Adaptive AI Task Partitioning and Offloading in Heterogeneous Edge-Cloud Networks

REAP: **R**untime **E**nergy-aware **A**daptive **P**artitioning and Offloading Framework

Akuen Akoi Deng

Eimantas Butkus

Master's Degree Project in
Computer and System Science
Spring term 2026
Supervisor: Praveen Kumar Donta
Department of Computer
and Systems Sciences

Stockholm University

Abstract

In recent years, the use of AI on resource-constrained IoT devices has grown significantly. However, most existing solutions for partitioning and offloading AI inference tasks across edge-cloud networks rely on static methods that are fixed before deployment and do not adapt to changes during runtime. Additionally, many of these solutions are evaluated in simulated environments rather than on real hardware. This thesis addresses this gap by designing and implementing an adaptive partitioning and offloading framework that dynamically determines where to split a neural network's layers across a three-node heterogeneous network. The framework was built in Python and tested on a physical testbed consisting of a Raspberry Pi 5 edge device, a laptop fog, and a high-performance GPU desktop PC as the cloud. Three CNN models were used for evaluation: VGG-16, AlexNet and MobileNetV2. The framework profiles the model at startup, measures the network link conditions between nodes, and periodically re-evaluates the partition to react to changes in the environment. Results show that compared to a static partitioning baseline, the framework achieved energy reductions of 35.82% for VGG-16, 35.70% for AlexNet and 27.09% for MobileNetV2. End-to-end latency was also reduced by 6.34%, 22.92% and 14.20%, respectively. These results show that adaptive partitioning can reduce energy consumption on resource-constrained devices while maintaining acceptable latency in a real heterogeneous edge-cloud network.

Acknowledgements

We would like to give special thanks to our supervisor Praveen Kumar Donta for all the help and guidance we received throughout the process. We also thank our peers and friends for their valuable support and encouragement.

Eimantas Butkus

This thesis is dedicated to the memory of my grandfather, whose support and encouragement continue to inspire me beyond his passing. I also want to thank my family for their unwavering support and belief in me throughout this journey.

Akuen Akoi Deng

Special acknowledgements and gratitude goes to my grandmother, mother, and siblings whose support and unwavering encouragement has been valuable throughout this thesis. This thesis is dedicated to my grandmother Aguil Ajang Duot and niece Nyankiir Bior Akoi.

Abbreviations

Ordered alphabetically for easy finding

Abbreviation	Description
AI	Artificial Intelligence
AIoT	Artificial Intelligence of Things
Alg	Algorithm
API	Application Programming Interface
AR	Augmented Reality
AWAS	Auction-based Workload Assignment Scheme
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DNN	Deep Neural Network
DPFL	Fused-Layer parallelization strategy
DQN	Deep Reinforcement Learning
DSRM	Design Science Research Methodology
GPU	Graphics Processing Unit
IIoT	Industrial Internet-of-Things
IoT	Internet of Things
LLM	Large Language Models
LSTM	Long Short-Term Memory
M2M	Mobile-To-Mobile
ML	Machine Learning
NLP	Natural Language Processing

OS	Operating System
PC	Personal Computer
QoE	Quality of Experience
RFID	Radio-Frequency Identification
RL	Reinforcement Learning
RNN	Recurrent Neural Network
RQ	Research Question
SAC	Soft Actor-Critic
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
USB	Universal Serial Bus
VR	Virtual Reality

List of Figures

List of Tables

Contents

1

Introduction

1.1 Introduction

In recent years, we have seen the widespread adoption of the Internet of Things (IoT) devices, leading to unprecedented growth in the volume, velocity, and variety of data generated [1] [2] [3]. These devices range from smartphones to Radio-Frequency Identification (RFID) readers, wearable devices to tablets, and gadgets [4]. They increasingly support applications and systems with strict requirements for latency, energy efficiency, and computational workload [2]. Few examples include smart home, cities, or health, robotics, Augmented Reality (AR) or Virtual Reality (VR), cognitive assistance, autonomous driving, video crowd sourcing, and M2M communications [3] [5]. At the same time, we have witnessed significant improvements in the capabilities of new Artificial Intelligence (AI) technologies, especially in machine learning and deep neural networks (DNNs), with models such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) [6]. These models have enabled fast, precise, and sophisticated data-driven decision-making across diverse fields of application [7] [6]. Consequently, the convergence of IoT and AI shows promising capabilities today and in the future [8].

Traditionally, AI models have been deployed and executed in a centralised environment, such as the cloud, which provides unlimited computational resources. However, this approach introduces limitations when applied to IoT systems [2]. In an IoT environment, factors such as network latency,

privacy concerns, bandwidth constraints, and poor connectivity can significantly degrade system performance in real time [9]. This, as a result, has led to a novel approach to combine fields such as edge & cloud computing, fog computing, AI, and IoT, leading to a growing interest in distributing AI computation across edge devices and cloud infrastructure. This approach allows some parts of the AI workload to run on edge devices alongside central cloud servers, a model known as edge-cloud AI. It promises to combine the low-latency benefits of edge computing and the scalability of the cloud [10] [8] [3].

Despite the promising potential, designing an adaptive and efficient AI-enabled IoT system in resource-constrained edge devices remains challenging due to the inherent heterogeneity of the edge-cloud environment. Typically, edge-cloud environments comprise devices with varying degrees of computational capabilities, energy budgets, needs, and network conditions [11]. Due to this, researchers have tried different approaches such as distributed training, deployment, and inference of AI models and workload [11] [12] [13].

In the case of distributed inference, the AI workload is distributed across several edge and cloud devices, where the aim is to execute the less computationally intensive and latency-critical task on the edge devices close to the data source, while the more computationally intensive tasks are offloaded to the cloud [14] [15]. However, most current solutions using this partitioning and offloading method for AI workloads rely on static deployment, where AI inference pipelines are predefined and fixed in advance and do not adapt to real-time or runtime changes. Such methods and solutions often fall short and lead to suboptimal performance when resources are constrained or have fluctuating availability. This can be due to varying network load, competing workloads, or device mobility [2]. Consequently, there is a need for an adaptive partitioning and offloading solution that can dynamically adjust how AI workloads are partitioned and executed across edge and cloud resources efficiently without violating other system requirements such as latency.

The literature on adaptive AI task partitioning and offloading in heterogeneous edge–cloud networks has expanded significantly in recent years [16] [17] [2]. Several frameworks have been proposed to address this challenge.

For example, some approaches employ reinforcement learning to determine optimal model layer partitioning strategies [17] [18], while others explore collaborative caching mechanisms [19] or edge–end joint parallel inference architectures [15]. However, a key limitation of many existing solutions is that they are primarily evaluated in simulated environments that assume ideal network conditions [16] [18] [20]. Consequently, the practical performance of these methods in real-world edge–cloud deployments remains insufficiently explored.

In one study, Li et al. [16] proposed a new framework that jointly optimises fine-grained DNN layer partitioning and magnitude-based channel pruning to reduce inference latency. The approach employs an LSTM-based controller that alternately generates discrete partitioning and pruning actions for each layer and is trained using a policy gradient algorithm with a delay-aware reward function that balances accuracy and total processing delay [16]. Despite promising results, the evaluation of the adaptive model partitioning and pruning was done through extensive simulation experiments in the Mobile-Edge Computing (MEC) environment.

Other available but limited studies conducted in real environments, such as [21], mainly focus on optimising metrics such as task completion time and waiting time while overlooking other metrics such as energy consumption and computational load balancing. In this study, Fang et al. [21] proposed EdgeDI, a partitioning method which first compresses models using structured pruning and a lightweight convolutional block (SCAR) with attention, then partitions feature-map workloads across multiple Industrial Internet-of-Things (IIoT) devices using an execution-time–aware scheme (OODP) that accounts for both computation capability and network bandwidth [21]. Results show that EdgeDI can provide better inference speedups while still maintaining the desired accuracy. However, this literature focuses only on improving inference performance.

These few examples, along with other related literature discussed more in-depth later in 2.1, show a consistent trend of inadequate real environment experimental evaluation of partitioning and offloading frameworks and a limited focus on energy consumption when designing a partitioning solution in a heterogeneous edge-cloud environment.

This thesis focused on real-world experiments using resource-constrained devices with the aim of reducing energy consumption without violating real-time latency. The objective was to design an adaptive partitioning and

offloading algorithm in heterogeneous edge-cloud networks to achieve the above-mentioned objective.

To achieve the purpose and objective of the research, we answered the main research question stated below

- **RQ:** To what extent can an adaptive model partitioning algorithm reduce the energy consumption of resource-constrained devices without violating real-time latency constraints in a heterogeneous edge-cloud network?

To address the research question, and achieve the research goals, this thesis proposes REAP (Runtime Energy-aware Adaptive Partitioning and Offloading Framework), an adaptive partitioning and offloading framework, as the name suggests. To reduce energy consumption in resource-constrained devices while maintaining acceptable end-to-end inference latency in a realistic physical testbed, the framework was designed to dynamically determine where AI workload should be partitioned by considering runtime computation cost, communication overhead, energy consumption, and latency. Instead of relying on a fixed partition chosen before deployment, REAP profiles the model, estimates the cost of candidate split points, monitors network and execution conditions, and periodically selects the most suitable partition during runtime. An indepth description of the framework is provided later in the methodology section 4

The scope of the thesis only covered the partitioning and offloading of AI inference tasks, with the exclusion of others such as training and data processing. Experimental validation was conducted using distributed edge-to-cloud testbeds incorporating a Raspberry Pi as the end node, a laptop as the edge node, and a GPU-enabled desktop PC as the cloud.

Our key contributions were as follows.

- Designed and implemented REAP, an adaptive partitioning and offloading framework for AI inference on a heterogeneous edge-cloud networks.
- We developed a fully functional setup comprising an end device (Raspberry Pi), edge servers (Laptop), and a Graphics Processing Unit (GPU) cloud server - Personal Computer (PC) to evaluate vertical offloading strategies in a realistic hardware environment.

- Unlike simulation-based studies, we designed a real-time testbed experiment and measured metrics such as energy consumption to justify the superiority of our method. Our results primarily focus on the trade-off between latency and energy constraint.

The rest of this thesis is structured as follows; The **problem section** 2 contains the related works and the description of research questions. **Extended Background** 3, which has key descriptions for the background and key concepts. **Methodology** 4, which contains the description of the methodology, the experiment procedure, and the proposed framework. **Results section** 5, which contains the key results and findings presented in tables and graphs. **Discussion** 6, which has an expanded interpretation of the results, conclusions, limitations, and future works.

2

Problem

2.1 Related works

Recent literature on Adaptive AI task partitioning and offloading in heterogeneous edge-cloud networks is vast. However, they focus on the different aspects, aims, and criteria. Some focus on reducing the inference time [17], latency [22], or minimising energy consumption [23] [19], while others focus on achieving optimal resource efficiency [14]. In this section, we will look at related works done on partitioning, offloading, and, lastly, joint adaptive partitioning and offloading.

2.1.1 Task Partitioning Frameworks

In an attempt to optimise resource efficiency, reduce latency, and reduce inference time in an edge-cloud heterogeneous network, some effort has been focused on how best to partition AI inference tasks before addressing the challenge of offloading. Recent literature aiming to achieve this includes Kim et al. [24], who proposed CO-PILOT, a DNN partitioning framework that jointly controls partition points, loss feature compression, and transport protocol selection, Transmission Control Protocol (TCP)/User Datagram Protocol (UDP). As a result, this leads to lightweight latency and accuracy prediction models (based on mean absolute deviation) to rapidly adapt collaboration settings at runtime without retraining [24]. However,

this work emphasises only improving the end-to-end latency while preserving the original model accuracy without regard to other metrics such as energy consumption or computational load balancing.

Masud et al. [25] proposed ParetoPipe, an open-source framework that explicitly captures the trade-off between end-to-end latency and inference throughput under realistic network variability. By looking at DNN partitioning as a multi-objective optimisation problem, the algorithm performs exhaustive block-level pipeline partitioning across heterogeneous edge hardware (Raspberry Pis and a GPU edge server) and identifies Pareto-optimal split points that balance competing objectives [25]. Despite this unique approach, ParetoPipe lacks adaptive capability and only focuses on finding the optimal balance between the two trade-offs(latency and inference throughput), while others, such as energy consumption, are out of scope. The experiment is also performed between only two nodes unlike our approach using three nodes.

Similarly, by looking at partitioning as a mixed-integer multi-objective optimisation task, Ma et al. [20] designed a joint optimisation framework that jointly determines the optimal model partitioning point and the corresponding edge resource allocation ratios, addressing the interdependence between partition decisions and server-side resource scheduling. This is achieved by using a deep reinforcement learning approach based on Proximal Policy Optimisation (PPO) within an Actor–Critic architecture [20]. Similar to this thesis, the literature focuses on joint optimisation of latency and energy consumption as the core objectives. However, the experiments were carried out in a simulation, which contrasts with our goal.

2.1.2 AI Inference Offloading approaches

In a scenario, where successfully partitioning AI workload is gained, it becomes vital to address the challenge of determining where inference tasks should be executed (mobile device, edge cloud, or central cloud), and when tasks should be offloaded [26], to achieve the aim of minimizing overall power consumption under dynamic content demands and heterogeneous resource constraints, different approaches have been proposed [19].

Ko et al. [26] propose ST-CODA, a spatial–temporal computation offloading decision algorithm which includes an opportunistic delay to exploit low-cost access such as Wi-Fi. Using value iteration, ST-CODA optimises transmission cost, mobile energy use, and deadline satisfaction [26]. While

it is effective for general mobile tasks, ST-CODA's use of intentional transmission delays makes it impractical for real-time AI inferences. Additionally, their framework does not explore intra-model layer partitioning or the inclusion of the fog node, both of which are important for optimising our system.

On a different case, Fang et al. [19] proposed another framework in which they formulate offloading as a power minimization problem that jointly considers task execution, in-network caching, and request aggregation, and solve it using a deep reinforcement learning (DQN)-based policy that adaptively selects collaborative caching and offloading decisions based on historical content requests and current network state [19]. However, using deep reinforcement learning requires significant computing power just to make routing decisions. In addition, their framework focuses on offloading or caching the entire task at once. In contrast, our system avoids this heavy processing overhead by using a fast, lightweight algorithm to split the AI model layer-by-layer across three distinct hardware levels.

Similarly, Chen et al. [23] propose an energy-efficient layer offloading algorithm, SPSO-GA, which combines self-adaptive particle swarm optimisation with genetic operators. With SPSO-GA, the system energy consumption is reduced as the deadline becomes looser. With the looser deadline constraints, more layers tend to be allocated to the more energy-efficient servers in the same situation [23]. The main drawback of their approach is that the genetic algorithms are too computationally heavy for fast real-time routing.

2.1.3 Joint Adaptive Partitioning + Offloading Approaches

To address the challenging nature and complexity of running distributed inference across an end-edge-cloud environment, other recent literature has attempted to tackle both partitioning and offloading simultaneously in an adaptive manner. These solutions try to achieve a great balance between partitioning and offloading of inference tasks in a heterogeneous environment.

In one study, Zhao et al. [15] proposed AdapCP, a collaborative inference framework that employs an inter-layer partitioner to determine multiple split points for the offloading phase, and an intra-layer parallel partitioner using a Dirichlet-guided DDPG reinforcement learning policy to distribute

convolutional filters and fully connected neurons across servers for parallel execution . The results show that AdapCP achieves substantial latency reductions, improving inference speed by up to 2.21× while maintaining accuracy [15]. The main drawback of this work is that facilitating collaborative inference involved the use of multiple increasing edge servers, which introduced high data transmission overhead along with data redundancy from multiple runs. Additionally, unlike our thesis, this literature in part aims to address data privacy and security challenges in distributed workload execution, improving inference time while preserving accuracy.

In another, Shen et al. [17] proposed EECRL, a reinforcement learning–based task scheduling framework which models online scheduling as an RL decision process, where an adapted Q-learning/DQN agent selects execution placement and optional delay actions based on real-time cluster state and task characteristics, implemented within an extended Kubernetes–Rancher orchestration architecture. Experiments show that it achieves substantial performance gains [17]. The main limitation of this work is that it focuses only on addressing the burst of AI task scenarios in an End-Edge-cloud computing framework. Lastly, the method is not extended to multiple optimisation objectives, such as server cost and energy consumption.

Chen et al. [9] proposed EdgeCI, a low-latency distributed CNN inference framework that exploits cooperative execution among locally connected devices instead of relying solely on edge servers or the cloud. Introduces an Auction-based Workload Assignment Scheme (AWAS) mechanism that balances expanded feature-map partitions across devices while accounting for convolutional overlap costs, and a Dynamic Programming–based Fused-Layer parallelisation strategy (DPFL) that determines the optimal fused-block partitioning configuration to trade off computation and communication overhead [9]. However, one limitation is that this approach is not applicable to graph neural networks and partitioning of non-convolutional layers (i.e., fully connected layers). An interesting detail is that this literature uses a similar approach and methodology as we later propose (experimental testbed). A key distinction, however, is that Chen et al. address the challenge of unstable performance and privacy leaks in a heterogeneous environment, while we look into energy consumption and preservation of other metrics such as latency and computational load balancing.

In another study, Peng et al. [27] proposed APT-SAT, a framework that integrates a prediction stage that models layer-wise delay and energy consumption using regression-based estimators, an adaptive multi-point DNN

partitioning algorithm that jointly optimises workload balance, transmission delay, and energy consumption, and a Soft Actor-Critic (SAC)–based reinforcement learning offloading strategy that determines routing and task allocation under dynamic network conditions [27]. One key notable drawback from this work, however, is the increased computational complexity that results from finding the optimal partition point. Peng et al. acknowledge this by calculating the Big-O notation of the complexity equivalent to exponential.

Lastly, Zhang et al. [18] proposed SEMA (Semantic-aware Meta-learning-based Adaptation), a meta-reinforcement learning framework for adaptive DNN partitioning in dynamic cloud–edge systems that jointly optimises the model partition point, offloading target, and semantic compression level of intermediate feature data to maximise a unified Quality of Experience (QoE) metric encompassing latency, energy consumption, and task accuracy [18]. Similar to other frameworks mentioned before, SEMA was evaluated in a simulated experimental setup.

In an empirical survey conducted by Chen et al. [2] evaluating recent and common partitioning and offloading frameworks, we observe that, despite the various partitioning and offloading algorithms proposed over the years, most of this existing work evaluates the framework of task partitioning and offloading with simulation experiments. This observation is consistent with most of the literature summarised above and in table 2.1 below. However, the setting provided by the simulation can be ideal and hardly reflect the performance of the algorithms in actual practice. Due to this, there is a real gap and a need to design and migrate to a real experimental environment to detect the ideal performance and outcomes in real scenarios [2].

Additionally, we find that a great number of these existing studies focus on optimising latency, reducing inference time, while a limited few focus on reducing energy consumption on resource-constrained networks also shown in table 2.1. This provides a gap for designing and implementing a novel solution with the aim of reducing energy consumption while preserving latency and other requirements based on a real experimental environment instead of using a simulated one.

Table 2.1: Comparison of related works on AI task partitioning and offloading in edge-cloud networks.

Study	**Framework**	**Optimisation Target**	**Technique**	**Environment**
Task Partitioning				
Li et al. [16]	—	Latency	LSTM + policy gradient RL	Simulation
Kim et al. [24]	CO-PILOT	Latency, accuracy	MAD-based prediction models	Real testbed
Masud et al. [25]	ParetoPipe	Latency, throughput (Pareto)	Exhaustive block-level search	Real testbed
Ma et al. [20]	—	Latency, energy	Deep RL (PPO, Actor–Critic)	Simulation
AI Inference Offloading				
Ko et al. [26]	ST-CODA	Transmission cost, energy, deadline	Value iteration	Simulation
Fang et al. [19]	—	Power minimisation	Deep RL (DQN) + caching	Simulation
Chen et al. [23]	SPSO-GA	Energy	Particle swarm + genetic alg.	Simulation
Joint Adaptive Partitioning + Offloading				
Zhao et al. [15]	AdapCP	Latency (inference speed)	DDPG RL (inter + intra layer)	Simulation
Shen et al. [17]	EECRL	Inference time	Q-learning / DQN	Real (Kubernetes)
Chen et al. [9]	EdgeCI	Latency, privacy	Auction-based (AWAS) + DP	Real testbed
Peng et al. [27]	APT-SAT	Latency, energy, workload balance	Regression + SAC RL	Simulation
Zhang et al. [18]	SEMA	QoE (latency, energy, accuracy)	Meta-RL	Simulation
Fang et al. [21]	EdgeDI	Inference speed	Structured pruning + OODP	Real (IIoT)
Our work	**REAP**	**Energy, latency**	**Weighted scoring + periodic re-eval.**	**Real testbed**

2.2 Research Question

To what extent can an adaptive model partitioning algorithm reduce the energy consumption of resource-constrained devices without violating real-time latency constraints in a heterogeneous edge-cloud network?

2.3 Delimitation

Due to both limited time and resources, the scope of this study was kept within the following boundaries to ensure a feasible outcome.

- Inference Focus: Research is strictly limited to the inference phase. The computationally intensive task of model training is excluded, as the primary objective is to optimise real-time decision-making on pre-trained models.
- Model Architecture: The evaluation is limited to Convolutional Neural Networks CNN and standard DNNs, such as VGG, MobileNetV2, and AlexNet. Large Language Models (LLMs) and audio processing models are outside the scope of this work.
- Privacy: Network privacy, security, and encryption mechanisms are excluded from the experimental design. This ensures that latency and energy measurements reflect the pure algorithm overhead.
- Hardware Constraints: The testbed utilises commercial off-the-shelf hardware, such as Raspberry Pi, laptops, PC. Specialised industrial-grade hardware is not utilised. Instead, network conditions are emulated via software.

3

Extended Background

3.1 IoT

IoT is defined as a network of interconnected physical devices such as sensors, actuators, servers and more that communicate over the internet to collect, exchange, and process data [28] [29]. These devices range from simple household objects, such as smart light bulbs, to complex industrial tools or systems [4]. The main objective of IoT is to connect the physical and digital worlds, enabling remote monitoring, automation, and intelligent decision-making [3].

Early IoT systems mainly consisted of simple sensing devices that transmitted small volumes of data to centralised cloud servers for processing and storage. These devices performed minimal local computation and relied heavily on cloud infrastructure for data processing and decision-making. However, this approach has slowly evolved with the emergence of intelligent IoT systems or Artificial Intelligence of Things (AIoT) [23]. In AIoT systems, devices are increasingly capable of performing local data processing and inference. As an example, modern smart surveillance cameras can perform small tasks such as human detection or face recognition locally before transmitting relevant data to the cloud [8].

An IoT environment is usually heterogeneous, consisting of a wide range of hardware platforms with varying computational capabilities. These include resource-constrained microcontrollers as well as more capable computers such as the Raspberry Pi [4]. Despite the increasing computational capabilities of modern IoT devices, they remain significantly constrained compared to traditional computing systems.

One of the key limitations of IoT devices is their limited energy capacity, as many operate on batteries and must maintain low power consumption to achieve long operational lifetimes [23]. In addition to energy constraints, IoT devices usually have limited processing power, memory, and storage compared to desktop or server systems. These limitations make it challenging to execute computationally intensive tasks such as deep learning inference directly on edge devices [2].

3.2 Artificial Intelligence

AI refers to the simulation of human intelligence in machines that are programmed to think and learn. Unlike traditional rule-based programming, where a developer explicitly codes every decision. AI systems use data to identify patterns and make decisions with minimal human intervention [7] [30].

In recent years, AI sub-fields such as Machine Learning (ML) and Deep Learning (DL) have increased in both capability and complexity. These capabilities include learning from complex labelled and unlabelled data, pattern recognition, natural language processing, and image processing. This, in turn, has led to increased application and integration of AI across different fields, such as IoT, thereby driving the emergence of intelligent IoT systems, or Artificial Intelligence of Things (AIoT) [23], as mentioned in the previous subsection.

3.2.1 Machine learning

ML is a subset of AI that provides systems with the ability to automatically learn and improve from experience without being explicitly programmed. ML algorithms use statistical techniques to build a model from sample data, known as “training data”, in order to make predictions or observations [7].

The main types of Machine learning are supervised, unsupervised learning and reinforcement learning. Supervised machine learning models are

trained with labelled data sets, which allow the models to learn and grow more accurate over time. In unsupervised machine learning, a program looks for patterns in unlabeled data, while reinforcement machine learning trains machines through trial and error to take the best action by establishing a reward system. Other sub-fields of machine learning include Natural Language Processing (NLP) and DL [31] [32]

3.2.2 Deep learning

DL is a specialised subset of machine learning inspired by the structure of the human brain. While traditional ML requires manual feature extraction, Deep Learning uses multi-layer artificial neural networks to learn these features automatically from raw data [6]. This ability to learn from unstructured data, such as images and video, makes DL the standard for computer vision tasks in IoT Applications [8]. The figure below 3.1 shows an illustration of the different subsets of AI as discussed above.

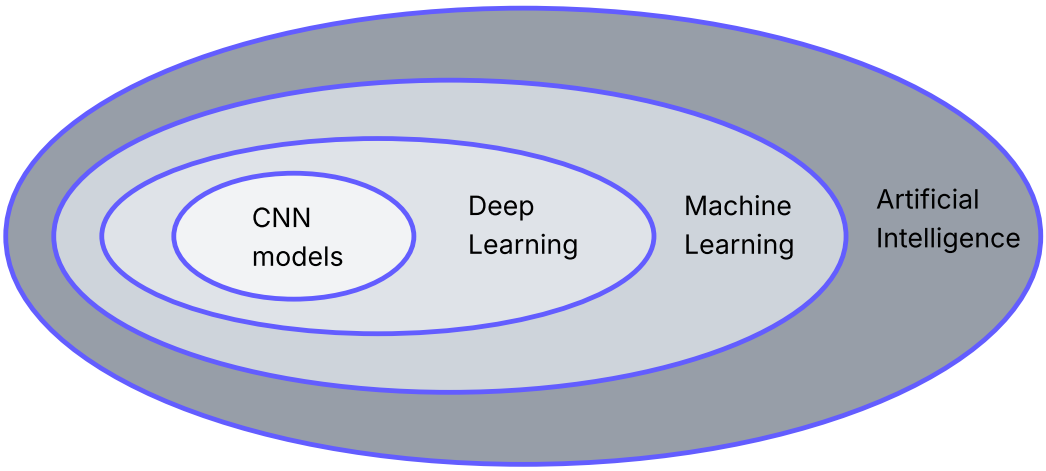

Figure 3.1: Subsets of AI (Adapted from [33])

3.2.3 Convolutional Neural Networks (CNN)

For computer vision tasks, such as image classification or object detection, the dominant architecture is the Convolutional Neural Network (CNN) [9]. The CNN is composed of a sequence of layers. Which typically include convolutional layers, pooling layers and fully connected layers. Convolutional layers act as filters that detect features like edges, texture or shapes. Pooling

layers reduce the spatial size of the representation to decrease the computational power required. Fully connected layers perform the final classification based on the features detected by previous layers [14]. It is very important in this thesis to understand how layers work and how to utilise them. Partitioning the model is possible because of the layers. It is possible to have the first 5 layers in Raspberry Pi and the rest of them, which are computationally heavy, offloaded to a more powerful edge server [15, 24].

3.3 Hierarchical Edge-cloud

Traditional cloud-only computing architectures are no longer sufficient or efficient for modern AI systems. In such use cases, large volumes of data generated by devices used to be transmitted to centralised cloud data centres for processing. This approach can lead to increased latency and significant bandwidth consumption, particularly in systems that require real-time responses [10].

To address these challenges, hierarchical computing architectures have been introduced. In this model, an intermediate layer known as the *edge* is positioned between end devices and centralised cloud infrastructure. Edge computing enables data processing to occur closer to the data source, thereby reducing communication latency and lowering network bandwidth usage when handling large volumes of data [3]. An example illustration of a Hierarchical Edge-cloud architecture is shown below in Figure 3.2.

3.3.1 Cloud Computing

At the highest level of the computing hierarchy is cloud computing, which refers to large-scale, centralised data centres that provide substantial computational capacity, storage resources, and scalability. In the context of this thesis, the cloud tier is represented by a high-performance desktop PC equipped with a GPU, serving as the most computationally capable node in the proposed edge–cloud architecture.

The primary advantage of the cloud tier is its ability to execute computationally intensive deep neural network models with significantly fewer resource constraints compared with edge or intermediate devices. Unlike battery-powered or thermally constrained devices, cloud infrastructure is typically less limited by energy availability, memory capacity, and heat dissipation. However, the main limitation of relying on the cloud is its physical and network distance from the data source. This can introduce increased

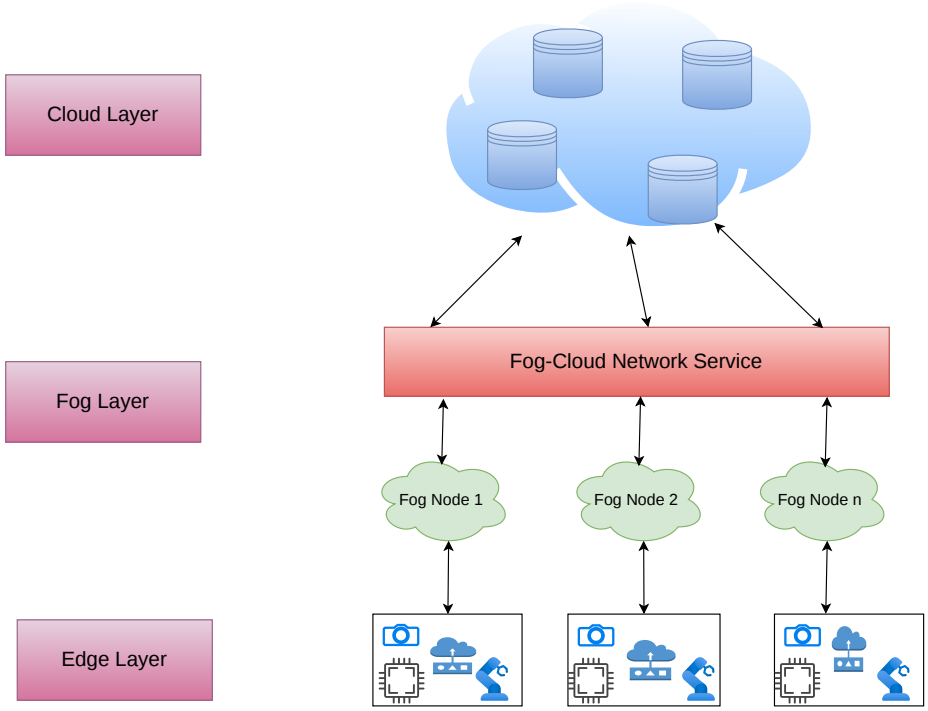

Figure 3.2: Hierarchical Edge-cloud architecture [34]

communication latency, bandwidth variability, and network jitter, which may negatively affect the responsiveness of real-time or latency-sensitive applications [10, 26].

3.3.2 Edge and Fog Computing

Edge computing is a new computing paradigm for performing tasks at the edge of the network. Unlike cloud computing, it emphasizes the approach of running tasks directly on the end devices or closer to the user and the data source [35]. The main aim and advantage of introducing this paradigm was to avoid or reduce unnecessary data transfer hence reducing latency [3]. Since data does not need to travel far before being processed, it can also reduce bandwidth usage and improve privacy. However, edge devices usually have limited CPU power, memory, battery capacity, and thermal headroom [35].

On the other hand, fog computing refers to the intermediate bridge sitting between cloud and edge computing. It acts as an intermediate computing layer that has more resources than the edge device but fewer resources than

the cloud [10]. In our case, this is represented by the CPU laptop node. In some cases, fog layer provides a fall safety layer that can guarantee system availability and resilience. All the three aforementioned layers; edge-fog-cloud form the 3-Tier continuum.

3.3.3 The 3-Tier Continuum

A 3-tier computing continuum refers to a distributed architecture in which computational tasks are shared across three hierarchical layers: the edge, the fog, and the cloud as described in the previous sections [36]. The term continuum implies that all three nodes form a connected pipeline treated as a monolithic entity where tasks can be partitioned, offloaded and distributed across the network depending on resource availability, latency, and energy consumption [36] [37].

In this thesis, the three-tier continuum is represented by a Raspberry Pi, a laptop, and a GPU-enabled desktop PC. The Raspberry Pi forms the edge tier, the laptop represents the fog tier, and the GPU desktop PC represents the cloud tier. Together, these devices form a continuous edge–fog–cloud execution environment in which deep neural network inference can be partitioned and offloaded across multiple nodes referred to as "Cloud-Edge-End Collaboration" [2, 19]

3.4 Edge–Cloud Collaborative Intelligence

As AI advances more and more every day in all aspects of society today, we are starting to see its integration in IoT systems, autonomous vehicles, smart cities and mobile applications as discussed in the earlier sections. However, to perform AI inference, learning and decision making on these IoT systems [38], while considering the various challenges they present, such as the heterogeneity of their environments, capability and resources availability, a new approach had to be considered [39]. This led to a new computing paradigm, Edge-cloud collaborative intelligence, which allows AI tasks to be executed on both edge devices and cloud servers simultaneously, hence allowing them to collaborate intelligently [39] [38].

The benefit of this new approach is that it allows us to balance latency, obtain better energy consumption, reduce bandwidth usage and improve computational efficiency. This is usually achieved by performing partitioning, task offloading and adaptive scheduling [2].

Key challenges from this approach include how to achieve optimal partitioning, resource efficiency, and handling energy constraints. To advance this paradigm, there is a need for solutions that enable us to achieve optimal partitioning and offloading of AI tasks while addressing key trade-offs such as energy consumption, latency, and system performance [39]. Through tackling the challenge of optimal partitioning and offloading with the aim of reducing energy consumption while preserving latency and performance by providing an adaptive partitioning and offloading solution, this thesis aimed to fulfil the needs mentioned above and hence contribute to the current knowledge and further the edge-cloud collaborative intelligence computing paradigm.

3.5 AI workload Partitioning in Edge-Cloud Networks

As briefly mentioned above, AI workload partitioning is one of the key techniques and methods used to achieve collaborative edge-cloud intelligence. In an edge-cloud heterogeneous network, instead of executing AI workloads such as inference on a single node, the workload is split and distributed across different node devices [27]. The two main types of partitioning are static and dynamic partitioning. In static partitioning, the partitioning is fixed beforehand before the model deployment, while in dynamic partitioning, the system adaptively changes the partition points during runtime based on metrics such as network latency, battery levels, bandwidth availability and more.

In most systems, such as CNN-based IoT systems, the partitioning is always done on the layer level [15] [9]. Some of the main benefits of applying partitioning in an edge-cloud collaborative system include reduced latency, energy consumption, and network traffic [24] [25] [21] [20]. The main challenge in partitioning is how to achieve optimal partitioning of the workload. To solve this challenge, different approaches and techniques of partitioning have been attempted, with some of the most common including layer-wise partitioning [15], reinforcement learning-based partitioning [17], joint partitioning and pruning [16], just to mention a few. Adding to this knowledge base, in this thesis, we developed a dynamic task-level-based partition approach.

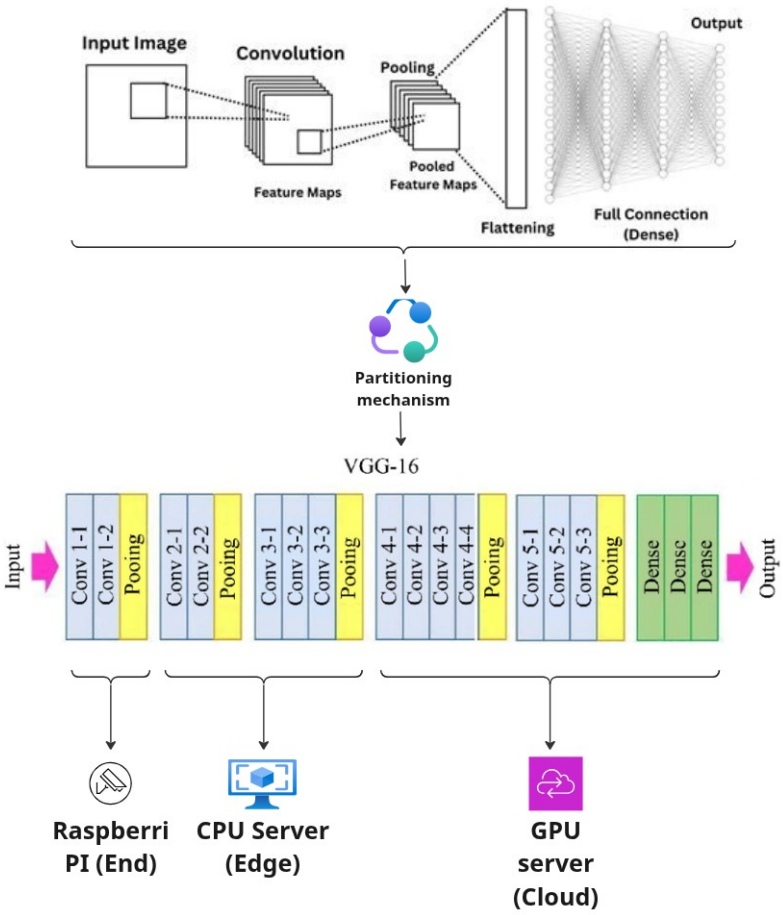

Figure 3.3: AI workload Partitioning in Edge-Cloud Networks

3.6 AI Inference Offloading in Edge-Cloud Networks

Similarly, as mentioned earlier, computation offloading is another technique used when executing AI workload across an edge-cloud collaborative network. It is the process of transferring computational workload from one node to another, a more powerful or appropriate node for execution across the same network [39] [2].

The main objective is to improve system performance, reduce energy consumption and reduce latency in a resource-constrained heterogeneous environment [39]. Types of offloading include full, partial, and dynamic offloading. In full offloading, the whole task is offloaded and executed on a

single node device, while in partial offloading, only some parts of the task are transferred for execution on another node. Lastly, for the dynamic offloading, the system decides at runtime where the task should be executed based on metrics such as network latency, deadlines, bandwidth, or battery levels [5] [2].

Some of the main benefits of this technique are reduced node workload, better system performance, and energy saving [39] [26] [19] [23]. One of the main challenges, however, includes the introduction or increase of latency overhead and how to achieve optimal resource management [39]. System offloading approaches usually rely on the design decisions made on the partitioning part of the system. How we partition the workload, for example, aspects such as decision granularity, can highly affect the resulting effects from offloading. In an AI-enabled IoT system, a lot of effort and emphasis has been placed on how to develop the optimal adaptive partitioning and offloading solution [2]. Similarly, this thesis contributes to this existing work by proposing a new approach that aims to achieve this.

4

Methodology

4.1 Selection of Research methodology

From the state-of-the-arts research methodologies, we chose to use **experiment as the main research methodology**. According to Stol & Fitzgerald [40], an experiment aims to investigate, evaluate, or compare techniques, practices, processes, or approaches within a real-world and pre-existing setting. However, to address our research question:

RQ: *"to what extent can an adaptive model partitioning algorithm reduce the energy consumption of resource-constrained devices without violating real-time latency constraints in a heterogeneous edge-cloud network?"*

The specific method chosen within experimental research for this study was a quantitative physical testbed experiment. This involved creating a real-world, controlled hardware environment to measure system performance on different AI task partitioning strategies. It was chosen because the research question requires measurable evidence of energy consumption, communication delay, and end-to-end latency under real hardware conditions. It also provides enough experimental control to compare static and adaptive partitioning fairly using the same models, devices, and conditions. Therefore, this method offers the best balance between realism, control, reproducibility, and direct performance measurement.

4.1.1 Alternative methods considered within experiment

When deciding on the specific type of experiment methodology to use, there were two other choices that were considered. Firstly, a pure software simulation, and on the other hand, a fielded experiment. Software simulation was rejected because it cannot accurately account for real-world environmental variables, as well as there already existing extensive research done using this approach. For example, Raspberry Pi experiences real thermal throttling when running a heavy AI workload, which drastically impacts both latency and power consumption. In simulation, these environmental variables are optimised and would not provide accurate real world measurements. On the other hand, a fielded experiment would need a deployment of the system in a production environment such as autonomous vehicle or system. This was deemed to be unnecessary, overly complicated and time-consuming within the scope of a master's thesis. Since the focus of this study was system performance metrics such as latency and energy consumption rather than AI model's accuracy, introducing unpredictable real-world variables would be hard to isolate and measure the performance of the partitioning algorithm itself.

Therefore, a physical testbed evaluation was found to be the most suitable and justified choice to address the research question. using real devices in a controlled environment, we could feed the DNN models dummy data and precisely isolate how the partitioning algorithm impacts latency and energy consumption. This methodology is widely accepted and utilised in similar Edge-Cloud computing research. For example, the authors of the DeepThings framework [14] and EdgeCI [9] successfully employed physical testbeds using Raspberry Pi devices to evaluate latency and energy performance for their distributed inference frameworks.

4.1.2 Alternative primary methods considered

Alternative research methodologies such as Design Science Research Methodology (DSRM) [41] [42], and case study research [41] [43] were considered but not used as the primary methodology. DSRM [42] was considered because this thesis develops a technical artifact, REAP: an adaptive partitioning and offloading framework; however, it was not selected as the main methodology because the central aim of the thesis was not only to design an artifact, but to quantitatively measure its performance against a static baseline using energy consumption and latency as evaluation metrics.

A case study [43] was also considered because the framework was evaluated in a specific three-node edge–cloud continuum, but this method was less suitable because the research required controlled comparison, repeated measurements, and numerical performance analysis rather than an in-depth descriptive investigation of one deployment context. Therefore, although elements of design science are present in the construction of the proposed framework, the thesis mainly follows a quantitative physical testbed experimental methodology, as this best supports the objective of measuring whether adaptive partitioning reduces energy consumption without violating latency constraints in a physical heterogeneous edge–cloud testbed.

4.1.3 Data collection

To gather results, the data collection methods involved a hybrid approach. For energy consumption measurement, the use of USB electricity was considered for edge devices. However, this approach was later considered inconvenient. As a result, software-based OS-level monitoring tools were applied to measure CPU and GPU data, as well as to log the exact execution latency for each inference task. This aspect is discussed further in detail in the next sections below.

4.2 Research Experiment Design and Setup

To properly evaluate the adaptive partitioning algorithm, a heterogeneous computing environment was established. This setup was designed to reflect a realistic Cloud-Edge-End continuum, utilising different tiers of hardware.

4.2.1 Hardware Environment

The physical testbed included 3 computing tiers, representing the different levels of resource availability in a typical IoT network:

- End Device (Raspberry Pi): This represented a resource-constrained IoT edge device. It acted as a data source and the primary initiator of the inference tasks. It is strictly limited by thermal throttling and power capacity.
- Edge Node (standard laptop): This represented the local edge server, providing a significant boost to the computational power over the

Raspberry Pi while also maintaining low network latency via a local network connection.

- Cloud Node (high-performance Desktop PC (RTX 4070 TI)): This represented the centralised cloud server, offering the highest computational capabilities for completing the heaviest inference tasks.

The hardware specifications of the nodes mentioned above are shown in the table below. Server 2 (GPU cloud is the most computationally capable, while Raspberry Pi is the least. The table shows the processor type and speeds of each node device, along with the RAM memory type, speed, and capacity.

Table 4.1: Hardware specifications of the devices in the end-edge-cloud nodes

Node	**Processor**	**RAM Memory**
Raspberry Pi (End)	CPU: 4-core Arm Cortex-A76, 2.4 GHz	8 GB LPDDR4X
Server 1 (Edge)	CPU: 4-core Intel Core i7-10510U, 1.80–4.90 GHz	16 GB DDR4, 2667 MT/s
Server 2 (Cloud)	GPU: NVIDIA GeForce RTX 4070 Ti, 7680 CUDA cores, 40.09 FP32 TFLOPS	32 GB DDR5, 5600 MHz; 12 GB GDDR6 VRAM

4.2.2 Software and Platform

The experimental framework and the adaptive partitioning algorithm were developed using Python. The neural network models chosen are implemented using standard deep learning libraries such as PyTorch. To monitor system health and resource bottlenecks during the experiment, standard OS-level monitoring software was utilised to track real-time Central Processing Unit (CPU) and GPU utilisation percentages.

For the distributed network connection across all the nodes, Tailscale, a secure private networking system, was used along with ZeroMQ, an open source asynchronous messaging library for messaging and sending the packets across one node to another. Tailscale was also used to introduce throttle, bottleneck and network traffic constraints across the network.

4.2.2.1 Energy consumption measurement

Raspberry Pi

On the Raspberry Pi, energy was estimated using the Pi's internal clock to track compute time.

$$E_{\mathrm{Pi}} = P_{\mathrm{Pi}} \times t_{\mathrm{Pi}} \tag{4.1}$$

where E_{Pi} denotes the estimated energy consumed by the Raspberry Pi during execution of its assigned model segment, P_{Pi} is the assumed constant Raspberry Pi power in watts which is 12W in this case, and t_{Pi} is the measured computation time in seconds for the Raspberry Pi segment. The Raspberry PI constant of 12 W is multiplied by the measured computation time to give us the estimated energy consumption during the inference execution.

Laptop

For the laptop node, energy was measured using Intel RAPL via the Linux powercap, which is considered the most accurate estimation of energy usage on an Intel-based Linux system such as ours.

$$E_{\mathrm{laptop}} = e_1 - e_0 \tag{4.2}$$

where E_{laptop} denotes the energy consumed by the laptop during execution of its assigned model segment, e_0 is the RAPL cumulative energy counter reading taken immediately before computation, and e_1 is the RAPL cumulative energy counter reading taken immediately after computation.

PC

On the GPU server, energy consumption was estimated using NVIDIA NVML through pynvml, which returns an instantaneous GPU power in milliwatts.

$$E_{\mathrm{PC}} = \frac{P_0 + P_1}{2}\, t_{\mathrm{compute}} \tag{4.3}$$

where E_{PC} denotes the estimated energy consumed by the PC during execution of its assigned model segment

P_0 is the GPU power measured at the start of computation, P_1 is the GPU power measured at the end of computation, and t_{compute} is the PC computation time in seconds.

4.2.3 DNN Model Selection

The primary objective of this experiment was to evaluate system performance metrics, specifically latency and energy consumption. Because of that, the predictive accuracy of image recognition was outside the scope. Therefore, instead of using real images, the system used randomly generated dummy data that match the input dimensions of the respective models. This approach isolates the computational workload and eliminates unnecessary variables related to dataset loading or image pre-processing. For the experiments, we chose models such as VGG, AlexNet, and MobileNet, which have been used time and time again in most of the recent related literature [15] [16] [9].

The table below shows the three models that were chosen for the experiment based on their differences in terms of parameters, number of blocks, and size. Among other models that were considered, such as NiN, ResNet-34 and others. We choose VGG16 due to its high parameters, blocks and size, AlexNet for its medium parameters, blocks and size and MobileNetV2 for its low parameters and size. These three represent the full range of complexity in terms of parameters, blocks and size. Two stand at the ends of the spectrum in terms of size and complexity, while AlexNet sits in the middle.

Table 4.2: DNN Models Used

Model	**Parameters**	**Blocks**	**Size (MB)**
MobileNetV2	2,236,682	21	8.8
AlexNet	61,100,840	21	234
VGG16	138,357,544	39	528

4.2.3.1 VGG-16

The model consists of 13 convolutional layers organised into five blocks, each block ending with a 2 x 2 max-pooling layer that halves the spatial resolution. The convolutional stack is followed by three fully-connected layers and a softmax classifier. All convolutions use a fixed 3 x 3 kernel size with stride 1 and padding 1, which makes the architecture conceptually simple but computationally expensive: VGG-16 has 138 million parameters and requires approximately 15.5 GFLOPs per inference. The model was included as a representative of large, dense convolutional architectures that are difficult to run on resource-constrained devices.

4.2.3.2 AlexNet

The architecture consists of five convolutional layers, three of which are followed by max-pooling, and three fully-connected layers ending in a 1000-way softmax classifier. The first convolutional layer uses an 11 x 11 kernel with a stride of 4, which aggressively downsamples the input and produces a much smaller intermediate activation tensor than VGG-16. AlexNet has approximately 61 million parameters and requires 0.7 GFLOPs per inference, making it considerably less computationally demanding than VGG-16 despite having a comparable number of parameters. The model was included as a representative of mid-sized convolutional architectures with classical layer composition.

4.2.3.3 MobileNetV2

The architecture replaces the standard convolutions used in VGG-16 and AlexNet with inverted residual blocks that combine depthwise separable convolutions and linear bottlenecks. A depthwise separable convolution factorises a standard convolution into a depthwise convolution that operates on each input channel independently, followed by a 1 x 1 pointwise convolution that combines channels. This factorisation reduces computation by roughly a factor of k^2, where k is the kernel size, at the cost of a small accuracy reduction. Each inverted residual block also contains a skip connection between its input and output, which means the block must be treated as atomic for partitioning purposes - splitting inside a block would break the residual addition. MobileNetV2 has approximately 2.2 million parameters and requires 0.3 GFLOPs per inference. The model was included as a representative of architectures explicitly designed for edge deployment.

4.3 Experimental Procedure

To ensure that this study is reproducible, the experiment was divided into three distinct phases. Baseline, the execution of the proposed adaptive partitioning algorithm, performance evaluation and comparison.

4.3.1 Baseline

Before introducing the partitioning algorithm, it was necessary to establish the baseline performance limits of hardware. In this phase, the chosen

deep learning models were executed on a single device without any task offloading. The models were each executed entirely on Raspberry Pi to check how long it took, as well as how much energy was consumed. Similarly, they were also executed on both the laptop and a PC to observe and record the performance data before partitioning. The recorded performance data were significant for later comparison and evaluations. These baseline executions were performed ten times on each device as mentioned previously, then averaged to achieve more consistent measurements of both energy and latency.

In the next phase, the distributed network was set up, using a systematically chosen static partition on each model, each of the three models were ran on the heterogeneous network to give us the baseline data for a static partitioning approach. The static partition execution approach was also run ten times on each model to give us consistent baseline data. The static split chosen were aimed to ensure that each node of the distributed network does at most a third of the model's total workload. We chose this approach to create a baseline where the workload is spread roughly evenly. This ensured that no single node ended up executing a significantly high percentage of the computation. This also reflected the kind of split a developer might reasonably pick when setting up a distributed pipeline without any runtime profiling, hence making it a practical reference point to compare our adaptive method against. For example, for VGG-16, we chose the split of (10,30), which means that the Raspberry Pi executes the first ten blocks of the feature extraction layer, the laptop finishes the remaining 11-30, while the PC completes the other pooling and classifier layers.

4.3.2 Partitioned Execution

After baseline executions, using the same models (VGG-16, AlexNet and MobileNetV2) and dummy input tensors, we executed similar inference runs on the heterogeneous environment, along with REAP: the proposed adaptive partitioning algorithm. In this phase, the inference task is no longer statically partitioned and distributed on the heterogeneous network, but the algorithm dynamically determines where to split the model's architecture. The Raspberry Pi executes the initial layers of the model, and based on where the algorithm determines to cut it, it will send the remaining layers to either the edge node or cloud node, depending on how complex it is.

Just as in the baseline runs, the software tools continuously monitor Raspberry Pi's energy draw, computational load, and the total time taken to complete the split task.

4.3.3 Performance Evaluation and Comparison

The final phase of the experiment included a direct quantitative comparison between data collected in Phase 1 (baseline) and Phase 2 (partitioned execution), specifically between the static partition results and the adaptive partitioning results.

Evaluation metrics

The evaluation focused on two primary metrics:

- ***Total End-to-End Latency*** : Measured in milliseconds (ms). For both static and adaptive partition execution, this total time includes local computation time on the Pi, the network transmission time to send the intermediate features, and the remote computation time on the Edge/Cloud server.
- ***Energy Consumption*** : Measured in Joules (J). The total physical power drawn by the Raspberry Pi, laptop and PC during the inference task, as recorded through the previously explained calculations, both for static and adaptive partition approaches.

By comparing the partitioned results to the baseline, the evaluation determined if the adaptive algorithm has successfully reduced the energy consumption on resource-constrained devices while maintaining or even improving overall latency. This will, as a result, answer the question of the extent to which the adaptive model partitioning algorithm has reduced the energy consumption without violating real-time latency constraints in our heterogeneous edge-cloud network.

4.4 REAP: Runtime Energy-aware Adaptive Partitioning and Offloading Framework

As described in the previous sections, the main architecture of our solution is a 3-tier system made up of end(Raspberry Pi), edge (CPU-Laptop), and cloud (GPU-PC). In this architecture, the inference begins on the Pi, intermediate activations are forwarded to the laptop, then the rest to the PC, which produces the final results. The chosen split is represented as (i, j)

The partitioning scheme is defined as follows:

- The Raspberry Pi executes feature layers $0 \ldots i$
- The laptop executes feature layers $i+1 \ldots j$
- The PC executes feature layers $j+1 \ldots N-1$, followed by the classifier head

where;

$$0 \leq i < j < N_{\text{features}}$$

REAP determines at runtime how the CNN model inference workload should be partitioned across end, edge, and cloud nodes to reduce the energy consumption while maintaining an acceptable latency. During runtime, the framework is designed to periodically evaluate if the current split is still the most optimal choice given the resources at that particular time. Based on runtime measurements of computation and communication cost, the framework only changes the current partition if it predicts a better choice than the current one.

Similar to many recent solutions from related literature, our solution was designed and implemented in Python, as shown in the pseudocode from Algorithms 1 2 3 4 5 6 in the sections below.

The framework has five major algorithmic stages.

- Stage 1: Offline profiler shown in Algorithm 1
- Stage 2: Two-point link probe, Algorithm 2
- Stage 3: Candidate split latency and energy estimator as shown in Algorithm 3
- Stage 4: Best candidate split search happen in Algorithm 4
- Stage 5: Run-time inference scheduler execution probe will perform according to Algorithm 5

4.4.1 Offline profiler

Before we could determine where to partition the workload in an adaptive runtime manner, we used the offline profiler, which profiles the model in use. The profiling phase results in two main tables: the activation size table and the relative compute weight table.

The activation size table stores the size of the activation tensor in bytes. This implies that, if the model is cut after layer i , the Pi must transmit a certain number of bytes to the laptop; similarly, if we cut after layer j, the laptop must transmit a certain number of bytes to the PC. This lets the algorithm know the size of bytes to be transmitted from one node to another, if we split at a candidate point (i, j).

The profiler also measures the execution time of each layer once and then normalises the time. This weight is stored in the relative compute weight table. In this part, each layer is assigned a fraction of the total model work. Later, runtime measurements from a few partitions can be scaled to estimate unseen partitions using the profiled baselines as a reference for the estimation calculations.

The offline profiling phase records the activation sizes at every feature boundary, then assigns each layer a normalised compute weight. The algorithm is shown in the pseudocode below.

Algorithm 1 Offline profiling of a neural network $\mathcal{N}$

Require: Pretrained network $\mathcal{N}$ with ordered feature children $\mathcal{F} = (f_0, f_1, \ldots, f_{N-1})$ and classifier head H

Ensure: Per-layer activation sizes B and relative compute weights W

$x \leftarrow \text{randn}(1, 3, 224, 224)$
for $k \leftarrow 1$ to 3 **do** ▷ warmup
    $x' \leftarrow H(\mathcal{F}(x))$
end for
$x \leftarrow \text{randn}(1, 3, 224, 224)$
for $i \leftarrow 0$ to $N - 1$ **do**
    $t_0 \leftarrow \text{now}()$
    $x \leftarrow f_i(x)$
    $T[i] \leftarrow \text{now}() - t_0$
    $B[i] \leftarrow \text{numel}(x) \cdot \text{elementSize}(x)$
end for
$t_0 \leftarrow \text{now}();\ \_ \leftarrow H(x);\ T[N] \leftarrow \text{now}() - t_0$
$W[k] \leftarrow T[k] \,/\, \sum_j T[j] \quad \text{for all } k \in \{0, \ldots, N\}$
return B, W

4.4.2 Two-point link probe

After measuring the activation sizes in bytes, we need to estimate the communication cost between our nodes. For this, we use the two-point link

probe to measure the round-trip time of sending two payloads of different sizes (one large, one small), denoted as s_1 and s_2. We transmit each payload multiple times, and the average round-trip time is computed for each in order to reduce the effect of short-term timing noise. This is denoted as;

$$rtt_h(s) = \omega + \frac{s}{\beta} \tag{4.4}$$

where $rtt_h(s)$ is the round-trip for payload size s on hop h, w is the fixed communication overhead, and β is the effective throughput in bytes per second.

after obtaining the average round-trip times $\tau[s_1]$ and $\tau[s_2]$, the throughput is estimated as shown in Eq.(4.5).

$$\beta = \frac{s_2 - s_1}{\tau[s_2] - \tau[s_1]} \tag{4.5}$$

and the fixed overhead as;

$$\omega = \max\left(0,\ \tau[s_1] - \frac{s_1}{\beta}\right) \tag{4.6}$$

The resulting pair (ω, β) provides a concrete model of the link between two nodes, which can later be used to estimate the transfer time of any activation tensor resulting from a chosen split pair. This part of the algorithm is shown in the pseudocode illustration below.

Algorithm 2 Two-point link probe

Require: Hop h with round-trip function $\mathrm{rtt}_h(s)$; sizes $s_1 \ll s_2$; repeat count r

Ensure: Link model (ω, β) where ω is fixed overhead and β is throughput in bytes/second

for $s \in \{s_1, s_2\}$ **do**
    $\tau[s] \leftarrow \mathrm{mean}\big(\{\mathrm{rtt}_h(s) : k = 1, \ldots, r\}\big)$
end for
if $\tau[s_2] \leq \tau[s_1]$ **then return** previous model ▷ malformed probe; keep stale values
end if
compute β using Eq.(4.5)
$\omega \leftarrow \max\big(0,\ \tau[s_1] - s_1/\beta\big)$
return (ω, β)

4.4.3 Candidate split latency and energy estimator

As previously described, a candidate split is described by a pair of indices (i, j), where i is the index of the last feature layer executed on the Raspberry Pi, and j is the index of the last feature layer on the laptop. The PC always executes the remaining feature layers and the classifier head. Algorithm 3 takes this candidate together with the per-layer compute weights from algorithm 1, the per-node compute speeds and energy rates measured during execution, the link models from algorithm 2, and the activation byte sizes from each cut point. It returns predicted end-to-end latency, the predicted Raspberry Pi energy, and the predicted total system energy for that candidate.

The latency prediction is the sum of three computation times, one per node and two transfer times, one per link. The Raspberry Pi energy prediction is its compute time multiplied by the constant 12 W power model. The laptop and PC energy predictions multiply each node's compute time by the empirical energy rate measured from previous runs. The total energy is the sum of all three. These predictions are estimates rather than measurements; the actual values will be observed once the candidate is run in production, and any discrepancy between the estimate and the measurement informs the next re-evaluation cycle. This part of the framework is illustrated through the pseudocode below.

Algorithm 3 Estimate latency and energy of a candidate split (i, j)

Require: Candidate (i, j); per-layer compute weights W; per-node speeds $(\sigma_{\text{Pi}}, \sigma_{\text{Lap}}, \sigma_{\text{PC}})$; per-node energy rates $(\rho_{\text{Lap}}, \rho_{\text{PC}})$; link models $(\omega_{pl}, \beta_{pl})$ and $(\omega_{lp}, \beta_{lp})$; payload table B

$w_{\text{Pi}} \leftarrow \sum_{k=0}^{i} W[k]$

$w_{\text{Lap}} \leftarrow \sum_{k=i+1}^{j} W[k]$

$w_{\text{PC}} \leftarrow \sum_{k=j+1}^{N} W[k]$

$t_{\text{Pi}} \leftarrow \sigma_{\text{Pi}} \cdot w_{\text{Pi}}$; $t_{\text{Lap}} \leftarrow \sigma_{\text{Lap}} \cdot w_{\text{Lap}}$; $t_{\text{PC}} \leftarrow \sigma_{\text{PC}} \cdot w_{\text{PC}}$

$t_{pl} \leftarrow \omega_{pl} + B[i]/\beta_{pl}$

$t_{lp} \leftarrow \omega_{lp} + B[j]/\beta_{lp}$

$\hat{L} \leftarrow t_{\text{Pi}} + t_{\text{Lap}} + t_{\text{PC}} + t_{pl} + t_{lp}$

$\hat{E}_{\text{Pi}} \leftarrow P_{\text{Pi}} \cdot t_{\text{Pi}}$ ▷ P_{Pi}: fixed Pi power model (12 W)

$\hat{E}_{\text{Lap}} \leftarrow \rho_{\text{Lap}} \cdot t_{\text{Lap}}$; $\hat{E}_{\text{PC}} \leftarrow \rho_{\text{PC}} \cdot t_{\text{PC}}$

$\hat{E}_{\text{tot}} \leftarrow \hat{E}_{\text{Pi}} + \hat{E}_{\text{Lap}} + \hat{E}_{\text{PC}}$

return $(\hat{L},\ \hat{E}_{\text{Pi}},\ \hat{E}_{\text{tot}})$

4.4.4 Best candidate split search

Algorithm 4 evaluates every valid partition point and returns the one that minimises the score function defined below. A partition (i, j) is considered valid if the Raspberry Pi runs at least the minimum of 1 layer, and the laptop runs at least one layer as well.

For each valid candidate, Algorithm 4 first calls Algorithm 3 to obtain the predicted latency and energies. Then two filters are applied. The first filter rejects candidates whose predicted latency exceeds the deadline, ensuring that no candidate with an unacceptable latency can be selected, regardless of how energy-efficient it might be. The second filter rejects candidates whose normalised score exceeds the score of the static baseline split; this ensures that the adaptive framework can never produce a result worse than the static baseline that it is intended to improve upon. Candidates who pass both filters are scored using the weighted sum

$$S = w_{\text{Pi}} \cdot \frac{\hat{E}_{\text{Pi}}}{n_{\text{Pi}}} + w_{\text{tot}} \cdot \frac{\hat{E}_{\text{tot}}}{n_{\text{tot}}} + w_{\text{lat}} \cdot \frac{\hat{L}}{n_{\text{lat}}} \tag{4.7}$$

where the weights are chosen by the user, and the normalisation anchors are average measured values from a set of probe splits run at startup. Normalisation makes the score dimensionless and ensures that each weight

Algorithm 4 Find best candidate split

Require: Rates and link models (as in Alg. 3); weights $\mathbf{w} = (w_{\text{Pi}}, w_{\text{tot}}, w_{\text{lat}})$; normalization anchors $\mathbf{n} = (n_{\text{Pi}}, n_{\text{tot}}, n_{\text{lat}})$; baseline score $S^{\star}$; deadline $L_{\max}$; minimum Pi layers m; current split c

$(s^{\star}_{\text{best}},\, S_{\text{best}}) \leftarrow (\textsc{None},\, +\infty)$
for $(i, j) \in \{(i, j) : m-1 \leq i < j < N\}$ **do**
  if $(i, j) = c$ **then continue** ▷ exclude currently-running split
  end if
  $(\hat{L}, \hat{E}_{\text{Pi}}, \hat{E}_{\text{tot}}) \leftarrow \textsc{Estimate}(i, j, \ldots)$ ▷ Alg. 3
  if $L_{\max} > 0$ **and** $\hat{L} > L_{\max}$ **then continue** ▷ deadline pre-filter
  end if
  $S \leftarrow w_{\text{Pi}} \cdot \dfrac{\hat{E}_{\text{Pi}}}{n_{\text{Pi}}} + w_{\text{tot}} \cdot \dfrac{\hat{E}_{\text{tot}}}{n_{\text{tot}}} + w_{\text{lat}} \cdot \dfrac{\hat{L}}{n_{\text{lat}}}$
  if $S > S^{\star}$ **then continue** ▷ must beat static baseline
  end if
  if $S < S_{\text{best}}$ **then**
    $(s^{\star}_{\text{best}},\, S_{\text{best}}) \leftarrow ((i, j),\, S)$
  end if
end for
return $s^{\star}_{\text{best}}$

has a comparable influence on the results regardless of the absolute magnitudes of the underlying quantities. This part of the framework is illustrated in the pseudocode shown below in algorithm 4.

All the phases of the framework algorithm described through algorithm 1, 2 & 3 are clearly illustrated in diagram 4.1 above. The diagram shows the process flow from all three and how we proceed from one to the other.

4.4.5 Adaptive distributed inference scheduler

The adaptive scheduler is designed and implemented in two main phases

- phase 1a: Measures the initial split and defines the baseline threshold to achieve the improvements to.

- phase 1b: Run three additional probe splits, which are chosen automatically from the network's layer count.
- phase 1c: Use the measured data to estimate all possible splits and choose the best starting one
- phase 2: Run, re-evaluate the resource availability and environmental conditions in the network and switch only when a new split that beats the current one by more than 3% is found.

In phase 1a of the scheduler, the algorithm starts with a user-defined initial split. It then runs the split for the first few inferences to compute and record the nodes' latency and energy consumption. These data are used as the baseline reference point threshold to be beaten. This part is important because it allows the scheduler to record and learn from actual runs of the model in use. The two main outcomes from this phase are the algorithm's reference operating point and a clear threshold to beat, which subsequent estimated candidates are compared against as we later adaptively improve the partitioning and offloading processes.

In phase 1b, after measuring the baseline, we probe three new splits: a PC-heavy, a balanced, and a PI-heavy split to expose different workload distributions to the scheduler. The layer count per node is calculated automatically based on the model's layer count. The purpose is to help the estimator infer device-specific execution rate parameters across a wide operating range. This phase is important because the splits are used to identify system behaviour when under different workload weights. Both phase 1a and 1b as described above can be seen clearly illustrated and simplified in figure 4.1

In phase 1c, armed with the baseline and probe data, the scheduler computes a normalised objective scaling factor, which is the main scheduler's decision policy.

$$S(i,j) = w_\pi \frac{\hat{E}_\pi(i,j)}{N_\pi} + w_{\text{tot}} \frac{\hat{E}_{\text{tot}}(i,j)}{N_{\text{tot}}} + w_{\text{lat}} \frac{\hat{L}(i,j)}{N_{\text{lat}}} \tag{4.8}$$

where: $\lambda_\pi,\ \lambda_{\text{tot}},\ \lambda_{\text{lat}}$ are user defined weights.

- λ_π weight for the PI energy
- λ_{tot} weight for the total system energy
- λ_{lat} weight for the latency

By calculating the estimated latency and energy consumption of each candidate split, and their corresponding objective weight as shown in figure 4.1, we can compare whether the new split can better meet the optimisation objective, which is to reduce the energy consumption without violating the latency deadline window. On the objective scale, we give both pi and overall energy more weight compared to latency. In most instances, the default weight for pi energy was set at the range of 0.9 to 0.6, while that of total energy cost was at 0.2 to 0.3. The latency weight always ranges between 0.1 and 0.3 at most.

In phase 2, the scheduler enters steady-state operation and runs the selected split for R_{steady} inferences at a time. At the end of each window, it refits the per-layer compute speeds using both the phase 1 data and the most recent window combined, re-probes both network links using Algorithm 2, and re-runs the candidate search of Algorithm 4 with the updated parameters. If the new candidate's score improves on the current split by at least the threshold (set to 3%), the scheduler switches to it. If the current split had violated the latency deadline $L_{\max}$ during the window, the switch is forced regardless of the improvement margin. If the deadline is violated but no better candidate exists, the scheduler falls back to the static baseline c_0 as the safest known configuration. This periodic re-evaluation is what makes the framework adaptive: changes in network bandwidth or node compute load are captured by the re-probing and re-fitting steps and reflected in the next candidate selection. This logic and process is what we called the flow state and can be seen as illustrated in figure 4.2.

The 3% threshold was chosen to prevent the scheduler from repeatedly switching between splits that only differ marginally in score. Since the profiling measurements for energy and latency carry some natural run-to-run variance, a very small improvement could simply reflect noise rather than a genuinely better configuration. Setting the bar at 3% filters out these fluctuations while still being low enough to let the system react when network or device conditions change in a meaningful way. This phase of the framework is shown in the pseudocode illustrations in Alg 5 showing the first initialisation part and Alg 6 showing the steady flow state part which has the periodic re-evaluation.

Algorithm 5 Adaptive distributed inference scheduler – Initialisation

Require: Model $\mathcal{N}$; initial split c_0; weights $\mathbf{w}$; deadline $L_{\max}$; R_{profile} baseline runs; R_{probe} runs per probe split; R_{steady} runs per re-evaluation window; warmup k_{warm}; switch improvement threshold θ

Phase 1a: baseline run (defines threshold to beat)

$(B, W) \leftarrow$ PROFILE($\mathcal{N}$) ▷ Alg. 1
$\mathcal{D}_{\text{base}} \leftarrow \emptyset$; $c \leftarrow c_0$
for $r \leftarrow 1$ to R_{profile} **do**
    $s \leftarrow$ RUNINFERENCE(c)
    if $r > k_{\text{warm}}$ **then** $\mathcal{D}_{\text{base}} \leftarrow \mathcal{D}_{\text{base}} \cup \{s\}$
    end if
end for
$(b_{\text{Pi}}, b_{\text{tot}}, b_{\text{lat}}) \leftarrow$ mean energies and latency over $\mathcal{D}_{\text{base}}$

Phase 1b: probe reference splits to ground per-layer rates

$\mathcal{P} \leftarrow$ PROBESPLITS(N, m) ▷ 3 splits at fifths of the feature range
$\mathcal{D}_{\text{probe}} \leftarrow \emptyset$
for all $p \in \mathcal{P} \setminus \{c_0\}$ **do**
    for $r \leftarrow 1$ to R_{probe} **do**
        $s \leftarrow$ RUNINFERENCE(p)
        if $r > k_{\text{warm}}$ **then** $\mathcal{D}_{\text{probe}} \leftarrow \mathcal{D}_{\text{probe}} \cup \{s\}$
        end if
    end for
end for

Phase 1c: fit rates, probe links, choose starting split

$\mathbf{n} \leftarrow$ mean energies and latency over $\mathcal{D}_{\text{probe}}$ ▷ normalization independent of c_0
$S^\star \leftarrow w_{\text{Pi}}(b_{\text{Pi}}/n_{\text{Pi}}) + w_{\text{tot}}(b_{\text{tot}}/n_{\text{tot}}) + w_{\text{lat}}(b_{\text{lat}}/n_{\text{lat}})$
$\sigma, \rho \leftarrow$ FITRATES($\mathcal{D}_{\text{base}} \cup \mathcal{D}_{\text{probe}}$, W)
$(\omega_{pl}, \beta_{pl}), (\omega_{lp}, \beta_{lp}) \leftarrow$ LINKPROBE() ▷ Alg. 2
$c \leftarrow$ FINDBEST($\sigma, \rho, \omega, \beta, \mathbf{w}, \mathbf{n}, S^\star, L_{\max}, m$, NONE) ▷ Alg. 4

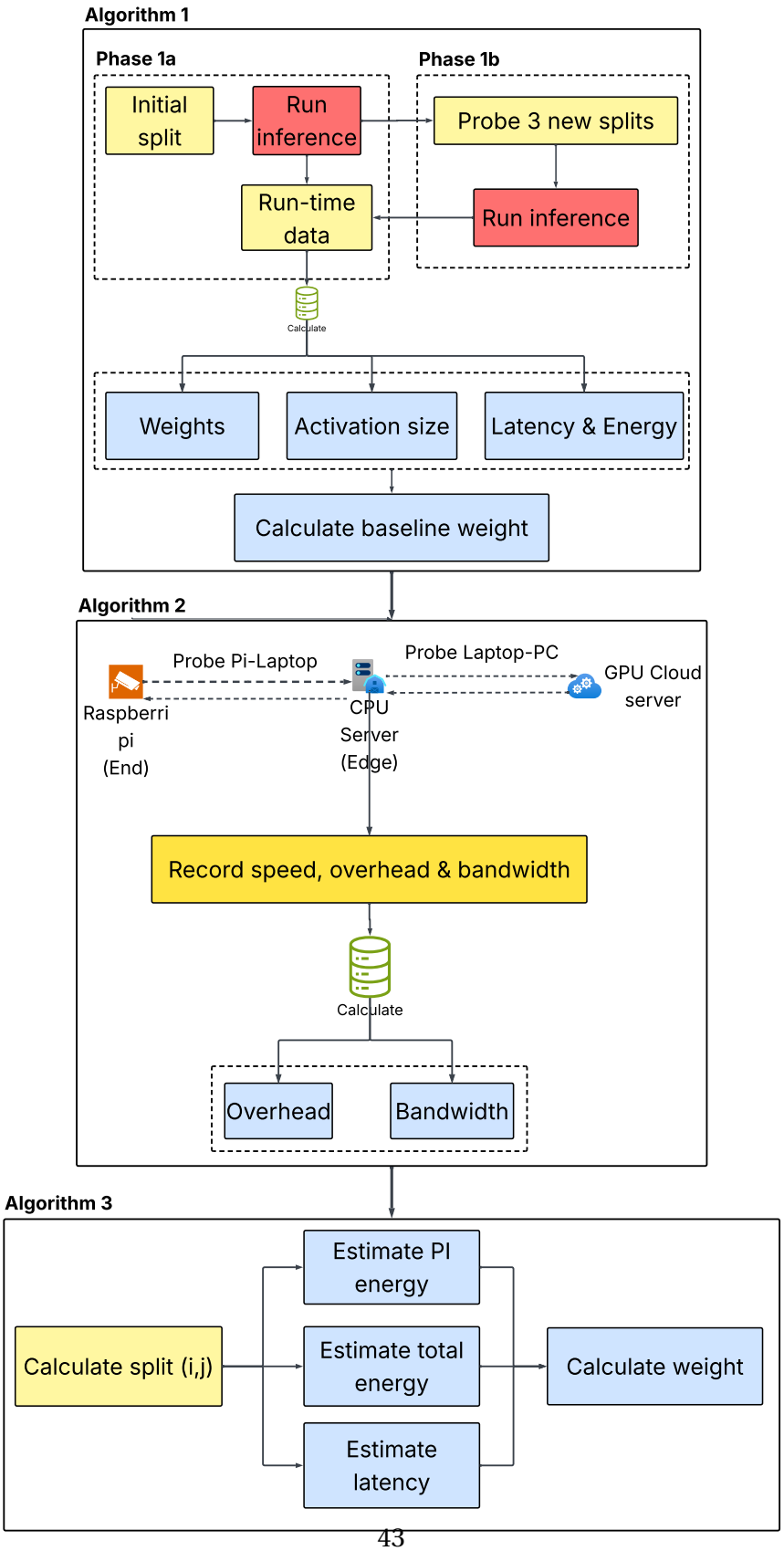

Figure 4.1: Algorithm 1, 2 and 3 flow diagram

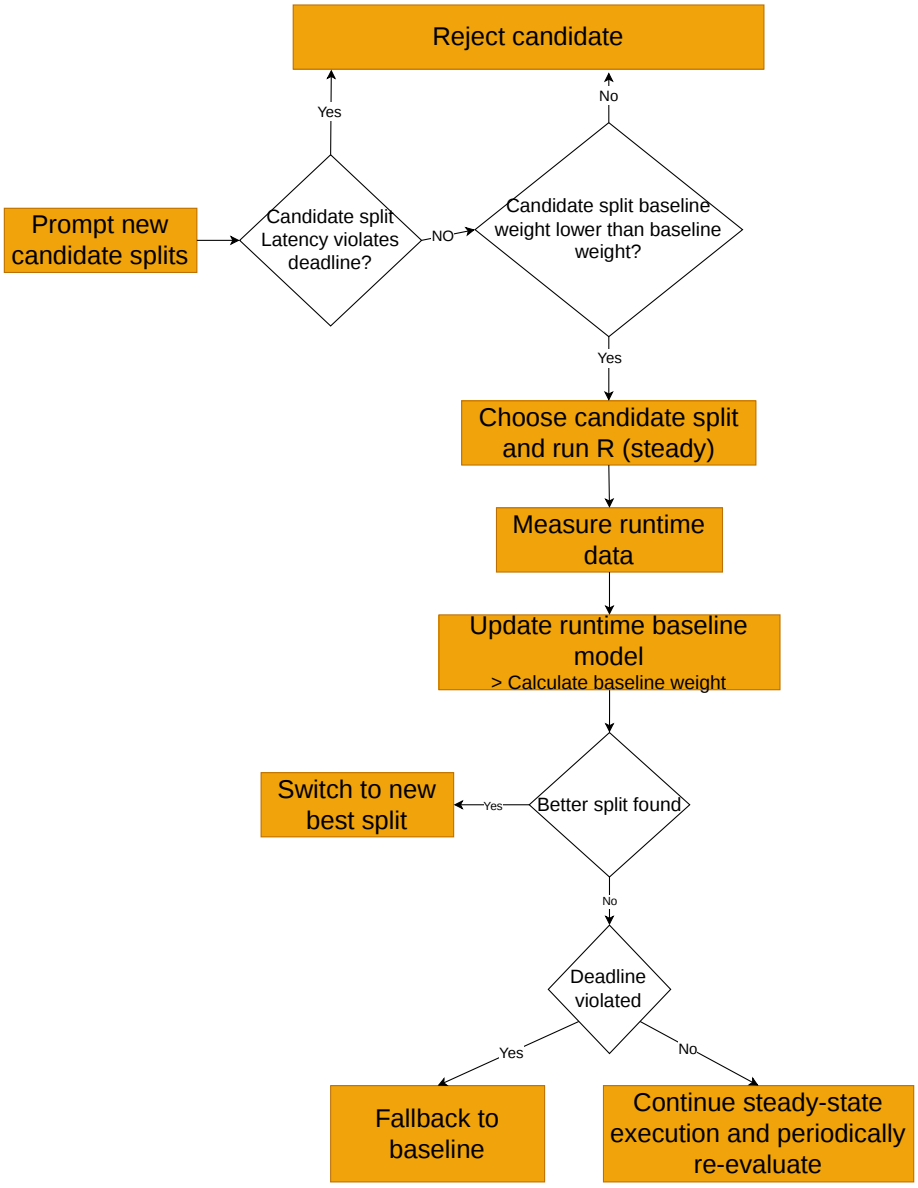

Figure 4.2: Scheduler flow steady state loop diagram

Algorithm 6 Adaptive distributed inference scheduler – Steady state

Require: Current split c; baseline split c_0; weights $\mathbf{w}$; deadline $L_{\max}$; R_{steady} runs per re-evaluation window; warmup k_{warm}; switch improvement threshold θ; Phase-1 data $\mathcal{D}_{\text{phase1}} = \mathcal{D}_{\text{base}} \cup \mathcal{D}_{\text{probe}}$

Phase 2: steady state with periodic re-evaluation

while budget not exhausted **do**
  $\mathcal{W} \leftarrow \emptyset$
  for $r \leftarrow 1$ to R_{steady} **do**
    $s \leftarrow \text{RUNINFERENCE}(c)$
    if $r > k_{\text{warm}}$ **then** $\mathcal{W} \leftarrow \mathcal{W} \cup \{s\}$
    end if
  end for
  $\bar{L} \leftarrow$ mean latency over $\mathcal{W}$
  $\sigma, \rho \leftarrow \text{FITRATES}(\mathcal{D}_{\text{phase1}} \cup \mathcal{W}, W)$ ▷ Phase-1 data keeps rates grounded
  $(\omega, \beta) \leftarrow \text{LINKPROBE}()$
  $c' \leftarrow \text{FINDBEST}(\sigma, \rho, \omega, \beta, \mathbf{w}, \mathbf{n}, S^\star, L_{\max}, m, c)$
  $S_c, S_{c'} \leftarrow \text{SCORE}(c, \ldots), \text{SCORE}(c', \ldots)$
  $\Delta \leftarrow (S_c - S_{c'})/S_c$ ▷ relative improvement
  deadlineHit $\leftarrow (L_{\max} > 0 \wedge \bar{L} > L_{\max})$
  if deadlineHit $\wedge\, c' \neq c$ **then**
    $c \leftarrow c'$ ▷ FORCED switch (deadline violation)
  else if $c' \neq c \wedge \Delta \geq \theta$ **then**
    $c \leftarrow c'$ ▷ NORMAL switch
  else if deadlineHit $\wedge\, c \neq c_0$ **then**
    $c \leftarrow c_0$ ▷ fall back to static baseline
  end if
end while

5

Results

This chapter presents the quantitative data collected during the experimental phase described in Chapter 4. The evaluation consisted of three sets of measurements. First, single-device baselines recorded for each of the three models running entirely on the Raspberry Pi, on the laptop CPU, and on the PC GPU. Second, the same models executed using a three-node static partitioning scheme in which split points are fixed before the run begins. Third, the models executed using the adaptive partitioning framework, in which the scheduler selects split points at runtime. All measurements used 500 inference passes with a dummy input tensor of shape 1 x 3 x 224 x 224, and were repeated 10 times to obtain a more accurate average. Energy consumption was recorded using the in-process instrumentation where the Raspberry Pi used a constant 12 W power model multiplied by measured compute time, laptop used Intel RAPL interface and PC through the NVIDIA NVML library as described in the experimental procedure. Latency was measured using the Python time library.

5.1 Single-Device Baselines

Tables 5.1, 5.2, and 5.3 report the average per-inference latency and energy consumption when each model were executed entirely on a single device, with no network transfers.

The three devices differ by roughly two orders of magnitude in both latency and energy cost. The PC GPU completes VGG-16 in 1.16 ms and AlexNet in

Table 5.1: PC (GPU) baseline results

Model	Average Latency (ms)	Average Energy (J)
VGG	1.164	0.03686
AlexNet	0.83	0.0240
MobileNetV2	4.175	0.0923

Table 5.2: Laptop (CPU) baseline results

Model	Average Latency (ms)	Average Energy (J)
VGG	169.908	2.5486
AlexNet	20.988	0.3150
MobileNetV2	15.954	0.2394

0.83 ms, while the Raspberry Pi requires 666.87 ms and 132.40 ms, respectively. MobileNetV2 shows a smaller gap because the model is specifically designed for mobile inference: the Pi completed it in 71.90 ms and the GPU in 4.18 ms, a difference of 17 times rather than the 570 times observed for VGG-16. The laptop CPU sits between the two extremes, with per-inference latency of 169.91 ms for VGG-16, 20.99 ms for AlexNet, and 15.95 ms for MobileNetV2. The same ordering holds for energy consumption: the Raspberry Pi consumes between 0.86 J (MobileNetV2) and 8.00 J (VGG-16) per inference, the laptop between 0.24 J and 2.55 J, and the GPU between 0.024 J and 0.0092 J.

5.2 Static Partitioning Baseline

Table 5.4 reports the per-inference results when the three models were executed across the three-node chain using a fixed partitioning scheme. For VGG-16, the Raspberry Pi executed feature layers 0 through 10, the laptop executed feature layers 11 through 30, and the PC executed the classifier head. For AlexNet, the Raspberry Pi executed feature layers 0 through 9, and the laptop executed feature layers 10 through 12 together with the adaptive average pooling module, and the PC executed the classifier. For MobileNetV2, the Raspberry Pi executed the first 10 blocks of the feature stack, the laptop executed the remaining 9 blocks, and the PC executed the

Table 5.3: Raspberry Pi baseline results

Model	Average Latency (ms)	Average Energy (J)
VGG	666.87	8.0024
AlexNet	132.40	1.5888
MobileNetV2	71.90	0.8628

Table 5.4: Total average performance data across static partitioning

Model	Pipeline Latency (ms)	Edge (Pi) Energy (J)	Fog (Laptop) Energy (J)	Cloud (PC) Energy (J)	Total Energy (J)
VGG	525.142	2.297	2.491	0.905	5.693
AlexNet	78.148	0.237	0.082	0.356	0.675
MobileNetV2	98.457	0.624	0.268	0.027	0.919

pooling and classifier head. These specific cut points were chosen to place roughly one-third of the feature extraction work on each node.

Relative to the Raspberry Pi single-device baseline, static partitioning reduced the end-to-end latency for VGG-16 by 21.2 percent (from 666.87 ms to 525.14 ms) and for AlexNet by 41.0 percent (from 132.40 ms to 78.15 ms). For MobileNetV2, the chain introduced additional latency overhead rather than reducing it: the end-to-end time grew from 71.90 ms on the Pi alone to 98.46 ms across the chain, an increase of 36.9 percent. The total system energy for the chain is 5.69 J for VGG-16, 0.68 J for AlexNet, and 0.92 J for MobileNetV2. The distribution of energy between the three nodes differed markedly between models. For VGG-16, the Raspberry Pi consumed 2.30 J, the laptop consumed 2.49 J, and the PC consumed the remaining 0.91 J. For AlexNet and MobileNetV2, the Raspberry Pi remained the largest contributor per inference (0.24 J and 0.62 J, respectively), while the laptop contribution was considerably smaller (0.08 J and 0.27 J).

5.3 Adaptive Partitioning

Table 5.5 reports the same per-inference metrics when the adaptive partitioning scheduler controled the split points. For each model, the scheduler first ran a profiling phase consisting of 50 inferences at the initial static

Table 5.5: Total performance average results from our proposed partitioning framework

Model	Pipeline Latency (ms)	Edge (Pi) Energy (J)	Fog (Laptop) Energy (J)	Cloud (PC) Energy (J)	Total Energy (J)
VGG-16	491.855	1.489	1.235	0.930	3.654
AlexNet	60.233	0.078	0.097	0.259	0.434
MobileNetV2	84.479	0.494	0.078	0.099	0.670

split, followed by 15 inferences at each of three probe splits selected automatically from the model's layer count. The scheduler then computed the per-layer compute rates and per-link bandwidth parameters, selected the split that minimizes the weighted score function, and entered the steady state phase with re-evaluation every 100 inferences.

Under adaptive partitioning using REAP, the total system energy was 3.65 J for VGG-16, 0.43 J for AlexNet, and 0.67 J for MobileNetV2. End-to-end latency was 491.86 ms for VGG-16, 60.23 ms for AlexNet, and 84.48 ms for MobileNetV2. For all three models, the REAP framework consumed less energy and completed inference faster than the static partitioning baseline reported in Table 5.4. The per-node energy distribution also changed between the two configurations. For VGG-16, the Raspberry Pi energy contribution dropped from 2.30 J under static partitioning to 1.49 J under adaptive partitioning, and the laptop contribution dropped from 2.49 J to 1.24 J. For AlexNet, the Raspberry Pi contribution dropped from 0.24 J to 0.08 J. For MobileNetV2, the Raspberry Pi contribution dropped from 0.62 J to 0.49 J, while the PC contribution rose from 0.03 J to 0.10 J

5.4 Comparison of Static and Adaptive Partitioning

Table 5.6 shows the results of the adaptive framework as percentage reductions in latency and total system energy relative to the static partitioning baseline.

The energy reduction ranges from 27.09 percent for MobileNetV2 to 35.82 percent for VGG-16, and the latency reduction ranges from 6.34 percent for VGG-16 to 22.92 percent for AlexNet. For all three models, the energy reduction is larger in magnitude than the latency reduction. Figure 5.1 and Figure 5.2 show the absolute values for latency and total system energy side

Table 5.6: overall percentage improvement achieved by adaptive partitioning over static partitioning

Model	Latency Reduction (%)	Energy Reduction (%)
VGG-16	6.34	35.82
AlexNet	22.92	35.70
MobileNetV2	14.20	27.09

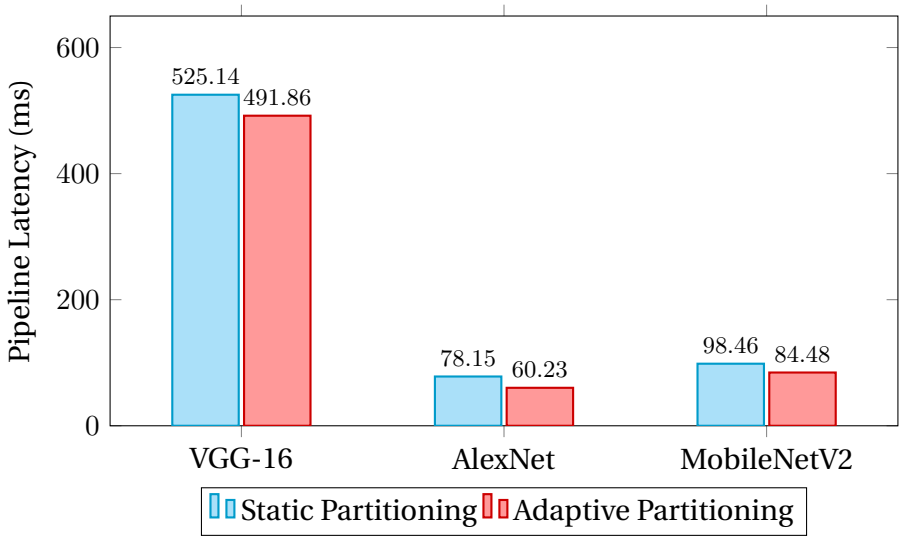

Figure 5.1: Pipeline Latency Comparison: Static vs. Adaptive Partitioning

by side for the static and adaptive configurations, and Figure 5.3 shows the relative reductions as percentages.

The three models differ in how the adaptive framework distributed the savings between energy and latency. VGG-16 shows the largest absolute energy reduction (2.04 J per inference) and the largest percentage energy reduction (35.82 percent), while also showing the smallest latency reduction (33.28 ms, or 6.34 percent). AlexNet shows both a large energy reduction (0.24 J, or 35.70 percent) and the largest latency reduction (17.92 ms, or 22.92 percent). MobileNetV2 shows the smallest percentage energy reduction of the three (27.09 percent) and a latency reduction of 14.20 percent. Across all three models, the adaptive framework produces measurable reductions in both metrics relative to the static chain baseline, with energy reductions consistently exceeding latency reductions in percentage terms.

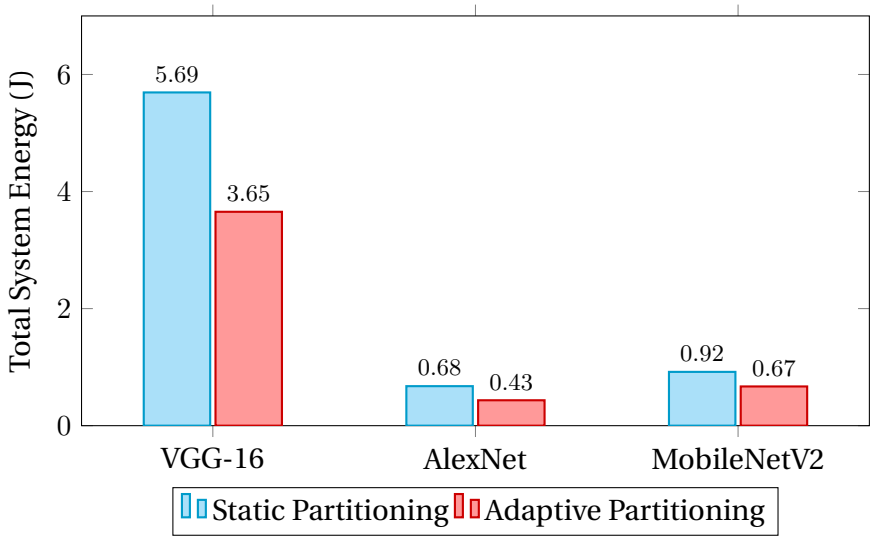

Figure 5.2: Total System Energy Comparison: Static vs. Adaptive Partitioning

Table 5.7: Variability of results across experimental runs (mean ± standard deviation)

Model	Approach	Pipeline Latency (ms)	Total Energy (J)
VGG-16	Static	525.142 ± 11.31	5.693 ± 0.158
	Adaptive	491.855 ± 17.00	3.654 ± 0.202
AlexNet	Static	78.148 ± 11.81	0.675 ± 0.091
	Adaptive	60.233 ± 4.65	0.434 ± 0.082
MobileNetV2	Static	98.457 ± 1.65	0.919 ± 0.025
	Adaptive	84.479 ± 2.05	0.670 ± 0.063

Table 5.7 above shows the standard deviation of the per-run averages across the experimental repetitions for both the static and adaptive configurations.

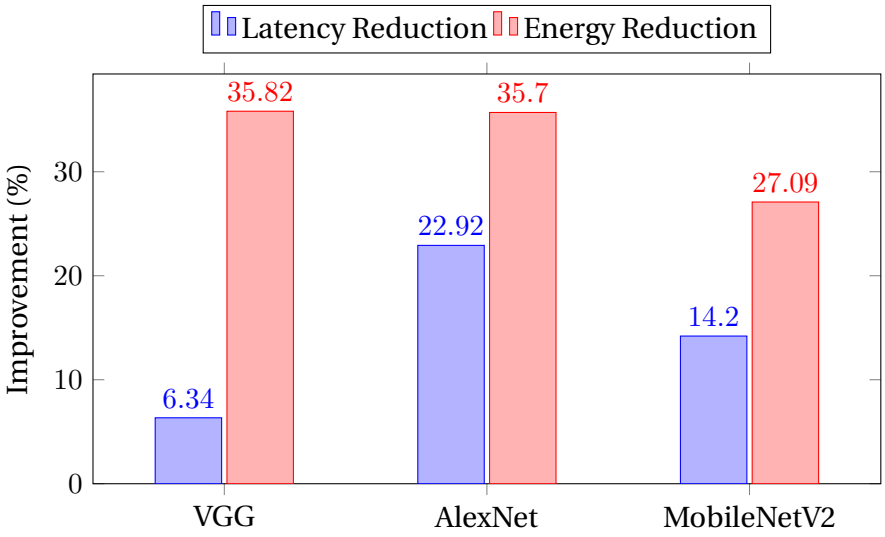

Figure 5.3: Relative improvement achieved by adaptive partitioning over static partitioning.

6

Discussion

To answer our research question, to what extent can an adaptive model partitioning algorithm reduce the energy consumption of resource-constrained devices without violating real-time latency constraints in a heterogeneous edge-cloud network? We will address it in two main parts: the energy-related part first, followed by the latency part.

6.1 Energy reduction

To address the energy focus part of the research question, we needed to answer the sub-question: to what extent can an adaptive model partitioning algorithm reduce the energy consumption of resource-constrained devices? To fully cover this, in the results, we measured and compared the energy consumption on each independent device (standalone), statically and lastly on our adaptively partitioned framework. The results from the experiment showed that our proposed adaptive framework resulted in a significant energy reduction across all models: VGG16, AlexNet and MobileNetV2. Using REAP in comparison to the static partition baseline, the energy was reduced by 35.82% in VGG16, 35.70% for AlexNet and by 27.09% for MobileNetV2. This effectively shows that the adaptive framework approach is significantly better in preserving energy than other approaches, such as static. The significance of this outcome is that it shows that the employment of an adaptive partitioning and offloading approach across a real-life heterogeneous network outside of a simulation environment can

realistically reduce the energy consumption across the network despite the unpredictability of such networks and systems. It also shows and implies that an energy reduction of 27% to 35% on a larger scale can lead to a significant energy reduction overall.

6.2 Latency reduction

To address the latency focus part of the research, we had to answer the subquestion: Can energy reduction be achieved without violating real-time latency? The results according to the previous table 5.6 showed that, not only were we able to achieve energy reduction without violating the initial real-time latency, but were also able to achieve a reduction of the network latency. From the results, we observed that the network end-to-end latency was reduced by 6.34% in VGG16, 22.92% in AlexNet and by 14.20% for MobileNetV2. This was significant because it clearly demonstrated that, even though at times the process of task offloading may introduce additional latency overhead, it is possible to achieve latency reduction while simultaneously reducing the energy consumption. This scenario was observed when dealing with the MobileNetV2 model. When ran solely on Raspberry Pi, the latency was 71.90ms, when statically partitioned across the three nodes, the latency went up to 98.457ms, and when on the adaptively partitioned framework, it went back down to 84.479ms. This showed that the process of offloading, along with the unique architecture of MobileNetV2, led to the introduction of additional latency overhead. However, our adaptive framework reduced this overhead resulting from the static partition of the model's workload.

6.3 Model's evaluation

The three models were different in a meaningful way in how they respond to adaptive partitioning, and these differences are explained by the architectural properties of each network and the hardware characteristics of the Raspberry Pi

- VGG-16

  Showed the largest absolute energy saving (2.04 J) and smallest latency reduction 6.34%. This is because VGG-16 was the most computationally expensive network in this thesis, it took 666.82 ms and consumed 8.00 J when executed entirely on the Raspberry Pi. The

high cost in Raspberry Pi gave the scheduler substantial headroom to reduce energy by offloading layers to faster nodes. However, VGG-16's activation tensors are large, the output of the first convolutional block alone was 3.2 MB, which meant that pushing the cut point earlier in the network increased the data that must be serialised and transferred across the network link. The scheduler was therefore constrained in how aggressively it could offload. Moving too many layers from raspberry Pi saved compute energy but introduced transfer latency that partially offset the gain.

- AlexNet

  Showed the highest latency reduction of 22.92% alongside strong energy reduction of 35.70%. AlexNet's first convolutional layer uses an 11x11 kernel with a stride of 4, which aggressively downsamples the input and produces activation tensors that are considerably smaller than VGG-16's at comparable cut points. This gave the scheduler more freedom to place the cut point early in the network without paying a large transfer cost. The combination of meaningful raspberry Pi compute savings and low transfer overhead resulted in improvements on both metrics simultaneously rather than trading one for the other.

- MobileNetV2

  Showed the smallest energy reduction of 27.09% and latency slower than raspberry Pi-alone (84.48 ms vs 71.9 ms). MobileNetV2 was specifically designed for resource-constrained mobile devices and uses depthwise separable convolutions that are computationally cheap. Raspberry PI could already execute the entire model inference in 71.90 ms at 0.86 J, which left the scheduler with relatively little energy headroom to exploit compared to VGG-16 or AlexNet. At the same time, the overhead of serialising the intermediate activations, transferring them across two network hops, and deserialising them on the receiving node exceeds the compute time saved by offloading. The adaptive framework reduced this overhead compared to the static baseline (from 98.46 ms to 84.48 ms) by selecting splits that minimise the transfer cost, but it could not fully eliminate the network overhead. This result suggested that for lightweight models where end devices such as raspberry Pi are already fast, the value of distributed inference lies more in reducing the end devices' energy than reducing latency.

6.4 Statistical consistency of the results

As shown in Table 5.7, for all three models, the energy distributions of the static and adaptive configurations showed no overlap, confirming that the reported energy reductions were not artifacts of measurement noise. For latency, AlexNet and MobileNetV2 showed clear separation, while VGG-16 showed marginal overlap consistent with the modest 6.34 percent improvement explained by the high transfer cost of its large activation tensors. A secondary observation was that for AlexNet, the adaptive framework produced noticeably lower latency variance than the static baseline (4.65 ms vs 11.81 ms), which indicated that periodic re-evaluation not only improved average performance but also increased consistency by reacting to runtime fluctuations.

6.5 Adaptivity in edge cases

To test out how adaptive the algorithm was, during the experiment, network throttling was introduced from the command line for Tailscale. It was used on the laptop to artificially reduce the data transmission speed to the PC node and see how well the adaptive framework would handle it. Since most models' partitioning started early, due to the high internet speeds and PC's ability to compute layers much faster when the laptop node and PC node speed were reduced, the scheduler detected the bandwidth drop through the link probe and migrated to splits with smaller WAN payloads. when normal speed was restored, it migrated back to the original splits. This was the strongest evidence of adaptivity that the framework showcases. It was not only picking the best static split at startup, but also reacted to environmental changes during operation.

Specifically, when the laptop-to-PC link was throttled to approximately 5 Mbit/s using the Linux traffic control utility, the scheduler shifted from a split that sent 1.6 MB across the WAN to one that sent only 98 KB by keeping more layers on the laptop and sending a smaller activation tensor to the PC. When the throttle was removed, and the link returned to its normal speed of approximately 40 MB/s, the scheduler detected the bandwidth recovery on the next re-evaluation cycle and migrated back to the original split that utilised the PC more heavily. The transition in both directions occurred within a single re-evaluation window of 100 inferences, which corresponds to approximately 50-60 seconds of wall-clock time depending on the model. This demonstrated that the framework's adaptation was not

merely theoretical but operated on a timescale that is practical for real deployments where network conditions fluctuate.

6.6 Ethical Considerations

This thesis did not involve human subjects, personal data, or privacy-sensitive information, as all inference tasks were executed using randomly generated dummy input tensors. The primary ethical implication of this work concerned energy consumption and environmental sustainability. As AI inference workloads scale across large IoT deployments, the cumulative energy cost becomes a meaningful concern, and the 27% to 35% energy reductions demonstrated in this thesis suggest that adaptive partitioning can contribute to more resource-efficient AI systems at scale. A secondary consideration concerned reliability in safety-critical contexts. Although the framework included safeguards such as a deadline pre-filter and fallback mechanism to the static baseline, the system has only been validated in a controlled testbed environment. So, deploying it in latency-critical applications such as autonomous driving or medical monitoring systems would require additional validation beyond the scope of this thesis. No external funding or conflicts of interest influenced the design or reporting of the results.

6.7 Conclusion

Due to the limited research work focusing objectively on reducing energy consumption in AI workload partitioning and offloading in a heterogeneous edge-cloud network, and the dominating use of simulation approaches in the available works. This thesis explored and investigated the extent to which an adaptive model partitioning algorithm can reduce energy consumption of resource-constrained devices without violating real-time latency constraints in a heterogeneous edge-cloud network. The work focused on implementing and testing an adaptive partitioning framework in a real-world experiment. To create a heterogeneous edge-cloud environment, we used a Raspberry Pi as an end node, a CPU laptop as an edge node and a GPU server PC as the cloud node. Using models such as VGG16, AlexNet and MobileNetV2, the adaptive partitioning and offloading framework was tested and validated. The results showed that the algorithm achieved up to 35% energy reduction for both VGG16 and AlexNet, and a 27% reduction in MobileNetV2. These results showed that an adaptive partitioning

solution can significantly reduce energy consumption while successfully reducing or maintaining the overall latency.

Despite the difference in the optimisation objective, in comparison to other related works, these results aligned with some of the most recent proposed solutions. Li et al. [16] adaptive model partitioning and pruning framework achieved an average inference reduction of 27.313%. Similarly, Chen et al. [9] EdgeCI achieved an inference latency improvement of 34.72% to 43.52%. In another energy-related work, Fang et al. [19] achieved substantial power efficiency using a Deep Reinforcement Learning (DRL) content offloading solution. Consequently, this makes our new framework REAP one of the most recent solutions that focuses on energy consumption instead of latency and was able to achieve similar percentage improvements, hence addressing a very essential gap and contributing to the current knowledge. Additionally, unlike other partitioning and offloading solutions that use techniques such as RL, parallel inference mechanisms and more, our solution used a single objective function as the main decision policy and could be considered more adaptive due to its constant periodic evaluation of the environmental conditions, factors, and resources at runtime. This shows that, regardless of the optimisation objective, adaptive partitioning and offloading can be a great solution for efficiently managing computational load balance and resources in a heterogeneous edge-cloud network.

6.8 Limitations

Despite the positive results from the experiment, some limitations are worth mentioning. One of the potential limitations is that the partitioning and offloading framework was tested only on CNN models and not on other architectures such as transformers, RNN or Encoder-Decoder models. This, as a result, only limits it to a specific category of DNN models and hence cannot be fully considered generalised. Another limitation is that the performance and results of our new framework have not been compared hand-in-hand to other state-of-the-art partitioning and offloading frameworks such as EdgeCI, DeepThing or any other that optimises energy consumption. This would give our framework a good comparative analysis and a full view of its performance when compared with other similar solutions.

Additionally, the energy measurement approaches and methods used for both the Raspberry Pi and the CPU laptop can be considered as estimates

rather than accurate and actual measures. Raspberry Pi energy consumption was estimated using a fixed constant of 12 W instead of a direct measurement with an external power meter. This was necessary because the Raspberry Pi does not provide a built-in energy counter equivalent to Intel RAPL on the laptop or NVIDIA NVML on the GPU. As a result, Pi energy values should be interpreted as comparative estimates rather than as precise physical power measurements. However, because the same constant-power model was applied consistently to both the static and adaptive partitioning experiments, the comparison between approaches remains valid, any reduction in raspberry Pi execution time leads to a proportional reduction in estimated its energy. Nevertheless, future work should replace this approximation with direct hardware-based power measurement, such as an inline USB-C power meter or an external power sensor, to improve the accuracy of the absolute energy values.

Lastly, for the inferences, we used dummy inputs instead of real image inputs to the models. This choice was made because the primary objective of the thesis was to evaluate system-level performance metrics such as latency, energy consumption, and partitioning behaviour instead of accuracy. Since CNN inference cost is mainly determined by the model architecture, layer operations, and input tensor dimensions, dummy tensors with the correct shape still produce representative computational workloads and activation sizes for evaluating partitioning and offloading performance. Therefore, using real images would introduce additional variables such as dataset loading and preprocessing, which could make it harder to isolate the direct impact of the adaptive partitioning algorithm. Therefore, the use of dummy data was appropriate for measuring the runtime behaviour of the proposed framework.

6.9 Future works

Having achieved the research goal to some extent, there is still considerate room for improvement to improve the adaptability and efficiency of the partitioning and offloading framework. Some of the future works and improvements would include attempting to extend and generalise the framework to work on other DNN model architectures mentioned in the previous section. Additionally, in a large deployment environment and scale, the energy measurement methods could be done using other approaches that would provide more accurate real energy consumption data. Lastly, for future work, we would attempt a multi-objective function approach rather

than the current single objective function in our solution. This would enable us to measure more metrics, such as temperature, memory usage, and current GPU/CPU load, determine how they influence the energy consumption of the network and use them to objectively reduce it.

6.10 AI-Tools

In this research, AI tools like ChatGPT, Claude and Gemini were used mostly for debugging Python scripts and fixing errors received during testing. Along with other tools such as Grammarly, It was also used for grammar checks, spell checks, and other minor errors.

Bibliography

[1] Harald Edquist, Peter Goodridge, and Jonathan Haskel. “The Internet of Things and economic growth in a panel of countries”. In: *Economics of Innovation and New Technology* 30.3 (2021), pp. 262–283.

[2] Haiming Chen, Wei Qin, and Lei Wang. “Task partitioning and offloading in IoT cloud-edge collaborative computing framework: a survey”. In: *Journal of Cloud Computing* 11.1 (2022), p. 86.

[3] Jianli Pan and James McElhannon. “Future Edge Cloud and Edge Computing for Internet of Things Applications”. In: *IEEE Internet of Things Journal* 5.1 (2018), pp. 439–449. DOI: 10.1109/JIOT.2017.2767608.

[4] Mike O. Ojo et al. “A Review of Low-End, Middle-End, and High-End Iot Devices”. In: *IEEE Access* 6 (2018), pp. 70528–70554. DOI: 10.1109/ACCESS.2018.2879615.

[5] Chuan Feng et al. “Computation offloading in mobile edge computing networks: A survey”. In: *Journal of network and computer applications* 202 (2022), p. 103366.

[6] Shaveta Dargan et al. “A survey of deep learning and its applications: A new paradigm to machine learning.” In: *Archives of computational methods in engineering* 27.4 (2020).

[7] Pramila P. Shinde and Seema Shah. “A Review of Machine Learning and Deep Learning Applications”. In: *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*. 2018, pp. 1–6. DOI: 10.1109/ICCUBEA.2018.8697857.

[8] Zhichao Zhang and Abbas Z Kouzani. “Implementation of DNNs on IoT devices”. In: *Neural Computing and Applications* 32.5 (2020), pp. 1327–1356.

[9] Yanming Chen et al. “Edgeci: Distributed workload assignment and model partitioning for cnn inference on edge clusters”. In: *ACM Transactions on Internet Technology* 24.2 (2024), pp. 1–24.

[10] Wei Yu et al. “A Survey on the Edge Computing for the Internet of Things”. In: *IEEE Access* 6 (2018), pp. 6900–6919. DOI: 10.1109/ACCESS.2017.2778504.

[11] Hoa Nguyen Thanh Doan et al. “Optimizing Edge Device Routing in Edge Computing: Harnessing the Synergy of Distributed Processing and Correlation Analysis”. In: *Proceedings of the 2024 13th International Conference on Software and Computer Applications*. 2024, pp. 368–372.

[12] Shen Li et al. “Pytorch distributed: Experiences on accelerating data parallel training”. In: *arXiv preprint arXiv:2006.15704* (2020).

[13] Matthias Langer et al. “Distributed training of deep learning models: A taxonomic perspective”. In: *IEEE Transactions on Parallel and Distributed Systems* 31.12 (2020), pp. 2802–2818.

[14] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. “Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018), pp. 2348–2359.

[15] Sifan Zhao et al. “AdapCP: Collaborative Inference with Adaptive CNN Partition on Distributed Edge Servers”. In: *ACM Transactions on Autonomous and Adaptive Systems* 20.4 (2025), pp. 1–28.

[16] Hui Li et al. “Adaptive model partitioning and pruning for collaborative DNN inference in mobile edge-cloud computing networks”. In: *IEEE Transactions on Mobile Computing* (2025).

[17] Wangbo Shen et al. “Reinforcement learning-based task scheduling for heterogeneous computing in end-edge-cloud environment”. In: *Cluster Computing* 28.3 (2025), p. 179.

[18] Yuankun Zhang, Zhaoxuan Zhang, and Hengye Zhao. “Adaptive DNN Partitioning for Edge-Cloud Systems with Meta-Reinforcement Learning”. In: *Proceedings of the 18th IEEE/ACM International Conference on Utility and Cloud Computing*. UCC ’25. Association for Computing Machinery, 2026. ISBN: 9798400722851. DOI: 10.1145/3773274.3774271. URL: https://doi-org.ezp.sub.su.se/10.1145/3773274.3774271.

[19] Chao Fang et al. “AI-Driven Energy-Efficient Content Task Offloading in Cloud-Edge-End Cooperation Networks”. In: *IEEE Open Journal of the Computer Society* 3 (2022), pp. 162–171. DOI: 10.1109/OJCS.2022.3206446.

[20] Yuer Ma, Yanyan Wang, and Bin Tang. “Joint Optimization of Model Partitioning and Resource Allocation for Multi-Exit DNNs in Edge-Device Collaboration”. In: *Electronics* 14.8 (2025). ISSN: 2079-9292. DOI: 10.3390/electronics14081647. URL: https://www.mdpi.com/2079-9292/14/8/1647.

[21] Weiwei Fang et al. “Joint Architecture Design and Workload Partitioning for DNN Inference on Industrial IoT Clusters”. In: *ACM Trans. Internet Technol.* 23.1 (Feb. 2023). ISSN: 1533-5399. DOI: 10.1145/3551638. URL: https://doi-org.ezp.sub.su.se/10.1145/3551638.

[22] Daniela Costa and Rafael Lima. “Dynamic Deep Neural Network Partitioning For Low-Latency Edge-Assisted Video Analytics: A Learning-To-Partition Approach”. In: *International Journal of Modern Computer Science and IT Innovations* 2.10 (2025), pp. 40–50. URL: https://aimjournals.com/index.php/ijmcsit/article/view/303.

[23] Xing Chen et al. “Energy-efficient offloading for DNN-based smart IoT systems in cloud-edge environments”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.3 (2021), pp. 683–697.

[24] Hyochan Kim et al. “A DNN partitioning framework with controlled lossy mechanisms for edge-cloud collaborative intelligence”. In: *Future Generation Computer Systems* 154 (2024), pp. 426–439. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2024.01.006. URL: https://www.sciencedirect.com/science/article/pii/S0167739X24000062.

[25] Adiba Masud et al. “Where to Split? A Pareto-Front Analysis of DNN Partitioning for Edge Inference”. In: *2025 IEEE 11th International Conference on Edge Computing and Scalable Cloud (EdgeCom)*. IEEE. 2025, pp. 27–33.

[26] Haneul Ko, Jaewook Lee, and Sangheon Pack. “Spatial and Temporal Computation Offloading Decision Algorithm in Edge Cloud-Enabled Heterogeneous Networks”. In: *IEEE Access* 6 (2018), pp. 18920–18932. DOI: 10.1109/ACCESS.2018.2818111.

[27] Shifeng Peng et al. “APT-SAT: An Adaptive DNN Partitioning and Task Offloading Framework Within Collaborative Satellite Computing Environments”. In: *IEEE Transactions on Network Science and Engineering* 13 (2026), pp. 597–610. DOI: 10.1109/TNSE.2025.3585287.

[28] Pradyumna Gokhale, Omkar Bhat, and Sagar Bhat. “Introduction to IOT”. In: *International Advanced Research Journal in Science, Engineering and Technology* 5.1 (2018), pp. 41–44.

[29] Dan-Radu Berte et al. “Defining the iot”. In: *Proceedings of the international conference on business excellence*. Vol. 12. 1. Pearson Educational. 2018, pp. 118–128.

[30] Earl B Hunt. *Artificial intelligence*. Academic Press, 2014.

[31] Jafar Alzubi, Anand Nayyar, and Akshi Kumar. “Machine Learning from Theory to Algorithms: An Overview”. In: *Journal of Physics: Conference Series* 1142.1 (Nov. 2018), p. 012012. DOI: 10.1088/1742-6596/1142/1/012012. URL: https://doi.org/10.1088/1742-6596/1142/1/012012.

[32] Zhi-Hua Zhou. *Machine learning*. Springer nature, 2021.

[33] Development Seed. *Introduction to Machine Learning, Neural Networks and Deep Learning*. Accessed: 2026-03-09. 2021. URL: https://developmentseed.org/tensorflow-eo-training/docs/Lesson1a_Intro_ML_NN_DL.html.

[34] Holisun Research. *Edge-Fog-Cloud Computing*. Accessed: 9 March 2026. 2023. URL: https://research.holisun.com/trending-topics/edge-fog-cloud-computing.

[35] Keyan Cao et al. “An Overview on Edge Computing Research”. In: *IEEE Access* 8 (2020), pp. 85714–85728. DOI: 10.1109/ACCESS.2020.2991734.

[36] Schahram Dustdar, Victor Casamayor Pujol, and Praveen Kumar Donta. “On Distributed Computing Continuum Systems”. In: *IEEE Transactions on Knowledge and Data Engineering* 35.4 (2023), pp. 4092–4105. DOI: 10.1109/TKDE.2022.3142856.

[37] Victor Casamayor Pujol et al. “Distributed computing continuum systems–opportunities and research challenges”. In: *International conference on service-oriented computing*. Springer. 2022, pp. 405–407.

[38] Luhui Wang et al. “Ace: Toward application-centric, edge-cloud, collaborative intelligence”. In: *Communications of the ACM* 66.1 (2022), pp. 62–73.

[39] Jing Liu et al. “Edge-cloud collaborative computing on distributed intelligence and model optimization: A survey”. In: *IEEE Communications Surveys & Tutorials* (2026).

[40] Klaas-Jan Stol and Brian Fitzgerald. “The ABC of software engineering research”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27.3 (2018), pp. 1–51.

[41] Martyn Denscombe. *EBOOK: The good research guide: For small-scale social research projects*. McGraw-Hill Education (UK), 2017.

[42] Roel Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[43] Winston Tellis et al. “Application of a case study methodology”. In: *The qualitative report* 3.3 (1997), pp. 1–19.

Appendix A — Implementation

For experimental reproducibility, the implementation of our proposed framework has been documented and made openly available in the repository below.

https://github.com/Akuien/Thesis-experiment

The paper has been submitted at the International Conference on Availability, Reliability and Security (ARES) in the Workshop on Security, Availability, and Fault-Tolerance in Edge AI Systems (SAFE-EDGE) and published as a preprint in ArXiv.

https://arxiv.org/abs/2605.09623

Appendix B — Reflections

Akuen Akoi Deng

How does your thesis align with the objectives of the thesis course? Why? Focus on the objectives that were particularly well met and those that were less well met.

Our thesis aligns well with the objectives of the thesis course because it demonstrates the ability to independently carry out qualified academic work within computer and systems sciences. Through the thesis, we were able to identify a relevant research problem in adaptive AI task partitioning and offloading, reviewed and built upon existing scientific literature, and developed a practical solution through a physical testbed experiment. The chosen methodology was appropriate because the research question focused on measurable system performance, especially energy consumption and latency, which required real hardware evaluation rather than only simulation. By designing, implementing, and evaluating an adaptive framework across a Raspberry Pi, laptop, and GPU-enabled PC, our thesis contributed to knowledge development in edge-cloud collaborative intelligence and showed that adaptive partitioning can improve energy efficiency while maintaining acceptable latency.

The objectives that were particularly well met were those related to independent research, correct application of scientific methods, literature-based argumentation, and writing a structured academic report. The thesis also supported oral presentation and defence because the problem, method, results, and contribution were clearly defined. However, some objectives were less strongly met, especially the reflection on ethical and societal aspects. This was because security and privacy were out of the thesis scope.

How did the planning for the thesis work? What could have been done better?

The planning for the thesis work generally went well. We were able to start early and prioritised doing much of the work right from the beginning. We made concrete plans and timelines for doing each phase of the thesis meticulously. For example we design our own work flow, gantt chart and internal deadlines to ensure we did not fall behind at any point. To work more smoothly, we ensured constant open and frequent communication with our supervisor. This approach was extremely helpful throughout the whole thesis work. If we had to redo it again, we would probably put more focus on starting the experimental setup and work early and also dedicate some time for addressing feedback before they build up.

How does the thesis relate to your education? Which courses and areas have been most relevant to the thesis?

The thesis work really depended and related to most of the prior courses and experience. Courses such as Internet of Things (IoT), foundations of data science and scientific communication and research methodology were really relevant in the thesis work. Other prior knowledge such as research methodology in software engineering and bachelor thesis course from my previous studies also came handy. Having used experiment research methodology in my previous research project was valuable.

How valuable is the thesis for your future work and/or studies?

The thesis work and process is very important for my future career pursuits. I have always enjoyed research work and have plans of pursuing doctoral education, this with the fact that the thesis is my first published paper, makes it really essential for the next career plans that I have. Additionally, the technical development part of the thesis has given me more experience working with distributed systems and AI systems in general. This is important for my technical interest in systems AI engineering. It also gave me the opportunity to apply my prior software engineering technical skills.

How satisfied are you with the execution and outcome of the thesis? Why?

I am very satisfied with the thesis process life-cycle, execution and the outcome. Building the framework has made me learn more about distributed, parallel and cluster computing, AI systems, Machine learning and systems design. Meticulously applying the state-of-the-art research methodology and practices has deepened my knowledge and experience in research in

general. I can recognize that we have built something to be proud of, published a paper that others can learn from and hopefully we will get the opportunity to present the work in the workshop conference which will be a great experience.

Which AI tools or similar tools did you use in the thesis process, and how did you use them? How well did the tools work? Please provide some examples.

We used AI tools such as ChatGPT, Claude, and Gemini. They were used in generating automation scripts, troubleshooting the network, debugging and fixing errors. Sometimes, they were helpful in small isolated implementations where the system design or context was not relevant or I could not recall specific syntax or linux commands. For example, we used ChatGPT to compare and evaluate the most suitable method for energy consumption measurement. It was also used to give us the commands for introducing the network throttling via the linux terminal.

Eimantas Butkus

How does your thesis align with the objectives of the thesis course? Why? Focus on the objectives that were particularly well met and those that were less well met.

While looking back at the course objectives, I think the thesis met most of them well. The objective of independently carrying out qualified academic work was met strongly. My thesis partner and I designed, built, and tested a complete adaptive partitioning framework from scratch on real hardware, which required making independent technical and methodological decisions throughout the process. Contributing to knowledge development was also well met. Our work directly addresses a gap in literature where most partitioning frameworks are evaluated in simulation rather than on physical devices, and where energy consumption is often overlooked as an optimization target. Selecting and applying relevant scientific methods was straightforward. We chose a quantitative physical testbed experiment, justified why simulation and fielded experiments were not appropriate, and followed a clear three-phase experimental procedure. Searching for and building upon scientific literature was also a strength – we reviewed many recent studies, categorized them into groups, and identified the specific gaps our work addresses. The objectives that were less naturally met were the ones related to ethical and societal reflection. Since our

thesis does not involve human subjects or personal data, the ethical discussion was limited to energy sustainability and the need for validation before deploying the framework in safety-critical contexts. It is a valid discussion, but it does not go as deep as it might in a thesis that directly involves people or sensitive data.

How did the planning for the thesis work? What could have been done better?

The overall planning worked reasonably well, but the implementation phase took longer than expected. Setting up the distributed testbed across three physical devices introduced problems that are hard to predict in advance – network configuration with Tailscale, getting ZeroMQ messaging to work reliably across all nodes, and debugging energy measurement inconsistencies all consumed more time than planned. If I could do it again, I would have started the hardware setup earlier, ideally in parallel with the literature review. I would also have set aside buffer time for writing – we underestimated how long it takes to write clear pseudocode, produce accurate diagrams, and iterate on sections like the abstract.

How does the thesis relate to your education? Which courses and areas have been most relevant to the thesis?

The thesis connects directly to several courses from my master's programme. Parallel and Distributed Programming was the most relevant – it gave me the foundation for partitioning workloads across nodes, managing communication between processes, and thinking about synchronization and overhead. The IoT course was also highly relevant, as it covered edge devices, resource constraints, and the kind of heterogeneous environments our testbed represents. Machine Learning provided the background on neural network architectures, which was essential for understanding how CNN layers can be split and why MobileNetV2 behaves differently from VGG-16 under partitioning. Data science helped with structuring experimental evaluation and interpreting results statistically. And scientific communication and research methodology helped with writing thesis as we learned structuring and how to write well for the thesis format. My earlier BSc in Computer Games Programming from the University of Derby also played a role. Working with real-time systems in game engines taught me to think about performance budgets and optimization under hardware constraints, which transferred well to the problem of splitting inference workloads across devices with different capabilities.

How valuable is the thesis for your future work and/or studies?

Very valuable. I am targeting ML engineering and software development roles, and this thesis gave me hands-on experience with the kind of work these roles involve – not just training models but deploying them on real hardware with real constraints. I think the experience of building something that runs across three physical devices and dealing with all the problems that come with that will be useful no matter what I end up working on.

How satisfied are you with the execution and outcome of the thesis? Why?

I am satisfied with the outcome. The framework achieved energy reductions of 27-35% across all three models while also reducing latency, which directly answers our research question. Validating this on real hardware rather than in simulation makes the results more credible. I am also pleased that we demonstrated adaptivity by throttling the network during experiments and observing the scheduler react in real time. I wish we had time to compare our framework against an existing solution like EdgeCI or DeepThings more directly rather than only against a static baseline. But given the time constraints, focusing on a thorough static baseline comparison and leaving direct comparison for future work was the right call.

Which AI tools or similar tools did you use in the thesis process, and how did you use them? How well did the tools work? Please provide some examples.

We used Claude, ChatGPT, and Gemini throughout the thesis, primarily for debugging Python scripts and fixing errors during testing. For example, when implementing the ZeroMQ communication layer, we used Claude to help diagnose serialization issues with PyTorch tensors being sent across nodes. It was also useful for writing LaTex pseudocode for the five algorithms – getting the notation consistent across all of them required several iterations. We also used them for grammar checks, spell checks and other minor errors. The tools worked well for targeted tasks like these, but they were not used to generate wholesale thesis content. All the actual research, code, and analysis is our own work.