TU WIEN Informatics

Edge Inference Using ML-Based Resource-Aware Offloading

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Stefan Hampejs, BSc
Matrikelnummer 01631842

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ. Prof. Dr. Schahram Dustdar
Mitwirkung: Univ.Ass. Dipl.-Ing. Philipp Alexander Raith, BSc

Wien, 10. September 2024

[redacted]

Stefan Hampejs — Schahram Dustdar

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Informatics

Edge Inference using ML-based model-aware offloading

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Stefan Hampejs, BSc
Registration Number 01631842

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dr. Schahram Dustdar
Assistance: Univ.Ass. Dipl.-Ing. Philipp Alexander Raith, BSc

Vienna, September 10, 2024

Stefan Hampejs Schahram Dustdar

Erklärung zur Verfassung der Arbeit

Stefan Hampejs, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel" habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 10. September 2024

Stefan Hampejs

Acknowledgements

I would like to thank my advisor Schahram Dustdar and my co-supervisor Philipp Raith who made all of this possible for me. Without their willingness to help and their passion, I could not finish this work as I did. Using their infrastructure, including the testbed, made this work very interesting and time-saving. Furthermore, I want to thank my family and friends, who never gave up on me and encouraged me to continue. Finally, I want to thank my girlfriend who also listened and stayed with me even when I was down and stressed.

Kurzfassung

Mit der Ausbreitung von Künstlicher Intelligenzen (KI) wird die Cloud zu einem zunehmenden Engpass. Darüber hinaus benötigen Anwendungen immer schnellere Antwortzeiten. Dadurch sind Lösungen gefragt, welche diesem Engpass abschwächen können und eine schnelle Antwortzeit ermöglichen. Eine mögliche Lösung könnte die dezentrale Datenverarbeitung sein, welcher dieser Herausforderung gewachsen ist. Dabei werden die Anfragen nicht zur Cloud gesendet, sondern zu Geräten, welche sich näher zu den Endgeräten befinden. Diese Lösung bietet nicht nur eine verkürze Wartezeit durch die verringerte Distanz, sondern auch eine erhöhte Sicherheit. Jedoch sind die Geräte, welche zur Verarbeitung der Anfragen genutzt werden, generell sehr unterschiedlich. Die Geräte unterscheiden sich in ihrer Größe und Leistung. Darüber hinaus sind sie Leistungstechnisch schlechter als die Cloud, was eine Herausforderung darstellt. Aufgrund eingeschränkter Netzwerkbedingungen und schwacher Client-Leistung ist das Auslagern von Aufgaben nicht immer möglich, aber sich ausschließlich auf den Client zu verlassen, ist ebenfalls ineffizient. Unsere Auslagerungsstrategien zielen darauf ab, die Gesamtleistung zu verbessern, indem diese Einschränkungen ausgeglichen werden.

In dieser Abschlussarbeit untersuchen wir einen Fall im 'Erweiterter Realität' Bereich. Ein Endgerät will Bilder durch ein Semantik Segmentation Modell analysieren, welches Bilder in die unterschiedlichen Objektgruppen aufteilt. Dieses KI-Modell befindet sich sowohl am Endgerät als auch auf einem Server. Das Endgerät kann den Server über einen Router erreichen. Um das Systemverhalten zu analysieren, haben wir eine Serie von Experimenten durchgeführt. Dabei haben wir uns auf die Informationen beschränkt, welche einem Endgerät immer zur Verfügung stehen und wir nicht davon ausgehen, dass der Server oder Router uns weitere Informationen zur Verfügung stellt. Da uns innere Prozesse vorenthalten sind handelt es sich hier um ein Black-Box System. Aus diesem Grund sind ausschließlich Informationen über das eigene Gerät verfügbar, sowie Informationen zum verbundenen Router. In unseren Experimenten betrachten wir verschiede Ausgangslagen. Diese unterscheiden sich von mehreren Router und Server, unterschiedliche Umgebungsbedingungen, sowie unterschiedliche Ausgangsbelastungen am Endgerät. Ziel dieser Experimente ist es Leistungseigenschaften der KI-Modells am Endgerät zu evaluieren und mit dem Verarbeiten am Server zu vergleichen. Durch die Analyse soll das abarbeiten von KI-Modell Anfragen im 'Erweiterter Realität' Bereich optimiert werden.

Die Daten, welche wir durch unsere Experimente sammeln, verwenden wir, um verschiedene Offloading-Ansätze zu erstellen. Diese unterscheiden sich in der Anzahl der Eingabeparameter und in der Komplexität. Mit den ersten Ansätzen sammeln wir grundlegende Informationen um die effektivität unserer Offloading-Ansätze zu überprüfen. Die Ergebnisse zeigen, dass es nicht ausreicht sich auf die Netzwerkinformationen zu verlassen. Aus diesem Grund sind die weiteren Ansätze ressourcenbewusst. Dies bedeutet, dass zusätzlich zu den vorhanden Netzwerkinformationen auch Hardwareinformationen, wie die derzeitige CPU Auslastung, des eingenen Gerätes genutzt werden. Unter den Ansätzen befindet sich einer mit einem Entscheidungsbaum, sowei einer welcher mit KI-Modellen arbeitet. Ziel ist es KI-Modelle zu generieren, welche den Ist-Zustand richtig evaluieren können. Dabei muss das KI-Modell die interne Leistung des Endgerätes, die Netzwerkleistung und Rechenauslastung interpretieren und dementsprechend handeln, mit dem Ziel die Leistung im dezentralen System zu verbessern um den KI-Anwendungsanforderungen gerecht zu werden.

Abstract

With the increasing prevalence of artificial intelligence (AI) applications, the strain on cloud processing resources has become a notable bottleneck. This requires solutions that can alleviate this strain, particularly as applications increasingly demand faster response times. Edge computing has emerged as a viable solution to address these challenges by offloading computational tasks to devices closer to the end-user. Edge computing not only offers reduced latency due to its proximity to end devices but also provides enhanced privacy protection. However, the heterogeneity and limited resources of edge devices pose significant challenges compared to cloud resources. In this thesis, we focus on evaluating a scenario in which an end device is tasked with processing images using a semantic segmentation model (SSM). A SSM is deployed on the end device and also on a remote server, to which clients can offload their images through an access point (AP). Due to limited network conditions and poor client performance, offloading tasks is not always feasible, but relying on the client alone is also inefficient. Our offloading strategies aim to improve overall performance by balancing these limitations. We conduct a series of experiments in a black-box manner in which only information about the end device and its connection to the AP is available. Our experiments involve various scenarios. These scenarios differentiate in diverse AP, diverse servers, varying environmental conditions, and different load on the client. Through these experiments, our aim is to gain insight into the performance characteristics of the SSM running on the end devices and to compare their efficacy with the models deployed on remote servers. The data gathered from these experiments will be used to construct different offloading approaches. They differ in the number of input features and in their complexity. The first approaches build a baseline that shows that the use of network conditions alone is not sufficient for the offloading decision. Based on that, we formulate two resource-aware offloading approaches as well, which use hardware resources of the client combined with network information. The first one works with a decision tree model and the other one works with a selected ML model. The approaches are compared with each other. The goal is to identify models that demonstrate remarkable effectiveness in evaluating the current state of edge devices and guiding appropriate actions. Using these insights and deploying efficient machine learning models, we aim to optimize the performance of edge computing systems and enhance their overall effectiveness in meeting the demands of AI applications. With that, the standard deviation can be improved and therefore increase the stability of the response times for the requests.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The popularity of artificial intelligence (AI) has increased in the last few years. One reason is the success in the fields of speech recognition, computer vision, and natural language processing. This rising popularity leads to a tremendous amount of data that needs to be sent between devices to fulfill AI requests. These tremendous amounts of data are mostly sent to the cloud. The reason is that the computation of such requests is very complex and resource intensive and, therefore, infeasible to be handled by mobile devices or edge devices themselves. However, using only the cloud for computation includes several disadvantages. One disadvantage is the long distance between the device sending the request and the cloud. This large distance leads to a long latency. For many use cases, it does not matter whether the result of the request has a delay of several seconds or not. However, there are also use cases where it is important that the result is available in a fraction of a second. In edge intelligence (EI), many requests have to be offloaded due to resource constraints, there long latency could have disastrous consequences, e.g. driving an autonomous vehicle, where a delayed response could lead to a car crash [TLL+20]. Another downside is availability. Using the cloud for computation implies that there has to be a continuous connection between the devices. However, stable connections are present mainly in large cities, but are not available within small villages, tunnels, or sequestered areas in general. Other downsides are poor hardware utilization of edge devices and privacy concerns, as well. For these reasons, the trend is to execute the request on edge servers or on edge devices themselves. Edge servers are an alternative to the cloud. They are more decentralized on the edge of the network, increasing availability and reducing latency for the request. In addition, it is a better solution in terms of privacy. However, edge servers and edge devices have, in contrast to the cloud, restricted hardware components, which is a major disadvantage. This means that the models used in the cloud to process requests are too heavy and computationally intensive to be executed on edge servers and edge devices. For that reason, some modifications are necessary to the model or execution to overcome this bottleneck. Such modifications are not trivial, which means that they should be done in aspects of the current state of the available resources and the network connection.

In our scenario, we analyze the use case in which a client uses augmented reality (AR) on his device. This device should be considered to be capable of processing images needed for AR on its own. Additionally, this device is able to connect to an AP in the surrounding, however, does not have the possibility to connect to the cloud or to the Internet directly. These APs allow the client to connect to an edge device, which is also able to process the requests with an increased speed because of a better hardware compared to the clients' device hardware. Since there are two options where the client can process the image, which is local or external, we are interested under which circumstances the client should execute the requests local or external. For that we can use the different load of our hardware resources and the information available from the AP we connect to. Therefore, we investigate several offloading approaches and compare them with each other. The first approaches are used to build a baseline, measuring the effectiveness of our offloading strategies under the given network and limited client hardware performance conditions. The approaches show that resource awareness is required, which means that an offloading approach uses hardware and network features for their offloading approach. Therefore, the following approaches are resource-aware. One includes a decision tree, and the other includes a machine learning model.

1.2 Research Questions

In our work, we want to improve the overall performance of processing images needed in the AR. We focus on the following Research Questions:

RQ1: *Which metrics can be used for offloading in a black box system?*

There are already many approaches to build a mathematical equation when it is better to offload the requests or not, such as the paper by Yue Zhang and Jingqi Fu. [ZF21], which provide a perfect solution in theory. However, in papers like this, they make a lot of assumptions which can differ in practice, such as to know the time a request takes to send to another device. For that reason, it is important to investigate this topic in a real-world setting.

For that, we first investigate the kind of metrics available in general. Some of them are perhaps misleading or not related for our purpose. Therefore, we want to filter the information that we can use for our offloading decision. Furthermore, it is interesting to know in what quality and frequency we can get this information. As a result, we expect information we can use for deciding where we should process the requests.

RQ2: *What offloading strategies can be used?*

In general, we can think of many different ways in which and where a request can be executed. The possibilities are restricted by the environment and the capabilities of the device in use. Offloading strategies introduce some overhead due to decision-making, so our aim is to keep this overhead minimal while evaluating each request individually. This is necessary because network and client conditions are constantly changing. Our offloading decisions are based solely on the information available to the client, treating it as a black-box approach.

In our approach, we investigate a client with its own device that has the opportunity to connect to one of the available APs in the area. Connecting to an AP allows the client to offload the request to a server that is also connected to the AP. Therefore, there are only two options for the client. Either it processes the images on their own device, or it offloads them to the server behind the AP. Relying on only the device on itself or the external is a bad idea because the device on itself is weak and offloading could be done, although the connection is miserable. Therefore, we want to offload over the AP when there is a stable connection to the AP, and we can assume that the request can be processed faster there. For that we are using the metrics available for our client, which are the information about the used HW with the current load and the APs the client is able to connect to. As a result, we expect that the offloading decision we use is aware of the current situation, including HW load and AP connection, and decides where the requests should be executed.

RQ3: *How do ML models handle real-world settings?* In our final step, we are curious whether the use of a ML model can improve the overall result. We want to train models which are resource-aware, which means that they derive the current state by analyzing the metrics we collect during the experiments. These trained models are then used to make the offloading decision for us. Depending on the resources available for us and the quality of the data, it might still lead to bad results anyway. This can be the case, when the available resources are not representative for the current state or the quality is not good enough to distinguish whether offloading is advisable or not.

However, using an ML model introduces additional latency because the information has to be processed before the ML model can make a decision where the request should be processed. Therefore, we train different models and compare them with each other in a real-world evaluation setting. This means that the models are compared in experiments that are done in a lab where the client moves during the experiment to change the network condition. In addition, different loads are added to the client to simulate different initial states on the client. As a result, we expect to find a ML model which is able to evaluate the current situation and offload according to it with the aim of improving overall performance.

1.3 Structure of the Thesis

The remainder of this thesis is structured as follows. Chapter 2 covers the relevant background information of this work, including general information about AI, edge computing, edge intelligence, and testing techniques, but also the tools we used in this work. Chapter 3 describes the related work, highlighting the work relevant to this work. In particular, it covers work in the area of distributed execution, compression, hardware accelerator, benchmark, and WiFi. Chapter 4 provides information on our approach. This includes the offloading approaches we compare, the systems and tools we use, the data we collect, and the mechanism to filter and preprocess the acquired data to train an ML model. Chapter 5 describes the offloading approaches in a deeper sense and explains how they are compared to each other. Chapter 6 presents the results of different experiments performed in this work. In the end, the results are summarized and discussed in Chapter 7 and Chapter 8 answers the research questions and describes future work including the boundaries of our work.

CHAPTER 2

Background

In this section, we cover the fundamentals that are important to this thesis. In Section 2.1 we start with an overview of edge intelligence, including the six-level rating of EI. In Section 2.2 we provide a general description of AI, continuing with different deep learning approaches. In Section 2.3 we elaborate edge computing. A common task in Edge Computing is offloading which is discussed in more detail in Section 2.4. It also provides some related topics such as offloading decisions, the degree of offloading, machine learning in general, and the steps needed to train a ML model. Section 2.5 covers testing and Section 2.6 summarizes the tools used in this thesis.

2.1 Edge Intelligence

Edge intelligence is the fusion of AI and edge computing. It enables performing computationally intensive tasks, such as AI tasks in the immediate vicinity of the client. In the past, most data, such as online shopping records, social media content, and business information, were primarily collected and stored in large data centers. Since social media data need to be stored in the data center anyway, it is logical to also perform computationally intensive tasks there. However, with the widespread use of mobile computing and the Internet of Things, the trend is changing [ZCL+19].

Generally, data collected from mobile devices and IoT devices is sent to the data center for analysis [HCS15]. However, due to this emerging trend and the growing number of mobile and IoT devices, a significant amount of data would have to be sent between the end devices and data centers. Since this occurs on the wide area network (WAN), it can include prohibitively high monetary cost and transmission delay.

Furthermore, data transmission to the data center can potentially lead to privacy breaches along the way to the data center [PAB+15]. The privacy problem could be easily solved by running all AI tasks directly on the IoT device. However, IoT devices are relatively small with limited resources, resulting in poor performance and energy efficiency. This is because AI tasks require significant computational power that exceeds the capabilities of IoT devices. The solution to this problem is edge computing [SCZ+16a], which offloads tasks from the network core to the network edge, i.e., from end devices to edge devices. These edge devices are closer to the end devices compared to the cloud and have better resources compared to IoT devices, making them capable of efficiently executing AI tasks.

2.1.1 Six-Level rating for EI

Different solutions for addressing a given task within a specific environment can be categorized under various Edge Intelligence (EI) paradigms. Zhou et al. [ZCL+19] have classified these EI solutions into six levels according to the devices used and their proximity to the end user. This classification is visually represented in Figure 2.1, with the following description of where the Deep Neural Network (DNN) model undergoes training and inference:

1. **Cloud Intelligence**: Training and inference occur in the cloud.
2. **Cloud-Edge Co-inference and Cloud Training**: Training is conducted in the cloud, while inference follows a cloud-edge approach, involving execution partly on the edge and partly offloaded to the cloud.
3. **In-Edge Co-inference and Cloud Training**: Training is performed in the cloud, whereas inference is carried out in an in-edge manner, involving execution primarily at the end device with partial offloading to edge devices.
4. **On-Device Inference and Cloud Training**: Training takes place in the cloud, while inference is performed solely on the end device without any offloading.
5. **All In-Edge**: Training and inference occur within an in-edge framework.
6. **All On-Device**: Training and inference are conducted on-device.

Since we are not using the cloud our work is located at level 5, where inference occurs within an in-edge framework. At this level, we benefit from the short distances between the devices, which reduces latency compared to the use with the cloud. However, these devices are rather small compared to the cloud, which in contrast increase the time needed to process tasks.

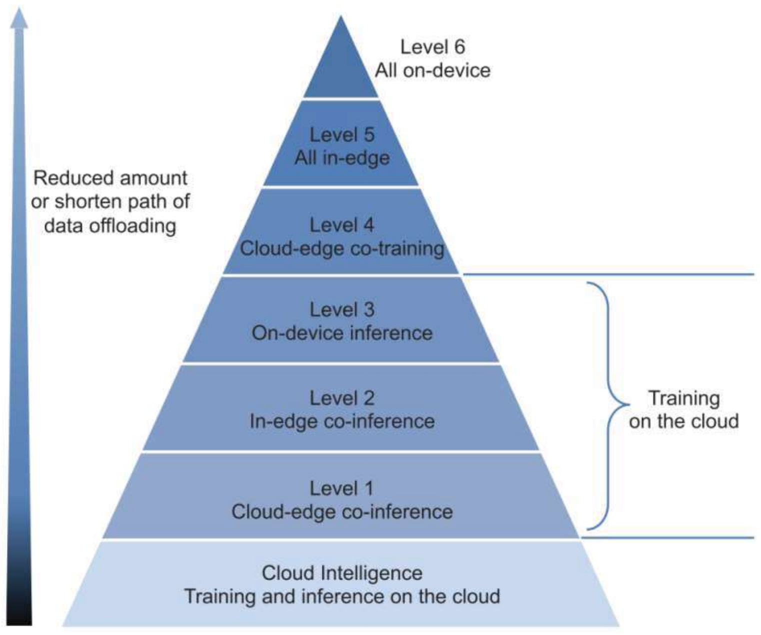


Figure 2.1: Six level of EI defined by Zhi Zhou et al. [ZCL+19]

In the next two sections, we take a closer look at AI and edge computing.

2.2 Artificial Intelligence

The objective of artificial intelligence (AI) is to perform tasks similar to humans. Although AI has attracted considerable attention in the past decade, it is not a new concept. The term has existed since 1956.

Given the broad scope of AI, its precise definition is not always clear. The term is used to simulate behaviors associated with human intelligence, including planning, learning, problem solving, reasoning, and, to a lesser extent, creativity. Since 1959, AI has experienced ups and downs, with the most recent resurgence in the 2010s due to breakthroughs made by deep learning [ZCL+19].

2.2.1 Deep learning

Deep learning enables computational models comprising multiple layers to learn intricate data representations with varying levels of abstraction. These methodologies have significantly advanced various fields, including speech recognition, visual object detection, and object recognition, along with domains such as drug discovery and genomics. By employing the backpropagation algorithm, deep learning uncovers complex structures within extensive datasets, guiding the machine on how to adjust its internal parameters to compute representations at each layer based on the preceding layer's representation.

In particular, deep convolutional networks have revolutionized image, video, speech, and audio processing, while recurrent networks have proven effective in handling sequential data such as text and speech [LBH15].

Popular Deep Learning Models are covered by Zhi Zhou et al. [ZCL$^+$19]:

Convolution Neural Network (CNN): Designed primarily for image classification tasks, CNNs excel at learning hierarchical representations from image data.

Recurrent Neural Network (RNN): Tailored for sequential input data, RNNs are particularly effective in addressing time-series problems. They possess a feedback loop that allows them to maintain information over time.

Generative Adversarial Network (GAN): Comprising two main components, a generator and a discriminator, GANs aim to generate realistic data samples. The generator learns to produce new data points by understanding the underlying data distribution, while the discriminator evaluates these samples to distinguish between real and fake data.

Deep Reinforcement Learning (DRL): DRL combines Deep Neural Networks (DNNs) with Reinforcement Learning (RL) techniques. The objective is to develop efficient policies that maximize rewards over extended periods, particularly in scenarios with controllable actions. DRL is often used in tasks that require long-term planning and decision making.

Neural Networks

An artificial neural network (ANN) consists of interconnected artificial neurons, organized into three layers: the input layer, the hidden layer(s), and the output layer.

The input layer neurons receive information, such as test data, which is filled in the input weights and are evaluated using the activation function and transmitted to the next neuron. Following this, one or more hidden layers further process the input using the activation function and pass the result to the next neuron. Finally, the output layer, executing the activation function one last time, delivers the result.

Each connection within the network is characterized by the strength of the connected pair of nodes, and these connections can adapt over time, initiating a learning process throughout the ANN. To facilitate learning, the ANN must interact with the environment multiple times, which is achieved by providing it with test data. The learning process constitutes a key feature of the ANN, enabling it to comprehend the environment as it progresses [GB08]. In Figure 2.2 we can see a visualization of a neural network [1].

[1] https://www.geeksforgeeks.org/artificial-neural-networks-and-its-applications/

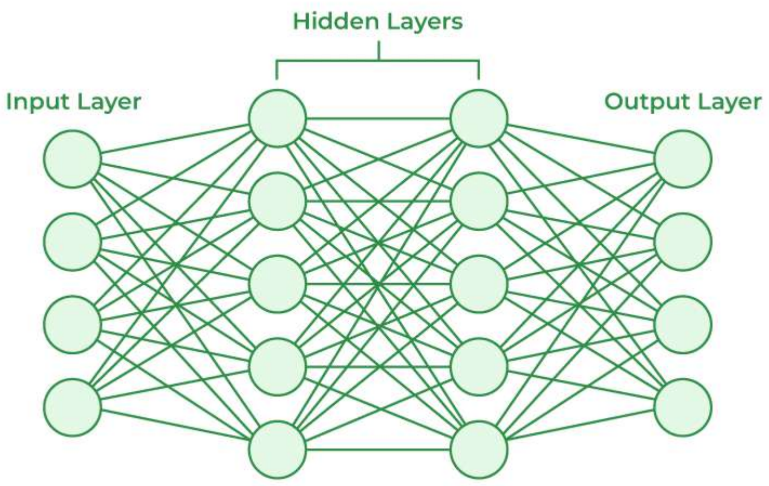


Figure 2.2: Illustration of a neural-network with the three layer separation [2]

2.3 Edge Computing

Edge computing involves technologies that enable computational tasks to be executed in the periphery of the network, handling data downstream of cloud services and upstream of Internet of Things (IoT) services. The *edge* refers to any computing and networking resources located between data sources and cloud data centers, forming a continuous spectrum.

The edge of the Internet is of distinct significance. Typically placed just one step away from end devices, it provides an ideal location for rapid processing of data, supporting various applications like augmented reality, public safety, autonomous driving, smart manufacturing, and healthcare. This proximity facilitates low-latency offloading infrastructure, which is crucial for these emerging applications. In addition, it serves as a strategic point for collecting, analyzing, and refining data from bandwidth-intensive sensors such as video cameras [SCZ$^+$16b].

A graphical visualization of edge computing is presented in Figure 2.3 [ZCL$^+$19]. Featured are various application areas for end devices, including autonomous driving, smart home, smart glasses, smart traffic, and more. Depending on the service they provide, factors such as privacy and latency are crucial. However, not only do the end devices vary, but the edge nodes themselves also differ in size and computational capabilities. Whether from credit card size to microdatacenter, these differences explain the variations in computational power. Nevertheless, they all share a common characteristic, their proximity to the end devices. Using these edge nodes reduces latency, alleviates the overall load on the WAN, and improves privacy. A common form of edge computing implementation is offloading.

[2] `https://media.geeksforgeeks.org/wp-content/cdn-uploads/20230602113310/Neural-Networks-Architecture.png`

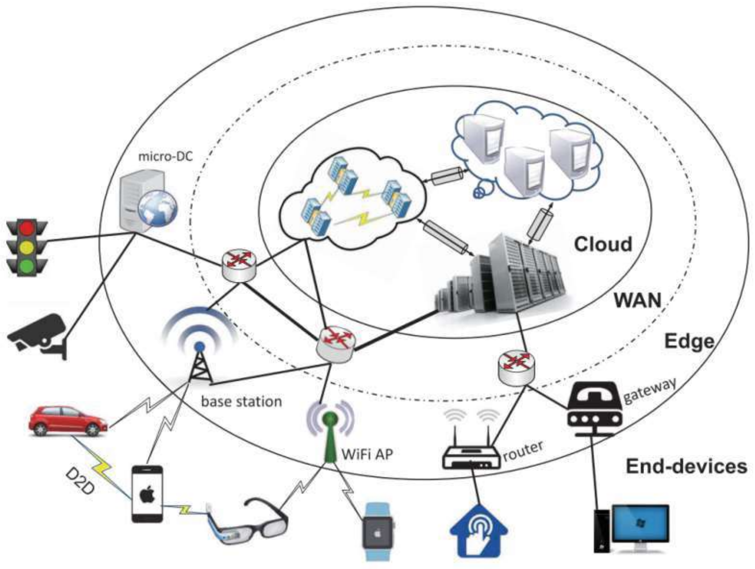


Figure 2.3: Visualization of Edge Computing with different end devices, edge devices and the cloud by Zhi Zhou et al. [ZCL+19]

2.4 Offloading

The principle of offloading is to increase the computing capacity of less powerful devices, such as mobile phones, by utilizing more powerful infrastructure, such as edge devices or cloud servers. In the past, offloading was mainly done to cloud servers with more powerful devices centralized far away. However, since latency has become an important factor, closer devices, such as edge devices, are now being used for offloading [LLJL19].

2.4.1 Offloading Decisions

The offloading decision can be made in several ways, depending on various factors. These include the capabilities of the device that performs the offload, the general offloading strategy, the target device for the offload, and the quality of connectivity. In the following, seven different approaches are summarized.

Rule Based approaches have static thresholds on specific criteria such as CPU load. This approach varies according to the application and needs to be adjusted in updates [KMDN18].

Heuristic Algorithm use heuristics to find a near-optimal solution for the offloading decision. It is based on specific criteria such as minimizing latency or maximizing energy efficiency. This approach heavily depends on expert knowledge or accurate analytical models [WHM+22].

Optimization technique formulate the offloading decision as an optimization problem with the aim of maximizing or minimizing specific metrics such as latency or energy consumption [KMDN18].

Decision Trees are used to make decisions based on multiple criteria. These criteria form a tree-like model in which the nodes represent the criteria and the leaves represent the results [RCC+17]. This approach is used in our thesis.

Machine Learning uses a ML model for the offloading decision. It can be distinguished into three different approaches called: Reinforcement learning, Supervised and Unsupervised learning [CZL+19]. This approach is also used in our thesis and is elaborated in more details in Section 2.4.2.

Game-Theory uses an ML model approach in a multi-user manner where the users interact with each other to achieve a mutually satisfactory solution. It can be distinguished between Non-cooperative and Cooperative Games. [CC18]

Context-Aware uses an ML model approach with addition information such as time or location which helps by the offloading decision [CYN22].

2.4.2 Granularity of Offloading

Offloading can be classified into three different categories. The categories differentiate in granularity and are called full computation offloading, task/component, and method/thread [LLJL19].

Full computation offloading migrates the entire computational workload of applications, leaving mobile devices responsible only for the user interface, input/output, and data sensing.

Task / Component involves dividing the application into subtasks. After analyzing the application workflow, the compute-intensive tasks are sent to a remote infrastructure. The partitioning is application dependent and can also require different partitioning strategies for a specific application.

Method/Thread granularity involves fine-grained computing migration. Typically, an application contains a substantial number of methods, therefore special partitioning mechanisms to assist at this level are needed. These methods are usually compute-intensive and involve a small to moderate amount of data transmission.

In our approach, we concentrate on full computation offloading, which presents a simple approach.

2.4.3 Machine Learning Offloading approach

In the previous section, we discussed various approaches to offloading, including several that involve Machine Learning (ML). In this thesis, we also use a ML approach. For that reason, we take a closer look into ML in general and what to consider when using an ML Model containing data preprocessing and model training.

Machine Learning

Similar to humans, computers can also learn from experience through a process called Machine Learning (ML). ML learns in the way of data by utilizing learning algorithms that analyze and process the provided data to construct a model. This model is able to predict the outcome of new observations. When this technique is used, the overall performance of the system can be increased. [Zho21].

For ML training, a large data set $D = x_1, x_2, ..., x_m$ is required. Each entry x in D, also known as a record or instance, is described by a number of attributes d and an output that represents the result of the attributes. The value m represents the number of events stored in the data set.

For example, when considering dogs, attributes could include fur length, color, size, tail length, and other characteristics. The output could indicate the breed of the dog or a simple yes/no comparison with other animals. These attributes are referred to as features and, when combined, form a feature vector. The number of attributes d determines the dimension of the space. In our dog example, each dog can be placed in the space according to the attribute values [Jos22].

Typically, the life cycle of machine learning models involves two primary phases: model training and inference [DZF$^+$20].

2.4.4 Data Pre-processing

The subsequent data is sourced from Volkan Cetin and Oktay Yıldız [CY22] and is supplemented by other authors, specifically identified in the text. Data preprocessing is a crucial step in data analysis applications, aimed at improving the quality of raw data by addressing various flaws such as inconsistencies, out-of-range values, missing values, noise, and excesses. Failure to address these issues can weaken the performance of subsequent mining and learning algorithms. To improve the quality of raw data, it undergoes several pre-processing stages. This involves using various algorithms tailored to specific tasks, usage scenarios, and underlying algorithms. In recent years, several effective preprocessing methods have emerged, primarily categorized into three main branches: data cleaning, data reduction, and data transformation.

1. Data cleaning

   Data collected in real-world scenarios frequently contain missing and noisy values. Addressing these discrepancies is a key challenge in data analytics, as failing to do so can lead to inaccurate analyzes and unreliable decision-making processes [CIKW16].

   a) Noice Filtering

      Inaccurate data gathering from faulty measurements or human errors within the dataset is referred to as noise. This concept is typically categorized into two main types: feature noise, which relates to inaccuracies in the measurement values themselves, and label noise, which relates to inaccuracies in the class labels assigned to the data points [PVI+17].

b) Missing value imputation

In the raw data set, the values for one or more properties of some samples may be missing. When such a data set with missing values is fed into learning algorithms, it can lead to decreased accuracy rates of the models or even failure to form a model altogether due to algorithmic limitations. To effectively extract knowledge from the dataset, it is imperative to clean and pre-process the data, addressing these missing values.

2. Data Reduction

In addition to increasing complexity and prolonging the time to obtain results, the presence of unnecessary and irrelevant data in a data set can hinder data analysis algorithms from extracting accurate information. Therefore, it is essential to reduce the size of the data set without compromising its quality.

a) Feature selection

Feature selection is a preprocessing technique that aims to identify and select key attributes relevant to a given problem. This process involves reducing the number of features, which corresponds to reducing the number of columns in a data set.

b) Instance selection

In addition to features, irrelevant and redundant instances within the rows of raw datasets can also occupy unnecessary space. These instances may adversely affect model performance and time efficiency. Instance selection involves identifying a subset of the original dataset that best represents its characteristics without compromising the success of the model.

3. Data transformation

Even after addressing issues such as noise, missing values, and unnecessary features through data cleaning and reduction methods, the resulting dataset may still not be suitable for analysis by data analysis applications. Inappropriately structured data can adversely affect the performance and efficiency of data mining models, leading to decreased effectiveness. Methods that convert data into a suitable format for use by data mining algorithms are referred to as data transformation methods.

a) Normalization

Normalization involves scaling the data of a feature to specific intervals, such as [-1.0, 1.0] or [0.0, 1.0], and is typically necessary when the features in a dataset have considerably different scales. Failure to normalize features with significantly different scales may result in decreased effectiveness of lower-scaled features, leading to reduced accuracy performance of data mining models. To address this issue, normalization is applied to bring all features to a comparable scale. The three most common methods of normalization are *min-max normalization*, *z-score normalization*, and *decimal scale normalization*.

b) Aggregation

Data aggregation involves presenting data in a condensed or summarized form by combining two or more attributes under a single attribute. This process is crucial for converting data collected from various sources into an appropriate format. Additionally, data aggregation not only transforms the data, but also reduces the size of the data set, thereby improving memory and time efficiency during data analysis.

2.4.5 Model training

The process of generating a prediction model involves two phases: learning or training, and testing. For the data set D is divided into two smaller data sets L and T, for learning and testing purposes, respectively. It follows that $L \cup T = D$ and $L \cap T = \emptyset$. The ratio between L and T can vary, and adjusting this ratio may impact the performance of the model. Generally L is larger than T, with a ratio of about 80% and 20% for the data set [Jos22].

During the learning/training process, ML uses the learning data set L, also known as the learning set. Individual samples in the learning set are called learning examples. Each training example consists of attributes and an output, also known as a label, which serves as the ground truth. If the output is discrete, it is called a classifier, and regression is used otherwise. In the case of a classifier, there are binary and multiclass classifiers for two or more distinct outputs, respectively. Training examples are used to find or approximate the ground truth. In addition, a subset of the training set is typically used for validation during the training process to assess the current accuracy. The validation process calculates the prediction error of the sample [SBH19].

Following the learning phase, the testing phase is conducted to evaluate the accuracy of the model. This involves the use of the testing data set T , also known as the test set. The samples in the test set are called test samples. During testing, ML attempts to predict the labels of the training samples and records the number of correct predictions as t. At the end of the testing phase, the accuracy is calculated as $\frac{|t|}{|T|}$. If the accuracy is low, the training and testing phases may be repeated.

With this approach, that is, by splitting the data set into training, validation, and test sets, we avoid overfitting as well. Overfitting means that our model learned aspects that are not generalized to the population but specific to our provided data. To avoid overfitting, regularizations are used, which controls the complexity of the model. In this case, complexity describes the ability to estimate a wide range of functions described by variables. Therefore, it can be regulated by changing the number of variables used to train the model. High regulations, which means less variables, tend to reach high bias with low variance. Bias in this case means that the prediction is high. However, the result is more or less constant, which means that the input does not affect the output. On the other hand, low regulations lead to low bias and high variance [SBH19].

Supervised Learning

Supervised learning, whether deep or not, is the most prevalent form of machine learning. Let us consider the scenario of developing a system capable of classifying images, such as identifying whether an image contains pets, people, or devices. Initially, we collect a substantial dataset comprising images of pets, people, and devices, each tagged with its respective category. During the training phase, the machine is presented with an image and generates an output represented as a vector of scores, each score corresponding to a category.

Ideally, we aim for the correct category to receive the highest score, although this is unlikely to happen without training. To evaluate the performance of the model, we calculate an objective function that quantifies the inequality ,or distance, between the output scores and the desired score distribution. The machine then adjusts its internal parameters, often referred to as weights, to minimize this error. These weights act as adjustable "knobs" that define the machine's input-output relationship. In a typical deep-learning setup, there could be millions of such adjustable weights, trained with hundreds of millions of labeled examples.

To effectively fine-tune the weight vector, the learning algorithm computes a gradient vector. This gradient vector indicates the change in error associated with each weight when adjusted by a small increment. Subsequently, the weight vector is updated in the opposite direction to the gradient vector to minimize the error [LBH15].

2.4.6 Analyzing an Image with Artificial Intelligence

Machine learning models are multifaceted and can be applied to various tasks, including analyzing images, which is crucial for augmented reality applications. Common approaches for image analysis include image classification, object detection, and semantic segmentation. In addition to these, there are two more advanced techniques: instance segmentation and panoptic segmentation, both of which extend the capabilities of semantic segmentation. These techniques provide a more detailed understanding and interpretation of images, supporting a wide range of applications in computer vision [HZG20].

Image classification involves assigning one or more category labels to an entire image, indicating the presence of objects such as people, cars, roads, or buildings. Object detection takes it further by not only identifying objects, but also pinpointing their locations within the image using annotated rectangles. Semantic segmentation is even more demanding as it aims to precisely divide each object region from the background, accurately delineating their boundaries. Unlike image classification and object detection, semantic segmentation requires a deeper understanding and precise delineation of objects, making it a more challenging task [HZG20]. A visual presentation of the work of Shijie Haoa, Yuan Zhoua, and Yanrong Guo is presented in Figure 2.4

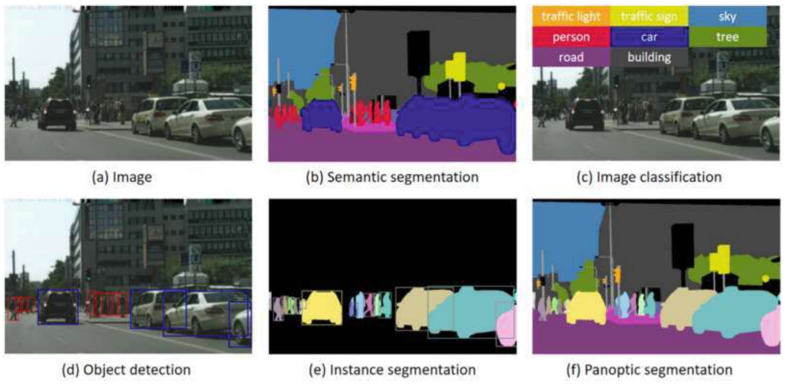


Figure 2.4: Image analyzing approaches by Shijie Haoa, Yuan Zhoua, Yanrong Guo [HZG20]

2.5 Testing Techniques - Black, Gray, White Box Testing

Software testing is crucial to identify defects, flaws, or errors in application code that require correction. It serves as a process for assessing the functionality and accuracy of the software through analysis. The primary goals of testing include ensuring quality assurance, estimating reliability, and validating and verifying the software. It plays a vital role in software quality assurance, involving a thorough review of specifications, design, and coding. The overarching objective of software testing is to confirm the quality of the software system under controlled conditions, ensuring completeness, correctness, and uncovering any previously unnoticed errors [Kha10].

The three main approaches are [Kha10]:

White Box Testing:

It comprises a thorough examination of the internal logic and structure of the code or system. Testers need to have a comprehensive knowledge of the source code to perform this type of testing effectively.

Black Box Testing:

Black-box testing focuses on testing the software without any understanding of its internal workings. Basically, it assesses the external behavior and functionalities of the system without considering its internal structure.

Gray Box Testing:

Is a combination of white- and black-box testing. Gray-box testing involves assessing the application with partial knowledge of its internal workings. The testers have some understanding of the internal structure, while also considering the fundamental aspects of the system's behavior.

2.6 Tools

2.6.1 Galileo

Galileo is a framework designed to perform benchmarks on edge-cloud compute clusters. During our study, we extended this framework originally created by Raith et al. [RRP+22].

The setup of the framework is illustrated in Figure 2.5. This tool facilitates various tasks, such as generating requests, hosting applications, distributing messages, and storing data. Galileo leverages the edge-oriented K3s[3] Kubernetes distribution to host applications and clients. The nodes within the cluster are labeled accordingly, distinguishing between controller nodes, client nodes, and worker nodes. The worker nodes are responsible for hosting applications, while the controller node hosts the load-balancing instance. In our setup, we utilized only one client, and thus the controller node forwarded requests accordingly. The client node is tasked with executing the requests, which are published via Redis. Consequently, the client nodes run Galileo and subscribe to Redis. The Galileo shell serves as the component that initiates the experiment and subscribes to Redis. The experiment data are stored in InfluxDB and MariaDB.

The framework divides experiments into three phases: pre-experiment, runtime, and post-experiment. In the pre-experiment phase, users define various parameters such as workload, applications, clients, and nodes. Galileo supports two types of workload: parameterized and profile-based. For profile-based workloads, utilized in our study, an array containing inter-arrival times is required. Each inter-arrival time list is then sent to a corresponding client.

During the runtime phase, the framework manages the system. It monitors experiment metadata and data collected during the runtime. For that, a fine-grained monitoring tool, *telemd*, is used. Once the experiment is complete, results can be analyzed with Jupyter Notebooks which provide a gateway to the information stored in InfluxDB and MariaDB.

[3]https://k3s.io/

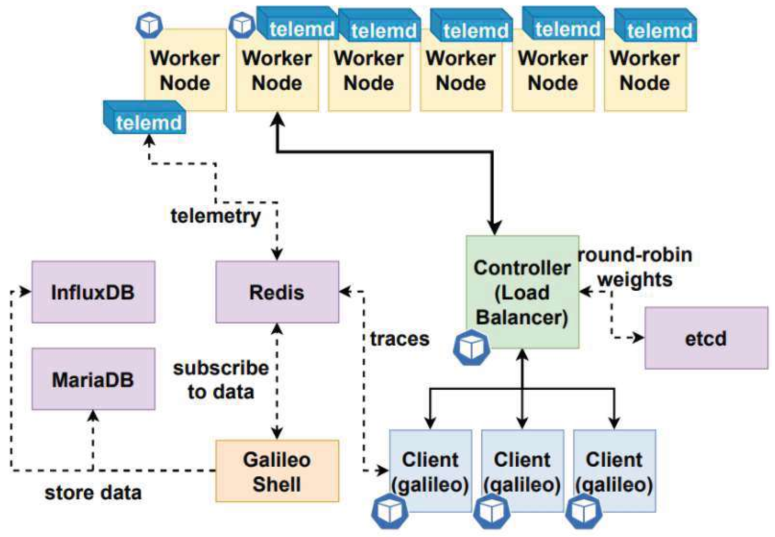


Figure 2.5: Architecture of Galileo by Raith et al. [RRP+22]

2.6.2 TPOT

TPOT is an automated machine learning (AutoML) tool that finds the proper pipelines for a given data set [4]. TPOT searches through the pipeline space considering multiple ML algorithms with different preprocessing steps. For this reason, to get a proper result from TPOT it can take up to days. During the pipeline space, it goes through several generations and, within each generation, through several populations. Generation is the number of iterations to run the pipeline optimization process, and population size is the number of individuals to retain in the genetic programming (GP) population for every generation. This amount is defined by the mutation rate. Using a too small mutation rate would result in evaluating a small search space, which could lead to getting stuck in a local optima. Depending on the data set and the available time and space, TPOT provides different default configurations that provide a good starting point. TPOT comes with six default configurations.

- Default
  - uses 100 generations with each 100 populations,
  - searches over a broad range of preprocessors, feature constructors, feature selections, models, and parameters,
  - goal is to minimize the error of the model prediction.
- Light
  - restricted search,

[4]http://epistasislab.github.io/tpot/

  - only simple and fast-running operations,
  - finding quick and simple pipeline.
- MDR
  - search over Multifactor Dimensional Reduction,
  - goal to maximize model prediction accuracy,
  - specialized for genome-wide association studies (GWAS).
- Sparse
  - uses configuration dictionary with one hot encoder, which is a transformation to category representation to binary representation,
  - support sparse metric, which are datasets with majority of 0.
- NN
  - extension of default configuration,
  - additionally use neural network estimators
- cuML
  - restricted configuration,
  - uses GPU-accelerated estimators

These default configurations can also be modified as needed to improve the ML model for a specific data set.

Genetic Programming

Genetic programming (GP) draws inspiration from Darwinian evolution to solve problems through computer-based simulations. In simple terms, GP involves creating and evolving populations of computer programs where these programs are represented using suitable data structures. In this context, a "population" refers to a group of individual computer programs represented in a particular format. Each generation of programs undergoes evolutionary processes such as pairing and mutation, guided by a fitness measure that evaluates their performance relative to the desired task [BKR+00].

CHAPTER 3
Related Work

In this chapter, we will outline different approaches presented in existing papers to improve the overall utilization of devices. Increasing utilization can be achieved in different ways, such as dividing a task. A task can be divided within a device or over several devices. Another way is to reduce the time that a task needs by compressing it. Additionally, hardware accelerators can also be used that are optimized for the execution of deep neural networks (DNNs). Furthermore, it is important which model is used for execution, since ML models are specialized for a specific kind of task. Finally, it is also important that devices such as WiFi access pointers (APs) are not overloaded because this introduces a lot of latency. The papers presented below use at least one of these optimization techniques.

3.1 Distributed Execution

One way to reduce overall latency and increase the usage of hardware resources is to divide the load between different hardware components and processors as well. This can be separated into internal and external distributions.

3.1.1 Local Distribution

In the local distribution, the load is distributed between different processors such as the central processing unit (CPU), the graphics processing unit (GPU), and the low power unit (LPU).

Bhattacharya et al. introduce DeepX, a new software accelerator that allows processing of a deep learning model on heterogeneous processors such as GPU, LPU, and cloud as well. To achieve this goal, two inference-time resource control algorithms are used. The control algorithms are called runtime-layer compression and deep-architecture compression. With the first one, the complexity of the layers can be scaled. The second is used to divide the deep model, so it can be run on different processors [LBG+16].

Georgie et al. introduce LEO, a purpose-built sensing algorithm scheduler designed to maximize the utilization of sensor applications executed on mobile devices. This scheduler runs on a LPU consuming a fraction of the battery per day. Parts of the scheduler are pipeline partitioning, pipeline modularization, and feature sharing. They split the computation into chunks, group the sensor samplings together, and detect overlaps in sensor applications [GLRM16].

Romero et al. introduce a modelless system called InFaas. With InFaas, a developer only needs to define the high-level performance, cost, or accuracy required for the requests. InFaas helps to handle diverse application requirements and changes in hardware resource availability by selecting a model variant for each query. It combines VM-level horizontal scaling to add or remove worker machines and model-level autoscaling, which changes the model variants to dynamically react to changing application requirements. Additionally, it automatically manages the place, for example, GPU or CPU, where the model variant is executed to improve resource utilization [RLYK21].

3.1.2 External Distribution

In the external distribution, the load is distributed between different components. This is mostly done in a chain starting with the device which has a request itself, or an edge device in case of a sensor. The next devices in the chain are further away but have more process power. In many cases, one or more edge servers are next to the edge device. At the end of the chain is the cloud, which is the most powerful device but also has the largest distance to the request device.

Bahreini et al. focus on a multi-component application placement problem (MCAPP). In MCAPP, there are users and servers. A user wants to execute an application that consists of several components. Servers are either an edge or a cloud server. The users and servers are placed in a two-grid system that represents the current position. The position of the users can vary between time slots, but the positions of the servers are fixed. The goal is to execute the components on the server at minimal cost. First, they use a variant of the Hungarian method algorithm [Kuh55] to assign components to servers, without taking into account the communication cost of the components. This is done in a second step with a local search algorithm called L-SEARCH [BG17].

There are also a lot of papers that answer the question of when it is useful to offload a request and when it is better to execute it directly on the device. Mengyu Liu and Yuan Liu [LL18], Yue Zhang and Jingqi Fu. [ZF21], Luan N. T et al. [HPN+21], Junyan Hu et al. [HLLL20], Xing Chen and Guizhong Liu [CL21], and Qinglin Chen, Zhufang Kuang, and Lian Zhao [CKZ22] papers present different algorithms for that. Mengyu Liu and Yuan Liu [LL18], and Xing Chen and Guizhong Liu [CL21] allow partial offloading, while the other papers only allow full offloading. In Yue Zhang and Jingqi Fu. paper [ZF21] the edge server gets all the information about the requests and, therefore, has good knowledge about the whole system. Each paper simulates the experiments and has concrete knowledge about the tasks. Therefore, they know how much bandwidth each user will have, how complex each task is in terms of CPU cycles, and how far the users are away from the next powerful device like a Mobile Edge Device (MED). These are the parameters they are using in their algorithms to decide when it is useful to offload and when it is better to execute it locally. However, we don't have this kind of information, or they can vary quite fast. An example is the signal power of the WiFi. The signal power can change rapidly when the user moves his device away from the access point (AP).

Hyuk-Jin et al. implemented a web app which allows the user to evaluate the content of an image. For that an image can be uploaded to the website, after pressing the inference button, the image will be analyzed, and the result is presented in the web app. The execution of the task is split. The first part is done on the node where the web app is running. Before the inference part starts, which is the resource intensive part, the client makes a screenshot of the DOM-tree including all relevant information like the content of the image and sends the result to the server. The server receives the DOM-tree and evaluates the image. After the evaluation, another screenshot is taken and sent back to the client. The client loads the DOM-tree received from the server, which includes the result of the ML model executed on the server, and presents it on the screen [JJLM18].

[LLB21] Francesc Lordan, Daniele Lezzi, and Rosa M. Badia; in their paper, they introduce a colony approach. Devices are grouped in a colony and allow other devices in the colony to use their resources in a Function-as-a-Service manner. For that, a task is converted into a workflow which splits the task into subtasks including an order. These subtasks can then be executed on the own device or on other devices in the colony. To handle tasks, each component of the colony has an agent, which is divided into resource management (RM), task scheduler (TS), data manager (DM), and execution engine (EE). When a new task has to be executed, it sends the task to the agent. The TS decides where the task is executed. If locally missing data are fetched to the DM and executed at EE after the resources are allocated in the RM. If external, the agent of the other device is called via the API and the needed data are sent to the other agent.

3.2 Compression

Another technique is to reduce the overall execution time of a request by simplifying the problem statement.

Montanari et al. implemented a framework called ePerceptive for batteryless sensors. They were able to build a model that can be used for several resolutions. This is possible with Global Average Pooling, which reduces the output matrix to the same dimension, regardless of the chosen resolution. Depending on the energy budget, a different resolution is taken. In addition, they use Anytime DNN to provide several exit points. It is a solution where the model has not only one exit point, but several. An exit point at an earlier stage delivers a faster but less accurate result. However, in the case of energy depletion, a result with a lower accuracy can be received [MSJ+20].

A similar approach exists for edge devices. Li et al. introduce Edgent which splits the computation between an edge server and a mobile device. In addition, it also has several exit points. The edgent consists of three stages. In the first stage, a model is generated with several exit points. In the second stage, an exit point is chosen depending on bandwidth between the devices and a predefined maximum computation latency. In the last stage, the first part of the model is executed on the edge server. The partial result is sent to the mobile device, where the remaining model is executed [LZC18].

Teerapittayanon et al. also split the computation and used several exit points. In their approach, they combine edge devices, the edge, and the cloud. First, the computation runs on the edge devices directly and a combined result can be taken at the edge. If the result is below a specific threshold, the edge continues with the computation to improve the result. After that, the computation can continue on the cloud, which happens if the result is again below a specific threshold. With this approach, easy cases can be directly computed on edge devices, which delivers a fast result. Other cases, which need more computation, are outsourced to the edge and cloud [TMK17].

Bhattacharya and D. Lane implement a framework called SparseSep that modifies neural networks. SparseSep uses different variants of SVD to adjust the complexity and, therefore, the computation of layers in DNNs and CNNs. For DNNs, a K-SVD is used on top of a weight factorization. For CNNs, the calculation of a convolution task is approximated with two convolutions, which uses a pre-trained filter. However, these adjustments lead to a reconstruction error. For that reason, there is a variable in both cases that defines the degree of change. Therefore, changes are made to meet hardware constraints and keep the reconstruction error as small as possible [BL16].

Liu et al. developed a framework called AdaDeep that uses specific compression techniques with respect to predefined performance and resource constraints. The novelty in this paper is the combined usage of different compression techniques. These techniques are weight compression, convolution decomposition, and special architecture layers [LLZ+18].

Han et al. introduce a new framework that stores different variants of models in a catalog. These variants are built using different model optimization techniques, such as matrix factorization, matrix pruning, and architectural changes. Variants are selected with respect to specific accuracy and time constraints. To further optimize the accuracy, they introduce two novel techniques, called specialization and sharing. The idea behind the specialization is that classes are heavily clustered by context, which means that, for example, a specific category is more often present in the result set than other categories. With the specialization technique, a specific focus is placed on such inputs to optimize these cases. With sharing, fractions of models are used for several requests, which improves utilization [HSP+16].

3.3 Hardware Accelerators

Finally, it is also possible to use Hardware Accelerators which are optimized for the execution of DNNs.

Antonini et al. compare different hardware accelerators against each other. In their study, they compared the memory usage, execution time, and energy consumption of different accelerators. They are compared with Raspberry Pis (RPi 3B+ and RPi 4B). The result shows that the accelerators are optimized for different performance features. Additionally, the execution time and energy overhead depend on the Raspberry Pi. RPi 3B+ has a higher execution time and energy overhead, however, the idle power of it is much lower. Therefore, the battery life of RPi 3B + will be longer [AVM+19].

Reagen et al. introduce Minerva, an approach operating on the algorithm, architecture, and circuit level, to optimize DNN hardware accelerators. With their approach, a power reduction of a factor of eight could be achieved. This is done in five stages. The first two stages are for choosing the right baseline implementation for the accelerator, this is done by choosing one of 1000 uniquely trained DNNs. In the last three steps, different novel optimizations are processed [RWA+16].

3.4 Benchmark

Another important point is to use the right model for the specific problem that it should solve. Models are mostly specialized for one specific problem and bad for another. They are also optimized for specific hardware, which means that the performance of a model depends on the hardware on which it is executed. For that, it is important to compare the different components available for the specific problem.

Reddi et al. introduced an industry-wide standard for ML benchmarks. In their study, over 600 inference models were submitted from 14 companies. They differentiated between three different tasks. These tasks are image classification, object classification, and machine translation. In addition, they differentiated between different scenarios. These scenarios are single-stream, multi-stream, server, and offline. Depending on the scenario, the time between requests and the number of queries per request vary [RCK+20].

3.5 WiFi

Reducing the overhead of devices is also possible by reducing the number of devices who access it. Aili Ashipala and Joshua A. Abolarinwa demonstrate in different scenarios how users can be divided between several WiFi APs and a base station. For that, a controller is used that has all the information about the network topology which includes a base station (BS), the WiFi APs, and the users. Additionally, the controller knows about the flow of requests and users per APs. In these scenarios, the idea is to relieve the BS. New users are added through the controller. Since the controller wants to relieve the BS, it checks which access points are in the near surroundings of the user. It will connect a new user to an AP if the signal power on the AP is good enough and the AP has enough capacity for an additional user. Only if there are no APs available, it checks whether the BS has enough capacity for an additional user. The request will be rejected if there is no capacity available [AA23].

Another idea is introduced by Bhanu Priya and Jyoteesh Malhotra. They implemented an environment that helps to find the best component for a specific purpose. The environment contains a cloud, a global Software-defined Networking (SDN) controller, several local SDN controllers, and several WiFi APs. A user who wants to execute a task provides information about the service they use, such as the maximum delay. WiFi APs, which are connected to a specific local SDN controller, provide information about bandwidth capacity, network delay, and network capacity to the agent in the local SDN controller. The assignment of the agent is to connect the users to a valid AP. For that, an agent will use the information provided about the APs, the capabilities and current workload, together with the information about the service, to find the perfect match for the user. AP requests are forwarded to the local SDN controller. All local SDN controllers are connected to a global SDN controller whose task is to handle the tasks of all local SDN controllers and forward them to the cloud where all tasks are executed [PM21].

CHAPTER 4

Approach

In this section, we present our approach to improve the performance of executing machine learning (ML) requests by evaluating the current state of the system and the environment. We start by introducing various offloading strategies that are compared in this thesis. Followed by a description of the different systems and tools we use in our experiments. It continues with a discussion of feature and data collection, where we explain which tools we use to collect data and how experiments are differentiated to gather informative data. Finally, we elaborate on how the collected data are processed to create meaningful test data, including steps for filtering and preprocessing.

4.1 Resource-Aware Offloading Strategies

With the available hardware resources and information from the Access Point (AP), various strategies can be implemented. These strategies differ according to the type of information considered and the way the data is processed. Several approaches are compared in this theses.

4.1.1 Naive Offloading

In our first offloading approach, we use a naive offloading approach. This approach is used to establish a baseline and analyze the capabilities of the different components and effects on the devices. In this approach, the offloading decision is not depending on some input parameters, rather we focus on collecting and evaluating the output parameters. Resulting in getting knowledge of the effects of using the local or external device as executor and how it affects the collected data.

A pseudocode is presented in Algorithm 1. It shows that the offload decision o is an input parameter and therefore independent of other factors such as image size or hardware resources. Depending on the offload parameter o, we process the request n_i on our client machine or send it to the server. The results are stored for our analysis.

Algorithm 1 Naive Offloading Approach

```
    Input Requests:   $n_1, n_2, n_{...} \in N$
    Input Offloading:   $o \in \{true, false\}$
    Output List of results:   list
    Var Result of the request:   res

 1: function MAIN(o)
 2:     for $n_i \in N$ do
 3:         res ← offloadHandler($n_i, o$)
 4:         list.add(res)
 5:     end for
 6:     return list
 7: end function

 8: function OFFLOADHANDLER($n_i, o$)        ▷ Is connected with client and server, where the modules are running
 9:     if o then
10:         res ← sendToServer($n_i$)
11:     else
12:         res ← sendToClient($n_i$)
13:     end if
14:     return res
15: end function
```

4.1.2 Simple Decision Offloading

In this offloading approach, we are interested in whether the system is capable of switching between local and external execution, in general. For that we take a single variable. This variable can be one of our available hardware resources, or a value referring to the connection to the AP.

A pseudocode is presented in Algorithm 2. In addition to our request n_i, we need the resource of interest for our offloading decision $rName$ and the boundary for our offloading decision $rBoundary$. The *main(...)* and *offloadHandler(...)* functions are similar to the one presented in Algorithm 1, except that the offload decision o is now calculated by the function getOffloadDecision(...) with the two parameters $rName$ and $rBoundary$. The getOffloadingDecision(...) function is connected to the resource controller, which has access to all hardware resources and current connection strength to the connected access point (AP). The actual decision is made in the evalDecisionFuction(...) presented in Table 4.1. There we can see that the offloading decision varies for the different resources.

The reason is that the ping increase but the signal power decrease when the distance between the client and AP increase.

Algorithm 2 Simple Decision Offloading Approach

```
    Input Requsts:    $n_1, n_2, n_{...} \in N$
    Input Resource name of interest :    $rName$
    Input Boundary from resource of interest:    $rBoundary$
    Output List of results:    list
    Var Offloading decision:    $o \in \{true, false\}$
    Var Current Value from resource of interest:    $rValue$

1: function OFFLOADHANDLER($n_i$, $rName$, $rBoundary$)
2:     o ← getOffloadingDecision($rName$, $rBoundary$)
3:     ...
4: end function

5: function GETOFFLOADINGDECISION($rName$, $rBoundary$)
6:     rValue ← callResourceController(rName)    ▷ Resource Controller has access to all Resources at the current time
7:     return evalOffloadingDecision($rName$, $rValue$, $rBoundary$)
8: end function
```

evalOffloadingDecision(rName, rValue, rBoundary)	
rName	offload
Toggle	not Toggle
Signal Power	rValue < rBoundary
Ping	rValue > rBoundary

Table 4.1: The offload decision depends on the resource of interest. The resources are the toggle which is a pseudo resource that changes after each iteration, and the two environment specific resources signal power and ping

4.1.3 Decision Tree Offloading

From the data we derive from our first offloading approach, we build a DT for our offloading decision. The data we use for the DT are the system variables of the local device and the connection to the AP. This approach should show whether a DT is capable of making an accurate decision and how long it takes to evaluate the input parameter.

A pseudocode is presented in Algorithm 3. Compared to the previous Algorith 2 we now take a list of resources as input parameter, r_{list}. Therefore, we need to change the parameters, from one resource r to a list of resources r_{list}, for our *main(...)* and *offloadHandler(...)* functions. First, the image size is evaluated and stored in *imgSize*.

After that, all the resources of interest r_{list} are collected from the resource controller, the decision tree (DT) is executed in the *evalOffloading(...)* function. The resources of interests are hardware and environment resources. The DT then returns the offloading decision according to the input resource list r_{list} and the current image size *imgSize*.

Algorithm 3 Decision Tree / ML-Based Offloading Approach

```
    Input Requsts:  $n_1, n_2, n_{...} \in N$
    Input Resource names of interest :  $rName_{list} = \{rName_1, rName_2, \cdots\}$
    Output List of results:  list
    Var Offloading decision:  $o \in \{true, false\}$
    Var Resource:  $r$
    Var Resources of interest:  $r_{list} = \{r_1, r_2, \cdots\}$
    Var Image size:  imgSize
1: function GETOFFLOADINGDECISION($n_i, rName_{list}$)
2:     imgSize ← evalSizeOfImage($n_i$)        ▷ Depending on the byte size of the image the result will be small, medium or large
3:     for $rName \in rName_{list}$ do
4:         r ← callResourceController(rName)        ▷ Resource Controller has access to all Resources at the current time
5:         $r_{list}.add(r)$
6:     end for
7:     return evalOffloading($r_{list}, imgSize$)        ▷ This functions stores the median values for our different resources.
8: end function
```

4.1.4 ML-Based Offloading

Our final approach, ML-based offloading, is by far the most advanced method discussed in this thesis. For this approach, we need to train an ML model. This approach requires test data including system data, environment data, and the round-trip time (RTT) to be able to distinguish when to offload or not. The success of training an effective ML model that provides meaningful results requires pre-processing of the data in advance. In Section 2.4.4, we discuss how data need to be preprocessed and in Section 2.4.5, we explain how a model needs to be trained.

A pseudocode is presented in Algorithm 3. In this offloading approach, the evalOffloading function executes the ML model of interests, instead of the DT used in the previous explained approach in Section 4.1.3. Therefore, the ML decides whether the request should be offloaded with the given input resource list r_{list} or run on the client machine.

Before going into detail about the approach, we first provide an overview of the system, including a description of the two systems and the two tools used in this thesis. Then we discuss how feature and data are collected.

4.2 Implementation

We execute our experiments on two different setups, which differ in location and number of devices. For some baseline information, we are working on our first setup because it is a rudimentary setup that only contains a few devices to be able to run the experiments. We then change to a more sophisticated setup including more devices. There we focus on how the feature changes by changing the circumstances. In the following, we call the first one the preliminary system and the second the evaluation system. In both systems, we use *telemd* to monitor our features. In addition, we use *edge-chaos* in the evaluation system to change the current state of the client. Our goal is to evaluate the differences between local and external executions, to be able to decide whether it should be offloaded or not.

4.2.1 Preliminary System

In our preliminary system, we minimize as much noise as possible by using as few devices and additional programs as possible. For that reason, the preliminary system only contains a client and a server which are connected via an AP. Information about the hardware used can be found in Table 4.2. As we can see in the table, we use a Lenovo laptop for our client, a common 4G router, and a PC as a server, which represents a stronger device compared to our client.

DEVICE	DESCRIPTION
LAPTOP	Lenovo ThinkPad 13-20J2S00G00
ROUTER	hh71vm link hub, 4g router
PERSONAL COMPUTER	AMD Ryzen 5 2600 Six-Core Processor 3.40 GHz with 16GB RAM

Table 4.2: Used hardware in the small setup

We also keep the procedure for the experiments as simple as possible. A visualization of the setup is presented in Figure 4.2. In this representation, we can see the general approach to evaluating images. In a predefined frequency, the client sends an image in the form of a request to a handler. Depending on the experiment, the handler decides which device executes the request. Since both devices, the client and the server, are running a ML model, the handler can send the request to any of these devices. The device can then process the image, which can be done with two different ML models on the client and one ML model at the server, and sends the result back to the handler. The handler then forwards it to the next level.

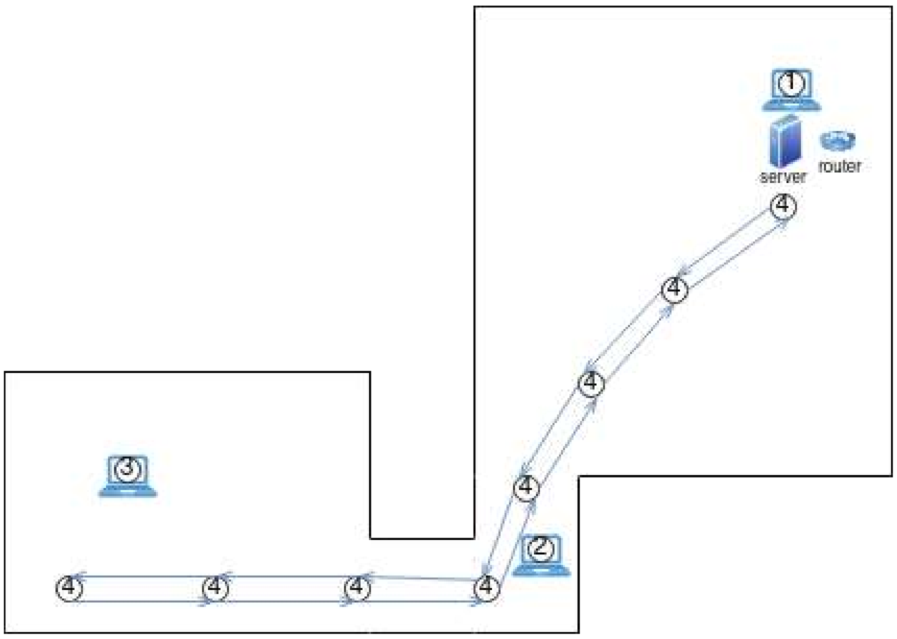


Figure 4.1: Environment overview in the preliminary system

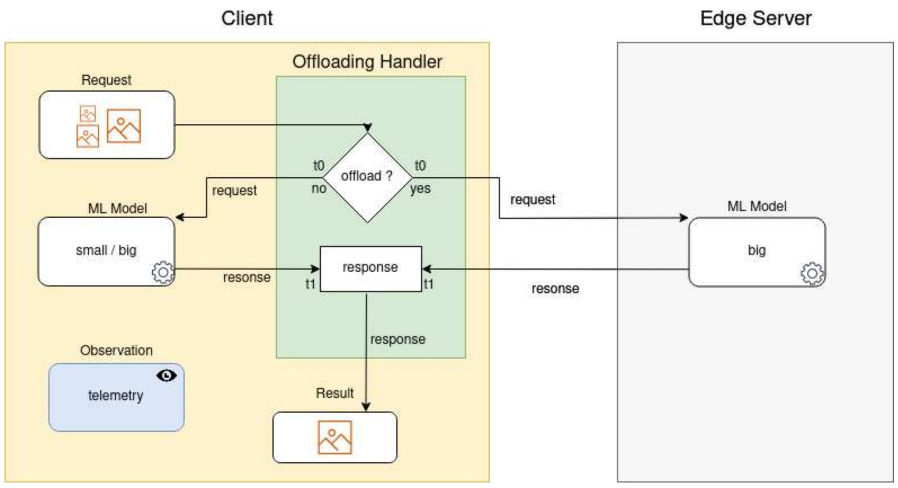


Figure 4.2: Request handling in the preliminary system

In Table 4.3 the information about the two different models is summarized. As we can see, the two models differ in depth. For that reason, the one with the lower depth is faster and smaller. However, this leads to lower accuracy.

Description	Model Name	Input Size	Depth Multi.	Latency	Model Size
Small	MobileNet v2 DeepLab v3	513x513x3	0.5	36.8 ms	1.1 MB
Big	MobileNet v2 DeepLab v3	513x513x3	1.0	43.0 ms	2.9 MB

Table 4.3: Summarized information about used semantic segmentation model

4.2.2 Evaluation System

In our evaluation system, we changed to a more realistic test environment. This new environment contains several APs and a new server as well. A list of all devices with the description of the underlying hardware is presented in table 4.4.

Device Name	Description
Local Device	Lenovo ThinkPad 13-20J2S00G00
Iss	Linksys WRT1200AC
Starlinke	Netgear EX6120
Stargate	Netgear EX6120
Server	ASRock Industrial

Table 4.4: Hardware description of the environment

A visualization of our setup and the location of the devices is presented in Figure 4.3 We can see that there are three APs, called *starlink*, *stargate*, and *iss*. The APs *starlink* and *stargate* are repeaters and only forward traffic to the third AP *iss*. *Iss* is a router that is connected to repeaters, the server, and the Internet. Additionally, the figure also contains information on two journeys and the place for our experiments with a fixed location. The first journey, in green, starts with the green circle and goes through the whole building according to the arrows. The second journey, in red, starts with the red circle next to the symbol for the local device. After the client exits the room, the journey continues by going up and down several times. For experiments in which the client stays at one place, the location is used next to the AP *starlink* marked with an X. We will refer to these different movements of the client's experiment setup as Journey 1, Journey 2, and Stationary.

The server in this approach runs the same model as before, but has better HW resources, which allows the server to execute the requests faster, compared to the server in our last approach. The changes in the new system give us new opportunities. Compared to the previous system, we now not only have the opportunity to decide whether to execute it locally or remotely, but also the opportunity to choose between different APs. Using

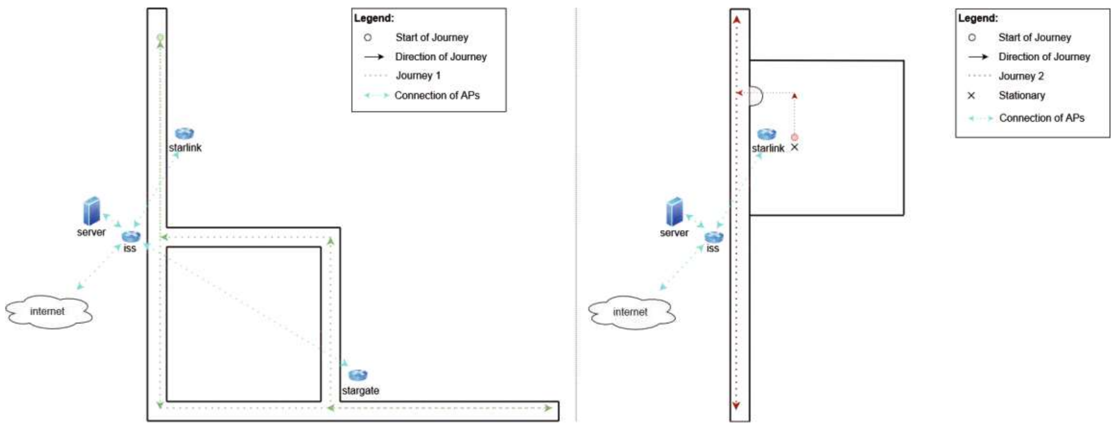
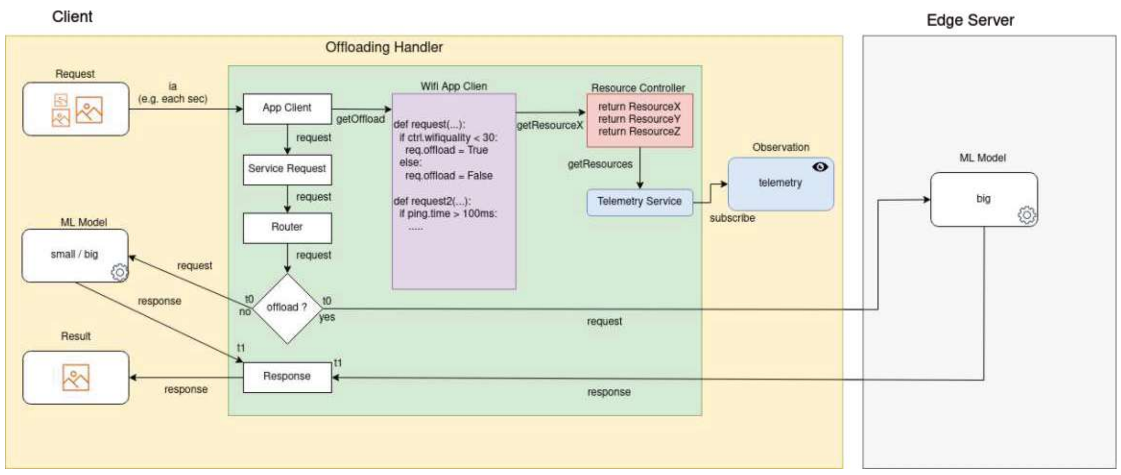


Figure 4.3: Setup of the test bed environment with the Journeys 1 and 2

different APs includes different amounts of latency, which is also more realistic compared to the last system.

We also change the way in which the decision is made about the execution of the request. Figure 4.4 visualizes it. It presents an extension of the Galileo framework described in Section 2.6.1. The extension includes the *Wifi App Client* and the *Resource Controller*. We can see that the processing of requests in general is now much more complex than before, compared to the approach in our preliminary system, visualized in Figure 4.2.



```
def request(...):
  if ctrl.wifiquality < 30:
    req.offload = True
  else:
    req.offload = False

def request2(...):
  if ping.time > 100ms:
    .....
```

Figure 4.4: Request handling in the evaluation system

Not only is the offload handler much more sophisticated now compared to before, but it also starts the client a worker. The worker could run on a different device and send the requests from there. However, we start the worker on our own machine. The worker is responsible for sending the request in a predefined interval. This worker does not have any knowledge about the current state of the client or the server. Therefore, it is now possible that the worker overloads the server with too many requests, leading to additional latency. Additionally, the offload handler can now use HW resources and information about the environment for the offloading decision. For that, the *App Client* collects the offload decision from the *Wi-Fi App Client* and propagates the request with the offloading decision to the *Service Request*. The *Service Request* prepares the request and propagates it further to the *Router*, which decides whether the request is offloaded or not, depending on the decision of the *Wi-Fi App Client*. The *Wi-Fi App Client* uses the Resource Controller, which provides data of *telemd* which it gets from the *Telemetry Service*. We defined different functions in the *Wi-Fi App Client*, giving us the opportunity to regulate when requests should be offloaded or not. We also intend to incorporate another machine learning model into this decision-making process which is now possible with these changes.

4.3 Tools

In our thesis we are using two tools, which allow us to collect different information during the experiments. The first tool, called *telemd* is used to monitor hardware resources and data from the network connection. The second tool, called *edge-chaos* is used to cause chaos on the local machine.

4.3.1 Telemd

Telemd [1] is a lightweight tool that reports fine-grained system runtime data into Redis. It provides information about the system in general, including CPU, RAM, Freq., etc., and information about running containers on the system. It also provides information about the signal power and ping to the currently connected AP. With *telemd*, it is possible to decide which information we want to collect and at what frequency we want to collect them. A list of monitored metrics with associated frequency is presented in Table 4.5. Most instruments provide information about the current state of the system, e.g., *load* is a measure of CPU utilization, where the processes currently executed by the CPU or on the waitlist are counted [2]. However, there are some instruments that store accumulated information over time. An example is *psi*, which provides information on pressure stall by aggregating productivity losses caused by resource scarcity [3]. Additionally, the container measures CPU usage over time. To be able to use these instruments properly and to get information about the moment, we preprocessed these data. We do that by changing the value to $x_{new} = x_n - x_{n-1}$. Therefore, we subtract the current value from the last to obtain the change in the current period of time. To collect our last feature, the RTT, we use two timestamps t_0 and t_1, taken when a request is sent and when the result is received, respectively. The resulting RTT is calculated by t_1 - t_0.

[1]https://github.com/edgerun/telemd
[2]https://www.digitalocean.com/community/tutorials/load-average-in-linux
[3]https://www.kernel.org/doc/html/latest/accounting/psi.html

Instruments	Description	Frequency
cpu	CPU utilization	1s
$docker_{xy}$	docker container x ∈ {redis, telemd, model}	1s
	y ∈ {blkio, cpu, memory}	1s
freq	Sum of clock frequencies of the main CPUs	1s
$load_x$	System load average of the last x minutes x ∈ {1,5}	1s
procs	number of procs	1s
psi_x	host's x pressure; x ∈ {cpu, io, memory}	1s
ram	RAM used in kilobytes	1s
rd	reads	1s
signal_wlp3s0	signal power of the WiFi	1s
wr	writes	1s

Table 4.5: Hardware-Resources monitored with telemd

4.3.2 Edge-chaos

Edge-chaos [4] allows the user to start additional programs to disrupt co-located applications. It runs in another container and waits to receive commands. With this tool, it is currently possible to provide commands for *stress-ng* [5] and *tc* [6]. *stress-ng* is a tool that will stress the system by executing different operations such as floating-point, integer, bit manipulation, and control flow. *tc* is a Linux tool that can modify traffic on the network interface. A use case would be to limit the available bandwidth. However, we decided to focus on the *stress-ng* tool. With *stress-ng* it is possible to start more than 200 different attacks. The parameters we use are summarized in Table 4.6 [7]. With *malloc* we are able to increase current CPU usage by 25% per worker. Additionally, *malloc-bytes* increase current RAM. Finally, we also include *io* to simulate save operations, since the data are stored in the underlying file system. With that, we aim to simulate a different application running on the system with a roughly static load during the different experiments.

Option	Parameter	Description
Malloc	number of workers	calls malloc, calloc, realloc, and free
Malloc-Bytes	max. bytes per (re)allocation	allocation between 1 and N (parameter) bytes
IO	number of workers	calling sync

Table 4.6: Used stress-ng parameters

[4] https://github.com/edgerun/edge-chaos
[5] https://github.com/ColinIanKing/stress-ng
[6] https://man7.org/linux/man-pages/man8/tc.8.html
[7] https://manpages.ubuntu.com/manpages/bionic/man1/stress-ng.1.html

4.4 Feature Collection

In our approach, we can collect features from three different categories. These three categories contain information about the HW resources of the client's device, the environment, and the resulting information we collect through our experiments, which are different time intervals, like the RTT in particular. To collect features from the first two categories, we use the lightweight open source monitor tool *telemd*. The third category is collected by measuring the elapsed time. To be able to collect proper test data, it is important to get a variance in our features. Our features of the first two categories have an impact on the feature in our last category, the RTT. Therefore, varying the features in the first two categories is sufficient.

To vary the features in our first category that contains HW resources, we use the *edge-chaos* tool. To vary the feature in the environment category, we have to change the connection to the AP. Instead of using *tc*, which is included in the *edge-chaos* tool, we decided to move the device during experiments and therefore change the quality of WiFi.

4.4.1 Acquiring of Divergent Data

Our features are collected through different experiments. The goal is to run different experiments in different scenarios to obtain a large variety in our test data. With that, we hope to find the characteristics of our features, depending on the current situation. A scenario is defined by the choice of some variables. We are using different scenarios to assess the impact of the variables. The following variables, all except the first, are also later used for training our machine learning (ML) model.

- Devices per experiment
- Device who executes the request
- Kind of model which processes the request
- Distance between the client and the AP
- Size of the request (image size)
- Confounding factors (additional load)

To minimize additional costs, such as latency introduced by transmitting data over different devices, we first divided the experiments by the number of devices.

We then categorize the experiments on the basis of the location of request execution. When requests are executed on the same device on which they were created, it is termed a local experiment, as only one device is involved in sending and processing the request. An external experiment, on the other hand, includes another device as the request is not processed on the same device on which it was created.

For the processing part, we also distinguish the ML Model, which is used to process the request. For our client, we use two different models summarized in Table 4.3, the server on the other hand uses the larger model of them, since it represents a stronger device where the impact of the model execution is negligible compared to the additional latency costs included through sending the request and receiving the result. In addition, we categorized the distance between the client and the connected AP.

For local experiments, the distance is irrelevant since the data is not transmitted to another device. For external experiments, we distinguish between static and dynamic tests. Static tests refer to experiments where the position of the client remains constant, while dynamic tests refer to experiments where the client moves during the experiments.

Furthermore, we perform experiments with different image sizes. Using different image sizes changes the period of time it takes to transmit the request to another device, depending on the current signal power on the connected AP. Additionally, it changes the time the ML Model needs to process the request, since the ML Model is designed for a specific resolution and therefore needs to adjust the resolution depending on the size of the image. The differences in the images are presented in Table 4.7. For the small image, we use an open source collection called VOC12[8]. Since they only provide small images, we have created some images, which we are using for the other two image sizes. The images were taken on a Samsung Galaxy S9 with the main camera.

Description	Resolution	Source
Small	512x512	VOC12
Medium	2018×1512	Samsung Galaxy S9
Large	4034×3024	Samsung Galaxy S9

Table 4.7: Description of the used images through all experiments

Finally, we differentiated between experiments where the client only ran the necessary programs for the tests and those where the client had an additional workload from running other tasks unrelated to the experiment. To introduce an additional workload, we utilized the *edge-chaos* tool, as previously described.

We perform these experiments in the preliminary and evaluation system. In both systems, we have at least a client, an AP, and a server. In our scenario, the client is running an Augmented Reality application, where information from images has to be evaluated at a predefined frequency. Each image is a request and can be processed either directly on the client or offloaded to the server. For that, the client and the server run a semantic segmentation ML model. With a semantic segmentation model, different types of objects can be identified. A more detailed description can be found in Section 2.4.6

In our experiments, we are starting the models on both devices before we start the experiments. Starting the model from the beginning simplifies the request, since we do

[8]http://host.robots.ox.ac.uk/pascal/VOC/voc2012/

not have to consider whether the model is already started or not. In the other case, the first request would suffer from additional latency since the model would have to start before it could process the request. An option to prevent that would be a pseudo request to signal the model to start. However, this would complicate the experiments. Therefore, our models are running and are ready already at the beginning of our experiments.

4.5 Creating test data

In this section we will describe the steps necessary to gather the proper test data needed for an DT or ML model Offloading approach.

First, it is important to understand that the data are environment-specific. Therefore, it is not advisable to mix the data from the preliminary and evaluation system, because it would distort our data. This means that the different devices and setups we use in the two systems have an impact on the result. Since the data collected in the preliminary system lack diversity and are not representative, because we avoided noises when possible, we use the data from the evaluation system.

Second, we need to define when it is advisable to offload or not. This is also very important for our supervised learning approach, explained in Section 2.4.5, which we use to train our ML models. In our case, we label the experiments with 0 and 1 for local and external execution, respectively.

In the end, it is important to filter and preprocess our test data. This is done according to our research result mentioned in Section 2.4.4. These steps include removing outliers, normalization, and adjusting the test size.

Labeling

For the two labels 0 and 1, which represent the experiments that should be performed locally or externally, respectively, we need to define a common rule. Since we do not have deadlines to follow, we decided to compare each request with the median value of the round-trip time (RTT) for the different experiments, presented in Equation 4.1.

$$t_{ref}^{l,s} = \forall_l \forall_s median(\sum_{i=0}^{n} r_i^{l,s}) \tag{4.1}$$

There we can see that the time reference $t_{ref}^{l,s}$ for a specific load l presented in Table 4.6, a specific image size presented in Table 4.7 and a specific amount of requests n, is defined by the median value over all requests $r_i^{l,s}$, where $r_i^{l,s}$ is the RTT of the request i, for a specific l and s. The label is defined in Equation 4.2. We can see that the label is 1 when the request is below and 0 when above the median.

$$l_i^{l,s} = \left\{ \begin{array}{ll} 1, & r_i^{l,s} \leq t_{ref}^{l,s} \\ 0, & r_i^{l,s} > t_{ref}^{l,s} \end{array} \right\} \tag{4.2}$$

4.5.1 Filter and Preprocessing

Removing outliers

Before normalizing, it is crucial to filter the data to eliminate outliers. Normalization processes tend to bring all data points closer together, potentially resulting in significant information loss if outliers are not addressed beforehand.

When outliers are present, normalization can cause most values to converge around a single point, effectively reducing the variability of the data set. Consequently, the data would lose much of their informative value, making it challenging to discern meaningful patterns.

To mitigate this issue, we employ a common outlier removal technique based on the z-score, which involves removing values that deviate from the mean by more than a specified multiple of the standard deviation. The choice of this multiple impacts the extent of the removal of outliers, thereby influencing the meaning and distribution of the remaining data. Striking a balance is essential, as overly aggressive removal may bias the dataset, leading to sub-optimal training of the ML model. In contrast, insufficient removal could result in data points being too closely clustered, making it difficult to differentiate between them and hindering decision making. Therefore, careful consideration must be taken to strike an appropriate balance between the removal of outliers and the retention of valuable information.

Normalization

After we removed the outliers, we continued to normalize our data with min-max normalization. In Equation 4.3 we can see that we first calculate the min $v_{min}^{l,s}$ and max $v_{max}^{l,s}$ values for each load l and image size s. After that we can calculate the normalized value by subtracting $v_{min}^{l,s}$ from the current value $v_i^{l,s}$ and then divide it by the difference of $v_{max}^{l,s}$ and $v_{min}^{l,s}$. As a result, all values are now between 0 and 1.

$$\begin{aligned} v_{max}^{l,s} &= \forall_l \forall_s max(\forall_i v_i^{l,s}) \\ v_{min}^{l,s} &= \forall_l \forall_s min(\forall_i v_i^{l,s}) \\ v_i^{l,s} &= \forall_l \forall_s \forall_i \frac{v_i^{l,s} - v_{min}^{l,s}}{v_{max}^{l,s} - v_{min}^{l,s}} \end{aligned} \tag{4.3}$$

Proper test data

Now our data are normalized and free of outliers, however, they are not yet good enough as test data. The reason is that the ratio of our test data is still biased. Depending on the stress level and the image size, our data are biased to either a local or external execution. This means that we would train our model for a specific stress level and image size with a preferred result for local or external execution. This would lead to a general bad result for the other execution. For that reason, we train the models with approximately the same amount of data from both decisions.

4.5.2 Test data for Decision Tree and ML-Based Offloading

In Section 4.5 we describe an approach to evaluate performance gains. This approach involves calculating the median, as visualized in Equation 4.1. We will employ this method to compute the median value for different experiments, taking into account hardware load and image size. Specifically, we use the results from the *stress-level* measurements. There we execute 12 experiments, both local and external. These 12 differ in four different stress levels and three different image sizes. We then use these 12 local execution experiments to calculate 12 distinct median values. These median values serve as references for the external execution results. During external execution, the RTT is compared with the corresponding median value for the same hardware load and image size. Requests with RTT faster than the median are labeled as *true*, while those slower are labeled as *false*. This labeling scheme enables us to train our models using a supervised learning approach, where each label indicates whether a specific request should be offloaded to the server or executed locally on the machine.

CHAPTER 5

Evaluation

In this section, we address the different offloading approaches mentioned in Section 4.1. After discussing the different approaches in more depth, the section ends with the key performance indicators. These indicators explain how the different approaches are compared with each other.

5.1 Naive Offloading

In this approach we cover all tests that provide us with general information about our systems and devices and are processed solely on the local or external machine.

5.1.1 Preliminary System

In this setup, we execute three different sets of experiments. The position of the devices during these experiments is visualized in Figure 4.1. There we can see that the AP and the server are next to each other. Due to the small distance between the AP and the server, the transmission should be quite fast, the connection should be strong, and therefore the latency should be low. The client can have several locations. The location depends on the experiment. With the experiments called *Model overhead*, *Local execution*, and *External execution*, we want to gain some general information about our client and the execution of ML requests on different devices. The experiments are explained in the following.

Model overhead

In our first experiment, we aim to assess the impact of running different models on the client. To this end, we monitor the hardware resources using *redis* and *telemd*. To obtain the overhead that the model produces, we first run the test without starting any model at all. After that, we repeat the experiment twice, starting the two different models separately. Then, we compare the results. The overhead of the model is the additional use of resources compared to the resources the client needs without the model. With this experiment, we evaluated how the model affects the client and whether the client is capable of executing a model, in general.

Local execution

In our next experiment, our objective was to assess the impact of processing requests with our different models on the client. As before, we used *redis* and *telemd* to monitor hardware resources. The requests were distinguished by image size, with the sizes of small, medium, and large images listed in Table 4.7 and the used models listed in Table 4.3. In this experiment, we analyze the additional resources the client needs to execute requests and the duration of the execution of a request. We can evaluate the overhead of the resources by running the requests of one of the models on the client directly and compare the results with the results we got in the last experiment where we ran the model solely. To evaluate the duration of the experiment, we measure the round-trip time (RTT), which can be calculated by the absolute difference of the two timestamps t_0 and t_1. In Figure 4.2, we can see that t_0 is taken at the time when the offload handler sends the request to a ML model and t_1 when the offload handler receives the result from the executed ML model. The overhead of the resources shows us how the execution of requests affects the hardware resources. The RTT is stored and will be used in the next experiment as a reference value.

External execution

In the last experiment in this setup, our objective was to assess the impact of the connection between the client and the AP on the performance gain by executing the request on a stronger device. In this experiment, the offload handler on the client sends the request to the server over the connected AP. The server uses the same model compared to the bigger model of the two available to the client. To evaluate the impact of the connection, we changed the position of the client, resulting in a change in bandwidth and network latency. In Figure 4.1 the different locations of the client are labeled with numbers 1 to 4. The first three numbers are fixed locations, and the last four-labeled number represents a route. There, the client moves during the experiment. The start and end are next to the server and the AP. The farthest point is near number three, where the signal power is bad. In total, there are eight stations. Therefore, it is possible to go there and back visiting all stations, except the last one, two times for one minute each. A total of 12 measurements are taken, with the experiment being carried out on each position using three different images. The resulting HW data collected by the client with *redis* and *telemd* should be similar compared to the *model overhead* experiment, since the model does not have to execute any requests and therefore similar programs are running on the system. The resulting RTTs collected in the 12 measurements can be compared with the RTTs collected in our *local execution* experiment, described in the previous section.

5.1.2 Evaluation System

For the remaining experiments, we switch to the evaluation system. The first two experiments present changes in the signal power depending on the AP and location used. The third one presents a boundary test and the last one shows how the load affects the performance of the local machine.

Variance with different AP and Locations

1. Change in signal power in a fixed position and while moving

In a small first experiment in this new environment, we evaluated the range of the different APs and how they vary. We collect data with a command line tool called *nmcli* [1]. With that tool, we collect the signal power quality of the surrounding APs. We repeat this experiment several times.

One experiment is on a fixed location, the other one we walk on Journey 1. The route is visualized in Figure 4.3 with the green arrows that represent the way there and back, starting at the location of the green circle. In that time, moving the green arrows, we collected 20 data samples with a 5 second break between the samples. The data samples contain information on the WLAN connection at that particular moment. With that experiment, we want to analyze how the signal power can vary in a fixed location and in a dynamic scenario.

2. Changes depending on the used access point

In another experiment, we evaluated the experiments from a stationary location. The location of the device is visualized in Figure 4.3 with the laptop icon. At this location, the client has access to the APs *starlink* and *iss*. With this experiment, we analyze the differences in the APs. We can see that the *starlink* is next to our client. Therefore, the connectivity to *starlink* will probably be better compared to the connectivity to *iss*. However, *starlink* is not directly connected to the server, it forwards the request further to *iss*. Therefore, sending the request to *starlink* not only sends the request in a wrong direction first, but also increases the hop count by one.

Boundary Test

In this experiment, we evaluate the quality of the connection when we move the client to the edge of the signal power range. Moving the client to the edge drops the signal power and increases the ping. We are interested to see how the RTT changes with these conditions.

Stress test

In all previous experiments, we offloaded all experiments to analyze how the RTT changes depending on the circumstances. However, not only the environment can change, but

[1] https://developer-old.gnome.org/NetworkManager/stable/nmcli.html

also the circumstances of the client itself. Therefore, in this experiment, we put an extra load on the client. As mentioned above, we use a tool called *edge-chaos*, which simplifies the use of *stress-ng* for us. With *edge-chaos*, we can tell *stress-ng* to add / remove an extra load on our HW resources during the experiment. We use three options in *stress-ng*, these options are described in Table 4.6. In our experiments, we used four different stress levels. Starting with level zero, where no extra load is introduced and used as a baseline, and increasing the load on each level.

For this experiment, we use Journey 2 visualized in Figure 4.3. We let time lapse away in the first room before we go through the door and move up and down in the hall. The reason for this is that we want to collect data in different signal power ranges. After leaving the first room, the signal power drops because we increase the distance to the AP *starlink*. Therefore, we stay a bit in the room to get data with a good signal power as well.

We execute this experiment on the client and repeat it on the server. We use these data later for our baseline because the data contain all the information we need. This information is a variance in the environment and the current state of the client.

5.2 Simple Decision Offloading

The experiments for this offloading approach are executed in the evaluation system and should show whether offloading or not with a single parameter is possible and sufficient. The experiments are carried out in the same way as the *stress test* experiment. Therefore, we again use Journey 2.

We evaluate and compare three different parameters with each other. The first parameter is a flag that we toggle after each execution. Therefore, the *Wi-Fi App Client* alternately responses with *true* and *false*. As a result, we alternate between local and external execution. We executed this experiment two times; first, the client stays at the stationary location, and second, it moves according to journey 2. With this measurement, we analyze whether the system is capable of changing between different devices or whether the execution of the request on changing devices increases the RTT in some way.

Second, we use the resource controller, visualized in Figure 4.4, to obtain recent measurements of our signal power. We then use the signal power as a constrained on different boundaries to decide whether to offload or execute the request locally.

Last but not least, we repeat this experiment with the ping instead of the signal power. Again, we use different boundaries for our offloading decision.

With the last two, we analyze whether it is sufficient to use one of these parameters alone for our offloading decision.

The data we have collected in all experiments so far give us the opportunity to evaluate how the feature changes depending on the circumstance. We use the data we collect in these experiments for filtering and prioritizing.

5.3 Decision Tree Offloading

At this approach, we have complete control over the decision. We can decide how important the different features are and at which point the request should be offloaded. For that, we analyze the data on our own and create a Decision Tree (DT). In this approach, we can find obvious correlations between the features and the RTT. However, some features could also be relevant but less important compared to other features. These features are hard to find by simply analyzing the data without any tool. Therefore, this approach is a very broad approach but, for that reason, a fast one.

5.4 ML-Based Offloading

For the remaining approaches, we entrust the decision-making process to a machine learning (ML) model. Numerous ML models exist to address this problem, each offering varying levels of evaluation time, efficiency, and resource utilization.

Although it is possible to tailor the ML model to specific needs, this process can be time consuming and may not always yield satisfactory results. To streamline this process, we use a tool known as TPOT, as detailed in Section 2.6.2. TPOT relieves us of the burden of manually selecting features and ML models, instead allowing the tool to autonomously determine the most suitable features and ML model based on our input dataset.

TPOT offers several default configurations, differing in generation, population, search grid, and goal. We intend to generate ML models using all default configurations and subsequently compare their performance with each other.

However, before we can use a Decision Tree and ML-Based offloading approach, two important steps are necessary. First, in both cases, it is necessary to make a decision on the features available during the experiment. For that both offloading approaches need to know whether a feature is good or bad, otherwise the resulting decision is not related to the experiment. Therefore, we have to define how we want to compare and evaluate different experiments. Second, our current data are raw data. Using these raw data can affect the performance of both approaches. To avoid that the data lead to poor results, they have to be filtered and preprocessed. The next two sections describe these steps.

5.5 Key Performance Indicators

For evaluating the different offloading approaches with each other, we will run the same experiment over and over. We will stick to the variety we started to use in the *stress-level* experiment where the client moves according to Journey 2 visualized in Figure 4.3. These measurements include both local and external executions, thereby offering data reflecting various hardware loads, environmental circumstances, and image sizes.

Additionally, our ML models are trained with these data and are therefore designed for this experiment. Otherwise, it would be harder to argue why a ML model performs good or bad in these tests. With the results of the experiments we want to compare the approaches on the following criteria:

- Introduced overhead by using the approaches
- Accuracy
- Improvement compared to baseline

Offload evaluation overhead

Since our goal is to reduce the total time the client has to wait for the response of the request, it is important to evaluate the time the different approaches are taking to evaluate whether the request should be offloaded or not. Therefore, waiting too long for the response of the evaluation process leads to a bad result, in general.

Accuracy

The estimated accuracy is a value automatically generated by training a ML model. Therefore, this value is only available for the ML model offloading approach, but not for the other approaches.

5.5.1 Points 🪙

It is an experiment-oriented value calculated after we run the experiments. We can compare whether the result is better or not by using Equation 4.1, which was originally used to define the labels in our supervised learning approach.

$$p_i^{l,s} = \left\{ \begin{array}{ll} 1, & r_i^{l,s} \leq t_{ref}^{l,s} \\ 0, & r_i^{l,s} > t_{ref}^{l,s} \end{array} \right\} \tag{5.1}$$

$$P^{l,s} = \forall_l \forall_s (\sum_{i=0}^{n} p_i^{l,s}) \tag{5.2}$$

With that we then calculate the sum over all requests which are at least as good as the reference value, presented in Equations 5.1 and 5.2. To evaluate which decision performs better, we can compare the different scores $P^{l,s}$, also called points 🪙. With that, we can count how many requests are at least as good as the reference value, but it gives us no information about the total time save we get by using this approach.

Improvement compared to baseline

In the last evaluation, we are interested in the absolute improvement. Since our experiments differ in the number of samples for each request, we will use the mean value for the various experiments. To calculate the mean, we use an equation similar to Equation 4.1. The main difference is that we use the mean instead of the median, which means that outliers will have a greater impact compared to using the median value.

CHAPTER 6

Results

In this chapter, we present the results of our experiments. As described in Section 4 our experiments are separated in different offloading approaches. These approaches are naive, simple, decision tree-based, an ML-based offloading approach. In the naive offloading approach, experiments are executed in both systems, called preliminary and evaluation systems. The preliminary system is used to collect fundamental data from the devices with the tools we are using. After collecting and evaluating the data, the remaining experiments are executed on the evaluation system. These include the experiments in the naive offloading approach, but also the remaining offloading approaches.

In this system, we evaluate different features and how they change when the state of the client or the environment changes. We then use these data to prioritize and filter the collected features. We want to find the features that correlate best with the round-trip time (RTT) to be able to decide when to offload or not. For this reason, we compare the different offloading approaches with the baseline and each other.

For a better presentation of the tables, we will use icons that are listed in Table 6.1

[▮▯▯]	Small Image
[▮▮▯]	Medium Image
[▮▮▮]	Large Image
$\Uparrow$	Uploaded to Server
🖼 $\Uparrow$	Amount of Images uploaded to the Server
🪙	Points
$\bar{x}$	Mean value
$\tilde{x}$	Median value
σ	Standard deviation

Table 6.1: List of Symbols used in Tables

6.1 Naive Offloading

In this section we run different experiments on two systems. In the first system, the tests are used to collect rudimentary information about the tools and the devices. In the seconds system, we change to a more sophisticated system. This system represents a more realistic approach including different services working together. Additionally, this system allows one to switch between the processor of the images during a single experiment.

6.1.1 Preliminary System

In this section, we present the results of the experiments carried out in the preliminary system, visualized in Figure 4.1. This system consists of three devices: the client, the server, and an AP. The client is able to send the requests to the server via the AP, or process them directly. Information about the different devices is presented in Table 4.2. The approach we use in this system is rudimentary and is visualized in Figure 4.2. Here, we send the images to an offload handler. This offload handler sent the request to the client or the server, depending on a predefined flag. After execution, the results are sent back and collected for evaluation. In addition to this approach, we avoid the execution of any other program which is not part of the experiments. With these decisions, we try to reduce the amount of noise and improve the quality of our data.

In our experiments, we use two different models, the details are summarized in Table 4.3. They distinguish in the depth and therefore in the accuracy and processing time of an image. These models will process different images, which are categorized by size, summarized in Table 4.7. Each experiment takes 15 minutes. In this time, we collect hardware resources with *telemd*. With this duration, we minimize the impact of significant background tasks as we had no control over them, and they might otherwise affect our results. Any significant background tasks that might occur during the experiments and consume more resources than the idle mode could be identified as an outlier. In experiments in which requests are sent, we choose a time interval of three seconds. The reason for this decision is the sequential execution of requests in our model, and we decided not to include a queue in these experiments.

In our first series of experiments, we solely use the client's machine and focus on some baseline tests. In these tests, we execute the model, process requests on the client, and evaluate its impact. After that, we continue with sending the requests to the server. Finally, we evaluate the differences in RTT between local and external executions.

Model overhead

Before processing requests, we are interested in the idle state. For that we measure different HW data with *telemd*. As we can see in Table 6.2 we measured the HW data three times. The first time we even run it without any model at all. In the remaining two, we run it once with the small model and once with the big model. For the 16 features, we present the medians and standard deviations.

These features are hardware resources, including information about the running container. In our case, we are running three different containers, *model*, *telemd*, and *redis*. The information of the two docker container *telemd* and *redis* are unimportant since the tasks of these containers are the same, independent of the current running model. However, information about the container where the model is running inside could be interesting when requests are processed in the model.

In our result presented in Table 6.2 we can see the following facts. To begin with, when no model is running, the CPU is almost constant with a standard deviation (std.) of $7.50e-01$, with models, the cpu is a bit higher with a mean utilization of $2.50e+00$ instead of $2.00+00$ and a higher σ with above $3.00e+00$. This also has an impact on $psi\_cpu/full$. PSI stands for Pressure Stall Information information and identifies and quantifies the disruptions caused by resource crunches and the time impact it has on complex workloads [1]. We can see that the order of magnitude for the mean value is around $e+03$ but for the std. It is about $e+02$ for no model, but $e+03$ and even $e+04$ for small and big models, respectively. Therefore, using the model increases the probability of experiencing disruptions due to excessive over-commitment, which is a side effect on high psi values. For the docker container information, we can see that running the model also introduces a constant increase of cpu and memory independent of the model size.

As we can see, in the idle mode, the container more or less needs the same CPU and memory independently of the running model. We also measured but did not mention different read and write features, since most of the results are zero.

[1] https://www.kernel.org/doc/html/latest/accounting/psi.html#psi

Instruction	No Model		Small Model		Big Model	
	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
cpu	$2.00e+00$	$7.50e-01$	$2.54e+00$	$3.22e+00$	$2.53e+00$	$3.47e+00$
freq	$5.98e+06$	$1.96e+06$	$5.84e+06$	$2.31e+06$	$6.22e+06$	$2.19e+06$
load1	$1.50e-01$	$8.12e-01$	$1.10e-01$	$6.56e-02$	$1.65e-01$	$8.53e-01$
load5	$4.50e-01$	$3.12e-01$	$1.80e-01$	$8.40e-02$	$4.75e-01$	$3.15e-01$
psi_io/full	$0.00e+00$	$6.87e+03$	$1.05e+03$	$9.03e+03$	$1.09e+02$	$7.80e+03$
psi_io/some	$0.00e+00$	$7.51e+03$	$2.44e+03$	$9.86e+03$	$1.01e+03$	$8.66e+03$
psi_cpu/full	$3.54e+03$	$7.20e+02$	$6.06e+03$	$1.19e+03$	$4.02e+03$	$7.15e+04$
psi_cpu/some	$7.32e+03$	$2.12e+03$	$1.12e+04$	$5.85e+03$	$9.38e+03$	$6.21e+03$
procs	$5.00e+00$	$2.27e+00$	$6.00e+00$	$1.80e+00$	$6.00e+00$	$1.87e+00$
ram	$2.38e+06$	$6.39e+03$	$2.43e+06$	$1.41e+04$	$2.43e+06$	$5.65e+03$
docker_cgrp_cpu/model	*nan*	*nan*	$1.18e+05$	$1.08e+07$	$1.19e+05$	$1.25e+07$
docker_cgrp_cpu/telemd	$2.08e+10$	$1.19e+10$	$2.85e+10$	$1.75e+10$	$2.20e+10$	$1.26e+10$
docker_cgrp_cpu/redis	$2.49e+09$	$1.41e+09$	$3.03e+09$	$1.74e+09$	$2.96e+09$	$1.68e+09$
docker_cgrp_memory/model	*nan*	*nan*	$3.53e+07$	$1.81e+05$	$3.53e+07$	$1.70e+05$
docker_cgrp_memory/telemd	$8.93e+06$	$4.15e+05$	$9.54e+06$	$4.52e+05$	$8.75e+06$	$3.64e+05$
docker_cgrp_memory/redis	$2.27e+06$	$3.09e+04$	$2.27e+06$	$1.61e+04$	$2.26e+06$	$4.03e+04$

Table 6.2: Collected information from *telemd* with mean $\bar{x}$ and std. value σ

Local execution

In this experiment, we start using the models by running requests with different images on the client machine. As a result, we measure the effect on the HW and the round-trip times (RTTs) of each request. We collect these RTTs and compare them later with the execution on the server. Figure 6.1 visualizes the RTTs of the different models and images.

We can see that the RTTs of the experiments, independently of the image size, executed on the smaller model are significantly lower compared to the bigger model. This gap is due to the different depths of the models, as described in Table 4.3. Since the depth of the smaller model is half as deep as that of the larger model, the evaluation time on the model will also be half as long. We can also see that there is also a difference in the RTT depending on the image size that is processed. These differences are due to two facts. First, the image is sent to the model, since the size of the image is different, it takes more or less time to send it, depending on the image size. Second, and more importantly, the model is designed for a specific resolution. Therefore, rescaling also takes time, which is longer when the input image is larger.

In the two Figures 6.2 and 6.3 we see the visualization of the HW resources. All plots are grouped by the model we use for the execution of the requests and then by the image size that is sent in the experiment. As we can see, the figures contain a *zero* image.

The data for image *zero* are the data we collect with an idle application with our models loaded, executed in our previous experiment. We use these data to highlight the differences between the processing requests on the models and the models in their idle state. Since we are interested in those HW resources that change with the use of the model and different image sizes, we separated the HW resources into two sets.

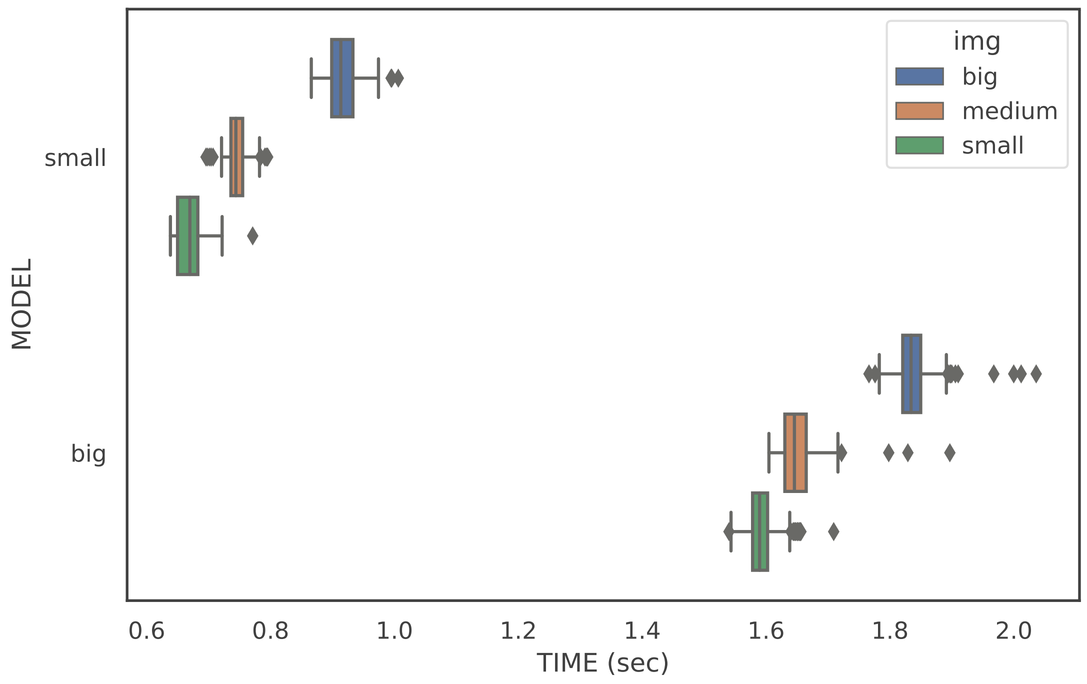
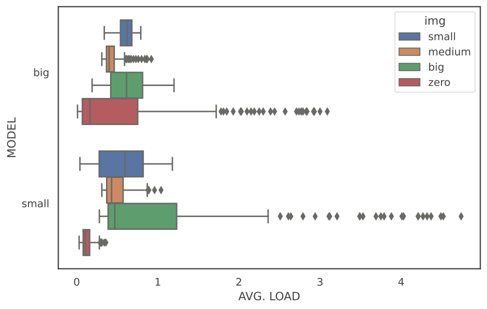


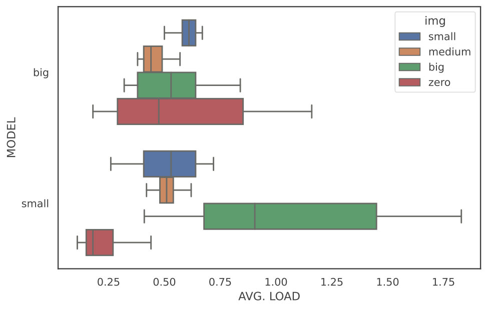
Figure 6.1: RTT from experiment on the client with different images and models

In the first set of HW resources visualized in Figures 6.2, we see that the execution of the request does not directly affect the HW data, because there is no relation between our baseline data, in red, and the other data. Instead, the baseline is sometimes even higher compared to the values collected with the models.

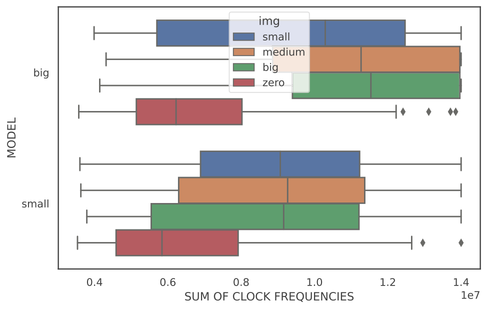
In the second set of plots, we can see that the evaluation of images on the models affect the HW resources. As we can see, the baseline in all the plots is smaller compared to the other measurements. For the plot presenting memory usage, the difference in usage compared to the baseline is enormous. Furthermore, we can see that the memory plots visualized in Figures 6.3a and 6.3c are very similar. This indicates that most of the memory is used for the model, which is trivial since we execute only the programs that we need to measure the HW resources to avoid any noise.



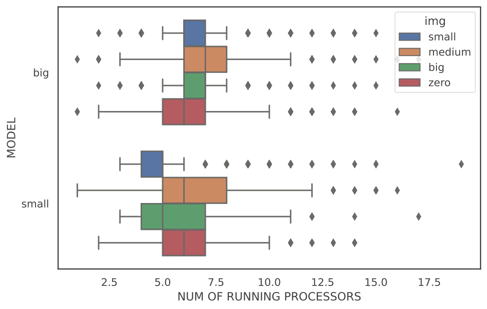
(a) Measurement of load1, which contains the average load of the last second



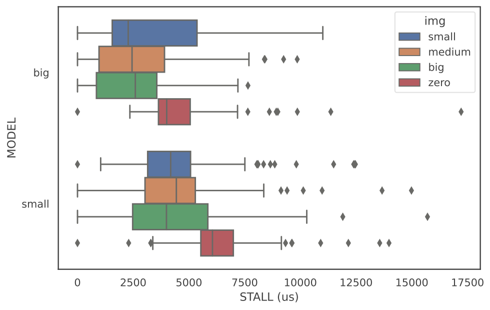
(b) Measurement of load5, which contains the average load from the last five second



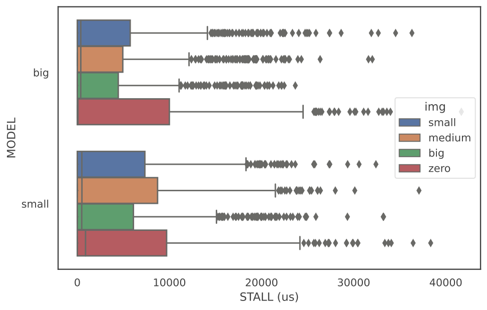
(c) Measurement of the frequency



(d) Measurement of the procs

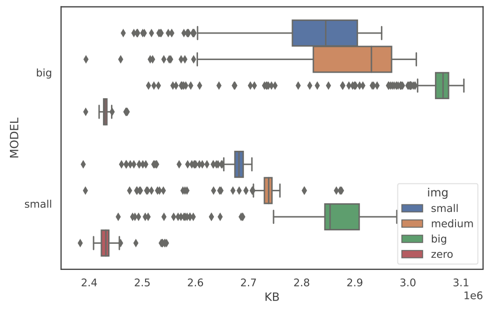


(e) Measurement of the CPU pressure

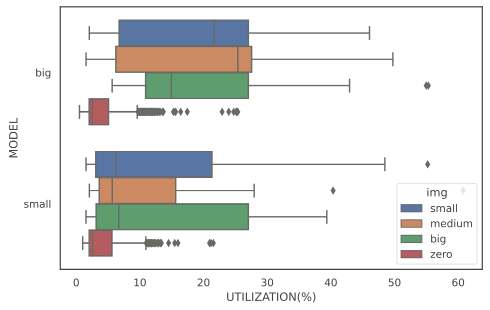


(f) Measurement of the input/output pressure

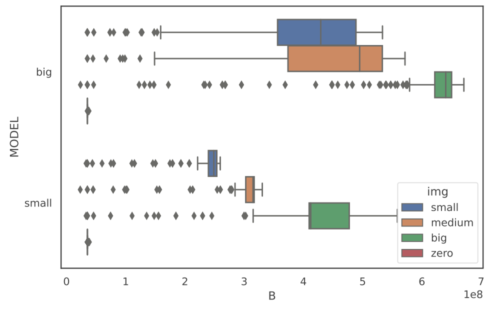
Figure 6.2: Measured HW resources using small or big model during the local experiments, with no correlation to the RTT of the experiments



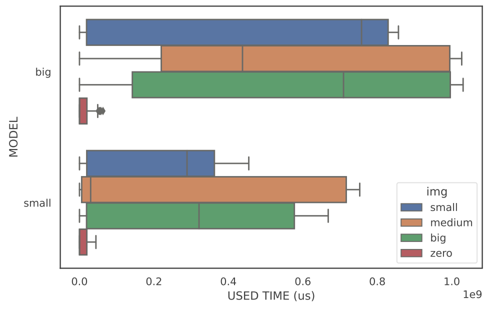
(a) Measurement of the RAM



(b) Measurement of the CPU



(c) Measurement of the RAM in the model container



(d) Measurement of the CPU in the model container

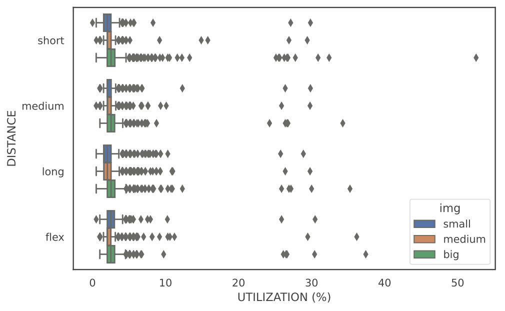
Figure 6.3: Measured HW resources using small or big model, during the local experiments, with correlation to the RTT of the experiments

External execution

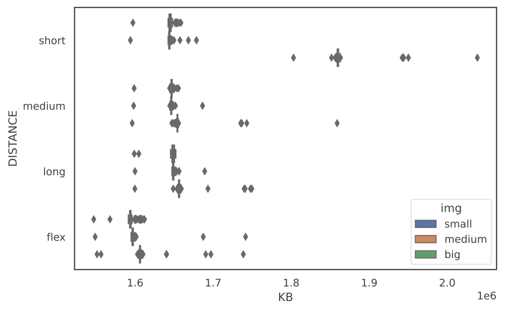
In this set of experiments we use the client only for sending the images and receiving the results. Therefore, the usage of HW on the client should be similar to the experiment in Section 6.1.1. Since the only difference is the additional connection to the server, which should be negligible. For the evaluation of the images we use our server which runs only the big model described in Table 4.3. The setup of our experiment is visualized in Figure 4.1.

Figure 6.4 visualizes the results. As we can see in Figure 6.4a, the CPU utilization is around 5% most of the time. However, there are some outliers, where the CPU utilization goes up to 50%. These outliers most likely are from background tasks, which are running automatically in parallel. In Figure 6.4b we can see the used RAM during the experiment. There we can see, that the RAM usage is between 1.6 GB and 2 GB, which is a usual range when only some tasks are running on the system.

In addition to the RAM and CPU usage, we also measure the signal power and the RTT. Figure 6.5 shows the signal power at different positions. These positions are visualized

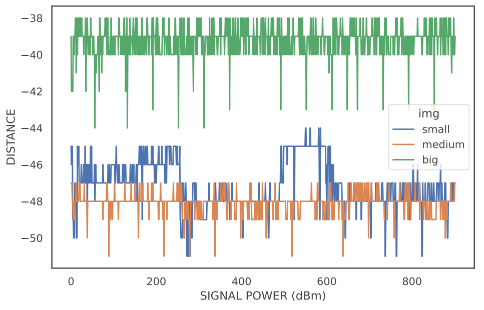


(a) Measurement of the CPU

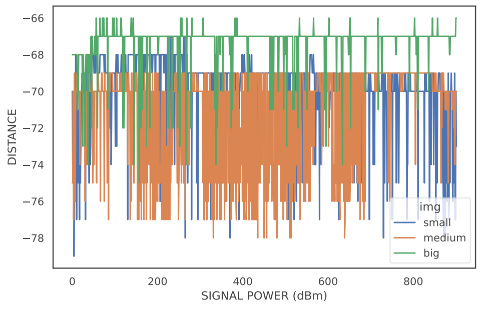


(b) Measurement of the RAM

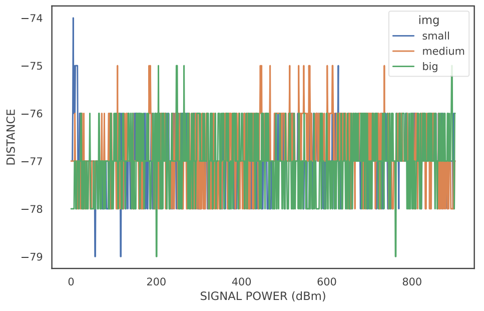
Figure 6.4: CPU and RAM usage during the external experiments for experiments with different images and distances to the server



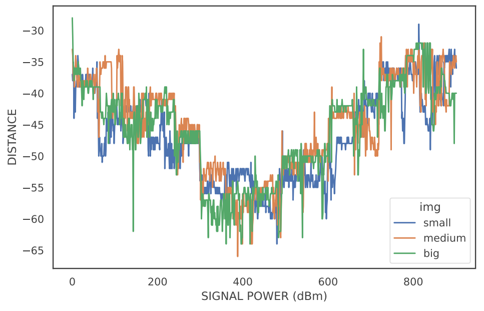
(a) Signal power for the short distance



(b) Signal power for the medium distance

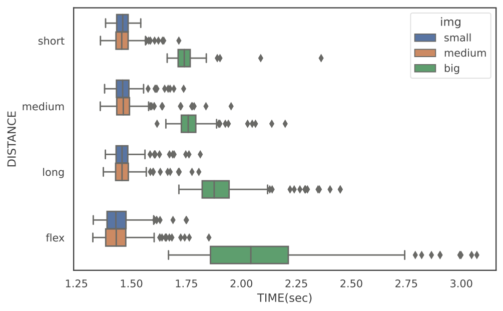


(c) Signal power for the large distance

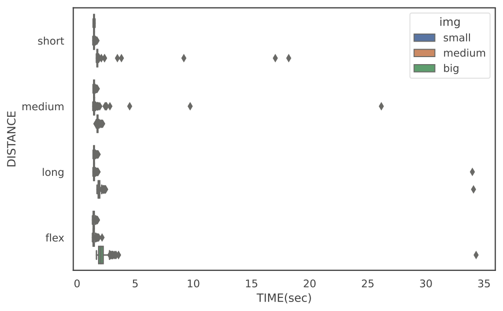


(d) Signal power for the dynamic distance

Figure 6.5: Signal powers that distinguish in the distance between the client and the server. In (a) to (c) the distance is static and in (d) dynamic.



(a) RTT with outliers



(b) RTT without outliers

Figure 6.6: RTT for different image sizes and distances to the server

in Figure 4.1. In Figure 6.5a, 6.5b, and 6.5c, we can see the signal power at the three static locations numbered one, two, and three, respectively. In Figure 6.5d we can see the signal power for number four, which is the experiment where the client is moving during the experiment. These figures show how the signal power changes depending on the client's location. In summary, the range of signal power we observed was from -40 dBm to -80 dBm. dBm stands for decibel milliwatt and is a logarithmic scale of values and is calculated by $10 \cdot log_{10} \cdot (\frac{P}{1mW})$. P describes the power of the remaining signal power. Increasing the distance or introducing obstacles between the sender and receiver decreases the power of the remaining signal power. Therefore, a value of -40 dBm is much better than -80 dBm. We observed a signal power of around -40 dBm to -50 dBm for short distances, around -65 dBm to -80 dBm for medium distances, and around -75 dBm to -80 dBm for large distances. Interestingly, in the fourth measurement, where the client moves during the measurement, the signal power does not fall below -65 dBm, although the distance is at least as far away as in the measurement with the large distance, as we can see in Figure 4.1. Therefore, we can conclude that the signal power decreases as the distance increases, but it is not possible to define how good or bad the signal power is depending solely on the distance. In our example, it could be the case that there are fewer obstacles in the measurement labeled with number four compared to the measurement labeled with number three.

In Figure 6.6, we can see the RTTs of our measurements. In Figure 6.6a shows the unfiltered result, where we can see that the RTT is not as predictable as in the previous experiment. Although we only use two devices connected via an AP, problems arising through a wireless connection. Even in the measurements with a short distance, there are requests that take more than 15 seconds. As we increase the distance, the time of the outliers also increases, as seen in the long-distance measurements, where requests take up to 35 seconds. However, these are only a few cases, so we will ignore them and concentrate on the remaining results. Figure 6.6b shows a filtered result, where we can observe two characteristics: increased RTT and increased spreading as the distance between the client and the AP increases.

Comparison of the RTTs between local and external execution

In the Figures 6.1 and 6.6b the RTTs for the local and external execution are visualized, respectively. We can see that the local execution with the small model is between 0.6 and 1 seconds and 1.6 to 2 seconds for the big model. For the external execution, we see that the small and medium images take between 1.25 and 1.5 seconds and around 1.75 seconds or more for the big image. Therefore, the external execution is up to 0.4 seconds faster if we compare it with the big model from the local execution. However, it is at least 0.6 seconds slower than the execution of the small model. As a result, we can see that we could only use the server for the execution if we need the evaluation of the big model with its deeper processing and we give the process enough time for it. In the other case, local execution is the better option.

6.1.2 Evaluation System

In our second series of experiments, we change to a new system visualized in Figure 4.3. As we can see, this system not only includes a client and a server, like our previous system, it also includes three APs. Information about the new components is presented in Table 4.4. In addition to the new HW components, the request handling is also changed. In Figure 4.4 we can see the new sophisticated approach, which is explained in Section 4.2.2.

In this section, we perform the following experiments.

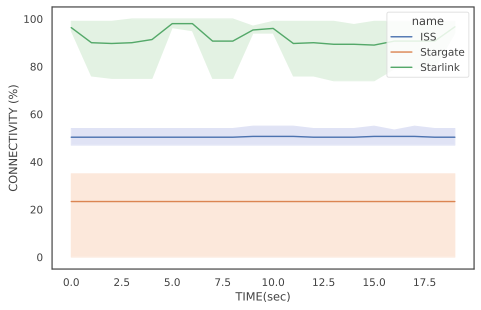
- Variance with different AP and Locations
  - Changes in the signal power on a fixed position and while moving
  - Changes depending on the used access point
  - Conclusion
- Boundary Test (experiment at the end of the signal power range to increase latency)
- Stress Test (introducing of additional load on HW with *edge-chaos*)
  - On the client (used for training ML later)
  - On the server (reference value, also used for training ML later)
  - Conclusion

Variance with different AP and Locations

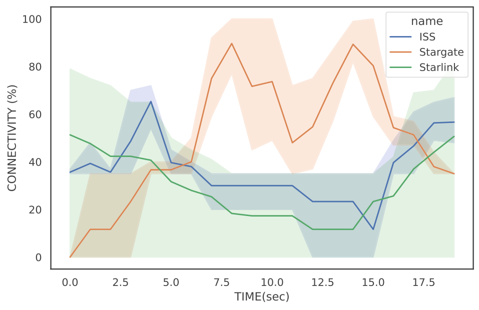
1. Changes in the signal power on a fixed position and while moving

In our first experiment in this series, we start by evaluating how stable the signal power is in general. For that we measure the signal power of different APs. First, at a fixed location and then while the client is moving around. For that we use a command-line tool called *nmcli* which measures the signal strength in the surrounding, from a good connection of 100 down to 0 for a bad connection. The setup of the experiment is visualized in Figure 4.3. The three APs *starlink*, *iss*, and *stargate* are visualized as router icons. In the first part, the client will stay at the place marked with an X (Stationary), and in the second part it will move according to the green arrows (Journey 1), starting at the green circle. In Figure 6.7a, we can observe the strength of the connection to each AP at that specific location. The connection to AP *starlink* is strong, while the connection to AP *iss* is slightly worse, and the connection to AP *stargate* is terrible.

In Figure 6.7b we can see the result while the client is moving according to Journey 1. We can see that the range of the signal power varies greatly. The AP *starlink* has the largest range, which could be due to placement in a separate room and the effect of a closed door, which was the case in some experiments, leading to such a variation in signal power. Especially the lower boundary with zero value almost through the whole experiment is recognizable. This is because it was not visible to our client device. A reason for a zero value can be that the device is not in range or, in general, offline. However, the other two APs, *stargate* and *iss*, also show a variance. One reason for this is the exact positions where the signal powers were collected. These points can vary a bit, leading to different results. However, there is also a difference at the beginning and end, which were the same in all experiments. Therefore, there is a variance depending on the time at which the signal power is measured. This means that a position that provides a good WiFi connection at a certain time may not necessarily deliver it at another point in time. As a result, it is important that we are aware of the current WiFi connection and that we do not rely on old data.



(a) Collected data at fixed point (Stationary) of the client



(b) Collected data while moving according to Journey 1

Figure 6.7: WiFi connections of the APs *starlink*, *stargate* and *iss*

2. Changes depending on the used access point

In this experiment, we start to run requests to the server. Since we are interested which affect the different signal powers have on the RTT we execute the requests on the fixed position from our previous experiment. As mentioned above, we have the APs *starlink*, *iss*, and *stargate* with good, medium and bad connection presented in Figure 6.7a.

Although there is no problem connecting to the APs *starlink* and *iss* we have an issue connecting to *stargate*. The reason for this was the too weak connection to this AP, which means that not all APs that can be found with *nmcli* have a stable connection to connect to. Due to this, we only execute the experiment on the APs *starlink* and *iss*.

The results of our experiment are presented in Figures 6.9 and 6.8. In these figures, we use *REQUEST_ID(cnt)*, which describes the current request counter. Therefore, we sent 500 requests to the APs *starlink* and *iss*. As we can see in the results presented in Figure 6.9 we are interested in four different values. These values are signal power, ping, latency, and RTT. This time only the signal power with the connected AP is measured. For that, *telemd* is used, which provides this information as well, but measures the connection differently. Here, the signal power is measured in decibel-milliwatt (dBm) with -40 dBm as a strong connection and -80 dBm as a bad connection. To obtain general information about the workload on the network, we send a ping to google (*https://www.google.at*). Latency and RTT are the results of the request to the server, where latency describes the time required to send the request and receive the result. The RTT also contains the time it takes to process the request.

In our experiments, the connection to the AP *starlink* is between -48 and -58 dBm, which is strong. The connection to AP *iss* is weaker, ranging between -62 and -78 dBm. Since our client is much closer to the AP *starlink*, as we can see in Figure 4.3 it is clear that this connection is stronger. The ping in both experiments is generally between 20 and 30, however, there are two spikes in the experiment where we are connected to the AP *starlink*. In this experiment, the ping increases to 80 and 120 at a certain point in time, as we can see in Figure 6.8. The first spike, which is 120, is also visible in the latency. We can see that the latency increases to half a second instead of the usual 0.1 and 0.2 seconds. However, the spike with 80 does not result in additional latency. What we also see is that the spikes in the latency, with 0.5 and 0.6 seconds, result in an increased RTT. Therefore, we see that a strong signal power is useful for having low ping and low latency, which leads to low RTT. In Figure 6.9a we see the results of the experiment in which we are connected to AP *iss*. In this experiment, the signal power is weaker compared to the AP *starlink*. However, the ping is constant between 20 and 30 and the latency is around 0.05 and 0.4 seconds. This is higher compared to the experiment with AP *starlink*, but also not very high. However, the RTT for large images ranges from 1.4 to 3 seconds, which is a factor of 2. Another interesting observation is that the distance between the APs *starlink* and *iss* is at best as the distance between the client and the AP *iss* as we can see in Figure 4.3. However, as we can see from the results, it is faster to send the request first to AP *starlink*, which forwards the request to AP *iss*. The reason for this is probably that the AP *starlink* has a better hardware and larger antennas, resulting in a better connection to the AP *iss*, compared to the connection from the client to the AP *iss*.

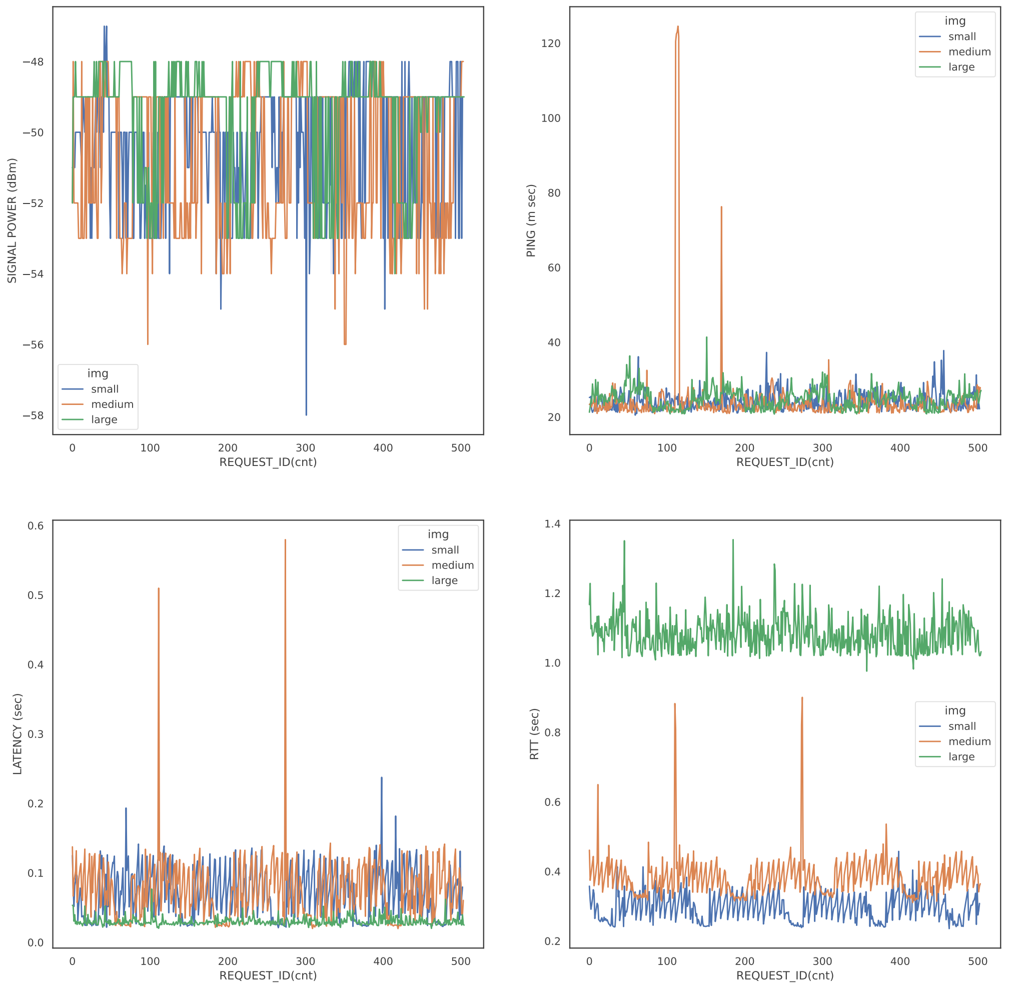
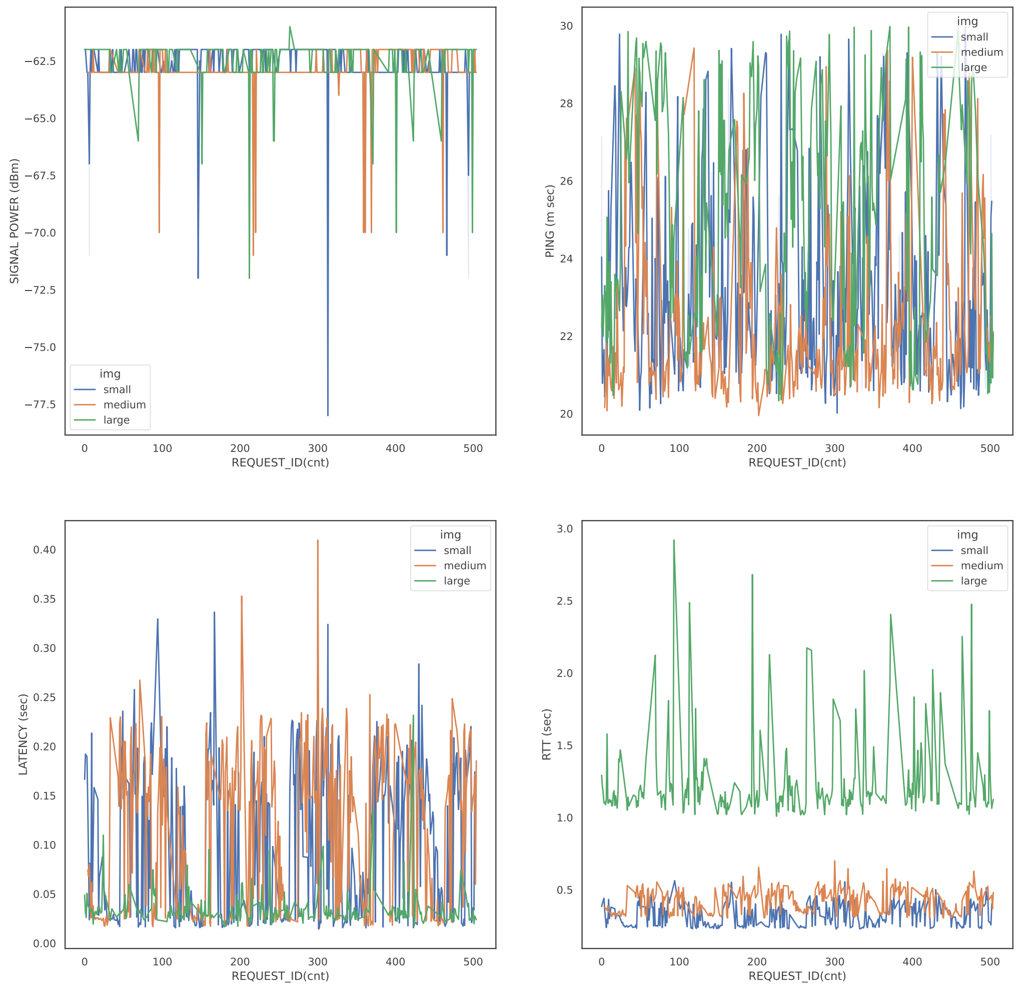


Figure 6.8: Collected data with Starlink AP



(a) Collected data with ISS AP

Figure 6.9: Signal power, ping, latency and RTT experiment with stationary location

3. Conclusion

With these two experiments, we evaluated which access point (AP) we use for our remaining research. From the first set of experiments, visualized in Figure 6.7, we could already see that the AP *stargate* has by far the worst connection. When we tried to connect to *stargate* in our second set of experiments, we realized that the connection is not good enough. Between APs *iss* and *starlink* we analyze the results visualized in Figures 6.8 and 6.9a. We came to the conclusion that it is faster to send requests to AP *starlink*. For that reason, the remaining experiments are performed with AP *starlink*.

Boundary Test

In the previous experiment, we could see that the signal power sometimes drops below -70 dBm. However, the ping remained more or less the same, around 20 to 30, as we can see in Figure 6.9a. For that reason, we analyze the behavior of the RTT when the ping increases radically by moving the client to the edge of the signal power range of the connected AP.

The results of the experiment, depicted in Figure 6.10, reveal a significant decrease in signal power, reaching -90 dBm. This reduction in signal strength correlates with a drastic increase in ping latency. Specifically, the ping spikes to 2500, as illustrated in the top-right plot. Furthermore, there is a recognizable correlation between ping and latency, notably visible when the *REQUEST_ID* approaches 200, with ping exceeding 2000. Consequently, this elevated ping duration leads to a latency exceeding 14 seconds, resulting in an RTT reaching 15 seconds. Thus, this experiment underscores the impact of ping on latency.

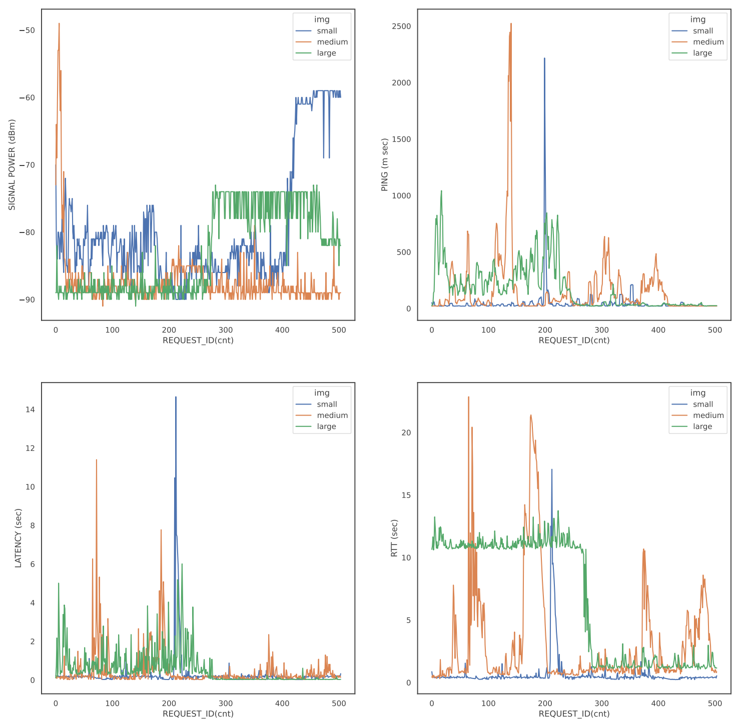


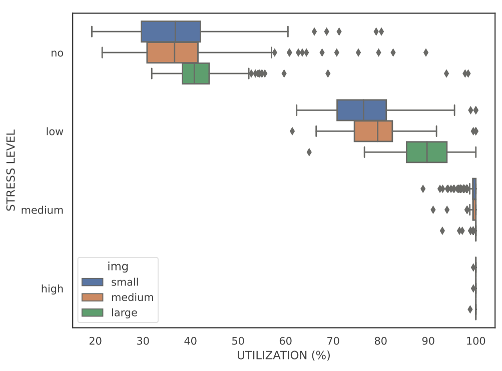
Figure 6.10: Measurement of the ping at the edge of AP connection

Stress Test

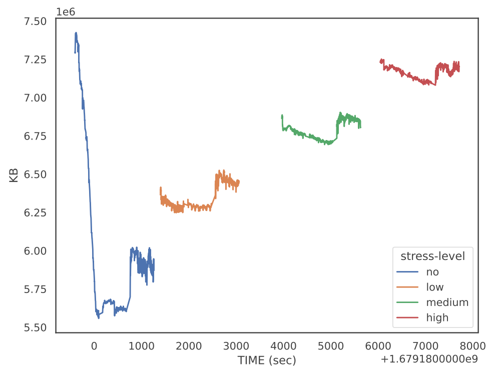
1. Execution on the Client

In this experiment, we change the focus from the APs to the client. We are interested in how fast the client can process requests, depending on the current load on the HW. For that we increase the load of the HW by simulating additional programs running on the client. For that we are using a tool called *stress-ng*. We differentiate between four different stress levels, numbered 0 to 3, which represents the multiplication factor of a predefined additional load listed in Table 6.3.

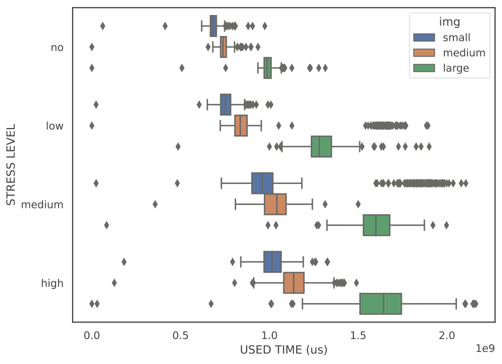
With these parameters, CPU utilization, RAM usage, and write operations are increased, which simulates another program in our experiment. During the experiment, the client moves according to Journey 2, described in Section 4.2.2 and visualized in Figure 4.3. Moving, although all experiments are executed locally, is not necessary, but it provides more variety in the result, which we need later for training our ML model. The results are visualized in Figure 6.11.



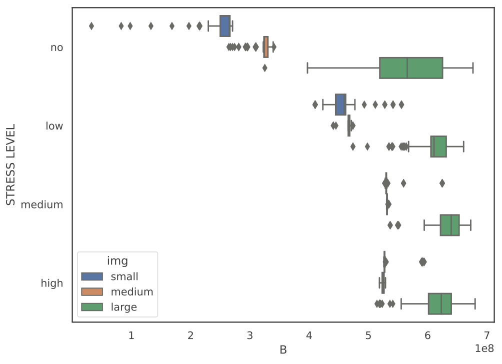
(a) overall CPU usage



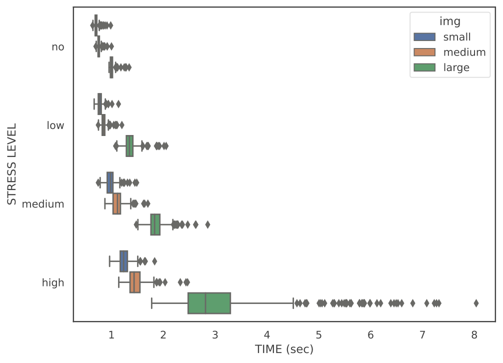
(b) overall RAM usage



(c) container CPU usage



(d) container RAM usage



(e) RTT

Figure 6.11: Collected CPU and RAM usage and resulting RTT

Stress-Level	Malloc	Malloc-Bytes	IO
No	0	0	0
Low	1	70000	1
Medium	2	140000	2
High	3	210000	3

Table 6.3: Parameter values of *stress-ng* for different stress levels

In Figure 6.11a, we see how *stress-ng* increases the cpu utilization depending on the current level. We also see that utilization is already very high with the *medium* stress level. Increasing the level again to *high* leads to CPU utilization of 100%. In Figure 6.11b, we see that the RAM usage also increases per level. We also see that RAM usage starts very high with around 7.5 GB, but drops to around 5.5 GB in the experiment with *no* stress level. The reason is that we start a lot of programs before starting the experiments because we need them to measure, store, and execute the experiments. These programs are the reason for the high RAM usage at the beginning. We also see that the start of each experiment and the use of large images increase RAM usage. A higher stress level not only increases the overall CPU and RAM usage, as we see in Figure 6.11a and 6.11b, but also increases the CPU and RAM usage of the model docker container, where requests are executed, which we can see in Figures 6.11c and 6.11d. Finally, we can also see that the different stress levels have an impact on the RTT of the request, visualized in Figure 6.11e.

In Table 6.4 we summarize the median and std. RTTs of the experiments. We decided to use the median instead of the mean value to prevent the effect of outliers. As already mentioned, we can see that the RTT is increased with a higher load on the client. The factor is around two from no to high load for small and medium sized images and around three for large sized images. We can also see that the std. is almost always negligible except for the large image with a high load on the client. The reason for this is that with the highest stress level all resources are in use and the increased effort for processing large sized images leads to this high std. value.

In summary, we can say that a general increase in HW resources utilization also increases the time usage the ML model needs to process the requests. This experiment delivers perfect data for our baseline, where we are interested in how fast the client can process a request depending on the current state.

Stress-Level	▭		◧		▬	
	$\tilde{x}$	σ	$\tilde{x}$	σ	$\tilde{x}$	σ
No	0.6988	0.1917	0.7524	0.1769	0.9955	0.1721
Low	0.7698	0.0478	0.8430	0.0491	1.3454	0.1161
Medium	0.9808	0.0862	1.1153	0.1077	1.8314	0.1587
High	1.2327	0.1150	1.4351	0.1576	2.8132	1.0140

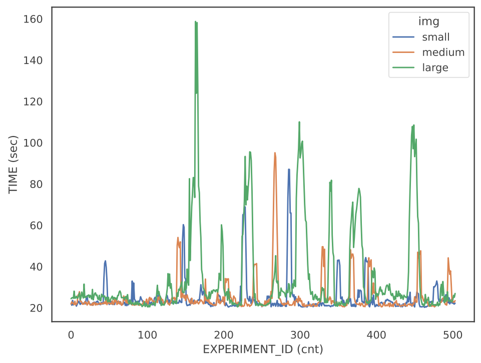
Table 6.4: Baseline of RTTs from local execution with different stress-levels. $\tilde{x}$ for median value and σ for std. value of RTTs measured in seconds. ▭, ◧, and ▬ for small to large images

2. Execution on the Server

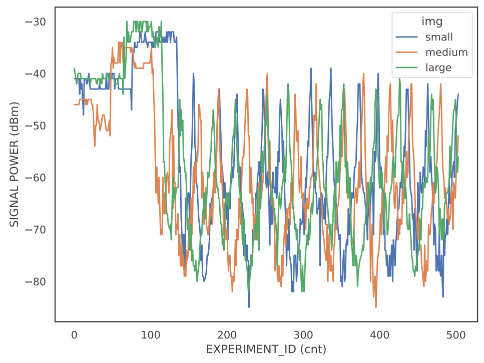
We repeat the same experiment, but this time we send everything to the server. We are connected to *starlink* and follow journey 2. The results we get from this experiment are compared to the previous ones when all requests are executed on the client. Given that the results on the client side vary depending on the stress levels, we will compare each stress level against one another. From the comparison, we can then see when it is better to execute the request on the client or the server in general.

The results are presented in Figure 6.12 and will be used as a reference value for future experiments. In the first two plots 6.12a and 6.12b, we can see the resulting ping and signal power, respectively. These values depend on the location of the client, but not on the current load. Therefore, the results of ping and signal power are very similar through all stress levels. Since we offload all requests, the RTTs are also very similar, independent of the current load on the client, which we can see in Figures 6.12c and 6.12e for no and medium extra loads, respectively. However, since the RTTs are compared with different reference values, we can see that the points increase with a higher load, as visualized in Figures 6.12d and 6.12f. The reason is that the server has more time to process a request when the load is higher, since the client would also need more time to process it.

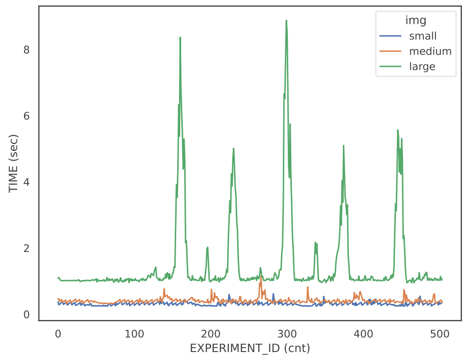
The RTTs and points 🪙 of the experiment are summarized in Table 6.5. As already mentioned, the RTT is independent of the stress level, which is why the values for the mean and std. for the same images are similar. We can see that the images take around 0.3, 0.4, and 1.05 seconds for the image sizes small, medium, and large, respectively. We can also observe that the std. for small and medium sized images is negligible but relevant for large sized image with a std. above the mean value. This means that the variation is very broad for this image size.



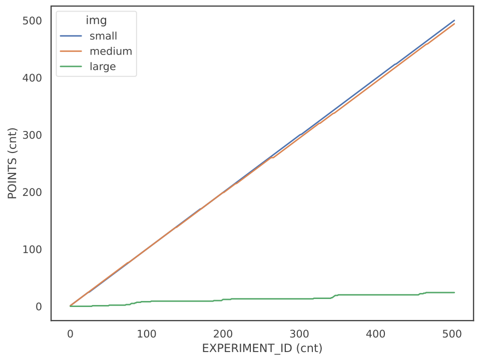
(a) Ping



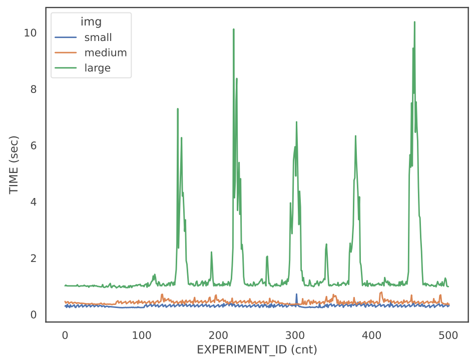
(b) Signal power



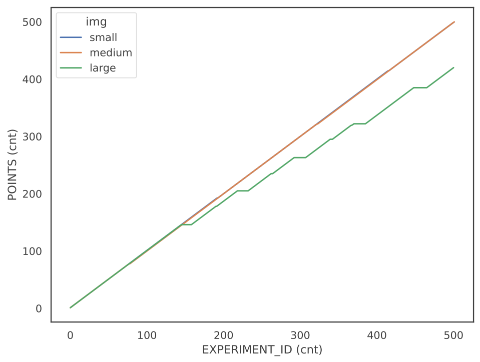
(c) RTT, no extra load



(d) Points, no extra load



(e) RTT, medium extra load



(f) Points, medium extra load

Figure 6.12: Baseline, results collected by offloading to server with no and medium extra load

Stress-Level	🔋			🔋			🔋		
	$\bar{x}$	σ	🪙	$\bar{x}$	σ	🪙	$\bar{x}$	σ	🪙
No	0.3098	0.0499	500	0.3934	0.0832	494	1.0561	1.2344	24
Low	0.3180	0.0606	499	0.3833	0.0615	499	1.0632	1.3007	408
Medium	0.3013	0.0449	500	0.4253	0.0679	500	1.0633	1.4582	420
High	0.3050	0.2536	496	0.3907	0.1740	497	1.0545	1.2701	449

Table 6.5: Offloading requests and comparison of RTTs with local execution. $\bar{x}$ for mean value, σ for std. value from RTT and measured in seconds. 🪙 points compared to local execution for different stress-levels. 🔋, 🔋, and 🔋 for small to large images

Conclusion From the results in this section we can summarize the following. First, we can see in the Tables 6.4 and 6.5 that the RTT for small- and medium-sized images is much better when they are executed on the server. This even did not change when the signal power dropped to around -80dBm. We can also see that the difference between the client and the server is closer compared for the large-sized image. Although it is clearly faster on the client when no extra load is introduced, the execution on the client benefits from the low std. values for the first three stress levels. In addition, it is advisable to process large images with the highest stress level on the server.

In the following sections, we will analyze the behavior when image processing is distributed between the client and server in a single experiment.

6.2 Simple Decision Offloading

In this approach, we go a step further and want to change between the devices that execute the requests. We are interested in whether a single decision is sufficient to improve the RTT of the requests. For that, we change *Wi-Fi App Client*, which is part of the offload handler visualized in Figure 4.4. Depending on the decision of the *Wi-Fi App Client*, the request is executed locally or remotely. We evaluate the decisions toggling, signal power, and ping by moving the client according to Journey 2 while connected to AP *starlink* visualized in Figure 4.3.

To evaluate the performance boost, we compare these decisions by the median RTT of the results from Section 6.1.2, where we offload everything to the server. First, we evaluate how many of our 500 requests perform at least as well as the reference value and give points for these requests. At the end of the 500 requests, we can see how well the decisions perform by the score of the points of the different decisions.

6.2.1 Toggling

First, we introduce a toggle bit, which we change after each execution. The results are presented in Table 6.6. We can see that the RTTs for small and medium images increase but stay the same for large images, compared to the results of the offloading approach in

Table 6.5, therefore this decision does not provide a performance boost. We expect this result since half of the requests are executed locally, which takes more time for small- and medium-size images and similarly for large-size images, as we can see in the Tables 6.4 and 6.5. Therefore, our system is capable of switching between models, allowing us to evaluate more advanced approaches.

		▭			◧			▬	
	$\bar{x}$	🖼 $\Uparrow$	🪙 ($\Uparrow$)	$\bar{x}$	🖼 $\Uparrow$	🪙 ($\Uparrow$)	$\bar{x}$	🖼 $\Uparrow$	🪙($\Uparrow$)
Static	0.6493	250	365(249)	0.7026	250	365(250)	1.0303	250	69(20)
Dynamic	0.6429	250	376(249)	0.7107	250	314(248)	1.0641	250	15(10)

Table 6.6: Results of changing between and client and server after each request. ▭, ◧, and ▬ for small to large images, $\bar{x}$ for the mean value given in seconds, 🖼 $\Uparrow$ for the number of requests we upload, 🪙 points compared to the reference value, and ($\Uparrow$) points collected by offloading to server.

6.2.2 Signal power

In this experiment, we offload requests based on the current signal power. We run the tests with four different boundaries: -40 dBm to -70 dBm in steps of -10 dBm. We choose this range because our signal power is most of the time in this area, as we can see in Figure 6.12b. The boundary defines until which signal power requests are offloaded to the server, e.g., -40 dBm means that the current request is offloaded until -40 dBm, but executed locally by -41 dBm. The results are presented in Table 6.7. There we can see the number of images sent to the server in the column 🖼 $\Uparrow$ for the three different image sizes: small ▭, medium ◧, and large ▬. We can see that more requests are executed on the server as the range increases. Only around $\frac{1}{5}$ requests are offloaded compared to around $\frac{4}{5}$ requests with a boundary of -40 dBm and -70 dBm, respectively. Therefore, the score for small and medium images increases and decreases for large images with increasing signal power range.

Signal Power		▭			◧			▬	
	$\bar{x}$	🖼 $\Uparrow$	🪙 ($\Uparrow$)	$\bar{x}$	🖼 $\Uparrow$	🪙 ($\Uparrow$)	$\bar{x}$	🖼 $\Uparrow$	🪙($\Uparrow$)
-40 dBm	0.6905	101	316(101)	0.7445	104	328(104)	1.0097	125	169(7)
-50 dBm	0.6878	110	320(109)	0.7546	116	233(116)	1.0213	125	121(9)
-60 dBm	0.6705	232	367(229)	0.7322	207	329(206)	1.0315	200	88(9)
-70 dBm	0.3396	395	437(391)	0.4128	392	435(392)	1.0465	405	59(20)

Table 6.7: Comparison of RTTs with the reference values for different signal powers as offload decision. ▭, ◧, and ▬ for small to large images, $\bar{x}$ for the mean value given in seconds, 🖼 $\Uparrow$ for the amount of requests we upload, 🪙 points compared to the reference value and ($\Uparrow$) points collected by offloading to server. The signal power is measured in decibel-milliwat (dBm)

6.2.3 Ping

In this experiment, we offload requests based on current ping. Since we expect pings around 30 and above, we used 30 and 50 as boundaries. A boundary of 30 means that requests are offloaded if the ping is 30 or below, therefore requests are executed locally with a ping of 31. The results are presented in Table 6.8. We can see that with a ping of 30, most of the requests are executed remotely. The number increases when setting the ping limit to 50. Due to this fact, the RTT is very similar to our reference values.

Ping		🔋			🔋			🔋	
	$\bar{x}$	🖼 ⇑	🪙 (⇑)	$\bar{x}$	🖼 ⇑	🪙 (⇑)	$\bar{x}$	🖼 ⇑	🪙(⇑)
30	0.3305	433	460(425)	0.4002	463	468(460)	1.0632	390	39(7)
50	0.3094	488	491(488)	0.4065	475	472(465)	1.0709	467	20(16)

Table 6.8: Comparison of RTTs with the reference values for different pings as offload decision. 🔋, 🔋, and 🔋 for small to large images, $\bar{x}$ for the mean value given in seconds, 🖼 ⇑ for the amount of requests we upload, 🪙 points compared to the reference value, and (⇑) points collected by offloading to server.

Conclusion

In all experiments in Section 6.2, we can see that it is advisable to offload small and medium-sized images to the server and execute large-sized images locally when there is no load on the client. Since all decisions we make in this section are independent of the current hardware utilization on the client and of the size of the image, we cannot expect a good score from these decisions. In summary, we can say that the image size is crucial to the decision. After that, it is also relevant whether the client has no load or additional load. However, how much extra load does not play an important role, since we already get more than 1400 out of 1500 points with a low extra load, as we can see in Table 6.5.

6.3 Decision Tree Offloading

In the previous section we can see that it is possible to offload depending on the current state of the environment. In this section, we go a step further and create our own DT, which is visualized in Figure 6.13, for the offloading decision.

It contains four conditions which include the information of some hardware resources and the current state of the environment. The reference values in this DT are the values collected from the training set. Since the training set is normalized, all values are between zero and one. Except for the signal power, a lower value is better compared to a higher value. The first condition ensures that the current state of the environment is not catastrophic. We can see that we decide to execute the request locally if *ping > 0.7* or *signal_power < 0.25*. Therefore, we decide to execute the request locally independently on our current state of the hardware, because we assume that the network is too unstable, and therefore it is better to rely on our own device. In the next condition, we evaluate the image size. Since we know that offloading requests are generally faster for small- and medium-sized images, we offload in these cases. In the third condition, we check again the current state of the network. The idea is that large images should be offloaded if the network allows it. In the last condition, we evaluate the current state of the hardware by using cpu, ram, and proc. Is the hardware almost on the max capacity, we also offload. For the remaining requests, we have a questionable network and hardware that is not overloaded yet. Here, we decided to execute the request locally.

We run experiments with the DT following Journey 2 visualized in Figure 4.3 on the right side. The results are presented and compared with different TPOT-configurations in the next section.

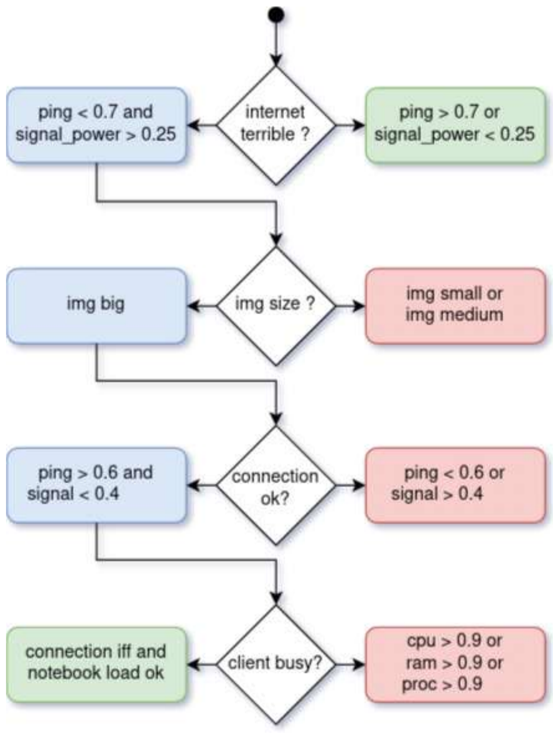

internet terrible ?
ping < 0.7 and signal_power > 0.25
ping > 0.7 or signal_power < 0.25
img size ?
img big
img small or img medium
connection ok?
ping > 0.6 and signal < 0.4
ping < 0.6 or signal > 0.4
client busy?
connection iff and notebook load ok
cpu > 0.9 or ram > 0.9 or proc > 0.9

Figure 6.13: Simple DT for offloading decision

6.4 ML-Based Offloading

In this section, we train and compare different ML models, generated by TPOT, and compare them with each other and with an additional solution. TPOT provides six different default configurations, as mentioned in Section 5.4. From the six available configurations, which are called: *Default TPOT*, *TPOT light*, *TPOT MDR*, *TPOT sparse*, *TPOT NN*, and *TPOT cuML* we will use all models except *TPOT cuML* which is exclusively for GPU, which is not supported on our client. Therefore, it was not possible to generate a model with this configuration. The remaining five configurations are compared in Tables 6.9, 6.10, 6.11 with an additional DT model. The DT model contains some decisions about the environment and some HW resources and is visualized in Figure 6.13.

In Table 6.9, we compare the size of the model and the accuracy of the model according to the training set. These values are already available without addition experiments. We can see that the sizes of the models range from 8KB up to almost 5MB, with MDR as the lowest and NN as the model with the largest model size. Additionally, we see the avg. success rate of the models, which is approximately the same for all models and is between 95% and 97%. Since the DT was generated without any training, we cannot provide any CV. For the size, we estimate it near *TPOT-MDR* since it only takes some conditions that are included in our program.

In Table 6.10 we compare the data collected in our experiment. We evaluate differences in the decision duration, local execution and points for twelve different experiments for each TPOT configuration and our DT. These are for the four stress levels and three images for each stress level. The duration of the decision describes the time that the ML model takes to make a decision on which device the request should be executed. The time for the TPOT configurations variate from 8 to 56 msec for the lowest stress level, up to 47 to 315 for the highest stress level. Compared to that the DT almost need no time at all for all stress levels. For the decision, the size of the image had no impact and is therefore combined to a single value. For the remaining two categories *local execution* and points we can see the icons , , and . These icons represent small, medium, and large images, respectively. The local execution shows how many from the 100 requests are executed locally. We can see that TPOT configurations almost never execute it locally for the small and medium size images. For large images, the offloading decision strongly depends on the current stress level. We can see that for lower stress levels, requests are more likely to be executed locally and offloaded for higher stress levels. We can also see that the TPOT configurations *light* and *mdr* offload everything except for a small amount of requests through all stress levels and image sizes. The remaining three TPOT configurations decided to execute the request locally with a high probability for the lowest stress level and a lower probability for the higher stress levels. Additionally, we can see for the execution decision, that the *sparse* TPOT configuration executes the majority of request for the large images independently of the current stress level. For our DT we can see that it executes it locally approximately the same time over all stress levels and image sizes, which means that the conditions we provided are not sufficient to evaluate the current state of the device and environment. In the remaining columns, we compare the points. The points describe whether the request was faster than the average time it takes to execute it locally. We use this value for training our model, discussed in Section 4.5.2, with a supervised learning approach described in Section 2.4.5. We can see that for the TPOT configurations at small and medium size images the points are almost 100 everywhere, which is the maximum amount of points. Since the decision of all models is similar for the small and medium size image, the resulting points are similar. We can see that point lost are there where the model decided to execute it locally. For large images, the points hardly depend on the stress level. For the lowest stress level from 2 to 17 points and for the highest stress level from 58 to 97 points. Since our DT executes it similar for all experiments we see point losses for small and medium images. For the large images the result is comparable with the other models except at the high stress-level where it has less points compared the majority of the other TPOT-configurations.

	TPOT				
	Default	Light	MDR	NN	Sparse
Size (MB)	2.885	0.443	0.008	4.668	2.177
Average CV on training set (%)	96.49	96.22	95.34	96.62	96.45

Table 6.9: TPOT models characteristics

In Table 6.11 we compare the RTT with our reference value. For each experiment, we can see the mean $\bar{x}$ and std. σ value. First, we will take a closer look at the TPOT configuration. We can see that the RTTs for the small experiments are around 0.3 and 0.4 seconds. For the medium image, it is a bit higher, around 0.4 to 0.5 seconds. We can see that for these experiments, the stress level plays a minor role since all values are approximately the same. This fact change for large images. First, there is a larger variance, in general, with 1 to up to 3 seconds. Furthermore, we can see that the results depend on the TPOT configuration and the stress level. The two configurations *TPOT-Default* and *TPOT-Light* almost offload all requests, which we can see in Table 6.10, however, their outcomes vary due to the current connection to the AP, despite the identical experimental setup in both cases. Since *TPOT-Sparse* offloads the majority of requests, we can see that the RTTs go up to around 3 seconds for the highest stress level. The remaining two configurations *TPOT-Default* and *TPOT-NN* are both providing great results and are always below 2 seconds. Compared to that, the RTT for our DT are worse. Since it offloaded less for the small and medium large images, the RTT is longer compared to the TPOT-configurations with more than 0.6s for the images small and medium and a stress level of medium and higher. Also, for the big image the RTT is worse compared to the TPOT-configuration, with an exception with no extra load.

	Stress-Level	Decision Duration (msec)	LOCAL EXEC.			🪙		
			🔋	🔋	🔋	🔋	🔋	🔋
Default	No	56.4	0	5	64	100	92	17
	Low	186.7	0	0	11	100	100	71
	Medium	276.4	0	0	16	100	100	73
	High	314.9	0	0	13	100	100	90
Light	No	19.2	0	0	2	100	99	2
	Low	90.5	0	0	1	100	99	70
	Medium	123.3	0	1	1	100	99	82
	High	162.9	0	0	1	100	100	97
MDR	No	8.1	1	0	1	99	100	3
	Low	9.6	0	1	0	100	97	68
	Medium	27.8	0	0	2	100	100	83
	High	47.3	0	0	1	100	100	86
NN	No	29.2	0	0	80	97	99	7
	Low	35.0	0	0	28	99	98	71
	Medium	51.5	0	0	20	100	100	73
	High	66.4	0	0	7	100	100	90
Sparse	No	44.8	0	0	99	100	96	10
	Low	70.0	0	0	92	99	100	23
	Medium	96.0	0	0	66	100	99	53
	High	117.8	0	0	70	100	100	58
DT	No	0.004	30	28	33	92	81	10
	Low	0.006	35	28	31	71	76	70
	Medium	0.008	36	30	30	70	73	69
	High	0.022	26	29	23	79	76	77

Table 6.10: Decision duration describes the time the model needed to decide whether to execute it local or external. Local execution is the amount a ML decided to execute it locally. The points 🪙 describe how often it was faster than the reference values. 🔋, 🔋, 🔋 for small to large images.

TPOT-Config.	Stress-Level	▮▯▯		▮▮▯		▮▮▮	
		$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
Default	No	0.318	0.054	0.478	0.182	1.294	1.045
	Low	0.329	0.070	0.431	0.083	1.259	0.314
	Medium	0.35	0.072	0.454	0.133	1.49	0.797
	High	0.355	0.133	0.435	0.103	1.46	0.847
Light	No	0.326	0.055	0.412	0.083	1.945	1.527
	Low	0.322	0.058	0.425	0.087	1.361	0.58
	Medium	0.329	0.071	0.428	0.15	1.49	0.805
	High	0.323	0.051	0.435	0.109	1.347	0.468
MDR	No	0.333	0.110	0.408	0.058	1.651	1.136
	Low	0.326	0.060	0.446	0.169	2.367	2.36
	Medium	0.324	0.067	0.409	0.08	1.751	1.505
	High	0.32	0.083	0.411	0.092	1.656	1.299
NN	No	0.344	0.111	0.418	0.094	1.096	0.128
	Low	0.330	0.075	0.428	0.135	1.231	0.258
	Medium	0.325	0.062	0.414	0.083	1.63	1.089
	High	0.327	0.082	0.434	0.095	1.647	1.428
Sparse	No	0.327	0.059	0.447	0.232	1.119	0.176
	Low	0.321	0.069	0.417	0.066	1.491	0.353
	Medium	0.363	0.074	0.446	0.119	1.999	1.138
	High	0.319	0.053	0.412	0.064	3.198	1.895
DT	No	0.429	0.190	0.541	0.268	1.11	0.150
	Low	0.546	0.368	0.634	0.472	1.579	1.192
	Medium	0.674	0.570	0.735	0.561	2.556	2.327
	High	0.755	0.894	1.137	1.484	2.973	3.334

Table 6.11: RTT of the different TPOT configurations. $\bar{x}$ for mean and σ for std. value given in seconds. ▮▯▯, ▮▮▯, ▮▮▮ for small to large images.

Conclusion

In this section, we conducted a series of experiments with DT and ML models. We can see that the DT is worse compared to the ML models, probably because the DT is rather simple. However, the goal was to build the DT without any tools, since they were used anyway for the ML models. Of the five ML models, only two are capable of properly evaluating the current state. The reason is the goal of the different TPOT configurations. The *light configuration* with a very small model size is done by a restricted search, which in our case leads to a poor result. The *sparse configuration* is good for data sets with a majority of zeros, but this is not the case in our experiments, which leads to a bad result. However, *MDR configuration* should aim to maximize the accuracy of the model prediction, still leads to poor results in our experiments. One reason could be that it searches over Multifactor Dimensional Reductions and is specialized for genome-wide association studies, which is not the case here. This leads us to the two configurations. Both configurations search over a broad range of different ML settings with the goal of minimize the error of model prediction. The two configurations *default* and *NN* are the only two that were able to offload almost all requests for small- and medium-sized images and to increase the offloading decision for large-sized images steadily to almost up to 100% from no to high stress-level.

CHAPTER 7

Discussion

This chapter provides a summary of all relevant information needed to answer the research questions, split into three sections. These sections concentrate on hardware, network, the client, and offloading related topics, respectively.

7.1 Hardware

7.1.1 Selection of correlating HW metrics

In Section 6.1.1, experiments were carried out in an isolated setting to collect information about the utilized hardware. Initially, in Section 6.1.1, system information was collected using an open source tool called *telemd* [1] and presented in Table 6.2. The metrics provided by *telemd* includes system load and the application load, the system load being further classified into current load and accumulated load.

Current load metrics include CPU usage, frequency, number of processes, and RAM consumption. The introduction of ML models led to an increase in current load metrics, although the difference between the two models remained similar. Accumulated load metrics such as load1, load5, and psi exhibited variations in the three executions.

In addition, application load metrics depicted the CPU and memory usage of the three applications utilized in the experiment. In Section 6.1.1, image processing with the ML models was performed locally, with variations in the experiments arising from the model used and the size of individual images. The experimental setup is illustrated in Figure 4.1, with Figure 6.1 presenting the RTT and Figures 6.3 and 6.2 visualizing the hardware resource loads.

The latter figures are grouped based on their correlation with the use of the model, indicating an increase in load when using a model, as shown in Figure 6.3. However,

[1] https://github.com/edgerun/telemd

metrics with no discernible correlation with model usage were excluded in subsequent analyzes.

Subsequently, requests were sent from the client device to the server, with the setup shown in Figure 4.1. Twelve experiments were conducted, differing in distance to the access point (AP) and image size. In Figure 6.4, CPU and RAM loads are illustrated. Since requests were forwarded to the server, the load on the client machine remained consistent across experiments, with observed variations likely attributed to background tasks, highlighting a critical distinction between real-world experiments and mathematical simulations.

7.2 Network

7.2.1 Analyzing the Signal Power

Moreover, in comparison to previous experiments, we also assessed the signal strength to the access point. The results are illustrated in Figure 6.5. Specifically, in Figure 6.5b, there is a noticeable fluctuation of up to 10dBm in the small images. Furthermore, as shown in Figure 6.5a, there exists a variance of up to 12dBm between different experiments carried out at the same distance. This significant range in signal power motivated us to conduct further research.

To ensure the acquisition of reliable test data, we moved to a more realistic setup. This updated test bed, shown in Figure 4.3, contains a single client, three APs, and a server.

The subsequent experiment aimed at determining signal power is described in Section 6.1.2 and is represented graphically in Figure 6.7. In this case, we used a command line tool named *nmcli*, which allowed a comprehensive measurement of all APs within the vicinity. The experiment was carried out repeatedly, revealing recognizable variations in signal power quality in different trials. Consequently, we made the decision to consistently monitor the current signal power and not rely on some abstraction or completely ignore it.

7.2.2 Choosing the right AP

As previously mentioned, our updated setup comprises three APs, two of them acting as repeaters that forward data to the third AP. Given that the client can only connect to one AP at a time, we conducted performance tests to determine the AP that offers the optimal results. Although we were able to measure the signal strength of the three APs, as shown in Figure 6.7, connecting to the weakest AP, called *stargate*, proved unfeasible. Consequently, we restricted our comparison to APs *starlink* and *iss*. The outcome of this comparison is illustrated in Figure 6.9. In particular, AP *starlink* demonstrated superior performance in terms of RTT. This finding underscores that proximity or the fewest hops do not invariably translate to improved performance. Based on this analysis, we selected AP *starlink* for our subsequent experiments.

7.2.3 Network limitations

In a limited-scale experiment, we examined the constraints of our network by intentionally offloading requests with a notably poor connection to the AP. The results of this experiment are shown in Figure 6.10. In particular, we observed that a signal power of approximately -90dBm corresponded to ping values exceeding 2500 and RTT exceeding 10 seconds. Given these findings, we decided to include the ping metric into our offloading decision-making process. This decision was made because ping values do not consistently correlate with signal power and can provide additional insight into network quality. This experiment underscores the inadequacy of solely relying on connections to stronger devices, emphasizing the importance of considering multiple factors in the offloading strategy.

7.3 Client state

7.3.1 Impact of additional load on the performance

In another experiment, we investigated the limitations of the client device using a tool called *stress-ng*. This tool facilitated the introduction of an additional static load on the hardware. Tables 4.6 and 6.3 outline the parameters utilized and their respective values. The results of this experiment are presented in Figure 6.11. In particular, the introduction of additional load resulted in the CPU reaching a utilization of 100% and almost a full utilization of the available RAM, particularly at the highest stress level. Consequently, there was a noticeable degradation in RTTs, as depicted in Figure 6.11e. These findings underscore the importance of assessing the current state of the client device when making offloading decisions. It highlights the need to consider factors beyond network conditions alone for effective optimization strategies.

7.4 Offloading

7.4.1 Evaluating different offloading approaches

To demonstrate that offloading decisions cannot be solely based on network-related information, we performed several experiments detailed in Section 6.2. These experiments compared various approaches with complete offloading. Initially, we ran a toggle experiment to assess the system's capability to switch between the executive devices after each request. Subsequent experiments focused on the measurement of signal power and ping. The results of these experiments are presented in Tables 6.6, 6.7, and 6.8, respectively. The results indicate that relying solely on a single piece of information, such as signal power or ping, is not sufficient to accurately represent the current state. Consequently, such approaches lead to poor results compared to complete offloading. These findings emphasize the need to consider multiple factors beyond network-related metrics for effective offloading decisions.

7.4.2 Resource-Aware approaches to evaluate the current state

Throughout the experiments conducted in this thesis, the findings consistently emphasize the importance of considering both the current load of the client device and the quality of the network connection in the decision-making process of offloading. To facilitate the implementation of more sophisticated approaches, the collected data was compared, filtered, and preprocessed to improve their usefulness for training ML models. Section 4.5 provides a comprehensive description of the process used to obtain meaningful data for the training process. Furthermore, the resulting data were normalized to fall within the range of 0 and 1. Initially, we developed our own Decision Tree (DT), as illustrated in Figure 6.13. Despite its simplicity, the DT was unable to adequately evaluate the current state. However, it presented a lightweight approach that, in comparison to alternative methods, offered rapid processing capabilities.

At the end of our study, we made the decision to leverage ML models to automate the evaluation of the current state. To achieve this, we used a tool called *TPOT*, as outlined in Section 5.4. This tool offers default configurations that we thoroughly compared in Section 6.4, and the results are detailed in Tables 6.9, 6.10, and 6.11. For training these ML models, discussed in Section 4.5.2, we used a supervised learning approach mentioned in Section 2.4.5. In Table 6.9, it's evident that *TPOT-Light* and *TPOT-MDR* are the only TPOT configurations with a size below 1MB. Despite their compact sizes and relatively good RTTs, these models are not recommended because they offload almost all requests to the server, rendering them ineffective in adjusting to the current situation. Similarly, *TPOT-Sparse*, although smaller in size, also exhibits poor performance by sending too many requests to the client, thereby failing to adapt to the current conditions.

Conversely, *TPOT-Default* and *TPOT-NN*, while larger in size, emerge as robust models in Table 6.10. They demonstrate excellent adaptation to the current image size and workload on the client. The similarity in the results between these two models is attributed to the fact that *TPOT-NN* is an extension of *TPOT-Default*, as elaborated in Section 5.4. Compared to others, our DT is poor in competitiveness. Its advantage lies in its small size and the quick decision-making process, which outperforms the TPOT configurations in terms of speed. However, as shown in Table 6.10, the decisions made by the DT are generic and do not accurately represent the current state of the client and the environment.

Although it may be possible to develop a superior DT that provides better results than the current one, this attempt would be time consuming and limited in applicability to the specific client and environment. Therefore, it is advisable to rely on ML models to automate this process, thereby ensuring adaptability to varying conditions without the need for manual intervention.

When comparing the RTT across various scenarios, it is crucial to consider the baseline RTT values for both local and external executions. These baseline RTT values are detailed in Tables 6.4 and 6.5. Upon analysis, it becomes evident that external execution demonstrates significantly faster RTT for small and medium-sized images, being approximately 2 to 4 times faster and 1 to 3 times faster for larger images. Consequently, choosing local execution as the default choice may not be advisable in most cases. However, it should be noted that the standard deviation column indicates a greater RTT variability in external executions. This variance is often attributed to requests executed externally, particularly requests with a poor connection to the AP. Therefore, while local execution might not yield substantial time savings, it can offer consistency, as evidenced by the lower standard deviation observed in local executions. This consistency sets a limit, ensuring that the RTT does not deviate significantly from the expected value. Consequently, the reasonable use of local execution at favorable moments can enhance the overall stability of the system.

When we compare the RTTs shown in Tables 6.5 and 6.11, we can see that the RTTs for low- and medium-sized images are naturally about 10% slower than when we run requests solely externally. This is mainly because making decisions gets more complicated, especially when we add extra ML models into the equation. So, it generally takes a bit longer to process requests that involve these ML models, which is something to keep in mind when comparing RTTs. Also, if we check the time it takes for large images, it is generally higher overall. But, as I mentioned earlier, that is in part because doing things locally usually takes 1 to 3 times longer. But what is interesting is that when we compare the variation in processing time for the *TPOT-Default* model, it is consistently better for large images compared to our baseline, where we offload everything to the server.

CHAPTER 8

Conclusion

The surge in AI requests has created a bottleneck in the cloud infrastructure. To alleviate this burden, alternative solutions are essential to redistribute the load from the cloud to other devices. Edge intelligence (EI) emerges as a promising alternative, but it encounters challenges due to the heterogeneity of devices. Ensuring time efficiency is crucial, yet it is difficult to guarantee solely relying on edge devices. Failing to meet deadlines can have severe consequences, with scenarios such as car accidents illustrating the potential gravity of such situations. Therefore, it is imperative to explore robust strategies to mitigate these risks and ensure the smooth operation of AI-driven systems.

EI can be categorized into six levels, as shown in Figure 2.1. Depending on the level, the training and inference of machine learning (ML) models occur in the cloud, on edge devices, or on the device itself. In our thesis, we align with level five, all in-edge, indicating that all processes, from training to executing ML models, take place between the local and edge devices. This implies that in this thesis, no cloud services are utilized.

In our thesis, we conducted numerous experiments to gather real-world data, which is subsequently utilized to determine the optimal offloading of tasks to edge devices. We provide a tangible real-world example juxtaposed with existing mathematical approaches. The data presented in our thesis underscore the fundamental importance of experiments with real-world data, highlighting their unique insights that are not captured by purely mathematical approaches. Furthermore, these data serve as a compelling example of the challenges inherent in the search for a perfect solution, particularly due to the inherent variability and spread of real-world data. Following the execution of numerous requests and the acquisition of meaningful data through filtering, extensive experiments are conducted to discern various offload decisions. These experiments reveal that solutions relying solely on single metrics yield poor results. This motivation led us to train the ML models using the data collected throughout this thesis. We used an AutoML tool called TPOT, which evaluates various candidates and identifies the best solution based on the conditions provided. The tool offers different default configurations, and we meticulously investigated each configuration, comparing them with each other and with our own decision tree (DT). Remarkably, some of the configurations were able to accurately represent the current state, resulting in exceptional performance that far surpassed our previous results.

8.1 Research Questions

This Section contains the answers to the main research questions that are mentioned at the beginning and should summarize the main findings of this work.

RQ1: *Which metrics can be used for offloading in a black box system?*

The primary goal of this work was to develop a load balancing system that does not require additional information from the system, thus offering a universal solution to this problem applicable across various environments. However, this approach involves a limitation in terms of available information, as many systems could already provide endpoints for collecting essential data, such as the current server load, such as the solution of Álvaro Brandón et al. [BPMS18].

To address this, we collected all necessary information directly from the device itself. These data were obtained using a tool called *telemd*, which can be seamlessly integrated with *redis* to facilitate convenient use of these data. The information provided by *telemd* can be categorized into device-wide HW load, application HW load, and network-related information. Throughout our experiments, we meticulously examined all relevant information available using this tool. However, we found that only certain pieces of data were correlated with the processing of our requests. Specifically, CPU and RAM usage from both the system and the application, as well as network-related information such as the current signal connection to the connected Access Point (AP) and the Ping, was found to be relevant. In contrast, other data appeared to be arbitrary and did not demonstrate a demonstrable increase with additional activity during our experiments.

RQ2: *What offloading strategies can be used?*

The resolution of this question depends on the characteristics and capabilities of the system. Initially, in our preliminary setup, the system's capabilities were notably limited, resulting in the adoption of a broad offloading strategy, as depicted in Figure 4.2. Subsequently, a more sophisticated and realistic setup was used in subsequent experiments, as illustrated in Figure 4.4. In this situation, we were able to use the insights from our previous experiments and discussions related to the previous research questions. Another important factor is the AP used by the client and its connection, as detailed in Section 5.1.2 and analyzed in Section 6.1.2. In Figure 6.7 we can see the continued changes in signal power, which is the reason why we decide for each request whether to offload or not. In Sections 5.2, 5.3, 6.2, and 6.3, various offloading strategies were described and analyzed. These approaches encompass a single decision-making process based on a singular parameter, as well as the implementation of a more intricate DT that incorporates multiple junctions derived from various HW metrics. The findings indicate that offloading can be facilitated through these approaches. However, compared to a simplistic offloading-only approach, they yield imperfect outcomes due to their inability to precisely assess the current context, resulting in local execution where offloading would have been more efficient.

RQ3: *How do ML models handle real-world settings?*

Since the single-parameter offloading and DT offloading approach were not suitable for our scenario, resulting in an increase in round-trip time (RTT), we explored various machine learning (ML) techniques to automate the decision-making process. To accomplish this, we used a tool called TPOT, as detailed in Section 2.6.2.

From the default configurations provided by TPOT, all but one were applicable to our setup. The exception required specialized hardware not available on our client device. Subsequently, the remaining configurations were compared in Section 6.4. Although all configurations exhibited an average cross-validation accuracy greater than 95%, as shown in Table 6.9, some struggled to accurately assess the current state. Specifically, the configurations *TPOT-Light* and *TPOT-MDR* predominantly offloaded tasks to the server, rendering the use of an ML model unnecessary. In contrast, *TPOT-Sparse* tended to execute tasks locally, especially for large images, resulting in pure outcomes. Among the various configurations, only *TPOT-default* and *TPOT-NN* displayed a noteworthy capacity to adapt effectively to the environment. They achieved favorable outcomes by modifying their behavior according to the prevailing conditions. Furthermore, in addition to their adaptability, these TPOT configurations contribute to enhancing the overall stability of the system. As extensively discussed in Section 7.4.1, while these configurations may not necessarily decrease the overall RTT, they notably diminish fluctuations in RTT, thereby promoting a more consistent and stable system overall.

To conclude, in this thesis we were able to build an ML model which is aware of the current state by using a resource-aware offloading approach containing information about the current state of the client and the network conditions. Different ML models were created with TPOT and trained with an supervised learning approach. Two of the trained models are capable of improving the offloading decision and therefore improving the stability of the RTT by minimizing the standard deviation.

8.2 Future Work

While our study serves as a foundational exploration, it is inherently limited. However, our findings suggest several areas for further research, as outlined below. As discussed in Section 4.2.2, our setup incorporates multiple APs, which may cover specific areas. Depending on the environment, there may be opportunities to optimize performance by dynamically switching between APs. During these transitions, requests could initially be executed locally and later transferred to the next AP, potentially leading to performance enhancements. Another constraint in our study is that our server was dedicated solely to this experiment, consistently outperforming local execution. Future research could assess the impact of additional load on a server, as servers typically serve multiple applications concurrently. Furthermore, the introduction of alternative devices into the mix, such as the distribution of requests between multiple servers, could open up new research avenues. Additionally, it would be intriguing to explore the training of ML models tailored to specific client and server hardware and apply them to devices with similar characteristics. Are performance metrics sufficient to evaluate offloading decisions, or are additional insights required? Lastly, our study focuses on black-box testing, in which no information is available from the server. Exploring alternative testing approaches, such as white-box testing, where the server provides clients with insight into its current state, could offer valuable insights and opportunities for further analysis.

Overview of Generative AI Tools Used

ChatGPT: Used to find spelling and grammar errors.

Writefull: Used to find spelling and grammar errors.

List of Figures

List of Tables

Bibliography

[AA23] Aili Ashipala and Joshua A. Abolarinwa. Sdn-enabled data offloading and load balancing in wlan and cellular networks. 2023.

[AVM$^+$19] Mattia Antonini, Tran Huy Vu, Chulhong Min, Alessandro Montanari, Akhil Mathur, and Fahim Kawsar. Resource characterisation of personal-scale sensing models on edge accelerators. AIChallengeIoT'19, page 49–55, New York, NY, USA, 2019. Association for Computing Machinery.

[BG17] Tayebeh Bahreini and Daniel Grosu. Efficient placement of multi-component applications in edge computing systems. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[BKR$^+$00] W. Banzhaf, J.R. Koza, C. Ryan, L. Spector, and C. Jacob. Genetic programming. *IEEE Intelligent Systems and their Applications*, 15(3):74–84, 2000.

[BL16] Sourav Bhattacharya and Nicholas D. Lane. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, SenSys '16, page 176–189, New York, NY, USA, 2016. Association for Computing Machinery.

[BPMS18] Álvaro Brandón, María S. Pérez, Jesus Montes, and Alberto Sanchez. Fmone: A flexible monitoring solution at the edge. *Wireless Communications and Mobile Computingl*, 2018.

[CC18] Huijin Cao and Jun Cai. Distributed multiuser computation offloading for cloudlet-based mobile cloud computing: A game-theoretic machine learning approach. *IEEE Transactions on Vehicular Technology*, 67(1):752–764, 2018.

[CIKW16] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. Data cleaning: Overview and emerging challenges. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 2201–2206, New York, NY, USA, 2016. Association for Computing Machinery.

[CKZ22] Qinglin Chen, Zhufang Kuang, and Lian Zhao. Multiuser computation offloading and resource allocation for cloud–edge heterogeneous network. *IEEE Internet of Things Journal*, 9(5):3799–3811, 2022.

[CL21] Xing Chen and Guizhong Liu. Energy-efficient task offloading and resource allocation via deep reinforcement learning for augmented reality in mobile edge networks. *IEEE Internet of Things Journal*, 8(13):10843–10856, 2021.

[CY22] Volkan Cetin and Oktay Yıldız. A comprehensive review on data preprocessing techniques in data analysis. *Pamukkale Univ Muh Bilim Derg*, 28(2):299–312, 2022. doi: 10.5505/pajes.2021.62687.

[CYN22] Bo Chen, Zhisheng Yan, and Klara Nahrstedt. Context-aware image compression optimization for visual analytics offloading. In *Proceedings of the 13th ACM Multimedia Systems Conference*, MMSys '22, page 27–38, New York, NY, USA, 2022. Association for Computing Machinery.

[CZL+19] Bin Cao, Long Zhang, Yun Li, Daquan Feng, and Wei Cao. Intelligent offloading in multi-access edge computing: A state-of-the-art review and framework. *IEEE Communications Magazine*, 57(3):56–62, 2019.

[DZF+20] Shuiguang Deng, Hailiang Zhao, Weijia Fang, Jianwei Yin, Schahram Dustdar, and Albert Y. Zomaya. Edge intelligence: The confluence of edge computing and artificial intelligence. *IEEE Internet of Things Journal*, 7(8):7457–7469, 2020.

[GB08] Enzo Grossi and Massimo Buscema. Introduction to artificial neural networks. *European journal of gastroenterology hepatology*, 19:1046–54, 01 2008.

[GLRM16] Petko Georgiev, Nicholas D. Lane, Kiran K. Rachuri, and Cecilia Mascolo. Leo: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, page 320–333, New York, NY, USA, 2016. Association for Computing Machinery.

[HCS15] Benjamin Heintz, Abhishek Chandra, and Ramesh K. Sitaraman. Optimizing grouped aggregation in geo-distributed streaming analytics. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, page 133–144, New York, NY, USA, 2015. Association for Computing Machinery.

[HLLL20] Junyan Hu, Kenli Li, Chubo Liu, and Keqin Li. Game-based task offloading of multiple mobile devices with qos in mobile edge computing systems of limited computation capacity. 19(4), jul 2020.

[HPN+21] Luan N. T. Huynh, Quoc-Viet Pham, Tri D. T. Nguyen, Md. Delowar Hossain, Young-Rok Shin, and Eui-Nam Huh. Joint computational offloading and data-content caching in noma-mec networks. *IEEE Access*, 9:12943–12954, 2021.

[HSP+16] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, page 123–136, New York, NY, USA, 2016. Association for Computing Machinery.

[HZG20] Shijie Hao, Yuan Zhou, and Yanrong Guo. A brief survey on semantic segmentation with deep learning. *Neurocomputing*, 406:302–321, 2020.

[JJLM18] Hyuk-Jin Jeong, InChang Jeong, Hyeon-Jae Lee, and Soo-Mook Moon. Computation offloading for machine learning web apps in the edge server environment. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1492–1499, 2018.

[Jos22] V. Roshan Joseph. Optimal ratio for data splitting. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 15(4):531–538, 2022.

[Kha10] Mohd Khan. Different forms of software testing techniques for finding errors. *International Journal of Computer Science Issues*, 7, 05 2010.

[KMDN18] Jitender Kumar, Amita Malik, Sanjay K. Dhurandher, and Petros Nicopolitidis. Demand-based computation offloading framework for mobile devices. *IEEE Systems Journal*, 12(4):3693–3702, 2018.

[Kuh55] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[LBG+16] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 1–12, 2016.

[LBH15] Yann LeCun, Y. Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–44, 05 2015.

[LL18] Mengyu Liu and Yuan Liu. Price-based distributed offloading for mobile-edge computing with computation capacity constraints. *IEEE Wireless Communications Letters*, 7(3):420–423, 2018.

[LLB21] Francesc Lordan, Daniele Lezzi, and Rosa M. Badia. Colony: Parallel functions as a service on the cloud-edge continuum. In *Euro-Par 2021: Parallel Processing: 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1–3, 2021, Proceedings*, page 269–284, Berlin, Heidelberg, 2021. Springer-Verlag.

[LLJL19] Li Lin, Xiaofei Liao, Hai Jin, and Peng Li. Computation offloading toward edge computing. *Proceedings of the IEEE*, 107(8):1584–1607, 2019.

[LLZ$^+$18] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, page 389–400, New York, NY, USA, 2018. Association for Computing Machinery.

[LZC18] En Li, Zhi Zhou, and Xu Chen. Edge intelligence: On-demand deep learning model co-inference with device-edge synergy. In *Proceedings of the 2018 Workshop on Mobile Edge Communications*, MECOMM'18, page 31–36, New York, NY, USA, 2018. Association for Computing Machinery.

[MSJ$^+$20] Alessandro Montanari, Manuja Sharma, Dainius Jenkus, Mohammed Alloulah, Lorena Qendro, and Fahim Kawsar. Eperceptive: Energy reactive embedded intelligence for batteryless sensors. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, SenSys '20, page 382–394, New York, NY, USA, 2020. Association for Computing Machinery.

[PAB$^+$15] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 421–434, New York, NY, USA, 2015. Association for Computing Machinery.

[PM21] Bhanu Priya and Jyoteesh Malhotra. Qaas: Qos provisioned artificial intelligence framework for ap selection in next-generation wireless networks. *Telecommun. Syst.*, 76(2):233–249, feb 2021.

[PVI$^+$17] Charlotte Pelletier, Silvia Valero, Jordi Inglada, Nicolas Champion, Claire Marais Sicre, and Gérard Dedieu. Effect of training class label noise on classification performances for land cover mapping with satellite image time series. *Remote Sensing*, 9(2), 2017.

[RCC$^+$17] Paulo A. L. Rego, Elaine Cheong, Emanuel F. Coutinho, Fernando A. M. Trinta, Masum Z. Hasan, and Jose N. de Souza. Decision tree-based approaches for handling offloading decisions and performing adaptive monitoring in mcc systems. In *2017 5th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, pages 74–81, 2017.

[RCK+20] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459, 2020.

[RLYK21] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. Infaas: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411. USENIX Association, July 2021.

[RRP+22] Philipp Raith, Thomas Rausch, Paul Prüller, Alireza Furutanpey, and Schahram Dustdar. An end-to-end framework for benchmarking edge-cloud cluster management techniques. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*, pages 22–28, 2022.

[RWA+16] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 267–278, 2016.

[SBH19] Hugo Storm, Kathy Baylis, and Thomas Heckelei. Machine learning in agricultural and applied economics. *European Review of Agricultural Economics*, 47(3):849–892, 08 2019.

[SCZ+16a] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[SCZ+16b] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[TLL+20] Jie Tang, Shaoshan Liu, Liangkai Liu, Bo Yu, and Weisong Shi. Lopecs: A low-power edge computing system for real-time autonomous driving services. *IEEE Access*, 8:30467–30479, 2020.

[TMK17] Surat Teerapittayanon, Bradley McDanel, and H.T. Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 328–339, 2017.

[WHM+22] Jin Wang, Jia Hu, Geyong Min, Wenhan Zhan, Albert Y. Zomaya, and Nektarios Georgalas. Dependent task offloading for edge computing based on deep reinforcement learning. *IEEE Transactions on Computers*, 71(10):2449–2461, 2022.

[ZCL+19] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proceedings of the IEEE*, 107:1738–1762, 2019.

[ZF21] Yue Zhang and Jingqi Fu. Energy-efficient computation offloading strategy with tasks scheduling in edge computing. 2021.

[Zho21] Zhi-Hua Zhou. *Machine Learning*, page 459. 2021.