TU WIEN Informatics

ML-based Power Consumption Prediction Models for Edge Devices

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Theresa Christina Müller, BSc
Matrikelnummer 11931212

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 10. März 2023

Theresa Christina Müller   Schahram Dustdar

Informatics

ML-based Power Consumption Prediction Models for Edge Devices

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering & Internet Computing

by

Theresa Christina Müller, BSc
Registration Number 11931212

to the Faculty of Informatics

at the TU Wien

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

Vienna, 10th March, 2023

Theresa Christina Müller   Schahram Dustdar

Erklärung zur Verfassung der Arbeit

Theresa Christina Müller, 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.

Wien, 10. März 2023

Theresa Christina Müller

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor Schahram Dustdar and my co-supervisor Philipp Raith. This endeavor would not have been possible without their continuous and valuable feedback throughout this journey. Their extensive knowledge and expertise helped me to shape my research approach, sharpen my focus and finally be able to accomplish this thesis.

Furthermore, I am also extremely grateful to my family for their unwavering emotional support during my studies and beyond, as well as their belief in me. Words cannot express my deep thankfulness for their unconditional backing.

Lastly, I want to genuinely thank Daniel for his invaluable patience, his endless support and his encouragement to pursue my goals. I could never have finished this work without him.

Kurzfassung

Edge Computing ist ein aufkommendes Paradigma, das darauf abzielt, den Nachteilen zentraler Cloud Computing-Ansätze in Bezug auf Latenzzeiten entgegenzuwirken. Da die Anzahl an Geräten innerhalb des Edge-Cloud-Kontinuums stetig wächst, ist der steigende Energiebedarf von Edge-Computing-Umgebungen eine unvermeidliche Folge. Aufgrund zunehmender Energiekosten und globaler Stromversorgungsprobleme gewinnt die Optimierung des Stromverbrauchs von Hardwaresystemen immer mehr an Bedeutung. Obwohl präzise Strommessungen für die Bestimmung der Energieeffizienz unerlässlich sind, fehlt es an standardisierten Methoden, um den Stromverbrauch von Edge Devices einheitlich erfassen zu können. Dieser Mangel ist auf die heterogenen Hardwareeigenschaften gängiger Edge Devices zurückzuführen, weshalb die Messung des Stromverbrauchs unterschiedlicher Geräte eine zeitaufwändige, komplexe und kostspielige Aufgabe darstellt. Dadurch entsteht ein dringender Bedarf an Tools zur Vorhersage des Stromverbrauchs verschiedener Edge Devices im Kontext realistischer Anwendungsfälle.

In dieser Arbeit wird daher die Entwicklung von Modellen zur Vorhersage des Energieverbrauchs auf Grundlage von Ressourcenauslastung und unter Verwendung von Machine Learning Techniken als mögliche Lösung für dieses Problem vorgeschlagen. Um die Anwendung der Modelle in größeren Maßstäben zu ermöglichen, werden diese in den bestehenden *faas-sim* Function-as-a-Service Simulator integriert, wodurch die Energieeffizienz von Edge Devices verglichen werden kann. Zu diesem Zweck werden Experimente mit Hilfe eines dedizierten Testbeds durchgeführt, um empirische Messungen bezüglich der Ressourcennutzung und des Stromverbrauchs verschiedener Edge-Computing-Plattformen zu erfassen. Die gewonnenen Datensätze bilden anschließend die Grundlage für die Entwicklung der Vorhersagemodellen anhand eines AutoML-Tools.

Die Evaluierungsergebnisse zeigen die erfolgreiche Implementierung von Machine Learning Modellen, die in der Lage sind, den Stromverbrauch von Serverless Functions präzise mit Abweichungen von durchschnittlich 190 bis 620 mW zu prognostizieren. Gleichzeitig wird die Skalierbarkeit des Simulators durch die Modelle nicht gravierend beeinträchtigt, da die verlängerten Ausführungszeiten, die aus der zusätzlichen Vorhersagefunktionalität resultieren, noch vertretbar sind. Die zeitliche Effizienz der Simulationen kann daher unabhängig vom Overhead, der durch die Modelle entsteht, gewährleistet werden, wobei eine Vorhersage zwischen 0.8 und 2.6 ms dauert. Infolgedessen weisen die entwickelten Modelle ein zufriedenstellendes Verhältnis zwischen Performance und Genauigkeit auf.

Abstract

Edge computing is an emerging paradigm that aims at circumventing the disadvantages that centralized cloud computing approaches exhibit in terms of latency. However, as the number of interconnected devices that operate within the edge-cloud continuum is constantly growing, the total energy demand of edge computing environments increases accordingly. Due to rising energy costs and global power supply issues, optimizing the power consumption of hardware platforms and therefore reducing operational expenses becomes a critical and predominant goal. Even though obtaining accurate and uniform power measurements is therefore crucial for determining the energy efficiency of computing platforms, there is a lack of uniform methods for profiling the power draw of edge devices in a platform-agnostic way due to the severe hardware heterogeneity of common edge devices. This makes power monitoring a very time-consuming, complex and costly task and thus arises the need for easy to use facilities to forecast the power usage of different edge computing platforms in the context of realistic use case scenarios.

In this thesis, the development of power prediction models based on resource usage metrics by using machine learning techniques is proposed as a potential solution to this problem. For the purpose of applying the models in large-scale edge computing topologies, they are integrated into the existing *faas-sim* serverless simulation framework. This allows users to determine the expected overall energy consumption of a certain scenario and to rapidly and easily compare the energy efficiency of various devices. To this end, an extensive set of experiments is conducted by means of a dedicated testbed in order to obtain empirical measurements regarding the resource usage and power consumption of different edge computing platforms. The retrieved data sets subsequently form the foundation for constructing power prediction models using an automated machine learning tool.

The evaluation results demonstrate the successful establishment of generalizable machine learning models that are able to precisely estimate the power consumption of serverless function invocations solely based on resource utilization rates with MAEs between 190 and 620 mW. At the same time, they do not severely impair the scalability of the simulator as the prolonged execution time which stems from the additional power forecasting functionality is still reasonable. Therefore, the time-efficiency of the simulations can be guaranteed regardless of the overhead that is caused by the predictions, whereby one inference call takes between 0.8 and 2.6 ms on average. As a result, the developed models exhibit a satisfactory performance-accuracy trade-off.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Due to the continuous increase in the number of physical objects and devices that are interconnected over the Internet as a result of the profileration of technological trends like the Internet of Things (IoT), edge or cloud computing, the total energy demand associated with the edge-cloud continuum is constantly rising [AZS^{+}17]. According to a report published by IoT Analytics[1] in 2022, the total number of connected IoT devices will rise up to 27 billion by 2025, while 12.3 billion active endpoints were registered in 2021. Since modern applications, such as smart city, smart healthcare or smart transport systems, are spread all over the spectrum of IoT, edge, fog and cloud computing, the terms distributed computing continuum and edge-cloud continuum emerged to embrace all of these domains and to provide a fundamental computational fabric for novel large-scale distributed systems.

Considering the carbon footprint that results from the ever-increasing energy wastage caused by the growing number of devices, energy efficiency becomes increasingly important within the distributed computing continuum [AESI^{+}17, ASA^{+}21]. In addition, the rising energy costs, energy supply issues and the resulting global energy crisis further exacerbate the ever-increasing energy demand of the edge-cloud continuum and thus reinforce the importance of optimizing and reducing power usage as well as operational expenses [AQPJ21, LAA^{+}21]. Green IoT is a term that arose in this context over the past few years in order to tackle the steady energy increases within the distributed computing continuum by developing novel energy-efficient concepts for IoT-based applications. The overarching goal of green IoT is therefore to reduce the greenhouse gas emissions and thereby curbing global warming and climate change. As a consequence, the wide application of such energy-aware concepts will make modern distributed systems deployed within the edge-cloud continuum more sustainable and eco-friendly [FLLFC21].

[1]`https://iot-analytics.com/number-connected-iot-devices/`, Accessed: May 21, 2022

Apart from the environmental and monetary aspects, there are other issues that further intensify the need for innovative energy-efficient strategies, particularly when considering the fact that resource-constrained mobile devices are widely used in the edge-cloud continuum. Even though the computational resources of edge devices became more powerful over the recent years, various platforms are still restricted in terms of energy supply as compared to high-performance edge or cloud servers [JFG+20]. Specifically, many of those devices are even only battery-powered due to mobility requirements, so their power supply and therefore their lifetime is significantly limited, which has to be taken into consideration when developing edge-based applications [ASA+21]. Extending the battery lifetime of edge devices should thus be an essential design goal during application development. As as result, energy efficiency and energy-aware computation plays a critical role in such scenarios since the limited power supply represents a major constraint that can impose severe restrictions on certain use cases.

Furthermore, the edge-cloud continuum is characterized by heterogeneous devices with a wide variety of hardware capabilities regarding computational power and memory capacity [JFG+20]. Due to the diverse hardware characteristics, the electricity consumption of different types of edge devices can vary significantly. The recent advancements in terms of specialized compute platforms might even intensify those discrepancies. This new kind of computing infrastructure is specifically designed for the requirements of edge intelligence applications, that focus on Artificial Intelligence (AI) tasks, and are thus equipped with hardware accelerators such as Graphical Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs) or Tensor Processing Unit (TPUs) [CLX+21, JFG+20]. Consequently, this development of edge device infrastructure leads to the question whether traditional statistical models, e.g. linear models, that compute the electrical energy usage in proportion to the CPU utilization, are still adequate or if they are outdated and inaccurate with respect to modern hardware platforms. This would in turn amplify the need for revised power calculation models which take additional factors into account.

1.2 Problem Statement

As the number of devices that operate within the edge-cloud continuum rises, the increasing energy demand is an inevitable consequence. The rising energy costs additionally enhance the need for energy reduction and optimization. Furthermore, the resource scarcity, as well as the hardware heterogeneity of edge devices regarding computational power, memory and power supply, require edge computing platforms with high energy efficiency. These issues elicit the necessity for designing and developing energy-aware hardware and software artifacts in order to mitigate the growing carbon footprint caused by the large amount of devices operating within the distributed computing continuum, to preserve scarce energy resources and to reduce operational costs.

Currently, device profiling in terms of power consumption is very time-consuming, complex and costly. Obtaining accurate and reproducible power consumption data across different types of platforms represents a vital but non-trivial task, especially when it comes to vendor- or platform-specific measurement facilities [KHH+20]. Even though the power usage of computing systems is a critical factor, there is a lack of standardized and uniform methods for measuring the power consumption of devices in a platform-agnostic way. Therefore, there is a growing demand for easy to use facilities to forecast the power usage of different edge device types for realistic large-scale scenarios. Such a prediction tool would also allow users to determine the expected overall energy consumption of a certain scenario, to rapidly and easily compare the energy efficiency among various devices and to foster the development and evaluation of novel energy-aware task placement strategies, routing policies or other resource management algorithms [WT21]. Therefore, a system that enables the assessment of the power usage of edge devices for a certain use case configuration and also helps with identifying the most energy-efficient device among a set of edge devices would be crucial for settings where energy awareness is a key aspect.

1.3 Solution Approach

Power consumption prediction models that are able to estimate the power usage of edge devices based on given resource utilization metrics could help to solve this issue and overcome the lack of power measurement and prediction facilities. This way, electrical energy wastage could be reduced and the selection of appropriate edge device types for application deployment and execution could be fostered. Moreover, such forecasting models could support future energy consumption estimation and therefore assist in energy management and planning, e.g. if the power supply is limited in a certain scenario.

For this purpose, the overall aim of this work is to develop generalizable forecasting models that are able to predict the power consumption of widely used edge devices solely on the basis of resource usage metrics. Hence, a resource utilization-based power modeling approach which incorporates various hardware parameters is used. As a consequence, the final models should be independent of the underlying application and thus be capable of making predictions for unknown applications. For this purpose, real-time profiling of common edge computing hardware using black-box system monitoring and power measurement facilities are performed during a series of experiments for gathering data about power draw, CPU utilization, network I/O rate, memory usage, etc. The sample applications used for the profiling experiments are serverless functions that focus on Machine Learning (ML) inference workloads as well as controlled stress tests provoked by a dedicated workload generation tool. While the execution of different serverless functions on the edge devices can give valuable insights into the impact of hardware heterogeneity on the resource usage behavior of individual computing platforms, the stress tests can reveal potential platform-specific variations in terms of power consumption across devices despite equal workloads. This way, the impact of hardware heterogeneity on the resource utilization rates and on the power draw of distinct edge devices can be assessed.

Afterwards, machine learning techniques, more precisely regression-based ML algorithms, are applied to the experimentally obtained data as part of the model development process to establish sophisticated power prediction models. By incorporating the results of the stress tests into the model training phase, more generalizable predictions can be achieved, since the additional samples foster more diverse and more comprehensive data sets. In order to provide developers with an easy and ready-to-use prediction tool for large-scale edge computing scenarios, the forecasting functionality is not developed individually as a standalone artifact, instead, the models are made publicly available as an additional feature of an existing Function-as-a-Service (FaaS) simulation framework. The prediction models are therefore developed using a simulator-based energy consumption estimation approach. As a result, making use of the prediction capability integrated into the existing simulation framework does not require complicated and laborious experiments for device profiling and also eliminates the effort of a complex setup.

For this purpose, the *faas-sim*[2], which is a trace-driven FaaS simulation framework developed by the Distributed Systems Group at TU Wien, is chosen as fundamental platform for embedding the additional power prediction functionality. The *faas-sim* simulator can imitate serverless container-based platforms such as OpenFaaS by simulating the execution of serverless functions and workloads on cluster nodes. By incorporating the power models into a simulator platform, simulations of realistic large-scale scenarios are facilitated, which would not be feasible to carry out on small-scale testbeds. Nevertheless, using a simulator framework imposes several limitations and thus challenges which need to be addressed within the model development process.

In order to put the envisaged work into a broader context, the resulting machine learning models can be used to assess the differences in terms of power consumption between multiple nodes running the same application and therefore compare the energy-efficiency of various types of edge devices. However, the applicability of the models is not limited to their usage for decision support regarding the energy efficiency of edge devices. The power consumption prediction functionality of the *faas-sim* simulation platform could also be used for energy management and planning, e.g. in case battery-powered devices are employed, so their lifetime can be extended. Furthermore, the developed models might be helpful for developing intelligent energy-aware scheduling, task offloading or application placement strategies, so the applicability of the prediction models is not restricted to the utilized simulator as such. Instead, the models could also be transferred to other problem domains associated with the distributed computing continuum.

To sum up, the focus of this work lies on the development of power consumption prediction models for edge devices using ML techniques in order to support energy efficiency and awareness for applications developed and deployed within the edge-cloud continuum. More precisely, elaborate models based on various resource usage metrics that have a pivotal impact on the power draw of a device are targeted. In order to enable power forecasting for large-scale edge computing scenarios and to provide an easy-to-use prediction facility for serverless functions, the models are integrated into the existing *faas-sim* simulation framework. The accuracy and performance of the developed models is ascertained by means of several evaluation criteria.

[2]`https://github.com/edgerun/faas-sim`, Accessed: May 21, 2022

1.4 Research Questions

- **RQ 1**: What are appropriate methods for measuring the power consumption of edge devices and how do they differ?

  In general, measuring the power consumption of edge computing platforms is a challenging task due to the hardware platform heterogeneity that is associated with the edge domain. However, accurate and consistent power measurements are crucial for optimizing the power demand as well as the energy efficiency of computer system components and for developing energy-efficient software artifacts. Especially battery-operated devices, such as mobile platforms, wireless sensors or wearables, require low power and high energy efficiency in order to maximize the battery lifetime and therefore the operating time.

  Up until now, a wide variety of measurement facilities exists and in many cases various techniques need to be applied in order to determine the power consumption of disparate device types with diverse hardware architectures. This makes power measurements on heterogeneous platforms complex and laborious [KHH$^+$20, GCZY21]. As a result, this thesis aims at examining different kinds of measurement instruments for retrieving the power consumption of edge devices. Furthermore, it should be analyzed how the individual methods differ in terms of granularity, measuring approach, power data source, profiling overhead, sampling frequency, setup costs and equipment costs. The chosen measurement facilities can then be used to profile edge devices at runtime during a series of experiments while also monitoring the resource usage by collecting black-box system metrics. This way, a relationship between the power consumption and the resource utilization parameters of edge devices can be established, which builds the foundation for the development of machine learning models.

- **RQ 2**: How severe is the impact of hardware heterogeneity on the resource usage and power consumption of different devices and what does this mean for power consumption modeling approaches in the edge domain?

  The edge-cloud continuum is characterized by a large number of partially resource-constrained devices with varying capabilities regarding computational power and storage capacity, as well as different energy sources. As a consequence, it is presumed that this infrastructure heterogeneity has a substantial impact on the resource usage and power consumption of devices. Furthermore, the recent advancements in terms of hardware accelerator platforms, such as novel computing architectures equipped with high-end GPUs or TPUs, could potentially even intensify this effect.

  Therefore, the level of variation with respect to resource utilization and power draw between multiple types of edge computing hardware has to be ascertained. For this purpose, the experiments have to be designed in a way that enables a comparison of resource usage metrics across devices by executing the same applications on different platforms. In addition, further experiments have to be defined which focus on straining the hardware components of heterogeneous edge devices on equal levels so that the power consumption readings for similar resource utilization rates can be contrasted. Depending on the results of these analyses, the impact of hardware heterogeneity might have implications for power consumption modeling techniques in the edge domain.

- **RQ 3**: How can the power consumption of edge devices accurately be modeled in a simulation environment considering the strict performance requirements demanded by the underlying simulation framework, and how does the chosen energy modeling approach affect the execution time and scalability of the simulator?

  Since the final power prediction functionality is integrated into the *faas-sim* simulation framework, which already incorporates methods for modeling the performance and the resource utilization of serverless functions, a suitable energy modeling technique has to be applied in order to be able to accurately estimate the power consumption of edge devices during simulations. For the purpose of developing generalizable models that are capable of making predictions for unknown applications, the energy modeling approach has to rely solely on resource usage metrics, so no information about the corresponding serverless function is required. As the aim of this work is to develop power models based on machine learning algorithms, it has to be analyzed whether this represents a viable power modeling approach for heterogeneous edge devices and simulation environments.

  By using a serverless simulation platform as the basis for incorporating the power models, certain limitations caused by inherent simulator characteristics have to be accepted. As compared to executing a serverless function on a real FaaS platform, such as OpenFaaS, the *faas-sim* framework aims at providing a time-efficient simulation environment, whereby the execution time of the simulator should be significantly lower than the actual runtime of the simulated scenario in the real world. Hence, the performance of the developed models plays a crucial role, since periodically predicting the power consumption of a large number of devices during a simulation introduces a certain degree of overhead on the execution time.

  Depending on the complexity of the models and therefore the computation time per prediction, the additional computational burden imposed by the forecasting models might impede the performance and the scalability of the simulator. As a consequence, the performance requriements imposed by the underlying simulation environment have to be taken into account during the model development process. The inference accuracy is thus not the only predominant design goal of the models, instead, lightweight models that exhibit a satisfactory compromise between prediction precision and performance need to be established.

1.5 Structure of the Thesis

The remainder of this thesis is structured as follows. Chapter 2 outlines the fundamental background with respect to concepts and technologies used in the course of this work. Afterwards, Chapter 3 presents related work in order to provide an overview of existing research efforts in this subject area and to highlight key differences in comparison with the intended approach and outcome of this thesis. The methodological approach applied for the purpose of establishing machine learning models that are able to predict the power consumption of edge devices is covered in Chapter 4. Therefore, this chapter encompasses the design and setup of the experiments, the analysis of the empirical measurement data and the actual construction of the prediction models. Subsequently, Chapter 5 details the specific approach and the results of the model evaluation procedure. This includes an in-depth assessment of the prediction accuracy as well as a profound examination of the performance of the models. The following chapter, namely Chapter 6, points out known limitations in terms of the conducted measurements and the resulting power prediction functionality integrated into the simulation framework. Finally, Chapter 7 concludes the thesis by summarizing the key findings and contributions of this work and provides an outlook regarding future work on this topic.

CHAPTER 2

Background

This chapter outlines the fundamental background that serves as the basis for this work. Therefore, the following sections are supposed to put the topic into context and to provide a basic understanding of the underlying concepts and technologies that are used in this thesis. Since this work resides in the context of edge computing, Section 2.1 outlines the basics of the emerging edge computing paradigm by presenting the underlying motivation for this novel computing model, its benefits and open challenges, as well as different methods for the evaluation of edge computing components. Furthermore, this section tries to give a basic understanding of edge intelligence, which represents a recent interdisciplinary research field that gained interest over the past years, and also deals with the multifaceted heterogeneous characteristics of edge computing environments. Afterwards, Section 2.1 concludes with the tightly thesis-related notion of energy-awareness in the context of edge networks.

In order to introduce the existing simulation platform that is used in the course of this work, the subsequent section, i.e. Section 2.2, presents detailed information about the *faas-sim* serverless simulation framework. The *faas-sim* simulator is ultimately equipped with a power prediction functionality based on the machine learning models which are developed throughout this thesis. As the employed simulation framework is based on the concept of serverless computing, the following section, namely Section 2.3, covers the basics of the serverless computing paradigm. This section therefore focuses on the characteristics and benefits of the serverless paradigm, also with special regard to the usage of the associated concepts in edge computing scenarios. Additionally, Section 2.3 encompasses a non-extensive compilation of challenges that stem from the integration of serverless into edge computing. Finally, since the solution approach applied in this work focuses on the establishment on machine learning models for predicting the power consumption of edge devices, the terms Artificial Intelligence, Machine Learning and Deep Learning are described in Section 2.4. In order to emphasize the delimitation between these terms, this section also clarifies their interrelation and differentiation.

2.1 Edge Computing

Edge computing is a popular computing paradigm that evolved over the last years in response to the high latency that cloud computing approaches exhibit [Sat17]. In traditional cloud computing models, IoT devices sense the surrounding environment and transfer the generated data over a network to a cloud data center for central processing of the large amounts of sensor data. The cloud data center is thereby located remotely in the core of the network and comprised of high-performance cloud servers [CLMS20]. However, many IoT-based applications have real-time demands where low latency and rapid feedback are crucial for successful operation, which cannot be guaranteed by the cloud computing paradigm due to the considerable delay that is caused by sending all the data to the cloud and the related high pressure on the network bandwidth [HS19]. Furthermore, due to the constantly rising amount of data generated by IoT devices, processing all sensor data in the cloud will become infeasible at some point because of the transmission bottleneck cloud computing exhibits [CCPB21, WT21]. As a result, cloud computing approaches might not be able to ensure the strict Quality-of-Service (QoS) guarantees of delay-critical applications [JFG$^+$20].

In order to circumvent the shortcomings that are associated with processing all the raw sensor data centrally in cloud data centers, edge computing was introduced as a novel computing paradigm, whereby computational workloads and storage capabilities are shifted from the cloud to the edge of the network [JFG$^+$20, ZLH$^+$18]. This kind of computation offloading is achieved by introducing a new layer at the edge of the network, i.e. between the IoT and the cloud domain, that consists of distributed compute nodes, which are able to preprocess and aggregate the raw sensor data produced by IoT devices [WT21]. Edge computing therefore represents a decentralized and local data processing approach, since edge nodes are highly geographically distributed but still in close vicinity to the end devices, as compared to the centralized, remote cloud [CCPB21].

By providing computational and storage resources at the edge of the network and thus in immediate proximity to the end devices and end users, the latency and response times can be enhanced, since the physical distance between the IoT devices and the processing unit is minimized. Consequently, the data transmission delay can be reduced and user experience can eventually be improved [HS19, WT21]. Especially time-sensitive applications that demand real-time processing and instant feedback in order to fulfil certain QoS requirements can benefit from the edge computing paradigm and therefore represent typical edge computing application scenarios.

Additional benefits of decentralized edge computing approaches, that stem from the inherent characteristics of edge computing environments, are reduced bandwidth requirements and usage, high scalability, mobility support, as well as location and context awareness, which are particularly useful for applications that depend on local context information such as the location of the user [CCPB21, DDTD19]. Example use cases that are characteristic for edge computing include smart cities, smart home or smart healthcare systems as well as augmented reality applications [CCPB21].

However, edge devices exhibit limited storage and computing capacities in comparison to high-performance cloud servers. For long-term, permanent storage, in-depth analysis of pre-processed sensor data, resource-intensive calculations without real-time demands and for integrating global sensor data information, edge computing approaches can therefore still rely on powerful cloud data centers. Hence, the emerging edge computing paradigm should not be considered as a replacement of the cloud computing approach, but rather as a complement, so the two technologies can coexist and augment each other [CLMS20, CCPB21].

In summary, edge computing provides the following benefits:

- Low latency and fast response times enabling real-time applications and services
- Less bandwidth utilization and transmission overhead
- Better service delivery, user experience and QoS
- Mobility support
- Location and context awareness
- High scalability

Even though the edge computing paradigm mitigates various disadvantages of cloud computing approaches, there are several remaining challenges and open issues that still need to be addressed. These challenges include computation offloading, hardware and networking technology heterogeneity, security and privacy mechanisms, as well as reliability [CCPB21]. Computation offloading involves multiple decisions, i.e. when to process tasks locally and when to outsource them to other devices, how to perform the offloading process and which nodes to select for offloading. More specifically, these decisions are influenced by manifold optimization metrics such as energy consumption, in particular the trade-off between computation energy consumption and transmission energy consumption, bandwidth, latency, cost and computational performance [CCPB21, CZS18].

Regarding security and privacy, edge computing can increase data security and privacy by aggregating, anonymizing and processing the data in close proximity to its source, but at the same time, the novel architecture of edge computing environments introduces additional attack surfaces due to their highly dynamic and distributed nature. Moreover, well-established privacy and security measures are often too heavyweight to be applied to edge scenarios, so new lightweight mechanisms are required [CLMS20]. Edge computing environments also evoke new challenges in terms of reliability and fault tolerance because of potential device failures, network fluctuations, mobility requirements or battery constraints [CCPB21]. Implications of edge device and edge networking technology heterogeneity are thoroughly discussed in Section 2.1.3.

2.1.1 Experimentation and Evaluation Methods for Edge Environments

In order to be able to design, develop, deploy and evaluate edge computing applications and scenarios under controlled and repeatable conditions, different methods exist. In general, they can be classified into three categories, namely testbeds, simulators and emulators, whereby all of them aim at the experimental evaluation of networking and computing tasks, e.g. algorithms, protocols or resource management strategies.

Testbeds Physical testbeds provide configurable environments that are similar to the real deployment environments, so they try to reproduce the actual scenario by supplying various interconnected nodes, i.e. real devices, that can be used for developing and evaluating applications. Therefore, testbeds typically represent the most realistic setup and thus enable accurate analyses. However, they are usually rather costly in terms of installation and maintenance, which is why testbeds are commonly used for small-scale experiments, while edge computing environments typically require large-scale infrastructure evaluation with a huge number of heterogeneous devices. Apart from their restricted scalability, testbeds often exhibit limited flexibility when it comes to network topologies and face issues regarding reproducibility and failures of physical hardware components [ZCS19].

Simulators Simulation frameworks aim at modeling and predicting system behavior by simulating the execution of code on cluster nodes with configurable topology. As such, simulations enable convenient and reproducible experiments that can be conducted in a time- and cost-efficient way [ZCS19, ISH10]. Therefore, realistic large-scale scenarios with a vast number of devices, including emerging computing infrastructures, can be modeled which would not be feasible to perform on small-scale testbeds. Furthermore, the use of simulators does not require an elaborate setup of a physical hardware environment [GCZY21]. As a result, simulations are widely used to evaluate resource management strategies. Advantages of such frameworks are therefore low costs, rapidity, high scalability and ease of use. Nevertheless, they might generate non-realistic results, which might not be as accurate as testbed outputs, since there is an inherent discrepancy between the real world and the simulation environment. Furthermore, simulators typically rely on estimations and assumptions about the real world that often represent imprecise simplifications [ZCS19, ISH10]. Simulation frameworks thus exhibit a trade-off between performance and accuracy. Besides general purpose simulators, specialized platforms exist, e.g. for simulations in the context of edge-cloud continuum the IoTSim-Osmosis [AJH^{+}21] was proposed, which estimates the battery draining of IoT devices for sensing tasks.

Emulators Emulators represent a middle ground between testbeds and simulators by combining real components with simulated ones and thereby enabling realistic large-scale scenarios. Emulations merge the realism of physical testbeds and the configurability, reproducibility and scalability of simulations. As a result, emulators are able to generate more realistic results than simulations and facilitate portability of the used code to real devices. Nonetheless, emulations are not as time-efficient as simulations and also require more hardware resources, which is why the costs are typically higher [ZCS19, BRKP22].

In this work, a simulator is used for the development and evaluation of the machine learning models, which requires the modeling of power consumption for different devices. Therefore, the method applied in the course of this thesis represents a simulator-based energy consumption estimation approach. The *faas-sim* simulation framework that is employed for this purpose is thoroughly described in Section 2.2.

2.1.2 Edge Intelligence

Edge intelligence (EI) or edge AI is a novel paradigm that describes the symbiosis of edge computing and artificial intelligence in order to enable the usage of AI capabilities at the edge of the network by performing model training and inference on edge devices [DZF+20]. As such, edge intelligence represents an interdisciplinary research field. AI applications can be deployed on the edge and thus taking advantage of edge computing benefits such as low latency, real-time processing and context awareness. Furthermore, by bringing AI applications to the edge of the network, the ability of AI techniques for rapid analysis of large amounts of data can be exploited in order to obtain meaningful insights into sensor data bulks and to make valuable, high-quality decisions in real time [ZCL+19, DZF+20].

A key enabler for this innovative technological trend is the development of hardware accelerators for edge devices with novel computing architectures, i.e. specialized computing platforms tailored at AI tasks and applications. The demand for this new kind of dedicated computing hardware emerged due to the fact that AI applications are typically very resource-intensive, so they require high computational power that cannot be provided by conventional resource-constrained edge devices. As a result, several high-end processors and AI chips that are able to fulfil these specific requirements were developed, such as Graphics Processing Units (GPUs) or Tensor Processing Units (TPUs) [ZCL+19, DZF+20]. The authors of [CLX+21] identified four categories of specialized edge AI platforms based on their internal physical structure, namely Application-Specific Integrated Circuit (ASIC) chips, GPUs, Field Programmable Gate Arrays (FGPAs) and brain-inspired chips.

Commercially available examples for edge devices that are equipped with such hardware accelerators include Google's Coral product line[1] which encompasses a single-board computer with an embedded Edge TPU, i.e. an ASIC designed by Google that is able to conduct high performance on-device inference, as well as an USB accelerator which can extend existing devices with Google's Edge TPU. Furthermore, GPU-enabled edge devices comprise Nvidia's Jetson boards[2] with integrated graphics processing units, such as the Jetson Nano, Jetson Xavier NX or Jetson TX2, among others. Due to their

[1]`https://coral.ai/products/`, Accessed: Sept 5, 2022
[2]`https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/`, Accessed: Sept 5, 2022

high processing power, edge devices equipped with hardware accelerators are capable of efficiently performing AI tasks on the edge of the network and thus pave the way for edge intelligence applications. Additionally, many of these novel hardware platforms exhibit a high level of energy efficiency [DZF+20].

Since the model training phase generally requires significantly more resources than the actual inference, many researchers propose to perform the training of machine learning models in powerful cloud data centers and only carry out the inference phase on the edge [KHH+20, WBC+19]. Nevertheless, within the edge-cloud continuum, edge nodes and cloud servers can collaboratively train and run AI models for performance optimization, so the domain boundaries become blurred. The decision on whether model training and inference are carried out at the edge of the network, within the cloud or in a hybrid fashion is dependent on the application-specific requirements, since the individual solutions exhibit different trade-offs between latency, bandwidth, privacy and energy consumption [ZCL+19]. The authors of [ZCL+19] therefore introduce a system to classify edge intelligence platforms, which comprises six levels that differ in terms of path length and the amount of data being offloaded from the cloud to the edge devices. The individual levels range from cloud-edge co-inference with cloud-only model training, i.e. level 1, to on-device model training an inference, i.e. level 6. As described at a later stage, the focus of this work lies on level 3, which is characterized by on-device inference at the edge while the model training is performed in the cloud.

Advanced Driver-Assistance Systems (ADAS) represent a typical edge AI application, where edge devices are able to carry out AI-based tasks such as vehicle or pedestrian detection and traffic sign recognition in order to provide instant feedback to the driver as required in highly dynamic and fast-changing traffic situations. Other examples of edge AI use cases include real-time video analytics, public transportation planning or traffic control systems [ZCL+19].

2.1.3 Heterogeneity in Edge Computing Environments

As mentioned earlier, the edge domain is characterized by a vast number of heterogeneous devices that differ in terms of computational power, storage capacity and energy supply. Even though edge nodes generally provide more powerful computing and storage resources than IoT devices, they are still restricted regarding processing power and memory as compared to high-performance cloud servers.

The diverse hardware capabilities of individual edge nodes range from resource-constrained mobile devices, that are typically battery-powered and therefore have a limited lifetime, to more powerful edge servers with higher processing performance for computationally intensive tasks [JFG+20]. Typical edge devices encompass Single Board Computers (SBCs), such as Raspberry Pis, mobile devices including smartphones, drones or health monitoring devices, embedded AI hardware like Nvidia's Jetson board series with GPU support, as well as conventional server computers, commonly called edge servers [ATC+21, RLF+20].

Especially battery-powered platforms are significantly impaired due to the limited power supply and the finite lifetime of batteries, which represents a severe constraint that can impose serious restrictions on certain use cases. As a consequence, lightweight and durable devices are required for the deployment of edge-based applications [JFG+20]. The emergence of specialized compute platforms for AI tasks, as described in Section 2.1.2, even intensifies the computing hardware discrepancy among edge devices.

In cloud computing scenarios, resources are locally clustered and rather homogeneous, whereas edge computing environments consist of geographically widespread resources that are highly heterogeneous, so hardware heterogeneity represents a challenge that emerged due to the novel architecture of edge computing networks. Apart from the device heterogeneity, the operating systems and software stacks highly vary between different types of edge nodes as well, which hampers a seamless integration and interaction [JFG+20]. These varieties lead to several implications and difficulties. For example, the integration of different device types leads to complex heterogeneous systems with a mix of proprietary and open hardware and software from different vendors without uniform standards. Hence, standardized protocols and well-defined interfaces are necessary in order to ensure interoperability and compatibility among various platforms [CCPB21].

In addition to heterogeneity in terms of individual hardware and software components, the networking technologies also vary in edge computing architectures. Wireless networking technologies might include Bluetooth, Wifi, 3G, 4G (LTE) or 5G networks. Therefore, a seamless and smooth transfer between diverse networking technologies needs to be guaranteed, which represents an additional challenge in such scenarios [CCPB21]. This results in highly heterogeneous systems concerning not only the devices themselves but also the available networking technologies and software stacks.

With respect to this thesis, which focuses on the energy consumption of edge devices, the aforementioned hardware disparities presumably lead to considerable differences regarding the power usage of individual edge device types, especially since hardware accelerators often incorporate advanced energy efficiency mechanisms in contrast to traditional edge computing architectures [DZF+20].

2.1.4 Energy-aware Edge Computing

Since the number of edge devices is constantly rising and the total energy consumption of edge computing environments is tremendous, research regarding energy-aware edge computing gains more and more interest. Another important factor that promotes research efforts in this field is the limited power supply as exhibited by battery-powered devices or power-constrained edge nodes. By deploying energy-efficient hardware and implementing energy-aware software solutions, the lifetime of devices with limited power sources can be extended and quality of service guarantees can be ensured. Considering energy awareness in edge computing environments as an overarching research area, multiple aspects need to be taken into account, including hardware design, resource management and scheduling as well as computation offloading, among others [JFG+20].

Especially energy-aware computation offloading or workload allocation strategies are non-trivial, since streamlining the power consumption might compete with other optimization metrics such as latency, bandwidth and cost. Therefore, multi-objective optimization approaches are crucial in order to meet application demands. Furthermore, there is a trade-off between the transmission energy consumption and the computation energy usage, so the transmission overhead that stems from task offloading needs to be compared to the energy required for processing a workload locally. However, energy-aware workload allocation strategies should also consider the total power consumption of all devices involved, including transmission and computation energy consumption, which makes it an even more complex task [CZS18, JFG+20].

Regarding energy-aware hardware design, the energy efficiency of different edge computing platforms widely ranges. Due to the ever-increasing computing demands, low power hardware design gains importance. Because of the widespread deployment of edge devices, energy efficiency at the hardware level is inevitable in order to achieve power savings and meet application demands. As mentioned earlier, energy awareness benefits from the recent advancements in AI-enabled edge devices that are equipped with hardware accelerators, since they often include energy consumption optimizations, while providing high computing power at the same time [JFG+20, DZF+20]. In the course of this thesis, the differences in terms of power consumption between various edge devices are investigated and assessed, so implications for power modeling approaches can be inferred. Furthermore, the integration of the power models into an existing simulation framework foster the development of energy-aware applications and the selection of energy-efficient hardware infrastructures.

2.2 Faas-sim Serverless Simulation Framework

The *faas-sim*[3,4] represents a trace-driven simulation framework for serverless Function-as-a-Service platforms, which is employed in the course of this thesis for developing, validating and deploying of the power prediction models. As such, it can be used to simulate the execution of serverless functions and workloads of container-based FaaS platforms like OpenFaaS. Details about the concept of serverless computing can be found in Section 2.3. The *faas-sim* framework is developed and maintained by the Distributed Systems Group at TU Wien within the scope of research efforts targeting serverless edge computing systems, whereby its design and architecture is strongly coined by OpenFaaS. As such, it is publicly available under the MIT license.

The open-source simulator supports the development and evaluation of operational strategies such as load balancing or scheduling, as well as optimization techniques thereof. It therefore offers plug-in support for self-developed schedulers, load balancers or autoscalers. In order to be able to simulate function execution on cluster nodes, the *faas-sim* simulator provides different node types, such as commonly used mobile edge devices,

[3]`https://github.com/edgerun/faas-sim`, Accessed: Sept 22, 2022
[4]`https://edgerun.github.io/faas-sim/`, Accessed: Sept 22, 2022

and already comprises traces from conventional computing devices and representative workloads for modeling different infrastructure scenarios. Traces are triggered through requests made by clients and typically refer to program execution or function invocation logs that record various system actions during a program run including timestamps and durations, such as the function execution time.

By providing a serverless simulation framework, *faas-sim* allows for modeling realistic large-scale scenarios, which would not be feasible on small-scale testbeds. As a result, users can configure and experiment with different scenarios and use cases in a time-efficient manner. However, simulation platforms generally exhibit certain challenges that are caused by inherent simulator characteristics. On the one hand, there is a discrepancy regarding resource utilization between the simulation environment and the real world, since the resource usage of devices has to be modeled and thus estimated to a certain degree during a simulation. On the other hand, simulation frameworks aim at providing a time-efficient simulation environment, whereby the execution time of the simulation should be significantly lower than the actual runtime of the simulated scenario in the real world. Consequently, performance and execution time play a crucial role, especially with respect to the scalability of the simulator. These aspects have to be taken into account when developing the power prediction models and also need to be validated afterwards by evaluating the performance and the inference accuracy of the final models.

In the current version of the *faas-sim* simulation platform, a power prediction functionality is not included. Hence, by adding this capability to the simulator, multiple problems can be addressed. Firstly, the determination of energy efficiency of different edge device types is facilitated, which eases the selection of appropriate edge devices for application deployment. This could in turn lead to reduced electrical energy wastage. Regarding the power prediction models themselves, they could also be exploited for the existing simulator scheduler in order to take energy efficiency and low power consumption into account as an optimization goal for scheduling decisions. Other simulation frameworks, such as the IoTSim-Osmosis [AJH+21], already incorporate power prediction functionalities to some extend, as detailed in Section 3.2, but some of the power models that are used for this purpose are rather simple and rigid. Therefore, the final power forecasting feature of the *faas-sim* should represent an elaborate and comprehensive approach to precisely estimate the power usage of edge devices.

2.3 Serverless Computing

Serverless computing represents a cloud-native paradigm that provides a simplified programming and execution model for deploying applications and services. As such, it abstracts away various operational aspects developers using Infrastructure-as-a-Service (IaaS) offerings are usually confronted with, e.g. resource provisioning, deployment, scaling, fault tolerance, maintenance and so on. Serverless computing platforms are able to execute small pieces of software as the basic computing and deployment unit, i.e. so-called stateless functions, whereby users are not responsible for managing the underlying resources, since they are administered by the service provider. Hence, developers can also benefit from the dynamic autoscaling capabilities of serverless computing without explicit resource provisioning. Additionally, due to the pay-per-use or pay-as-you-go billing model of serverless computing, charges only apply to resources that are actually used instead of charges based on resource allocation, which typically results in lower costs. The term *serverless* in no way signifies that no servers are involved in the process, but rather that users can deploy application code without the burden of server provisioning and administration, which significantly simplifies application deployment [BCC+17, JSSS+19]. As described in Section 2.2, the *faas-sim* simulation framework also relies on the serverless computing paradigm for function deployment and execution.

Serverless computing evolved due to a lack of paradigm that offers a pay-per-use model and provides effortless scalability of resources. The emergence of serverless computing was enabled by technological advances in multiple areas, namely microservice architectures, the Function-as-a-Service (FaaS) model, event-driven programming as well as containerization, whereby the serverless concept encompasses all of these technologies [ATC+21]. FaaS thereby represents a key enabler and the core of serverless computing, which inherently differs from traditional cloud provider offerings, such as Infrastructure-as-a-Service (IaaS) or Platform-as-a-Service (PaaS). FaaS platforms provide an additional level of abstraction, as depicted in Figure 2.1, which outlines the differences of IaaS, PaaS and FaaS in terms of user responsibilities regarding the control and management of the underlying infrastructure.

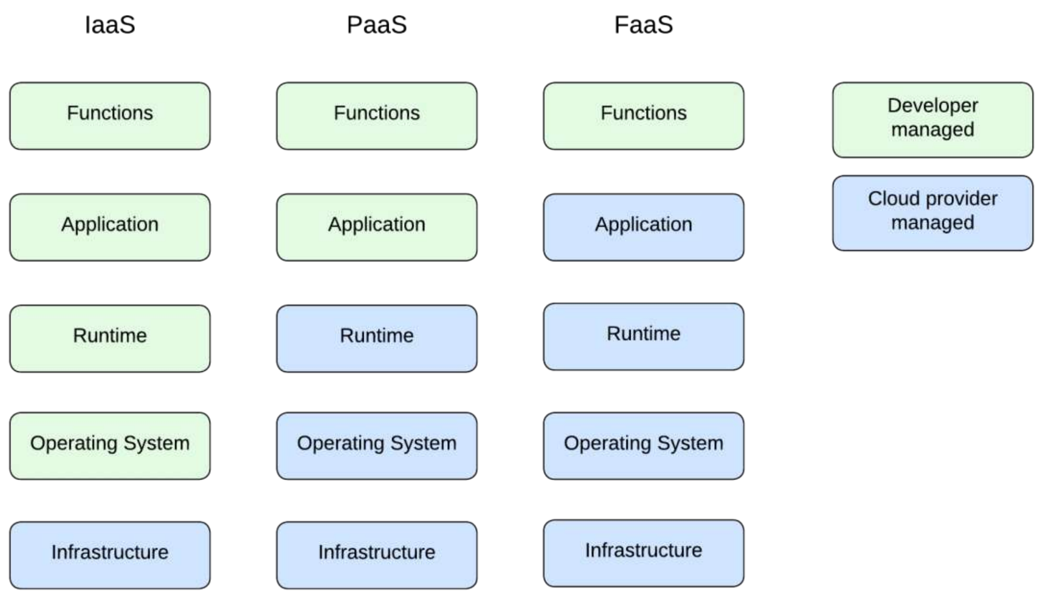


Figure 2.1: Comparison between IaaS, PaaS and FaaS.

While the IaaS model provides the most control over hardware resources, users have to manually configure and manage the underlying infrastructure, i.e. virtual machines and bare metal servers, as a consequence, including resource setup, provisioning and administration as well as scaling. This makes IaaS highly customizable, but also very complex and cumbersome. In PaaS offerings, developers are no longer responsible for server administration, since this is done by the service provider, but they still have to configure scaling strategies to some extend, i.e. the number of instances the application should run on. Furthermore, pre-packaged applications are required to be deployed on PaaS platforms. The FaaS model offers the least control over the physical infrastructure, but provides automatic resource provisioning, deployment and fault tolerance, besides other services, and uses functions as basic deployment units. Additionally, the dynamic autoscaling mechanisms of FaaS platforms enable resource allocation on demand [BCC+17].

OpenFaaS[5] is a well-known example of an existing open-source serverless platform that enables the easy deployment of microservices and event-driven functions, which can be submitted as Docker containers. Furthermore, it can be used to package code, binaries or containers as serverless functions. OpenFaaS incorporates different container orchestrators such as Kubernetes or OpenShift and supports various programming languages like Python, Java, Go or Ruby[6]. The auto-scaling capability of OpenFaaS facilitates demand-dependent scaling based on the current workload and also includes scaling down to zero when the application is idle. With OpenFaaS, developers can run their code on any public or private cloud, so portability is fostered and vendor lock-in is impeded.

Serverless Edge Computing

Initially, serverless computing was devised for cloud environments, but since it provides significant benefits, serverless is adapted and integrated into edge computing scenarios as well. As IoT-based applications often exhibit unpredictable and varying workloads, the autoscaling capability of serverless computing models can be harnessed in order to automatically respond to dynamically changing workloads. Many applications deployed in edge environments do not require the application to be up and running continuously due to periodical instead of steady data collection and processing. Therefore, the serverless scale to zero ability for idle applications and services could lead to reduced energy consumption, save scarce edge device resources and extend battery lifetime, while the user only needs to pay for the resources that are actually used [ATC+21]. As a result, serverless edge computing represents a promising computing paradigm that leverages both serverless and edge computing.

Nonetheless, integrating the serverless paradigm into edge computing environments raises certain challenges that need to be addressed in order to fully unleash the potential of

[5]`https://github.com/openfaas/faas`, Accessed: Sept 22, 2022
[6]`https://docs.openfaas.com/`, Accessed: Sept 22, 2022

serverless edge computing. Some of these concerns not only stem from the combination of the two paradigms, instead, they are inherent to serverless computing, but due to the integration into edge networks, these issues are exacerbated. First of all, cold starts of serverless functions upon the first function invocation or after scaling to zero might pose a problem for latency-sensitive applications, since they impose a certain delay which might lead to performance degradation. Therefore, innovative solutions are required in order to circumvent the cold start delays for time-critical applications [ATC+21].

Furthermore, edge AI application may not be able to fully benefit from serverless computing, because they encompass long-running I/O-intensive tasks, whereas serverless is most cost-efficient for short-running CPU-bound jobs. This cost-efficiency issue does not only affect edge AI scenarios, but applications with continuous workloads in general. Moreover, reliability and fault tolerance constitute two major challenges of serverless edge computing, since appropriate robust and resilient techniques still need to be adopted in edge computing environments. Security represents another challenge that serverless edge computing platforms are confronted with, since edge networks are more vulnerable to attacks and containers are less isolated than virtual machines as used in the cloud [ATC+21]. Nevertheless, the aforementioned aspects only cover a fraction of demanding factors that need to be faced in the context of serverless edge computing.

2.4 Artificial Intelligence, Machine Learning and Deep Learning

2.4.1 Artificial Intelligence

Artificial Intelligence (AI) is a technological trend that aims at enabling machines to simulate intelligent human behavior by providing rapid analysis of vast data bulks in order to obtain meaningful insights, identify complex patterns, facilitate reasonable future predictions and foster real-time decision making [ZCL+19, CLX+21]. Applying AI to a problem domain is generally reasonable if the solution cannot be implemented by an explicit set of rules that can be followed straightforwardly.

Simply put, AI tries to generate knowledge from data without human involvement, whereby the term artificial intelligence represents an umbrella term for the simulation of human intelligence embedded into machines. Machines are therefore able to mimic cognitive tasks typically performed by humans like learning, reasoning, problem solving or natural language processing. As a result of the recent efforts in AI research and development over the past years, self-driving vehicles and chess playing agents are becoming part of our daily lives [Ong17].

2.4.2 Machine Learning

Machine Learning (ML) represents a well-established subdomain of AI, which is considered the most popular subset of AI methods and is focused on building models for classification and prediction tasks [ZCL+19, CLX+21]. ML models are complex algorithms that are able to learn from data to make decisions or forecasts based on previous knowledge and experience. As such, machine learning is tightly intertwined with the field of computational statistics [Ong17].

Typical examples of traditional and well-known machine learning models are Support Vector Machines (SVMs), Decision Trees (DTs), Bayesian networks or k-means clusters. In practice, machine learning models have already been successfully implemented for several use cases, such as image recognition, text classification or even medical diagnoses [ZCL+19, CLX+21]. Apart from using machine learning techniques for developing prediction models, conventional statistical methods, such as time series models like the linear Autoregressive Integrated Moving Average (ARIMA) model or regression models, can also be used for this purpose [WLP+19]. Their suitability and prediction accuracy however depends on the specific application area.

In general, the life cycle of a ML model consists of two main phases, namely model training and inference [DZF+20]. As mentioned earlier, the model training phase generally requires significantly more resources than the actual inference [WBC+19, KHH+20]. During the training phase, sample data is used as training dataset and fed into the ML algorithm so it can learn from it, identify patterns, etc., while the model inference phase represents the actual usage of the pre-trained machine learning model for making predictions. Furthermore, pre-defined workflows or pipelines for establishing ML models encompassing multiple steps and subtasks, such as data preprocessing or model validation, exist [WCW+17]. A simplified but commonly used version of such a ML pipeline is comprised of three main steps, namely data preprocessing, model training and model serving, i.e. model deployment [RRD21]. The machine learning workflow used in the course of this thesis is thoroughly outlined in Section 4.1.

Basically, there are two major approaches to machine learning, namely supervised and unsupervised learning. The main difference between these two concepts is the type of input data they expect. Supervised learning requires the input data to be labeled in order to make predictions about discrete, categorical or continuous values, whereas unsupervised learning relies on unlabeled data and strives to identify patterns within the data set. While classification and regression algorithms represent typical examples for supervised learning, unsupervised learning techniques include clustering and anomaly detection [Raj20].

2.4.3 Deep Learning

Deep Learning (DL) is a special machine learning method, i.e. a machine learning subdomain, that makes use of Artificial Neural Networks (ANNs) in order to solve tasks like image classification or face recognition with high inference accuracy [ZCL+19]. Neural networks are neuroscience-inspired, multi-layered, hierarchical structures that are able to extract a deep data representation, whereby the output of each successive layer serves as the input for the subsequent layer [CLX+21, Ong17]. Different types of neural networks exist, e.g. Convolutional Neural Networks (CNNs), Generative Adversarial Networks (GANs) and Recurrent Neural Networks (RNNs). Typical application areas of deep learning involve natural language processing, speech recognition and computer vision [DZF+20].

Due to its outstanding performance regarding big data processing, forecasting and decision making, deep learning represents one of the most popular AI techniques. As a result, DL models are often considered superior to other machine learning algorithms. However, the high-level precision and efficiency comes at the cost of complexity, which is why deep learning models are typically highly computation- and thus resource-intensive. Therefore, machine learning models generally exhibit a trade-off between inference performance and prediction accuracy, which needs to be taken into account when selecting the right machine learning model for a certain use case [DZF+20, Ong17].

Regarding the ML-based power prediction models that are developed in the course of this thesis, traditional ML algorithms are used instead of deep learning methods due to the high resource intensity and complexity of deep learning models, which presumably makes them impractical for the usage within the *faas-sim* simulation framework. As mentioned earlier, *faas-sim* requires efficient prediction models with high inference speed, so efficient execution and high scalability can be guaranteed.

CHAPTER 3

Related Work

This chapter presents related work on relevant topics in the context of this thesis in order to give an overview of thematically associated existing research efforts. Furthermore, key differences compared to the approach of this work can be identified and demonstrated. Section 3.1 outlines previously published scientific approaches that focus on power prediction models. Subsequently, simulation frameworks in the context of the edge-cloud continuum are described in Section 3.2, particularly with regard to their energy prediction capabilities. Finally, in Section 3.3, energy-aware scheduling and task offloading strategies are considered as broader domains where power prediction models also play a crucial role.

3.1 Power Prediction Models

Due to the emergence of green IoT and green edge computing, energy consumption prediction within the edge-cloud continuum represents an active field of research, including energy efficiency of cloud data centers or power forecasting methods for energy-aware edge computing. However, many of the proposed approaches in the context of energy prediction are trying to predict future energy consumption based on historical data and previously identified usage patterns. As such, various research projects focus on predicting the electricity demand of smart homes [CMPR22], buildings [ZWJ$^+$19], or even whole cities [Oyi21], but not on the application level with regard to hardware utilization-based power modeling of individual devices that run edge-based applications, as intended by this work.

Similar to the approach proposed by Lee et al. [LLKP19], multiple other researchers employ Deep Learning techniques for energy prediction in edge computing environments. To reach this goal, complex algorithms, such as neural networks, are developed. They thereby primarily focus on high prediction accuracy when developing and evaluating their forecasting models and do not consider the applicability of the models in resource-constrained environments or simulation scenarios. Instead, their energy models are mainly designed for more powerful computing platforms, such as edge or cloud servers. Even though prediction precision is an important aspect, it is not the only and predominant design goal of the envisaged work because of the computational burden the resulting models would impose, which would be irreconcilable with the inherent simulator limitations. Therefore, model complexity and suitability have to be considered in addition to high prediction accuracy in the course of this thesis, so more lightweight models can be developed for utilization within the *faas-sim* simulator.

The work presented by Carvalho et al. [CCSF19] includes a power model for mobile devices that is based on a non-linear k-Nearest Neighbors (k-NN) regression algorithm, which represents a well-known machine learning algorithm. Due to the fact that battery-powered mobile devices constitute the target platform for the power model, the authors not only focus on high accuracy, but also on suitability of the algorithm for such devices in terms of execution time. The proposed power model is validated by comparing it with a linear regression model and a neural network-based model. The results of their evaluation show that the k-NN power model exhibits the best trade-off between prediction precision and computation time. Their approach therefore resembles the approach applied in this work, since achieving an optimal accuracy-performance trade-off for power models is also a fundamental part of this thesis.

Shi et al. [SLHM22] also highlight the necessity for considering the complexity-accuracy trade-off in the algorithm design phase, since their work focuses on edge computing scenarios, where devices with restricted resources are used. The developed lightweight forecasting model is thus optimized for minimized complexity as it is implemented on a Raspberry Pi. The feasibility and suitability of the algorithm for edge devices is verified afterwards. Although the complexity-accuracy trade-off of forecasting models is also a key aspect of this thesis, the usage of deep learning methods as applied by Shi et al. is not anticipated. Additionally, the research domain of the proposed approach differs as it is based on photovoltaic-assisted charging stations for electric vehicles.

Rodrigues et al. [RRL18] developed a framework for energy measurement and prediction in order to determine and forecast the power usage of deep neural networks that run on ARM-based mobile platforms. As such, the resulting framework called SyNERGY operates on the device level, as planned for this thesis, but further breaks down the energy consumption to the separate layers of the neural network models that are investigated. For the actual power consumption forecasting of the individual neural network layers, a multi-variable linear regression model is used, which considers device-specific hardware performance counters, namely Single Instruction/Multiple Data (SIMD) instructions and bus accesses, i.e. main memory accesses. Even though the proposed approach is similar to the one applied in this thesis, it varies in terms of prediction granularity as well as the targeted applications, i.e. deep learning applications.

To sum up, the key differences between the related work presented above and the envisaged approach of this thesis lie in the usage of deep learning methods, i.e. neural networks, for developing power prediction models and the underlying scope of application. Some of the proposed models are developed for very specific application domains, whereas the applicability of the power models established in this work should not be limited to a specific use case but rather be generally usable for various different programs and scenarios.

3.2 Simulation Frameworks for the Edge-cloud Continuum

As the power prediction functionality is integrated into an existing serverless simulation platform for edge computing systems, other simulation frameworks in the context of the edge-cloud continuum are also considered as relevant related work. The existence of energy models within other simulators is thereby of particular interest.

The CloudSim [CRB+11] is an extensible simulation framework for cloud computing environments that is able to model and simulate cloud components such as cloud data centers and Virtal Machines (VMs). Furthermore, it enables the evaluation of different provisioning policies in terms of resources, VMs and applications by simulating various allocation strategies. This also allows for testing energy-conscious resource management techniques, since the CloudSim simulation toolkit includes basic power consumption models for cloud system components. These models were built upon the assumption that the total power consumption consists of a static share, i.e. a constant fraction, and a dynamic share, whereby the dynamic part represents a function that computes the consumption in proportion to the current CPU utilization. Therefore, this approach is considered rather simple and rigid as it is solely based on the CPU utilization without taking other resource metrics into account.

The iFogSim [GVDGB17] and its extension, iFogSim2 [MPGB22], are both simulation frameworks for IoT, edge and fog environments. The initial iFogSim simulator version is based upon CloudSim and therefore already incorporates a power prediction functionality for cloud centers and fog nodes. The resource utilization of each device is thereby monitored and used as the input for pre-defined power models that are able to predict the power consumption of each device at the end of the simulation. Like the models used in the CloudSim, the available iFogSim power models focus on the CPU usage in order to calculate the electricity consumption and use linear, square or cubic formulae, among others. The presented approach is in principle similar to the one intended in the course of this thesis when it comes to incorporating power models into a simulator for providing an energy consumption prediction at the end of a simulation. But instead of using rather simple statistical functions, that calculate the energy consumption based on the CPU utilization, this work aims at developing more elaborate models using machine learning techniques.

Besides iFogSim, EdgeCloudSim [SOE18] and IoTSim-Edge [JAA+20] are also extensions of the CloudSim simulator toolkit, which both encompass the simulation of IoT and edge computing scenarios. While the EdgeCloudSim framework does not include an energy consumption model in its initial version, the IoTSim-Edge simulator incorporates an energy calculation feature in terms of battery consumption of portable IoT and edge devices. The power consumption of the battery is calculated by estimating the energy necessary for data processing, while also including the size of the data that needs to be processed, and adding the estimated energy required for the transfer of the data, which also depends on the transmission protocol that is used. As such, this represents a more detailed energy consumption prediction approach as compared to the previously described ones.

IoTSim-Osmosis [AJH$^+$21] is another simulator framework that enables the simulation of IoT applications within the edge-cloud continuum and focuses on osmotic computing scenarios. Regarding the power consumption of the IoT layer, IoTSim-Osmosis is able to estimate the battery draining of IoT devices, which is composed of the draining rate required for sensor measurements and data transmission, respectively. The battery consumption of the individual IoT devices is then updated upon every sensing of the environment. Furthermore, the framework also provides means to estimate the energy consumption of the edge and cloud layer, but the corresponding scientific work [AJH$^+$21] does not specify the details for these estimations. Their specific approach concerning the power consumption of edge devices is therefore not clearly evident from the published work.

The LEAF simulator [WT21] represents a simulator for large-scale energy-aware fog computing environments that was developed with a focus on realistic simulations, holistic and granular energy modeling, support for energy-aware decision making, as well as simulation performance and scalability. Using the LEAF simulator, each compute node and each network link is assigned an individual power model, which allows the separate assessment of the power usage of edge devices, data centers and network links. The current power consumption of an entity is composed of a static, i.e. load-independent, and a dynamic, i.e. load-dependent, fraction, whereby the static part can also take energy-saving mechanisms such as Dynamic Voltage and Frequency Scaling (DVFS) into account. Depending on the type of entity, the authors apply linear or non-linear power models. The applicability of the LEAF simulator for simulating the energy consumption of heterogeneous, distributed environments with resource-constrained devices and for evaluating energy-aware task placement strategies or energy-saving mechanisms are demonstrated using a smart city traffic scenario.

In summary, multiple simulation frameworks for the edge-cloud continuum already exist. While some of them exhibit a lack of energy modeling mechanisms, many of the existing simulators include approaches for estimating the power consumption of the devices used during the simulation process. However, the level of comprehensiveness and maturity of the implemented solutions highly differs, whereby the LEAF simulator [WT21] is assumed to provide the most extensive energy modeling techniques. Hence, not all of the proposed approaches are considered adequate in order to obtain realistic results, since some of the proposed energy models only implement limited and partly rigid forecasting methods, rely on simplifications and assumptions or neglect the importance of hardware heterogeneity. The aim of this thesis is thus to develop more elaborate and comprehensive ML-based models that incorporate various influencing factors in addition to CPU utilization and generate satisfactory prediction results.

3.3 Energy-aware Resource Management Strategies

Other domains where energy prediction capabilities play a major role are energy-aware scheduling algorithms, routing policies and other resource management strategies, which aim at optimizing a system's power consumption by using intelligent placement algorithms that minimize the total energy demand of a system. As mentioned earlier, the final power consumption prediction models that are developed in the course of this thesis could also be used as the basis for such energy-aware mechanisms in the future.

Neurosurgeon [KHG$^+$17] represents a dynamic and lightweight scheduler that offers computation partitioning for the individual layers of Deep Neural Networks (DNNs) for the sake of low end-to-end latency or low mobile device energy consumption. Therefore, the Neurosurgeon system is able to determine the optimal partition point for the collaborative execution of a DNN in order to distribute the computation among cloud data centers and mobile devices. A partition point can be located after each neural network layer and represents the delimitation of layers, whereby the one portion is executed on the mobile devices and the other one on the cloud servers. The partitioning depends on the primary optimization goal, which can either be low end-to-end latency or low mobile device energy consumption. The approach is similar for both of the competing goals, i.e. the latency and energy consumption required for the computation of each DNN layer is estimated by predefined prediction models and then the optimal partition point is chosen. For developing the prediction models, the authors profile a state-of-the-art mobile device, namely an Nvidia Jetson TK1, and a modern server platform and establish a regression model for each layer type and on each hardware platform based on the layer's configuration parameters. As a consequence, the prediction models are platform-specific, but at the same time, they are reusable for different neural network architectures. The improvements in terms of latency and energy consumption of the Neurosurgeon scheduler are verified using an evaluation suite comprising eight DNN applications.

Ale et al. [AZF$^+$21] propose a delay-aware and energy-efficient computation offloading method for mobile edge computing networks that makes use of Deep Reinforcement Learning (DRL) techniques. Their approach is able to optimize the number of completed tasks in order to comply with delay constraints, while the energy consumption is minimized at the same time due to the intelligent offloading mechanism. The energy consumption is estimated by calculating the power used for data transmission based on the size of the data, the transmission rate and the transmission power, and adding up the power used for computation, which depends on the required CPU cycles for a task and the CPU frequency. By using a reinforcement learning framework, the system is able to learn from previous offloading decisions and can thereby optimize itself for future decisions. Besides the ability to determine the optimal edge server for computation offloading, the system is also capable of optimizing the computational resource allocation of edge nodes in order to improve the long-time utilization of the system. The effectiveness of the presented solution is demonstrated by a simulation analysis.

Furthermore, the work presented by Liu et al. [LCH+18] includes an energy-aware resource allocation scheme for edge networks, called On-demand Energy-efficient Resource Allocation (OERA), that aims at minimizing the energy consumption on the network and the device level. Therefore, a so-called Network Device Power Model (NDPM) based on empirical measurements is constructed, which serves as the initial groundwork for the OERA scheme. In order to establish the power model for the network devices, i.e. routers and switches, a testbed is utilized for obtaining the actual power consumption data of the devices. For estimating the power drainage of a network device, the authors consider the energy consumption of the device itself and the corresponding network interface card that is used to connect the device to the network. Thereby, the energy consumption of the device is composed of the power required for transmission, reception and packet processing, respectively. Additionally, the model incorporates the optional usage of frequency scaling for the network devices as energy saving technique. The final OERA algorithm, which is developed using Mixed Integer Linear Programming (MILP), is compared with two other existing algorithms based on three performance metrics, namely acceptance ratio, total edge network power consumption as well as host and link utilization. The performance evaluation shows that OERA outperforms the other two algorithms in all three factors.

In conclusion, power models that are able to predict the power consumption of devices can be applied to other problem domains, such as scheduling strategies, routing policies or other resource management algorithms, in order to enable energy-aware decision making and optimize such algorithms for reduced energy consumption.

CHAPTER 4

Methodology

The following sections thoroughly describe the methodology applied in the course of this thesis to develop machine learning models that are able to predict the power consumption of edge devices based on resource usage values. Section 4.1 therefore outlines the general methodological approach as well as the model development process. Subsequently, Section 4.2 presents the experimental design and setup for the empirical measurements conducted for the sake of data acquisition. This includes the setup of the dedicated testbed used for the experiments, the devices being profiled, the system metrics being measured, the measurement instruments used for obtaining the power consumption of the devices and the experimental procedure including the individual experiment configurations. After the experiments are carried out, the empirical measurement data can be used for data preprocessing and a preliminary, exploratory data analysis, as detailed in Section 4.3, where key observations are highlighted. Finally, Section 4.4 focuses on the construction of the envisaged machine learning models by means of automated machine learning tools such as TPOT. This section concludes the chapter by presenting the results of the model development procedure, i.e. the final machine learning models, which represent the main contribution of this work.

4.1 Methodological Approach

4.1.1 Overview

As the objective of this work is to develop machine learning models that are able to predict the power consumption of edge devices based on resource usage, a data-driven, empirical approach based on a series of experiments is used as methodology. Ultimately, the final outcome is a software artifact that extends the existing *faas-sim* simulation framework by the additional forecasting capability. Therefore, an experimental method is considered as an appropriate scientific test and verification procedure for this purpose.

In order to give an overview over the conceptual design of the methodological process, Figure 4.1 reflects the individual steps of the empirical approach that is applied in this thesis.

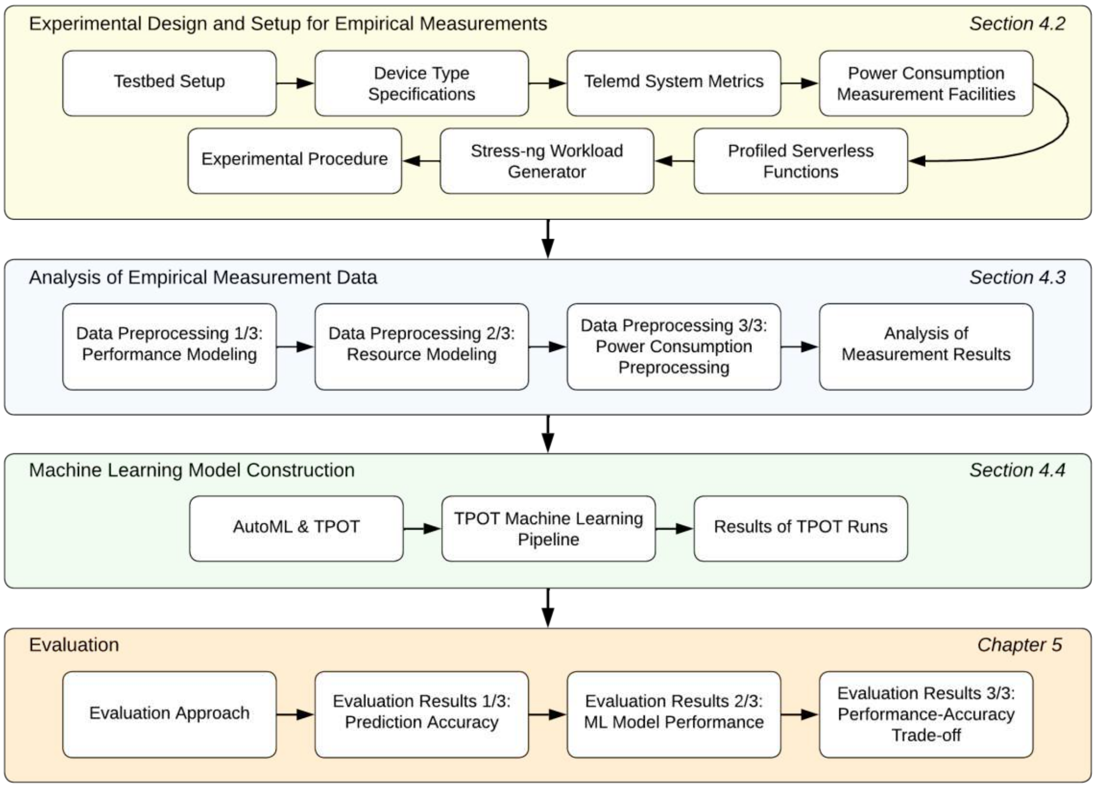

Experimental Design and Setup for Empirical Measurements
Section 4.2
Testbed Setup
Device Type Specifications
Telemd System Metrics
Power Consumption Measurement Facilities
Profiled Serverless Functions
Stress-ng Workload Generator
Experimental Procedure
Analysis of Empirical Measurement Data
Section 4.3
Data Preprocessing 1/3: Performance Modeling
Data Preprocessing 2/3: Resource Modeling
Data Preprocessing 3/3: Power Consumption Preprocessing
Analysis of Measurement Results
Machine Learning Model Construction
Section 4.4
AutoML & TPOT
TPOT Machine Learning Pipeline
Results of TPOT Runs
Evaluation
Chapter 5
Evaluation Approach
Evaluation Results 1/3: Prediction Accuracy
Evaluation Results 2/3: ML Model Performance
Evaluation Results 3/3: Performance-Accuracy Trade-off

Figure 4.1: Methodology of the empirical approach.

First of all, a series of experiments including real-time measurements is conducted for gathering power consumption and resource usage data from different edge device types. This requires a sophisticated design and setup of the individual experiment components, such as the testbed used for the experiments, the edge devices to be profiled, the resource usage metrics to be reported, the means of power consumption monitoring and so forth. All of these experiment components are thoroughly described in Section 4.2. The raw empirically obtained data is then preprocessed for easier interpretation and for usage within the *faas-sim* simulation framework, as explained in Section 4.3. Furthermore, the preprocessed data serves as the basis for the development of predictive models using data-driven machine learning techniques. The individual steps required for the construction of these models are detailed in Section 4.4. Since the developed models need to be validated, a profound evaluation in terms of model accuracy and performance needs to be performed, which is covered in Chapter 5. Finally, the validated power models can be provided as an additional simulator feature for public usage. While the first three phases of the experimental approach, namely the experimental design and setup for the empirical measurements, the analysis of the empirical measurement data and the machine learning model construction, are detailed in the following sections, the evaluation of the developed models is covered in the subsequent chapter, see Chapter 5.

4.1.2 Model Development Process

The development of the machine learning models follows a specific, predefined process. However, the individual steps of this process can also be mapped to the methodological approach outlined in Figure 4.1. The model development process used in this work, which is derived from the well-known and established CRoss-Industry Standard Process for Data Mining (CRISP-DM) and based on the iterative processes presented in [BB21] and [WCW^{+}17], respectively, involves several steps. These steps include *Data Preparation*, *Data Analysis*, *Model Construction*, *Model Validation* and *Model Delivery*, as depicted in Figure 4.2. If necessary, steps 2–4 can be iterated in order to obtain an enhanced prediction model.

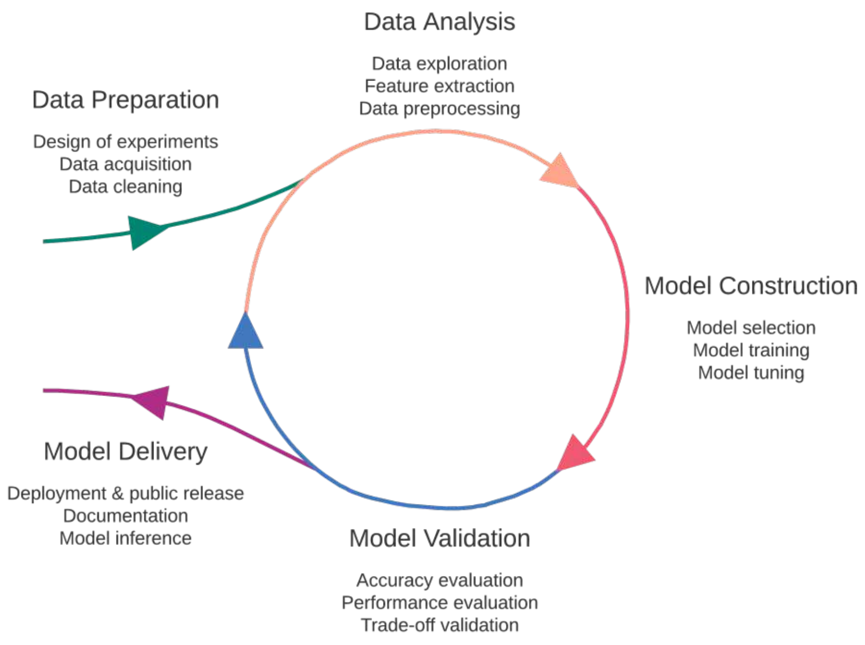


Figure 4.2: The model development process based on [BB21]

Data Preparation As a first step, power consumption and resource usage data is collected through a series of experiments that are conducted by means of a dedicated testbed, where different edge device types are profiled at runtime. This way, empirical real-time measurements can be obtained. During the experiments, the actual power consumption of the individual hardware platforms that run certain applications will be measured in combination with other resource utilization parameters that conceivably have an impact on the energy usage. These metrics might include CPU utilization, GPU utilization, network I/O rate, memory (RAM) usage, etc. The runtime monitoring of edge devices during the experiments is achieved by collecting black-box system metrics using monitoring agents and by monitoring the power consumption of the hardware platforms accordingly. The ensuing data cleaning step might be required in order to ensure a good data quality, e.g. if measured values are incomplete, corrupt or incorrect.

Data Analysis The subsequent data analysis step encompasses several tasks, namely data exploration, feature extraction and data preprocessing. An exploratory data analysis will take place in order to better understand the structure of the data and to be able to identify the relevant variables, so-called features, for the upcoming model construction phase. Feature extraction focuses on identifying and extracting the appropriate, i.e. the most correlated, factors that have the greatest effect on the power consumption of the platforms, while the irrelevant ones should be disregarded. Furthermore, the raw measurement data needs to be preprocessed before it can be fed into the forecasting models, since the simulator can only model the resource consumption of edge devices through estimates and therefore the actual resource utilization values cannot be used as input for the prediction models. So instead of continuously measuring the actual resource consumption and calculating the power usage based on the measured values, as it would be possible in the real world, the simulator has to estimate and model the resource utilization of the individual devices during the simulation in some way. Therefore, the measured values have to be mapped to inputs the simulator can work with in order to be able to serve as training data for the ML models. By preprocessing the raw data, the inherent discrepancy of resource consumption between the real world and the simulation environment can be taken into account in the model development step, so more accurate results can be achieved.

Model Construction After the data analysis step, the empirically obtained and already preprocessed data serves as the basis for a data science-driven approach, where machine learning techniques are applied to construct a power consumption prediction model. For this purpose, an Automated Machine Learning (AutoML) tool will be used, which enables automatized and easy development of high quality ML models, instead of manually building, training and tuning a prediction model [TWG+19]. Hence, the model creation process can be accelerated by using an AutoML tool. In the context of this work, the Tree-based Pipeline Optimization Tool (TPOT) is applied, which is an AutoML Python library that aims at automating ML pipelines and optimizing the model accuracy on the basis of genetic programming [OM16]. As such, TPOT is available as an open-source Python library[1]. Further details about TPOT in general and the specific TPOT configuration used in the course of this work in order to construct the predictive models are included in Section 4.4.1.

[1]`https://github.com/EpistasisLab/tpot`, Accessed: Jan 30, 2023

Model Validation As a subsequent step, the resulting models have to be programmatically integrated into the existing *faas-sim* simulator platform in order to facilitate the evaluation of the models with respect to prediction accuracy and performance. This way, the forecasting precision can be investigated by comparing the predicted power consumption from the simulation with the actual power consumption measured by means of experiments within the dedicated testbed. Furthermore, the performance can be analyzed by determining the degree of latency the models impose on the simulator execution time as a result of the additional computational burden caused by the model complexity. As the model performance also influences the simulator scalability, the results of the performance evaluation can also indicate whether the developed models are suitable for an integration into the existing simulation framework or not. Finally, the resulting performance-accuracy trade-off needs to be assessed and weighed out. If a model turns out to generate imprecise predictions or does not perform well in terms of execution time, the previous three steps, namely data understanding, model creation and model validation, can be iterated as a result of feedback loops, so an enhanced model can be obtained.

Model Delivery As a final step, the validated forecasting models are deployed and made publicly available for developers and users as part of the open source *faas-sim* simulation framework. The models can then be used for inference, i.e. for estimating the power consumption of devices during simulations as an additional simulator feature. This represents the conclusive step of the process.

4.2 Experimental Design and Setup for Empirical Measurements

4.2.1 Testbed Setup

As described in the previous section, the empirical measurements represent the data preparation step, i.e. the first step, of the model development process. It includes the design of the experiments, the actual execution of the experiments for data acquisition and a successive data cleaning step if necessary. The empirical measurements are required as input data for the construction of the machine learning models and are conducted by means of a dedicated testbed. For this purpose, an existing testbed established by the Distributed System Group of the TU Wien is used. The present testbed is already equipped with an open source experimentation framework called Galileo.

The Galileo experimentation framework[2,3] enables users to define and execute experiments on the testbed. As such, the framework is targeted at distributed load testing experiments and enables users to observe the resource usage and application performance, to perform profiling tasks and to evaluate certain cluster components like the load balancer or

[2]https://github.com/edgerun/galileo, Accessed: Oct 3, 2022
[3]https://github.com/edgerun/galileo-experiments, Accessed: Oct 3, 2022

scheduler. Furthermore, it aims at easing experiment setup and deployment by providing different building blocks, for example configurable workload generation, telemetry data collection, trace recording and a container orchestration adaption [RRP+22]. Telemetry data represent metrics for system observation, e.g. resource utilization measurements, whereas traces refer to program execution logs that record various system actions and are triggered through function calls, i.e. user requests. Trace recording includes several function invocation-related data, such as various timestamps of a function invocation lifecycle and the total function execution time. For the usage in edge environments, all of the experiment components of the Galileo framework can run on resource-constrained devices. Details about the telemetry data collection are covered in Section 4.2.3, while the specific workloads used for the experiments are contained in Section 4.2.7.

In the context of the Galileo framework, applications are submitted and deployed as serverless functions via Kubernetes because the testbed is based on OpenFaaS and uses a Kubernetes cluster as OpenFaaS runtime as well as container orchestrator. In terms of Kubernetes, the smallest deployment unit is called a Pod, whereby one or more containers can run inside a single Pod[4]. The functions that are deployed as part of the experiments are described in Section 4.2.5.

In order to reflect the hardware heterogeneity of edge environments, nodes with different underlying architectures are included in the testbed setup. The testbed is divided into three zones, i.e. clusters, namely zone A, B and C, whereby each node is statically assigned to one of these zones. While zone A and B primarily encompass commonly used edge devices, zone C represents the cloud and therefore comprises virtual machines only. Zone A and B both include a client node, i.e. an Intel NUC i7, and multiple worker nodes, i.e. different types of edge computing platforms, for the execution of distributed load testing experiments by means of the Galileo framework. All nodes in the testbed are remotely accessible via SSH through the TU Wien VPN. As the focus of this work lies on edge devices, only nodes in zone A and B are used for the experiments.

In general, the Galileo experiment framework supports two kinds of experiments, namely profiling and scenario experiments [RRP+22]. *Profiling experiments*, which are used in this work, aim at profiling the resource usage and performance of one particular node that runs one type of application. This also enables cross-device evaluations of individual applications. *Scenario experiments* are targeted at large-scale experiments, e.g. to evaluate the system performance for different resource management strategies. The workloads, i.e. the client request patterns, for an experiment can either be generated by existing random probabilistic interarrival time generators or prerecorded arrival time profiles, such as a constant or a sine-based profile, can be used. Additionally, manually defining the number of requests and the corresponding interarrival time is also possible.

[4] `https://kubernetes.io/docs/concepts/workloads/pods/`, Accessed: Jan 30, 2023

4.2.2 Device Type Specifications

Since different types of common edge computing platforms are being profiled, the heterogeneous hardware characteristics of the individual devices need to be outlined. Three device types with diverse architectures are chosen from the pool of nodes included in the existing testbed that is used for the empirical measurements. Table 4.1 provides details about the hardware specifications of the selected edge devices. These platforms include two types of GPU-enabled Nvidia Jetson boards[5] with different underlying physical components and an Intel Xeon PC equipped with an Nvidia GPU[6,7]. From the device characteristics displayed in Table 4.1, it can be concluded that the Intel Xeon is the most powerful edge device in terms of hardware equipment, whereas the Jetson Nano is the least powerful one in this sense. Besides the heterogeneous hardware specifications, all of these computing platforms can be considered as high-performance, AI-enabled edge devices in general.

Device	CPU	Accelerator	RAM
Intel Xeon	Quad-core Xeon E-2224 CPU	1408-core Nvidia Turing GPU (GeForce GTX 1660)	16 GB
Jetson Xavier NX	6-core Nvidia Carmel Arm v8.2 CPU	384-core Nvidia Volta GPU with 48 tensor cores	8 GB
Jetson Nano	Quad-core Arm Cortex-A57 MPCore processor	128-core Nvidia Maxwell GPU	4 GB

Table 4.1: Device type specifications.

In order to be able to uniquely identify the different hardware platforms within the testbed, hostnames are assigned to the individual nodes. The hostnames typically include the zone the node resides in. Since the telemetry and power monitoring data make use of the hostname instead of the device type for the sake of unambiguity, the following table, i.e. Table 4.2 provides a mapping from devices to hostnames. As shown in this table, the Xeon GPU node is located in zone B, while the other nodes reside in zone A of the testbed.

4.2.3 Telemd System Metrics

For the purpose of identifying the factors that have a decisive impact on the power consumption of edge devices, various system metrics regarding resource usage are measured in addition to the power consumption during the execution of the experiments. Therefore,

[5] `https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/`, Accessed: Sept 29, 2022

[6] `https://www.intel.com/content/www/us/en/products/sku/191036/intel-xeon-e2224-processor-8m-cache-3-40-ghz/specifications.html`, Accessed: Oct 7, 2022

[7] `https://www.nvidia.com/en-us/geforce/graphics-cards/16-series/`, Accessed: Oct 7, 2022

Device	Hostname
Intel Xeon	eb-b-xeongpu-0
Jetson Xavier NX	eb-a-jetson-nx-0
Jetson Nano	eb-a-jetson-nano-0

Table 4.2: Device to hostname mapping within the testbed.

telemetry data from each node of the testbed can be collected by observing the platforms at runtime by means of telemd[8], which is part of the Galileo experimental framework and represents an open-source black-box monitoring agent for gathering time-series system metrics. The telemd daemon runs on all nodes of the testbed and is thus able to capture system-wide fine-grained telemetry data. Telemd reports the measured values into different Redis topics depending on the individual metrics, whereby Redis acts as a publish-subscribe system and the persistent storage of measured data is accomplished by the open-source time series database InfluxDB. The Redis topics for telemetry data follow a predefined scheme, namely `telem/<nodename>/<metric>[/<subsystem>]`. As an example, the topic `telem/eb-b-xeongpu-0/cpu/0` is used for reporting the *CPU utilization* metric of the subsystem *CPU core 0* of the host with node name *eb-b-xeongpu-0*. Per default, the sampling frequency of all telemd telemetry values is one second, but it can also be modified if desired.

Telemd Metric	Description
`kubernetes_cgrp_cpu`	Total CPU usage time in nanoseconds of individual containers inside Kubernetes Pods
`kubernetes_cgrp_blkio`	Total block I/O usage in bytes of individual containers inside Kubernetes Pods
`kubernetes_cgrp_net`	Total network I/O usage in bytes of individual containers inside Kubernetes Pods
`kubernetes_cgrp_memory`	Current memory (RAM) usage in bytes of individual containers inside Kubernetes Pods
`gpu_util`	Overall GPU utilization in percent (differentiation between common Nvidia GPUs and Jetson boards)
`gpu_power`	Overall GPU power usage in milliwatts (only available on common Nvidia GPUs)

Table 4.3: Relevant resource utilization metrics reported by telemd.

[8]`https://github.com/edgerun/telemd`, Accessed: Sept 30, 2022

The resource utilization metrics recorded by telemd[9] that are relevant for this work are listed in Table 4.3, whereby the default sampling frequency of one second is maintained, so each metric is reported once per second. The `gpu_power` metric is additionally added to the telemd monitoring agent as part of this work, which is thoroughly described in Section 4.2.4. In general, telemd reports system-wide runtime data on the node level and therefore aggregates the measured values, e.g. the total CPU utilization of all CPU cores of a system, but it may also obtain metrics of individual subsystems such as a specific network device, disk or CPU core. Regarding the system-wide GPU utilization metric, i.e. `gpu_util`, it is important to note that the approach to obtaining the relevant values differs between common Nvidia GPUs and Jetson boards, which is why the definition of GPU utilization or GPU load also differs depending on the platform[10]. The interpretation of the specific GPU utilization values therefore depends on the underlying architecture. On common Nvidia GPUs, i.e. amd64 systems such as the Intel Xeon PC, the GPU utilization is defined as the percentage of time during which the GPU was used within the elapsed sample period[11]. In contrast, the tegrastats utility of Nvidia Jetson-based development boards reports the percentage of the GPU that is currently used with respect to the current GPU frequency[12].

Apart from system-wide metrics, telemd can additionally collect certain resource utilization measurements for individual Docker containers and for containers inside Kubernetes Pods running on testbed nodes and thereby enables fine-grained monitoring on the container level[13]. This is done by so-called cgroup metrics, whereby control groups (cgroups) are hierarchically organized collections of processes, which can be isolated, restricted and monitored in terms of resource usage[14,15]. These cgroup metrics include CPU usage time, total block I/O usage, total network I/O usage and memory, i.e. RAM, usage. Besides the memory (RAM) usage, i.e. `kubernetes_cgrp_memory`, all other cgroup parameters reported by telemd, which can be identified by the prefix `kubernetes_cgrp_`, represent continuous counters instead of current utilization values. For example, the `kubernetes_cgrp_cpu` measurements published at a certain point in time indicate the total CPU usage time of individual containers running inside Kubernetes Pods up until the time of measurement, not the usage time since the last measurement and not the current CPU utilization. This differentiation between the cgroup metrics and the system-wide metrics, such as the GPU utilization, has to be considered when interpreting the measured values.

[9]`https://github.com/edgerun/telemd/tree/gpu-support`, Accessed: Jan 14, 2023
[10]`https://github.com/edgerun/telemd/tree/gpu-support`, Accessed: Jan 14, 2023
[11]`https://docs.nvidia.com/deploy/nvml-api/structnvmlUtilization__t.html#structnvmlUtilization__t`, Accessed: Jan 14, 2023
[12]`https://docs.nvidia.com/jetson/archives/r34.1/DeveloperGuide/text/AT/JetsonLinuxDevelopmentTools/TegrastatsUtility.html`, Accessed: Jan 14, 2023
[13]`https://github.com/edgerun/telemd#instruments`, Accessed: Jan 14, 2023
[14]`https://man7.org/linux/man-pages/man7/cgroups.7.html`, Accessed: Jan 14, 2023
[15]`https://docs.docker.com/config/containers/runmetrics/#control-groups`, Accessed: Jan 14, 2023

4.2.4 Power Consumption Measurement Facilities

In addition to the telemetry data that is automatically obtained by the experimental framework, power consumption data need to be captured on the individual devices during the experiments as well, which is not supported by the testbed out of the box. Therefore, power consumption measurement facilities need to be incorporated into the existing testbed setup.

Measuring the power consumption of edge computing platforms is a challenging task due to the hardware platform heterogeneity that is associated with the edge domain. Nonetheless, accurate power measurements are crucial for optimizing the power demand and energy efficiency of computer system components and for developing energy-efficient software artifacts, which is why the energy demand represents an essential design criteria in modern computing systems. Especially battery-operated devices, such as mobile platforms, wireless sensors or wearables, require low power and high energy efficiency in order to maximize the battery lifetime and therefore the operating time. Even though the power usage of computing systems is a critical factor, there is a lack of standardized and unified methods or tools for measuring the power consumption of devices in a platform-agnostic way. Therefore, a wide variety of measurement facilities exists and in many cases various techniques need to be applied in order to determine the power consumption of different device types with diverse hardware architectures. This makes power measurements on heterogeneous platforms complex and laborious [KHH$^+$20, GCZY21].

In general, there are two different types of measurement facilities for retrieving the power consumption of devices, namely on-board, i.e. internal, and external methods. These methods differ in terms of granularity, measuring approach, power data source, overhead, frequency, setup costs and equipment costs. On-board techniques encompass power monitoring mechanisms that are integrated into the hardware platform, such as internal power sensors or monitors. The power values of these on-board facilities can then be obtained using software. Such internal power readings often include fine-grained power consumption breakdowns for individual subsystems such as CPU, GPU or memory, which enables a thorough inspection of the system's power demand based on multiple components. As such, on-board measurement instruments typically provide accurate results at a high level of granularity. These methods however impose a certain, potentially negligible but still present, overhead, since the internal power values need to be retrieved by the system itself through software. The specific interfaces that can be used to read the power values and their sampling frequency also differ among devices and are therefore highly platform-dependent [KHH$^+$20]. Hence, the equipment costs are low, since no external power measurement instruments are needed, but the setup costs are rather high, because each device might require a different setup depending on the specific internal power measurement techniques integrated into the systems, which can be time-consuming and laborious. Furthermore, some hardware platforms, such as Google's Coral Dev Board, do not provide any means to obtain the current power draw through internal power monitoring mechanisms, so these techniques generally cannot be applied to all devices.

External power measurement facilities require auxiliary devices that are connected to the platform to be measured and can thereby determine and monitor their power consumption. These measuring instruments include Power Distribution Units (PDUs) and USB sticks with power metering capabilities, as well as power monitors that intercept the power supply of a device and meter the power consumed by the corresponding power outlet or battery, such as smart plugs. Since these external power meters can report energy consumption data without relying on software that runs on the device under test, they do not impose a overhead in this regard. As a result, external power sampling tools can be attached to multiple hardware architectures and can therefore foster uniform, platform-agnostic power measurements. Using auxiliary measuring instruments involve reasonable setup costs, since the setup is identical for all devices, but might come along with high equipment costs depending on the type of instrument used. The sampling frequency of power meters can also be restricted, so they might not be applicable to certain use cases where a high sampling frequency is crucial. Additionally, they only provide coarse-grained power values at the device level as compared to on-board monitoring mechanisms, which facilitate fine-grained measurements at component level [KHH+20, GCZY21].

Implementation of Power Monitoring in this Work

In order to be able to obtain the power readings of all edge device types in a uniform and comparable way, external power measuring instruments are used. Specifically, each platform listed in Table 4.1 is connected to a smart plug[16], which uses an open source software called Tasmota[17] to publish the real-time energy values in a predefined interval via the MQTT[18] messaging protocol. In the course of this work, Tasmota version 7.2.0 is used for all smart plugs. In order to store the power measurements gathered by the smart plugs in a database, two MQTT clients are set up using the Eclipse Paho MQTT Python client library[19], i.e. one on the eb-a-controller node for zone A and one on the eb-b-controller for zone B, that both connect to an Eclipse Mosquitto MQTT broker[20] of their respective zone and publish the power readings reported by the smart plugs to dedicated Redis channels. Redis in turn automatically forwards the energy values to InfluxDB, an open-source time series database, where all other system metrics obtained by the telemd monitoring agent[21] are also stored, as described in Section 4.2.3. This way, all measured metrics are consistently recorded in one database, which eases later data analysis steps. The source code for the MQTT client scripts can be found in Appendix A.

The smart plugs used in this work are able to report different energy metrics. The metrics that are of particular relevance for this thesis are the measured voltage and current, since the power can be calculated using Ohm's law and the corresponding formula for power

[16]`https://www.reichelt.com/de/en/wifi-outlet-switch-power-measurement-delock-11827-p262109.html?r=1`, Accessed: Nov 28, 2022
[17]`https://github.com/arendst/Tasmota`, Accessed: Nov 28, 2022
[18]`https://mqtt.org/`, Accessed: Nov 28, 2022
[19]`https://pypi.org/project/paho-mqtt/`, Accessed: Jan 13, 2023
[20]`https://mosquitto.org/`, Accessed: Jan 13, 2023
[21]`https://github.com/edgerun/telemd`, Accessed: Nov 28, 2022

calculation, i.e. $Power\,(watts) = Voltage\,(volts) \cdot Current\,(amperes)$. As the internal power monitors obtain the power consumption in milliwatts, the calculated power is also converted from watts (W) to milliwatts (mW) in order to facilitate comparability among the different measurement facilities. Apart from these values, other energy data such as apparent power or reactive power are also measured and published by the smart plugs.

While the Galileo framework generally reports telemetry metrics once per second, the shortest telemetry period supported by Tasmota is 10 seconds. In order to bridge this gap and receive more frequent updates, the PowerDelta[22] configuration of Tasmota is used, which enables the smart plugs to immediately report on power changes via MQTT telemetry messages that are emitted in addition to the frequent updates defined by the telemetry period. Therefore, the maximum delta can be set to report on percentage power change, e.g. 1%, or on absolute power change, e.g. 1 W. In terms of the energy monitoring devices used in this work, the maximum delta of the PowerDelta configuration of all smart plugs is set to 1%. This way, it can be assumed that the power consumption changed less than 1% when no update is issued, which is negligible.

In addition to the external power meters that are connected to each edge device, the Jetson boards are monitored using an internal, on-board power measurement mechanism in order to be able to compare the software-based power readings with the power values reported by the external monitoring instruments. This way, the extent of deviations between these two inherently different measurement facilities can be determined. The two Nvidia Jetson boards specified in Table 4.1 can be measured uniformly using their integrated power monitors. For this purpose, a Python daemon script is developed for retrieving the power values of the boards using the `jetson-stats` Python package[23,24], which can be applied for monitoring and controlling Nvidia Jetson boards. Internally, the `jetson-stats` package includes jtop, which is a system monitoring utility that can access the real-time status of Nvidia Jetson boards, such as CPU, GPU, disk and fan status, as well as power stats[25]. Using the `jetson-stats` package, it is therefore possible to read the current power consumption of a Jetson board in milliwatts, among other internal values. The sampling granularity for the power readings in the Python daemon script is set to one second as default value, but it can be modified as desired. The script, which is contained in Appendix A, is then executed on every Jetson board, whereby the power consumption overhead of running this program is considered rather small and can thus be neglected.

Regarding the Intel Xeon PC equipped with an Nvidia GPU, the total power usage is also measured by the smart plugs as described above. Additionally, similar to the Jetson boards, internal power measurements of the Intel Xeon are also envisaged. For this purpose, the Nvidia Management Library (NVML) library[26] is applied, which provides

[22]`https://tasmota.github.io/docs/Commands/#power-monitoring`, Accessed: Dec 14, 2022
[23]`https://github.com/rbonghi/jetson_stats`, Accessed: Oct 3, 2022
[24]`https://pypi.org/project/jetson-stats/`, Accessed: Oct 3, 2022
[25]`https://github.com/rbonghi/jetson_stats#jtop`, Accessed: Oct 3, 2022
[26]`https://developer.nvidia.com/nvidia-management-library-nvml`, Accessed: Nov 28, 2022

means to monitor and manage Nvidia GPU devices. NVML thereby includes information about the power usage of integrated GPUs of the Nvidia Tesla product line. As the GPUs embedded into the Jetson boards do not belong to this product category, this feature is only available for the Intel Xeon GPU. In order to be able to obtain the power consumption of the GPU, a C script that retrieves the internal power usage of the GPU and the connected circuitry such as memory by means of NVML[27] is developed. As mentioned above, this GPU power reading functionality is integrated into the telemd monitoring agent as an additional metric, since telemd does not include any power consumption measurements. Analogous to the jtop script, the source code of the developed C script is also attached in the appendix, namely in Appendix A.

To sum up, Table 4.4 provides an overview over the specific power measurement mechanisms that are applied to the individual edge devices.

Device	**Internal**	**External**
Intel Xeon	Total power: ✗ GPU power: ✓	Total power: ✓
Jetson Xavier NX	Total power: ✓ GPU power: ✗	Total power: ✓
Jetson Nano	Total power: ✓ GPU power: ✗	Total power: ✓

Table 4.4: Internal and external power measurement mechanisms used for each device.

4.2.5 Profiled Serverless Functions

In order to be able to deploy applications on the testbed, serverless functions are required. Therefore, different tasks that are implemented as OpenFaaS-based functions are chosen, since they can be deployed and profiled by means of the Galileo experimentation framework. All functions that are employed in this thesis are implemented in Python and are based on TensorFlow (TF), which is a library for machine learning and artificial intelligence. The specific functions used for profiling generally reside in the context of deep learning models that perform inference tasks, because all of the targeted devices are equipped with hardware accelerators.

Inference functions are chosen as reference applications since they reflect data-intensive serverless edge computing applications and therefore represent typical examples of edge intelligence use cases. Training ML models generally requires powerful high-end computing hardware, whereas performing inference with pre-trained models is usually less computation- and resource-intensive [WBC+19]. As the focus of this work lies on heterogeneous edge devices that exhibit different levels of resource constraints, only

[27]`https://docs.nvidia.com/deploy/nvml-api/group__nvmlDeviceQueries.html#group__nvmlDeviceQueries`, Accessed: Mar 6, 2023

inference tasks are carried out in order to ensure that the hardware characteristics of the used platforms are sufficient for processing the given workloads. Revising the classification system for edge intelligence platforms introduced by the authors of [ZCL+19] as described in Section 2.1.2, the approach applied in this thesis targets level 3 of the proposed scheme, i.e. on-device inference at the edge with cloud-based model training. A common use case for such level 3 applications is cognitive augmentation, which uses augmented reality and AI methods to perform inference at the edge and thereby enables real-time video processing for example [RHS+21].

In general, two types of functions are employed in the course of this thesis, namely TensorFlow 2 and TensorFlow Lite (TFLite) functions. As compared to TensorFlow 2, TensorFlow Lite is a more lightweight library that is optimized for the deployment of ML models on mobile, IoT or edge devices. However, TFLite-based functions can only run on GPUs of Android or iOS devices, besides common CPUs. Hence, these functions do not support the execution of a ML model on Nvidia GPUs, which is why TensorFlow 2 functions are required in order to be able to profile the Nvidia GPUs of the chosen edge computing platforms. TensorFlow 2-based functions can be executed on both CPUs and GPUs, which can be specified using an environment variable, so one TF2 function can be used for CPU-centered and for GPU-centered profiling.

Regarding the specific functions used for the experiments, existing inference applications are applied. In general two TF2 functions and one TF Lite function is selected. While the TF Lite function is already incorporated into the Galileo experimentation framework, which requires only minor modifications, the TF2 functions have to be manually integrated into the framework before being able to deploy and profile them. Table 4.5 provides a summary of the three functions that are used, including the type of task they perform and the possible execution target, which depends on the usage of TensorFlow 2 or TensorFlow Lite. All these functions can be found on GitHub in the galileo-experiments-functions repository[28].

Function	**Task**	**TF Version**	**Execution Target**
Resnet	Image Classification	TensorFlow 2	CPU: ✓ GPU: ✓
Efficientnet	Image Classification	TensorFlow 2	CPU: ✓ GPU: ✓
Objectdetection	Object Detection	TensorFlow Lite	CPU: ✓ GPU: ✗

Table 4.5: Serverless functions used for the experiments.

[28]`https://github.com/edgerun/galileo-experiments-functions`, Accessed: Jan 13, 2023

All inference applications used in this work generally preprocess the input data they receive from the user via an HTTP request, e.g. an image, load a pre-trained ML model and perform inference by applying the loaded model to the given input data and generating an output such as a prediction or classification. The result of the inference is then transmitted to the user via an HTTP response. In terms of the two TF2 functions used in this work, loading the model is triggered upon the first user request, while the loaded model is then cached for successive requests. Therefore, initially calling one of the TF2 functions by sending an HTTP request generally takes significantly longer than the subsequent requests. Regarding the TF Lite functions, the model is reloaded upon every request, so all requests - including the first one - approximately take a similar amount of time, assuming that there is no overlap between multiple requests.

4.2.6 Stress-ng Workload Generator

In order to generate a more comprehensive and more diverse training data set for the machine learning models, a versatile stress workload generator is used to execute stress tests under controlled conditions. To this end, the *stress-ng* tool is utilized, which is able to strain a system in many different, highly customizable ways. *Stress-ng* offers over 290 types of stress test[29], which are called stressors and can be arbitrarily combined[30]. The different stressors are grouped together into stressor classes, such as CPU for CPU-intensive stress tests or IO for generic Input/Output stress tests[31]. The CPU stressor for example encompasses various stress methods that demand a lot of computing power and thus CPU resources, such as Fast Fourier Transform (FFT) or matrix multiplication. If the CPU stressor is used, all related stress methods are performed sequentially in a round-robin mode. Different stressors can either be executed separately or combined as desired, while combined stressors run in parallel by default, but can also be invoked one by one if required. Unfortunately, *stress-ng* does not support GPU stress tests, so all computing tasks can only be executed on CPUs.

Using *stress-ng* within the Galileo experimentation framework enables more targeted and controlled experiments, since specific workloads can be generated while monitoring the resource usage and power consumption. This also fosters more heterogeneous data sets for the model training phase, so that the final prediction models are presumably more generalizable and as independent from the serverless functions as possible, because they are solely based on the resource utilization metrics. As a result, the models are more likely to be able to deal with unknown functions as they do not need to rely on any information about the serverless function associated with certain resource usage values.

Furthermore, platform-specific variations in terms of power consumption can easily be identified using *stress-ng* tests since they can strain the devices equally based on predefined CPU or memory load for example, so the resource utilization rates should be on a similar level for all devices. Therefore, the resulting power consumption of different

[29] `https://github.com/ColinIanKing/stress-ng`, Accessed: Jan 8, 2023
[30] `https://github.com/edgerun/edge-chaos`, Accessed: Jan 8, 2023
[31] `https://wiki.ubuntu.com/Kernel/Reference/stress-ng`, Accessed: Jan 8, 2023

edge devices for the same workload, i.e. for similar resource utilization values, can be compared, which enables the assessment of the impact of hardware heterogeneity on the power draw of computing platforms.

4.2.7 Experimental Procedure

All experiments conducted in the course of this thesis are executed by means of the Galileo experimentation framework as described earlier in Section 4.2.1, whereby every experiment is submitted as a profiling experiment. The framework is responsible for deploying the Kubernetes Pods associated with each experiment on the appropriate testbed nodes.

The empirical measurements are composed of two types of experiments. While the *function invocation experiments* are targeted at profiling the resource usage of certain serverless functions, as detailed in Section 4.2.5, the *stress-ng experiments* aim at generating workload under controlled conditions and monitoring the corresponding resource usage, as introduced in Section 4.2.6. The *function invocation experiments* therefore focus on executing identical tasks on different edge devices in order to assess the impact of hardware heterogeneity on the resource usage of computing platforms, whereas the *stress-ng experiments* strain the devices using equal loads, so the influence of the hardware heterogeneity on the power consumption under similar resource utilization rates can be ascertained.

Table 4.6 comprises the individual experiment configurations of the function invocation experiments, whereas Table 4.7 contains the experiment configurations of the *stress-ng* experiments. Each row of these two tables therefore represents one separate configuration that encompasses the specific parameters which are used for the empirical measurements.

Function	**Execution Target**	**# Requests**	**Request Pattern**
Resnet	CPU	100	interval.pkl
Resnet	GPU	100	interval.pkl
Efficientnet	CPU	100	interval.pkl
Efficientnet	GPU	100	interval.pkl
Objectdetection	CPU	100	IA time: 2s

Table 4.6: Configurations for function invocation experiments (five configurations).

In general, each of the experiment configurations listed in Table 4.6 and Table 4.7 is executed on every edge device outlined in Section 4.1, namely the Intel Xeon PC, the Jetson Xavier NX board and Jetson Nano board. Furthermore, in order to ensure consistency across measurement results, every experiment configuration is sequentially repeated five times in a row with a 15 seconds break in between the individual runs, which results in a series of five repetitions of the same configuration on each platform. The 15 seconds break in between the runs is required so that the Kubernetes Pods

Stressor Class	**Stressor**	**Value**	**Option**	**Value**	**Duration**
CPU	`cpu`	**1**	–	–	100s
	`cpu`	**2**	–	–	100s
	`cpu`	**4**	–	–	100s
	`cpu`	**8**	–	–	100s
CPU	`cpu`	0 (all)	`cpu-load`	**25%**	100s
	`cpu`	0 (all)	`cpu-load`	**50%**	100s
	`cpu`	0 (all)	`cpu-load`	**75%**	100s
	`cpu`	0 (all)	`cpu-load`	**100%**	100s
Virtual Memory	`vm`	1	`vm-bytes`	**20%**	100s
	`vm`	1	`vm-bytes`	**40%**	100s
	`vm`	1	`vm-bytes`	**80%**	100s
Generic input/output	`iomix`	**1**	–	–	100s

Table 4.7: Configurations for *stress-ng* experiments (12 configurations).

of an experiment run can be terminated and removed before the next experiment starts, otherwise conflicts may occur. In summary, there are 17 disparate experiment configurations, five for function invocations and 12 for *stress-ng* tests, which need to be carried out on all three devices and are replicated five times. As a result, 255 experiment runs are conducted in total. A complete list of the experiments performed for the function invocation configurations can be found in Appendix B, whereas the full set of experiments carried out for the stress-ng configurations is attached in Appendix B.

Regarding the request patterns used for the function invocation experiments, a distinction needs to be made between TF2 and TF Lite functions, since the TF2 functions are able to cache the loaded ML model, while the TF Lite function reloads the model upon every function call. The request patterns used for each function are also contained in Table 4.6. In general, the goal is to isolate each request as much as possible in order to facilitate successive data preprocessing tasks. Therefore, the Round Trip Time (RTT) of a request is crucial so that the requests do not overlap. As the RTT of all functions is considerably smaller than two seconds, an interarrival time of two seconds is sufficient to prevent overlaps. However, since the first request typically takes significantly longer than the subsequent ones in terms of the TF2 functions, because the ML model is loaded upon the first request, the interval between the first and the second request is set to 60 seconds for the TF2 functions. This interval is chosen in order to ensure that the first two requests do not overlap. As a result, the second request is sent 60 seconds after the first request and all subsequent requests are sent in a two second interval. This request pattern is reflected by the interval.pkl file. Regarding the TF Lite function, i.e. the *objectdetection* function, this differentiation of interarrival times is not necessary, since the model is loaded in every request. Consequently, a universal interval of two seconds can be set for all requests.

Baseline Profiling

In addition to the experiments described above, baseline profiling tests are conducted for the purpose of determining the average power consumption of each device in idle state. Therefore, 100 consecutive power measurements by means of the smart plugs are taken when there is no load on the devices. The resulting values for each edge device are then aggregated using the mean values. As a result, the mean idle power consumption can be used as a baseline for comparisons with the power draw measured during the experiments.

4.3 Analysis of Empirical Measurement Data

4.3.1 Data Preprocessing

Before exploring the empirical measurement data, the raw values need to be preprocessed in order to facilitate interpretations and comparisons, since especially the resource usage metrics that represent continuous counters cannot simply be rated without any preparatory steps. Furthermore, the obtained readings need to be converted into a simulator-friendly data format, which is also achieved through data preprocessing. This transformation is required because the final power prediction models are integrated into the *faas-sim* simulation framework, which cannot deal with the actual resource utilization values available as time-series data. Instead, it can only model the resource consumption of edge devices through estimates and approximations. Consequently, these preprocessing tasks therefore also enable the models to be trained with samples that have the same structure as the final input parameters provided by the framework for inference tasks. This is necessary since the prediction models should be able to perform inference based on the input they receive from the simulator during a simulation. By preprocessing the raw measurement readings, the inherent discrepancy between the actual resource usage values in the real world and the resource modeling of the simulation environment can be taken into account during the model development step, so more accurate results can be achieved.

Preparatory Steps

Prior to the actual data preprocessing, which aims to map the raw measured values to inputs the simulator can work with, can take place, the data has to be cleaned in preparation for further processing.

The data cleaning step for the function invocation experiments involves eliminating the first request of every experiment run in order to only consider the relevant measurement values, because only warm starts of serverless functions are focused in this work. As mentioned earlier, the first request of an experiment run initially loads the model into the processing unit and therefore requires significantly longer and also demands more resources than the subsequent calls, because the model can be cached for successive requests once loaded. Even though this only holds true for the TensorFlow 2 functions and not for the TensorFlow Lite functions, this procedure is applied for every function invocation experiment for the sake of consistency and convenience. Since the data preprocessing focuses on determining the function execution time and the average resource usage of one function call, leaving these traces in the data set would distort the calculated values for the function execution time as well as the resource consumption per request, which is why they are omitted. In order to take the removal of the first trace in every function invocation experiment into account for calculating the resource consumption per request, the telemetry readings associated with each of the removed traces need to be eliminated from the data set as well.

Regarding the *stress-ng* experiments, no data cleaning tasks are required. Due to the design of the Galileo experimentation framework, the *stress-ng* experiment runs also encompass two serverless function invocations, one at the beginning and one at the end of each run. However, the preprocessing of the *stress-ng* experiments is designed in a way that only the resource usage of the container running the stress tests is considered, not the resource usage of the container hosting the serverless function. This way, solely the resource usage caused by the *stress-ng* tests is utilized for further processing, while the additional computational burden imposed by the two function calls is disregarded.

Data Preprocessing Tasks

Preprocessing the raw measurement data is done by determining the resource usage of one function invocation based on the type of function and the underlying device on which the function is executed. The preprocessing therefore consists of three subtasks. Firstly, since the *faas-sim* is a trace-driven simulator, the traces of the profiled functions, more precisely the Function Execution Time (FET) as reported by the traces, need to be fitted to the simulation framework. This way, the performance of a device on a certain function invocation can be modeled in the simulation framework. Secondly, the telemetry readings must be transformed, because the simulator cannot deal with the actual resource utilization during a simulation but has to rely on approximations thereof for modeling the resource usage of requests. Hence, the resource usage demanded by a single function call has to be computed for each kind of function and for every edge device type. Thirdly, the measured power consumption values have to be preprocessed in order to be able to analyze the interrelation between resource usage and power consumption. All of these steps, which are performed in Python by means of Jupyter notebooks[32], are thoroughly described in the following.

Performance Modeling Due to the trace-driven nature of the *faas-sim* simulation framework, the traces recorded during the experiments need to be converted into a format that can be handled by the simulator, whereby one trace represents one function invocation triggered by a user request. For this purpose, the performance modeling approach of the *faas-sim* simulator is applied. Therefore, the Function Execution Time (FET) of a function call is utilized, which represents the performance of a device on a single function

[32]`https://jupyter.org/`, Accessed: Jan 18, 2023

call in terms of execution speed. The FET is then mapped to a simulator-friendly data format by taking the execution time of each trace and fitting it using a log-normal distribution. This way, the FET of a function call can be simulated by sampling from the distribution, which yields more disparate FETs as compared to only taking the average over all FETs. Consequently, this makes the simulations more realistic. Since the runtime of a function invocation highly depends on the function itself, i.e. the program that is run upon a user request, and the underlying hardware platform the function is executed on, the preprocessed FET is always associated with a particular function and a certain device. As a result, the performance of the devices with respect to serverless function calls can be modeled within the simulation framework.

Resource Modeling As the *faas-sim* simulates systems based on traces, the mean resource utilization of a single function invocation, i.e. a single trace, has to be determined so the simulator is able to model the resource consumption of requests. Therefore, the traces and telemetry metrics are utilized in order to compute the hardware utilization of each individual function call before the calculated values are averaged across all requests belonging to the same function and executed on the same device. As a result, analogous to the performance modeling, the resource modeling is also conditional on the invocation of a specific function on a particular node, since the hardware usage strongly depends on the underlying platform and the executed application. This resource modeling procedure is thereby derived from the approach applied in [Rai21]. Due to the fact that the simulator can only estimate the resource usage of function calls, the preprocessing of the telemetry data is based on the assumption that each request has a constant resource utilization throughout the whole function invocation. This represents an inherent approximation and thus simplification made by the *faas-sim* simulation framework.

Apart from the system-wide GPU utilization metric, all other resource usage measurements represent Kubernetes cgroup metrics, which means that only the resource consumption of the container running the serverless function inside a Kubernetes Pod is considered for the data preprocessing tasks. As mentioned in Section 4.2.3, all cgroup metrics except for the RAM usage, i.e. CPU usage time, block I/O and network I/O, which are published by the telemd monitoring agent represent continuous counters. Consequently, for example the CPU usage time at a certain point can be interpreted as the total usage time of the CPU so far, not the time of use since the last measurement and not the current CPU utilization. To retrieve the usage time since the last measurement, the difference between two consecutive measurements has to be calculated. This way, the average resource usage for one function invocation can be determined.

Contrarily, the system-wide GPU metric and the cgroup memory (RAM) metric have to be interpreted and processed in a different manner. The reason for this distinction is the fact that these two metrics do not report continuous counters, but the current utilization values at the time of measurement. Hence, they only represent the resource usage at a certain point instead of the average utilization or the total usage time up to a specific measurement. This evidently affects the accuracy of the measured values as compared to the other metrics and is thus a limitation of telemd. In order to obtain the mean GPU utilization and RAM usage per request, the corresponding measurements need to be averaged for each function call and then across all calls.

Telemetry metrics that are reported in bytes, namely block I/O and network I/O usage, are processed twofold in order to determine the data rate, i.e. the amount of bytes read or written per second on the one hand, and the total amount of data read or written per request on the other hand. As listed in Table 4.8, the resource usage preprocessing includes multiple metrics, namely CPU utilization, block I/O rate, total block I/O per request, network I/O rate, total network I/O per request, memory (RAM) utilization and GPU utilization. Regarding the CPU utilization, the resulting percentage can be above 100%, since the CPU usage time is summed up across CPU cores and no normalization in terms of CPU cores is applied during preprocessing. Consequently, on multi-core hosts, the CPU usage can reach values up to $N \cdot 100\%$, where N represents the number of cores.

Raw Telemd Metric (Unit)	**Preprocessed Metric (Unit)**
`kubernetes_cgrp_cpu`: total CPU usage time (ns)	CPU utilization per request (%)
`kubernetes_cgrp_blkio`: total block I/O usage (bytes)	Total block I/O usage per request (kilobytes), block I/O data rate (kilobytes/second)
`kubernetes_cgrp_net`: total network I/O usage (bytes)	Total network I/O usage per request (kilobytes), network I/O data rate (kilobytes/second)
`kubernetes_cgrp_memory`: memory (RAM) usage (bytes)	Memory (RAM) usage per request (megabytes)
`gpu_util`: GPU utilization (%)	GPU utilization per request (%)

Table 4.8: Resource usage preprocessing of individual telemetry metrics.

An additional task of the resource usage preprocessing is to determine the average resource consumption of the *stress-ng* stress test in order to be able to incorporate these data sets into the model development process as well. Although the *stress-ng* experiments do not represent serverless function calls, the correlation between resource usage and power consumption of the stress tests should still be integrated into the prediction models for the purpose of developing more generalizable models, as described earlier. Hence, the resource usage of the individual stress tests is averaged for each experiment run. This way, the *stress-ng* data sets can also serve as training data for the machine learning models, which facilitates a more distinct set of training samples. Due to the fact that *stress-ng* can only issue stress tests on the CPU and therefore does not provide GPU support, the GPU utilization can be set to zero.

Power Consumption Preprocessing Since the external measurements performed by the smart plugs represent the only uniform method of power monitoring across all devices, the values reported by the smart plugs are used for the development of the machine learning models. Analogous to the GPU and memory (RAM) measurements, the power consumption values also represent the current power consumption at the time of measurement, i.e. a snapshot of the system's energy wastage. Therefore, the preprocessing of the power usage is similar to the one applied for GPU and memory utilization. In order to obtain the mean power consumption of one request, all power values measured during a function invocation are averaged for each trace and then the mean across all requests of a function is taken. The number of power readings per request may differ due to the PowerDelta configuration applied for the smart plugs as described in Section 4.2.4. Since the plugs can only periodically publish the energy values in a 10 second interval, the PowerDelta value is set to 1%, which enables additional recordings in case of power changes over 1%. This way, it can be assumed that the power consumption between two consecutive measurements diverges at most 1% from the reported values, which is negligible and therefore supports the preprocessing concept of averaging the values. As the external power values are recorded in milliwatts, the mean power consumption per request has the same measuring unit.

Similar to the performance and the resource modeling, the power consumption preprocessing is also dependent on the type of function that is executed as well as the underlying hardware. However, it has to be noted that the resource modeling focuses solely on the resource usage of the function invocations or *stress-ng* tests respectively, but the power wastage is attributed to the whole device. Hence, the power consumption also takes the resource utilization of other (sub-)processes running on the system into account, while the resource usage preprocessing does not. As a result, the interrelationship between resource usage and power consumption represents an approximation, since the fine-grained power draw of an individual container running in a Kubernetes Pod cannot be determined by means of the power measurement facilities used in this work.

To give an example of the structure of the preprocessed data sets in terms of resource usage and power consumption, Table 4.9 shows the resulting values for the five repetitive runs of the *resnet* function executed on the CPU of the Intel Xeon. Therefore, the data set shown in this table contains the records for one experiment configuration. These records thus represent the average resource usage and power consumption of one function invocation on a particular node. The CPU and GPU metric are presented as percentages, for the block I/O and network I/O the total amount of data in kilobytes is depicted, the RAM usage is defined in megabytes and the power consumption is illustrated in milliwatts.

4.3.2 Analysis of Measurement Results

After performing the experiments according to the procedure detailed in Section 4.2.7 and preprocessing the raw data, the empirically obtained measurements can be examined in the context of a preliminary, exploratory data analysis in order to better understand the structure of the data and to generate first insights from the preprocessed measured values. This represents a subtask of the data analysis step in the model development process. For this purpose, the raw readings could not be used since the resource usage counters are hard to interpret and to compare across experiments, so the preprocessed

CPU	GPU	Block I/O	Network I/O	RAM Usage	Power
112.279074	0.0	0.0	2699.669837	722.561526	17224.894737
120.459088	0.0	0.0	2701.658510	659.388712	16804.000000
120.783417	0.0	0.0	2700.954367	665.511894	17319.720588
124.735307	0.0	0.0	2704.950388	705.664995	17280.012048
119.159790	0.0	0.0	2703.303786	719.569732	17507.196721

Table 4.9: Preprocessed average resource usage and power consumption per request of five repetitive runs.

values are employed. The preprocessed data sets represent the average resource usage and power consumption of one function invocation, i.e. one trace, and encompass five records for each experiment configuration, as shown in Table 4.9. There are several analytical investigations that are of particular interest for this work, which are discussed in the following.

Resource Usage and Power Consumption of Repetitive Runs

In order to determine the measurement errors, the measured values of the repetitive runs of each experiment configuration executed on all edge devices can be examined. As stated earlier, every configuration is replicated five times on each platform to ensure consistency and reliability of measured values. By comparing the preprocessed variables of repeated experiments, the deviations in terms of resource usage measurements and power readings can be ascertained, which can give indications about the distribution of the data and the measurement errors.

For this purpose, box plots, or more specifically box-and-whisker plots[33], are used to visualize these discrepancies. The boxes of these plots contain the quartiles of the data with the bottom and top of a box representing the borders of the first quartile and the third quartile, respectively. The line inside the boxes shows the median of the distribution, whereas the whiskers extend the data set based on a function using the inter-quartile range. Outliers are visualized as individual points beyond the whiskers. Moreover, the function name contained in these plots describes the serverless function executed on the testbed node, whereby the suffix indicates the execution target, i.e. whether the function is carried out on the CPU or on the GPU. The power consumption values come from the smart plugs and thus represent the externally measured power draw of the devices, since the smart plugs constitute the only uniform measurement facility across all three types of edge devices.

Figure 4.3 illustrates the series of repetitive runs conducted for the *resnet* function on the GPU of the Intel Xeon PC, i.e. the eb-b-xeongpu-0 node of the testbed, and the corresponding resource usage and power consumption per request on average. For the

[33]`https://seaborn.pydata.org/generated/seaborn.boxplot.html`, Accessed: Feb 14, 2023

sake of completeness, the mean Function Execution Time (FET) is also integrated. As the values highly vary, the plots could not be aggregated into a single plot. The average CPU utilization values across all five runs, which are represented by the second, i.e. the orange, box range from 106.3% to 110%. The absence of outliers implies an acceptable distribution of data. As mentioned earlier, the CPU utilization values are not normalized based on the number of cores, so utilization rates of over 100% are plausible. Nonetheless, even though the function is executed on the GPU, the GPU utilization is noticeably low with a median around 0.68%, as indicated by the line inside the third, i.e. the green, box. However, this observation is verified manually using the NVIDIA System Management Interface (nvidia-smi) command-line utility tool[34], which is able to monitor the real-time GPU utilization of Nvidia GPUs. In general, the nvidia-smi tool also reports GPU utilization values between 0% and 1% during the function invocations on the Intel Xeon. Consequently, the workload caused by the function executions is presumably too low to heavily stress the available GPU on the Intel Xeon, which is why the GPU is only minimally utilized.

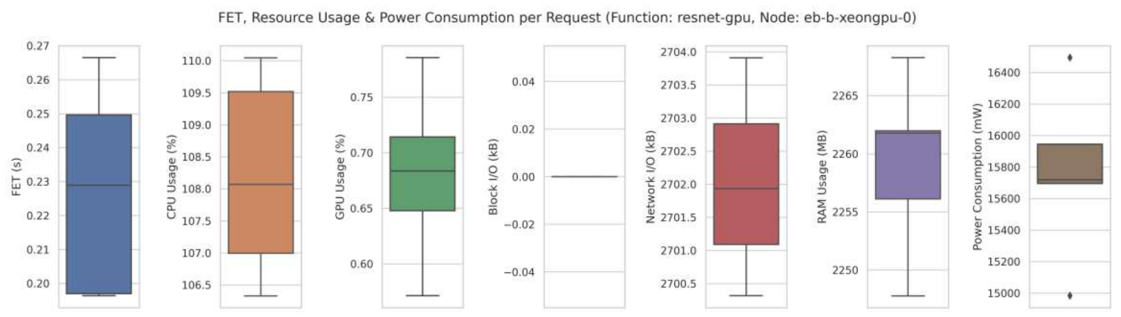


Figure 4.3: Average FET, resource usage and power consumption of the *resnet-gpu* function on the Intel Xeon.

Moreover, the absence of any block I/O in Figure 4.3 indicates that the images that are used for the input of the functions are so small that they can be stored in-memory. The network I/O can be attributed to receiving incoming requests including images and sending back the corresponding responses. Furthermore, it becomes evident that the average power readings differ across the individual repetitions, since they approximately range between 15,000 mW and 16,500 mW, which makes up a difference of 1.5 W. This discrepancy indicates that the power measurements slightly fluctuate, which is also the case for the resource usage. However, the measurement errors in terms of the power consumption of these five consecutive runs on the Intel Xeon are still tolerable and fairly reasonable.

This key finding regarding the consistency and stability of the smart plug measurements can also be observed in other experiment configurations, where either the serverless function or the edge device type varies. Figure 4.4 shows the distribution of the data for

[34]`https://developer.nvidia.com/nvidia-system-management-interface`, Accessed: Feb 12, 2023

the same function as shown in Figure 4.3, but executed on a different hardware platform, namely the Jetson Nano board. From the rightmost plot, which shows the average power consumption across the five repetitive runs of this function on the eb-a-jetson-nano-0 node, it can be derived that the power values are also not widely scattered but rather concentrated with a maximum difference of 300 mW. As a result, the distribution of power consumption data is even lower on the Jetson Nano than on the Intel Xeon.

Another interesting discovery that can be deduced from the plot shown in Figure 4.4 is that the mean GPU utilization rates of the individual runs are approximately spread between 31.5% and 43%, which constitutes a considerable difference of 11.5%. Contrarily, the CPU values only range between nearly 62.5% and 64.5%, so only 2% fluctuation can be noted here. As a consequence, it can be concluded that the GPU utilization measurements are presumably more error-prone than the CPU measurements. This observation can be traced back to the fact that the CPU metric represents a continuous counter, whereas the GPU utilization constitutes punctual measurements of the current usage at a certain point in time, which is highly dependent on the timing and the frequency of the measurements. Since all telemd metrics including the GPU metric are reported once per second and the function execution takes less than one second with this experiment configuration, as shown by the mean FET, the GPU utilization metric of telemd is very volatile and fragile to rapidly changing values as they might not be captured.

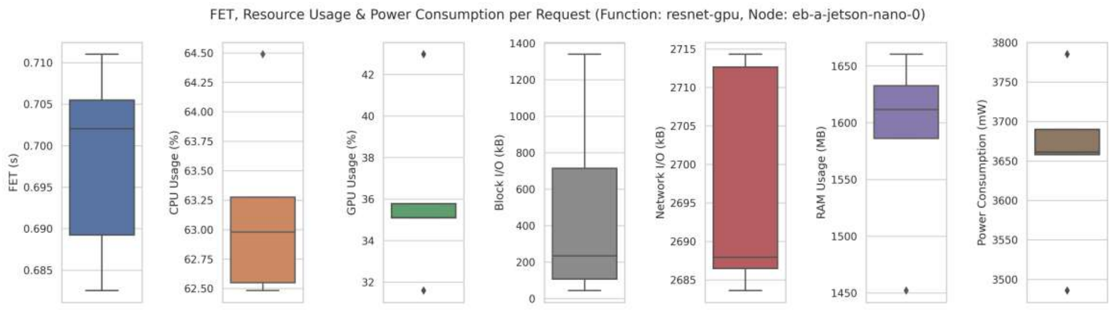


Figure 4.4: Average FET, resource usage and power consumption of the *resnet-gpu* function on the Jetson Nano.

In addition, when comparing the FET, resource usage and power consumption of the *resnet* function executed on the GPU of the Intel Xeon, see Figure 4.3, with the execution on the GPU of the Jetson Nano, see Figure 4.4, the results clearly show the impact of the disparate hardware characteristics of these two computing architectures. The hardware heterogeneity of edge devices therefore significantly influences the function execution time of a serverless function executed on two different platforms, as well as the resource usage, which highly differs across devices, and the power consumption. Regarding the FET, the *resnet-gpu* function has a median FET of 0.23 seconds on the Intel Xeon as compared to roughly 0.7 seconds on the Jetson Nano, so the execution speed deviates among these two edge devices. Furthermore, besides the CPU utilization which exhibits a gap of around 45%, the GPU utilization is of particular interest, since the *resnet* function is executed on the GPU in these two experiment configurations. While one function invocation utilizes approximately 0.7% of the GPU on the Intel Xeon, about 35% of the GPU is utilized when calling the same function on the Jetson Nano. The block I/O and RAM usage also varies between these two devices. Hence, even though the same task, i.e. the same serverless function with an equal request pattern, is executed on both platforms, the resulting resource usage variables indisputably diverge.

Moreover, the power consumption of the two devices also notably differs. With a median value of 15,700 mW across all five runs, the Intel Xeon has a substantially higher power consumption per request than the Jetson Nano with not even 3,700 mW. Due to these severe discrepancies in terms of execution speed, resource usage and power consumption across hardware platforms, constructing a single, cross-platform predictive model that achieves sufficient power prediction accuracy across different hardware architectures is assumed to be unfeasible. Therefore, the establishment of multiple, platform-specific power prediction models is as it is anticipated that they are able to provide a viable and more appropriate approach for modeling the power consumption of heterogeneous edge devices. This way, the considerable impact of hardware heterogeneity on resource utilization rates and power draw in the edge domain can be taken into account for energy modeling. As a consequence, this valuable insight is specifically important for the subsequent machine learning model development phase and is henceforth considered in this context.

By looking at the average resource usage of the *efficientnet* experiments conducted on the CPU of the Jetson Xavier NX board as presented in Figure 4.5, another striking observation, namely the presence of GPU utilization values above zero, becomes evident. Since these experiments are executed on the CPU only, the GPU utilization is expected to be 0.0%. However, since this is not the case, manual verifications of the measured values are required for proving the authenticity of the data. To this end, the real-time GPU usage of the Jetson Xavier NX board in idle state is manually monitored by means of jtop. Hereby, random GPU utilization peaks of up to 30% are registered, although no function and no stress tests are executed on the device during the validation procedure. Potential causes for this behavior are other processes that run on the GPU. Therefore, an average GPU utilization of around 1-1.5%, as measured by the experimentation framework, can be attributed to this unexpected behavior of the Jetson board. This finding however can only be observed for the Jetson Xavier NX, not the Jetson Nano.

For the purpose of affirming the inexplicable GPU utilization on the Jetson Xavier NX board, the five *objectdetection* runs carried out on this device are visualized in Figure 4.6. As recognizable in this figure, these experiments also exhibit GPU utilization values above 1% even though the *objectdetection* function represents a TF Lite function that cannot be executed on Nvidia GPUs, because it only provides GPU support for Android or iOS devices. Consequently, this observation also refutes the potential suspicion that the *efficientnet-cpu* function depicted in Figure 4.5 is erroneously executed on the GPU instead of the CPU.

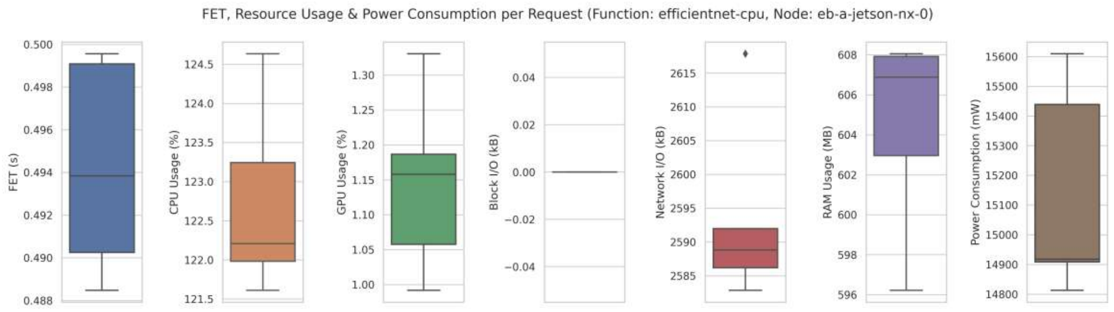


Figure 4.5: Average FET, resource usage and power consumption of the efficientent-cpu function on the Jetson Xavier NX.

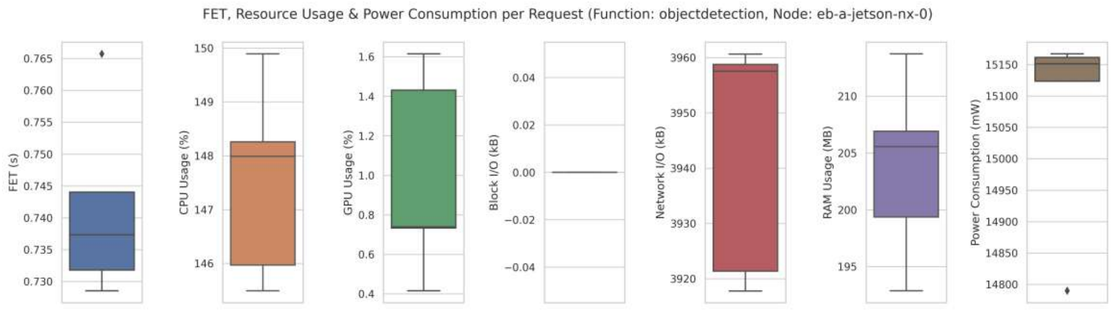


Figure 4.6: Average FET, resource usage and power consumption of the *objectdetection* function on the Jetson Xavier NX.

In order to be able to compare the extent of fluctuations of the power values across the respective repetitive runs more easily, the plots in Figure 4.7 and Figure 4.8 provide the distributions of power readings for each individual function executed on each hardware platform. Since the Intel Xeon and the Jetson Xavier NX exhibit a similar level of power consumption, the data sets of these two devices are aggregated into one plot while the data sets of the Jetson Nano are illustrated in a separate plot due to its significantly lower level of energy wastage. This way, the quality of the power consumption measurements and the measurement deviation of the smart plugs can be assessed.

In terms of the mean power draw variations of the Intel Xeon and the Jetson Xavier NX, it can be noted that the individual distribution vary across devices and also across functions on the same platform. The most extreme outliers can be identified for the *resnet-gpu* function on the Xeon PC, whereas the majority of distributions does not exhibit any outliers at all. However, as mentioned above, the two outliers of the *resnet-gpu* function on the Intel Xeon are still only 1.5 W apart while all other dispersions for both the Intel Xeon and the Jetson Xavier NX show lower fluctuations of around 0.5-1 W, which represent tolerable deviations.

With respect to the results of the Jetson Nano, the extend of divergences within repetitive runs also differs, but the highest discrepancy is around 1.2 W due to an outlier, which is

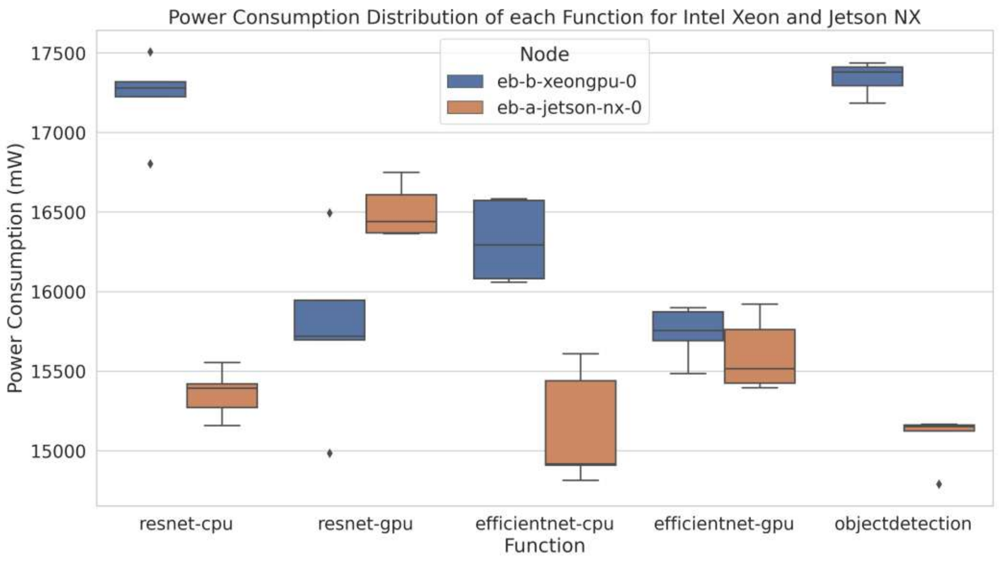


Figure 4.7: Power consumption distribution of function invocations for Intel Xeon and Jetson Xavier NX.

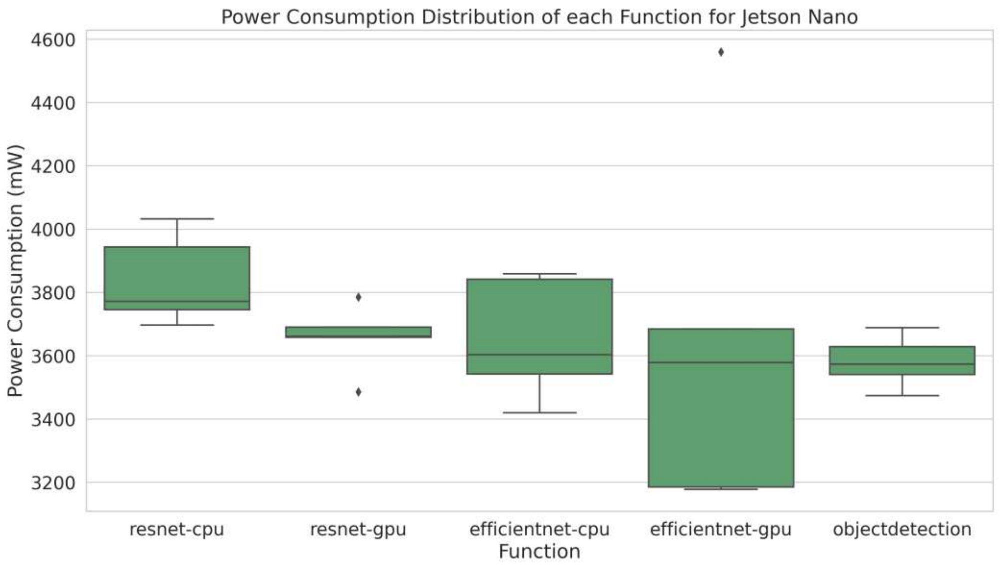


Figure 4.8: Power consumption distribution of function invocations for Jetson Nano.

also acceptable. The other variations are generally in the order of 200-400 mW. As a result, the measurement fluctuations are minor, so the measured values are considered as relatively stable and consistent in general.

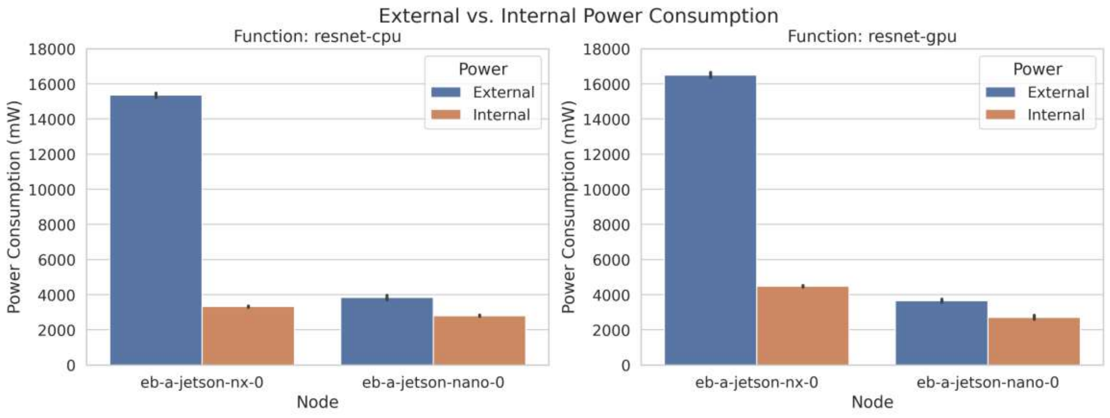
External vs. Internal Power Consumption

In order to compare the external and the internal power readings of the profiled edge devices, the power values reported by the different power consumption measurement facilities are contrasted with each other. Regarding the external power measurements, the values obtained by the smart plugs are available for all three platform types. As described in Section 4.2.4, the internal measurement instruments differ, since the two Jetson boards can be monitored using jtop, which reports the total power usage of the module, while the Intel Xeon PC only provides internal power readings of the GPU component. As a result, the comparisons between external and internal power consumption of the Jetson boards can be combined, whereas the results of the Xeon should be examined separately.

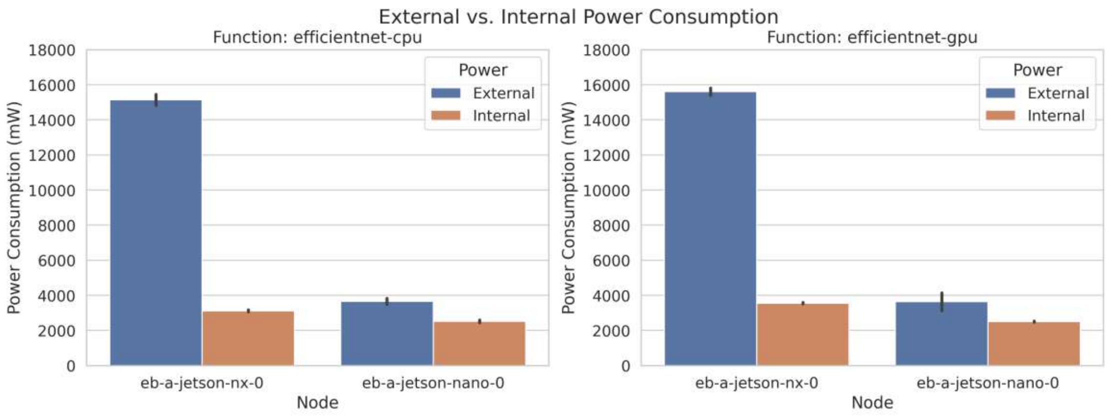
With respect to the Jetson boards, the plots in Figure 4.9 visualize the average external and internal power wastage per request for the *resnet* (Figure 4.9a), *efficientnet* (Figure 4.9b) and *objectdetection* (Figure 4.9c) function. Regarding the *resnet* and *efficientnet* serverless functions, both the CPU-based and the GPU-based execution is contained. Since the *objectdetection* function cannot be executed on Nvidia GPUs, such a distinction is not possible for this type of function. The most striking observation is the considerably difference between the external and internal power values of the Jetson Xavier NX board, whereby the externally measured power consumption is consistently significantly higher than the power readings captured by the internal sensors. This mismatch can be observed in all five plots, i.e. across all functions. However, this observation only holds true for the Jetson Xavier NX but not for the Jetson Nano board, since the external and internal power readings do not diverge as much on this device. Although higher external measurements are expected, the extent of discrepancies of the Jetson Xavier NX is larger than initially estimated and the behavior of the two Jetson boards unforeseeably differs.

Regarding the Jetson Xavier NX board, the reported power values of the smart plugs are still plausible, since the Nvidia NVP model[35], i.e. the energy-performance profile of a device, is set to `20W 6CORE` on the Jetson Xavier NX, which means that the power budget of the board is fixed at 20 W and all six CPU cores are online and available for computing. Hence, an externally measured power consumption of around 15 W is considered to be realistic. The `20W 6CORE` profile of the Jetson Xavier NX thereby represents the model with the highest power budget and also the highest number of online cores. For the Jetson Nano board, the NVP model is set to `MAXN` which imposes a power budget of only 10 W. Furthermore, the Jetson Nano has less CPU cores in total and a weaker GPU than the Jetson Xavier NX, so a lower power consumption as compared to the Jetson Xavier NX is logical. This would therefore explain the significant divergence of the two hardware platforms in terms of the smart plug measurements. Due to the power budget imposed by the NVP profiles and the corresponding energy-performance trade-off, specifying other power profiles for the Jetson boards would potentially yield different results.

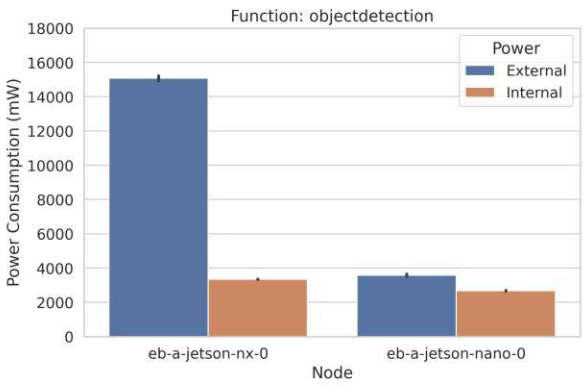
[35]`https://docs.nvidia.com/jetson/archives/r35.2.1/DeveloperGuide/text/SD/PlatformPowerAndPerformance/JetsonXavierNxSeriesAndJetsonAgxXavierSeries.html#supported-modes-and-power-efficiency`, Accessed: Feb 14, 2023



(a) *Resnet* function execution on CPU and GPU.



(b) *Efficientnet* function execution on CPU and GPU.



(c) *Objectdetection* function execution on CPU.

Figure 4.9: Comparison between external and internal power consumption per request on Jetson Xavier NX and Jetson Nano.

In general, the divergence in terms of external and internal measurements is caused by the fact that the internal power monitors queried by jtop only include the power consumption of the module itself, while the smart plugs cover additional components, such as the fan, the carrier board or the power supply [KHH+20]. Furthermore, the measuring frequency differs between the two measurement instruments and also the measuring points are not synchronized, so they might acquire the current power draw at different points in time, which might also lead to discrepancies between the reported values. Since jtop obtains the current internal power consumption once every second, whereas the smart plugs include the PowerDelta configuration for instantly reporting power changes above 1%, jtop might not be able to capture fast-changing power values. For example, if the power consumption of a device changes for a short period of time between two jtop measurements, i.e. within a few milliseconds, jtop is not able to ascertain these changes while the smart plugs can also report such changes if the power value increases or decreases more than 1%.

Regardless of the severe differences in terms of external and internal power consumption, the plots in Figure 4.9a and Figure 4.9b also show that the average power consumption of the Jetson Xavier NX board as registered by the smart plugs is slightly higher for the GPU-based execution of the functions than for the CPU-based execution. This behavior can be observed for the external as well as the internal power measurements. However, the difference is only about 0.5-1 W, whereby it has to be noted that the invocation of functions on the GPU is typically faster than on the CPU. In terms of the Jetson Nano, this finding does not hold true because the mean external and internal power consumption values are fairly consistent and independent of the execution target. In general, as stated earlier, the external power readings of the Jetson Xavier NX and the Jetson Nano do substantially differ across all experiment configurations as depicted in Figure 4.9, so they exhibit a considerably different level of power wastage. Therefore, due to the heterogeneous hardware characteristics of the devices being profiled, individual power models need to be developed for each device type, as the accuracy of the predictions would presumably suffer from a joint model.

Concerning the third edge device profiled in the course of this work, the results of the external and internal power consumption of the Intel Xeon PC are illustrated in Figure 4.10. Since the internal measurements of the Intel Xeon only incorporate the power wastage of the GPU, solely the two experiment configurations including executions on the GPU are presented. From the two plots shown in Figure 4.10, it can be seen that the internal power readings, which only report the power draw of the GPU and associated circuits, are noticeably higher than the external measurements conducted by the smart plugs. This represents a very interesting but counterintuitive finding, since the assumption would have been the exact opposite, as it is the case for the Jetson boards. Because the externally measured power values take additional hardware components like the CPU into account, they should be explicitly higher than the internal GPU power readings. At this juncture, however, it cannot be determined where this divergence stems from and whether the external or the internal measurements are more accurate. It can only be deduced that these two types of measurements do not correspond.

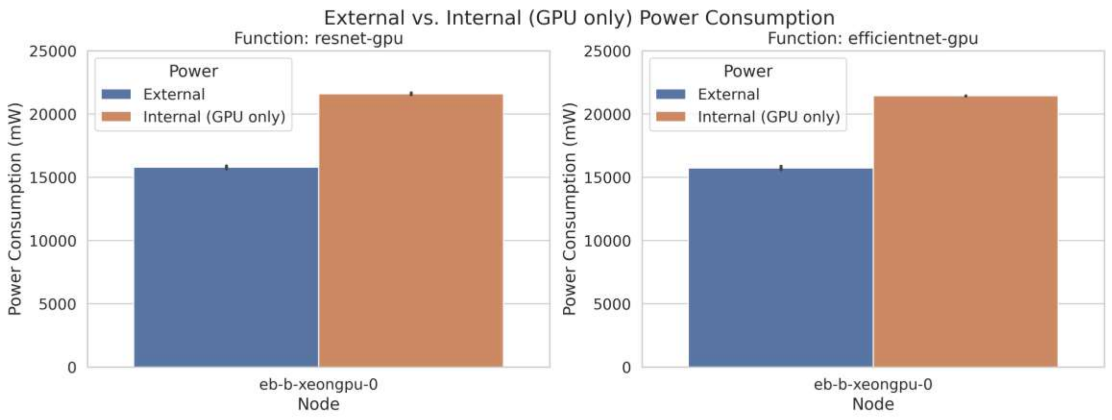


Figure 4.10: Comparison between external and internal (GPU only) power consumption per request on Intel Xeon.

Power Consumption of Stress-ng Workloads

The different workloads generated by the *stress-ng* tool can give valuable and additional insights into the interrelationship between resource usage and power consumption, since the stress tests enable experiments with controlled resource usages. By comparing the different parameter sets passed to *stress-ng* in terms of resource usage and the resulting power draw, potential correlations might become evident. Furthermore, since the stress tests provoke similar resource usage values on all devices according to predefined loads, the impact of hardware heterogeneity on the power consumption of edge computing platforms can be assessed.

Moreover, for the purpose of contrasting the power draw measured during the stress tests with the mean baseline power consumption in idle state, additional measurements are taken for each device. The baseline power consumption is then calculated by taking the average of 100 consecutive measurements of the respective device in idle state. The results can be found in Table 4.10. Since the idle power values substantially vary, they already indicate that the hardware heterogeneity has a significant impact on the power draw of edge devices.

Device	**Baseline Power Cons.**
Intel Xeon	16,246.60 mW
Jetson Xavier NX	14,699.38 mW
Jetson Nano	3,457.91 mW

Table 4.10: Mean baseline power consumption of the edge devices.

As previously detailed in Table 4.7, different parameters are used for *stress-ng* in order to achieve distinct resource utilization values. Figure 4.11 summarizes the results of the CPU-centered experiments that include *stress-ng* CPU stressors for generating 25%, 50%, 75% and 100% load on each online CPU, respectively. This plot shows the mean power consumption for each stress test configuration on each of the computing architectures being profiled, and includes the individual error bars indicating the 95% confidence intervals. All five repetitions of each experiment configuration are considered in this plot. Additionally, the baseline power consumption of each node, as presented in Table 4.10, is included and visualized through dashed lines in the appropriate color. The top blue line therefore represents the baseline power of the Intel Xeon, the middle orange line indicates the baseline power of the Jetson Xavier NX and the bottom green line displays the baseline power of the Jetson Nano.

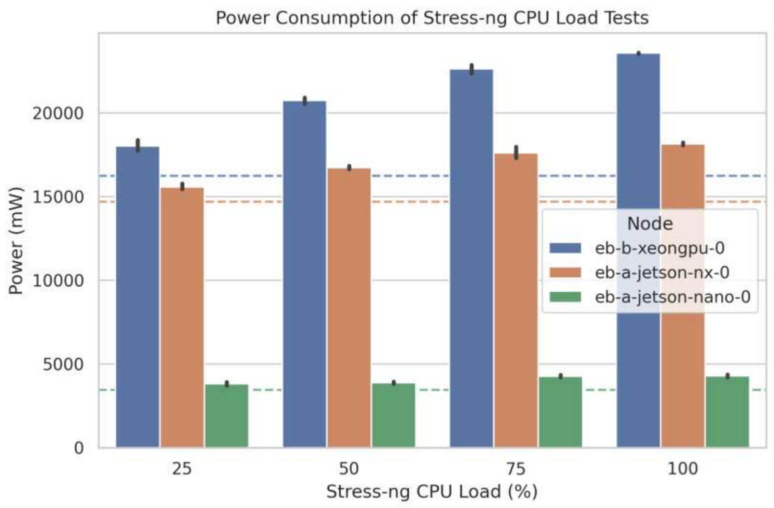


Figure 4.11: Mean power consumption of *stress-ng* CPU load tests.

When investigating the changes in power consumption depending on the CPU load in Figure 4.11, it is clearly visible that the power draw consistently increases with rising CPU load, which represents an expectable behavior. Although the extend of the elevations varies across platforms, the general tendency can be identified for all three types of edge devices. While the greatest and most apparent increments can be recorded for the Intel Xeon, i.e. the eb-b-xeongpu-0 node, only marginal, but still recognizable increases can be registered for the Jetson Nano board. The results of the Jetson Xavier NX board lie somewhere in between the Xeon and the Jetson Nano.

Furthermore, by taking the baseline power consumption values into account, it can be seen that the power draw of the Intel Xeon on 25% CPU load is already about 2 W higher than the baseline power consumption, which is around 16,250 mW on average. In case of a CPU load of 100%, the mean power drainage of the Intel Xeon increases up to 23,600 mW, which makes up a difference of over 7 W compared to the baseline. A similar behavior but with lower absolute increases can be observed for the Jetson Xavier

NX, whereby the baseline is about 14,700 mW. The average power consumption of this device on 25% CPU load is around 15,500 mW, whereas it reaches 18,100 mW on 100% CPU load. The least significant increments can be identified for the Jetson Nano, where the difference between the baseline and the mean power consumption measured at 100% CPU load is around 800 mW. This indicates that the Jetson Nano operates in a very energy-efficient manner. In summary, it can be concluded that the CPU utilization has a considerable impact on the power consumption of edge devices.

Another observation that can be derived from the plot illustrated in Figure 4.11 is the high variation in terms of power consumption across devices due to their heterogeneous hardware characteristics. While the workloads provoked by the *stress-ng* tests are equal for all three edge devices, i.e. the load is consistent on each platform, the power readings significantly differ, especially with respect to the Intel Xeon and the Jetson Nano. This finding therefore confirms the previous assertions which state that the power consumption is heavily influenced by varying hardware capabilities. As a result, choosing a multi-model approach for modeling the power draw of heterogeneous edge devices in order to overcome the severe power consumption discrepancies is supported by these results.

In addition to the CPU-centered stressors, stress tests targeting the RAM usage are included in the experiments. These kinds of stress tests are performed by means of the *stress-ng* virtual memory stressors and setting the memory usage to 20%, 40% and 80% of the total available memory, respectively. The average power consumption of the individual nodes is outlined in Figure 4.12, whereby all five repetitive runs of each experiment configuration are taken into account for this plot. Regarding the Xeon PC, the power consumption at 80% RAM load is definitely higher than at 20%, but the lowest power usage is reported at 40%. Since the difference between the readings for 20% and 40% is only about 100 mW on average and the error bars for both values are comparatively high, this mismatch might results from measurement inaccuracies. The mean power consumption of the Jetson Xavier NX rises when increasing the RAM usage from 20% to 40%, but then slightly decreases when incrementing the RAM usage to 80%. Therefore, no clear and consistent trend can be detected for these two devices. Considering the Jetson Nano, no obvious change in power consumption can be identified at all. As a result, no universally valid statement about the specific impact of different RAM usage levels on the power draw of these platforms can be made.

Similar to the CPU load plot, the RAM usage plot illustrated in Figure 4.12 also incorporates the baseline power consumption of the individual devices, which is indicated by the dashed lines. Again, the top blue line represents the baseline power of the Intel Xeon, the middle orange line indicates the baseline power of the Jetson Xavier NX and the bottom green line displays the baseline power of the Jetson Nano. Even though no continuous increase in terms of power draw can be observed, the power consumption values of all devices are consistently higher than the baseline measurements. Hence, it can be concluded that the RAM usage does indeed influence the power consumption of devices in general, however, as stated above, an explicit behavioral pattern with regard to ascending RAM usage rates cannot be determined based on the available data.

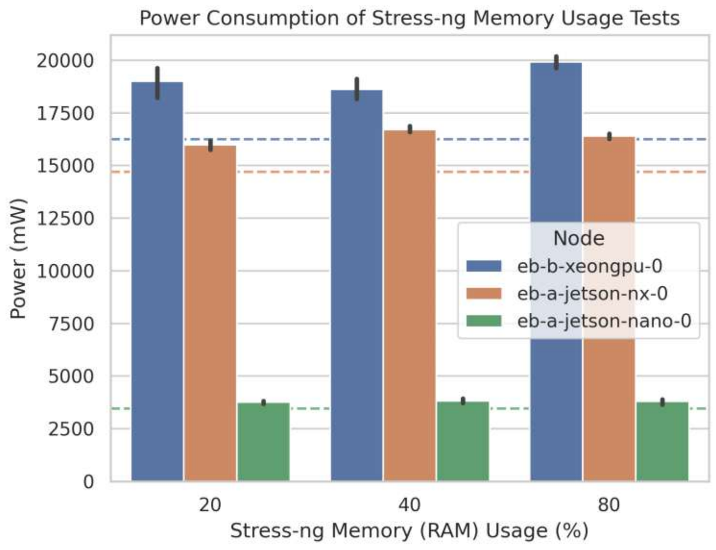


Figure 4.12: Mean power consumption of *stress-ng* memory usage tests.

4.4 Machine Learning Model Construction

4.4.1 AutoML and TPOT

After the raw measurement data is preprocessed for usage within the *faas-sim* simulation framework, the data can be used for developing and training the ML models. Since the preceding data analysis demonstrates the significant impacts of hardware heterogeneity in terms of resource usage, one machine learning model per device type is developed instead of one single model that can be applied to all devices. The final ML models should be able to predict the average power consumption of one function invocation based on the resource usage of the underlying device, which represents a typical regression problem, since the purpose of the models is to forecast a continuous value. The model construction phase of the model development process introduced in Section 4.1.2 and illustrated in Figure 4.2 consists of three main steps. These include model selection, model training and model tuning, i.e. model optimization.

For the model construction phase, where machine learning techniques are applied to develop power prediction models, an Automated Machine Learning (AutoML) tool is used. Designing efficient and accurate machine learning pipelines is generally very challenging and requires substantial domain knowledge and ML experience [OM16]. AutoML tools aim at saving time and effort associated with manually building ML models by automating certain repetitive and tedious tasks involved in this process [TWG$^+$19]. They therefore enable an automatized and easy development of ML models with minimal human invention, which considerably accelerates the model construction process and also facilitates this process to be carried out by non-experts [OBUM16].

In the context of this work, the Tree-based Pipeline Optimization Tool (TPOT) is chosen as AutoML system to streamline the model development process. In general, TPOT supports supervised learning for classification and regression problems[36]. As the name indicates, TPOT generates and optimizes tree-based machine learning pipelines for finding the most suitable pipeline for a given data set, which is achieved by applying genetic programming techniques. TPOT thus represents a data science wizard that assists with exploring the provided data set, discovering alternative models, comparing and tuning the discovered pipelines, as well as recommending the pipeline with the highest score as representative pipeline to the user. Internally, TPOT uses a k-fold cross-validation strategy for optimization, which should prevent model overfitting issues, where the generated models are not sufficiently generalizable[37]. The score of a pipeline thereby depends on the scoring function that is utilized in order to determine the quality of a pipeline, whereby the default scoring function is conditional on the type of problem, i.e. classification or regression. This way, TPOT enables automating the design and optimization of ML pipelines and thereby aims to maximize the prediction accuracy for a specific problem domain. TPOT is available as an open source Python package[38] and is based on the well-known Python machine learning library scikit-learn [OM16, OBUM16, LFM20]. Regarding this thesis, TPOT version 0.11.7 is used.

TPOT already comes with a range of built-in configurations that affect the operators and parameters used during the automation process. Since the final prediction models are integrated into the *faas-sim* simulator, the performance of the models is crucial, so an appropriate performance-accuracy trade-off is essential. Therefore, not the default TPOT configuration, but the *TPOT Light* configuration is applied in this work, which only takes simple and fast operators and preprocessors into account for the pipelines[39]. This way, particularly lightweight and fast-running models can be developed.

Furthermore, TPOT already incorporates multi-objective Pareto optimization into the model selection step of the process, which aims at enhancing multiple distinct objectives in order to find efficient solutions by making certain trade-offs. In TPOT Pareto optimization is implemented by trying to maximize the model performance in terms of prediction accuracy while minimizing the complexity of the pipeline, i.e. the total number of operators within the pipeline, at the same time. As a result, TPOT can take model complexity into account during the model development and is able to discover compact yet effective pipelines, that also make interpretation easier [OM16, OBUM16].

TPOT Machine Learning Pipeline

As mentioned earlier, according to the defined model development process, the model construction phase consists of model selection, model training and model tuning. However,

[36]https://epistasislab.github.io/tpot/using/, Accessed: Jan 30, 2023
[37]https://epistasislab.github.io/tpot/api/, Accessed: Jan 30, 2023
[38]https://github.com/EpistasisLab/tpot, Accessed: Jan 30, 2023
[39]https://epistasislab.github.io/tpot/using/#built-in-tpot-configurations, Accessed: Feb 2, 2023

using TPOT none of these steps has to be carried out manually, since TPOT is able to automatically perform these steps without manual human intervention. In order to illustrate the tasks that TPOT is able to automate, Figure 4.13 displays the TPOT automation process within a machine learning pipeline. The tasks that are automated by TPOT include feature selection, feature preprocessing, feature construction, model selection and parameter optimization. As shown in this figure, the data acquisition and data cleaning at the beginning, as well as the model validation at the end of the pipeline have to be done manually [OBUM16, OUA+16]. Since the measurement data obtained during the experiments is already available in the right format due to the preceding data preprocessing step, it can directly be used as input data set for TPOT. Furthermore, a manual feature selection procedure is not required since this task is also automated by TPOT as shown in Figure 4.13.

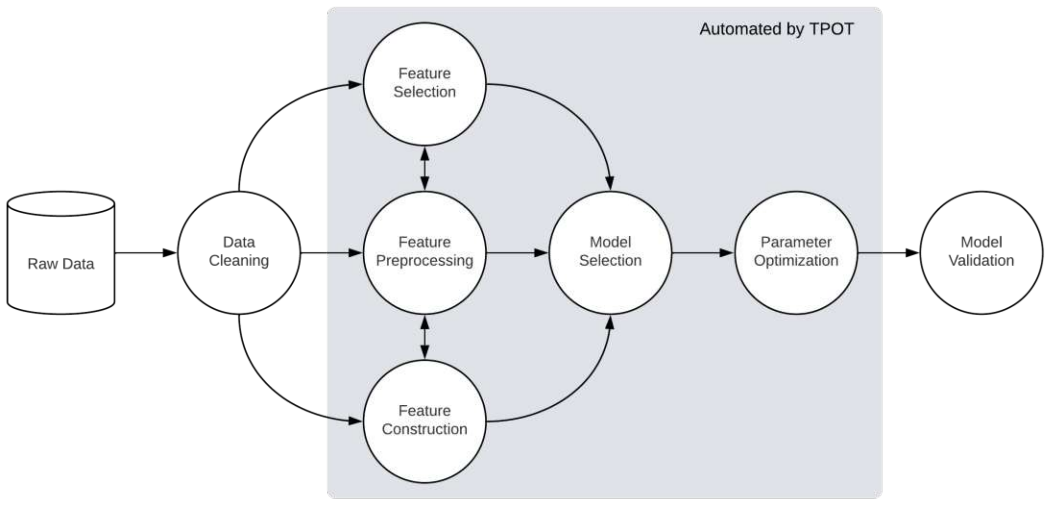


Figure 4.13: The TPOT automation process based on [OBUM16].

Developing machine learning models typically requires three data sets, namely the training, validation and test set, whereby these sets have to be independent and mutually exclusive. The training set is used as input for the model, so it can learn from the given data, whereas the validation set is required to evaluate the model in terms of prediction accuracy during the training phase and is thus essential for the model optimization procedure. Finally, after the training phase is completed, the data samples contained in the test set are fed into the final model in order to determine its accuracy on an unseen data set [ABC+21]. With regard to TPOT, providing a separate validation set is not necessary, since TPOT automates the optimization procedure and internally uses a cross-validation strategy to evaluate the individual pipelines during this process. Therefore, when using TPOT, the data sets only have to be split into a distinct training and test set.

In general, splitting the data into training and test sets is done randomly based on a predefined ratio. However, with respect to this work, this step is done manually in order to ensure comparability and consistency across the individually developed models. Since three separate models have to developed due to the severe hardware heterogeneity of edge devices, the training and test sets should be split in the same way. The division of the data into training and test sets is outlined in Table 4.11, which indicates that specific workloads are used for training and the remaining ones are used for testing. This division is applied to all three data sets, i.e. to the data set of each computing platform. In total, each platform-specific data set encompasses 85 entries, whereby all five repetitions of each experiment configuration are included. While the training set comprises the *resnet-gpu*, *efficientnet-cpu* and *stress-ng* experiment workloads, which make up 70 records for each edge device, the test set consists of the *resnet-cpu*, *efficientnet-gpu* and *objectdetection* workloads, which encompass 15 records in total for each platform. Each training and test set is then further split according to the features that should serve as the model input, i.e. the resource usage data, and the prediction target which represents the output of the final models, i.e. the power consumption.

Workload	**Training Set**	**Test Set**
Resnet-cpu	✗	✓
Resnet-gpu	✓	✗
Efficientnet-cpu	✓	✗
Efficientnet-gpu	✗	✓
Objectdetection	✗	✓
Stress-ng	✓	✗

Table 4.11: Division of data into training and test set.

Listing 4.1 provides the code that is used for the automated pipeline optimization by means of TPOT after the data sets are appropriately split. Since three individual models are developed, these steps have to be executed three times. First of all, the TPOT parameters have to be defined. For this purpose, the statement in line 2 is required. As the power prediction represents a regression problem, the TPOTRegressor class is employed, which can be customized through a number of parameters. The number of generations is set to 100, while the population size is set to 1000. Since TPOT generally evaluates $population\_size + generations \cdot offspring\_size$ pipelines in total[40], whereby the offspring size is equal to the population size by default, 101,000 pipelines, i.e. $1000 + 100 \cdot 1000 = 101,000$, are analyzed with this set of parameters. The verbosity parameter is only used to tell TPOT how much information should be printed while running, whereas the random state variable defines the random number generator seed, which is intended to improve the reproducibility of a TPOT run.

[40]`https://epistasislab.github.io/tpot/api/`, Accessed: Feb 6, 2023

Furthermore, the configuration dictionary parameter (config_dict) specifies a built-in or custom TPOT configuration, which defines the operators and parameters TPOT applies. In this case, the TPOT Light configuration is chosen, as described above. The subsequent parameter (n_jobs) can be used to define the number of processes TPOT should run in parallel, whereby the special value -1 indicates that all available cores on the system should be utilized. Moreover, the negative Mean Absolute Error (MAE) is chosen as scoring function to evaluate the quality of an individual pipeline, since the default scoring function applied to regression problems, which is the negative Mean Squared Error (MSE), is very sensitive to outlier prediction with large errors, while the MAE weights all errors equally. In general, the MAE indicates the mean absolute error over all predictions and thus has the same unit of measurement as the data, i.e. milliwatts. The negated value of the MAE is required because by default the score of the scoring function is aimed to be maximized, so maximizing the negated value means minimizing the actual MAE.

In order to be able to determine the duration of the TPOT optimization process, the time is recorded before and after the optimization procedure using the `timeit` Python package, as done in line 5 and line 11, respectively. To ensure reproducibility and comparability of the measured time, all three TPOT runs, i.e. one run for each device, are executed on the same machine, namely a virtual machine instance with 16-CPU cores based on 2.1 GHz Intel Xeon processors (Cascadelake) and 32 GB RAM, which is part of the testbed used for the empirical measurements. The code in line 8 finally starts the TPOT pipeline optimization process based on the provided training data set. After the TPOT run finishes, the eventually recommended pipeline can be evaluated by means of the test set and the previously set scoring function, which is the negative MAE in case of this work. The corresponding code required for this step is shown in line 17. Lastly, the pre-trained pipeline represented as a scikit-learn pipeline object can be exported by means of the `joblib` Python package. The corresponding code fragments for this final step can be found in line 20 and 21.

4.4.2 Results of TPOT Runs

Due to the TPOT Light configuration, only a subset of the available preprocessors, models, parameters, etc. is considered for the optimization procedure, whereby the focus lies on simple and rapid operators. Futhermore, using the parameters defined in Listing 4.1 on a multi-core machine, TPOT evaluates pipelines in parallel on all available cores on the system and thus additionally speeds up the process. Consequently, the durations of the three TPOT runs are rather low, as shown in Table 4.12. Although the number of records in the training set is the same for all three devices, the durations of the individual runs differ. However, different data sets and even repeated runs on the same data set trigger different pipelines to be explored and evaluated by TPOT, so these variations are plausible.

Besides the durations of the TPOT runs, Table 4.12 also includes the average internal cross-validation (CV) score achieved by the recommended pipeline on the training set of each run. The internal CV score is based on the given scoring function, which is the

```python
# Define the TPOT parameters
tpot = TPOTRegressor(generations=100, population_size=1000, verbosity=2,
    random_state=42, config_dict='TPOT light', n_jobs=-1,
    scoring='neg_mean_absolute_error')

# Record the start time of TPOT pipeline optimization process
start_time = timeit.default_timer()

# Optimize the pipeline based on the given data set
tpot.fit(df_train_features, df_train_target)

# Calculate the duration of TPOT run in minutes
elapsed = (timeit.default_timer() - start_time) / 60

# Print the time TPOT used for the pipeline optimization procedure
print(f'Elapsed Time: {elapsed}')

# Evaluate the final pipeline
print(f'Score: {tpot.score(df_test_features, df_test_target)}')

# Export the pre-trained pipeline
filename = 'tpot-model.sav'
joblib.dump(tpot.fitted_pipeline_, filename)
```

Listing 4.1: Python code for the TPOT pipeline optimization process.

Target Device	Duration	CV Score
Intel Xeon	$\sim$ 55 min.	-487.26
Jetson Xavier NX	$\sim$ 100 min.	-442.13
Jetson Nano	$\sim$ 135 min.	-85.37

Table 4.12: Duration and mean internal cross-validation (CV) score of TPOT runs.

negative MAE as defined in Listing 4.1. Since the power values are recorded in milliwatts and the MAE has the same unit of measurement as the data, the MAE represents the mean absolute error over all predictions in milliwatts. For example, regarding the Intel Xeon, a CV score of around 487 means that on average the error describing the difference between the predicted and the actual power values is 487 mW, i.e. less than 0.5 W. The internal CV scores of the Jetson Xavier NX and the Jetson Nano are even smaller. However, these values cannot be directly projected to the final MAE on the independent test set, i.e. on a diverse and unseen set of inputs, since the training set is internally used by TPOT for the cross-validation. The accuracy of the models on the test set samples therefore has to be evaluated separately, which is covered in the subsequent chapter.

The final pipelines for each device that achieved the best internal score are presented in Listing 4.2, Listing 4.3 and Listing 4.4, respectively. When comparing all three models, it becomes clear that they include different operators, which consist of regression models, preprocessors, transformers and their corresponding hyperparameters[41]. Furthermore, also the number of incorporated operators varies between the individual pipelines, whereby the pipeline for the Jetson Nano contains the most operators. The reason for this outcome is the fact that each pipeline is tailored to the provided training set and since the training sets differ between the three edge devices, the final outcomes also diverge. This observation however supports the previously made decision to develop an individual model for each platform due to the severe hardware heterogeneity and its implications on resource usage and power consumption. Furthermore, it can be stated that the Jetson Nano pipeline has the best internal CV score while it also exhibits the longest training duration and the highest number of operators, so it represents the most complex pipeline. However, the exact consequences of these assertions are ascertained in the following chapter.

```
LassoLarsCV(Normalizer(RidgeCV(ElasticNetCV(MaxAbsScaler(SelectFwe(
    ElasticNetCV(input_matrix, l1_ratio=0.15000000000000002, tol=0.1),
    alpha=0.026000000000000002)), l1_ratio=0.75, tol=0.0001)), norm=max),
    normalize=True)
```

Listing 4.2: Recommended pipeline for the Intel Xeon.

```
DecisionTreeRegressor(CombineDFs(PCA(CombineDFs(input_matrix,
    SelectPercentile(ElasticNetCV(input_matrix, l1_ratio=0.25, tol=1e-05),
    percentile=83)), iterated_power=3, svd_solver=randomized),
    ElasticNetCV(input_matrix, l1_ratio=0.25, tol=1e-05)), max_depth=7,
    min_samples_leaf=1, min_samples_split=8)
```

Listing 4.3: Recommended pipeline for the Jetson Xavier NX.

```
RidgeCV(DecisionTreeRegressor(KNeighborsRegressor(SelectPercentile(
    ElasticNetCV(LassoLarsCV(RidgeCV(RobustScaler(ElasticNetCV(
    KNeighborsRegressor(SelectFwe(SelectFwe(LassoLarsCV(input_matrix,
    normalize=False), alpha=0.01), alpha=0.002), n_neighbors=19, p=1,
    weights=uniform), l1_ratio=0.4, tol=0.01)))), normalize=False),
    l1_ratio=0.9, tol=0.1), percentile=81), n_neighbors=20, p=1,
    weights=uniform), max_depth=7, min_samples_leaf=18, min_samples_split=3))
```

Listing 4.4: Recommended pipeline for the Jetson Nano.

[41]https://epistasislab.github.io/tpot/api/, Accessed: Feb 18, 2023

CHAPTER 5

Evaluation

This chapter covers the evaluation of the developed machine learning models which is done by validating the models in terms of prediction accuracy and performance. To give an overview of the general procedure, Figure 5.1 outlines the individual steps of the evaluation process. The specific validation approach applied in this work is thoroughly described in Section 5.1. As stated in this section, a prerequisite for the assessment of the model precision and inference speed is the integration into the *faas-sim* simulation framework, which is elucidated in Section 5.1.1. The simulator thus represents the primary evaluation environment, as explained in Section 5.1.2. Afterwards, the approach utilized for the model accuracy evaluation is detailed in Section 5.1.3, whereas the procedure for evaluating the performance of the models is contained in Section 5.1.4. The results of the evaluation are then presented in Section 5.2. This section therefore encompasses the conclusions that can be drawn from the simulations performed by means of the *faas-sim* simulator in terms of model precision, see Section 5.2.1, and inference speed, see Section 5.2.2. While the accuracy has an impact on the level of generalization of the developed models, their complexity can negatively affect their performance and thus impede the scalability of the simulator. The trade-off between these two distinct objectives is of particular interest with respect to the suitability of the power models for the simulator, which is why the performance-accuracy trade-off is discussed at the end of this chapter in Section 5.2.3. This section thereby concludes the evaluation.

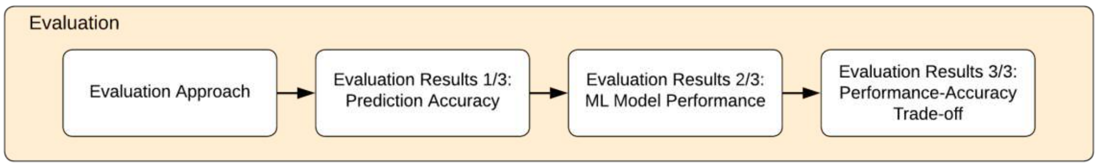

Evaluation
Evaluation Approach
Evaluation Results 1/3: Prediction Accuracy
Evaluation Results 2/3: ML Model Performance
Evaluation Results 3/3: Performance-Accuracy Trade-off

Figure 5.1: The machine learning model evaluation process.

5.1 Evaluation Approach

The primary objective of the model validation, which constitutes one of the final steps in the model development process, is to assess whether the power values predicted by the models correspond with the actual power consumption of the edge devices in the real world. Furthermore, the level of overhead the models impose on the simulator due to inference latency is also of particular interest for determining the performance of the developed power models. This way, it can be ensured that the predictions can be generalized to data samples beyond the training sets while they do not severely impede the scalability of simulations conducted with the *faas-sim* simulator at the same time. As a result, the evaluation of the developed ML models is two-fold and consists of multiple assessments that are carried out separately. The specific findings of these evaluations can then be combined in order to be able to reason about the inherent performance-accuracy trade-off of the models. Nevertheless, before the models can be evaluated using the simulation framework, they have to be integrated into the existing *faas-sim* project, which ultimately enables the simulator to make predictions about the power consumption of devices during the simulation of a scenario. This procedure is thoroughly described in the following section.

5.1.1 Simulator Integration

Before the power models can be validated, they have to be integrated into the *faas-sim* simulation framework, which is introduced in Section 2.2. This enables the assessment of the prediction accuracy on the hand and the model performance on the other hand. As a result, the *faas-sim* simulator represents the main evaluation environment for the model validation.

As described in Section 4.13, the pre-trained models are already exported to files after the TPOT training process. In order to use the models during simulations, they have to be loaded from the files at the beginning of a simulation run. From then on, the power prediction functionality provided by the machine learning models is ready for use. For the purpose of making predictions, the resource usage of a function invocation, namely CPU, GPU, block I/O, network I/O and RAM utilization, must be fed into the appropriate model according to the device being simulated. The predicted power consumption associated with each function call can then be accessed after the simulation finishes. Consequently, the *faas-sim* framework is extended by an additional power prediction feature.

However, not only the ML models, but also the preprocessed function execution times and resource usages of the individual serverless functions have to be incorporated into the simulation framework in order to be able to simulate the functions used for the experiments in the *faas-sim* framework. To this end, the log-normal distribution of the function execution times is included in the simulator for each device and for each type of function, i.e. the *resnet* function executed on CPU and GPU, the *efficientnet* function invoked on CPU and GPU, and the *objectdetection* function, which can only be deployed on CPUs. Therefore, as stated earlier, the FETs are dependent on the underlying device and the serverless function being simulated. During a simulation, the function execution time of a serverless function is sampled from the appropriate log-normal distribution according to the device that should be modeled. As described before, this approach reflects the performance modeling method applied by the *faas-sim* simulator.

Regarding the resource usage required for a function invocation, the values of the five repetitive runs for each experiment configuration are averaged, so the mean resource utilization of one function call can be embedded into the simulator. As mentioned before, the simulator is not able to simulate the real resource consumption of devices, instead, it uses the preprocessed resource data for estimating and approximating the resource utilization. Similar to the FET, the resource consumption also depends on the underlying device and the serverless function that should be simulated. Hence, for each platform, the average resource usage of every function has to be embedded into the *faas-sim* framework. Since there are five serverless functions, namely *resnet-cpu*, *resnet-gpu*, *efficientnet-cpu*, *efficientnet-gpu* and *objectdetection*, five resource usage characterizations are available for each edge device. The resource usage values can then be passed to the ML models, so the predicted power consumption for a certain function call can be obtained after a simulation. In general, the input for the final machine learning models represents the preprocessed mean resource usage per request, whereas the output should be the expected power consumption associated with the given resource utilization values.

5.1.2 Evaluation Environment

Besides a rather straightforward accuracy validation performed by means of TPOT, all other evaluation tasks are conducted using the *faas-sim* simulator. Therefore, the *faas-sim* framework represents the primary evaluation environment of this work. The results of the simulation runs are then either contrasted among each other or compared with the measurement results of the experiments carried out by means of the testbed. Hereby, the most important metrics that need to be considered for validating the models include the predicted power consumption yielded by the simulator, the actual power draw measured during the experiments and the execution time of different simulation runs, among others.

Regarding the *faas-sim* framework, a simulation is described through a simulation scenario, whereby a simulation scenario, in turn, is defined by a topology and a benchmark. The topology encapsulates the configuration of the simulation environment, i.e. the cluster being simulated, with respect to the specific nodes involved in the simulation and the network setup. Furthermore, the benchmark specifies the container images, function deployments and request profiles, which are used for setting up the runtime system of a simulation[1]. This way, the simulator is able to simulate serverless workloads, which are represented by function requests, on cluster nodes. The simulation scenarios are therefore fully customizable and can contain arbitrary numbers of nodes and functions.

[1] `https://edgerun.github.io/faas-sim/concepts/index.html#simulation`, Accessed: Feb 20, 2023

After a simulation run finishes, the time it took to simulate a certain scenario, i.e. the execution or wall-clock time of the simulation run, and the total period of time that was simulated can be obtained. Moreover, various metrics are continuously logged during a simulation, such as the resource utilization for each request, the function execution time, the timestamps of the individual invocations or the deployed container images. The predicted power consumption of each function call is also included in the resource utilization metrics, along with the corresponding CPU utilization, GPU utilization, total block I/O, total network I/O and RAM usage. The relevant data sets that are necessary for further processing or analysis tasks can be extracted from the simulator outputs and saved as CSV files at the end of a simulation. Even though the *faas-sim* simulator is able to simulate the download of container images in addition to function invocations, this feature is not used in the scenarios of this thesis for the sake of comparability, because the Galileo experimentation framework does not take the image pulling into account for the total duration of an experiment. Therefore, the focus lies on warm start execution of functions.

Since the specific simulation scenarios and the pertinent outputs differ across the various analysis tasks depending on the evaluation goal, the detailed settings of the simulation scenarios and the particular outputs used for the assessments can be found below in the appropriate sections. In general, the underlying hardware on which the simulations are run influences the execution time of the simulator. In order to ensure reproducibility and comparability across simulation runs, all simulations conducted in the course of the evaluation are executed on the same local machine, i.e. an Apple MacBook Pro with a 2,8 GHz quad-core Intel Core i7 processor and 16 GB LPDDR3 RAM.

5.1.3 Prediction Accuracy Approach

The main goal of the prediction accuracy evaluation is to ascertain whether the models can be generalized to inputs that are not contained in the initial training sets or not. This is done by testing how the models perform on unseen data samples they are not fitted on and analyzing the difference between the predicted values and the actually measured ones. The empirical power measurements thereby represent the ground truth that is used for comparisons. In terms of this work, this validation can be accomplished in three different ways.

Firstly, TPOT is able to evaluate a predefined test set with respect to the scoring function specified for the TPOT run. This yields a score that indicates the accuracy of the predictions on the given test set based on the mean absolute error. Secondly, the MAE scores of the individual models is contrasted to the MAEs of an existing statistical model for predicting the power consumption of devices. Therefore, the prediction accuracies can be compared in order to be able to interpret the results of the ML models more comprehensively. Thirdly, the precision of the ML models can be assessed by conducting simulations with the *faas-sim* framework and comparing the results to the real-world measurements. This way, the integration of the models can be used to determine whether the power values predicted during a simulation correspond with the actual power readings obtained throughout the experiments. Furthermore, the total amount of energy estimated by the simulator can be contrasted with the overall energy consumed by the experiment runs. As a result, all three approaches, i.e. a TPOT-based analysis, a comparison with an existing power model and a simulator-based examination, are applied for the prediction accuracy evaluation of the developed ML models and are thus detailed in the following.

Accuracy Evaluation using TPOT

For the purpose of evaluating the prediction accuracy of the models by means of TPOT's built-in model validation, the required code is already presented in Listing 4.1 on page 68, specifically in line 17. The score function of TPOT takes the testing features, i.e. the resource usage values of the test set, and the prediction labels, i.e. the measured power consumption for the resource usage data, as parameters. As return value this function yields the score of the previously optimized pipeline on the provided test set based on the scoring function defined earlier. This score therefore represents the estimated average inference accuracy on the given test set. As the scoring function is set to the MAE, the precision score can be interpreted as the mean absolute error of the testing samples.

While the training sets for each edge device type contain the preprocessed records of the *resnet-gpu*, *efficientnet-cpu* and *stress-ng* experiments, the test sets encompass the data of the *resnet-cpu*, *efficientnet-gpu* and *objectdetection* functions, as outlined in Table 4.11 on page 66. This way, it can be ensured that the test set only involves records that are excluded from the training phase of the models. In every data set, five repetitive runs of each experiment configuration are present. As stated in Section 4.4.1, the test set for every individual device comprises 15 records, because three functions are used for this set and five repetitions are carried out per function. Since the code contained in Listing 4.1 is executed for all platforms separately, the score can directly obtained after each TPOT run. The resulting scores are presented in Section 5.2.1.

Accuracy Evaluation using a Statistical Power Model

In order to be able to rank and interpret the accuracy of the machine learning models more easily, the prediction precision of the developed models is contrasted with an existing power model. To this end, a simple statistical model, which calculates the power consumption linearly according to the CPU utilization rate, is chosen as the basis for comparison. Since the accuracy of the ML models can be rated using the mean absolute error as reported by the preceding TPOT evaluation, the MAE is selected as reference value. Therefore, the MAE of the statistical model has to be computed analogous to the TPOT model validation.

The existing model computes the estimated power consumption based on a static, i.e. fixed, and a dynamic fraction, whereby the dynamic share represents a linear function depending on the CPU utilization [WT21, CRB$^+$11]. Such a statistical power model also forms the foundation of the energy modeling techniques used by other simulators, for example the LEAF simulator[2] or the iFogSim[3], as introduced in Section 3.2. Consequently, this model only considers the CPU usage as the sole metric for resource consumption as compared to the ML models, which incorporate multiple variables. The specific formula employed by the linear model to predict the power consumption of devices is contained in Equation 5.1. This formula is applied to the preprocessed data sets of each device, namely the data samples containing the mean resource usage per request. Similar to the data sets used for the TPOT evaluation as described above, the records envisaged for the linear model also encompass all repetitions of individual experiment configurations. However, while the TPOT evaluation is only targeted at the test samples, the complete data is taken into account for this kind of accuracy validation.

As a preparatory step, the preprocessed CPU utilization rates need to be divided by 100 to obtain the decimal number instead of the percentage and then normalized based on the number of CPU cores of each device. While the Intel Xeon and the Jetson Nano both have four CPU cores, the Jetson Xavier NX comprises six cores. As a result, the CPU usage required for the formula ranges between zero and one, whereby a value of one represents full utilization of all cores. Furthermore, the baseline power measurements of each platform, as carried out earlier, can be used as the idle power, whereas the maximum power usage values for each device have to be defined initially. For the Jetson boards, the power budget of the NVP models is applied, so the maximum power usage of the Jetson Xavier NX is set to 20 W and the power draw of the Jetson Nano is limited to 10 W. Regarding the Intel Xeon, such power restrictions are not known, so the maximum power consumption is estimated based on the power readings published during the *stress-ng* stress tests. Therefore, 25 W is considered as a realistic value. This way, the power consumption of every serverless function executed on each edge device type can be approximated by means of the statistical model.

$$\underbrace{Idle\ Power}_{\text{Static Part}} + \underbrace{(Max.\ Power - Idle\ Power) \cdot CPU\ Util.}_{\text{Dynamic Part}} = Total\ Power \tag{5.1}$$

Afterwards, similar to the TPOT evaluation approach, the predicted values can be compared with the ground truth values for each individual device using the MAE. The formula used for calculating the MAE of each platform is contained in Equation 5.2, whereby y_i represents the predicted power consumption, x_i is the actual power value and n constitutes the number of samples contained in the data set[4]. This enables a comparison between the MAEs of the ML models as returned by TPOT and the MAEs of the linear power model, which can give insights into the accuracy of the ML models in contrast to the precision of the statistical model.

[2]https://github.com/dos-group/leaf, Accessed: Mar 5, 2023
[3]https://github.com/Cloudslab/iFogSim, Accessed: Mar 5, 2023
[4]https://scikit-learn.org/stable/modules/model_evaluation.html#mean-absolute-error, Accessed: Mar 5, 2023

$$\frac{\sum_{i=1}^{n} |x_i - y_i|}{n} = \text{MAE} \tag{5.2}$$

Accuracy Evaluation using *faas-sim*

The second kind of accuracy evaluation involves different simulations carried out using the *faas-sim* simulation framework. After the FET, the average resource usage per request and the power models are integrated into the simulation framework, as outlined in Section 5.1.1, simulation scenarios for the accuracy evaluation have to be defined and implemented. The main objective of these scenarios is to reproduce the profiling experiments that are initially conducted for data acquisition. This way, comparability between the simulation results and the measurement results of the experiments can be ensured. Hence, the simulation scenario parameters used for assessing the accuracy of the predictions should reflect the experiment configurations defined earlier. As a result, the same set of scenarios facilitates different analyses regarding the predicted and the actual power consumption. Since the *faas-sim* simulator only models serverless functions and the *stress-ng* experiment configurations are already utilized for the training of the models, they are omitted for the evaluation.

Simulation Scenarios For the purpose of guaranteeing comparability, the scenarios need to reproduce the experiments in terms of 1) the devices used, i.e. the Intel Xeon, Jetson Xavier NX and Jetson Nano, 2) the serverless functions executed during the experiments, namely *resnet-cpu*, *resnet-gpu*, *efficientnet-cpu*, *efficientnet-gpu* and *objectdetection*, and 3) the request patterns of the functions. Therefore, each scenario consists of only two nodes, whereby one node represents the device on which the function invocations are simulated and the other node is used to simulate the data transmission caused by the HTTP requests.

Since the resource usage metrics that are integrated into the simulator represent the average values across five repetitive runs of the same function on the same device, these values differ from the ones used during the model training phase, where the resource usages of all five individual runs are taken into account. Hence, all functions can be used for testing the prediction accuracy by means of the simulation framework. Analogous to the experiment configurations listed in Table 4.6 on page 44, five different functions need to be simulated on each of the three devices, which results in $5 * 3 = 15$ distinct scenarios. In order to ensure consistency among the simulation results and the testbed results, the same request patterns are used for each function.

The implementation of these scenarios facilitates different kinds of examinations, which are performed on the basis of the simulation outputs, i.e. the exported CSV files. First of all, the power consumption per request predicted during the simulation runs can be compared to the average power draw of a function invocation determined by the measurement results of the testbed experiments. While the predicted power consumption is based on the resource utilization metrics of a function call, the actual power consumed during a request is obtained from the smart plugs. As a second analysis, the total amount of energy estimated by the *faas-sim* for a certain scenario and the overall amount of energy consumed during the associated profiling experiment on the testbed can be calculated and contrasted. The results of these two assessments can then be used to draw conclusions about the accuracy of the predictions yielded by the power models.

Comparison of Power Consumption per Request Regarding the comparison between the predicted and the actual power consumption per request, the approach is rather straightforward. As described in Section 5.1.1, for each set of experiments, the average resource usage values across all five repetitive runs are integrated into the simulator. Therefore, the predicted power wastage per request is based on the mean resource usage across all repetitions of an experiment configuration. Similarly, the power consumption per request of the individual runs also have to be aggregated using the average. As a result, for every type of function invocation experiment, the predicted power consumption per request returned by the models can be contrasted with the mean power draw measured by the smart plugs.

Comparison of Total Energy Demand In addition to directly comparing the power consumption per request between the simulation results and the experimental measurements, the power values can be used to compute the total amount of energy consumed by a device over a certain period of time. By calculating the expectable energy demand estimated by the simulation scenarios as well as the actual energy usage of the corresponding real-world experiments, further assertions about the accuracy of the power prediction models can be made. This way, another comparison between the predicted and the actual values can be established.

In general, the total energy usage of a device over a certain time span depends on two variables, namely the power consumption of the device and the overall time during which the device is used. Hence, the formula presented in Equation 5.3 [GA19] can be applied, which yields the total energy consumed over the associated period of time in watt-seconds, whereby one watt-second is equivalent to one joule.

$$Power\ (watts) \cdot Time\ (seconds) = Energy\ (watt\text{-}seconds) \tag{5.3}$$

However, since the power consumption during a function invocation differs from the baseline power draw when the device is in idle state, the energy used for function calls and the energy consumed during idle state have to be computed separately and added up afterwards to get the total energy demand. The computations therefore involve several steps. While the energy calculations for the *faas-sim* scenarios are based on the estimations derived from the simulations, the computations for the experiments are based on the actual measurements obtained through the smart plugs and the Galileo experimentation framework. Even though the origin of the data differs, the procedure, i.e. the sequence of arithmetic steps, is the same for the simulations and the experiments. In order to cover all devices and serverless functions, the following steps have to performed for every simulation scenario and for every series of repetitive experiment runs.

As a first step, the function execution time of every request is taken and added up. The resulting sum indicates the overall time in which a function was executed. By subtracting this sum from the total duration, the remaining period in which the node was in idle state can be determined. With respect to the simulator, the total period of time that was simulated is used for this purpose, whereas the total runtime of an experiment is utilized for the actual measurements. Afterwards, the energy usage of these two individual time frames can be computed by converting the power values of the underlying device from milliwatts to watts and then multiplying the power in watts with the time in seconds, as defined by the formula shown in Equation 5.3. For the idle state energy demand, the mean baseline power readings as measured earlier are applied. These are identical for both the simulations and the experiments. Regarding the energy used for function invocations, the predicted power consumption per request is applied for the simulations and the actual power draw per request is utilized for the experiments. Finally, by summing up the two energy usage values, the total energy demand can be acquired. In terms of the experiments, the measured values of repetitive runs are aggregated across all five runs by using the average.

5.1.4 ML Model Performance Approach

Regarding the model performance evaluation, the primary goal is to ascertain the inference speed of the predictions and to determine the level of overhead the machine learning models impose on the execution time of the simulator, which could have negative effects on the scalability of the simulation framework. Similar to the approach applied for assessing the prediction accuracy, the model performance evaluation also consists of multiple investigations.

Initially, the analysis of the inference speed per prediction across models might reveal differences and can therefore give first insights into the performance of the models. Furthermore, the execution time of simulations including the power models can be compared with the runtime of the simulator when the average power consumption is reported instead of making predictions. Analyzing the results of these two distinct configurations might indicate the overhead caused by incorporating the machine learning models into the *faas-sim* framework. Moreover, the impact of the models on the scalability of the simulator can be examined by contrasting the wall-clock time of different scaling scenarios, i.e. scenarios with a varying number of nodes. Since the simulation framework is able to simulate large numbers of nodes and typical edge computing use cases encompass topologies consisting of numerous nodes, the scaling behavior of the models is crucial for time-efficient simulations. While all of these evaluations are based on the *faas-sim* simulator and use the same simulation scenarios as their basis, they require individual modifications regarding the scenarios or the simulator configuration itself, which are outlined in the following along with the general approaches applied for each kind of performance analysis.

Comparison of Inference Speed

As described in Section 4.4, three individual models are developed due to the significant impact of hardware heterogeneity on the resource usage and power consumption of the devices. Therefore, the computation time per prediction during a simulation, i.e. the duration of a single inference call, has to be determined for each model separately. This however enables the comparison of inference time across models.

Simulation Scenarios Since the execution time of a power prediction depends on the complexity of the model and the model varies depending on the device being simulated, one scenario for each edge device type is required, which results in three scenarios in total. For this purpose, the scenarios reproducing the testbed experiments that are already implemented for the accuracy evaluation approach as defined in Section 5.1.3 can be reused. The CPU-based *resnet* function is chosen as serverless function, however, every other function could be selected as well because the inference speed is not influenced by the type of function.

The approach for ascertaining the duration of a single power prediction is rather straightforward but requires some small adjustments of simulator source code. The execution time of each call to the predict method of a model during a simulation is measured using the `timeit` Python package, so the predictions for all requests of a scenario are profiled. Additionally, the individual measurements are logged in order to be able to obtain them after a simulation finishes. Finally, the values can be aggregated by taking the average over all reported computation times. The code for profiling the execution time of the predictions is therefore added to the existing simulation scenarios and removed again after the simulations are performed. In total, three scenarios have to be simulated for determining the inference speed for each of the power models.

Overhead of Power Models on Simulator Execution Time

In addition to analyzing the computation time of a single prediction during a simulation, the total overhead that is caused by the integration of the models into the *faas-sim* framework has to be considered for the performance evaluation of the models. This is essential because the machine learning models do not only delay the runtime of the simulator by making predictions, but also by initially loading them into the simulator before they can be used to forecast any power values. Of course, the model loading is only done once at the beginning of a simulation run, but in case multiple different nodes are simulated, several models have to be loaded. As a baseline for comparison, the average power consumption measured during the experiments is integrated into the simulator. This way, instead of predicting the power consumption upon every function invocation, the mean power draw is reported along with the resource usage values for each request.

Simulation Scenarios As the model loading time and the prediction duration is independent of the function that is simulated, only one function is chosen for the simulation scenarios. Hence, the simulation scenarios previously defined for the comparison of inference speed can be reused and also serve as the basis for analyzing the overhead of the power models. These scenarios include one scenario for each device type, i.e. three scenarios in total, whereby all of them simulate invocations of the CPU-centered *resnet* function.

In order to obtain the simulator execution time of the scenarios where the power models are integrated and used for predictions, the available scenarios can be applied without any changes. Afterwards, source code modifications are needed in order to implement the baseline functionality instead of the prediction models. In contrast to the approach of the inference speed comparison, not the scenarios but the simulator configuration needs to be adopted, so the power models can be removed and the actual power consumption per request on average can be embedded in the framework. The scenarios can then be simulated again with the modified version of the simulator. This enables comparisons between the wall-clock time of the simulations including the models with the ones that simply report a predefined value for the power consumption per request. As a result, six simulation scenarios in total are required for this kind of performance evaluation.

Impact of Models on Simulator Scalability

Assessing the impact of the model integration on the scalability of the simulator can be achieved by simulating scenarios with a varying number of nodes. By observing potential changes regarding the simulator execution time, conclusions about the scaling behavior of the models can be drawn. Since realistic use cases involve topologies that encompass hundreds of nodes, it must be ensured that the wall-clock time of a simulation does not significantly increase due to the rising number of predictions before the models can ultimately be incorporated into the framework. This is particularly important for assuring the time-efficiency of simulations conducted with the *faas-sim* simulator.

Due to the fact that an increasing number of nodes automatically leads to a longer simulator execution time, because more requests have to be simulated, a comparison with a baseline, i.e. a simulator configuration that does not include the power models, is inevitable. Such a baseline version is also implemented for evaluating the overhead of the power models on the simulator execution time as described above, so the changes made to the source code of the simulator are identical and can therefore be reapplied. This enables comparisons between the scaling behavior of the simulator with and without the machine learning models in terms of execution time, which extends the previous evaluation targeting the overhead of the models to a varying number of nodes.

Simulation Scenarios As the simulation scenarios should represent scaling scenarios in which the number of nodes is continuously incremented, one scenario per device type is used as the basis and the subsequent ones are generated by duplicating the topology, i.e. the amount of nodes being simulated. For this purpose, the same scenarios as before, i.e. the *resnet-cpu* function simulated on each device, can be utilized as the starting point and adapted according to the predefined scaling factors.

With respect to the existing scenarios, each topology consists of two nodes, namely the device on which the function invocations shall be simulated and an additional node that enables modeling the data transmission, i.e. sending a request containing an image as input to the device. In order to simulate scaling scenarios, these node tuples are multiplied by 1, 10, 50 and 100. The first scaling scenario therefore represents the existing one, whereas the other ones encompass a varying number of nodes.

All scaling scenarios are then simulated one by one before adaptations to the simulator are made that remove the machine learning models from the framework and include the actual mean power consumption per request instead, as already done for the previous evaluation. Afterwards, all scaling scenarios have to simulated again with the modified version of the simulator, i.e. without the power models. In total, 24 scenarios are used for evaluating the impact of the models on the simulator, since the four scaling scenarios are executed twice, namely with and without the power models, and on each of the three devices. Furthermore, for this kind of evaluation, the topologies of the scenarios as well as the simulator itself have to be adapted.

5.2 Evaluation Results

In the subsequent sections, the results of the individual accuracy and performance evaluations are presented. If not indicated otherwise, all the values contained in the following tables are rounded to two decimal places.

5.2.1 Prediction Accuracy

Accuracy Evaluation using TPOT

Regarding the prediction accuracy evaluation in terms of TPOT, the resulting negative mean absolute error scores of each model on the test data set are summarized in Table 5.1. Hereby, the term *target device* is used to distinguish the three machine learning models based on the underlying edge device the model is tailored to. As mentioned earlier, negating the scoring function is a TPOT-specific method, which is applied to minimize the error as scoring functions naturally try to maximize the accuracy scores.

Target Device	Negative MAE
Intel Xeon	−619.27 mW
Jetson Xavier NX	−458.13 mW
Jetson Nano	−191.04 mW

Table 5.1: Comparison of negative MAE scores on the test sets across devices.

The MAE scores presented in Table 5.1 can give first insights into the accuracy of the models by demonstrating the performance of the models on the test data samples, that are distinct from the training data sets. As indicated in the table, the scores are based on the negative MAE, i.e. the predefined scoring function, and can thus be interpreted in milliwatts. Hence, a MAE score of 619.27 mW, as achieved by the model for the Intel Xeon, means that on average the difference between the predicted and the actual power values is around 620 mW. When comparing the three scores, it becomes clear that the models exhibit different levels of errors. While the highest MAE can be registered for the Intel Xeon, the Jetson Nano model attains the lowest mean absolute error with only about 190 mW. Therefore, the discrepancy between the best and the worst MAE is around 430 mW.

Nevertheless, it can be pointed out that all scores represent acceptable results considering the fact that the measurement deviations of repetitive experiments are even higher in some cases, as identified earlier. For example, previous observations regarding the power measurements of the smart plug connected to the Intel Xeon show that the maximum divergences are around 1.5 W. So even though the models for this device has the poorest rating in terms of prediction accuracy, it is still a promising outcome that is way below the maximum measurement variation. With respect to the Jetson Nano, most measurement distributions exhibit a variance of 200-400 mW, which is potentially the reason for the really good MAE score of the corresponding power model. The results of the Jetson Xavier NX are located somewhere in between the Intel Xeon and the Jetson Nano. These observations in terms of the device ranking also correspond to the internal CV scores of the individual models, as shown in Table 4.12 on page 68. Furthermore, the results of the TPOT evaluation indicate that more complex model generally leads to more accurate results, as it is the case with the Jetson Nano model. In summary, all three models are able to achieve remarkable results on the test set samples, which represent promising prediction accuracies.

Accuracy Evaluation using a Statistical Power Model

With respect to the comparison between the MAE scores of the machine learning models with an existing statistical model, i.e. a linear power model, the results are contained in Table 5.2. In general, the statistical model exhibits a poorer accuracy than the developed platform-specific models on all three devices. The most striking observations that can be derived from this table are the significant differences between the mean absolute errors in terms of the Intel Xeon and the Jetson Nano. While the linear power model exhibits an MAE of over 2.3 W for the Intel Xeon, which is substantially greater than the MAE of the corresponding ML model, the discrepancy regarding the Jetson Nano is even more substantial, since the MAE of the linear model is ten times higher than the one of the ML model for this device. In contrast, the accuracy of the statistical model for the Jetson Xavier NX is only slightly worse than the corresponding TPOT model and therefore also represents the best MAE score across all devices. In fact, the maximum differences between the predicted and the actual power values are around 3.6 W for the Intel Xeon, 1.2 W for the Jetson Xavier NX and 3.3 W for the Jetson Nano. Especially the prediction errors of more than 3 W are considered as dissatisfying and inadequate.

The higher variances between the predicted and the actual power consumption of the linear model presumably stem from the fact that it only incorporates the CPU utilization into the computation, whereby all other resource usage metrics are disregarded. Due to technical advancements of computing infrastructures, particularly in the field of hardware accelerators such as GPUs or TPUs, solely relying on the CPU usage of a system is considered as an insufficient approach for accurately predicting the power draw of modern edge devices.

Target Device	**ML Model MAE**	**Linear Model MAE**
Intel Xeon	619.27 mW	2,343.27 mW
Jetson Xavier NX	458.13 mW	691.03 mW
Jetson Nano	191.04 mW	1,863.72 mW

Table 5.2: Comparison of MAEs between the ML models and the linear statistical power model.

As a result, the existing linear power model that only takes the CPU utilization into account for power consumption forecasts cannot keep up with the developed ML models in terms of prediction accuracy, since these models outperform the statistical model regarding the mean absolute errors. Hence, the linear model might be reliably applicable to certain devices, but it is too generic and too rigid because it does not involve the hardware heterogeneity of different computing platforms. Furthermore, as the power models of other simulators are based on this linear energy modeling technique, it can be deduced that the power forecasting functionality provided by the ML models enables the *faas-sim* framework to deliver more accurate predictions than the other simulators. Consequently, it can be concluded that the platform-specific machine learning models represent a viable and appropriate energy modeling approach for heterogeneous edge devices that achieves a comparatively low level of mean absolute errors.

Comparison of Power Consumption per Request using *faas-sim*

After the TPOT accuracy evaluation and the comparison of the developed models with a statistical power model, the results of the various validations conducted by means of the *faas-sim* simulation framework are presented and compared to the measurements performed during the actual experiments. All of these analyses are facilitated using several simulation scenarios, which reproduce the testbed experiments as defined in the evaluation approach in Section 5.1.3.

First of all, the power consumption per request as forecasted by the ML models during the simulations is compared to the preprocessed power draw measurements obtained of the actual experiments on the testbed. This comparative analysis is done for each model separately, starting with the model developed for Intel Xeon.

Intel Xeon Model Table 5.3 shows the comparison of results for the Intel Xeon power model, whereby the predicted and the real power values for each function are displayed along with the absolute difference between respective values. In order to additionally visualize the outputs, Figure 5.2 contains a barplot of the power predictions and the actual power draw per request. From the table it can be deduced that the absolute difference between the predicted and the true power consumption per request on average on the Intel Xeon ranges from around 11 mW in terms of the *resnet-cpu* function to 1,240 mW as registered for the *efficientnet-gpu* function. The results for the *efficientnet-gpu* function thus represent a large discrepancy. Even though both functions are not contained in the initial training set, the deviation of the forecasted value from the real measured value is over 1,2 W more for the *efficientnet-gpu* function than for the *resnet-cpu* function. This behavior is also clearly evident in the plot shown in Figure 5.2. The other three functions exhibit differences between 200 mW and 400 mW, which represent satisfactory results. From the plot it can also be derived that the predicted values are higher than the actual ones regarding the *resnet-gpu*, *efficientnet-cpu* and *efficient-gpu* function, whereas the true values are higher than the forecasted ones with respect to the *resnet-cpu* and *objectdetection* function. Hence, in this sense, no clear tendency can be observed for the results of the Intel Xeon-based power model.

Function	**Pred. Power**	**Actual Power**	**Absolute Diff.**
Resnet-cpu	17,216.55 mW	17,227.16 mW	10.62 mW
Resnet-gpu	15,972.78 mW	15,767.98 mW	204.80 mW
Efficientnet-cpu	16,517.23 mW	16,318.54 mW	198.69 mW
Efficientnet-gpu	16,979.84 mW	15,740.80 mW	1,239.04 mW
Objectdetection	16,933.43 mW	17,341.99 mW	408.56 mW

Table 5.3: Comparison between the predicted and the actual power consumption per request for the Intel Xeon model.

In summary, it is important to highlight that there is one outlier prediction, i.e. the one for the *efficientnet-gpu* function, which is considerably inferior compared to the other predicted values of the Intel Xeon model. However, taking the maximum measurement error of 1.5 W for the Intel Xeon smart plug into account, this output is still inside the tolerance range. Sine the measurement variance of 1.5 W is recorded for the *resnet-gpu* measurements and this function is contained in the training set, the poor prediction for the *efficientnet-gpu* function can be attributed to this mismatch, as both functions are GPU-centered. The other predictions are all below the MAE as previously determined by TPOT, so these values are highly satisfactory.

Jetson Xavier NX Model Regarding the power model constructed for the Jetson Xavier NX, similar findings as observed for the Intel Xeon model can be derived from the results presented in Table 5.4 and the corresponding plot shown in Figure 5.3. While

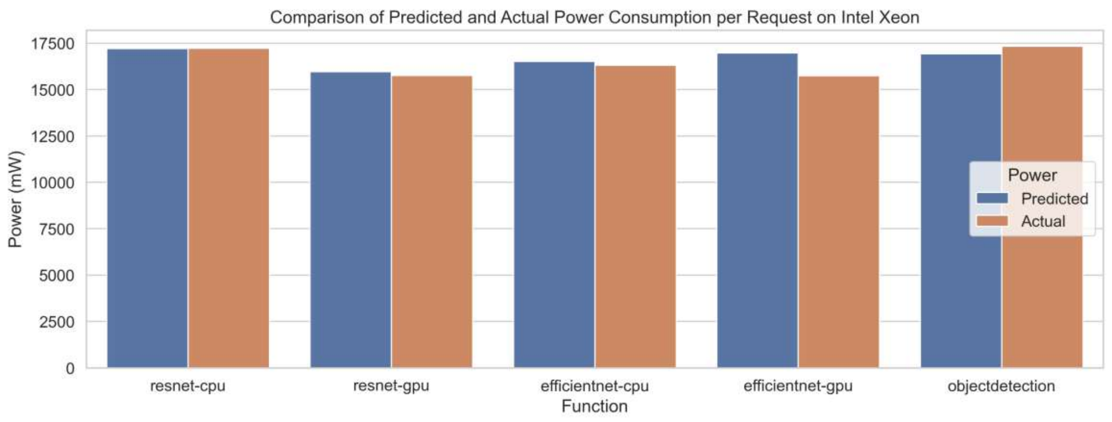


Figure 5.2: Comparison between the predicted and the actual power consumption per request for the Intel Xeon model.

the predicted power values for four of the functions are satisfactory, since the absolute difference between the forecasted and the actual power consumption are on a very low level, the prediction made for the *efficientnet-gpu* function is striking. Similar to the previously analyzed model, the power consumption per request predicted for the *efficientnet-gpu* function misses the actually measured average value by more than 1 W. Nevertheless, in contrast to the Intel Xeon model, the machine learning model developed for the Jetson Xavier NX achieves the best prediction, i.e. the one closest to the true value, for the *resnet-gpu* function.

Function	**Pred. Power**	**Actual Power**	**Absolute Diff.**
Resnet-cpu	15,458.15 mW	15,359.42 mW	98.74 mW
Resnet-gpu	16,535.50 mW	16,507.14 mW	28.36 mW
Efficientnet-cpu	15,019.83 mW	15,137.78 mW	117.95 mW
Efficientnet-gpu	16,690.36 mW	15,604.01 mW	1,086.35 mW
Objectdetection	15,019.83 mW	15,078.63 mW	58.80 mW

Table 5.4: Comparison between the predicted and the actual power consumption per request for the Jetson Xavier NX model.

Besides the single poor prediction, this model also accomplishes outstanding results. When looking at the plot displayed in Figure 5.3, only minimal differences between the predictions and the true values can be identified for four out of five functions. The comparably large discrepancy in terms of the *efficientnet-gpu* function is clearly visible in the plot as well. In order to find out the exact cause for this mismatch, additional investigations regarding the resource usages of the individual functions would be required, which is out of the scope of this work. However, a prediction error of 1 W is still

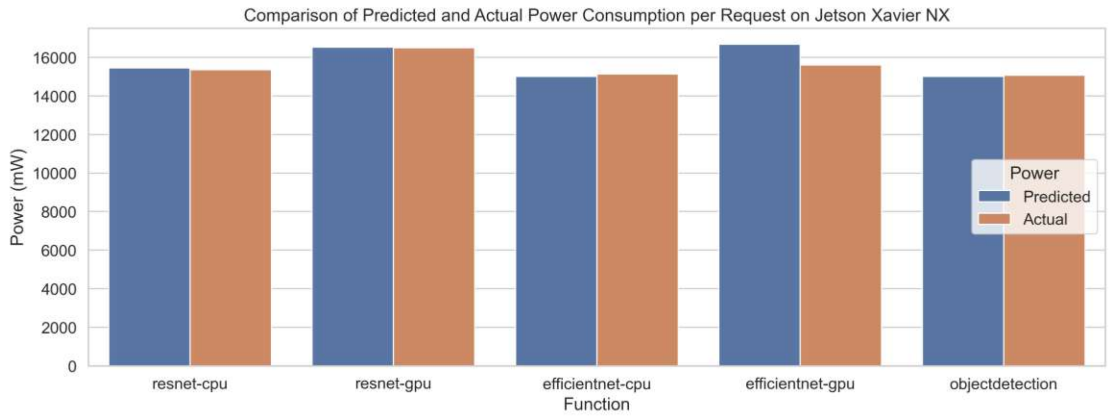


Figure 5.3: Comparison between the predicted and the actual power consumption per request for the Jetson Xavier NX model.

considered as tolerable when keeping the measurement deviations in mind. In general, it can be also be noted that the predictions of the Jetson Xavier NX model are more accurate than the ones obtained from the Intel Xeon model, which corresponds with the findings of the TPOT accuracy evaluation.

Jetson Nano Model Finally, the difference between the predicted and actual power values of the machine learning model tailored to the Jetson Nano is assessed. The evaluation results are presented in Table 5.5 and Figure 5.4, respectively. As a general observation, it can be stated that the prediction errors are extremely low for this model across all functions, whereby the minimum deviation is less than 2 mW and the maximum divergence is around 26 mW.

Function	**Pred. Power**	**Actual Power**	**Absolute Diff.**
Resnet-cpu	3,820.83 mW	3,838.20 mW	17.36 mW
Resnet-gpu	3,657.88 mW	3,656.06 mW	1.82 mW
Efficientnet-cpu	3,675.70 mW	3,653.22 mW	22.48 mW
Efficientnet-gpu	3,644.82 mW	3,637.37 mW	7.45 mW
Objectdetection	3,607.41 mW	3,581.05 mW	26.36 mW

Table 5.5: Comparison between the predicted and the actual power consumption per request for the Jetson Nano model.

This behavior can probably be attributed to the fact that the power consumption values of the Jetson Nano are the most homogeneous ones with a low level of measurement errors. As clearly evident in the plot in Figure 5.4, the power values of the individual workloads do not differ as much as they do with respect to the Intel Xeon and the Jetson

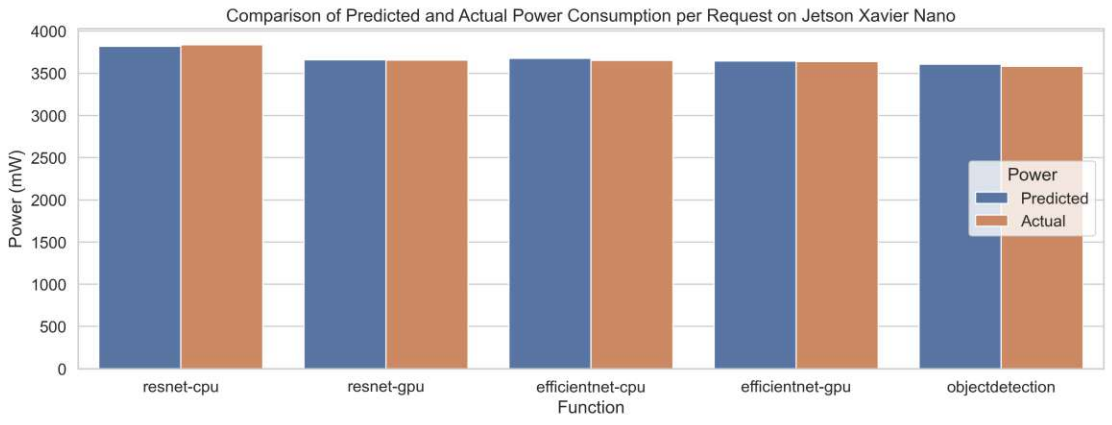


Figure 5.4: Comparison between the predicted and the actual power consumption per request for the Jetson Nano model.

Xavier NX. Furthermore, the TPOT MAE score already indicates that the predictions of the Jetson Nano model are the most accurate ones. Another interesting finding is that except for the *resnet-cpu* function, all other predictions are higher than the actual power consumption per request measured for the different serverless functions.

Summary In conclusion, by comparing the predictions yielded by the machine learning models during simulations with the real-world measurements conducted by means of the experiments, it becomes clear that the predicted values generally correspond to the actual power readings for most serverless functions. While there are in total two outlier predictions, these mismatchs still reside in a tolerable range with respect to the measurement errors of the smart plugs as determined earlier. Therefore, the power models are able to predict the power consumption per request with a satisfying accuracy. However, no generally valid assertion about a tendency of the models to consistently predict values higher or lower than the real power measurements can be made. Furthermore, the exact accuracy of the models slightly differ, as already indicated by the TPOT evaluation.

Comparison of Total Energy Demand using *faas-sim*

Similar to the comparison of the predicted and actual power consumption per request, assessing the differences between the total energy demand estimated by the simulator with the total amount of energy actually consumed during the experiments is also done for each model individually. These differences can help with gaining additional knowledge about the accuracy of the prediction models. Since joule is the most common unit of energy, joules (J) are used in the following instead of watt-seconds, but both units are equivalent in general. Apart from the overall energy, the period of time that is simulated by the *faas-sim* framework for a certain scenario can be contrasted with the real-world duration of the corresponding experiment. This can lead to valuable insights regarding the comparability of the scenarios simulated through the framework, which should reproduce the experiments conducted by means of the testbed as accurately as possible.

Intel Xeon Model At first, the results of the model constructed for the Intel Xeon are analyzed. Table 5.6 therefore comprises the duration modeled by the simulator and the actual length of the experiment for each function, as well as the total amount of energy estimated based on the prediction model and the one based on the real-world measurements.

Function	Sim. Time	Act. Time	Pred. Energy	Act. Energy
Resnet-cpu	286.50 s	293.75 s	4,679.80 J	4,796.98 J
Resnet-gpu	286.50 s	293.74 s	4,648.15 J	4,761.55 J
Efficientnet-cpu	286.50 s	293.74 s	4,661.18 J	4,773.84 J
Efficientnet-gpu	286.50 s	293.58 s	4,671.61 J	4,757.27 J
Objectdetection	200.50 s	209.68 s	3,301.40 J	3,476.86 J

Table 5.6: Comparison between the estimated and the actual energy demand for the Intel Xeon model.

Regarding the total duration, both the simulated and the actual time are very similar for all *resnet* and *efficientnet* functions since the same request pattern is used for these workloads. The request profile for the *objectdetection* function differs, which is why the both durations are lower. When comparing the estimated period of time with the true execution time of the experiments on average, it becomes clear that the simulated values are 7-9 seconds lower. This behavior can be observed for all functions and probably stems from certain setup and teardown tasks that are performed by the Galileo experimentation framework before the first request and after the last request. This assumption is validated through manual checks, which confirm that additional 2-3 seconds pass between the start of the experiment and the transmission of the first request. Additionally, 5-6 seconds elapse between finishing the last request and stopping the experiment. These setup and teardown delays are therefore not reflected by the *faas-sim* framework.

When it comes to comparing the predicted total energy demand with the actually consumed energy, deviations between 85 J and 175 J can be discovered for the Intel Xeon model. Therefore, the differences are on a consistently low level, which means that the simulations are able to accurately estimate the total amount of energy required for a certain scenario. Since the forecasted energy demand is lower than the real energy usage for all functions, additional investigations are carried out to find out the reason for this behavior. As stated above, the duration estimated by the simulator is about 7-9 seconds shorter than the actual execution time of the experiments. Because the total energy demand is calculated based on the function execution time and the idle duration, it is assumed that the idle period of the real experiments is higher than the estimated idle time due to the experiment setup and teardown delays as identified earlier. This assumption is verified by comparing the idle periods of the simulations with the ones registered during the experiments. The results of these additional validations show that the idle time of the experiments is indeed 7-9 seconds longer than the estimated idle period. Adapting the computations according to this finding for test purposes yields even more accurate results for the total energy demand. Since the biggest difference between the estimated and the true duration can be identified for the *objectdetection* function, it is plausible that the predicted and the actual energy values of this function also diverge more than the ones calculated for the other functions.

Nevertheless, the initial results contained in Table 5.6 already represent very satisfactory outcomes, which show that the Intel Xeon model can be applied for accurately estimating the total energy demand of a scenario. In general, it can also be observed that the function executions on the CPU require slightly more energy than the corresponding GPU experiments on the Intel Xeon, which represents an interesting finding in terms of energy efficiency.

Jetson Xavier NX Model Table 5.7 comprises the total energy demand based on the predicted power values and based on the power consumption measurements of the experiments on the Jetson Xavier NX. Furthermore, the period of time simulated by the scenarios as well as the actual execution time of the corresponding experiments on average is included in the table. Regarding the durations, the conclusions that can be drawn from the comparison between the simulated and the real-world values are identical to the findings that are documented for the Intel Xeon model, which is why they are not repeated at this point.

Function	**Sim. Time**	**Act. Time**	**Pred. Energy**	**Act. Energy**
Resnet-cpu	286.50 s	293.94 s	4,264.14 J	4,366.58 J
Resnet-gpu	286.50 s	294.27 s	4,274.40 J	4,386.84 J
Efficientnet-cpu	286.50 s	293.74 s	4,227.09 J	4,339.26 J
Efficientnet-gpu	286.50 s	293.86 s	4,282.81 J	4,350.87 J
Objectdetection	200.50 s	210.04 s	2,971.09 J	3,115.54 J

Table 5.7: Comparison between the estimated and the actual energy demand for the Jetson Xavier NX model.

With respect to the total energy demand, the insights that can be gained from the results of the Jetson Xavier NX model are also similar to the ones of the Intel Xeon model. The estimated total energy demands are consistently lower than than the actually consumed energy, which is again caused by the differences in terms of execution time. As discovered above, the reason for this mismatch is the fact that the simulated duration is shorter than the real runtime for all functions. The specific discrepancies between the estimated and the actual amount of energy range between 68 J and 145 J, whereby the *objectdetection* function again exhibits the biggest deviation. In general, the level of differences is lower than the ones calculated for the Intel Xeon model, which corresponds with the previous findings regarding the slightly varying accuracy of the individual machine learning models.

In contrast to the Intel Xeon, the total amount of energy consumed by the CPU-based functions is lower than the energy required for the GPU-based functions. However, both devices exhibit a comparable level of energy demand with regard to the experiments conducted in the course of this work.

Jetson Nano Model Finally, Table 5.8 presents the results of the energy demand comparison based on the power model developed for the Jetson Nano. In fact, the differences between the estimated energy demand and the actually consumed energy range from 24 J to 28 J, so the Jetson Nano model yields even more accurate results than the Jetson Xavier NX model, which coincides with the key findings of the preceding accuracy evaluations.

Function	**Sim. Time**	**Act. Time**	**Pred. Energy**	**Act. Energy**
Resnet-cpu	286.50 s	293.58 s	1,025.83 J	1,051.77 J
Resnet-gpu	286.50 s	294.11 s	1,004.22 J	1,030.71 J
Efficientnet-cpu	286.50 s	293.91 s	1,002.62 J	1,027.08 J
Efficientnet-gpu	286.50 s	294.43 s	1,001.11 J	1,027.91 J
Objectdetection	200.50 s	209.52 s	709.62 J	737.97 J

Table 5.8: Comparison between the estimated and the actual energy demand for the Jetson Nano model.

Since the Jetson Nano has a lower power consumption on average both during idle state and during function invocations, the total amount of energy consumed by this edge device is significantly lower than the energy demand of the Intel Xeon and the Jetson Xavier NX, although the Jetson Nano takes longer for processing a single request. This represents a notable observation, since the power models embedded into the *faas-sim* framework can therefore also be used to compare the energy efficiency of different device types.

Summary The results of the energy demand comparison represent remarkable outcomes for all three models, which demonstrate and thus certify that the developed machine learning models can be applied for accurately estimating the total amount of energy required for a certain scenario. This in turn confirms that the power models can reliably be used to determine and compare the energy efficiency of different devices. As elucidated above, the model tailored to the Jetson Nano performs best in terms of inference accuracy across all evaluations, whereas the Intel Xeon model exhibits the poorest forecasting precision. However, the prediction errors are still very low with respect to the measurement errors of the smart plugs, so all models are able to yield satisfactory outcomes.

5.2.2 ML Model Performance

The model performance evaluation also consists of multiple parts in order to provide an in-depth analysis of the complexity of the power models and the resulting level of overhead they impose on the simulator execution time. If the models turn out to produce a considerable delay, it could have negative effects on the scalability of the simulation framework. All the performance assessment results presented in the following are based on simulations conducted with the *faas-sim* framework and executed on a MacBook Pro as described earlier.

Comparison of Inference Speed

The average runtime of a single prediction per model can be found in Table 5.9, whereby the results are calculated by measuring the inference speed of each prediction within a scenario comprising 100 requests and aggregating the values using the mean. In general, it can be noted that all three models make very fast predictions in the range of a few milliseconds. When comparing the inference time of all three models, clear differences can be identified. While the Intel Xeon-based model makes the fastest predictions, which only run for 0.78 ms on average, the model developed for the Jetson Nano takes over three times longer to perform a single inference call and thus requires 2.6 ms per prediction. The inference time of the Jetson Xavier NX model lies between these two values.

Target Device	Inference Speed
Intel Xeon	0.78 ms
Jetson Xavier NX	1.16 ms
Jetson Nano	2.60 ms

Table 5.9: Comparison of the average inference speed of each model.

These findings correspond with the assertions made about the complexity of the models in Section 4.4.2, where it is stated that the model pipeline of the Intel Xeon is the most compact one, whereas the TPOT pipeline for the Jetson Nano has the highest number of operators. Consequently, as expected, the complexity of the models clearly has an impact on the inference speed. Furthermore, the length of the model training, i.e. the duration of the individual TPOT runs, can also be related to these insights, which leads to the assumption that a longer training period implies a more complex and thus slower, but also more accurate model. As shown in Section 5.2.1, the Jetson Nano model exhibits the highest accuracy across all three ML models, but also has the longest pipeline optimization period, the most exhaustive pipeline and therefore the slowest inference speed. In comparison, the Intel Xeon model is the most compact one with the fastest training phase and the best inference speed while its predictions are not as accurate as the ones yielded by the Jetson Nano model. Hence, the results clearly show the interrelationship between the model training duration, the complexity of the final pipelines, the inference speed and the prediction accuracy of the models. As a consequence, these observations are presumably the reason for the varying inference time of the models. However, the exact implications of the differences in terms of inference speed only become visible through the subsequent evaluations.

Overhead of Power Models on Simulator Execution Time

The preceding evaluation only considers the computational burden caused by a single power prediction, but the integration of the models additionally adds overheads to the simulator execution time, since the models need to be loaded before they can be used for power forecasting. Therefore, following evaluation takes both delays into account by determining the overhead of the models in comparison with a baseline approach that does not include the power models. As described in Section 5.1.4, the baseline configuration of the simulation framework simply reports a predefined value for the power consumption, namely the average power draw per request, instead of making predictions.

Table 5.10 includes the results of the simulator execution time of scenarios that are simulated with and without the machine learning models. In order to illustrate the differences in terms of simulator runtime, the resulting values are also plotted as shown in Figure 5.5. The scenarios used for these comparisons involve the simulation of 100 requests of the same serverless function on each device type. While the wall-clock times of the scenarios executed without the models are fairly similar across devices, the simulator runtimes regarding the simulator configuration with the integrated power models exhibit more variations. Nevertheless, the magnitude of these differences is rather small as the minimum and maximum value are only about 250 ms apart. These slight discrepancies certainly stem from the fact that the Jetson Nano model is the slowest model with the longest inference time of all three devices as indicated in Table 5.9, whereas the Intel Xeon model is the fastest one in terms of inference speed.

Target Device	**With Models**	**Without Models**
Intel Xeon	306.37 ms	83.91 ms
Jetson Xavier NX	378.44 ms	88.73 ms
Jetson Nano	553.12 ms	86.89 ms

Table 5.10: Comparison of the overhead imposed by the models on the simulator execution time.

In general, depending on the device and thus the model used for predictions, the execution time of the simulations in case the models are included is approximately 3.5-6.5 times higher than the simulator runtime of the baseline configuration without the models. Even

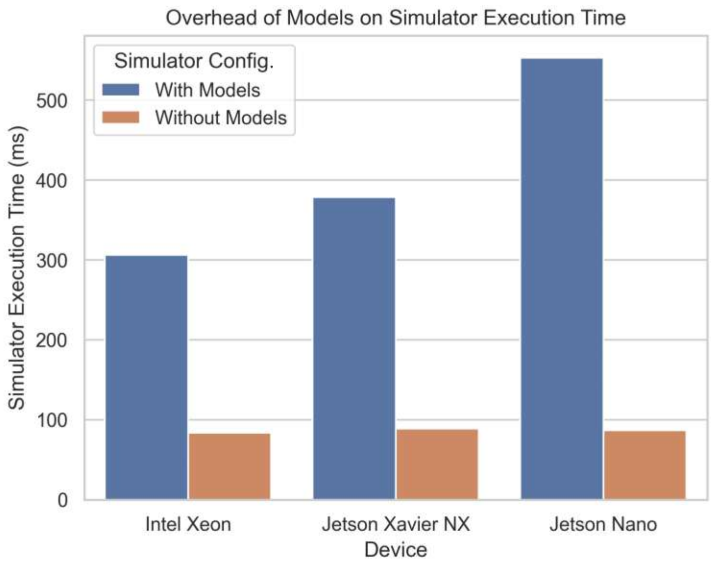


Figure 5.5: Overhead caused by the models on the simulator execution time.

though the overheads might seem severe at first sight, they have to be considered in proportion to the overall context. Since the simulations with the power models encompass 100 requests, which implies the computation of 100 predictions and the loading of the respective model at the beginning, the absolute values of the delays are plausible. Due to the fact that further functionality is added to the simulator, an increased wall-clock time is expectable and thus justifiable. Furthermore, the simulator wall-clock times recorded for the simulations including the models are still extremely low compared to the actual period of time they simulate. While the simulations run between 300 ms and 550 ms, the duration that these scenarios are simulating amounts to 286.5 s, so the simulator operates very time-efficiently despite the overhead caused by the models.

To sum up, the integration of the machine learning models has a reasonable impact on the execution time of a simulation, since the benefit the models add to the simulation framework prevails and thus justifies the delays imposed by the predictions and the model loading time.

Impact of Models on Simulator Scalability

Simulating scenarios with a varying number of nodes represents another interesting evaluation that can be performed to assess the level of overhead caused by the models. Since 100 requests are simulated per node tuple and the tuples are multiplied by the scaling factor, a factor of one implicates scenarios with 100 requests in total, whereas a factor of 100 implies that 10,000 requests are modeled during a simulation. The scaling behavior of the models in terms of simulator wall-clock time can therefore be observed and the results can be compared to the runtimes recorded for the baseline configuration.

This way, the impact of the models on the scalability of the *faas-sim* framework can be assessed.

The simulator execution time of the individual runs can be found in Table 5.11. For the purpose of visualizing the results, Figure 5.6 encompasses one plot for each device, i.e. each machine learning model. According to these plots and the exact values contained in the corresponding table, the wall-clock times of the scaling scenarios without the models, which are indicated by the orange lines in the plots, are pretty much identical across all three devices. However, when comparing the simulator runtimes of the scaling scenarios where the models are included, as displayed by the blue lines in the plots, it becomes evident that the Jetson Nano model performs worse than the other two models. The integration of the Jetson Xavier NX model does not increase the simulator wall-clock time as much as the Jetson Nano model, but still more than the Intel Xeon model. These findings again correspond with the previous observations, namely the finding that the Jetson Nano model has the slowest inference speed, whereas predictions for the Intel Xeon take the least amount of time.

Target Device	**Scaling Factor**	**With Models**	**Without Models**
Intel Xeon	1	0.31 s	0.08 s
Intel Xeon	10	2.03 s	0.93 s
Intel Xeon	50	11.95 s	7.09 s
Intel Xeon	100	29.87 s	20.49 s
Jetson Xavier NX	1	0.38 s	0.09 s
Jetson Xavier NX	10	2.34 s	0.92 s
Jetson Xavier NX	50	13.64 s	7.37 s
Jetson Xavier NX	100	33.32 s	20.14 s
Jetson Nano	1	0.55 s	0.09 s
Jetson Nano	10	3.39 s	0.91 s
Jetson Nano	50	18.85 s	7.13 s
Jetson Nano	100	45.42 s	20.56 s

Table 5.11: Comparison of the execution time with and without the models based on the scaling scenarios.

Furthermore, all simulations conducted with the models are considerably slower than the simulations performed without the models. Regarding the scenarios with a small number of nodes, the slight delays caused by the integration of the models is considered as reasonable. The magnitude of the differences however increases with rising numbers of nodes because of the higher amounts of requests, which in turn imply a larger number

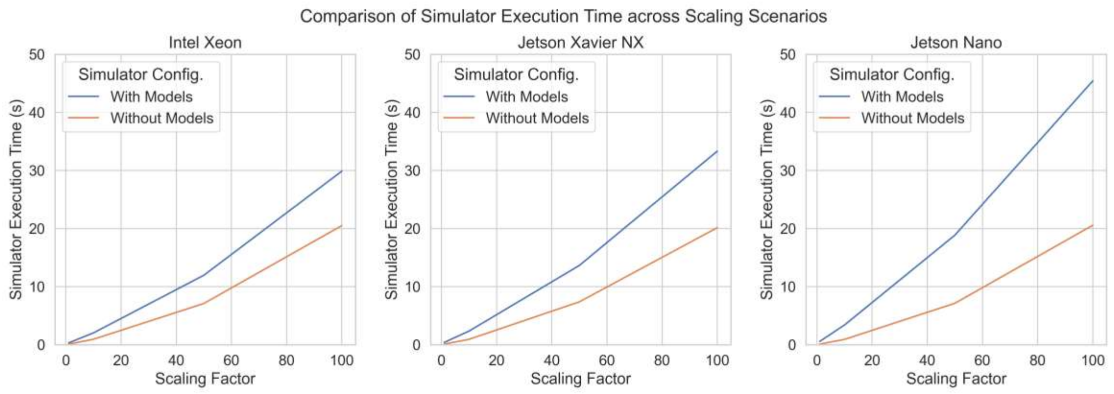


Figure 5.6: Comparison of simulator execution time across scaling scenarios.

of predictions. Hence, the biggest divergences with respect to the simulator execution time can be observed for the scenarios with a scaling factor of 100. Regarding the Intel Xeon model, the integration of the models causes a delay of more than 9 seconds for the scenario with 100 node tuples. The predictions made with the Jetson Xavier NX model impose a latency of around 13 seconds, whereas the Jetson Nano model increases the simulator wall-clock time by nearly 25 seconds. This behavior can also be recognized in the plots shown in Figure 5.6. Nonetheless, when taking the amount of time that is simulated by these scenarios into account, it can be concluded that the simulations conducted with the power models are still very time-efficient. Since the simulations with a scaling factor of 100 simulate scenarios with a total duration of 28,602 seconds, i.e. nearly 8 hours, a wall-clock time of 45 seconds is still bearable and thus acceptable.

Summary

In conclusion, the machine learning models impose diverse levels of overheads on the simulator execution time, which is confirmed by all evaluations results. Especially in scenarios where large numbers of nodes are simulated, which represents a realistic scenario with respect to typical edge computing use cases, the delays are clearly perceptible. However, even though every model negatively impacts the wall-clock time of the simulator, the scalability of the *faas-sim* framework is not severly impaired, since the degree of latencies is still reasonable considering the significant benefit the models add to the simulations. Furthermore, when taking the amount of time that is simulated by these scenarios into account, it can be concluded that the simulations are very time-efficient despite the model overhead and that the extended execution times are therefore acceptable.

5.2.3 Performance-Accuracy Trade-off

After the accuracy of the predictions and the overall performance of the models are assessed separately, the findings can be combined in order to reason about the inherent trade-off between these two essential ML model properties. This trade-off can give valuable insights into the applicability and the suitability of the power models for the given simulation framework. On the one hand, the predictions should exhibit a satisfying level of precision, which means that the predicted power consumption during simulations does not significantly diverge from the actual power consumption of the devices in the real world. However, on the other hand, the complexity of the models is crucial as well, since this affects the inference speed, i.e. the performance of the models. The models should therefore only add a minor overhead to the simulations, so that the simulator scalability is not severely impaired.

While the results demonstrate that the models are able to achieve remarkable outcomes in terms of accuracy, as thoroughly analyzed in Section 5.2.1, the performance of the models, which directly impacts the overhead imposed on the simulator execution time, considerably differs depending on the number of nodes that are simulated in a scenario. However, in order to correctly interpret the results, they have to be put into perspective and thus considered in proportion to the overall context, as explained in Section 5.2.2. Since the prediction functionality is a major enhancement of the simulator and the model accuracy is in general very satisfactory, the increased wall-clock times of the simulations are reasonable. Additionally, the simulator can still finish the simulations within an acceptable amount of time, which is why the simulation runs including the power models are very time-efficient regardless of the overhead they impose. Therefore, when keeping the actual period of time these scenarios simulate in mind, even the delays recorded for larger numbers of nodes are tolerable. As a result, the scalability of the simulator is slightly affected by the models, but not severely impaired.

Since the models are able to accurately estimate the power consumption of a single serverless function invocation and can therefore be applied to reliably calculate the total amount of energy required for a certain scenario, they should be provided as an additional simulator feature despite the overhead they cause. Due to their ability to predict the power consumption of a request independent of the underlying serverless function and solely based on the resource usage metrics, the machine learning models are generalizable to unknown functions and thus constitute a valuable extension to the *faas-sim* framework. Furthermore, the accuracy evaluation results also show that the platform-specific machine learning models represent a viable and appropriate energy modeling approach for heterogeneous edge devices, which is able to outperform an existing statistical power model. The multi-model approach chosen in this work can thus overcome the device-dependent discrepancy in terms of resource usage and power consumption that is associated with the edge domain.

As it is anticipated that the majority of scenarios, which are simulated by means of the *faas-sim* simulation framework, comprise large-scale topologies with several nodes, the power prediction functionality of the machine learning models should not necessarily be activated by default. Instead, it is suggested to offer this feature as an optional simulator extension that can be enabled if desired. Otherwise, the power models could increase the simulator execution time even if they are not needed by every user. Hence, as a compromise, it should be left to the users whether they want to include this additional functionality into their simulations or not. Besides that, the implications of integrating the power models into a scenario on the simulator runtime must be clearly documented for potential users of the *faas-sim* framework, since time-efficiency is one of the main objectives of a simulation.

Moreover, from the results presented in Section 5.2.1 and Section 5.2.2, conclusions regarding the performance-accuracy trade-off of the individual models can be drawn. The evaluations clearly show that the Jetson Nano model is the most accurate one, whereby it also imposes the biggest overhead due to its slower inference speed. These observations are direct causes of the elevated model complexity as compared to the other two ML models. In contrast, the more compact model established for the Intel Xeon performs best in terms of inference time across all evaluations and thus imposes less overhead on the simulator wall-clock time, but at the same time it exhibits larger, but still bearable, deviations in terms of forecasting precision. Therefore, it can be stated that a higher prediction accuracy generally can only be achieved at the cost of model complexity and therefore weaker model performance. Nevertheless, the models demonstrate that TPOT enables the construction of lightweight ML models which are able to balance the accuracy-complexity trade-off, whereby a satisfactory trade-off between model performance and accuracy is a key requirement imposed by incorporating the models into the *faas-sim* simulator. This achievement can be attributed to TPOT's internal Pareto optimization and the TPOT Light configuration used for the model establishment phase.

In summary, the integration of the developed machine learning models into the *faas-sim* simulation framework proves that it is possible to establish power prediction models with an acceptable performance-accuracy trade-off. While these models are able to precisely estimate the power consumption of a function invocation solely based on the provided resource usage metrics, they do not severely impair the scalability of the simulator. Even though the overhead imposed by the additional power forecasting functionality increases with rising numbers of nodes contained in a scenario, which represents a plausible behavior, the time-efficiency of the simulations can still be guaranteed. As a result, the applicability and suitability of the models in simulation environments is verified by the evaluation results. However, the benefit that the models add to the simulation framework, i.e. providing power predictions for resource planning and energy efficiency comparisons, needs to be balanced with the extended simulator execution time that is caused by the computation overhead of the models. Consequently, as a compromise, the power prediction feature should only be embedded into the *faas-sim* as an optional simulator extension.

CHAPTER 6

Limitations

In the following, known limitations of the approach applied in this work and restrictions of the final power models are highlighted and discussed. Thereby the limitations are divided into constraints related to the power consumption and resource usage measurements on the one hand, as presented in Section 6.1, and restrictions concerning the integration of the models into the *faas-sim* framework on the other hand, as covered in Section 6.2.

6.1 Measurement-related Limitations

Regarding the smart plugs used in the course of this thesis to measure the power consumption of edge devices, several limitations need to be noted. First of all, the sampling frequency of ten seconds is in general too coarse, since the telemd resource usage metrics are reported once per second during the experiments. In an ideal setup, the time of the power consumption readings should be aligned and therefore synchronized with the telemd sampling rate, so the timestamps of the power values correspond with timestamps of the resource usage metrics. Nevertheless, this constraint of the smart plugs is compensated by using the PowerDelta configuration of the Tasmota firmware running on the plugs, which is able to publish all energy-related metrics upon power changes of more than one percent in addition to the periodic reports. Consequently, it can be assumed that the power consumption between two consecutive measurements did not change more than one percent. However, this leads to irregular power readings as compared to the steady resource usage publishments, which represents an inherent shortcoming of the smart plugs.

Furthermore, two of the smart plugs had to be replaced due to hardware malfunctions that caused the plugs to report unrealistic power consumption values. Therefore, the overall reliability and correctness of the power readings returned by the smart plugs could be questioned and should thus be verified by other external power measurement instruments. Considering the striking findings derived from the comparisons between the external and internal measurements, the need for such a verification is further amplified.

Another caveat that needs to be mentioned is the assumption that the Jetson NVP models, which define the operation mode of the underlying Jetson module, presumably have an impact on the power consumption of a device. As the specified NVP profile also determines the power budget, setting a different NVP mode could potentially lead to power consumption readings that differ from the ones obtained in the course of this work, which adds another level of heterogeneity. This would imply that the developed power models for the Jetson boards do not only depend on the platform, but also on the NVP model that is currently set on the devices, which could result in inaccurate predictions in case other NVP profiles are chosen. However, this is only a hypothesis, so the actual implications of changing the operation modes of the Jetson Xavier NX and the Jetson Nano are unknown at this juncture.

In addition, the data analysis results presented in Section 4.3 indicate that the GPU utilization metric is rather volatile and more error-prone than the CPU measurements. Furthermore, since the GPU utilization metric does not represent a continuous counter, it may not be able to capture rapidly changing values. Hence, this metric is highly dependent on the timing of the measurements and the sampling frequency. This however does not present a limitation of this work, but a shortcoming of the telemd monitoring agent. Reporting the GPU usage time as a continuous counter instead of the current GPU utilization would therefore be a highly desirable improvement of the telemd daemon.

Moreover, especially the GPU-targeted functions executed on the Intel Xeon exhibit relatively low utilization rates, since the provided workloads are not sufficient to heavily stress the powerful GPU. In general, all of the profiled serverless functions represent inference tasks of deep learning models, so the resource usage and power consumption of training processes performed on edge devices is not considered in this work. As the training phase usually takes significantly longer than performing inference with a pre-trained model, exact measurements of inference applications are more difficult due to their short runtime. Consequently, as compared to inference, training a ML model is typically more resource-intensive, so additionally profiling serverless functions that focus on training on the available edge nodes could be beneficial in terms of measurement robustness and higher GPU workloads.

6.2 Simulator-related Limitations

With respect to the additional power prediction functionality of the *faas-sim* simulation framework, this novel simulator feature is restricted to the three edge devices used in this work, i.e. the Intel Xeon, the Jetson Xavier NX and the Jetson Nano. Due to the severe hardware heterogeneity these platforms exhibit, the developed models are tailored to one type of device. Therefore, predicting the power consumption per request is only supported for these three devices, whereas other hardware platforms that can also be simulated by means of *faas-sim* do not include this functionality.

Although the *faas-sim* framework is able to simulate the parallel invocation of multiple serverless functions on a single node, this functionality is not utilized in the course of this work. The reason for this is the fact that the simulations conducted with the *faas-sim* simulation for evaluation purposes aim at reproducing the testbed experiments. Since the testbed experiments focus on profiling single and isolated requests, the simulations do not need to simulate more than one request at a time. Hence, only single tenancy is considered in this thesis, while multi-tenancy scenarios are disregarded. Nonetheless, if the multi-tenancy feature of the simulator is intended to be used in combination with the power models, taking the resource usage of all parallel function calls, predicting the power consumption for each one individually and adding them up is most likely not the correct approach, because the models predict the total power consumption of the device instead of the difference between the baseline and the function invocation power draw. As an alternative, the combined resource usage of the parallel requests needs to be determined and fed into the power models in order to obtain more accurate results. However, due to the fact that this kind of scenario is not analyzed in the scope of this work, separate evaluations would be required to assess the correctness and feasibility of the described approach. As a result, this does not represent a real limitation, but it should be kept in mind that scenarios with parallel function invocations are not covered and thus not tested in the course of this thesis.

CHAPTER 7

Conclusion

Accurately and consistently measuring the power consumption of edge devices is a challenging and laborious task due to the high hardware heterogeneity that is associated with computing platforms operating in the edge-cloud continuum and the lack of uniform, platform-agnostic measurement facilities. However, obtaining the power draw of devices is crucial for optimizing the energy efficiency and reducing operational costs in edge environments. Therefore, this thesis aims to provide an easy to use prediction functionality embedded into an existing open-source FaaS simulation framework, which enables forecasting the power usage of different edge computing platforms in the context of realistic large-scale scenarios.

To this end, the development of power prediction models based on resource usage metrics by means of machine learning techniques is proposed for efficiently modeling the energy consumption of heterogeneous edge devices. This work therefore encompasses the design, construction and evaluation of ML models which rely on empirical measurements conducted during a series of experiments on a dedicated testbed. For this purpose, an AutoML tool, namely TPOT, is applied during the model development process. In order to enable the utilization of the models in typical edge computing use cases, they are integrated into the *faas-sim* serverless simulator.

The evaluation results demonstrate the successful establishment of generalizable ML models that are able to precisely estimate the power consumption of serverless function invocations solely based on resource utilization rates. The models therefore generalize well to new inputs from unknown serverless functions. At the same time, the extended simulator execution time, which stems from the additional power forecasting functionality, is still reasonable. Hence, the prediction models do not severely impair the scalability of the underlying simulation framework. Considering the actual period of time the scenarios simulate, the time-efficiency of the simulations can thus be guaranteed regardless of the overhead that is caused by the predictions. As a result, the findings show that efficient machine learning models with a satisfactory performance-accuracy trade-off can be established and therefore represent an appropriate energy modeling technique for heterogeneous edge devices.

Consequently, the supplementary prediction functionality embedded into the *faas-sim* framework represents a major enhancement to the simulator. As such, it can be useful for energy management and planning, which is crucial for settings where energy awareness is a key aspect, e.g. due to limited power supply. Furthermore, it allows developers and researchers to rapidly and easily compare the energy efficiency of various devices and facilitates the development and evaluation of novel energy-aware scheduling, load balancing and task placement strategies as well as other resource management algorithms. In summary, the power models foster the development of energy-efficient solutions and thus contribute to the goal of optimizing and reducing energy demands as targeted by research efforts in the context of green IoT.

7.1 Research Questions

This section summarizes the answers to the main research questions that should be answered in the course of this thesis and thereby highlights the key findings and contributions of this work.

RQ 1: What are appropriate methods for measuring the power consumption of edge devices and how do they differ?

Since the edge domain is characterized by highly heterogeneous computing platforms, power profiling of edge devices is typically complex, time-consuming and costly. For the purpose of finding appropriate methods that are suitable for measuring the power consumption of various platforms, different types of measurement facilities are described, classified into external and internal instruments and contrasted by means of various aspects in the course of this thesis. The factors that build the basis for this comparison include granularity, measuring approach, power data source, profiling overhead, sampling frequency, setup costs and equipment costs of the individual techniques.

Furthermore, one external and two internal power profiling methods are chosen and applied during the series of experiments in order to be able to compare the measurement values reported by different instruments across devices. Since the internal techniques cannot uniformly be used for all three devices due to platform-dependent internal sensors, two distinct measurement facilities have to be implemented, i.e. one for the Intel Xeon and one for the Jetson boards.

The results of these distinct types of energy measurements show clear differences. While the smart plug readings are consistently higher than the power consumption captured by the internal sensors with respect to the two Jetson boards, the external measurements of the Intel Xeon are lower than the obtained power values of the GPU and its associated circuits. The findings of the Jetson boards therefore correspond with the expected behavior, whereas the outputs of the Intel Xeon profiling are counterintuitive and contradictory. As a consequence, the external power readings conducted by means of the smart plugs are used for the development of the machine learning models, because it is considered as infeasible to consistently and comparably monitor all three devices using internal measurement instruments. Therefore, the smart plugs represent the only uniform approach for measuring the power consumption of diverse edge computing platforms.

RQ 2: How severe is the impact of hardware heterogeneity on the resource usage and power consumption of different devices and what does this mean for power consumption modeling approaches in the edge domain?

In order to ascertain the impact of heterogeneous hardware capabilities on the resource usage and power consumption of edge computing platforms, two kinds of profiling experiments are conducted. The function invocation experiments, which focus on executing the same serverless functions on each device type using a consistent workload, clearly demonstrate that the platform heterogeneity in the edge domain has a severe impact on the resource usage of devices. For instance, regarding the *resnet-gpu* function, the GPU utilization rates of the Intel Xeon and the Jetson Nano exhibit a divergence of approximately 35%. Furthermore, the stress tests performed by means of the *stress-ng* workload generator show that identical loads on different computing infrastructures, i.e. equal resource utilization rates, result in diverse levels of power consumption. As an example, while the Intel Xeon exhibits maximum power readings of 23.5 W on 100% CPU load, the Jetson Nano only consumes 4.2 W under the same load, which is a difference of nearly 20 W. Therefore, the hardware heterogeneity of edge devices also has a significant impact on their power draw.

As a consequence, these findings are incorporated into the applied power consumption modeling method by developing multiple platform-specific models in order to overcome the severe discrepancies in terms of resource usage and power consumption between various edge devices. The results of the accuracy evaluation finally confirm the utilization of a multi-model approach, since the developed models are able to precisely estimate the power draw of devices.

RQ 3: How can the power consumption of edge devices accurately be modeled in a simulation environment considering the strict performance requirements demanded by the underlying simulation framework, and how does the chosen energy modeling approach affect the execution time and scalability of the simulator?

As the power prediction functionality is integrated into the *faas-sim* simulation framework, the final power models have to meet strict requirements regarding the performance-accuracy trade-off they exhibit due to inherent simulator characteristics. For the purpose of incorporating this trade-off into the model development process, TPOT is chosen as AutoML tool, since it internally uses multi-objective Pareto optimization to balance model complexity and prediction precision. Furthermore, the TPOT Light configuration enables restricting the set of pipeline operators during the optimization process, so only fast ones are considered and thus simple and lightweight models can be established.

The evaluation results show that applying machine learning techniques by using TPOT to develop platform-specific and resource utilization-based power prediction models represents an efficient and viable energy modeling approach for heterogeneous edge devices. While these models are able to achieve accurate predictions with a MAE between 190 and 620 mW, they do not severely impair the scalability of the simulator since the prolonged execution time caused by the overhead of the models is reasonable. Hence, the time-efficiency of simulations can still be guaranteed, as the inference speed of the models varies between 0.8 and 2.6 ms. Moreover, the models are able to outperform a linear statistical model, which only considers CPU utilization, across all three devices in terms of MAE scores. As a result, the developed ML models exhibit a satisfactory trade-off and can therefore be reliably and effectively used to estimate the power consumption of edge devices during simulations.

7.2 Future Work

While known limitations of the applied approach are already outlined in Chapter 6, this section proposes future work and research directions on this topic in order to highlight potentials for optimization and to give a conclusive outlook.

For the purpose of incorporating more distinct resource usage values into the training data set and therefore potentially enhancing the prediction accuracy, the models could be re-trained with a larger set of samples. These samples should also include heavier workloads especially for the Intel Xeon, since the GPU-targeted functions executed on this device exhibit relatively low utilization rates, so they are not able to heavily stress the powerful GPU integrated into the Intel Xeon PC. Furthermore, executing additional *stress-ng* tests could also help with achieving a more heterogeneous and more comprehensive data set. Using such a data set for training would probably yield even better results in terms of accuracy than the available records.

Another optimization of the prediction functionality would be to expand it to other devices, since the feature is currently limited to the three devices profiled in the course of this thesis. Nevertheless, supporting power forecasting for a large set of devices would be beneficial in case heterogeneous topologies with several types of platforms are being modeled. As edge domain use cases typically encompass numerous nodes with different hardware characteristics, such topologies are very common in general. In order to expand the power prediction feature to more kinds of platforms and thereby enabling large-scale, heterogeneous scenarios, the approach used in this work might be reapplied to other computing architectures, so additional machine learning models can be developed for other edge device types. Alternatively, the untrained pipelines of the existing models can be used and trained with data samples of other platforms. Furthermore, since the ML algorithms considered in this work do not encompass DNNs, developing deep learning models could also be an opportunity for future research.

Moreover, in the current version of the simulator extension, the power consumption is predicted for each request simulated by the framework even though the average resource usage is utilized for every function invocation. However, this constitutes an intended behavior, since a remaining task that needs to be carried out before the power models can ultimately be released for public usage is to sample the resource usage values from a log-normal distribution instead of using the average for each request. Hence, similarly to the FET sampling, the resource usage of each function call would differ, which represents a more realistic behavior, and so one prediction per request is logically necessary. Nevertheless, the implementation of a caching strategy that prevents the model from repeating the predictions for recurring resource usage metrics would be a potential optimization for the future.

In addition to the prospective public release of the developed simulator extension, which facilitates energy management and planning tasks as well as comparisons regarding the energy efficiency of different edge devices, the power models could serve as the basis for any kind of energy-aware resource management algorithms. These might include, but are not limited to, energy-aware scheduling and load balancing strategies, routing policies or other algorithms where energy efficiency should be considered. However, the data structure of the inputs fed into the models in other contexts must resemble the one used for training, which represents an inherent prerequisite imposed by the models. If this requirement can be satisfied by a certain use case, the general applicability of the power models to other fields where energy awareness plays a critical role is ensured.

APPENDIX A

Scripts

This chapter contains different Python scripts that are developed in the course of this thesis. For confidentiality reasons, constants that define IP addresses, ports and hostnames are removed from the scripts.

MQTT Client Script

Since the code for the MQTT client of zone A and B of the testbed is almost identical, only the script for zone A is included. Since the two Jetson boards are contained in zone A and the Intel Xeon PC resides in zone B, this script is targeted at publishing the smart plug power readings of the Jetson Xavier NX and the Jetson Nano.

```python
import redis
import paho.mqtt.client as mqtt
import json
import datetime


MQTT_TOPIC_NX = 'tele/eb-a-jetson-nx-0/SENSOR'
MQTT_TOPIC_NANO = 'tele/eb-a-jetson-nano-0/SENSOR'


# Client callback for CONNACK response from the server
def on_connect(client, userdata, flags, rc):
    print("Connected with result code " + str(rc))

    # Subscribe to topics
    client.subscribe([(MQTT_TOPIC_NX, 0), (MQTT_TOPIC_NANO, 0)])


# Client callback for PUBLISH messages from the server
```

```
def on_message(client, userdata, msg):
    # Decode message payload
    try:
        decoded_payload = json.loads(str(msg.payload.decode("utf-8",
            errors='ignore')))
    except:
        print("Error during json.loads")
        return

    print(f'Received message on topic {msg.topic} with payload:
        {decoded_payload}')

    # Extract host from MQTT topic
    host = ''
    if 'eb-a-jetson-nx-0' in msg.topic:
        host = 'eb-a-jetson-nx-0'
    elif 'eb-a-jetson-nano-0' in msg.topic:
        host = 'eb-a-jetson-nano-0'

    # Convert timestamp
    datetime_ts = datetime.datetime.strptime(decoded_payload['Time'],
        "%Y-%m-%dT%H:%M:%S")
    unix_ts = datetime_ts.timestamp()

    # Get current and voltage to calculate power consumption
    value = float(decoded_payload['ENERGY']['Current']) *
        float(decoded_payload['ENERGY']['Voltage']) * 1000
    message = f'{unix_ts} {value}'
    publish_message_to_redis(host, 'plug-calc-power', message, userdata)


def publish_message_to_redis(host, suffix, message, redis_client):
    # Assemble Redis channel
    channel = f'telem/{host}/{suffix}'

    # Publish message to channel
    redis_client.publish(channel, message)


if __name__ == '__main__':
    print(f'Starting mosquitto client')

    # Establish Redis connection
    r = redis.Redis(
        host=REDIS_HOST,
        port=REDIS_PORT,
        password=REDIS_PASSWORD)

    # Create MQTT client
    client = mqtt.Client()
    client.on_connect = on_connect
    client.on_message = on_message
    client.user_data_set(r)
```

```python

    # Establish MQTT connection
    client.connect(host=MQTT_HOST)

    # Blocking call that processes network traffic, dispatches callbacks
        and handles reconnecting
    client.loop_forever()
```

Listing A.1: MQTT client script for testbed zone A.

Jtop Power Monitoring Daemon Script

The following script is used to internally measure the power consumption of the Jetson boards by means of the jtop utility. This script has to be run on both Jetson boards during the execution of profiling experiments.

```python
import sys
import jtop
import time
import redis
import multiprocessing


def read_power_stats(jetson):
    # Read power data from tegra stats
    value = jetson.stats['power cur']
    message = f'{time.time()} {value}'
    queue.put(message)


def redis_loop(queue, channel):
    # Establish Redis connection
    r = redis.Redis(
        host=REDIS_HOST,
        port=REDIS_PORT,
        password=REDIS_PASSWORD)

    while True:
        # Get item from queue
        item = queue.get()
        print(f'Received message with payload: {item}')

        # Publish queue item to Redis channel
        r.publish(channel, item)


if __name__ == '__main__':
    # Get node name from command line arguments (if available)
    nodename = 'jetson'
    if len(sys.argv) > 1:
        nodename = sys.argv[1]

    # Define name of Redis channel
    channel = f'telem/{nodename}/jtop-power-cur'

    # Get monitoring interval from command line arguments (if available)
    interval = 1
    if len(sys.argv) > 2:
        interval = sys.argv[2]

    print(f'Starting jtop monitoring for node with name {nodename} and interval of {interval}s')

    # Create queue for communication between processes
    global queue
    queue = multiprocessing.Queue()

    # Start process for Redis loop
    redis_process = multiprocessing.Process(target=redis_loop, args=(queue, channel,))
    redis_process.start()

    # Open jtop with pre-defined interval
    jetson = jtop.jtop(interval)

    # Attach function to read board stats
    jetson.attach(read_power_stats)
    jetson.loop_for_ever()

    redis_process.join()
```

Listing A.2: Jtop power monitoring daemon script.

NVML-based GPU Power Script

As the C script used to monitor the power consumption of Nvidia GPUs is mainly based on existing telemd scripts, only the relevant code fragment is contained here. Specifically, the `nvmlDeviceGetPowerUsage` method called in line 18 is required for this purpose.

```c
1  nvmlDevice_t device;
2  char name[64];
3  int i = atoi(argv[1]);
4
5  result = nvmlDeviceGetHandleByIndex (i, &device);
6  if (NVML_SUCCESS != result) {
7      printf ("Error: failed to get handle for device %i: %s\n", i, nvmlErrorString (result));
8      fail();
}

result = nvmlDeviceGetName (device, name, sizeof (name) / sizeof
    (name[0]));
if (NVML_SUCCESS != result) {
    printf ("Error: failed to get name of device %i: %s\n", i,
        nvmlErrorString (result));
    fail();
}

int power;
result = nvmlDeviceGetPowerUsage (device, &power);
printf ("%d-%s-gpu_power-%d\n", i, name, power);
```

Listing A.3: NVML-based GPU power script.

Baseline Measurement Script

Similar to the MQTT client scripts, the baseline measurements are also dependent on the testbed zone the corresponding devices belong to. Therefore, only the script used for zone B, i.e. for the Intel Xeon, is presented.

```
import paho.mqtt.client as mqtt
import json


MQTT_TOPIC_XEON = 'tele/eb-b-xeongpu-0/SENSOR'


# Client callback for CONNACK response from the server
def on_connect(client, userdata, flags, rc):
    print("Connected with result code " + str(rc))

    # Subscribe to topics
    client.subscribe(MQTT_TOPIC_XEON, 0)


# Client callback for PUBLISH messages from the server
def on_message(client, userdata, msg):
    # Decode message payload
    try:
        decoded_payload = json.loads(str(msg.payload.decode("utf-8",
            errors='ignore')))
    except:
        print("Error during json.loads")
        return

    value = float(decoded_payload['ENERGY']['Current']) *
        float(decoded_payload['ENERGY']['Voltage']) * 1000
    userdata.append(value)

    print(len(userdata))
    print(f'Baseline average: {sum(userdata) / len(userdata)}')


if __name__ == '__main__':
    print(f'Starting baseline measurements')

    measurements = []

    # Create MQTT client
    client = mqtt.Client()
    client.on_connect = on_connect
    client.on_message = on_message
    client.user_data_set(measurements)

    # Establish MQTT connection
    client.connect(host=MQTT_HOST)

    # Blocking call that processes network traffic, dispatches callbacks
        and handles reconnecting
    client.loop_forever()
```

Listing A.4: Baseline measurement script for Intel Xeon (zone B).

APPENDIX B

Empirical Experiments

The following tables contain the complete set of experiments that are conducted in the course of this thesis by means of the testbed.

Function Invocation Experiments

Function	Node	Image
resnet-cpu	eb-b-xeongpu-0	resi5/resnet-inference:v1.0.0
resnet-cpu	eb-b-xeongpu-0	resi5/resnet-inference:v1.0.0
resnet-cpu	eb-b-xeongpu-0	resi5/resnet-inference:v1.0.0
resnet-cpu	eb-b-xeongpu-0	resi5/resnet-inference:v1.0.0
resnet-cpu	eb-b-xeongpu-0	resi5/resnet-inference:v1.0.0
resnet-gpu	eb-b-xeongpu-0	resi5/resnet-inference:v1.0.0
resnet-gpu	eb-b-xeongpu-0	resi5/resnet-inference:v1.0.0
resnet-gpu	eb-b-xeongpu-0	resi5/resnet-inference:v1.0.0
resnet-gpu	eb-b-xeongpu-0	resi5/resnet-inference:v1.0.0
resnet-gpu	eb-b-xeongpu-0	resi5/resnet-inference:v1.0.0
resnet-cpu	eb-a-jetson-nx-0	resi5/resnet-inference:v1.0.0
resnet-cpu	eb-a-jetson-nx-0	resi5/resnet-inference:v1.0.0
resnet-cpu	eb-a-jetson-nx-0	resi5/resnet-inference:v1.0.0
resnet-cpu	eb-a-jetson-nx-0	resi5/resnet-inference:v1.0.0
resnet-cpu	eb-a-jetson-nx-0	resi5/resnet-inference:v1.0.0
resnet-gpu	eb-a-jetson-nx-0	resi5/resnet-inference:v1.0.0
resnet-gpu	eb-a-jetson-nx-0	resi5/resnet-inference:v1.0.0
resnet-gpu	eb-a-jetson-nx-0	resi5/resnet-inference:v1.0.0
resnet-gpu	eb-a-jetson-nx-0	resi5/resnet-inference:v1.0.0

resnet-gpu	eb-a-jetson-nx-0	resi5/resnet-inference:v1.0.0
resnet-cpu	eb-a-jetson-nano-0	resi5/resnet-inference:v1.0.0
resnet-cpu	eb-a-jetson-nano-0	resi5/resnet-inference:v1.0.0
resnet-cpu	eb-a-jetson-nano-0	resi5/resnet-inference:v1.0.0
resnet-cpu	eb-a-jetson-nano-0	resi5/resnet-inference:v1.0.0
resnet-cpu	eb-a-jetson-nano-0	resi5/resnet-inference:v1.0.0
resnet-gpu	eb-a-jetson-nano-0	resi5/resnet-inference:v1.0.0
resnet-gpu	eb-a-jetson-nano-0	resi5/resnet-inference:v1.0.0
resnet-gpu	eb-a-jetson-nano-0	resi5/resnet-inference:v1.0.0
resnet-gpu	eb-a-jetson-nano-0	resi5/resnet-inference:v1.0.0
resnet-gpu	eb-a-jetson-nano-0	resi5/resnet-inference:v1.0.0
efficientnet-cpu	eb-b-xeongpu-0	resi5/efficientnet-inference:v1.0.0
efficientnet-cpu	eb-b-xeongpu-0	resi5/efficientnet-inference:v1.0.0
efficientnet-cpu	eb-b-xeongpu-0	resi5/efficientnet-inference:v1.0.0
efficientnet-cpu	eb-b-xeongpu-0	resi5/efficientnet-inference:v1.0.0
efficientnet-cpu	eb-b-xeongpu-0	resi5/efficientnet-inference:v1.0.0
efficientnet-gpu	eb-b-xeongpu-0	resi5/efficientnet-inference:v1.0.0
efficientnet-gpu	eb-b-xeongpu-0	resi5/efficientnet-inference:v1.0.0
efficientnet-gpu	eb-b-xeongpu-0	resi5/efficientnet-inference:v1.0.0
efficientnet-gpu	eb-b-xeongpu-0	resi5/efficientnet-inference:v1.0.0
efficientnet-gpu	eb-b-xeongpu-0	resi5/efficientnet-inference:v1.0.0
efficientnet-cpu	eb-a-jetson-nx-0	resi5/efficientnet-inference:v1.0.0
efficientnet-cpu	eb-a-jetson-nx-0	resi5/efficientnet-inference:v1.0.0
efficientnet-cpu	eb-a-jetson-nx-0	resi5/efficientnet-inference:v1.0.0
efficientnet-cpu	eb-a-jetson-nx-0	resi5/efficientnet-inference:v1.0.0
efficientnet-cpu	eb-a-jetson-nx-0	resi5/efficientnet-inference:v1.0.0
efficientnet-gpu	eb-a-jetson-nx-0	resi5/efficientnet-inference:v1.0.0
efficientnet-gpu	eb-a-jetson-nx-0	resi5/efficientnet-inference:v1.0.0
efficientnet-gpu	eb-a-jetson-nx-0	resi5/efficientnet-inference:v1.0.0
efficientnet-gpu	eb-a-jetson-nx-0	resi5/efficientnet-inference:v1.0.0
efficientnet-gpu	eb-a-jetson-nx-0	resi5/efficientnet-inference:v1.0.0
efficientnet-cpu	eb-a-jetson-nano-0	resi5/efficientnet-inference:v1.0.0
efficientnet-cpu	eb-a-jetson-nano-0	resi5/efficientnet-inference:v1.0.0
efficientnet-cpu	eb-a-jetson-nano-0	resi5/efficientnet-inference:v1.0.0
efficientnet-cpu	eb-a-jetson-nano-0	resi5/efficientnet-inference:v1.0.0
efficientnet-cpu	eb-a-jetson-nano-0	resi5/efficientnet-inference:v1.0.0
efficientnet-gpu	eb-a-jetson-nano-0	resi5/efficientnet-inference:v1.0.0
efficientnet-gpu	eb-a-jetson-nano-0	resi5/efficientnet-inference:v1.0.0
efficientnet-gpu	eb-a-jetson-nano-0	resi5/efficientnet-inference:v1.0.0
efficientnet-gpu	eb-a-jetson-nano-0	resi5/efficientnet-inference:v1.0.0

efficientnet-gpu	eb-a-jetson-nano-0	resi5/efficientnet-inference:v1.0.0
objectdetection	eb-b-xeongpu-0	edgerun/objectdetection:1.1.0
objectdetection	eb-b-xeongpu-0	edgerun/objectdetection:1.1.0
objectdetection	eb-b-xeongpu-0	edgerun/objectdetection:1.1.0
objectdetection	eb-b-xeongpu-0	edgerun/objectdetection:1.1.0
objectdetection	eb-b-xeongpu-0	edgerun/objectdetection:1.1.0
objectdetection	eb-a-jetson-nx-0	edgerun/objectdetection:1.1.0
objectdetection	eb-a-jetson-nx-0	edgerun/objectdetection:1.1.0
objectdetection	eb-a-jetson-nx-0	edgerun/objectdetection:1.1.0
objectdetection	eb-a-jetson-nx-0	edgerun/objectdetection:1.1.0
objectdetection	eb-a-jetson-nx-0	edgerun/objectdetection:1.1.0
objectdetection	eb-a-jetson-nano-0	edgerun/objectdetection:1.1.0
objectdetection	eb-a-jetson-nano-0	edgerun/objectdetection:1.1.0
objectdetection	eb-a-jetson-nano-0	edgerun/objectdetection:1.1.0
objectdetection	eb-a-jetson-nano-0	edgerun/objectdetection:1.1.0
objectdetection	eb-a-jetson-nano-0	edgerun/objectdetection:1.1.0

Table B.1: Experiments conducted for function invocation configurations.

Stress-ng Experiments

Application	Node	Parameters
stress-ng	eb-b-xeongpu-0	cpu: 1
stress-ng	eb-b-xeongpu-0	cpu: 1
stress-ng	eb-b-xeongpu-0	cpu: 1
stress-ng	eb-b-xeongpu-0	cpu: 1
stress-ng	eb-b-xeongpu-0	cpu: 1
stress-ng	eb-b-xeongpu-0	cpu: 2
stress-ng	eb-b-xeongpu-0	cpu: 2
stress-ng	eb-b-xeongpu-0	cpu: 2
stress-ng	eb-b-xeongpu-0	cpu: 2
stress-ng	eb-b-xeongpu-0	cpu: 2
stress-ng	eb-b-xeongpu-0	cpu: 4
stress-ng	eb-b-xeongpu-0	cpu: 4
stress-ng	eb-b-xeongpu-0	cpu: 4
stress-ng	eb-b-xeongpu-0	cpu: 4
stress-ng	eb-b-xeongpu-0	cpu: 4
stress-ng	eb-b-xeongpu-0	cpu: 8
stress-ng	eb-b-xeongpu-0	cpu: 8
stress-ng	eb-b-xeongpu-0	cpu: 8

stress-ng	eb-b-xeongpu-0	cpu: 8
stress-ng	eb-b-xeongpu-0	cpu: 8
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 25
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 25
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 25
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 25
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 25
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 50
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 50
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 50
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 50
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 50
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 75
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 75
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 75
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 75
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 75
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 100
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 100
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 100
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 100
stress-ng	eb-b-xeongpu-0	cpu: 0, cpu-load: 100
stress-ng	eb-b-xeongpu-0	vm: 1, vm-bytes: 20%
stress-ng	eb-b-xeongpu-0	vm: 1, vm-bytes: 20%
stress-ng	eb-b-xeongpu-0	vm: 1, vm-bytes: 20%
stress-ng	eb-b-xeongpu-0	vm: 1, vm-bytes: 20%
stress-ng	eb-b-xeongpu-0	vm: 1, vm-bytes: 20%
stress-ng	eb-b-xeongpu-0	vm: 1, vm-bytes: 40%
stress-ng	eb-b-xeongpu-0	vm: 1, vm-bytes: 40%
stress-ng	eb-b-xeongpu-0	vm: 1, vm-bytes: 40%
stress-ng	eb-b-xeongpu-0	vm: 1, vm-bytes: 40%
stress-ng	eb-b-xeongpu-0	vm: 1, vm-bytes: 40%
stress-ng	eb-b-xeongpu-0	vm: 1, vm-bytes: 80%
stress-ng	eb-b-xeongpu-0	vm: 1, vm-bytes: 80%
stress-ng	eb-b-xeongpu-0	vm: 1, vm-bytes: 80%
stress-ng	eb-b-xeongpu-0	vm: 1, vm-bytes: 80%
stress-ng	eb-b-xeongpu-0	vm: 1, vm-bytes: 80%
stress-ng	eb-b-xeongpu-0	iomix: 1
stress-ng	eb-b-xeongpu-0	iomix: 1
stress-ng	eb-b-xeongpu-0	iomix: 1

stress-ng	eb-b-xeongpu-0	iomix: 1
stress-ng	eb-b-xeongpu-0	iomix: 1
stress-ng	eb-a-jetson-nx-0	cpu: 1
stress-ng	eb-a-jetson-nx-0	cpu: 1
stress-ng	eb-a-jetson-nx-0	cpu: 1
stress-ng	eb-a-jetson-nx-0	cpu: 1
stress-ng	eb-a-jetson-nx-0	cpu: 1
stress-ng	eb-a-jetson-nx-0	cpu: 2
stress-ng	eb-a-jetson-nx-0	cpu: 2
stress-ng	eb-a-jetson-nx-0	cpu: 2
stress-ng	eb-a-jetson-nx-0	cpu: 2
stress-ng	eb-a-jetson-nx-0	cpu: 2
stress-ng	eb-a-jetson-nx-0	cpu: 4
stress-ng	eb-a-jetson-nx-0	cpu: 4
stress-ng	eb-a-jetson-nx-0	cpu: 4
stress-ng	eb-a-jetson-nx-0	cpu: 4
stress-ng	eb-a-jetson-nx-0	cpu: 4
stress-ng	eb-a-jetson-nx-0	cpu: 8
stress-ng	eb-a-jetson-nx-0	cpu: 8
stress-ng	eb-a-jetson-nx-0	cpu: 8
stress-ng	eb-a-jetson-nx-0	cpu: 8
stress-ng	eb-a-jetson-nx-0	cpu: 8
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 25
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 25
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 25
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 25
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 25
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 50
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 50
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 50
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 50
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 50
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 75
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 75
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 75
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 75
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 75
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 100
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 100
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 100

stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 100
stress-ng	eb-a-jetson-nx-0	cpu: 0, cpu-load: 100
stress-ng	eb-a-jetson-nx-0	vm: 1, vm-bytes: 20%
stress-ng	eb-a-jetson-nx-0	vm: 1, vm-bytes: 20%
stress-ng	eb-a-jetson-nx-0	vm: 1, vm-bytes: 20%
stress-ng	eb-a-jetson-nx-0	vm: 1, vm-bytes: 20%
stress-ng	eb-a-jetson-nx-0	vm: 1, vm-bytes: 20%
stress-ng	eb-a-jetson-nx-0	vm: 1, vm-bytes: 40%
stress-ng	eb-a-jetson-nx-0	vm: 1, vm-bytes: 40%
stress-ng	eb-a-jetson-nx-0	vm: 1, vm-bytes: 40%
stress-ng	eb-a-jetson-nx-0	vm: 1, vm-bytes: 40%
stress-ng	eb-a-jetson-nx-0	vm: 1, vm-bytes: 40%
stress-ng	eb-a-jetson-nx-0	vm: 1, vm-bytes: 80%
stress-ng	eb-a-jetson-nx-0	vm: 1, vm-bytes: 80%
stress-ng	eb-a-jetson-nx-0	vm: 1, vm-bytes: 80%
stress-ng	eb-a-jetson-nx-0	vm: 1, vm-bytes: 80%
stress-ng	eb-a-jetson-nx-0	vm: 1, vm-bytes: 80%
stress-ng	eb-a-jetson-nx-0	iomix: 1
stress-ng	eb-a-jetson-nx-0	iomix: 1
stress-ng	eb-a-jetson-nx-0	iomix: 1
stress-ng	eb-a-jetson-nx-0	iomix: 1
stress-ng	eb-a-jetson-nx-0	iomix: 1
stress-ng	eb-a-jetson-nano-0	cpu: 1
stress-ng	eb-a-jetson-nano-0	cpu: 1
stress-ng	eb-a-jetson-nano-0	cpu: 1
stress-ng	eb-a-jetson-nano-0	cpu: 1
stress-ng	eb-a-jetson-nano-0	cpu: 1
stress-ng	eb-a-jetson-nano-0	cpu: 2
stress-ng	eb-a-jetson-nano-0	cpu: 2
stress-ng	eb-a-jetson-nano-0	cpu: 2
stress-ng	eb-a-jetson-nano-0	cpu: 2
stress-ng	eb-a-jetson-nano-0	cpu: 2
stress-ng	eb-a-jetson-nano-0	cpu: 4
stress-ng	eb-a-jetson-nano-0	cpu: 4
stress-ng	eb-a-jetson-nano-0	cpu: 4
stress-ng	eb-a-jetson-nano-0	cpu: 4
stress-ng	eb-a-jetson-nano-0	cpu: 4
stress-ng	eb-a-jetson-nano-0	cpu: 8
stress-ng	eb-a-jetson-nano-0	cpu: 8
stress-ng	eb-a-jetson-nano-0	cpu: 8

stress-ng	eb-a-jetson-nano-0	cpu: 8
stress-ng	eb-a-jetson-nano-0	cpu: 8
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 25
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 25
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 25
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 25
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 25
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 50
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 50
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 50
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 50
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 50
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 75
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 75
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 75
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 75
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 75
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 100
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 100
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 100
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 100
stress-ng	eb-a-jetson-nano-0	cpu: 0, cpu-load: 100
stress-ng	eb-a-jetson-nano-0	vm: 1, vm-bytes: 20%
stress-ng	eb-a-jetson-nano-0	vm: 1, vm-bytes: 20%
stress-ng	eb-a-jetson-nano-0	vm: 1, vm-bytes: 20%
stress-ng	eb-a-jetson-nano-0	vm: 1, vm-bytes: 20%
stress-ng	eb-a-jetson-nano-0	vm: 1, vm-bytes: 20%
stress-ng	eb-a-jetson-nano-0	vm: 1, vm-bytes: 40%
stress-ng	eb-a-jetson-nano-0	vm: 1, vm-bytes: 40%
stress-ng	eb-a-jetson-nano-0	vm: 1, vm-bytes: 40%
stress-ng	eb-a-jetson-nano-0	vm: 1, vm-bytes: 40%
stress-ng	eb-a-jetson-nano-0	vm: 1, vm-bytes: 40%
stress-ng	eb-a-jetson-nano-0	vm: 1, vm-bytes: 80%
stress-ng	eb-a-jetson-nano-0	vm: 1, vm-bytes: 80%
stress-ng	eb-a-jetson-nano-0	vm: 1, vm-bytes: 80%
stress-ng	eb-a-jetson-nano-0	vm: 1, vm-bytes: 80%
stress-ng	eb-a-jetson-nano-0	vm: 1, vm-bytes: 80%
stress-ng	eb-a-jetson-nano-0	iomix: 1
stress-ng	eb-a-jetson-nano-0	iomix: 1
stress-ng	eb-a-jetson-nano-0	iomix: 1

stress-ng	eb-a-jetson-nano-0	iomix: 1
stress-ng	eb-a-jetson-nano-0	iomix: 1

Table B.2: Experiments conducted for stress-ng configurations.

List of Figures

List of Tables

Bibliography

[ABC$^+$21] Nongnuch Artrith, Keith T Butler, François-Xavier Coudert, Seungwu Han, Olexandr Isayev, Anubhav Jain, and Aron Walsh. Best practices in machine learning for chemistry. *Nature chemistry*, 13(6):505–508, 2021.

[AESI$^+$17] Mahmoud AM Albreem, Ayman A El-Saleh, Muzamir Isa, Wael Salah, Muzammil Jusoh, MM Azizan, and A Ali. Green internet of things (iot): An overview. In *2017 IEEE 4th International Conference on Smart Instrumentation, Measurement and Application (ICSIMA)*, pages 1–6. IEEE, 2017.

[AJH$^+$21] Khaled Alwasel, Devki Nandan Jha, Fawzy Habeeb, Umit Demirbaga, Omer Rana, Thar Baker, Scharam Dustdar, Massimo Villari, Philip James, Ellis Solaiman, et al. Iotsim-osmosis: A framework for modeling and simulating iot applications over an edge-cloud continuum. *Journal of Systems Architecture*, 116:101956, 2021.

[AQPJ21] Muhammad Saidu Aliero, Kashif Naseer Qureshi, Muhammad Fermi Pasha, and Gwanggil Jeon. Smart home energy management systems in internet of things networks for green cities demands and services. *Environmental Technology & Innovation*, 22:101443, 2021.

[ASA$^+$21] Mahmoud A Albreem, Abdul Manan Sheikh, Mohammed H Alsharif, Muzammil Jusoh, and Mohd Najib Mohd Yasin. Green internet of things (giot): applications, practices, awareness, and challenges. *IEEE Access*, 9:38833–38858, 2021.

[ATC$^+$21] Mohammad S Aslanpour, Adel N Toosi, Claudio Cicconetti, Bahman Javadi, Peter Sbarski, Davide Taibi, Marcos Assuncao, Sukhpal Singh Gill, Raj Gaire, and Schahram Dustdar. Serverless edge computing: vision and challenges. In *2021 Australasian Computer Science Week Multiconference*, pages 1–10, 2021.

[AZF$^+$21] Laha Ale, Ning Zhang, Xiaojie Fang, Xianfu Chen, Shaohua Wu, and Longzhuang Li. Delay-aware and energy-efficient computation offloading in mobile-edge computing using deep reinforcement learning. *IEEE Transactions on Cognitive Communications and Networking*, 7(3):881–892, 2021.

[AZS$^+$17] Rushan Arshad, Saman Zahoor, Munam Ali Shah, Abdul Wahid, and Hongnian Yu. Green iot: An investigation on energy saving practices for 2020 and beyond. *Ieee Access*, 5:15667–15681, 2017.

[BB21] Przemyslaw Biecek and Tomasz Burzykowski. *Explanatory Model Analysis - Explore, Explain, and Examine Predictive Models*. CRC Press, Boca Raton, Fla, 1 edition, 2021.

[BCC$^+$17] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research advances in cloud computing*, pages 1–20. Springer, 2017.

[BRKP22] Sarath Babu and Arun Raj Kumar P. A comprehensive survey on simulators, emulators, and testbeds for vanets. *International Journal of Communication Systems*, 35(8):e5123, 2022.

[CCPB21] Gonçalo Carvalho, Bruno Cabral, Vasco Pereira, and Jorge Bernardino. Edge computing: current trends, research challenges and future directions. *Computing*, 103(5):993–1023, 2021.

[CCSF19] Sidartha A.L. Carvalho, Daniel C. Cunha, and Abel G. Silva-Filho. Autonomous power management in mobile devices using dynamic frequency scaling and reinforcement learning for energy minimization. *Microprocessors and Microsystems*, 64:205–220, 2019.

[CLMS20] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE access*, 8:85714–85728, 2020.

[CLX$^+$21] Zhuoqing Chang, Shubo Liu, Xingxing Xiong, Zhaohui Cai, and Guoqing Tu. A survey of recent advances in edge-computing-powered artificial intelligence of things. *IEEE Internet of Things Journal*, 2021.

[CMPR22] Emanuele Cuncu, Marco Manolo Manca, Barbara Pes, and Daniele Riboni. Towards context-aware power forecasting in smart-homes. *Procedia Computer Science*, 198:243–248, 2022. 12th International Conference on Emerging Ubiquitous Systems and Pervasive Networks / 11th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare.

[CRB$^+$11] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011.

[CZS18] Jie Cao, Quan Zhang, and Weisong Shi. *Challenges and Opportunities in Edge Computing*, pages 59–70. Springer International Publishing, Cham, 2018.

[DDTD19] Michele De Donno, Koen Tange, and Nicola Dragoni. Foundations and evolution of modern computing paradigms: Cloud, iot, edge, and fog. *Ieee Access*, 7:150936–150948, 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.

[FLLFC21] Paula Fraga-Lamas, Sérgio Ivan Lopes, and Tiago M Fernández-Caramés. Green iot and edge ai as key technological enablers for a sustainable digital transition towards a smart circular economy: An industry 5.0 use case. *Sensors*, 21(17):5745, 2021.

[GA19] Elena Gracheva and Alsu Alimova. Calculation methods and comparative analysis of losses of active and electric energy in low voltage devices. In *2019 International Ural Conference on Electrical Power Engineering (UralCon)*, pages 361–367. IEEE, 2019.

[GCZY21] Chen Guo, Song Ci, Yanglin Zhou, and Yang Yang. A survey of energy consumption measurement in embedded systems. *IEEE Access*, 9:60516–60530, 2021.

[GVDGB17] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K. Ghosh, and Rajkumar Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.

[HS19] Jun-Ho Huh and Yeong-Seok Seo. Understanding edge computing: Engineering evolution with artificial intelligence. *IEEE Access*, 7:164229–164245, 2019.

[ISH10] Muhammad Imran, Abas Md Said, and Halabi Hasbullah. A survey of simulators, emulators and testbeds for wireless sensor networks. In *2010 International Symposium on Information Technology*, volume 2, pages 897–902. IEEE, 2010.

[JAA$^+$20] Devki Nandan Jha, Khaled Alwasel, Areeb Alshoshan, Xianghua Huang, Ranesh Kumar Naha, Sudheer Kumar Battula, Saurabh Garg, Deepak Puthal, Philip James, Albert Zomaya, et al. Iotsim-edge: a simulation framework for modeling the behavior of internet of things and edge computing environments. *Software: Practice and Experience*, 50(6):844–867, 2020.

[JFG$^+$20] Congfeng Jiang, Tiantian Fan, Honghao Gao, Weisong Shi, Liangkai Liu, Christophe Cérin, and Jian Wan. Energy aware edge computing: A survey. *Computer Communications*, 151:556–580, 2020.

[JSSS$^+$19] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[KHG$^+$17] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News*, 45(1):615–629, 2017.

[KHH$^+$20] Sven Köhler, Benedict Herzog, Timo Hönig, Lukas Wenzel, Max Plauth, Jörg Nolte, Andreas Polze, and Wolfgang Schröder-Preikschat. Pinpoint the joules: Unifying runtime-support for energy measurements on heterogeneous systems. In *2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, pages 31–40. IEEE, 2020.

[LAA$^+$21] Jingyi Liu, Qaiser Abbas, Majed Alharthi, Muhammad Mohsin, Farhat Rasul, and Nadeem Iqbal. Managerial policy and economic analysis of wind-generated renewable hydrogen for light-duty vehicles: Green solution of energy crises. *Environmental Science and Pollution Research*, 28(9):10642–10653, 2021.

[LCH$^+$18] Pengcheng Liu, Saqib Rasool Chaudhry, Tao Huang, Xiaojun Wang, and Martin Collier. Multi-factorial energy aware resource management in edge networks. *IEEE Transactions on Green Communications and Networking*, 3(1):45–56, 2018.

[LFM20] Trang T Le, Weixuan Fu, and Jason H Moore. Scaling tree-based automated machine learning to biomedical big data with a feature set selector. *Bioinformatics*, 36(1):250–256, 2020.

[LLKP19] Sang Hyeon Lee, Tacklim Lee, Seunghwan Kim, and Sehyun Park. Energy consumption prediction system based on deep learning with edge computing. In *2019 IEEE 2nd International Conference on Electronics Technology (ICET)*, pages 473–477. IEEE, 2019.

[MPGB22] Redowan Mahmud, Samodha Pallewatta, Mohammad Goudarzi, and Rajkumar Buyya. ifogsim2: An extended ifogsim simulator for mobility, clustering, and microservice management in edge and fog computing environments. *Journal of Systems and Software*, 190:111351, 2022.

[OBUM16] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 485–492, New York, NY, USA, 2016. ACM.

[OM16] Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on automatic machine learning*, pages 66–74. PMLR, 2016.

[Ong17] Pariwat Ongsulee. Artificial intelligence, machine learning and deep learning. In *2017 15th international conference on ICT and knowledge engineering (ICT&KE)*, pages 1–6. IEEE, 2017.

[OUA$^+$16] Randal S. Olson, Ryan J. Urbanowicz, Peter C. Andrews, Nicole A. Lavender, La Creis Kidd, and Jason H. Moore. Automating biomedical data science through tree-based pipeline optimization. In Giovanni Squillero and Paolo Burelli, editors, *Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 – April 1, 2016, Proceedings, Part I*, pages 123–137. Springer International Publishing, 2016.

[Oyi21] Tobi Oyinlola. Energy prediction in edge environment for smart cities. In *2021 IEEE 7th World Forum on Internet of Things (WF-IoT)*, pages 439–442, 2021.

[Rai21] Philipp Alexander Raith. Container scheduling on heterogeneous clusters using machine learning-based workload characterization. Diploma thesis, Technische Universität Wien, 2021.

[Raj20] Bashar Rajoub. Supervised and unsupervised learning. In Walid Zgallai, editor, *Biomedical Signal Processing and Artificial Intelligence in Healthcare*, Developments in Biomedical Engineering and Bioelectronics, pages 51–89. Elsevier, 2020.

[RHS$^+$21] Thomas Rausch, Waldemar Hummer, Christian Stippel, Silvio Vasiljevic, Carmine Elvezio, Schahram Dustdar, and Katharina Krösl. Towards a platform for smart city-scale cognitive assistance applications. In *2021 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*, 2021.

[RLF$^+$20] Thomas Rausch, Clemens Lachner, Pantelis A Frangoudis, Philipp Raith, and Schahram Dustdar. Synthesizing plausible infrastructure configurations for evaluating edge computing systems. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.

[RRD21] Thomas Rausch, Alexander Rashed, and Schahram Dustdar. Optimized container scheduling for data-intensive serverless edge computing. *Future Generation Computer Systems*, 114:259–271, 2021.

[RRL18] Crefeda Faviola Rodrigues, Graham Riley, and Mikel Luján. Synergy: An energy measurement and prediction framework for convolutional neural networks on jetson tx1. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 375–382. The Steering Committee of The World Congress in Computer Science, Computer ..., 2018.

[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.

[Sat17] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[SLHM22] Jiaqi Shi, Nian Liu, Yujing Huang, and Liya Ma. An edge computing-oriented net power forecasting for pv-assisted charging station: Model complexity and forecasting accuracy trade-off. *Applied Energy*, 310:118456, 2022.

[SOE18] Cagatay Sonmez, Atay Ozgovde, and Cem Ersoy. Edgecloudsim: An environment for performance evaluation of edge computing systems. *Transactions on Emerging Telecommunications Technologies*, 29(11):e3493, 2018.

[TWG+19] Anh Truong, Austin Walters, Jeremy Goodsitt, Keegan Hines, C. Bayan Bruss, and Reza Farivar. Towards automated machine learning: Evaluation and comparison of automl approaches and tools. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1471–1479, 2019.

[WBC+19] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE international symposium on high performance computer architecture (HPCA)*, pages 331–344. IEEE, 2019.

[WCW+17] Mowei Wang, Yong Cui, Xin Wang, Shihan Xiao, and Junchen Jiang. Machine learning for networking: Workflow, advances and opportunities. *Ieee Network*, 32(2):92–99, 2017.

[WLP+19] Nan Wei, Changjun Li, Xiaolong Peng, Fanhua Zeng, and Xinqian Lu. Conventional models and artificial intelligence-based models for energy consumption forecasting: A review. *Journal of Petroleum Science and Engineering*, 181:106187, 2019.

[WT21] Philipp Wiesner and Lauritz Thamsen. Leaf: Simulating large energy-aware fog computing environments. In *2021 IEEE 5th International Conference on Fog and Edge Computing (ICFEC)*, pages 29–36. IEEE, 2021.

[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(8):1738–1762, 2019.

[ZCS19] Yukun Zeng, Mengyuan Chao, and Radu Stoleru. Emuedge: A hybrid emulator for reproducible and realistic edge computing experiments. In *2019 IEEE International Conference on Fog Computing (ICFC)*, pages 153–164, 2019.

[ZLH$^+$18] Ke Zhang, Supeng Leng, Yejun He, Sabita Maharjan, and Yan Zhang. Mobile edge computing and networking for green and low-latency internet of things. *IEEE Communications Magazine*, 56(5):39–45, 2018.

[ZWJ$^+$19] Hai Zhong, Jiajun Wang, Hongjie Jia, Yunfei Mu, and Shilei Lv. Vector field-based support vector regression for building energy consumption prediction. *Applied Energy*, 242:403–414, 2019.