Untangling the Cloud
from
Edge Computing for the Internet of Things

by

Nabeel Nasir

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
in the

Department of Computer Science
School of Engineering and Applied Science
University of Virginia
May 2024

Doctoral Committee:

Assistant Professor Bradford Campbell
Professor Emeritus John Stankovic
Associate Professor Felix Xiaozhu Lin
Associate Professor Arsalan Heydarian
Assistant Professor Yuan Tian, University of California Los Angeles

APPROVAL SHEET

This

Dissertation

is submitted in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

Author: Nabeel Nasir

This Dissertation has been read and approved by the examing committee:

Advisor: Bradford Campbell

Advisor:

Committee Member: John Stankovic

Committee Member: Felix Xiaozhu Lin

Committee Member: Arsalan Heydarian

Committee Member: Yuan Tian

Committee Member:

Committee Member:

Accepted for the School of Engineering and Applied Science:

Jennifer L. West, School of Engineering and Applied Science

May 2024

ABSTRACT

The Internet of Things (IoT) is growing at a rapid pace, with billions of devices expected in the near future generating zettabytes of data. The current design of IoT is dependent on the cloud for data storage, processing, and control decisions, which does not scale well to handle this massive influx of data. Edge computing is viewed as a key solution to handle this, and prescribes executing applications closer to devices rather than in the cloud, consequently reducing latency, minimizing bandwidth, as well as improving privacy by operating on premises. The current model of edge computing relies on server-class machines on other public edge infrastructure nearby IoT devices to execute applications, with all control and configuration handled at the cloud. However, having the control plane away from devices, and in the cloud, reduces scalability and reliability, incurs considerable cost, is impractical for deployments without Internet access, and leads to limited data privacy controls for users.

Our work moves away from a cloud-centric design to a device-centric design for edge computing. We identify untapped compute potential in gateway devices present in IoT deployments and utilize it for edge computing, rather than relying on cloud-controlled edge infrastructure. This shift presents several key challenges: handling interoperability of IoT devices, operating on constrained resources, addressing user privacy, and supporting heterogeneous gateways and dynamic workloads. To address these challenges, we first use a decentralized architecture and a thin middleware to enable multiple gateways to operate together, combining their compute capabilities to offer more than the sum of its parts, supporting a good set of edge IoT applications. Further, we create an IoT ecosystem in which resource-constrained IoT devices can offload tasks to more resource-powerful devices, enabling a host of more compute-intensive realtime edge applications to be supported. We also provide users in shared IoT spaces with better transparency and control over their data, by utilizing our gateway-based edge computing platform to enforce user privacy preferences to filter data from edge applications. Together, these solutions enable edge computing which is cost-effective, scalable, general-purpose, and privacy-aware, without the drawbacks of cloud dependency.

ACKNOWLEDGEMENTS

I want to thank my parents, Dr. Abdul Nasir, and Dr. Safiya Nasir. I am truly appreciative of how they've raised me, how much independence they've given me, and how they've supported my dreams. I am happy that I now have a "Dr." tag and I can finally feel at ease with all the medical doctors in the family!

I also want to thank my wife, Nishara, for making this journey a lot easier than it would have. It's hard to put into words how much of a difference she has made in my life. I am really appreciative of all those times she has helped me bounce back from setbacks, encouraged me to manage my physical and mental health better, and pushed me on when I didn't have any belief in myself.

I would like to thank my advisor, Brad Campbell, who I've had a great relationship with, and who has supported me throughout my PhD journey. He was compassionate during my times of struggles, and was patient with me, handling some of the excessive skepticism I had about my own ideas. He has ingrained a culture of great work-life balance in our research group, while setting high standards for our research. I am grateful for the various sessions he conducted on identifying research problems, writing effectively, framing hypotheses and thesis statements, which have helped me immensely in becoming a better researcher. I'm thankful for the opportunity to co-teach with him, which helped me gain confidence to pursue a career in teaching, and for his unwavering support throughout my job interview process for a teaching faculty position.

I would like to thank my committee members, John Stankovic, Felix Lin, Arsalan Heydarian, and Yuan Tian, for helping shape this dissertation.

I'm lucky to have made some close friends at UVA who kept me sane through my PhD. I'm indebted to Ishika Paul for positively impacting my research and teaching, introducing me to so many new things, playing table tennis and board games, influencing some of my life philosophies, and supporting me through some tough times. I'd like to thank Samyukta Venkat for giving great feedback on my work, but equally importantly, partaking in candid discussions about life, sharing a brand of weird silly comedy, and taking me to some great concerts and events.

Victor Sobral has been an amazing collaborator, graciously offering his time whenever I or anyone in the group needed help, whose feedback has improved this work substantially. I'm grateful to Viswajith Rajan for supporting me through my co-teaching experience, and helping me better navigate my PhD life. The three of us have gone on countless walks, shared memes about grad student life, watched football matches, and played board games, making my time so much more memorable.

I'm thankful to other members of our research group. Marshall Clyburn, Rabbi Masum, Li-Pang Huang have been amazing collaborators and also great people to hang out with. I would like to thank Nurani Saoda and Fateme Nikseresht, who have been great sources of positive energy for me. I would also like to mention Wenpeng Wang, Jiechao Gao, Tushar Routh, and Alexander Sarris who have helped me settle in and navigate grad school.

I am thankful to my master's advisor, Dr. Krithi Ramamritham, who sparked my interest in doing research. I have been lucky to work with the wonderful research group we had at the SEIL

Lab in IIT Bombay: Vivek Chil Prakash, Kartik Palani, Uddhav Arote, and Anand Prakash. Vivek taught me how to plan long term projects, and has always believed in my abilities as a researcher and an educator, while the others have been an excellent support system for me over so many years.

I am also thankful to people and organizations who've helped shape my teaching while at UVA. Dr. John Stankovic once interrupted a seminar talk I was giving for his course, and complimented me on the effort I put into my slides to explain a difficult concept to the audience. This moment has given me great confidence whenever I've taught afterward. I am grateful to the Center for Teaching Excellence for the great programs they offer like the Tomorrow's Professor Today and the Teaching Fellowship Program, which have been pivotal in making me a more inclusive, feedback-driven, and empathetic teacher. Computers4Kids (C4K) is an organization I mentored for, which fosters a great learning environment for young people from the community. I have learned a lot while mentoring there, and I wish to take those learnings with me in my career as an educator.

I am thankful to a few others who've positively influenced my life. Vivek Chhabbarwal has always been a great cheerleader for my work and has always believed in me. I'm indebted to Gadha Raj for getting my PhD application over the line during a rather chaotic time of my life. I am happy to have spent some awesome discussions over food with Alan Wang and Feng-Yi Chang. Avinash Ramesh and Srijith Santhosh have been a source of silliness and strength over the years. I am grateful to my therapist, Josh, for helping me manage my mental health better. I'm thankful to Lahiru Nuwan for making my life in Charlottesville so much better with our times playing cricket, volleyball, and hiking. I am also thankful for the wonderful times I've spent with Nithin Revi, Ajay S, Joseph Malliakkal, Anushruti Huilgol, Arjun Ajith Mohan, and Apinop Atipiboonsin.

I am thankful to all the people I've mentioned here and the countless others who I've crossed paths with who made my PhD life a period I will look back at fondly.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

The Internet of Things (IoT) is being widely adopted in different areas including smart cities, smart homes, precision agriculture, and healthcare. It is expected to continue growing, with the global market for IoT to reach a value of $1,386 billion by 2026 from $761 billion in 2020 [1]. Even with the upsurge in popularity, the design of current IoT systems has mostly stayed simple and involves three key pieces: the IoT device, the cloud, and a gateway to bridge communication between device and cloud. For instance, a Philips Hue smart light setup has the bulb (device), the Hue Bridge (the gateway), and their cloud service [2]. This IoT implementation allows users to connect to the cloud service using a smartphone application or a cloud API to obtain data or control the IoT devices [3, 4]. Another more ad-hoc implementation of this IoT design is to deploy sensors or actuators, set up gateways like the Raspberry Pi [5] to collect data from the devices, and send data to an application hosted in some cloud service [6–8]. Most major cloud computing services offer IoT platforms to support this second type of IoT implementation [9–11]. In both implementations, data storage, data processing, and control decisions are performed at the cloud.

Although this design is useful for simple use cases, dependence on the cloud incurs high bandwidth usage, applications suffer from high latency, and users may have privacy concerns with their data in the cloud.

1.1 Edge Computing to Address Cloud Limitations

Edge computing is touted as a solution to handle these concerns with cloud dependency, which prescribes executing applications closer to devices, consequently reducing latency, minimizing bandwidth, and improving privacy by operating on premises [12]. Edge computing for IoT introduces a new edge layer between the cloud layer and gateway layer, which consists of server-class machines deployed within the local network that can execute applications. Due to their proximity to IoT devices, applications can process data at very low latencies, without sending the data to the cloud. This design also allows offloading tasks that require heavier compute to the cloud from the edge server. The edge computing design is centered on the cloud with all control and configuration handled at the cloud for easier management [13, 14].

However, having the control plane away from devices, and in the cloud, introduces several issues. First, it is impractical for remote deployments without a stable Internet connection or in secure deployments which have restricted access to the Internet. For example, applications like precision agriculture in a remote farm field, or a smart healthcare application using sensitive patient data would benefit from edge computing but are deployments in which a cloud connection is either unavailable or restricted. Second, it reduces reliability and incurs considerable cloud and

bandwidth monetary costs. Third, it hinders scalability due to configuration overhead at the cloud. For instance, most IoT edge computing platforms like Microsoft's Azure IoT [10] require devices and gateways in the deployment to be configured on the cloud prior to usage, which reduces scalability as devices increase [15]. Fourth, current platforms provide inadequate privacy mechanisms for users to control how edge applications use their IoT data, and existing mechanisms require users to configure such rules from the cloud [16, 17], excluding deployments without access to the Internet.

These issues highlight the need for a design that is not cloud-centric, but instead focuses on the IoT devices and users, to realize the full potential of edge computing for IoT. In this dissertation, we focus on utilizing unused compute power available near IoT devices to provide edge computing for IoT which doesn't depend on the cloud.

1.2 Utilizing Unused Compute for Providing Edge Intelligence

Edge computing for IoT uses server-class machines close to IoT devices to execute applications that primarily require low latency processing, operate on high bandwidth data or use highly private data. However, compute power at the edge is a scarce commodity and not all IoT deployments have access to such an infrastructure due to high setup costs, or limited availability of third party edge infrastructure [18, 19].

We instead explore an alternative approach to providing edge computing. IoT device deployments typically leverage gateways or distributed networking equipment that translates between the low power wireless networks IoT devices use and more conventional IP-based networks. These gateways are essential in IoT deployments, yet underutilized, and we utilize them to provide edge intelligence for the IoT devices that are connected to them. Applications executing in situ on an edge node, requiring only low/moderate amounts of compute power (CPU, GPU, memory) and storage, and run indefinitely once deployed, are good candidates to execute on gateways.

However, creating a new paradigm of edge computing relying on edge gateways having no cloud support, raises several key challenges: *handling interoperability of IoT devices*, *operating on constrained resources*, *addressing user privacy in IoT environments with interoperating devices*, *supporting devices with heterogeneous computational capabilities*, and *supporting dynamic workloads in edge computing environments*.

Current IoT devices are inherently heterogeneous, coming from different manufacturers with different sensors, energy sources, operating modes, form factors, data formats, and wireless protocols, and supporting useful edge applications must address interoperability among IoT devices. In shared spaces like smart buildings, users have limited knowledge of how their IoT data is being collected and used. This is exacerbated when building an edge computing platform which enables better interoperability among devices.

Shifting away from cloud data centers or edge server machines to gateway devices limits the available compute resources for applications. Also, applications no longer have the luxury of utilizing the elasticity of resources like storage, CPU, GPU, etc. available on the cloud.

Additionally, utilizing gateways for compute requires handling the heterogeneity in capabilities that are present in gateways. IoT environments are also dynamic and tasks, devices, and device behavior could change, including new tasks being added, task QoS requirements changing, new devices being added, or gateway loads changing.

This dissertation explores solutions to mitigate these challenges to support a cloud-independent edge computing paradigm. The first of these solutions is to identify a feasible architecture to enable multiple gateways to operate together, combining their compute capabilities to offer more than the sum of its parts, and support responsive edge applications and cutting-edge IoT devices. This architecture enables the gateways to operate without any Internet access, support heterogeneous IoT devices, support streaming data from devices, and scale well with more devices and applications.

Our second solution is to convert a disjoint network of gateways into a cohesive platform by building a thin middleware which can handle application load balancing, optimize latency and network communication overhead, provide automatic discovery and scalability, ensure resilience to failures, and operate autonomously without any Internet connectivity. Additionally, we design an open-source edge computing platform, which aims to manage heterogeneity for better interoperability, minimize configuration, aid in network management, and support applications, and easily interoperate with other IoT systems. The middleware supports user devices to remotely manage applications and devices, as well as abstracts out the underlying distributed network complexity, providing developers a simple centralized application model to ease development. This enables support for applications like sensing and actuating in large-scale IoT deployments, simple if-this-then-that (IFTTT) applications, simple machine learning at the edge, and inherently distributed applications.

The next solution enables supporting realtime IoT applications with much higher compute requirements and tighter quality of service (QoS) requirements. We explore how resource-constrained IoT devices like AR/VR headsets, smartwatches, and other low-power IoT sensors can improve the quality of service of their applications by relying on another class of smart home and office devices which include smart TVs, gaming consoles, and smart doorbell cameras. This second category of IoT devices, which we refer to as gateways as well since they topologically provide services to resource-constrained devices, have relatively better computing resources, considerable idle time, implying computational resources to spare, and a steady source of energy, making them a candidate to provide edge intelligence.

We provide this edge intelligence using a task offloading ecosystem which utilizes heterogeneous edge gateways to support realtime requirements of diverse edge tasks. We design a scalable realtime task scheduler which eliminates the need for *a priori* task profile data by profiling tasks on the go, to satisfy task QoS while minimizing overall energy of the ecosystem. This opportunistic approach enables us to support dynamic IoT workloads which include short realtime tasks with high compute requirements and tight QoS requirements. It enables combining compute power from multiple heterogeneous gateways to provide a form of elasticity of resources on the edge, reducing the need for a dedicated cloud-dependent edge infrastructure.

Our final solution aims at providing users in shared IoT spaces with better transparency and

control over their data. We collect privacy preferences from users on how applications should use their sensor data and use the edge intelligence on the gateways to enforce these preferences. Gateways in our edge paradigm already act as routers that route sensor data from IoT devices to edge applications, and we extend this design to enforce privacy policies on streaming sensor data. This solution provides an alternative to the inadequate privacy mechanisms in current edge computing systems, which unnecessarily exclude deployments without Internet access.

We present real-world deployments of these solutions, and these solutions together enable our vision for an edge computing paradigm which does not require dedicated edge infrastructure, and is untangled from a control plane from the cloud.

1.3 Thesis Statement

Requiring cloud support for edge computing moves the control plane away from IoT devices, hinders scalability and reliability, is impractical for remote deployments, and incurs considerable cost. A cloudless system that utilizes inexpensive IoT gateways for compute, scales up seamlessly across multiple gateways, exploits their diverse special-purpose capabilities to replace cloud elasticity, provides an always-on "front desk" infrastructure on the edge to replace cloud's centrality, and empowers users to control which applications can use their data through privacy policies, enables edge computing which is cost-effective, scalable, general-purpose, and privacy-aware without the drawbacks of cloud dependency.

1.4 Contributions of This Dissertation

This dissertation focuses on building a gateway-based privacy-aware edge computing platform which operates independent of the cloud, and makes the following contributions.

First is a qualitative analysis of the state of the art to identify different edge computing architectures which can support IoT use cases. We compare the architectures on their suitability for supporting edge computing on IoT gateways. We identify a decentralized architecture which enables gateways to operate without any Internet access, support heterogeneous IoT devices, and scale up with devices and applications.

For the decentralized architecture identified by our analysis, we develop a middleware that enables disjoint gateways to coordinate together to execute edge computing applications. This work is a novel approach to demonstrate the feasibility of using IoT gateways to build a cohesive platform that can execute edge computing applications. We also develop an open-source, cloud-independent, resilient, edge computing platform, which can handle device heterogeneity and access control of data, and interoperates with other edge computing platforms. This work was done in collaboration with Victor Sobral, Li-Pang Huang, and Bradford Campbell, and presented at ACM/IEEE Symposium on Edge Computing (SEC) '22 [20].

We then develop an IoT device ecosystem in which resource-constrained devices offload edge

tasks to more resource-powerful devices, in lieu of requiring a dedicated edge computing infrastructure. We design a realtime task scheduler for this ecosystem which can meet the offloaded task's QoS requirements without requiring a priori task profiling, making it scalable, heterogeneity-aware, and well-suited for the dynamic nature of IoT. This work was done in collaboration with Marshall Clyburn, Md Fazlay Rabbi Masum Billah, Victor Sobral, Dong Chen, Jiechao Gao, Fateme Nikseresht, and Bradford Campbell, and is under preparation.

We then design a user-centric privacy enforcement mechanism for users in shared IoT spaces to have better transparency and control over their data. This enables user privacy preferences to be enforced one hop from the devices rather than aggregating data in the cloud and applying filtering at a later stage.

We also extensively compare the privacy approach in our work to the one employed by, Matter [21], an emerging unifying standard for smart home IoT devices. We compare between the device-centric approach of Matter and the application-centric approach of our work, and explore how privacy controls in one approach can be represented in the other.

We also present multiple real world applications which are supported by the edge computing platform, including actively supporting sensor data collection in the Link Lab. Additionally, we present a case study of the design and implementation of a robot offloading a realtime room classification task to our edge infrastructure. These applications highlight the scalability, usability, and cost-effectiveness of our platform.

Finally, we identify multiple open problems that arise from this work, creating further opportunities for research to further improve the performance of our gateway-based cloudless edge computing infrastructure, as well as assist in improving other emerging IoT standards.

CHAPTER 2
BACKGROUND AND RELATED WORK

2.1 Edge Computing

Edge computing refers to offloading computation from the cloud closer to end devices. There are three major implementations of edge computing, namely *Mobile Edge Computing* [22], *Cloudlet Computing* [12], and *Fog Computing* [13]. Mobile edge computing provides storage and processing capabilities at base stations in a Radio Area Network (RAN) to enable cloud computing services for mobile subscribers. Cloudlet computing uses micro data centers called "cloudlets" near to mobile users to reduce the latency of applications without requiring round trips to the cloud. Fog computing utilizes a hierarchical set of IoT gateways, routers, and edge servers to provide a computing and storage layer between end devices and the cloud. Recent works [23–25] have tried to disambiguate between these types based on factors like the type of edge nodes used, their location, mechanisms to access end devices etc. Using the decision tree in [25], our work can be classified under fog computing, since IoT gateways are considered fog nodes, and we use them to directly connect with sensors and actuators.

2.2 Edge Computing Platforms

Cloud platforms like Amazon Web Services and Microsoft Azure provide edge computing solutions (AWS IoT Greengrass [26], Azure IoT Edge [27]) to execute applications in the edge network. We identify four shortcomings of these platforms. First, they assume a rather simple centralized deployment architecture in which all device data is available at a central edge node and applications execute only on this node. Second, there is no access control of device data to applications, and they expose all available data to all applications. Third, they require a cloud connection for configuration and application deployment, which restricts use cases with unreliable Internet connectivity. Finally, the development experience is deterred by configuration overhead and lack of simple application abstractions. Deploying an application that interact with even tens of devices requires substantial configuration [15]. Developers need to set up data flow between apps, devices, and gateways, which is intractable at scale.

2.3 Collaborating Edge Gateways

There are a few papers which study cooperation among IoT gateways. Clemente et al. present a framework where gateways form a mesh network to cooperate or provide services to each other [28]. Their work does not describe the interface between devices and the platform, or applications and devices, which are key components. Ooi et al. present an architecture in which gateways

coordinate to provide additional routes from devices to the cloud to improve reliability [29]. This leverages multiple gateways, but applications are all executed on the cloud, and gateways only cooperate to reliably deliver data to the cloud.

2.3.1 Tiered Wireless Sensor Networks

The Tenet Architecture support 2-tier networks, with multiple decentralized *master nodes* (equivalent to IoT gateways) in one tier, and motes in the lower tier [30]. The architecture allows multiple applications to execute concurrently on the master nodes. However, in their architecture they assume motes can execute tasks, which are delivered to them by the master nodes by splitting the application. This assumption does not hold for IoT devices as they are not programmable. Also, the motes they consider are very homogeneous, unlike IoT devices which are very heterogeneous.

2.4 Device Heterogeneity on The Edge

IoT devices come with a wide range of wireless radios, network protocols, and data formats, which complicate application development. SemIoTic [31] maps semantic user commands to device actions by abstracting the underlying device heterogeneity with a *DeX API* that provides support for protocols including CoAP, MQTT, and XMPP. However, this excludes resource-constrained devices which cannot support these application protocols. TinyLink 2.0 [15] is a programming language for IoT which automatically generates programs and configuration for the cloud, device, and mobile layers. However, they require devices to be programmable to support specific functions which is not always feasible, and is difficult to scale as the device API varies based on the underlying device.

2.4.1 Task Offloading in Edge Computing

To improve the quality of experience (QoE) of applications running on mobile devices, Satyanarayanan et al. [32] introduced the concept of Cloudlets, edge servers that provide to mobile devices the task offloading benefits of cloud computing without suffering from its fundamental limitations in terms of latency and bandwidth-induced delays. Their work considers mobile devices as thin clients with one-hop and high bandwidth wireless access to Cloudlets, which run loosely coupled soft-state applications. The benefits of computation offloading were demonstrated in the following works, which aimed to save battery power for mobile devices [33] and increase the computational capabilities of wearable devices [34]. Lin et al. conducted a comprehensive literature review on edge computing in 2019 [35], and more recently, in 2021, Luo et al. presented the latest research trends in edge computing [36].

2.4.2 Real-time Task Scheduling at the Edge

Many real-time schedulers have been introduced during the last few years to minimize the energy consumption of the executors and meet the QoS requirements of the offload tasks. Zhang et al. [37] developed a game-theoretic task allocation framework called CoGTA to allocate real-time social sensing tasks to cooperative edge computing nodes. Their evaluation results show that their system satisfies tasks' QoS requirements. On the other hand, HeteroEdge [38] proposes a resource management framework that addresses the heterogeneity issue of the edge devices by providing a uniform interface to conceptualize the device details. Kim et al. [39] also propose a collaborative task scheduling scheme for edge devices to offload their tasks among idle IoT devices according to the tasks' execution time and energy consumption.

The main drawback of these real-time task schedulers is that they require a priori task profiling; they need to know the execution time and energy usage of each task on the executing devices in advance. However, IoT environments are heterogeneous consisting of a variety of tasks as well as executor devices, and profiling every task-device combination can get intractable at scale. It is also not practical to assume that such profiling data is available for all user environments. Second, changes to tasks or executor devices could require re-profiling tasks and updating the scheduler. However, IoT environments are dynamic and tasks, devices, and device behavior could change. For instance, changes like new tasks being added, task QoS requirement changing, new devices being added, or device loads changing are all events which can happen in IoT environments. Re-profiling of tasks and updating of the scheduler for such changes results in poor adaptability of the scheduler.

2.5 Privacy Controls for Users in Shared Spaces

Prior works like Pappachan et al. [40] outline a framework to design privacy-aware smart buildings and build a machine-readable policy language to specify privacy policy rules, but do not detail how privacy policies are actually enforced. Ghayyur et al. [41] show how to filter data streams based on rules from a privacy policy and apply certain Privacy Enhancing Technologies (PET) on the streams, but do not discuss the infrastructure to implement such a design. Das et al. [42] solves the problem of collecting privacy policy rules from users and how they can be made aware of data sensing in their environment, but they expecting IoT devices to provide an opt-in/opt-out API to enforce such policy rules. However, this is impractical for most IoT sensors as they do not host RESTful APIs, and highlights the usefulness of having gateways enforce privacy policies. Al-Hasnawi et al. [43] demonstrate enforcing privacy policy rules on Edge Fog Nodes (similar to gateways) to provide access control of IoT data to locally running edge applications. But their policies do not operate on streaming data, and only enforces privacy policies when edge applications make requests for previously stored data at the gateway.

CHAPTER 3
MOTIVATING APPLICATIONS

In this chapter, we set the scope for our work by enumerating five classes of applications which would benefit from using a cloudless gateway-based edge computing system. These applications fall at the intersection of edge computing and IoT, and benefit from executing at proximity to IoT devices without depending on the cloud. For each application class, we provide specific application examples from literature, and reason why this class fits our edge computing model. Table 3.1 indexes which chapters of this dissertation covers the various motivating application types listed in this chapter.

3.1 Low-Latency Applications

These are applications that require IoT data but also need to operate at low latencies. For instance, an augmented reality (AR) application that helps visualize hot and cold areas in a large hall, using data from multiple temperature sensors [44]. Or an AR application which can control non-smart appliances with embedded IoT devices [45]. Even though the devices may not be producing data at high data rates, it is essential that the data is received at the AR application fast for the visualization to be seamless and without stutter.

3.2 Applications Using High Bandwidth Sensors

This includes applications that need streaming IoT data particularly from sensors with a high data rate, like cameras, Doppler radar, etc. These devices usually have large data packets, large sampling requirements, or both. Examples of applications using such devices include an edge application that receives images of license plates from several cameras and performs license plate detection on them [46], or an application which monitors heart rate using a Doppler radar sensor [47]. For such applications, it is beneficial to execute the applications as close to the sensors as possible, to reduce network traffic and sensor latency, due to the high bandwidth of the sensors.

3.3 Applications Operating on Highly Private Data

These include classes of applications that operate on IoT data but prefer no data or limited data to be sent to the cloud. For instance, an edge application at a hospital which obtains data from wearable sensors or other bedside sensors to alert for events like patient falls [48]. Or an Industrial Internet of Things (IIoT) application which constantly monitors equipment health and performs

anomaly detection [49]. Relying on an edge platform which doesn't depend on the cloud can reduce privacy breaches (eg: HIIPA violations [50]), or improve regulatory compliance [51, 52].

3.4 Applications in Remote IoT Deployments

These are applications that operate on IoT data in deployments which don't have reliable Internet connectivity, or where Internet access is expensive. For instance, an application that performs agricultural irrigation using sprinklers and soil moisture level sensors [53, 54], or an application which does drainage monitoring in a smart city [55]. These applications cannot rely on Internet connectivity for configuration or operation.

3.5 Inherently Distributed Applications

Some IoT applications *must* be spatially distributed and cannot run in a centralized manner in the cloud or at an edge server machine. For example, SeamBlue [56] provides cellular-like handover functionality to BLE devices, and must run on spatially distributed gateways to transfer BLE connection state as the BLE device moves throughout a space. Similarly, IoT services for wireless devices including time updates [57], firmware updates [58], and fault detection heartbeat messages [59] require gateways near devices that can provide those services.

Application Type	Dissertation Chapter
Low-Latency Applications	Ch6
Applications Using High Bandwidth Sensors	Ch5, Ch6, Ch8
Applications Operating on Highly Private Data	Ch7
Applications in Remote IoT Deployments	Ch5, Ch6
Inherently Distributed Applications	Ch5

Table 3.1: Shows how the chapters that follow cover the various motivating application types listed.

CHAPTER 4
EDGE COMPUTING ARCHITECTURES FOR INTERNET OF THINGS

Developing an edge computing system that supports the proposed applications outlined in Chapter 3 in challenging scenarios starts with defining a suitable architecture. In this chapter, we first outline the requirements that a valid architecture must meet, and a set of components that existing architectures contain. We then explore a series of potential designs, starting with existing approaches, and discuss the benefits and tradeoffs with various approaches. We build towards a suitable and scalable architecture that can support the needs of complex IoT networks while including acceptable drawbacks. Finally, we evaluate the architectures to identify their suitability for edge computing for IoT.

4.1 Architecture Requirements and Components

Building an edge computing system that is cloud independent, and utilizes edge gateways for compute requires an architecture that supports state-of-the-art IoT devices, satisfies requirements of real world edge applications, and scales up well with compute requirements and devices. We argue a valid architecture must meet four requirements derived from the intended use cases.

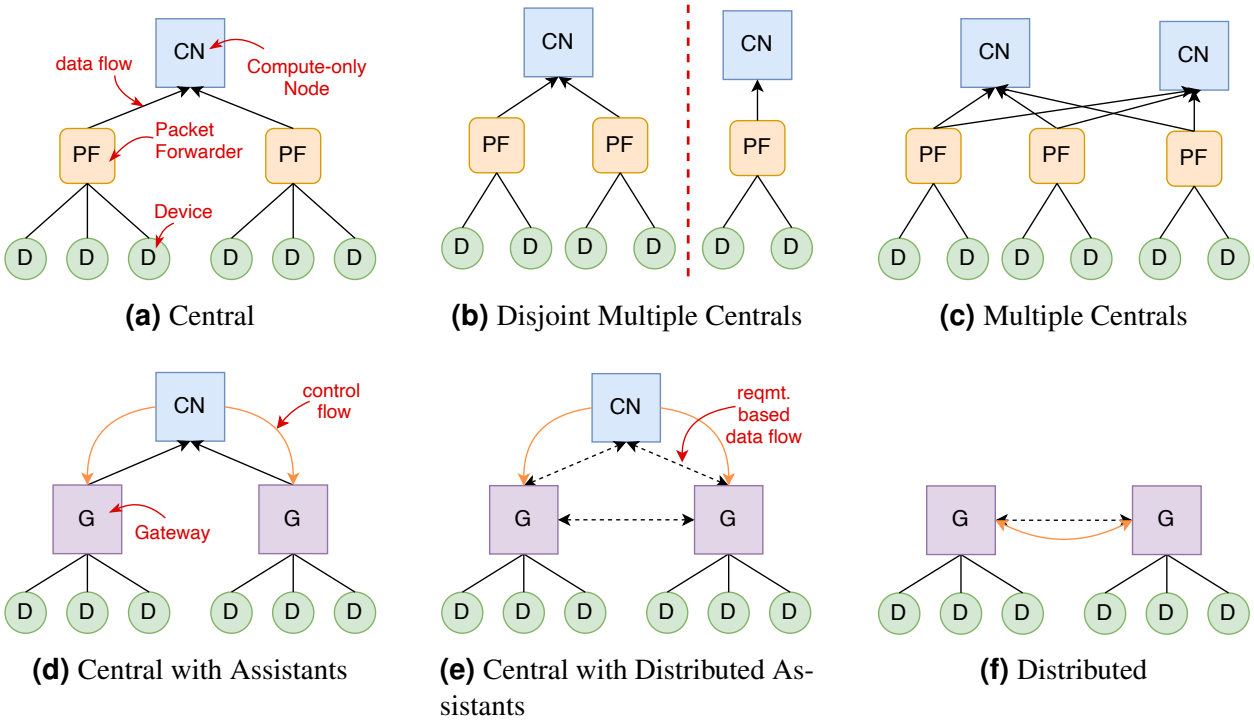
1. The approach must be able to operate without ongoing cloud support. This is necessary for remote or air gapped environments or if internet access is intermittent.

2. The edge architecture needs to support heterogeneous IoT devices with various compute and communication capabilities. Requiring only sophisticated devices (e.g. modern smartphones) or specific networks (e.g. 5G) would prohibit an approach from supporting the long-tail of IoT devices.

3. The architecture must support streaming data. This enables responsive and "smart" IoT applications.

4. The architecture must be able to scale to support both more devices and new gateways as coverage needs, protocol support, and deployment requirements change.

After surveying various edge computing for IoT approaches, we identify four generic components that existing architectures are comprised of: compute-only nodes, packet forwarders, gateways, and devices. Compute-only nodes (CN) are capable of executing applications but cannot directly communicate with IoT devices over low power networks. These are often servers or unused desktop machines. Packet forwarders act as bridges between networks, and relay packets from low power networks to conventional IP-based networks. This could be a simple ESP32 [60]



(a) Central **(b)** Disjoint Multiple Centrals **(c)** Multiple Centrals

(d) Central with Assistants **(e)** Central with Distributed Assistants **(f)** Distributed

Figure 4.1: Design space for IoT edge computing architectures. Architectures tradeoff complexity, scalability, resource utilization, network overhead, and performance. We advocate for the distributed model in Figure 4.1f to encourage scalability, promote interoperability, and avoid centralized failures.

WiFi device. Gateways offer more compute than packet forwarders, and can communicate with low power devices as well as execute applications. These are typically single-board computers like the Raspberry Pi [5]. Devices are sensors or actuators and connect to gateways or packet forwarders. The architectures we identify connect these components in different ways to enable certain features.

4.2 Architectures

We now explore various architectures that can help meet the overall goal of utilizing gateways for designing a cloudless edge computing system, and discuss how they meet the four additional requirements from Section 4.1. We start with conventional architectures used by existing systems, and build to an ideal architecture which supports real world edge computing for IoT use cases. Figure 4.1 depicts the various approaches that are further explained in this section. Table 4.1 provides a quick overview of how each architecture fares against the requirements that we outlined earlier.

Architecture	Requirement				
	(1) Cloud Independent	(2) Device Heterogeneity	(3) Streaming Data	(4a) Scale Up Compute	(4b) Scale Up with Devices
C	✗	✗	✗	✗ requires server replacement	✗ high network traffic & latency
DMC	✗	✗	✗	✗ requires adding servers	✗ high network traffic & latency
MC	✗	✗	✗	✗ requires adding servers	✗ high network traffic & latency
CWA	✓	✓	✓	✓ by adding assistant gateways	✗ high network traffic & latency
CWDA	✓	✓	✓	✓ by adding assistant gateways	✓ smart routing to reduce traffic & latency
D	✓	✓	✓	✓ by adding gateways	✓ smart routing to reduce traffic & latency

Table 4.1: Comparison of various edge computing architectures and how they fare with the requirements outlined in Section 4.1

4.2.1 Extending the Cloud to the Edge

Building on the success of cloud computing, the first edge computing architecture largely tries to mimic cloud abstractions but runs the computation on edge resources. AWS IoT Greengrass [26] takes this approach where the cloud sends applications to a central compute node. As shown in Figure 4.1a, this compute node interacts with all devices through packet forwarders and executes all local applications. Just as serverless computing exploits small, short-lived pockets of unused cloud compute, Greengrass enables small *lambda* functions to leverage the capability of the local compute node.

While conceptually attractive, trying to map the centralized cloud architecture to the edge has several drawbacks. First, scaling up compute capabilities can be difficult or expensive (i.e. replacing an existing server with a new server) [61, 62], or unsupported. Second, all data streams are centralized, incurring network traffic overhead [63]. Third, to move applications from the cloud to the edge, devices must use a standard protocol like MQTT [26], which either excludes emerging resource constrained devices or necessitates numerous packet forwarders. Even resource optimized protocols like CoAP [64] and CBOR [65] can be difficult to map to protocols like BLE, and standardization and agreement has also remained elusive.

This architecture cannot operate without cloud support or handle device heterogeneity. Scaling up compute is expensive and requires replacing servers, and adding new devices leads to increased

network traffic and end-to-end sensor latency overheads.

4.2.2 More Flexible Approaches

Building on the centralized approach from Section 4.2.1, we address the issue of scaling up compute by simply running multiple edge networks, as shown in Figure 4.1b. This architecture uses multiple compute nodes to load balance while still supporting familiar cloud abstractions. AWS Greengrass can be deployed in this manner [26]. However, this effectively bifurcates the deployment, resulting in independent edge networks. In certain deployments, this may be acceptable, but general purpose applications may need data streams from both networks. This architecture also cannot operate without cloud support or handle device heterogeneity. Scaling up compute can be done by installing a new server but is expensive, and adding new devices leads to increased network traffic and end-to-end sensor latency overheads.

One approach to address this is to simply connect all devices to all compute nodes, enabling increased compute while also allowing any application to access any device. This architecture is shown in Figure 4.1c. This is appealing because it is a straightforward extension of the original architecture from Section 4.2.1. It does, however, increase the burden on devices and packet forwarders, as they must send to multiple compute nodes, and must be updated any time a new compute node is added. If a device cannot be reconfigured to send to multiple destinations, then additional packet forwarders must be introduced. Compute nodes can share the application execution workload, but each compute node has to handle every data stream, even for devices not needed by the locally running applications. Even worse, these effects compound, and scaling the network up further with new devices and compute nodes requires both configuring the new infrastructure *and* reconfiguring the existing. This architecture is primarily used in fault tolerant edge computing systems, with multiple compute nodes obtaining data from all devices [66].

This architecture also cannot operate without cloud support or handle device heterogeneity. Scaling up compute can be done by installing a new server but is expensive, and compared to the previous two architectures, due to all devices sending data to multiple compute nodes, scaling up with devices has even worse network traffic and end-to-end sensor latency overheads.

4.2.3 A Case for Decentralization

To address the roadblocks with centralized approaches, we conclude that a new architecture with decentralized components is required. Therefore, rather than adding new "central" compute nodes, existing packet forwarders are updated to gateways to execute applications, and a new edge coordination layer allows the compute nodes to offload applications to the gateways. This approach is shown in Figure 4.1d, and is pursued by HeteroEdge [67] and Samie et al. [68], which both perform computation offloading in a tiered architecture. As before, the compute node aggregates all data streams and forwards them to the correct gateways as needed. This approach reduces the network traffic duplication issue as data packets are only relayed if necessary. Also, since there is

only one cohesive network, devices only have to be configured to send to a single compute node. But, the burden on this compute node can still be quite high, and using the coordinator as the central dispatch for data leads to additional network overhead.

This architecture can operate without cloud support and handles device heterogeneity better. Scaling up compute can be done by achieved by adding new assistant gateways, but can lead to increased coordination overhead on the compute node. Scaling up with devices is difficult due to increased network traffic and end-to-end sensor latency overheads.

4.2.4 Decentralization with More Cooperation

Embracing a distributed architecture further to reduce the network overhead, we next propose configuring gateways to forward data directly to where applications that need the data streams are running (Figure 4.1e). This is used in deployments with multiple compute nodes forming a mesh network with one central coordinating compute node [69], or in multi-level arrangements of compute nodes with a coordinating compute node [70]. This increases the coordination complexity, but enables leveraging the compute resources of multiple gateways, minimizes the burden of deploying additional devices, and exploits the available spatial properties of an application's device requirements (i.e. many applications use data streams from clustered devices) to reduce network overhead.

This more decentralized architecture can operate without cloud support and handles device heterogeneity well. Scaling up compute can be done by achieved by adding new assistant gateways, but can still lead to increased coordination overhead on the compute node. This architecture fares much better in scaling up with devices by providing smart routing optimizations based on data proximity to reduce network traffic and end-to-end sensor latency overheads.

This architecture forms the basis for our proposed approach, which is shown in Figure 4.1f. Our proposed architecture observes that as gateways can execute applications, the compute node is largely redundant. Removing the compute node simplifies deployments and re-uses hardware that is commonly already deployed. A fully decentralized model also eliminates a central point of failure. This approach does incur overhead costs related to setup, consensus, and organization among gateways. However, this is mostly one-time, and does not affect the critical path of applications (i.e. the time taken for sensing, processing, and actuating). Also, even in multi-tier centralized approaches, there is an extra overhead in aggregating data streams and providing those to applications, which is comparable to this decentralization overhead.

4.3 Evaluation

To evaluate the proposed architectures, we conduct two studies. First, we measure the CPU usage, memory usage, and the network traffic as the number of connected devices increases. Next, we measure the CPU usage, memory usage, network traffic, and the application latency as the number

of applications increases and the sensor requirement of applications changes. We use five different Raspberry Pi [5] devices as the Compute-only Nodes, Gateways, and Packet Forwarders, and generate synthetic data to simulate a deployment of sensors.

4.3.1 Architectures and Evaluation Setup

We set up a test deployment for each architecture as follows: 1) Central (C): A single CN and four packet forwarders. All packet forwarders forward data to the CN. Applications are executed only on the CN. 2) Multiple Centrals (MC): Extension of Central with two CNs and three packet forwarders. Each packet forwarder forwards data to both CNs. Applications are executed on either of the CNs. We omit the Disjoint Multiple Centrals architecture (Figure 4.1b) as applications do not have access to all data. 3) Central with Assistants (CWA): One CN and four assistant gateways. All assistant gateways send data to the CN. The central deploys applications on any of the assistants and also provides the streams needed for the applications. 4) Central with Distributed Assistants (CWDA): Similar to CWA, but assistants do not pass data to the central. The central deploys an application to one of the assistants based on the availability of data streams at the assistant. Assistants either use locally available data streams, or request data from other gateways if unavailable. 5) Distributed (D): Five gateways operate without a central entity to coordinate application execution. Gateways communicate with each other to decide which gateway gets to execute the application. They also request device streams from each other.

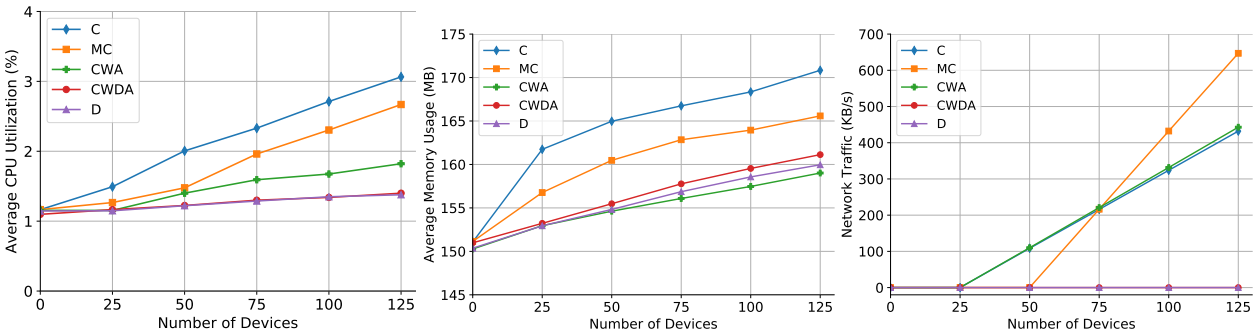
We assumed this deployment to be within the floor of a smart building, and simulated data for five classes of devices, based on the requirements of the applications in our evaluation. We considered a total of 125 devices with different payload sizes, and streaming rates. We simulated 10 smart meters at 500 bytes/s, 50 temperature sensors at 100 bytes every 5 min., 50 occupancy sensors at 100 bytes every 5 min., 10 CO_2 sensors at 100 bytes every 15 min., and 5 IP cameras at 100 KB/s assuming 720p at 15 fps [71].

4.3.2 Evaluation of Device Scaling

We first measure CPU utilization, memory usage, and network traffic of the gateways as the number of devices increases. For this experiment, the 125 devices are equally distributed among each CN, packet forwarder, or gateway, i.e. each hosting 25 devices (we assume devices over a WiFi network, so even CNs can support them). We do not take into account packet forwarders while averaging CPU and memory usages, as they cannot run applications and are not considered as computing resources. The measurements are illustrated in Figures 4.2a to 4.2c.

4.3.2.1 Results

1) From Figure 4.2a and Figure 4.2b, the C and MC architectures can support a limited number of devices due to resource constraints, since they get flooded with device data, and can be poor architecture choices if the deployment has lots of high streaming devices. Interestingly, CWA fares



(a) CPU Utilization (b) Memory Usage (c) Network Traffic

Figure 4.2: Illustrates the CPU usage, memory usage, network traffic evaluated for different architectures as number of devices increases.

much better, although it has a similar centralized architecture. This can be attributed to having other assistant gateways to work with, in effect reducing the average CPU and memory utilization. In contrast, other distributed architectures like CWDA and D fare slightly better in terms of resource utilization. 2) Figure 4.2c shows that CWDA and D have substantially lower network traffic as compared to centralized architectures, as gateways in CWDA and D only stream data over the network when applications request for it. For gateways with limited CPU and memory resources, the high network traffic in central architectures could potentially overload the gateways.

4.3.3 Evaluation of Application Scaling

We developed five different applications for this evaluation, each with its own device requirements. We deployed the applications successively, and measured the CPU utilization, memory utilization, and network traffic of the gateways, and the applications' sensor data reception latency.

We use the following applications and deploy them in this order. 1) *Power meter anomaly detection*: Monitors power meter data from 10 smart meters, and sends an alert if it detects a spike in the power profile. 2) *Object detection in secure areas of a building*: Obtains the camera feed from five IP cameras, extracts images from them, and uses a machine learning model trained on the ImageNet dataset [72], to check for any suspicious objects. 3) *User comfort monitor*: Collects data from the 50 temperature sensors from different office rooms and open areas, monitors for overheating or overcooling, and sends alerts to facilities. 4) *Air quality monitoring*: Periodically checks readings from ten CO_2 sensors and alerts if certain areas cross dangerous ppm levels. 5) *Room scheduling*: Collects information from all occupancy sensors from the building floor, and also data from user calendars, to display available conference rooms.

Since architectures deploy apps based on how devices are distributed among nodes (we use node to commonly refer to a CN, PF, or a gateway), we considered three different device distribution schemes in this experiment. 1) *Colocated*: All devices of a class are exclusively available

on one node. Specifically, all five IP cameras on one node, all ten CO_2 sensors on a second node, etc. 2) *Distributed*: Devices are equally distributed on all five nodes. Specifically, each node has 1 (of the 5) IP camera, 10 (of the 50) occupancy sensors, etc. 3) *Random*: Devices are randomly distributed on the five nodes.

For each of these schemes, we measured the average CPU utilization, memory usage, network traffic, and the application latency (time it takes for an application to receive a sensor packet after it was created). The measurements are shown in Figure 4.3 and Figure 4.4.

4.3.3.1 Results

1) Memory usage of C and MC architectures are roughly 90% and 36% higher respectively than other architectures, across device distributions. This can be attributed to having limited number of nodes to execute applications. 2) Central architectures (C, MC, CWA) again suffer from a higher network traffic. CWA suffers more than C as it receives all device streams and then has to forward a subset of this data to applications that run on the assistant gateways. CWDA and D deploys applications based on the locality of devices on gateways, thereby reducing additional network traffic, as shown in Figure 4.4 i-a and Figure 4.4 iii-a. However, if devices are equally distributed among gateways (Figure 4.4 ii-a), applications end up needing data from all gateways, effectively performing similar to the C architecture. 3) The CWDA and D architectures offer better application latency when devices are colocated (Figure 4.4 i-b), and can at least match the latency offered by other architectures in other device distributions. 4) Also, CPU and memory usages are very low, at 3% and 250 MB (with 4 GB available), even with five apps running. This shows that gateways can indeed support edge applications, and also scales up well.

Overall, both studies highlight the benefit of adopting a decentralized architecture (CWDA or D) with substantial improvement in network load, better resource usage, and reduced latency when device requirements involve spatial locality. Embracing a distributed architecture is therefore beneficial, and our choice of a fully decentralized architecture (D) removes the central compute-only node and thus eliminates a single point of failure problem.

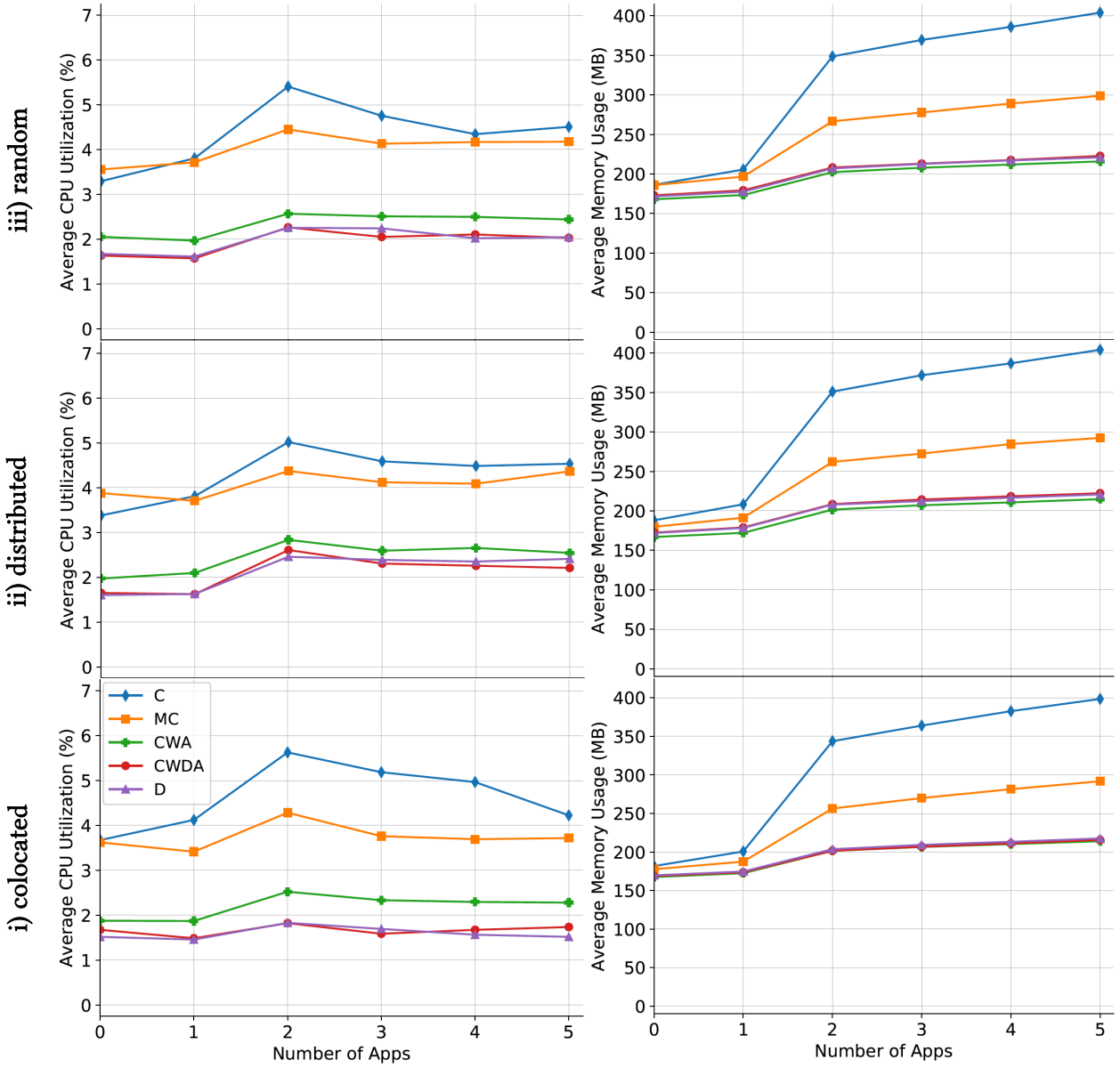


Figure 4.3: CPU usage, and memory usage measurements for the application scalability study. Each row denotes a device distribution scheme (colocated, distributed, random).

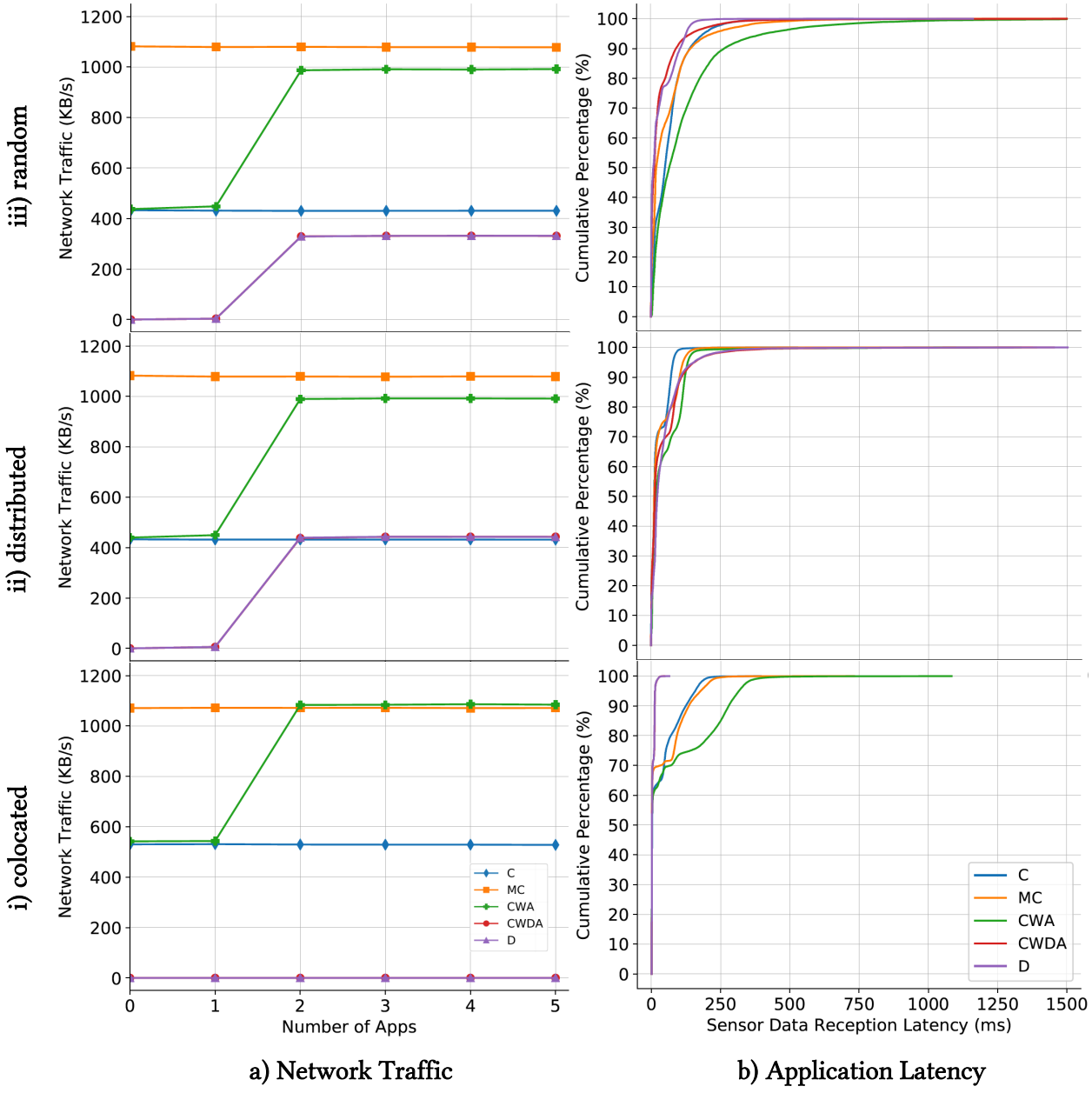


Figure 4.4: Network traffic and latency measurements for the application scalability study. Each row denotes a device distribution scheme (colocated, distributed, random).

CHAPTER 5
NEXUSEDGE: LEVERAGING IOT GATEWAYS FOR A DECENTRALIZED EDGE COMPUTING MIDDLEWARE

As motivated in the previous chapter, we finalized on using a decentralized architecture to support edge computing use cases for IoT devices on gateways. We hypothesize that an edge computing platform built using a network of IoT gateways can support and scale up with responsive edge applications and cutting-edge IoT devices, and eases application development, without requiring cloud support. To convert a disjoint network of gateways into a cohesive platform, we build a thin management layer which can handle application load balancing, optimize latency and network communication overhead, provide automatic discovery and scalability, ensure resilience to failures, and operate autonomously without any Internet connectivity. Then, we design an open-source edge computing platform, NexusEdge, which aims to manage heterogeneity, minimize configuration, aid in network management, and support applications.

In comparison, a cloud-based approach utilizes cloud APIs to support device interoperability, and elastic resources to operate with constrained resources.

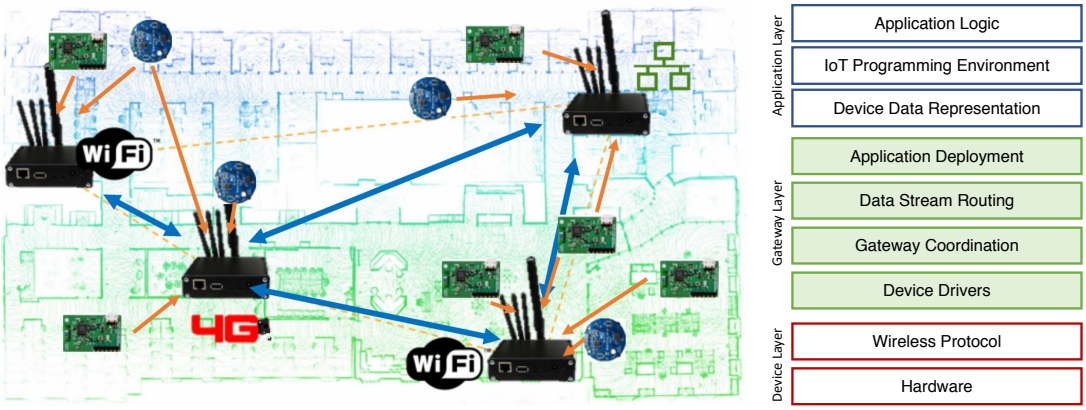
In this chapter, we describe the design of our middleware, NexusEdge, that runs on gateways which implements this decentralized architecture. We describe the NexusEdge design and deploy it to support an IoT testbed. We then validate availability of gateways in IoT deployments, compare our middleware implementation with a real world edge computing platform, and finally, we present a case study to showcase our platform's utility for real world applications.

5.1 Design

We first outline the goals necessary for a successful design to realize the selected decentralized architecture. We describe an overview of our design and the layers of the IoT stack that it focuses on. We then detail other key design elements which enable the outlined goals, and also discuss design alternatives that we considered but were ultimately insufficient.

5.1.1 Design Goals

To realize the selected decentralized architecture and evaluate potential design alternatives, we first describe the goals necessary for a successful design. **G1**: Manage heterogeneity. The Internet of Things is ripe with heterogeneity, encompassing different hardware platforms, data formats, communication protocols, operating modes, energy sources, and mobility patterns. Further, gateways themselves can be attached to different networks (e.g. WiFi vs. 4G). A successful design should cope with this heterogeneity. **G2**: Support applications. Ultimately, the goal of any edge platform is to enable applications to leverage the available data streams. The design should provide useful



(a) Platform Deployment (b) IoT Stack

Figure 5.1: NexusEdge platform across a floor of a building, and layers of the IoT stack for edge computing. We focus on the highlighted gateway layers, while ensuring that developments in other layers are compatible with the platform.

APIs and abstractions for applications while reducing complexity for developers. **G3**: Minimize configuration overhead. In small deployments the configuration required to add a new device or gateway is inconsequential, however, at scale per-device configuration is laborious. A design that simplifies this enables edge computing systems to be more widely deployed. **G4**: Simplify network management. Any deployment will require management over time, and a design that aids with this management is preferable to one that does not. **G5**: Provide resiliency. As gateways are typically not as robust as server-class machines, the design should still provide resiliency for applications and data streams.

5.1.2 Design Overview

NexusEdge uses multiple gateways to provide connectivity for IoT devices deployed throughout a space, as shown in Figure 5.1a. Each gateway is connected to one or more IoT devices. The gateways automatically discover each other to form the gateway platform, even if the gateways are connected to different backhaul networks. Once discovered, the gateways coordinate to provide applications with the illusion that there is a single gateway with connections to all devices. Any gateway can execute applications, and the platform optimizes where to execute applications locally.

The NexusEdge platform supports IoT-on-the-edge deployments, but does not address all challenges in developing IoT systems and applications. Figure 5.1b provides a view of many common IoT layers, and highlights the gateway-focused aspects that NexusEdge addresses. Other layers are meant to plug into the platform, and the platform's APIs are designed to ensure compatibility as new advancements are made in other areas, including wireless protocols, device standardization, and IoT programming frameworks.

5.1.3 Gateway Discovery

NexusEdge gateways must coordinate to create a cohesive platform, and the first step is a gateway discovery process. The simplest approach would be to expect manual discovery, where a user or network administrator explicitly configures a new gateway and adds it to the network. This is not consistent with **G3**, however, and complicates scaling the platform.

To avoid manual configuration, and since we assume the gateways are connected to some backhaul network (although not necessarily the wider internet), gateways could use an established network discovery protocol, such as the Simple Service Discovery Protocol (SSDP) [73] or IPv6's Neighbor Discovery Protocol (NDP) [74]. This fails in complex deployments—exactly the deployments where manual configuration is most difficult—as gateways may not be on the same LAN, and routers typically do not forward discovery messages.

Instead, we observe that since gateways by definition support wireless IoT devices that use common wireless protocols, gateways must be distributed spatially based roughly on the communication range of these protocols. Therefore, we perform wireless discovery using a standard IoT wireless protocol. Gateways send encrypted beacons and listen for other beacons to discover nearby gateways. This allows gateways to form networks based on their physical proximity, even if from a LAN networking perspective they are several hops apart.

5.1.4 Unified Gateway Platform

After gateways discover each other and create a common "gateway network", they must coordinate to provide a cohesive edge computing platform. Following the lead of today's edge-to-cloud IoT platforms like AWS IoT Greengrass [26], the platform could take a gateway-centric view and expose each gateway and the network topology to users, developers, and network administrators. This would enable the administrator to deploy applications to specific gateways and configure exactly how gateways share data streams. Since administrators know which applications they require and the physical deployment environment, they may be able to help optimize the platform. The drawback is that a real-world deployment is likely quite complex, and it is difficult to understand the optimal placement and expected wireless communication patterns between devices and gateways. Also, this sets a high bar for using the platform, and complicates development as applications need to understand the deployment topology. In fact, existing edge computing platforms aimed at smartphones require significant overhead for edge resource discovery and usage [75, 76].

Instead, our design folds the topology complexity into the platform, and presents the abstraction of the "single gateway model", as if all devices in the deployment are connected to the same gateway. This abstraction is essential for meeting goals **G1** and **G2**. Developers write apps as if all data streams are immediately available, and the platform itself chooses which gateway to actually execute the application on and ensures the necessary data streams are available to that gateway. The tradeoff is increased routing and platform complexity, but as we show in our evaluations, the overhead is minimal.

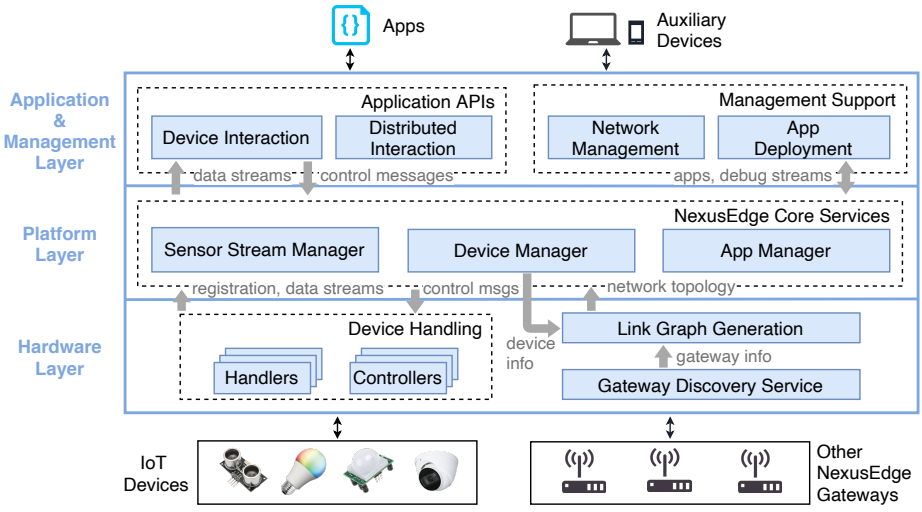


Figure 5.2: Layered architecture of NexusEdge.

To enable this abstraction, we maintain a network-wide graph data structure that encodes the topology of the gateway network with gateways as nodes, and discovery links (Section 5.1.3) as edges. This "link graph" also tracks which devices are connected to which gateways, the backhaul addresses of each gateway, and applications that are currently running on each gateway. All algorithms for routing data, placing applications, and managing device mobility only need the link graph data structure. The gateways update and propagate changes to the link graph as the network changes over time (e.g. if a new gateway or device is added).

5.1.5 Handling Device Heterogeneity

The key task of the gateway platform after discovery and unification is to actually interface with the distributed IoT devices. However, IoT devices are inherently heterogeneous, coming from different manufacturers with different sensors, energy sources, operating modes, form factors, data formats, and wireless protocols. Addressing this heterogeneity requires answering the question: between which layers of Figure 5.1b should the "narrow waist" for IoT networks be put? We explore the available options.

Unifying at the wireless hardware level (e.g. Thread [77], ANT [78], DigiMesh [79], and Sigfox [80]) has gotten little traction. As wireless protocols inherently contain numerous tradeoffs, it is not clear what the one wireless standard and data format every device should agree on. Alternatively, if every device did conform to some networking standard, for example IPv6 6LoWPAN, then IoT gateways could directly support that protocol, much in the way that a WiFi access point works. If or when such a standardization occurs, the device driver layer in Figure 5.1b could be simplified and the rest of the gateway platform would operate as described. Again, consensus

remains elusive in practice.

To avoid the overhead of full standardization, NexusEdge could specify only the data format (e.g. requiring JSON or protobufs). The gateways would need to support multiple communication protocols, but device data would be in a known format with a known schema. For example, the KubeEdge [81] platform requires a device model based on YAML for each device configured on its platform [82], and Azure IoT supports an optional device format in JSON [83]. Since not all devices currently share a common schema (although efforts like One Data Model [84], Open Data Format [85], and Zigbee Alliance's Dotdot [86] are attempting to), gateways would include small "converter" software modules that re-format data from devices into its canonical format. In attempting this design, we found it unsuitable as crafting a common data format works well for a few types of sensors, but it became increasingly difficult to conform to the standard format as we added more devices to the platform.

Thus, our platform design eschews any attempt to manage the heterogeneity, and instead embraces it, as shown in the hardware layer of Figure 5.2. We only require that devices have some identifier so we can track them on the platform, and some general type so we can cluster them to support application APIs. Our experience indicates these requirements enable IoT scaling and broad interoperability. Data sent to or from devices is treated entirely as a binary sequence with no expectations of structure, much like the application layer data of a traditional networking packet. Interpreting data or understanding devices is left entirely to upper layers. This approach maximizes device scalability by imposing the fewest requirements on devices, and allows the design to entirely support **G1**. Note, however, this design choice does not preclude using an IoT data schema with NexusEdge, just that the platform does not enforce it. As we show in Figure 5.1b, NexusEdge only provides a few layers of the overall IoT stack, and other layers can be plugged in to support different deployment and application objectives.

To support this device interface, the platform includes a three-function API for managing devices. This API must be implemented for new classes of devices.

- `register(deviceId, deviceType)`: Informs the platform of a new connected device with given ID and type.
- `deliver(deviceId, data)`: Pass the specified binary data from the device to the platform. The platform ensures that it receives data only from registered devices.
- `dispatch(deviceId, data)`: Pass binary data from the platform to the specified device.

5.1.6 Auxiliary Devices

Without a central entity to manage the distributed network, there is no clear interface to manage the overall platform. To provide such an interface, the platform enables devices including laptops and smartphones to temporarily connect to the network. These devices connect to a neighboring gateway using the same discovery radio as any other gateway. This is analogous to searching for

a WiFi connection from a mobile device where the device needs to be present in a WiFi router's coverage area. Once a device is connected to the platform as an auxiliary device, it can use the same internal platform APIs to manage the platform. This includes visualizing the link graph to study the network topology, deploying applications on the gateway platform, monitoring which applications are currently executing, viewing the standard logs of applications for troubleshooting, and terminating applications.

5.1.7 Application Support

The ultimate goal of the platform is to support applications that run locally on the edge operating on the rich sensor data. We do not specify the exact runtime and execution format for applications, but only specify how they interact with the core platform. Once an application is loaded (via an auxiliary device) to an arbitrary gateway, that gateway uses the link graph to identify gateways with spare compute resources, chooses a gateway which minimizes data transfer to execute the application (i.e. it tries to choose a gateway with the relevant sensors connected to it), loads the application on that gateway, and finally configures the relevant gateways to forward any relevant data streams to the executing gateway.

Due to the single gateway abstraction, developers can write applications as though data from every device in the network is immediately available. These applications interface with the platform through a narrow API with four functions. This API can be simple due to the design choice of using opaque binary data. Additional software layers outside the scope of this work can help with developers to manage the binary data.

- `receive(deviceId, callback(data))`: Request data from specific devices. Due to the event-based nature of sensor data, data is returned as a callback.
- `receiveAll(deviceType, callback(data))`: Request data from all devices of a certain type.
- `send(deviceId, data)`: Send data to a specific device. The transmitted data is similarly treated as a binary sequence, and could contain an actuation command.
- `sendAll(deviceType, data)`: Send data to all devices of a certain type. For example, an application can send a new set point temperature to all thermostats.
- `poll(deviceId, request, callback(data))`: Query a specific device by sending it binary request data and expecting a response. For example, an application can request an air quality sensor to get the latest measurements.

In addition to executing an application on a single gateway, developers can also leverage the distributed nature of the platform to execute the same application on multiple gateways. The platform provides a message passing interface to let application instances interact with each other.

This is especially useful to split and run applications at spatially close spaces, for example, different apartments in an apartment complex, or labs in a university building. The provided API follows.

- `disseminate(tag, data)`: Send a message to all other instances of the same application, with a tag to differentiate message types.
- `query(tag, query, callback(response))`: Retrieve disseminations for a specific tag from other instances. Similar to the data API, the events arrive as callbacks.

5.1.8 Providing Resilience

A key benefit of a distributed architecture is that the platform can provide better resilience for applications in the event of faults, when compared to a centralized architecture. Gateways can fail due to power loss, gateway movement, OS bugs, physical disturbances, or other issues. In the event of a gateway failure, the platform ensures resiliency to meet **G5**.

If a gateway that was executing an application fails, the application must be restarted on a different gateway. When the platform receives an application to schedule, it identifies an *executor* gateway to execute the application and a *watcher* gateway to watch for failures on the executor gateway. The watcher stores a copy of the application and periodically checks the link graph to see if the executor failed, and if so, the watcher chooses a new pair of executor and watcher gateways for the application, restarts the application, and stops watching. Additionally, the executor also checks for failures for its watcher, and if the watcher fails it nominates a new watcher for the application.

If a gateway that is collecting and forwarding data to an application fails, then the platform identifies one or more alternative gateways which can provide the same data streams to the application (since devices could be sending their data to multiple gateways). The executor gateway receives periodic heartbeat messages from each provider gateway, and if a provider fails, the executor uses the link graph to identify new provider gateways for the required data streams, trying to minimize the number of provider gateways (i.e., maximize the number of streams from each provider) to reduce dependency.

5.2 Implementation

In this section, we describe the testbed we use and the implementation specifics of our platform.

5.2.1 Hardware and Testbed

To prototype gateways we use Raspberry Pi 4 Model B [5] boards which support onboard BLE and WiFi, and we augment with an EnOcean [87] radio via USB. Any single-board computer should be sufficient. Our testbed has BLE and EnOcean wireless devices including temperature, door,

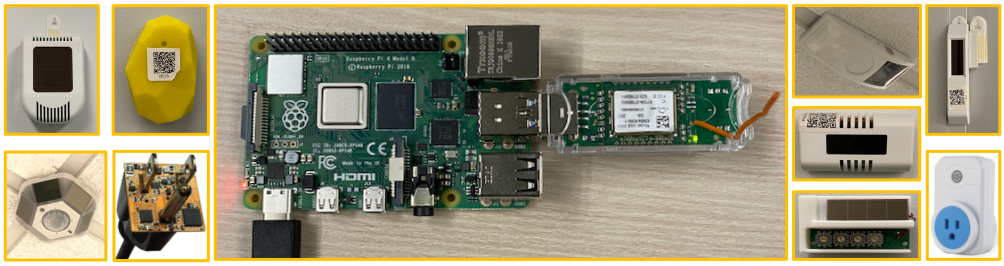


Figure 5.3: Gateways and devices in our deployment. Column-wise, top to bottom: CO_2 sensor, Occupancy sensor, Estimote beacon, Power meter, Raspberry Pi, Light sensor, Temperature sensor, Location beacon, Contact sensor, Smart socket.

lighting, air quality, and occupancy sensors, as well as Estimote beacons [88], power monitors, and smart sockets. The gateways and devices are shown in Figure 5.3.

5.2.2 Gateway Discovery using BLE

To minimize configuration overhead, gateways use their Bluetooth Low Energy (BLE) radio to discover each other. They send and receive discovery messages as BLE advertisements, using a pre-shared key that is made available during their initial configuration. The discovery messages contain the IP address of a gateway's backhaul network, and discovered gateways use this higher bandwidth network for further communication. The ubiquity of BLE in IoT gateways, its low power draw and cost, and native support for advertisements makes it a compelling choice for our discovery radio. Also, since laptops and smartphones have BLE radios, they can be *auxiliary devices* for management and application deployment.

5.2.3 Link Graph Network Abstraction

The link graph encapsulates knowledge about the current network topology. Each gateway tracks its discovered neighbors, connected devices, and running applications and encodes this as a graph. Additionally, each gateway hosts a web server which exposes this information to other gateways via well-defined endpoints. Once a gateway learns another's IP address, it can query the peer gateway to retrieve the link graph and update it with its local state. If the graph has changed, it also notifies the peer gateway. For simplicity, we encode the link graph as a JSON file. Other layers that rely on the link graph information simply parse the graph to adapt to the current network topology and structure.

5.2.4 Interfacing with Devices

NexusEdge handles device heterogeneity by not enforcing any constraints on communication protocols or data formats. It supports this with two module types: *controllers*, to communicate using

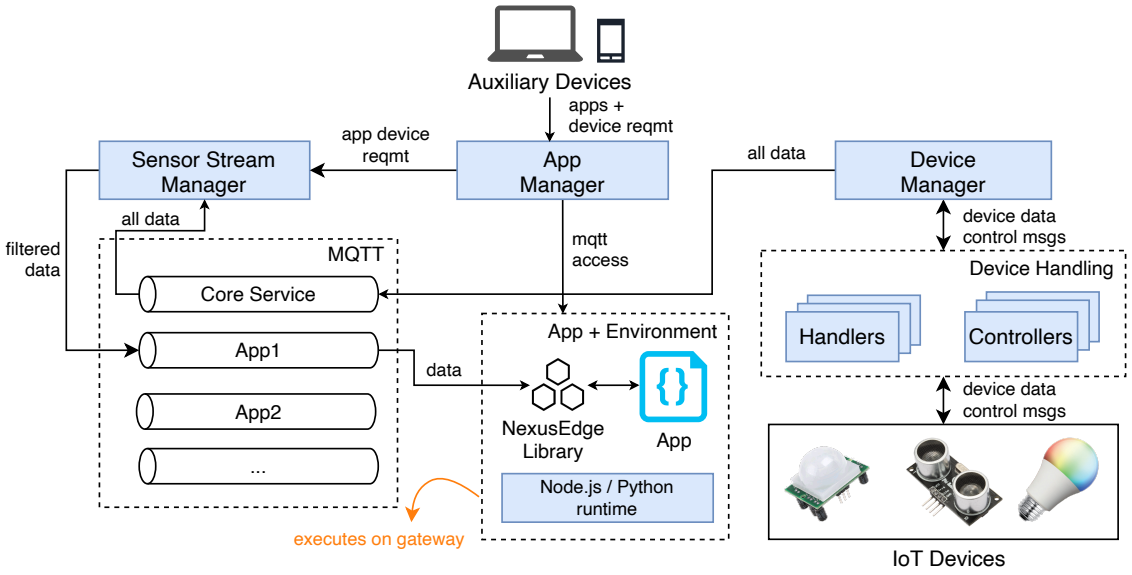


Figure 5.4: NexusEdge core service overview.

a specific wired/wireless technology, and *handlers*, the device-specific code to communicate with devices. Each handler plugs into a specific controller. Handlers must implement the device-facing API defined in Section 5.1.5. Because of a lack of standardization, each device type needs a custom handler (as in IoT systems like openHAB [89] or Home Assistant [90]), and users can load their custom controllers and handlers. The intent is a fixed number of controllers per-gateway (one for each communication channel), and device manufacturers can provide handlers for their devices.

5.2.5 Core NexusEdge Services

To enable the NexusEdge design, each gateway runs three background services, as shown in Figure 5.4. The *Device Manager* service loads controllers and handlers, tracks registered devices, and associates specific devices and device types with handlers. It receives data streams from registered devices, and publishes them to a *Core Service* MQTT topic for other core services to use. The *Application Manager* service accepts incoming requests to execute applications. It tracks which applications are currently running, sets up their execution environments, maintains log files, and processes termination requests. The *Sensor Stream Manager* service ensures access control of device data to applications. It branches data from the Core Service topic to different applications based on their requirements. If data is unavailable at the gateway, it requests other gateways to send data to the app.

```
// moisture sensors: "m1", "m2", "m3", "m4", actuators: "sprinkler1"

receiveAll("moisture", function(data) { // subscribe to "moisture" sensors
  const moisture = data.values.moistureLevel;
  storeDataPoint(data.deviceId, moisture);
});

setTimeout(checkMoistureLevels, period); // periodically observe moisture

function checkMoistureLevels() {
  const minMoisture = min(sensorIds.map(id => getLatestMoistureLevel));
  if (minMoisture < threshold) // turn on sprinkler, if low moisture level
    setSprinklerState(ON_STATE);
}

function setSprinklerState(state) {
  send("sprinkler1", {"state": state});
}
```

Figure 5.5: Sample NexusEdge application in Node.js to control a sprinkler based on soil moisture levels.

5.2.6 Applications on NexusEdge

The application APIs described in Section 5.1.7 support arbitrary execution environments. In our prototype, we support Node.js and Python based environments, to aid in asynchronous programming and machine learning. We package the APIs as a library, using MQTT for streaming data, and HTTP for other request-based operations. The application interface uses the device-facing interface for device interaction. An example NexusEdge application is illustrated in Figure 5.5.

5.2.7 Deploying and Scheduling Applications on the Platform

We provide a tool on auxiliary devices (user laptops) for users to deploy applications by selecting the runtime environment and the necessary sensor streams. The application is sent to the App Manager on any NexusEdge gateway, which selects the best gateway to execute the application. Our scheduler uses a simple heuristic based on data locality to identify the gateways that can provide the most number of required sensor streams. The scheduler also considers the CPU and memory usage of each gateway, and selects a gateway to promote locality while avoiding overburdening a single gateway.

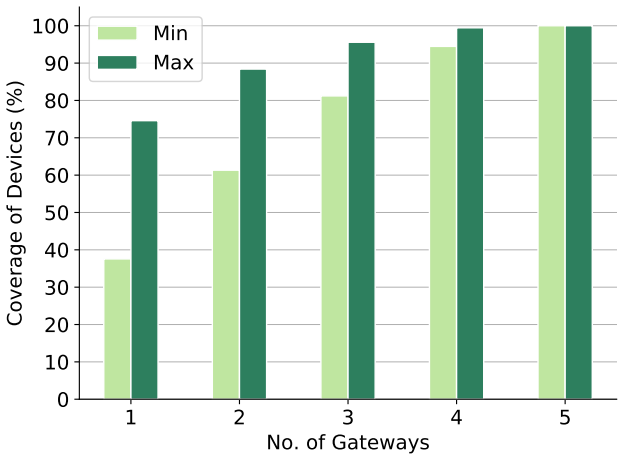


Figure 5.6: Device Coverage

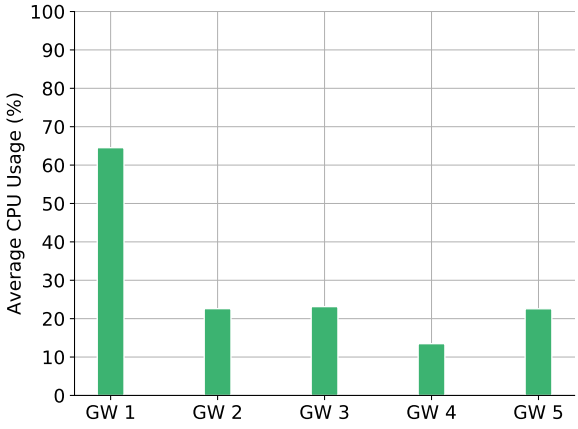


Figure 5.7: Gateway Load

5.2.8 Containerization and Deployability of NexusEdge

Gateway devices are heterogeneous and have different hardware or runtime environments with various subtle differences. To handle this heterogeneity, we containerize our middleware using Docker [91] to provide operating system-level virtualization. This vastly improves the ease of deployment, and abstracts out the hardware, OS, and runtime environment. The NexusEdge Docker container is public on Docker Hub [92].

Additionally, to reduce the tedious burden of deployment on many gateways, we utilize K3S [93], a lightweight version of Kubernetes [94] container orchestrator built for IoT and Edge Computing. Gateways join a cluster, and the cluster executes a DaemonSet [95] to start the NexusEdge container on all cluster nodes. DaemonSet ensures that NexusEdge starts automatically on all new gateways.

5.3 Evaluation

In this section, we first validate the availability of multiple gateways in device deployments, which are available to be utilized to execute edge computing applications. We then compare our platform to a real-world edge computing platform. We also evaluate the resiliency of our middleware to gateway failures. We compare the decentralization overhead for application deployment on NexusEdge to a centralized edge server. Finally, we focus on a case study to showcase our platform's utility for real world applications.

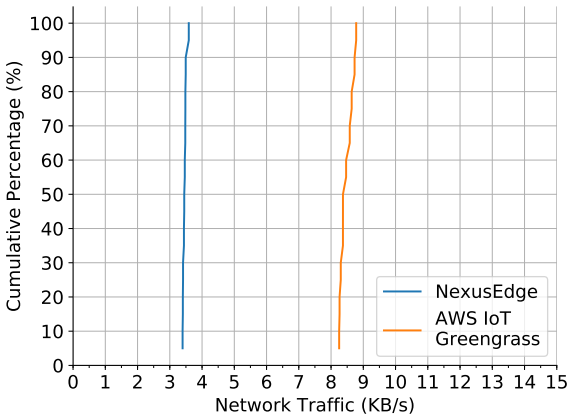


Figure 5.8: CDF for network traffic

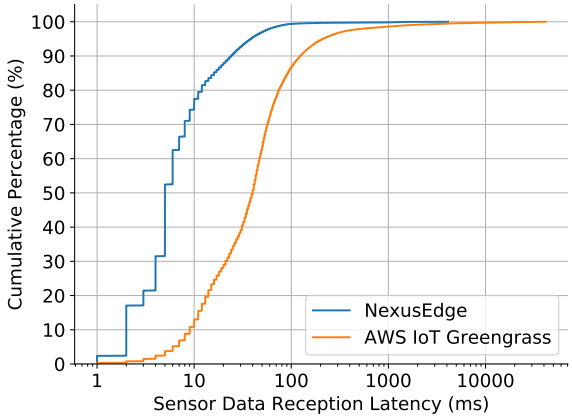


Figure 5.9: CDF for application latency

5.3.1 Availability of Underutilized Gateways in IoT

Our motivation for this work stems from our deployment experience of setting up a testbed of 181 wireless IoT devices in the floor of a building spanning 17,000 sq.ft. As our IoT devices have limited transmission ranges and were spread across a large area, we deployed five gateways to collect data from the devices and send it to a cloud service. We conducted some preliminary experiments with the gateways to study the number of devices they covered and their CPU loads.

Figure 5.6 shows the minimum and maximum number of devices that can be covered with different combinations of our gateways. We observe that for our deployment, a single gateway at best can cover 74% of devices. To cover more than 95% of the devices, we need at least 3 well-positioned gateways, and 5 are required for full device coverage. This suggests that **multiple gateways are required to cover IoT devices** in moderate-sized deployments like ours.

We then investigated the workload of the gateways in our deployment, which were collecting sensor data on multiple wireless interfaces, formatting data packets, and publishing data to a cloud time-series database. We measured the average CPU usage of our gateways for 60 days and the results are shown in Figure 5.7. We noticed that most gateways are lightly loaded and spend around 80% of their CPU time idle, with the exception of Gateway 1 which is fetching additional data streams from four different web APIs. This suggests that **IoT gateways have underutilized computational power**.

These insights indicate the availability of multiple underutilized gateways in IoT deployments, which could be leveraged for edge computing.

5.3.2 Comparison with AWS IoT Greengrass

To validate the results from the evaluation in Section 4.3, we compare our middleware's performance with AWS IoT Greengrass. Greengrass has an architecture that is similar to the centralized

edge computing architecture (Figure 4.1a). We evaluate with our testbed and an application for monitoring users' thermal comfort.

5.3.2.1 Evaluation Setup

This test application obtains data from 70 actual sensors from our deployment (14 occupancy sensors and 56 temperature sensors). The occupancy sensors only send data when they detect motion, and the temperature sensors send data every 5 minutes. We partition the space into 14 regions based on the location of the occupancy sensors, and an app analyzes the average temperature of each region. Every 5 minutes, the app checks if the temperature of all regions are within an acceptable comfort range. It also reports energy wastage for unoccupied and overcooled regions. We built the application for our platform and for the Greengrass platform.

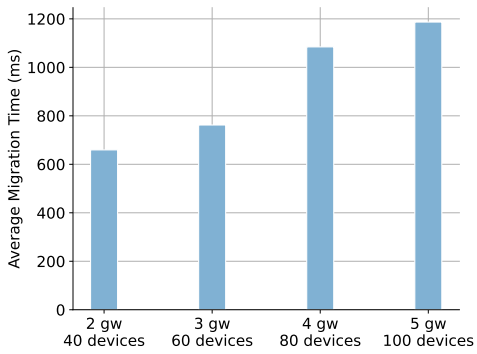
5.3.2.2 Preliminary Results

We let the application run for 30 minutes, and measured network traffic, and time taken for sensor data to reach the app. We repeat this for 10 runs and plot CDFs for the network traffic and the sensor data reception latency, as shown in Figure 5.8 and Figure 5.9. The mean network traffic for our platform is 3.46 KB/s, 2.5x lower than the 8.48 KB/s of Greengrass. The application latency for Greengrass is 142.74 ms and 13.83 ms for NexusEdge, which is 10x faster. This is because Greengrass requires one gateway to be configured as the "Greengrass Core" which is the only one that can run apps. This limits application optimizations like shifting apps closer to device streams. We take advantage of this to reduce the latency and network traffic.

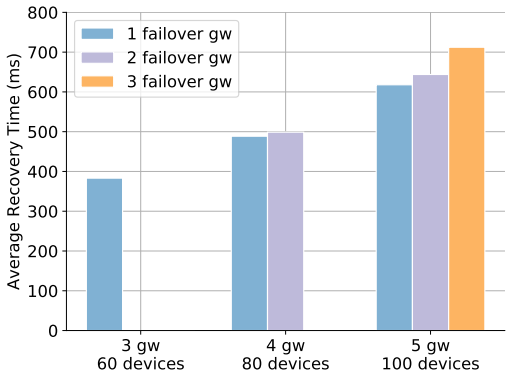
The results shown here are for a single application, and in centralized platforms it will exacerbate when there are more applications, devices, or devices with higher data rates. Increased network traffic would also result in applications that are less responsive. This experiment also highlights the benefits of executing applications at the gateway layer, one hop closer to the devices, rather than at a central point like an edge server.

5.3.3 Resiliency to Gateway Failures

Since gateways are not as sturdy as edge servers, they can be subject to faults such as unexpected failures, deployment changes, etc. The NexusEdge middleware ensures application resiliency in two scenarios: (a) *executor failure*: if the executing gateway fails, the application is migrated to a different gateway, and (b) *provider failure*: if a gateway providing data streams to an application fails, the middleware reinstates data streams from other gateways if possible. For both scenarios, we use a periodic timer of 60 seconds to detect the failure, which is configurable. As the time taken to detect a failure depends on this period, we skip measuring the failure detection and instead evaluate the time it takes to recover once a failure is detected. We describe our evaluation setup and results below.



(a) Resilience to Executor Failures



(b) Resilience to Provider Failures

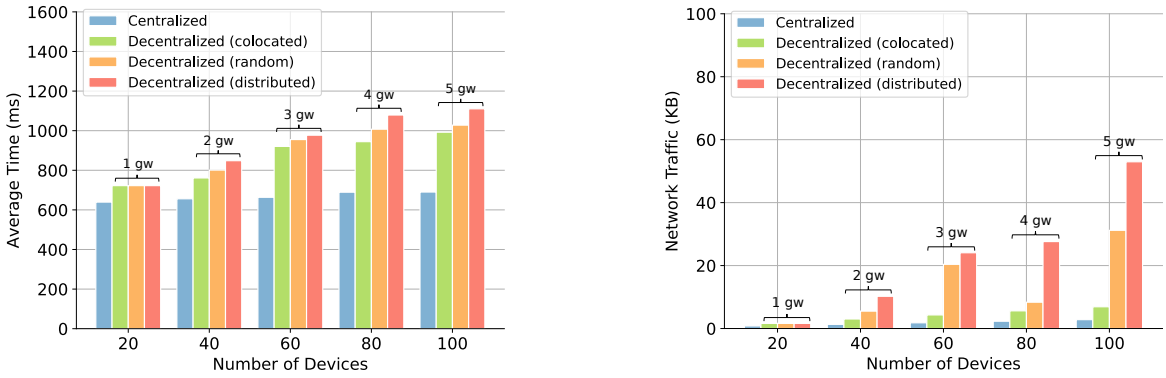
Figure 5.10: a) and b) illustrate the application migration and recovery time respectively for the middleware when gateways fail.

(a) Executor Failure: We define migration time as the time taken to restart the application on a different gateway, when the executor fails. This includes time for rescheduling the application, generating the link graph, dispatching the application to the new executor, and setting up data streams from provider gateways. Since the link graph generation time is dependent on the number of gateways and devices, we measure the migration time with varying number of gateways and devices.

We start with 2 gateways and 40 devices and go up to 5 gateways with 100 devices, with each gateway hosting 20 devices. We generate synthetic data for devices, with each device connected to *exactly* one gateway. We execute an application on one of the gateways which receives data streams from all devices. We then fail the executor gateway and measure the time taken to migrate the application once the failure is detected. We repeat this 10 times and plot the average migration time as shown in Figure 5.10a. Migrating an application that requires data from 100 devices distributed among 5 gateways only takes around 1.2 s, and the migration time follows a linear trend as gateways and devices increase.

(b) Provider Failure: If a gateway that was providing data streams fails, and there are other *failover gateway(s)* which can provide the same data streams, the middleware requests the failover gateway(s) for the data. We define the recovery time as the time it takes to complete these requests and reinstate data streams. This includes generating the link graph, identifying failover gateways, and requesting them for data.

Similar to the previous experiment, we varied the number of devices and gateways, but from 3 gateways and 60 devices to 5 gateways and 100 devices. We excluded the 2 gateways case, since there is no failover gateway when one of the gateway fails. For 4 gateways, after a providing gateway fails, there can be 1 or 2 failover gateways that can provide the streams. As the number of failover gateways increase, the recovery time slightly increases as more gateways need to be



(a) Deployment time overhead

(b) Network traffic overhead

Figure 5.11: Illustrates the decentralization overhead in terms of (a) deployment time, and (b) network traffic, for deploying an application on NexusEdge compared to a centralized edge server.

requested to forward data streams. We distributed devices among gateways so that there are some redundant providers and thus a scope to recover when the original provider gateway fails.

We induced failure for one of the provider gateways, and measured the average recovery time for 10 runs as shown in Figure 5.10b. Since reinstating data streams doesn't require migrating the application, the recovery time is faster than migration time, taking only 712.3 ms even with 5 gateways, 100 devices while using 3 failover gateways.

Both these experiments show that NexusEdge can recover quickly from gateway failures, by migrating applications or reinstating data streams from alternative providers. It highlights the significance of our middleware in building a cohesive edge platform with gateways, that can operate with resiliency without suffering from a single point of failure.

5.3.4 Decentralization Overhead

Shifting from an edge server to decentralized gateways adds some overhead for gateway coordination, and we evaluate if this overhead is reasonable. We compare the time taken to deploy an application on a centralized edge server and on NexusEdge. For the edge server, scheduling is not needed, as the application executes on the same machine. Also, since all device streams are available on the server, deploying the application only includes sending it to the server, and to subscribe to the MQTT broker providing all streams. However, for NexusEdge, deploying an application involves sending it to a scheduling gateway, generating the link graph, scheduling the application, dispatching the application to an executing gateway, setting up streams from provider gateways, and subscribing to all data streams.

We deploy an application which receives data from multiple devices (virtual sensors sending a small payload every 5s), and measure the deployment time at a varying number of devices. For centralized, the application always executes on the server, and we vary the number of devices from

20 to 100, at steps of 20. For decentralized, we vary the numbers of gateways and devices, starting from 1 gateway with 20 devices and going up to 5 gateways with 100 devices. We also consider three device distribution schemes i.e., how devices are distributed among the gateways: colocated (best case), random (average case), and fully distributed (worst case), as the distribution affects the latency (Section 4.3.3). We measure the deployment and the results are shown in Figure 5.11a. We also analytically calculate how much network traffic deploying an application in each of these scenarios would consume, and show the results of this analysis in Figure 5.11b.

As expected, the deployment time for centralized is lower than decentralized. But even with 5 gateways, 100 devices, and the worst case device distribution, the deployment time for NexusEdge is 1110.4ms compared to 690.1ms for centralized, with an overhead of 420ms. We also find that in the worst case, the network traffic for exchanging decentralization messages is only around 50KB compared to 2KB for the centralized server. These overheads are reasonable since deployment for long running applications happens infrequently. Also, applications in centralized require additional time to filter out data streams, which is handled for NexusEdge and included in its deployment time.

5.3.5 Case Study: Federated Machine Learning without Cloud Support

We describe an illustrative example application on NexusEdge to predict turn on times of an appliance based on its power data. Accuracy of the application can be improved by training with data of the same appliance type from different sources. For instance, usage data of dryers from multiple users can be used, but users may not be comfortable sharing such data. So we modelled this as a federated learning problem.

Federated ML is typically used by mobile devices to learn a shared prediction model. Devices keep their data private, but can still run predictions, and retrain their models. Changes from all local models are sent to the cloud to improve the shared model, which is then synchronized back. We demonstrate how our platform can natively run federated ML on gateway devices without the need of a cloud.

5.3.5.1 Modeling Federated ML on NexusEdge

As described in Section 5.1.7, NexusEdge allows an application to be distributed on different gateways. Each app instance uses the `receive` function to receive power meter data from the gateway it runs on. The apps are initialized with the global model, and retrain their models whenever they get substantial amount of data. Over time, when the data received by an app crosses a threshold, it uses the `query` function to request model parameters from all other apps. It then performs a federation step by taking a weighted average of model weights to generate an updated model. It uses the `disseminate` function to share this updated model to all other apps.

Measuring Overhead and Performance: The query and disseminate calls only amount to 25 LOC. The model parameter exchanges were each of 8 KB. One federation step required 32 KB of data

transfer between the apps. For the federated learning model, we used a neural network with two hidden layers and used all time-based features. We simulated appliance level power data from the UK-DALE dataset [96] (around 900,000 data points). The model's accuracy increased from 73.39% to 78.93% after one round of federated learning. To measure the effectiveness of the distributed approach, we also compared this with a non-federated baseline (which has access to all data) which reported an accuracy of 79.93%.

The accuracy numbers do not mean much in the context of our platform. The key takeaway is that NexusEdge can ease the burden in developing complex IoT applications by providing useful abstractions, including a distributed interaction API.

CHAPTER 6
MEDLEY: UTILIZING AMBIENT COMPUTE FOR SUPPORTING REALTIME, COMPUTE-INTENSIVE IOT APPLICATIONS

Using the NexusEdge middleware on decentralized edge gateways can support a good set of IoT applications on the edge, without relying on the cloud or other expensive edge infrastructure. The IoT applications we evaluated on NexusEdge are limited to long-running tasks without any quality of service (QoS) requirements. However, there is a wide range of edge IoT applications which are short realtime tasks with higher compute requirements and tight QoS requirements. For example, assisting users in cognitive decline using augmented reality (AR) headsets [97], and detecting handwashing events on smartwatches for preventing diseases [98], are representative realtime edge IoT applications with firm QoS requirements.

Supporting compute-intensive, realtime IoT applications require increased compute power, which is a scarce commodity at the edge. Ideally, we would have preferred to have a high performing computing infrastructure on the edge, as assumed in several edge computing works in the literature. However, not all IoT deployments have access to such an infrastructure due to high setup costs, or limited availability of third party edge infrastructure.

We explore an alternative approach to bridge this gap by utilizing unused compute available in certain edge IoT devices. We study how resource-constrained IoT devices like AR/VR headsets, smartwatches, and other low-power IoT sensors can improve the quality of service of their applications by relying on another class of smart home and office devices which include smart TVs, gaming consoles, and smart doorbell cameras. Compared to resource-constrained IoT devices, these devices have three pertinent characteristics: i) better computing resources, ii) considerable idle time, implying computational resources to spare, and iii) a steady source of energy. In this work, we answer the question: "can we offload tasks from resource-constrained IoT devices to nearby resource-capable IoT devices to meet QoS requirements while minimizing energy overhead?".

We hypothesize that an IoT ecosystem where devices cooperate through task offloading could help improve the quality of applications on resource-constrained devices. For instance, we can reduce the latency of cognitive assistance on AR headsets by submitting video frames to a nearby Google Nest Doorbell [99] for object detection, or improve the prediction quality of hand washing detection on the smartwatch by utilizing a more powerful GPU on a nearby PlayStation 5 [100] to run a deep learning model. This enables resource-constrained devices to operate on a lower energy budget while trading off for the data transmission cost.

We design Medley, which is an IoT device ecosystem, in which resource-constrained "offloader" devices are designed and manufactured with the assumption that ambient compute resources from resource-capable "executor" devices are available, just like how current IoT devices

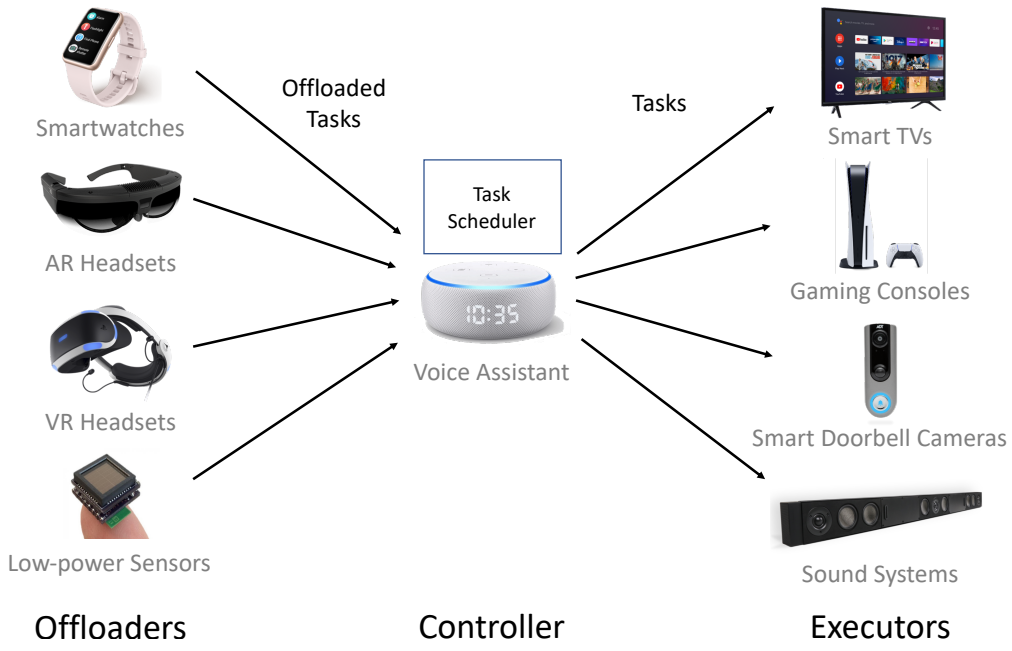


Figure 6.1: An overview of the Medley ecosystem.

are designed with an assumption that WiFi connectivity would be available at the user's home. Offloader devices are not "thin clients" [101, 102] which have no compute available, but rather devices which have enough compute to do their basic functionality, with an opportunity to improve their overall quality of service when present in a Medley ecosystem. Additionally, the ecosystem will have a "controller" device which could, for instance, be a voice assistant like Google Home, which accepts offloaded tasks from offloaders, schedules the tasks on executors, and sends back task responses to offloaders. An overview of how devices in the Medley ecosystem interact is shown in Figure 6.1. Such an ecosystem utilizes compute resources on executor devices in lieu of a dedicated edge computing infrastructure for running offloaded tasks. The key component of our Medley ecosystem is a realtime task scheduler that identifies the best executor to offload to, given the QoS requirements of the offloaded task, executor resource usages, and energy constraints. It also requires a realtime task scheduler that identifies the best executor to offload to, given the QoS requirements of the offloaded task, executor resource usages, and energy constraints.

We design a task scheduler that is better suited to provide real-time guarantees and minimize executor energy usage while being heterogeneity-aware without requiring a priori task profiling. Our key insight is that tasks in a given environment usually repeat, and it gives an opportunity for

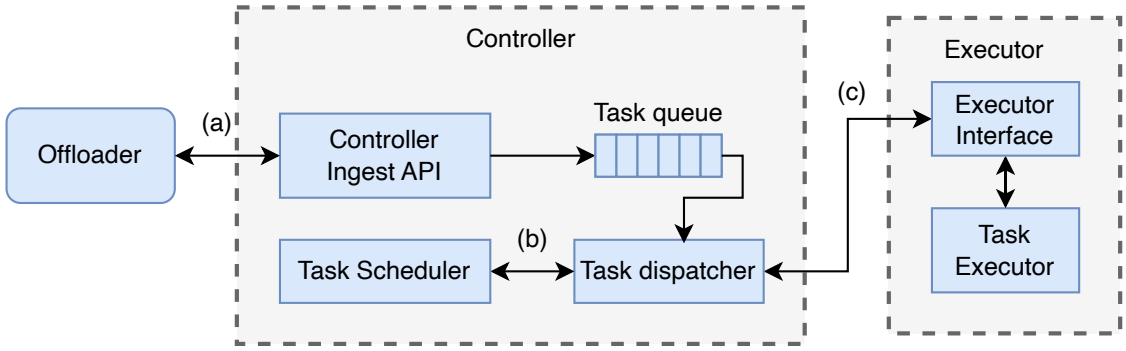


Figure 6.2: An overview of the design of the NexusEdge system. Offloaders send tasks to the controller (a), the controller assigns tasks to executors (b) through a task scheduler, and the controller oversees the completion of the task, sending the result to the offloader.

a task scheduler to see how they perform on each executor device. Over time, for each task, the scheduler can learn the best executor device to satisfy the task's QoS requirement while minimizing the energy consumed by the executor devices. Thus, such a task scheduler eliminates the need for *a priori* task profiling, and supports profiling tasks on the go, learning their execution time, resource impact, and energy impact on each executor to satisfy task QoS while minimizing overall energy of the ecosystem.

In comparison, a cloud-based approach utilizes cloud APIs to handle gateway heterogeneity, and Function-as-a-Service (FaaS) to support dynamic IoT workloads.

In this chapter, we discuss the design of our Medley ecosystem and its task scheduler, our prototype implementation, and the evaluation of the Medley ecosystem.

6.1 Design of the Medley Ecosystem and its Task Scheduler

We design the Medley ecosystem to enable resource-constrained IoT devices to offload diverse tasks with varying QoS requirements to more resource-powerful IoT devices. In this section, we first describe the various components of the ecosystem. We then describe the design of the task scheduler, including the design requirements and the scheduling algorithm we use.

6.1.1 Components of the Medley Ecosystem

In the Medley ecosystem, devices can fall into three categories: offloaders, executors, or the controller. An overview of how these devices interact with each other is shown in Figure 6.2. We describe the design choices we made for different components in the ecosystem below.

6.1.1.1 Tasks

A task is the unit of work in Medley that a device is capable of offloading to another device within the ecosystem. A developer describes tasks as part of their application. Each task, like a function in code, accepts some input and computes some corresponding output. In addition to the other parameters, for each task, we have a deadline associated with it, which indicates the time the offloader expects the task to be finished by. For example, an offloader could request to run an image classification task with an input image and the output could be a label for the image, and the deadline could be 5 s.

We assume that the code for the task exists on all the devices which execute the task on behalf of the offloading device. We envision that much like smartphone applications have corresponding companion applications for smartwatches, devices that may accept work from offloading devices will install task code for applications in the ecosystem.

6.1.1.2 Offloaders

We classify any device with constrained computing capabilities relative to most computers or with tight energy budgets a candidate for being an offloader within Medley. These kinds of devices would benefit from having other, more powerful devices performing a portion of their work, which may take much longer or have prohibitive energy costs if the offloader completed it locally. For example, this class of devices could include hardware such as smartwatches, smart glasses, AR headsets, or any energy-harvesting sensing devices.

In order to offload a task, the offloader communicates with the Medley *controller*, identifying the task it is offloading and providing input data for the task. While the task is executing elsewhere in the ecosystem, the offloader can conserve energy in a low-power state or perform other work it is better suited to take on locally. The offloader will receive the result of the task it offloaded from the controller once the executing device completes it.

6.1.1.3 Executors

We define an executor as any device within Medley that accepts work from offloaders. Devices that may spend considerable amounts of time idle, have more computing capabilities than offloaders, and have stable sources of energy are candidates for being executors. For example, these include devices such as smart doorbell cameras, gaming consoles, smart TVs, and conventional laptop and desktop computers. As previously described, executors store the code necessary to execute tasks from offloaders. Upon receiving input data for a task, the executor will run the task and send the result back to the offloader through the controller.

Because an executor's primary purpose is not to run offloaded tasks, we take into account the fact that the executor may already be sufficiently loaded to handle its own workload (e.g., a gaming console may be in use rendering a game state). An executor's load will also vary depending on its hardware capabilities and load from concurrent tasks. We consider this by having each executor

provide metrics, such as CPU load, available memory, and GPU usage, that represent its current load available for use by the scheduler in the Medley ecosystem.

6.1.1.4 Controller

We use a designated controller device in the Medley deployment to manage the process of accepting tasks from offloaders and distributing them to executors. The controller is aware of all executors available to perform tasks and performs scheduling, minimizing the offloaders' role in task distribution and allowing them to conserve energy. It is also responsible for routing offload requests from offloaders to executors and their corresponding results back to the offloaders.

Upon receiving an offload request from an offloader, the controller makes a scheduling decision to choose an executor it believes can meet the task's QoS requirement (deadline). It then sends the input data to the executor and requests it to run the task and awaits the result. Once it receives the output data from the executor, it communicates the result back to the offloader.

6.1.2 Task Scheduling Problem

We make three key requirements for the task scheduler we design for the Medley ecosystem. First, profiling a task ahead of time for execution time or energy consumption is not scalable for IoT given its hardware and task heterogeneity, and thus profiling is avoided. Second, even though executors are wall-powered, there is an opportunity to reduce the energy consumption of the executors. It is beneficial to choose executors that use less energy and still satisfy deadlines, in lieu of always assigning tasks to the most powerful executor. Finally, since the Medley ecosystem is opportunistic and executor devices are not guaranteed to be available at all times, the design should tolerate device failures.

Based on these requirements, the scheduling problem is stated below:
Given a task with some input data and a deadline, identify an executor from a set of executor devices, which can meet the task's deadline, along with the following requirements for the scheduler: i) does not require any profiling data, ii) attempts to save energy on the executors, and iii) can tolerate device failures on the executors.

Designing a scheduler is challenging due to some uncertain factors with the tasks and the executing devices. First, the set of tasks the scheduler may handle is large and diverse because of the many types of devices that may be offloaders in the ecosystem; even offloaders with a similar function may have different tasks to offload (e.g., an AR headset may offload image recognition, and another may offload natural language processing tasks). It is especially challenging for the scheduler to choose an optimal device because it cannot rely on any profiling data for the tasks. Second, the execution time of a task will vary based on how well-suited an executor is for completing it, which itself will depend on both static and dynamic characteristics of the executor itself. Ever-changing executor load state with numerous possible concurrent task combinations make schedules based on static task execution times unreliable; executors will additionally have their

own workloads to run, further contributing to the load state. It is also possible that execution times for the same task vary depending on the input data.

6.1.3 Design of the Medley Ecosystem's Task Scheduler

Our key insight is that in a given environment we can expect offloaders' tasks to repeat, the scheduler will have ample opportunity to learn and quickly improve its decisions over time. By giving the scheduler visibility into likely relevant characteristics of tasks and executors, it enables it to explore how well each task runs on the ecosystem's executors and discover what properties are most relevant to improve the task's quality of service.

We design a stateful task scheduler which uses static and dynamic characteristics of executors to inform its scheduling decision, and uses feedback to further improve its future decisions. The scheduler is initialized with each executor's static characteristics including number of CPU cores, some CPU performance benchmark score, and the electrical power required to execute the benchmark. Additionally, before making a scheduling decision, it collects the current CPU load state of all the executors. It uses these static and dynamic characteristics of executors to choose an executor for a task. More specifically, it tries to pick an executor with the best performance per watt score. It also keeps track of how many active tasks it has scheduled on a given executor and doesn't schedule more than the number of CPU cores on the machines. Additionally, it doesn't pick an executor if the 1-minute average CPU load is greater than some threshold value. The algorithm we use for scheduling a task is listed in Listing 6.1.

```
MAX_NORM_CPU_LOAD = 0.75
MAX_ALLOWED_DEADLINE_MISSES = 1
task_executor_candidate = [[True] * num_executors for i in range(num_tasks)]
no_of_active_tasks = [0] * num_executors  # number of tasks already scheduled
deadline_miss_count = [[0] * num_executors for i in range(num_tasks)]

function schedule(task_id, normalized_cpu_loads):
    """
    Returns the id of an executor to schedule a given task to.

    task_id: id of the task to be executed.
    normalized_loads: current 1 min avg CPU Load/CPU count for each executor.
    """
    # sort executor_ids according to their CPU benchmark score / Power values
    sorted_executor_ids = sorted(executor_ids, key=scores, reverse=True)
    picked_executor_id = sorted_executor_ids[0]

    for executor_id in sorted_executor_ids:
        if task_executor_candidate[task_id][executor_id]:
            if no_of_active_tasks[executor_id] < cpu_counts[executor_id]:
                if normalized_loads[executor_id] < MAX_NORM_CPU_LOAD:
                    picked_executor_id = executor_id

```

```
    no_of_active_tasks[picked_executor_id] += 1
    return picked_executor_id
```

Listing 6.1: Algorithm for the Medley schedule function.

When an executor finishes executing a task, the scheduler receives feedback on whether the executor was able to meet the task's deadline or not. We define two modes for our task scheduler based on how it reacts to this feedback, a Performance Mode (Medley-perf) and an Energy Savings Mode (Medley-energy). In performance mode, if an executor fails to meet the deadline for a task, it never picks that executor again. Next time, it picks an executor which performs better but might incur a higher energy on the executor machine. However, in the energy savings mode, it allows multiple deadline misses on an executor (we choose five for our implementation), as defined by MAX_ALLOWED_DEADLINE_MISSES in our algorithm. The energy savings mode is beneficial for more energy conscious deployments, where Medley tries to save executor energy in addition to meeting deadlines. The algorithm for how Medley handles feedback upon finishing a task is shown in Listing 6.2.

```
function task_finished(task_id, executor_id, deadline_met):
    """
    Called when an executor finishes executing a task.

    task_id: the id of the task which finished executing.
    executor_id: id of the executor which executed the task.
    deadline_met: indicates whether the tasks's deadline was met or not.
    """
    no_of_active_tasks[executor_id] -= 1
    if not deadline_met:
        deadline_miss_count[task_id][executor_id] += 1

        if deadline_miss_count[task_id][executor_id] >=
   MAX_ALLOWED_DEADLINE_MISSES:
            task_executor_candidate[task_id][executor_id] = False
```

Listing 6.2: Algorithm for the Medley task_finished function which is called when an executor finishes executing a task.

For our Medley task scheduler design, we make some assumptions. We measure the execution time of a task as the time it takes for the controller to send the task input to the executor, the time for the executor to compute the task, and the time for the executor to return the response back to the controller. We do not consider the time it takes for the offloader to send the task data to the controller in our time measurements. Second, we assume that deadlines for a given task do not change at a later point of time. This is important for our task scheduler to improve based on the feedback it receives. Finally, since our task scheduler has to do some learning from feedback as it does not use task profiling data, we assume the tasks running in our system to have some room for failure, i.e., they are not hard real-time tasks.

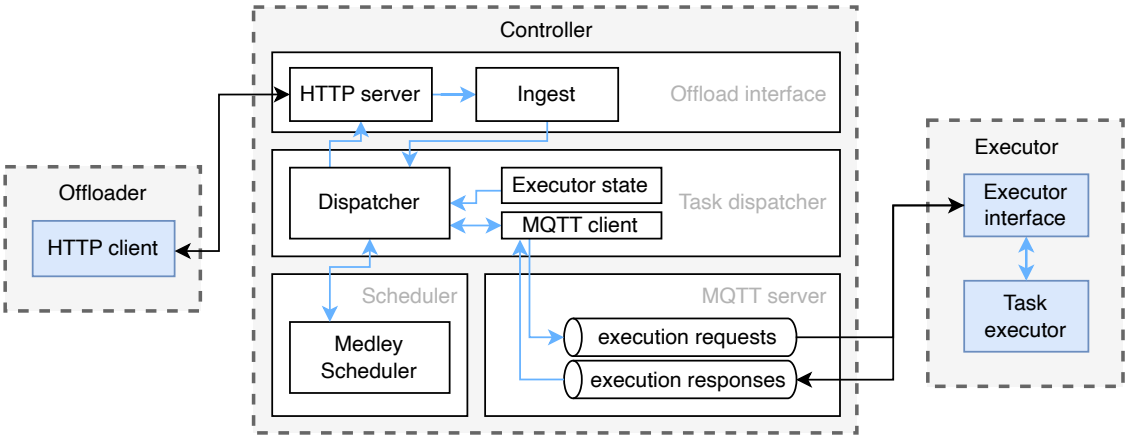


Figure 6.3: Overview of Medley ecosystem implementation.

6.2 Implementation

We describe the workflow of how a task is executed in Medley, the implementation details of the controller, executors, offloader, and the task scheduler.

A task submitted from an offloader is received by a controller, which schedules it on an executor and sends back the response to the offloader. The controller runs an HTTP server to accept task requests from offloaders. It houses a task dispatcher module which processes incoming tasks and sends back responses to offloaders, and the task scheduler. When a task arrives, the task dispatcher sends HTTP requests to each executor to get their current load state. It then sends the tasks and current load state to the task scheduler and requests it to find an executor. Once an executor is identified, the task is sent to the executor via MQTT for execution. The executor executes the task, and responds back on MQTT the time taken and energy consumed to execute the task, and its load state after execution. This response is given to the task scheduler to complete its feedback loop. The task response is sent to the offloader over MQTT. The block diagram for our implementation is shown in Figure 6.3. We use Python for our implementation.

For obtaining the capabilities and load state in the executors, we use the psutil [103] Python library. To obtain the GPU load, and available VRAM, we use the nvidia-smi [104] utility on the desktop and tegrastats [105] for the NVIDIA Jetson SBCs. The executor reports the following capabilities and load states: CPU count, 1-minute CPU load average, free RAM, total RAM, GPU available, GPU load, free VRAM, total VRAM, process count, and thread count.

We don't use the GPU to speed up any of our tasks, since we found that the CPU outperformed the GPU for these tasks due to the overhead in shifting input data to the GPU memory.

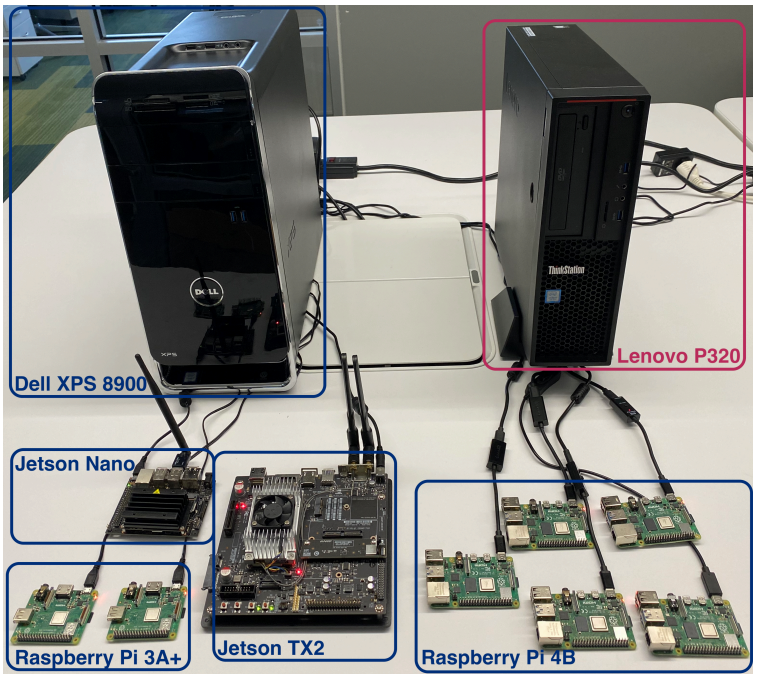


Figure 6.4: The devices we use in our evaluation. There are ten executors (blue) and a controller (red).

6.3 Evaluation

We evaluate the Medley implementation to see how well it can support realtime edge computing tasks offloaded from resource-constrained IoT devices. Specifically, we evaluate the task scheduler that we implemented with other state-of-the-art schedulers and common scheduling algorithms. We also evaluate the Medley ecosystem on how well it can support edge computing on the ecosystem of IoT devices, compared to a dedicated edge infrastructure.

Since we evaluate on realtime tasks with deadlines (specifying QoS requirements), the primary evaluation metric is the Deadline Satisfaction Ratio (DSR), which measures the ratio of the number of deadlines that were met out of all offloaded tasks.

6.3.1 Evaluation Setup

A desktop PC hosts the *controller* which performs the task scheduling. Our *executors* consist of ten systems: two Raspberry Pi 3A+s, five Raspberry Pi 4Bs, one NVIDIA Jetson Nano, one NVIDIA Jetson TX2, and one desktop computer. Table 6.1 details their relevant technical specifications. All executors are connected to the same LAN over Wi-Fi except the desktop computer, which is connected via Ethernet. This hardware resembles the computing resources that would be available on a medium-sized home network or a small office: for instance, these could represent desktop and

Device	CPU	RAM
Lenovo ThinkStation P320	Intel Core i7-7700	16 GB
Dell XPS 8900	Intel Core i7-6700K	16 GB
NVIDIA Jetson TX2	ARM Cortex-A57	8 GB
NVIDIA Jetson Nano	ARM Cortex-A57	4 GB
Raspberry Pi 4B ($\times 5$)	ARM Cortex-A72	4 GB
Raspberry Pi 3A+ ($\times 2$)	ARM Cortex-A53	4 GB

Table 6.1: Specifications of the devices in our evaluation.

laptop computers, gaming consoles, smart appliances, smart doorbells, smart home assistants, and more. The hardware also offers a likely variety of heterogeneous computing capabilities.

We mimic the behavior of *offloaders* by submitting task requests to the controller via a Python script that allows us to control the experimental conditions, which include the tasks themselves, their inputs, and the rate of their arrival.

6.3.2 Tasks and Setting Task Deadlines

We use six types of tasks to test our scheduler's performance. Each task requires a varying amount of computation, depending on its input data, as we can expect from real-world tasks. And the computation for each task may leverage the unique hardware capabilities of executors to accelerate task execution, a possibility that is essential for the scheduler to recognize.

The task types we use are:

1. **Loop**: this task computes a sum in a for-loop for a variable number of iterations. It takes an initial value as its input and returns the final sum as its output.

2. **Matrix Multiplication**: this task multiplies two square matrices together to compute their product. The size of the input matrices vary. The two matrices are the inputs, and the product is the output.

3. **Fast Fourier Transform (FFT)**: this task computes the FFT of an audio data. The audio data is sampled at 44.14KHz and the input size varies based on the number of seconds sampled.

4. **Human Activity Recognition (HAR)**: this task uses data collected from various wearable sensors to classify activities. This uses an LSTM model trained on the MHEALTH dataset [106]. The input sensor data is sampled at 50Hz and varies according to the number of seconds of data collected.

5. **Object Detection**: this task uses the YOLOv5 vision model [107] to identify objects in a given image. We use the same image at different images sizes to vary the input.

Task Information					Profiled Time on Executors (ms)				
Task Type	**Description**	**Input Size**	**Task Id**	**Deadline (ms)**	**Nano**	**RPi3**	**RPi4**	**TX2**	**Desktop**
Loop	Running a loop 1M times	4B	1	666.20	416.35	605.63	306.40	353.61	186.69
	Running a loop 10M times	4B	10	4,711.81	2,353.61	4,283.46	1,931.60	1,703.08	517.77
Matrix Multiplication	Multiplying 500x500 matrices	2MB	11	1,377.63	2,634.29	3,309.31	1,252.40	1,203.36	464.90
	Multiplying 1000x1000 matrices	8MB	20	8,769.76	12,217.47	30,257.08	7,972.51	6,625.46	1,856.82
FFT	FFT on 3s audio @ 44.1KHz	1.06MB	21	1,306.79	2,484.60	560.20	1,187.99	1,235.00	417.84
	FFT on 30s audio @ 44.1KHz	10.6MB	30	5,316.52	9,850.85	5,814.92	4,833.20	2,839.78	3,724.11
HAR	HAR on 1s data @ 50Hz	920KB	31	882.70	1,521.56	1,338.50	802.45	1,390.03	362.65
	HAR on 10s data @ 50Hz	9.2MB	40	5,558.66	7,913.33	14,474.55	5,879.98	5,053.32	1,990.57
Object Detection	500x500 image	123.47KB	41	7,017.53	7307.04	NA	7533.64	10385.07	3597.15
	5000x5000 image	2.33MB	50	8,278.53	8,671.28	NA	9,288.47	11,316.44	4,089.90
Room Classification	400x400 image	44.67KB	51	3,107.38	716.48	2,824.89	940.83	1,499.69	403.27
	4000x4000 image	2.54MB	60	4,746.43	1,835.27	4,314.94	1,637.64	2,075.76	662.02

Table 6.2: Describes tasks used in our evaluation and their execution times (incl. network round trip time from controller) on the executors. We use 10 tasks per task type by varying input size (only the lightest, tasks and heaviest of these are shown). A green or red color is used to indicate whether an executor can meet a task's deadline or not. Loop and Room Classification have lax deadlines, Matrix Multiplication and FFT have average deadlines, and Object Detection and HAR have strict deadlines.

6. **Room Classification**: this task uses a SqueezeNet DNN image classification model [108] to classify images of rooms. We use the same image at different images sizes to vary the input.

To provide a variety of tasks to the scheduler, we create 10 different versions of each task type based on the input data to the task. For example, for the FFT task type, we have 10 tasks that compute FFT for audio data of different sizes (3s, 6s, ... 30s). Similarly, we have 10 tasks each for the other task types, totaling to 60 tasks in the evaluation.

We have three types of deadlines for these tasks based on how tight the task's deadline is: i) lax: achievable on all 10 executor devices, ii) average: achievable on roughly half of the devices, iii) strict: achievable on a few executors.

To set the deadlines, we profile these tasks on all of our executor devices. We choose the 100th percentile of the profiled average execution time as the deadline for the lax deadlines. We choose the 100th, 50th, and 25th percentile of the profiled average execution times to select the lax, average, and strict deadlines respectively.

We set lax deadlines for the Loop and Room Classification tasks. We set average deadlines for Matrix Multiplication and FFT tasks. We set strict deadlines for HAR and Object Detection tasks.

We list the various task information, input sizes, deadlines, and average profiling times in Table 6.2.

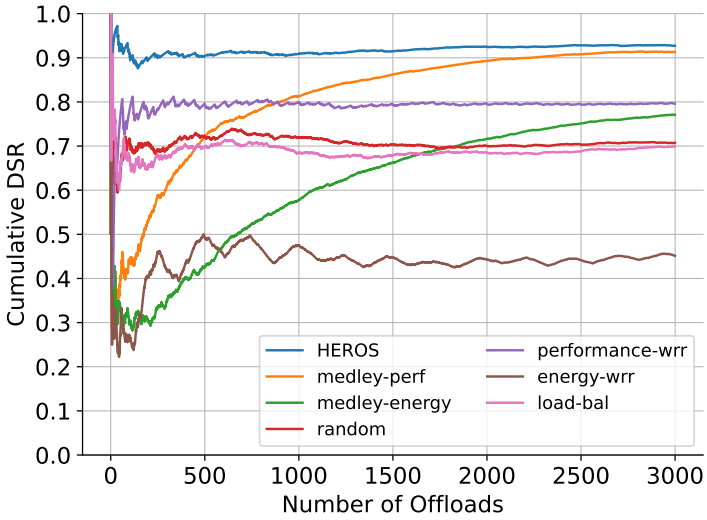


Figure 6.5: Cumulative DSR for the Medley schedulers compared to other common scheduling algorithms and the HEROS [109] scheduler.

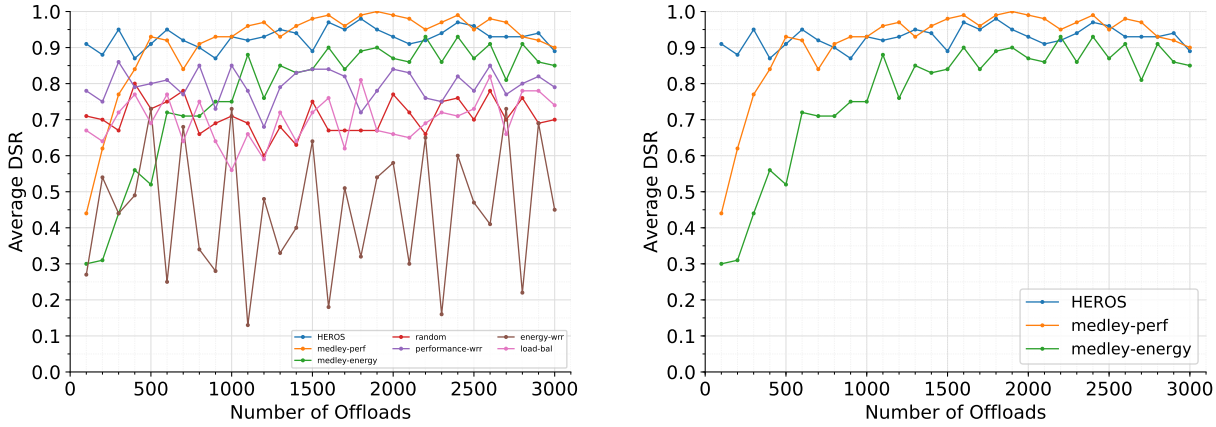
It is worth noting that the profiling information is *not* available for the Medley scheduler.

6.3.3 Energy and Time Measurements

Some of our evaluations consider executors' energy usage. To estimate this usage, we measure the power of all executors while they are idle and while they are executing under the load of a task. We use a plug-in watt meter to observe these values over a period of time and average these observations to obtain average values for both idle and active power. We then take the difference between the average active power and the average idle power to estimate the power of executing a single task. To estimate the energy usage for executing a task, we multiply the executor's task execution power by the length of time it executes the task. Each executor keeps track of the time it takes to execute a task by recording the time it begins executing a task and subtracting that time from the time it finishes executing a task.

6.3.4 Comparing Medley's Task Scheduler to Other Schedulers

We compare the Medley scheduler's performance and energy savings modes to some common scheduling algorithms: weighted round-robin, load balancing, and random choice. The round-robin scheduler statically balances the load and assigns tasks based on either the performance of executors relative to other executors (Performance WRR), or the energy consumption of executors relative to other executors (Energy WRR). The load balancing scheduler assigns tasks to the executor with the lowest 1-minute CPU load value. The random-choice scheduler randomly selects



(a) All schedulers

(b) Top-3 schedulers

Figure 6.6: DSR averaged for every 100 offloads. This helps better gauge instantaneous performance of the scheduler. After around 500 offloads, Medley-perf scheduler performs similarly to HEROS, whereas Medley-energy takes around 1500 offloads to reach a steady state.

an executor to run a task.

For the Performance WRR scheduler, we use Geekbench [110] scores to determine executors' performance relative to each other. For example, the Raspberry Pi 3 has a score of 166, and the Raspberry Pi 4 has a score of 513, so a Raspberry Pi 4 would receive three times the number of tasks a Raspberry Pi 3 would. For the Energy WRR scheduler, to determine the number of offloads each executor would get in a single round in the power-weighted round-robin schedule, we take the difference between the power of each executor when under the load of one task and when idling (as described in Section 6.3.3) and compare them relatively to obtain the weighting.

We also compare the Medley scheduler with the HEROS [109] load-balancing scheduler, which requires *a priori* task execution times on all executors to perform scheduling. Guzek et al. designed the HEROS scheduler to perform energy-efficient load balancing among a set of heterogeneous systems. It maintains an inventory of all systems' capabilities: CPUs and their core counts, the amount of memory a system has, available persistent storage, etc. The scheduler uses the resource information alongside task requirements to allocate tasks to computing nodes. To achieve energy efficiency, it uses a heuristic selection function that uses candidate nodes' current load state and performance-per-watt metrics to calculate selection scores. The scheduler assigns the task to the server with the highest selection score. The selection function favors the most energy efficiency nodes and nodes that are already active with jobs (but not overloaded such that they cannot handle additional work). We use the authors' heuristic parameter recommendations of $\alpha = 110$, $\beta = 0.9$, and $\gamma = 1.2$ which quickly degrades the selection score once a node reaches above 90% of its maximum load.

We offload the same sequence of 3000 randomly selected tasks to executors using each scheduling strategy at a rate of 15 tasks per minute. The cumulative DSR for each of the schedulers are shown in Figure 6.5. The HEROS scheduler performs well with an overall DSR of 92.73%. However, this is expected as it has access to the profiling data of how each of the tasks perform on all the executor devices. In contrast, the Medley schedulers (-perf and -energy) do not have access to the profiling data and learns this on the fly, and the Medley-perf scheduler is able to achieve an overall DSR of 91.33%, even with this learning.

To better gauge the instantaneous performance of the scheduler, we also plot the average DSR every 100 offloads, as shown in Figure 6.6. Figure 6.6a shows this for all schedulers, and Figure 6.6b shows DSR average for the top 3 schedulers (HEROS and the Medley schedulers). We see that the Medley-perf scheduler actually starts performing similarly to HEROS, once it learns from around 500 offloads.

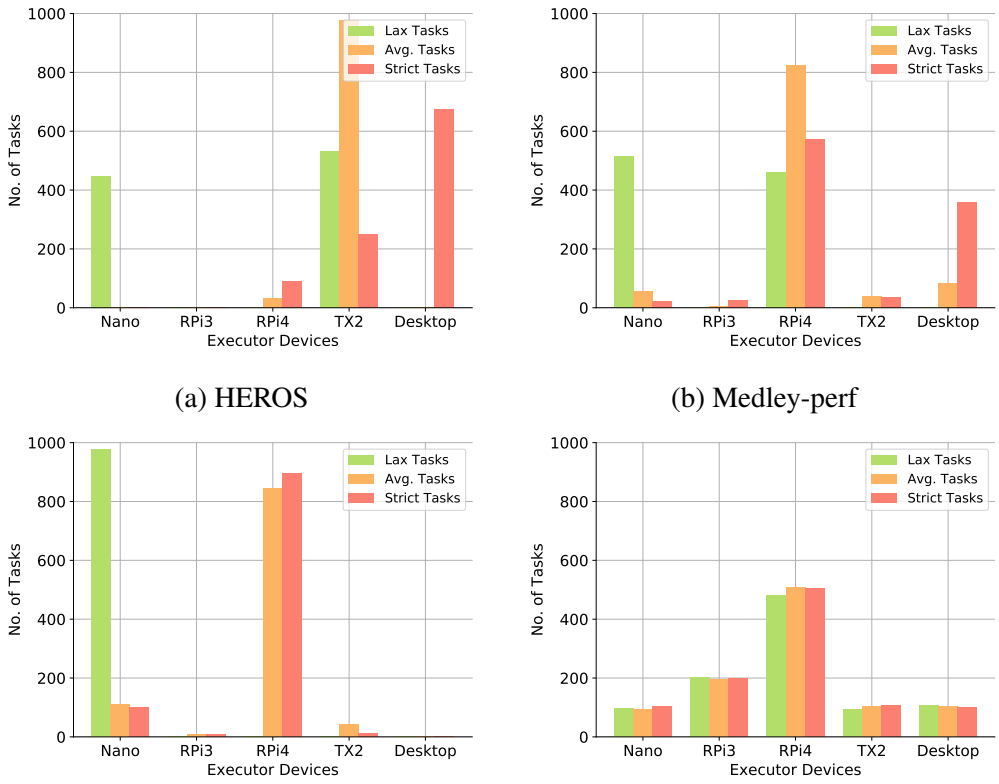
These plots highlight the fact that our task scheduling approach of learning from feedback can outperform other conventional schedulers, and reach similar performances compared to a scheduler like HEROS require a priori task profiling data.

We also study how these schedulers distribute the task types (with three levels of deadlines) on the executor devices, as shown in Figure 6.7 and Figure 6.8.

6.3.5 Comparing Medley's Task Scheduler to Heteroedge

In this subsection, we compare the Medley scheduler with an offline realtime task scheduler, which uses energy and execution time task profiling data on executor devices and tasks to find optimal scheduling decisions. Our implementation of the offline scheduler is based on the Heteroedge scheduler [67], where the executor device energy consumption and delay are modeled as supply chain graphs. Heteroedge models an edge application as a set of concurrent jobs, where each job consists of several sequential tasks, necessitating a separate evaluation in addition to the compared we performed in Section 6.3.4. Additionally, it is interesting to compare against Heteroedge since this work also opportunistically offload edge tasks to other IoT devices nearby.

For our implementation of the Heteroedge scheduler, we make some asssumptions. We assume energy consumption is proportional to the time taken to execute a task as estimated by the Medley scheduler, which is equivalent to considering the base energy cost as zero in the linear energy model used in [67]. We define a set of tasks being executed sequentially as a job. As the scheduling algorithm, we assume an optimization heuristic where the minimal cost (shortest path) is selected from the supply chain graph for each job at a time, then we remove from the graph links representing occupied threads and proceed to search the next new optimal path for the following job. If all links representing threads of an executor device are removed, we restore all thread links of the device and double their energy costs to represent a penalty of waiting for all threads to finish their previous tasks before taking new ones. To compare our scheduler against the Heteroedge scheduler, we consider multiple jobs running concurrently. Each job does 6 tasks sequentially: computes loops 10M times, multiplies two 1500x1500 matrices, performs FFT on 30s audio data,



(a) HEROS

(b) Medley-perf

(c) Medley-energy

(d) Random

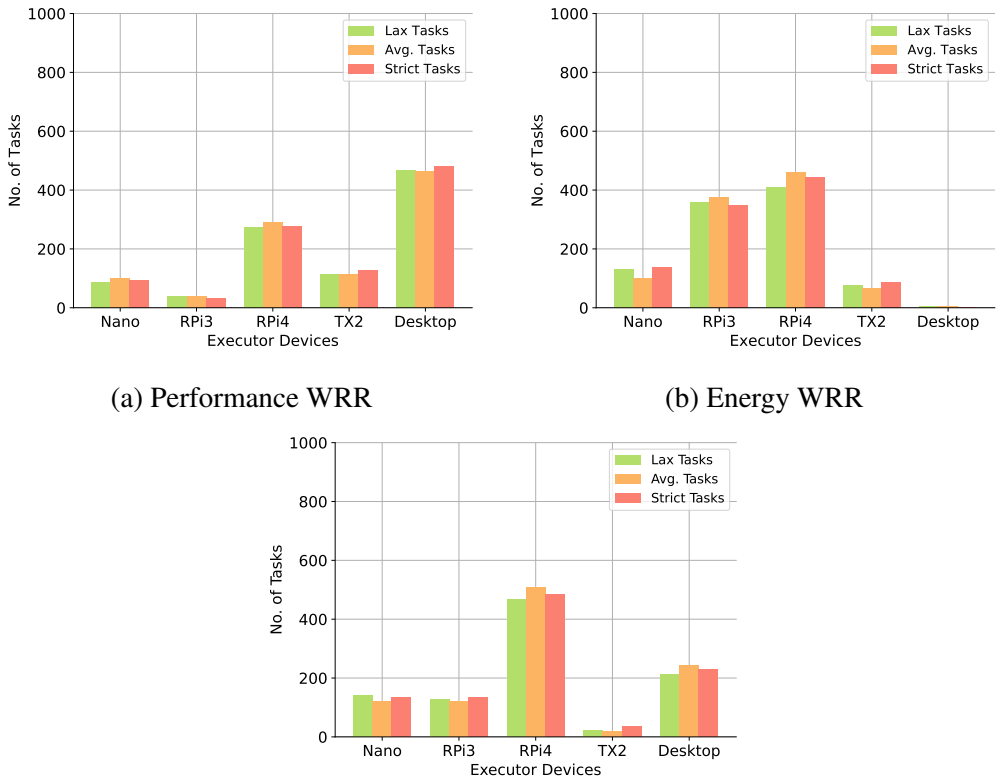
Figure 6.7: Shows how the various schedulers schedule different task types on the executor devices.

performs HAR on 10s sensor data, performs object detection on a 5000x5000 image, and then performs room classification on a 4000x4000 image. These tasks are listed in Table 6.2 with task ids 10, 20, 30, 40, 50, and 60 respectively. We use all 10 of our executor devices for the experiment.

We choose different number of jobs: 5, 10, 15, 20, 25, and 50, with each job comprising 6 tasks, and use the offline Heteroedge scheduler to identify the optimal executor sequence for each of the tasks. We then submit these jobs, spaced every 30s, and compare the DSR for Heteroedge and Medley, and the results are shown in Figure 6.9.

Heteroedge clearly performs better for a small number of jobs. This is because it uses task profiling data to identify suitable executors for each task. In contrast, the Medley scheduler learns this on the fly, and when the number of jobs are higher, Medley-perf can match and even exceed the DSR of Heteroedge.

We study the total energy for executing the jobs in the executor machines, and the total execution times for the jobs, and the results are shown in Figure 6.10. Heteroedge is designed to pick executors to reduce energy consumption on the executing devices as its primary criteria. This



(a) Performance WRR

(b) Energy WRR

(c) Load Balancing

Figure 6.8: Shows how the various schedulers schedule different task types on the executor devices.

is why it consumes significantly less total energy compared to Medley-perf (Figure 6.10a). For instance, it never picks the desktop machine for any of the strict deadline tasks due to the high energy cost to run tasks on the desktop. However, the Medley-perf scheduler and particularly the Medley-energy scheduler do not exclude the desktop and as an option if it can meet the deadline for a task.

Our scheduler implementations attempt to save energy when possible, but prioritize meeting deadlines, which might be better suited for an edge computing ecosystem like Medley which support realtime tasks.

6.3.6 Comparing Performance of Our Ecosystem to a Single Server Machine

In this evaluation, we answer the question "Is it viable to use an ecosystem of IoT devices like Medley to replace an edge server machine?". To do so, we assume that the desktop in our evaluation setup as a server machine, and evaluate how well the ecosystem with the rest of the machines excluding the server performs.

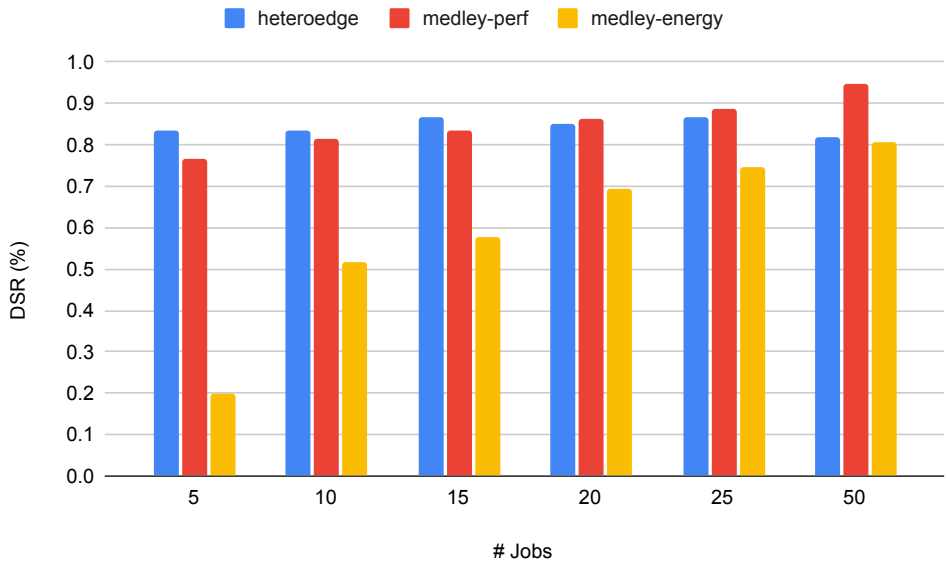


Figure 6.9: Compares the Heteroedge scheduler with the Medley schedulers. Shows the Overall DSR for different number of jobs, which are sequence of tasks executed in order.

To provide some context on how the desktop performs compared to the other executors, we visualize their average computation time for all six of the tasks in our evaluation. The results are shown in Figure 6.11 and Figure 6.12. For all tasks, the Desktop is able to easily meet the deadlines and outperform other devices. This indicates the challenge in replacing an edge server machine.

We first look at how a single executor device, a singular Raspberry Pi 4 performs for 3000 task offloads, as shown in Figure 6.13a. The overall DSR reaches 26.73%.

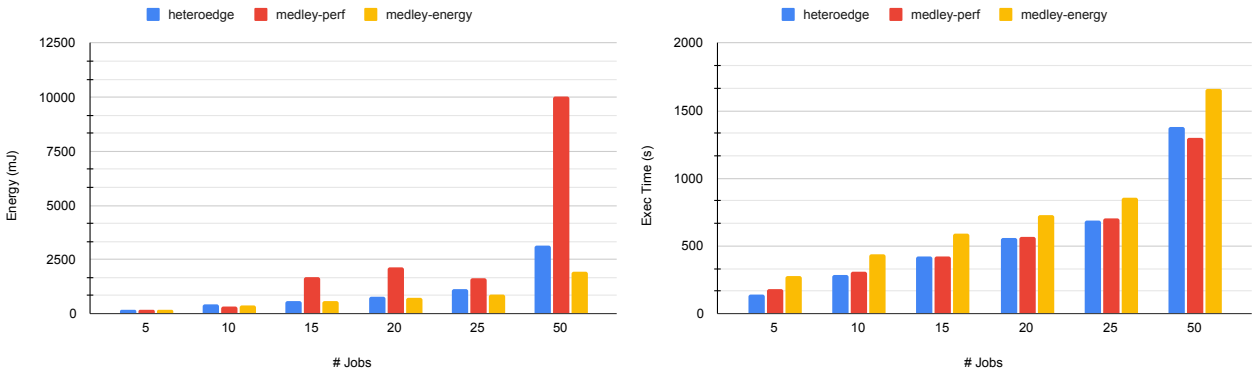
We then add some homogeneous compute to this singular RPi4, i.e., we add four more RPi4s (Figure 6.13b), and use the Medley-perf scheduler to achieve an improved DSR of 60.93%, an improvement of 2.3x.

We make the Medley ecosystem more heterogeneous by adding the remaining executors, the Jetson Nano, Jetson TX2, and two Raspberry Pi 3A+ (Figure 6.13c), and adding more heterogeneous compute resources to the mix further improves the DSR to 78.43%.

We then offload the same set of tasks to the server machine and compare the results of Medley with the server as shown in Figure 6.14a. The server doesn't miss a single deadline, and has a DSR of 100%, compared to the server, the Medley ecosystem misses 21.57% of the deadlines.

We identify that for the tasks with strict deadlines (HAR and Object Detection), the deadlines are in fact impossible to achieve without the server. This is because the computation time and network round trip time to execute these tasks easily exceed the deadline (refer Figure 6.11, Figure 6.12 for just the task computation times, and Table 6.2 for the computation and network round trip times), whereas this is not the case for the server.

Based on this, we remove HAR and Object Detection tasks from our previous experiment and



(a) Total Energy

(b) Total Execution Time

Figure 6.10: Show the (a) total energy consumed by executors, and (b) total execution time for tasks, for different numbers of jobs for Heteroedge and Medley schedulers.

	DSR	**Total Energy**	**Total Execution Time**
Performance Mode	91.33%	75.79J	8,942.9s
Energy Savings Mode	77.06%	14.46J	11,502.9s

Table 6.3: Measurements for the two operations modes of the Medley scheduler after 3000 task offloads.

then plot the results, as shown in Figure 6.14b. This leads to a DSR of 94.89% for Medley. We also note that if we had only removed the Object Detection tasks and included the HAR tasks, the DSR would still be at a respectable 90.65%.

We learn two key things from this evaluation. First, we acknowledge that tasks with tight QoS requirements exist in the real world, and thus excluding those tasks is not the right solution. On one hand, the progression in performance when shifting from a single Raspberry Pi 4 to a heterogeneous ecosystem of IoT devices certainly highlight the viability of an ecosystem like Medley to tackle edge computing for a good set of IoT applications. But on the other hand, it does highlight that there are real world limits on our "sum of parts" approach, but it can be mitigated by carefully choosing the tasks to be run based on the ecosystem's capabilities.

6.3.7 Different Modes of Operation for the Medley Task Scheduler

The Medley scheduler can operate in either performance mode or energy savings mode. Both modes try to maximize DSR. Additionally, the performance mode tries to reduce execution time, and the energy mode tries to reduce energy consumption of the executors. The Medley-perf scheduler is aggressive in its approach to find the best executor for a given task. If it misses the deadline for a particular task on an executor, it will never pick that executor for that task again. In com-

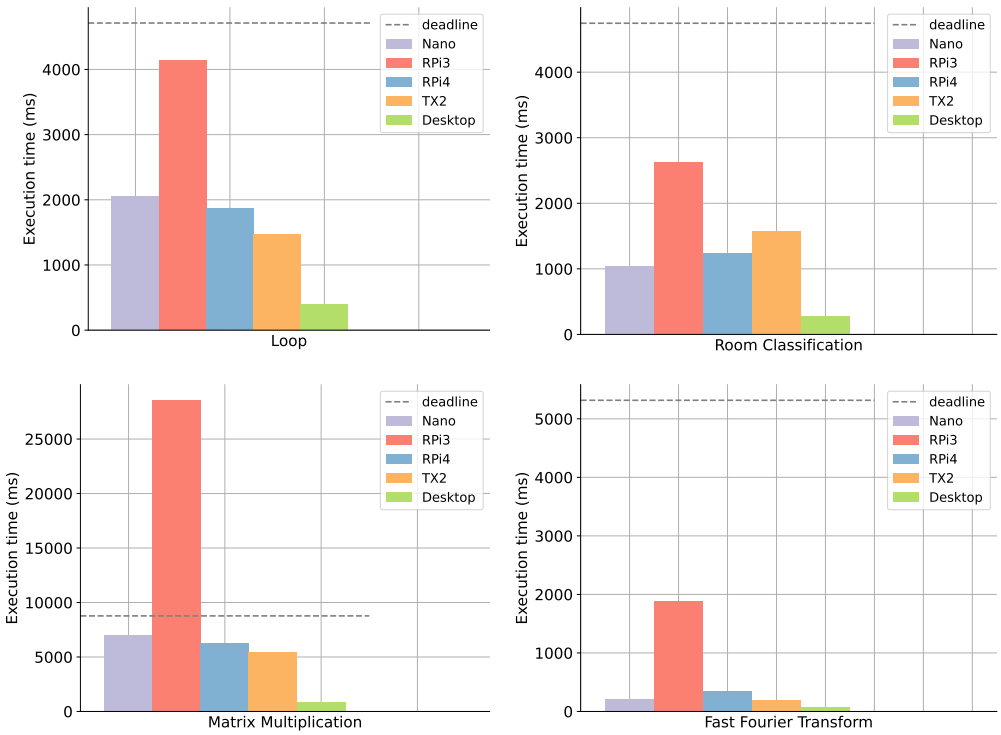


Figure 6.11: Average Computation Times for each of four of the six evaluation tasks on different executor devices. The deadline for the task is shown as a dotted line. Times doesn't include network round trip.

parison, Medley-energy gives each executor 5 deadline misses before switching to a more energy consuming yet computationally better device.

We showcase how these modes differ in their operation by showing the DSR, total energy, and total execution time in Table 6.3. Depending on the mode, either execution time or energy consumption can be traded off to reduce the other. It is also interesting to see how the scheduler distributes tasks to the different executors in each mode, as shown in Figure 6.7. Notice how the desktop machine is completely avoided by Medley-energy, while switching to other executors like the Raspberry Pi 4 and Jetson Nano.

These operation modes provide users with an option to optimize whether they want to get the best performance for their offloaded tasks or to reduce their home or office energy consumption.

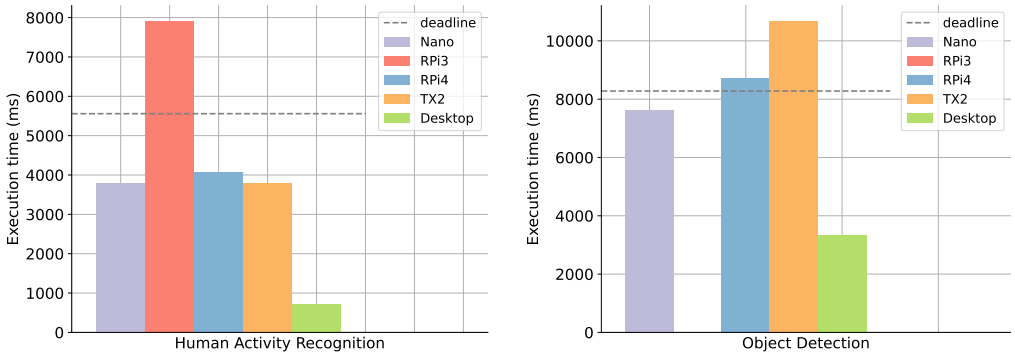
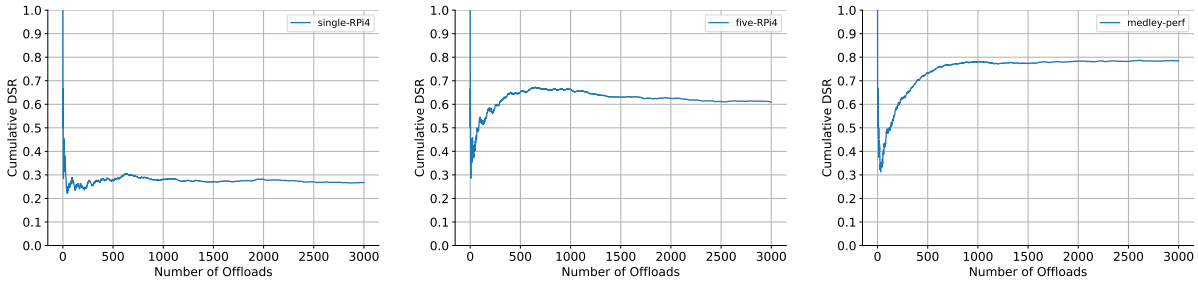
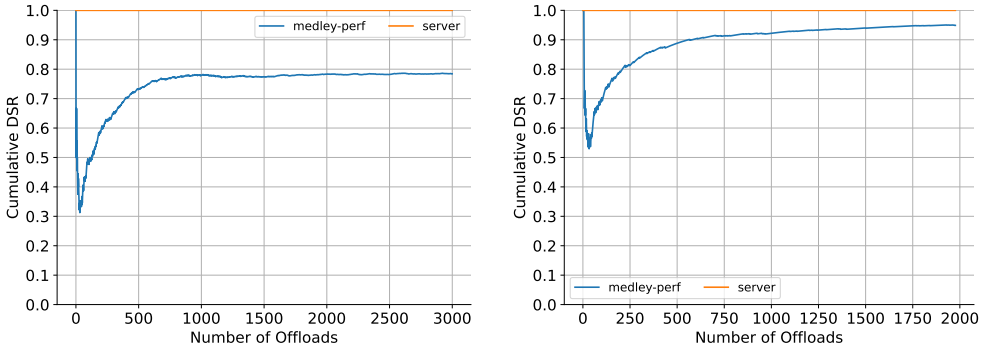


Figure 6.12: Average Computation Times for two of the six evaluation tasks on different executor devices. The deadline for the task is shown as a dotted line. Times doesn't include network round trip.



(a) Single RPi4 (b) Five RPi4s (c) All except Desktop

Figure 6.13: Shows cumulative DSR for 3000 task offloads for different combinations of executor devices: (a) A singular device, (b) A homogeneous ecosystem, and (c) A heterogeneous ecosystem.



(a) All Tasks

(b) No tasks with strict deadlines

Figure 6.14: Shows how the Medley ecosystem without a desktop machine compares to a server machine. Shows cumulative DSR for 3000 task offloads for (a) all six tasks, and (b) all tasks except HAR and Object Detection.

CHAPTER 7
PRIVACY POLICIES TO EMPOWER USERS WITH ACCESS CONTROL OF THEIR IOT DATA

The distributed middleware described in Chapter 5 and the Medley task scheduler described in Chapter 6 enables a sizable set of edge IoT applications. Supporting more application use cases at the edge in turn leads to an increase in the consumption of IoT data by edge computing applications. Although this aligns with the vision of IoT to provide users with better services to improve their everyday life, we have to be cognizant of the privacy of users as well. This is especially relevant in shared spaces like smart buildings, where users have little to no knowledge of what data is being collected and how it is being used [42]. It is essential that we provide users with better control over their sensed data.

One approach to better privacy controls is to first send all IoT data from a deployment to a cloud data store, and enforce access control on the collected data based on user privacy preferences. This is how typical cloud-based edge computing solutions employ, but, this is not a scalable solution since edge computing applications require real-time processing of data and thus relying on the cloud is infeasible. Also, rather than collecting data and filtering retroactively, it is a better privacy practice to not collect it in the first place if users do not prefer so. As we propose shifting the control plane of edge computing from the cloud, we envision a platform where data flow from IoT devices to edge applications is controlled by end users based on their privacy preferences.

In this chapter, we first compare the user privacy among different domains to underline why privacy in shared IoT spaces is important. We then describe the design of our policy enforcement system. We provide more detail on how we implement our privacy policies and do a qualitative comparison of our implementation with the Matter protocol for IoT [21].

7.1 User Privacy in Different Domains

We compare privacy models in smartphone apps, social media apps, smart home apps (assuming IoT devices are self deployed), and in shared IoT spaces (where device deployment and data collection is not handled by the end users using the space). Our observations are outlined in Table 7.1. Unlike other domains, users in shared IoT spaces have no knowledge about what is being sensed or which applications are using their data (limited transparency). Also, users are not provided with any mechanisms to control how their data is being used.

Domain	Transparency	Data Usage	Control
Smartphone Apps	Knows what data is collected from requested permissions	- App data isolated by OS - Supports data sharing with user request	OS provides permissions model for users to control app data access
Social Media Apps	Knows what data is collected from requested permissions	- App data fully isolated - Data sharing allowed based on user permissions	Users can revoke permissions or remove apps from using their data
Smart Home Apps	Knows what data is being sensed to some extent since installed by self	- Knows which smartphone apps use data since installed by self - App data fully isolated - Data sharing allowed using user enabled APIs	- Can disable devices or control device data collection schedules - Can revoke APIs to stop other apps from using data
Shared IoT Spaces	No knowledge of what is being sensed	- No knowledge of which apps use their data - Data sharing with other cloud or edge apps possible	No mechanism to control data usage

Table 7.1: Comparison of user privacy in different domains.

7.2 Design for our User-Driven Privacy Policy

To provide better transparency and control for users in shared IoT spaces, we propose collecting a privacy policy, which is a set of preferences from users. A preference indicates how a device's data stream is used by an application. These preferences can be very elaborate; for example, a user can allow access to their occupancy data to applications only during their working hours, they can opt to share occupancy data to an emergency evacuation application, but not to a room scheduling application and so on. We design a platform where data flow from IoT devices to edge applications can be controlled by end users based on such privacy preferences.

Building such a data stream based access control mechanism would be a natural extension of the existing NexusEdge middleware. Our key insight is that NexusEdge gateways already act as routers that route sensor data from IoT devices to edge applications, and thus a privacy policy can be enforced on the gateways, which by allowing or denying data streams to flow to edge applications based on preferences specified in the policy. This can be viewed as similar to how firewall rules operate on network traffic in routers.

One key requirement that we had for our design was to have a machine-readable privacy policy format that is expressive enough. It should work for different combinations of sensors and apps, as well as at different time granularities. Users should be able to specify preferences for single, multiple, or all sensors, and similarly for apps. They should also be able to specify preferences for days of week, month, day of month, hour, minute, and second. For instance, a user should be able to block a specific occupancy sensor from all applications between 9AM to 5PM on Sundays.

We make some assumptions for our design. First, preferences in existing systems are collected in different ways from the users; this could be a user initiated process, or they could be notified

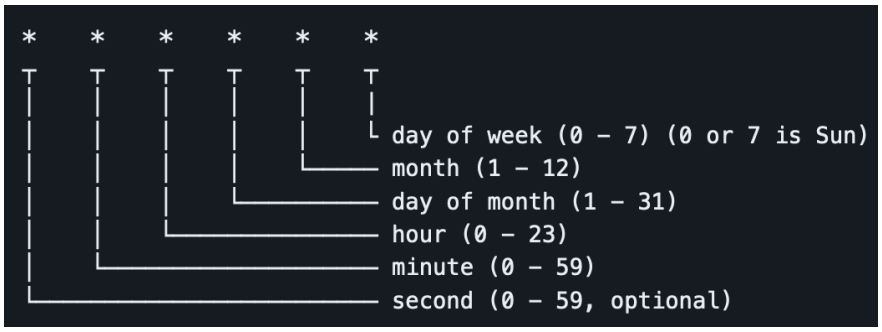


Figure 7.1: Time Interval in a NexusEdge privacy policy preference is represented similar to crontab time specifiers used in the Unix utility cron.

when a new app requests for data collection. We assume a user-initiated process where users can provide the preferences at any time. Second, it is important to know which IoT devices are associated to a user. In shared and open spaces, this might be difficult because there are multiple people within sensing range of the same device. We assume that this mapping is known beforehand, and that each device is exactly associated to a single user, for sake of simplicity.

7.2.1 Privacy Policy Format

A privacy policy consists of multiple preferences. Each preference is represented as a tuple (Sensors, Apps, Time Interval, Block/Allow). Sensors is a list of sensors (e.g.: [s1, s2]). Similarly, Apps is a list of applications (e.g.: [a1, a2, a3]). We use the wildcard character * to specify "all". Time Interval is a string which represents the time associated with the preference. Block/Allow is a flag value indicating whether to block or allow the sensor stream during this time interval. All sensor traffic is allowed by default.

The Time Interval is represented as a string, similar to how lines in a crontab file are written for the Unix system utility, cron [111]. The Time Interval representation is shown in Figure 7.1. For example, a policy "Block data from s1 and s2 to all applications between 9AM to 5PM on Sundays" is represented as: ([s1,s2], *, * * 05-09 * * 0, block).

7.2.2 Privacy Policy Enforcement

We design our system such that users can provide preferences and this is added to the current privacy policy and disseminated to all gateways. The NexusEdge middleware on the gateways then coordinate to enforce the policy by filtering out sensor streams based on the preferences in the policy.

We show an example of such a policy enforcement in Figure 7.2. The App shown in the figure requires data from temperature sensors T1 (connected to Gateway 1) and T2 (connected to Gateway

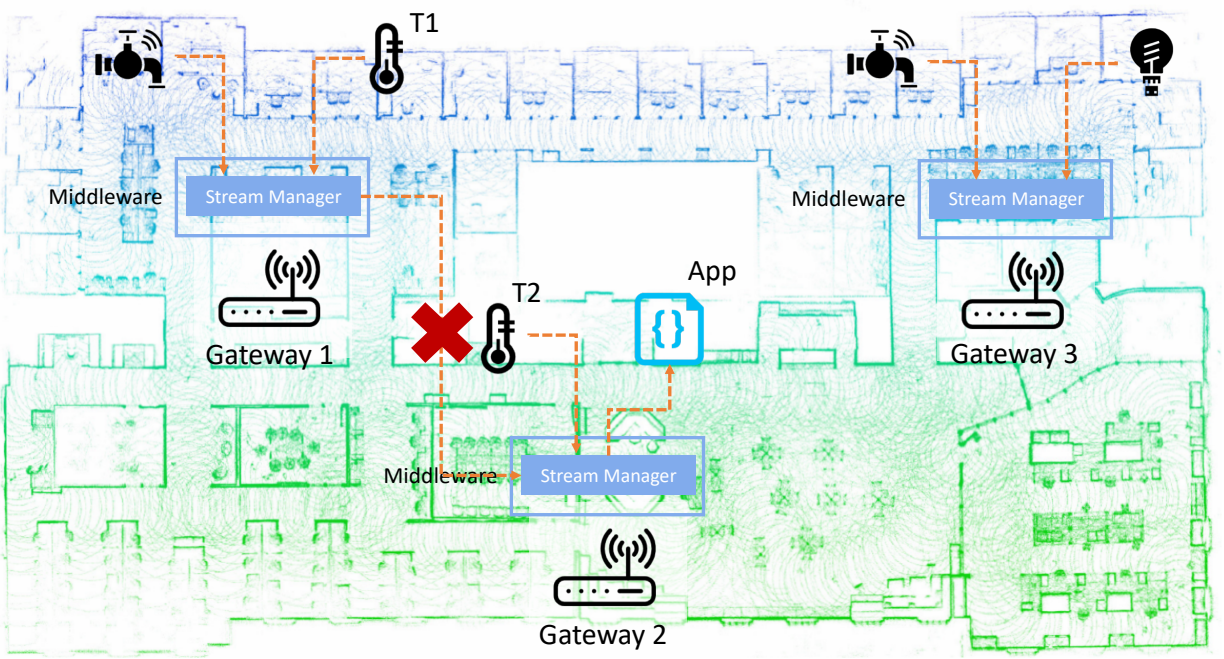


Figure 7.2: Example of privacy policy enforcement. If there is a preference to not share the data of T1 to App, the data stream is not routed to App.

2). If there is a privacy preference to not share the data of T1 to App, the data stream from T1 is not routed to App. This policy is represented as: ([T1], [App], * * * * * *, block).

7.3 Implementation

In this section, we discuss two aspects of our implementation, namely how the policy preferences are collected from users, and how the device streams are filtered from applications.

7.3.1 Policy Preference Collection

To collect the policy preferences from the user, we use a web application which shows a user form allowing users to create a new preference. A screenshot of this application is shown in Figure 7.3. This eases the burden on the users to not specify the policy preference as a tuple per our design. The tuple is generated from the web application.

7.3.2 Policy Enforcement Implementation

Privacy policy preferences are collected and routed to the Sensor Stream Manager (SSM) service on the NexusEdge gateway (refer Section 5.2.5, for the different NexusEdge services and how they

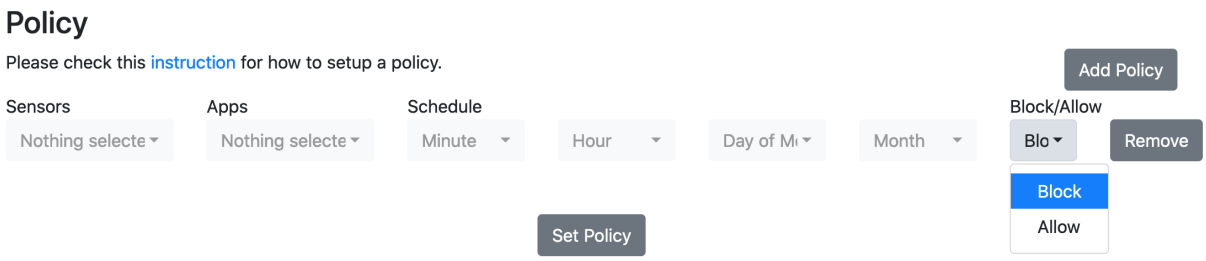


Figure 7.3: Users of NexusEdge use a web application to create privacy policy preferences. This preference is collected and converted to a tuple (Sensors, Apps, Time Interval, Block/Allow) representation internally.

interact with each other). Whenever a new application is deployed on NexusEdge, it runs on a particular gateway and has a dedicated MQTT topic on that gateway to receive the data streams it had requested for at deployment time. Gateways coordinate to make sure an application receives these requested data streams. The SSM on each gateway maintains a routing table of which device data streams it needs to forward to various application MQTT topics, both locally and on other gateways.

We implement a new module within the Sensor Stream Manager, the Privacy Enforcer, which stores the received privacy policy and enforces it. Whenever the SSM has to forward a data stream from it Core Service MQTT topic (which receives all device data) to an application's MQTT topic (locally or remotely), it first checks with the Policy Enforcer if the application is allowed to receive that stream or not. Based on this, the SSM decides to either forward or not forward the data stream.

An example of this policy enforcement is shown in Figure 7.4. In the given example, two NexusEdge gateways, Gateway 1 (G1) and Gateway 2 (G2), receive data respectively from temperature sensors T1 and T2. An application App1 runs on G2, and at deployment time had requested data streams from T1 and T2. At App1's deployment time, SSM on G2 had requested SSM on G1 to forward data streams from T1 to App1's MQTT topic on G2. When a user sets up a new privacy policy preference, "Block T1 from App1", SSMs on both gateways receive this preference. When SSM is forwarding the data streams, the Privacy Enforcer module intercepts these data streams and enforces this preference. G1's stream to App1 is blocked, whereas G2's stream to App1 is allowed.

7.4 Comparison with Matter Protocol

To evaluate our work, we compare it qualitatively to an emerging unifying standard for smart home IoT devices, the Matter protocol. This comparison is relevant because similar to the overall objective of our work, which is to create an edge computing infrastructure which supports IoT applications, Matter also aims to improve interoperability between currently siloed IoT ecosystems

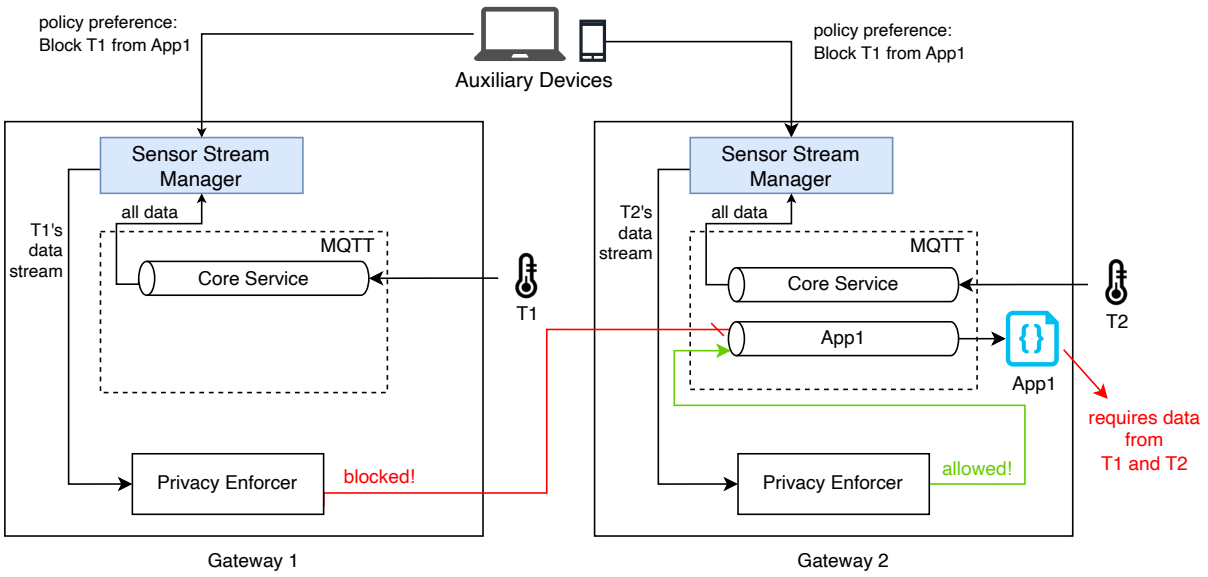


Figure 7.4: Gateways collaborate to enforce collected user privacy preferences. In the example, Gateway 1 (G1) and Gateway 2 (G2), each receive data respectively from temperature sensors T1 and T2. App1 runs on G2, and receives data streams from T1 and T2. At App1's deployment time, G2 had requested G1 to forward data streams from T1 to App1's MQTT topic on G2. When a gateway receives a preference to "Block T1 from App1", the Privacy Enforcer module intercepts data streams from the Sensor Stream Manager to both local and remote MQTT topic to enforces this preference. G1's stream to App1 is blocked, whereas G2's stream to App1 is allowed.

for supporting useful applications at the edge. We study how Matter handles privacy controls and how they compare with our implementation.

7.4.1 Background on Matter Protocol

Matter is a unified, open-source application-layer connectivity standard built to enable developers and device manufacturers to connect and build reliable, and secure ecosystems and increase compatibility among connected home devices [112]. It was born out of the need to improve interoperability among IoT devices in smart homes, which currently are siloed within their own ecosystems consisting of the device, a dedicated gateway device, and the cloud. Applications which require any interaction across smart home devices are required to run on the cloud with the interactions handled by cloud APIs, leading to lower latency applications.

Matter is an application layer protocol which tries to shift applications to the edge by supporting interoperability among devices, using Internet Protocol (IP) and is compatible with Thread and Wi-Fi network transports. As per the specification of Matter, the protocol defines the application layer that will be deployed on devices as well as the different link layers to help maintain inter-

operability [113]. Matter was developed by a Working Group within the Connectivity Standards Alliance (Alliance).

Most of the examples and definitions in this background section have been borrowed from element14's starter's guide to Matter [114].

7.4.1.1 Matter Controller

A Matter controller is a device used to add and control the Matter devices to a network, pairing it with BLE and communicating it with IPv6. The controller, along with a Thread border router running on the same device, be a central hub for the ecosystem using the Matter protocol. The Matter controller can run on a standalone device like a smart home hub, or on a smartphone or tablet.

7.4.1.2 Matter Data Model

Once a device is connected to a Matter network after approval from the controller, it can communicate to other devices using the Matter data model, which is a standard language defined by the Matter spec [113]. This data model is crucial to interoperate with devices from different manufacturers.

It includes a set of common commands and data formats that devices might need. For example, a light bulb that supports the Matter protocol might use the language to send commands to turn on or off, adjust brightness, or change colors. A thermostat might use the language to report temperature readings, set heating and cooling schedules, or adjust temperature settings.

7.4.1.3 Matter Interaction Model

Interactions between Matter devices are based on the Interaction Model layer defined in the Matter spec. There are four types of interactions defined in this model:

1. **Read**: used to retrieve the values of attributes or events. For example, a Matter controller might read the LockType attribute of a door lock device to display the correct icon to the user.

2. **Write**: used to modify attribute values. In the door lock example, a Matter controller might update the OperatingMode attribute of the device to put the lock in privacy mode.

3. **Invoke**: used to send commands. With a door lock device, a Matter controller can invoke the UnlockDoor command to unlock the door.

4. **Subscribe**: allows a device to receive data from a target device periodically, rather than having to poll for the data each time. In the door lock example, a Matter controller can subscribe to the LockState attribute of the device in order to be notified when the door is unlocked by another user.

7.4.1.4 Matter Applications

There are two ways in which Matter supports applications:

1. **Device-Device Interaction**: In this model, the application runs on one device, and it interacts with another device. For example, a smart light switch can be configured to turn on/off a specific light bulb [115, 116].

2. **App-Device Interactions**: In this model, the application is executed externally (i.e., not on any of the devices) on a smartphone or other dedicated device, and interacts with devices. For example, a Matter phone app controlling a light bulb [117].

This distinction is important when we compare Matter's privacy controls to our work in this dissertation.

7.4.2 Privacy Controls in Matter

Matter approaches handles device security and privacy using access control lists (ACLs) [113, 118]. Every interaction operation in Matter must be verified by the Access Control mechanism. Whenever a client device and a server device use any of the interaction model operations to interact with one another by reading (or subscribing) attributes or events, writing attributes, or invoking commands, the Access Control mechanism must verify that the client has sufficient privileges to perform the operation on the server device. The operation is blocked if no sufficient privilege is obtained.

Each device has an ACL with these key fields: Privilege, Subjects, and Targets. Privilege can be one of View, Operate, Manage, or Administer. Subjects could be a node's id or a group's id. Targets specify the scope of the privilege.

We provide some examples of how these ACLs are used to enforce access controls.

- "Allow an Application to administer the device".
  ACL: {"fabricIndex": 1, "privilege": Administer, "subjects": [112233], "targets": null}.
  fabricIndex is the index of the current device ecosystem. Administer privilege allows managing privileges, and can observe and modify the Access Control. 112233 is the ID of the Application. The target: null means the access control is widely scoped.

- "Allow a light switch device to operate the On/Off and Light Level controls of a light bulb".
  ACL: {"fabricIndex": 1, "privilege": Operate, "subjects": [2], "targets": [{"cluster": 6, "endpoint": 1, "deviceType": null}, {"cluster": 8, "endpoint": 1, "deviceType": null}]}
  This ACL would be added to the light bulb. 2 is the ID of the light switch which is given permission to operate the specific controls. Matter uses *clusters* in its data format to separate operations. For example, cluster no. 6 is the On/Off cluster, and cluster no. 8 is the Level Control cluster.

Although Matter doesn't have an explicit privacy control, administrative users can specify ACL rules on devices to allow specific applications or devices (or groups of applications or devices) to collect data from, or to operate them.

7.4.3 Comparison with our Privacy Policy Implementation

There are some similarities with how Matter and our work enforce privacy policies. In our work, when deploying an application, we require the application developer to state which sensors the application needs to subscribe to. Similar to this, when setting up an application, the developer on Matter needs to create a device binding for each device the application needs access to. Additionally, you also create ACL rules for each device stating which other devices/applications have access to the device's data. In both implementations, a privacy policy can be updated after application deployment time. For example, a new time based preference can be collected from the user in our implementation, whereas a new ACL rule could be added by an administrator in Matter.

There are two key difference we identify between the platforms. Matter takes a device-centric approach, whereas we take an application-centric approach for enforcing access control. This is because by design we don't support device to device interactions like Matter (Section 7.4.1.4). We assume that not all IoT devices, for example, energy-harvesting devices, have the capability to support such device interaction code.

Another difference is that we provide a more higher semantic layer for privacy preferences, compared to ACL lists. We allow users to write time-based preferences, enabling better expressiveness for the privacy policy. For example, the preference "block data from my occupancy sensor to a meeting app (which shows if I'm available or not for a meeting), outside my office hours of 9AM–11AM on Fridays", is a very reasonable and useful preference for an IoT user. This is easy to express in our implementation, but the timing aspect is hard to enforce with Matter's ACLs. One potential solution for Matter to implement this is to run an administrator app on the controller, which can take such time-based preferences and constantly update ACLs on the devices based on current time to enforce the policy.

We also study if Matter's ACL rules can be implemented in our privacy policy implementation. We check this for the two types of applications that can be created on Matter (Section 7.4.1.4). One advantage with Matter's ACL is that it allows specific clusters on a device to be accessed by other nodes. We don't support this on our platform, and our access granularity is either all data from the device or no data from the device. For example, the ACL {"fabricIndex": 1, "privilege": View, "subjects": [112233], "targets": null}, allows an app (with id 112233) to view data from the device. This can be represented as: ([s1], [112233], * * * * * *, allow) with our implementation. However, more granular ACLs like {"fabricIndex": 1, "privilege": Operate, "subjects": [2], "targets": [{"cluster": 6, "endpoint": 1, "deviceType": null}, {"cluster": 8, "endpoint": 1, "deviceType": null}]} is not possible to implemented.

From this qualitative evaluation, we find that our policy implementation provides better expressiveness to specify time-based preferences, which is useful for end users in spaces with inter-

operating IoT devices. However, our implementation could be further improved by adopting finer access control of elements of the data model as used by Matter.

CHAPTER 8
CASE STUDY: ROOM CLASSIFICATION EDGE APPLICATION ON THE TEMI ROBOT

In this chapter, we describe a case study we built to demonstrate the usefulness of our platform.

There are cheap robots available with LIDAR sensors which can move around and generate maps of the space using Simultaneous Localization and Mapping (SLAM). During the mapping process, the robot can figure out where walls and other objects in space are. However, it doesn't know what kind of room it has mapped out, for example, whether it is a kitchen or a living room. This information is useful to enable context-based applications. For instance, for a cleaning robot, once a user tags locations or zones, they can ask the user to clean a specific room. The information about rooms are typically supplied manually by the user [119].

For our case study, we use the Medley ecosystem to provide edge intelligence to aid with this problem. We use the robot's camera to click a picture of the room, offload the image to the Medley ecosystem to perform room classification.

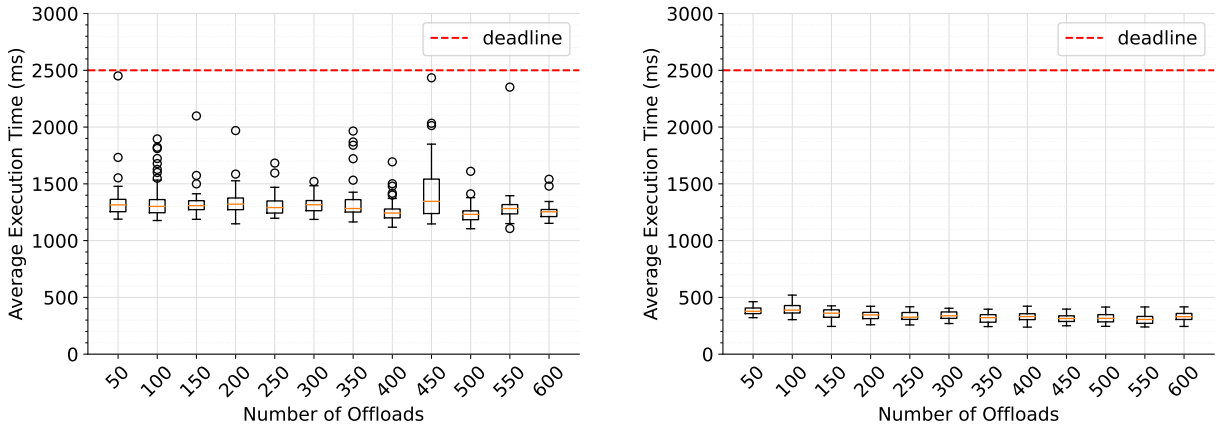
There are two key advantages in using the Medley-powered edge computing for the robot. First, although the robot could potentially run the room classification model locally, there is high value in not doing so, to minimize energy and prolonging the time the robot is in action (i.e., not in its charging station). Utilizing the nearby Medley edge ecosystem enables the robot to do so. Second, the tasks that a robot tend to run usually repeats, and repeating tasks are the ones that Medley ecosystem excels at handling.

This case study showcases a use case in which a robot (or an IoT device) with relatively old hardware or software can benefit from using edge intelligence to run more cutting-edge tasks. This prolongs the lifetime of the robot (or an IoT device) in a user's home without having to replace them frequently, provided there is ambient compute available.

8.1 Implementation

We use the robotic platform, Temi v2 [120] for our implementation. The Temi robot costs around $2,000, uses a LIDAR sensor to map out space, and has a wide angle and a regular camera that it can use to click pictures. The Temi platform allows Android application to be executed on it.

We implement an Android application which runs on Temi, which can click a picture, and make an HTTP request to the controller machine of a Medley ecosystem implementation. We run a SqueezeNet DNN image classification model [108] on the Medley executor machines to classify images of rooms. The model can classify rooms into the following categories: bathroom, bedroom, dining room, exterior, interior, kitchen, and living room. The Android application subscribes to the MQTT broker running on the Medley controller machine to receive responses.



(a) Medley Ecosystem without Desktop

(b) Server Machine

Figure 8.1: Shows the box-and-whisker for the execution time every 50 offloads for a total of 600 offloads of a room classification task. The deadline for the task is set to 2500ms.

Another reason for choosing this platform is its limited OS support, further necessitating task offloading. Temi v2, although being only a few years old, only supports the Android version 6.0.1 and SDK level 23 [121], which was released in 2015 [122]. This highlights the necessity of having a platform like Medley to offload tasks to, which makes the robot support cutting-edge software and continue using the same hardware without being obsolete.

8.2 Evaluating the Case Study Application

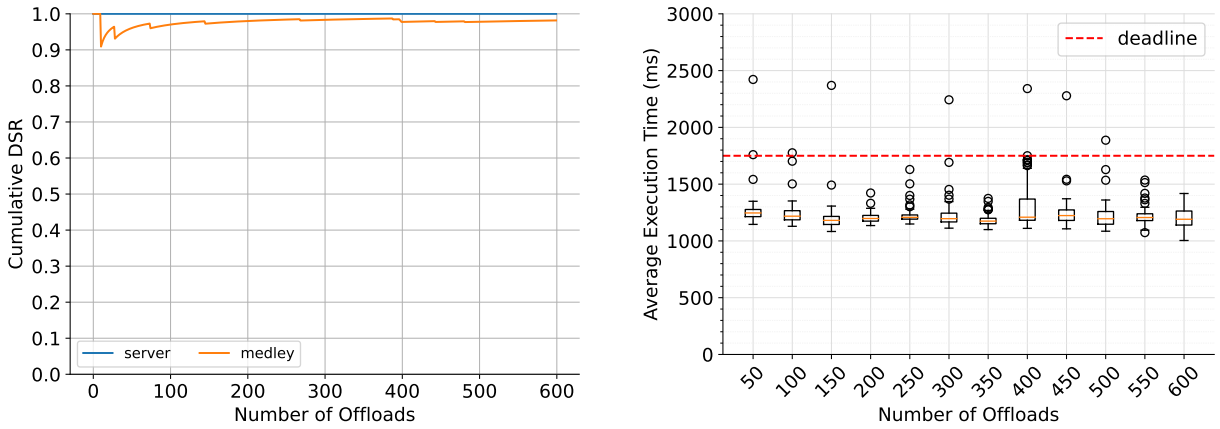
We perform two evaluations to evaluate our case study application. We compare how an Medley ecosystem without the desktop machine can support the application's QoS, compared to just a single server machine. We then make the application's QoS requirement stricter, and see how the ecosystem adapts to this change.

8.2.1 Comparing Application Performance on a Medley Ecosystem and on a Server

We deploy the room classification application multiple times to evaluate how the Medley ecosystem without the desktop machine can support the application's QoS, compared to a single server machine. We use the camera on the Temi robot to click a picture and offload it to perform the room classification task. The images captured by the Temi robot are of size 1600×1200, and we set a deadline of 2500ms, which is reasonable for a room classification task. We repeat the image capture and offload 600 times and measure the cumulative DSR, the execution time, and total energy consumed.

	DSR	Total Energy	Total Execution Time
Medley Ecosystem	100.00%	516.73 mJ	790.98 s
Server Machine	100.00%	6,496.06 mJ	202.87 s

Table 8.1: Shows the DSR, Total Energy, and Total Execution Time for 600 offloads from the Temi robot to the Medley ecosystem and to the server machine.



(a) Cumulative DSR

(b) Box-and-whisker plot for execution time

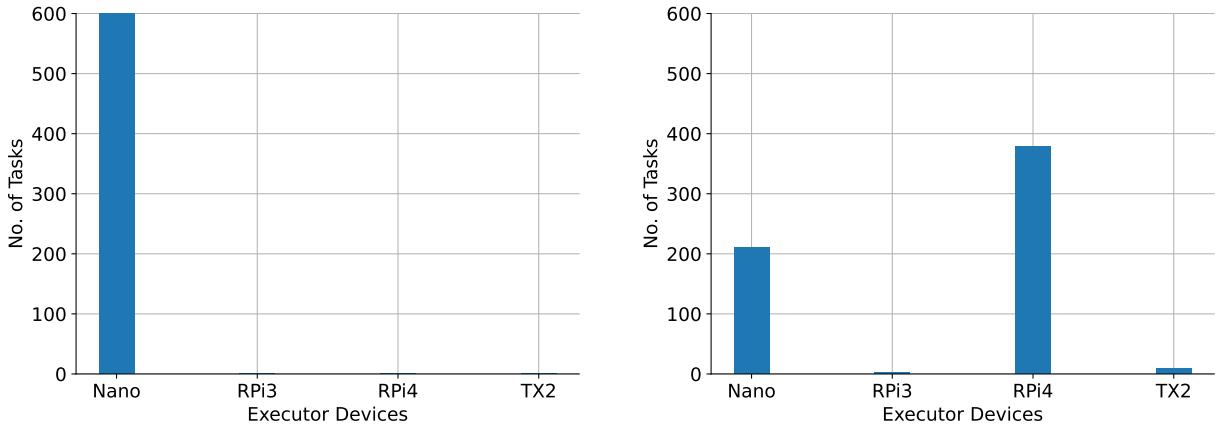
Figure 8.2: Shows the (a) cumulative DSR, and (b) box-and-whisker for the execution time every 50 offloads, after tightening the task deadline from 2500ms to 1750ms. Measurements are for a total of 600 offloads of a room classification task.

The Medley ecosystem is able to match the server machine in terms of QoS provided for the app, by meeting all of the 600 deadlines, and both obtain a DSR of 100%.

We further examine how the Medley ecosystem and the server performed for the offloads, by plotting a box-and-whisker plot for every 50 offloads, as shown in Figure 8.1. We find that the server machine is able to meet the deadline comfortably, meanwhile the median execution time on Medley is close to 1500ms, with some outlier values just below the deadline of 2500ms. Per design, Medley is trying to optimize for energy savings as long as the deadlines of the tasks are being met, which is highlighted in Table 8.1. Medley is able to achieve a 12.5x reduction in energy savings on the executing machines, while still meeting the deadlines. For all the offloaded tasks on Medley, the Jetson Nano is chosen as the executing device.

8.2.2 Measuring Adaptability of the Medley Ecosystem

We also study how the Medley ecosystem adapts to a tightened deadline for the room classification application. We offload an additional 600 room tasks from the robot, but now with a deadline of



(a) Task Distribution (deadline=2500ms)

(b) Task Distribution (deadline=1750ms)

Figure 8.3: Shows the task distribution in the Medley ecosystem (a) before, and (b) after tightening the deadline.

1750ms.

The cumulative DSR is shown in Figure 8.2a, and even at the tightened deadline, the scheduler maintains a high DSR of 98.16% after 600 offloads, missing only a few task deadlines. We also observer the task execution time spread for every 50 offloads is shown in Figure 8.2b. Compared to Figure 8.1a, there is a very slight downward shift in the median execution times, which the scheduler achieves by switching to other executors in the ecosystem. This is highlighted in the task distribution of the Medley scheduler before and after tightening the deadline, as shown in Figure 8.3. With a lax deadline, the ecosystem is able to meet the task deadline on the Jetson Nano itself, but when it starts missing deadlines, it attempts to switch to other executors to try and meet the deadline.

This case study highlights a real world use case where unused compute is used to satisfy the quality of service of edge tasks without requiring a dedicated edge server machine. It emphasizes the importance of an ecosystem like Medley to utilize this unused compute and to adapt based on user requirements. It showcases how devices like robots could benefit from task offloading to support cutting-edge software and save energy, without risking device obsolescence and increasing the usable life of the device.

CHAPTER 9
CONCLUSION

Edge computing is touted as a solution to manage the massive amounts of data expected to be generated from the "trillion device vision" of IoT, by improving quality of service of applications by reducing latency, and bringing processing closer to devices. However, current edge computing solutions require a "data center on the edge" infrastructure, either physically present near the devices, or as a geographically close service provided by a cloud service. The high cost to deploy and maintain such infrastructures, and limited availability of edge services at present mean that the majority of IoT deployments have limited access to edge computing.

We explore other alternatives to provide edge computing, specifically using unused compute resources opportunistically available in gateway devices near IoT devices, without requiring any cloud support. However, this comes with its own challenges. There are challenges due to the heterogeneous and dynamic nature of IoT: how to provide interoperability among diverse IoT devices with a plethora of differences, how to support gateways with a varying set of compute capabilities, and how to handle ever-changing dynamic IoT workloads. There are challenges on how to operate on constrained resources when you take away infinite resources of the cloud or limited yet relatively much better resources of an edge server. There are also challenges on how to provide users with a better control over their data when working with interoperable IoT devices.

We provide multiple solutions to handle these challenges. The first solution, NexusEdge, enables multiple disjoint gateways to collaborate together to form a decentralized cohesive edge computing platform. NexusEdge exploits data locality of IoT applications, provides resilience by handling gateway failures, and handles IoT device heterogeneity to improve interoperability, to support a good subset of IoT applications.

To support IoT applications which are more compute-intensive and have realtime requirements, we introduce Medley, an IoT device ecosystem. We show with Medley ecosystem how it can support several cutting-edge IoT applications by supporting task offload from resource-constrained device to more resource-powerful devices.

We also show how the NexusEdge middleware and the decentralized architecture it uses enables a natural way to enforce user-driven privacy policies to control how IoT applications use their data. These solutions together open up a new paradigm of exploiting unused compute to support edge computing. We deploy these solutions in real world deployments and find that using such a system is indeed viable for edge computing, and that it is cost-effective, general-purpose, privacy-aware, scalable, all without any dependency on dedicated edge infrastructure or the cloud.

It will be fascinating to see how the IoT landscape evolves, particularly how devices are expected to interoperate in smart home and other environments. IoT devices currently are heterogeneous in its use of wireless radios and protocols that they use, causing it a major challenge for

devices to interoperate. The NexusEdge middleware's approach to this problem is to let devices stay heterogeneous and instead shift the responsibility to the gateways to handle this heterogeneity and aid interoperability. Although, this aids end devices to be thin, not restricting them to a specific communication protocol, there is the tradeoff to have multiprotocol gateways. An alternative is to follow the architecture supported by Thread networks for IoT, which also advocates for thin end devices by requiring all Thread devices to use IP as the network layer. However, this means all devices are restricted to very limited payloads, which may not be viable for *all* IoT devices. This IoT interoperability problem is a ripe area for future work, and an architecture that makes use of the best of both NexusEdge and Thread approaches could be viable.

Other future directions include translating the time-based privacy policy preferences from our work on the Matter application layer for IoT. Time-based preferences could be a better semantic fit for end users compared to device-specific access controls, and implementing a Matter controller application which enforces these preferences on applications is an option. If Matter becomes the primary standard for IoT, this could be an opportunity for user-centric privacy controls for IoT to reach a lot more devices.

Another direction is to introduce more encompassing privacy preference like "collect only 10 samples of temperature per minute from a space". This is challenging, because if there are two different temperature sensors sampling from the space, and only do so at 10 samples/min, but they are interleaved, then this preference may not be met. It throws light on the difficulty in enforcing privacy preferences for richer preferences that users might have in IoT.

Other future work could be to reduce the "learning" start up time present for the Medley task scheduler. One approach could be to use non-consequential task offloads, which are tasks not sent on-demand from offloading devices. During periods of low activity, like overnight, the scheduler can explore executor performance by itself offloading tasks it has previously seen or missed deadlines for. In some deployments, it may be possible to allow such occasional exploration for offloaded tasks for non-urgent tasks. But the performance improvement with this approach could have energy implications on the executor devices, and this needs to be studied.

Another approach would be to use reinforcement learning (RL) for the task scheduler. Our current task scheduler also learns from feedback but is very procedural, and could be replaced with RL which could improving profiling tasks on the go, learning their execution time, resource impact, and energy impact on each executor to satisfy task QoS while minimizing overall energy of the ecosystem. We believe it might also help with adaptability; when new tasks or devices are added, or device behaviors change, enabling the scheduler to adapt without requiring updates to the system. Compared to other machine learning approaches, reinforcement learning doesn't require a labelled task execution dataset, which would need to be collected by task profiling and such data sets would not be adaptable to different environments.

Another future direction would be to study the impact of our work when the IoT deployment shifts from indoors to outdoors. Our current deployments are all indoors, and it will be interesting to see if there will be any changes if the deployments move outdoors. For instance, does the wireless discovery mechanism used in NexusEdge get affected when it is moved outdoors. Would

it be better to rely on some other wireless radio for discovery, are there more chances of creating secluded gateways due to limited network reach etc. are all interesting questions to investigate. It would also be interesting to see if there are potential unforeseen challenges which arise with this shift.

We believe the techniques presented in this dissertation help expand the vision of edge-centric computing by enabling a new way of providing compute resources for edge computing, without requiring dedicated cloud-dependent edge infrastructure.

BIBLIOGRAPHY

[1] M. Intelligence. "Internet of things (iot) market - growth, trends, covid-19 impact, and forecasts (2022 - 2027)." (2022), [Online]. Available: https://www.mordorintelligence.com/industry-reports/internet-of-things-moving-towards-a-smarter-tomorrow-market-industry.

[2] Philips. "Philips hue." (2019), [Online]. Available: https://www2.meethue.com/en-us.

[3] SmartThings, *Smartthings hub*, 2019. [Online]. Available: https://www.smartthings.com/gb/products/smartthings-hub.

[4] Centerfield, *ADT Smart Security*. [Online]. Available: https://www.adtsecuritysystem.com.

[5] T. R. P. Foundation. "Raspberry pi 4." (2021), [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-4-model-b.

[6] N. A. Zakaria, F. N. B. M. Saleh, and M. A. A. Razak, "Iot (internet of things) based infant body temperature monitoring," in *2018 2nd international conference on biosignal analysis, processing and systems (ICBAPS)*, IEEE, 2018, pp. 148–153.

[7] D. Davcev, K. Mitreski, S. Trajkovic, V. Nikolovski, and N. Koteli, "Iot agriculture system based on lorawan," in *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*, IEEE, 2018, pp. 1–4.

[8] R. K. Kodali, G. Swamy, and B. Lakshmi, "An implementation of iot for healthcare," in *2015 IEEE Recent Advances in Intelligent Computational Systems (RAICS)*, IEEE, 2015, pp. 411–416.

[9] A. W. Services. "Aws iot." (2019), [Online]. Available: https://aws.amazon.com/iot/.

[10] Microsoft. "Azure iot." (2019), [Online]. Available: https://azure.microsoft.com/en-us/overview/iot/.

[11] Google. “Google cloud iot solutions.” (2022), [Online]. Available: https://cloud.google.com/solutions/iot.

[12] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, “The case for vm-based cloudlets in mobile computing,” *IEEE pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[13] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog computing and its role in the internet of things,” in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, 2012, pp. 13–16.

[14] M. Satyanarayanan, Z. Chen, K. Ha, W. Hu, W. Richter, and P. Pillai, “Cloudlets: At the leading edge of mobile-cloud convergence,” in *6th International Conference on Mobile Computing, Applications and Services*, 2014, pp. 1–9. DOI: 10.4108/icst.mobicase.2014.257757.

[15] G. Guan, B. Li, Y. Gao, Y. Zhang, J. Bu, and W. Dong, “Tinylink 2.0: Integrating device, cloud, and client development for iot applications,” in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, 2020, pp. 1–13.

[16] I. Microsoft. “Tutorial: Use the azure cli and azure portal to configure iot hub message routing.” (2022), [Online]. Available: https://docs.microsoft.com/en-us/azure/iot-hub/tutorial-routing.

[17] I. Amazon Web Services. “Aws iot greengrass tutorial: Connect and test client devices.” (2022), [Online]. Available: https://docs.aws.amazon.com/greengrass/v2/developerguide/client-devices-tutorial.html.

[18] M. Dano. “Informa.” (2021), [Online]. Available: https://www.lightreading.com/the-edge-network/mapping-out-edge-computing-how-dense-is-it-.

[19] Spiceworks. “Edge platforms.” (2024), [Online]. Available: https://www.spiceworks.com/tech/edge-computing/articles/best-edge-computing-platforms/.

[20] N. Nasir, V. A. L. Sobral, L.-P. Huang, and B. Campbell, “Nexusedge: Leveraging iot gateways for a decentralized edge computing platform,” in *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*, 2022, pp. 82–95. DOI: 10.1109/SEC54971.2022.00014.

[21] C. S. Alliance. “Build with matter.” (2024), [Online]. Available: https://csa-iot.org/all-solutions/matter/.

[22] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing—a key technology towards 5g," *ETSI white paper*, vol. 11, no. 11, pp. 1–16, 2015.

[23] M. De Donno, K. Tange, and N. Dragoni, "Foundations and evolution of modern computing paradigms: Cloud, iot, edge, and fog," *Ieee Access*, vol. 7, pp. 150 936–150 948, 2019.

[24] J. Ren, D. Zhang, S. He, Y. Zhang, and T. Li, "A survey on end-edge-cloud orchestrated network computing paradigms: Transparent computing, mobile edge computing, fog computing, and cloudlet," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–36, 2019.

[25] K. Dolui and S. K. Datta, "Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing," in *2017 Global Internet of Things Summit (GIoTS)*, IEEE, 2017, pp. 1–6.

[26] A. W. Services. "Aws iot greengrass." (2020), [Online]. Available: `https://aws.amazon.com/greengrass/`.

[27] Microsoft. "Azure iot edge." (2020), [Online]. Available: `https://azure.microsoft.com/en-us/services/iot-edge/`.

[28] J. Clemente, M. Valero, J. Mohammadpour, X. Li, and W. Song, "Fog computing middleware for distributed cooperative data analytics," in *2017 IEEE Fog World Congress (FWC)*, IEEE, 2017, pp. 1–6.

[29] B.-Y. Ooi, Z.-W. Kong, W.-K. Lee, S.-Y. Liew, and S. Shirmohammadi, "A collaborative iot-gateway architecture for reliable and cost effective measurements," *IEEE Instrumentation & Measurement Magazine*, vol. 22, no. 6, pp. 11–17, 2019.

[30] O. Gnawali *et al.*, "The tenet architecture for tiered sensor networks," in *Proceedings of the 4th international conference on Embedded networked sensor systems*, 2006, pp. 153–166.

[31] R. Yus, G. Bouloukakis, S. Mehrotra, and N. Venkatasubramanian, "Abstracting interactions with iot devices towards a semantic vision of smart spaces," in *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, 2019, pp. 91–100.

[32] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The Case for VM-Based Cloudlets in Mobile Computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, Oct. 2009,

Conference Name: IEEE Pervasive Computing, ISSN: 1558-2590. DOI: 10.1109/MPRV.2009.82.

[33] E. Cuervo *et al.*, "MAUI: Making smartphones last longer with code offload," en, in *Proceedings of the 8th international conference on Mobile systems, applications, and services - MobiSys '10*, San Francisco, California, USA: ACM Press, 2010, p. 49, ISBN: 978-1-60558-985-5. DOI: 10.1145/1814433.1814441. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1814433.1814441 (visited on 12/09/2022).

[34] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, ser. MobiSys '14, New York, NY, USA: Association for Computing Machinery, Jun. 2014, pp. 68–81, ISBN: 978-1-4503-2793-0. DOI: 10.1145/2594368.2594383. [Online]. Available: https://doi.org/10.1145/2594368.2594383 (visited on 10/28/2022).

[35] L. Lin, X. Liao, H. Jin, and P. Li, "Computation offloading toward edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1584–1607, 2019.

[36] Q. Luo, S. Hu, C. Li, G. Li, and W. Shi, "Resource Scheduling in Edge Computing: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 4, pp. 2131–2165, 2021, Conference Name: IEEE Communications Surveys & Tutorials, ISSN: 1553-877X. DOI: 10.1109/COMST.2021.3106401.

[37] D. Zhang, Y. Ma, C. Zheng, Y. Zhang, X. S. Hu, and D. Wang, "Cooperative-competitive task allocation in edge computing for delay-sensitive social sensing," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, IEEE, 2018, pp. 243–259.

[38] D. Zhang, T. Rashid, X. Li, N. Vance, and D. Wang, "Heteroedge: Taming the heterogeneity of edge computing system in social sensing," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, 2019, pp. 37–48.

[39] Y. Kim, C. Song, H. Han, H. Jung, and S. Kang, "Collaborative task scheduling for iot-assisted edge computing," *IEEE Access*, vol. 8, pp. 216 593–216 606, 2020.

[40] P. Pappachan *et al.*, "Towards privacy-aware smart buildings: Capturing, communicating, and enforcing privacy policies and preferences," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, IEEE, 2017, pp. 193–198.

[41] S. Ghayyur, P. Pappachan, G. Wang, S. Mehrotra, and N. Venkatasubramanian, "Designing privacy preserving data sharing middleware for internet of things," in *Proceedings of the Third Workshop on Data: Acquisition To Analysis*, 2020, pp. 1–6.

[42] A. Das, M. Degeling, D. Smullen, and N. Sadeh, "Personalized privacy assistants for the internet of things: Providing users with notice and choice," *IEEE Pervasive Computing*, vol. 17, no. 3, pp. 35–46, 2018.

[43] A. Al-Hasnawi and L. Lilien, "Pushing data privacy control to the edge in iot using policy enforcement fog module," in *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*, 2017, pp. 145–150.

[44] A. Erickson, K. Kim, R. Schubert, G. Bruder, and G. Welch, "Is it cold in here or is it just me? analysis of augmented reality temperature visualization for computer-mediated thermoception," in *2019 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, IEEE, 2019, pp. 202–211.

[45] D. Jo and G. J. Kim, "Ariot: Scalable augmented reality framework for interacting with internet of things appliances everywhere," *IEEE Transactions on Consumer Electronics*, vol. 62, no. 3, pp. 334–340, 2016.

[46] Y. Yuan, W. Zou, Y. Zhao, X. Wang, X. Hu, and N. Komodakis, "A robust and efficient approach to license plate detection," *IEEE Transactions on Image Processing*, vol. 26, no. 3, pp. 1102–1114, 2016.

[47] A. Høst-Madsen, N. Petrochilos, O. Boric-Lubecke, V. M. Lubecke, B.-K. Park, and Q. Zhou, "Signal processing methods for doppler radar heart rate monitoring," in *Signal processing techniques for knowledge extraction and information fusion*, Springer, 2008, pp. 121–140.

[48] T. N. Gia *et al.*, "Iot-based fall detection system with energy efficient sensor nodes," in *2016 IEEE Nordic Circuits and Systems Conference (NORCAS)*, IEEE, 2016, pp. 1–6.

[49] L. Zhou and H. Guo, "Anomaly detection methods for iiot networks," in *2018 IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI)*, IEEE, 2018, pp. 214–219.

[50] U. D. of Health & Human Services. "Health information privacy." (2022), [Online]. Available: `https://www.hhs.gov/hipaa/index.html`.

[51] I. World. "How to build an industrial iot project without the cloud." (2022), [Online]. Available: https://www.iiot-world.com/industrial-iot/connected-industry/how-to-build-an-industrial-iot-project-without-the-cloud/.

[52] I. World. "Industrial iot law – iiot legal and regulatory aspects." (2022), [Online]. Available: https://www.iiot-world.com/industrial-iot/connected-industry/industrial-iot-legal-and-regulatory-aspects/.

[53] D. A. Winkler and A. E. Cerpa, "Wisdom: Watering intelligently at scale with distributed optimization and modeling," in *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, 2019, pp. 219–231.

[54] T. Meyer and G. Hancke, "Design of a smart sprinkler system," in *TENCON 2015-2015 IEEE Region 10 Conference*, IEEE, 2015, pp. 1–6.

[55] K. L. Keung, C. K. M. Lee, K. Ng, and C.-K. Yeung, "Smart city application and analysis: Real-time urban drainage monitoring by iot sensors: A case study of hong kong," in *2018 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, IEEE, 2018, pp. 521–525.

[56] S. R. Hussain, S. Mehnaz, S. Nirjon, and E. Bertino, "Seamblue: Seamless bluetooth low energy connection migration for unmodified iot devices," in *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks*, ser. EWSN ’17, Uppsala, Sweden: Junction Publishing, 2017, pp. 132–143, ISBN: 978-0-9949886-1-4. [Online]. Available: http://dl.acm.org/citation.cfm?id=3108009.3108027.

[57] T. Beke, E. Dijk, T. Ozcelebi, and R. Verhoeven, "Time synchronization in iot mesh networks," in *2020 International Symposium on Networks, Computers and Communications (ISNCC)*, IEEE, 2020, pp. 1–8.

[58] S. Choi and J.-H. Lee, "Blockchain-based distributed firmware update architecture for iot devices," *IEEE Access*, vol. 8, pp. 37 518–37 525, 2020.

[59] M. Banerjee, C. Borges, K.-K. R. Choo, J. Lee, and C. Nicopoulos, "A hardware-assisted heartbeat mechanism for fault identification in large-scale iot systems," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[60] E. Systems. "Esp32 wi-fi and bluetooth mcu." (2021), [Online]. Available: https://www.espressif.com/en/products/socs/esp32.

[61] F. Wang, X. Huang, H. Nian, Q. He, Y. Yang, and C. Zhang, "Cost-effective edge server placement in edge computing," in *Proceedings of the 2019 5th International Conference on Systems, Control and Communications*, ser. ICSCC 2019, Wuhan, China: Association for Computing Machinery, 2019, 6–10, ISBN: 9781450372640. DOI: 10.1145/3377458.3377461. [Online]. Available: https://doi.org/10.1145/3377458.3377461.

[62] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Communications Surveys Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017, ISSN: 1553-877X.

[63] J. Ren, D. Zhang, S. He, Y. Zhang, and T. Li, "A survey on end-edge-cloud orchestrated network computing paradigms: Transparent computing, mobile edge computing, fog computing, and cloudlet," *ACM Comput. Surv.*, vol. 52, no. 6, Oct. 2019, ISSN: 0360-0300. DOI: 10.1145/3362031. [Online]. Available: https://doi.org/10.1145/3362031.

[64] IETF. "The constrained application protocol (coap)." (2014), [Online]. Available: https://tools.ietf.org/html/rfc7252.

[65] IETF. "Rfc 8949 concise binary object representation." (2020), [Online]. Available: https://cbor.io/.

[66] A. Javed, K. Heljanko, A. Buda, and K. Främling, "Cefiot: A fault-tolerant iot architecture for edge and cloud," in *2018 IEEE 4th world forum on internet of things (WF-IoT)*, IEEE, 2018, pp. 813–818.

[67] D. Y. Zhang, T. Rashid, X. Li, N. Vance, and D. Wang, "Heteroedge: Taming the heterogeneity of edge computing system in social sensing," ser. IoTDI '19, Montreal, Quebec, Canada: ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3302505.3310067.

[68] F. Samie, V. Tsoutsouras, L. Bauer, S. Xydis, D. Soudris, and J. Henkel, "Computation offloading and resource allocation for low-power iot edge devices," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, IEEE, 2016, pp. 7–12.

[69] Y. Shi, S. Abhilash, and K. Hwang, "Cloudlet mesh for securing mobile clouds from intrusions and network attacks," in *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, IEEE, 2015, pp. 109–118.

[70] A. M. M. Ali, N. M. Ahmad, and A. H. M. Amin, "Cloudlet-based cyber foraging framework for distributed video surveillance provisioning," in *2014 4th World Congress on Information and Communication Technologies (WICT 2014)*, IEEE, 2014, pp. 199–204.

[71] Arxys. "Nvr bandwidth and storage calculator." (2021), [Online]. Available: https://www.arxys.com/bandwidth-storage-calculator/.

[72] O. Russakovsky *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015. DOI: 10.1007/s11263-015-0816-y.

[73] S. Albright, P. J. Leach, Y. Gu, Y. Y. Goland, and T. Cai, "Simple Service Discovery Protocol/1.0," Internet Engineering Task Force, Internet-Draft draft-cai-ssdp-v1-03, Nov. 1999, Work in Progress, 18 pp. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-cai-ssdp-v1-03.

[74] W. A. Simpson, D. T. Narten, E. Nordmark, and H. Soliman, *Neighbor Discovery for IP version 6 (IPv6)*, RFC 4861, Sep. 2007. DOI: 10.17487/RFC4861. [Online]. Available: https://www.rfc-editor.org/info/rfc4861.

[75] I. Murturi, C. Avasalcai, C. Tsigkanos, and S. Dustdar, "Edge-to-edge resource discovery using metadata replication," in *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*, 2019, pp. 1–6. DOI: 10.1109/CFEC.2019.8733149.

[76] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, "Challenges and opportunities in edge computing," in *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, 2016, pp. 20–26. DOI: 10.1109/SmartCloud.2016.18.

[77] T. Group. "Thread protocol." (2021), [Online]. Available: https://www.threadgroup.org/.

[78] G. C. Inc. "Ant / ant+ defined." (2021), [Online]. Available: https://www.thisisant.com/developer/ant-plus/ant-antplus-defined.

[79] DigiMesh. "Digimesh products." (2021), [Online]. Available: https://www.digi.com/products/browse/digimesh.

[80] S. S.A. “Sigfox - the global communications service provider for the internet of things (iot).” (2021), [Online]. Available: https://www.sigfox.com/.

[81] C. N. C. Foundation. “Kubeedge.” (2020), [Online]. Available: https://kubeedge.io/en/.

[82] K. Project. “Device model for kubeedge.” (2021), [Online]. Available: https://kubeedge.io/en/docs/developer/device_crd/.

[83] Microsoft. “Understand and use device twins in iot hub.” (2021), [Online]. Available: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-device-twins.

[84] O. D. Model. “Onedm: Solving the problem of lack of a common data model for iot and iot devices.” (2021), [Online]. Available: https://onedm.org/.

[85] T. O. Group. “Open data format (o-df), an open group internet of things (iot) standard.” (2021), [Online]. Available: http://www.opengroup.org/iot/odf/.

[86] Z. Alliance. “Dotdot: A common language for the smart objects.” (2021), [Online]. Available: https://zigbeealliance.org/solution/dotdot/.

[87] E. GmbH. “Energy harvesting wireless sensor solutions and networks from enocean.” (2019), [Online]. Available: https://www.enocean.com/.

[88] Estimote. “Estimote proximity beacons.” (2021), [Online]. Available: https://estimote.com/.

[89] openHAB Community. “Openhab.” (2021), [Online]. Available: https://www.openhab.org/.

[90] H. Assistant. “Open source home assistant.” (2021), [Online]. Available: https://www.home-assistant.io/.

[91] D. Inc. “Docker.” (2022), [Online]. Available: https://www.docker.com/.

[92] DockerHub. “Docker hub.” (2022), [Online]. Available: https://www.docker.com/products/docker-hub/.

[93] K. P. Authors. “Lightweight kubernetes.” (2022), [Online]. Available: https://k3s.io/.

[94] T. K. Authors. “Kubernetes - production-grade container orchestration.” (2022), [Online]. Available: https://kubernetes.io/.

[95] DaemonSet. “Kubernetes daemonset.” (2022), [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/.

[96] J. Kelly and W. Knottenbelt, “The UK-DALE dataset, domestic appliance-level electricity demand and whole-house demand from five UK homes,” *Scientific Data*, vol. 2, no. 150007, 2015. DOI: 10.1038/sdata.2015.7.

[97] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, “Towards wearable cognitive assistance,” in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, 2014, pp. 68–81.

[98] M. A. S. Mondol and J. A. Stankovic, “Harmony: A hand wash monitoring and reminder system using smart watches,” in *proceedings of the 12th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services on 12th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2015, pp. 11–20.

[99] Google. “Google nest doorbell.” (2023), [Online]. Available: https://store.google.com/us/product/nest_doorbell?hl=en-US.

[100] S. Corporation. “Playstation 5.” (2023), [Online]. Available: https://www.playstation.com/en-us/ps5/.

[101] J. Nieh, S. J. Yang, and N. Novik, “Measuring thin-client performance using slow-motion benchmarking,” *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 1, pp. 87–115, 2003.

[102] S. J. Yang, J. Nieh, M. Selsky, and N. Tiwari, “The performance of remote display mechanisms for thin-client computing.,” in *USENIX Annual Technical Conference, General Track*, 2002, pp. 131–146.

[103] G. Rodola. “Psutil 5.9.4.” (2023), [Online]. Available: https://pypi.org/project/psutil/.

[104] P. Mavrogiorgos. “Nvsmi 0.4.2.” (2023), [Online]. Available: https://pypi.org/project/nvsmi/.

[105] NVIDIA. “Tegrastats utility.” (2023), [Online]. Available: https://docs.nvidia.com/drive/drive_os_5.1.6.1L/nvvib_docs/index.html#page/DRIVE_OS_Linux_SDK_Development_Guide/Utilities/util_tegrastats.html#wwpID0EEHA.

[106] O. Banos, R. Garcia, and A. Saez, *MHEALTH*, UCI Machine Learning Repository, DOI: https://doi.org/10.24432/C5TW22, 2014.

[107] G. Jocher, *YOLOv5 by Ultralytics*, version 7.0, May 2020. DOI: 10.5281/zenodo.3908559. [Online]. Available: https://github.com/ultralytics/yolov5.

[108] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, “Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size,” *arXiv preprint arXiv:1602.07360*, 2016.

[109] M. Guzek, D. Kliazovich, and P. Bouvry, “Heros: Energy-efficient load balancing for heterogeneous data centers,” in *2015 IEEE 8th International Conference on Cloud Computing*, IEEE, 2015, pp. 742–749.

[110] Geekbench. “Geekbench 6 - cross-platform benchmark.” (2023), [Online]. Available: https://www.st.com/en/evaluation-tools/nucleo-f446re.html.

[111] L. system programming training. “Cron - linux manual page.” (2024), [Online]. Available: https://man7.org/linux/man-pages/man8/cron.8.html.

[112] C. S. Alliance. “Matter - github.” (2024), [Online]. Available: https://github.com/project-chip/connectedhomeip.

[113] C. S. Alliance. “Matter - spec v1.0.” (2024), [Online]. Available: https://csa-iot.org/wp-content/uploads/2022/11/22-27349-001_Matter-1.0-Core-Specification.pdf.

[114] element14. “A starter guide for the new matter smart home automation protocol.” (2024), [Online]. Available: https://community.element14.com/learn/learning-center/essentials/w/documents/28137/a-starter-guide-for-the-new-matter-smart-home-automation-protocol.

[115] N. Semiconductor. “Matter nrf connect light switch example application.” (2024), [Online]. Available: https://github.com/project-chip/connectedhomeip/tree/master/examples/light-switch-app/nrfconnect.

[116] Espressif. “Matter: Device-to-device automations.” (2024), [Online]. Available: https://blog.espressif.com/matter-device-to-device-automations-bdbb32365350.

[117] Espressif. “Matter: Device-to-device automations.” (2024), [Online]. Available: https://blog.espressif.com/matter-multi-admin-identifiers-and-fabrics-a291371af365.

[118] C. S. Alliance. “Matter acl - github.” (2024), [Online]. Available: https://github.com/project-chip/connectedhomeip/blob/master/docs/guides/access-control-guide.md.

[119] iRobot Corporation. “Guide to imprint smart maps.” (2024), [Online]. Available: https://homesupport.irobot.com/s/article/64102.

[120] temi USA. “Temi robots.” (2024), [Online]. Available: https://www.robotemi.com/robots/.

[121] temi USA. “Robotemi github - wiki.” (2024), [Online]. Available: https://github.com/robotemi/sdk/wiki.

[122] Wikipedia contributors, *Android marshmallow — Wikipedia, the free encyclopedia*, [Online; accessed 3-April-2024], 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Android_Marshmallow&oldid=1216587653.