Universidade Estadual de Campinas
Instituto de Computação

Flávia Pisani

Leveraging Constrained Devices for Custom Code Execution in the Internet of Things

Utilizando Dispositivos Limitados para Execução de Código Personalizado na Internet das Coisas

CAMPINAS
2019

Flávia Pisani

Leveraging Constrained Devices for Custom Code Execution in the Internet of Things

Utilizando Dispositivos Limitados para Execução de Código Personalizado na Internet das Coisas

Tese apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutora em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Supervisor/Orientador: Prof. Dr. Edson Borin

Este exemplar corresponde à versão final da Tese defendida por Flávia Pisani e orientada pelo Prof. Dr. Edson Borin.

CAMPINAS
2019

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

P672L

Pisani, Flávia, 1992-
Leveraging constrained devices for custom code execution in the Internet of things / Flávia Pisani. – Campinas, SP : [s.n.], 2019.

Orientador: Edson Borin.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Internet das coisas. 2. Dispositivos limitados. 3. Computação em névoa. I. Borin, Edson, 1979-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Utilizando dispositivos limitados para execução de código personalizado na Internet das coisas
Palavras-chave em inglês:
Internet of things
Constrained devices
Fog computing
Área de concentração: Ciência da Computação
Titulação: Doutora em Ciência da Computação
Banca examinadora:
Edson Borin [Orientador]
Jó Ueyama
Markus Endler
Edmundo Roberto Mauro Madeira
Luiz Fernando Bittencourt
Data de defesa: 24-06-2019
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0003-2670-7197
- Currículo Lattes do autor: http://lattes.cnpq.br/5258302049114084

Universidade Estadual de Campinas
Instituto de Computação

Flávia Pisani

Leveraging Constrained Devices for Custom Code Execution in the Internet of Things

Utilizando Dispositivos Limitados para Execução de Código Personalizado na Internet das Coisas

Banca Examinadora:

- Prof. Dr. Edson Borin
  IC/UNICAMP
- Prof. Dr. Jó Ueyama
  ICMC/USP
- Prof. Dr. Markus Endler
  DI/PUC-Rio
- Prof. Dr. Edmundo Roberto Mauro Madeira
  IC/UNICAMP
- Prof. Dr. Luiz Fernando Bittencourt
  IC/UNICAMP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 24 de junho de 2019

Acknowledgements

First, I would like to thank my PhD advisor (and undergraduate co-advisor), Prof. Edson Borin. After seven years, I am still learning a lot from you and I hope we have other opportunities to work together in the future.

I would also like to thank my friends and colleagues for the many conversations, collaborations, and good times. Your presence made this journey a lot more fun.

I would like to thank my parents and the rest of my family. You have supported me through my whole life and I am forever grateful for everything you have done for me.

I would like to thank Samuel. Your inputs have helped me grow as a person and a researcher, and I would not have made it without your love and support.

Finally, I would like to thank the professors and staff of the Institute of Computing, who were always there to give me advice and help me throughout this process. In particular, I would like to thank my undergraduate advisor, Prof. Ricardo da Silva Torres, who invited me to work with him and encouraged me to do a PhD.

This study was financed in part by the Fundação de Desenvolvimento da Unicamp (FUNCAMP) - Samsung grant, the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) - grant 140653/2017-1.

Resumo

Perspectivas atuais para a Internet das Coisas (IoT, do inglês, *Internet of Things*) indicam que, dentro de alguns anos, uma rede global de objetos irá conectar dezenas de milhares de dispositivos através da Internet. A maioria deles conterá sensores que geram fluxos de dados constantemente e é comum que usuários precisem processar esses dados antes de armazenar os valores finais. No entanto, com o crescimento progressivo da IoT, esta estratégia precisa ser reavaliada, já que transmitir um imenso volume de informações pela rede será muito custoso. Uma maneira de atender a estes futuros requisitos é trazer a computação para onde a informação já está. Com essa premissa, surgiu um paradigma chamado computação em névoa que propõe processar os dados perto de dispositivos da borda da rede. Outra característica comum dentre o grande número de dispositivos IoT será que seus recursos (ex.: energia, memória e processamento) serão limitados. O Instituto Nacional de Padrões e Tecnologia dos EUA chama a atenção para este tipo de dispositivo dentro do contexto da computação em névoa, nomeando-os nós de neblina, pois são nós de névoa leves. Contudo, eles caracterizam estes nós como mais especializados, dedicados e geralmente compartilhando a mesma localidade que os dispositivos finais inteligentes a quem eles atendem. Em nosso trabalho, nós propomos uma abordagem diferente, onde estes dispositivos limitados estão integrados com sensores e são capazes de executar código de propósito geral, podendo assim executar códigos personalizados enviados pelo usuário. A análise desta proposta é dividida em duas partes. Primeiro, nós desenvolvemos uma plataforma chamada LibMiletusCOISA (LMC), que permite que usuários executem seus códigos em dispositivos limitados e a comparamos com o arcabouço Apache Edgent. O desempenho do LMC foi melhor que o do Edgent quando ambos foram executados no mesmo dispositivo e ele possibilitou a execução de código em dispositivos limitados que eram muito restritos para executar a outra ferramenta. Depois, criamos dois modelos onde o usuário escolhe uma certa métrica de custo (ex.: número de instruções, tempo de execução ou consumo de energia) e a emprega para decidir onde o seu código deve ser executado. Um deles é um modelo matemático que usa uma equação linear para auxiliar o usuário a dividir o seu problema em um conjunto de variáveis e então determinar os custos de realizar a sua computação usando a nuvem e a névoa. O outro é um modelo visual que permite que o usuário conclua facilmente qual é a abordagem mais vantajosa em um cenário específico. Nós usamos estes modelos para identificar situações onde a abordagem escolhida é executar o código no dispositivo que coletou os dados e também simulamos cenários futuros onde mudanças na tecnologia podem impactar a decisão do usuário.

Abstract

Current prospects for the Internet of Things (IoT) indicate that, within the next few years, a global network of objects will connect tens of billions of devices through the Internet. Most of them will contain sensors that constantly generate data streams and it is common for users to need to process these data before storing the final values. However, with the ever-growing scale of the IoT, this strategy must be re-evaluated, as transmitting an immense volume of information through the network will be too taxing. One way to meet these coming requirements is to bring the computation to where the information already is. With this premise, emerged a paradigm called fog computing that proposes to process data closer to network edge devices. Another common characteristic among a large number of IoT devices will be that their resources (e.g., power, memory, and processing) will be limited. The USA's National Institute of Standards and Technology calls attention to this type of device within the fog computing context, naming them mist nodes, as they are lightweight fog computing nodes. Nevertheless, they characterize these nodes as more specialized, dedicated, and often sharing the same locality with the smart end-devices they service. In our work, we propose a different approach, where these constrained devices are integrated with sensors and capable of general-purpose code execution, thus being able to run custom code sent by users. The analysis of this proposal is divided into two parts. First, we developed a platform called LibMiletusCOISA (LMC), which allows users to execute their code on constrained devices and compared it to the Apache Edgent framework. LMC outperformed Edgent when both were executed on the same device, and made it possible to execute the code on a constrained device that was too limited to execute the other tool. Then, we created two models where the user chooses a certain cost metric (e.g., number of instructions, execution time, or energy consumption) and employs it to decide where their code should be executed. One of them is a mathematical model that uses a linear equation to help the user break down their problem into a set of variables and then determine the costs of performing their computation using the cloud and the fog. The other is a visual model that allows the user to easily conclude what is the most profitable approach in a specific scenario. We employed these models to identify the situations where executing the code on the device that collected the data is the chosen approach and also simulated future scenarios where changes in the technology can impact the user's decision.

List of Figures

List of Tables

List of Abbreviations

API Application Programming Interface 19, 35, 91

ARM Advanced RISC Machine 28

BLE Bluetooth Low Energy 28, 29, 60

CAT Cloud-Assisted Translation 37–39

CBIR Content-Based Image Retrieval 23

CoAP Constrained Application Protocol 79

COISA Constrained OpenISA 28–30, 35, 37, 65

CPU Central Processing Unit 78, 84

cURL Client URL 29

FFT fast Fourier transform 27, 36

HLL High-Level Language 20

HP Hewlett-Packard 23

HTTP Hypertext Transfer Protocol 79

I/O Input/Output 19

IaaS Infrastructure as a Service 85, 86

IDE Integrated Development Environment 28

IEC International Electrotechnical Commission 79

IETF Internet Engineering Task Force 78, 79

IoT Internet of Things 9, 16–21, 26, 27, 29, 34, 37, 41, 42, 64, 65, 86–89, 91, 93–95

IP Internet Protocol 79

ISA Instruction Set Architecture 20, 21, 28

JIT just-in-time 8, 37–39

JSON JavaScript Object Notation 28

JVM Java Virtual Machine 35, 37–39

LAN Local Area Network 22, 92

LMC LibMiletusCOISA 8, 10, 17, 25, 27–30, 32–40, 65–67

LTE Long-Term Evolution 60

MCC Mobile Cloud Computing 92, 93

mDNS Multicast Domain Name System 28

MEC Mobile-Edge Computing 25, 92, 93

MIPS Microprocessor without Interlocked Pipelined Stages 28

NFC Near-Field Communication 88, 89

NIST National Institute of Standards and Technologies 17, 83, 91, 94

OS Operating System 19, 20, 35

PaaS Platform as a Service 85, 86

RAM Random Access Memory 28, 34, 35, 78, 80

RFID Radio-Frequency Identification 86–89

RISC Reduced Instruction Set Computer 20

SaaS Software as a Service 85, 86

SSL Secure Sockets Layer 22

TCP Transmission Control Protocol 60

TLS Transport Layer Security 79

UDP User Datagram Protocol 60, 79

VM Virtual Machine 19–21, 27–30, 35, 37

WSAN Wireless Sensor and Actuator Network 92, 93

WSN Wireless Sensor Network 19, 20, 92, 93

List of Symbols

Table 1: Notation for the cloud and fog computing cost models.

Notation	**Restriction**	**Definition**
b	$0 < b \leq z$	Number of blocks into which the stream is divided to compute several f values along the stream.
$C_{_C}$	$C_{_C} > 0$	Cost of sending the data to be processed on the cloud.
$C_{_F}$	$C_{_F} > 0$	Cost of processing the data on the fog and sending the data that pass the filter to the cloud.
f	$0 \leq f \leq 1$	Probability that a value will pass the filter.
i	$i > 0$	Cost of being idle between processing a value and reading the next one.
n	$0 \leq n \leq v$	Number of values that passed the filter among the tested values.
p	$p > 0$	Penalty for processing a value on the fog when it would cost less to do so on the cloud.
r	$r > 0$	Cost of reading a value.
s	$s > 0$	Cost of sending a value to the cloud.
t	$t > 0$	Cost of executing a custom code that decides if the value should be sent to the cloud.
v	$0 < v \leq z$	Number of values that we are allowed to test to estimate f considering the maximum increase in cost defined by the user.
z	$z > 0$	Stream size (i.e., number of processed values).

Contents

Chapter 1

Introduction

Current prospects for the Internet of Things (IoT) indicate that, within the next few years, a global network of objects will connect tens of billions of devices through the Internet [54]. As this technology becomes more widespread, we can also expect that a large number of devices with limited resources (e.g., power, memory, and processing) will become part of it. These devices are known as constrained devices. Current examples of smart constrained devices are the 2nd-generation Nest Learning Thermostat (which has 128 KiB of RAM, 16 KiB of Flash, and a microprocessor with a frequency of up to 32 MHz) and the Fitbit activity tracker (which has 256 KiB of RAM, 64 KiB of Flash, and a microprocessor with a frequency of up to 80 MHz).

Most IoT devices will contain sensors to help them interact with the environment around them, and the data they collect will create many opportunities. For instance, this will allow the improvement of strategies for resource usage and management in settings such as urban planning and environmental sustainability, thus advancing agriculture and contributing to an increase in the number of smart cities. It will also promote automation and the use of cyber-physical systems, prompting the widespread adoption of industry 4.0 [10]. Moreover, considering that important processes such as data analytics can benefit from working with more data, this will lead to data scientists not only being able to better understand the world we live in, but also making more accurate predictions and creating improved systems based on people's behaviors and tastes.

Even though storage space on the cloud can be scaled according to the needs of its users, with petabytes of data being produced by IoT devices every day, the sheer volume of information will make transmitting every single byte to the cloud prohibitively expensive in terms of both time and money [69]. Furthermore, moving data streams from sensor nodes to servers will not only present the aforementioned costs, but also have an impact on the energy consumption of devices that may have constricted power budgets (e.g., devices that are battery-operated or depend on limited energy sources such as solar power).

A possible way to meet these expected requirements is not sending all the data to be processed by machines that are far from the data source, but instead bringing the computation closer to where the information already is. With this premise, a new paradigm called fog computing emerged, proposing to process data closer to network edge devices, such as switches and routers [14]. Enabling data to be processed near its origin (e.g., on a local access point, router, or even on the sensor device itself) allows us to address prob-

lems related to transmission latency and network congestion. It also opens up the space for new possibilities, such as filtering and discarding unnecessary information, analyzing readings in search of outliers to report, and actual real-time response to local queries.

The USA's National Institute of Standards and Technologies (NIST) calls attention to constrained devices within the fog computing context, naming them mist nodes [41], as they are lightweight fog computing nodes. However, NIST expects the purpose of these devices to be more specialized and dedicated, while we see a valuable opportunity in leveraging them for custom user code execution due to their proximity to the data source. For instance, mist nodes have the potential to perform simple custom operations on sensor streams, such as aggregation and filtering, to reduce network traffic and latency. Given that the mist is part of the fog, when we refer to fog computing in this dissertation, we are considering both fog and mist nodes.

Despite the potential of employing constrained IoT devices as a part of the fog hierarchy, many current fog computing frameworks [2, 22, 30, 76] still require more resources than what these devices provide, and thus cannot be used to enable them to execute custom code. Therefore, investigating solutions that involve resource-constrained devices and creating a lean infrastructure that enables their seamless incorporation into the IoT is a gap that both industry and academia must explore to enable the full potential of this technology.

In particular, we highlight the importance of understanding and characterizing the scenarios where these devices can be used efficiently, as this is the key to developing solutions that use the most profitable approach for each specific problem. For example, although fog computing brings several advantages, there may be cases where the computation takes longer or requires more energy to be completed on the fog device than it would it take to send the values to the cloud. In addition, there are situations where the whole computation depends on many different data sources. In these instances, it might be more profitable to send the data to be processed by the more robust cloud servers instead of executing the program locally.

Considering this context, this dissertation works toward the answer to the research question "*In what cases is it more profitable to perform computation on a constrained IoT device instead of using the cloud?*".

Unlike many works in the area, which focus on either the technical or theoretical aspects of handling constrained devices and offloading the computation from these devices to more powerful ones, our investigation used a broader approach, which combined both the implementation and the formalization of our solution.

In the first part of our analysis, we developed a platform called LibMiletusCOISA (LMC), which allows users to execute their code on constrained devices, and compared it to the Apache Edgent framework [2]. LMC performed well when executed on the same device as Edgent and made it possible to execute the code on a constrained device, which does not have enough memory to support the execution of existing fog computing tools.

With that, we established that indeed there are cases where it is faster to perform the computation on a constrained IoT device (that is, more profitable in terms of time), therefore giving us more evidence to support the investigation of our research question. The main contributions brought by this part of the analysis are the infrastructure that

we developed, which allows us to send user programs to constrained devices and execute them, and the fact that we employed it to obtain real-world values for our test cases.

In the second part of our analysis, we created two models where the user chooses a certain cost metric (e.g., number of instructions, execution time, or energy consumption) and employs it to decide where they should execute their code. One of them is a generic mathematical model that uses a linear equation to determine the costs and the other is a visual model that allows the user to conclude quickly what is the most profitable approach in a specific scenario.

For instance, consider that the user chose energy consumption as their cost metric. The mathematical model would allow them to calculate, from the point of view of the device that is collecting the data, how much energy would be required for the device to perform the computation and send its result to the cloud, as well as how much energy would be required to send all data to the cloud instead. The visual model would then allow the user to visualize easily the approach where less energy is used (that is, the approach that is more profitable in terms of energy).

We used datasheets and the infrastructure built in the first part of our analysis to obtain real-world values to use in our test cases, and then used these values as the input for simulations that employed our models to identify the situations where executing the code on the device that collected the data would be the chosen approach. We also simulated future scenarios where changes in communication and processing technologies can affect whether the fog or the cloud is the most profitable solution.

With that, we formalized an approach to identifying the cases where it is more profitable to perform computation on a constrained IoT device instead of using the cloud, which is what we intended with the investigation guided by our research question. The main contributions brought by this part of the analysis are the mathematical and visual models, a procedure to estimate the probability of a value passing a filter based on the cost penalty that the user is willing to pay for this calculation, and the simulation of future scenarios.

This dissertation is organized as follows: Chapter 2 covers studies that are related to our work on the topics of execution of code on sensors, fog computing tools, general computation offloading schemes, and edge/fog/cloud computing offloading schemes. Chapter 3 has the first part of our analysis, where we introduce the LibMiletusCOISA framework and discuss the experimental results related to it. Chapter 4 describes the second part of our analysis, in which we present our mathematical and visual models and discuss the experimental results related to them. Chapter 5 gives our conclusions and possibilities for future work. Appendix A characterizes important background concepts for this research: constrained devices, cloud computing, IoT, fog computing, and mist computing. Finally, Appendix B shows more detailed data for the experimental results discussed in the dissertation.

Chapter 2

Related Work

This chapter presents the result of our research into the state-of-the-art technologies related to our study and the opportunities and challenges we can tackle to advance it.

2.1 Infrastructure for Code Execution on Sensors

Throughout the years, much work has been put into developing technologies that provide the necessary infrastructure for Wireless Sensor Networks (WSNs). Although this type of network operates on a considerably smaller scale than the IoT, the knowledge obtained by WSN studies can provide valuable insights for this emerging paradigm. For instance, we have Operating Systems (OSs) that were specifically created to be executed on constrained devices.

Tiny Operating System (TinyOS) [1] can be considered one of the most robust, innovative, energy-efficient, and widely used OSs for WSNs. It was designed for low power sensing motes and it is a flexible solution that supports concurrency, has easy memory management, and uses good scheduling algorithms, while also complying with resource limitations. Furthermore, compared to other OSs for sensors, TinyOS code is very simple and short, and its installation and applications require less memory.

More recently, other OSs have been created targeting not only sensors but the "things" that are part of the IoT as well. For instance, there is Android Things [31], which integrates additional Application Programming Interfaces (APIs) to the core Android framework, allowing developers to work with hardware that is not found on mobile devices. By removing the need for kernel and firmware development, this tool also enables developers without previous knowledge of embedded system design to build applications. One more example is the ARM Mbed OS [4], which focuses on ARM Cortex-M microcontrollers. It provides security, connectivity, a real-time OS, and drivers for sensors and I/O devices, making prototyping for IoT applications quicker on low-cost development boards.

Another important feature for code execution on sensors is the ability to update software through wireless connections instead of physical access to the hardware, be this update a simple bug fix or a complete re-tasking of the sensor. Approaches to remotely reprogramming a WSN can be divided into four main categories: Full-Image Replacement, Differential Image Replacement, Dynamic Operating Systems, and Virtual Machines [61].

Full-Image Replacement techniques, such as XNP [23] or Deluge [40], work by propagating a new binary image of both an application and an OS in the network. Since the image is recompiled and relinked in every iteration, this type of solution offers a very fine-grained control over the possible new configurations. However, this may lead to bandwidth overhead, as unchanged parts of an application also need to be retransmitted.

Differential Image Replacement approaches, like the ones used by Zephyr [66] and other tools [43, 71], aim to reduce bandwidth consumption by only disseminating the changes between an executable deployed in the network and a new image. Still, they do not possess high-level knowledge of the application structure and, therefore, suffer from the intrinsic drawback of image replacing.

Dynamic Operating Systems, like Contiki [28], SOS [35], FiGaRo [60], FlexCup [56], TOSthreads [44], and Dynamic TinyOS [61], allow fine-grained code updates with low dissemination and runtime overhead. Nevertheless, most of these solutions present some disadvantages, such as requiring position independent code, which is usually not fully supported by common WSN runtime systems; only allowing one-way linking for loaded modules, consequently leading to the use of more energy-intensive, polling-based services for interrupts; and using nonstandard tools that need to be ported to a wide range of development platforms.

Virtual Machines (VMs), such as COISA [8] and others [5, 6, 12, 16, 18, 32, 36, 46, 75, 79, 83], typically only enable application updates, but reduce the energy cost of propagating a new functionality in the network, given that their code is commonly more compact than native code [61] (due to being processed by software, which allows it to be arbitrarily complex). They also allow developers to implement a program once and deploy it to many different hardware platforms, which can be very useful in heterogeneous settings such as the one expected for the IoT.

Darjeeling [16], simpleRTJ [75], uJ [32], and NanoVM [36] are bare-metal High-Level Language (HLL) VMs that are capable of running on limited hardware environments. While these implementations handle the complexity of Java by only supporting a subset of the Java bytecode, projects like TakaTuka [5] choose to mitigate the disparity between the resources required by Java and the ones available on constrained devices by using bytecode compression and optimization, as well as a compact file format to reduce the memory overhead of .class files.

Although most HLL VMs are restricted to a single programming language, there are some solutions that target a wider range of implementations. One such case is the IBM Mote Runner [18], which is designed to be executed on embedded systems and works with all strictly typed programming languages, such as Java and C#. This VM runs a specialized bytecode, called Mote Runner intermediate language (SIL), and uses a stack-based approach to achieve a more compact implementation. Another example is CILIX [83], which works with the Common Intermediate Language (CIL) and is compatible with many programming languages, such as C++, C#, F#, J++, and Visual Basic.

One more approach that supports different languages is COISA [8]. This VM uses a single intermediate format to encode programs that run on several hardware platforms. By translating the native code to OpenISA, an intermediate language that is as low level as a RISC ISA, it becomes an architecture-neutral solution, thus playing to the strengths

of VM technology in the highly heterogeneous IoT scenario. Furthermore, this tool allows the use of Cloud-Assisted Translation to create binary codes without the overhead of code interpretation, which can greatly reduce the energy required to emulate the code.

Table 2.1 shows a comparison between the features of these virtual machines regarding some of the parameters that are relevant for their execution on constrained devices.

Table 2.1: Comparison between Darjeeling [16], SimpleRTJ [75], uJ [32], NanoVM [36], TakaTuka [5], IBM Mote Runner [18], CILIX [83], and COISA [8]. Modified from [8].

Project	CPU (bits)	RAM (kB)	Flash (kB)	ISA	Focuses on
Darjeeling	8/16	2-10	32-128	Java bytecode	Java
SimpleRTJ	8/16/32	2-24	32-128	Java bytecode	Java
uJ	8/16/32/64	4	60-80	Java bytecode	Java
NanoVM	8	1	8	Java bytecode	Java
TakaTuka	8/16	4	48	Java bytecode	Java
IBM Mote Runner	8/16/32	4	32	SIL	C#, Java
CILIX	8/16	4	32	CIL	C++, C#
COISA	8/16/32/64	2	6	OpenISA	Any (C a.t.m.)

2.2 Fog Computing Tools

Despite the novelty of the fog computing paradigm, several tools that employ it for general-purpose computation, or even specific applications such as data analytics, have appeared since its inception.

Apache Edgent [2] is an open source programming model and runtime that executes parts of analytics solutions on IoT devices to reduce the amount of data they transmit. Instead of sending a continuous flow of possibly trivial data to a server, Edgent analyzes the events at the device, and if they are meaningful, the data is sent to a back-end system for further analysis, action, or storage. A few use cases for this mechanism are analyzing data on distributed edge devices and mobile devices to provide local feedback and reduce data transmission costs, and analyzing machine health and application server error logs in real time without impacting network traffic.

IOx [22] is an application environment created by Cisco that allows the use of docker tools for the development of IoT applications that are independent of the network infras-

tructure. It provides a way to execute these applications on the fog with secure connectivity with Cisco IOS software and it is used by businesses varying from manufacturing and energy corporations to public sector organizations such as cities and transportation authorities.

FogHorn [30] provides a solution for industrial customers, enabling efficient testing of potential edge use cases and on-site delivery of analytics and machine learning. This technology operates in markets such as manufacturing, oil and gas, transportation, smart buildings, renewable energy, and power and water. FogHorn combines services that run on the edge and the cloud, and its edge intelligence platform can be used to prevent costly machine failures or downtime, as well as improve the efficiency and safety of industrial operations and processes in real time.

Parstream [76] is a geo-distributed data analytics platform that creates value for businesses by analyzing a combination of historical and real-time data. As data is now being generated much faster than bandwidth is growing, solutions that store data and query it at its source may become a requirement for data management. An example where this approach is useful is the German railway system, which stores and analyzes data on board of its trains, sending only the result back to the management teams that need it to make decisions. Another instance is Siemens, which used Parstream to locally store the data generated by 5,000 sensors making 100 readings per second on a gas turbine, and then send only the results of the analysis to the Siemens' network.

2.3 General Computation Offloading Schemes

Mathematical models are an effective way to describe the behavior of systems and help users to better understand and optimize system performance. Therefore, many models have been proposed to estimate costs related to computing and transferring data. Nevertheless, these costs are dependent on the execution behavior of the task being considered and the highly-variable performance of the underlying resources. As such, remote execution systems must employ sophisticated prediction techniques that accurately guide computation offloading. These models are even more important for offloading systems, as they may need to decide whether or not it is worth sending a computation to other devices. To this end, many works have been proposed. However, they are usually complex, computationally expensive, or specific for some metric.

Li, Wang, and Xu [50] considered offloading media encoding and compression computation from a small handheld device to a desktop through a wireless Local Area Network (LAN). They used profiling information from applications to construct cost graphs, which were then statically partitioned, and each part was set to run on either the small or the large device. The results showed considerable energy savings by offloading computation following this partition scheme. In another work [51], the authors tested the same infrastructure with benchmarks from the Mediabench suite using a Compaq iPAQ. Their scheme significantly saved energy in 13 of the 17 programs tested. Later, Li and Xu [52] considered the impact of adding security to the offload process for the same infrastructure. They added secure mechanisms such as the Secure Sockets Layer (SSL) in all offloaded wireless

communication and concluded that despite the extra overhead, offloading remained quite effective as a method to reduce program execution time and energy consumption.

Kremer, Hicks, and Rehg [47] presented a prototype for an energy-aware compiler, which automatically generates client and server versions of the application. The client version runs on small devices, offloading computation to the server when necessary. The client also supports checkpoints to allow server progress monitoring and to recover from connection failures. They tested this compiler with multiple programs and mobile devices, obtaining the energy consumption with actual power measurements, and showed that they could save up to one order of magnitude of energy for the small devices.

Rong and Pedram [74] built a stochastic model for a client-server system based on the theory of continuous-time Markovian decision processes and solved the offloaded dynamic-power management problem by using linear programming. Starting with the optimal solution constructed off-line, they proposed an online heuristic to update the policy based on the channel conditions and the server behavior, resulting in optimum energy consumption in the client and outperforming any existing heuristic proposed until the publication of their work by 35%.

Chen *et al.* [20] introduced a framework that uses Java object serialization to allow the offloading of method execution as bytecode-to-native code compilation (just-in-time compilation). This tool takes into account communication, computation, and compilation energies to decide where to compile and execute a method (locally or remotely). As many of these variables vary based on external conditions, the decision is made dynamically as the methods are called. Finally, they tested the framework in a simulator and showed that the technique is very effective at conserving the energy of the mobile client.

O'Hara *et al.* [65] presented a system-wide model to characterize energy consumption in distributed robot systems' computation and communication. With the model, they showed that it was possible to make better decisions about where to deploy each software and how to do the communication between robots. They tested this on a simulator and showed that for a search-and-rescue mission, there are several counterintuitive energy trade-offs. By using their cost model scheme, AutoPower, they were able to improve by up to 57% over the baseline energy consumption.

Unlike other authors, Xian, Lu, and Li [89] chose to offload the computation from one device to another by using a timeout approach instead of analyzing the software statically or dynamically. They set a specific amount of time in which the application would be allowed to run on the client and after a timeout, the program execution was entirely offloaded to the server. They showed that this heuristic works well and that it can save up to 17% more energy than approaches that tried to compute the execution time of the application beforehand.

Hong, Kumar, and Lu [38] showed a method to save energy in mobile systems that perform Content-Based Image Retrieval (CBIR). They proposed three offloading schemes for these applications (local search, remove extraction, and remote search) and physically measured that their approaches were able to save energy on an HP iPAQ hw6945 running CBIR applications.

Gu *et al.* [33] presented a dynamic partition scheme to decide when and which parts of a program to offload to nearby surrogates. They took into consideration both application

execution patterns and resources fluctuations to use a fuzzy control policy model. They ran a large number of tests, measuring the total overhead, the average interaction delay (time to send and receive requested data), and the total bandwidth required. The results showed that their model was effective as an offloading inference engine.

Gurun, Krintz, and Wolski [34] argued that offloading systems must predict the cost of execution both locally and remotely. Moreover, they also say that these techniques must be efficient, that is, they cannot consume significant resources (e.g., energy, execution time), given that they are performed on mobile devices. Thus, the authors proposed NWSLite, a predictor of resource consumption for mobile devices. They empirically analyzed and compared both the prediction accuracy and the cost of NWSLite and a number of different forecasting methods from existing remote execution systems, showing its advantages over the other approaches.

Wang and Li [85] used parametric analysis to deal with the issue of programs having different execution behaviors for different input instances. They proposed a cost analysis for computation and communication of a program that generates a function of the program's inputs. With this strategy, better decisions can be made when partitioning a program and making offload decisions based on the program input.

Wolski *et al.* [88] proposed a method to make offloading decisions in grid settings by using statistical methods. When only considering the network bandwidth of the system, they showed that a Bayesian approach can be superior to prior methods.

Nimmagadda *et al.* [64] showed that offloading mechanisms can even be feasible in real-time scenarios such as real-time moving object recognition and tracking. They considered the real-time constraints when building the offloading decision system and tested motion detection and object recognition using offloading in multiple network and server configurations.

2.4 Edge/Fog/Cloud Computation Offloading Schemes

Jayaraman *et al.* [42] presented equations for a cost model that focuses on energy usage and evaluates the energy gain of their approach when compared to sending all the raw data to the cloud. They also introduced the CARDAP platform, which was implemented for Android devices and uses energy-efficient data delivery strategies to distribute mobile analytics applications among mobile devices on the fog.

Deng *et al.* [25] investigated the trade-off between power consumption and transmission delay in fog-cloud computing systems. Their model was formulated as a workload allocation problem where the optimal workload allocations between fog and cloud intend to minimize power consumption given a constrained service delay. Their aim was to provide guidance to other researchers studying the interaction and cooperation between the fog and cloud.

Xu and Ren [90] analyzed an edge system consisting of a base station and a set of edge servers that depend on renewable power sources. Their models considered workload, power, battery, and delay cost, and they used them to formulate the dynamic offloading and autoscaling problem as an online learning problem to minimize the system cost.

Liu *et al.* [53] used queuing theory to study a mobile fog computing system. Their model was formulated as a multi-objective optimization problem with a joint objective to minimize energy consumption, execution delay, and payment cost by finding the optimal offloading probability and transmission power for each mobile device. They addressed the multi-objective optimization problem with an interior point method-based algorithm.

Neto *et al.* [63] created a framework called ULOOF that takes location awareness into account and uses empirical profiles to estimate the energy consumption and execution time of Android application methods. Their model employs a scaling factor to prioritize either saving execution time or battery charge. Without having to modify the underlying operating system or requiring superuser privileges, their decision engine determines whether to offload a task to an external Mobile-Edge Computing (MEC) server or not.

2.5 Discussion

In Chapter 3, we present our proposed framework, called LMC. In order to compare it to the tools described in Section 2.2, we chose four main characteristics that are relevant to our investigation of code execution on heterogeneous constrained devices (see the constrained devices definition in Appendix A.1). Table 2.2 has the results of this comparison.

Table 2.2: Comparison between our proposed approach (LMC) and the tools Edgent [2], IOx [22], FogHorn [30], and Parstream [76].

	LMC	Edgent	IOx	FogHorn	Parstream
Runs on end devices?	✓	✓	✗	✗	✗
Runs on Class 2 constrained devices?	✓	✗	✗	✗	✗
Supports many programming languages?	✓	✗	✓	✓	✗
Open source?	✓	✓	✗	✗	✗

Given that Edgent is an open-source framework and can be executed on end devices, we chose it as a baseline for the experiments described in Chapter 3.

In Chapter 4, we introduce two models to assist the analysis of the cost trade-off between fog and cloud computing. One of them is a general mathematical model and the other is a visual model inspired by the roofline model introduced by Williams, Waterman, and Patterson [87].

We note that our main goal is different from that of the works mentioned in Sections 2.3 and 2.4, given that we aim to enable users to select the most profitable platform according to a metric of their choosing, while the other studies focus on optimizing specific metrics (mostly time and energy).

Furthermore, the presented mathematical model is intended to be simple so its implementation can be executed on constrained devices, which is not the case for most of the listed models. Although some approaches such as NWSLite [34] also discuss the cost

model needing to be mindful of resource usage, it is still more complex than our approach and too large to be used in the devices that are constrained according to the definition presented in Appendix A.1. We consider this to be an important distinction between our approach and the others, as constrained devices will have a growing importance in the IoT scenario due to their low power consumption and reduced cost, size, and weight.

Chapter 3

A Framework for Custom Code Execution on Constrained Devices

Considering the prediction that there will be tens of billions of devices connected to the IoT in the near future [54], the demand for efficient ways to process data streams generated by sensors grows ever larger, highlighting the necessity to re-evaluate current approaches, such as sending all data to the cloud for processing and analysis.

In this chapter [68], we explore one of the methods for improving this scenario: bringing the computation closer to data sources. By executing the code on the IoT devices themselves instead of on the network edge or the cloud, solutions can better meet the latency requirements of several applications, avoid problems with slow and intermittent network connections, prevent network congestion, and potentially save energy by reducing communication.

To this end, we propose the LibMiletusCOISA (LMC) framework and compare it with Edgent [2], an open-source project that is under development by the Apache Incubator. In our experiments, we used a DragonBoard 410c to execute a simple filter, an outlier detector, and a program that calculates the FFT. Our results indicate that LMC outperforms Edgent when dynamic translation is disabled for both of them and is more suitable for lightweight quick queries otherwise. More importantly, LMC also enables us to perform cross-platform code execution on small, cheap devices that do not have enough resources to run Edgent, like the NodeMCU 1.0.

3.1 Proposed Framework

The two main components of our proposed framework are the LibMiletus [59] library and the COISA [8] VM. Therefore, we named it "LMC", which stands for LibMiletusCOISA.

3.1.1 LibMiletus

The central goal of LibMiletus is to enable developers to easily design and implement IoT devices. To achieve this, the library hides communication specifics so the programmer can focus on device functionalities instead of system infrastructure.

One example of the features included in this tool is the discovery of new devices through mDNS or BLE. It also provides an easy way to describe a device's characteristics using JSON so it can respond to user queries about its traits (e.g., a device with a temperature sensor that returns values as degrees Celsius would contain the property `"state": {"temperature": {"unit": "Celsius"}}`).

Moreover, LibMiletus is compatible with low-cost devices, such as NodeMCU and HM-10 BLE hardware, and it can be easily installed and used through the Arduino IDE, thus being accessible to programmers with little or no experience with embedded systems [59].

3.1.2 COISA

One relevant aspect of enabling the computation to be performed close to the data source is the ability to update software through wireless connections rather than physical access to the devices. An interesting approach for this is the use of VMs, such as Constrained OpenISA (COISA).

COISA is a compact virtual machine that is capable of running guest applications in host platforms with as little as 2 kB of RAM. The version of COISA that we used in our experiments currently emulates OpenISA 0.1, a MIPS-compatible user-level instruction set. OpenISA is an Instruction Set Architecture (ISA) that aims at easy emulation in order to allow a single program to be deployed on many platforms, including microcontrollers. Even though OpenISA has evolved since version 0.1, it remains similar to MIPS in functionality and number of instructions and using the most recent version of OpenISA should also result in compact guest applications and emulation engine, while also emulating guest applications on ARM and x86 host processors at near native performance [8].

3.1.3 Overview

Figure 3.1 shows an overview of the main components of the LMC framework and how they are connected.

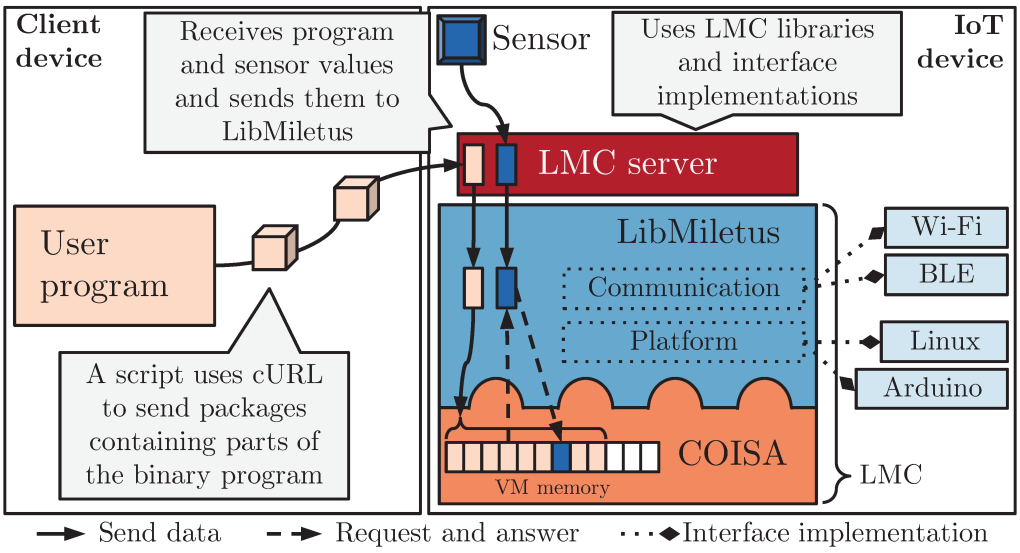


Figure 3.1: Overview of the main components of the LMC framework.

On the left side, we have the client device, which hosts a program written by the user with the help of libraries from the LMC framework. The program is compiled to create a binary file and sent to the IoT device through the network. In our experiments, we use a script that makes simple cURL requests to send this file. As there may be a space limitation on the IoT device's memory, we chose to split the binary file into several smaller packages that can be individually received by the LMC server and then copied to COISA's VM memory until the whole program is stored.

On the right side, we have the IoT device where the LMC server is running. The server program's main function is divided into two parts: setup and event handling.

The first step of the setup consists in creating a new `MiletusDevice` object and defining the properties that identify the device and represent its characteristics (e.g., what type of sensor the device contains) and the methods that allow it to answer requests and execute user programs. Once the object is created, the setup defines its Platform interface, which is responsible for handling the output (e.g., a GNU/Linux Platform interface outputs with `printf`, while an Arduino Platform interface uses `Serial.print`), as well as its Communication interface, which handles the connection using a certain wireless network technology (e.g., Wi-Fi or BLE).

The event handling part is placed inside of an endless loop. It starts by calling the `handleEvents` method from the `MiletusDevice` object, which checks if there are any pending requests. In case there is custom code installed on the device, it is executed until an `exit` syscall is performed. If an event such as a sensor value update happens, the method sends it to the COISA event queue so it can be properly treated by the VM. When the `handleEvents` method returns, the current sensor values are requested and the `MiletusDevice` object is updated with this information (as we used simulated values in our experiments, the implementation of this function simply reads the next value instead of polling the sensors).

In order for COISA to be compatible with other intercommunication frameworks, we kept it completely independent from LibMiletus. On the other hand, we modified LibMiletus to allow access to certain COISA functions and structures, such as COISA's VM memory. This way, when the user program invokes a syscall provided by LMC for reading values, the server event handling process can respond to this request with the current value of a certain sensor by storing it in a place allocated by the user program in the VM memory.

As a final note, we call attention to the fact that LMC is an open-source project[1].

3.1.4 Implementation Details

The development of LMC required us to integrate the LibMiletus library and the COISA VM, while also documenting the capabilities of the new framework.

Although we did not change COISA functionalities, we had to extend the implementation of its instruction set to support single and double precision floating-point operations, which are part of OpenISA, but still needed to be added to the VM. We also included counters to COISA that allowed us to get the statistics required for our experiments.

[1] https://github.com/fpisani/LibMiletusCOISA

We modified LibMiletus to support a new `_coisa` trait and a command that installs the user code on the device that is running the server. This installation process occurs by copying the code to the VM's memory. If a new event happens and there is a code to be executed, we made LibMiletus use COISA for this job.

Furthermore, we included the mechanism that allows the server to respond a reading request from the user program with the current sensor value. This is done by storing the value in a place allocated by the user program in the VM memory. We also added counters to LibMiletus that allowed us to get the statistics required for our experiments. In order to verify our code, we implemented unit tests in C++ that allowed us to guarantee that LibMiletus functionalities were still working after the changes.

We implemented the test cases for our experiments in C and wrote Bash scripts that automated compiling them and testing their functionalities. To verify if the test cases were working correctly with LMC, we created Python scripts to automate the process of starting the LMC server on a computer running GNU/Linux, sending the user program to the server, and then checking the results of the computation. We did the same for the Edgent test cases, but the programming language used to implement them was Java.

3.2 Experiments

In this section, we consider the following scenario to evaluate LMC and compare it with Edgent: a device with processing capabilities is equipped with a capture instrument (e.g., a microphone or a sensor) that periodically measures an environmental variable such as temperature, humidity, or luminosity. A user is interested in querying this device for information that can be useful for synthesizing knowledge. To this end, the user expresses their query as a program, sends it to be executed on the device, and then receives its result. The answer to the user's query can be a log file stored in the device itself, a stream of values sent to the cloud, or any other reply that they may find convenient. For the purpose of these experiments, however, we will simply consider that the answer was reported back to the user and will not look further into the implementation of this procedure.

3.2.1 Test Cases

To explore our test case scenario, we implemented three different operations that a user might want to use to query the device.

MinMax is a simple filter that checks whether a numeric value is outside of a certain interval (Algorithm 1). It can be used to detect malfunctions in systems where the expected upper and lower bounds for collected values are known.

Algorithm 1 Algorithm for the MinMax operation.

Input: $sensor\_value$: floating-point value to be evaluated; min, max: lower and upper bounds for the interval of values that do not need to be reported, respectively.

1: **if** $sensor\_value < min$ **or** $sensor\_value > max$ **then**
2: report_value($sensor\_value$)
3: **end if**

Outlier is a program that executes Tukey's test [37] to detect outliers in a window of stream values (Algorithm 2). It can be used to detect possible anomalies or failures in data models.

Algorithm 2 Algorithm for the Outlier operation.

Input: *sensor_values*: array containing floating-point values to be used in the calculation; *size*: number of values.

insertion_sort(*sensor_values*)
▷ These calculations use the definition of quartile that does not include the median:
$q1$ = find_median(0, $(size/2) - 1$) ▷ The parameters indicate the first and last indices of the array to consider for the calculation
if $size$ % $2 == 0$ **then** ▷ The operator % represents the remainder of the division
    $q3$ = find_median($size/2$, $size - 1$)
else
    $q3$ = find_median($size/2 + 1$, $size - 1$)
end if
$range = q3 - q1$
$inner\_lower\_boundary = q1 - range \times 1.5$
$inner\_upper\_boundary = q3 + range \times 1.5$
$outer\_lower\_boundary = q1 - range \times 3.0$
$outer\_upper\_boundary = q3 + range \times 3.0$
for each $value \in sensor\_values$ **do**
    if $value < outer\_lower\_boundary$ **then**
        report_major_outlier(*value*)
    else
        if $value < inner\_lower\_boundary$ **then**
            report_minor_outlier(*value*)
        else
            if $value > outer\_upper\_boundary$ **then**
                report_major_outlier(*value*)
            else
                if $value > inner\_upper\_boundary$ **then**
                    report_minor_outlier(*value*)
                end if
            end if
        end if
    end if
end for

FFT is a recursive algorithm that calculates the Fast Fourier Transform (Algorithm 3) for a window of stream values. Among other applications, it can be useful as a step in problems such as voice activity detection [19].

Algorithm 3 Algorithm for the FFT operation.

Input: *sensor_values*: array containing the complex numbers to be considered in the calculation (all imaginary parts are equal to zero and real parts are the values read by the sensor); *size*: number of values.

```
1:  out = copy of sensor_values
2:  _FFT(sensor_values, out, size, 0, 1)
3:  report_values(sensor_values)
    ▷ Auxiliary procedure:
4:  procedure _FFT(buf, out, size, first, step)
5:      if step < size then
6:          _FFT(out, buf, size, first, step × 2)
7:          _FFT(out, buf, size, first + step, step × 2)
8:          c = 0
9:          while c < size do
10:             z = −i × π × c/n                ▷ i is √−1
11:             t = e^z × out[c + step]         ▷ e is Euler's number
12:             buf[c/2] = out[c] + t
13:             buf[(c + n)/2] = out[c] − t
14:             c = c + 2 × step
15:         end while
16:     end if
17: end procedure
```

When it comes to stream queries, it is possible to make two different types of requests: *one-time queries*, which consider the data of a snapshot taken from the stream at a certain moment in time, are evaluated only once, and return the answer to the user, and *continuous queries*, which are evaluated continuously as new values arrive and produce answers over time, always reflecting the stream data seen so far [9].

In our experiments, we implemented mechanisms for both one-time and continuous queries using LMC and Edgent. Figure 3.2 shows an overall scheme of how these implementations work using the MinMax operation as an example.

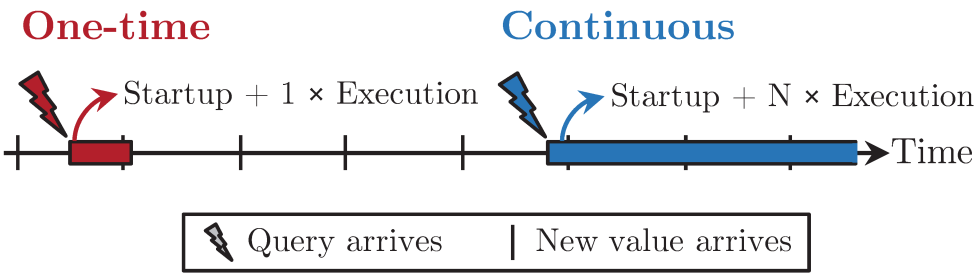


Figure 3.2: Behavior of one-time and continuous queries for MinMax.

In the case of LMC, when the query request is triggered, the corresponding program is sent to the device that is running the LMC server and will execute the query. The user program is then registered as an event handler for sensor readings. In the case of MinMax, whenever an event is delivered to the user program, it requests a value from the sensor and performs the operation. As for Outlier and FFT, the program reads the value and

stores it in an array until *size* values are collected. Then, it runs the query kernel. If the query is of the one-time type, the event handler will be unregistered after the execution, and if it is continuous, the query will keep on being executed until a new request arrives.

We used a similar process for Edgent, except that there is no server running on the device, only the user program written in Java with the help of the framework. Also, Edgent's `TStream<T>` interface offers a `filter` method which produces the same result as MinMax, thus our Edgent program uses this method instead of the algorithm presented above.

3.2.2 Datasets

In order to create the datasets used in our experiments, we simulated a stream of values coming from a sensor by generating random floating-point numbers using the sine function combined with a Gaussian noise. We decided to use Gaussian noise due to having easy access to a working implementation of a function that calculates it. The sine was used to add the characteristic of a wave to the values, similar to what would happen, for example, with temperature variations throughout the day, and the random noise was introduced to create a sufficiently varied dataset. Considering this, we calculated the i-th value of each set as follows:

$$\sin\left(\frac{i+1}{4}\right) \times \text{GaussianFunction}(\text{mean} = 30, \text{std\_dev} = 15)$$

3.2.3 Setup

For our experiments, we selected two devices with distinct characteristics: the development board DragonBoard 410c, which we will simply call "DragonBoard" or "DB" from now on, and the NodeMCU 1.0 (ESP8266, ESP-12E) IoT platform, which will be called "NodeMCU" or "NMCU".

As the DragonBoard can execute both LMC and Edgent, we employed it to compare the performance of these two frameworks in the same setting. The NodeMCU, however, is a less powerful device with more limited resources and was thus used to show that LMC can be executed in devices that do not comply with Edgent's requirement of either Java 7, Java 8, or Android. Table 3.1 presents the hardware and software setup that we used for both devices. Furthermore, we used Arduino 1.8.2 to program for the NodeMCU and the OS Debian GNU/Linux 8.6 (4.4.23-linaro-lt-qcom), Java openjdk-1.8.0_102, and Edgent 1.1.0-incubating for the DragonBoard.

Although the NodeMCU does not belong to the constrained device classes presented in Figure A.1, we verified that the number of resources used by LMC is compatible with C2 devices, as described in Section 3.3.3.

Lastly, we also point out we created the OpenISA binary files that are sent to the LMC server on a computer running Ubuntu 14.04.1 LTS GNU/Linux (4.4.0-81-generic) using the following software: GCC Ubuntu 4.8.4-2ubuntu1 14.04.3 (-O3 flag), ELLCC x86_64-linux-0.1.34, and GNU Binutils 2.24.

Table 3.1: Hardware and software setup for our experiments.

	NodeMCU	DragonBoard
RAM	64 KiB (instruction) + 96 KiB (data)	1 GB LPDDR
Flash	1 MiB + 3 MiB (File System)	8 GB
CPU	Tensilica Xtensa L106 @ 80 MHz	Quad-core ARM® Cortex® A53 Snapdragon 410E up to 1.2 GHz
Compiler	xtensa-lx106-elf-gcc (crosstool-NG 1.20.0) 4.8.2 (-Os flag)	gcc Debian/Linaro 4.9.2-10 (-O3 flag)

3.3 Analysis

We divided our analysis into five parts. Section 3.3.1 evaluates the impact of the filters used in our tests on the number of values reported by the device. We compared the number of host and guest instructions in Section 3.3.2 and used it as an indicator of LMC's interpreter quality. Section 3.3.3 reports the Flash and RAM used by both the LMC and Edgent frameworks for each of the test cases, as these are the values used to determine if they are compatible with the constrained device classification presented in Appendix A.1. Section 3.3.4 analyzes the performance of both frameworks in terms of startup and handler execution time. Finally, Section 3.3.5 considers their performance with regard to the number of events processed per second.

3.3.1 Filter Efficacy

The predicted increase in the number of devices connected to the IoT may lead to the approach of sending all sensor values to the user becoming too taxing due to network congestion or the cost of keeping an ever-increasing storage space. Therefore, as a first experiment, we looked into the impact that executing a filter on the device itself may have on the amount of data that is reported to the user, as this can be a strong motivation for the approach of bringing the computation closer to the data source.

Figure 3.3 shows the results for continuous MinMax queries considering different ranges of values that the user may want to filter out. As expected, we see that for larger ranges such as [-30, 30] (i.e., only values below -30 or above 30 are reported) there is a drastic decrease in the size of the output, reaching as low as only 20% of the stream values being reported. We also observe that a similar behavior occurs even when the query is executed over larger numbers of stream values before being interrupted.

We also executed tests with continuous Outlier queries with window sizes of 5, 10, and 20 and stream lengths of 10, 100, and 1000. In this case, we noted that the maximum number of values that were reported is 15 on a stream of length 1000 using a window of size 10. Most cases reported either zero or one, a likely outcome considering that this filter only lets through abnormal readings.

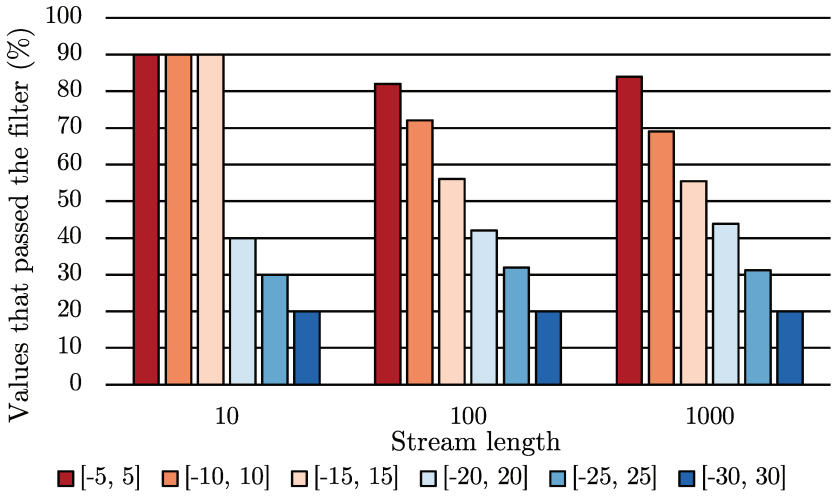


Figure 3.3: Percentage of values that passed the MinMax filter for each range.

3.3.2 Number of Host and Guest Instructions

As an indication of LMC's interpreter quality, we measured the average number of host instructions to emulate one guest instruction on the DragonBoard platform, described in Section 3.2.3. On average, the COISA VM takes approximately 60, 66, and 80 host instructions to emulate one guest instruction when executing the MinMax, Outlier, and FFT, respectively. These results are in line with the ones reported by Auler *et al.* [8].

Due to the lack of APIs to access register counters on the NodeMCU device, we were unable to measure the number of host instructions on this platform. Nonetheless, our manual examination of the interpreter's binary code on both platforms indicates that the result would be similar on the NodeMCU.

3.3.3 Code and Data Size

In this experiment, we measured the amount of Flash and RAM memory required by both Edgent and LMC frameworks. Tables 3.2 and 3.3 show the memory required by the Runtime/OS, the LMC/Edgent framework, and the user's program. The LMC framework consumes 87.0 kB (126.5 kB) of Flash memory to store the framework code on the NodeMCU (DragonBoard) platform, which is compatible with C2 constrained devices. The Edgent framework, on the other hand, uses 287.8 kB of Flash memory to store the framework code on the DragonBoard, which is more than what is available to a C2 device. Moreover, the Edgent framework requires the Java Virtual Machine (JVM), which occupies 128 MB when installed on the DragonBoard platform, going way above what is defined for a C2 device. The LMC framework consumes from 22.8 to 33.6 kB of RAM memory when executing, which is mainly due to the array that is allocated to represent the virtual machine memory. This array has been configured to be 15 kB in size in our experiments. Nonetheless, this value can be adjusted to fit more constrained host platforms.

The LMC MinMax code uses 1.2 kB of memory, being 1129 bytes used by code, 25 bytes by data, and 32 bytes by the program stack. The same code occupies about 5.2 kB

of memory when compiled for the Edgent framework. We were unable to measure the amount of memory required to store the program data and stack when using Edgent.

The Outlier code program occupies 2.0 to 3.0 kB of memory on the LMC platform, being 1.9 kB spent by code, 32 bytes by the program stack, and 64 to 1056 bytes spent by the program data section, which hosts the array that stores the sensor values. The Edgent framework uses roughly 6.9 kB of memory to store the program code.

The FFT code program consumes 12.5 to 14.5 kB of memory on the LMC platform, being 10.4 kB used by code, 2080 bytes by the program stack, and 100 to 2084 bytes spent by the program data section. This user code implements a recursive FFT function, which consumes stack memory. The Edgent framework uses only 7.1 kB of memory to store the program code, however, this code uses the sine and cosine methods available on the Java libraries, while the FFT code for the LMC platform implements these functionalities. In case these functions are removed, the FFT would occupy only 2.6 kB of code and 100 to 946 bytes of data on the LMC platform.

Table 3.2: Comparison between the infrastructure of the LMC and Edgent frameworks.

		Flash (bytes)		RAM (bytes)
		Arduino Runtime / GNU/Linux	LMC / Edgent	Arduino Runtime / GNU/Linux
NMCU	LMC	239,369	87,012	49,912
DB	LMC	-	126,524	88,080,384
	Edgent	-	287,826	88,080,384

Table 3.3: Comparison between the LMC and Edgent frameworks during execution.

			RAM (bytes)			
			LMC / Edgent+JVM	User Prog. Code	User Prog. Data	User Prog. Stack
NMCU	LMC	MinMax	26,258	1,129	25	32
		Outlier	28,258	1,887	64 - 1,056	32
		FFT	36,258	10,367	100 - 2,084	2,080
DB	LMC	MinMax	22,789	1,129	25	32
		Outlier	25,248	1,887	64 - 1,056	32
		FFT	33,578	10,367	100 - 2,084	2,080
	Edgent	MinMax	-	5,237	-	-
		Outlier	-	6,903	-	-
		FFT	173,957	7,112	-	-

3.3.4 Startup and Handler Execution Time

Deploying and executing a user application on LMC implies that the code must be transferred to the IoT device and configured. In LMC, this means that the application is copied to the VM memory and the `main` function is called so the user can register its event handlers. Once registered, whenever a new event is generated, the proper user handler is invoked to process it. In this experiment, we exercise the infrastructure with different stream lengths (16, 256, 4096, and 65,536) and perform a linear regression analysis to derive the time it takes to set the user program up (startup) and the time it takes to execute the user handler. For the sake of simplicity, our implementation of FFT requires the window size to be a power of two, therefore, from this point forward we always used window and stream sizes that are powers of two for our tests.

Figure 3.4 shows the amount of time the LMC and the Edgent frameworks take to execute the user handler on each benchmark configuration. We compare the LMC COISA VM running with interpretation with Edgent running on the JVM with only interpretation by turning off JVM's JIT mechanism. We note that the LMC framework is 2.1 to 4.7 times faster than the Edgent framework. Although interpretation is used, LMC is capable of running the user handler 10,000 to 27,000 times per second on the DragonBoard and 300 to 1500 times per second on the NodeMCU.

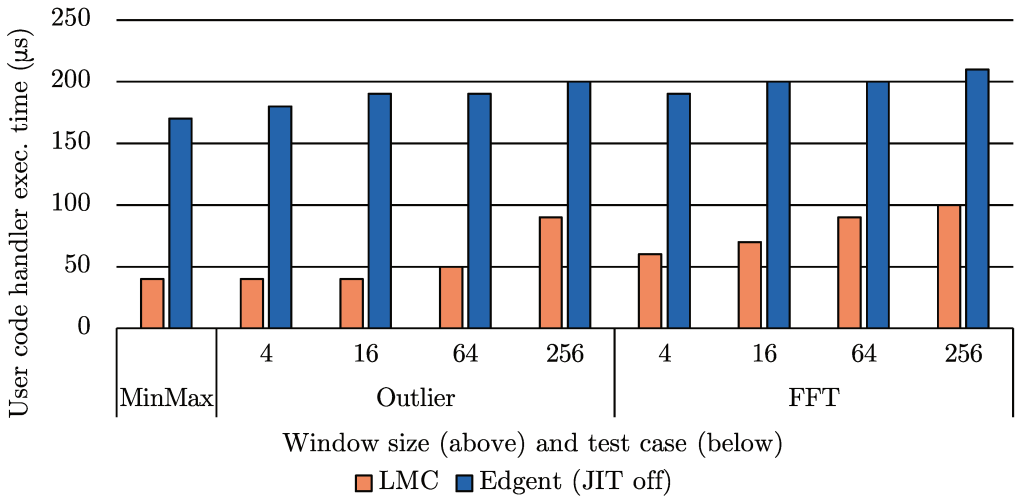


Figure 3.4: User code handler execution time on DragonBoard for LMC and Edgent with JIT off.

In case we turn the JIT mechanism on, the user code is eventually compiled and Edgent's user handler is accelerated by 24 to 56 times. The COISA VM can also employ translation to accelerate the code emulation. The translation mechanism, called Cloud-Assisted Translation (CAT) [8], leverages servers on the cloud to translate the code from OpenISA to the host machine. As this mechanism is not yet implemented in our infrastructure, we can instead estimate the performance that would be achieved by CAT by compiling and running the user handler as native code[2]. Figure 3.5 shows the amount of

[2]As discussed by Auler *et al.* [8], CAT is capable of producing code that achieves near-native performance.

time the Edgent framework and the native binary code take to execute the user handler on each benchmark configuration. Note that there is a large gap between Edgent and native performance (4 to 67 times), which means that even if the code generated by the CAT mechanism was twice slower than native execution, it would still be much faster than Edgent.

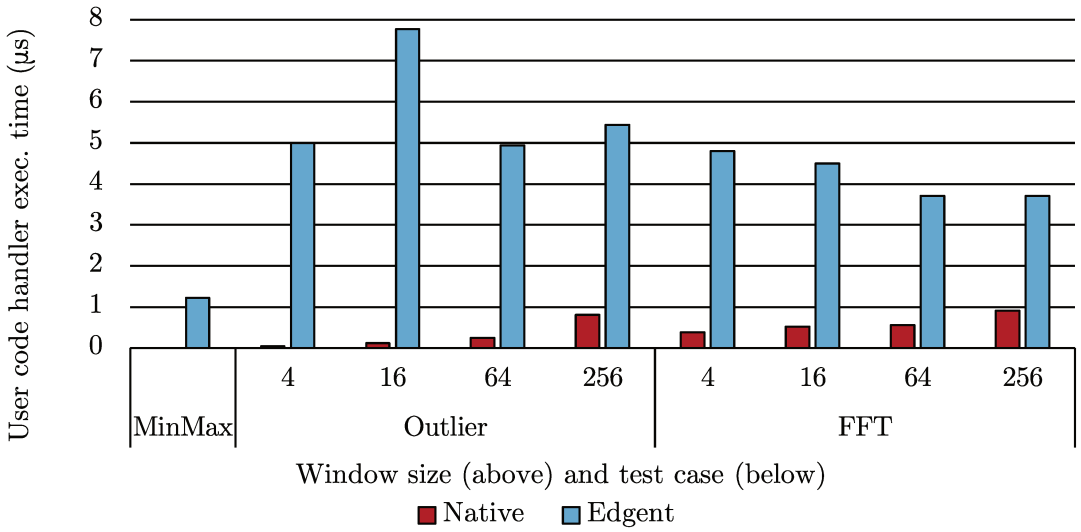


Figure 3.5: User code handler execution time on DragonBoard for Native and Edgent.

Our results also indicate that the LMC framework takes 1 to 5 ms to setup the user code while the Edgent framework (with and without JIT) takes 2200 to 2500 ms to start the JVM and setup the user code, which can impose a high overhead on one-time queries.

3.3.5 Events Processed per Second

Given that being able to execute a query before a new value arrives avoids data losses and is therefore an interesting characteristic for a system, we have that an important measure to define the performance of LMC in comparison to Edgent is query throughput (i.e., the number of user queries that can be executed per second).

In order to evaluate both platforms according to this metric, we executed continuous queries for each test case with increasing stream lengths. For Outlier and FFT, we also measured the impact of using different window sizes, as computations on larger windows not only take longer to be executed but also require more memory in order to store their values.

The numbers reported in this subsection were calculated by the inverse of the running time of each scenario (including the time the system takes to setup the user code). In turn, the execution time was obtained by determining the median between three executions. In the case of Outlier and FFT, the number was also divided by the window size, as we consider that filling up a window and performing the correspondent computation represents one query.

Figure 3.6 shows the results for MinMax queries. First, we see that LMC running on the DragonBoard had the best throughput overall. However, Edgent's performance

is growing linearly as input size increases, almost reaching the same rate as LMC for 65,536 values. This growth in efficiency happens due to JVM's JIT and declines when the dynamic translation is turned off. As expected, the results indicate that Edgent with JIT enabled will outperform LMC in long running queries, but native LMC (or LMC translated with CAT, as argued in Section 3.3.4) still would have the best performance altogether.

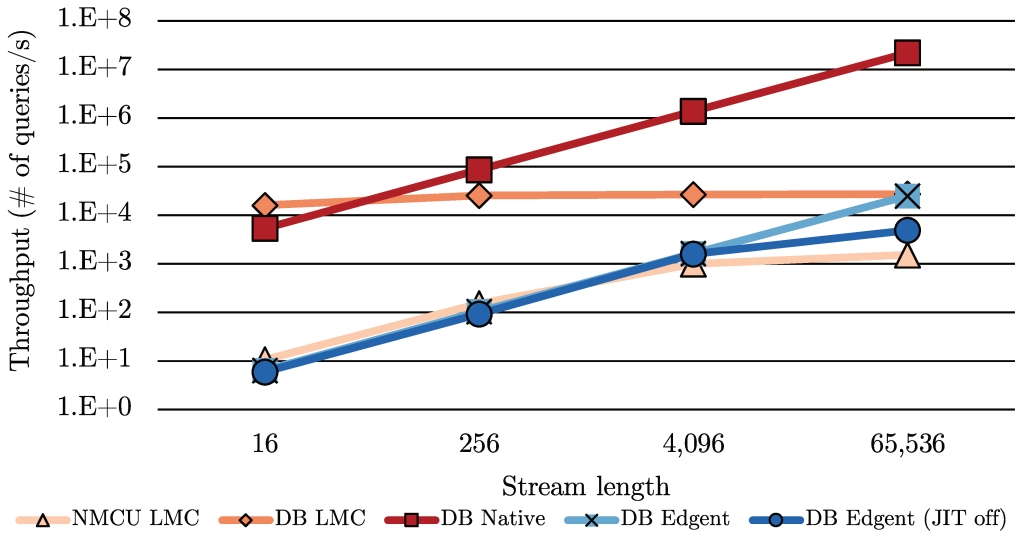


Figure 3.6: Throughput of MinMax queries for different stream lengths.

Another observation that can be made is that LMC running on the NodeMCU outperforms Edgent on the DragonBoard for queries that are running on up to 256 values. This establishes interesting use cases for LMC: for example, a user may want to run a query that lasts less than 5 minutes on a sensor that produces one value per second, or they may want to check if there are any readings outside of the expected range within the next half minute on a sensor that generates 10 readings per second.

Figures 3.7 and 3.8 illustrate the results for the Outlier and FFT applications, respectively. We note that the same trend that was detected for MinMax also happens in these examples, with the exception that for larger window sizes, Edgent's Outlier outperforms LMC on the DragonBoard, and for larger stream lengths Edgent's FFT has the best throughput. All in all, LMC still presents itself as the best choice for quick lightweight queries and would have the best performance as a whole if we consider native or translated code.

3.4 Conclusion

In this chapter, we presented LMC, a framework that enables cross-platform code execution on constrained IoT devices. We validated the framework using three test cases on two IoT platforms, NodeMCU 1.0 and DragonBoard 410c, and showed experimental results that indicate that LMC is: *i)* very compact and compatible with Class 2 constrained devices; *ii)* overall faster than the Edgent framework when they are both statically translated; and *iii)* faster than Edgent at lightweight quick queries when they are both being

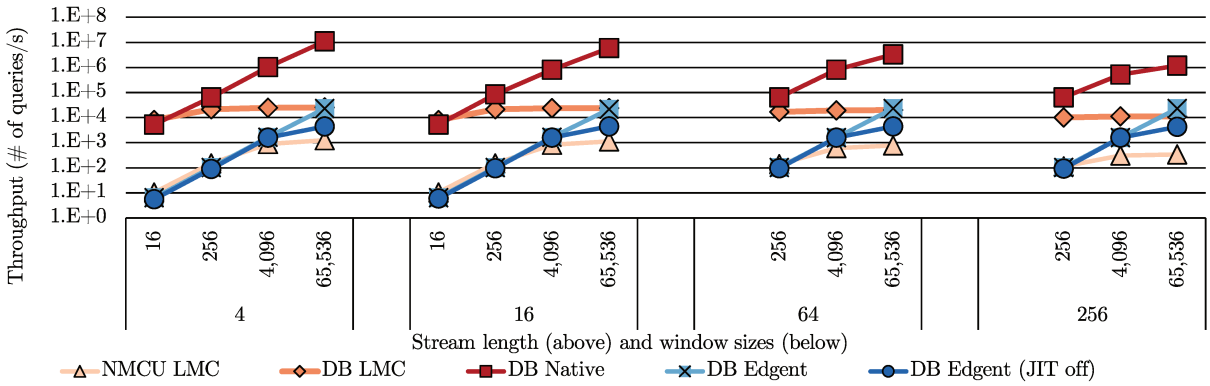


Figure 3.7: Throughput of Outlier queries for different stream lengths and window sizes.

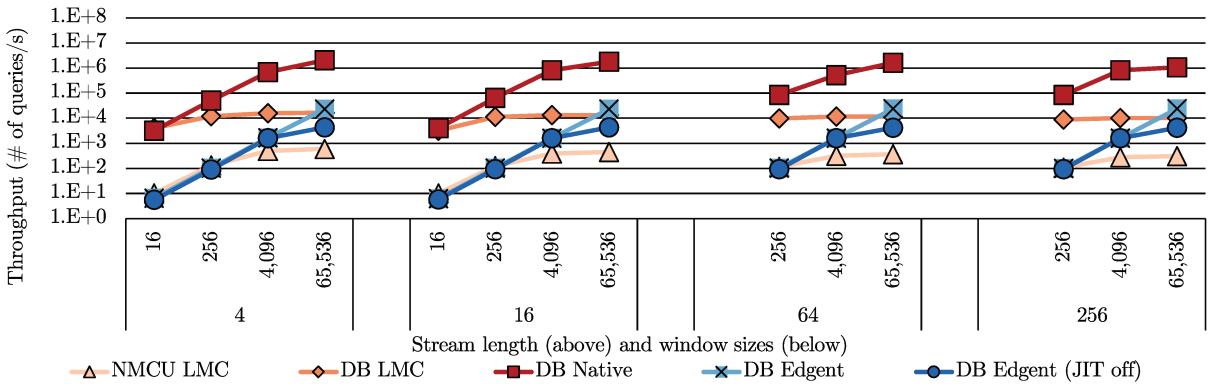


Figure 3.8: Throughput of FFT queries for different stream lengths and window sizes.

interpreted, in some cases, even if LMC is running on the small NodeMCU platform while Edgent is running on the DragonBoard 410c.

Chapter 4

Modeling Cloud and Fog Execution Costs

In this chapter [69], we work toward the answer to the question "is it worth processing a data stream on the device that collected it or should we send it to be processed by more powerful and distant devices?". As it is often the case in Computer Science, the response is "it depends". To find out the cases where it is more profitable to stay in the device (which is part of the fog) or to go to a different one (for example, a device on the cloud), we propose two models that intend to help the user evaluate the cost of performing a certain computation on the fog or sending all the data to be handled by the cloud. In our generic mathematical model, the user can define a cost type (e.g., number of instructions, execution time, energy consumption) and plug in values to analyze test cases. As filters have a very important role in the future of the IoT and can be implemented as lightweight programs capable of running on resource-constrained devices, this kind of procedure is the main focus of our study. Furthermore, our visual model guides the user in their decision by aiding the visualization of the proposed linear equations and their slope, which allows them to find if either fog or cloud computing is more profitable for their specific scenario. We validated our models by analyzing four benchmark instances (two applications using two different sets of parameters each) being executed on five datasets. We use execution time and energy consumption as the cost types for this investigation.

4.1 Modeling Platforms

When working with the data streams generated by sensors, it is common for users to need to transform the raw collected data into something else before storing the final values. For example, they may wish to standardize or clean the data (parsing), discard unwanted readings (filtering), or generate knowledge (predicting).

With the task at hand, they will then need to decide where to execute it, and there are a few options to choose from: they can process the data on the same device that collected the values; they can send the data to other nearby devices, among which the task may be split; or they can send the data to be processed by more powerful and distant devices, such as cloud data centers.

In this dissertation, we discuss the first and third cases. Therefore, our main goal will be working toward the answer to the question presented at the beginning of this chapter. In a way, looking for the answer to this question is akin to a search for cases where fog computing (performing computation close to the data source) is more profitable than cloud computing (sending data to be processed by faraway devices).

For the purpose of this analysis, we consider that we are working with devices that are capable of executing custom code, connected to the Internet, and able to send packets to other devices. Furthermore, we assume that we are handling non-empty data streams of limited size (i.e., the stream size is neither zero nor infinite) and that we are only dealing with cases where there is no overlap between processing and transferring the data.

We chose this last restriction due to the fact that treating concurrent data processing and transferring would require a more complex model, as the cost of performing both events (or part of them) at the same time may not be the same as the sum of the costs of performing them separately. For instance, devices may handle these cases differently from the point of view of enabling/disabling parts of the hardware and buffering the data to be transmitted.

4.1.1 General Equations

First, we look at the generic cost of performing the task on the device itself, which we call fog computing cost ($C_{_F}$). We model this cost with regard to the steps that must be executed to complete the computation. We start by reading the sensor value (cost r). Then, we perform a custom code operation (cost t). We are particularly interested in filtering procedures, given that, as seen in Section 3.3, these can be simple enough to run on resource-constrained devices and have the potential to decrease the amount of data sent to the cloud drastically, a situation that is posed to become a problem with the large-scale adoption of technologies such as the IoT. For that reason, in this investigation, we consider that our custom operations are filters, that is, operations that decide whether or not each stream value should be sent to the cloud. In this case, we must also consider the cost of sending the data to the cloud (s) and the probability that the value will pass the filter in question (f). After completing the operation, the device is then idle until a new reading comes along (cost i). If we are working with a data stream of size z, Equation (4.1) shows the fog computing cost.

$$C_{_F} = (r + t + s \cdot f + i) \cdot z \tag{4.1}$$

Next, we look at the generic cost of performing the task on the cloud. As some of the resources used by the cloud are not under the user's control, from the point of view of the device that is collecting the data, the cloud computing cost ($C_{_C}$) only includes, for each of the z values in the data stream, reading (r), sending (s), and being idle (i), as shown by Equation (4.2).

$$C_{_C} = (r + s + i) \cdot z \tag{4.2}$$

Given that this is a generic model, it is possible to plug in values to calculate different types of costs (e.g., number of instructions, execution time, energy consumed). We will use

this approach in Section 4.3 to analyze a few test cases. We note that we are considering that all of the values are positive, with the exception of f, which is a probability and, as a result, a number between zero and one. Table 1 in the List of Symbols has a summary of the notation introduced in this section.

In order to answer our question, we would like to choose situations where the fog computing cost is less than or equal to the cloud computing cost (we use the fact that processing data locally decreases network traffic congestion as a tiebreaker in favor of fog computing when fog and cloud costs are the same). Thus, in the cases where $C_{_F} \leq C_{_C}$, we have:

$$\begin{aligned}
(r + t + s \cdot f + i) \cdot z &\leq (r + s + i) \cdot z \\
r + t + s \cdot f + i &\leq r + s + i \\
t + s \cdot f &\leq s \\
\frac{t}{s} + \frac{s \cdot f}{s} &\leq \frac{s}{s} \\
\frac{t}{s} + f &\leq 1 \\
\frac{t}{s} &\leq 1 - f \\
1 - f &\geq \frac{t}{s}
\end{aligned} \tag{4.3}$$

Since f is the probability of a value passing the filter, $1 - f$ is the probability of a value being filtered out. Therefore, from Equation (4.3), we have that the probability that a value will be filtered must be greater than or equal to t/s for us to choose fog computing in this cost model.

4.1.2 Estimating f

We can find s and t by measuring the hardware and software infrastructures according to the type of cost for which we are looking. However, f also depends on the data, so it cannot be as easily calculated unless we know all data points in advance. What we can do instead is estimate f by looking at a subset of the values.

We can start by plotting a graph to guide our analysis of the relationship between $C_{_F}$ and $C_{_C}$. In this graph, the horizontal axis represents the fog computing cost as a function of the number of stream values being processed on the fog. Therefore, a case where all z data points are processed on the fog would be represented by a point crossing this axis. The value of this point, which we call $C_{_{F_0}}$, can be calculated by Equation (4.1). In the same way, the vertical axis represents the cloud computing cost from the point of view of the device as a function of the number of stream values being processed on the cloud, and a case where all values are handled by the cloud is represented by a point crossing this axis. The value of this point, which we call C_{C_0}, can be found with the help of Equation (4.2).

We can also add more points to the graph by considering the cases where part of the

data is processed by the fog, while the rest is processed by the cloud. For each additional sensor reading that we process on the fog instead of just sending to the cloud, we add $r + t + s \cdot f + i$ to the value in the horizontal axis and subtract $r + s + i$ from the value in the vertical axis. It is possible to repeat this process until all possibilities are represented in the graph.

Lastly, we draw a line that passes through these points, which will give us the linear equation represented in Equation (4.4).

$$C_C - C_{C_0} = \frac{0 - C_{C_0}}{C_{F_0} - 0} \cdot (C_F - 0)$$

$$C_C - C_{C_0} = -\frac{C_{C_0}}{C_{F_0}} \cdot C_F \tag{4.4}$$

Figure 4.1 shows an example of a graph constructed with the method described in this section. In this graph, $z = 10$ and all the lines have the same values for all variables, except f.

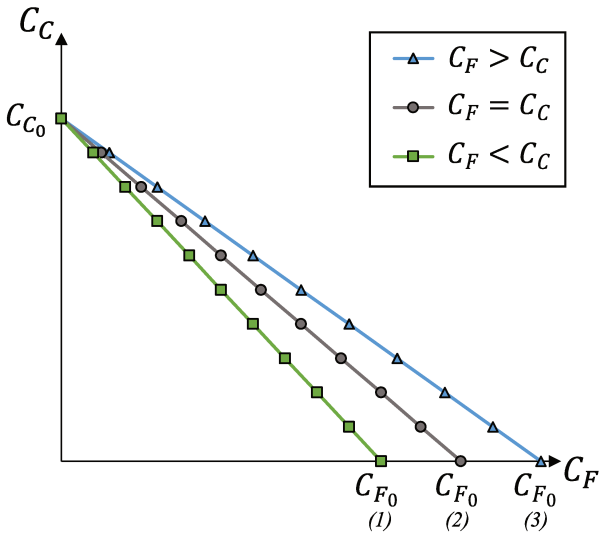


Figure 4.1: Graph of the relationship between fog and cloud computing costs.

Considering that we are estimating f on the device itself while it is collecting and processing the data, we do not need to worry about situations where $C_F \leq C_C$, as the values are processed on the device before we make our decision and nothing will change if fog computing is considered to be more profitable. To evaluate the scenario where $C_F > C_C$, we can think about the worst-case scenario of $f = 1$ (i.e., all data are passing the filter) and determine the size of the penalty that we are willing to pay to be able to make a more informed decision about the value of f. First, we calculate the penalty (p) of processing each additional value on the fog when it would be more profitable to do so on the cloud using Equation (4.5). This value is a percentage of increase in cost.

$$p = \frac{\frac{C_{F_0}}{C_{C_0}} - 1}{z} \tag{4.5}$$

As $C_F > C_C$, $C_{F_0}/C_{C_0} > 1$, and thus, p is always positive. Then, we can establish a

threshold for the maximum penalty that we are willing to pay. For example, if $p = 0.0175$, each additional point processed on the fog will increase the computational cost by 1.75%. Therefore, if we are willing to risk up to a 10% increase in cost, we can test if the first $v = \lfloor 10/1.75 \rfloor = 5$ data points pass the filter or not. In this case, if $n = 3$ out of the five values pass the filter, we can estimate that $f = n/v = 3/5 = 0.6$ and then plug this value into Equation (4.3) to make our decision. It is up to the user to define what is a reasonable trade-off given the parameters of their specific scenario. Table 1 in the List of Symbols has a summary of the notation introduced in this section.

Examining Equation (4.4) and Figure 4.1 also gives us additional insights when we evaluate the slope of the line, which is $-C_{C_0}/C_{F_0}$. When $C_F = C_C$, the slope is -1 (as is the case of the gray line with circular markers). If $C_F < C_C$, the slope is less than -1 (green line with square markers), and for $C_F > C_C$, it is greater than -1 (blue line with triangular markers). From that, we have the following:

$$\begin{aligned} -1 &= -\frac{C_{C_0}}{C_{F_0}} \\ -1 &= -\frac{(r+s+i)\cdot z}{(r+t+s\cdot f+i)\cdot z} \\ 1 &= \frac{r+s+i}{r+t+s\cdot f+i} \\ r+t+s\cdot f+i &= r+s+i \\ t+s\cdot f &= s \end{aligned} \tag{4.6}$$

Considering that all values are positive (or possibly zero in the case of f), we can follow a similar logic for the cases where $C_F < C_C$ and $C_F > C_C$. We see that Equation (4.6) is analogous to Equation (4.3), but in this case, it helps us notice that the values of r, i, and z do not affect the comparison between the slope and -1. Therefore, we can look only at the values of f, s, and t when deciding whether to perform the computation on the fog or the cloud using our cost model. We will use this simplification for our analysis in Section 4.3.

4.2 Experiments

In this section, we put our theory to practice by using real-world data to perform simulations using our mathematical model and then analyze the results with the help of our visual model. In the future, we intend to implement a system that uses our model to choose between processing values on the fog or the cloud so that the user can execute it on the device while it is collecting the data.

4.2.1 Test Cases

We executed our tests using two of the test cases presented in Section 3.2.1. Namely, the MinMax and Outlier benchmarks. We chose four instances of the benchmarks: MinMax

$[-15, 15]$, which filters out numbers in the $[-15, 15]$ range; MinMax $[-5, 5]$, the same program, but using the $[-5, 5]$ range instead; Outlier 16, which finds outliers in a window of 16 values; and Outlier 256, the same program, but with a window of 256 values instead. The parameters of each benchmark were selected due to the results of Section 3.3.1, which showed that these four instances filter out very different percentages of stream values.

4.2.2 Datasets

We executed the benchmarks for five different datasets: four real-world datasets downloaded from the United States National Oceanic and Atmospheric Administration's National Centers for Environmental Information website [62] and the artificial dataset described in Section 3.2.2, which represents a stream of sensor readings. The four real datasets are a subset of the hourly local climatological data collected at Chicago O'Hare International Airport between September 2008 and August 2018 and are named HOURLYRelativeHumidity (HRelHumidity, the relative humidity given to the nearest whole percentage, with values between 16 and 100), HOURLYVISIBILITY (HVisibility, the horizontal distance an object can be seen and identified given in whole miles, with values between 0 and 10), HOURLYWETBULBTEMPC (HWBTempC, the wet-bulb temperature given in tenths of a degree Celsius, with values between -27.3 and 27.4), and HOURLYWindSpeed (HWindSpeed, the speed of the wind at the time of observation given in miles per hour, with values between 0 and 55). We call the artificial dataset Synthetic, and its values are random floating-point numbers between -84.05 and 85.07.

4.2.3 Setup

In our tests, we used the stream size $z = 65{,}536$, which is one of the stream sizes analyzed in Chapter 3, while also being a reasonably large stream size for which our test cases can be executed in a feasible time. Furthermore, as we have the infrastructure employed in that same chapter to measure the execution time of our benchmarks on a NodeMCU device, we used time as one of the cost types for this analysis.

The other cost type is energy consumption, which we obtained in two different ways: the first way was by combining the NodeMCU execution times with electric current and voltage information to calculate the energy consumption of executing the benchmarks on this device; the second way was to use the number of instructions, host per guest instruction, and average cycles per instruction combined with electric current, voltage, and clock rate information to calculate the energy consumption of executing the benchmarks on other devices, such as a Raspberry Pi 3. We chose the Raspberry Pi 3 instead of the DragonBoard 410c for these tests due to it being easier to obtain all the information required for our calculations from the Raspberry Pi 3 datasheet. Section 4.3.3 has a more detailed explanation of these two approaches.

4.3 Analysis

We divided our analysis into four parts. As we need the value of f in order to calculate the fog computing cost, Section 4.3.1 investigates different ways to implement the estimation approach described in Section 4.1.2. Section 4.3.2 then uses the result of the previous subsection to calculate the fog and cloud computing costs using execution time as the cost type. Similarly, Section 4.3.3 presents the costs regarding energy consumption. These two subsections evaluate the efficiency of the chosen estimation technique and how well our model employs it to choose between processing values on the fog or on the cloud. Finally, Section 4.3.4 describes a way to use our model to simulate different scenarios, allowing us to observe how changing the values of our parameters would affect the decision between fog and cloud.

4.3.1 Choosing an Approach to Estimate f

When following the steps presented in Section 4.1.2 to estimate the probability that a value passes the filter (f), we come across the question of whether it would be more profitable to make a decision at the beginning of the stream and then process the values accordingly or to make several decisions along the stream with the intent of accounting for possible changes in data patterns.

In order to evaluate the possibility of making several decisions along the stream, we implemented two different estimation procedures, which we call Local and Cumulative. In both approaches, we first divide our stream into a certain number of blocks (b). As the number of values we are allowed to test (v) is still the same, we can now check $\lfloor v/b \rfloor$ values in the first $b-1$ blocks and $v-(b-1)\times\lfloor v/b \rfloor$ values in the last block. The total number of elements in each block depends on the stream size (z) and can be calculated as $\lfloor z/b \rfloor$ for the first $b-1$ blocks and $z-(b-1)\times\lfloor z/b \rfloor$ for the last one.

In the Local approach, we leverage the information obtained by testing local values in order to try to better estimate f for each block. To that end, we test the values at the beginning of each block and count how many of them passed the filter, then divide this number by the number of tested values. Continuing the example from Section 4.1.2, for $b = 4$, we would be able to check $\lfloor 5/4 \rfloor = 1$ data point in each of the first three blocks and $5-3\times\lfloor 5/4 \rfloor = 2$ data points in the last one. If the number of values that passed the filter (n) was one in each of the first and second blocks, none passed the filter in the third, and one passed it in the fourth, we would have $f = 1/1 = 1$, $f = 1/1 = 1$, $f = 0/1 = 0$, and $f = 1/2 = 0.5$ for each block, respectively.

In the Cumulative approach, we attempt to use the results from all previously-tested blocks in order to make a more informed estimate using a larger number of data points. In this case, we also test the values at the beginning of each block and count how many passed the filter, but we then accumulate the number of both tested and passed values with the count from previous blocks. In our example, that would lead to $f = 1/1 = 1$ for the first block, $f = (1+1)/(1+1) = 1$ for the second block, $f = (1+1+0)/(1+1+1) = 0.67$ for the third, and $f = (1+1+0+1)/(1+1+1+2) = 0.6$ for last block.

Using these two approaches to estimate f, we calculated the costs in terms of execution

time for all benchmarks and datasets using the simplified versions of Equations (4.1) and (4.2) obtained in Section 4.1.2 (i.e., $C_F = t + s \cdot f$ and $C_C = s$). The figures in this subsection present a summary of these results. For the sake of simplicity, we did not include all possible combinations between datasets and applications, as some of the graphs are very similar to the ones shown. All figures employ the same values for stream size (z = 65,536) and custom execution code cost (t, shown in Tables B.1 and B.2 in Appendix B), but Figures 4.2–4.5 use the measured time of 7.3 ms [27] as the cost of sending data to the cloud (s), while Figures 4.6–4.9 show what the costs would be like if s was ten times smaller (0.73 ms). In the first case, the results are fog-prone, that is, the fog is more likely to be profitable, as s is about one order of magnitude larger than t. On the other hand, the results in the second case are cloud-prone, as the cloud is more likely to be profitable when we have close values for s and t.

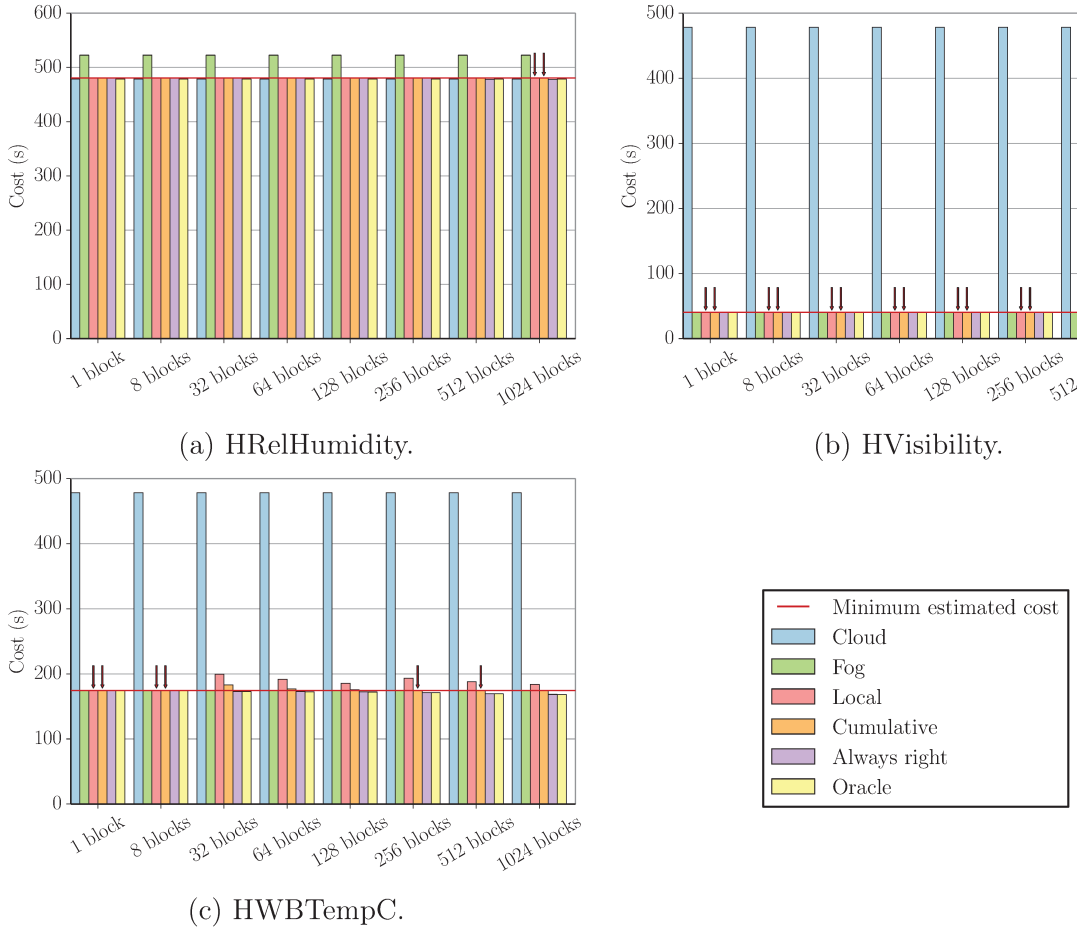
These figures compare the cost of processing all data on the cloud (Cloud); processing all data on the fog (Fog); the cost of using the Local and Cumulative approaches to estimate f and then decide where the data should be processed in each block (Local and Cumulative, respectively); what the cost would be if there was a procedure that always chose correctly between the fog and the cloud after testing the values at the beginning of each block (Always right); and what the cost would be if we knew where to process the data in each block without any testing (Oracle). The line in each graph shows the minimum estimated cost among all the tested numbers of blocks for a certain combination of dataset and benchmark, considering only Local and Cumulative estimates. The arrows point to all Local and Cumulative bars that have the same value as the minimum cost.

We start by analyzing the fog-prone results. The graphs for the MinMax $[-15, 15]$ benchmark present three different sets of characteristics. The first can be seen in Figure 4.2a. Despite being fog-prone, the most profitable choice in this case is the cloud. The division into 1,024 blocks gives us the best estimate, increasing the cost by 0.44% in comparison to sending all values to the cloud. However, we note that the estimate obtained by using one block increases the cost by only 0.50%.

The second case is shown in Figure 4.2b, which is similar to the results for the HWindSpeed and Synthetic datasets (the difference being that the cost values are around 100 ms and 300 ms for these two datasets, respectively). Here, all the divisions result in the same estimate for both the Local and Cumulative approaches. For HWindSpeed and Synthetic datasets, the Cumulative approach results in the same estimate for all values, with Local starting to have increasingly worse estimates with 256 blocks and 512 blocks, respectively.

The third case is the one depicted in Figure 4.2c. Although the Cumulative approach shows good estimates for a higher number of blocks, we see that both Local and Cumulative obtain the minimum cost value for one and eight blocks, as well.

The graphs for MinMax $[-5, 5]$ show two sets of characteristics. Figure 4.3a illustrates the first one, which is a similar result to that of Figure 4.2a, and Figure 4.3b depicts the second, which is akin to the graphs for the HVisibility, HWBTempC, and Synthetic benchmarks. In this case, dividing the stream into only one block results in the minimum cost value for all benchmarks except HWBTempC, for which the Cumulative approach reduces the cloud cost by 22.20% when dividing by 128 blocks, compared to 22.03% for just one block.

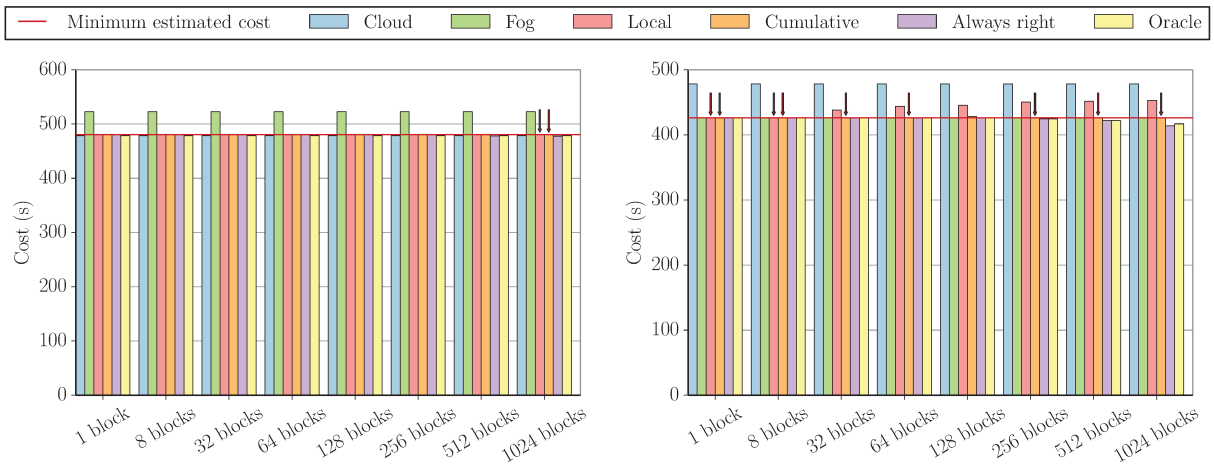


(a) HRelHumidity.

(b) HVisibility.

(c) HWBTempC.

Figure 4.2: Results summary for MinMax $[-15, 15]$ fog-prone test cases.



(a) HRelHumidity.

(b) HWindSpeed.

Figure 4.3: Results summary for MinMax $[-5, 5]$ fog-prone test cases.

All graphs for the Outlier 16 benchmark are similar to Figure 4.4, with every estimate resulting in the minimum cost apart from the Local approach using 1,024 blocks (for the Synthetic benchmark, even this case results in the minimum cost).

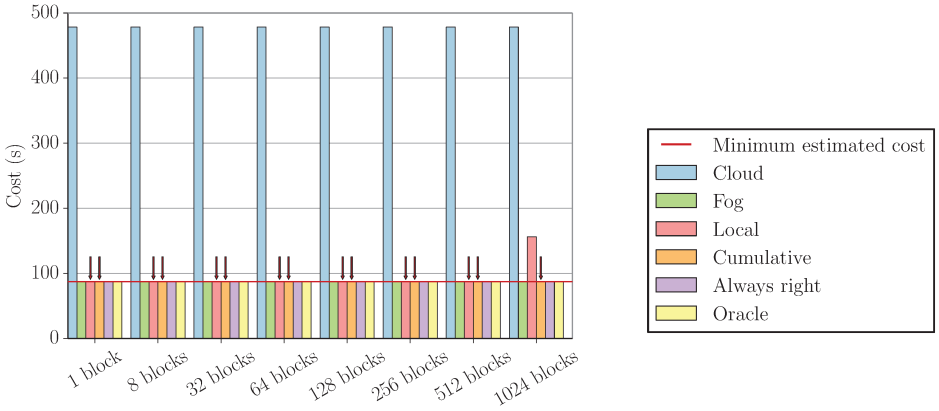
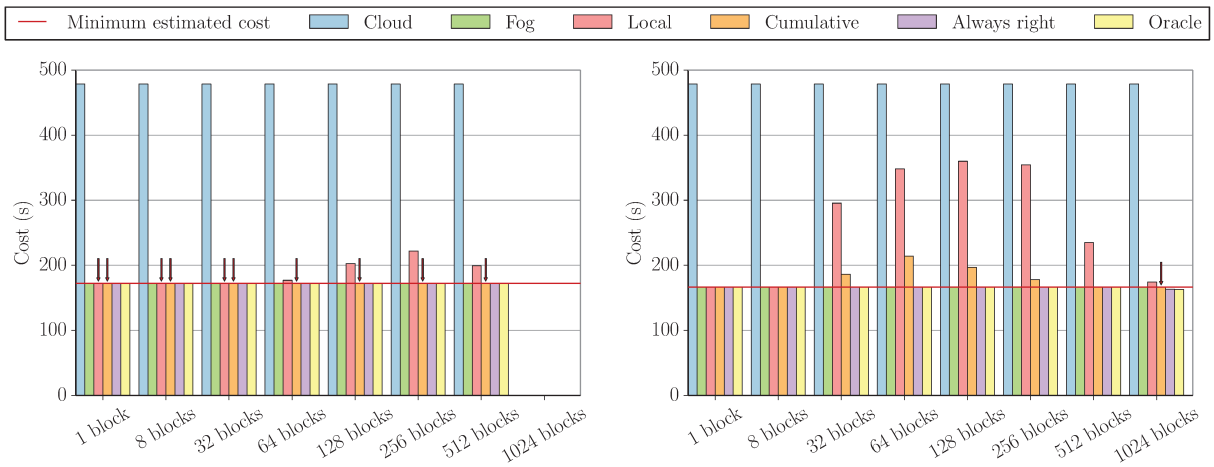


Figure 4.4: Results summary for Outlier 16 fog-prone test cases (HVisibility).

For Outlier 256, most graphs look like Figure 4.5a, where dividing by up to 32 blocks in both approaches results in the minimum cost. Unlike the other datasets, HVisibility allows us to test the division by 1,024 blocks, as seen in Figure 4.5b, given that the number of values we can test in this case is larger than 1,024, and therefore there is at least one value per block.



(a) HRelHumidity.

(b) HVisibility.

Figure 4.5: Results summary for Outlier 256 fog-prone test cases.

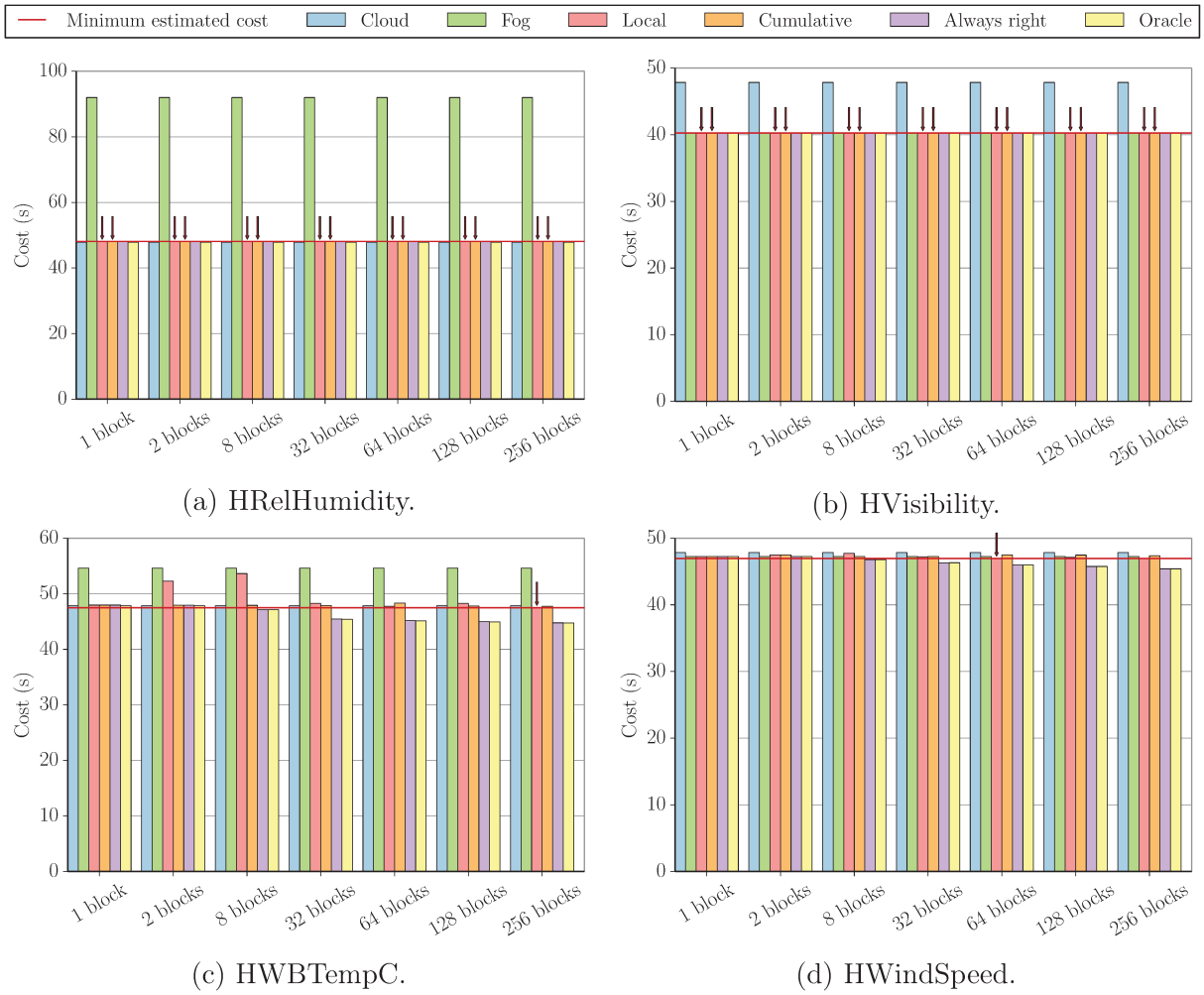
Analyzing these results, we observe that in most cases, changing the number of blocks or the estimation procedure only affects the costs by a slight margin, with the difference between the fog and cloud computing costs being much more prominent.

We now look at the cloud-prone results. Figure 4.6a has the results for the MinMax $[-15, 15]$ benchmark running on the HRelHumidity dataset. All estimates result in the minimum cost, which is an increase of 0.5% over the cost of processing all data on the cloud.

While the results shown in Figure 4.6b are cloud-prone, we see that in fact, the fog is more profitable when executing the MinMax $[-15, 15]$ benchmark on the HVisibility dataset, with all estimates having approximately the same cost as processing all values on the fog (less than a 0.01% difference).

Figure 4.6c also shows one of the datasets for the MinMax $[-15, 15]$ benchmark, HWBTempC. This result is somewhat similar to that of the Synthetic dataset in the sense that the cloud is more profitable in both cases; the Local approach presents a few estimates with high cost (with two and eight blocks for HWBTempC and with 128 and 256 blocks for Synthetic), and only one estimate has the lowest cost value (Local with 256 blocks for HWBTempC and Cumulative with 64 blocks for Synthetic).

Figure 4.6d displays the final dataset for the MinMax $[-15, 15]$ benchmark. Similar to Figure 4.6b, this test also presents a lower cost for processing all data on the fog. However, the difference between cloud and fog costs is much smaller in this case, and the best estimate (Local with 64 blocks) reduces the cost of processing all data on the fog by 1.85%.



(a) HRelHumidity.

(b) HVisibility.

(c) HWBTempC.

(d) HWindSpeed.

Figure 4.6: Results summary for MinMax $[-15, 15]$ cloud-prone test cases.

Figure 4.7 illustrates the results for MinMax $[-5, 5]$, which are similar for all datasets. In this benchmark, processing all the data on the cloud costs much less than doing so on

the fog, and the Local approach usually yields higher estimates for several block numbers (with the exception of HRelHumidity, where all estimates are the same in both approaches). Using only one block results in the minimum cost for the HRelHumidity and HWindSpeed datasets, but the difference in cost of using only one block when compared to the minimum cost in the other datasets is only 0.03%, 0.23%, and 0.01% for HVisibility, HWBTempC, and Synthetic, respectively.

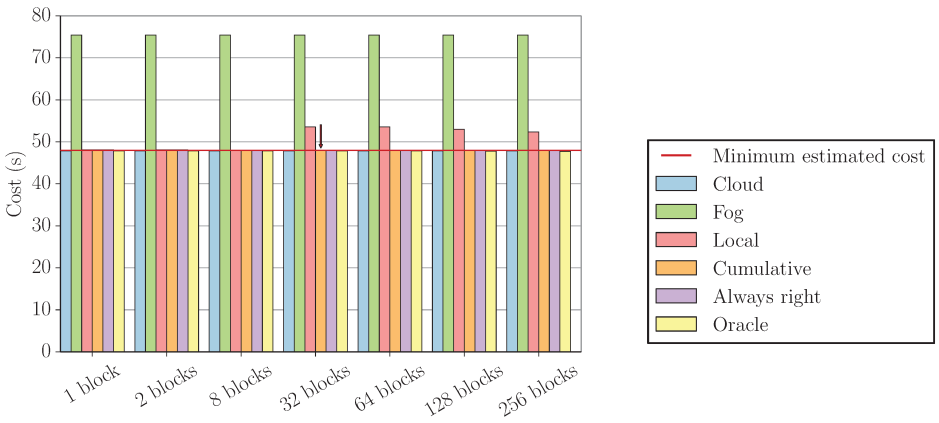


Figure 4.7: Results summary for MinMax $[-5, 5]$ cloud-prone test cases (HWBTempC).

As we can see in Figure 4.8, the values for the cost of processing all data on the cloud and on the fog are very similar for Outlier 16. This is the case for all datasets in this benchmark, but the minimum value is achieved by different numbers of blocks in distinct datasets. For HRelHumidity and HWBTempC, the minimum is obtained by using one block; for HWindSpeed, two; for Synthetic, eight; and for HVisibility, 256. Both Local and Cumulative approaches present the same results for the minimum cost case in all datasets except HVisibility, where the Local approach presents a lower cost.

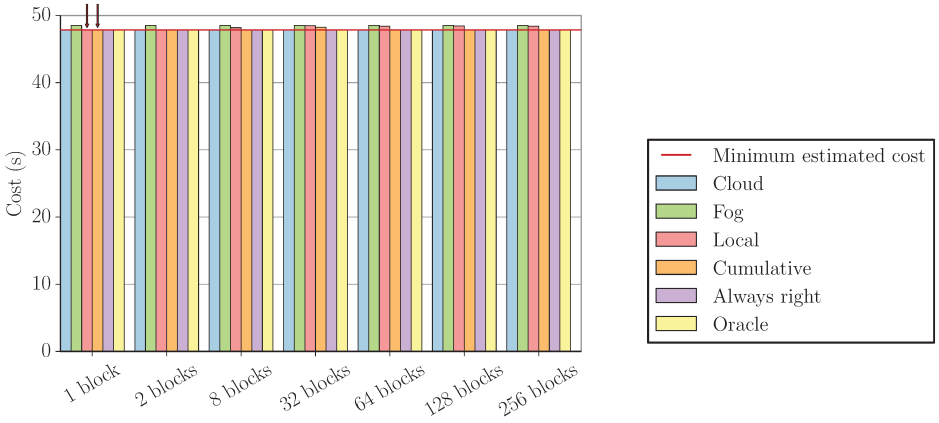
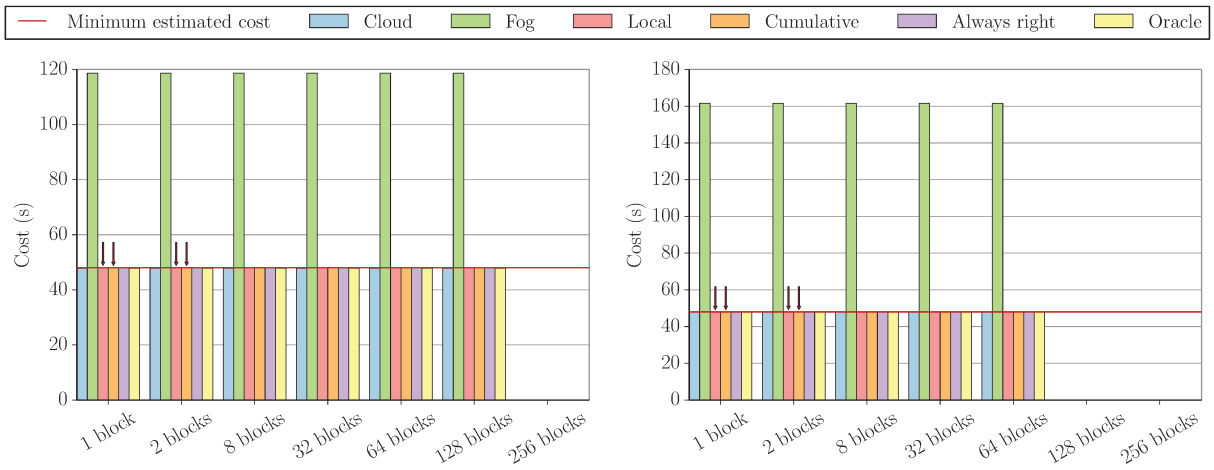


Figure 4.8: Results summary for Outlier 16 cloud-prone test cases (HWBTempC).

Similarly to Figures 4.5a,b, most of the graphs for the Outlier 256 benchmark apart from the dataset HVisibility (Figure 4.9a) look like the one in Figure 4.9b. Again, this is due to the fact that only the HVisibility dataset allows us to test at least one value per block, in this case for 128 blocks. In all cases, dividing by one or two blocks results

in the minimum cost for both the Local and Cumulative approaches. For HWBTempC, dividing by eight blocks also yields this result.



(a) HVisibility. (b) HWindSpeed.

Figure 4.9: Results summary for Outlier 256 cloud-prone test cases.

Again, we can see that in most cases, changing the number of blocks or the estimation procedure does not have a large impact on the cost, and simply choosing correctly between the fog and the cloud will lead to most of the performance gains.

Therefore, this evaluation allows us to conclude that using the straightforward solution of looking at the first values of the stream (that is, when there is only one block) leads to good results in terms of cost for several test cases. Furthermore, in the instances where this is not the best approach, the increase in cost is very small, not justifying the use of more complex division methods. Considering this, in the following subsections, we will look at the values at the beginning of the stream to estimate f.

4.3.2 Deciding between Fog and Cloud for Execution Time

As seen in Section 4.1, we only depend on the variables s, t, and f for our calculations, so we start by finding these values. Tables B.1 and B.2 in Appendix B show the values for t in milliseconds measured for each benchmark and dataset, and we use the measurement of $s = 7.3\,\text{ms}$ [27] in all fog-prone test cases and $s = 0.73\,\text{ms}$ for the cloud-prone ones. Having these values, we can now calculate C_{F_0} and C_{C_0} for the worst-case scenario from the fog point of view, that is, when $f = 1$.

We are considering that $z = 65{,}536$, so we can use Equation (4.5) to determine the penalty for processing a value on the fog when it would cost less to do so on the cloud (p). Given our large stream size, the penalty for each value is very small in fog-prone cases (less than 0.001%) and still relatively small in cloud-prone ones (less than 0.01%), as can be seen in Tables B.1 and B.2 in Appendix B. This difference is due to the fact that values for s and t are closer to each other in our cloud-prone cases.

Therefore, we can look at a reasonable number of values and still have a very low increase in cost. We decided that we are willing to have a maximum increase in cost of 0.5%, so that enables us to test over 3,000 values for Outlier 16 and both MinMax

cases, over 900 values for Outlier 256 for fog-prone cases, and ten times fewer values for cloud-prone cases (the exact numbers are also displayed in the aforementioned table). By examining the output of the benchmarks, we are able to count the values that passed the filter (n), which leads us to the f estimates, as well as the real f values for comparison. Again, all of these results are reported in Tables B.1 and B.2.

Finally, we use the calculated f values to plot a graph similar to the one in Figure 4.1 for each benchmark. Like Section 4.3.1, here, we also used the simplified versions of Equations (4.1) and (4.2) (i.e., $C_{_F} = t + s \cdot f$ and $C_{_C} = s$). The result is illustrated in Figure 4.10. The continuous lines represent the tests that use the estimated values of f (Est.), and the dashed lines represent the tests that use the real values of f (Real). The continuous thick gray line represents the case where $C_{_F} = C_{_C}$, that is, a line with a slope of -1. By observing the graphs, we can determine that most of the lines are below the $C_{_F} = C_{_C}$ threshold for fog-prone cases and above it for cloud-prone cases, as expected. As discussed in Section 4.1, the lines below the threshold mean that it is more profitable in terms of execution time to run these filters in the device that is collecting the data instead of sending the values to be processed on the cloud. On the other hand, lines above the threshold mean that, from the point of view of the device, it is more profitable to process these data on the cloud.

A few notable exceptions are the MinMax benchmarks being executed on the HRelHumidity dataset in the fog-prone scenario and the MinMax $[-15, 15]$ benchmark being executed on the HVisibility and HWindSpeed datasets on the cloud-prone scenario. In the first case, we see that although it was more likely that the fog would be more profitable, the lines are above the threshold, indicating that in fact the cloud is the correct choice. This can be explained by looking at the values of f for the MinMax benchmarks on the HRelHumidity dataset, which are equal to one, meaning that all values passed the filter. We can see that the cloud is more profitable every time this happens, as $C_{_F} = t + s \cdot 1$ is always larger than $C_{_C} = s$. In the second case, the lines are below the threshold, indicating that the fog is more profitable instead of the cloud. This can again be explained by looking at the values of f. MinMax $[-15, 15]$ has $f = 0$ for HVisibility and $f = 0.1393$ for HWindSpeed. Using the values of s and t in each case, we can calculate what f would be when $C_{_F} = C_{_C}$. For HVisibility, $0.614381 + 0.73 \cdot f = 0.73$, leading to $f = 0.1584$. For HWindSpeed, $0.619163 + 0.73 \cdot f = 0.73$ and $f = 0.1518$. The f values for both test cases are below the f value for the threshold, indicating that the fog is indeed the correct choice for them.

We can also calculate the values of the slopes in order to verify the accuracy of our f estimation. We do so by determining the slope of each of the lines using both the estimated and real f values. The slopes obtained, as well as the error of the estimated values in comparison to the real ones, are shown in Tables B.1 and B.2 in Appendix B. In 16 out of the 20 fog-prone cases and in 16 out of the 20 cloud-prone cases, the error is less than 5% in our predictions, thus we are able to get a close estimate of f while only risking an increase of up to 0.5% in the processing cost. Furthermore, it is worth noting that while the eight remaining test cases present larger errors, the choice between fog and cloud is the correct one in all of them.

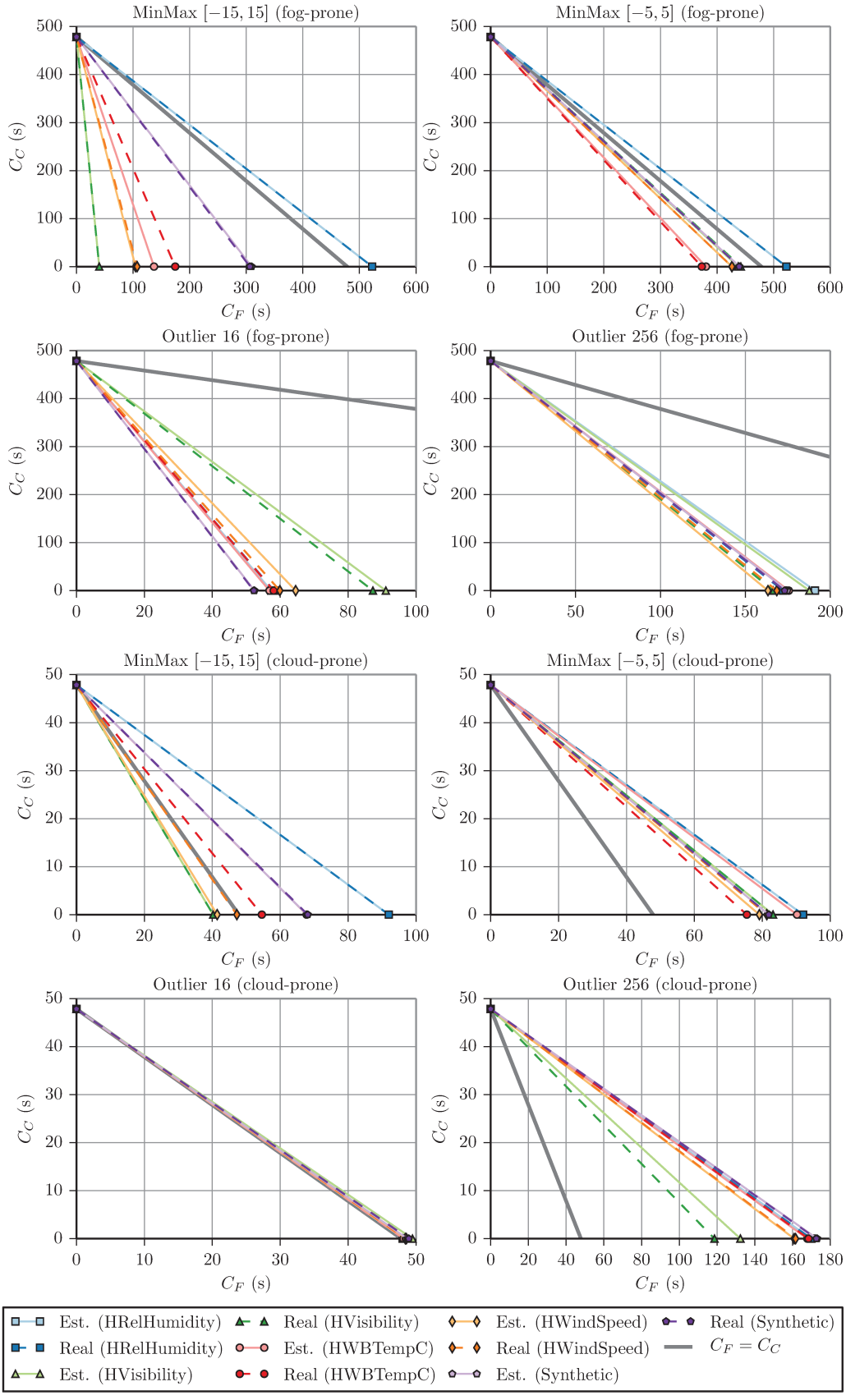


Figure 4.10: Graphs of the linear equations for each dataset and benchmark using execution time as the cost. Est., estimated.

4.3.3 Deciding between Fog and Cloud for Energy Consumption

Similarly to what we did in the previous subsection, we start by obtaining values for s, t, and f. However, we will now use energy consumption as our cost type and evaluate our test cases on two different devices, namely NodeMCU and Raspberry Pi 3.

For the NodeMCU, we calculate t by taking the execution time for each test case (displayed in Tables B.1 and B.2 in Appendix B) and multiplying it by the voltage of the device (2 V) and the electric current for when no data are being transmitted (17.2 mA) [58], resulting in the values reported in Tables B.3 and B.4. We use the same method to calculate the values of s, multiplying the time it takes to send the data to the cloud (7.3 ms for fog-prone cases and 0.1825 ms for cloud-prone cases) by the voltage and by the electric current for when data are being transmitted (70 mA) [58], producing $s = 1.022$ mJ for fog-prone cases and $s = 0.025\,55$ mJ for cloud-prone ones. The next step is to calculate C_{F_0} and C_{C_0} for $f = 1$, the worst-case scenario from the fog point of view.

Given $z = 65{,}536$, we use Equation (4.5) to determine the penalty for processing a value on the fog when it would cost less to do so on the cloud (p). Tables B.3 and B.4 have all the values for p, which are less than 0.0002 for fog-prone scenarios and less than 0.006 for cloud-prone scenarios. We again use 0.5% as the limit for the increase in cost that we are willing to pay to estimate the value of f, which allows us to test over 13,000 values for most fog-prone cases (with the exception of Outlier 256, where v ranges between 3,600 and 5,700, as this benchmark executes a more costly procedure in comparison to the others) and over 320 values for most cloud-prone ones (here, the values of v range between 90 and 150 for Outlier 256), as can been seen in Tables B.3 and B.4.

The last step is to estimate f and plot the resulting linear equations using the simplified versions of Equations (4.1) and (4.2) (i.e., $C_F = t + s \cdot f$ and $C_C = s$), as illustrated by Figure 4.11. From Tables B.3 and B.4, we see that the slope estimate error is less than 5% in 13 out of the 20 fog-prone cases and in 16 out of the 20 cloud-prone ones, which again is a close estimate for risking only up to 0.5% increase in the processing cost. Moreover, from the 11 remaining test cases, 10 correctly choose the more profitable approach. We see that these results are analogous to the ones obtained in Section 4.3.2, with the expected choice being made on most fog- and cloud-prone cases and the same four test cases appearing as exceptions (fog-prone MinMax $[-15, 15]$ and MinMax $[-5, 5]$ executed on HRelHumidity and cloud-prone MinMax $[-15, 15]$ on HVisibility and HWindSpeed).

One interesting difference in the cloud-prone Outlier 16 cases is that as all lines are very close to $C_F = C_C$. In this type of situation, it is necessary to check the slope values to get a more accurate view of the decisions being made. From Table B.4, we have the following slope estimates (and real values): for HRelHumidity, -1.0070 (-1.0128); for HVisibility, -0.9836 (-1.0105); for HWBTempC, -1.0106 (-1.0032); for HWindSpeed, -1.0019 (-1.0048); and for Synthetic, -0.9990 (-0.9945). From that, we have that the fog is chosen for HRelHumidity, HWBTempC, and HWindSpeed, with the cloud being chosen for the other two datasets. However, in the case of HVisibility, the real slope value tells us that the best choice would have been processing the values on the fog. Even so, considering how close the fog (0.025 29 mJ) and cloud (0.025 55 mJ) costs are in this case, choosing the less profitable option will not greatly affect the performance of the system.

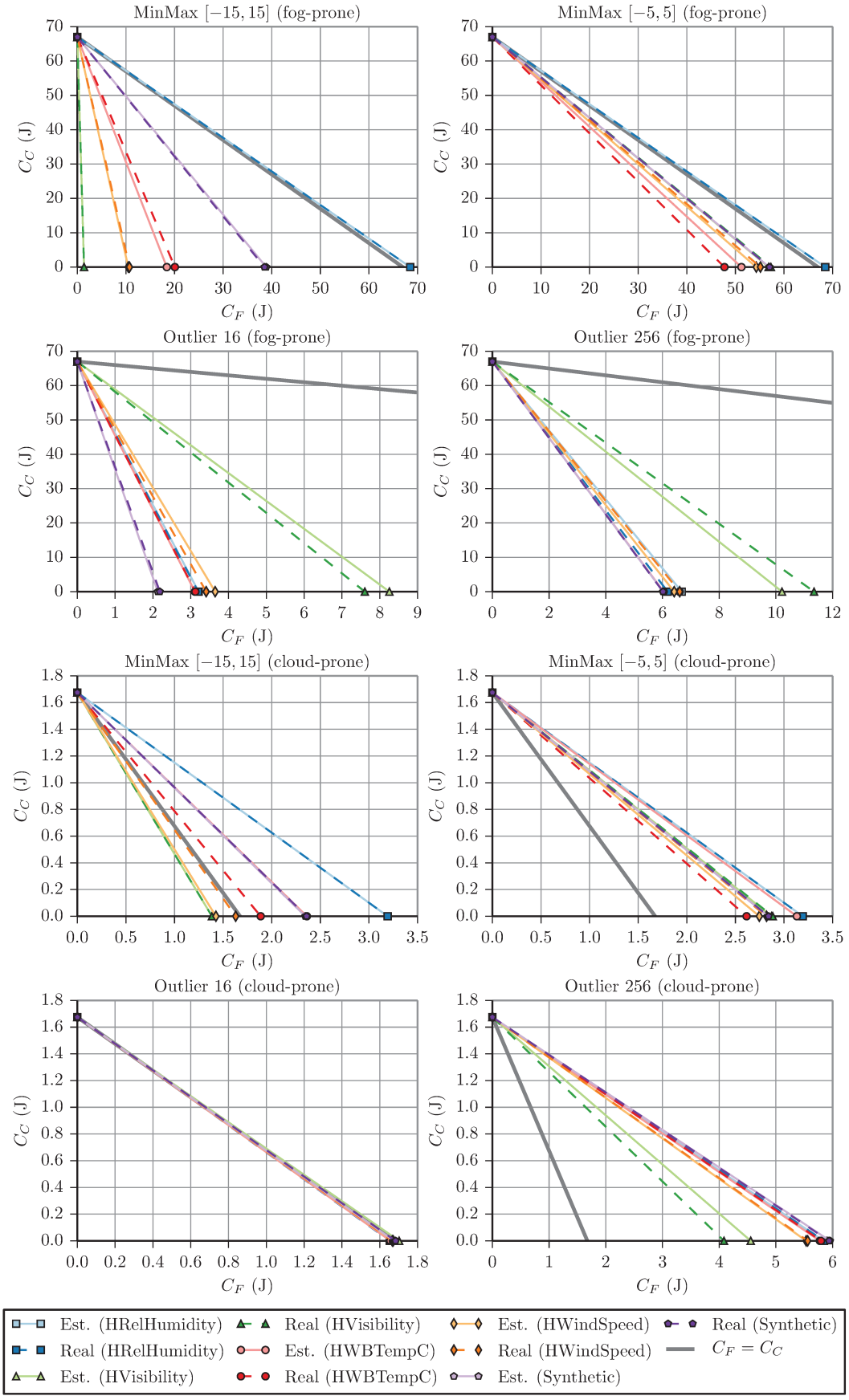


Figure 4.11: Graphs of the linear equations for each dataset and benchmark using energy consumption as the cost: NodeMCU.

Instead of also executing our test cases on a Raspberry Pi 3 device, we employ a different approach to calculate the value of t. First, we use the infrastructure presented in Chapter 3 to count the number of instructions for each test case (displayed in Tables B.5 and B.6 in Appendix B). As our infrastructure is a virtual machine, we then multiply this number by the number of host per guest instructions (60 for MinMax and 66 for Outlier, as seen in Section 3.3.2) and by the average number of cycles per instruction (which we estimate to be one). This gives us an estimate of the number of cycles that each test case would take to process the whole data stream on the Raspberry Pi 3. We then divide this result by the clock rate (1.2 GHz [26]) to find out the time each test case takes to process all stream values and by the stream size (z = 65,536) to finally obtain the execution time of each test case. We then proceed with the same method that we used for NodeMCU to obtain the t values reported in Tables B.5 and B.6. In this device, the voltage is 3.3 V, and the electric current for when no data are being transmitted is 330 mA.

We determine the value of s in the same way as we did for NodeMCU, that is, by multiplying the time it takes to send the data to the cloud (7.3 ms for fog-prone cases and 0.001 825 ms for cloud-prone cases) by the voltage and by the electric current for when data are being transmitted (500 mA) [26], which gives us s = 12.045 mJ for fog-prone cases and s = 0.003 011 25 mJ for cloud-prone ones.

After that, we calculate C_{F_0} and C_{C_0} for $f = 1$ and use Equation (4.5) to determine the penalty for processing a value on the fog when it would cost less to do so on the cloud (p, which can be found in Tables B.5 and B.6). In the fog-prone cases, p is less than 0.00001, and in the cloud-prone cases, it is less than 0.03. This time, we use 0.01% as the limit for the increase in cost that we are willing to pay to estimate the value of f in the fog-prone cases and 1% as the limit in cloud-prone cases. This is done due to the fact that we have very small values for p in the former and larger values for p in the latter. With these limits, we can test over 11,000 values for most fog-prone cases (with the exception of Outlier 256, where v ranges between 1,700 and 2,900) and over 290 values for most cloud-prone ones (here, the values of v range between 40 and 80 for Outlier 256), as indicated in Tables B.5 and B.6.

Finally, we estimate f using the simplified versions of Equations (4.1) and (4.2) to plot the graphs in Figure 4.12. Tables B.5 and B.6 show us that 10 out of the 20 fog-prone cases and 16 out of the 20 cloud-prone ones have a slope estimate error of less than 5%. Although there are four fog-prone scenarios where the error is higher than 30% and three cloud-prone scenarios where it is higher than 15%, the most profitable option between cloud and fog is chosen in all cases.

Like the previous cases, most fog-prone and cloud-prone tests result in the expected choice, with the exception of fog-prone MinMax $[-15, 15]$ and MinMax $[-5, 5]$ executed on HRelHumidity and cloud-prone MinMax $[-15, 15]$ on HVisibility and HWindSpeed.

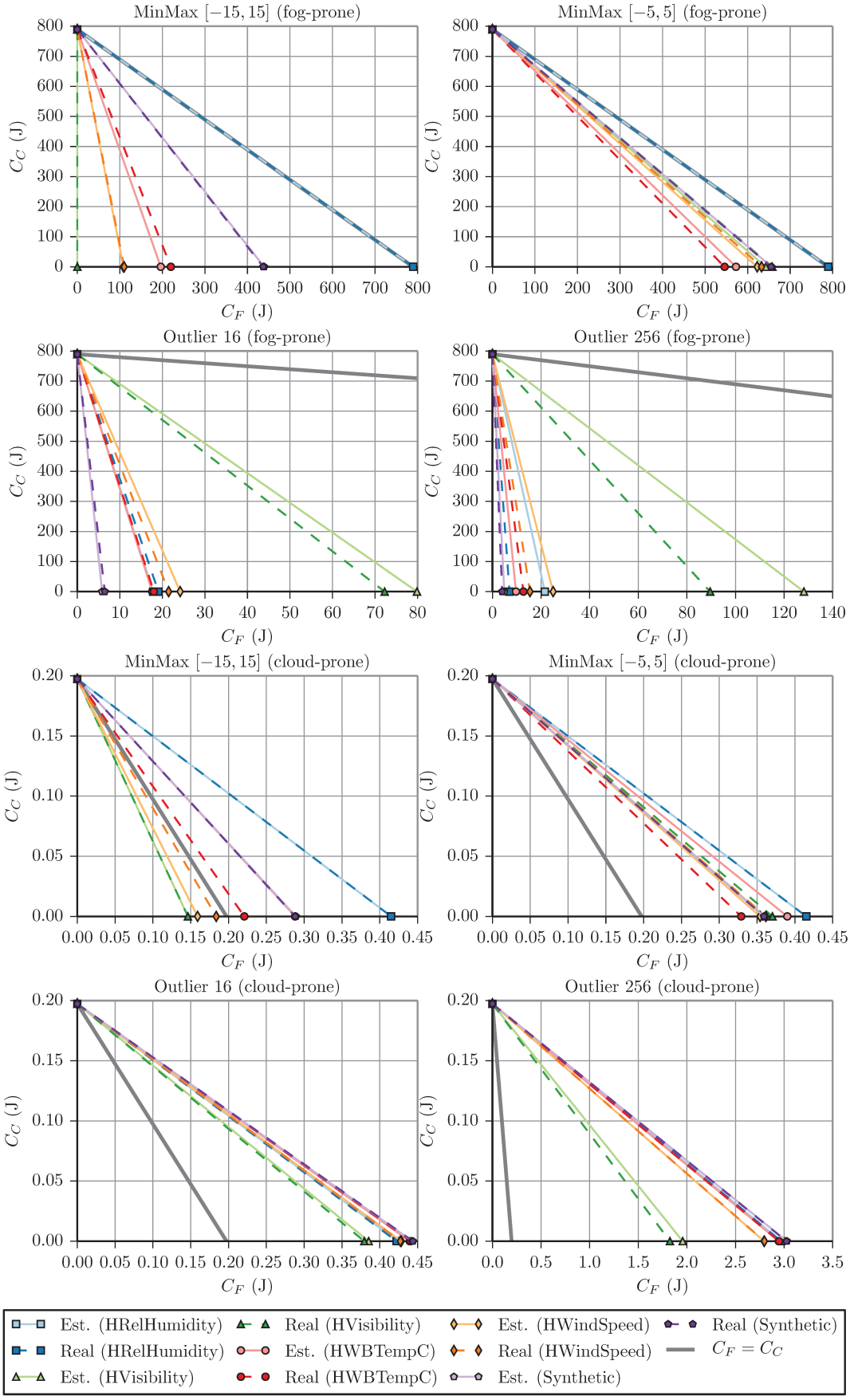


Figure 4.12: Graphs of the linear equations for each dataset and benchmark using energy consumption as the cost: Raspberry Pi 3.

4.3.4 Simulating Other Scenarios

As we have seen with our study of fog-prone and cloud-prone scenarios in the previous subsections, another application for our model is simulating the decision process for different ranges of values. This is useful to help us visualize how changes in technology may affect the decision to filter values in the device instead of sending them to be processed on the cloud. It also allows us to analyze how far we can change s and t and still keep the same decisions.

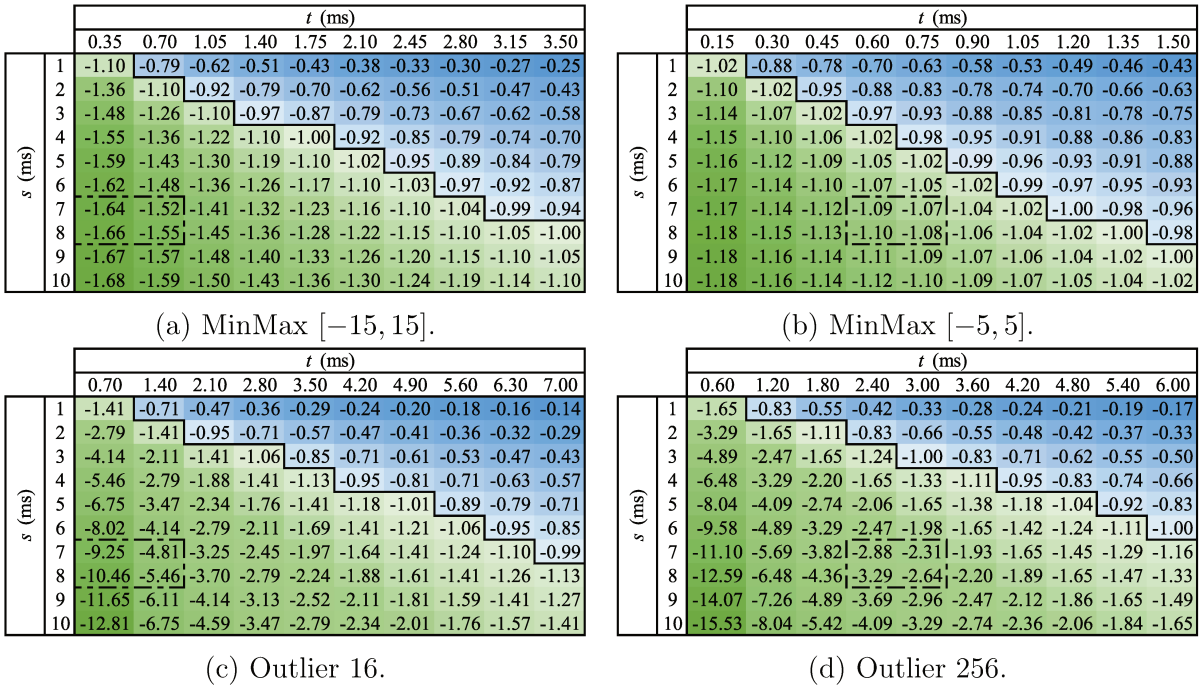
The value of s is related to factors such as the network protocol being used (e.g., TCP, UDP); the implementation of network processes (e.g., routing); and the communication technology employed by the device (e.g., Wi-Fi, Bluetooth, BLE, LTE, Zigbee, WiMax). Moreover, it may include costs related to information security like authentication and data confidentiality. Therefore, improving the performance of any of these elements would decrease the value of s and lead us to change the decision from processing values on the fog to sending them to be processed on the cloud.

At the same time, t is related to factors such as the quality of the filtering procedure's code and the technology of the processing unity used by the device. While we do not expect the performance of these elements to decrease with time, it is possible that t may increase as progressively more limited devices are employed or more robust features are added to the procedures being executed, which would also lead to processing data on the cloud being more profitable than doing so on the fog.

In our simulations, we use the estimated f values for the fog-prone cases of the Synthetic dataset. The upper values for t are approximations of the result of Equation (4.6), and the s values are numbers close to the 7.3 ms, when the cost type is execution time, and 1.022 mJ, when the cost type is energy consumption. We calculate the slope of the lines obtained with these coefficients to observe how changing them affects our results.

In all figures in this subsection, the dashed area represents the space where the values used in Sections 4.3.2 and 4.3.3 are located. The green cells (below the continuous line) are the ones where the slope is less than or equal to -1, that is, the cases where fog computing is more profitable. The blue cells (above the continuous line) are the cases where cloud computing has the lower cost.

Figure 4.13a depicts the simulation results for MinMax $[-15, 15]$. In this case, we need to either increase t or decrease s by $5.1\times$ to reach a situation where fog computing no longer has the lowest cost. Figure 4.13b shows the simulation results for MinMax $[-5, 5]$, for which we need to increase t or decrease s by only $1.9\times$ to change our decision. This is due to this case having a higher f than the previous one, which brings its initial slope already close to -1. Figure 4.13c illustrates the simulation results for Outlier 16, and an increase in t or decrease in s of $9.8\times$ is necessary in this instance to make cloud computing the more profitable choice, as the initial slope is far from -1 due to a very low value of f. Figure 4.13d has the simulation results for Outlier 256. In this test, we need to increase t or decrease s by $2.8\times$ to change our decision. Although this case also presents a very low f value, this is compensated by a t value that is much closer to s than the other cases, explaining why its initial slope is much closer to -1 than Outlier 16.



(a) MinMax $[-15, 15]$.

(b) MinMax $[-5, 5]$.

(c) Outlier 16.

(d) Outlier 256.

Figure 4.13: Slope simulations for each test case using execution time as the cost.

The energy consumption values used in this analysis are the ones obtained for the NodeMCU device. Figure 4.14a has the MinMax $[-15, 15]$ simulation results, for which there is a need to either increase t or decrease s by 20.6× for cloud computing to become the most profitable choice. Similarly to the results in Figure 4.13b, the MinMax $[-5, 5]$ values displayed in Figure 4.14b also needs to be adjusted by a much smaller factor than what is required by MinMax $[-15, 15]$ in order to change the decision between fog and cloud, that is, we need to increase t or decrease s by only 7.9×. Again, this can be explained by this case having a higher f than that of MinMax $[-15, 15]$, which brings its initial slope already close to -1. Figure 4.14c shows the simulation results for Outlier 16, which require a very large adjustment of 39.8× to change our decision. Again, this can be explained by the initial slope being far from -1 due to a very low value of f. Figure 4.14d presents the simulation results for Outlier 256, and here, we need to increase t or decrease s by 11.3× to change our decision. As in the case of Figure 4.13d, the low f value of this instance is compensated by a larger t value, which brings its slope to -1 when compared to Outlier 16.

s (mJ) \ t (mJ)	0.02	0.07	0.12	0.17	0.22	0.27	0.32	0.37	0.42	0.47
0.01	-0.39	-0.13	-0.08	-0.06	-0.04	-0.04	-0.03	-0.03	-0.02	-0.02
0.16	-1.46	-1.00	-0.76	-0.62	-0.52	-0.45	-0.39	-0.35	-0.31	-0.29
0.31	-1.60	-1.27	-1.06	-0.90	-0.79	-0.70	-0.63	-0.57	-0.52	-0.48
0.46	-1.66	-1.41	-1.22	-1.08	-0.96	-0.87	-0.80	-0.73	-0.68	-0.63
0.61	-1.69	-1.48	-1.32	-1.19	-1.09	-1.00	-0.92	-0.86	-0.80	-0.75
0.76	-1.71	-1.54	-1.39	-1.28	-1.18	-1.09	-1.02	-0.96	-0.90	-0.85
0.91	-1.72	-1.57	-1.45	-1.34	-1.25	-1.17	-1.10	-1.04	-0.98	-0.93
1.06	-1.73	-1.60	-1.49	-1.39	-1.30	-1.23	-1.16	-1.10	-1.05	-1.00
1.21	-1.74	-1.62	-1.52	-1.43	-1.35	-1.28	-1.21	-1.16	-1.10	-1.06
1.36	-1.74	-1.64	-1.54	-1.46	-1.39	-1.32	-1.26	-1.20	-1.15	-1.11

(a) MinMax $[-15, 15]$.

s (mJ) \ t (mJ)	0.02	0.04	0.06	0.08	0.10	0.12	0.14	0.16	0.18	0.20
0.01	-0.35	-0.21	-0.15	-0.11	-0.09	-0.08	-0.07	-0.06	-0.05	-0.05
0.16	-1.05	-0.93	-0.83	-0.75	-0.69	-0.63	-0.59	-0.55	-0.51	-0.48
0.31	-1.12	-1.04	-0.98	-0.92	-0.87	-0.82	-0.78	-0.74	-0.71	-0.68
0.46	-1.15	-1.09	-1.04	-1.00	-0.96	-0.92	-0.88	-0.85	-0.82	-0.79
0.61	-1.16	-1.12	-1.08	-1.04	-1.01	-0.97	-0.94	-0.92	-0.89	-0.86
0.76	-1.17	-1.13	-1.10	-1.07	-1.04	-1.01	-0.99	-0.96	-0.94	-0.92
0.91	-1.18	-1.15	-1.12	-1.09	-1.07	-1.04	-1.02	-1.00	-0.97	-0.95
1.06	-1.18	-1.15	-1.13	-1.11	-1.08	-1.06	-1.04	-1.02	-1.00	-0.98
1.21	-1.18	-1.16	-1.14	-1.12	-1.10	-1.08	-1.06	-1.04	-1.02	-1.01
1.36	-1.19	-1.16	-1.15	-1.13	-1.11	-1.09	-1.07	-1.06	-1.04	-1.02

(b) MinMax $[-5, 5]$.

s (mJ) \ t (mJ)	0.02	0.12	0.22	0.32	0.42	0.52	0.62	0.72	0.82	0.92
0.01	-0.50	-0.08	-0.05	-0.03	-0.02	-0.02	-0.02	-0.01	-0.01	-0.01
0.16	-7.59	-1.32	-0.72	-0.50	-0.38	-0.31	-0.26	-0.22	-0.19	-0.17
0.31	-14.01	-2.54	-1.40	-0.96	-0.73	-0.59	-0.50	-0.43	-0.38	-0.34
0.46	-19.87	-3.74	-2.06	-1.42	-1.09	-0.88	-0.74	-0.64	-0.56	-0.50
0.61	-25.24	-4.91	-2.72	-1.88	-1.44	-1.16	-0.98	-0.84	-0.74	-0.66
0.76	-30.16	-6.07	-3.37	-2.34	-1.79	-1.45	-1.22	-1.05	-0.92	-0.82
0.91	-34.70	-7.21	-4.02	-2.79	-2.14	-1.73	-1.45	-1.25	-1.10	-0.98
1.06	-38.90	-8.33	-4.66	-3.24	-2.48	-2.01	-1.69	-1.46	-1.28	-1.14
1.21	-42.80	-9.43	-5.30	-3.69	-2.83	-2.29	-1.93	-1.66	-1.46	-1.30
1.36	-46.42	-10.52	-5.93	-4.13	-3.17	-2.57	-2.16	-1.86	-1.64	-1.46

(c) Outlier 16.

s (mJ) \ t (mJ)	0.02	0.14	0.26	0.38	0.50	0.62	0.74	0.86	0.98	1.10
0.01	-0.50	-0.07	-0.04	-0.03	-0.02	-0.02	-0.01	-0.01	-0.01	-0.01
0.16	-7.91	-1.14	-0.61	-0.42	-0.32	-0.26	-0.22	-0.19	-0.16	-0.15
0.31	-15.15	-2.21	-1.19	-0.81	-0.62	-0.50	-0.42	-0.36	-0.32	-0.28
0.46	-22.23	-3.27	-1.76	-1.21	-0.92	-0.74	-0.62	-0.53	-0.47	-0.42
0.61	-29.17	-4.33	-2.34	-1.60	-1.22	-0.98	-0.82	-0.71	-0.62	-0.55
0.76	-35.95	-5.38	-2.91	-1.99	-1.52	-1.22	-1.03	-0.88	-0.77	-0.69
0.91	-42.60	-6.44	-3.48	-2.39	-1.82	-1.46	-1.23	-1.06	-0.93	-0.83
1.06	-49.10	-7.49	-4.05	-2.78	-2.11	-1.71	-1.43	-1.23	-1.08	-0.96
1.21	-55.47	-8.53	-4.62	-3.17	-2.41	-1.95	-1.63	-1.40	-1.23	-1.10
1.36	-61.72	-9.58	-5.19	-3.56	-2.71	-2.19	-1.83	-1.58	-1.38	-1.23

(d) Outlier 256.

Figure 4.14: Slope simulations for each test case using energy consumption as the cost.

Lastly, it is relevant to point out that we also simulated all of these cases employing the real f values. Although that leads us to slightly different slopes, the decision between fog and cloud stays the same in all scenarios. If we had used a smaller value for the maximum increase in cost allowed, we would have seen some marginal discrepancies in borderline cases (i.e., cases where the slope is very close to -1), as f would have been less precise due to being estimated using fewer data points. Divergences can also occur due to rounding differences in borderline cases. We do not consider this an issue, as borderline cases have similar fog and cloud costs.

4.4 Conclusion

In this chapter, we introduce two cost models for fog and cloud computing. The first is a mathematical model that can be used to estimate the cost of processing a data stream on the device that collected it (fog computing cost) and the cost of sending the data to be processed on the cloud (cloud computing cost). The second is a visual model that is a cloud computing cost vs. fog computing cost graph, which aims at helping the user decide which of the strategies presents the lowest cost according to a certain metric.

One of the parameters of our model is the probability that a number will pass the filter, which we call f. We created a procedure to use our mathematical model to estimate this value based on the cost penalty that the user is willing to pay for this calculation, and analyzed different strategies for executing it. We observed that looking at a contiguous set of elements at the beginning of the stream is a straightforward approach that yields good estimates. The two other approaches that we tested, which involve continuously monitoring the stream and dynamically adjusting the estimate, presented very little performance gains, not justifying their use.

We applied our models to two instances of a filter that allows numbers outside of a certain range to pass and two instances of a filter that finds outliers in a window of values. We used execution time and energy consumption as the cost types for our analysis and executed the tests on five different datasets (four datasets with real-world climatological data and one dataset with artificial data). Our experimental results indicate that: *i)* our estimation process for fog and cloud computing costs works well, presenting less than 5% error in most of our test cases and correctly choosing the most profitable strategy to process the values in all but one case; *ii)* in order to visualize possible changes in technology, we were able to use our mathematical model to simulate a different range of values for our test cases and found out how changing the time it takes to send the data to the cloud (parameter s of our model) or changing the time it takes to execute the application on the fog (parameter t of our model) would affect our decision between fog and cloud computing.

Chapter 5

Conclusions

In this dissertation, we worked toward the answer to the research question "*In what cases is it more profitable to perform computation on a constrained IoT device instead of using the cloud?*".

5.1 Scope of This Work

We started by defining the restrictions of our solution. That is, we established that the devices that we are working with must be: *i)* capable of executing custom code; *ii)* connected to the Internet; *iii)* able to send packets to other devices; *iv)* equipped with a capture instrument, such as a microphone or a sensor that periodically measures an environmental variable. Furthermore, we assumed that we are handling: *i)* non-empty data streams of limited size (i.e., the stream size is neither zero nor infinite); *ii)* cases where there is no overlap between processing and transferring the data.

Then, we delimited the type of application that would be the focus of our analysis, which can be expressed as: a user (e.g., a data scientist) wants to query a device looking for meaningful information. To reach this goal, the user expresses their query as a program, sends it to be executed on the device, and then receives its result. These requests are of one of two types: one-time queries, which consider the data of a snapshot taken from the stream at a certain moment in time, are evaluated only once, and return the answer to the user; or continuous queries, which are evaluated continuously as new values arrive and produce answers over time, always reflecting the stream data seen so far. Moreover, we noted that filtering procedures are particularly interesting for these queries, as they can be simple enough to run on constrained devices and may become very useful with the large-scale adoption the IoT due to their potential to drastically decrease the amount of data sent to the cloud.

5.2 Proposed Framework

Considering the expected scope of our solution, we proceeded to look for existing frameworks that allowed users to send custom code to be executed on constrained devices. As the studied tools were not compatible with the resource limitations of these devices, we

developed LMC, a framework that enables cross-platform code execution on constrained IoT devices by combining a compact virtual machine (COISA) with a lean event handling mechanism (LibMiletus).

We used LMC to deploy and execute one-time and continuous queries on constrained IoT devices and compared its performance with the Apache Edgent framework using three test cases on two IoT platforms, the simple NodeMCU 1.0 and the more robust DragonBoard 410c. Our experimental results indicated that LMC is: *i)* very compact and compatible with Class 2 constrained devices; *ii)* overall faster than the Edgent framework if we disable dynamic translation mechanisms; *iii)* faster than Edgent at lightweight quick queries when they are both being interpreted, in some cases, even if LMC is running on the small NodeMCU platform while Edgent is running on the DragonBoard 410c.

With this result, we were able to establish that indeed there are cases where it is profitable to leverage constrained IoT devices for custom code execution instead of employing more powerful devices.

5.3 Proposed Model

In order to be able to systematically identify the cases where it is more profitable to leverage constrained IoT devices for custom code execution instead of employing more powerful devices, it was necessary to characterize the scenarios where this holds true. Thus, we introduced two cost models for fog and cloud computing. The first is a mathematical model that can be used to estimate the cost of processing a data stream on the device that collected it (fog computing cost) and the cost of sending the data to be processed on the cloud (cloud computing cost). Both of these costs are calculated from the point of view of the device, as this would allow it to choose where to process the data stream while collecting its elements. The second is a visual model that is a cloud computing cost vs. fog computing cost graph. It intends to help the user decide which of the strategies presents the lowest cost according to a certain metric. Using this visual model, the user can better understand the test cases they are working with and improve the implementation of their processing strategy.

Considering that our models are based on linear equations, we observed that the user can make their decision by analyzing the slope of a line drawn in the visual model. As discussed, fog computing is more profitable in the cases where the slope is less than or equal to -1, with cloud computing having the lower cost otherwise.

One of the parameters of our model is the probability that a number will pass the filter, which we call f. We analyzed different strategies to estimate this value and observed that looking at a contiguous set of elements at the beginning of the stream is a straightforward approach that yields good estimates. The two other approaches that we tested, which involve continuously monitoring the stream and dynamically adjusting the estimate, presented very little performance gains, not justifying their use.

We applied our models to two instances of two different filters, used execution time and energy consumption as the cost types for our analysis, and executed the tests on five different datasets (four datasets with real-world climatological data and one dataset with

artificial data). By comparing the slope of the linear equation obtained with the real and estimated values of f, we noticed that our estimation process worked well, as it presented an error of less than 5% in most of our test cases and allowed us to decide correctly on the more profitable strategy to process the values in all but one case.

We also simulated a different range of values for our test cases and found out how different parameters would affect our decision. We looked at how much it would be necessary to decrease the time it takes to send the data to the cloud (which we call s) or increase the time it takes to execute the application on the fog (which we call t) for cloud computing to become the more profitable approach in cases where the fog was the chosen solution. When using execution time as the cost type, the values of the parameters had to change from 1.9× to 9.8× to affect our decision, and in the case of energy consumption as the cost type, they had to change from 7.9× up to 39.8×. We noticed that the size of these alterations depends on factors such as the value of f and how close s and t are to each other. We point out that this type of investigation is very useful to visualize possible changes in technology. Again, our estimation process proved to be effective, as the simulations using the real and predicted values presented the same decisions in all cases.

By employing these models, we were able to systematically identify a set of cases where performing the computation on a constrained IoT device is more profitable than sending the values to be processed on the cloud.

5.4 Main Contributions

The main contributions of this work are as follows:

- We developed an open-source platform called LibMiletusCOISA (LMC), which allows users to execute their code on constrained devices;
- We used our infrastructure to obtain real-world values to use in our test cases;
- We created two models where the user chooses a certain cost metric (e.g., number of instructions, execution time, and energy consumption) and employs it to decide where they should execute their code:
  - A generic mathematical model that uses a linear equation to determine the costs;
  - A visual model that allows the user to conclude quickly what is the most profitable approach in a specific scenario.
- We created a procedure to use our mathematical model to estimate the probability of a value passing a filter based on the cost penalty that the user is willing to pay for this calculation;
- We simulated future scenarios where changes in communication and processing technologies can affect whether the fog or the cloud is the most profitable approach.

5.5 Future Work

There are still several paths to be explored within this research topic for both the LMC framework and the cost models.

Regarding LMC, we have the possibility to compare it to other frameworks, the use of more benchmarks and larger test cases, and the execution of tests on constrained devices that belong to Class 2 or lower. Another possibility is the implementation of the Cloud-Assisted Translation mechanism and using it to obtain more precise results related to statically translating the user programs.

In the case of the cost models, we can test them using other custom code operations, such as aggregation, parsing, or other more general queries. We can also consider that the calculated costs may change over time (due to, for example, device mobility, packet collisions, radio-frequency interference from other devices, or changes in the network). A possible approach to doing this is using a network simulator to generate new values for the cost of sending the data to the cloud.

The models can also be improved to account for scenarios not included in this dissertation, such as the device having multiple antennas with different wireless technologies, splitting tasks and sending data to other devices in the fog hierarchy, overlapping processing and transferring times, and the device receiving data for actuation.

Another interesting path is to not only consider the costs from the point of view of the device, but also to calculate the costs looking at the whole system in order to analyze the impact of employing fog computing in the studied applications. Moreover, there is the possibility to implement a system that uses our model to choose between processing values on the fog or the cloud, and execute it on constrained devices.

Bibliography

[1] Muhammad Amjad, Muhammad Sharif, Muhammad Khalil Afzal, and Sung Won Kim. TinyOS-New Trends, Comparative Views, and Supported Sensing Applications: A Review. *IEEE Sensors Journal*, 16(9):2865–2889, May 2016. doi:10.1109/JSEN.2016.2519924. (Cited on page 19.)

[2] Apache Software Foundation Incubator. Apache Edgent Overview. https://edgent.apache.org/docs/overview. Accessed: May 01, 2019. (Cited on pages 10, 17, 21, 25, and 27.)

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, April 2010. doi:10.1145/1721654.1721672. (Cited on pages 83, 84, and 85.)

[4] ARM Ltd. Mbed OS. https://www.mbed.com/en/platform/mbed-os/. Accessed: May 01, 2019. (Cited on page 19.)

[5] Faisal Aslam, Luminous Fennell, Christian Schindelhauer, Peter Thiemann, Gidon Ernst, Elmar Haussmann, Stefan Rührup, and Zastash A. Uzmi. Optimized Java Binary and Virtual Machine for Tiny Motes. In *Proceedings of the 6th IEEE International Conference on Distributed Computing in Sensor Systems*, DCOSS '10, pages 15–30, June 2010. doi:10.1007/978-3-642-13651-1_2. (Cited on pages 10, 20, and 21.)

[6] Dale Athanasias. PyMite. https://wiki.python.org/moin/PyMite. Accessed: May 01, 2019. (Cited on page 20.)

[7] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, October 2010. doi:10.1016/j.comnet.2010.05.010. (Cited on pages 86, 87, 88, and 89.)

[8] Rafael Auler, Carlos Eduardo Millani, Alexandre Brisighello, Alisson Linhares, and Edson Borin. Handling IoT platform heterogeneity with COISA, a compact OpenISA virtual platform. *Concurrency and Computation: Practice and Experience*, 29(22):e3932, November 2017. doi:10.1002/cpe.3932. (Cited on pages 10, 20, 21, 27, 28, 35, and 37.)

[9] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *Proceedings of the Twenty-first ACM*

SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '02, pages 1–16, June 2002. doi:10.1145/543613.543615. (Cited on page 32.)

[10] Luiz Bittencourt, Roger Immich, Rizos Sakellariou, Nelson Fonseca, Edmundo Madeira, Marilia Curado, Leandro Villas, Luiz da Silva, Craig Lee, and Omer Rana. The Internet of Things, Fog and Cloud continuum: Integration and challenges. *Internet of Things*, 3–4:134–155, October 2018. doi:10.1016/j.iot.2018.09.005. (Cited on page 16.)

[11] Luiz Fernando Bittencourt, Javier Diaz-Montes, Rajkumar Buyya, Omer F. Rana, and Manish Parashar. Mobility-Aware Application Scheduling in Fog Computing. *IEEE Cloud Computing*, 4(2):26–35, March–April 2017. doi:10.1109/MCC.2017.27. (Cited on page 90.)

[12] Genom Bob. HaikuVM - A Java VM for ARDUINO and other micros using the leJOS runtime. http://haiku-vm.sourceforge.net/. Accessed: May 01, 2019. (Cited on page 20.)

[13] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog Computing: A Platform for Internet of Things and Analytics. In Nik Bessis and Ciprian Dobre, editors, *Big Data and Internet of Things: A Roadmap for Smart Environments*, volume 546 of *Studies in Computational Intelligence*, pages 169–186. Springer, Cham, March 2014. doi:10.1007/978-3-319-05029-4_7. (Cited on page 91.)

[14] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh 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*, MCC '12, pages 13–16, August 2012. doi:10.1145/2342509.2342513. (Cited on pages 16, 90, and 93.)

[15] Carsten Bormann, Mehmet Ersue, and Ari Keranen. Terminology for Constrained-Node Networks. Technical report, Internet Engineering Task Force, May 2014. doi:10.17487/RFC7228. (Cited on pages 78 and 79.)

[16] Niels Brouwers, Peter Corke, and Koen Langendoen. A Java Compatible Virtual Machine for Wireless Sensor Nodes. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 369–370, November 2008. doi:10.1145/1460412.1460456. (Cited on pages 10, 20, and 21.)

[17] Charles C. Byers. Architectural Imperatives for Fog Computing: Use Cases, Requirements, and Architectural Techniques for Fog-Enabled IoT Networks. *IEEE Communications Magazine*, 55(8):14–20, August 2017. doi:10.1109/MCOM.2017.1600885. (Cited on page 92.)

[18] Alexandru Caracaş, Thorsten Kramp, Michael Baentsch, Marcus Oestreicher, Thomas Eirich, and Ivan Romanov. Mote Runner: A Multi-language Virtual Machine for Small Embedded Devices. In *Proceedings of the 2009 Third International Conference on Sensor Technologies and Applications*, SENSORCOMM '09, pages

117–125, June 2009. doi:10.1109/SENSORCOMM.2009.27. (Cited on pages 10, 20, and 21.)

[19] Joon-Hyuk Chang, Nam Soo Kim, and Sanjit K. Mitra. Voice Activity Detection Based on Multiple Statistical Models. *IEEE Transactions on Signal Processing*, 54(6):1965–1976, June 2006. doi:10.1109/TSP.2006.874403. (Cited on page 31.)

[20] Guangyu Chen, Byung-Tae Kang, Mahmut Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Rajarathnam Chandramouli. Studying Energy Trade Offs in Offloading Computation/Compilation in Java-Enabled Mobile Devices. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):795–809, September 2004. doi:10.1109/TPDS.2004.47. (Cited on page 23.)

[21] Mung Chiang and Tao Zhang. Fog and IoT: An Overview of Research Opportunities. *IEEE Internet of Things Journal*, 3(6):854–864, December 2016. doi:10.1109/JIOT.2016.2584538. (Cited on page 86.)

[22] Cisco Systems. Introduction to IOx. https://developer.cisco.com/docs/iox/. Accessed: May 01, 2019. (Cited on pages 10, 17, 21, and 25.)

[23] Inc. Crossbow Technology. TOS In-Network Programming User Reference. http://cs.uccs.edu/~cs526/tinyos/doc/Xnp.pdf. Accessed: May 01, 2019. (Cited on page 20.)

[24] Andy Davis, Jay Parikh, and William E. Weihl. Edgecomputing: Extending Enterprise Applications to the Edge of the Internet. In *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters*, WWW Alt. '04, pages 180–187, May 2004. doi:10.1145/1013367.1013397. (Cited on page 93.)

[25] Ruilong Deng, Rongxing Lu, Chengzhe Lai, Tom H. Luan, and Hao Liang. Optimal Workload Allocation in Fog-Cloud Computing Toward Balanced Delay and Power Consumption. *IEEE Internet of Things Journal*, 3(6):1171–1181, December 2016. doi:10.1109/JIOT.2016.2565516. (Cited on page 24.)

[26] Behnam Dezfouli, Immanuel Amirtharaj, and Chia-Chi Li. EMPIOT: An Energy Measurement Platform for Wireless IoT Devices. *Journal of Network and Computer Applications*, 121:135–148, November 2018. doi:10.1016/j.jnca.2018.07.016. (Cited on page 58.)

[27] Vanderson Martins do Rosario, Flávia Pisani, Alexandre Rodrigues Gomes, and Edson Borin. Fog-Assisted Translation: Towards Efficient Software Emulation on Heterogeneous IoT Devices. In *Proceedings of the 7th Workshop on Parallel Programming Models - Special Edition on Edge/Fog/In-Situ Computing*, MPP '18, pages 1268–1277, May 2018. doi:10.1109/IPDPSW.2018.00196. (Cited on pages 48 and 53.)

[28] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 455–462, November 2004. doi:10.1109/LCN.2004.38. (Cited on page 20.)

[29] Mohamed Firdhous, Osman Ghazali, and Suhaidi Hassan. Fog Computing: Will it be the Future of Cloud Computing? In *Proceedings of the Third International Conference on Informatics & Applications*, ICIA '14, October 2014. (Cited on pages 10, 91, and 92.)

[30] FogHorn Systems. Real-Time Edge Intelligence for Industrial IoT. www.foghorn.io/. Accessed: May 01, 2019. (Cited on pages 10, 17, 22, and 25.)

[31] Google Inc. Android Things - Overview. https://developer.android.com/things/get-started. Accessed: May 02, 2019. (Cited on page 19.)

[32] Dmitry Grinberg. uJ - a Java VM for microcontrollers. http://dmitry.gr/index.php?r=05.Projects&proj=12.%20uJ%20-%20a%20micro%20JVM. Accessed: May 02, 2019. (Cited on pages 10, 20, and 21.)

[33] Xiaohui Gu, Klara Nahrstedt, Alan Messer, Ira Greenberg, and Dejan Milojicic. Adaptive Offloading Inference for Delivering Applications in Pervasive Computing Environments. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, PerCom '03, pages 107–114, March 2003. doi:10.1109/PERCOM.2003.1192732. (Cited on page 23.)

[34] Selim Gurun, Chandra Krintz, and Rich Wolski. NWSLite: A Light-Weight Prediction Utility for Mobile Devices. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services*, MobiSys '04, pages 2–11, June 2004. doi:10.1145/990064.990068. (Cited on pages 24 and 25.)

[35] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A Dynamic Operating System for Sensor Nodes. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, pages 163–176, June 2005. doi:10.1145/1067170.1067188. (Cited on page 20.)

[36] Till Harbaum. The NanoVM - Java for the AVR. http://www.harbaum.org/till/nanovm/index.shtml. Accessed: May 02, 2019. (Cited on pages 10, 20, and 21.)

[37] David C. Hoaglin, Boris Iglewicz, and John W. Tukey. Performance of Some Resistant Rules for Outlier Labeling. *Journal of the American Statistical Association*, 81(396):991–999, December 1986. doi:10.2307/2289073. (Cited on page 31.)

[38] Yu-Ju Hong, Karthik Kumar, and Yung-Hsiang Lu. Energy Efficient Content-Based Image Retrieval for Mobile Systems. In *Proceedings of the 2009 IEEE International Symposium on Circuits and Systems*, ISCAS '09, pages 1673–1676, May 2009. doi:10.1109/ISCAS.2009.5118095. (Cited on page 23.)

[39] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile Edge Computing - A key technology towards 5G. Technical report, European Telecommunications Standards Institute, September 2015. Accessed: May 01, 2019. URL: https://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp11_mec_a_key_technology_towards_5g.pdf. (Cited on page 93.)

[40] Jonathan W. Hui and David Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 81–94, November 2004. doi:10.1145/1031495.1031506. (Cited on page 20.)

[41] Michaela Iorga, Larry Feldman, Robert Barton, Michael J. Martin, Nedim Goren, and Charif Mahmoudi. Fog Computing Conceptual Model - Recommendations of the National Institute of Standards and Technology. Technical report, National Institute of Standards and Technology, March 2018. doi:10.6028/NIST.SP.500-325. (Cited on pages 17, 91, and 94.)

[42] Prem Prakash Jayaraman, João Bártolo Gomes, Hai Long Nguyen, Zahraa Said Abdallah, Shonali Krishnaswamy, and Arkady Zaslavsky. CARDAP: A Scalable Energy-Efficient Context Aware Distributed Mobile Data Analytics Platform for the Fog. In *Proceedings of the 2014 East European Conference on Advances in Databases and Information Systems*, ADBIS '14, pages 192–206, September 2014. doi:10.1007/978-3-319-10933-6_15. (Cited on page 24.)

[43] Jaein Jeong and David Culler. Incremental Network Programming for Wireless Sensors. In *Proceedings of the First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, IEEE SECON '04, pages 25–33, October 2004. doi:10.1109/SAHCN.2004.1381899. (Cited on page 20.)

[44] Kevin Klues, Chieh-Jan Mike Liang, Jeongyeup Paek, Răzvan Musăloiu-E, Philip Levis, Andreas Terzis, and Ramesh Govindan. TOSThreads: Thread-safe and Non-invasive Preemption in TinyOS. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 127–140, November 2009. doi:10.1145/1644038.1644052. (Cited on page 20.)

[45] Hermann Kopetz. Internet of Things. In *Design Principles for Distributed Embedded Applications*, Real-Time Systems, pages 307–323. Springer, Boston, MA, USA, February 2011. doi:10.1007/978-1-4419-8237-7_13. (Cited on pages 86, 87, 88, 89, and 90.)

[46] Joel Koshy and Raju Pandey. VMSTAR: Synthesizing Scalable Runtime Environments for Sensor Networks. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, SenSys '05, pages 243–254, November 2005. doi:10.1145/1098918.1098945. (Cited on page 20.)

[47] Ulrich Kremer, Jamey Hicks, and James Rehg. A Compilation Framework for Power and Energy Management on Mobile Computers. In *Proceedings of the 2001 International Workshop on Languages and Compilers for Parallel Computing*, LCPC '01, pages 115–131, August 2001. doi:10.1007/3-540-35767-X_8. (Cited on page 23.)

[48] Sam Leroux, Steven Bohez, Elias De Coninck, Tim Verbelen, Bert Vankeirsbilck, Pieter Simoens, and Bart Dhoedt. Multi-Fidelity Matryoshka Neural Networks for Constrained loT Devices. In *Proceedings of the 2016 International Joint Conference*

on Neural Networks, IJCNN '16, pages 1305–1309, July 2016. doi:10.1109/IJCNN.2016.7727348. (Cited on page 78.)

[49] Shancang Li, Li Da Xu, and Shanshan Zhao. The internet of things: a survey. *Information Systems Frontiers*, 17(2):243–259, April 2015. doi:10.1007/s10796-014-9492-7. (Cited on pages 86, 87, and 88.)

[50] Zhiyuan Li, Cheng Wang, and Rong Xu. Computation Offloading to Save Energy on Handheld Devices: A Partition Scheme. In *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '01, pages 238–246, November 2001. doi:10.1145/502217.502257. (Cited on page 22.)

[51] Zhiyuan Li, Cheng Wang, and Rong Xu. Task Allocation for Distributed Multimedia Processing on Wirelessly Networked Handheld Devices. In *Proceedings of the 16th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '02, April 2002. doi:10.1109/IPDPS.2002.1015589. (Cited on page 22.)

[52] Zhiyuan Li and Rong Xu. Energy Impact of Secure Computation on a Handheld Device. In *Proceedings of the 2002 IEEE International Workshop on Workload Characterization*, WWC-5, pages 109–117, November 2002. doi:10.1109/WWC.2002.1226499. (Cited on page 22.)

[53] Liqing Liu, Zheng Chang, Xijuan Guo, Shiwen Mao, and Tapani Ristaniemi. Multiobjective Optimization for Computation Offloading in Fog Computing. *IEEE Internet of Things Journal*, 5(1):283–294, February 2018. doi:10.1109/JIOT.2017.2780236. (Cited on page 25.)

[54] Sam Lucero. IoT platforms: enabling the Internet of Things. Technical report, IHS Technology, March 2016. (Cited on pages 16 and 27.)

[55] Chris Mack. The Multiple Lives of Moore's Law. *IEEE Spectrum*, 52(4):31–37, April 2015. doi:10.1109/MSPEC.2015.7065415. (Cited on page 79.)

[56] Pedro José Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel. FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks. In *Proceedings of the Third European Workshop on Wireless Sensor Networks*, EWSN '06, pages 212–227, February 2006. doi:10.1007/11669463_17. (Cited on page 20.)

[57] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards and Technology, September 2011. doi:10.6028/NIST.SP.800-145. (Cited on pages 83, 84, and 86.)

[58] Jozef Mocnej, Martin Miškuf, Peter Papcun, and Iveta Zolotová. Impact of Edge Computing Paradigm on Energy Consumption in IoT. *IFAC-PapersOnLine*, 51(6):162–167, May 2018. Special issue: 15th IFAC Conference on Programmable Devices and Embedded Systems (PDeS '18). doi:10.1016/j.ifacol.2018.07.147. (Cited on page 56.)

[59] MotorolaMobilityLLC. LibMiletus - IoT prototyping made easy! Accessed: May 02, 2019. URL: http://libmiletus.ic.unicamp.br. (Cited on pages 27 and 28.)

[60] Luca Mottola, Gian Pietro Picco, and Adil Amjad Sheikh. FiGaRo: Fine-Grained Software Reconfiguration for Wireless Sensor Networks. In *Proceedings of the 5th European Conference on Wireless Sensor Networks*, EWSN '08, pages 286–304, January–February 2008. doi:10.1007/978-3-540-77690-1_18. (Cited on page 20.)

[61] Waqaas Munawar, Muhammad Hamad Alizai, Olaf Landsiedel, and Klaus Wehrle. Dynamic TinyOS: Modular and Transparent Incremental Code-Updates for Sensor Networks. In *Proceedings of the 2010 IEEE International Conference on Communications*, ICC '10, May 2010. doi:10.1109/ICC.2010.5501964. (Cited on pages 19 and 20.)

[62] National Centers for Environmental Information. Climate Data Online: Dataset Discovery - Local Climatological Data. https://www.ncdc.noaa.gov/cdo-web/datasets#LCD. Accessed: May 02, 2019. (Cited on page 46.)

[63] José Leal D. Neto, Se-Young Yu, Daniel F. Macedo, José Marcos S. Nogueira, Rami Langar, and Stefano Secci. ULOOF: A User Level Online Offloading Framework for Mobile Edge Computing. *IEEE Transactions on Mobile Computing*, 17(11):2660–2674, November 2018. doi:10.1109/TMC.2018.2815015. (Cited on page 25.)

[64] Yamini Nimmagadda, Karthik Kumar, Yung-Hsiang Lu, and C. S. George Lee. Real-time Moving Object Recognition and Tracking Using Computation Offloading. In *Proceedings of the 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IROS '10, pages 2449–2455, October 2010. doi:10.1109/IROS.2010.5650303. (Cited on page 24.)

[65] Keith J. O'Hara, Ripal Nathuji, Himanshu Raj, Karsten Schwan, and Tucker Balch. AutoPower: Toward Energy-Aware Software Systems for Distributed Mobile Robots. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation*, ICRA '06, pages 2757–2762, May 2006. doi:10.1109/ROBOT.2006.1642118. (Cited on page 23.)

[66] Rajesh Krishna Panta, Saurabh Bagchi, and Samuel P. Midkiff. Zephyr: Efficient Incremental Reprogramming of Sensor Nodes Using Function Call Indirections and Difference Computation. In *Proceedings of the 2009 USENIX Annual Technical Conference*, USENIX '09, pages 411–424, June 2009. (Cited on page 20.)

[67] Poonam Pingale, Kalpana Amrutkar, and Suhas Kulkarni. Design aspects for Upgrading Firmware of a Resource Constrained Device in the Field. In *Proceedings of the 2016 IEEE International Conference on Recent Trends in Electronics, Information Communication Technology*, RTEICT '16, pages 903–907, May 2016. doi:10.1109/RTEICT.2016.7807959. (Cited on page 78.)

[68] Flávia Pisani, Jeferson Rech Brunetta, Vanderson Martins do Rosario, and Edson Borin. Beyond the Fog: Bringing Cross-Platform Code Execution to Constrained IoT Devices. In *Proceedings of the 29th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '17, pages 17–24, October 2017. © 2017 IEEE. Reprinted (modified), with permission. doi:10.1109/SBAC-PAD.2017.10. (Cited on page 27.)

[69] Flávia Pisani, Vanderson Martins do Rosario, and Edson Borin. Fog vs. Cloud Computing: Should I Stay or Should I Go? *Future Internet*, 11(2), February 2019. doi:10.3390/fi11020034. (Cited on pages 16 and 41.)

[70] Deepak Puthal, Bibhudutta P. S. Sahoo, Sambit Mishra, and Satyabrata Swain. Cloud Computing Features, Issues, and Challenges: A Big Picture. In *Proceedings of the 2015 International Conference on Computational Intelligence and Networks*, CINE '15, pages 116–123, January 2015. doi:10.1109/CINE.2015.31. (Cited on pages 84 and 86.)

[71] Niels Reijers and Koen Langendoen. Efficient Code Distribution in Wireless Sensor Networks. In *Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications*, WSNA '03, pages 60–67, September 2003. doi:10.1145/941350.941359. (Cited on page 20.)

[72] Industrial Internet Consortium. The Industrial Internet Consortium and OpenFog Consortium Unite. https://www.iiconsortium.org/press-room/01-31-19.htm. Accessed: August 09, 2019. (Cited on page 91.)

[73] OpenFog Consortium Architecture Working Group. OpenFog Reference Architecture for Fog Computing. Technical report, OpenFog Consortium, February 2017. Accessed: August 12, 2019. URL: https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf. (Cited on page 90.)

[74] Peng Rong and Massoud Pedram. Extending the Lifetime of a Network of Battery-Powered Mobile Devices by Remote Processing: A Markovian Decision-based Approach. In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pages 906–911, June 2003. doi:10.1145/775832.776060. (Cited on page 23.)

[75] RTJ Computing Pty. Ltd. SimpleRTJ - a small footprint Java VM for embedded and consumer devices. Accessed: May 02, 2019. URL: https://rtjcom.com/files/simpleRTJ-TechInfo.PDF. (Cited on pages 10, 20, and 21.)

[76] Amanda Saint. Where next for the Internet of Things? *Engineering & Technology*, 10(1):72–75, February 2015. doi:10.1049/et.2015.0111. (Cited on pages 10, 17, 22, and 25.)

[77] Muhammad Sajjad, Khan Muhammad, Sung Wook Baik, Seungmin Rho, Zahoor Jan, Sang-Soo Yeo, and Irfan Mehmood. Mobile-cloud assisted framework for selective encryption of medical images with steganography for resource-constrained

devices. *Multimedia Tools and Applications*, 76(3):3519–3536, February 2017. doi:10.1007/s11042-016-3811-6. (Cited on page 78.)

[78] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):14–23, October–December 2009. doi:10.1109/MPRV.2009.82. (Cited on page 93.)

[79] Doug Simon and Cristina Cifuentes. The Squawk Virtual Machine: Java™ on the Bare Metal. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 150–151, October 2005. doi:10.1145/1094855.1094908. (Cited on page 20.)

[80] Karolj Skala, Davor Davidovic, Enis Afgan, Ivan Sovic, and Zorislav Sojat. Scalable Distributed Computing Hierarchy: Cloud, Fog and Dew Computing. *Open Journal of Cloud Computing*, 2(1):16–24, 2015. doi:10.19210/1002.2.1.16. (Cited on page 90.)

[81] Ivan Stojmenovic. Fog computing: A cloud to the ground support for smart things and machine-to-machine networks. In *Proceedings of the 2014 Australasian Telecommunication Networks and Applications Conference*, ATNAC '14, pages 117–122, November 2014. doi:10.1109/ATNAC.2014.7020884. (Cited on page 92.)

[82] Hao Sun, Xiaofeng Wang, Rajkumar Buyya, and Jinshu Su. CloudEyes: Cloud-based malware detection with reversible sketch for resource-constrained internet of things (IoT) devices. *Software: Practice and Experience*, 47(3):421–441, March 2017. doi:10.1002/spe.2420. (Cited on page 78.)

[83] Takayuki Suyama, Yasue Kishino, and Futoshi Naya. Abstracting IoT Devices using Virtual Machine for Wireless Sensor Nodes. In *Proceedings of the 2014 IEEE World Forum on Internet of Things*, WF-IoT '14, pages 367–368, March 2014. doi:10.1109/WF-IoT.2014.6803190. (Cited on pages 10, 20, and 21.)

[84] Luis Miguel Vaquero and Luis Rodero-Merino. Finding Your Way in the Fog: Towards a Comprehensive Definition of Fog Computing. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, October 2014. doi:10.1145/2677046.2677052. (Cited on pages 90 and 91.)

[85] Cheng Wang and Zhiyuan Li. Parametric Analysis for Adaptive Computation Offloading. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 119–130, June 2004. doi:10.1145/996841.996857. (Cited on page 24.)

[86] Andrew Whitmore, Anurag Agarwal, and Li Da Xu. The Internet of Things–A survey of topics and trends. *Information Systems Frontiers*, 17(2):261–274, April 2015. doi:10.1007/s10796-014-9489-2. (Cited on pages 86, 87, 88, and 89.)

[87] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, April 2009. doi:10.1145/1498765.1498785. (Cited on page 25.)

[88] Rich Wolski, Selim Gurun, Chandra Krintz, and Dan Nurmi. Using Bandwidth Data To Make Computation Offloading Decisions. In *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing*, IPDPS '08, April 2008. doi:10.1109/IPDPS.2008.4536215. (Cited on page 24.)

[89] Changjiu Xian, Yung-Hsiang Lu, and Zhiyuan Li. Adaptive Computation Offloading for Energy Conservation on Battery-Powered Systems. In *Proceedings of the 2007 International Conference on Parallel and Distributed Systems*, ICPADS '07, December 2007. doi:10.1109/ICPADS.2007.4447724. (Cited on page 23.)

[90] Jie Xu and Shaolei Ren. Online Learning for Offloading and Autoscaling in Renewable-Powered Mobile Edge Computing. In *Proceedings of the 2016 IEEE Global Communications Conference*, GLOBECOM '16, December 2016. doi:10.1109/GLOCOM.2016.7842069. (Cited on page 24.)

[91] Shanhe Yi, Cheng Li, and Qun Li. A Survey of Fog Computing: Concepts, Applications and Issues. In *Proceedings of the 2015 Workshop on Mobile Big Data*, Mobidata '15, pages 37–42, June 2015. doi:10.1145/2757384.2757397. (Cited on pages 91 and 93.)

Appendix A

Background

This appendix outlines the key concepts to which this dissertation is related. The aim here is not to cover each topic exhaustively, but rather lay out important characterizations to facilitate the comprehension of the discussion presented in the text.

A.1 Constrained Devices

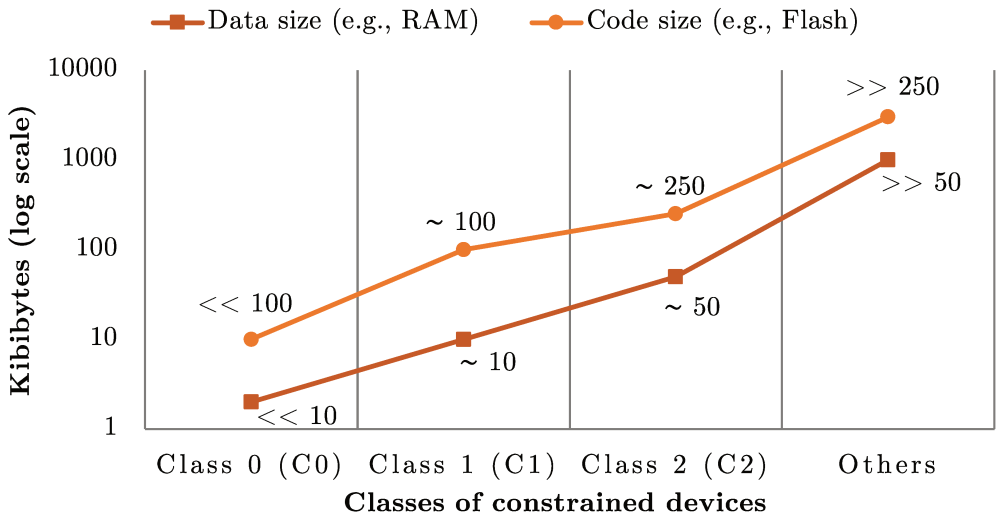
The concept of *constrained device*, or *resource-constrained device*, relates to restrictions imposed on power, energy, memory, and processing resources of small devices [15]. However, there is no established threshold for the size of these constraints. For instance, there have been cases where a system with a microcontroller such as the ATSAM4E8C [67], a device like the Intel Edison [48], nodes with CPU frequencies ranging from 512 MHz to 1 GHz and RAM going from 768 MB to 1 GB [82], and even smartphones [48, 77] were considered constrained devices.

With that in mind, we adopt the definition that is presented in the "Terminology for Constrained-Node Networks" technical report [15], which describes several terms with the intent to help the standardization work for constrained-node networks and represents the consensus of the Internet Engineering Task Force (IETF) community.

According to the report, a constrained device is one where some of the characteristics that are expected to be present on other devices currently connected to the Internet are not achievable. This is often due to cost constraints and/or physical restrictions of features such as size, weight, and available power and energy. The tight limits on power, memory, and processing resources lead to hard upper bounds on power state, code space, and processing cycles, making the optimization of energy and network bandwidth usage a primary concern in all design requirements. Although this may not be a rigorous definition, it was created based on the state of the art and it can clearly separate constrained devices from server systems, desktop and laptop computers, and more powerful mobile devices such as smartphones [15].

The IETF proposed classifications for constrained devices considering the size of available memory, energy limitations, and strategies for power usage and network communication. In this appendix, we present the proposed categorization with visual aids to better convey the expected behavior of the devices that belong to the classes in each category.

Figure A.1 shows the IETF classification for constrained devices. It is important to note that, while the boundaries presented in this classification[1] are expected to change over time, Moore's law [55] tends to be less accurate when it comes to embedded devices in comparison to personal computing settings, as gains obtained by increasing transistor count and density are more likely to be invested in reducing cost and power requirements than into continual increases in computing power [15].



Name	Data size (e.g., RAM)	Code size (e.g., Flash)	Description
Class 0	<< 10 KiB	<< 100 KiB	Sensor-like mote
Class 1	~ 10 KiB	~ 100 KiB	Can use CoAP over UDP
Class 2	~ 50 KiB	~ 250 KiB	Can use HTTP and TLS
Others	>> 50 KiB	>> 250 KiB	Can use existing protocols unchanged

Figure A.1: Approximate boundaries for the classes of constrained devices (lower = more constrained).

Class 0 (C0) contains sensor-like devices. Although they may answer keep-alive signals and send on/off or basic health indications, they most likely do not have the resources to securely communicate with the Internet directly (larger devices act as proxies, gateways, or servers) and cannot be secured or managed comprehensively in the traditional sense.

Devices that belong to *Class 1 (C1)* are quite constrained in code space and processing capabilities and cannot easily talk to other Internet nodes employing a full protocol stack (e.g., HTTP, TLS). Nonetheless, they can use the Constrained Application Protocol (CoAP) over UDP, participate in meaningful conversations without the help of a gateway node, and be integrated as fully developed peers into an IP network.

Class 2 (C2) devices are less constrained and capable of supporting most of the same protocol stacks as servers and laptop computers. They can benefit from lightweight and energy-efficient protocols and from consuming less bandwidth. Also, using the protocol

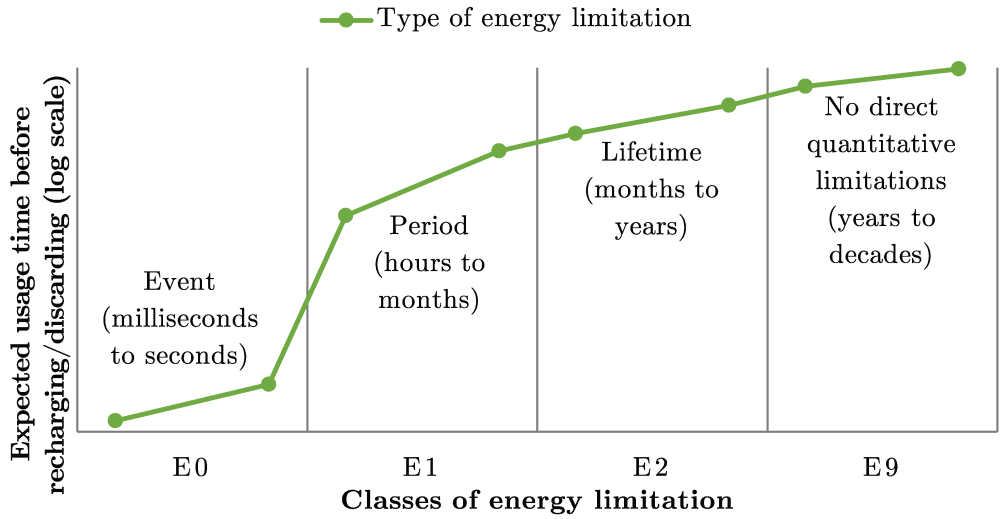
[1]The constrained devices classification was defined using the units established by the International Electrotechnical Commission (IEC), where 1 kibibyte (KiB) = 1024 bytes and 1 mebibyte (MiB) = 1024 kibibytes.

stacks defined for more constrained devices may reduce development costs and increase the interoperability of these devices.

Devices with capabilities significantly beyond that of Class 2 are left uncategorized (*Others*). They may still be constrained by a limited energy supply but can largely use existing protocols unchanged.

The graph in Figure A.1 shows the approximate order of magnitude of data (e.g., RAM) and code (e.g., Flash) memory sizes for the devices in each of these classes. We can see that there is a significant difference in resources between C0 and uncategorized devices. Moreover, we point out that this classification makes a very important distinction from one class to another in practice, as memory size has an impact on the communication protocols that can be used by the device.

The second classification, which we present in Figure A.2, pertains to limitations to the total electrical energy available before the device's energy source is exhausted.



Name	Type of energy limitation	Example power source
E0	Event	Event-base harvesting
E1	Period	Battery that is periodically recharged or replaced
E2	Lifetime	Non-replaceable primary battery
E9	No direct quantitative limitations	Permanently connected to a stable electrical power grid

Figure A.2: Expected behavior of classes of energy limitation in terms of time before recharging/discarding (lower = more constrained).

Devices that belong to *Class E0* depend on event-based harvesting and have a limited amount of energy available for a specific event (e.g., a button press in an energy-harvesting light switch).

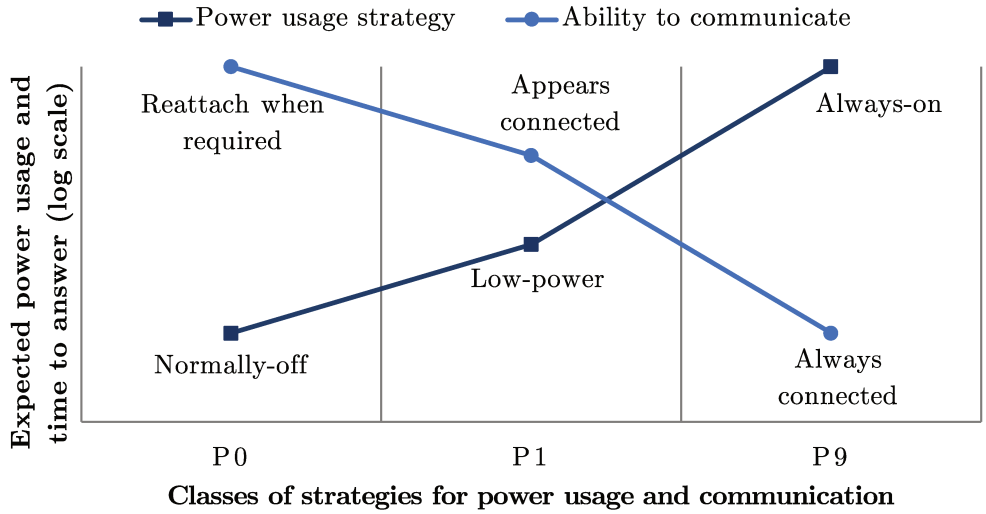
Class E1 devices have batteries that can be periodically recharged or replaced, which leads to relevant limitations on specific periods (e.g., a solar-powered device with a limited amount of energy available for the night, a device that is manually connected to a charger and has a period between recharges, or a device with a periodic primary battery replacement interval).

Devices from *Class E2* rely on non-replaceable primary batteries, thus having their total energy available limited by the usable lifetime of the device (e.g., a device that is discarded when its non-replaceable primary battery is exhausted). In a sense, many E1 devices are also E2, as the rechargeable battery has a limited number of useful recharging cycles.

Devices that are permanently connected to a stable electrical power grid have no relevant limitations with respect to energy and are placed on *Class E9*.

The graph in Figure A.2 has the expected order of magnitude for the period in which the device will work connected to the power source that describes its class before needing to be recharged or discarded. In the case of events, we would expect the interaction to last from milliseconds to seconds. As for periods, the device may work for several hours up to a few months before it needs to be recharged. Lifetime devices can be anticipated to perform from months to years before needing replacement, and devices connected to a stable power grid should work for many years or even a few decades before failing for possibly unrelated reasons. In practice, it is important to differentiate between these classes because the type of energy source used by the device affects the types of applications for which this device can be used.

Lastly, constrained devices can also be classified according to the strategies they employ for power usage and network attachment. This classification is displayed in Figure A.3.



Name	Power usage strategy	Ability to communicate
P0	Normally-off	Reattach when required
P1	Low-power	Appears connected, may have high latency
P9	Always-on	Always connected

Figure A.3: Expected behavior of classes of strategies for power usage in terms of power usage (lower = more constrained) and communication in terms of time to answer (higher = more constrained).

Class P0 devices sleep for long periods and reattach to the network as they are woken up (normally-off). If communication is infrequent, the relative increase in energy expen-

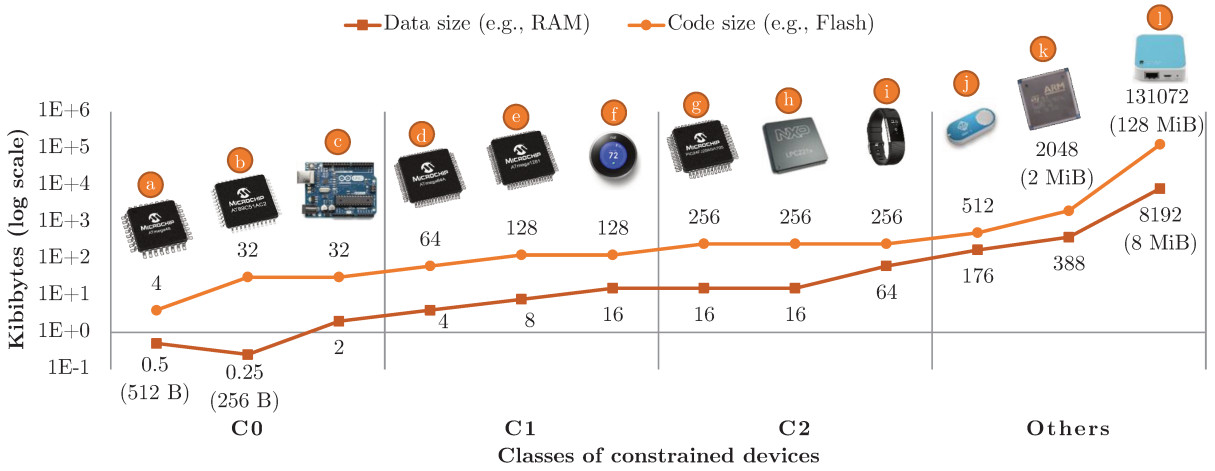
diture during reattachment may be acceptable. The main optimization goal, in this case, is to minimize the effort during the reattachment process and any resulting application communications.

The strategy of *Class P1* (low-power) is most applicable to devices that operate on a very small amount of power but still need to be able to communicate rather frequently. Some form of attachment to the network may be retained due to the small extent of time between transmissions, so even though the device may be in a sleep state, it appears connected. In this case, parameters such as the frequency of communication can be optimized and *duty cycling* (i.e., regularly switching components on and off) can be performed.

The approach used for *Class P9* (always-on) is most applicable if extreme power saving measures are not required. The device can always stay on as usual for non-constrained devices and be connected to the network. Power-friendly hardware or limiting the number of wireless transmissions, CPU speed, and other aspects may be useful for general power-saving and cooling needs.

The graph in Figure A.3 shows two different aspects of the expected behavior of these devices. First, we have the expected power usage, which grows along with the time that the device stays on. It is important to note that this is due to the communication frequency that is being assumed for each of these classes, as using a normally-off strategy for a situation where communication is very frequent may cause the device to spend a lot of power by executing continual shutting down and powering on processes. Then, we have the time necessary for the device to answer requests, which is larger in the cases where more aggressive power saving techniques are used. We expect that this delay is much more prominent on devices that need to be woken up for communication, with low-power devices also having high latency. Devices that are always connected should not present any meaningful lag unless network availability is disrupted. The placement of the points in this graph does not intend to show specific quantities but rather what we view as the general tendency in the behavior of these devices. Like the two previous classifications, the measures used by this categorization also have an important real-world impact when deciding which devices are suitable for certain applications.

Furthermore, we can think of real-world devices and how they fit into the classification of constrained devices. Figure A.4 shows twelve current devices, which are either uncategorized or belong to one of the classes C0, C1, or C2. In particular, we would like to highlight popular devices that belong to each category: for C0, we have the Arduino Uno board, which is the most-used board in the Arduino family; the 2nd-generation Nest Learning Thermostat belongs to C1 and is a smart device that programs itself automatically and helps homeowners to save energy; and, in C2, we have the Fitbit activity tracker, which assists people with exercising, eating, and sleeping habits.



	Description		Description
a	ATmega48 8-bit AVR RISC-based microcontroller	g	Microchip PIC24FJ256GA705
b	AT89C51AC2 Atmel Enhanced 8-bit microcontroller	h	NXP 60-MHz, 32-bit ARM7TDMI-S Processors LPC221x
c	ATmega328P 8-bit microcontroller (Arduino Uno Rev3)	i	STM32L433 ARM Cortex-M4 32-bit MCU+FPU (FitBit)
d	ATmega64A microchip 8-bit AVR RISC-based microcontroller	j	SAM G55G / SAM G55J Atmel | SMART ARM-based Flash MCU (Amazon dash button)
e	ATmega1281 8-bit microcontroller (Waspmote)		
f	ST Microelectronics STM32L151VB 32-MHz ARM Cortex-M3 MCU (2nd-generation Nest Learning Thermostat)	k	STM32F469xx ARM Cortex-M4 32-bit MCU+FPU
		l	Atheros 560-MHz AR9344 (TP-Link mini-router)

Figure A.4: Categorization of real-world devices according to the constrained devices classification.

A.2 Cloud Computing

As it is often the case with emerging paradigms, cloud computing elicited a debate about its definition in the early days of its popularization [3]. Since then, a well-known definition was consolidated by the USA's NIST [57]. In short, cloud computing is a model that aims at enabling the access to a shared pool of computational resources, such as networks, servers, storage, applications, and services. However, there are a few essential characteristics that must be present in order for a combination of software and hardware to be called a cloud: on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service.

On-demand self-service means that a consumer has to be able to adjust the number of resources they are allocating without requiring human interaction with the service providers. Examples of resources that must be independently controlled by the consumer are server time and network storage.

Broad network access guarantees that consumers are able to access the cloud over the network through standard mechanisms. It is important that this access be provided to different platforms, ranging from a smartphone up to a workstation.

Resource pooling enables multiple consumers to be served by the provider using a multi-tenant model, with different physical and virtual resources dynamically allocated according to demand (e.g., storage, processing, memory, and network bandwidth). Although consumers usually have no control or knowledge regarding the exact location where their computation is being performed, they may be able to specify some location details like country, state, or data center. It is important to highlight the role played by machine virtualization in this cloud characteristic, since this is the technology that allows multiple

operating systems and software configurations to concurrently be executed on the same physical machine, sharing its resources while operating under the assumption that they have sole access. For example, with the use of virtualization, each consumer may see that they have been granted access to one of the server's Central Processing Units (CPUs), while in fact all virtual machines are taking turns using the same processor [70].

Rapid elasticity is one of the key cloud features, since it enables the consumer to provision or release resources according to their needs, giving the impression that the capabilities of the system are unlimited and can be requested in any quantity at any time. Being able to elastically resize resources also enables the pay-as-you-go economic strategy, which is one of the most compelling use cases of cloud computing [3]. Figure A.5 shows a few scenarios where being able to adjust the infrastructure to demand would result in benefits to either revenue or quality of service.

Figure A.5a represents a scenario where even though the peak demand for a service has been correctly predicted, resources are being wasted during low-usage times. On the other hand, there are the situations where demand differs from the initial forecast: Figure A.5b displays a case where an initial investment for a service was made corresponding to a certain estimate of demand. However, after launching the service, the number of users dropped below the expected, resulting in several resources being underutilized. Figure A.5c shows the opposite case, where demand was higher than the system's capacity. It illustrates one of the possible outcomes for this situation, in which users become frustrated with the unavailability of the service and stop using it.

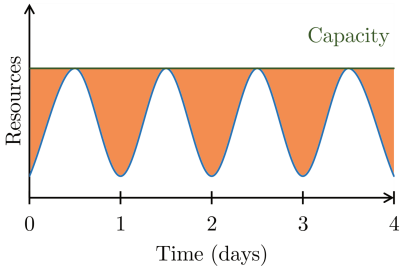
A system with rapid elasticity is depicted in Figure A.5d. Being able to dynamically provide or release resources enables the cloud consumer to adjust (automatically or not) the number of allocated machines according to demand, reducing the total price paid for using the cloud provider without compromising service quality.

Another interesting case that points out the utility of the pay-as-you-go strategy is the one where you can either use 1,000 machines running for one hour or one machine running for 1,000 hours, paying the same price for both options. This approach can be particularly favorable for organizations that perform batch analytics [3].

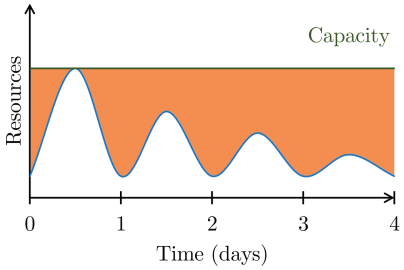
Having a *measured service* is important to enable monitoring, controlling, and reporting on the resource usage. This way, cloud systems can automatically control and optimize resource allocation according to a certain metric (e.g., storage, processing, bandwidth, and active user accounts) and this process is transparent to both the consumer and the provider.

In sum, these characteristics ensure that the cloud provides ubiquitous, convenient, on-demand access to its services in a way that resources can be quickly scaled up or down with as little management effort and service provider interaction as possible [57]. Advantages such as the appearance of infinite computing resources, the elimination of up-front commitment by consumers, and the ability to pay for needed resources on a short-term basis are some of the main differences between clouds and small and medium-sized data centers [3].

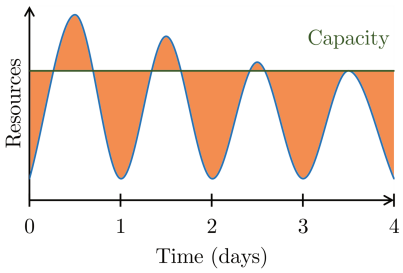
Knowing the necessary features for an infrastructure to be categorized as a cloud, we can now look into different models for cloud deployment: private cloud, community cloud, public cloud, and hybrid cloud.



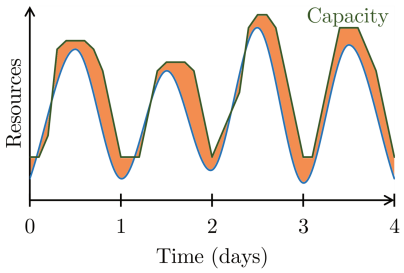
(a) Knows exact peak demand, still overprovisions during low-usage periods.



(b) Buys infrastructure at the beginning, cannot downsize if the estimate is wrong.



(c) Underprovisions, so demand drops due to low quality of service.



(d) Pay-as-you-go model allows server size customization according to demand.

Figure A.5: Possible resource management scenarios. Shadowed areas represent misused resources. Figures (a), (b), and (c) show cases where there is no elasticity, leading to resources being often under or overprovisioned. Figure (d) shows a case where it is possible to reallocate resources according to demand predictions, resulting in better usage.

Private clouds are infrastructures provisioned for exclusive use by a single organization, which may be composed of several business units that act as consumers. They may be owned, managed, and operated by the organization, a third party, or some combination of them, and may be physically located inside the company or not.

Community clouds are similar to private clouds, with the difference that they may be shared by a specific community of consumers from organizations with the same concerns regarding topics such as mission, security requirements, policy, and compliance considerations.

Unlike the previous deployment models, *public clouds* are open to the general public, and may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. Their location depends on the service provider.

Hybrid clouds are the ones that are comprised of two or more distinct cloud infrastructures that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability. One use case of hybrid clouds is employing a public cloud to capture extra tasks that cannot be easily run on a private cloud due to temporarily heavy workloads [3].

Finally, we discuss cloud service models. Services are capacities through which a cloud can collaborate with a consumer and three major types exist across the web: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS).

The *SaaS* model provides an application that is running on a cloud infrastructure. The consumer can access this application through a web browser or a program interface, and

they have no control over the underlying cloud infrastructure or application capabilities, with the exception of occasional customizable settings [57]. GoogleDocs and Salesforce are SaaS examples [70].

PaaS also does not give the consumer access to the cloud infrastructure. It does, however, allow the deployment of consumer-created or acquired applications, as long as its development tools are supported by the provider. This model may also allow the consumer to control the configuration settings for the application-hosting environment [57]. Examples for PaaS are Google App Engine and Microsoft Azure [70].

In contrast to the other models, *IaaS* enables consumers to provision processing, storage, networks, and other fundamental computing resources. They may use this model to deploy and run any kind of software, including operating systems. In this model, the consumer has permission to manage operating systems, storage, and deployed applications; and possibly has limited control of select networking components, such as host firewalls. Nonetheless, IaaS also does not give the authorization to manage and control the underlying cloud infrastructure [57]. Amazon Web Services EC2 and S3 are IaaS examples [70].

A.3 Internet of Things

When the Internet of Things (IoT) was first proposed, the focus was to create a network connecting objects that would be inter-operable and uniquely identifiable by Radio-Frequency Identification (RFID). Since then, this definition has evolved to include ideas that work on a much larger scope: the IoT must be a dynamic network infrastructure with self-configuring capabilities based on standards. Also, its components are now seen as physical or virtual "things" that have attributes as well as identities and are able to use intelligent interfaces and integrate an information network [49].

Nevertheless, the IoT is still in its infancy, thus opening it up to different definitions. For instance, the components of the IoT may also be expected to have sensing and processing capabilities, allowing everyday objects to not only communicate with one another but also work with services available on the Internet to achieve useful goals [86].

The ubiquity of objects such as mobile phones, sensors, actuators, and RFID tags will create a scenario where the majority of content will be created and received by these "things", surpassing the scale that we see nowadays on the Internet, and leading to a system that ensures connectivity anytime, anyplace, for anyone and anything [7].

The key to the innovation of the IoT lies precisely in its never-before-seen size. While there already exist embedded systems connected to the Internet in modern day appliances, the IoT presents the perspective of billions or trillions of smart objects that bring about the creation of a smart planet where people can benefit from the intelligence gathered by combining information coming from different sources [45]. This rapidly growing number of connected devices is expected to generate data at an exponential rate, soon reaching the mark of petabytes being produced daily [21].

There are several areas in today's society that can profit from the wide deployment of the IoT. For didactic purposes, we can group them into different domains: healthcare,

novelty, personal and social, smart environments, smart infrastructure, and transportation and logistics.

Healthcare applications for the IoT propose to improve the quality of life by allowing medical professionals to provide better service with reduced cost. Automatic *data collection* from patients and hospital property can reduce form processing time, as well as facilitate procedure auditing and inventory management. Patient *identification and authentication* can prevent incidents such as incorrect administration of drugs and dosages and procedures being executed on the wrong patient. It can also ease identification of newborn children in maternities. Using the IoT to identify hospital assets can avoid important instruments being misplaced or stolen, and staff authentication can improve safety inside the premises [7]. In fact, *medication* delivery can even be automated through the use of smart implants [45], which can also monitor allergic reactions and adverse interactions [86]. Continuously *monitoring* the patient's vital statuses (e.g., heart rate, blood pressure, body temperature, glucose level) and combining this information to form a comprehensive picture of their health at each moment can improve treatment and responsiveness, even if the patient is not in the hospital [7, 45, 49, 86]. Body area networks can be used to *track* daily activities for patients in assisted living and trigger message alarms in case of emergencies [45, 49]. Tracking the position of patients can also improve the workflow inside the hospital and tracking materials can prevent items from being left inside the patient during surgery [7].

Novelty IoT applications are those that depend on technologies that are currently under development but are expected to not be fully adopted in the short/medium term. Examples of this type of application are: *augmented reality*, where smartphones or other devices can provide background information about smart objects by accessing several context-dependent databases [45]; *city information models*, which monitor the status of buildings and structures such as sidewalks, sewers, and railways, providing information about environmental performance and enabling them to share resources like energy in the most efficient and cost-effective way; *enhanced game rooms*, which measure the excitement and energy of the players by sensing movement, noise, heart rate, blood pressure, etc., and then set achievements and adjust the game accordingly; and *robot taxis*, which operate with or without a driver, providing service in a timely manner, responding to real-time traffic updates in the city, and automatically returning to set points for maintenance, recharging, and cleaning [7].

There are many applications for the IoT in *personal and social* settings. They enable users to interact with other people, maintain and build social relationships, and manage their personal needs. For example, IoT-enabled mobile phones could connect to each other and *share contact information* when their owners have matching dating or friendship profiles [86]. *Missing objects* could easily be located through the use of RFID tags and a search engine that queries their last recorded position. User-defined events can also be used to notify the owner whenever the object's location matches some predefined conditions. A similar use case for this technology would be *preventing thefts*, as the stolen items would be able to inform their owner that they are being relocated without authorization. Furthermore, it would be possible for sensors to record events that happen to people or objects and build timelines that enable *historical queries*. Users would be

able to look back and see how and with whom they have spent their time and trend plots could be automatically generated to help them organize their future schedule [7]. Another feasible application would be tracking user *location* and letting them know when they are close to friends, social events, or other interesting activities [86]. Finally, IoT devices can be integrated with *social networks* such as Twitter and Facebook to help users save time by providing automatic real-time information about their activities and locations [7, 86].

The intelligence of objects in the IoT can help bring more comfort to people through the creation of *smart environments*. For instance, *homes and offices* can be made more *comfortable* by having heating that adjusts itself to the weather and lighting that is regulated by the time of the day [7]. It is also possible to *reduce wastes and costs* by tracking the usage of objects and resources like electrical energy and then use this data to make decisions such as turning off pieces of equipment when they are not needed [7, 49, 86]. Other spaces can be enhanced by the IoT as well. Mass deployment of RFID tags can enable automation in *industrial plants* [7] and reduce maintenance costs and safety issues by detecting equipment anomalies before they lead to failures [45]. *Gyms and museums* are good examples of smart leisure environments, as the IoT can help these facilities to provide more customized service. In the gym, exercise machines can change their parameters for each person and monitor their health during the training session, while the museum can adjust the climate of each of its rooms corresponding to their current exhibition [7].

We can also have *smart city infrastructures*. By spreading sensors around public areas, it is possible to monitor many factors that affect the life in the city. To name a few, we have *air quality*, *traffic*, *availability of parking spaces*, and whether *trash containers* are full or not. This information enables the development of more efficient city services, leading to a better quality of life for its inhabitants. Integrating smart objects into the *urban physical infrastructure* also contributes to improving flexibility, reliability, and efficiency in infrastructure operation, as well as increasing safety and reducing both the costs and the number of workers necessary for building and maintenance [86]. Another application is IoT-based *surveillance*, which can be used to enhance overall security for the population. Lastly, *smart grids* can help to better coordinate energy supply and demand through the use of smart meters; detecting energy usage patterns in the collected data allows the creation of plans to reduce energy consumption [45, 86].

Transportation and logistics can both benefit greatly from the IoT. If public and private means of transportation were equipped with sensors, actuators, and processing power, and perhaps even the roads themselves with tags and sensors, it would be possible to see considerable advances in technologies such as *assisted driving*. Route optimization would be improved through automatic real-time reports about traffic jams and incidents, and systems for collision avoidance and accurate monitoring of important vehicles (e.g., the ones transporting hazardous materials) could be implemented. Overall, providing drivers with more information about traffic and road conditions would lead to better navigation and safety. *Business transactions* such as purchasing tickets for public transportation could be facilitated by the use of Near-Field Communication (NFC) tags and visual markers in posters and panels, since hovering a mobile phone over a tag or pointing it towards a marker would allow users to find information about services, such as costs and available

seats, and buy tickets. NFC could also be used to *augment touristic maps*, centralizing up-to-date information about hotels, restaurants, monuments, and events in the tourist's area of interest [7]. From the supply chain standpoint, some efforts to increase the efficiency of *logistics* through the use of technologies like RFID have already been made. However, the wide deployment of IoT-connected, low-cost RFID tags and readers in packages, delivery trucks, and storehouses would enable real-time information to be shared with consumers and across companies, regardless of geographical distance [86]. Improved traceability of products would result in better information about the availability of items at stores and more precise delivery time estimates [7, 86]. It would also enhance the management of shelf space and inventory, enabling automatic stocking according to demand [7, 45, 86]. Reducing human involvement in the storage and delivery process could contribute to shorter completion times for sales as well [7, 45]. Another effect of having items that can be traceable at all times is *preventing counterfeiting* of, for example, products and currency bills [86]. In addition to tracking items, supply chains can also *monitor the environment* of perishable foods with regards to parameters like temperature, humidity, and shock in order to assure quality levels from production to consumption [7].

Figure A.6 summarizes the IoT applications we presented. However, it is important to emphasize the fact that many more uses for this technology will emerge as its development matures.

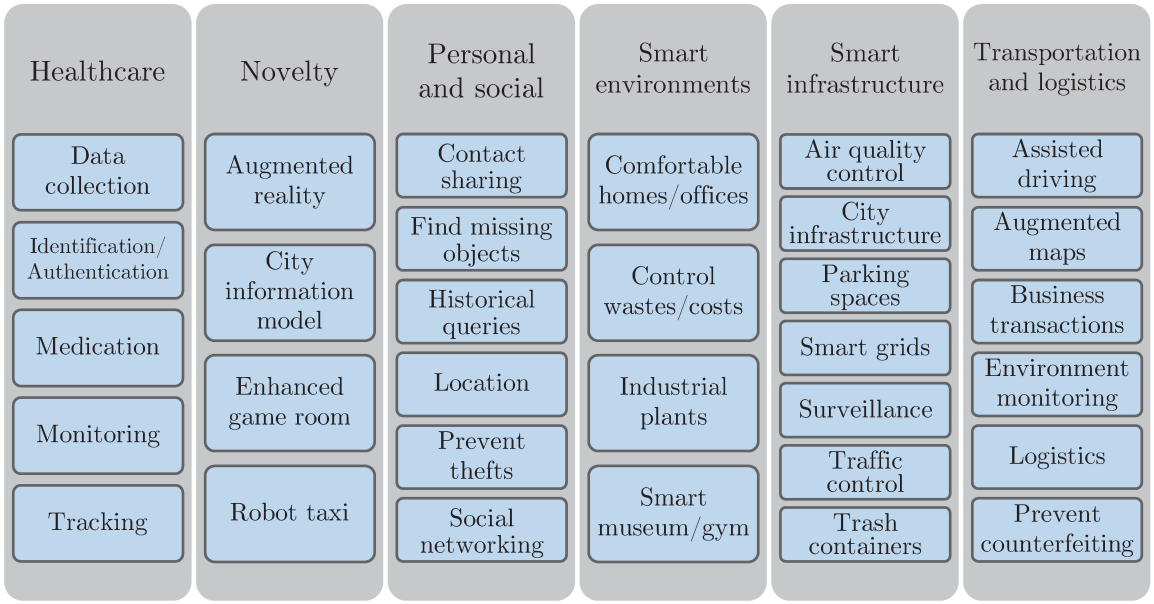


Figure A.6: Applications for the Internet of Things.

Lastly, it is important to note that the Internet of Things will work as a complement to the cloud, not a replacement. Smart objects will be able to use services provided by the cloud and the tasks attributed to each of them will depend on several factors. To name a few, we have privacy, since sending sensitive data to third-party servers risks the information not being used as agreed; energy consumption, as executing a task locally may or may not require more energy than sending its parameters to a data center; autonomy of the smart objects, seeing that it would be unfeasible to work with a huge number of objects that require frequent human intervention; response time, given that some latency

requirements will not be met when we consider the time necessary for sending data to and receiving data from the cloud; reliability, for tasks are more prone to failure if they are distributed to a wide range of machines that are not very robust; and security, considering that sending data to the cloud opens it up to attacks, possibly jeopardizing confidentiality and integrity [45].

A.4 Fog Computing

Similarly to many other emerging paradigms, fog computing is a concept that is still under discussion.

In 2012, Bonomi *et al.* [14] from Cisco Systems defined fog computing as a "highly virtualized platform that provides compute, storage, and networking services between end devices and traditional Cloud Computing Data Centers, typically, but not exclusively located at the edge of network".

The authors claim that the origins of the fog concept can be traced to early proposals to support endpoints with rich services at the edge of the network, including applications with low latency requirements, such as gaming, video streaming, and augmented reality. As fog and cloud computing share many similarities, they credit the creation of the term to the definition of fog as a cloud that is close to the ground.

In the same text, they also present a comprehensive list of defining characteristics for the fog: low latency and location awareness, widespread geographical distribution, mobility [11], very large number of nodes, the predominant role of wireless access, the strong presence of streaming and real-time applications, and heterogeneity.

In 2014, Vaquero and Rodero-Merino [84] submitted an editorial note saying that the fog should not be seen as simply an evolution of the cloud model, for doing so may lead to the false interpretation that there is little novelty in this idea. Instead, they propose that the fog should be analyzed as a result of many emerging trends in technology usage patterns (e.g., the need for network connectivity and privacy for a extremely large number of small devices) and the advance of enabling technologies (e.g., 3D micro batteries, new ways to manage networks through software, new wireless communication protocols). They added ubiquity, improved network capabilities as a hosting environment, and better support for cooperation among devices to the list of important features for fog computing.

Skala *et al.* [80] proposed an even further separation of the distributed computing hierarchy in 2015, placing a new structural layer called dew computing below the fog with the intent of scattering information among end-user devices so data can be accessed even when an Internet connection is unavailable.

In November of 2015, the OpenFog Consortium was founded with the goal of establishing an open architectural framework that can help business leaders, software developers, hardware architects, and system designers to create and maintain the hardware, software, and system elements necessary for implementing fog computing [73]. Since then, they have published many reports such as a Reference Architecture for fog computing [73] with the goal of fostering the creation of fully interoperable and secure systems. At the beginning of 2019, the OpenFog Consortium and the Industrial Internet Consortium decided to work

together under the Industrial Internet Consortium name [72].

Other entities, such as the USA's NIST, are also making efforts to help define the fog, and to this end they published a Conceptual Model [41] which places fog computing in relation to cloud and edge computing. They characterize fog computing as "a layered model for enabling ubiquitous access to a shared continuum of scalable computing resources," saying that it "facilitates the deployment of distributed, latency-aware applications and services, and consists of *fog nodes* (physical or virtual), residing between *smart* end-devices and centralized (cloud) services." These fog nodes "can be organized in clusters - either vertically (to support isolation), horizontally (to support federation), or relative to *fog nodes*' latency-distance to the *smart* end-devices." [41]. In this dissertation, we adopt this definition of fog computing.

It is important to emphasize that, much like the IoT, the fog does not intend to supersede the cloud but rather complement it [13]. For instance, the cloud has shortcomings that could be mitigated by the fog, such as requiring high bandwidth client access links and being unable to meet the latency and security requirements of certain problems [29]. Due to its characteristics, the fog is a good approach to handling large-scale distributed control systems and geo-distributed or time-sensitive mobile applications [13].

There are many situations where fog and cloud can be used in conjunction, leveraging both fog localization and cloud globalization. For example, there are the IoT applications presented in Appendix A.3. The majority of the use cases described can benefit from the fog's low latency, while cases such as data collection, historical queries, control wastes/costs, and city infrastructure can use it to filter and pre-process streaming data.

Along with all the possibilities brought by fog computing, there also come a series of challenges that stand in the way of its full realization [84, 91]:

- The possible need for centralization in applications running on devices at the edge;
- Compute and storage limitations in edge devices and how to manage computation offloading when necessary;
- Managing, maintaining connectivity, and providing reliable services using an extensive number of devices in a way that is scalable and decentralized;
- Privacy, authentication, and visibility in cases where the data is processed using third-party hardware and software;
- Standardization of communication mechanisms;
- Accountability, monitoring, and monetization for users that share their resources to host applications;
- The development of APIs and programming models that allow the portability of applications to the fog and the use of small units spread across several locations;
- Provisioning and managing resources for applications executed on the fog.

The hierarchical structure envisioned for the fog can also provide many advantages when we think about its increased potential for scalability and its ability to reduce response latency. By enabling infrastructures to continue working even when the connection to the cloud is unavailable, the fog increases their reliability and availability. As each fog

layer addresses data security and privacy, the fog allows for solutions that are overall more secure as well (e.g., it is possible to aggregate privacy-sensitive data before they even leave the network edge) [17].

Finally, in order to better place the fog in the current distributed computing scenario, we compare the features of this platform with other approaches: the cloud, cloudlets, edge computing, Mobile Cloud Computing, Mobile-Edge Computing, Wireless Sensor Networks, and Wireless Sensor and Actuator Networks.

As mentioned before, it is easy to see that the *cloud* and the fog share many similarities, with the major difference being that the latter is closer to end users. However, a more in-depth look reveals that this divergence can impact a series of characteristics in both paradigms. Firdhous, Ghazali, and Hassan [29] made such a comparison and the summary of their results is presented in Table A.1. From this analysis, we again see that fog computing excels in time-sensitive tasks, while the cloud still presents advantages for more demanding jobs such as processing batches.

Table A.1: Comparison between cloud and fog computing. Modified from [29].

Characteristic	Cloud computing	Fog computing
Latency	High	Low
Delay jitter	High	Very low
Location of server nodes	Within the Internet	Edge of the LAN
Client-server distance	Multiple hops	One hop
Security	Undefined	Can be defined
Attack on data en route	High probability	Very low probability
Location awareness	No	Yes
Geographical distribution	Centralized	Distributed
Number of server nodes	Few	Very large
Computational power of server nodes	High	Low
Support for mobility	Limited	Supported
Real-time interactions	Supported	Supported
Last mile connectivity	Leased line	Wireless

Stojmenovic [81] defines *cloudlets* as layers placed between the cloud and mobile devices which operate as "data centers in a box", that is, they are often owned by a local business and have little or no professional attention. When working with these intermediate layers, mobile devices play the role of thin clients and are able to rapidly instantiate custom virtual machines to perform the required computation. This way, they avoid part of the overhead brought by the use of the cloud, and therefore minimize their response time.

Akin to the fog, the key behind this idea is to bring the cloud closer to the data, but in the case of cloudlets, this only affects a few users at a time. Both models are also similar in regard to demanding new solutions for issues such as privacy, software licensing, and business models.

Although cloudlets were introduced in 2009 [78], years before Bonomi *et al.* [14] presented their definition of fog computing, an analysis of the characteristics of both platforms points to cloudlets being an important special case of fog computing.

The concept of *edge computing* is also very similar to the fog and sometimes they are even used interchangeably [91]. Nevertheless, edge computing was created with the focus of distributing Web applications on the Internet's edge [24], so we can consider that it represents another subset of fog computing.

Mobile Cloud Computing (MCC) is defined as an infrastructure where the data is both stored and processed outside of mobile devices. It relies on the cloud to provide processing power and data storage, thus allowing smartphone users and other mobile subscribers to benefit from mobile cloud applications and mobile computing [91].

Mobile-Edge Computing (MEC) is a natural development in the evolution of mobile base stations and can be seen as a cloud server running at the edge of a mobile network. Characterized by proximity, low latency, and high bandwidth, this paradigm enables the execution of tasks which cannot be handled by traditional infrastructures due to issues like network congestion [39, 91].

While fog computing may seem like a combination of MCC and MEC, the fact that it is applied to a broader set of devices and networks makes it stand out as a more promising and well-generalized computing paradigm in the context of the IoT [91].

Wireless Sensor Networks (WSNs) usually connect a large number of low bandwidth, low energy, low processing power, small memory motes, i.e., wireless sensor nodes that run on extremely low power or even extend their battery life by harvesting energy. Each mote on a WSN acts as sources to one or more sinks (collectors). This type of network is suitable for tasks such as sensing the environment, performing simple processing, and sending data to the sink. However, WSNs fall short in cases that not only require sensing and tracking, but also performing physical actions with the help of actuators (e.g., opening, closing, targeting, or even carrying and deploying sensors).

Actuators, which can control either a system or the measurement process itself, bring a new dimension to sensor networks, as the communication of *Wireless Sensor and Actuator Networks (WSANs)* goes from both sensors to sinks and controller nodes to actuators. Unlike networks that only connect motes, issues of stability and possible fluctuations in behavior cannot be ignored in this case, and latency and jitter become the main concerns if the system also requires that the actions be performed in near real time.

If we consider that fog computing is characterized by proximity and location awareness, geo-distribution, and hierarchical organization, we see that it is a suitable platform to support energy-constrained WSNs and WSANs [14].

Overall, the comparison between fog computing and other current paradigms leads us to the conclusion that in each case the fog either represents a concept that is broader than previous ideas or a complementary platform that enables previous approaches to thrive within the immense scale of the Internet of Things.

A.5 Mist Computing

In addition to defining fog computing, the NIST Conceptual Model report [41] also introduces the concept of *mist computing*. According to them, mist computing can be seen as a "lightweight and rudimentary form of fog computing that resides [...] at the edge of the network fabric." They also point out that it "uses microcomputers and microcontrollers to feed into fog computing nodes and potentially onward towards the centralized (cloud) computing services." Figure A.7 shows an example of the fog computing hierarchy, including how it relates to the cloud and the mist.

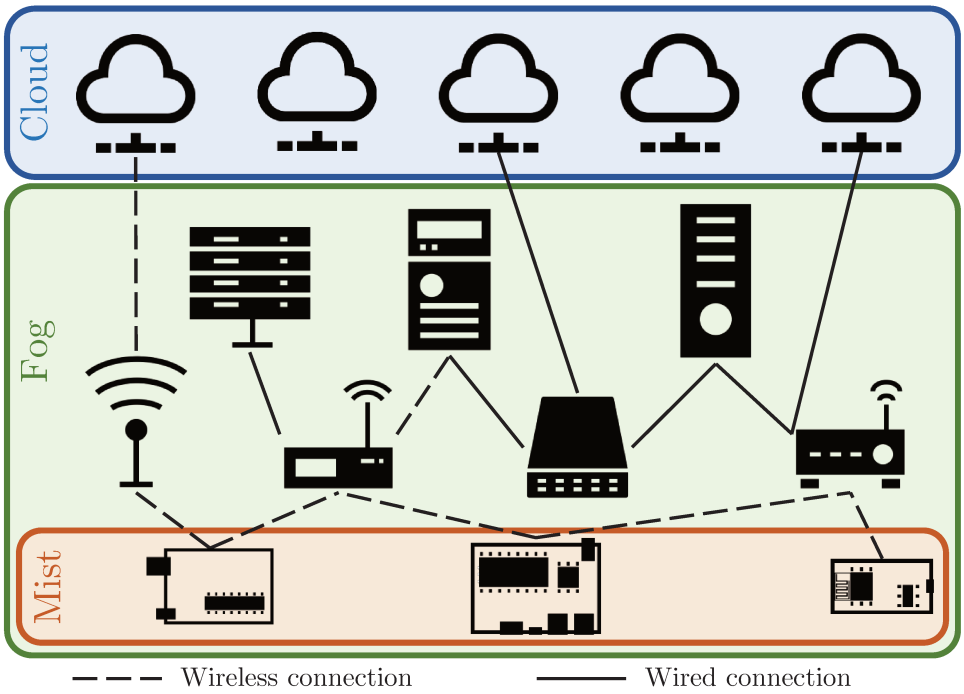


Figure A.7: Fog computing hierarchy.

The report also states that mist computing nodes are "more specialized, dedicated nodes that exhibit low computational resources" which are "placed even closer to the peripheral devices than the more powerful fog nodes they collaborate with, often sharing the same locality with the smart end-devices they service." However, we have a different proposal for the type of characteristics that should be present in mist nodes, as we consider that resource-constrained IoT devices that are part of the fog hierarchy can be seen as part of the mist. In this case, the characterization would differ in two main aspects: purpose and placement.

While they expect the purpose of mist nodes to be more specialized and dedicated, we see a valuable opportunity in leveraging resource-constrained devices for custom user code execution, as their proximity to the source of the data will allow them to perform lightweight operations on sensor streams generated by IoT devices, thus reducing network traffic and latency. In addition, they characterize the placement of the data source and the mist nodes as separated devices which might share the same locality, while we see that IoT devices usually have both sensors and resources that can be used to service user requests. This way, some IoT devices would be able to receive and execute the code

instead of sending the data to more powerful fog nodes.

With this alternative view on mist computing, we get closer to a setup that is more in line with the advances expected for the IoT, which we consider an important step toward reaching the full potential of this emerging technology.

Appendix B

Experimental Data

This appendix shows more detailed data for the experimental results discussed in Section 4.3.

Table B.1: NodeMCU execution time results in fog-prone cases.

Benchmark	**Dataset**	t **(ms)**	p **(%)**	v	n	f **(Est.)**	f **(Real)**	**Slope (Est.)**	**Slope (Real)**	**Slope Error (%)**
MinMax $[-15, 15]$	HRelHumidity	0.673551	0.000141	3,551	3,551	1.0000	1.0000	−0.9155	−0.9155	−0.0000
	HVisibility	0.614381	0.000128	3,893	0	0.0000	0.0000	−11.8819	−11.8819	−0.0000
	HWBTempC	0.629857	0.000132	3,797	760	0.2002	0.2784	−3.4911	−2.7420	+27.3195
	HWindSpeed	0.619163	0.000129	3,863	515	0.1333	0.1393	−4.5844	−4.4623	+2.7351
	Synthetic	0.635126	0.000133	3,766	2,108	0.5597	0.5536	−1.5462	−1.5611	−0.9566
MinMax $[-5, 5]$	HRelHumidity	0.673655	0.000141	3,550	3,550	1.0000	1.0000	−0.9155	−0.9155	−0.0000
	HVisibility	0.660127	0.000138	3,623	2,984	0.8236	0.8336	−1.0940	−1.0823	+1.0881
	HWBTempC	0.646680	0.000135	3,698	2,613	0.7066	0.6911	−1.2576	−1.2826	−1.9489
	HWindSpeed	0.656031	0.000137	3,646	2,920	0.8009	0.8013	−1.1227	−1.1222	+0.0440
	Synthetic	0.643882	0.000135	3,715	3,087	0.8310	0.8299	−1.0880	−1.0892	−0.1187
Outlier 16	HRelHumidity	0.715757	0.000150	3,342	73	0.0218	0.0236	−8.3408	−8.2180	+1.4952
	HVisibility	0.667375	0.000139	3,584	355	0.0991	0.0911	−5.2501	−5.4794	−4.1852
	HWBTempC	0.723876	0.000151	3,304	65	0.0197	0.0222	−8.4151	−8.2367	+2.1662
	HWindSpeed	0.719371	0.000150	3,325	121	0.0364	0.0267	−7.4110	−7.9862	−7.2024
	Synthetic	0.741199	0.000155	3,227	26	0.0081	0.0076	−9.1248	−9.1606	−0.3902
Outlier 256	HRelHumidity	2.590709	0.000542	923	41	0.0444	0.0051	−2.5043	−2.7776	−9.8403
	HVisibility	1.729101	0.000361	1,383	215	0.1555	0.1112	−2.5489	−2.8731	−11.2837
	HWBTempC	2.561749	0.000535	933	15	0.0161	0.0124	−2.7248	−2.7522	−0.9963
	HWindSpeed	2.454752	0.000513	974	5	0.0051	0.0160	−2.9291	−2.8386	+3.1893
	Synthetic	2.633129	0.000550	908	4	0.0044	0.0012	−2.7389	−2.7629	−0.8681

Table B.2: NodeMCU execution time results in cloud-prone cases.

Benchmark	**Dataset**	t **(ms)**	p **(%)**	v	n	f **(Est.)**	f **(Real)**	**Slope (Est.)**	**Slope (Real)**	**Slope Error (%)**
MinMax $[-15, 15]$	HRelHumidity	0.673551	0.001408	355	355	1.0000	1.0000	−0.5201	−0.5201	−0.0000
	HVisibility	0.614381	0.001284	389	0	0.0000	0.0000	−1.1882	−1.1882	−0.0000
	HWBTempC	0.629857	0.001317	379	210	0.5541	0.2784	−0.7058	−0.8762	−19.4563
	HWindSpeed	0.619163	0.001294	386	7	0.0181	0.1393	−1.1543	−1.0127	+13.9844
	Synthetic	0.635126	0.001328	376	206	0.5479	0.5536	−0.7053	−0.7024	+0.4010
MinMax $[-5, 5]$	HRelHumidity	0.673655	0.001408	355	355	1.0000	1.0000	−0.5201	−0.5201	−0.0000
	HVisibility	0.660127	0.001380	362	302	0.8343	0.8336	−0.5752	−0.5754	−0.0392
	HWBTempC	0.646680	0.001352	369	369	1.0000	0.6911	−0.5303	−0.6341	−16.3797
	HWindSpeed	0.656031	0.001371	364	275	0.7555	0.8013	−0.6045	−0.5883	+2.7673
	Synthetic	0.643882	0.001346	371	308	0.8302	0.8299	−0.5840	−0.5841	−0.0189
Outlier 16	HRelHumidity	0.715757	0.001496	334	10	0.0299	0.0236	−0.9897	−0.9959	−0.6239
	HVisibility	0.667375	0.001395	358	43	0.1201	0.0911	−0.9668	−0.9947	−2.8069
	HWBTempC	0.723876	0.001513	330	5	0.0152	0.0222	−0.9933	−0.9863	+0.7048
	HWindSpeed	0.719371	0.001504	332	10	0.0301	0.0267	−0.9847	−0.9880	−0.3395
	Synthetic	0.741199	0.001549	322	1	0.0031	0.0076	−0.9819	−0.9775	+0.4442
Outlier 256	HRelHumidity	2.590709	0.005415	92	0	0.0000	0.0051	−0.2818	−0.2814	+0.1445
	HVisibility	1.729101	0.003614	138	55	0.3986	0.1112	−0.3614	−0.4033	−10.3846
	HWBTempC	2.561749	0.005355	93	0	0.0000	0.0124	−0.2850	−0.2840	+0.3539
	HWindSpeed	2.454752	0.005131	97	0	0.0000	0.0160	−0.2974	−0.2960	+0.4765
	Synthetic	2.633129	0.005504	90	1	0.0111	0.0012	−0.2764	−0.2771	−0.2729

Table B.3: NodeMCU energy consumption results in fog-prone cases.

Benchmark	**Dataset**	t **(mJ)**	p **(%)**	v	n	f **(Est.)**	f **(Real)**	**Slope (Est.)**	**Slope (Real)**	**Slope Error (%)**
MinMax $[-15, 15]$	HRelHumidity	0.023170	0.000035	14,453	14,453	1.0000	1.0000	−0.9778	−0.9778	−0.0000
	HVisibility	0.021135	0.000032	15,845	0	0.0000	0.0000	−48.3565	−48.3565	−0.0000
	HWBTempC	0.021667	0.000032	15,456	3,920	0.2536	0.2784	−3.6387	−3.3376	+9.0198
	HWindSpeed	0.021299	0.000032	15,723	2,131	0.1355	0.1393	−6.3949	−6.2452	+2.3970
	Synthetic	0.021848	0.000033	15,327	8,567	0.5589	0.5536	−1.7232	−1.7393	−0.9288
MinMax $[-5, 5]$	HRelHumidity	0.023174	0.000035	14,451	14,451	1.0000	1.0000	−0.9778	−0.9778	−0.0000
	HVisibility	0.022708	0.000034	14,747	12,184	0.8262	0.8336	−1.1787	−1.1685	+0.8687
	HWBTempC	0.022246	0.000033	15,054	11,189	0.7433	0.6911	−1.3071	−1.4028	−6.8176
	HWindSpeed	0.022567	0.000034	14,839	11,701	0.7885	0.8013	−1.2336	−1.2145	+1.5716
	Synthetic	0.022150	0.000033	15,119	12,530	0.8288	0.8299	−1.1759	−1.1743	+0.1301
Outlier 16	HRelHumidity	0.024622	0.000037	13,601	324	0.0238	0.0236	−20.8708	−20.9521	−0.3880
	HVisibility	0.022958	0.000034	14,587	1,470	0.1008	0.0911	−8.1144	−8.8072	−7.8668
	HWBTempC	0.024901	0.000037	13,448	297	0.0221	0.0222	−21.5284	−21.4534	+0.3493
	HWindSpeed	0.024746	0.000037	13,532	409	0.0302	0.0267	−18.3694	−19.6518	−6.5253
	Synthetic	0.025497	0.000038	13,134	90	0.0069	0.0076	−31.4457	−30.6958	+2.4432
Outlier 256	HRelHumidity	0.089120	0.000133	3,757	47	0.0125	0.0051	−10.0289	−10.8308	−7.4044
	HVisibility	0.059481	0.000089	5,630	531	0.0943	0.1112	−6.5567	−5.9035	+11.0641
	HWBTempC	0.088124	0.000132	3,800	15	0.0039	0.0124	−11.0896	−10.1371	+9.3965
	HWindSpeed	0.084443	0.000126	3,965	52	0.0131	0.0160	−10.4449	−10.1371	+3.0363
	Synthetic	0.090580	0.000135	3,697	5	0.0014	0.0012	−11.1133	−11.1277	−0.1295

Table B.4: NodeMCU energy consumption results in cloud-prone cases.

Benchmark	**Dataset**	t **(mJ)**	p **(%)**	v	n	f **(Est.)**	f **(Real)**	**Slope (Est.)**	**Slope (Real)**	**Slope Error (%)**
MinMax $[-15, 15]$	HRelHumidity	0.023170	0.001384	361	361	1.0000	1.0000	−0.5244	−0.5244	−0.0000
	HVisibility	0.021135	0.001262	396	0	0.0000	0.0000	−1.2089	−1.2089	−0.0000
	HWBTempC	0.021667	0.001294	386	217	0.5622	0.2784	−0.7091	−0.8878	−20.1222
	HWindSpeed	0.021299	0.001272	393	7	0.0178	0.1393	−1.1745	−1.0278	+14.2665
	Synthetic	0.021848	0.001305	383	209	0.5457	0.5536	−0.7139	−0.7099	+0.5616
MinMax $[-5, 5]$	HRelHumidity	0.023174	0.001384	361	361	1.0000	1.0000	−0.5244	−0.5244	−0.0000
	HVisibility	0.022708	0.001356	368	304	0.8261	0.8336	−0.5831	−0.5806	+0.4365
	HWBTempC	0.022246	0.001329	376	376	1.0000	0.6911	−0.5346	−0.6403	−16.5127
	HWindSpeed	0.022567	0.001348	370	280	0.7568	0.8013	−0.6097	−0.5936	+2.7142
	Synthetic	0.022150	0.001323	377	312	0.8276	0.8299	−0.5901	−0.5894	+0.1345
Outlier 16	HRelHumidity	0.024622	0.001470	340	10	0.0294	0.0236	−1.0070	−1.0128	−0.5816
	HVisibility	0.022958	0.001371	364	43	0.1181	0.0911	−0.9836	−1.0105	−2.6609
	HWBTempC	0.024901	0.001487	336	5	0.0149	0.0222	−1.0106	−1.0032	+0.7445
	HWindSpeed	0.024746	0.001478	338	10	0.0296	0.0267	−1.0019	−1.0048	−0.2919
	Synthetic	0.025497	0.001523	328	1	0.0030	0.0076	−0.9990	−0.9945	+0.4576
Outlier 256	HRelHumidity	0.089120	0.005322	93	0	0.0000	0.0051	−0.2867	−0.2863	+0.1470
	HVisibility	0.059481	0.003552	140	55	0.3929	0.1112	−0.3675	−0.4100	−10.3520
	HWBTempC	0.088124	0.005263	95	0	0.0000	0.0124	−0.2899	−0.2889	+0.3601
	HWindSpeed	0.084443	0.005043	99	0	0.0000	0.0160	−0.3026	−0.3011	+0.4848
	Synthetic	0.090580	0.005410	92	1	0.0109	0.0012	−0.2812	−0.2820	−0.2709

Table B.5: Raspberry Pi 3 energy consumption results in fog-prone cases.

Benchmark	**Dataset**	**Instruction Count**	t **(mJ)**	p **(%)**	v	n	f **(Est.)**	f **(Real)**	**Slope (Est.)**	**Slope (Real)**	**Slope Error (%)**
MinMax $[-15, 15]$	HRelHumidity	3,997,730	0.003321	0.000000	23,765	23,765	1.0000	1.0000	−0.9997	−0.9997	−0.0000
	HVisibility	2,687,010	0.002232	0.000000	35,358	0	0.0000	0.0000	−5395.3493	−5395.3493	−0.0000
	HWBTempC	3,047,723	0.002532	0.000000	31,174	7,744	0.2484	0.2784	−4.0222	−3.5891	+12.0664
	HWindSpeed	2,869,570	0.002384	0.000000	33,109	4,541	0.1372	0.1393	−7.2806	−7.1695	+1.5502
	Synthetic	3,285,947	0.002730	0.000000	28,914	16,086	0.5563	0.5536	−1.7967	−1.8058	−0.4997
MinMax $[-5, 5]$	HRelHumidity	3,997,730	0.003321	0.000000	23,765	23,765	1.0000	1.0000	−0.9997	−0.9997	−0.0000
	HVisibility	3,779,590	0.003140	0.000000	25,137	20,454	0.8137	0.8336	−1.2286	−1.1993	+2.4413
	HWBTempC	3,537,571	0.002939	0.000000	26,857	19,474	0.7251	0.6911	−1.3787	−1.4465	−4.6872
	HWindSpeed	3,737,250	0.003105	0.000000	25,422	20,039	0.7883	0.8013	−1.2682	−1.2476	+1.6506
	Synthetic	3,584,428	0.002978	0.000000	26,506	22,025	0.8309	0.8299	−1.2031	−1.2047	−0.1299
Outlier 16	HRelHumidity	6,972,197	0.006372	0.000001	12,388	273	0.0220	0.0236	−44.3135	−41.3824	+7.0831
	HVisibility	6,035,976	0.005516	0.000001	14,309	1,441	0.1007	0.0911	−9.8850	−10.9245	−9.5154
	HWBTempC	7,285,119	0.006658	0.000001	11,856	258	0.0218	0.0222	−44.8151	−43.8595	+2.1788
	HWindSpeed	7,045,547	0.006439	0.000001	12,259	369	0.0301	0.0267	−32.6425	−36.7553	−11.1898
	Synthetic	7,384,825	0.006749	0.000001	11,695	80	0.0068	0.0076	−135.1194	−122.1042	+10.6591
Outlier 256	HRelHumidity	49,584,961	0.045317	0.000006	1,741	41	0.0235	0.0051	−36.6140	−112.4953	−67.4529
	HVisibility	30,115,961	0.027524	0.000003	2,868	459	0.1600	0.1112	−6.1604	−8.8124	−30.0942
	HWBTempC	49,178,545	0.044946	0.000006	1,756	15	0.0085	0.0124	−81.4756	−61.9114	+31.6004
	HWindSpeed	46,647,122	0.042632	0.000005	1,851	52	0.0281	0.0160	−31.6132	−51.1218	−38.1610
	Synthetic	50,492,036	0.046146	0.000006	1,710	4	0.0023	0.0012	−162.0664	−197.3519	−17.8795

Table B.6: Raspberry Pi 3 energy consumption results in cloud-prone cases.

Benchmark	**Dataset**	**Instruction Count**	t **(mJ)**	p **(%)**	v	n	f **(Est.)**	f **(Real)**	**Slope (Est.)**	**Slope (Real)**	**Slope Error (%)**
MinMax $[-15, 15]$	HRelHumidity	3,997,730	0.003321	0.001683	594	594	1.0000	1.0000	−0.4755	−0.4755	−0.0000
	HVisibility	2,687,010	0.002232	0.001131	883	0	0.0000	0.0000	−1.3488	−1.3488	−0.0000
	HWBTempC	3,047,723	0.002532	0.001283	779	482	0.6187	0.2784	−0.6851	−0.8934	−23.3159
	HWindSpeed	2,869,570	0.002384	0.001208	827	12	0.0145	0.1393	−1.2403	−1.0741	+15.4754
	Synthetic	3,285,947	0.002730	0.001383	722	403	0.5582	0.5536	−0.6827	−0.6848	−0.3149
MinMax $[-5, 5]$	HRelHumidity	3,997,730	0.003321	0.001683	594	594	1.0000	1.0000	−0.4755	−0.4755	−0.0000
	HVisibility	3,779,590	0.003140	0.001591	628	491	0.7818	0.8336	−0.5480	−0.5329	+2.8348
	HWBTempC	3,537,571	0.002939	0.001489	671	671	1.0000	0.6911	−0.5061	−0.5998	−15.6321
	HWindSpeed	3,737,250	0.003105	0.001573	635	484	0.7622	0.8013	−0.5576	−0.5457	+2.1783
	Synthetic	3,584,428	0.002978	0.001509	662	550	0.8308	0.8299	−0.5495	−0.5498	−0.0523
Outlier 16	HRelHumidity	6,972,197	0.006372	0.003229	309	10	0.0324	0.0236	−0.4655	−0.4674	−0.4062
	HVisibility	6,035,976	0.005516	0.002795	357	43	0.1204	0.0911	−0.5122	−0.5200	−1.5042
	HWBTempC	7,285,119	0.006658	0.003374	296	5	0.0169	0.0222	−0.4488	−0.4478	+0.2404
	HWindSpeed	7,045,547	0.006439	0.003263	306	10	0.0327	0.0267	−0.4606	−0.4619	−0.2767
	Synthetic	7,384,825	0.006749	0.003420	292	1	0.0034	0.0076	−0.4455	−0.4447	+0.1873
Outlier 256	HRelHumidity	49,584,961	0.045317	0.022963	43	0	0.0000	0.0051	−0.0664	−0.0664	+0.0341
	HVisibility	30,115,961	0.027524	0.013947	71	55	0.7746	0.1112	−0.1009	−0.1081	−6.6915
	HWBTempC	49,178,545	0.044946	0.022775	43	0	0.0000	0.0124	−0.0670	−0.0669	+0.0832
	HWindSpeed	46,647,122	0.042632	0.021603	46	0	0.0000	0.0160	−0.0706	−0.0706	+0.1132
	Synthetic	50,492,036	0.046146	0.023383	42	1	0.0238	0.0012	−0.0652	−0.0652	−0.1471

Appendix C

Publications

This appendix presents a list of journal and conference articles published during this Ph.D. research.

Publications which resulted from this project:

1. **Flávia Pisani**, Vanderson Martins do Rosario, Edson Borin. Fog vs. Cloud Computing: Should I Stay or Should I Go?. *Future Internet*, 11(2):27–32, February 2019. doi:10.3390/fi11020034.

2. **Flávia Pisani** and Edson Borin. Fog vs. Cloud Computing: Should I Stay or Should I Go?. In: *Proceedings of the Workshop on INTelligent Embedded Systems Architectures and Applications*, INTESA '18, Turin, Italy, pages 27–32, October 2018. doi:10.1145/3285017.3285026.

3. **Flávia Pisani**, Jeferson Rech Brunetta, Vanderson Martins do Rosario, and Edson Borin. Beyond the Fog: Bringing Cross-Platform Code Execution to Constrained IoT Devices. In *Proceedings of the 29th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '17, pages 17–24, October 2017. doi:10.1109/SBAC-PAD.2017.10.

Publications which resulted from previous studies and from the collaboration with other students and researchers as a part of their work:

1. Tiago Lobato Gimenes, **Flávia Pisani**, and Edson Borin. Evaluating the Performance and Cost of Accelerating Seismic Processing with CUDA, OpenCL, OpenACC, and OpenMP. In *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '18, Vancouver, Canada, pages 399–408, May 2018. doi:10.1109/IPDPS.2018.00050.

2. Vanderson Martins do Rosario, **Flávia Pisani**, Alexandre Rodrigues Gomes, and Edson Borin. Fog-Assisted Translation: Towards Efficient Software Emulation on Heterogeneous IoT Devices. In *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops*, IPDPSW '18, Vancouver, Canada, pages 1268–1277, May 2018. doi:10.1109/IPDPSW.2018.00196.

3. **Flávia Pisani**, Daniel Carlos Guimarães Pedronette, Ricardo da Silva Torres, and Edson Borin. Contextual Spaces Re-Ranking: accelerating the Re-sort Ranked Lists step on heterogeneous systems. Concurrency and Computation-Practice & Experience, 29(22):e3962, November 2017. doi:10.1002/cpe.3962.

4. Rafael Schmid, **Flávia Pisani**, Edson Borin, and Edson Cáceres. An Evaluation of Segmented Sorting Strategies on GPUs. In *Proceedings of the 2016 IEEE 18th International Conference on High Performance Computing and Communications*, HPCC '16, Sydney, Australia, pages 1123–1130, December 2016. doi:10.1109/HPCC-SmartCity-DSS.2016.0158.

5. Edson Borin, Caian Benedicto, Ian Liu Rodrigues, **Flávia Pisani**, Martin Tygel, and Mauricio Breternitz. PY-PITS: A Scalable Python Runtime System for the Computation of Partially Idempotent Tasks. In *Proceedings of the 2016 International Symposium on Computer Architecture and High Performance Computing Workshops*, SBAC-PADW '16, Los Angeles, USA, pages 7–12, October 2016. doi:10.1109/SBAC-PADW.2016.10.

6. Hércules Cardoso da Silva, **Flávia Pisani**, and Edson Borin. A Comparative Study of SYCL, OpenCL, and OpenMP. In *Proceedings of the 2016 International Symposium on Computer Architecture and High Performance Computing Workshops*, SBAC-PADW 2016, Los Angeles, USA, pages 61–66, October 2016. doi:10.1109/SBAC-PADW.2016.19.

7. **Flávia Pisani**, Daniel Carlos Guimarães Pedronette, Ricardo da Silva Torres, and Edson Borin. Improving the Performance of the Contextual Spaces Re-Ranking Algorithm on Heterogeneous Systems. In *Anais do XVI Simpósio em Sistemas Computacionais de Alto Desempenho*, WSCAD '15, Florianópolis, Brazil, pages 132–143, October 2015. URL: http://www.lbd.dcc.ufmg.br/colecoes/wscad/2015/012.pdf.