Automating Computational Placement for the Internet of Things

Peter Michalák

School of Computing
Newcastle University

In Partial Fulfilment of the Requirements for the Degree of
Doctor of Philosophy

June 2020

In memory of my grandfather

wise, uplifting, visionary man ..

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Peter Michalák

June 2020

Acknowledgements

I would like to thank Professor Paul Watson for his guidance, expertise and insights, always positive attitude, and encouragement throughout my PhD. I would not be standing here without him. My deep gratitude goes to Dr Sarah Heaps for her support and for never making me afraid to ask even the simplest statistical question; and to Professor Mike Trenell, I enjoyed every walking meeting discussing leading-edge medical discoveries or business management. I feel truly lucky to have this supervisory team.

I would like to thank Dr Matt Forshaw for his guidance, being a great travel companion, and a hard-working role model. Thank-you Dr Devki Jha for your expertise, determination, and patience whilst working with me.

During this journey I was fortunate to meet many inspiring people that I'm grateful for: Professor Mike Catt, I have always looked forward to our conversations about exciting and visionary ideas; Dr Rawaa Quasha, the conversations helped me understand life better, and her constant encouragement and strength affected me greatly; Dr Antonia Kontaratou for always listening, encouraging and looking at the bright side, I'm very lucky to call her my friend; Dr Stuart Wheater for never becoming tired of hearing my 'moon shot' IoT ideas, I'm grateful for his continuous supply of useful tips and handy electronic components and his kindness.

I'd like to thank all my PhD colleagues from Centre for Doctoral Training in Cloud Computing for Big Data, especially Dr Mario Parreño, Dr Shane Halloran, Dr Richard Cloete – I'm thankful that our centre brought us together so we can learn from each other. Thank you, Barry Hodgson, for all your support and an always friendly reality-check.

Thank you, Jenny Brady, for your guidance through the three-month Action for Impact Fellowship. Your uplifting can-do attitude, expertise and mentorship skills made it an incredible, very much outside of my comfort zone, experience. Thank you, Vlad González Zelaya, for your friendly attitude, and never-ending well of positivity, and all the neat LaTeX tricks. Thank you Oonagh McGee and Jen Wood for making everything run smoothly, always happy to help, and having genuine care for our wellbeing.

Thank you, Joanne Allison, for proofreading this manuscript. I'm also very grateful to every 'focusmate' that I shared the virtual coworking session with. A wonderful, well-intentioned, and supportive community.

I would like to thank my family for all the support throughout my research work, especially my mum, who has seen me as a successful candidate even when I doubted myself. Thanks to my dad, my sisters and my goddaughters Nicole and Eliška, who I have thought of every time I needed a reason to smile. Thank you all for the energy that kept me going forward.

Abstract

The *PATH2iot* platform presents a new approach to distributed data analytics for Internet of Things applications. It automatically partitions and deploys stream-processing computations over the available infrastructure (e.g. sensors, field gateways, clouds and the networks that connect them) so as to meet non-functional requirements including network limitations and energy. To enable this, the user gives a high-level declarative description of the computation as a set of Event Processing Language queries. These are compiled, optimised, and partitioned to meet the non-functional requirements using a combination of distributed query processing techniques that optimise the computation, and cost models that enable *PATH2iot* to select the best deployment plan given the non-functional requirements. This thesis describes the resulting *PATH2iot* system, illustrated with two real-world use cases. First, a digital healthcare analytics system in which sensor battery life is the main non-functional requirement to be optimized. This shows that the tool can automatically partition and distribute the computation across a healthcare wearable, a mobile phone and the cloud - increasing the battery life of the smart watch by 453% when compared to other possible allocations. The energy cost of sending messages over a wireless network is a key component of the cost model, and we show how this can be modelled. Furthermore, the uncertainty of the model is addressed with two alternative approaches: one frequentist and one Bayesian The second use case is one in which an acoustic data analytics for transport monitoring is automatically distributed so as enable it to run over a low-bandwidth LORA network connecting the sensor to the cloud. Overall, the paper shows how the *PATH2iot* system can automatically bring the benefits of edge computing to the increasing set of IoT applications that perform distributed data analytics.

Table of contents

List of figures

List of tables

List of acronyms

API	Application Programming Interface
AWS	Amazon Web Services
BLE	Bluetooth Low Energy
CEP	Complex Event Processing
CGM	Continous Glucose Monitors
CI	Confidence Intervals
CQL	Continuous Query Language
DAG	Directed Acyclic Graph
DSMS	Distributed Stream Management Systems
EI	Energy Impact
EPL	Event Processing Language
FGM	Flash Glucose Monitors
GCP	Google Cloud Platform
IoT	Internet of Things
JSON	Javascript Object Notation
LPWAN	Low Power Wide Area Network
NFC	Near-Field Communication
REST	Representational State Transfer
SDK	Software Development Kit
SPE	Stream Processing Engine
SQL	Structured Query Language
UDF	User Defined Function

CHAPTER 1
INTRODUCTION

Humans have always striven to understand the world they live in and to assist in this they have collected data about their environment. One example of data collection that has had a significant global impact on society is the case of Tycho Brahe, the astronomer, who made meticulous astronomical observations in late 16^{th} century. These were analysed by Johannes Kepler and led to the discovery of laws of planetary motion.

Four hundred years later, we have increasingly turned to computers to make sense of the world in which we live. Examples range from smart thermostats that use a temperature sensor to control home heating, to cameras in autonomous vehicles whose outputs are analysed to locate and identify nearby objects. Irrespective of the source, data needs to be analysed in order to derive actionable insights from it – collecting the data is not enough. This has become more challenging with the rise of "Big Data". Over the past decades, the volume of data collected and stored has increased dramatically, creating analytic challenges. One extreme example is the Large Hadron Collider (LHC) – the world's largest and most powerful particle collider built by the European Organization for Nuclear Research (CERN) – that requires more than 30 petabytes of data storage every year for data captured from its experiments. As well as data "Volume", analytics techniques have also been stretched by the increasing number of real-time data sources, including sensors and apps. This has led to the growing use of the term "Velocity" to refer to the rate at which data is being produced. This is an issue for the LHC: the total data that it generates has to be heavily filtered in real-time, while the experiments are running, in order for it to be processed by the rest of the data analytical pipeline [1].

The "Variety" of data — the range of different types of data that need to be merged in order to derive insights through analysis [110] – also presents challenges. Two additional Vs are also used to describe what has become known as Big Data. The first is "Veracity", a measure of trust or representativeness of captured data, while the second is the "Value" that can be gained by

the analysis and that might dissipate over time, if not derived shortly after or during the data collection.

1.1 Internet of Things

> We need to empower computers with their own means of gathering information, so they can see, hear and smell the world for themselves, in all its random glory.
>
> -Kevin Ashton [41]

Kevin Ashton is often credited as the first person to coin the term 'Internet of Things' in order to capture the attention of management at a presentation he gave at a Procter & Gamble meeting in 1999. Two decades later, the IoT passed the peak of expectations as seen from the hype cycle curve [105, 2] in Figure 1.1, assembled from Gartner's data. The term IoT appeared on the curve for the first time in 2011 with the expectation that it would reach a plateau of productivity in more than 10 years. In the years 2014 and 2015, the Internet of Things reached the Peak of Inflated Expectations. The term 'IoT platform' – the infrastructure that focusses specifically on managing data flows and processing within IoT environments – reached the peak of expectations in 2018. Afterwards, it started to decline into the "Trough of Disillusionment" with expectations that it will reach Plateau of Productivity in five to 10 years. Already many IoT applications have found their usefulness in a range of fields from lifestyle monitors and wellness recommendation systems [43], smart city use cases such as pedestrian and car detection [61] and understanding human behaviour related to traffic prediction [86], through smart energy management systems [74] and smart farming [93].

According to the McKinsey management consulting firm, 127 new devices connect to the Internet every second [3]. This is expected to grow, with the potential economic impact of $11.1 trillion ($10^{12}$) by 2025. Statista, a market research and business intelligence portal, estimates a total of 20.41 billion (10^9) of installed IoT devices by this year (2020) [4]. Future growth estimates might have to be re-adjusted as the world finds new ways of working and recovers from the Covid-19 health and financial crisis of 2020.

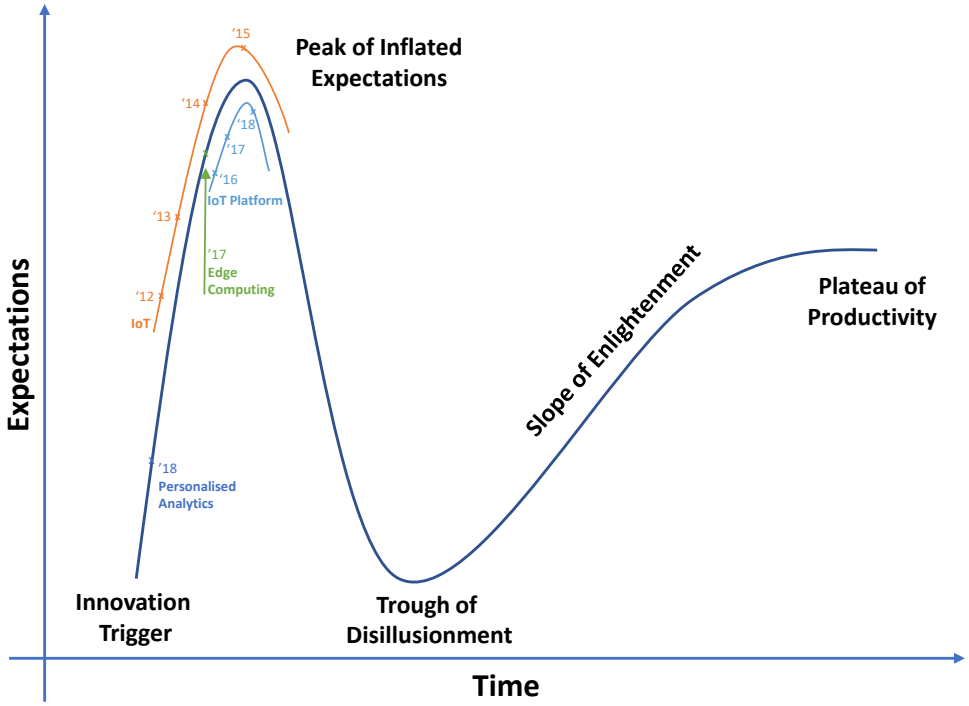


Fig. 1.1 Gartner Hype Cycle : Selected Emerging Technologies

> In CEP an 'event' is an object that can be subjected to computer processing. It signifies, or is a record of, an activity that has happened.
>
> *-David Luckham* [108]

The quote from David Luckham defines an event as an object that serves as a record of a specific activity that has occurred and can be analysed by a digital system. Events are typically passed within these systems as messages that carry the data from which the information is derived. In recent years, the possibility of placing connected sensors ubiquitously within the environment created opportunities for the stream of events to flow from their source and be processed in a data analytics pipeline so as to distil useful information from them. Complex Event Processing (CEP) was introduced in the late 1990s as a technology to extract information from distributed message-based systems. It provides a way to build applications that filter, transform and aggregate sensor data [109].

1.1.1 Centralised vs Distributed Analytics

Extracting value from IoT generated data can be challenging, especially when it exhibits the high velocity and/or high volume commonly referred to as Big Data. In order to meet this challenge, stream processing engines have been created, especially for the cloud (e.g. Apache Spark [5] and Apache Storm [6]).

While these systems are highly efficient, this cloud-centric approach can present major problems for some important stream processing scenarios as they require data collected by sensors to be sent to the cloud for analysis. The limited bandwidth of networking from the sensor to the cloud can cause problems, as can the drain on sensor battery life due to the energy cost of sending messages. One way to address these challenges is to exploit the fact that in modern distributed computer systems, there is a range of options for where to deploy the operations that make up the data analytics pipeline. Typically, the IoT sensors will have some – though often limited – event processing capabilities [137]. In an IoT system, the sensor may then pass data on to an intermediate device such as a field gateway that can also perform some analysis before passing events on to the cloud.

One of the main reasons for distributing the processing in this way is to aggregate or filter data before it is transmitted to the cloud for final analysis. This so-called "edge computing" approach can reduce the required network bandwidth requirements and lower energy costs.

We can illustrate the opportunities and challenges using a real medical application – this will act as a running example through the thesis. We have worked with medical researchers on a healthcare application that uses wearable sensors to monitor the activity and glucose levels of type II diabetes patients and alert them to a possible hyperglycaemic episode before their health is endangered, A Continuous Glucose Monitor periodically collects accurate glucose measurements from a patient. These must be analysed in order to give short-term forecasts: if patients' glucose levels are predicted to exceed the upper threshold for a healthy individual, a behavioural prompt (text message/notification) can be issued to the user, asking them to heighten their physical activity in order to attenuate the upward trajectory of their glucose levels. Figure 1.2 presents an example of glucose and activity monitoring from this project, with an illustration of a behavioural prompt being issued based on the forecast glucose levels. When the type II diabetes patient recovers from the hyperglycaemic episode, the impact of their activity levels is estimated and feedback can be sent to the patient – a comparison between modelled (solid line) and actual glucose readings (dotted line). The impact of increased activity on glucose metabolic response is an active research area [80, 56].

To enable the incorporation of activity analysis into the model, the patient uses a healthcare wearable: a watch-like device that incorporates an accelerometer whose output can be processed to give a measure of their activity level. The wearable communicates data over the Bluetooth Low Energy networking protocol to a phone, which then sends the data over a mobile network to the cloud. If data is not partially processed in situ, then every reading taken by the glucose and activity sensors must be transmitted to the cloud (via the phone) for analysis. As sending a

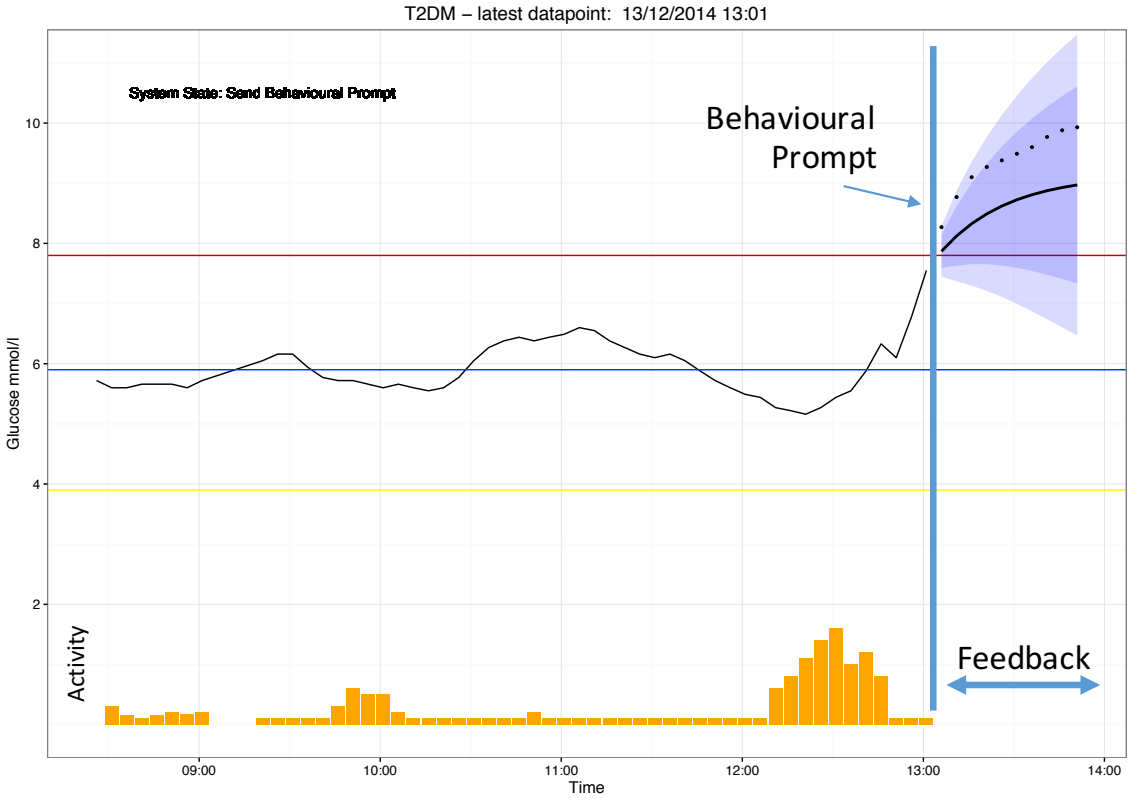


Fig. 1.2 Activity and glucose data processing with multivariate time series forecasting.

message has an energy cost, this approach severely affects the battery life of the wearable, and the phone.

Distributing the analytics can address this problem, if most of the messages are not required by the detailed analysis – for example, if only certain activity levels are of interest – then they could be filtered out at the sensor or phone. Further gains may be made by discarding unwanted data before it is sent to the cloud (e.g. the sub-fields of sensor readings that are not used in any computation), and by performing some simple analysis and data aggregation (e.g. averaging) on the wearable or phone. Overall, exploiting the limited computational power at the edge could:

- increase the battery life of the wearable and phone,
- reduce the number of messages transmitted over the mobile network, so reducing the required bandwidth and possibly data charges,
- reduce the load on the cloud-based processing, which could be a serious issue if the application becomes popular, resulting in tens of millions of wearables streaming data to the cloud.

Unfortunately, while this approach has been recognised by the growth of edge computing for IoT it raises a major problem. Building distributed data analytics systems can be extremely

challenging for a variety of reasons. One of the main ones is heterogeneity – analytics has to be distributed over a variety of platforms, each with very different capabilities. A typical distributed analytics configuration will consist of a sensor streaming data to the cloud via a field gateway or mobile phone. Clouds have seemingly infinite computing power and storage capacity, but sensors have very limited capabilities; mobile phones or field gateways fall in the middle of this capability range. As well as the differences in computational capabilities, each platform has a different development environment – clouds offer support for a variety of stream processing middleware (e.g. Apache Storm [6] and Apache Spark [5], while a mobile phone developer might use Objective C, Swift, Java or Kotlin. A sensor will typically be programmed in C.

To further complicate matters, there is also great diversity in the networks connecting the platforms. These range from GigaBit Ethernet in the Cloud to the much lower bandwidth protocols used to connect a sensor to a phone or field gateway (e.g. BLE [81] or Zigbee [77]). Finally, the system designer must take into account a set of non-functional requirements, ranging over performance, security, dependability and energy (e.g. the required battery-life of portable devices).

One way to build distributed data analytics applications that meet these challenges is to assemble a talented team of developers with expertise across all these types of platforms, and a knowledge of how to build systems that meet the required non-functional requirements. Unfortunately, these skills are in short supply.

In this thesis we therefore present an alternative approach that simplifies the design and implementation of efficient systems for processing streaming data. The *PATH2iot* is distributed stream processing framework that automatically partitions and distributes processing across the available components - considering it holistic as demonstrated in the case of the running example, the wearable, phone and cloud, depending on the computational capabilities of the platforms, and the non-functional requirements (e.g. battery life and network capabilities).

As a proof of concept, the system generates software components that are deployed on each platform automatically, demonstrating its capabilities for the healthcare scenario. Given the heterogeneity of the platforms, this approach removes a source of complexity for the programmer. The programmer needs to specify the stream processing computation in a high-level, platform-independent language (a set of Event Processing Language statements, as will be described in Section 3.2.1), provide the configuration file defining available infrastructure components with its capabilities, as described in Section 3.2.2 on Resource Catalogue, and a configuration file defining the non-functional requirements, see Section 3.2.3. An optimiser module applies logical (Section 3.3.2) and physical optimisation (Section 3.3.3) and a cost model to determine the best

to partition the computation defined by those statements across the available set of platforms. The best partitioning options is determined from the total cost of individual plans. This thesis explores Energy cost model in Chapter 4 and a Bandwidth cost model in Chapter 5. The framework takes into account the functional capabilities of the platforms (e.g. not all computations that can be performed on a cloud can be performed on a wearable or a phone). *PATH2iot* can, therefore, be viewed as a way to automatically bring the advantages of edge computing to IoT stream processing.

1.2 Problem Definition and Motivation

1.2.1 Research Question

This project investigates whether a high-level, declarative description of computation on streaming data can be used to automatically generate a run time execution plan that meets non-functional requirements. In the thesis, we use energy and bandwidth as examples of non-functional requirements, but the approach has the ability to encompass other requirements including accuracy, performance and monetary cost.

1.2.2 Main Contributions

Following is the list of the main contributions of this research work:

- *PATH2iot* allows the application administrator or domain expert to implement and deploy distributed computational infrastructure with little programming knowledge.
- *PATH2iot* uses optimisation techniques to make computational placement decisions taking into consideration the available infrastructure, a description of the computation and a set of non-functional requirements.
- *PATH2iot* automatically generates and distributes the software components across the platforms (e.g. wearable, phone and cloud). This removes a significant source of complexity for the programmer as each platform typically presents different software interfaces and challenges.

We evaluate *PATH2iot* using two substantial use cases: the healthcare example described earlier, and a smart city application. Each requires a different non-functional parameter to be optimised. These show the range and the effectiveness of *PATH2iot*: improvements in battery life of up to 453% were achieved in the healthcare use case [114].

PATH2iot is comprehensively evaluated through two real-world use cases, each with a different non-functional attribute that must be optimised. The first is the healthcare example presented earlier. Here the non-functional requirement being optimised is the battery life of the wearable. The other focusses on the monitoring of smart cities -– in this case transport – using devices connected over a low bandwidth network. Our solution offloads a significant amount of computation to the edge, and so reduces the amount of data that needs to be transmitted to the cloud over a bandwidth-constrained LoRaWAN network, that is disucussed in Section 5.3.4.

Another contribution of this work is in the way in which the energy cost model was designed and implemented. A Bayesian Regression model with binary covariates [98] is introduced to improve the accuracy of the predictions (see Section 4.2.2). Section 4.3 informs reader of experiments that were carried out the compare the predicted power consumption and empiracally gathered power measurements estimating the battery life of the wearable device confirming its accuracy.

The *PATH2iot* system is made available as open-source software and has been built to be modular, so it can accommodate additional needs. That this is possible has been demonstrated by an extension that has been added by another research group to support user-preference optimisation [88].

1.3 Publications

During the course of this PhD, the following publications have been produced:

- **Michalák, P.**, Heaps, S., Trenell, M. and Watson, P.,
  Automating computational placement in IoT environments: a doctoral symposium, ACM International Conference on Distributed and Event-Based Systems, DEBS, 2016 [113].
  This short paper was presented at the DEBS conference in a Doctoral Symposium track. The paper outlines challenges in automating computational placement in IoT environments and presents a novel approach utilising a high-level declarative language.

- **Michalák, P.** and Watson, P.,
  PATH2iot: A Holistic, Distributed Stream Processing System, IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2017 [114].
  This main track paper introduces the *PATH2iot* system, and its capabilities which are demostrated using a real-world healthcare use case, monitoring of type II diabetes patients. The system can decompose high-level declarative language queries that define the necessary computation for a step count algorithm, processing triaxial data. Also, this paper

presents logical and physical optimisation steps, with an Energy cost model. As a result, a battery life of a smart watch device was improved by 453 % when compared to other possible operator deployment allocations.

- Roberts, L., **Michalák, P.**, Heaps, S., Trenell, M., Wilkinson, D., and Watson, P., *Automating the placement of time series models for IoT healthcare applications*, IEEE International Conference on e-Science, 2018 [134].
  This abstract paper was written in collaboration with a group of statisticians and outlined future work focused on advanced time series forecasting of heterogeneous streaming data in the healthcare domain, and the automated partitioning of such prediction models on the available infrastructure.

- Jha, D., **Michalák, P.**, Ranjan, R., Watson, P., *Multi-objective Deployment of Data Analysis Operations in Heterogeneous IoT Infrastructure*, IEEE Transactions on Industrial Informatics, 2019 [88].
  This journal paper was published in collaboration with another research group. An external decision-making module, based on Analytic Hierarchical Processes was developed by the other research group. This external module allows the application administrator to place a set of conflicting non-functional requirements on the system. These are automatically resolved, with the best execution plan being selected. Our contribution to this work was to adapt the *PATHfinder* optimisation module to work with external cost models; we have extended EPL grammar to work with variable time-based window length; and provided additional energy coefficients for a mobile phone.

1.4 Thesis Overview

The rest of the thesis is structured as follows:

The background to the research is provided in Chapter 2, with the definition of terminology, introduction to database systems and stream processing optimisation techniques that have been adopted in this work.

Chapter 3 describes the *PATH2iot* system, its main components, optimisation techniques, the automated operation placement process and deployment strategy. The energy model used to cost individual plans during the physical optimisation phase is detailed in Chapter 4. Two approaches to Energy Impact calculations are presented and compared with techniques to capture uncertainty.

Chapter 5 describes the smart city use case, which exploits audio analytics, and has network bandwidth as the main constraint.

Chapter 6 – Conclusions – summarises what has been achieved and compares it to the goals. It ends with suggestions for further work.

CHAPTER 2
BACKGROUND

Since the very first general-purpose computer – ENIAC – was commissioned by the US military in 1943, rapid technological progress in computing, storage and networking technology has opened up new possibilities for collaboration, information exchange and resource sharing. The first cloud solutions only offered consumers limited remote disk space. Now, developers have under their fingertips a seemingly infinite pool of storage, compute, and network resources available on-demand with pay-as-you-go pricing structure – a key feature defining the modern cloud. Recently, rapid technological progress has led to embedded devices and sensors, which has created new opportunities for Internet of Things use cases that exploit local processing.

2.1 Internet of Things and Clouds

As early as 1961, Professor John McCarthy envisioned customers paying only for the computational services they use – treating them as a public utility. Sixty years later, the global cloud computing industry is worth over $100 billion [148] with increasing usage and a growing number of services offered by the three major providers in the western world: Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). They allow tech start-up companies to rapidly build new services that are scalable, without the prohibitive upfront costs of setting up a sufficiently large private cloud to satisfy future growth and peaks in demand. The flexibility of clouds also allows companies and investors to purchase the computational, storage and network resources they use, and once the services are not needed anymore, they can be switched off.

The two vital enablers for cloud computing that were needed for this industry to become the success it is today were (i) pervasive networking and (ii) virtualisation. Both are still drawing vast interest from both academic and industry researchers, and there have been significant improvements over the last decade.

Recent developments in networking that are relevant to this thesis are discussed in detail in Chapter 5: Bandwidth. An important milestone was reached by the introduction of containers [112] and microservices [145], allowing a large number of highly specialised virtualised resources to be loosely coupled to offer better scalability, clearer architectural design and more straightforward orchestration. Following Moore's Law, which states that the number of transistors on a dense integrated circuit doubles nearly every two years, this dramatically increases the server's computational capabilities. This growth has also permeated to devices outside of the cloud, ranging from simple sensors, field gateways and other small single-board computers, such as the Raspberry Pi[1]. This gave rise to the Internet of Things, Fog and Edge processing.

Widely spread, geographically distributed, often mobile devices are becoming ubiquitous. They predominantly use wireless access to the Internet, and so the term "Internet of Things" has become used to describe them. This term was coined by Kevin Ashton in 1999. Estimates from McKinsey suggest that by 2023 there will be over 43 billion IoT-connected devices, almost a three-fold increase from 2018 [7].

As technology progressed and more heterogeneous devices with sensing capabilities were placed in the environment, there was the need for a way to achieve a low latency response to events, so enabling near real-time applications. These requirements gave rise to "Fog Computing". Fog computing was defined by Cisco as a "highly virtualized platform that provides compute, storage, and networking services between end devices" and "extends the Cloud Computing paradigm to the edge of the network" [47, 46]. With increasing processing capabilities the focus shifted to "Edge processing" which was defined as "any computing and network resources along the path between data sources and cloud data center" [137]. There is much overlap between the two terms, and some research groups use them interchangeably, such as [65]. Both techniques aim to shift the computation closer to the data source, utilising the computational power either directly on the sensor device, or very close to it.

As well as reducing the latency for responding to events, there are other advantages to this approach. It can reduce energy costs from battery powered devices by using local processing to reduce the amount of data sent over a network. This can also enable a reduction in the network bandwidth needed to transmit data for further processing in the cloud. Finally, it can also increase privacy by supporting local analytics, for example as in Databox, where data collected in people's homes are used to collaboratively train a deep neural network model without transferring raw data to the cloud [117, 161].

[1] https://www.raspberrypi.org

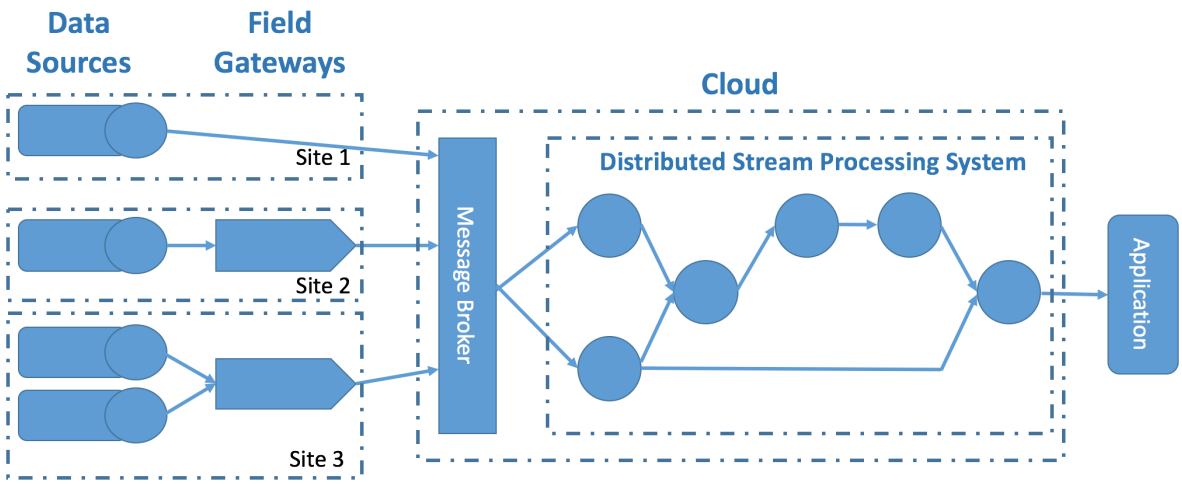


Fig. 2.1 Distributed Stream Processing System - typical event flow and processing diagram.

2.2 Data Analytics for IoT

Data analytics is needed to extract value from data. A traditional approach is to use databases; an application administrator designs a set of queries that are executed on a database server [78] into which all the data has been loaded.

However, as technology progressed, companies investigated options for how to efficiently extract value from the vast amounts of data being generated by IoT, and user interaction with mobile devices. A well-established approach was for the administrator to configure the system to regularly process batches of data at pre-set intervals, such as nightly or weekly to gain insights into customer behaviour or company performance. The MapReduce model [63] is often suitable for this. Non-critical tasks such as click tracking for personalised advertisement display can function sufficiently with one update in 24 hours. However, this approach is not enough for many time-sensitive applications. An example is fault finding in the telecommunication industry where it can be essential to detect and remove faults as quickly as possible.

Realising the benefit of processing data as quickly as possible also gives companies a competitive advantage in a dynamic market. Conventional batch processing of historical data cannot address the ever-growing, dynamic nature of stream flows in “Big Data” environments.

As a result, real-time distributed stream processing has emerged to address this challenge.

Figure 2.1 presents a typical data streaming architecture. Data can be generated at multiple geographically distributed sites from various data sources (e.g. sensors, user and system interaction and applications). A set of sensors produce measurements about the environment they are placed in, and transmit it to a field gateway downstream. A field gateway is an intermediary device connected to the Internet, that can be used as a bridge in between the sensor device and

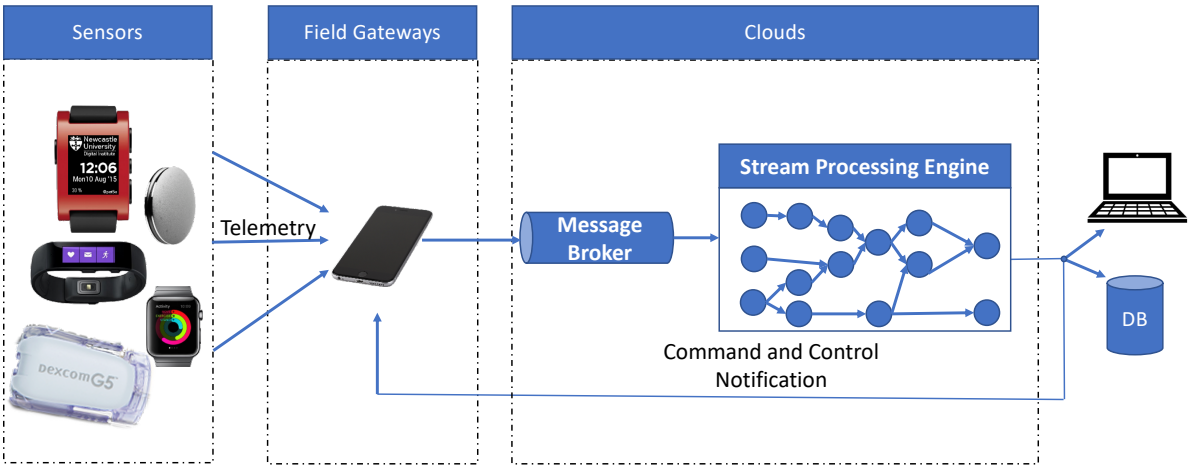


Fig. 2.2 Stream processing architecture overview for healthcare use case.

cloud; simple sensors do not necessarily have the capability to stream the generated data there directly – for example they may only be able to transmit over Bluetooth, Zigbee, or LPWAN. We explore wireless data transmission over Bluetooth and LPWAN in Chapter 4, and Chapter 5 respectively. As well as acting as a network bridge, a field gateway can also do some processing of the data, store it locally or send it through a message broker to the cloud for final analysis.

Typical IoT data analytics scenarios, such as [86, 74, 93], use a range of IoT sensors to generate data from the environment and send these data via field gateway to the cloud either for offline analysis or for analysis in real-time. However, as sensor hardware is becoming more affordable and more computationally powerful, a portion of data analysis can be placed directly on these devices in order to extend the battery life, lower the bandwidth required for data transmission, or improve performance by reducing latency. This thesis includes two use cases that demonstrate these advantages.

2.3 Data Transmission Evolution

Before the invention of wireless data transfer, data was moved either by a wired connection between electronic devices, or by using a variety of physical devices. Examples of the latter include IBM punch cards – "perhaps the earliest icon of the Information Age" [8], magnetic tapes, floppy disks and removable hard disk drives.

The ground-breaking theory of propagation of electromagnetic waves, formulated by James Clerk Maxwell in 1873 [44] paved the way for wireless data transfer, which is ubiquitous today. Shortly after the theory was published, Heinrich Hertz proved the existence of these waves experimentally. Gugliemo Marconi then improved upon Hertz's experiment by adding an antenna

to his apparatus, so dramatically increasing the range of transmitted signals. Further breakthroughs were made by a brilliant Serbo-Croatian inventor, who was an early radio pioneer and a futurologist, Nikola Tesla, in area of wireless data transmission and power distribution [135].

The data rate for wireless communications depends on several factors. There is a trade-off between using higher frequencies that have higher bandwidth but shorter range, or lower frequencies with lower bandwidth but better signal propagation.

With the explosion in mobile phone and mobile Internet usage, cellular networks transitioned through several generations of data transmitting technologies, from slow Circuit Switched Data (CSD) with speeds of 9.6 kbps and 14.4 kbps (depending on frequency), through General Packet Radio Service (GPRS – also referred to as 2.5G) from the year 2000 with theoretical speeds of up to 171 kbps. Subsequent upgrades such as EDGE (384 kbps), 3G (up to 42 Mbps) and 4G improved not only download speeds (of up to 150 Mbps) but also upload speeds, allowing more user content to be generated. Rolling out currently is 5G, with speeds of up to 10 Gbps utilising frequency bands between 24-28 GHz. This will also cut energy requirements by two thirds and enable full-duplex connections for the first time. This is likely to revolutionise how people connect, not just on the move, but also at home, potentially replacing fixed broadband connections. Additional forms of data exchange in mobile phones for wireless data exchange include NFC (near-field communication) technology for minimal range communication (for example between the device and an RFID tag), Bluetooth for short-range connection to smart devices, and Wi-Fi for connections to local area networks [54].

2.4 Event Processing

The first generation of Stream Processing Engine (SPE) designs, such as Aurora [36] were based on database architectures. They were built purely to process incoming streams of data without consideration for fault-tolerance or scalability. These shortcomings were quickly identified, and led to the second generation represented by Borealis [35], which was based on Aurora's system. The second-generation enriched the previous functionality with dynamic revision of query results – a system's ability to update previous records in case missing data becomes available – that allowed the data source to correct errors in previously produced results. The importance of dynamic query modification – changing query attributes at runtime – became clear as the systems had to be able to adapt to load changes and the volatility of data sources. Volatility can have many causes when processing data coming from sensors. For example, it can be due to: a temporary

network overload, a power cut (either on the sensor site or on the network infrastructure), the battery running out of charge, or a sensor software or a hardware failure.

STREAM [39] was designed to scale to high data rates, and introduced a declarative query language CQL [118]. This language was designed as an extension to SQL with a precisely-defined semantics, resource sharing in query plans, and an ability to approximate query answers if the system is forced to degrade gracefully.

Other tools then enhanced these capabilities, as in the case of IBM's SPC [38], which allowed administrators to use relational and non-relational operators. Support for user-defined functions (UDFs) proved very useful, and shaped the design of subsequent stream processing engines. Since then, declarative languages have been successfully used to express complex stream processing logic and are now used to process tens of terabytes of input events every day by spreading the load over thousands of servers.

Microsoft's 20,000-server production clusters use the StreamScope distributed stream computation engine [104] which achieves 10-millisecond processing latencies for simple applications. Furthermore, it provides a strong guarantee for exactly-once event processing even in the presence of server node failures; this highly desirable feature is difficult to guarantee in distributed systems, which usually only offer at least once processing. This is traditionally achieved by regular checkpointing and, in case of a server node failure, restarting the processing from the last persisted state.

Stream engines with Record-At-A-Time processing, such as Apache Samza [9], Apache Storm [6], or Microsoft MillWheel [37], achieve very low latency, scalability and fault-tolerance with additional engine specific features. Apache Samza introduced *ChangeLog Capture*, which enables every node to keep part of a remote database as a local snapshot. This speeds up any request to the database, as the up-to-date information necessary for a particular task is cached locally on a processing node. The MillWheel architecture is capable of handling delayed events as it automatically calculates a *low watermark* for individual input streams – this represents an amount of time before the engine produces a result. However, a careful balance has to be taken into consideration, as waiting for the delayed events will add additional processing time, hence must be taken into consideration in time-critical use cases. Apache Storm allows system administrators to create a bespoke stream processing solution by creating a chain of operators that are scheduled using a round-robin scheduler onto free nodes. This simplistic scheduling strategy has been an active research interest as it has direct effect on the system performance [73].

Apache Flink [49], MillWheel and PrIter [159] allow administrators to construct cyclic computations. These are computations that execute the same operations many times in a loop and can be used to train machine learning models efficiently.

The throughput vs. latency trade-off presents a common challenge for Distributed Stream Management Systems (DSMS). We have used this difference to group individual Stream Processing Engines into two categories: **Batch Processing** and **Record-At-A-Time Processing** engines.

Record-At-A-Time processing creates more overhead compared to a batch processing approach. This is primarily due to the method call and inter-node communication overheads – fewer resources are required if dozens of input data can be sent within a single message and a single method call. However, an artificial delay is introduced within batch systems, as a batch of events, either count or time based has to be assembled before processing happens. As this is the case, an architectural decision has to be made to ensure individual components are optimised for the chosen design. One way to increase throughput at the cost of latency, for example, is by micro-batching, as implemented in Trident [10]. Micro-batching is a technique for minimising the size of the window before processing, for example instead of waiting for 10 seconds' worth of events, a smaller window is chosen, and hence increasing the overhead, but lowering the latency.

DSMS system architectures are constantly being evolved to improve their features. UC Berkeley's Apache Spark [158], engine was designed to overcome the performance limitations of Hadoop MapReduce processing by implementing in-memory processing. This has been achieved by limiting the materialisation of intermediate results between operations, where the intermediate result is moved from the RAM into a disk storage. This worked well and the solution came to dominate the data analytics market space. An additional improvement was introduced with the extension presented as Spark Streaming, which splits the input data into micro-batches in an attempt to compete with Record-At-A-Time solutions on latency. A very well fine-tuned Spark Streaming engine allows stream processing to perform with latency as small as tens of milliseconds [50]. However, this latency is still five orders of magnitude larger than latency achieved by a purely Record-At-A-Time processing solution, such as Esper, which benchmarked its engine with average latency below 3 μs [11]. This is due to the overhead of micro-batching the unbounded stream, instead of processing it one event at a time.

An example of evolution from the other side of the architectural spectrum is Apache Storm, where the original architecture was built to process the stream on a per-record basis. However, to improve the throughput and to add a message delivery guarantee, a high-level abstraction

Name	Type	EH	MDG	Input	Deployment	License
Apache Flink [12]	SPE	R&B	=1	SQL-based	server	Apache 2.0
Apache Kafka [13]	SPE	R&B	=1	Java	library	Apache 2.0
Apache Samza [9]	SPE	R	>=1	Multi Lang	server	Apache 2.0
Apache Spark [158]	SPE	B	=1	Multi Lang	server	Apache 2.0
Apache Storm [6]	SPE	R&B	=1	Multi Lang	server	Apache 2.0
Aurora [36]	SPE	B	-	SQL-based	server	deprecated
Borealis [35]	SPE	B	=1	SQL-based	server	deprecated
Drools Fusion [14]	CEP	R	-	Rule Lang	module	Apache 2.0
Esper [15]	CEP	R	-	EPL	library	GPL v2
MillWheel [37]	SPE	R	=1	Multi Lang	server	proprietary
PrIter [159]	SPE	B	=1	Multi Lang	server	propriatery
Siddhi [144]	CEP	R	=1	Multi Lang	library	Apache 2.0

Table 2.1 Selected stream processing systems with key functionality overview. Event handling (EH): R - record at a time; B - batch processing; Message Delivery Guarantee (MDG): =1 exactly once; >=1 at least once.

was built on top of Storm – Trident [10]. Trident improves throughtput to millions of messages per second and also provides an exactly-once processing guarantee, an improvement from the at-least-once guarantee that Apache Storm offered.

Some tools aim to achieve the best trade-off between latency and throughput. One of these engines is Microsoft's Trill [53]. The Trill (Trillion messages per day) system is based on a tempo-relation model with an ability for the application administrator to specify the required latency response time. The core of this tool processes streams in batch mode using fast bit-vector operations. A unique feature – *punctuations* – allows the system to push the results out of operators prematurely to fulfil the latency criteria, only partially filling the data batch size. The punctuations are injected by the system based on the user-specified latency, which in effect dynamically adapts batch sizes. The authors observe that the adaptive batching improves systems throughput during periods of heavy load as larger batch sizes are used.

Table 2.1 presents a selection of stream processing systems that were discussed in this section with their key functionalities.

2.4.1 Complex Event Processing

Another important step in the development of stream processing systems was the ability to handle composite events. Complex Event Processing [108] (CEP) was introduced as a solution to a problem where the outcome of the query depended on multiple data streams being monitored.

Complex Event Processing tools, such as Esper [11], Drools Fusion [14], or Siddhi [144], provide powerful stream analytics libraries for processing composite events. The main differ-

ence between stream processing engines and CEP systems lies in whether or not they support heterogeneous streams of data. SPEs treat individual streams separately and apply any filtering, transformation, or aggregation only on one stream, independent of "neighbouring" data streams. In contrast, CEP systems are designed to allow an application administrator to define queries which detect patterns for a causal relationship between multiple streams originating from possibly geographically distributed data sources.

Esper uses an *Event Processing Language*, which allows application administrators to define advanced patterns to be detected over streams of events, using a variety of windows, aggregation, or negation. Drools Fusion implements a tool-specific rule language [16] which enables advanced temporal expressions with operators such as: after, before, coincides, includes and overlaps.

Data Windows

In the work described in the body of this thesis, we have adopted Esper's EPL to define a computation over streams of data. One of the main reasons for this decision was that Esper is implemented as a standalone library that runs on any platform that supports Java, and has sufficient computational and memory resources, such as a Raspberry Pi or, with some modifications to it, a mobile phone [17]. Another advantage it has over competitors is in its use of an SQL-like language, making it accessible to the many developers who have written database queries.

In order to process an unbounded data stream, Esper extends the language with data windows that specify over which sub-parts of the stream subsequent operations should be applied. Data windows can be set to be time-based or count-based with sliding or tumbling event batching. Time-based windows are triggered when a defined amount of time has passed. This can be driven by the system time, or by a timestamp contained in the events that are being processed. Count-based windows are used when it is the number of events that is the driving factor for subsequent operator execution.

To illustrate the most relevant types of windows used in our work Figure 2.3 visualises a "count" operator for events entering a stream processing system. Different types of windows are shown over the same time interval. A count operator simply returns a number of events within the window, regardless of their payload. The figure shows a flow of time from t_0 to t_{14} – for this example, each unit of time represents one second, denoted by the subscript. The "data_stream" row represents the events that are entering the system. The first data window is a count-based sliding window, 'data_stream:length(4)' – the argument defines the number of events that the window holds – that is triggered every time a new event arrives, and returns the current window count. The tumbling version of this type of window, 'data_stream:length_batch(4)', batches

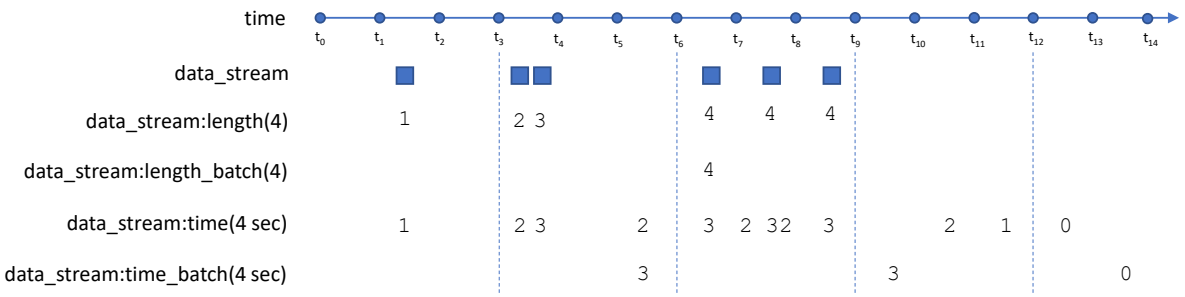


Fig. 2.3 Count-based and time-based data windows.

events and releases them when a given minimum number of events, in this example 4, has been collected. The time-based window, 'data_stream:time(4 sec)', is a sliding time window extending back into the past by the specified time interval (4 seconds in this example). It returns the current count for every time an event "expires" – for example at time point $t_{5.5}$ the window is triggered as the first event, collected at $t_{1.5}$, leaves it. Hence, this time-based window is triggered at t_5.5, t_10.5, t_11.5, and t_12.5. The last type of a window in this example is a time-based tumbling window, 'data_stream:time_batch(4 sec)', that batches events, and releases them every specified time interval, in this case, 4 seconds. As with the sliding version of this type of window, the tumbling time-based window will be triggered on event exit, hence a 0 count at $t_{13.5}$.

In the volatile world of the Internet of Things, the programmer must not expect all the events to arrive in the proper order. This may, for example, be due to network delays. Even sensor readings from the same device can arrive in an unexpected order. The administrator therefore has to bear this in mind when designing an application.

A window that is offered in EPL to assist in these cases is *out of order correcting*. This can be applied to a stream to automatically reorder the events before they will be processed. The challenge with this correcting action is to set a waiting period – a time for which the window waits for any delayed messages before re-ordering occurs. When an event arrives after the pre-set time interval, it is discarded. The choice of time interval is highly use-case specific and requires a domain expertise. For example, missing a single reading from a high frequency vibration sensor might not impact the computation if the reading is discarded due to a short waiting period.

An associated issue is the important question of what time to use when executing analysis – processing or event time? Processing time is the time that an operator can request from the system the operator runs on. This might work very well for many use cases where the decisions are made immediately after the event is digested. Event time is specified within the event itself. It is independent of the system time, or the time of the day that the processing is carried out. Event time is useful in data analytics that does not happen in real-time, or when the outcome of

data processing is very sensitive to time delays that may be introduced by the network or earlier processing.

2.4.2 Delivery Guarantees

Many tools allow the programmer to focus on the application logic without the need to handle possible node failure. The traditional approach to this in computing is to detect a failure or an underperforming machine and restart the partial computation with the appropriate data on another available node.

A novel approach to this introduced in Spark Streaming [158] uses Resilient Distributed Datasets (RDD), which allows for the parallel recovery of lost work. An RDD is a storage abstraction that allows the system to quickly recompute the lost data without replication. The system keeps track of computations associated with the RDD, and in the case of failure, it spreads the data and the failed task across the other, surviving nodes. This reduces the waiting time of the whole cluster as the failed task is recomputed in parallel. Spark Streaming captures relationship between interdependent operations as a lineage graph of dependencies with fine granularity that allows system to track and periodically checkpoint state of RDDs so it can relaunch recomputation of any tasks when it detects missing RDDs.

Message delivery guarantees vary across Distributed Stream Management Systems (DSMS). It is important to select the DSMS based on the application requirements and how possible lost data might effect the correct functionality. Delivery guarantees fall into three categories:

- **At-most-once** – tools such as Apache S4 [123], an actor-based framework, utilises checkpointing as a fault-tolerance strategy to achieve this objective. In case of a loss of individual events during node failure, the state of the system can be restored from the last checkpoint.
- **At-least-once** – Apache Samza and Apache Storm use 'at-least-once' processing guarantee, as they replicate the state of the node after a failure and replay the events that were being processed when the failure occurred, leaving duplicate-elimination up to the application logic, if it is necessary.
- **Exactly-once** – there is an additional cost involved in ensuring exactly-once message processing, but many DSM systems provide this feature as it is required by some applications. This guarantee naturally introduces a processing overhead, as there must be a way of uniquely recording which events have been processed already. One approach assigns a unique ID on every record upon entry into a processing framework, and the system keeps track of its progress, which allows for precise recovery in the case of processing failure.

For highly mission critical systems, such as in healthcare, exactly-once delivery guarantee might be needed, as any lost data might result in patient harm. On the other hand, up-to-date 'like' count on a social media post can be off by one or two without causing any critical harm, and it can be recomputed later.

2.4.3 Data Sources

We have explored a variety of devices for both the healthcare and smart cities use cases used in this thesis. This section provides a detailed overview of devices explored for our two use cases: healthcare and transport. Specifically, this section describes the data coming from activity, glucose, and audio streams.

A continuously growing range of activity trackers is available on the market. They vary by the type of data they can collect, by battery life and by additional capabilities, such as access to the raw sensor data. The summary below outlines a non-exhaustive list of the main ones that were investigated for this project. This gives an idea of the functional and non-functional capabilities of such devices.

Pebble Steel Watch

The Pebble smartwatch started as a crowdfunding idea campaign on Kickstarter on 11th April 2011. The promised programmable wearable activity tracker device with seven days of battery life and an always-on display was delivered two years later, with first shipments in January 2013. The company continued developing new products and raised a total of $43.8M in three separate crowdsourcing campaigns [89]. Unfortunately, the company found itself in manufacturing and financial problems, filed for insolvency, and was bought by Fitbit in 2016. Fitbit was subsequently acquired by Google in late 2019. The development of the original Pebble watch products has ceased, however, new programmable models appeared after the acquisition, notably Fitbit Versa and Fitbit Ionic.

The smartwatch had run on TinyOS [101], an event-based operating system designed for embedded networked sensors which allows a software developer to program in C++ and JavaScript. The API gives direct access to the onboard accelerometer sensor, where different rates of sampling frequencies can be selected from 10, 25, 50 or 100 Hz. The developer can also select the batch size specifying how frequently the callback method is executed to process the accelerometer readings.

The Pebble Steel Watch was a pioneer in offering the capability to write and deploy programs as built-in apps with unprecedented control. This created an opportunity to pre-process the

accelerometer data locally. However, what could be implemented was restricted by the per app memory limit of 24 KB, that was shared between the application and the data.

Fitbit Ionic

As previously mentioned, after Fitbit acquired Pebble and retained a portion of the developer base, the company introduced the first two programmable watches: Ionic in 2017 followed by Versa in 2018. A web-based programming environment [2] similar to the previously used "CloudPebble" web IDE with smartwatch simulators was offered, with Javascript as the main programming language.

The Fitbit Ionic offers an accelerometer, gyroscope, heart rate, light, GPS, magnetometer and altimeter sensors that are accessible through API. The device communicates with a mobile phone over Bluetooth 4.0 LE, but can also connect to WiFi, and use NFC for contactless payments.

Microsoft Band

Microsoft introduced the first version of their smartwatch, Microsoft Band, in 2014. This was followed by an upgraded version with an impressive range of sensors: optical heart rate monitor, three-axis accelerometer, gyroscope, GPS, microphone, ambient light sensor, galvanic skin response, UV sensor, skin temperature, capacity sensor and barometer. As far as we know, the variety of sensors within a single wearable device with a battery life of up to 48 hours has not been superseded to date. However, Microsoft did not document how to access all of the onboard sensors, and also severely restricted the programmability of the watch. A programmer could however build a dedicated 'Tile' for simple communication with the mobile phone. Even though Software Development Kits (SDKs) were available for all three major platforms (iOS, Android and Microsoft), due to the combination of high price and unusual design features, the watch was not a success, and Microsoft stopped distribution in 2016.

GENEActiv

A wearable device designed for use in clinical studies was developed by Activinsights[3], founded in 2008. The GENEActiv range offers products with a sampling frequency of up to 1000 Hz and maximum logging period of 45 days (for 10 Hz sampling). Currently, the latest model offers wireless data transfer. Even though none of the watches can be programmed to execute computation locally, many research groups used them in their trials, such as for estimating

[2] https://dev.fitbit.com
[3] www.activinsights.com

sleep parameters [147], for monitoring of Parkinson's disease patients [146] or for comparing the impact of intermittent walking when compared to standing, for both insulin and glucose response [130].

Accelerometer Data

The crucial component within any activity tracker, including all of the reviewed smartwatches, is the accelerometer sensor, most typically a triaxial accelerometer measuring acceleration force on the device in the three dimensions: x, y, and z. These are measured either in units of earth's gravitation force (g) or in metres per second squared (m/s^2) with standard gravitational constant $1g = 9.80665m/s^2$. If a highly precise conversion needs to be carried out, one must also consider the gravity anomaly and account for it [132].

Figure 2.4 presents four different raw accelerometer data scenarios measured by a GENEActiv device at 100 Hz sampling frequency. These were for a user working behind a computer, staying still/meditating, walking and running. A periodic pattern can be seen for walking and running, and this will be further explored in Chapter 3, where we implement a step count algorithm based on a fully-featured pedometer design [160].

Table 2.2 compares the main features of a selection of devices that have been considered for exploring computational offloading of stream processing in a healthcare application. The '!' marks partially programmable wearable devices and '!!' wearable devices without a screen.

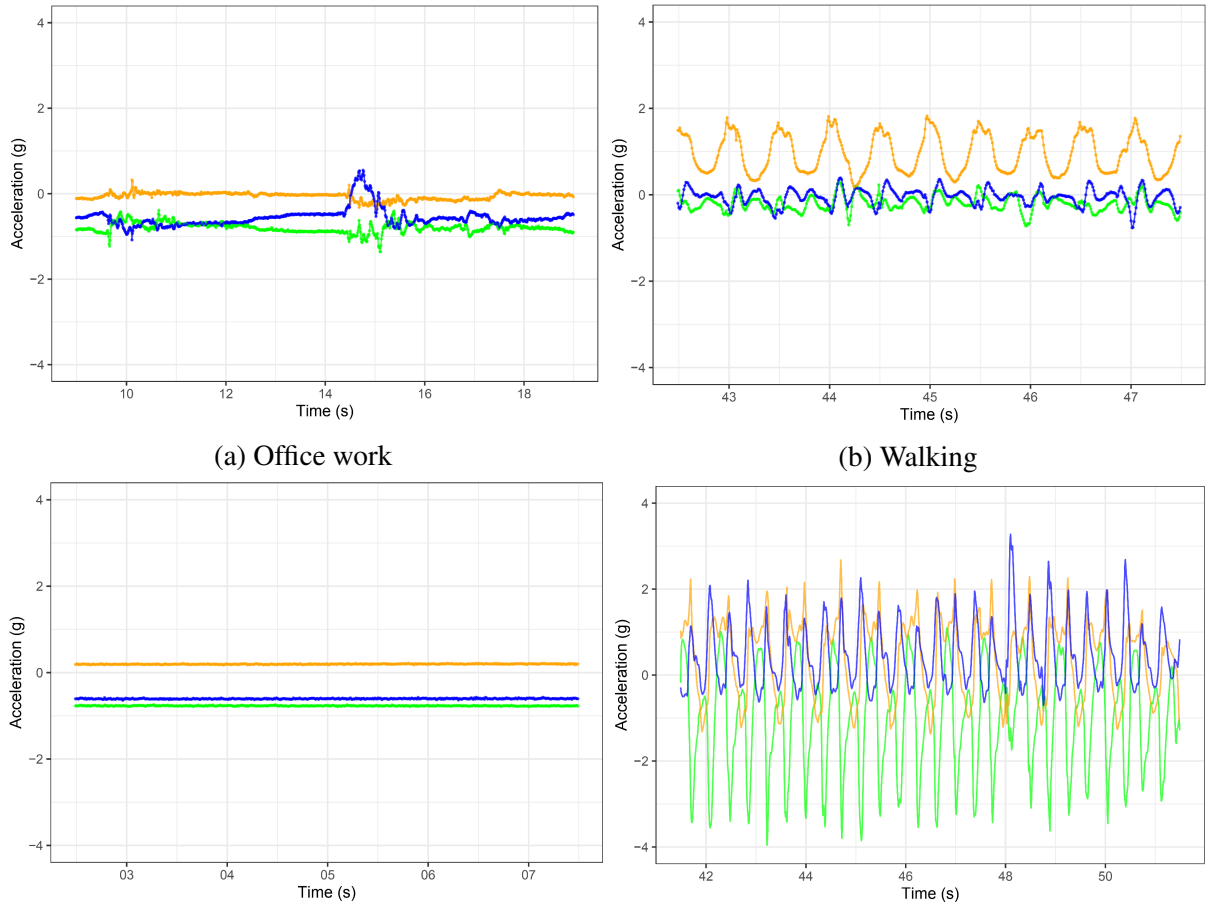
Continous Glucose Monitors

The current most prevalent method to measure blood glucose level is a finger prick test, where a lancet is used to break the skin, usually on the finger and a small drop of blood is applied on a blood glucose monitoring strip and inserted into a blood glucose monitor. After a few seconds, an accurate measurement is displayed on the device. The latest improvements allow the reading to be recorded and transmitted over a wireless network to a user's mobile phone or cloud. However, this approach requires the user to take action every time a new reading is needed, which can be painful and inconvenient. The information is used to determine whether any corrective action is needed, such as in cases of high blood glucose, when insulin is administered, or more immediately when there are dangerously low glucose levels that might induce a coma if glucose levels are not raised quickly enough.

Several companies have developed Continuous Glucose Monitors (CGMs) to alleviate this problem. These are small, minimally invasive sensor devices that are used mainly by diabetic patients to manage their condition better. These devices provide regular and accurate readings

Device	battery life (days)	sensor count	accelerometer	heart rate	gyroscope	GPS	microphone	light	galvanic skin resp	UV sensor	skin temperature	capacity sensor	barometer	magnetometer	colour screen	programmable	open-source	on the market
Pebble Steel	7	3	✓	✓					✓						✗	✓	✓	✗
Microsoft Band 2	2	11	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	!	✗	✗
GENEActiv Original	45	3	✓					✓			✓				!!	✗	✗	✓
Fitbit Ionic	4	7	✓												✓	✓	✗	✓
Apple iWatch 5	0.75	8	✓	✓(x2)	✓	✓	✓						✓	✓	✓	✓	✗	✓

Table 2.2 List of selected Activity wearable devices (as of Dec 2019)



(a) Office work

(b) Walking

(c) Meditating

(d) Running

Fig. 2.4 5-second raw triaxial accelerometer data.

of glucose from the interstitial fluid. They can be attached without medical supervision in the stomach, back, arms or legs. Originally, regular calibration still had to be carried out every 12 hours by the traditional blood finger prick test, but as the technology matures, even this step can be circumvented by a factory sensor calibration (such as in Dexcom G6 CGM model).

There are several types of CGM systems on the market, and a non-exhaustive list is presented in Table 2.3. These sensors either stream the readings regularly (e.g. every five minutes) over a Bluetooth connection to the dedicated reader or mobile phone, or store the readings locally, and transfer the data only upon proximity to the reader – so-called Flash Glucose Monitors (FGM). For example, the FreeStyle Libre uses a Near-Field Communication interface (NFC) for this. The disadvantage of FGM is that there is a finite capacity to store the readings, and if the collection does not occur frequently enough, some of the data might be lost. Another practical advantage of the CGM solution is that it can link with the insulin pump to create a closed-loop system, where the real-time readings affect the automatic insulin intake.

The sensor life of CGMs varies from 10 days with the Dexcom G6, to up to 90 days for the Eversense XL solution. Eversense uses a fully implantable sensor, placed under the skin

Device	Sensor life	Reader	Sensor	£/day	Type
Dexcom G5	7 days	£200	£51	£7.29	CGM
Dexcom G6	10 days	£200	£51	£5.10	CGM
Freestyle Libre	14 days	£58	£58	£4.14	FGM
Medtronic Guardian Connect	6 days	£299	£55	£9.17	CGM
Eversense XL	90 days	NA	NA	NA	CGM

Table 2.3 List of selected Glucose Monitors with key feature comparison. Eversense XL costs were not publicly available, as of December 2019.

on the arm to measure glucose levels. The price also plays a part in deciding which glucose monitoring approach to take. [79] outlines further advantages and disadvantages depending on the perspective and patient needs when choosing a CGM compared to an FGM solution.

We collaborated with a research group that focuses on development of an advanced statistical model used for personalised prediction of severity of hyperglycaemic episode for type II diabetes patients, under a variety of activity-based interventions [134].

2.5 Related Work on Moving Computation to the Edge

We present a new approach that simplifies the design and implementation of efficient systems for processing streaming data at the edge of the network. The edge computing is defined as any computation capability along the path between data source and cloud [137]. Extracting value from the data can be challenging, especially when it exhibits the high velocity and/or high volume commonly referred to as Big Data. In order to meet this challenge, stream processing systems have been created specifically for processing data in the cloud (e.g. Apache Spark[4] and Apache Storm[5]).

While these systems are highly efficient, this cloud-centric approach can present major problems for some important stream processing scenarios, as they require data collected by sensors to be sent to the cloud for analysis. In particular, sensor battery life can be a major problem due to the energy cost of sending messages to the cloud.

Comprehensive surveys have identified the need for further research to maximise the benefit of the recent unprecedented increase in internet-connected devices, whether it is sensors, actuators, smart appliances or wearable devices. These surveys have come both from academia [119, 139], and from industry [127, 60]. They all highlight ample opportunities that IoT technology can

[4]https://spark.apache.org/
[5]https://storm.apache.org/

bring to improve the services for its users. We have analysed the suggestions, identified the most critical requirements, and used them to shape the research reported in this thesis.

Energy reduction [67], lowering bandwidth [75], privacy preserving [161], decreasing cloud costs [121], or minimising latency [42] to satisfy ever-increasing user demands are some of the key requirements that are being actively researched in the field of IoT, Fog, and Edge processing. There is evidence that system administrators are seeking to reap the benefits of moving computation closer to the data source in almost every application type [149].

Energy-aware computation offloading of mobile code was explored in MAUI system [59]. Authors present three use cases: face recognition, arcane game, and a voice-based language translation. All of these use cases are limited to a mobile phone environment with considerably more computational resources than a smart watch device. On the other hand, the system offers continuous program and network profiling of the application and it handles failures, when the smartphone loses contact with the server while the server is executing a remote method. ThinkAir [97], used a similar approach to MAUI, where a method-level computation offloading from mobile phones to to cloud was explored. The main aim of this framework was to reduce the execution time and energy consumption focusing on memory-hungry applications. In addition to computation offloading the framework allows on-demand VM resource scaling to exploit parallel processing. Another system, Dandelion [76], offers code offloading for wearable computing, such as Google Glass. It can offload the code to nearby devices or cloud to gain improvement in execution speed and energy efficiency. It focuses on code offloading of applications such as text recognition, face detection, or gesture recognition.

A prime use case to demonstrate the need for the data analysis closer to the data source are video streams. Video cameras are common on high streets, police patrol cars, or in transportation with a data rate that would usually be too high to consider sending all the data to the cloud for analysis. One way to solve this problem, was proposed by the researchers in the form of: 'cloudlets' [136] – an architectural element that "brings the cloud closer [to the data source]". These are decentralised virtual machine-based elements that can run on mobile phones. However, not all resource-constrained devices can host a VM-based solution, for example, small wearables with programming restrictions, small memory and disk space.

Pervasive health monitoring is a very active area of research. Inexpensive and unobtrusive sensors collect biomedical information that can be processed in real-time to potentially provide life-saving interventions. Fall detection is one example, where the real-time nature of the events requires the action to be taken with as little response time as possible [48]. Authors present a new fall detection algorithm that was split into two parts, one to run at the edge, on a mobile

phone, and the second part to be evaluated in the cloud. Another example is dynamic data source selection, that chooses between the mobile phone and a smartwatch for the activity tracking of elderly people [122].

These previously described solutions were hand-crafted by experts and local processing programmed manually to create a bespoke solution. This is time-consuming and might not result in the most optimal placement of data processing operators, especially when conflicting requirements are placed upon the system [88], for example, low energy impact and minimum latency.

Partitioning the analytics pipeline over the available infrastructure is being actively researched, especially with the focus on IoT environments. LEONORE [150] provides provisioning of application components on resource-constrained heterogeneous edge devices. An auto-configuration solution for IoT platforms is offered in [126], where five algorithms are explored in order to achieve a high operator's utilisation targeting a large number of devices. Computational offloading from smartphone devices (specifically Android) has been investigated in [95]. Cuckoo [95] is a framework that provides a dynamic runtime system that decides whether a part of an application will be executed locally or remotely. This is achieved by intercepting all method calls during the runtime, and evaluating whether it is beneficial to offload the method invocation or not.

In [91], an algorithm is designed to reduce power usage and increase the battery life for a wearable device in a health-monitoring use case. The algorithm dynamically offloads computations and partitions the data processing between the wearable and the mobile phone with accuracy as the main criteria. This approach claims a 20 % system power reduction. However, it operates with a predefined set of classifiers and only two processing options: local and remote. An external signal specifies a certain level of accuracy, and the algorithm then loops through available classifiers and predicts how much power is required for each of them, selecting the one that fits the criteria.

Another energy-aware edge server placement algorithm is presented in [103], where an improvement of 10% of energy consumption was achieved on Dell PowerEdge R730 edge server that serves as a base station for a user accessing the Internet. Satisfying pre-specified energy constrains of individual operators placed on IoT devices were explored in [71], where a scheduling heuristics is developed to solve this NP-Complete problem. The prototype shows that the performance was improved by 40% adhering to the energy constraints using android smartphones, Intel Edisons computer boards, and Raspberry Pis.

CARDAP [87], a context aware real-time data analytics platform, explored energy efficient data stream mining for distributed mobile analytics applications. Authors stress the impact

of data reduction, especially before the wireless data transmission, delivers significant energy benefits. The implementation of the platform is focused on the Android mobile application for activity recognition using accelerometer data in combination with a light sensor to improve classification. The energy savings are evaluated using a software power monitor [6]. Our work focuses on even more resource-constrained IoT devices: a smart watch. Also, we use a hardware power monitor, capturing true power consumption for different deployment options. Furthermore, CARDAP does not provide any measure of uncertainty for the expected power savings as we do explore and compare two different approaches to tackle this challenge.

This raises a question of how much computation can be placed on these devices with a need for bespoke solutions for each use case scenario. As IoT streaming analytics place computation on the data source, estimation of energy requirements is needed for battery constrained devices. This is not a straightforward task. The impact of computation, networking and hardware modules needs to be taken into careful consideration. Another challenge is presented by limited bandwidth: what happens if the computation on the resource-constrained device produces too much or too frequent data that needs to be sent to the cloud?

2.6 Comparison with the Work Presented in this Thesis

In comparison to the other work described in this chapter, the new research presented in this thesis describes the design and evaluation of an end-to-end architecture for a streaming system aimed at IoT environments, with a focus on optimising for non-functional requirements. Whilst previous work has focussed on specific aspects of IoT stream processing, our work has required us to review and build on work across a wide range of areas in order to create a complete holistic system. In doing so we have identified gaps in previous work and found a way to address them.

We have based the system on the exploitation of a high-level, declarative stream processing language – EPL – to enable the automatic partitioning and deployment of a stream processing application across all parts of a system: the sensor, the edge and the cloud. In our use cases, we show how the system can optimise for Energy and Bandwidth - both of which have been highlighted extensively throughout literature review as important research challenges. Other non-functional requirements, such as monetary cost, or latency could be added to the system with additional development effort.

We have extended this language by allowing our system to perform a grid search to find the best window size, so improving the energy performance, but taking into account the increase in

[6] http://ziyang.eecs.umich.edu/projects/powertutor/

latency, similar to what was described in [65]. A new operator was added to the language that allows for this functionality, that is described in detail in Section 3.3.2.

The *PATH2iot* system not only decomposes the EPL into a directed acyclic graph of operators that are partitioned across the available platforms, but it is also capable of translating these partitions back, either to a new set of EPL statements or to a platform-specific set of configuration instructions for pre-installed IoT agents that can then perform the required stream data processing. As has been shown, automated deployment, including IoT platforms is an ongoing research field. However, our work presents *PATHdeployer* deployment module, that delivers required configuration to all infrastructure nodes available for processing, and we demonstrated its capabilities in Chapter 3 for automatic configuration deployment on a smartwatch, mobile phone, and cloud nodes.

CHAPTER 3
PATH2iot: SYSTEM ARCHITECTURE

3.1 System Overview

To answer the research question of whether a high-level declarative description of computation can be automatically partitioned, optimised according to a set of requirements, suitably transformed and deployed on IoT infrastructure, we have designed and built a system that can explore these possibilities. The *PATH2iot* system can automatically optimise operator placement and deploy the resulting distributed stream processing system spanning the cloud, the edge, and IoT devices. An operator is a smallest unit of computation that the system can operate with. All operators are created by decomposition of the high-level declarative description of computation that forms a directed acyclic graph representing the application's data flow.

A major challenge for a deployment system is to identify and partition computations across IoT environments, in order to satisfy a set of non-functional requirements. Our system accomplishes this by analysing a high-level declarative description of the computation and exploring possible partitionings by applying a set of cost models to arrive with a final execution plan that is then automatically deployed on the existing infrastructure. The *PATH2iot* system works only with one direction flow of data, that is reasonable in the typical IoT scenarios where the data is generated at the sensors and flow through the connected nodes to the cloud. The system is not suitable for mesh networks or ring networks as it uses Directed Acyclic Graph (DAG) to represent computation and does not permit cycles.

The overall system architecture is shown in Figure 3.1. It consists of three inputs: the Resource Catalogue that describes the capabilities and characteristics of available platforms (e.g. the wearable, phone and cloud), the Description of Computation, and the Non-Functional requirements such as energy and bandwidth. The optimiser (*PATHfinder*) takes these inputs and determines how to partition the computation over the active platforms in order to meet the

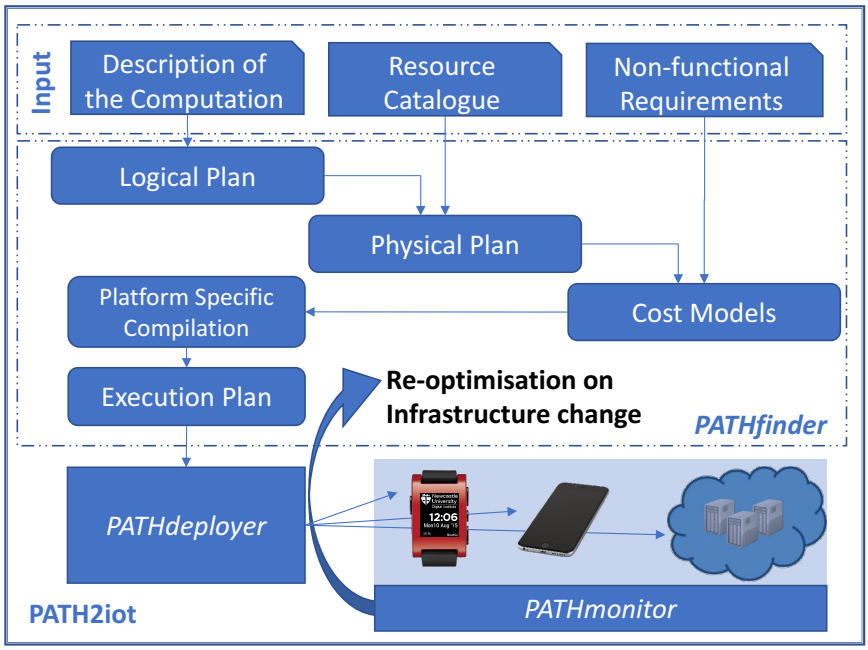


Fig. 3.1 The *PATH2iot* - System Architecture Overview.

non-functional requirements optimally; Energy (Chapter 4) and Bandwidth (Chapter 5) are the two examples implemented and evaluated in this thesis, though a performance model has also been built by another research group [88]. This output is passed to the deployer (*PATHdeployer*), which deploys it across IoT and Clouds. The architecture also supports a monitor (*PATHmonitor*) that could trigger re-optimisation and re-deployment, but its design and implementation is outside the scope of this thesis.

PATH2iot is a generic tool and can be used to cover a wide range of scenarios. However, as any tool there are limitations to its use. During this work we have identified three main limitations: UDFs, Multi-Site Deployment, and Licensing. These limitations are described in Section 6.2. To evaluate the system, including understanding the limits of the capabilities we have explored two different use cases that are driven by different non-functional requirements, one in healthcare and one in smart city monitoring. These use cases present different optimisation challenges for the tool, and the aim is to show that the tool is not specialised to fulfil a single requirement nor to satisfy a single use case within only a specific domain. There is an additional requirement for the energy use case as the system needs to have information of the energy use of individual operators on the devices that the optimisation should target, a list of estimates must be procurred and supplied to the tool. We detail the process in Chapter 4.

The first use case is in the area of digital healthcare monitoring to provide personalised behavioural prompts for type II diabetes patients to help them better understand and manage their condition; though summarised in the Introduction, we describe it here in more detail so that

it can be used as a running example in the description of the system architecture (the second use case is described in detail in Chapter 5).

3.1.1 Healthcare: Type II Diabetes Forecasting

We collaborated with medical researchers on a healthcare application that uses wearable sensors to monitor the activity and glucose levels of type II diabetes patients in near real-time in order to alert them to a possibility of upcoming hyperglycaemic episode so an action can be taken - in this case increase of physical activity. A Continuous Glucose Monitor (CGM) periodically collects accurate glucose measurements from patients. These must be analysed in order to give short-term forecasts: if it is predicted that a patient's glucose level will exceed the upper threshold for a healthy individual, a behavioural prompt (text message/notification) is issued to the user, asking them to increase their physical activity in order to attenuate the upward trajectory of their glucose levels. The intention of the behaviour prompt is not to enable the patient to avoid the hyperglycaemic episode as this might not be possible. Instead, it is to encourage them to stay closer to healthy glucose levels by taking corrective action through physical activity; this is in addition to taking insulin where necessary [134].

When the patient recovers from the hyperglycaemic episode, the impact of his/her activity levels are estimated and feedback sent to back the patient. The impact of increased activity on glucose metabolic response is an active research area [80, 56]. For example, evidence shows the benefits of high-intensity interval training to improve the glycaemic control in type II diabetes patients under unsupervised conditions [51].

The overall application requirements were therefore that it should analyse streams of activity and glucose data, utilise an appropriate time-series forecasting method and issue a behavioural prompt followed by feedback after a hyperglycaemic episode. We will now define all of these stages in detail.

As seen from Figure 3.2 healthcare sensor devices, such as smartwatches, typically need to rely on an intermediate component – field gateway – if they are to send data to the cloud. In this use case a field gateway, in the form of a mobile phone, receives the telemetry data from the sensors. It has the capability to do some processing of the received messages, and pass them to the cloud for further analysis. As seen from this figure, a message broker is used as an intermediate component in between the IoT devices and the Cloud. Message brokers are frequently used in streaming scenarios to decouple processing tasks. They offer robust, fault-tolerant and scalable solutions and reduce the need for mutual awareness of IoT devices

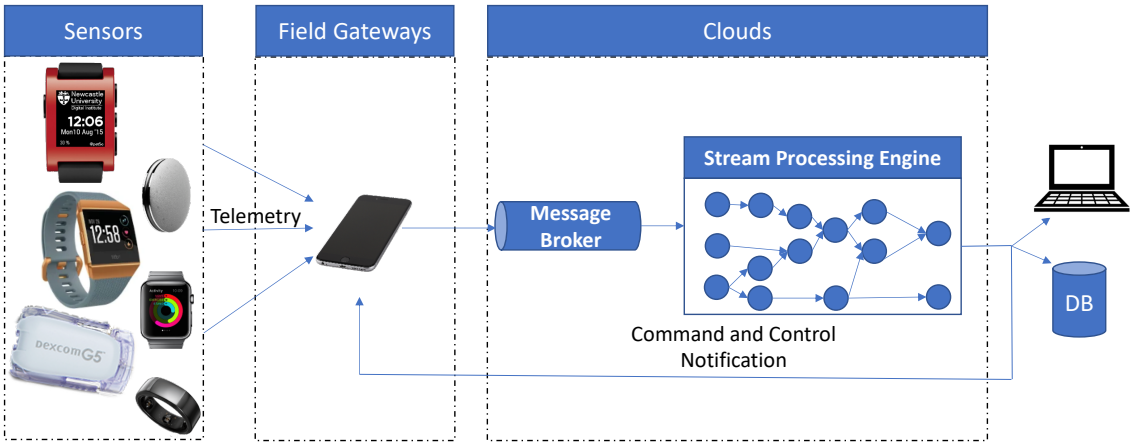


Fig. 3.2 Stream processing in IoT. Sensors (from top to bottom): Pebble Watch, Misfit Shine, Fitbit Ionic, Apple Watch, Dexcom G5, Oura ring.

and applications in the cloud. Once events have reached the cloud, they can be further processed (for example to issue prompts to the user) and stored in a database for future analysis.

3.1.2 Healthcare Data Collection

As a part of the research project and with cooperation with the research group focused on the forecasting of the healthcare time series, we have carried out a series of real-world data collection experiments using the following wearable devices to capture a range of healthcare related data:

- Dexcom G5 and G6 Continuous Glucose Monitors[1](CGMs) are approved by the US Federal Drug Administration (FDA) and provide real-time glucose readings in 5 minutes intervals directly to the dedicated receiver or a mobile phone. This information can be shared with other applications, family members or clinicians. For our data collection we have used the Sugarmate application[2] that allowed simple data export and also allowed for easy logging of carbohydrates and exercise.

- Oura ring[3] is a smart ring, that collects activity information with five minute granularity. Also, and uniquely, the watch activates its optical heart rate sensor when the wearer is asleep and starts to monitor the resting heart rate and heart rate variability. Heart rate variability is a measure of constant variation between heart beats that links to the autonomic nervous system and can be effected by stress, medical conditions, and other factors [92].

[1]https://www.dexcom.com/en-GB
[2]https://sugarmate.io
[3]https://ouraring.com

- Fitbit Ionic[4] is a programmable smartwatch with accelerometer, optical heart rate and GPS sensor with battery of four to five days. Fitbit allows for all collected data to be exported from the web site, including additional information, such as estimated VO2 max (an estimate for maximum rate of oxygen consumption), stride length, active minutes, food diaries, estimated oxygen levels (only during the night), and daily resting heart rate and cardio scores.

- GENEActiv Original[5] accelerometer watch sensing 100 Hz of triaxial accelerometer data, with ambient light and temperature sensor. The watch has a capacity to store up to seven days of data; hence data had to be downloaded and the watch recharged during the 10 day monitoring period.

This data was collected for the collaborative work with a group of statisticians, who explore time series forecasting models using Bayesian framework to generate personalised models that could be used for online forecasting [134]. This is an ongoing research project and the proactive personalised predictions for type II diabetes patients, as well as the behavioural prompts asking users to heighten their activity in order to attenuate the upward glucose trajectory followed by a personalised feedback are dependant on completion of the advanced time-series statistical model.

We have made available all of the data that the author of this thesis collected to the DataSHIELD research group[6]. DataSHIELD is an open-source initiative that provides an infrastructure platform for non-disclosive analysis of sensitive data, specifically suitable for biomedical, healthcare and social-science data analysis. It also allows protected data visualisation and the platform shows capability to handle broad applications even beyond biomedical sciences [155]. The DataShield team outlined potential uses in these areas [18]:

- use for development and testing of non-disclosive statistical algorithm;

- use for development and testing of non-parametric and non-disclosive algorithm;

- use for development and testing of synthetic data generation, in particular time series data;

- use for development and testing of machine learning techniques;

- provide non-disclosive access to the data to the public (upon request) to use for training researchers on how to do analysis on non-disclosive data;

- development, training and testing in the wider DataSHIELD community;

[4]https://www.fitbit.com/us/products/smartwatches/ionic
[5]https://www.activinsights.com/products/geneactiv/
[6]http://www.datashield.ac.uk

This will allow DataSHIELD researchers to use a real-world sensitive data for development and testing of new non-disclosive algorithms on following datasets:

- Continuous Glucose Data;
- Activity, sleep, heart rate, calory estimates, active minutes summaries, vo2max;
- Raw triaxial accelerometer data (100 Hz);
- Personalised sleep quality scores, resting heart rate, heart rate variability, temperature and estimates for daily expenditure (Metabolic Equivalents - METs) as calculated by the Oura ring;
- Detailed food diary and exercise.

Furthermore, the DataSHIELD store the data and will allow a non-disclosive access to the data sets to the public (upon request).

Appendix A provides a visualisation of the collected data from a single individual. Daily graphs of continuous glucose data from Dexcom G6 CGM, with measurements every 5 minutes, for a period of 9 days are plotted with all carbohydrate entries extracted from a sugarmate food diary. The second part of the Appendix contains visualisation of hearth rate and step count data from Fitbit Ionic wearable, with measurements every minute.

3.1.3 Accelerometer Data Processing

In this use case, the source of activity data is an accelerometer in the smartwatch.

An accelerometer is a sensor that measures a gravitational pull and can be designed with single or multi-axis detection. Three linear accelerometers are sufficient for the measurement of movement in three dimensions. Depending on the location of the sensor such as waist [45], wrist [106] or hip [85] the accelerometer data stream can be used to analyse a variety of information about the wearer, such as activity profile, activity recognition or detection of walking patterns respectively. As a result, accelerometer data is widely used in a range of healthcare applications, including estimating the activity of a monitored patient, sleep [147], calorie expenditure [52] and also for fall detection [64].

A basic step count algorithm has been explored for the diabetes management application, based on work in [160], in which a full-featured pedometer design is realised using a 3-axis digital accelerometer. The sampling frequency of an accelerometer sensor depends on the use case and can be adjusted to fit the purpose. For example, the Pebble Steel watch we use in our

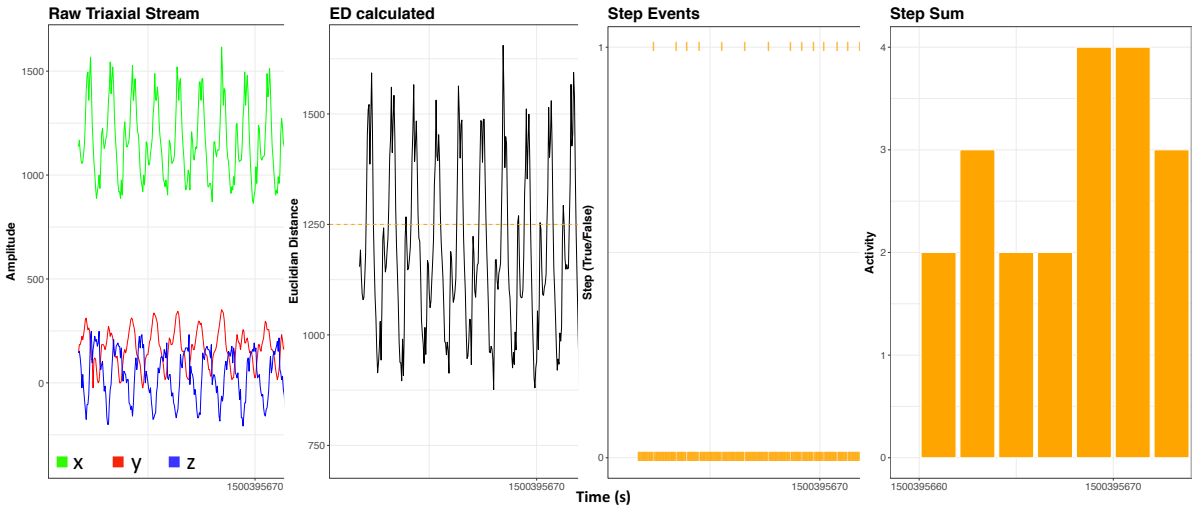


Fig. 3.3 Real-time monitoring of type II diabetes patient.

experiments is capable of being programmed to sense the accelerometer stream at 10/25/50 and 100 Hz. In our experiments, the accelerometers are sampled at 25 Hz.

The step count algorithm analyses the raw data stream from the accelerometer in the smartwatch. It computes the Euclidean Distance using the formula: $ED = \sqrt{x*x+y*y+z*z}$, where "x, y, and z" are the three axes measured by the accelerometer (a detailed explanation of the triaxial accelerometer data measurements can be found in Section 2.4.3). This stage transforms the three data sources for every collected sample into a new ED data stream, reducing the dimensions by a factor of three. Further processing occurs in order to calculate the total number of steps the patient walked in a period of time. Using a fixed threshold (THR), the algorithm determines whether a single step has been taken. Given consecutive samples, or events that are processed, ed_0 and ed_1, a step is detected when $ed_0 > THR$ and $ed_1 <= THR$.

Once the step is detected a time-based window is applied in order to have a step count summary, this serves as measure of activity levels for a patient and is used in an advanced time-series forecasting model as described in another research collaboration here [134]. The visual representation of the overall processing task is displayed in Fig. 3.3. This figure shows (from left to right) the raw tri-axial data as sensed by the watch, the calculated ED metric, the fixed threshold step detection and finally the aggregated step count for each time period – the 'step sum' part of the graph.

3.1.4 Glucose Data Stream

Glucose is a vitally important source of energy in humans. It is critical for bodily functions, and is particularly needed for normal brain operation. The standardised way of measuring glucose levels is either in molar concentration (mmol/L) or a mass concentration (mg/dL) (a linear

conversion between the two units can be applied for conversion: 1 mmol/L = 18 mg/dL). Blood glucose is traditionally measured by a finger-pricking method that is both painful, expensive and has an environmental impact as both the lancets that penetrate the skin to obtain the drop of blood needed for the test, and the blood glucose measuring strip are single-use only. However, Continuous Glucose Monitor (CGM) devices are now available. These consist of a sensor that sits just under the skin to measure sugar levels, and a transmitter that is attached to the sensor and periodically sends the levels to a device that displays them to the patient.

Type II diabetes patients must monitor their glucose levels throughout the day, especially after meals, so they can calculate how much insulin, an important hormone that allows for the glucose absorption by the cells, is required to bring their glucose levels as close to healthy levels as possible, and to meet personal long-term glucose targets agreed with their medical practitioner. Poor glucose control may result in cardiovascular disease, nerve, kidney and eye damage followed by a need for toe, foot or leg amputation [19].

We now give an overview of how *PATH2iot* enables a developer to create and deploy the application. Because the glucose analytics developed by our collaborators was not ready sufficiently early in the project, we focused our efforts on the activity analytics. Firstly, the developer must describe the computation in high-level declarative language, compile a list of resources in resource catalogue that are available for the computation to be placed upon, define non-functional requirements that the *PATH2iot* should optimise for with its configuration options. The three primary inputs are required by the tool to analyze the computation and make the decision on the placement of the operators in order to satisfy the defined non-fucional requirements. The system input will be described in the following section.

3.2 System Input

The *PATH2iot* system, as described in the previous sections requires three inputs from the application administrator: (i) set of EPL queries; (ii) current state of the infrastructure; (iii) definition of non-functional requirements.

3.2.1 High-level Declarative Language

The ability to automatically partition a computation over a set of platforms requires that the computation be described in a high-level, declarative way that is amenable to analysis, distribution and optimisation. To meet these requirements, we adopted a relational model in which a set of

linked Event Processing Language (EPL) queries define the computation. This has three main advantages over alternatives. Firstly, SQL based languages, such as EPL, have been used to describe stream processing in a number of systems: Apache Spark [40], Apache Flink [20], and Esper [21]; this confirms that they are expressive enough for a wide range of applications. Secondly, EPLs are based on SQL, which is familiar to a large portion of developers who will have used it to query databases. Thirdly, we can build on decades of work on optimising SQL queries in centralised systems [69, 162, 156], and in distributed query processing systems [96, 140], when designing the optimiser.

We therefore selected an Event Processing Language (EPL) developed by EsperTech[7]. It is an SQL-standard compliant, high-level declarative language designed for event stream processing with minimal latencies [11]. It has been extended to be able to operate on unbounded streaming data. A set of linked EPL queries define the overall processing of an application.

The step count algorithm, detailed in Section 3.1.3 can be expressed through a set of linked EPL queries. These process the raw accelerometer data stream generated in the Pebble Steel Watch. The algorithm is defined using the following five EPL statements:

1. **INSERT INTO** *AccelEvent*
   SELECT getAccelData(25, 60)
   FROM *AccelEventSource*

2. **INSERT INTO** *EdEvent*
   SELECT Math.pow($x*x+y*y+z*z$, 0.5) **AS** ed, ts
   FROM *AccelEvent* **WHERE** vibe = 0

3. **INSERT INTO** StepEvent
   SELECT ed1('ts') as ts **FROM** *EdEvent*
   MATCH_RECOGNIZE (**MEASURES** A **AS** ed1, B **AS** ed2 **PATTERN** (A B) **DEFINE** A **AS** (A.ed $>$ THR), B **AS** (B.ed $\leq$ THR))

4. **INSERT INTO** *StepCount*
   SELECT count(*) **AS** steps
   FROM *StepEvent*.win:time_batch(120 sec)

5. **SELECT** persistResult(steps, 'step_sum', 'time_series') **FROM** *StepCount*

Query 1 includes a user-defined function (UDF) to read the data from the accelerometer. It has two parameters: the sampling frequency and the mode of operation. The mode of operation

[7]http://www.espertech.com

is six bits that determine which of the full set of possible data fields should be generated on the device, e.g. mode 0 does not generate any data, while mode 63 generates all data fields: x, y, z accelerometer axes data, vibe (state of the vibration module on the watch: 0 or 1, for active and inactive respectively), timestamp and battery level (expressed as a percentage, in multiples of 10%). Mode 60, used in the running example, projects the first four event fields (x,y,z, and vibe). The naming conventions for the stream names is arbitrarily defined by the application developer. In this query, 'AccelEvent' in the INSERT INTO part of the EPL statement is a name for the new data stream and Esper's engine automatically creates a representation from these events, that can be used in the later statements. The initial 'AccelEventSource' is populated by the *PATH2iot* system from the 'STREAM_DEF' input file that is described within PATHfinder configuration in Section 3.2.4. In this case, it contains following event properties: 'x', 'y', 'z', 'vibe', and 'ts'.

Query 2 calculates the Euclidean distance from the raw accelerometer data stream for all of the data samples where the vibe parameter is set to 0. The 'Math.pow' function in a standard Java library function that is automatically imported by Esper as three other libraries: 'java.lang.*', 'java.text.*', and 'java.util.*'. The 'AS' keyword renames the output of the operation to defined name, in this case 'ed'. The 'ts' event property is carried from the previous statement that populated 'AccelEvent' data stream.

Query 3 processes the output of Query 2 and generates step events by detecting crossings of the specified threshold. To do this it uses the EPL's *match recognize* operator that Esper added to the standard SQL grammar to allow for pattern detection with familiar syntax of regular expressions – matching sequences of events instead of characters [15]. Note, that the *ed*1 and *ed*2 are arbitrary names defined within the *match recognize* operator that specify the order of the events. This allows the developer to define temporal relationship between the successive events. In this case the event *ed*1 represents earlier event and *ed*2 represents the event coming immediately after the *ed*1. The temporal relationship is then defined in the 'PATTERN' part of the 'MATCH_RECOGNIZE' statement as '(A B)', meaning that the event A must be considered as first, immediately followed by event B. This pattern can be more complex if needed, for example by using quantifiers within the 'PATTERN'. The quantifier can express number of events that need to be detected before the query produces an output. Furthermore, the 'PATTERN' can contain an alternation operator '(|)', which allows for including two conditions for the Esper engine to look for. The technical documentation contains several examples of this functionality addition to the SQL language [15].

Query 4 aggregates the input information using a 120s tumbling window, and sends to Query 5 events containing the number of steps taken in each 120s period. Section 2.4.1 describes the inner workings of time-based and count-based data windows on a toy example.

Query 5 then stores this in a database from where it can be accessed by healthcare applications – for example, to prompt the user with a text if action is needed to prevent a medical problem. This is achieved by a UDF function 'persistResult' that is not part of the Esper's built-in functions, and hence need to be registered at the beginning of the operation as any other UDF functions. The process of registering an arbitrary function to be used within the EPL statements is to use 'addImport' function when initialising Esper's engine. This is done automatically by *PATH2iot* system from the 'UDF_DEF' input file that is described within PATHfinder configuration in Section 3.2.4

3.2.2 Resource Catalogue

The Resource Catalogue contains information necessary for the *PATH2iot* system to optimise and deploy a distributed stream processing definition of computation based on the EPL description. This input file contains a description of the platforms over which the computation can be distributed including the computational capabilities of all the platforms, available resources and network connections. The optimiser accesses this information from the catalogue in the form that identifies all the platforms that can support a specific relational operator. In the running example, another key piece of information is the energy characteristics of all battery-powered platforms, that is also stored in resource catalogue files. Table 3.1 outlines the most important properties from the resource catalogue.

The information in the Resource Catalogue has to be provided by the system administrator before the *PATH2iot* system can run. Appendix B contains a JSON object containing the input information for the healthcare use case.

3.2.3 Non-Functional Requirements

The *PATH2iot* system currently supports two non-functional requirements that can be defined within an input file, which define constraints for the optimisation and directs the system to deliver the best possible plan that satisfies the requirements.

Non-functional requirements are loaded and parsed at the system start-up. The following attributes are needed for the requirements to be loaded correctly:

Parameter Key	Parameter	Description
nodes	state	active/disabled – the state of the node in the infrastructure
nodes	resourceId	the unique identifier of the infrastructure platform
nodes	resourceType	a type of the infrastructure, e.g. "PebbleWatch", "iPhone", "ESPer"
nodes	batteryCapacity	battery capacity of the infrastructure node, where applicable
nodes -> resources	CPU/RAM/disk	available processing, memory and disk
nodes -> connections	downstreamNode	a resource id of a downstream infrastructure platform
nodes -> capabilities	name	a name of the function or operator that the node supports with additional information about the specific type
messageBus	IP/port	connection details, type and credentials to connect to a message broker
energyResources -> EIcoefficients	type	a name of the function or an operator to which this impact coefficient applies; this must match to "nodes -> capabilities -> name" definition
energyResources -> EIcoefficients	cost	power impact cost
energyResources -> EIcoefficients	confInt	95% confidence interval
energyResources -> EIcoefficients	selectivityRatio	selectivity of the operator

Table 3.1 Resource Catalogue - description of key parameters. JSON format is used for Resource Catalogue records as seen from Appendix B.

- device – name of the infrastructure node that the non-functional requirement is placed upon;
- reqType – type of the non-functional requirement, in this case "energy";
- min – minimum acceptable value, if set to -1, the minimum value will not be applied;
- max – maximum allowed value;
- units – units that the requirement is defined in, in this case "hour".

All of the information in the Resource Catalogue is stored in a file in JSON format. This is loaded by *PATH2iot* at the beginning of the process of transforming the EPL into a running application. In this work we primarily focus on Energy and Bandwidth requirements. However, other types of non-functional requirements can include monetary costs, security or performance. This requires adding a new cost module to the tool. The implemented energy cost model module can be used for programers guidence for adding additional non-functional requirement. In addition, a provision for additional information for both the computational nodes and the

definition of operators have been made in the Resource Catalogue, making it easier to expand for additional attributes, for example security level of operator, network connection, or a device. An example of the requirement file for the health care use case can be found in Appendix C.

3.2.4 *PATHfinder* configuration options

All of these are loaded as JSON files by the system at the start of the execution. The application administrator needs to provide a path to the input files in the 'settings.conf' configuration file.

The primary configuration file for the system 'pathFinder.conf' allows an application administrator to set the parameters for the system run. The 'EPL' section requires file paths to the set of master queries, input streams, UDFs and infrastructure definition files, resource power coefficients, definition of high-level non-functional constraints to be satisfied and an optional file path for the output file, where a copy of execution file can be saved for later viewing.

Also the following flags can be set for the optimiser to select from the capabilities that it offers. We now go in detail through input file:

- **MASTER_QUERY_PATH** – a path to a file with a set of EPL queries defining the computation. Each query must be on a separate line. There is a possibility to comment out queries that the application developer wants *PATH2iot* to ignore, this is aachieved prepending the query line with '#' character. An example of the complete list of EPL queries for the healthcare example is in Appendix C.

- **STREAM_DEF** – a path to a file with a definition of existing data streams e.g. accelerometer, acoustic signal. The file must be in JSON format, with a 'inputStreams' key that has a list of streams defined within it. In the healthcare use case only a single stream is being processed: 'AccelEventSource' stream. This name must match the name of the first stream being processed in the first EPL statement. A list of 'streamProperties' with 'name', 'asName', and 'type' must be populated, additional optional property 'selectivity' that represents estimated ratio between the data in and data out that passes through the operator. In the healthcare example these are the 'x', 'y', 'z', 'vibe', and 'ts' property names, that are either 'double', 'integer', or 'long' type. The 'asName' property is a convienence that helps the rename the property to more human-readable name. An example of the stream definition file is in Appedix C.

- **UDF_DEF** – a path to a file with a definition of User Defined Function. This file must be in a JSON format with a key 'udf' and its value containing a list of application developer's User Defined Functions. Each function must have 'name' that correponds to a name

referred to in the EPL statements, 'output' this is the expected name of the stream that the output of the function should be pushed to, 'frequency' is the expected number of times the function will be called, 'generationRation' is an expected amount of data being produced by the function on every call, 'selectivityRatio' represents estimated ratio between the data in and data out that passes through the function, 'isSource' is a boolean flag that tells the optimiser whether this is a first operator in the DAG, and 'notes' that is a space for a free-text description of the functionality. Each function must include a 'support' key that lists all the devices that can perform this function. Each device has to have a name and a software version that supports is required from the infrastructure node to be able to run the operation and definition of 'metrics' - these are the list of estimated cpu, ram, disk, data out, monetary costs, and security level. These fields can be extended to support additional non-functional requirements, and are present in the configuration as an example of supported functionality. However, these are not relevant of the healthcare use case as this depends on the Energy Impact coefficients to calculate the cost of individual palns. An example of the UDF file is in Appendix C.

- **INFRA_DEF** – a path to a main Resource Catalogue infrastructure definition file, as described in Section 3.2.2;

- **RESOURCE_EI** – a path to a file with energy coefficient, if an energy cost model will be used. The file is in JSON format with main key 'energyRecources' with a list of resources that the energy impact coefficients are available. Each entry contains a 'resourceType' as a name of the resource, 'swVersion' that signifies for which version of the given resource type are the energy coefficients valid for, and a 'EIcoefficients' that contain a definition of energy impact coefficients per each supported operation with its 'type', 'operator', 'cost', 'confInt' that are all derived from the measurements that are detailed in Chapter 4 and a 'generationRatio' and 'selectivityRatio' defined as in previous input files. An example of a file is in Appendix C.

- **REQUIREMENT_DEF** – a path to a file defining the non-functional requirements. This file is in JSON format and contains a list of 'requirements' for the system to optimise. Each requirement has following properties 'device' is a name of the device for which to optimise, 'reqType' is a requirement type, such as 'energy' or 'bandwidth', 'min' and 'max' values that signify the permitted boundaries for which the plans must conform to, and 'units' that clarify the measure in which the boundaries are defined in. An example of a requirement file is in C.

- **EXEC_OUT_FILE** – (optional) – a path to a file, that a copy of the execution file will be stored for later viewing;
- **NEO4J** connection details – a section with IP, port, and credentials information needed to connect to the Neo4j database. Neo4j [152] is an open source JVM-based NOSQL graph database. The *PATH2iot* system offloads the graph operations, such as: traversing the acyclic graph of operators, linking individual EPL queries to a single graph, or ensuring all of the operators are connected within the graph structure, to the database during the runtime.
- **PATHdeployer** connection details – a section with IP and a port to connect to the *PATHdeployer* module;
- external module connection details – a section with an 'EX_MODULE_ENABLED' flag, that instructs the *PATH2iot* system to establish a socket connection to an external cost module, send all enumerated physical plans to it, and wait for the list of ids with associated costs as a result before proceeding to deploy a best plan; this section also includes IP and port information for the external module.

If any of the mandatory files or configuration settings are not found, or are not formatted as expected, the system will notify the administrator, and stop the execution. There are also additional settings available to fine-tune the optimisation process. For example, boolean flags for 'PUSH_WINDOWS' and 'PUSH_SELECT' can be set to true or false values to instruct the *PATHfinder* module to enable movement of these operators during logical optimisation. The next section will outline one of the key contributions of this work - the optimisation process.

3.3 *PATHfinder*: Optimisation Module

The optimisation module is at the heart of the *PATH2iot* system. It is the decision-making component that processes all the inputs, explores the space of possible deployment plans, applies optimisation techniques and cost models and selects the best plan that can be deployed on active components within the infrastructure taking into account the non-functional requirements.

This module is activated immediately after the system loads the users' input. All input files and configuration parameters as outlined in the Section 3.2.4 are validated. At first the configuration file for the tool is loaded and parsed through Apache Commons [8] configuration parser. The configuration's integrity is validated line by line and if any unexpected or missing

[8]https://commons.apache.org/

parameter is detected, the program is terminated and the user is notified. The resource catalogue is loaded using Gson library [9], the system creates an internal representation of the computing nodes within the Neo4j database with all of input parameters. The tool links the computing nodes together based on the "connections" entries within the file. The requirement definition is also loaded with information on which cost models need to be applied during the optimisation process. The master query text file is loaded line by line, as each of the lines represents a single EPL query. Each query is validates using the Esper's SODA API [21] that raises an error if the query contains a type, an unknown data stream, or unknown operation. We have added a variable window length operator which parameters are validated directly by the *PATH2iot* tool. The advantage of use of this operator are discussed in Section 3.3.2. The next steps of the optimisation process are performed as outlined in the following sections.

The *PATHfinder* – a self-contained module within *PATH2iot* – takes a set of EPL queries as input, and determines how best to partition the computation over the set of platforms, taking into account the non-functional requirements and the capabilities of the platforms. The best partition is selected as the one that has the least cost. Chapter 4 introduces an Energy cost model and Chapter 5. The *PATHfinder* contains following main stages: EPL query decomposition, followed by logical optimisation, and physical optimisation followed by application of cost models and device-specific compilation. These stages are now discussed in turn.

3.3.1 EPL decomposition

The high-level definition of the computation in EPL format needs to be decomposed into a form the optimiser can use. This is done in the EPL decomposition phase.

PATHfinder utilises the native Esper SODA API [21] for EPL decomposition. The queries are loaded, decomposed and linked together – `INSERT INTO` clauses contain stream names that are used for this inter-query linking e.g. `INSERT INTO AccelEvent` from Query 1 links with `FROM AccelEvent` clause in Query 2 in this phase.

The set of EPL queries is then decomposed into a computational graph of operators. Figure 3.4 shows this graph for the EPL queries in the running example. The 'Q1, Q2, ...' annotation is used to annotate from which query the decomposed operators came from. The description of operators is as follows:

- SELECT (σ): places a constraint on events (such as 'vibe=0' in Query 2), filtering out events from the stream so that they are not propagated to downstream operators;

[9] https://github.com/google/gson

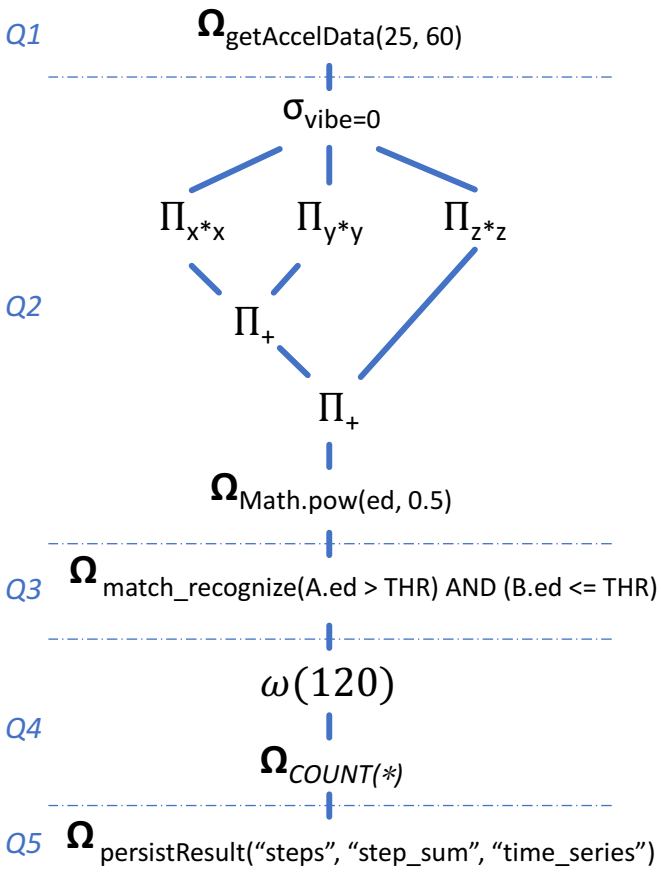


Fig. 3.4 The EPL statements from the running example decomposed to a Computational Graph.

- PROJECT (Π): removes fields that are not needed from the events; and/or creates new fields by transforming existing ones (e.g. $x*x+y*y+z*z \rightarrow ed$ in the running example);
- Windows (ω): a variety of different window constructs can be expressed by the event processing language [15], including tumbling, sliding and out-of-order correcting windows;
- User Defined Functions (Ω): a common use is for generating new events (e.g. at a source node) and persisting the events in a database (at a sink node). They are also used for all other use case specific computations that cannot be expressed using EPL statements. This allows arbitrary analysis computations, but should only be used where necessary as they limit the optimisation capabilities of *PATHfinder*, and narrow the placement options to those specific devices that support implementations of the given function.

Once all of the operators are decomposed and linked, optimisation techniques can be applied.

3.3.2 Logical Optimisation

The graph generated in the decomposition stage is then optimised through a set of optimisation transformations. These are drawn from previous work on distributed query processing [55, 39,

157], stream optimisation techniques [82] and some new window-based transformations that we have introduced based on our exploration of a set of use cases. These optimisations are:

- Select operators are moved as close as possible to the beginning of the stream graph so as to filter out events that are not needed in the computation as early as possible. For example, if an event has the vibe field not set to one then the haptic feedback was active, causing vibrations to notify the user, for example of an incoming phone call, text message or other notification. However, the accelerometer data collected during this event is not suitable for analysis as the measured acceleration is not only caused by the wearer, but by these vibrations, and should not be used in the step count. Hence, if a select operator that is filtering out these events is defined in later queries, the optimiser can move this operator closer to the data source. Placing the select operator at the beginning of the stream graph saves unnecessary processing, reduces the number of messages that are transmitted, and reduces the memory occupied by the window operator.

- Project operators are moved as close as possible to the beginning of the stream graph to remove event fields that are not used in the rest of the computation. This is done by scanning unused data fields that are produced by an operator and dropping them at the end of the logical optimisation phase, if this is selected within the settings. For example, some healthcare wearables may return fields that are not needed to compute the step count, e.g. light and galvanic skin response. Placing a Project operator to remove these fields as early as possible in the stream graph reduces the size of the messages that are transmitted, and reduces the memory occupied by the window operator.

- Window operators already present in the stream graph can be moved closer to the data source in order to reduce the number of messages transmitted between platforms. As will be seen in the evaluation, even though this does not reduce the total volume of data transmitted, as there is a large energy cost in transmitting a message, whatever the size, this optimization can have a dramatic effect on the battery life of resource-constrained platforms. The optimiser generates multiple options: one for each possible position of the window operator. Each of these options is then passed to the physical optimiser for costing as will be described in the next subsection. The Select, Project and UDF operators that are placed between the original position of the window operator and the new position are adjusted within the implementation of the system to work on each event in the window, rather than on individual events.

It is possible to extend the capability of the tool by adding additional operator movement. However, it is crucial to follow the safety rules in order not to modify the underlaying computation. The safety rules for moving operations were implemented into the tool. The 'window' operator cannot be moved upstream only if there is only one operator producing the results. Furthermore, the resource catalogue contains a parameter per each operator under 'capabilities', 'supportWin' that indicates whether a window can be moved before this operator as it can process both the single event at the time, but also can unpack the window operator and apply the transformation on individual events. The project operator can be moved as close as possible to the source of the computation removing unnecessary event fields and reducing the required bandwidth, only if any downstream operator does not require a field that would be discarded if this optimisation process would happen. The same applies for the Select operators. In addition ability for the user to disable this optimisation step was exposed through the configuration file where it can be disabled. Individual operators can be reordered if they satisfy the safety requirements, however, the tool does not reorder full queries. The catalogue of stream optimisation techniques [82] provides detailed discussion on safety rules for any other optimsiation technique.

Variable Window Length

Through exploring use cases we have identified the opportunity for a new optimisation – varying window sizes. Standard EPL queries offer a variety of windows as introduced in Section 2.4.1. Traditionally, the window size is fixed by the application administrator – one example was shown in Query 4 earlier in this chapter, which included a time-based tumbling window of length 120s.

Another way to use data windows is to batch up events before they are transmitted from one node to another, so long as the following operator permits data ingress by windowed events. This can dramatically reduce the energy expended. This may cause a problem in systems in which the processing of events is time sensitive. For example, it may be unacceptable to introduce a delay of minutes before informing someone that they need to take an immediate action to avoid their blood glucose level exceeding a healthy threshold. However, a delay of a few seconds may be acceptable if it dramatically increases the battery life of the sensor. This decision must be taken by the application developer as it is domain-specific.

A problem with leaving this decision up to the application administrator to find the best window size is that she would have to try all possible sizes one-by-one because the window size must be fixed in EPL statements. A best window size for given use case is determined as a trade-off between additional stream processing latency and an improvement in energy consumption caused by batching the events together or saving the bandwidth as will be described

in Chapter 4 and Chapter 5 respectively. As a solution, to further broaden the search space of deployment plans, we have extended the EPL grammar with the capability for the application administrator to define a variable window length.

The application administrator can then define the range of acceptable window sizes by providing the lowest and highest, along with a step size. This enables the optimiser to explore all windows sizes within the range, using the step size to set the granularity of the exploration within that range. As this is not directly supported by Esper, this is achieved by extending the EPL grammar as follows:

$flexi\_time\_batch(< start >, < stop >, < step >, < units >)$

where the first argument represents the minimum size of the window, the second argument is the maximum length, the third argument is the step and the last is the time unit. A pre-processor then expands this out into a set of options, each defined only by a valid EPL windows statement. These can then be explored by the optimiser.

We have demonstrated the usefulness of this approach in [88], where our system was enhanced by an additional multi-objective optimisation module that can make optimal deployment decisions based on conflicting non-functional requirements defined as user-preferences. An Analytical Hierarchical Process was used by this research group to achieve this objective. One of the queries presented in this paper uses the enhanced EPL grammar: $StepEvent.win : flexi\_time\_batch(30, 120, 15, sec)$. The *PATH2iot* system generates EPL queries for 30, 45, 60, 75, 90, 105, and 120 second duration with distinct plans for the optimiser in the pre-processing phase that are then passed through the logical and physical optimisation and the external ABMO costing module[88]. The usefulness of this in practise means that seven distinct sets of EPL queries were compared automatically by the system, instead of application administrator changing the setting one by one and re-running the optimisation process.

The usefulness of the added 'flexi_time_batch' operator in this example helps to evaluate all of the .

The variable window length approach can be easily extended to other supported windows.

3.3.3 Physical Optimisation

All of the plans generated in the logical optimisation phase are passed to the physical optimisation phase. The *PATHfinder* module applies additional optimisation techniques on each plan related to the placement of operators on the physical infrastructure.

The set of operator graphs generated by the logical optimiser serves as input to the physical optimisation phase. From these, a set of possible deployment plans is generated, and one or more cost models are then used to find the plan that best satisfies the non-functional requirements such as energy, or network bandwidth. Physical Optimisation proceeds in the following stages for each of the set of options passed by the logical optimiser. Firstly, the optimiser generates all possible partitionings of the stream graph. The set of platforms on which the system can be deployed is contained in the Resource Catalogue. The system generates one option for each possible partitioning (e.g. all possible allocations of operators to platforms). If there are p platforms and s operators then we might expect that the number of possible partitionings is simply p^s. However, this is reduced if we assume that in all practical cases events will travel in one direction (e.g. from wearable to phone to cloud). For a linear chain of operators, we can use the binomial co-efficient from the binomial theorem to calculate that the total number of partitionings [22] is

$$\frac{(p+s-1)!}{s!(p-1)!} \tag{3.1}$$

This formula confirmed the correct implementation of the physical optimisation phase as the number of plans explored by the optimiser was the same as the formula predicted. Even for small stream graphs, this can be a large number and shows why it is difficult for humans to pick the optimal placement – with only 10 operators and 3 platforms, there are 220 options to be considered. Device capabilities must now be considered as not all of the partitionings will be possible in practice:

- resource constrained devices do not have large amounts of RAM to store data. For example, the Versatile Pebble smart watch used in the running healthcare example can only hold up to 6000 Bytes of data. If this is exceeded, the plan is rejected and not pushed through to the costing phase.
- resource constrained devices do not have the processing power (and/or in some cases the library functions) to support all operators. For example, a UDF performing a complex data analytics function may only run in the cloud, and not on a wearable or phone.
- some operators can only be placed on one platform. e.g. in the healthcare example the operator that reads the accelerometer data can only run on the wearable.
- the optimiser was designed not to create any physical plans that would reverse the data flow in any of the processed streams. e.g. we assume an acyclic connection graph for events travelling between nodes.

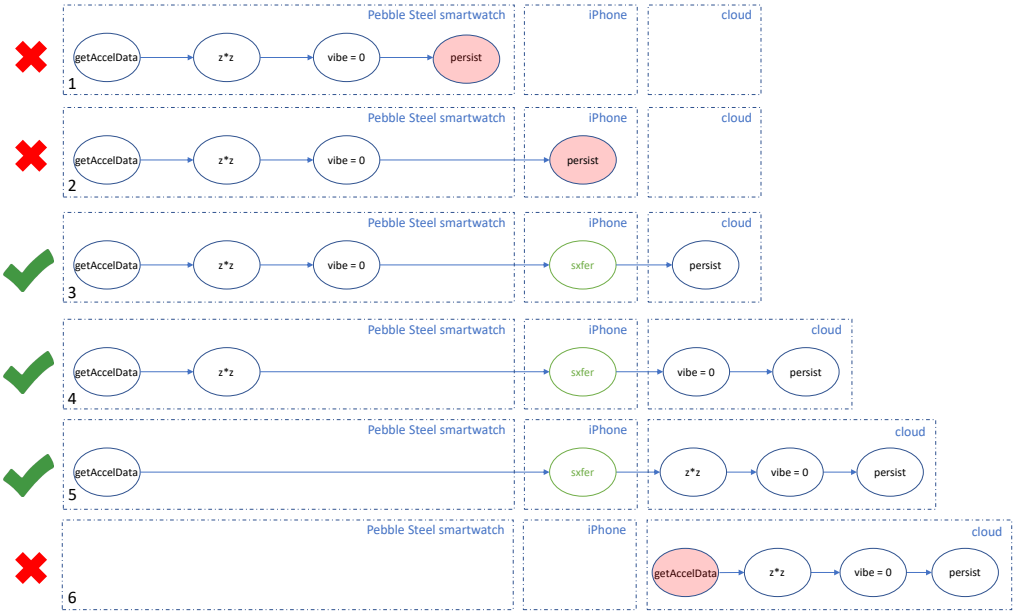


Fig. 3.5 Enumerating physical plans

- there may be cases where two platforms cannot directly communicate (e.g. the data from a Pebble Watch cannot be sent directly to the cloud as it transmits data over BLE [81]). To address this scenario we adopted an approach similar to the data exchange service in [151]: a special *sxfer* operator is injected into the logical plan to enable the data propagation, e.g. on the mobile phone, to relay data between the wearable and the cloud.

To demonstrate the capabilities of the physical optimisation phase, we return to the previously introduced example of four operators. We assume there are three platforms available in the infrastructure for operator placement: Pebble Steel smartwatch, iPhone, and cloud environment with Esper complex event processing support. Using Formula 3.1, given four operators and three platforms, we arrive with: $\frac{(3+4-1)!}{4!(3-1)!} = 15$ possible deployment options. As we have two logical plans, the original decomposed DAG of operators, and a second logical plan that was created during the logical optimisation phase by pushing the select operator closer to the data source, the optimiser arrives with 30 enumerated physical plans.

Figure 3.5 shows six examples selected from the pool of available physical plans. The first enumerated plan places all of the operators on the Pebble Steel smartwatch, leaving the iPhone and cloud resources underutilised. This plan will be pruned when the optimiser checks the capabilities of the platforms as defined in the resource catalogue, it will identify a conflict – the smartwatch is not capable of executing the 'persist' UDF. Physical plan number two will also be removed from the pool of suitable plans, as for this use case, the iPhone does not have any processing capabilities programmed, therefore cannot support the UDF operator. In the third physical plan, the first three operators are placed on the smartwatch and the last operator is placed

in the cloud. All of the operators are placed on platforms that are capable of executing them, hence this plan will not be discarded and its cost will be evaluated in the next phase. Furthermore, the optimiser identifies a data propagation problem, as the smartwatch is not directly connected to the cloud. Therefore, the optimiser injects the 'sxfer' operator automatically in the physical plan and places it on the iPhone to ensure the data can reach its destination for the final step of the analysis. In the fourth and fifth physical plans, the optimiser places more operators on the clouds – both of these options are checked and valid with the new 'sxfer' operator placed on the field gateway. However, when the optimiser evaluates the listed last plan, it will be discarded, as the 'getAccelData' UDF cannot be executed on the cloud resource, as defined in the resource catalogue only the Pebble Steel smartwatch is capable of sampling the raw accelerometer data.

All of the plans that successfully pass the physical plan pruning phase are passed to a cost model. There is a possibility that zero available plans remain after the physical plan pruning. In this case the application administrator will be notified and the system will be halted as there are no available plans to be deployed. Depending on the nature of the pruning process the application administrator will have to examine the logs produced during the physical optimisation to determine the root cause. The previous section outlined possible problems during this process and the application developer must check whether she defined operation that cannot be executed on any of the nodes in the system, especially the UDF operators, such as in the previously described scenario depicted on the Figure 3.5 in the plan 1, 2, and 6. If this is the case, the application administrator must amend the queries, or define the new UDF functions that are needed and the *PATH2iot* system can be rerun again. We expect that the most likely source of the behaviour that the optimiser stops at this point is when the system checks node capabilities used in the EPL statements do not match definitions within Resource Catalogue. We suggest that the spellings of any such UDFs is checked in the input files required for the system to run as defined in 3.2.

Physical Plan Enumeration for the Healthcare Use Case

In the introduced healthcare example, we focus on the analysis of the raw accelerometer data. The five EPL queries contain eight operators as shown in Table 3.2.

We now explain all of the operators and possible constraints:

- 'getAccelData' – UDF operator that generates the accelerometer data on a wearable device; it takes two parameters, the sampling frequency and mode of operation. The sampling frequency can be set to the following values: 10, 25, 50, and 100 Hz. The mode of operation lets the application administrator decide which of the data will be generated.

Operator	Wearable	Mobile phone	Cloud
getAccelData	✓	-	-
filter	✓	✓	✓
Math.pow	✓	✓	✓
window	✓	✓	✓
getCGMdata	-	✓	-
forecast	-	✓	✓
actionPrompt	-	✓	✓
parameterEstimation	-	-	✓
pushEstimates	-	-	✓
persistData	-	-	✓

Table 3.2 List of Healthcare Analytics operators with their possible placement options.

Table 3.3 list the fields that can be selected by passing the second argument to this function. Each field can either be selected – set to '1' – or deselected '0'. For example to set this UDF to sample only the 'z' accelerometer axis and a battery reading, the application administrator sets the binary flags to: 001001, which in decimal number is 9. In the running example, this is set to 60, therefore selecting x, y, z, and vibe fields (0b111100).

x	y	z	vibe	timestamp	battery
0/1	0/1	0/1	0/1	0/1	0/1

Table 3.3 Mode of operation for getAccelData UDF on a wearable.

- 'selectVibe' is an operator that examines the boolean value of the 'vibe' field and discards the events where this is true. The field is true when the haptic feedback on the smartwatch was activated. The current measurements should not be used in the analysis, as the readings are polluted by the vibrations from this module.

- 'arithmeticExpression' represents a first step in calculation of the Euclidian distance from the accelerometer measurements. The formula $x^2 + y^2 + z^2$ is applied on the data fields that are passed through this operator.

- 'Math.pow' calculates the square root on a number. As the Pebble Steel smartwatch does not contain a Floating Point Unit, a Newtonian approximation of the square root with maximum of 10 iterations has been implemented directly on the device in C++.

- 'Match Recognize' operator was introduced previously in this chapter as an extension to the SQL grammar that allows capturing of the patterns in the data streams.

- 'win' creates batches of events that are released to the downstream operator periodically.

- 'count' is an operator that counts number of events passed to it.
- 'persistResults' is a UDF that was implemented in Java on *d2Esper* stream processor. *d2Esper* is a dynamic wrapper for Esper library that was built as a part of the deployment system to test the end-to-end system functionality. It takes three arguments: the name of the event field to be persisted, the table and the name of the database. The implementation of the D2Esper stream processor handles connection to the database – in this case InfluxDB[10] time series database (see Appendix D for a Java implementation of this function).

Once all possible physical plans have been generated, a decision needs to be made on which one to select and deploy. Cost models are used to make the choice.

3.3.4 Cost Models

The logical and physical optimisation stages generate a set of viable plans, but only one must be selected for deployment. This decision is made based on the use of a cost model – each plan is costed, and depending on the non-functional requirement criteria, the plan with the lowest cost is selected.

The *PATH2iot* system has been designed to enable any cost model (or models) to be used to make this decision. In this thesis, we describe the two that were used in the running examples.

Energy Cost Model

The main focus in the healthcare use case is automating computation placement on a watch, a mobile phone and a cloud to satisfy energy non-functional requirements. The Pebble Steel watch can operate for up to seven days on a single charge. However, it has been observed that this can be significantly shorter when data processing and networking are heavily utilised.

We have developed and calibrated an Energy Impact cost model that is able to cost any physical plan passed to it and evaluate its energy impact. This informs the system of how long the battery life of a wearable device and mobile phone will be for a specific plan. A plan that satisfies the user's non-functional requirement (e.g. the battery of a wearable device must last at least two days) is selected and passed to the next stage (deployment). If more than one plan satisfies these constraints and no other requirement has been placed on the system to evaluate, the best performing plan (the one that provides the longest battery life) will be selected. This cost model is described in detail in Section 4.1.

[10]https://www.influxdata.com/products/influxdb-overview/

Bandwidth Model

The bandwidth cost model ensures that the physical plan is viable on infrastructure with networking constraints. It is described in Chapter 5.3.4 in the context of the bandwidth-constrained smart city monitoring use case, in which a raw audio signal is analysed to detect train movements.

External Modules

The *PATH2iot* system has been built as an open-source tool, and is designed to be easily extendable by adding additional modules that could further augment its capabilities.

This could be exploited when another use case requires the ability to satisfy a non-functional requirement for which there is no existing cost model.

As the *PATH2iot* system has been built in Java, a skilled programmer could follow the documentation and inline comments to make additions or changes to the code. However, external modules do not have to be written in the same programming language, as for example, the link between *PATHfinder* and *PATHdeployer* is over a socket connection. Hence its implementation is language-agnostic.

An example of this is that we have collaborated with another research group to build the first external module. ABMO [88] extends the decision-making capabilities of the tool. This multi-constrained optimisation tool, based on the Analytical Hierarchical Process (AHP) approach, has been designed to satisfy user preferences when conflicting non-functional requirements are placed on the system. User preferences include sustainability, performance and monetary cost, but others can be added. A configuration file that is loaded at *PATH2iot* when it is run contains the information needed to include or exclude an external module within the tool.

To enable ABMO to select the best plan, all of the physical plans that are left after pruning are transmitted to the external module along with their ID numbers. ABMO then returns the id of the plan that it has selected as the best to deploy.

3.3.5 Device-specific Compilation

Once the execution plan is derived, the tool translates the platform-agnostic operators into device specific code or configuration files. Because *PATH2iot* is designed to support a range of platforms – including clouds, phones and wearables – a translator is needed for each type of platform. If the platform supports EPL execution then the physical plan is converted back into a set of EPL statements. However, for other platforms, bespoke translators are needed. For example, the partition of the physical plan to be executed on the Pebble Watch is translated into device specific

configuration instructions, that the pre-installed IoT agent parses in order to start data sampling and executing of operators on the watch .

For any device, that is currently not supported by the *PATH2iot* system a new compiler must be added. Currently, compilers for supported operations utilised in the two use cases within this thesis are for Esper, iPhone, and a Pebble watch. A new compiler class must implement an abstract interface 'EplCompiler', define a standard constructor, and implement 'compile' function. We refer the developer to the existing compiler and the inline code documentation for further details.

This process is run for all infrastructure platforms that are selected for the data processing in the use case. The outputs from the device-specific compilation phase form part of the Execution Plan.

3.3.6 Execution Plan

The execution plan consists of all of the information needed to deploy software and activate the computation. It contains information such as the IP address of the message broker, the unique id of all resources, and device-specific configurations generated by the translators described in the previous section.

The final execution plan is sent over a socket connection to the *PATHdeployer* deployment module. In the current implementation there is no authentication level between the *PATHfinder* and *PATHdeployer* module. If the system is to have a public facing interface that could be exposed in the future an authentication layer would need to be added. One option to address the authentication could be with use of JSON Web Tokens (JWT), an industry standard for securely transmitting information between parties using signed tokens [90].

3.4 *PATHdeployer*: Automatic Deployment

The deployment module deploys the application across all the platforms, according to the configuration described in the execution plan file. The deployment system implemented within *PATHdeployer* consists of two parts: deployment in the cloud and IoT deployment. The deployment architecture overview is shown in Figure 3.6.

3.4.1 Deployment in the Cloud

As seen from Figure 3.6, the deployer receives the execution file in JSON via a socket connection from the *PATHfinder* module, and proceeds with deployment in the cloud, pushing the configuration details via ZooKeeper, a configuration management tool, to the *d2ESPer* stream processors. The *PATHdeployer* utilises several industry proven open-source technologies, such as Apache ZooKeeper and the Apache ActiveMQ scalable message broker. The stream computation is represented as EPL queries, and these are executed by D2Esper, which was developed in this project as a dynamic real-time data stream processor based on the Esper Complex Event Processing (CEP) library (it is described in detail in the following section).

Before the deployment in the cloud can be started, all of the cloud components must have been started by the system administrator. These include:

- the Docker instances of *d2Esper*,
- active installation of ZooKeeper server,
- active installation of ActiveMQ message broker,
- active installation of InfluxDB (if used),
- active installation of Flask REST API and MariaDB used for IoT configuration,
- pre-installed IoT agent on the mobile phones,
- pre-installed IoT agent running on the smart watches.

The *d2Esper* registers itself with a ZooKeeper node upon activation. Once a proper configuration file is delivered to the processor from *PATHdeployer*, it dynamically loads the event definition, parses the provided EPL statements and connects to a specified broker to start processing the real-time data stream. The output is forwarded to a specified destination (usually a different queue on the same broker) or a database.

Message Brokers can be used in between the field gateways and Distributed Stream Management Systems to buffer events from various sources and feed them into the processing system. The use of Message Brokers has many advantages as these tools are built to scale and provide features such as event replay (also called time travel [35]), fault-tolerance, and delivery guarantee. They traditionally operate based on a publish/subscribe model, where producers send messages to topics or queues and consumers linked to them receive these messages seamlessly.

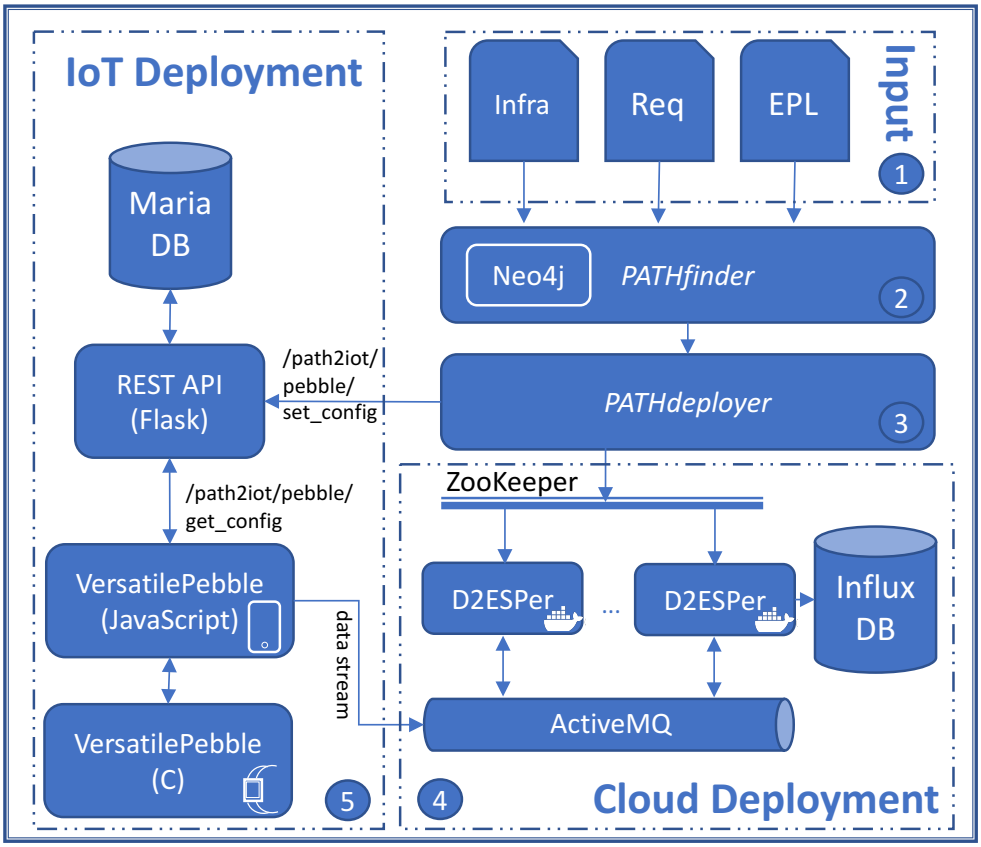


Fig. 3.6 IoT and Cloud Deployment Overview.

In the deployment architecture for the *PATH2iot* system, message brokers are used for message passing between the mobile phone and the *d2Esper* instances, but also for message passing between multiple *D2Esper* instances if a use case would require this.

3.4.2 IoT deployment

The deployment to IoT devices is triggered after the cloud deployment. This is to ensure that no events are generated from these devices before the cloud infrastructure is ready to accept them.

To enable the on-the-fly configuration of IoT devices, the system has to rely on the capabilities of agents that have been manually installed on the devices, in this case the phone and the wearable. Each agent periodically pulls and installs a configuration file from a REST API endpoint [111] as shown in Figure 3.6. REST (Representational State Transfer) API (Application Programming Interface) is set of definitions and protocols for building application software. It can be used for retrieving information from an external system or to perform a function. In this case, the IoT agent periodically queries the REST API endpoint to retrieve information about its configuration.

There are several advantages of this approach: being able to enact computation at a resource constrained device without the need for a dynamic firmware update over the air – a capability not

widely supported on edge devices – and the ability to change the computation during the runtime. The main disadvantage is that an agent offering a standardised approach for computational description and implementation must be designed and implemented for each IoT device.

Once the pre-installed agents on the IoT devices receive the configuration, they parse it and act according to it.

Although it has limited storage and computational capabilities, the agent designed and implemented for the healthcare wearable is capable of carrying out the following operations:

- Project for picking out a subset of the field in the events generated by the sensors
- Basic select operations that compare fields in an event (e.g. for equality or inequality)
- Window operations that are parameterised by time or number of events.
- A set of Library Functions that perform common operations such as basic arithmetic.

The exact computational plan to be deployed on the wearable is encoded as a binary string that the agent can interpret.

3.5 Summary

In this section we have presented the architectural design of the *PATH2iot* system. The three core inputs were described in detail: (i) high-level description of computation, (ii) a set of non-functional requirements; (iii) and the current state of infrastructure. The system parses the input from the application developer. The EPL queries are decomposed using Esper SODA API [21], the extension to the queires that allows variable window length is checked, and a acyclic graph of computation is built using Neo4j database. As well as, the resource catalogue is loaded and the representation of the computational nodes that the operators can be placed upon is constructed.

We have presented the inner workings of the core module of the system, *PATHfinder*, that uses logical and physical optimisation techniques to explore the space of the possible deployment plans. This is done in a set of stages:

- **Logical optimisation** including rules to improve the performance of the application. This process can move operations closer to the data source, it can clone the set of input queries if presence of variable window size operator is detected, and it applies safety checks to ensure logical consistency of the computation.

- **Physical optimisation** including partitioning over the available platforms. This process maps the decomposed queries and its operators to available infrastructure nodes. The system also performs a physical plan pruning where created plans are discarded if its not possible to carry out the deployment process due to constrains of the node capabilities or network connections.

- Using a **cost model** to select the best plan. This process loops through all of the available plans and calculates the cost for each. This cost is formed based on an implemented cost model. Two cost models are currently supported: energy and bandwidth. Both cost models will be detailed in the Chapter 4 and Chapter 5.

- **Code generation** for each of the available platforms. Three platform specific compilers were implemented in the *PATH2iot* system. These compilers are designed to take a high-level declarative description of computation into a device-specific implementation. In case of Esper queries, the translation output is a new set of EPL statements and for the iPhone and Pebble watch a configuration file is produced for device-specific agents. A guidence on adding new ones if needed is provided.

- Dynamic **deployment** using device-specific agents. The deployment is carried out using a pull model for the IoT devices, where each preinstalled agent regularly pulls the REST API endpoint in order to retrieve a device-specific configuration. Once this configuration is available, the agent sets the computation on the device and start processing the data stream. For the Esper enabled processing component - d2esper - the configuration update is passed to the nodes using ZooKeeper service [84] using a ZooKeeper watch function.

One of the limitations of the *PATH2iot* system are the node topology, as the system supports only unidirectional data stream flow. Another limitation to the optimisation process is presence of UDFs, that limit the ability of the optimiser to place the computation on any computation node. Further details on the limitations are discussed in Section 6.2.

All of the source code for the two core *PATH2iot* modules: *PATHfinder* and *PATHdeployer* are available at https://github.com/PetoMichalak/phd-PATH2iot under the ‘v0.1.0’ tag. The source code for further enhancements to the platform will be made available from the same repository and a placeholders for *PATHmonitor* and a *PATHviewer* modules are already present.

CHAPTER 4
ENERGY COST MODEL

Battery-powered sensors are now widely employed for many types of IoT monitoring. Their energy consumption is an essential factor in the success or failure of many of these applications. This is because battery-powered devices have a limit to how long they can operate without recharging or replacing the battery. It is therefore vital that battery life is a key non-functional requirement that is taken into account when designing applications. To achieve this, it is important to be able to estimate battery life before a device is deployed – especially if it is performing a critical healthcare-related function. If we have a way of performing these estimates, then it gives the *PATH2iot* system a significant advantage over manually designing stream processing applications that utilise battery powered devices; instead, the optimiser in *PATH2iot* can generate a set of possible options, and then – using an energy cost model that estimates battery life – select the one that best meets the battery life requirement. The focus of this chapter is therefore the design and validation of an energy cost model.

A simplistic approach to estimating the battery life of a sensor is by observation of a test deployment. This approach can be lengthy given that the battery may last for days, weeks or months. If the optimiser generates a large number of deployments then comparing their effect on battery life so as to select the best could easily be impractical. Also, any change in the processing performed on the device or in its networking behaviour would impact these tests, and the whole experiment would have to be re-run. An additional problem is that while some devices self-report battery levels, which could therefore be used to generate estimates, these are often inaccurate, as we will show through experiments. The work described in this chapter addresses these challenges by introducing cost models capable of predicting the lifetime of the battery in resource-constrained devices for a broad range of deployment options. It does this based on measuring power coefficients for the set of operations that are combined to create applications. The model takes into consideration computation, wireless data transmission and the battery capacity. It is important – especially for healthcare applications – to be able to express

the level of uncertainty in the battery-life estimates. For this we compare two approaches, one using traditional frequentist methods and another using an alternative Bayesian approach. The energy cost models are evaluated on the real-world healthcare example introduced in Chapter 3.

4.1 Cost Model

Resource-constrained IoT devices have limited hardware capabilities. One of the significant limitations is battery life. Battery-powered devices rely on carefully designed hardware and optimised software to satisfy users' expectations. Users are now expected to recharge their modern mobile phones daily, compared with the multi-week battery capacity of their predecessors, for example, the original Nokia 3310 offered up to 10 days' standby battery life.[1] However, the need for a short recharge cycle might not be so convenient for wearables. Consider a medical use case that aims to continuously capture essential data about a patient; if an application unexpectedly stops sensing in the middle of the night due to a low battery it might affect the quality of the results, and the user's healthcare. Devices currently on the market offer a variety of battery lifetimes: 18 hours in the case of Apple Watch Series 4; up to four days for Fitbit Ionic; and up to 25 days for the Nokia Steel HR smartwatch.

Application designers require a reliable way to estimate battery life under different networking and computational demands so as to ensure continuous monitoring, and predictable recharge needs. In this chapter, we show that an energy cost model can be used to estimate the battery life of a device running an application. The presence of accurate estimates is of key importance for many application areas, especially where a battery pack replacement is infeasible or prohibitively costly [66]. A feature of *PATHfinder* is the ability to use a cost model to select the best plan. The cost model combines the power impact of individual operators, data transmission costs and the initial state of the battery. *PATHfinder* can then generate a battery life estimate for each possible plan, and recommend the best. This gives the application administrator an estimate of the energy demand of the best plan, and the uncertainty of the estimate, before they make the decision to go ahead and deploy the plan.

4.1.1 Power Impact Model

For the healthcare use case, we can assume an infinite (or easily replenished) battery capacity on the mobile phone, as experience shows that the wearable's battery life is the critical issue. We therefore focus our analysis primarily on the wearable – however, the approach taken to generate

[1] https://www.gsmarena.com/

a battery life estimate can be applied to any device, including mobile phones. For a wearable, there are three major components of energy consumption:

- the operating system constantly running in the background; in the use case, the main computational impact on the Pebble Steel smartwatch comes from the FreeRTOS[2] real-time operating system;
- data sensing and processing;
- data transmission from the wearable (usually to a mobile phone).

The Energy Cost Model developed for *PATHfinder* combines estimates for the power consumed by these three components using the Formula 4.1. This formula estimates power usage, rather than energy, as we are modelling a constantly-running stream processing system.

$$\begin{aligned} PowerImpact[mW] = {} & OS_{idle}[mW] + \\ & \sum_{i}^{n} comp\_cost_i[mW] + \\ & msg\_cost[mJ] * msg\_count\_per\_s + \\ & \frac{\sum_{j}^{m} RF_{overhead_j}[mW] * RF_{duration_j}[s]}{cycle\_length[s]} \end{aligned} \tag{4.1}$$

In the first part of the formula 4.1 of the power impact includes operating system running without any computations. This is expressed as OS_{idle} and the measurement is in milliwatts as this is typically seen in the small wearable devices. Then, the formula introduces a sum of all operations that are placed on the device. The i is an index of computational operators and n is the total number of operators that the *PATHfinder* optimisation module placed on the device. These operations are in milliwats. The next part of the formula calculates the energy cost involved with transmission of the messages between individual devices. An energy, in millijoules, required to transmit a single message is multiplied by the number of messages that the system calculated to be transmitted per second, result returned in milliwatts. The last part of this formula calculates the necessary overhead when working with wireless data transmission. There are two types of overhead: establishing the wireless connection and its duration, and wireless transmittor being active even when not in use after the transmission has ended. The total number of wireless transmission factors is denoted by m and we use j to iterate over them. This cost is then normalised by the cycle length which we provide more detail in the following more detailed description.

[2]https://www.freertos.org

All power impact coefficients are summarised in the Table 4.2 with description on how they have been procurred. We now consider each of the components in more detail in turn:

- Operating System (OS_{idle}) – the power consumption on the IoT device caused by the operating system. This is independent of the energy expended by the application. The value varies depending on the device, and the version of the operating system running on it.

- Computation – every computation performed in an application adds to the overall power consumption of the device. Each operation deployed on an infrastructure platform has an assigned device-specific energy coefficient (*comp_cost*) derived from benchmarking tests. These are then all summed to give the overall power drawn by all operators on that platform. Performing the benchmark tests to measure *comp_cost* is a time-consuming process, which will be described in the next section.

- Networking – networking has a significant impact on battery life. Messages transmitted from the wearable to other platforms (e.g. the mobile phone) are costed in this part of the formula. A complex aspect of networking in the running example is the Bluetooth Radio Active State. This state is entered every time a message is sent from the wearable device. The OS then keeps the radio module active for a period of time in case there is another message to be sent. Figure 4.1 shows the power drawn during the four phases of the 120s Bluetooth Active State Cycle – 1st phase: connection establishment; 2nd wireless data transmission; 3rd $RF_{overhead}$; 4th OS_{idle} with data sensing, preprocessing and *RF_standby* cost, after which the cycle repeats (5th mark). We represent this in the model by including the power drawn in the Active State ($RF_{overhead}$) and multiplying this by the fraction of the time that the wearable is in the Active State ($RF_{duration}/cycle\_length$). The amount of time that the Bluetooth radio is in this state varies and is discussed further in the following section. To take into account the message passing costs, the power expended sending messages is calculated by multiplying the energy cost of sending a single message (*msg_cost*) by the number of messages sent per second (*msg_count_per_s*). This number is calculated in Formula 4.2 by taking the number of bytes transmitted from the watch to the mobile phone *data_out*, divided by the maximum payload size per message *max_msgpayload*, rounded up, and normalised by the duration of the cycle in the plan: $cycle\_length$.

$$msg\_count\_per\_s = \frac{\lceil data\_out[B]/max\_msg_{payload}[B] \rceil}{cycle\_length} \tag{4.2}$$

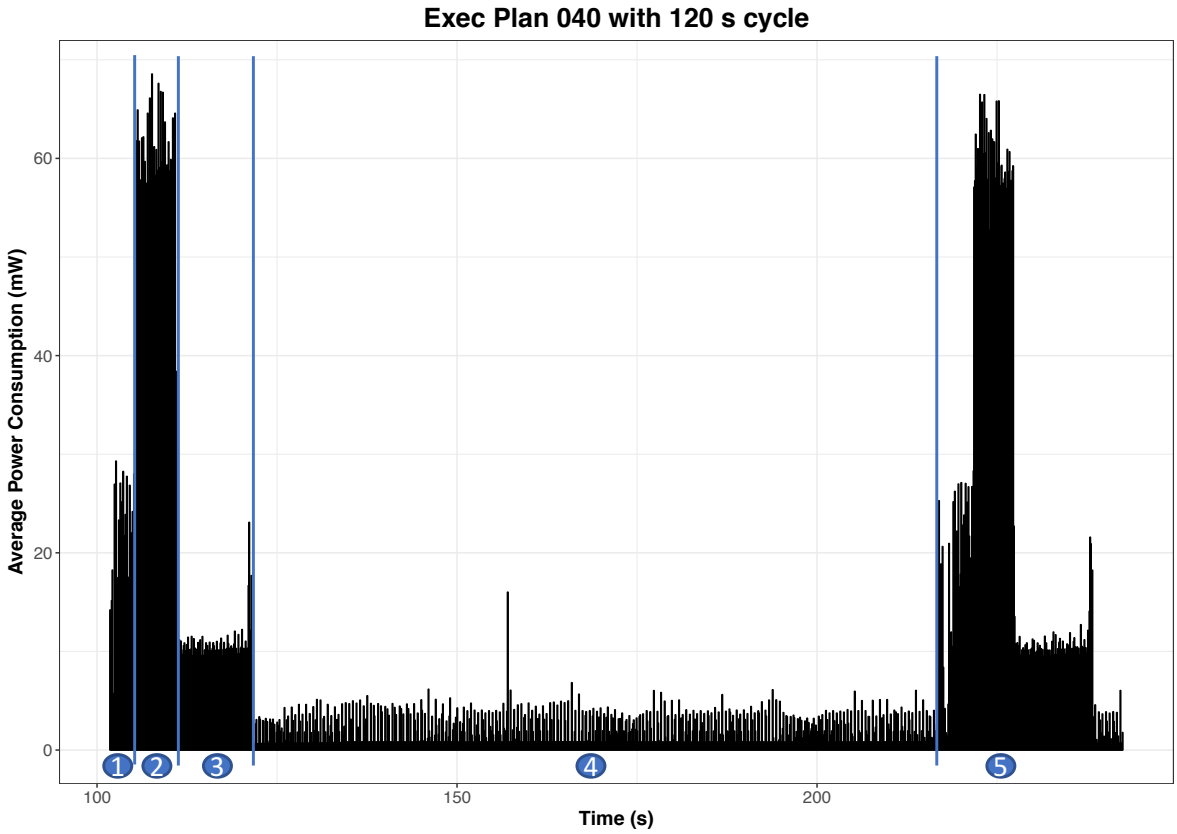


Fig. 4.1 Power consumption of Pebble Steel smartwatch: 120 second window split into four phases: (1) Bluetooth radio establishing connection; (2) data transmission; (3) Bluetooth radio active, but not transmitting; (4) data processing; (5) cycle repeats.

Units in Formula 4.2 result in number of messages per second. The $data\_out$ and $max\_msg_{payload}$ are in Bytes and the cycle length is in seconds.

4.1.2 Power Coefficients

To perform calibration measurements, an experimental testbed was set up with a Pebble Steel smartwatch, an LG G4 smartphone and a cloud environment. A set of experiments was carried out on the testbed to measure the power coefficients for the individual operations, and for the transmission of messages.

The experimental setup of the Pebble Steel smartwatch and a Monsoon Power Monitor is shown in Figure 4.2. This tool is a combination of a benchtop power supply and a measurement device with very high sampling frequency – up to 5,000 Hz. A battery bypass had to be performed so that the device is powered through the Monsoon Power Monitor.

A similar procedure has been carried out for the battery of the mobile phone. The main difference was that in this case the battery removal was straightforward. However, the battery microcontroller had to be stripped from the battery cell as the Android operating system monitors

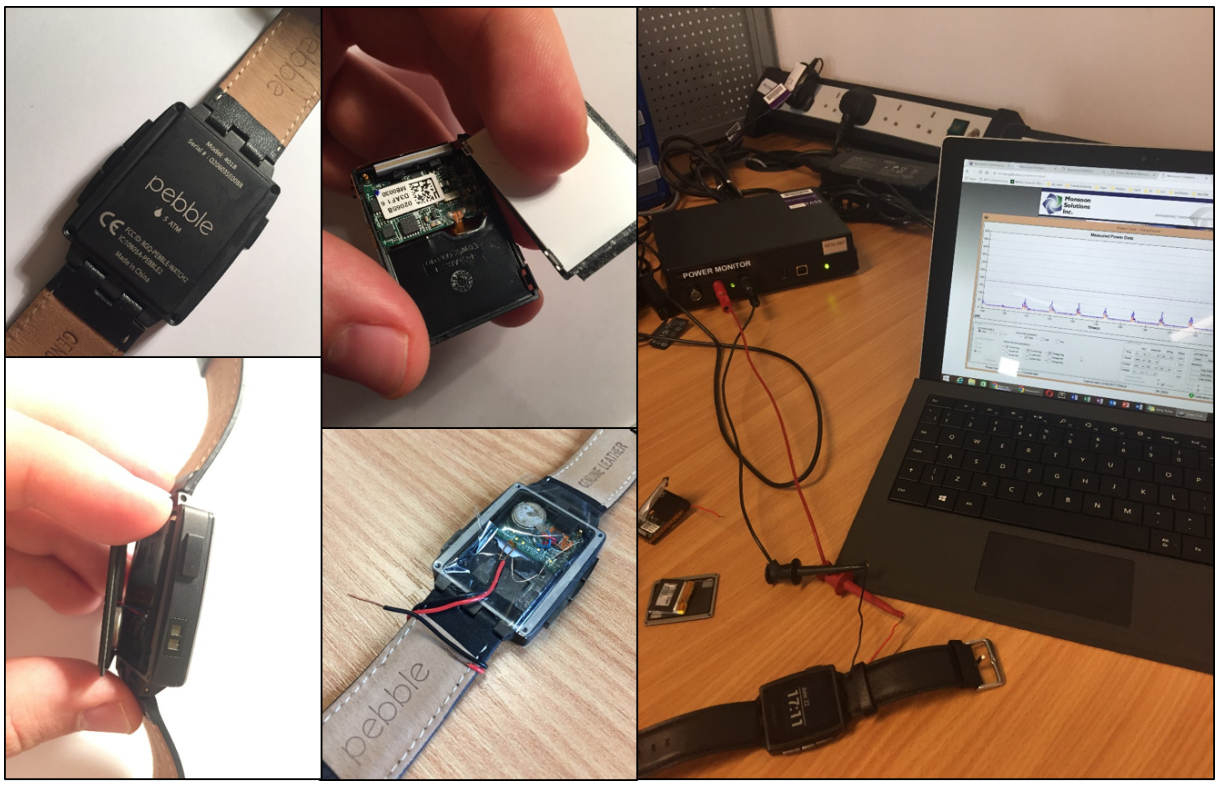


Fig. 4.2 The Pebble Steel smartwatch battery bypass procedure: the built-in battery is removed, and the device is powered directly through the Monsoon Power Monitor.

the battery condition through battery state indicator pins [23]. The need to use the battery microcontroller is due to Android checking the state of the battery, hence directly powering the phone results in Android detecting a missing microcontroller and shutting the phone down shortly after it boots.

This section outlines the approach used to calculate these power coefficients. They vary depending on the device and a version of the operating system running on it. They therefore need to be benchmarked for each device.

Computational Impact

The power coefficients for individual operations were calculated in a set of experiments. The baseline for the power consumption is formed by the smartwatch operating without any computations or data transmission running on it. During the experiment, the device was left idle without any human interaction as this would also cause additional workload in order to handle the events; this would also cause the backlit LED screen to switch on, so increasing power consumption. Further, flight mode was activated on the smartwatch to disable any network activity. The following protocol was observed for all of the experiments: (i) the watch was programmed with the new configuration; (ii) measurements did not begin until after a warm-up

period was over, to ensure the power impact coefficients when the watch was in a steady-state; (iii) the power expended over 1000 continuous one second cycles was measured by the monitor.

The following pseudo-code describes the operation of the pre-installed agent on the Pebble Steel watch which is configured by a message sent to it by the *PATHdeployer* deployment system.

```
// Pebble Agent - configurable accelerometer data processing agent
for sample in samples
  if select_flag && did_vibrate
    continue
  if ed_flag
    unint32_t ed = x*x + y*y + z*z
    if sqrt_flag # calculate euclidian distance
      buffer[buffer_pointer++] = newtonian_sqrt(ed)
    else
      buffer[buffer_pointer++] = ed
  else // buffer accelerometer readings - sending all data
    buffer[buffer_pointer++] = x
    buffer[buffer_pointer++] = y
    buffer[buffer_pointer++] = z
    if not select_flag
      buffer[buffer_pointer++] = did_vibrate

if data_pointer >= win_buffer
  buffer_pointer = 0
  send_buffer
```

The 'select_flag', 'did_vibrate' and 'ed_flag' in the pseudo-code are part of the configuration received from the *PATHfinder* module which determines what part of the computation should be enacted. Once the 'data_pointer' counter is greater than the 'win_buffer' a function to start the data transmission from the watch to the mobile phone is triggered.

This code is executed every time a callback method on the watch accumulates a batch of accelerometer samples, for this use case this number was set to 10, causing the callback method to fire 2.5 times every second. For each experiment, we have differed the amount of computation that was executed on the smart watch. For the real world deployment this is done by changing the flag value that was described in Section 3.3.3. Each sample consists of:

- a triaxial accelerometer measurement (x, y, z);
- a vibration module flag (did_vibrate);
- a timestamp.

ID	DATA	WIN	SELECT	ED	POW	Power Impact (mW)
001	-	-	-	-	-	0.8281218
002	✓	-	-	-	-	1.6381417
003	✓	✓	-	-	-	1.6390818
004	✓		✓	-	-	1.6692631
005	✓	-	-	✓	-	1.6623610
006	✓	-	-	-	✓	1.6611211

Table 4.1 Power impact experiments.

A comprehensive set of power consumption experiments was designed to calculate the individual power costs for each operator. Table 4.1 presents a summary of these experiments executed on the Pebble Steel smart watch: the checkmark symbol in a column indicates that the given operator was active for that plan.

A detailed explanation for all the columns within this table is:

- ID – a unique identifier for the plan;
- DATA – the operator that generates the triaxial accelerometer data stream on a Pebble Watch – this is configured to generate a window of 25 samples every second;
- WIN – an internal array used to store data in a buffer before transmission;
- SELECT – removing any events that were collected while the vibration module on the watch was turned off (`WHERE did_vibrate=0`);
- ED – calculating the Euclidean distance from the raw triaxial data. $ED = x^2 + y^2 + z^2$. This reduces the amount of data propagated to the next operator by a factor of 3.
- POW – a Newtonian approximation of a square root (limited to 10 iterations) that we implemented as the Pebble Watch does not have a Floating Point Unit on the chip;
- Power Impact – the power measurement from 1000 one-second continuous cycles were averaged, however as shown in Figure 4.1 there is a lot of variation in power measurements. Hence, we implemented two ways to capture uncertainty of these power measurements: one using frequentist approach, and second using alternative Bayesian approach to capture uncertainty.

It is not possible to run every computation operator individually without data being collected, but from these measurements, it is possible to separate out the operating system and other operator power coefficients.

Operation	Power Impact (mW)	Confidence Interval 95%
OS_{idle}	0.82812	$\pm$ 0.0008273
$Data$	0.81002	$\pm$ 0.0086507
WIN	0.00094	$\pm$ 0.0809910
$SELECT$	0.03112	$\pm$ 0.0089773
ED	0.02422	$\pm$ 0.0089684
POW	0.02298	$\pm$ 0.0089947

Table 4.2 Power impact coefficients for computation operations (rounded to 4 decimal places).

Table 4.2 presents these power coefficients, along with their confidence interval. The first row in Table 4.1 (ID 001) gives the power consumption of operating system (OS_{idle}). The cost of the *DATA* operator is calculated by subtracting the power consumed by the operating system from cost of running the DATA operator on the operating system (Table 4.1 ID 002). The same approach is used to calculate the power consumed by the remining operators: WIN, $SELECT$, ED and POW. Section 4.2 describes how the uncertainty in these coefficients was calculated.

Network Power Impact

Wireless data transmission between IoT and Edge devices incurs a significant energy cost. We have estimated the energy coefficients for the individual phases in the multi-stage process described earlier.

In the experiments, the Pebble Steel smartwatch was programmed to sample 25 Hz accelerometer data, select only events where 'vibe=0', calculate the square root of the Euclidean Distance, and window the output data stream using a 120 second tumbling window, then send the result to the mobile phone over Bluetooth radio. Figure 4.1 displays power measurements captured from the Monsoon Power Monitor for four Data Transmission phases: (i) establishing a connection; (ii) data transmission; (iii) Bluetooth radio is active and (iv) Bluetooth radio in standby mode. We analysed the power trace file so as to separate out these phases in order to calculate their energy impact and duration. Average durations of the phases were evaluated for 12 cycles, with a payload of 125 bytes per message transmitted from the smartwatch to the smartphone in 24 consecutive messages for each cycle.

Table 4.3 shows a sample of analysed cycles with power consumption and timing information for each of them.

The average power consumption for each of the Bluetooth radio phases was calculated: 33.3368 mW, 84.6330 mW, and 26.6801 mW for $RF\_establish$, $RF\_transmitting$ and RF_active respectively; as summarised in a Table 4.4. The table also provides duration for each phase.

message count	Cycle duration (s)			Power Impact (mW)		
	$RF_{establish}$	$RF_{transmitting}$	RF_{active}	$RF_{establish}$	$RF_{transmitting}$	RF_{active}
24	0.6	1.5	11.6	33.2322	78.4963	26.5284
24	0.9	1.4	11.5	21.4008	81.5545	26.5270
24	0.2	1.5	11.9	37.2701	80.1947	26.5880
24	0.7	1.3	11.7	26.2966	84.6375	26.9211
24	0.8	1.6	11.0	32.1490	76.3040	26.7027
24	0.7	1.2	11.7	27.6356	90.2488	26.7373
24	0.5	1.2	11.7	37.5392	87.1262	26.4826
24	0.7	1.2	11.6	33.1917	85.6604	26.6618
24	0.9	1.2	11.9	24.7142	88.8800	26.7785
24	0.5	1.2	11.6	34.3730	88.7293	26.7935
24	0.8	1.2	11.6	30.5015	87.5960	26.6355
24	0.1	1.3	12.1	61.7375	86.1686	26.8046

Table 4.3 Pebble Steel watch Bluetooth phase durations measurements for three phases: establishing connection, transmitting data, and transmitter being active after the data transmission with corresponding power impact measurements.

Operation	Power Impact (mW)	95% CI	Duration (s)
$RF\_establish$	33.3368	$\pm$ 5.7831	0.6
$RF\_transmitting$	84.6330	$\pm$ 2.5359	1.3
RF_active	26.6801	$\pm$ 0.0756	11.7

Table 4.4 Power impact coefficients for networking operations.

These measures explain why varying window length has a significant impact on battery life. The longer the window, the less power the wearable device consumes. This is the result of the two $RF_{overhead}$ phases included in the power impact formula: establishing a connection and keeping the Bluetooth module active. These phases have high energy impact on the battery. Sending the messages in rapid succession, and transmitting all the available data in short bursts reduces the overall power consumption, compared to sending every message individually. The power savings comes from reduced need for the re-establishing the connection with the mobile phone, and also the Bluetooth module can spend more time in the power saving mode. However, it must be borne in mind that there may be practical limits on the window length. In the running example, storage in the Pebble Watch is limited (as will be seen, this places an upper limit of 120s on the window length), while in time-sensitive use cases, delaying sending data until a long window fills may increase the time before a vital alert or action can be raised.

When an additional operation needs to be included in an application, an estimate needs to be provided of its energy cost. There are two possible options for this: (1) follow the previously outlined procedures to benchmark and derive new power coefficients; (2) compare the complexity

of the new operator with existing operations and use the energy cost of the nearest as an estimate of the impact of the new operation.

The former is the most accurate, but can be time-consuming and requires specialist equipment and software experimentation; for this reason, the latter may be acceptable.

4.1.3 Battery Capacity: Charging Strategies

Battery capacity refers to the charge held within the battery. This is measured in Ampere per hour (Ah), but usually given in milliampere per hour for IoT devices. It is a representation of the amount of energy that can be extracted from the battery under specific conditions. It is a key measure that can be combined with the result of the cost model presented in the previous subsection, in order to estimate how long a device will run before the battery needs recharging or replacing.

However, as will be explained, the battery capacity information on its own is not enough to calculate the estimated battery life. This is because it depends on which charging strategy is utilised.

In this section, we outline two significantly different charging strategies, and describe the impact they have on battery life estimations. To estimate the battery life of a smartwatch, information on the total energy capacity for the battery is needed. This was calculated using the following formula:

$$\begin{aligned} maxEnergyCapacity[J] &= batteryCharge[mAh] \times batteryVoltage[V] \times 3.6 \\ &= 130mAh \times 3.7V \times 3.6 = 1731.6J. \end{aligned} \tag{4.3}$$

The battery charge in milliampere per hour and the battery voltage in Volts are normally provided in the technical documentation, or can be read directly from the label on the battery. For this use case these are 130 mAh and 3.7 V respectively for the battery used. The product of these two values is multiplied by the normalisation constant 3.6 to convert the battery charge from milliampere hours to ampere seconds (the number of seconds in an hour – $60 \times 60 = 3600$ – is divided by 1000).

Self-reported Battery Level vs Real Energy Capacity

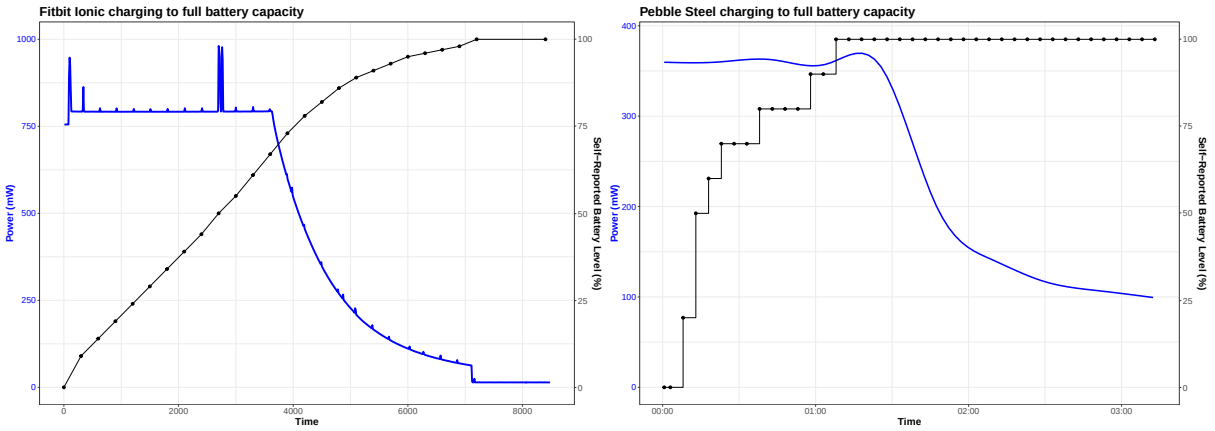
The battery life of an IoT device depends on several factors that introduce uncertainty into the expected battery life estimates. These include:

- battery hardware design – the material used to build the battery, battery age, previous charging cycles, degradation of battery capacity over time and the battery self-discharge if not used over a longer period of time all have an impact on the overall battery life.
- battery charge – devices often estimate the battery capacity based on the current battery voltage and provide a self-reported battery level to users. This information might not be precise, and can lead to overoptimistic expectations, as is demonstrated in this chapter. We show that the duration of the charging cycle after the self-reported battery levels display full capacity has a significant effect on battery life.
- battery discharge – as has been shown, when a device is being used it consumes energy due to:
  - computation – this includes data sensing, pre-processing, and windowing;
  - networking – wireless data transmission is one of the most energy demanding tasks that can run on a typical IoT device.
  - other factors, such as the impact of the operating system running on the device, powering up power hungry sensors (for example a Heart Rate sensor), powering up the display, haptic feedback etc.

This observation is significant for managing users' expectations and those of application administrators when estimating the remaining battery life of IoT devices under different execution plans. The users might expect that a self-reported 100% battery mark suggests several days of battery life, but be adversely affected when this does not come to pass. From the technical specifications, the Pebble Steel smart watch should be able to run on a full charge for up to seven days, when displaying only current time with low refresh rate (one per minute) and no additional processing or networking activity is executed. Application administrators might expect this to be the duration for the device to be used, but be surprised when this is not the case, especially when they do not have access to any measures of the uncertainty in such estimates.

In some instances, the user is misinformed by the self-reported battery life of a wearable device. This is the case for a Pebble Steel smartwatch, which self-reports a battery level of 100% even when this does not reflect the actual battery state. We will show that the real battery capacity can vary significantly from the self-reported battery charge due to the charging strategy.

Figure 4.3 presents a comparison of the battery charging cycles of two wearable devices: a Pebble Steel and a Fitbit Ionic smartwatch. This experiment was designed to show the differences in the strategies used to inform the user about the current battery capacity during charging. A



(a) Fitbit Ionic (b) Pebble Steel

Fig. 4.3 Self-reported battery levels: full charging cycle.

fully discharged device, i.e. a device unable to switch on when the power button is pressed due to empty battery capacity, was connected to a charging cable that was powered through the Monsoon Power monitor. This captures the amount of power that is supplied to these devices during the entire charging process. This is shown in blue on the graphs. The battery levels reported to the user are shown in black.

The power supplied to the Fitbit Ionic starts to drop once the battery levels are reported to be at 67% of maximum capacity, as shown in Figure 4.3a. The level then very slowly approaches 100%. Once it is reached, the power intake drops rapidly and plateaus. We argue that this is an accurate indication that the battery is at full capacity. However, the charging cycle of the Pebble Steel smart watch, presented in Figure 4.3b, gives a different view to the user. The user is told that full capacity has been reached shortly after the first hour of charging. However, the energy supplied via Monsoon Power Monitor does not plateau for almost another two hours. We further investigated this anomaly using a battery stress test experiment.

Pebble Steel Battery Stress Test Experiment

To further illustrate the importance of understanding charging cycles, we designed an experiment where the performance of the same Pebble Steel wearable device with a fully charged battery is compared with a battery that was disconnected from charging at the self-reported 100% battery capacity mark. Based on the graphs in Figure 4.3, we define a fully charged battery for this wearable device at a point when the charging cycle is considered fully completed after 190 minutes. A specially dedicated watch application was designed for this experiment - it rapidly drains the battery of the device by repeating the following cycle:

1. collect accelerometer data from the built-in accelerometer sensor at 25 Hz,

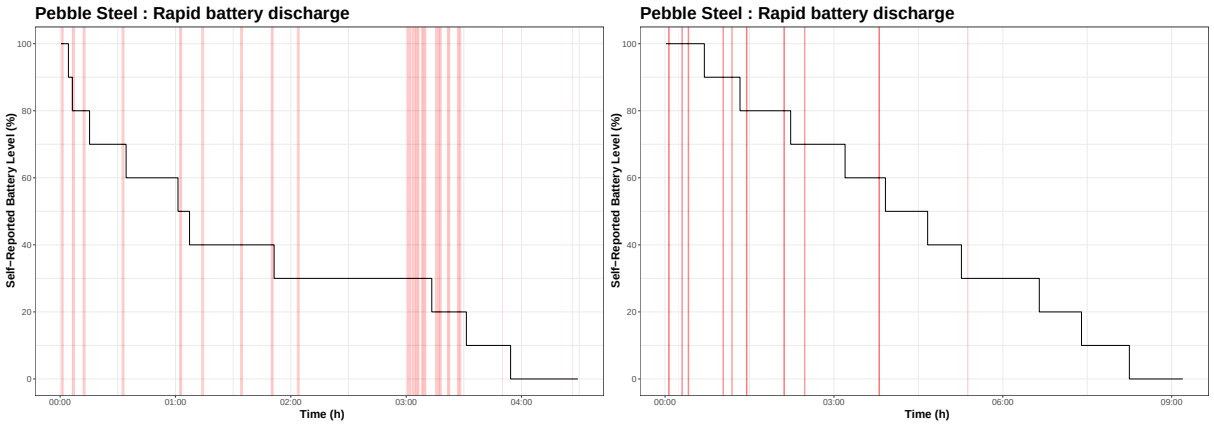


Fig. 4.4 Pebble Steel battery stress test under 100% self-reported and full charge.

2. batch the data into windows of 10 samples,

3. transmit it to the mobile phone over the Bluetooth connection every 400 ms, i.e. at 150 messages per minute with a message counter that is incremented with each cycle. The data payload of each message was 93Bytes: 8B timestamp, 80B accelerometer data, 1B battery level, 4B current message counter.

In the first experiment, the watch was charged until the self-reported 100% battery capacity notification was received (after approximately 70 minutes), then the dedicated watch application started to stream the data – rapidly draining the battery. The watch was left to expend all of its battery's energy, and so the last transmitted message was received after 4.5 hours, at which point the watch shut down. However, when the watch in the second experiment was charged for an extra two hours after the self-reported full capacity notification, the watch transmitted 43,720 more messages than in the first experiment. The extra charging time increased the battery time to 9.2 hours. Both tests are presented in Figure 4.4 with step plots displaying the remaining self-reported battery levels and the red vertical bars capturing the lost messages, extrapolated from the message counter information that was sent from the watch within each message. The process of detecting lost messages was based on comparing the message counter generated by the watch and passed through the payload to the mobile phone. As this counter is incremented linearly it is straightforward to detect when a message was missed by detecting any gaps in received messages on the mobile phone.

This battery stress test confirms our hypothesis that the battery charging cycle is not complete when the Pebble Steel watch notifies a user that it is fully charged. Further experiments confirmed this claim. Table 4.5 lists supplementary test runs.

Charging duration (m)	Discharge duration (m)
42	191
119	304
386	376

Table 4.5 Pebble Steel battery test results under rapid discharge.

Results show that charging the battery for an additional 77 minutes after it reached the self-reported full capacity extends battery life by 59%. Charging a battery to its full capacity extends the battery life by almost 97%.

The analysis in this section shows that the self-reported battery charge level cannot be used as the basis for our scientific experiments – we need to rely on the additional evidence provided by the power monitor.

In this work we operate under the assumption that the battery charge is at its full capacity – with the extra charging times – and ignore the self-reported battery level for the Pebble Steel smart watch device.

Data Transfer Energy Impact on Mobile Phone

The energy impact of the smartphone device, an LG G4 mobile phone, has also been examined by directly observing the power consumption using the Monsoon Power Monitor showing that the Energy cost model, previously introduced, can be reused for any IoT battery-powered device.

In the case of a mobile phone, some differences must be taken into consideration when measuring the power consumption of an application, when compared to a wearable, including:

- OS background tasks – an operating system has many more background services running independently of its foreground application (e.g. location services, periodic cellular communication with the nearest base station, wi-fi scanning, app updates). These background activities cannot be directly controlled for in a real-world use case;
- application lifecycle – when another application requires computational resources, the monitored application can be pushed into a paused state, or be terminated by the OS [24];
- the number of notifications and user interactions that cause the display activation, which has a significant energy drain, can vary greatly and modelling of these is out of scope for the research work.

The hardware setup used for this experiment is displayed in Figure 4.5, where a battery bypass of the LG G4 mobile phone was carried out to measure the power consumption of the

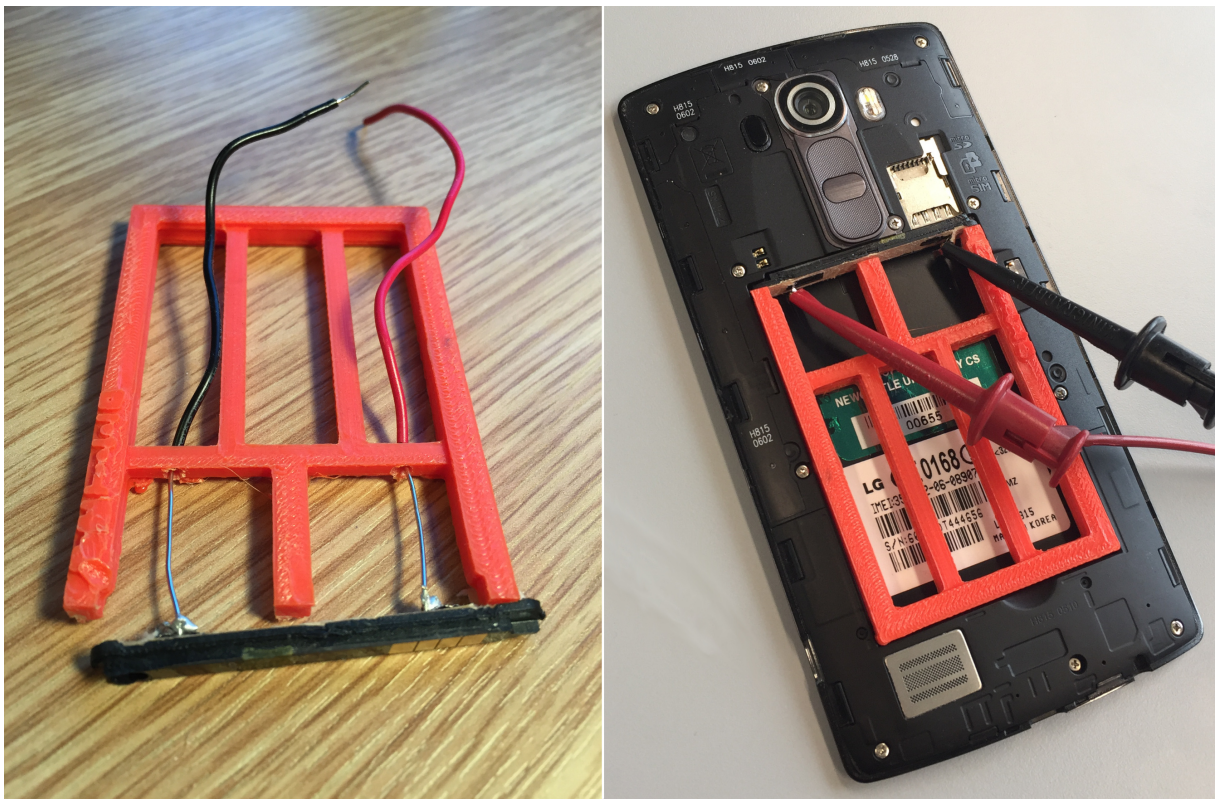


Fig. 4.5 LG G4 battery bypass with 3D-printed holder.

mobile phone directly. At first, we connected the Monsoon power monitor to the battery pins on the mobile phone directly. However, failing to supply the battery state information that the built-in battery microcontroller provides results in the immediate shut-down of the phone after it has booted up. To address this problem, we disassembled the battery cell itself, so that its built-in battery controller could continue to provide battery state information to the mobile phone, without detecting that it power comes from the Monsoon power monitor instead of battery cells. This required a physical holder to keep the battery microcontroller in place, hence we designed and 3D-printed a customised frame to secure the placement of the battery controller as shown in the Figure 4.5.

Table 4.6 gives the Power Impact coefficients for the OS_{idle}; this is the power consumption of the mobile phone with Bluetooth, WiFi and cellular connection idling. The cost of sending messages to the cloud, and the cost of RF_active are calculated with the phone only running the $sxfer$ data relay operator. These results were evaluated from two experiments to separate the energy impact of OS_{idle} and the data transmission costs. In the baseline plan, messages are streamed directly from the Pebble watch every 400 ms, and these are forwarded immediately to the ActiveMQ [141] message broker running on the cloud. In the optimised plan, messages are sent in rapid succession after a 120s window expires. This part of the experiment clearly shows

Operation	Energy Impact (mJ)	95% Conf Int
OS_{idle}	56.28	$\pm$ 4.520
msg_cost	161.62	$\pm$ 6.813
RF_active	2497.10	$\pm$ 226.288

Table 4.6 Power Consumption Coefficients for the LG G4 mobile phone.

that buffering the messages has a significant impact on the battery lifetime of the smartphone also.

We can calculate the total battery capacity of the mobile phone as: $3000mAh \times 3.85V \times 3.6 = 41,580J$. This information has been used to calculate the expected battery life under the two plans. To calculate the estimated battery life of the device, we use the total battery capacity and divide it by the energy impact. Without any additional processing or usage the mobile phone is estimated to run in idle mode for around 205 hours ($41580J/56.28mJ = 738806s = 205h13m$), with confidence intervals giving an interval of lower and upper boundary of 190 hours and 223 hours respectively.

The battery impact was calculated from two experiments. First, the mobile phone was monitored for 30 minutes where no data has been transmitted between the smartwatch and the mobile phone. A period of 1250 seconds was selected, omitting the initial warm-up where the power consumption fluctuated. One-second windows of power averages were calculated, and the main power consumption with 95% confidence intervals.

The second experiment was carried out from the optimised plan, where the smartwatch was transmitting the data every 120 seconds. These data were sent via a Wi-Fi connection to the ActiveMQ broker without delay; hence the *msg_cost* contains the combined power consumption of the Bluetooth incoming message and the forwarded MQTT message to the cloud-based message broker. The Monsoon power monitor log file has been analysed and annotated. Table 4.7 presents the transmission duration and power for the two phases from selected eight cycles: (i) Transmission – where messages were passed from the smart watch to the cloud; and (ii) RF Active – where the RF module was active, but not transmitting any data messages.

The average duration of data transmission is 10.45 seconds, and the average RF_active duration lasts for 15.7 seconds. The energy coefficients for the RF_active phase have been calculated as the average power consumption of this phase times the average duration: $15.7s \times 159.051mW = 2,497.1007mJ$. The calculation of the energy cost of sending a message has been evaluated as the average duration of a single message: $10.45s/60 = 0.17417s$ multiplied by the average power drain during the data transmission phase: $0.17417s \times 927.943mW = 161.6198mJ$.

Transmission Duration	Transmission Power	RF Active Duration	RF Active Power (mW)
10.4	912.994	15.5	161.617
10.3	917.422	15.7	146.392
10.1	937.280	15.7	185.368
10.9	893.962	15.7	146.034
9.7	997.584	15.6	187.398
12.1	850.493	15.7	152.469
9.8	988.645	16.1	146.256
10.3	925.164	15.6	146.875

Table 4.7 Data transmission phases - power consumption.

The 95% Confidence Intervals (CI) were calculated using the formula:

$$95\%CI = \bar{X} \pm t_{n-1,0.975} \times \frac{sd}{\sqrt{n}} \tag{4.4}$$

where sd is the standard deviation, n is the sample size and $t_{n-1,0.975} = 2.306$ is the 97.5% point in the Student's t distribution.

4.2 Uncertainty in Battery Life Estimates

When analysing the power measurements of an IoT device obtained from a set of experiments, it is evident that there is variation in the results. Point estimates for the power consumption of a specific physical plan would not provide information about this variability. It is therefore crucial to incorporate uncertainty so as to allow the application administrator to make an informed decision: for example she might want to be cautious in the case of a healthcare device that a patient relies on.

4.2.1 The 95% Confidence Interval Calculation

The use of Confidence Intervals (CI), a Frequentist approach to expressing uncertainty, provides additional information to the application administrator about the range of expected values compared to only a point estimate. As defined in [98], CI determines an interval of values that would not be rejected by a significance test. In the case of 95% CI this range allows for a 5% rate of false alarms. In this section we will explain how the confidence intervals were calculated and the way they are applied within the cost model [25].

We can estimate the 95% confidence intervals using the following formulae:

$$
\begin{aligned}
lower\_limit &= EI - 1.96 \times \sqrt{\sum_{i}^{n} a_i^2 Var(\bar{X}_i)} \\
upper\_limit &= EI + 1.96 \times \sqrt{\sum_{i}^{n} a_i^2 Var(\bar{X}_i)}
\end{aligned}
\tag{4.5}
$$

where n is a set of all operations contributing to the calculation of given Energy Impact coefficient; a_i is a coefficient that specifies the proportion of the duration within a cycle that the operation lasts; and $Var(\bar{X})$ denotes the variance of the sample means being calculated.

The first step is to calculate the 95% CI for OS_{idle} power impact coefficient using the Formula 4.5. The power measurements were split into one-second windows, and the arithmetic mean of each was calculated. This data was run through the formula to determine the Power Impact coefficient of 0.82812 mW with the lower_limit of 0.827293 and the upper limit of 0.828947. This approach was repeated for all experiments where the computational impact was to be calculated. It can be observed that the confidence intervals grow with each additional operation in the plan, as the uncertainty accumulates.

This approach to expressing uncertainty provides valuable information and gives the application administrator a deeper understanding of the battery performance that a user can expect. However, as we explained previously, the use of 95% confidence intervals with an assumption of mutual independence of the variables and an assumption of approximately normal distributions to calculate these intervals might result in narrower intervals than will be the case in reality. Hence, we investigated a more robust approach that could address these limitations.

4.2.2 Bayesian Approach to Capturing Uncertainty

Kruschke [98] argues that Confidence Intervals suffer the same problems as p values and a yes/no hypothesis testing, and they do not represent a probability distribution, but merely two end points. To see if we could improve on this approach, we employed more directly interpretable method based on Bayes' Theorem [72] to capture the uncertainty of all of the operations and utilise Bayesian Regression to make our model 'more useful' following Box's famous aphorism "All models are wrong, but some are useful". The main difference between the first – frequentist – approach and the second – Bayesian – approach is that the latter works with distributions instead of point estimates, therefore quantifying meaningfully all uncertainty within the process.

The mathematical definition of the Bayes' Theorem states

$$
P(A|B) = \frac{P(B|A) \times P(A)}{P(B)} \tag{4.6}
$$

where A and B are events, that $P(A|B)$ is the conditional probability of event A occurring given that event B has occurred and $P(B|A)$ represents the reverse of the previous statement. The $P(A)$ and $P(B)$ are marginal probabilities of individual events happening. The process of inductive learning via Bayes' rule is referred to as Bayesian Inference, where the expression of belief about an unknown quantity is expressed by a probability, followed by application of this rule the beliefs about the model parameters utilising the new information that comes from the dataset.

In Bayesian statistics everything is expressed in terms of a probability distribution, and the posterior distribution is calculated from the prior distribution, which describes the belief of true population characteristics before seeing the data, and a sampling model via Bayes' rule [72]:

$$p(\theta|y) = \frac{p(\theta) \times p(y|\theta)}{p(y)} \tag{4.7}$$

where θ represents the unknown quantity and y is the observed dataset.

The same data have been analysed as in the previous approach. These have been partitioned into one-second intervals, each categorised as follows:

- bluetoothState - categorical variable with four states
    1. inactive - the Bluetooth module is powered down,
    2. establishing connection - the Bluetooth module is attempting to establish a wireless network link with the mobile phone,
    3. data transmission - data transmission between the smartwatch and the mobile phone,
    4. Bluetooth module active - the Bluetooth module is in ready mode as the Bluetooth connection has not been severed yet.
- data - 0/1 – data is, or is not, being sampled from the accelerometer
- select - 0/1 – filtering the events where is_vibe = 0
- ed - 0/1 – calculating the Euclidian distance from the raw triaxial data
- pow – approximation of square root using Newtonian approach with 10 iterations
- win – applying a window to buffer the data

To make assumptions of normality more plausible, we model $\log y$, the natural logarithm of the power readings. We used a JAGS [128] (Just Another Gibbs Sampler) module to apply

Markov Chain Monte Carlo (MCMC) simulation to fit our model, detailed below, in a Bayesian framework. MCMC is a type of simulation algorithm that draws samples from a continuous random variable. It is well suited to estimated an expected value of a function and its variance. In our case, these are used to determine the individual energy impact coefficients from all the observation that were made using the Monsoon power monitor. This allows us to isolate impact of individual operators with corresponding uncertainty in the approximation. JAGS is a free and portable program that integrates well within the R language [131]. R is language for statistical computing and graphics. It is provided under open source conditions and offers a vast range of packages that can be used within the language. Our Bayesian regression model, with categorical covariates, is summarised in the JAGS model below:

```
model {
  # setting sampling model
  for (i in 1:n) {
    logy[i]~dnorm(mu + blueState[X[i,1]] + dat*X[i,2] +
                  select*X[i,4] + ed*X[i,5] + pow*X[i,6] +
                  win*X[i,3], tau)
  }

  # setting priors
  mu ~ dnorm(0,0.0001)
  dat ~ dnorm(0,0.0001)
  select ~ dnorm(0,0.0001)
  ed ~ dnorm(0,0.0001)
  pow ~ dnorm(0,0.0001)
  win ~ dnorm(0,0.0001)

  # corner constraint
  for (j in 2:5) {
    blueState[j] ~ dnorm(0,0.001)
  }
  blueState[1] <- 0

  # precision prior (1/variance)
  tau ~ dgamma(1,0.0001)
}
```

The sampling model defines the response variable (i.e. log power consumption) as normally distributed given six categorical covariates. The intercept *mu* represents the OS_{idle} parameter, as the OS is running in all deployment scenarios, followed by the Bluetooth state encoded as a four state categorical covariate, followed by data sampling, select operator, calculation of Euclidian distance, the square root operator, and the application of windowing - all modelled as a normal distribution with mean 0 and standard deviation of 0.0001 forming an uninformative prior. The response variable represents the overall log power consumption within a 1-second interval. The

Parameter	Mean	SD	2.5% Quantile	97.5% Quantile
mu	-0.19743	0.002673	-0.20264	-0.19226
dat	0.65906	0.003790	0.65166	0.66662
select	0.03964	0.003813	0.03217	0.04704
ed	0.03448	0.003787	0.02703	0.04187
pow	0.03357	0.003796	0.02602	0.04082
win	0.02032	0.003820	0.01289	0.02772
blueState[1]	0.00000	0.000000	0.00000	0.00000
blueState[2]	0.33514	0.010083	0.31561	0.35507
blueState[3]	2.53914	0.033054	2.47559	2.60359
blueState[4]	3.81306	0.023526	3.76717	3.86000
blueState[5]	2.69197	0.012111	2.66858	2.71593
tau	13.80491	0.073279	13.66139	13.94856

Table 4.8 Bayesian Regression Model Power Impact coefficient summary.

tau element represents the precision parameter. In the JAGS model, uninformative priors are used for all parameters, and a corner constraint is placed on the Bluetooth state as only one of the states can be active at a time, fixing the impact of the Bluetooth inactive power impact to 0.

The model is initialised with μ and τ parameters as 0 and 1 respectively. We allow the sampler to burn-in for 100 iterations, meaning discarding any samples that were drawn by the model in this phase, and then run it for 1,000,000 iterations with a thinning factor of 1000. The thinning factor discards all but kth sample, in our case every 1000th sample is persisted. This process helps to remove the correlations in between individual samples [72]. Figure 4.6 shows the diagnostic plots for two representative covariates of the fitted model which indicates the sampler converged and mixes well. Derived model parameters representing the power impact coefficients are summarised in Table 4.8.

For example, the posterior mean for data suggests we would expect log power consumption in one second to increase by around 0.65906 if data is being sampled from the accelerometer in that second, with all other things held constant. To draw from the posterior distribution, a physical plan is formatted into four phases as defined by the Power Impact Model in Section 4.1.1. For example to get an estimate for the optimised plan with active operators: DATA, SELECT, ED, and WIN with a window size of 120 seconds, the four phases will have the following durations:

1. blueState1_duration = $window\_size \times 10 - \sum_{i=2}^{4} blueState_i\_duration$
2. blueState2_duration = 0.6
3. blueState3_duration = 1.3
4. blueState4_duration = 11.7

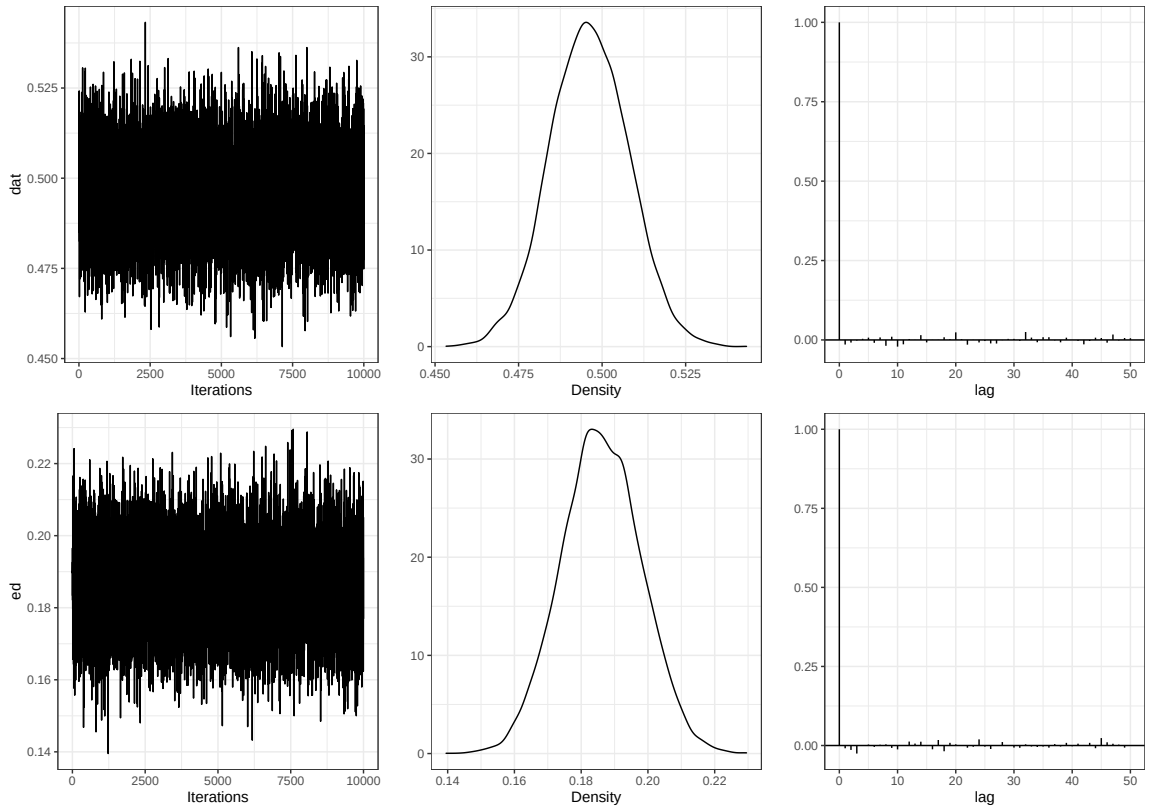


Fig. 4.6 MCMC Diagnostic plots for the fitted model.

Figure 4.7 shows the distribution of predictions generated from the trained Bayesian regression model for the six selected plans with the corresponding 95 % credible intervals delimited by the red vertical bars. This model was trained by running the previously described MCMC chain defined in JAGS. In comparison with the Confidence Intervals, used in the frequentist approach, where the 95 % confidence intervals only represented a range of plausible values, the Bayesian approach gives more interpretable results, representing the interval within which the prediction lies with (posterior) probability 0.95. Moreover the shape of the distribution is revealing. For example, the posterior distribution for plans 'pp04' and 'pp09' is fairly symmetric with the peak – signifying the highest probability for the power cost for a given plan – lying in the middle of the credible intervals. However for other plans, such as, 'pp00', 'pp01', and 'pp02' there is a skewness with the peak shifted more to the left, providing more information for the prediction. This is the crucial practical difference in between the first and the second appraoch to estimate uncertainy of the power energy coefficients that are then used in the *PATH2iot* system to cost individual physical plans to select the best one that will be the most energy efficient. The first approach is computationally simpler, however it doesn't offer the insights as seen from Figure 4.7. A more significant skewness might be revealed if modelled for different device or application.

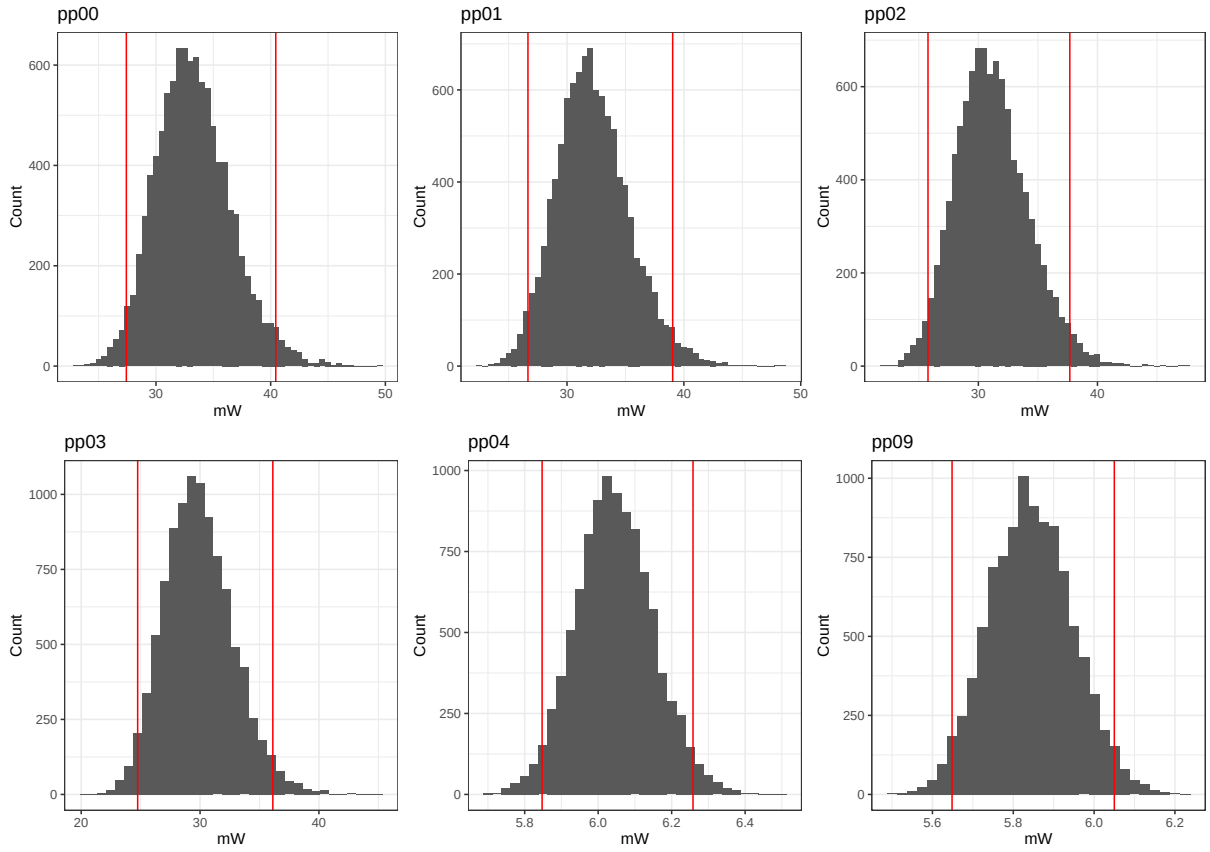


Fig. 4.7 Bayesian probabilities with 95% credible intervals for selected plans.

The output from the power modelling form the energy coefficients that are loaded as part of the resource catalogue into the *PATH2iot* system.

4.3 Evaluation of Healthcare Application

To evaluate the proposed Energy Model, and compare the two different approaches to capture the uncertainty of these battery estimates, we have compared the physical plans that the *PATH2iot* system returns as viable options for deployment them with the actual measurements.

The aim of the healthcare application has been outlined in Section 3.1.1 with operations defined in Section 3.2.1 to process raw accelerometer data in real time to calculate activity levels in the form of step count summaries. The definition of the infrastructure that was received by the system from the Resource Catalogue, as outlined in Section 3.2.2, was that there were three active infrastructure components across which the computation could be split: (i) a smartwatch; (ii) a smartphone; and (iii) a cloud resource. The information about the computational capabilities and the Power Impact coefficients has also been recorded within the Resource Catalogue. The primary objective for the optimiser was that the Energy consumption on the wearable device was

ID	DATA	SELECT	ED	POW	WIN	WIN size (s)	msg_count_per_s
pp00	✓	✓	✓	✓	✓	1	1
pp01	✓	✓	✓	-	✓	1	1
pp02	✓	✓	-	-	✓	1	1
pp03	✓	-	-	-	✓	1	1
pp04	✓	✓	✓	✓	✓	120	0.2
pp09	✓	✓	✓	-	✓	120	0.2

Table 4.9 Evaluated Physical Plans : Computation Placement.

ID	Measured	Frequentist			Baysian		
		Estimate	95% Conf Int	Error	Estimate	95% Cred Int	Error
pp00	29.7365	31.596	(31.09, 32.08)	6.22%	33.348	(27.53, 40.34)	12.15%
pp01	29.8051	31.563	(31.08, 32.04)	5.90%	32.291	(26.70, 39.20)	8.34%
pp02	30.2218	31.539	(31.07, 32.01)	4.36%	31.135	(25.63, 37.71)	3.02%
pp03	30.7207	31.507	(31.04, 31.97)	2.56%	29.954	(24.69, 36.27)	2.49%
pp04	6.0976	6.628	(6.36, 6.89)	8.69%	6.044	(5.85, 6.26)	0.88%
pp09	5.9905	6.604	(6.35, 6.86)	10.24%	5.845	(5.65, 6.05)	2.43%

Table 4.10 Evaluated Physical Plans : Estimated Power Consumption.

set to a minimum of 48 hours to satisfy a simulated doctor's requirement to guarantee 48 hours of battery life so as to ensure patients only need to recharge the wearable every two days.

The optimisation phase initially produced 228 possible plans. However, all but 18 were eliminated due to restrictions on operator placement. Six of these plans have a unique set of operators placed on the wearable device. These are shown in Table 4.9. The process of creation and elimination of the plans is carried out by the optimiser module that places each operator on a node in turn, thus exhaustively searching the space for all possible deployments. However, the implemented rules in physical plan pruning described in 3.5 discard the plans that cannot be deployed.

The last step in the optimisation process applies the energy model that has been described earlier in this chapter to each of the plans to identify their power cost. The plan with the lowest cost is selected, and if it passes all the non-functional requirements it is then passed to the device-specific compilation module that parses the individual operators and transforms them into a device-specific configuration based on the best placement.

Table 4.10 shows the results of applying the energy cost model to the six plans. Both of the methods of estimating uncertainty describe above were utilised: the frequentist with the confidence intervals, and the Bayesian with credible intervals and the error. The estimates produced by the cost model were compared with real measured values using the power monitor.

A detailed description of the experiment summary table is as follows:

- ID – plan identifier;
- DATA, SELECT, ED, POW, WIN – have been defined previously in Section 4.1.2;
- Measured (mW) – the average power consumed by the wearable as measured by the Monsoon Power Monitor;
- Frequentist Estimate (mW) – the average power consumed by the wearable as predicted by the cost model;
- 95% Conf Int – confidence intervals calculated for the power predictions. We have assumed mutual independence of the variables (energy used by the operations) and approximately normal distributions to calculate these intervals, as described above.
- Error (%) – the difference in estimated battery life based on measurements from the Power Monitor and the predictions of the cost model.
- Bayesian Estimate (mW) – the average power consumed by the wearable as predicted by the Bayesian Regression Model;
- 95% Cred Int – 95% credible intervals calculated from the Bayesian Regression Model;
- Error (%) – the difference in estimated battery life based on measurements from the Power Monitor and the predictions of the Bayesian cost model.

If we compare a default scenario where the Pebble watch is programmed to stream all of the raw accelerometer data to the mobile phone so that it can be relayed to the cloud for analysis, the expected battery life is 16.2 hours. In contrast, the best execution plan produced by the optimiser (pp09) has a measured power consumption of 5.99 mW, giving an estimated battery life of 80.3 hours – an improvement of 496%. The improvement of battery life by reducing the amount of data that needs to be transmitted will likely be observed in another use cases with similar properties. These specifically are: the large amount of data that is generated by the sensor that can be transformed locally in order to reduce the overall amount of the data that needs to be transferred. In addition, the use of windows operations on the sensors makes the use of wireless transmission less power demanding. On the other hand, if all of the information must be transmitted to the cloud for processing in real-time, such a use case won't benefit from this approach.

The frequentist approach gives us one point estimate of the expected power consumption of the physical plan – with a maximum error of 10.24%. However, the 95% confidence intervals

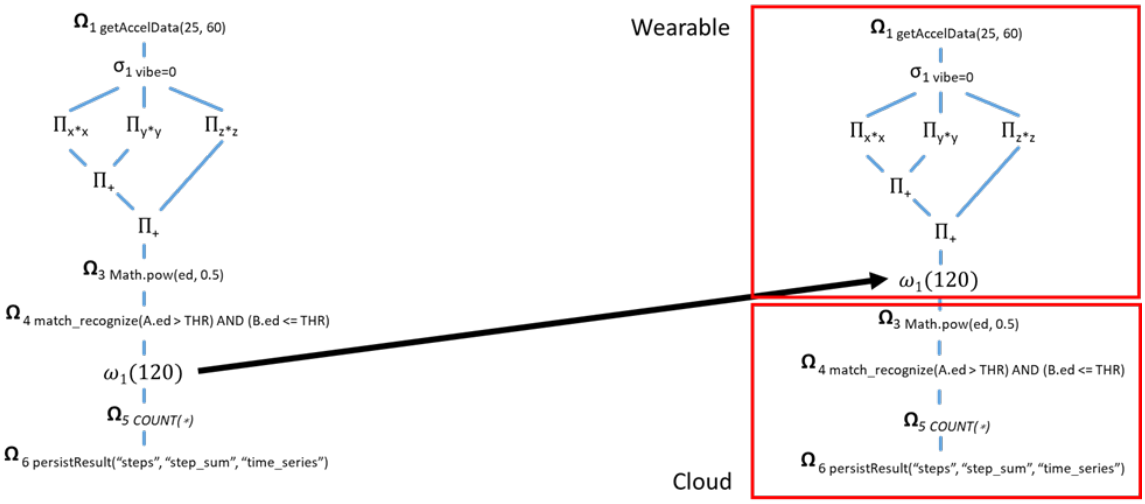


Fig. 4.8 Original DAG of operators (on the left) compared to the optimised plan.

of 70.1 – 75.7 hours of estimated battery life are too narrow to bound the measured power consumption, for the reasons outlined in Section 4.2.1.

In contrast, the Bayesian approach resulted in 95% Credible Intervals that provide an interval of 79.5 – 85.1 hours, so successfully capturing all of the test cases. This suggests that our Bayesian approach holds promise in the design of IoT applications. However, the maximum error using this approach was greater at 12.15%. The Bayesian approach is also significantly more computationally intensive to run an MCMC chain for over three hours to fit the model. However, this only has to be done once for each application.

4.4 Summary

If we compare the optimised plan (pp09) against the default plan we can see why the work of the optimiser has resulted in such a dramatic improvement. Figure 4.8 shows the original plan and the optimised plan.

The two key decisions made by the optimiser are to:

(a) Move the window operator earlier in the plan and place it on the wearable. This packs together multiple messages into one message sent from the wearable to the cloud via the phone. As we have seen from the analysis earlier in the chapter, reducing the number of messages can dramatically reduce the power consumption. However, this process increases the overall latency as the data is being delayed on the smartwatch. The application developer can use the variable window length introduced in Section 3.3.2 to determine the best trade-off between the latency and the energy savings.

(b) Perform the square root on the cloud, not on the wearable, so reducing the total operator energy costs.

Overall these decisions balance the need to minimise the number of messages sent from the wearable and the need to minimise the amount of computation performed on it.

Another advantage of the optimised plan is that the total amount of data transmitted to the cloud is reduced by a factor of three. This might be important in cases where there is a charge per byte for data transmission (e.g. over a mobile phone network).

This chapter has shown that it is possible to design a model to predict energy usage, and that, when used as part of the *PATH2iot* optimiser, this can result both in major improvements in battery life (almost 5x in the running example) and also the required data bandwidth. We have also shown the practical challenges in measuring power for real devices. However, the measurements did enable the creation of an energy cost model that performed well for a real-world use case.

We would also argue that both the frequentist and Bayesian approaches are useful for capturing uncertainty in the estimates. Depending on the use case requirements, the application administrator should consider whether to use the frequentist approach, or adopt the more easily interpretable Bayesian approach to capture the uncertainty. As discussed in this chapter, one of the important factors in making this decision is the computational power required for the Bayesian model, but more importantly the depth of the statistical knowledge. It might be appealing to use a simpler frequentist model with straightforward calculations, however, as seen from our model, this might result in narrower confidence intervals, which might give an impression of higher confidence in the time estimates for the battery-powered IoT device to run. On the other hand, utilising the Bayesian approach provided credible intervals; where confidence in the battery estimates is based on probabilities not a mere interval of values, however, underlying assumptions of operator mutual independence and normality have to be considered. The *PATH2iot* system requires energy estimates for the power impact calculations with the confidence intervals.

CHAPTER 5
BANDWIDTH COST MODEL

In the previous chapter, we explored how the *PATH2iot* system optimises EPL queries when an energy constraint is placed on a real-world healthcare application. This chapter explores how the system can handle another constraint – communications bandwidth – for smart cities, by examining a transport monitoring use case. The transfer of data between processing platforms needs to be considered in the design of many streaming data systems, especially for applications that utilise wireless-connected IoT devices. A bandwidth constraint must be applied when the rate at which data can be transferred between connected devices is limited by the communications protocol, or by environmental conditions, or by imposed duty cycles that restrict the periods when the transmitter is permitted to use the channel.

As well as the communications data rate, our bandwidth model considers legal requirements, fair usage policies, airtime requirements, and spreading factors [26]. These will be considered in this chapter as they play an integral part in the decision-making process during which the *PATHfinder* module selects the best plan to be deployed.

5.1 Smart Cities: the TrainBusters Application

The term 'Smart City' can entail many definitions as discussed in [120]. In this work we perceive the instrumentation of the city a first and crucial step in making the city smart. Embedding sensors to measure environmental factors, or behaviour of population that can generate data from which meaningful insight is gained. Smart City applications are becoming an increasingly important use case for streaming data analytics [70, 116, 86]. To extend and explore the capabilities of the *PATH2iot* system, in particular the generality of its optimiser, we have designed and developed a smart detection application to detect train arrivals at a platform – *TrainBusters*. This use case shows that the utilisation of local processing capabilities enables a service, such as notification of the train arrival time, to be delivered in cases where it would not be possible to offload analytics

to the cloud due to network constraints. We show that network constraints, regulatory alignment and fair usage policies can be integrated within our system to make automatic decisions on computational placement. The application we have built could be valuable to citizens, local council transport planners, and could also expose an API for other use in other applications, such as trip planning.

The quantity of data produced in a smart city environment by ubiquitous sensors varies greatly. For example, to measure heat dissipation within a heavily urbanised area a set of ambient temperature sensors would only need to produce small payload messages at low frequency. In comparison, CCTV monitoring would require much larger bandwidth to send the data to a cloud for storage and analysis [124, 136].

One of the main challenges in designing smart city scenarios is network connectivity. While Wi-Fi might be sufficient to connect all sensors within a well-designed shopping centre, however, it cannot provide connectivity in larger, outside areas, or highways. As described in the introduction, the ongoing developments in wireless data transmission, especially in the high-frequency spectrum such as 4G (and 5G) have allowed users to consume and produce vast amounts of digital content on-the-move. However, in this use case, we explore how *PATH2iot* can enable the use of another recent development in networking technologies – Low Power Wide Area Networks (LPWAN).

The rest of this chapter discusses the train detection use case, and the solution we implemented. We show how the *PATH2iot* system can be used to select the partitioning of a distributed analytics solution so as to minimise the network bandwidth needed, thereby enabling a LPWAN network to be realised. The proposed solution calculates all possible deployment options based on the amount of data transmitted between nodes and offloads a sufficient amount of computation to the edge device. This approach can reduce the amount of data needed to be transported over the network, satisfying the non-functional requirements, in this case the limited bandwidth link.

5.1.1 Objectives

There are multiple possibilities for how to detect when a train arrives at the station. We have considered the following for our experiments with the Newcastle Metro light rail system:

- Requesting the current location from the fleet of trains by the network operator. This was considered because digital information boards on each platform give real-time information on when the next train will arrive. Unfortunately, this information is not currently shared outside of the Metro's information system, for example by exposing a public API endpoint.

When asked, the operators told us that while there is a plan to share this information publicly, there are no time estimates for when this will become available. We therefore had to consider other options to collect the information.

- GPS sensors in each train – placing a GPS sensor that could periodically measure and transmit the current location of the train would allow for real-time tracking of the fleet. This is a typical tracking of vehicles or object of interest in an open environment. However, this would require agreement with the network provider, who would have to fund the cost of installation. However, this is not a good solution for a railway like the Newcastle Metro, as parts of the system, including stations, are underground.
- Proximity sensor – we could place a suitable sensing mechanism directly on the platform. For example, a proximity sensor could be placed to detect the coming train. Placement on, or near, the platform would be crucial, as a proximity sensor requires a direct line of sight and any obstacles would cause detection problems. For example, the sensor could be placed in between the tracks, pointing in the direction of an oncoming train. However, this solution would require the agreement of the operator, and maintenance could be a problem.
- Acoustic sensor – another approach would be to use an audio sensor. A simple microphone recording at a high enough frequency might enable the detection of the train if there is a unique sound signature when it is in the station. Deploying and testing this approach is quite straightforward, as it only requires locating the device somewhere on the train platform where the sound of the train can be picked up.

We have selected the acoustic sensor approach as it does not require installation of equipment inside the Metro train cars, nor does it require special permission to position the device near the edge of the platform, or on the rail tracks. Audio analytics also provides an interesting challenge for *Path2iot*, one that is very different to the previously explored healthcare example.

The audio captured by the microphone must be analysed and the information on train arrivals made available to applications. This means that an entirely self-contained application running on the platform is not enough. This raises the issue of how information can be conveyed from the platform. To achieve this we explored the use of a Low Power Wide Area Network.

5.1.2 Low Power Wide Area Network

Low Power Wide Area Networks offer low energy, long-range communication. It operates in unlicensed spectrum, in contrast to the high-frequency 4G and 5G networks, where companies

	Sigfox	LoRa	NB-IoT
Frequency	Unlicensed	Unlicensed	Unlicensed
Bandwidth (kHz)	0.1	125/250	200
Maximum payload (B)	8	243	1600
Private network	No	Yes	No
Range – urban (km)	10	5	1
Range – rural (km)	40	20	10

Table 5.1 Comparison of LPWAN technologies.

must bid at auctions to gain rights to use the frequency spectra on which they operate. The latter are – like GSM – assigned by governments and differ by region. In Europe these are 863-870 MHz, in the US 902-928 MHz, in China 779-787 MHz and 470-510 MHz, and in Australia 915-928 MHz [99]. It is crucial to purchase the LoRaWAN transmitter devices that operate in the correct frequency spectrum for the region they are dedicated to be used.

There are three dominant LPWAN technologies that are currently competing for large-scale IoT deployments: Sigfox, NB-IoT and LoRa [133]. These offer long-range, energy efficient coverage of up to 40km in rural zones and up to 5km in urban areas. LPWAN technology can be utilised both indoors and outdoors for IoT applications, so long as they only need to transmit small amounts of data. Table 5.1 outlines the key differences between these technologies.

LoRaWAN is the communication protocol that was standardised by the LoRa-Alliance. [1] Sigfox and LoRa operate within the unlicensed frequency spectrum, therefore the usage of airways is free. However, a regulatory framework based on the country of operation applies. For example, in Europe a duty cycle is set to 1 %. The duty cycle is proportion of the time a transmitter actively transmits data compared to the time it is not broadcasting. Also, the maximum power that the transmitter can use for wireless communication is restricted, but usually this is implemented directly within the hardware design. LoRa also allows users to create their own private networks that can be deployed, for example by purchasing a base station and set of LoRa-capable transmitters for sensors. Table 5.1 also outlines the theoretical range that each technology offers and is closely linked to the air time for a single message – the amount of time the transmitter requires to transmit the message – and also the maximum payload for a single packet.

LPWAN therefore provides a long range, low power connectivity solution that is ideal for many IoT use cases that monitor larger geographical areas and do not need high throughput. As a result, wide range of IoT use cases are benefitting from the long range and the low power that

[1] https://lora-alliance.org

this technology offers, including smart waste collection [107], precision agriculture [138], or wind-turbine monitoring [115].

Limiting the amount of data and the frequency of data transmission is a key challenge when using LPWAN technology. Accurate prediction of this is an essential step when designing a system architecture for a real-world deployment. We therefore explored whether *PATH2iot* could be used to automatically meet this challenge, by using a cost model that predicts bandwidth use that will be described in Section 5.3.4. The alternative – hand-crafting a bespoke system – would require a wide range of computational placement decisions to be considered to ensure the bandwidth limits were not crossed. This process would also have to be repeated, and the system redesigned if the system changed in any way, for example, to add information on how busy a platform is, is derived from counting the number of mobile phones that had Bluetooth enabled.

We selected the LoRA version of LPWAN for our experiments as it allows administrators to set up and maintain their own private networks and also allows for bidirectional half-duplex communication.

In the rest of this chapter we first describe how audio analysis can be used to identify train arrivals. We then show that even though a large quantity of data is generated by the audio sensor, LPWAN can be used because *PATH2iot* can make automatic computational placement decisions that ensure compliance with the technology constraints and regulatory alignment.

5.2 Audio Signal Analysis

In order to detect the train arriving at a platform using an acoustic sensor, we used a Raspberry Pi 4 with a microphone and a battery pack. To explore how to extract relevant information from the audio, we recorded and annotated signals from the platform at Monkseaton Metro station – platform 1. Figure 5.1 shows a map with the location of the LPWAN base station and the Metro platform. The distance between them is 531 meters.

We have collected all the data from the same location at the Monkseaton Metro station. The distance from the microphone to the train, once stationary, was approximately 3 meters. Figure 5.2 shows a LoRa base station positioned at a kitchen window sill, and the sensor node, that contains the Raspberry Pi 4, USB microphone, battery packs, the MultiTech mDot LoRaWAN ready transmitter, LoRa MultiTech developer kit MTUDK2-ST-MDOT module and MultiTech Conduit (MTCDT Series).

During this research project we have arranged for a LoRa base station to be deployed at the roof of Urban Sciences Building (Figure 5.3) covering parts of the Newcastle upon Tyne,

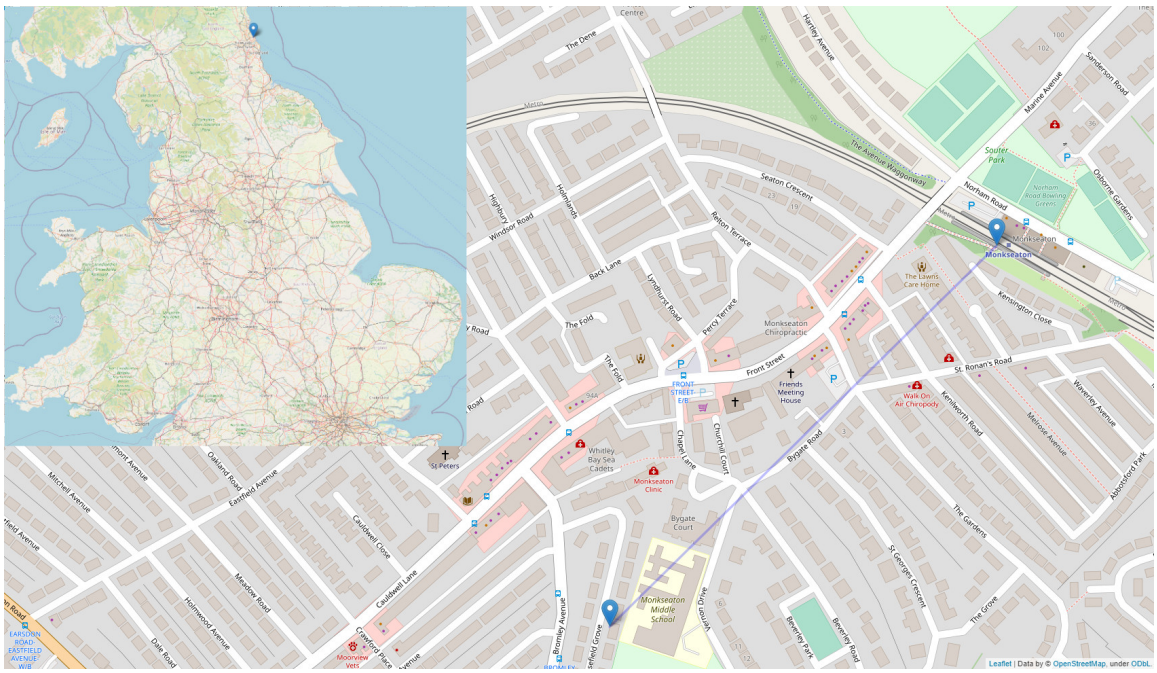


Fig. 5.1 Map showing the distance between the LoRa base station and the metro station with the LoRa mdot transmitter used in the TrainBusters smart city use case.

providing a LoRaWAN signal as a part of The Things Network. The figure illustrates possible range of the installed LoRa base station: 500 m, 1 km, and up to 5 km. However, this network signal doesn't propagate as far in heavily urbanised areas, hence a further signal range tests would be necessary. The map was generated using 'folium' [27] Python package using 'leaflet.js' [28] JavaScript library.

Figure 5.4 presents one sample of the collected acoustic signal of a train arriving and leaving a platform. The following is the composition of a typical set of events associated with the train arrival:

1. 'Safety announcement' – repeated every 15 minutes, it informs passengers that smoking is not allowed anywhere at the Metro station;
2. 'Train Announcement' – an automated message, triggered by the arriving train, notifying passengers of an incoming train and its destination;
3. 'Train Arrival' — approximately 18 seconds of the audio signal is the sound of the approaching train;
4. 'Door Opening Signal' – the first warning sound of the door opening;
5. 'Door Closing Signal' — the second warning sound of the door closing;
6. 'Train Departure' — approximately 20 seconds of the audio signal is the sound of the departing train.

Fig. 5.2 The Monkseaton Metro station with the experimental TrainBusters hardware setup at the platform; a LoRa base station on a kitchen's window sill.

The audio signal analysis could focus on three options to use for the detection of the train at the platform:

- Train Announcement – this announcement is triggered automatically before the train reaches the platform. It also contains both the platform information and the final destination of the train, which could potentially be processed by a form of speech recognition analysis. Some online services already provide real-time speech-to-text transcription, such as Microsoft's Cognitive Services [29] or Speech-to-Text API from Google [30]. However, both of these approaches would require an internet connection with sufficient bandwidth to send all the data over to the cloud for analysis. The ongoing improvement in deep learning models and sizes enable real-time transcription of the audio, such as Kaldi framework [129]. We have not proceeded with this approach.

- Train arrival/departure – a specific humming noise is produced by the electric train as it approaches and departs the platform. However, as can be seen from the spectrogram (a

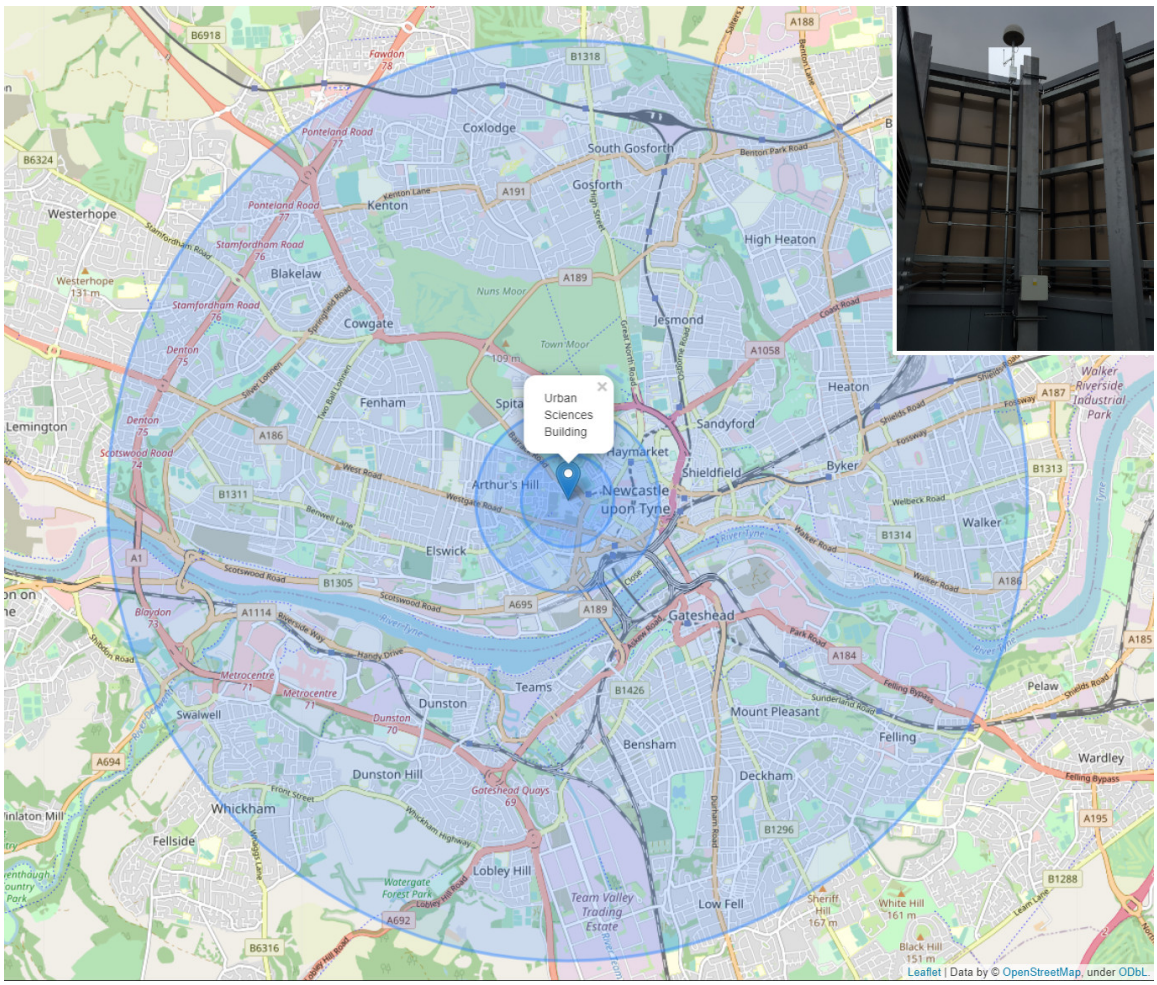


Fig. 5.3 LoRa base station deployed at the roof of Urban Sciences Building with an illustration of signal range.

plot of frequencies over time) in Figure 5.5, this noise does not produce strong signals at stable frequencies that could be easily detected;

- Door opening/closing – there are two distinct sounds that the train makes to alert passengers of door opening and closing. These high-pitched sounds, as we will show from the audio digital signal processing analysis, are clear enough to tell them apart from other noise, such as passengers talking, birds chirping or other background noise.

This work focused on the last option, train detection based on the door opening and door closing sounds. This guarantees that the train stopped to allow passengers to exit and board, as opposed to a train just passing by. Also, we judged that detecting two sounds should allow a simpler approach when compared to perform complex speech recognition.

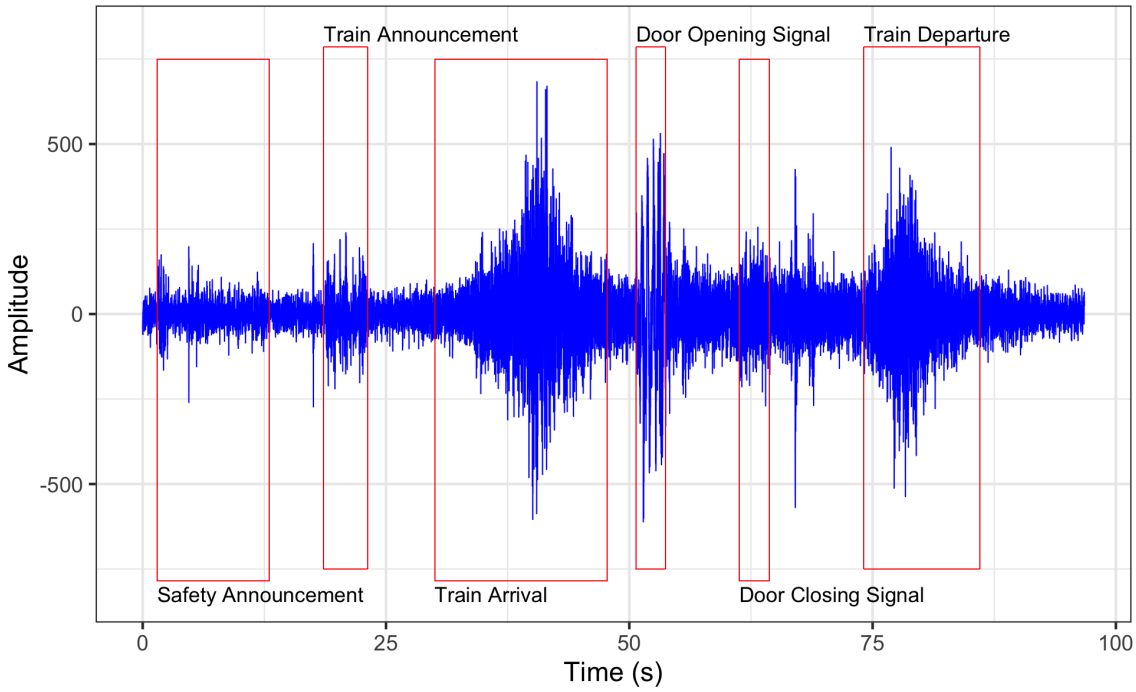


Fig. 5.4 Annotated acoustic signal recording of a train arriving at the Monkseaton Metro station.

5.2.1 Frequency Domain

The acoustic signal was recorded in dual-channel mode, using a 16-bit resolution signal with a sampling frequency of 44100 Hz. For the application purposes, it is sufficient to process the audio data from a single channel, hence only the left channel was used for the analysis.

As a first step in a digital signal processing [142] scenario is to transform the collected signal from time domain to frequency domain. This is done by a Fourier Transform. This converts the original signal from time (or space) domain into the frequency domain [68]. It can also be used to reverse the transformation. We have used the fast Fourier Transform (FFT), an extension to Fourier Transform, that increases the computational speed and is more suitable for computationally constrained devices - such as Raspberry Pi microcontrollers. The FFT is used in many areas of engineering and science, as it provides a unique way to represent signal visually as a sum of sinusoids functions [57, 94].

The transformed signal can be visualised through a spectrogram, as seen on Figure 5.5 for our sample signal with highlighted noise that is present in the lower frequency spectrum and visible distinct frequencies for the door opening (1) and the door closing (2). The signal processing to detect the doors opening and closing was implemented using the R language [31] 'tuneR' [32], 'signal' [33], 'seewave' [143] and 'tidyverse' [154] packages. Also, we have used 'ggplot2' [153] R package for plotting the figures in this chapter and the rest of the thesis.

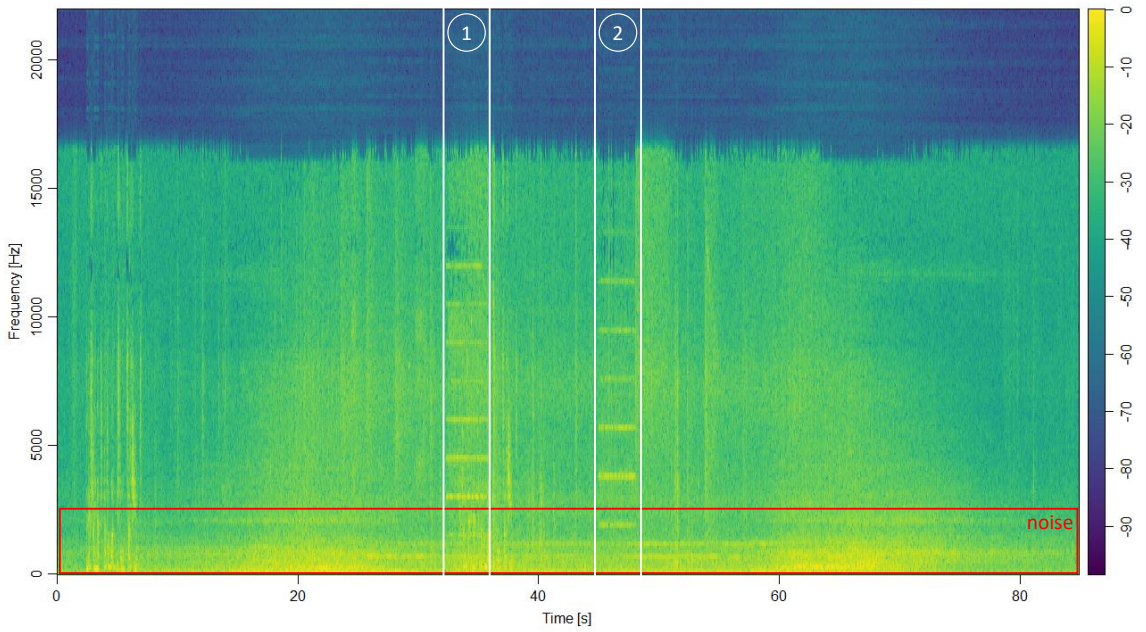


Fig. 5.5 Spectrogram of the acoustic signal of the train arriving at the platform.

The signal was split into windows of 512 samples and the Fast Fourier Transfer calculated for each one. As seen from the spectrogram in Figure 5.2 there is some noise present in lower frequencies. A high-pass Butterworth filter with a cut-off frequency of 2.5 kHz and order 3 was therefore applied to suppress this part of the signal. This significantly improved the detection of the dominant frequencies in the higher range – 2.5 kHz and above.

The result of this step is visualised in Figure 5.6 and reveals the presence of unique, dominant frequencies during the two warning sounds. It also shows that there are no discernible patterns for the other sound signals that we initially considered for detection. This confirms that focusing on the door opening and closing is a sensible, viable option.

The next step was to identify these dominant frequencies so that they could be used for real-time detection. Further exploration confirmed that it is sufficient to check for the three most dominant frequencies within the signal to uniquely confirm the presence of the warning sound – both for the door opening and the door closing. Table 5.2 lists the first seven dominant frequencies in the door opening and closing and also their strength.

To confirm the identified dominant frequencies used to detect the train, we have run the overall analysis on a set of recordings taken from the platform. Figure 5.7 shows the test audio sample with the three selected frequencies for the door opening and three selected frequencies for the door closing warning sound, overlaid on the top and the bottom of the audio signal respectively. The presence of each of the sounds is shown on the figure by red vertical bars. The figure shows that the detection algorithm is correctly picking dominant frequencies for each of these signals. However, selected frequencies are detected as dominant throughout the audio

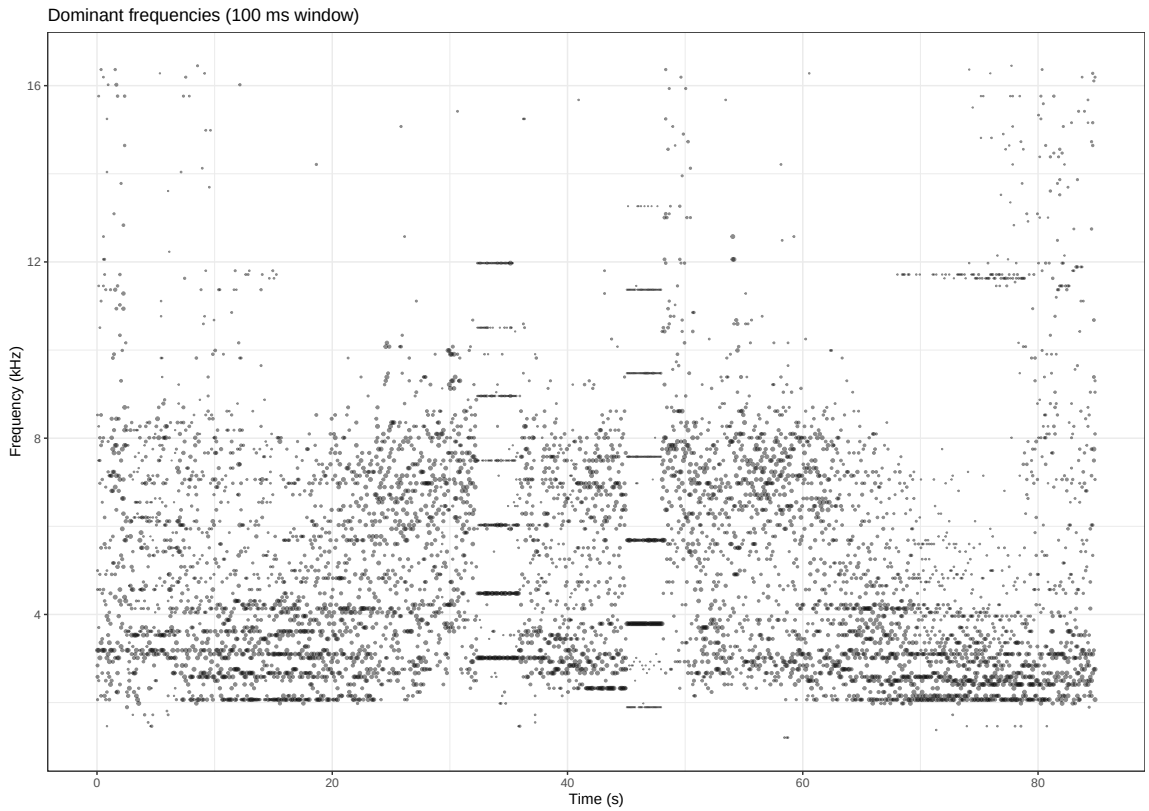


Fig. 5.6 Visualisation of top seven FFT dominant frequencies for the acoustic stream.

recording. This suggests that it is not enough to focus on a single frequency, but multiples must be taken into consideration to lower the rate of false positives. Our experiments show that when a continuous presence of the three dominant frequencies is detected in the signal, this will provide sufficient evidence that a warning sound is present in the signal, separating it from other noise that the microphone might have recorded. This raises an issue for how long the warning sounds are present each time the Metro train arrives at the station. We will explore that in the next section.

As expected, the train can be detected when all three dominant frequencies for the door opening are present, followed by all three dominant frequencies of the door closing. In order to increase the detection accuracy and lower false positive detection events, we also take into account an average duration for the two sounds, as well as the time interval between these two sounds. This was calculated from the annotated train audio data we collected. Figure 5.8 displays boxplot summaries of the three investigated event timings: (i) door opening; (ii) boarding; (iii) door closing. The outliers in the boarding time data can be caused by several unpredictable events, such as:

- passengers with a disability, or older passengers, who need extra time to board safely;

Door Opening		Door Closing	
Frequency (Hz)	Strength	Frequency (Hz)	Strength
3,015	6.000	3,790	6.000
4,479	0.620	2,584	3.207
2,584	0.366	5,685	3.114
6,029	0.315	2,239	1.350
7,494	0.189	1,895	1.163
2,239	0.131	9,475	0.870
10,508	0.125	7,580	0.594

Table 5.2 Dominant frequencies ordered by magnitude for the train warning sounds.

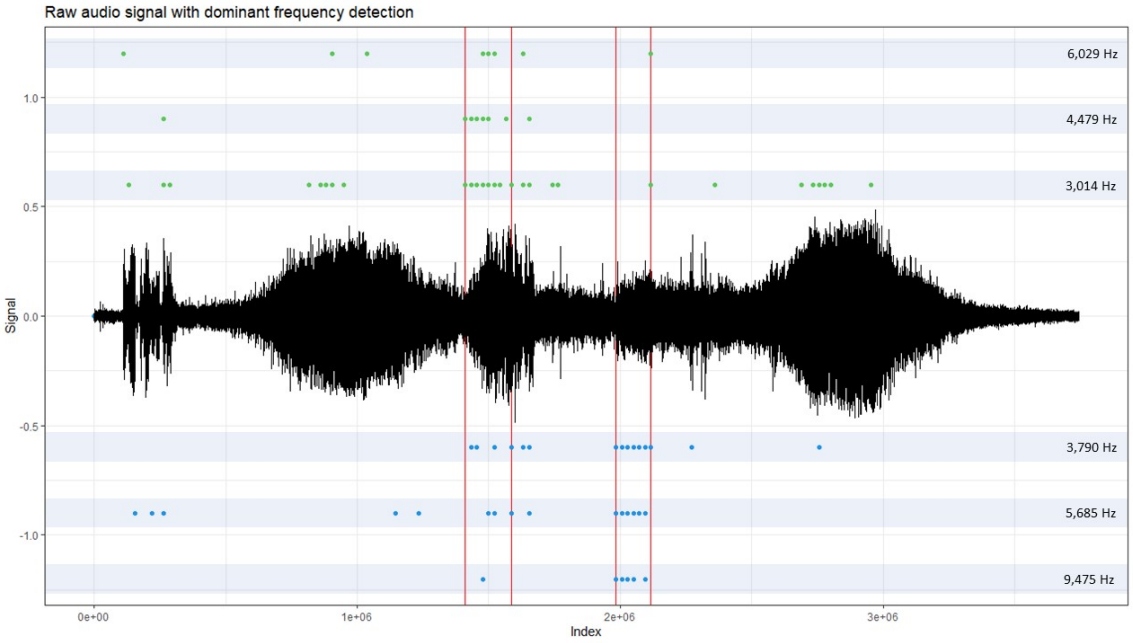


Fig. 5.7 Acoustic signal with door opening and door closing frequencies detection.

- other reasons for the driver to intervene, such as when more than a permitted number of bicycles board a train car;
- waiting for the track “clear” signal that indicates that the train can leave the platform.

As a result of this analysis, we have identified the following characteristics to use to identify a train stopping at a platform:

- the three dominant frequencies for the door opening (3,015 Hz; 4,479 Hz; and 6,029 Hz)
- the three dominant frequencies for the door closing (3,790 Hz; 5,685 Hz; and 9,475 Hz)
- the duration of each warning sound was set to 3 seconds;
- an upper limit for the boarding time of 30 seconds.

These results are encapsulated in the EPL queries defined in the next section.

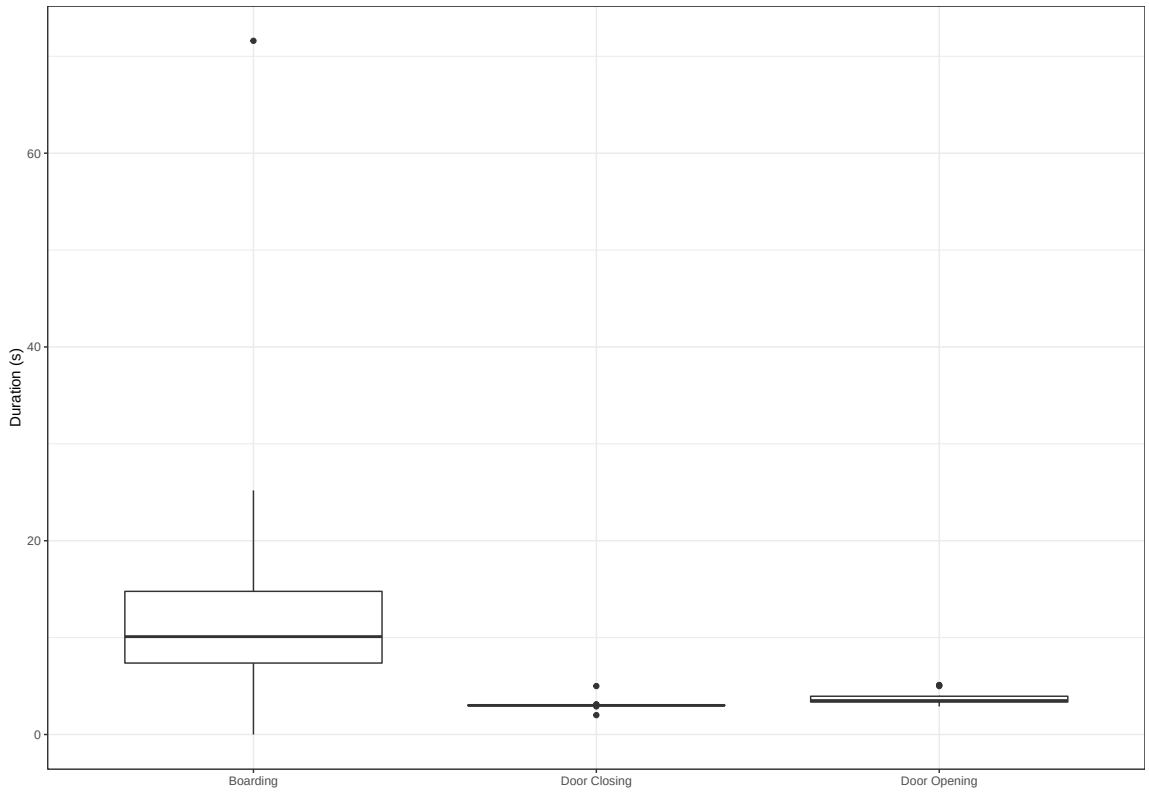


Fig. 5.8 Monitored events - duration.

5.3 Optimisation Process

The *PATH2iot* system requires a definition of the computation in the form of Event Processing Language queries, along with a definition of non-functional requirements and a definition of the available platforms within the infrastructure on which the computation can run. The system then decomposes the EPL queries into a Directed Acyclic Graph (DAG) of streaming operators, optimises the queries, and produces a final plan that satisfies the non-functional requirements. This section steps through each of these stages in detail for the TrainBusters use case.

5.3.1 TrainBusters: Input

This section will outline in detail the three inputs that the *PATH2iot* system requires to operate. These are the set of EPL queries, infrastructure description, and the definition of non-functional requirements that the tool will optimise for.

EPL: TrainBusters

We have translated the digital signal processing analysis of the collected acoustic signal in the previous section into a set of EPL queries for the *PATH2iot* system. These are:

1. **INSERT INTO** SpectralRecord
   SELECT recordAudio(44100, 16, 16384) **FROM** AudioStream

2. **INSERT INTO** SignalOpening(ts, size)
   SELECT current_timestamp, size
   FROM SpectralRecord(frequency **BETWEEN** 2985 **AND** 3045,
   magnitude > 0.45)#**time_batch(0.5 sec)**#size

3. **INSERT INTO** SignalOpening(ts, size)
   SELECT current_timestamp, size
   FROM SpectralRecord(frequency **BETWEEN** 4449 **AND** 4509,
   magnitude > 0.45)#**time_batch(0.5 sec)**#size

4. **INSERT INTO** SignalOpening(ts, size)
   SELECT current_timestamp, size
   FROM SpectralRecord(frequency **BETWEEN** 5999 **AND** 6059,
   magnitude > 0.45)#**time_batch(0.5 sec)**#size

5. **INSERT INTO** TrainSignal(ts, door)
   SELECT current_timestamp, 1
   FROM SignalOpening#**time(3 sec) HAVING** count(size) > 12

6. **INSERT INTO** SignalClosing(ts, size)
   SELECT current_timestamp, size
   FROM SpectralRecord(frequency **BETWEEN** 3765 **AND** 3815,
   magnitude > 0.45)#**time_batch(0.5 sec)**#size

7. **INSERT INTO** SignalClosing(ts, size)
   SELECT current_timestamp, size
   FROM SpectralRecord(frequency **BETWEEN** 5660 **AND** 5710,
   magnitude > 0.45)#**time_batch(0.5 sec)**#size

8. **INSERT INTO** SignalClosing(ts, size)
   SELECT current_timestamp, size
   FROM SpectralRecord(frequency **BETWEEN** 9450 **AND** 9500,
   magnitude > 0.45)#**time_batch(0.5 sec)**#size

9. **INSERT INTO** TrainSignal(ts, door)
   SELECT current_timestamp, 2
   FROM SignalClosing#**time(3 sec) HAVING** count(size) > 12

10. **INSERT INTO** TrainStream(ts)
    SELECT ts2 **FROM** TrainSignal#**time(30 sec)**
    MATCH_RECOGNIZE(MEASURES e1.ts **AS** ts1, e2.ts **AS** ts2
    PATTERN (e1 e2) define e1 **AS** e1.door = 1, e2 **AS** e2.door = 2)

11. **SELECT** sendToMessageBroker(ts, 'Monkseaton-p1')
    FROM TrainStream

The first query relies on a user defined function (UDF) 'recordAudio', that initialises the microphone on the Raspberry Pi, selects the preprogrammed input channel ('AudioStream') sets the sampling frequency (44100Hz) as specified by its first parameter, and specifies the bit resolution for the recording of the audio sample (16) through its second parameter.

This UDF function uses count-based windows (detailed comparison of time-based and count-based windows can be found in Section 2.4.1), that utilise a new processing thread within the Java application in order to minimise the processing impact on the main thread handling the audio recording.

The window of measurements from the microphone are stored as a byte array, encoded in Big Endian format, and are transformed inside the new thread into a double array that contains the raw audio signal as seen in Figure 5.4. Once a full array of received audio events has been transformed, the FFT is calculated for the window. An Apache Commons Java library is used with forward type FFT transform and standard DFT normalisation in the implementation. This returns an array of complex numbers representing spectral components of the input signal. We use standard formulae [142] to calculate the absolute magnitude, Formulae 5.1 and 5.2 for each item in the array.

$$magnitude = \sqrt{(real\_part \times real\_part + imaginary\_part \times imaginary\_part)} \tag{5.1}$$

To determine the frequency for individual magnitudes within the array, the following formula is used:

$$frequency = index \times \frac{sampling\_freq}{fft\_window} \tag{5.2}$$

where *index* corresponds to the position within the array; $sampling\_freq$ is the sampling frequency of the recorded audio signal; and fft_window is the length of the window on which the FFT transform was calculated. The size of this window can be set to any 2^n for this type of calculation to work correctly. In this case it is 512, with each thread handling an array of 8192 audio samples. The final step within this UDF is creating a new stream of events that each

contain the magnitude and frequency tuple that are individually pushed into a 'SpectralRecord' stream as defined by the INSERT INTO part of the first query.

This spectral record can then be used by queries 2–4 and 6–8 to detect the dominant frequencies relating to the doors opening and closing respectively. The EPL statement can define a range with use of 'BETWEEN' keyword, which we use in the queries to allow for a tolerance to the threshold that we have identified to correspond to the door opening and the door closing acousic signal. This range operator performs a filtering operation on the incoming stream and passes only the frequencies with the defined values.

We introduced a tolerance threshold for the dominant frequencies to be detected, so as to improve the detection results. A threshold of 25Hz below and above the dominant frequencies was found to perform well, and so is used in all the detection queries 2–4 and 6–8. All of these queries follow the same template, but each is configured to detect one of the dominant frequencies. They all consist of four parts:

1. Filter out all events whose frequency does not fall between the lower and upper thresholds;
2. Filter out all events whose magnitudes are below the frequency threshold magnitude (which is set to 0.45);
3. Time-based sliding window – all events that pass the filter operations will be collected within a 500 ms window;
4. Count operator – defined by 'size' operator in queries 2–4 and 6–8 is the last operator of these queries. It triggers the insertion of a new event into 'SignalOpening' and 'SignalClosing' streams. The events in the new streams contain the current timestamp and the size of the window, that is the number of times an occurrence of a dominant frequency has been detected within that window.

The fifth query aggregates all events from 'SignalOpening' – the output from queries 2–4 - in a 3-second sliding window with a counting operator triggering an output only if more than 15 events were detected. This threshold was selected as an outcome of the previous digital signal processing analysis because it was found to improve the accuracy. It shows that there was the sustained presence, at sufficiently high magnitude, of the dominant frequencies, and so it is reasonable to generate a door opening event. This detection event is forwarded to the 'TrainSignal' stream with the door signal identifier set to '1', which serves to identify that the door opening signal was detected.

Queries 6–8 are equivalent to queries 2–4, except that the upper and lower frequency thresholds are chosen to detect the door closing signal, and their output events form the 'SignalClosing' stream. Query 9 is identical to signal detection Query 5, except that, when a door signal is detected, a new event is created with its unique identifier set to '2' (identifying that the door close signal was detected).

Query number 10 accepts both the door opening and door closing events from queries 5 and 9 respectively. It performs Match Recognise was used in the accelerometer data processing set of queries in Section 3.2.1. The pattern to be detected here is defined as two consecutive events occurring in 'TrainSignal'. The first must be a signal with identification '1' – door opening, followed by signal '2' – the door closing. Further, these two events must happen within a maximum time interval of 30 seconds, as this was established previously to be the average boarding duration. Once this condition is detected, an event is sent on 'TrainStream' to signify that a train has arrived at the platform.

The last query (11) is triggered every time a new event is sent along TrainStream. It pushes an update into a message broker, allowing all current subscribers to this stream to get instant notification of the train's arrival. The single parameter for the sendToMessageBroker UDF is the timestamp from the incoming 'TrainStream' event. In a system with TrainBusters deployed on more than one station, the message broker must be preconfigured to know which station each stream represents. This would allow any consumers to filter only the events from the stations of interest.

In *PATH2iot*, the set of queries that make up the high-level declarative definition of the computation is saved in a plain text file with each query separated by a line break. The file path is specified within the configuration file that is loaded when the *PATH2iot* system starts. The *PATHfinder* module then verifies the query syntax, checks for any EPL grammar extensions, decomposes the queries, runs physical and logical optimisations, before applying one or more cost models in order to arrive at the best deployment plan, as is described later in this chapter. However to do this it needs further information as input.

Infrastructure Description

After the queries, the second input required for the *PATHfinder* optimisation module is the infrastructure description. This contains all necessary information about available platforms. This allows the module to make the best decision on placing operators on the available infrastructure. For this use case, two platforms are available for the computation to be deployed on (i) a resource constrained IoT device: a Rasberry Pi Model 4 B, and (ii) cloud resources. Another

key requirement for this use case is the definition of network capabilities. Here, the connection between the Raspberry Pi and the cloud is provided by a LoRa network.

Non-functional Requirements

The last input required for the optimiser is the definition of the non-functional requirements that must be met. This input guides the optimisation process to use the cost model(s) to arrive at the best plan. In this use case, the definition of requirements is set to optimise for bandwidth. It instructs the optimiser to select the plan that requires the least bandwidth between the infrastructure platforms, in this case the the Raspberry Pi and the cloud.

5.3.2 Logical Optimisation

Once the input files are loaded and parsed by the *PATH2iot* system, an extended grammar check is carried out. This is to detect use of variable window length which was introduced in Section 3.3.2. The next step within the *PATHfinder* optimisation module is query validation. The tool cycles through all of the input EPL queries and passes them through Esper SODA API [21] that returns an error if unrecognized language features are present, such as expressions from different SQL-like languages, or typos in the query statements. If the query validates correctly, it is automatically decomposed by the same API into individual operators. Each query results in at least one Computation node being created in a Neo4j graph database; if a query contains multiple operators these are chained together in a similar process to chaining individual queries together as described in Chapter 3.

For this use case, after the query decomposition phase, there are 32 operators that are to be placed on the infrastructure. A visual representation of all the operators is shown in Figure 5.9. From the graph database, an internal data structure within *PATHfinder* is created for each logical plan. As the focus of this use case was to satisfy strict bandwidth constraints, the next logical optimisation step - the movement of operators – was disabled. This option that can be set in the configuration file by the application developer will prevent optimiser from moving the operators. Having demonstrated the usefulness of this step in the previous use case, we experimented with moving the window and project operators in this use case; however, this resulted in an extremely large number of plans to be evaluated. The future work along with possible improvements to scale out the *PATHfinder* module.

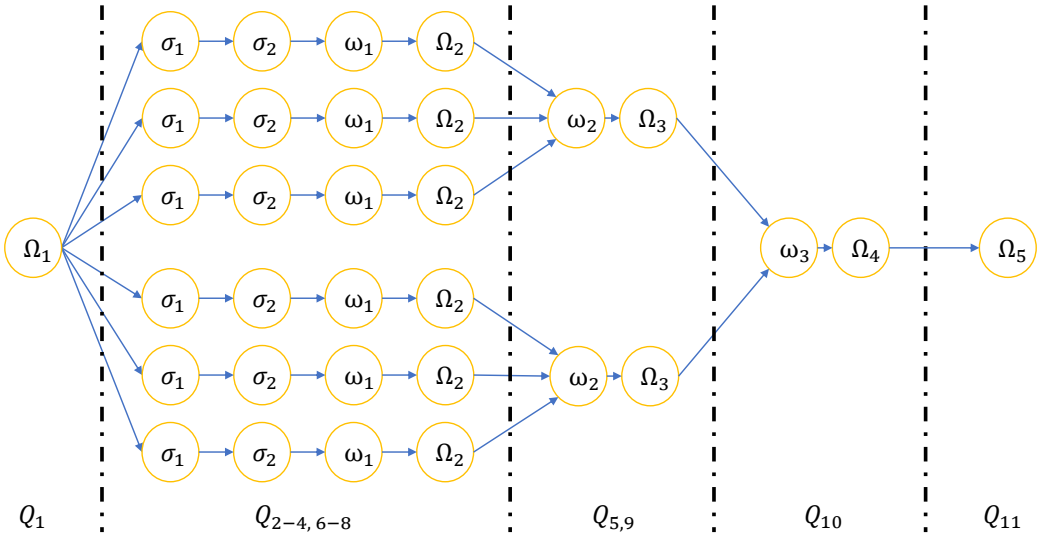


Fig. 5.9 Visual representation of decomposed EPL queries into 32 operators. User Defined Functions are represented as Ω, σ is used for select operators, Π for project operators, and ω for window operators.

5.3.3 Physical Optimisation

After the logical optimisation phase, the *PATHfinder* applies physical optimisations to each logical plan in turn. This consists of trying all possible placements of all operators on the available platforms, evaluating the cost of each option (using the cost model), removing physical plans that are not deployable, and applying safety rules before selecting the best plan. These steps will be outlined in the following section.

For the TrainBusters use case, two platforms are available – the Raspberry Pi and the cloud. With 32 operators to be placed on two platforms, there are 2^{32} possible physical plans. This is significantly higher than in the previous use case and as this is a special case where operators can be placed only on two platforms, a specific algorithm to enumerate all possible placements was implemented. It is based on the binary representation of the index of the physical plan. For example, a physical plan number 101,033 is represented as binary number "0b00000000000000011000101010101001" – which we take as a unique placement of operators that have a fixed order. This means that for this plan, the first 14 operators will be placed on the Raspberry Pi (which has '0' assigned), the next two operators will be in the cloud, the next three on the Raspberry Pi and so forth. This approach guarantees that all of the placement options will be evaluated. This approach improved the speed of the enumeration step within the physical optimisation phase compared to the previously used traversal approach, where the system started with the data source and placed every downstream operator only on the same or a downstream infrastructure platform. However, unlike the previously used approach, this is an exhaustive

search of the space of all possible deployment options, and so includes options in which an operator on the cloud sends events back to the Raspberry Pi. These physical plans are discarded after the enumeration phase is complete.

Once all the plans are enumerated, the physical plan pruning phase discards those plans that cannot be physically deployed. The physical plan pruning is described in Section 3.3.3. During this phase, two safety rules are enforced: (i) the infrastructure platform has to support all of the operators that were placed on it; and (ii) a downstream flow has to be guaranteed – for example, if given infrastructure platforms 'r' and 'c', where 'c' is a downstream platform, any operator 'o2' that accepts events from operator 'o1', which is placed on platform 'r', must be placed on the same platform 'r' or any downstream platform (in this case the only available platform is 'c'). Any plans that would force the stream of data to flow upstream are undeployable and will be discarded in this step. This rules out an operator on the cloud sending events to the Raspberry Pi. Current limitation of this approach is that the bandwidth model evaluates only the data flow in one direction: from the IoT sensor device to cloud. One of the advantages of using LoRaWAN for wireless communication is a bi-directional, half-duplex, data transmission. It is important to calculate any bandwidth required for the data transmissen. For example over the air firmware update of the device would not be possible to carry out over the LoRaWAN connection as the limitation of the bandwidth. This wireless network technology is very well suitable for a small payload sizes, any larger files that need to be transferred, would need to be carried out over another medium. Table 5.4 provides an example of payload size, air time and limit on maximum number of messages that can be sent between the LoRaWAN gateway and a device.

The non-functional requirement for this use case is the bandwidth. For the accurate calculation of bandwidth requirements, the optimiser must estimate the payload size and the data rate of the events that are output from each operator. The selectivity of an operator is used to support this calculation and is defined as the number of outgoing messages per number of received message. The selectivity coefficients for this use case are summarised in Table 5.3. These were calculated from a sample of acoustic data with a duration of 90 minutes, by running the acoustic data through the data analysis pipeline and keeping a record of the average number of events that were sent to an operator and those which were output – this creates a Selectivity ratio for each of the individual operators used within this set of EPL queries.

The last step before the cost model evaluation is to validate all remaining physical plans to ensure that data can successfully flow from one infrastructure platform to another. This step is necessary where more than two platforms are available, as some plans might place operators on the first and the last platform, which might not be directly connected. In this case, the optimiser

Query	Operator	Selectivity
2, 3, 4, 6, 7, 8	BETWEEN	0.001373
2, 3, 4, 6, 7, 8	magnitude	0.017367
2, 3, 4, 6, 7, 8	time_batch	0.161384
2, 3, 4, 6, 7, 8	size	1
5, 9	time	0.037652
5, 9	count > 12	0.012889
10	time	0.051555
10	match recognize	0.001445
11	sendToMessageBroker	1

Table 5.3 Selectivity ratio for operators used within the EPLs.

automatically injects a 'sxfer' transfer operator on an intermediate platform that can relay the data. This safety check is carried out, but is unnecessary for this use case as this scenario has only two, directly connected platforms.

5.3.4 Bandwidth Cost Model

The bandwidth requirements between the platforms on which the operators are to be placed have to be carefully evaluated before any deployment. This is especially important when a significant constraint on the bandwidth is present, such as in the TrainBusters use case where network connectivity between platforms is provided only by the LoRaWAN network.

The Bandwidth cost model implemented for this use case is defined by three important factors: (i) the payload size and transmission frequency; (ii) regulatory compliance; and (iii) The Things Network Fair Access Policy[2]. The first is the one that is used by *PATHfinder* in exploring the space of possible deployment options, as the second and the third factors are directly determined by the first, as will be demonstrated in this section.

As previously outlined in the comparison between the LPWAN technologies, LoRaWAN offers a maximum payload size of 243 B (including 13 B for the message header). This is the first, and the most straightforward, filtering criterion while selecting the best deployment plan. If the size of payload that needs to be transmitted between the platforms exceeds this limit, the physical plan is discarded. For the plans that do not exceed this upper payload limit, there is a need to calculate the airtime that the LoRaWAN mDot module on the Raspberry Pi will need to transmit the message. Airtime, is the amount of time the transmitter is actively broadcasting. This is important for regulatory and fair access policy compliance. Equations 5.3, 5.4 and 5.5 are used to calculate the airtime required for the data transmission over LoRaWAN as described in

[2]https://www.thethingsnetwork.org/docs/lorawan/duty-cycle.html

LoRa modem Designer's Guide AN1200.13 [34] and were implemented within the *PATHfinder* bandwidth cost model.

$$T_{sym} = \frac{2^{SF}}{BW} \tag{5.3}$$

where BW is a frequency bandwidth the transmitter operates in, and SF is a spreading factor [26]. The spreading factor sets the way the transmitter modulates the signal to improve the range, or energy consumption. The higher the spreading factor, the longer it takes for a single data packet to be transmitted, hence, more energy is used for the data transmission. However, this improves the chance for a receiver to pick the signal as it improves the transmission range. The lower the spreading factor, the less time and energy it takes to transmit a data packet, however, if a receiver is further away, or is in dense urban area, the higher the chance that the wireless signal will not be picked up [83]. T_{sym} is a duration to transmit a single data packet.

$$T_{preamble} = (n_{preamble} + 4.25) \times T_{sym} \tag{5.4}$$

where $n_{preamble}$ is the duration that it takes to transmit the preamble, the number of programmed preamble symbols that contain a modem configuration, based on the length of the preamble which is 8 symbols;

$$numberPayloadSymbols = 8 + max(ceil(\frac{8PL - 4SF + 28 + 16 - 20H}{4(SF - 2DE)}(CR+4)), 0) \times T_{sym} \tag{5.5}$$

Number of payload symbols *numberPayloadSymbols*:

- PL - number of payload bytes;
- SF - the spreading factor;
- H - set to 0 when the header is enabled; set to 1 when header is not present;
- DE – set to 0 when low data rate optimization is not enabled; set to 1 when it is enabled;
- CR – is error correction coding;

$$T_{payload} = numberPayloadSymbols \times T_{sym} \tag{5.6}$$

Once we know the number of payload symbols from Formula 5.5, we can calulate how long it will take for the payload data to transmit using Formula 5.6, and get the total time for the whole message to be sent by adding the time it takes for the payload (including the packet header) and the preamble to transmit by adding the two together.

$$T_{packet} = T_{preamble} + T_{payload} \tag{5.7}$$

Door Opening		Door Closing	
Payload (B)	SF	Airtime (ms)	Max msg/hour
1	7	46.34	27.0
1	9	164.86	7.6
1	12	1155.07	1.1
25	7	82.18	15.2
25	9	267.26	4.7
25	12	1974.27	0.6
50	7	118.02	10.6
50	9	390.14	3.2
50	12	2793.47	0.4

Table 5.4 Comparison of Airtime under different LoRaWAN configurations.

Table 5.4 provides an overview of nine different scenarios with varied payload size and spreading factor to show the significance of the two factors. It can be observed that doubling the payload size from 25 to 50 bytes increases the airtime by 30-40 %, however increasing the spreading factor from the minimum to the maximum increases the airtime by orders of magnitude. This has a direct impact on how many messages can be transmitted in a period of time.

European regulation in Section 7.2.3 of the ETSI EN300.220 standard sets a duty cycle of 1% – this is the proportion of time that a transmitter can transmit for on the same channel, and 99% of the time it has to be passive to allow other transmitters to use the frequency spectrum. This means that any wireless device using unlicensed frequency spectrum 863 – 870 MHz can only transmit for 1% of the time, followed by 99% of the time not transmitting. For example, if the air time is calculated to be 250 ms, after a successful data transmission, the transmitter must stay inactivate again before $99 * airtimeused = 24750ms$. This is to allow a large number of devices to share the unlicensed spectrum.

Another restriction is placed on devices using LoRaWAN technology – a fair access policy. This policy requires the devices to actively use air time for data transmission for only up to 30 seconds per day. Hence, if we go back to the first example in Table 5.4, as a single transmission uses 46.34 ms, to comply with a fair access policy, the device can transmit a maximum of $30,000ms/46.34ms = 647msg/day$ or 27 msg/hour. Now we return to the TrainBusters use case and describe how the Bandwidth cost model applies to the explored physical plans.

From the initial logical plan, $2^{32} = 4,294,967,296$ physical plans were considered (as there were two infrastructure platforms and 32 operators). During the pruning phase, as outlined in the previous physical optimisation section, 4,294,952,300 plans were discarded, leaving only 14,996 plans. The final plan is selected based on the payload size and transmission frequency, and compliance with the regulatory and the fair access policy. Table 5.5 outlines some of the

Plan	Raspberry Pi	Clouds	#events/trigger	msg (B)	msg/hour
pp00	Ω_1	31 ops	8,192.0000	131,072	9,690
pp01	Ω_1, σ_1	25 ops	11.2476	188	9,690
pp02	$\Omega_1, \sigma_1, \sigma_2$	19 ops	0.1953	11	9,690
pp03	$\Omega_1, \sigma_1, \sigma_2, \omega_1$	13 ops	0.0315	8.5	7200
pp04	$\Omega_1, \sigma_1, \sigma_2, \omega_1, \Omega_2$	7 ops	0.0315	8.5	7200
pp05	$\Omega_1, \sigma_1, \sigma_2, \omega_1, \Omega_2, \omega_2$	5 ops	0.0012	8.1	365
pp06	$\Omega_1, \sigma_1, \sigma_2, \omega_1, \Omega_2, \omega_2, \Omega_3$	3 ops	<0.0001	8	125
pp07	$\Omega_1, \sigma_1, \sigma_2, \omega_1, \Omega_2, \omega_2, \Omega_3, \omega_3$	2 ops	<0.0001	8	19
pp08	$\Omega_1, \sigma_1, \sigma_2, \omega_1, \Omega_2, \omega_2, \Omega_3, \omega_3, \Omega_4$	1 op	<0.0001	8	14

Table 5.5 Dominant frequencies ordered by the magnitude for the train warning sounds.

plans that were considered, along with the spread of the operators between the two platforms, the number of expected events, the payload size of each message and the transmission frequency.

In this scenario, the last plan in Table 5.5, where 31 out of 32 operators are placed locally on the Raspberry Pi, requires the least bandwidth to transmit the detection outcome over the LoRaWAN connection to the cloud and is therefore selected as the best deployment. Under this plan, there are an estimated 14 events every hour, which satisfies the presented use case, as the trains are arriving at the station every 8–15 minutes. Currently the bandwidth model selects the best plan, i.e. the plan that requires the least frequent data transmission with lowest payload, however, other plans could be successfully deployed. These might be considered when additional requirements needs to be satisfied, such as restrictions on energy usage. This and other possible extensions to this research work are discussed in the Conclusion, Chapter 6.

While only a timestamp (8 bytes) is transmitted for each train arrival event, the receiver could identify the source from the sender's MAC address, which is included in the message header.

5.4 Summary

In this chapter, we have examined how to make automatic operator placement decisions for two available infrastructure platforms with constrained bandwidth. We presented a real-world use case – TrainBusters – with the aim of detecting the arrival of a train on a Metro platform, and transmitting this information to interested parties, such as the council, who may wish to monitor the efficiency of an operator, or to passengers to help them plan their journeys. First, we discussed different ways of tackling this use case before showing that it is possible to accurately detect the train arrival on a platform using acoustic signal analysis. We then discussed the use of digital signal processing to isolate dominant frequencies for two distinct sound signals. Finally, we have translated the analysis into a set of EPL queries that are processed by the *PATH2iot*

system. This set of queries is then passed through physical optimisation, with a bandwidth cost model used to select the best deployment plan automatically from the total of 2^{32} possible plans. In future work, the system could be extended in the following ways:

- Second platform monitoring. The majority of the platform layouts on the Metro network have two platforms that are opposite one another. Hence, the signals used to detect train arrivals can be heard from the other platform and are therefore detected by the proposed IoT system. A possible solution to this problem is to have two interconnected audio sensors, continuously and cooperatively detecting trains at the station, one on each platform. A comparison of the signal magnitude for the extracted frequencies could identify which platform the train arrived at.

- Additional door opening and closing sounds – a small portion of the Metro fleet use different sounds for the door opening and closing events. Further analysis could be carried out to isolate and include additional frequencies to capture these trains.

- Full coverage – it is not sufficient to have a monitoring system at a single station. For further coverage, detection devices could be deployed at multiple Metro stations. However, not all of the stations have to be monitored, as the train movements are quite predictable and an acceptable level of service could be achieved by monitoring a fraction of stations, spread across the Metro network.

CHAPTER 6
CONCLUSION

This chapter reflects on how the research goals were achieved, identifies limitations, and outlines possible future work.

6.1 Research Overview

This research addressed the question set out in the Introduction 1: "This project investigates whether a high-level, declarative description of computation on streaming data can be used to automatically generate a run time execution plan, which meets non-functional requirements." Chapter 2 provides a background information of the distributed stream processing systems, edge computing and relevant research groups in the area of computation offloading.

To achieve this we have designed, implemented and evaluated the *PATH2iot* platform that introduces a novel approach to distributed data analytics for Internet of Things applications. We started with an introduction and review of literature relevant to this work, followed by technical chapters outlining the designed system and the two use cases that we have explored in detail.

Chapter 3 presents the architecture of the proposed system, and shows how it can automatically partition a computational graph, generate and deploy the software components for stream-processing analytics over heterogeneous devices – sensors, field gateways, and clouds – in order to meet a set of non-functional requirements. A high-level declarative description of the computation in the form of Event Processing Language queries written by a user is decomposed into a directed acyclic graph of operators, followed by logical and physical optimisation techniques that use cost models to arrive at the best plan satisfying the non-functional requirements. The plan is then automatically compiled into a device-specific configuration that is deployed onto the target infrastructure, without requiring any knowledge from the user of how to program the device.

Chapter 4 introduces an Energy cost model and describes its application to a real-world healthcare use case. A set of EPL queries is decomposed into possible plans that are costed by the energy model. The best deployment plan is automatically deployed by the *PATHdeployer* module on the real hardware: a Pebble Steel smartwatch, a mobile phone and a cloud. The optimised plan improves the battery life of the wearable device by 453% compared to other placement options. This part of the work used an extensive set of experiments to measure the power consumed by specific deployments, so as to provide the ground truth data needed by the cost model, and also so that the accuracy of the model could be determined. For critical applications, such as those that might have implications for the health of an individual, it is important to be conservative in estimating the battery life of the device for a deployment option. This chapter also presented an alternative approach to capture the cost by using a Bayesian model. We show that, for this use case, this is more accurate than the conventional, frequentist approach.

Chapter 5 introduced a second use case focused on the smart cities scenario. A set of 32 EPL queries was designed to analyse an acoustic signal to detect the arrival of a train at a Metro platform. A bandwidth constraint is placed on the system, as the only connection available to the embedded device is a LoRaWAN network. A bandwidth cost model ensures that only a plan that satisfies the strict constraints on payload, message rate, regulatory and fair policy usage is selected as a deployment option. Also, this chapter presented detailed calculations of the required air time for data transmission, a discussion on how payload size and other LoRaWAN specific factors affect the delivery of this real-world deployment that are crucial to understanding, and how these were fully automated within the optimiser.

6.2 Limitations

Whilst the work delivered the contributions enumerated in the Introduction, there are still some limitations that need to be considered when evaluating the suitability of this approach. These are in the use of User Defined Functions, open software licensing, and the multi-site deployment challenge.

6.2.1 UDF

The unique perspective explored in this thesis, and the resulting benefits, are predominantly tied to the use of a high-level declarative language that enables the system to automate the placement of operators on IoT platforms and clouds. In the healthcare use case, the EPL queries are used to transform an accelerometer data stream into a step count activity summary. However, two

parts of the computation cannot be expressed within the declarative language: (1) data sampling that happens on the watch and (2) persisting the data within a database. User-Defined Functions (UDFs) have to be used for these. As there are only two UDFs in the DAG of 8 operators, and the functionality of each of these is tied to a specific platform, we do not consider this to be a serious hindrance to the optimisation process. Also, it is the first and the last query in the operator pipeline that is pinned to specific platforms, hence all of the operators in between are free to be placed on any platform by the optimiser as described in Section 3.3.3.

In the second use case, train detection requires 32 operators, and two of them are again UDFs. However, the first user-defined function does a large amount of computation as it samples the acoustic signal, performs a Fourier transformation, and calculates the magnitude and frequency from the signal. These operations could not be expressed in the EPL language, as these calculations operate with complex numbers that are not supported in EPL. The second UDF sends the final train detection event to the message broker for any connected subscribers. Once again, Event Processing Language does not allow us to express this as a native query and it must be encapsulated within a UDF.

An attraction of Esper is that operations such as these, that cannot be implemented in EPL, can be programmed as a UDF in Java or a .NET language. However, the use of UDFs imposes significant limitations on the optimiser. Firstly the implementation may not be portable across all platforms, therefore restricting optimisation placement options. Secondly, the UDF is a black box whose behaviour is opaque to the optimiser. Thirdly, multiple data transformations may be encapsulated in a single UDF, reducing the ability to decompose the query into its constituent parts before optimisation.

Overall, these limitations decrease the potential for data reduction, energy-saving etc. through the automatic exploration of the space of deployment options by the *PATH2iot* system.

6.2.2 Multi-site Deployment

We have demonstrated our approach through two real-world use cases, on up to three types of platform: sensor, field gateway and cloud. However, in many IoT use cases, there might be a need to place the same queries on a multitude of homogeneous IoT devices. For example, to extend the functionality of TrainBusters, the smart city use case explored in Chapter 5, two Raspberry Pi boards would be placed, one at each platform at the station. Both of these programmable boards would have to analyse and pre-process the acoustic signal before merging the results in order to improve the accuracy of train arrival detection algorithm. An extension to the system

would enable the system administrator to express this without the need to duplicate queries that are run on multiple devices.

6.2.3 Licensing

Esper, the Complex Event Processing Engine, was used directly within the *PATH2iot* system and the D2Esper stream processor. It is used for query validation and decomposition (utilising the SODA API [21]), and also for the processing of all event streams that reach a cloud platform. Esper is an open-source library [1] that is licensed under GNU General Public License version 2 (GPLv2). This licence requires all of the copies or modified versions of the software to be made publicly available and all of the resulting software must offer the same freedoms. This is an important legal requirement that might affect commercial use of the *PATH2iot* software. The software we have developed for this work has an GPLv2 open-source licence, and is available at https://github.com/PetoMichalak/phd-PATH2iot and https://github.com/PetoMichalak/phd-d2esper.

6.3 Future Work

In this section, we will outline further research work that could directly extend the capabilities of the *PATH2iot* system, including platform monitoring, dynamic re-optimisation and extensions to the introduced use cases.

6.3.1 *PATHmonitor*: IoT monitor

Cloud infrastructure provides a relatively mature, stable and reliable platform for compute, network, and storage. However, fault detection, recovery, and prevention are still active research fields for IoT. The real-time monitoring of services while they are in use plays a crucial role in being able to respond, ideally proactively [58], to any behaviours that would affect the delivery of an IoT-based application. These detrimental situations can include a congested network node, an overload of any of the compute nodes, or a hard disk failure. Problems such as these can also occur for devices in IoT environments, though they can also have additional issues: many rely on battery power that might not be replenished in time to maintain a service, and they are often exposed to harsh conditions, such as bad weather, physical damage due to wear and tear, and poor maintenance when compared to the protected environment in which cloud servers are

[1] https://github.com/espertechinc/esper

kept. Hence, a dedicated monitoring system for all connected devices within IoT environments is unavoidable for real-world use cases.

The *PATH2iot* system architecture 3 included PATHmonitor. This is a (currently unimplemented) component that could capture run-time information on the behaviour of the system and re-optimise the computation if it detects that the non-functional requirements are not being met. Depending on the source of a data stream there are different options for directly or indirectly inferring the state of each system running stream processing tasks:

- Heartbeat – typically, heartbeat messages are used in distributed systems, where servers issue regular messages to each other to ensure all are online and there is no network disruption. This is especially important for data consistency [84] and consensus algorithms, such as Raft [125]. This approach would however require additional messages to be sent from the IoT devices, that would increase energy and bandwidth demands. Alternatively, existing networking activity could be logged by the monitoring system to provide some information about the system state. For example, in the healthcare use case, the preinstalled agent on the Pebble Steel smartwatch pulls a REST API endpoint for the initial configuration at regular intervals, set to 30 seconds. These messages could be used as a signal that the device is still active. Furthermore, once the configuration is received, there is regular messaging from the watch, in the best deployment scenario it occurs every 120 seconds. This could serve as another input signal to the monitoring module that could trigger an alert if the expected message did not arrive within a specified tolerance interval;

- Apache ZooKeeper – the *d2Esper* stream processor that was built for the end-to-end demonstration of the system, uses ZooKeeper [84] for configuration coordination. The implementation uses ZooKeeper watcher functionality, that can trigger an automatic alert when a node disconnects from it. At that point, the ephemeral record – a type of record that is only visible to other ZooKeeper nodes, if a node that created it is still connected – is removed from a nested directory structure of resources which triggers a watcher event and can be acted upon as it happens;

- Docker containers provide a range of run-time statistics for each container, such as CPU, RAM usage, disk and network access, that can be queried through APIs to ensure the services are active;

- Message Brokers enable indirect monitoring of IoT devices and cloud resources by monitoring message arrival rates. There would be a need for an additional layer that would contain information about the expected arrival rates for individual data streams. This could

also be learned during a 'training' period, where the system would be carefully observed. In 'live' data processing, anomaly detection could be running alongside the monitoring system to raise an alarm when the data stream arrival rate deviates significantly from the previously learned parameters;

- UDF information capture – it is possible to gather information in real-time such as in [62]. The additional information this gives could also be used to assist in the re-optimisation of a stream processing application.

6.3.2 Dynamic Adaptation

Once a monitoring system is in place, a further enhancement to the system would be to dynamically adapt to changes in the infrastructure. One of the triggers for adaptation might be the battery level of a resource-constrained device. In this case, the adaptation system could trigger the automatic re-optimisation process, which might result in: lowering the sampling frequency or increasing the batch size to conserve the energy; pausing the data collection altogether; stopping sending data to the cloud; or simply notifying the user or an administrator about the situation, and requesting a battery recharge or replacement.

Another trigger for adaptation might be when sensor devices becomes unavailable, or during network outages. The system could automatically notify the network administrator, and also notify an end-user of possible disruption – this would require the monitoring system to be not only present in the cloud, but also a monitoring module would have to be located on the device itself in order to trigger an alert for example to the mobile phone when connection is not possible for an extended period of time and might result in undesirable consequences. In the case of glucose monitoring, the user would have to keep checking their blood glucose levels, and not rely on a cloud notification to be sent through during the outage. Ideally, the prediction algorithm would be placed directly on the mobile phone, so the user could enjoy the continuity of the service. This capability to dynamically respond to changes in the IoT environment is left as future work but is seen as natural next step.

6.3.3 Additional Non-functional Requirements

The design of the *PATH2iot* system is modular, and allows additional non-functional requirements and cost models to be added. For example, performance and security cost models could extend the optimisation process. This would require additional information provided to the system. In the case of estimating the processing time, the timing of each operator would have to be provided

in order to evaluate individual deployment options to check whether they satisfy the constraint. In the case of security, the optimiser could be extended to discard the plans utilising untrusted network connections or processing data on devices that have not been updated with the latest security updates. The ability to add other cost models to *PATH2iot* has been validated by the fact that another research group has already been able to add a performance cost model [88].

The *PATH2iot* system can be extended by additional non-functional requirements, however, there is a need for the requirements to be defined quantitatively, and also a cost model has to be developed for each new requirement. The two cost models that were described in this thesis, and an additional cost model [88] developed by another research group, can serve as a starting point. Both a domain expertise and a software development skills will be required.

6.3.4 Searching for the Optimal Deployment Option

Another challenge for the optimiser is when there is a large operator placement space. As in the bandwidth use case explored in Chapter 5, the system works with 32 operators and can find the best deployemnt plan. The current near exhaustive search strategy will make this time consuming (or impractical) for large search spaces. However, this could be addressed through scalability: the design of the search is highly scalable as independent plans can be evaluated in parallel. Adding a multi-threaded capability to the module would therefore significantly increase the speed of the optimisation process. Furthermore, the modular design of the system could be extended by adding self-contained optimisation sub-modules that would work in parallel, for example as a set of docker containers – each taking a portion of the plans to evaluate, speeding up the evaluation of the physical plans using available cloud resources. As the individual plans are independant, they could be distributed among copies of the *PATHfinder* module for calculating the cost. There would be additional software development effort needed as to make sure that none of the plans are missed, or to support any node failures to make sure all of the plans are costed.

Finally, other optimisation strategies could be used to narrow the search space, for example a branch and bound approach could be used, or a binary decision diagrams approach [102].

6.3.5 Multitenancy

In the world where access to real-time insights is becoming more common, there is an expectation that the same real-time data will be available to a multitude of applications and users. This is a straightforward task when all of the users are posing the same questions, e.g. what is the current footfall in Newcastle High Street. However, the situation can get more complicated in a scenario

where a single set of sensors can serve multiple applications and users through differing sets of queries, there exist a few options on how to tackle this challenge:

- Stream all of the data to the cloud; in a typical publish/subscribe system, this would lead to multiple consumers processing the same stream of data. However, this would mitigate any possibility for local processing at the edge, as all of the data has to be delivered to a message broker to serve all connected consumers.
- Push all sets of queries from all applications as close to the sensors as possible. This might lower the bandwidth requirement by pre-processing the data locally as we have shown in our smart cities use case. However, it might also increase the computational requirement, as the resource-constrained sensor or edge device would have to manage and run the operators for each client separately.
- Merge the directed graph of operators for all of the application users' queries before pushing it to the resource-constrained devices. In this way, if there is some overlap in the queries from differing users, the optimisation process would merge these, so the same operation would have to be executed only once – saving both the bandwidth and the computational resources, and also lowering the energy impact.

SenShare [100] allows sharing the infrastructure by multiple applications and might provide useful insights in building the multitenancy functionality.

6.3.6 Exploiting Local Storage at the Edge

Local Storage – in addition to the working RAM memory – is offered on many resource-constrained devices. This is not utilised at all in the *PATH2iot* system. However, it could offer additional power savings. For example, if we could extend the EPL grammar further, to allow time-based windows to persist events outside of RAM on the Pebble smartwatch, we could increase the window size, and so lower the energy transmission cost by transmitting larger batches of events in each message from the watch to the phone.

6.3.7 Historical Data

Historical data is currently not included in any analysis within the *PATH2iot* system. However, a time-series forecasting model might need historical data to retrain itself, alongside the latest measurements. At the moment, this would have to be done either using longer time-based

windows, which might not be an optimal solution if the required data is longitudinal; alternatively, a UDF could query a database to obtain historical data. Both are suboptimal solutions.

6.3.8 On Demand Sampling

Smart "On Demand" Sampling could further extend the energy of a battery-powered IoT device by reducing the number of messages transmitted between nodes. For example, in the step count algorithm introduced in Chapter 4, there is no need to send activity summaries to the mobile phone from the Pebble Steel smartwatch if the previous data shows that the user is inactive. For example, a simple rule would be to not send a message unless the number of steps within a time interval reaches a low threshold.

6.4 Closing Remarks

In this thesis, we have explored the use of high-level declarative language for the holistic optimisation and deployment of IoT data analytics applications across heterogeneous platforms, such as sensors, field gateways and clouds. We have designed and implemented an open-source *PATH2iot* system to automatically process the input from the system administrator in the form of Event Processing Language queries that are automatically decomposed, and a final deployment plan is derived through deterministic optimisation techniques. Furthermore, we have identified an opportunity to extend the existing EPL grammar and have implemented the extension successfully. This extension allows the system administrator to define queries containing time-based windows in a way that allows the *PATHfinder* module to perform a grid search to find the best size of the time-based window.

An Energy-based cost model with coefficients for a Pebble Steel smartwatch was described and validated with the use of a Power Monitoring tool. Results showed that a dramatic improvement in battery life can occur when using the optimised execution plan rather than the baseline approach: this was 453% for the running healthcare example. This shows the potential for IoT management systems that can automatically exploit fog/edge computing to optimise non-functional requirements, including battery life for sensors. It has the additional advantage of being able to distribute stream processing computations across multiple platforms without the need for the application programmer to know how to program each type of platform: this is done automatically, by platform-specific compilers and deployers, from a high level, declarative description of the computation. Furthermore, we have identified problems when relying on self-reported battery levels to predict how long the battery will last under different deployment

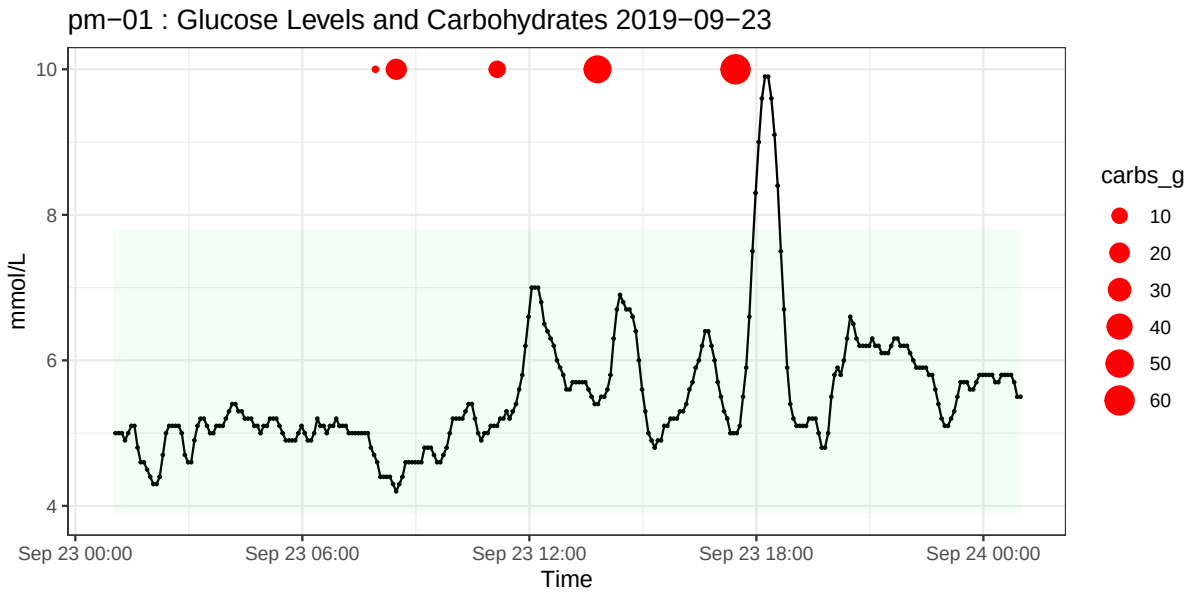
scenarios. This finding strengthens the need for working with uncertainties instead of point estimates when dealing with energy measurements. We have explored and compared two approaches – traditional frequentist, and an alternative Bayesian approach.

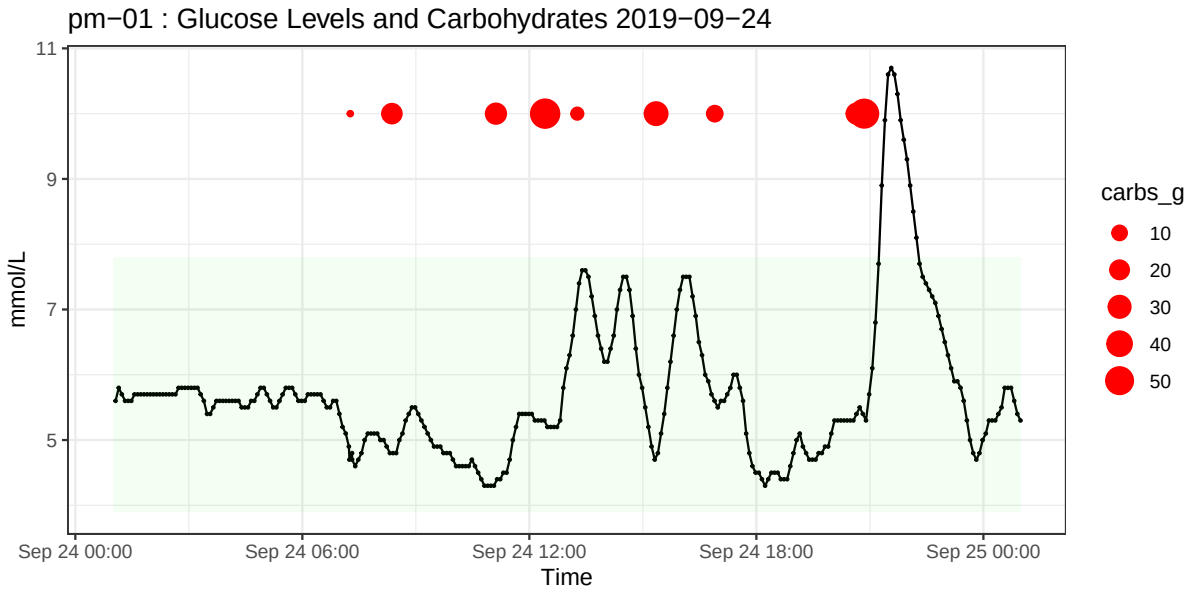
A bandwidth constraint presented by the use of LoRaWAN technology was explored in the second real-world use case, focused on the train arrival at a local Metro station. We have designed a system that can process an acoustic stream in real-time, and using an Fourier Fast Transform with determined sound frequencies, detecting the train in real-time, utilising only edge computational resources at the platform – a Raspberry Pi. The optimiser analysed 2^{32} logical plans and was able to determine the best plan to satisfy the network constraints, considerably reducing the amount of data that needs to be transmitted between the embedded device and clouds. We have also integrated regulatory requirements and The Things Network fair usage policy within the designed bandwidth model to ensure that these are met by the chosen plan.

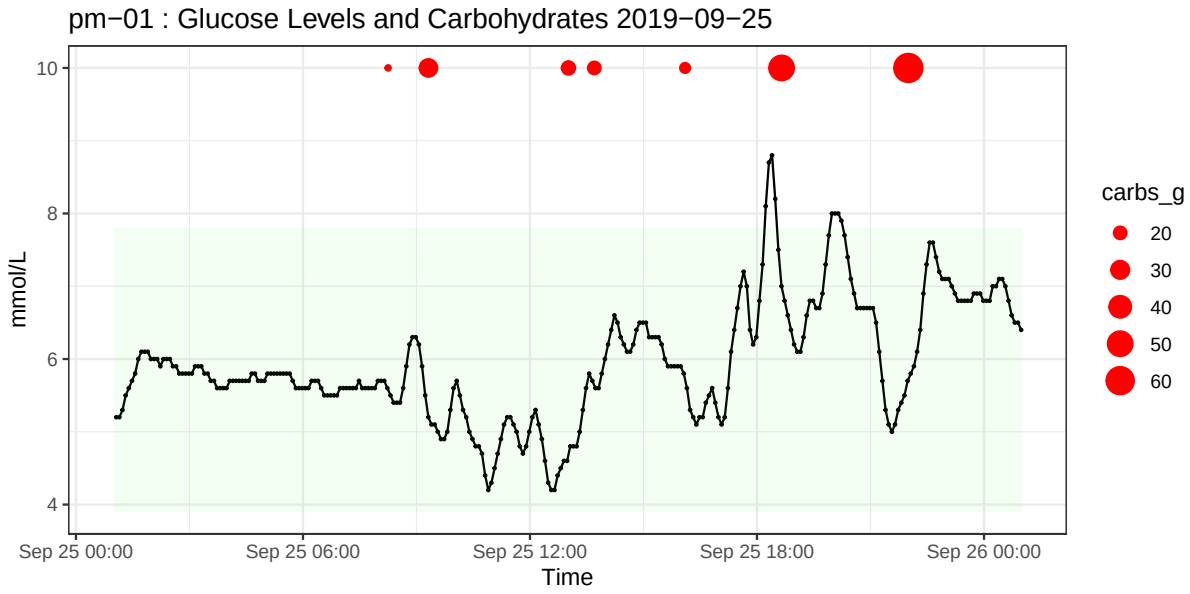
We have proved the advantages of this novel approach in real-world deployments, but also highlighted the limitations, especially the need for User Defined Functions that restrict optimisation capabilities. All the software is available as open source – another research group has exploited it for their own research [88], including adding a performance-based cost model. This work can be extended further as outlined in this chapter to suit future research and industry needs.

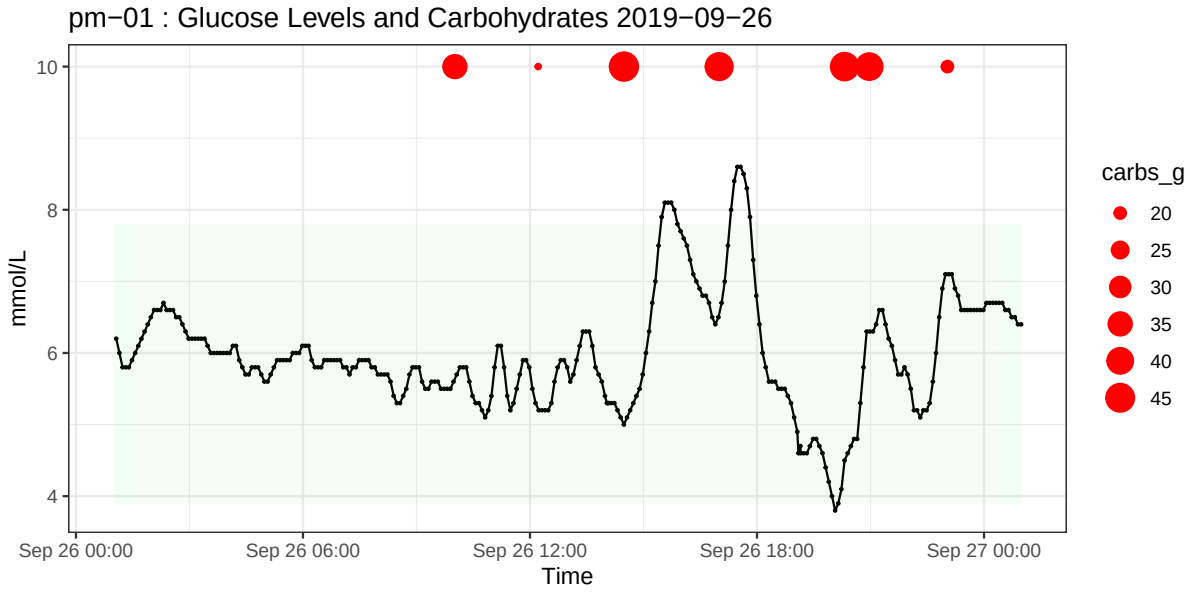
APPENDIX A
VISUALISATION OF COLLECTED HEALTHCARE DATA

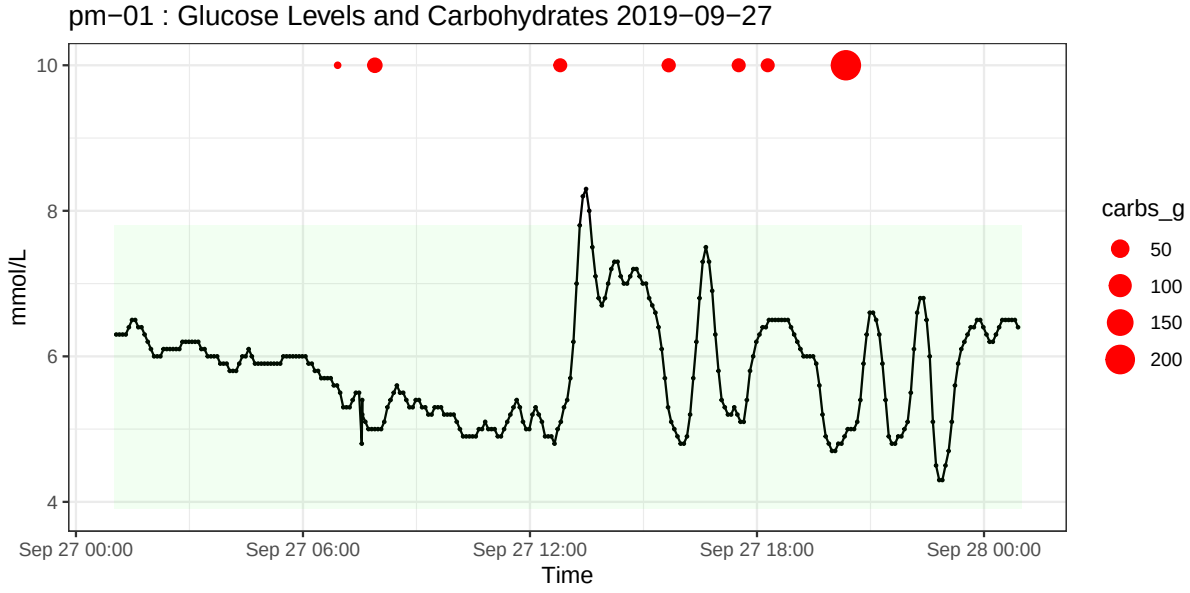
A.1 Glucose and Food Diary Data

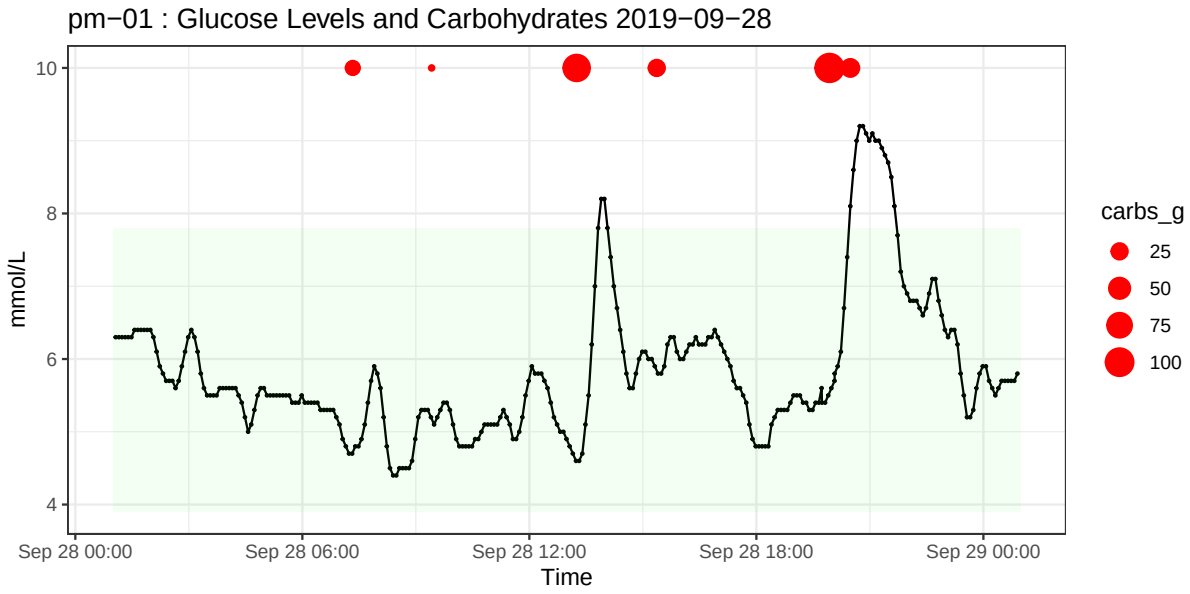


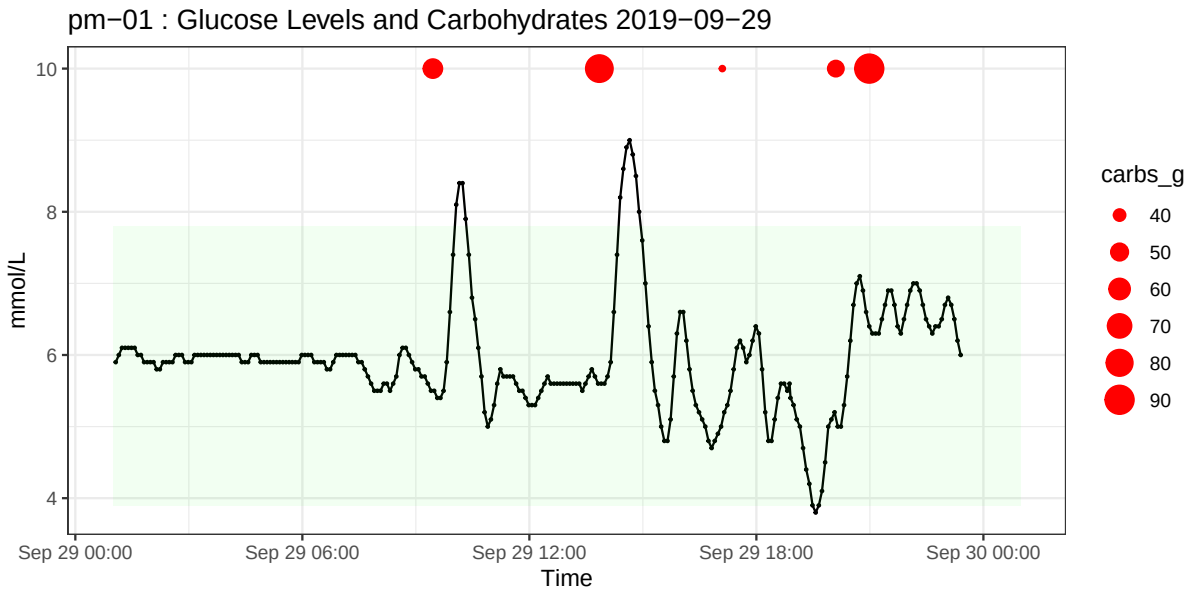


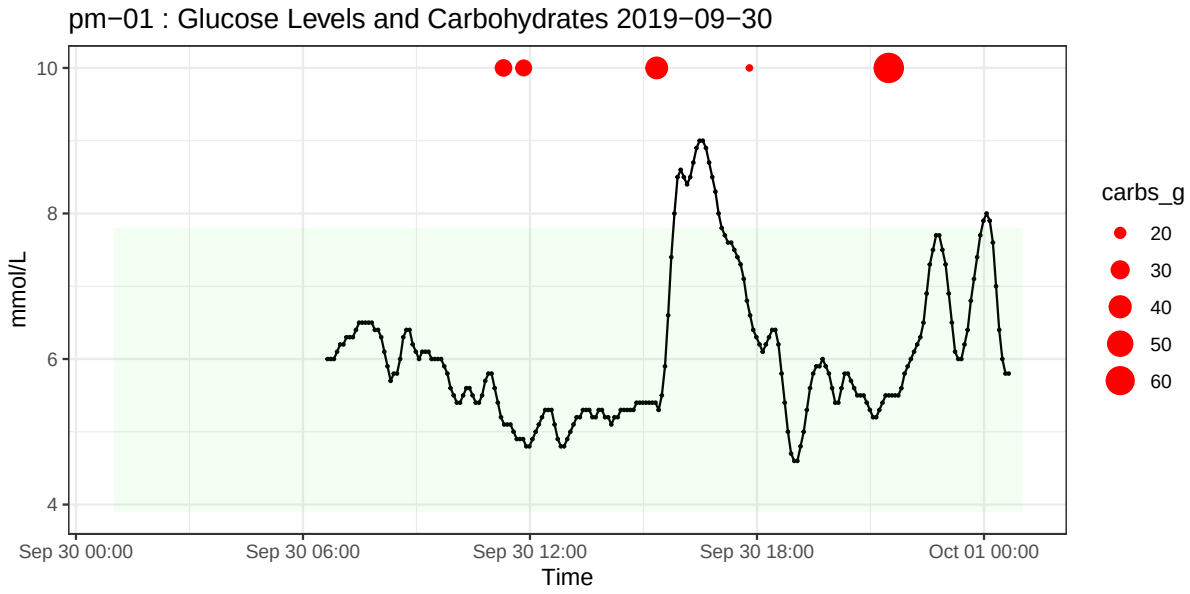

pm−01 : Glucose Levels and Carbohydrates 2019−09−25
mmol/L
10
8
6
4
Sep 25 00:00
Sep 25 06:00
Sep 25 12:00
Sep 25 18:00
Sep 26 00:00
Time
carbs_g
20
30
40
50
60

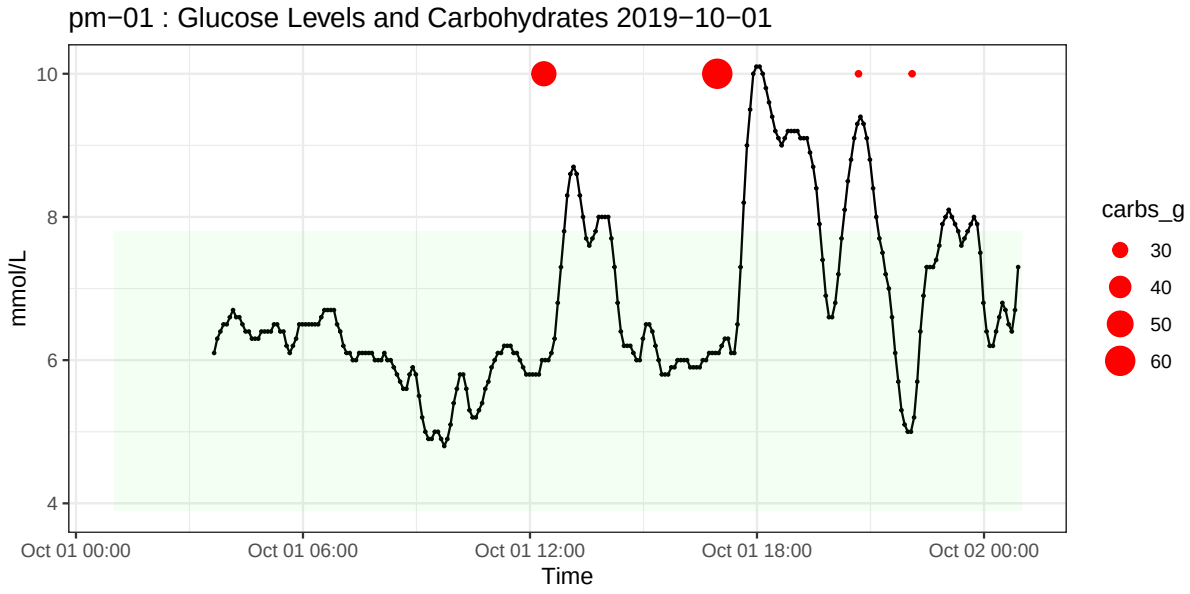

pm−01 : Glucose Levels and Carbohydrates 2019−09−26
mmol/L
10
8
6
4
Sep 26 00:00
Sep 26 06:00
Sep 26 12:00
Sep 26 18:00
Sep 27 00:00
Time
carbs_g
20
25
30
35
40
45


pm−01 : Glucose Levels and Carbohydrates 2019−09−27
mmol/L
10
8
6
4
Sep 27 00:00
Sep 27 06:00
Sep 27 12:00
Sep 27 18:00
Sep 28 00:00
Time
carbs_g
50
100
150
200

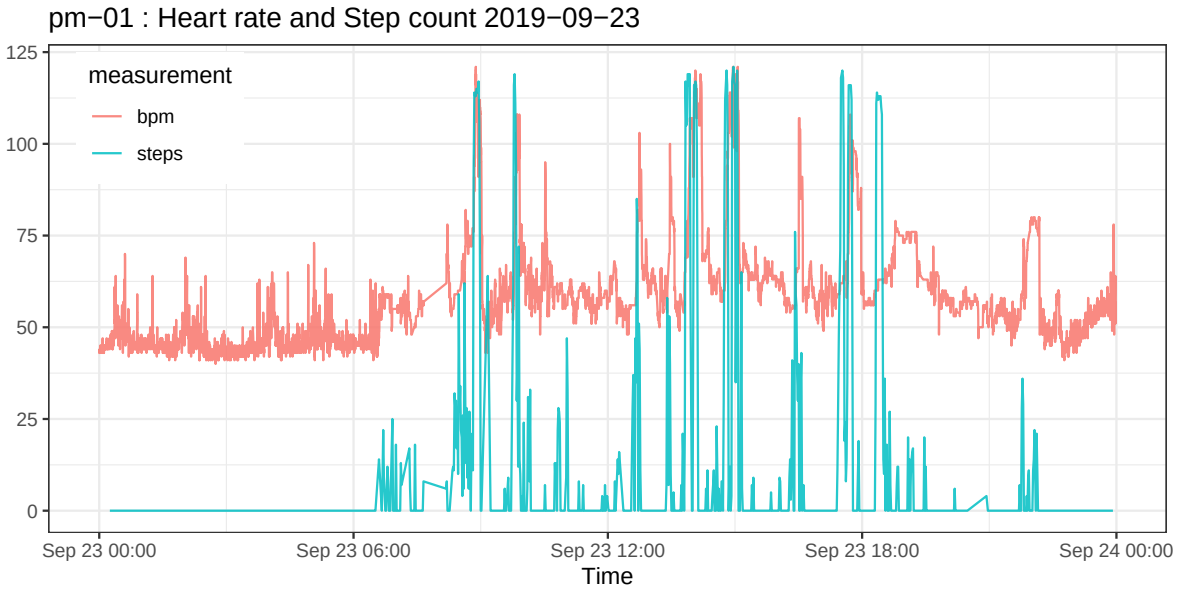

pm−01 : Glucose Levels and Carbohydrates 2019−09−28
mmol/L
10
8
6
4
Sep 28 00:00
Sep 28 06:00
Sep 28 12:00
Sep 28 18:00
Sep 29 00:00
Time
carbs_g
25
50
75
100

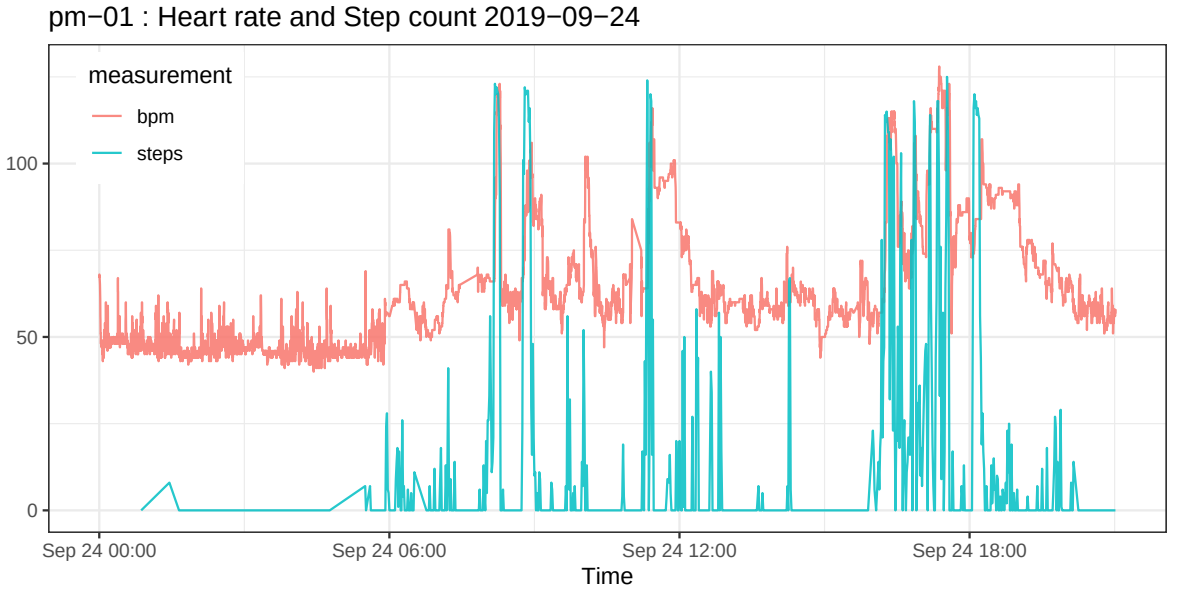

pm−01 : Glucose Levels and Carbohydrates 2019−09−29
mmol/L
10
8
6
4
Sep 29 00:00
Sep 29 06:00
Sep 29 12:00
Sep 29 18:00
Sep 30 00:00
Time
carbs_g
40
50
60
70
80
90

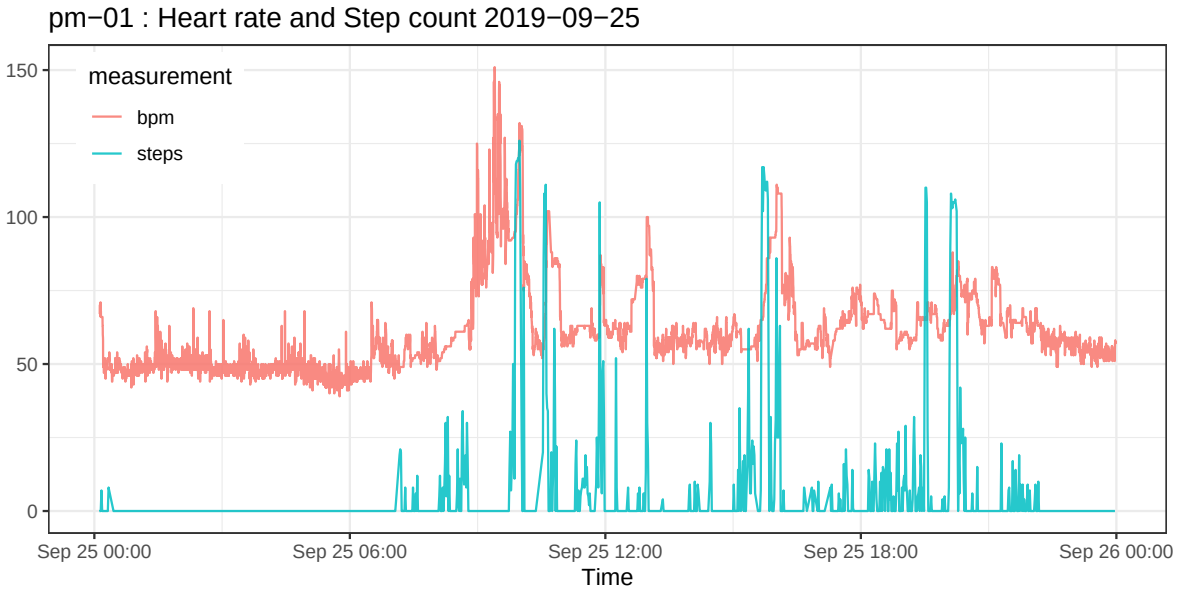

pm−01 : Glucose Levels and Carbohydrates 2019−09−30
mmol/L
10
8
6
4
Sep 30 00:00
Sep 30 06:00
Sep 30 12:00
Sep 30 18:00
Oct 01 00:00
Time
carbs_g
20
30
40
50
60


pm−01 : Glucose Levels and Carbohydrates 2019−10−01
mmol/L
10
8
6
4
Oct 01 00:00
Oct 01 06:00
Oct 01 12:00
Oct 01 18:00
Oct 02 00:00
Time
carbs_g
30
40
50
60

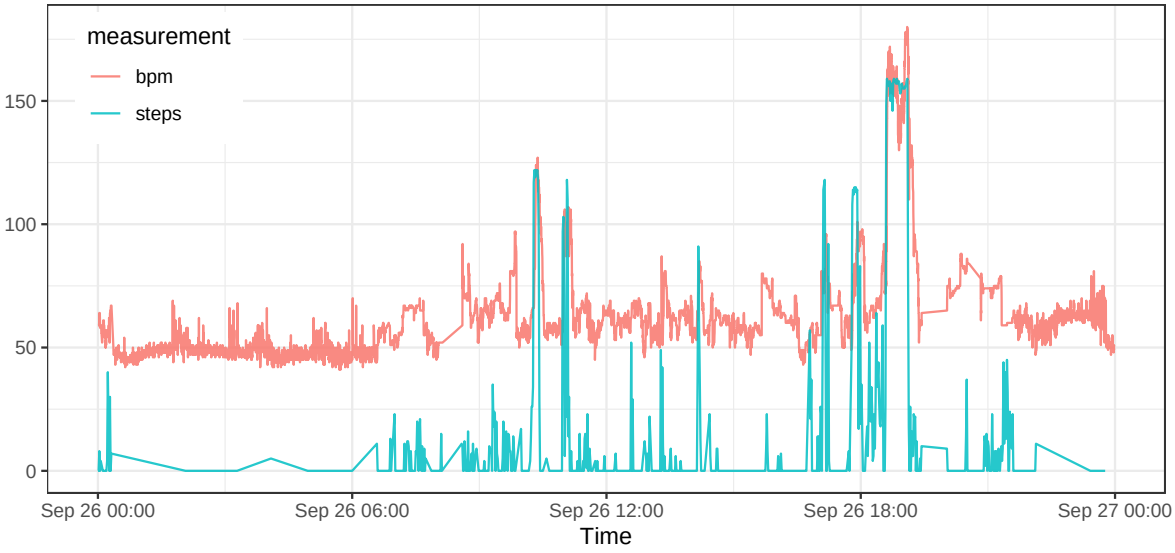
A.2 Heart Rate and Step Count Data



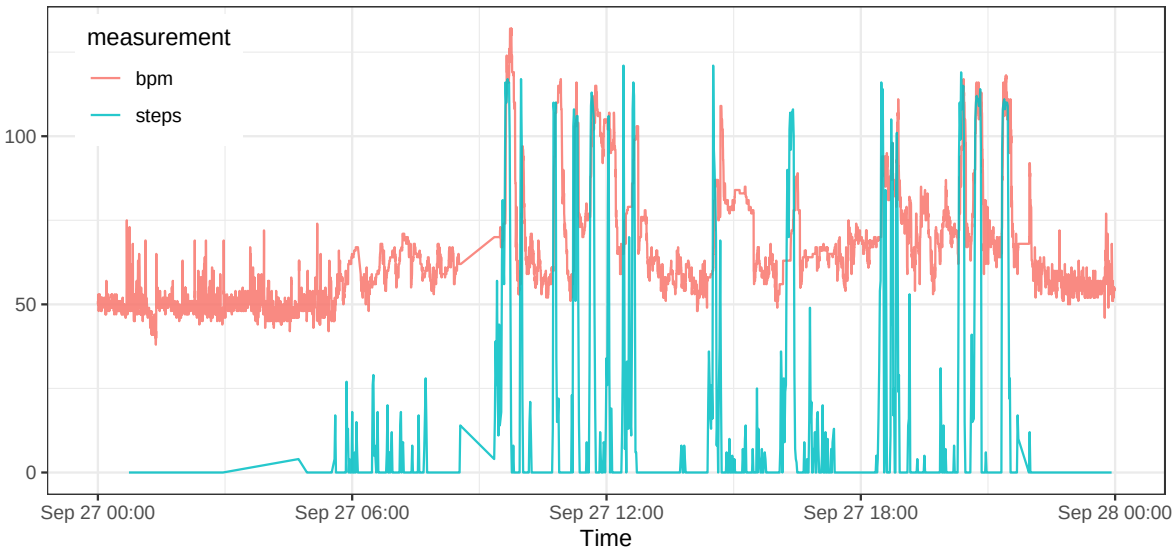




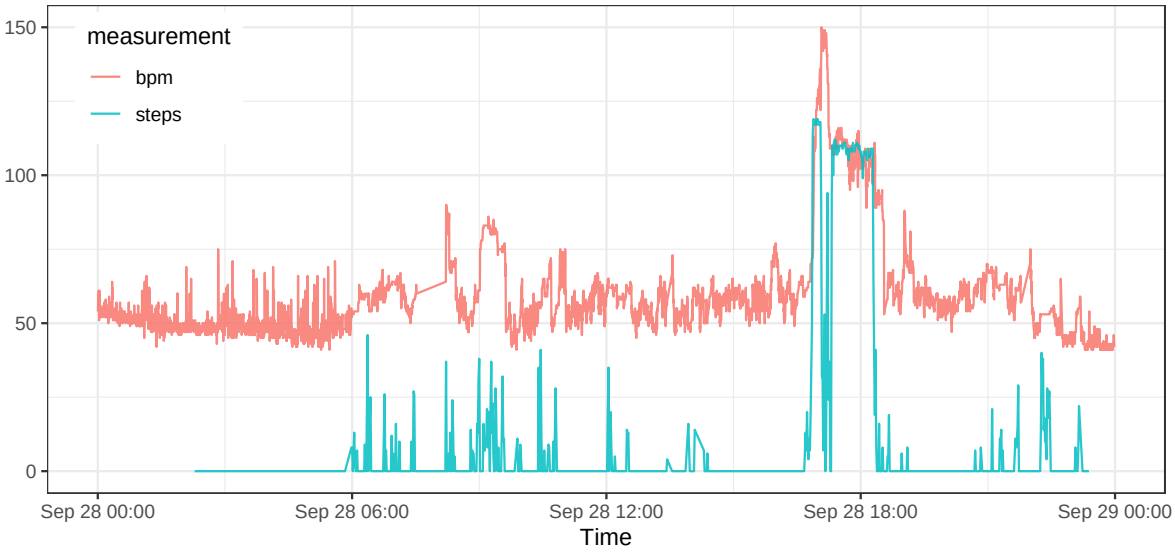
pm−01 : Heart rate and Step count 2019−09−26

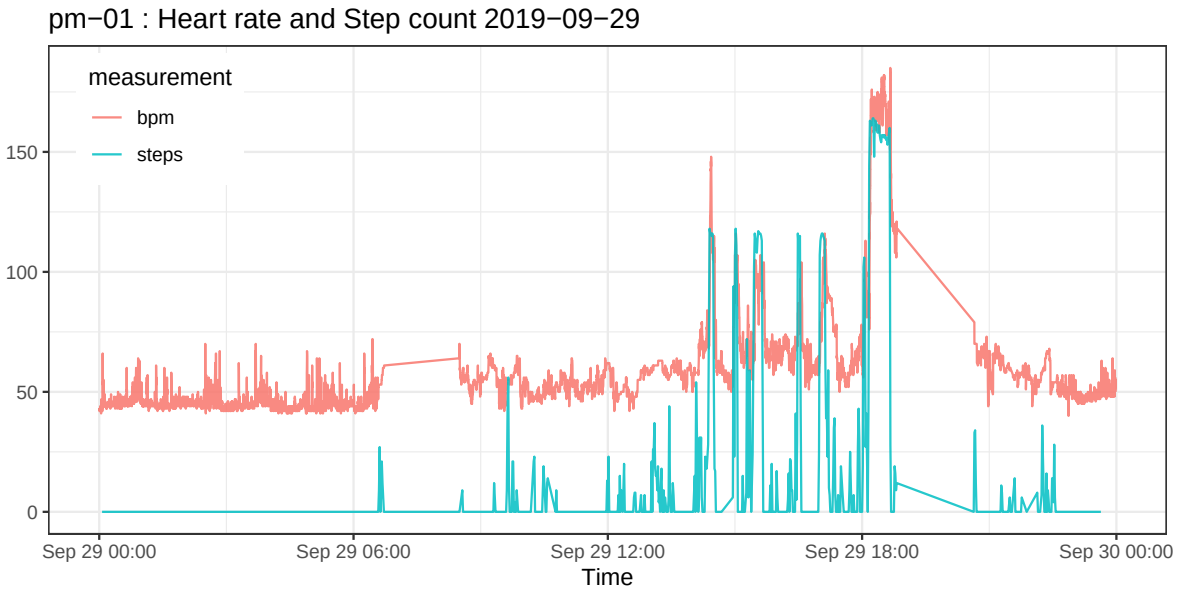


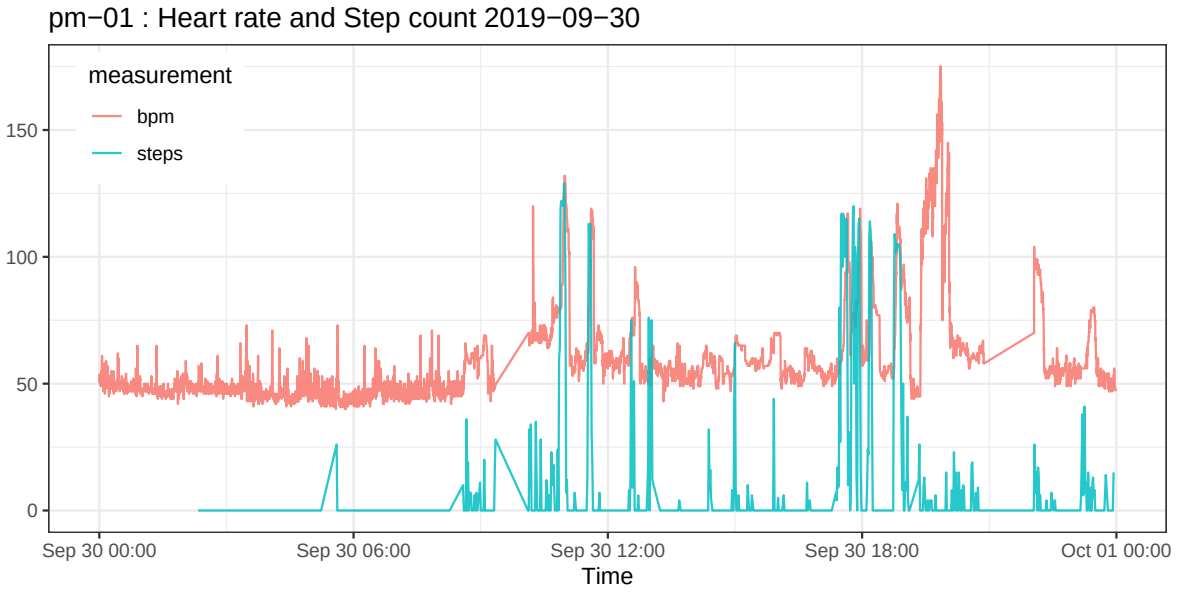
pm−01 : Heart rate and Step count 2019−09−27

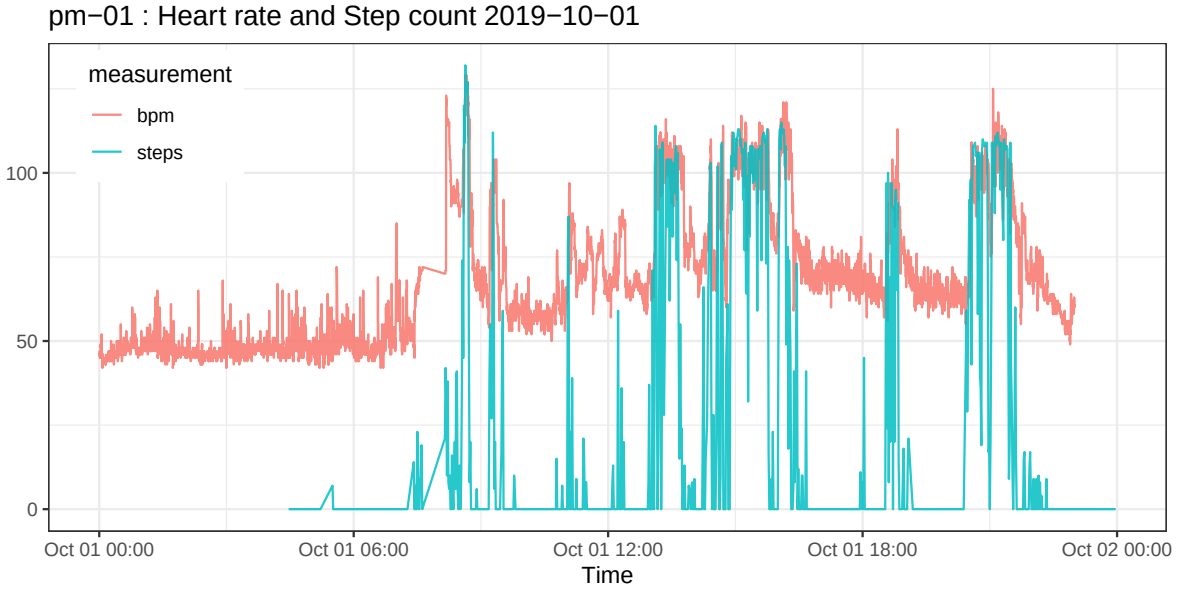


pm−01 : Heart rate and Step count 2019−09−28




pm−01 : Heart rate and Step count 2019−09−29
measurement
bpm
steps
150
100
50
0
Sep 29 00:00
Sep 29 06:00
Sep 29 12:00
Sep 29 18:00
Sep 30 00:00
Time


pm−01 : Heart rate and Step count 2019−09−30
measurement
bpm
steps
150
100
50
0
Sep 30 00:00
Sep 30 06:00
Sep 30 12:00
Sep 30 18:00
Oct 01 00:00
Time


pm−01 : Heart rate and Step count 2019−10−01
measurement
bpm
steps
100
50
0
Oct 01 00:00
Oct 01 06:00
Oct 01 12:00
Oct 01 18:00
Oct 02 00:00
Time

APPENDIX B
RESOURCE CATALOGUE - INPUT FILE

A example resource catalogue input file that was used for the healthcare use case:


```
{
        "nodes": [
        {
                "state": "active",
                "resourceId": 115001,
                "resourceType": "PebbleWatch",
                "swVersion": "1.0.0",
                "batteryCapacity_mAh": 130,
                "batteryVoltage_V": 3.7,
                "defaultNetworkFreq":2.5,
                "defaultWindowLength":0.5,
                "resources":
                {
                        "cpu": 40.0,
                        "ram": 6000,
                        "disk": 16.0,
                        "monetaryCost": 0.0,
                        "energyImpact": 100,
                        "securityLevel": 1
                }
                ,
                "connections": [
                {
                        "downstreamNode": 115002,
                        "bandwidth":100,
                        "monetaryCost":0.0
                }
                ],
                "capabilities": [
                {
                        "name":"UDF",
                        "operator":"getAccelData",
                        "supportsWin": false
                },
```

```
                {
                        "name":"RelationalOpExpression",
                        "operator":"=",
                        "supportsWin": false
                },
                {
                        "name":"ArithmaticExpression",
                        "operator":"*",
                        "supportsWin": false
                },
                {
                        "name":"DotExpression",
                        "operator": "Math.pow",
                        "supportsWin": false
                },
                {
                        "name":"win",
                        "operator":"*",
                        "supportsWin": false
                }
                ]
        },
        {
                "state": "active",
                "resourceId": 115002,
                "resourceType": "iPhone",
                "batteryCapacity_mAh": 3000,
                "batteryVoltage_V": 3.85,
                "swVersion": "1.0.0",
                "defaultNetworkFreq":1,
                "defaultWindowLength":0,
                "resources":
                {
                        "cpu": 200.0,
                        "ram": 1000000.0,
                        "disk": 4000.0,
                        "monetaryCost": 0.001,
                        "energyImpact": 0.001,
                        "securityLevel": 1
                },
                "connections": [
                {
                        "downstreamNode": 65001,
                        "bandwidth":54000,
                        "monetaryCost":0.0
                }
                ],
                "capabilities": [
                {
                        "name":"sxfer",
```

```
                "operator":"forward",
                "supportsWin": true
            }
            ]
        },
        {
            "state": "active",
            "resourceId": 65001,
            "resourceType": "ESPer",
            "swVersion": "1.0.0",
            "defaultNetworkFreq":1,
            "defaultWindowLength":0,
            "resources":
            {
                "cpu": 800.0,
                "ram": 16000000.0,
                "disk": 16000.0,
                "monetaryCost": 0.001,
                "energyImpact": 0.001,
                "securityLevel": 1
            },
            "connections": [
            {
                "downstreamNode": 65002,
                "bandwidth":100000,
                "monetaryCost":0.0
            }
            ],
            "capabilities": [
            {
                "name":"UDF",
                "operator":"persistResult",
                "supportsWin": true
            },
            {
                "name":"RelationalOpExpression",
                "operator":"*",
                "supportsWin": true
            },
            {
                "name": "ArithmaticExpression",
                "operator": "*",
                "supportsWin": true
            },
            {
                "name":"DotExpression",
                "operator":"*",
                "supportsWin": true
            },
            {
```

```
                "name": "MatchRecognizeClause",
                "operator": "*",
                "supportsWin": true
            },
            {
                "name":"win",
                "operator":"*",
                "supportsWin": true
            },
            {
                "name": "CountProjectionExpression",
                "operator": "*",
                "supportsWin": true
            }
            ]
    },
    {
            "state": "disable",
            "resourceId": 65002,
            "resourceType": "ESPer",
            "swVersion": "1.0.0",
            "defaultNetworkFreq":1,
            "defaultWindowLength":0,
            "resources":
            {
                "cpu": 800.0,
                "ram": 4000.0,
                "disk": 16000.0,
                "monetaryCost": 0.001,
                "energyImpact": 0.001,
                "securityLevel": 1
            },
            "connections": [
            {
                "downstreamNode": 65001,
                "bandwidth":100000,
                "monetaryCost":0.0
            },
            {
                "downstreamNode": 65003,
                "bandwidth":100000,
                "monetaryCost":0.0
            }
            ],
            "capabilities": [
            {
                "name":"UDF",
                "operator":"persistResult",
                "supportsWin": true
            },
```

```
                {
                        "name":"RelationalOpExpression",
                        "operator":"*",
                        "supportsWin": true
                },
                {
                        "name": "ArithmaticExpression",
                        "operator": "*",
                        "supportsWin": true
                },
                {
                        "name":"DotExpression",
                        "operator":"*",
                        "supportsWin": true
                },
                {
                        "name": "MatchRecognizeClause",
                        "operator": "*",
                        "supportsWin": true
                },
                {
                        "name":"win",
                        "operator":"*",
                        "supportsWin": true
                },
                {
                        "name": "CountProjectionExpression",
                        "operator": "*",
                        "supportsWin": true
                }
                ]
        }
        ],
        "messageBus": {
                "IP": "127.0.0.1",
                "port": 61616,
                "type": "activemq",
                "username": "",
                "pass": ""
        }
}
```

APPENDIX C

PATH2iot INPUT FILES

A template for the *PATH2iot* settings files:

```
# EPL
MASTER_QUERY_PATH=input/accelPebble/master_query.epl
STREAM_DEF=input/accelPebble/input_streams.json
UDF_DEF=input/accelPebble/udfs.json
INFRA_DEF=input/accelPebble/infrastructure_current_state.json
RESOURCE_EI=input/energy_coefficients.json
REQUIREMENT_DEF=input/accelPebble/requirements.json
EXEC_OUT_FILE=output/accel_exec_plan.csv

# NEO4J settings
NEO_IP=
NEO_PORT=
NEO_USERNAME=
NEO_PASSWORD=

# PATHdeployer connection details
PATH_DEPLOYER_IP=
PATH_DEPLOYER_PORT=

# External module connection details
EX_MODULE_ENABLED=false
EX_MODULE_IP=
EX_MODULE_PORT=
```

A complete set of EPL statements for the healthcare use case processing accelerometer stream (‘input/accelPebble/master_query.epl’):

```
INSERT INTO AccelEvent SELECT getAccelData(25, 60) \
    FROM AccelEventSource
INSERT INTO EdEvent SELECT Math.pow(x*x+y*y+z*z, 0.5) AS ed, ts \
    FROM AccelEvent WHERE vibe=0
INSERT INTO StepEvent SELECT ed1('ts') as ts FROM EdEvent \
    MATCH_RECOGNIZE (MEASURES A AS ed1, B AS ed2 \
    PATTERN (A B) DEFINE A AS (A.ed > 1.3), B AS (B.ed <= 1.3))
INSERT INTO StepCount SELECT count(*) as steps \
    FROM StepEvent.win:time_batch(120 sec)
# INSERT INTO StepCount SELECT count(*) as steps \
    FROM StepEvent.win:flexi_time_batch(30, 120, 15, sec)
SELECT persistResult(steps, "time_series", "step_sum") \
    FROM StepCount
```

An example used for the definition of the input streams in healthcare use case defining initial accelerometer data stream (‘input/accelPebble/input_streams.json’):

```
{
        "inputStreams": [
        {
                "streamName" : "AccelEventSource",
                "streamProperties" : [
                {
                        "name": "x",
                        "asName": "x",
                        "type": "double"
                },
                {
                        "name": "y",
                        "asName": "x",
                        "type": "double"
                },
                {
                        "name": "z",
                        "asName": "z",
                        "type": "double"
                },
                {
                        "name": "vibe",
                        "asName": "vibe",
                        "type": "integer",
                        "selectivity": "0.87"
                },
                {
                        "name": "ts",
                        "asName": "ts",
                        "type": "long"
                }
                ]
        }
        ]
}
```

An example of energy coefficients used by the *PATH2iot* system for the healthcare use case (‘input/energy_coefficients.json’):


```
{
        "energyResources": [
        {
                "resourceType": "PebbleWatch",
                "swVersion": "1.0.0",
                "EIcoefficients": [

                {
                        "type": "OSidle",
                        "operator": "",
                        "cost": 1.780797328,
                        "confInt": 0.0370,
                        "generationRatio":1,
                        "selectivityRatio":1
                },
                {
                        "type": "UDF",
                        "operator": "getAccelData",
                        "cost": 0.060977212,
                        "confInt": 0.0153,
                        "generationRatio":150,
                        "selectivityRatio":1
                },
                {
                        "type": "RelationalOpExpression",
                        "operator": "=",
                        "cost": 0.091714146,
                        "confInt": 0.0416,
                        "generationRatio":1,
                        "selectivityRatio":1
                },
                {
                        "type": "ArithmaticExpression",
                        "operator": "*",
                        "cost": 0.335136959,
                        "confInt": 0.0665,
                        "generationRatio":1,
                        "selectivityRatio":0.33333333333
                },
                {
                        "type": "DotExpression",
                        "operator": "Math.pow",
                        "cost": 0.032450218,
                        "confInt": 0.1039,
                        "generationRatio":1,
                        "selectivityRatio":1
                },
```

```
                {
                        "type": "win",
                        "operator": "*",
                        "cost": 0.062281175,
                        "confInt": 0.0605,
                        "generationRatio":1,
                        "selectivityRatio":1
                },
                {
                        "type": "netCost",
                        "operator": "",
                        "cost": 5.064357224,
                        "confInt": 0.2747,
                        "generationRatio":1,
                        "selectivityRatio":1
                },
                {
                        "type": "bleActive",
                        "operator": "",
                        "cost": 12.11834585,
                        "confInt": 0.2747,
                        "generationRatio":1,
                        "selectivityRatio":1
                }
                ]
        },
        {
                "resourceType": "iPhone",
                "swVersion": "1.0.0",
                "EIcoefficients": [
                {
                        "type": "OSidle",
                        "operator": "",
                        "cost": 0,
                        "confInt": 0,
                        "generationRatio":1,
                        "selectivityRatio":1
                },
                {
                        "type": "sxfer",
                        "operator": "forward",
                        "cost": 0,
                        "confInt": 0,
                        "generationRatio":1,
                        "selectivityRatio":1
                },
                {
                        "type": "netCost",
                        "operator": "",
                        "cost": 0,
```

```
                "confInt": 0,
                "generationRatio":1,
                "selectivityRatio":1
        },
        {
                "type": "bleActive",
                "operator": "",
                "cost": 0,
                "confInt": 0,
                "generationRatio":1,
                "selectivityRatio":1
        }
        ]
},
{
        "resourceType": "ESPer",
        "swVersion": "1.0.0",
        "EIcoefficients":
        [
        {
                "type": "OSidle",
                "operator": "",
                "cost": 0,
                "confInt": 0,
                "generationRatio":1,
                "selectivityRatio":1
        },
        {
                "type": "sxfer",
                "operator": "",
                "cost": 0,
                "confInt": 0,
                "generationRatio":1,
                "selectivityRatio":1
        },
        {
                "type": "UDF",
                "operator": "persistResult",
                "cost": 0,
                "confInt": 0,
                "generationRatio":1,
                "selectivityRatio":1
        },
        {
                "type": "RelationalOpExpression",
                "operator": "*",
                "cost": 0,
                "confInt": 0,
                "generationRatio":1,
                "selectivityRatio":1
```

```
},
{
        "type": "ArithmaticExpression",
        "operator": "*",
        "cost": 0,
        "confInt": 0,
        "generationRatio":1,
        "selectivityRatio":1
},
{
        "type": "DotExpression",
        "operator": "*",
        "cost": 0,
        "confInt": 0,
        "generationRatio":1,
        "selectivityRatio":1
},
{
        "type": "MatchRecognizeClause",
        "operator": "*",
        "cost": 0,
        "confInt": 0,
        "generationRatio":1,
        "selectivityRatio":1
},
{
        "type": "win",
        "operator": "*",
        "cost": 0,
        "confInt": 0,
        "generationRatio":1,
        "selectivityRatio":1
},
{
        "type": "CountProjectionExpression",
        "operator": "*",
        "cost": 0,
        "confInt": 0,
        "generationRatio":1,
        "selectivityRatio":1
},
{
        "type": "netCost",
        "operator": "",
        "cost": 0,
        "confInt": 0,
        "generationRatio":1,
        "selectivityRatio":1
},
{
```

```
                        "type": "bleActive",
                        "operator": "",
                        "cost": 0,
                        "confInt": 0,
                        "generationRatio":1,
                        "selectivityRatio":1
                }
                ]
        }
        ]
}
```

An example of the requirement file that was used in the healhcare use case (‘input/accelPebble/requirements.json’):

```
{
        "requirements": [
        {
                "device": "PebbleWatch",
                "reqType": "energy",
                "min": -1,
                "max": 48,
                "units": "hour"
        }]
}
```

An example of the UDF file that defines the computation added by the application developer (‘input/accelPebble/udfs.json’):

```
{
        "udf": [
        {
                "name": "getAccelData",
                "output": "AccelEvents",
                "frequency": 25,
                "generationRatio": 150,
                "selectivityRatio": 1.0,
                "isSource": true,
                "notes": "This is an UDF generating initial
                accelerometer events available from bespoke
                Pebble Watch App.
                Parameters: 1 - data (x,y,z,vibe,ts,battery)
                represented by binary switch flags
                (e.g. 111101b - 64d - all on except ts);
                2 - freq (0, 10, 25, 50, 100);
                3 - name of the queue for the data",
                "support": [
                {
                        "device": "PebbleWatch",
                        "version": "1.0.0",
                        "metrics": [
                        {
                                "cpuCost": 5.0,
                                "ramCost": 1.0,
                                "diskCost": 0.0,
                                "dataOut": 18,
                                "monetaryCost": 0,
                                "securityLevel": 1
                        }
                        ]
                }
                ]
        },
        {
                "name": "printResult",
                "output": "",
                "frequency": 0,
                "generationRatio": 1,
                "selectivityRatio": 0,
                "isSource": false,
                "notes": "This is a global sink representation
                via UDF.",
                "support": [
                {
                        "device": "ESPer",
                        "version": "0.0.1",
```

```
                        "metrics": [
                        {
                                "cpuCost": 1.0,
                                "ramCost": 1.0,
                                "diskCost": 0.0,
                                "dataOut": 0,
                                "monetaryCost": 0,
                                "securityLevel": 1
                        }
                        ]
                }
                ]
        },
        {
                "name": "persistResult",
                "output": "",
                "frequency": 0,
                "generationRatio": 1,
                "selectivityRatio": 0,
                "isSource": false,
                "notes": "This is a global sink UDF that saves
                all data in database.
                Parameters: 1. database name, 2. table",
                "support": [
                {
                        "device": "ESPer",
                        "version": "0.0.1",
                        "metrics": [
                        {
                                "cpuCost": 1.0,
                                "ramCost": 1.0,
                                "diskCost": 0.0,
                                "dataOut": 0,
                                "monetaryCost": 0,
                                "securityLevel": 1
                        }
                        ]
                }
                ]
        }
        ]
}
```

APPENDIX D
d2esper

d2Esper – Dockerised Dynamic Esper (read 'd-squared esper') – is a standalone stream processor that has been designed as a wrapper for the Esper CEP library. This open-source stream processor is a key part of *PATH2IoT* deployment strategy as the limitations of most IoT devices do not allow for all of the computation to be pushed onto them, so the remainder of the computation must occur on the cloud. The *D2ESPer* allows the system to offload to the cloud all operators that are either not deployable, or are too expensive to deploy in IoT. It is encapsulated in a Docker container [112], which makes it easy to deploy as all of the dependencies are packaged in the container.

Upon activation, *d2Esper* registers itself with a ZooKeeper node. When it receives a configuration set from *PATHdeployer* – a JSON formatted sample can be reviewed from D – it dynamically loads the event (type) definition, parses the provided EPL statements and connects to a specified broker (in this case ActiveMQ) to start the processing of the real-time data stream. The output of the processing is forwarded to a specified destination (usually a different queue on the same broker) or stored in a database such as Influx DB.

- RESOURCE_ID – a unique ID that identifies the d2esper instance;
- LOCAL_EPL – a boolean flag, if set to 0 the tool will register to the ZooKeeper; register itself by creating an ephemeral node with specified resource id, and setup a watcher for incoming configuration from *PATHdeployer*; the flag can be set to '1' if the configuration is already available, e.g. at the shared disk, or blob storage, the path to the configuration must be provided;
- ZooKeeper settings - connection details for the ZooKeeper instance.

A JSON formatted configuration sample file, that is distributed by *PATHdeployer* to each of the 'd2esper' docker containers through Apache Zookeeper:

```
{
  "streams" : [
    {
      "eventName" : "AccelEvent",
      "eventProperties" : [
        {"key": "ts", "type": "long"},
        {"key" : "x", "type": "double"},
        {"key": "y",  "type": "double"},
        {"key": "z",  "type": "double"}],
      "source": {
        "IP" : "localhost",
        "port": 61616,
        "queue": "AccelEvent",
        "type": "activemq"
      }
    },
    {
      "eventName": "EDEvent",
      "eventProperties" : [
        {"key": "ts", "type": "long"},
        {"key": "ed", "type": "double"}],
      "source": {
        "IP" : "",
        "port": 0,
        "queue": "",
        "type": "internal"
      }
    }],
  "statements" : [
    {
      "statement": "INSERT INTO EDEvent SELECT x*x+y*y+z*z AS ed,
                    ts FROM AccelEvent",
      "output": {
        "IP" : "",
        "port": 0,
        "queue": "",
        "type": "internal"
      }
    },
    {
      "statement": "SELECT Math.sqrt(ed) as sqrted, ts
                    FROM EDEvent",
      "output": {
        "IP" : "127.0.0.1",
        "port": 61616,
        "queue": "SqrtEdEvent",
        "type": "activemq"
      }
    }]
}
```

An example of the UDF implementation within the tool that can connect to the InfluxDB database and persist received measurements.

```
public static void persistResult(double count,
                                 String measurement,
                                 String field) {
    logger.debug(String.format("Received a message: %f, %s, %s.",
                               count, measurement, field));

    // establish connection to influxdb
    InfluxDB influxDB = InfluxDBFactory.connect(INFLUX_DB_CONNECTION,
                                                INFLUX_DB_USER,
                                                INFLUX_DB_PASS);
    influxDB.setDatabase(measurement);

    Point point = Point.measurement(measurement)
            .time(System.currentTimeMillis(), TimeUnit.MILLISECONDS)
            .addField(field, count)
            .build();

    influxDB.write(point);

    influxDB.close();
}
```

The tool is also made open-source, under the GPLv2 license, and the source code can be viewed at GitHub https://github.com/PetoMichalak/phd-d2esper.

References

[1] CERN; Facts and figures; online, accessed on 22/08/2021; https://home.cern/resources/faqs/facts-and-figures-about-lhc.

[2] Gartner; Gartner Hype Cycle: Interpreting Technology Hype; online, accessed on 22/08/2021; https://www.gartner.com/en/research/methodologies/gartner-hype-cycle.

[3] McKinsey; The Internet of Things: How to capture the value of IoT; https://www.mckinsey.com/featured-insights/internet-of-things/our-insights/the-internet-of-things-how-to-capture-the-value-of-iot.

[4] Statista; The Internet of Things (IoT) units installed base by category from 2014 to 2020 (in billions); online, accessed on 22/08/2021; https://www.statista.com/statistics/370350/internet-of-things-installed-base-by-category/.

[5] Apache Spark; Apache Spark: Fast and general engine for big data processing; online, accessed on 22/08/2021; https://spark.apache.org/.

[6] Apache Storm; Apache Storm: realtime computation system; online, accessed on 22/08/2021; https://storm.apache.org/.

[7] McKinsey; Growing opportunities in the Internet of Things; online, accessed on 22/08/2021; https://www.mckinsey.com/industries/private-equity-and-principal-investors/our-insights/growing-opportunities-in-the-internet-of-things.

[8] IBM; The IBM Punched Card; online, accessed on 22/08/2021; https://www.ibm.com/ibm/history/ibm100/us/en/icons/punchcard/.

[9] Apache Samza; online, accessed on 28/08/2021; http://samza.apache.org.

[10] Apache Storm - Trident; online, accessed on 28/08/2021; http://storm.apache.org/documentation/Trident-tutorial.html.

[11] Esper Performance; online, accessed on 22/08/2021; http://esper.espertech.com/release-5.3.0/esper-reference/html/performance.html.

[12] Apache Flink; online accessed on 22/08/2021; http://flink.apache.org.

[13] Apache Kafka; online, accessed on 22/08/2021, http://kafka.apache.org/.

[14] Drools Fusion; online, accessed on 22/08/2021; http://docs.jboss.org/drools/release/6.3.0.Final/drools-docs/html_single/index.html#DroolsComplexEventProcessingChapter.

[15] EsperTech Inc.; Esper Reference v8.5.0;; online, access on 22/08/2021; http://esper.espertech.com/release-8.5.0/reference-esper/pdf/esper_reference.pdf.

[16] Drools Rule Language; online, accessed on 22/08/2021; http://docs.jboss.org/drools/release/6.3.0.Final/drools-docs/html_single/index.html#DroolsLanguageReferenceChapter.

[17] Eggum, Marcel; Smartphone Assisted, Complex Event Processing (Master Thesis); Department of Informatics, University of Oslo, https://www.duo.uio.no/handle/10852/41663.

[18] Dr Wheater, Stuart, confirmed intention of the DataSHIELD team to use the health related data, that was collected during this research project, and also outlined the development scope. Private communication, 12/06/2020.

[19] Mayo Clinic; Diabetes, Symptoms and Causes; online, accessed on 22/08/2021; https://www.mayoclinic.org/diseases-conditions/diabetes/symptoms-causes/syc-20371444.

[20] Apache Flink; Stream Processing for Everyone with SQL and Apache Flink; online, accessed on 22/08/2021; https://flink.apache.org/news/2016/05/24/stream-sql.html.

[21] EsperTech; Esper Reference - SODA API; online, accessed on 22/08/2021; http://esper.espertech.com/release-6.0.1/esper-reference/html/api.html#api-soda.

[22] Dr Griffin, Jory, and, Prof Watson, Paul, contributed the formula for calculation of a number of physical plans under pipeline scenario. Private communication, 20/05/2017.

[23] Monsoon Solutions Inc.; Product Documentation > Power Monitor End User Manual; online, accessed on 22/08/2021; https://www.msoon.com/hvpm-product-documentation.

[24] Android, Android Developers > Docs > Guides > Understand the Activity Lifecycle; online, accessed on 22/08/2021; https://developer.android.com/guide/components/activities/activity-lifecycle.

[25] Dr Heaps, Sarah, provided ongoing input and support for the development of the frequentist and Bayesian approach to express the uncertainty used in the energy cost model. Private communication, 2016-2020.

[26] The Things Network; How Spreading Factor affects LoRaWAN device battery life; online, accessed on 22/08/2021; https://www.thethingsnetwork.org/article/how-spreading-factor-affects-lorawan-device-battery-life.

[27] Folium: Python visualisation package, online, accessed on 22/08/2021; https://python-visualization.github.io/folium/.

[28] Leaflet.js: open-source JavaScript library for mobile-friendly interactive maps; online, access on 22/08/2021; https://leafletjs.com.

[29] Microsoft Azure; Cognitive Services; online, accessed on 22/08/2021; https://azure.microsoft.com/en-us/services/cognitive-services/.

[30] Google Cloud; Speech-to-Text; online, accessed on 22/08/2021; https://cloud.google.com/speech-to-text/.

[31] R: A language and environment for statistical computing, Team, R Core and others; online, accessed on 22/08/2021.

[32] tuneR: Analysis of Music and Speech; online, accessed on 22/08/2021; https://CRAN.R-project.org/package=tuneR.

[33] signal: Signal processing R package; online, accessed on 22/08/2021; http://r-forge.r-project.org/projects/signal/.

[34] SEMTECH; SX1272/3/6/7/8: LoRa Modem Designer's Guide AN1200.13; online, accessed on 22/08/2021; https://www.rs-online.com/designspark/rel-assets/ds-assets/uploads/knowledge-items/application-notes-for-the-internet-of-things/LoRa%20Design%20Guide.pdf.

[35] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The Design of the Borealis Stream Processing Engine. In *Conference on Innovative Data Systems Research*, volume 5, pages 277–289, 2005.

[36] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal—The International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[37] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[38] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. Spc: A distributed, scalable platform for data mining. In *Proceedings of the 4th international workshop on Data mining standards, services and platforms*, pages 27–37. ACM, 2006.

[39] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream: The stanford data stream management system. In *Data Stream Management*, pages 317–336. Springer, 2016.

[40] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark SQL: Relational data processing in Spark. In *ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[41] Kevin Ashton et al. That 'Internet of Things' thing. *RFID journal*, 22(7):97–114, 2009.

[42] Cosmin Avasalcai and Schahram Dustdar. Latency-aware decentralized resource management for IoT applications. In *Proceedings of the 8th International Conference on the Internet of Things - IOT '18*, page 1–4. ACM Press, 2018.

[43] Oresti Banos, Muhammad Bilal Amin, Wajahat Ali Khan, Muhammad Afzal, Maqbool Hussain, Byeong Ho Kang, and Sungyong Lee. The mining minds digital health and wellness framework. *BioMedical Engineering OnLine*, 15(S1), Jul 2016.

[44] Alessandro Bettini. *A Course in Classical Physics 3 Electromagnetism*. Springer, 2016.

[45] N. Bidargaddi, A. Sarela, L. Klingbeil, and M. Karunanithi. Detecting walking activity in cardiac rehabilitation by using accelerometer. In *Sensor Networks and Information 2007 3rd International Conference on Intelligent Sensors*, page 555–560, Dec 2007.

[46] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog computing: A platform for Internet of Things and analytics. In *Big data and internet of things: A roadmap for smart environments*, pages 169–186. Springer, 2014.

[47] 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*, page 13. ACM Press, 2012.

[48] Yu Cao, Peng Hou, Donald Brown, Jie Wang, and Songqing Chen. Distributed analytics and edge intelligence: Pervasive health monitoring at the era of fog computing. In *Proceedings of the 2015 Workshop on Mobile Big Data*, pages 43–48. ACM, 2015.

[49] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[50] Joao Carreira and Jianneng Li. Optimizing latency and throughput trade-offs in a stream processing system. *University of California at Berkeley, Computer Science Division*, 2014.

[51] Sophie Cassidy, Vivek Vaidya, David Houghton, Pawel Zalewski, Jelena P Seferovic, Kate Hallsworth, Guy A MacGowan, Michael I Trenell, and Djordje G Jakovljevic. Unsupervised high-intensity interval training improves glycaemic control but not cardiovascular autonomic function in type 2 diabetes patients: A randomised controlled trial. *Diabetes and Vascular Disease Research*, 16(1):69–76, Jan 2019.

[52] Tyrone Gene Ceaser. The estimation of caloric expenditure using three triaxial accelerometers. 2012.

[53] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. Trill: a high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, 2014.

[54] Devaki Chandramouli, Rainer Liebhart, and Juho Pirskanen. *5G for the Connected World*. John Wiley & Sons, 2019.

[55] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R Madden, Fred Reiss, and Mehul A Shah. TelegraphCQ: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003.

[56] Sheri R Colberg, Ronald J Sigal, Jane E Yardley, Michael C Riddell, David W Dunstan, Paddy C Dempsey, Edward S Horton, Kristin Castorino, and Deborah F Tate. Physical activity/exercise and diabetes: a position statement of the American Diabetes Association. *Diabetes Care*, 39(11), 2016.

[57] Christian Constanda, Matteo Dalla Riva, Pier Domenico Lamberti, and Paolo Musolino. *Integral Methods in Science and Engineering, Volume 2: Practical Applications*. Birkhäuser, 2017.

[58] Thomas Cooper. Proactive scaling of distributed stream processing work flows using workload modelling: doctoral symposium. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 410–413. ACM, 2016.

[59] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62, 2010.

[60] Li Da Xu, Wu He, and Shancang Li. Internet of things in industries: A survey. *IEEE Transactions on Industrial Informatics*, 10(4):2233–2243, 2014.

[61] Jay Danner, Linda Wills, Elbert M. Ruiz, and Lee W. Lerner. Rapid Precedent-Aware Pedestrian and Car Classification on Constrained IoT Platforms. In *Proceedings of the 14th ACM/IEEE Symposium on Embedded Systems for Real-Time Multimedia - ESTIMedia'16*, page 29–36. ACM Press, 2016.

[62] Roshan Bharath Das, Nicolae Vladimir Bozdog, Marc X Makkes, and Henri Bal. Kea: A computation offloading system for smartphone sensor data. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 9–16. IEEE, 2017.

[63] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[64] Thomas Degen, Heinz Jaeckel, Michael Rufer, and Stefen Wyss. SPEEDY: A Fall Detector in a Wrist Watch. In *International Symposium on Wearable Computers*, 2003.

[65] 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, 2016.

[66] Matthew Forshaw, Nigel Thomas, and A Stephen McGough. The case for energy-aware simulation and modelling of internet of things (IoT). *ACM ENERGY-SIM*, 2016.

[67] Keke Gai, Meikang Qiu, and Hui Zhao. Energy-aware task assignment for mobile cyber-enabled applications in heterogeneous cloud computing. *Journal of Parallel and Distributed Computing*, 111:126–135, 2018.

[68] Woon Siong Gan. *Signal Processing and Image Processing for Acoustical Imaging*. Springer, 2020.

[69] Hector Garcia-Molina. *Database systems: the complete book*. Pearson Education India, 2008.

[70] Sandro Rodriguez Garzon, Sebastian Walther, Shaoning Pang, Bersant Deva, and Axel Küpper. Urban air pollution alert service for smart cities. In *Proceedings of the 8th International Conference on the Internet of Things - IOT '18*, page 1–8. ACM Press, 2018.

[71] Hend Gedawy, Karim Habak, Khaled Harras, and Mounir Hamdi. An energy-aware iot femtocloud system. In *2018 IEEE International Conference on Edge Computing (EDGE)*, pages 58–65. IEEE, 2018.

[72] Andrew Gelman, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. *Bayesian data analysis: Third Edition*. CRC press, 2013.

[73] Javad Ghaderi, Sanjay Shakkottai, and Rayadurgam Srikant. Scheduling storms and streams in the cloud. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 1(4):1–28, 2016.

[74] A. Ghosh, K. A. Patil, and S. K. Vuppala. PLEMS: Plug Load Energy Management Solution for Enterprises. In *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, page 25–32, Mar 2013.

[75] Tuan Nguyen Gia, Mingzhe Jiang, Amir-Mohammad Rahmani, Tomi Westerlund, Pasi Liljeberg, and Hannu Tenhunen. Fog Computing in Healthcare Internet of Things: A Case Study on ECG Feature Extraction. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, page 356–363, Oct 2015.

[76] Morteza Golkarifard, Ji Yang, Zhanpeng Huang, Ali Movaghar, and Pan Hui. Dandelion: A unified code offloading system for wearable computing. *IEEE Transactions on Mobile Computing*, 18(3):546–559, 2018.

[77] Dean A Gratton. *The handbook of personal area networking technologies and protocols*. Cambridge University Press, 2013.

[78] Jan L Harrington. *Relational database design and implementation: clearly explained*. Morgan Kaufmann/Elsevier, Amsterdam ; Boston, 3rd ed.. edition, 2009.

[79] Lutz Heinemann and Guido Freckmann. CGM Versus FGM; or, Continuous Glucose Monitoring Is Not Flash Glucose Monitoring. *Journal of Diabetes Science and Technology*, 9(5), Sep 2015.

[80] Joseph Henson, Melanie J Davies, Danielle H Bodicoat, Charlotte L Edwardson, Jason MR Gill, David J Stensel, Keith Tolfrey, David W Dunstan, Kamlesh Khunti, and Thomas Yates. Breaking up prolonged sitting with standing or walking attenuates the postprandial metabolic response in postmenopausal women: a randomized acute study. *Diabetes care*, 39(1):130–138, 2016.

[81] Robin Heydon and Nick Hunn. Bluetooth low energy. *CSR Presentation, Bluetooth SIG https://www. bluetooth. org/DocMan/handlers/DownloadDoc. ashx*, 2012.

[82] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46, 2014.

[83] Jack Kenneth Holmes. *Spread spectrum systems for GNSS and wireless communications*. Artech House Norwood, 2007.

[84] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX annual technical conference*, volume 8, 2010.

[85] Ronny K Ibrahim, Eliathamby Ambikairajah, Branko G Celler, and Nigel H Lovell. Time-frequency based features for classification of walking patterns. In *2007 15th International Conference on Digital Signal Processing*, pages 187–190. IEEE, 2007.

[86] Antonio J. Jara, Dominique Genoud, and Yann Bocchi. Big data for smart cities with KNIME a real experience in the SmartSantander testbed. *Software: Practice and Experience*, 45(8):1145–1160, Aug 2015.

[87] 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 *East European Conference on Advances in Databases and Information Systems*, pages 192–206. Springer, 2014.

[88] Devki Nandan Jha, Peter Michalak, Zhenyu Wen, Paul Watson, and Rajiv Ranjan. Multi-objective deployment of data analysis operations in heterogeneous iot infrastructure. *IEEE Transactions on Industrial Informatics*, 2019.

[89] Mike Jipping. *Learn C with Pebble*. Gitbooks.io, 2016. published online https://pebble.gitbooks.io/learning-c-with-pebble/content/.

[90] Michael Jones, John Bradley, and Nat Sakimura. JSON Web Token (JWT). RFC 7519, May 2015.

[91] Haik Kalantarian, Costas Sideris, Bobak Mortazavi, Nabil Alshurafa, and Majid Sarrafzadeh. Dynamic computation offloading for low-power wearable health monitoring systems. *IEEE Transactions on Biomedical Engineering*, 64(3), 2017.

[92] Markad V Kamath, Mari Watanabe, and Adrian Upton. *Heart rate variability (HRV) signal analysis: clinical applications*. CRC Press, 2012.

[93] A. Kamilaris, F. Gao, F. X. Prenafeta-Boldu, and M. I. Ali. Agri-IoT: A semantic framework for Internet of Things-enabled smart farming applications. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, page 442–447, Dec 2016.

[94] David W Kammler. *A first course in Fourier analysis*. Cambridge University Press, 2007.

[95] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Cuckoo: a computation offloading framework for smartphones. In *International Conference on Mobile Computing, Applications, and Services*, pages 59–79. Springer, 2010.

[96] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.

[97] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *2012 Proceedings IEEE Infocom*, pages 945–953. IEEE, 2012.

[98] John Kruschke. *Doing Bayesian data analysis: A tutorial with R, JAGS, and Stan*. Academic Press, 2014.

[99] Petri Launiainen. *A Brief History of Everything Wireless: How Invisible Waves Have Changed the World*. Springer, 2018.

[100] Ilias Leontiadis, Christos Efstratiou, Cecilia Mascolo, and Jon Crowcroft. Senshare: transforming sensor networks into multi-application sensing infrastructures. In *European Conference on Wireless Sensor Networks*, pages 65–81. Springer, 2012.

[101] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[102] Lun Li. *Digital system verification a combined formal methods and simulation framework*. Synthesis lectures on digital circuits and systems. Morgan and Claypool Publishers, San Rafael, Calif. (1537 Fourth Street, San Rafael, CA 94901 USA), 2010.

[103] Yuanzhe Li and Shangguang Wang. An energy-aware edge server placement algorithm in mobile edge computing. In *2018 IEEE International Conference on Edge Computing (EDGE)*, pages 66–73. IEEE, 2018.

[104] Wei Lin, Haochuan Fan, Zhengping Qian, Junwei Xu, Jingren Zhou, Lidong Zhou, and Sen Yang. Streamscope: Continuous reliable distributed processing of big data streams. page 15. A/B,MicrosoftResearch.

[105] Alexander Linden and Jackie Fenn. Understanding Gartner's hype cycles. *Strategic Analysis Report Nº R-20-1971. Gartner, Inc*, 2003.

[106] Clemens Lombriser, Nagendra B. Bharatula, Daniel Roggen, and Gerhard Tröster. On-body activity recognition in a dynamic sensor network. In *Proceedings of the Second International Conference on Body Area Networks BodyNets*. ICST, 2007.

[107] Alvaro Lozano, Javier Caridad, Juan Francisco De Paz, Gabriel Villarrubia Gonzalez, and Javier Bajo. Smart waste collection system with low consumption lorawan nodes and route optimization. *Sensors*, 18(5):1465, 2018.

[108] David Luckham. *The power of events*, volume 204. Addison-Wesley Reading, 2002.

[109] David C. Luckham and Brian Frasca. Complex event processing in distributed systems. *Computer Systems Laboratory Technical Report CSL-TR-98-754. Stanford University, Stanford*, 28, 1998.

[110] S. Madden. From databases to big data. *IEEE Internet Computing*, 16(3):4–6, May 2012.

[111] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.", 2011.

[112] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[113] Peter Michalák, Sarah Heaps, Michael Trenell, and Paul Watson. Automating computational placement in IoT environments: doctoral symposium. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 434–437. ACM, 2016.

[114] Peter Michalák and Paul Watson. PATH2iot: A Holistic, Distributed Stream Processing System. In *Cloud Computing Technology and Science (CloudCom), 2017 IEEE International Conference on*, pages 25–32. IEEE, 2017.

[115] Konstantin Mikhaylov, Abdul Moiz, Ari Pouttu, José Manuel Martín Rapún, and Sergio Ayuso Gascon. LoRaWAN for wind turbine monitoring: Prototype and practical deployment. In *2018 10th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*. IEEE, 2018.

[116] M. Victoria Moreno, Fernando Terroso-Sáenz, Aurora González-Vidal, Mercedes Valdés-Vela, Antonio F. Skarmeta, Miguel A. Zamora, and Victor Chang. Applicability of big data techniques to smart cities deployments. *IEEE Transactions on Industrial Informatics*, 13(2):800–809, Apr 2017.

[117] Richard Mortier, Jianxin Zhao, Jon Crowcroft, Liang Wang, Qi Li, Hamed Haddadi, Yousef Amar, Andy Crabtree, James Colley, Tom Lodge, et al. Personal data management with the Databox: What's inside the box? In *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking*, pages 49–54, 2016.

[118] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, resource management, and approximation in a data stream management system. Conference on Innovative Data Systems Research, 2003.

[119] Mithun Mukherjee, Lei Shu, and Di Wang. Survey of fog computing: Fundamental, network applications, and research challenges. *IEEE Communications Surveys Tutorials*, 20(3):1826–1857, 2018.

[120] Taewoo Nam and Theresa A Pardo. Conceptualizing smart city with dimensions of technology, people, and institutions. In *Proceedings of the 12th annual international digital government research conference: digital government innovation in challenging times*, pages 282–291, 2011.

[121] Yucen Nan, Wei Li, Wei Bao, Flavia C Delicato, Paulo F Pires, and Albert Y Zomaya. Cost-effective processing for delay-sensitive applications in cloud of things systems. In *Network Computing and Applications (NCA), 2016 IEEE 15th International Symposium on*, pages 162–169. IEEE, 2016.

[122] Ebrahim Nemati, Konstantinos Sideris, Haik Kalantarian, and Majid Sarrafzadeh. A dynamic data source selection system for smartwatch platform. In *2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 5993–5996. IEEE, 2016.

[123] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[124] S. Y. Nikouei, Y. Chen, S. Song, R. Xu, B. Choi, and T. R. Faughnan. Real-Time Human Detection as an Edge Service Enabled by a Lightweight CNN. In *2018 IEEE International Conference on Edge Computing (EDGE)*, page 125–129, Jul 2018.

[125] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference)*, 2014.

[126] Apostolos Papageorgiou, Manuel Zahn, and Ernö Kovacs. Auto-configuration system and algorithms for big data-enabled internet-of-things platforms. In *2014 IEEE International Congress on Big Data*, pages 490–497. IEEE, 2014.

[127] Charith Perera, Chi Harold Liu, Srimal Jayawardena, and Min Chen. A survey on Internet of Things from industrial market perspective. *IEEE Access*, 2:1660–1679, 2014.

[128] Martyn Plummer et al. Jags: A program for analysis of bayesian graphical models using gibbs sampling. In *Proceedings of the 3rd international workshop on distributed statistical computing*, volume 124, pages 1–10. Vienna, Austria, 2003.

[129] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, et al. The kaldi speech recognition toolkit. In *IEEE 2011 workshop on automatic speech recognition and understanding*, number CONF. IEEE Signal Processing Society, 2011.

[130] Richard M. Pulsford, James Blackwell, Melvyn Hillsdon, and Katarina Kos. Intermittent walking, but not standing, improves postprandial insulin and glucose relative to sustained sitting: A randomised cross-over study in inactive middle-aged men. *Journal of Science and Medicine in Sport*, 20(3):278–283, Mar 2017.

[131] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2020.

[132] Zhengyong Ren, Chaojian Chen, Kejia Pan, Thomas Kalscheuer, Hansruedi Maurer, and Jingtian Tang. Gravity anomalies of arbitrary 3d polyhedral bodies with horizontal and vertical mass contrasts. *Surveys in geophysics*, 38(2):479–502, 2017.

[133] Brecht Reynders, Wannes Meert, and Sofie Pollin. Range and coexistence analysis of long range unlicensed communication. In *2016 23rd International Conference on Telecommunications (ICT)*, pages 1–6. IEEE, 2016.

[134] Lauren Roberts, Peter Michalák, Sarah Heaps, Michael Trenell, Darren Wilkinson, and Paul Watson. Automating the Placement of Time Series Models for IoT Healthcare Applications. In *2018 IEEE 14th International Conference on e-Science (e-Science)*, pages 290–291. IEEE, 2018.

[135] Ralf Rudersdorfer. *Radio receiver technology: Principles, architectures and applications*. John Wiley & Sons, 2013.

[136] Mahadev Satyanarayanan, Pieter Simoens, Yu Xiao, Padmanabhan Pillai, Zhuo Chen, Kiryong Ha, Wenlu Hu, and Brandon Amos. Edge analytics in the internet of things. *IEEE Pervasive Computing*, 14(2), 2015.

[137] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016.

[138] Ritesh Kumar Singh, Michiel Aernouts, Mats De Meyer, Maarten Weyn, and Rafael Berkvens. Leveraging LoRaWAN Technology for Precision Agriculture in Greenhouses. *Sensors*, 20(7):1827, 2020.

[139] Eugene Siow, Thanassis Tiropanis, and Wendy Hall. Analytics for the internet of things: A survey. *ACM Computing Surveys (CSUR)*, 51(4):1–36, 2018.

[140] Jim Smith, Paul Watson, Anastasios Gounaris, Norman W Paton, Alvaro AA Fernandes, and Rizos Sakellariou. Distributed query processing on the grid. *The International Journal of High Performance Computing Applications*, 17(4), 2003.

[141] Bruce Snyder, Dejan Bosnanac, and Rob Davies. *ActiveMQ in action*, volume 47. Manning Greenwich Conn., 2011.

[142] Dag Stranneby and William Walker. *Digital Signal Processing and Applications*. Elsevier Science & Technology, 2004.

[143] J. Sueur, T. Aubin, and C. Simonis. Seewave: a free modular tool for sound analysis and synthesis. *Bioacoustics*, 18:213–226, 2008.

[144] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. Siddhi: A second look at complex event processing architectures. In *Proceedings of the 2011 ACM workshop on Gateway computing environments*, pages 43–50, 2011.

[145] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Florian Dudouet, and Andrew Edmonds. An architecture for self-managing microservices. In *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, AIMC '15, pages 19–24, New York, NY, USA, 2015. ACM.

[146] C. Tomasi, E. De Giovannini, N. Locallo, F. Favaron, C. Lain, P. Pianalto, V. Terrazzi, M. Pistacchi, and M. Rabuffetti. 7-Days actigraphy in patients with Parkinson disease: a 2-years follow-up. *Gait & Posture*, 74, Sep 2019.

[147] Vincent T. van Hees, Séverine Sabia, Kirstie N. Anderson, Sarah J. Denton, James Oliver, Michael Catt, Jessica G. Abell, Mika Kivimäki, Michael I. Trenell, and Archana Singh-Manoux. A novel, open access method to assess sleep duration using a wrist-worn accelerometer. *PLOS ONE*, 10(11):e0142533, Nov 2015.

[148] Blesson Varghese, Philipp Leitner, Suprio Ray, Kyle Chard, Adam Barker, Yehia Elkhatib, Herry Herry, Cheol-Ho Hong, Jeremy Singer, Fung Po Tso, and et al. Cloud futurology. *Computer*, 52(9):68–77, Sep 2019.

[149] Blesson Varghese, Nan Wang, Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S Nikolopoulos. Challenges and opportunities in edge computing. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 20–26. IEEE, 2016.

[150] Michael Vögler, Johannes Schleicher, Christian Inzinger, Stefan Nastic, Sanjin Sehic, and Schahram Dustdar. Leonore–large-scale provisioning of resource-constrained iot deployments. In *2015 IEEE Symposium on Service-Oriented System Engineering*, pages 78–87. IEEE, 2015.

[151] Paul Watson. A multi-level security model for partitioning workflows over federated clouds. *Journal of Cloud Computing: Advances, Systems and Applications*, 1(1):15, 2012.

[152] Jim Webber. A programmatic introduction to neo4j. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 217–218, 2012.

[153] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016.

[154] Hadley Wickham, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D'Agostino McGowan, Romain François, Garrett Grolemund, Alex Hayes, Lionel Henry, Jim Hester, Max Kuhn, Thomas Lin Pedersen, Evan Miller, Stephan Milton Bache, Kirill Müller, Jeroen Ooms, David Robinson, Dana Paige Seidel, Vitalie Spinu, Kohske Takahashi, Davis Vaughan, Claus Wilke, Kara Woo, and Hiroaki Yutani. Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686, 2019.

[155] Rebecca C Wilson, Oliver W Butters, Demetris Avraam, James Baker, Jonathan A Tedds, Andrew Turner, Madeleine Murtagh, and Paul R Burton. DataSHIELD–new directions and dimensions. *Data Science Journal*, 16(21):1–21, 2017.

[156] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and rich analytics at scale. In *ACM SIGMOD International Conference on Management of data*. ACM, 2013.

[157] Shusen Yang. Iot stream processing and analytics in the fog. *IEEE Communications Magazine*, 55(8):21–27, 2017.

[158] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing*, pages 10–10. USENIX Association, 2012.

[159] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Priter: a distributed framework for prioritized iterative computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 13. ACM, 2011.

[160] Neil Zhao. Full-featured pedometer design realized with 3-axis digital accelerometer. *Analog Dialogue*, 44(06), 2010.

[161] Yuchen Zhao, Hamed Haddadi, Severin Skillman, Shirin Enshaeifar, and Payam Barnaghi. Privacy-preserving activity and health monitoring on databox. In *Proceedings of the Third ACM International Workshop on Edge Systems, Analytics and Networking*, pages 49–54, 2020.

[162] Jingren Zhou, Per-Ake Larson, and Ronnie Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE, 2010.