FlexIoT: Flexible IoT Application Development With Stream Processing Engines

by

Pavel Golikov

A thesis submitted in conformity with the requirements
for the degree of Master of Science

Department of Computer Science
University of Toronto

FlexIoT: Flexible IoT Application Development With Stream Processing Engines

Pavel Golikov
Master of Science

Department of Computer Science
University of Toronto
2022

Abstract

As reliability and speed of wireless technologies continue developing, a multitude of Internet of Things use cases that were previously infeasible start becoming possible. These use cases employ various sensors that continuously collect data and transfer the data to the cloud. However, there is a class of applications for which it is infeasible to send all raw data to the cloud - some of it has to be pre-processed on the edge. Such applications necessitate the partitioning of the computational workload between the edge and the cloud. Determining and implementing such partition is an error-prone process with high engineering effort involved.

This work presents FlexIoT, a unified approach for development of applications that need to be partitioned between the edge and the cloud. The key idea is to determine the edge-cloud partition and generate the edge, the cloud, and the edge-cloud communication components automatically. We propose to express the workload as a streaming query, which allows to clearly delineate the stages of computation and allows to determine the edge-cloud partition automatically.

Compared to existing approach of developing such applications manually, FlexIoT allows the developer to save time and engineering effort. Our results show that we can reduce the lines of code written by an average of 30% at a modest overhead (average 2.5%) compared to manual implementation. In addition, we argue that because the partition is determined automatically, the developer does not have to perform the manual effort of determining the partition. We have performed a sensitivity study for our representative applications and it shows that partially offloading such applications to the edge can reduce cloud compute costs by up to 5X and cloud network costs by up to 1000X.

Acknowledgements

I want to thank Anand Jayarajan for constructive discussion about Stream Processing Engines. I also want to thank my supervisor, Professor Gennady Pekhimenko for his direction and patience during completion of this project and for making my masters studies a very rewarding and enjoyable experience.

Contents

Chapter 1

Introduction

Fuelled by development of 5G [1], recent years have seen rapid growth of Internet of Things (IoT) [60] applications. Most IoT applications rely on sensors that collect data and send it to the cloud[110] for processing and analysis. As such, most IoT applications consist of three parts: **sensor**, **communication**, and **cloud** components. Sensors collect the data, communication component sends the raw data to the cloud, and cloud component aggregates it and performs deeper analysis on data from multiple sensors. There is a class of applications in industrial [86] and medical [41] fields that rely on fast-collecting sensors [29, 44, 45] and for these applications it is not feasible to send raw data from the sensor to the cloud because of fast sensor collection frequency. For such applications, it is necessary to pre-process and aggregate the raw data on or near the sensor and send the aggregated results to the cloud. This approach is usually referred to as "edge computing" [59]. In edge computing, part of the workload is offloaded from the cloud to the edge so that raw data can be pre-processed and aggregated, thus reducing the frequency of messages from the edge to the cloud. Edge devices that we are targeting in this work are small and resource-constrained micro-controllers [20] which can be paired with IoT sensors.

Developing applications that require partitioning the workload between edge and cloud come with several fundamental challenges:

- Developers have to determine which part of the workload to offload to the edge, which is an error prone process that requires high engineering effort.
- Each edge-cloud partition requires manually implementing three different pieces of software, typically, in two different programming languages. Edge component [24] is usually firmware for a micro-controller written in low level programming language (C/C++ [31]) while cloud component [16, 10] is usually written in a high level programming language (Python [94], Java [62], Scala [97]) to run on the cloud. Edge and cloud require very different programming expertise and a developer will have to deal with different technologies (networks, embedded devices, mobile, cloud).
- Every time a developer wants to try a different partition, they have to change edge, cloud, and communication components. Partitioning the workload at a different point means moving part of the computation from the cloud to the edge or vice versa, which means the developer needs to

convert from a high level language to low level language or vise versa, which is not a trivial task and has to be performed manually.

To address these challenges, we propose FlexIoT, a framework which takes an integrated development approach to the problem. Developer uses a Stream Processing Engine (SPE) [104] to write a single streaming query in our intermediate representation. We have developed an intermediate representation that allows to **(1)** write a single high level streaming query as if it ran entirely on the cloud; **(2)** automatically identify the edge and the cloud components of the query, automatically test and determine which part of the workload can be offloaded to the edge device; and **(3)** keep track of all the required information to automatically generate the communication component for all possible edge-cloud splits. The query is automatically split between the edge and the cloud components based on different optimization criteria and communication component is inserted between the two. FlexIoT makes use of source code generation techniques to convert the high-level representation into source code for the edge and the cloud components. Since micro-controllers are very constrained in terms of compute resources, it is important not to offload too large of a workload onto a micro-controller, which can potentially produce invalid results. Given a micro-controller, FlexIoT automatically tests different edge-cloud splits to determine what part of the workload can be offloaded to the edge.

We have compared FlexIoT against the manual implementation and found that we reduce the amount of code written by approximately 30% and significantly reduced development time while incurring a maximum and average latency increase of 7.45% and 2.92% respectively.

In summary, this work makes following contributions:

- We perform a thorough study of an important class of IoT applications and identify challenges in development of this class of applications.
- Motivated by the aforementioned challenges of manual implementation, we present FlexIoT, an integrated system for development of an important class of IoT applications with signal processing pipelines. FlexIoT allows users to develop these challenging IoT applications with minimal programmer effort by automating several development stages and allowing developers to focus on important parts of the application.
- We perform a thorough evaluation of our system on live human data with real sensors and find that FlexIoT is able to automatically determine the edge-cloud split point and produce edge, communication, and cloud components while significantly reducing engineering effort (both lines of code written and development time). Moreover, FlexIoT does this with a minimal increase in latency compared to manual implementation.

The rest of this manuscript is structured as follows. Sections 2 and 3 will provide the background and describe the problem, Section 4 will list observations that drove our approach, Section 5 will describe the implementation, Sections 6 and 7 describe our methodology and evaluation, Section 8 discusses important related works and finally we conclude with Section 9.

Chapter 2

Background

In this section, we will give the necessary background on IoT sensors and micro-controllers, followed by discussion of cloud and edge and how they are related to IoT. After that we will briefly describe the applications we are targeting, including the use cases we have implemented. We will conclude with the discussion of Stream Processing Engines (SPEs).

2.1 5G and IoT

Advent of 5G [1] has enabled faster, more stable and more secure connectivity, which made it possible to connect very large number of devices to the internet. This has enabled rapid development of the Internet of Things (IoT) [60], where network-capable sensors are attached to various objects (from household refrigerators [70] to large industrial robots [34]) or people (medical use cases [2, 52]) in order to continuously collect diagnostic data, which is then sent to the cloud servers for processing and analysis.

2.2 Sensors and Microcontrollers

Sensors used for IoT applications range from fast-collecting accelerometers [75] with sampling frequency on the order of KHz used to monitor industrial machines [29] to air quality sensors in the cities that report readings once per minute [77]. Table 2.1 shows example of several such sensors, typical output frequency used, field, and applications[1].

These sensors can be paired with network capable microcontrollers into what is called "Smart Sensors", which can collect the data from the source (human being or industrial machine) and send the data to the cloud for analysis. Devices that we focus on are on the scale of Arduino [19] micro-controllers. We focus on this class of devices because they are the small, cheap, can be deployed en masse, and yet are still capable of processing raw signals from the types of sensors that are deployed for IoT applications and sending the results to the cloud. Because of their low cost, small size, and network capabilities, such micro-controllers would be primary candidates to be paired with IoT sensors. These devices normally have simple in-order pipelines with memory on the order of tens

[1]It is important to note that raw data produced by these sensors needs to be filtered and processed using signal processing techniques before meaningful information can be extracted.

Sensor	Typical sampling Rate(Hz)	Field	Application
ECG sensor [102]	250 - 2000	Medical	Electrocardiogram [44]
EMG sensor [84]	250 - 2000	Medical	Electromyography [45]
EEG sensor [43]	100 - 20,000	Medical	Electroencephalogram [95]
Accelerometer [75]	100 - 40,000	Industrial	Predictive Maintenance [29]
Proximity sensor [53]	2 - 42,000	Industrial	Machine Control / Safety [91]
Pressure sensor [51]	100 - 1000	Industrial	Machine Monitoring [100]

Table 2.1: Examples of Typical Sensors that are used for IoT and their respective applications.

or hundreds of KB [21, 22, 23, 92, 106, 87]. Such devices are normally programmed using a low level programming language like C\C++ [31]. In order to perform deeper analysis and extract actionable insights from *across* multiple sensors, that data needs to be analyzed and stored in some central location. In the current IoT architecture, this is happening at the cloud.

2.3 The Cloud

The term *cloud* commonly refers to datacenter servers that provide compute and data storage services to customers. Among major cloud providers are Alibaba Cloud [8], Amazon Web Services [11], Microsoft Azure [81], Google Cloud Platform (GCP) [48], IBM Cloud [54], and Oracle Cloud [88]. These organizations provide compute services to their users who usually pay on time-basis (e.g., hourly, daily, monthly) for the use of provider's cloud compute and storage instances. Cloud computing services are popular because cloud providers provide users with Infrastructure, Tools and Software that allow users to skip several stages of application development.

In the context of IoT, cloud is a natural place for accumulation and analysis of sensor data [13, 48, 89]. cloud providers virtualize seemingly infinite compute resources into a highly flexible, efficient, and scalable service. However, with all of its benefits, cloud computing has one major shortcoming - since the machines that perform computing are located in datacenters, data has to be delivered to these machines, which introduces additional *costs* and *latency*. In this work, we will demonstrate several applications that, due to their very nature, are not well suited to be deployed in a classical cloud computing framework. These applications would require a different approach - one that is commonly referred to as *edge computing*.

2.4 The Edge

Edge computing refers to the concept of performing a computation closer to the source of the data or the user. If cloud normally refers to datacenter servers, edge is a much more heterogeneous term which can mean much more diverse hardware resources. It can mean anything from tiny Arduino microcontrollers [20] to a cloudlet [76, 109] (micro-datacenter). The purpose of edge computing is to bring compute resources closer to the sensor or user, which is done in order to support latency-sensitive applications and reduce costs associated with moving data to the cloud. Part of the computational workload of the application is performed on an edge device and the result is then sent to the cloud. This allows to perform partial processing and aggregation of raw results closer to the source, which reduces the amount of compute that has to be done by the cloud, thus reducing cloud compute costs. Since the raw data is aggregated, this results in less frequent messages from the edge to the cloud, which reduces pressure on network infrastructure and relaxes bandwidth requirements, thus reducing network costs. There is a large set of Industrial and Medical applications which employ fast collecting sensors [99, 98, 44, 45, 95, 29, 91, 100] which can benefit from edge computing approach. Table 2.1 shows several examples of such applications. Applications listed in the table are only a subset of a large set of similar applications that require fast-collecting sensors. One important observation that we can make is that medical and industrial sensors collect at high frequencies because physical phenomena they are trying to detect *require* fast collection frequencies.

2.5 Applications

In this work, we focus on IoT applications that make use of fast-collecting sensors. Such applications are usually found in health [41] and industrial [86] IoT use cases. Several examples of such applications are given in the last column of Table 2.1. These applications make use of high-frequency (at least 100 Hz) sensors to collect raw data and require signal processing techniques to filter and pre-process the results of each individual sensor before meaningful information can be extracted. We select three use cases that are representative of these types of applications:

1. Electrocardiogram (ECG) [44] is one of the simplest and fastest tests used to evaluate the heart. Electrodes (small, plastic patches that stick to the skin) are placed at certain spots on the chest, arms, and legs. The electrodes are connected to an ECG machine by lead wires. The electrical activity of the heart is then measured and analyzed.

2. Surface Electromyography (EMG) [45] measures muscle response or electrical activity in response to a nerve's stimulation of the muscle. The test is used to help detect neuromuscular abnormalities. During the test, electrodes are attached to the skin near the muscle and the electrical activity is picked up by the electrodes and analyzed.

3. Bearing Health Measurement (BHM) [29]. Accelerometer is attached to a bearing in a factory robot in order to monitor the health of said bearing to determine if the bearing needs (will soon need) replacement. Accelerometer records the vibrations as time-series data, which is then analyzed and health of a bearing is assessed.

Applications such as these usually employ multiple sensors whose results are still aggregated on the cloud. For example, a doctor would have multiple patients with wearable ECG sensors all of whose data is being collected and sent to the cloud. A deeper analysis can be run on the data from across sensors on the cloud, for example computing the average heart rate across the patients. This data from multiple sensors can be thought of as a data stream. There are many tools to analyze data streams, but perhaps the most well known are Stream Processing Engines (SPEs).

2.6 Stream Processing Engines

The idea of stream processing originated almost 30 years ago [32]. Early stream processing engines (SPEs) [5, 17, 107, 50, 27] were designed as extensions to relational databases by modeling stream processing as incremental query execution over append-only tables [107]. Over the years, streaming engines have evolved considerably [7, 6, 26, 82, 73, 63, 80, 79, 93, 108] and have found adoption in a wide variety of fields, including health care [63], social media analysis [69], investment [105], and fraud detection [40]. Many of these applications process data at a high rate and demand latency and throughput requirements - something traditional data bases are not designed for. In addition, traditional data bases are not designed to process live data.

Sensors such as the ones in Table 2.1 produce readings at regular intervals (dictated by sensor's sampling rate). These readings can be thought of as a data stream and as such, SPEs are a natural choice for processing such data.

$$CF = max(|x_i|) \Big/ \sqrt[2]{\frac{1}{N}\sum_{n=1}^{N} x_i^2}$$

(a) Crest Factor formula.

$$\begin{tikzcd}
 & TwinSubq(512) \arrow[d] & \\
 & Parallel(2) \arrow[dl] \arrow[dr] & \\
x \to |x| \arrow[d] & & x \to x^2 \arrow[d] \\
Max() \arrow[dddr] & & Sum() \arrow[d] \\
 & & x \to x \,/\, 512 \arrow[d] \\
 & & x \to \sqrt[2]{x} \arrow[dl] \\
 & TWinFun(2, (x, y) \to x \,/\, y) &
\end{tikzcd}$$

(b) Implementation of Crest Factor using an SPE.

Figure 2.1: Formula for Crest Factor and it's implementation using a SPE.

SPEs usually provide users with a set of operators using which users build streaming queries. Since our work concerns itself with SPEs and not with databases, we will, unless otherwise stated, use queries to mean streaming queries. Streaming query consists of a series of operators, each of which can have parameters, input functions, or even subqueries. Each operator of the query receives an element (or several elements) of the stream as input, performs its operation on the input, and outputs the result to the next operator. We demonstrate how a mathematical formula for Crest Factor [36], a statistical measure of a signal that is often used in signal analysis, can be implemented using an SPE. Part (a) of Figure 2.1 shows the formula for the Crest Factor and part (b) shows how it would be implemented using an SPE. First, the input signal is windowed using a *Tumbling Window Subquery* operator with window size of 512 events, then the stream is split into two separate streams using *Parallel* operator (this operator passes its input to its subqueries). Left side of the stream computes the maximum of the absolute values of stream events. Right side of the stream computes the expression for the square root of the formula from part (a). Then two streams are joined using a *Tumbling Window Function* operator which takes two elements of the stream (first containing result of left side and second containing the result of the right side) and divides the first element by the second one.

SPEs normally provide a number of operators out of the box, as well as allow users to build their own custom operators. SPEs have the advantage of hiding some of the complexity of the overall computation from the user by implementing common computational paradigms for the user in the operators themselves and allowing the user to concentrate their effort on critical parts of the computation. For instance, in Figure 2.1 example, user will not have to write the code to window the stream with tumbling windows of size 512 as it is already implemented by the SPE's operator.

Chapter 3

Motivation

In this section we will demonstrate that there is a class of applications with high-frequency sensors which can not be placed entirely on the edge or the cloud. These applications require the programmer to partition the workload between the edge and the cloud. We will argue that determining and performing that partition forces the programmer to apply high engineering effort and potentially perform redundant work.

3.1 Why Not Doing Everything at the Edge?

There are several reasons why it is difficult and sometimes even impossible to completely move workloads to the edge:

- While there is a very large variety of edge devices, they are usually small, cheap, and deployed en masse. As such, these devices are often very constrained in terms of power, compute capabilities, and memory. This means that it is possible that these devices will not have enough memory to store the working set of the entire application or their CPU is not powerful enough to perform all required computations to obtain the final result within a required time interval. IoT applications with signal processing pipelines pre-process and filter raw sensor data, which needs to be collected at strict time intervals. If an edge device computes intermediate results between sensor collections too slow and violates these strict time intervals, this could produce incorrect final results. In addition to memory and computational constraints, it is often desirable to keep results from sensors over long time periods for logging and possible offline analysis, which requires significant amount of storage, something edge devices do not have. For example, a doctor might want to keep patient results over months or years for long term analysis, which is impossible on a small edge micro-controller due to memory constraints. Although it is possible to analyze (and to some degree) store this data on doctor's laptop, it would require the doctor to be able to implement and maintain everything required for the operation of what is essentially a mini data center. This demands from said doctor the skills they do not normally have.
- In order to perform analysis over data from multiple sensors one has to collect these sensor results into a single location. For example, in preventative maintenance application, one might want to know the average health of factory machines across factory floor or across multiple factories. This

Application	Sensor Freq. (Hz)	Monthly Messages (M/sensor)	Monthly Price ($/sensor)	Fleet Size (K)	Monthly Price (M$/fleet)
BHM	3000	7500	5200	1000	5200
ECG	1000	2500	1800	500	900
EMG	500	1250	900	250	218

Table 3.1: Monthly AWS network message costs without edge compute.

requires that data from multiple sensors is brought into a central location and this aggregated analysis is performed there.

3.2 Why Not Doing Everything on the Cloud?

While it is possible to send all sensor data to the cloud and perform computation there, there are applications for which it is prohibitively expensive. In this work, we demonstrate a class of applications for which it is infeasible to send raw data to the cloud. We implement three use cases, namely electrocardiogram (ECG), electromyography(EMG), and Bearing Health Measurement (BHM), which we treat as representative of the class of applications we focus on. All three applications require fast collecting sensors and signal processing pipelines to analyze raw signal. These use cases were described in Section 2.

If we do not perform processing at the edge for these applications, we have to send raw data from sensor to the cloud. Table 3.1 shows the cost to do that with the AWS [11] IoT Core in the US East Region and cheapest (MQTT and HTTP) pricing of $0.70 per million messages [13]. We are using 3 KHz (slower) collection speed for BHM for the calculations in the table.

Last column of the table indicates monthly bill for a user of IoT services without pre-processing or aggregating data on the edge. We conclude that for our three applications (and those with comparable sensor collection frequencies) it would be infeasible to send all the data to the cloud and perform all analysis there. Therefore, we conclude that it is necessary to offload the workload, at least partially, to the edge.

3.3 The Split

In order to continue the discussion of edge-cloud split workloads, it is useful to introduce some definitions. When we obtain the reading from each sensor, we call that component of the workload **sensor reading**. As was noted earlier, in order to extract meaningful information from sensors' data, it is necessary to perform signal processing techniques on raw data coming out of the sensors. We call that part of the workload **individual sensor component**. We define the **split point** as the point at which the workload is split between edge and cloud. It is often desirable to perform deeper analysis across sensors, for example, if a factory manager wants to know the average health of industrial machines across their factory floor or across multiple factories, one would need to aggregate the data from multiple sensors into a central location. We call the point in the computation where this is done an **aggregation point** and processing of data aggregated across sensors **aggregated processing**.

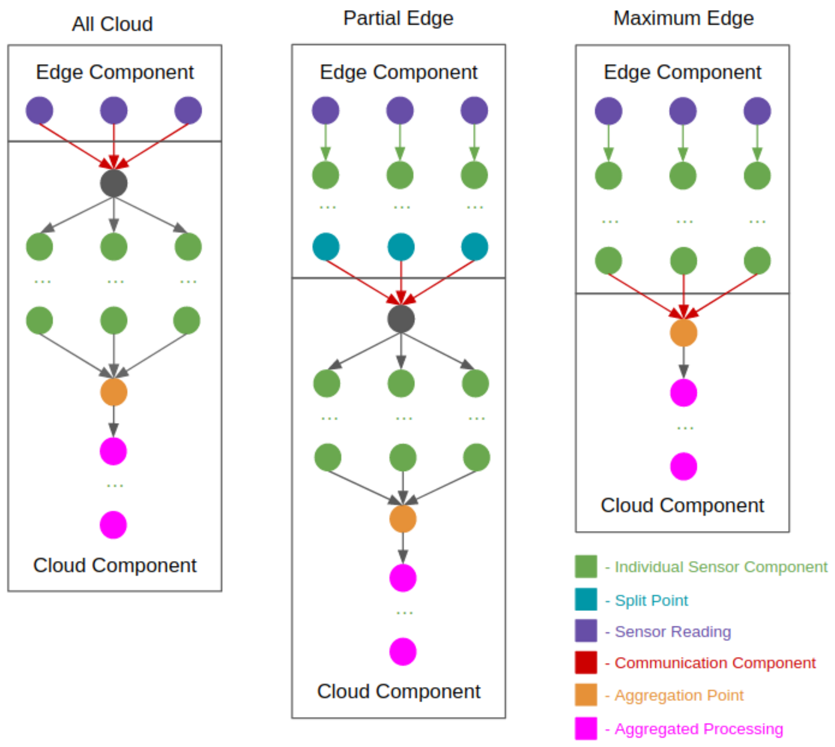


Figure 3.1: Scenarios for IoT Workloads.

Figure 3.1 illustrates definitions discussed above and demonstrates three possible scenarios for an IoT application (we are assuming a simple case of three sensors):

Everything in the Cloud. In this scenario there is no processing at the edge and everything is processed in the cloud. Edge component consists of reading sensor value and sending it to the cloud. Cloud receives the raw sensor readings and processes each individual sensor component separately, then aggregates the results and processes the rest of the workload.

Partially at the Edge. In this scenario part of the individual sensor component is placed on the edge device, but since edge device can only process part of that workload, it is divided at the *split point*. Everything up to the split point is processed at the edge and then the rest is sent to the cloud. What happens at the cloud is similar to the *Everything at the Cloud* case except with less processing on the cloud and potentially fewer network transmissions between the edge and the cloud.

Maximum at the Edge. In this scenario the entire individual sensor component is processed at the edge and the result of it is then sent to the cloud, where results are aggregated and the rest of the workload is processed. This is a special (and the most favourable) case of a *Partially at the Edge* scenario.

When a developer implements an application that is supposed to run partially on a small microcontroller on the edge and partially on the cloud, they come across the following challenges:

- Split point needs to be determined for each device and each application. Different devices have different compute capabilities and different applications have different compute requirements, which potentially implies different split points.
- Different user objectives dictate different split points. For example, if the user wants to reduce

the pressure on the network infrastructure or available network is not very fast, they would likely prefer to split the workload at a point that would result in the lowest transmission rate. On the other hand, if it is desirable to relieve the pressure from cloud resources, one would want to put the maximum number of operators on the edge.

- Split point is sensitive to changes in the application. For example, in Bearing Health Measurement, we need to filter the raw signal coming from the sensor using filters designed for sensor data collected at a certain frequency. If the collection frequency has been increased, it is not guaranteed that current split point satisfies sensor frequency collection requirements anymore. If the user processes sensor events between collection of said events, it is possible that the old split will not process events fast enough to satisfy collection frequency, i.e. we will not be able to collect the next sensor reading in time. This will violate strict collection frequency requirements in place for signal processing filter and could result in violation of result correctness. On the other hand, if the frequency has been reduced, more of the workload can now be placed on the edge, allowing for more possible splits with potentially better satisfaction of user objectives.

- Determining where to split the workload and implementing the split is an error-prone process with high engineering effort involved. As the input data makes its way through the application, it mutates into different data types. This means that at different split points, the data type sent to the cloud via communication component changes. For different split points we also have to change the amount of work the edge and cloud components perform. When determining the split point, the user will likely have to experiment with different split points, which means that they will have to manually re-implement all three components every time they attempt to split the workload at different points. These three components usually need to be written in at least two different programming languages. Edge and communication components are normally written using either C or C++ (because edge component is usually firmware for the small microcontroller which is not able to run high level languages efficiently or at all), while cloud component is normally written in a high-level programming language like Java, Python, or C#. If one needs to rebalance the workload between edge and cloud or vice versa, one would have to convert part of the workload written in low level programming language and write it in a high level programming language, which needs to be performed manually. Having to write 3 different pieces of programming in different programming languages requires high engineering effort with skills from different areas of computer science.

3.4 How is it done now?

Naturally, there are implementations of ECG [64], EMG [39], and similar applications which are implemented by performing part of the analysis at the edge (normally using a smartphone as the edge device), with the rest of the workload (aggregation and analysis across sensors) on the cloud. However, to our best knowledge these use cases (and similar ones with fast-collecting sensors) are implemented manually on a case-by-case basis for specific devices. This normally requires the user to write the part that will be executed on the edge (**edge component**), the part that will executed on the cloud (**cloud component**), and the part that sends data from the edge to the cloud and receives the data on the cloud (**communication component**) separately.

Today, if one wants to implement an application with edge and cloud components, one faces the challenges listed above and is forced to implement the split manually and to experiment with different split points for different objectives, which is very high engineering effort.

Chapter 4

Key Observations and Ideas

Our approach to IoT application development is motivated by four key observations. This section will describe those observations and outline key ideas behind our approach.

4.1 Key Observations

Observation 1: As we discussed before, determination of the split point of the application for different micro-controllers and different applications is currently performed by hand, leading to high engineering effort.

Key Idea 1: First observation drives our first key idea: edge-cloud split needs to be determined automatically.

Observation 2: When we aggregate the results of each individual sensor component across sensors for deeper analysis (i.e. past the aggregation point in Figure 3.1), the result of such aggregation is a data stream where events have sensor IDs (for example, to be able to pin-point outliers during deeper analysis). Such data stream is a natural workload for an SPE. We thus conclude that our workload past the aggregation point is likely to be processed with an SPE, which allows to program the desired application in a relatively high level interface and minimize programmer effort.

Observation 3: As was mentioned before, in order to extract usable information from raw sensor data, raw data needs to be filtered and pre-processed. This filtering and preprocesing is the part of the workload that can be moved to the micro-controller paired with the sensor because it is performed on the raw data of each individual sensor. It is denoted by individual sensor component in Figure 3.1. These individual sensor components consist of mathematical operations on the raw sensor signal. We observe that these individual sensor components can also be naturally implemented using an SPE.

Observation 4: Due to their nature, SPEs allow us to delineate workloads into clear steps. We observe that we can treat each SPE operator as a possible split point.

Key Idea 2: Observations 2, 3, and 4 drive our second key idea: for our class of applications (ones that are split between the edge and the cloud), we can express the individual sensor signal processing pipeline **and** the cross-sensor aggregation in a single streaming query, which will allow us to identify possible split points.

4.2 FlexIoT Overview

Based on the above ideas, we propose FlexIoT, a new approach which simplifies the development of IoT applications that have to be split between the edge and the cloud. FlexIoT allows the user to write their workload as a single streaming query using our lightweight SPE. User writes the streaming query as if there is **no** edge component and indicates the edge device they intend to use.[1] Since the query is written as a sequence of operators, this allows us to delineate different stages of computation and identify possible split points for the edge component. FlexIoT parses the query, identifies individual sensor component using a special operator, and automatically determines the split point given user objectives. After the split point has been identified, the edge, the cloud, and the communication components are generated.

Our high-level intermediate representation allows the user to avoid writing separate edge, cloud, and network components. Having the workload represented as a streaming query allows to automatically determine possible split points. Generating C++ Arduino Sketch provides an extra level of manipulation of edge, cloud, and network components when determining possible split points and testing for strict latency requirements. This makes it possible to split the workload automatically and with minimal programmer effort.

FlexIoT compiles and tests edge components according to one of two user-indicated objectives:

- **Maximize Edge Compute**. For this objective, FlexIoT attempts to put as many operators on the edge device as possible.
- **Minimize Network Transmissions**. For this objective, FlexIoT attempts to split the query in such a way as to minimize the number of edge-cloud transmission per second.

We have identified these two objectives because they correspond to two major components of operating costs for a potential provider of IoT services: cloud compute costs and communication costs. Maximizing edge compute will minimize cloud compute, which will minimize the bill from using cloud server machines. Minimizing network transmissions will minimize the bill associated with delivering data to the cloud and metadata overhead associated with transmissions.

[1]We currently support Arduino framework compatible devices.

Chapter 5

Implementation

FlexIoT consists of 2 major components. Stream Processing Engine and Compiler. Compiler is written in Python programming language [94], while the SPE is written in C++ [31]. We have implemented our own SPE because to our best knowledge, there is no SPE designed for Arduino-scale microcontrollers. These microcontrollers are very constrained in terms of memory and compute power, so the SPE has to be very lightweight and written with these constraints in mind. Compiler allows us to automatically generate the edge, the cloud, and the communication components, and test against user-defined latency requirements.

5.1 Streaming Engine

We have implemented a lightweight and highly customizable streaming engine. Each operator is written as a C++ Object with a constructor and 4 functions: *reset*, *connect*, *receive*, and *end*. This format for operators was inspired by StreamQL [73] SPE. *Reset* function resets the variables required for operator functionality. *Connect* is responsible for connecting current operator with the next operator. *Receive* receives an event from previous operator, processes said event and calls *receive* function of the next operator. *End* function receives end of stream marker from previous operator, processes it, and calls *end* function of the next operator. Operators are chained together using *Pipe* operators, which are generated automatically for each required size during code generation phase.

This simple and lightweight SPE allows us to break the computation into clear steps using its operators while introducing only negligible (as we will demonstrate in Section 7) increase in latency. We also avoid dynamic memory allocation as much as possible in our provided operators to optimize for performance. Only operators that are intended for use on the cloud use dynamic memory allocation. All operators are templated and require the user to indicate the input and output types of the operator, which is a common practice in SPEs [73, 82]. If the output of the operator is an array, FlexIoT requires that users indicate the size of the array in template arguments, which is necessary for communication component.

$$\begin{tikzcd}
\textit{Scatter}\ \{\text{Individual Sensor Component}\} \arrow[d] \\
\text{Timestamp assignment} \arrow[d] \\
\text{Join} \arrow[d] \\
\text{Aggregated Processing Operators}
\end{tikzcd}$$

Figure 5.1: Structure of the input Query.

5.2 Inputs

FlexIoT accepts as inputs the following:

- **Query.** Query description using our SPE.
- **Custom Sources file.** File with any custom constructs that are used in the query.
- **Network file.** File with network information that will be used in network component creation.
- **Default Arduino sketch.** File where user shows how to obtain a sensor event from the sensor.

The overall structure of the input query is shown in Figure 5.1. The first operator of the query is a special *Scatter* operator. Its subquery (denoted by curly braces) is the individual sensor component, which is the subquery whose operators FlexIoT will try to offload to the edge device. The output of *Scatter* is the Id-Value struct that is created automatically and is the data type that cloud component is supposed to receive from the edge component. Depending on where the split point is, Scatter operator automatically adjusts the Id-Value struct to accomodate the last operator of the edge component. Following is the timestamp assignment to the Id-Value struct and following that the streams are Joined and processed in Aggregated Processing component.

5.3 Parsing and Edge Component Generation.

FlexIoT parses the input query's operators into Python objects and identifies operators that can be offloaded to the edge using *Scatter* operator. FlexIoT then isolates the subquery of the Scatter operator and determines all possible split points. After identifying all split points, FlexIoT generates C++ source code for all possible edge components by incrementing the number of operators on the edge by one. Each edge component has communication component in the form of TCP\IP client code, which will communicate with TCP\IP server code of cloud component. Arduino programs are commonly called "sketches". Thus, FlexIoT generates a number of sketches, each corresponding to a possible split point.

5.3.1 Satisfying Memory Constraints

FlexIoT compiles each sketch using Arduino CLI toolchain [18], which allows us to determine the memory usage of the sketch. We capture this output and determine if the sketch will fit on the device.

Memory capacity can be tested without a device physically present, user just needs to indicate the device name.

5.3.2 Satisfying Latency Requirements

Before testing edge components with operators FlexIoT tests user-provided default Arduino sketch. In some cases (when the sensor or the device is not configured properly), collection of a sensor reading alone can take longer than the allowed latency. In this case, FlexIoT displays a warning, suggesting to the user to configure the sensor or the device (or both) appropriately.

In order to test latency of possible edge components, the developer has to have the microcontroller physically present and plugged into their development machine. If device has enough memory to run the sketch, we then upload the sketch on the device and test whether it satisfies latency requirements dictated by user-indicated sensor collection frequency. Individual sensor components are designed for certain sensor collection frequency and expect the sensor to collect readings at that frequency. If we violate that collection frequency, we risk obtaining incorrect results. When we ingress sensor readings into the SPE for processing, it takes time to process the reading. If the time to process the reading with N operators (suppose that we are testing whether N operators can be placed on the edge) takes longer than the interval defined by collection frequency (for example, in ECG application, the sensor collection frequency is 1 KHz, which means that interval is 1000 microseconds), this means that we can not put N operators on the edge.

5.4 Split Objective

When using FlexIoT the programmer indicates one of two objectives: **maximize sensor compute** where we try to put as many operators on the edge as possible, therefore maximizing the portion of workload that is processed at the edge, and **minimize network transmissions**, where we try to reduce the number of edge-cloud messages. In the former case, FlexIoT will try to increment the number of operators on the edge one by one until it either runs out of operators and we end up in the "Maximum Edge" case when the entire individual sensor component fits on the edge, or we end up in the "Partial Edge" case when some, but not all operators from individual sensor component fit on the edge device. In the latter case, FlexIoT detects all operators that output arrays and test edge-cloud splits where these operators are the last operators of the edge component. The logic behind this heuristic is that if the output is an array, it is likely that multiple values were aggregated into the array, and since multiple values were aggregated, the frequency of transmissions between edge and cloud will be reduced. After identifying all such operators, FlexIoT tests splits ending in these operators and records frequency of edge-cloud transmissions (messages per second). The split that satisfies memory and latency requirements with least frequent transmissions is then selected as the edge component. If no such splits are detected or eligible, FlexIoT defaults to **maximize sensor compute** objective. Although there can be other, more complicated, split objectives, we have identified these two as the typical objectives, since they correspond to two major parts of the cost of operating a fleet of IoT devices.

5.5 Cloud

Once edge component has been generated, FlexIoT generates C++ sources for cloud component. Recall from Figure 3.1 that we have two possibilities for edge component: Partial edge and Maximum edge. In the latter case, the entire individual sensor component fits onto the edge device. In this case, we do not need to generate *Scatter* operator on the cloud and we generate all of user's operators after *Scatter* into a C++ source code file. In Partial Edge case FlexIoT has to generate Scatter operator on the cloud because we need to process part of individual sensor component on the cloud. However, it is important to note that output type of last edge operator and the first operator of cloud portion of individual sensor component have to match. In order to account for that, during cloud component generation, FlexIoT automatically changes the input data type of *Scatter* operator to match what is being sent from the edge. After generating all query operators, FlexIoT generates TCP \IP server code to receive messages from edge devices. All operators after *Scatter* are generated on the cloud.

Chapter 6

Methodology

6.1 Data

We used real data that was collected using appropriate sensor for the application. Data from the sensors was in the IEEE FP32 form [55]. ECG sensor data was collected at 1000 Hz from real human subjects. EMG data was collected at 500 Hz also from real human subjects. For Bearing Health data, we have experimented with two frequencies: 3000 Hz and 6000 Hz [3] because these are common frequencies for analysis of vibration of rotating machinery. Bearing data was collected from an accelerometer that is embedded on the same board as the micro-controller we used.

6.2 Baselines

We compare FlexIoT implementation of our three use cases against manual implementation[2] of the use cases to find out how much performance is lost due to Stream Processing implementation of the workload. We compare SPE implementation against manual implementation of the workloads on a micro-controller by timing the processing time of each sensor event.

6.3 Metrics

When comparing manual vs. SPE implementations of our workloads on a micro-controller, we use event processing latency as a metric. Latency reported is the average of at least 4000 trials. For cloud scalability experiments we use throughput (which we define as the number of sensors a single cloud instance can service), network transmissions per second that edge component sends to the cloud, and monthly projected costs (is USDs) for cloud services with AWS [11]. For estimates of engineering effort we used the lines of code and the engineering time estimation as the most realistic metrics we can measure.

[2]To our best knowledge, there is no other system for IoT application development that is similar to FlexIoT against which we could compare. For detailed discussion of related work, please see Section 8.

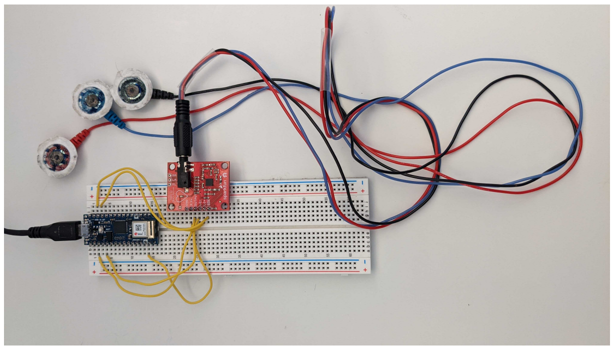


Figure 6.1: ECG Sensor AD8232 connected to Arduino Nano 33 IoT.

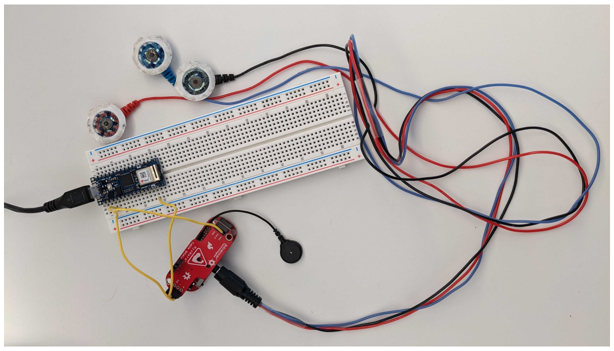


Figure 6.2: EMG Sensor AT-04-001 connected to Arduino Nano 33 IoT.

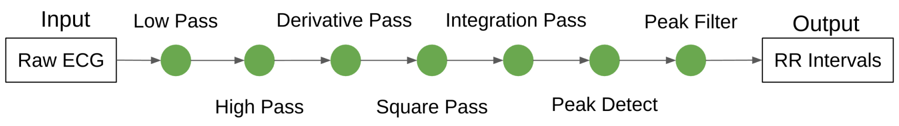
6.4 Hardware

For all experiments, we used AMD Ryzen 9 3900X [15] machine running at 3.8 GHz with 32 GB RAM, and running Ubuntu 20.04. For the microcontroller, we used Arduino Nano 33 IoT [23] with SAMD21 Cortex®-M0+ [25] 32-bit low power ARM MCU and 32 KB of RAM. For ECG sensor, we used Sparkfun AD8232 Heart Rate Monitor Sensor [101] attached to the micro-controller. Figure 6.1 shows the setup of ECG sensor. Electrodes were then connected to a human subject from who data was obtained and analyzed. For EMG we used Myoware Muscle Sensor (AT-04-001) [85]. Figure 6.2 shows EMG sensor set up and connected to Arduino. For Bearing Health application, we used LSM6DS3 [74] accelerometer that is embedded with Arduino Nano 33 IoT board. When conducting experiments on the cloud, we have linearly scaled our AMD [4] development machine to simulate AWS m6i [12] instance, a machine with 64 physical cores. When we ran throughput experiments, we recorded throughput on a single core of our local development machine and scaled linearly to 64 physical cores to estimate throughput on a cloud server machine. Thus our throughput experiments estimate the throughput in favour of cloud service since in practice performance scaling is not linear.

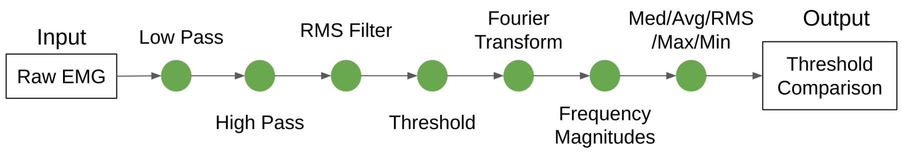
Chapter 7

Evaluation

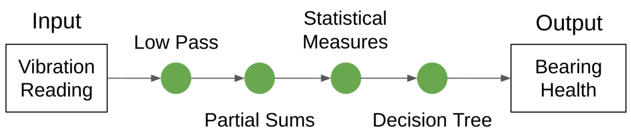
In this section, we present the signal processing pipelines of our selected applications, perform evaluation of FlexIoT against manual implementation, perform a sensitivity study of how much a user would be able to save from offloading parts of our selected workloads to the edge, and finally evaluate how much engineering effort can be saved from using FlexIoT.



(a) ECG signal processing pipeline.



(b) EMG signal processing pipeline.



(c) Bearing Health processing pipeline.

Figure 7.1: Signal processing pipelines for three selected applications.

7.1 Applications

Figure 7.1 shows signal processing pipelines of our selected applications.

1. ECG: Operations in Figure 7.1 filter raw ECG signal coming from lead wires and ultimately transform it into integers, which represent RR-intervals, which are intervals between consecutive heart beats. The algorithm used is Pan-Tompkins algorithm [90]. We sampled the ECG sensor at 1000 Hz.

2. EMG: Operations in Figure 7.1 filter, rectify, and perform Fast Fourier Transform on the signal to transform the signal into frequency domain, followed by extraction of statistical measures of signal frequencies. As the last step, statistical measures are compared against thresholds, which identifies if there is a possible muscle disorder [46]. We sampled the EMG sensor at 500 Hz.

3. BHM: Operations in Figure 7.1 filter the accelerometer signal, after which partial sums are extracted, followed by calculation of statistical measures using those partial sums, which is followed by using the statistical measures to classify the bearing with various types and degrees of fault using a decision tree [111] [30]. In our experiments we used 2 frequencies: 3000 Hz and 6000 Hz [3] because these are common frequencies for analysis of vibration of rotating machinery.

For all three applications, the operations depicted in Figure 7.1 are mathematical operations of a single input event or a window of input events. For example, RMS filter in the Figure 7.1b collects 80 events and calculates their Root Mean Square [96] and passes the result to the next operator. After each individual sensor component is computed, we assign timestamps, join the streams, and for aggregated processing component, we compute the average and the maximum of the values. We provide an example of implementation of a large part of ECG signal processing pipeline using our SPE in Appendix A.

7.2 Evaluation

We evaluate FlexIoT with the following questions in mind:

1. How much engineering effort can be avoided when using FlexIoT compared to manual implementation?

2. How much performance overhead does we incur by moving from manual to SPE implementation of the edge component and does it affect the split point?

3. How much can a programmer save by offloading computation to the edge for our selected applications?

7.3 Reducing Engineering Effort

We want to evaluate how much engineering effort and time is saved when using FlexIoT over the manual implementation. We have implemented our three use cases manually and directly compared the lines of code used for FlexIoT implementation against manual implementation. Table 7.1 demonstrates the results.

From the above table we can conclude that we can reduce the number of lines of code by approximately 30%. The difference in the number of lines comes from the fact that different applications have different number of custom operators. Applications that have larger number of custom operators will normally have less savings in terms of lines of code since the user has to implement the custom operator. For example, ECG has proportionally the largest number of custom operators and BHM has the least, which is reflected in more savings for BHM.

Application	Manual	FlexIoT	% Decrease
ECG	397	299	25
EMG	277	187	32
BHM	258	150	40

Table 7.1: Lines of code saved by using FlexIoT.

Time is another important metric in evaluation of engineering effort required to implement an application. We argue that FlexIoT allows the user to save a significant amount of time. When the user tries to determine an appropriate edge-cloud split for their application, they have to implement the edge, the cloud, and the communication components. If user wants to test a different split, they have to re-implement parts of the edge, the cloud, and the communication components for each attempt. Moreover, for each split the developer has to perform testing and verify that the current split satisfies sensor latency requirements. All this engineering work is avoided if one uses FlexIoT, which will find the split automatically, verify that the split satisfies sensor latency requirements, and produce edge, cloud, and communication components.

7.4 SPE vs. Manual

We want to evaluate how much performance is lost when we express the workload as a streaming query. As we mentioned in Section 2, Stream Processing Engines are convenient, because they implement important functions (such as windowing, summing, averaging), but they do so with some performance overhead. For example, splitting a stream into separate substreams has to normally be done by a special operator that casts the events of the incoming stream to each of its substreams. Manual implementation of this concept would be trivial. Thus, we need to evaluate how much performance is lost on the edge component if we implement it as a streaming query as opposed to manual implementation. We manually implement our three selected applications and split the manual version at the same split points as the SPE version.

Figure 7.2 shows the slowdown in % of edge component for each split point. Average and maximum slowdowns we incur for ECG is 2.45% and 3.70% respectively. For EMG, the average and maximum latency increases are 0.3625% and 1.55% respectively. Finally, for BHM the average and maximum latency increases are 4.756% and 7.45%. Higher latency increase for BHM can be explained by the fact that SubExpr in SPE implementation contains several Parallel subqueries and aggregations that require more involved data manipulation, which require more operators. It is important to note that at no point did the latency increase affect the split decision, meaning that there is no split point at which the latency increase was such that the manual version could execute on the micro-controller and the SPE version could not. *We conclude* that it is possible to implement signal processing pipelines using SPEs with minimal increase in latency for the edge component.

7.5 Cost Saving Sensitivity Study

In this section, we evaluate the cost aspect when offloading the operators from the cloud to the edge.

For this experiment, we estimated an upper bound on the throughput of the cloud node (AWS

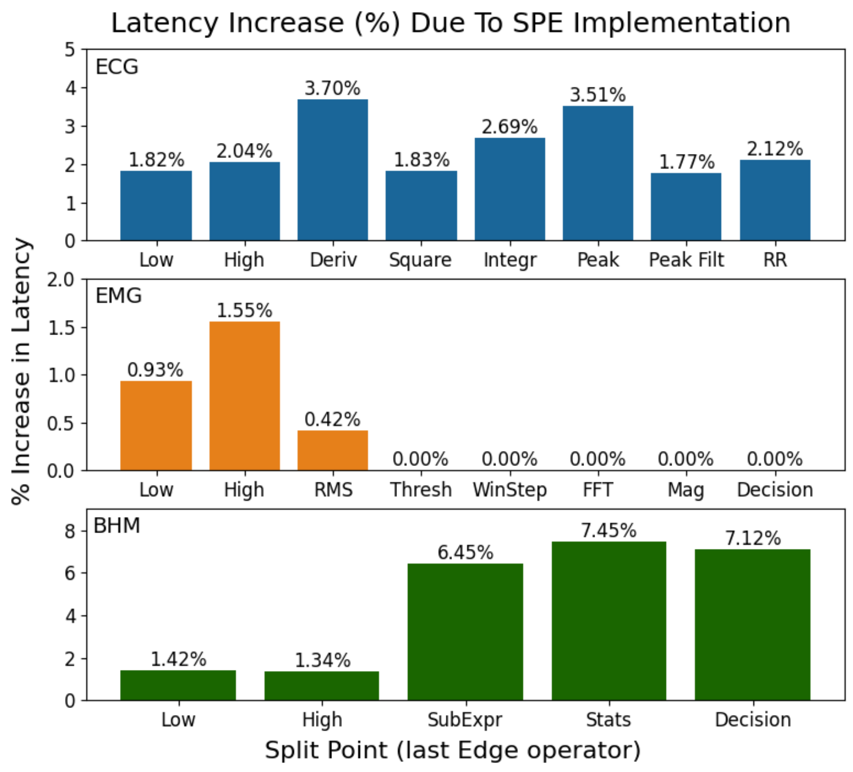
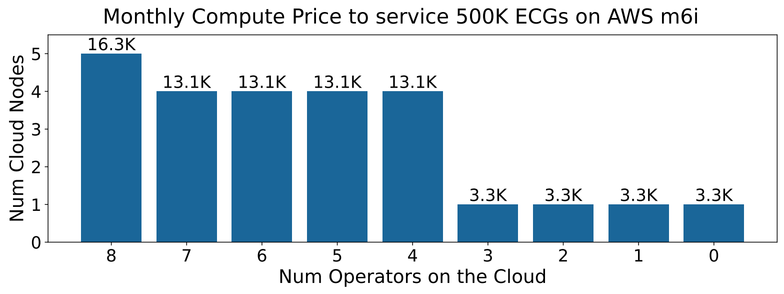


Figure 7.2: Edge component latency increase for every possible split for manual vs. SPE implementation.

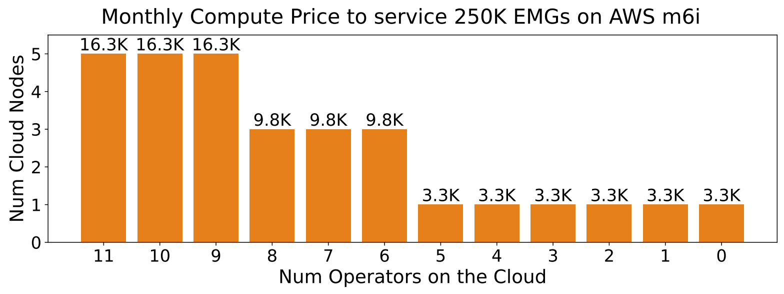
m6i [12]), i.e. how many sensor devices a cloud node can service for each split point. The price for this instance is 3263$ per month in a reserved role [14](Accessed Aug 19, 2022). In order to define what we mean by cloud **servicing** the applications, we need to define the failure criteria for each application. For **ECG**, we require that cloud processes a minute of ECG samples in a minute of time. Otherwise, the cloud will fail to process the incoming events in a live fashion and will accumulate the lag over time. For **EMG**, we require that individual sensor component produces output every 200 ms because we set up our EMG query to detect frequency content change over the window of 200 ms. For **BHM**, we require the query to output bearing health measurement every 1 second. BHM is a predictive manintenance application which happens in a factory where bearings are used in rotating elements and it is desirable to get the estimated bearing health in a live manner so as to stop the rotating element with unhealthy bearing before the mechanical problem spills to other parts of the machine.

In order to estimate costs of operating large number of sensors we estimate fleet sizes (number of sensors that cloud provider services) for each application. When available, we base our fleet size estimates on existing providers. AliveKor [64] claims to have sold over 1 million units [9]. We take half of that - 500K as the fleet size for ECG. EMG is a use case that is not as prevalent as ECG, so we take half of the fleet size of ECG - 250K units. In order to estimate BHM fleet size, we note that there were 373K robots installed in factories worldwide in 2019 and the total number of robots in operation worldwide is 2.7 million [56]. With these numbers in mind, we take 200K as the fleet size of robots. Supposing that each robot has 5 important bearings that need to be monitired, we arrive at 1 million accelerometer sensors that need to be serviced by the cloud.

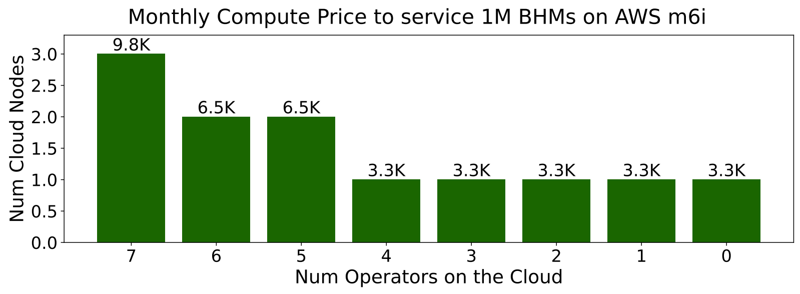
With these failure criteria and fleet sizes we evaluated cloud costs from two perspectives - from perspective of compute and network. Figure 7.3 shows estimated *compute* costs analysis for each application for each split. The estimates were obtained by estimating the throughput (in terms of number of queries that can be serviced) of a single cloud server node (AWS m6i) for each possible split and comparing to the total fleet size, thus obtaining the number of cloud nodes required to service the entire fleet, which is multiplied by the cost to rent the cloud node. Figure 7.4 shows estimated *network* costs analysis for each application for each split. The estimates were obtained by recording the number of messages edge component sends to the cloud for each split, followed by scaling by the fleet size and multiplying by the cost to receive message on AWS IoT Core [13]. Notice that we have used the cheapest pricing option of MQTT and HTTP messaging. The main takeaway from these figures is that for our class of applications offloading computation to the edge allows a provider of IoT services to cut compute costs by as much as 5X and network costs by as much as 1000X. Moreover, without offloading computation to the edge, network costs make deploying such applications prohibitively expensive.



(a) ECG signal processing pipeline.

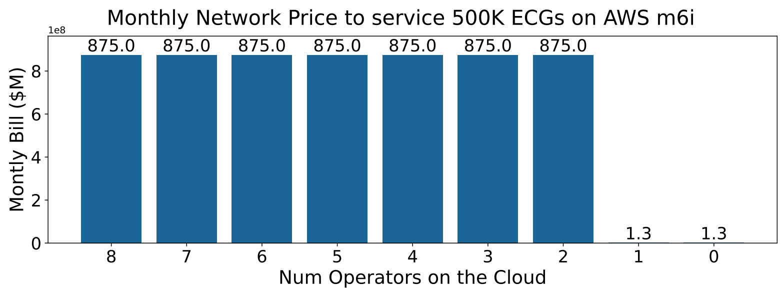


(b) EMG signal processing pipeline.

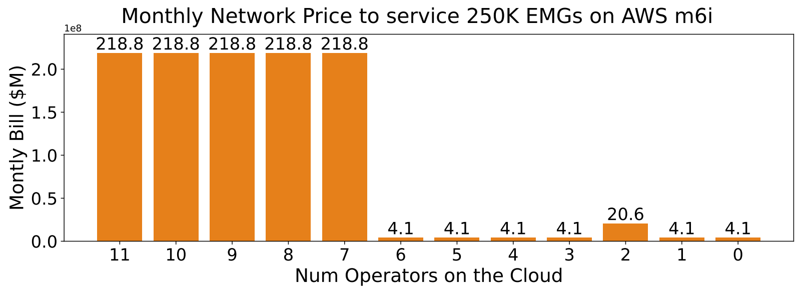


(c) Bearing Health processing pipeline.

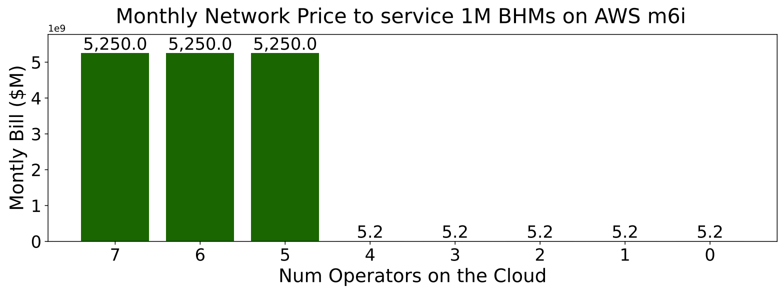
Figure 7.3: Monthly Prices to Service Fleets for each split.



(a) ECG signal processing pipeline.



(b) EMG signal processing pipeline.



(c) Bearing Health processing pipeline.

Figure 7.4: Monthly Prices to Service Fleets for each split.

Chapter 8

Related Work

Adaptive computation offloading is becoming increasingly popular in recent years because of the development of IoT and edge computing. The main distinction that can be made between these works is how they find *what* to offload. Some assume a set of tasks that are ready to be offloaded, while others partition user input automatically or semi-automatically.

TinyLink 2.0 [49] aims to make IoT application development easier by using IFTTT [57] programming model combined with their Domain Specific Language (DSL) to enable users to write different components of IoT applications in a single programming language. Framework itself can shift sections of code from cloud to edge device automatically. TinyLink 2.0 targets simpler event-driven IoT applications that require micro-controllers paired with sensors to perform actions (such as turning on LEDs and performing mechanical actions depending on user input) in domains such as edication, maker and startup.

EdgeProg [71] is an edge-centric approach to IoT application development which also uses IFTTT programing model and targets micro-controllers and applications similar to TinyLink 2.0. Unlike TinyLink 2.0 however, EdgeProg allows users to declare more sophisticated data processing pipelines using "stages", which EdgeProg automatically splits between edge and cloud. EdgeProg seeks to *offload heavy stages from edge to cloud*. Latency of these stages is estimated for different devices using simulators. Estimated stage runtimes and network speed are input into Integer Linear Program [58] whose goal is to determine the placement of stages on devices to minimize total application latency.

We are fundamentally different from both TinyLink 2.0 and EdgeProg in that our system is cloud-centric and we target different types of applications - ones where deep analysis is performed on results of multiple sensors of the same type. As such, these frameworks are not built to assign sensor IDs to output of each sensor or aggregate results from multiple sensors, which is key to applications that we target. Most importantly, both TinyLink 2.0 and EdgeProg do not test that edge component of the resulting split satisfies sensor collection intervals, which is critical to obtaining correct results for our targeted applications. Note that EdgeProg's optimization objective of minimizing total application latency does *not* guarantee that stages offloaded to the edge will respect sensor collection intervals.

WiProg is another work from the authors of TinyLink 2.0 and EdgeProg that targets micro-controllers and solves the problem of heterogeneous application development by compiling all components of IoT application to WebAssembly intermediate language but it requires user annotations and incurs overhead in terms of memory and latency because it requires WebAssembly interpreter to be deployed on the micro-controller.

There is a set of works in the domain adaptive computation offloading that determine which part of the workload to offload based on user annotations. MAUI [37], Cuckoo [65], ThinkAir [67] target mobile platforms for offloading heavy computation *from edge to cloud* and do so by requiring user annotations. CloneCloud [35] stands out among works that target mobile platforms in that it determines the partition automatically, but that is achieved at the cost of generality as the framework only applies to Android applications.

Sidewinder [72] targets very similar applications to our work, but on mobile platforms. Authors advocate for partitioning of the workload into two parts - one part that will run continuously on a small micro-controller and another part that runs on a more powerful main CPU when user-defined conditions are met. The workload is partitioned in order to allow the mobile device to perform continuous analysis of raw sensor information without having to continuously use the main CPU and thus allow the mobile device to save energy and stay ”asleep”. We are different from Sidewinder in that we are are proposing to split the workload between edge micro-controller and a cloud server instead of the micro-controller and main CPU and we propose to perform the split automatically.

There is a large group of works [112, 113, 28, 61, 33, 47, 66, 42] that target Deep Neural Network applications by splitting based on layers of the Neural Network, which can be done automatically.

Chapter 9

Discussion

9.1 Limitations

The work presented in this manuscript is aimed at IoT use cases that involve fast-collecting sensors with signal processing pipelines. When deploying IoT applications similar to ones discussed in this work, there are three major limiting factors when it comes to IoT architecture:

- Network costs. If the raw data is sent to the cloud, the cost for data ingress into a cloud computing environment will rise with sensor collection frequency. On the other hand, with slower sensor collection frequency, the network costs will be reduced. For example, if the sensor collection frequency is 10 Hz, the monthly network costs to service each sensor come out to roughly $500 on AWS IoT Core[13], which is expensive, but feasible. Thus, for applications with slow collecting sensors, the benefits of our approach will be diminished.

- Network bandwidth. With faster sensor collection frequency the bandwidth pressure on each edge node is increased. Naturally, if the application employs slower collecting sensor, the bandwidth pressure is reduced and it again becomes more feasible to send data to the cloud. Like in the previous case, the benefits of our approach will be diminished with slower collecting sensors.

- Compute costs. Offloading computation to the edge is also beneficial because it relieves compute pressure from the cloud infrastructure. If the application in question does not require pre-processing and aggregation of the raw sensor data, it becomes feasible to send data directly to the cloud and perform cross-sensor aggregation and cross-sensor processing there. In this case there will also be no individual sensor component or it will be so insignificant that it will be feasible to send the data directly to the cloud and process everything there. In this case splitting the workload will not be required and the benefits of our approach will be reduced.

9.2 Future Work

This work explored the case of two layers in the edge-cloud hierarchy (namely edge and cloud). There is prior work that explores the possibility of introducing intermediate layers into the hierarchy [83, 78, 38, 68, 103]. These works propose to include devices in between the edge and the cloud to further partition the computation. Such intermediate devices could include mobile devices, network devices and mini data-centers, among others. In future work we plan to extend our approach to include more than two layers of the hierarchy. We claim that our approach naturally extends to more than two layers in the edge-cloud hierarchy by recursively splitting the remaining workload among remaining devices. Suppose we have three devices (A, B, and C) with A being edge device, C being cloud device and B being intermediate device. Suppose we have determined the part of the total workload that can be placed on the edge device (A). We can then take the remainder of the workload and two remaining devices (B and C) and treat device B as the new edge device. At this point we have reduced the problem with three devices to the original problem with two devices.

Chapter 10

Conclusion

This work presents FlexIoT, a new and efficient system that gives developers an integrated way of writing IoT applications that employ fast-collecting sensors and require raw data pre-processing and aggregation at the edge. FlexIoT achieves this by representing the edge-cloud split workload as a single streaming query that organically includes both edge and cloud components and makes use of a Stream Processing Engine to automatically determine the part of the workload that should execute on the edge. This allows FlexIoT to automatically develop the edge-cloud partition and generate both edge and cloud components. We conduct a study and determine the challenges that arise while developing edge-cloud split applications. We implement three representative applications and evaluate FlexIoT with real human collected data. Our evaluation of our three representative workloads shows that FlexIoT allows the programmer to significantly reduce the amount of code that has to be written as well as save time when determining the edge-cloud split appropriate for the application being developed. All of this is achieved at a minimal increase in latency for the edge component. We also conduct a sensitivity study of our three representative workloads and demonstrate how much money can be saved by pre-processing on the edge for each edge-cloud split point. We conclude that FlexIoT is an efficient framework for development of challenging IoT applications that allows the developer to avoid a significant amount of engineering effort.

Bibliography

[1] 5G by Qualcomm. https://www.qualcomm.com/5g/what-is-5g. 2021.

[2] Jemal H. Abawajy and Mohammad Mehedi Hassan. "Federated Internet of Things and Cloud Computing Pervasive Patient Health Monitoring System". In: *IEEE Communications Magazine* 55.1 (2017), pp. 48–53. DOI: 10.1109/MCOM.2017.1600374CM.

[3] Accelerometers for rotating machinery. https://itestsystem.com/2020/02/24/general-purpose-accelerometers-for-rotating-machinery-vibration-measurements/. 2022.

[4] Advanced Micro Devices. https://www.amd.com/en. 2022.

[5] Yanif Ahmad et al. "Distributed Operation in the Borealis Stream Processing Engine". In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD '05. Baltimore, Maryland: Association for Computing Machinery, 2005, pp. 882–884. ISBN: 1595930604. DOI: 10.1145/1066157.1066274. URL: https://doi.org/10.1145/1066157.1066274.

[6] Tyler Akidau et al. "MillWheel: Fault-Tolerant Stream Processing at Internet Scale". In: *Very Large Data Bases*. 2013, pp. 734–746.

[7] Tyler Akidau et al. "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing". In: *Proceedings of the VLDB Endowment* 8 (2015), pp. 1792–1803.

[8] Alibaba Cloud services. https://www.alibabacloud.com/. 2022.

[9] AliveCor press release claiming over 1 million units sold. https://venturebeat.com/2020/11/16/alivecor-raises-65-million-to-detect-heart-problems-with-ai/. 2022.

[10] Amazon kinesis data analytics. https://aws.amazon.com/kinesis/data-analytics/. 2022.

[11] Amazon Web Services. https://aws.amazon.com/. 2022.

[12] Amazon Web Services EC2 m6i Instance Type. https://aws.amazon.com/ec2/instance-types/m6i/. 2022.

[13] Amazon Web Services IoT Core pricing. https://aws.amazon.com/iot-core/pricing/. 2022.

[14] Amazon Web Services Reserved pricing instances. https://aws.amazon.com/ec2/dedicated-hosts/pricing/. 2022.

[15] AMD Ryzen™ 9 3900X. https://www.amd.com/en/products/cpu/amd-ryzen-9-3900x. 2022.

[16] Apache Spark programming guide. https://spark.apache.org/docs/latest/streaming-programming-guide.html. 2022.

[17] S. Arasu A. Babu and J. Widom. “The CQL continuous query language: semantic foundations and query execution.” In: *The VLDB Journal 15, 121-142 (2006)* 15.2 (June 2006), pp. 121–142. ISSN: 1066-8888. DOI: 10.1007/s00778-004-0147-z. URL: https://doi.org/10.1007/s00778-004-0147-z.

[18] Arduino Command Line Interface tools. https://www.arduino.cc/pro/cli. 2022.

[19] Arduino hardware. https://www.arduino.cc/en/hardware. 2022.

[20] Arduino Home Web Page. https://www.arduino.cc/. 2022.

[21] Arduino MKR WAN 1310 specs. https://store-usa.arduino.cc/products/arduino-mkr-wan-1310. 2022.

[22] Arduino MKR Wifi 1010 specs. https://store-usa.arduino.cc/products/arduino-mkr-wifi-1010. 2022.

[23] Arduino Nano 33 IoT product page. https://store-usa.arduino.cc/products/arduino-nano-33-iot. 2021.

[24] Arduino programming language. https://www.arduino.cc/reference/en/. 2022.

[25] Arm Cortex M0+ processor. https://developer.arm.com/Processors/Cortex-M0-Plus. 2017.

[26] Michael Armbrust et al. “Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark”. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD ’18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 601–613. ISBN: 9781450347037. DOI: 10.1145/3183713.3190664. URL: https://doi.org/10.1145/3183713.3190664.

[27] Brian Babcock et al. “Models and Issues in Data Stream Systems”. In: *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS ’02. Madison, Wisconsin: Association for Computing Machinery, 2002, pp. 1–16. ISBN: 1581135076. DOI: 10.1145/543613.543615. URL: https://doi.org/10.1145/543613.543615.

[28] Arian Bakhtiarnia et al. *Dynamic Split Computing for Efficient Deep Edge Intelligence*. 2022. DOI: 10.48550/ARXIV.2205.11269. URL: https://arxiv.org/abs/2205.11269.

[29] Bearing Health Measurement. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3545588/. 2012.

[30] Mouloud Boumahdi et al. “On the extraction of rules in the identification of bearing defects in rotating machinery using decision tree”. In: *Expert Systems with Applications* 37.8 (2010), pp. 5887–5894. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2010.02.017. URL: https://www.sciencedirect.com/science/article/pii/S0957417410000564.

[31] C++ programming language. https://en.wikipedia.org/wiki/C%2B%2B. 2022.

[32] Paris Carbone et al. “apache flink™: stream and batch processing in a single engine”. In: *ieee data eng. bull.* 38.4 (2015), pp. 28–38. URL: http://sites.computer.org/debull/a15dec/p28.pdf.

[33] Xing Chen et al. “DNNOff: Offloading DNN-Based Intelligent IoT Applications in Mobile Edge Computing”. In: *IEEE Transactions on Industrial Informatics* 18.4 (2022), pp. 2820–2829. DOI: 10.1109/TII.2021.3075464.

[34] Jiangfeng Cheng et al. “Industrial IoT in 5G environment towards smart manufacturing”. In: *Journal of Industrial Information Integration* 10 (2018), pp. 10–19. ISSN: 2452-414X. DOI: https://doi.org/10.1016/j.jii.2018.04.001. URL: https://www.sciencedirect.com/science/article/pii/S2452414X18300049.

[35] Byung-Gon Chun et al. “CloneCloud: Boosting Mobile Device Applications Through Cloud Clone Execution”. In: *CoRR* abs/1009.3088 (2010). arXiv: 1009.3088. URL: http://arxiv.org/abs/1009.3088.

[36] Crest Factor Wikipedia Page. https://en.wikipedia.org/wiki/Crest_factor. 2022.

[37] Eduardo Cuervo et al. “MAUI: Making Smartphones Last Longer with Code Offload”. In: *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*. MobiSys ’10. San Francisco, California, USA: Association for Computing Machinery, 2010, pp. 49–62. ISBN: 9781605589855. DOI: 10.1145/1814433.1814441. URL: https://doi.org/10.1145/1814433.1814441.

[38] A.V. Dastjerdi et al. “Chapter 4 - Fog Computing: principles, architectures, and applications”. In: *Internet of Things*. Ed. by Rajkumar Buyya and Amir Vahid Dastjerdi. Morgan Kaufmann, 2016, pp. 61–75. ISBN: 978-0-12-805395-9. DOI: https://doi.org/10.1016/B978-0-12-805395-9.00004-6. URL: https://www.sciencedirect.com/science/article/pii/B9780128053959000046.

[39] Delsys mobile EMG with Tablet. https://delsys.com/mobile-emg-suite/. 2022.

[40] Fraud Detection and Prevention: A Data Analytics Approach. https://wso2.com/whitepapers/fraud-detection-and-prevention-a-data-analytics-approach/. 2015.

[41] Mrinai M. Dhanvijay and Shailaja C. Patil. “Internet of Things: A survey of enabling technologies in healthcare and its applications”. In: *Computer Networks* 153 (2019), pp. 113–131. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2019.03.006. URL: https://www.sciencedirect.com/science/article/pii/S1389128619302695.

[42] Maryam Ebrahimi et al. “Combining DNN Partitioning and Early Exit”. In: *Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking*. EdgeSys ’22. Rennes, France: Association for Computing Machinery, 2022, pp. 25–30. ISBN: 9781450392532. DOI: 10.1145/3517206.3526270. URL: https://doi.org/10.1145/3517206.3526270.

[43] EEG sensor sampling rate. https://en.wikipedia.org/wiki/Electroencephalography#Method. 2021.

[44] Electrocardiogram. https://www.hopkinsmedicine.org/health/treatment-tests-and-therapies/electrocardiogram. 2022.

[45] Electromyography. https://www.hopkinsmedicine.org/health/treatment-tests-and-therapies/electromyography-emg. 2022.

[46] Electromyography Analysis pipeline. https://www1.udel.edu/biology/rosewc/kaap686/notes/EMG%20analysis.pdf. 2019.

[47] John Emmons et al. "Cracking Open the DNN Black-Box: Video Analytics with DNNs across the Camera-Cloud Boundary". In: *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*. HotEdgeVideo'19. Los Cabos, Mexico: Association for Computing Machinery, 2019, pp. 27–32. ISBN: 9781450369282. DOI: 10.1145/3349614.3356023. URL: https://doi.org/10.1145/3349614.3356023.

[48] Google Cloud Platform. https://cloud.google.com/. 2022.

[49] Gaoyang Guan et al. "TinyLink 2.0: Integrating Device, Cloud, and Client Development for IoT Applications". In: *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*. New York, NY, USA: Association for Computing Machinery, 2020. ISBN: 9781450370851. URL: https://doi.org/10.1145/3372224.3380890.

[50] R. Gwadera et al. "Nile: A Query Processing Engine for Data Streams". In: *Proceedings. 20th International Conference on Data Engineering*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2004, p. 851. DOI: 10.1109/ICDE.2004.1320080. URL: https://doi.ieeecomputersociety.org/10.1109/ICDE.2004.1320080.

[51] HGSI GD4200-USB Digital Pressure Transducer. https://www.hgsind.com/product/gd4200-usb-esi-digital-pressure-transducer. 2021.

[52] Rami Hodrob et al. "An IoT Based Healthcare using ECG". In: *2020 21st International Arab Conference on Information Technology (ACIT)*. 2020, pp. 1–4. DOI: 10.1109/ACIT50332.2020.9300104.

[53] HRLV-ShortRange EZ Series Proximity sensor. https://www.maxbotix.com/documents/HRLV-ShortRange-EZ_Datasheet.pdf. 2021.

[54] IBM Cloud Services. https://www.ibm.com/cloud. 2022.

[55] IEEE. "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[56] IFR World Robotics Report 2020. https://ifr.org/ifr-press-releases/news/record-2.7-million-robots-work-in-factories-around-the-globe. 2020.

[57] IFTTT programming framework. https://ifttt.com/explore/new_to_ifttt. 2022.

[58] Integer Linear Programming. https://en.wikipedia.org/wiki/Integer_programming. 2022.

[59] Intel - What is Edge Computing. https://www.intel.ca/content/www/ca/en/edge-computing/what-is-edge-computing.html. 2022.

[60] Internet of Things by Orable. https://www.oracle.com/ca-en/internet-of-things/what-is-iot/. 2021.

[61] Jananie Jarachanthan et al. "AMPS-Inf: Automatic Model Partitioning for Serverless Inference with Cost Efficiency". In: *50th International Conference on Parallel Processing*. ICPP 2021. Lemont, IL, USA: Association for Computing Machinery, 2021. ISBN: 9781450390682. DOI: 10.1145/3472456.3472501. URL: https://doi.org/10.1145/3472456.3472501.

[62] Java programming language. https://docs.oracle.com/javase/7/docs/technotes/guides/language/. 2022.

[63] Anand Jayarajan et al. “LifeStream: A High-Performance Stream Processing Engine for Periodic Streams”. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 107–122. ISBN: 9781450383172. DOI: 10.1145/3445814.3446725. URL: https://doi.org/10.1145/3445814.3446725.

[64] Kardia AliveCor mobile ECG. https://www.kardia.com/. 2022.

[65] Roelof Kemp et al. “Cuckoo: A Computation Offloading Framework for Smartphones”. In: *Mobile Computing, Applications, and Services*. Ed. by Martin Gris and Guang Yang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 59–79. ISBN: 978-3-642-29336-8.

[66] Jong Hwan Ko et al. “Edge-Host Partitioning of Deep Neural Networks with Feature Space Encoding for Resource-Constrained Internet-of-Things Platforms”. In: *2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*. 2018, pp. 1–6. DOI: 10.1109/AVSS.2018.8639121.

[67] Sokol Kosta et al. “ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading”. In: *2012 Proceedings IEEE INFOCOM*. 2012, pp. 945–953. DOI: 10.1109/INFCOM.2012.6195845.

[68] Frank Alexander Kraemer et al. “Fog Computing in Healthcare–A Review and Discussion”. In: *IEEE Access* 5 (2017), pp. 9206–9222. DOI: 10.1109/ACCESS.2017.2704100.

[69] Sanjeev Kulkarni et al. “Twitter Heron: Stream Processing at Scale”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 239–250. ISBN: 9781450327589. DOI: 10.1145/2723372.2742788. URL: https://doi.org/10.1145/2723372.2742788.

[70] LG ThinQ Smart Refrigerator. https://www.lg.com/us/discover/thinq/refrigerators. 2022.

[71] Borui Li and Wei Dong. “EdgeProg: Edge-centric Programming for IoT Applications”. In: *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. 2020, pp. 212–222. DOI: 10.1109/ICDCS47774.2020.00038.

[72] Daniyal Liaqat et al. “Sidewinder: An Energy Efficient and Developer Friendly Heterogeneous Architecture for Continuous Mobile Sensing”. In: *SIGARCH Comput. Archit. News* 44.2 (Mar. 2016), pp. 205–215. ISSN: 0163-5964. DOI: 10.1145/2980024.2872398. URL: https://doi-org.myaccess.library.utoronto.ca/10.1145/2980024.2872398.

[73] Konstantinos Mamouras Lingkun Kong. “streamql: a query language for processing streaming time series”. In: *Proceedings of the ACM on Programming Languages, volume 4, issue oopsla*. 2020.

[74] LSM6DS3 Accelerometer Data Sheet. https://www.mouser.ca/datasheet/2/389/dm00133076-1798402.pdf. 2017.

[75] LSM6DSL Accelerometer by ST Electronics. https://www.st.com/en/mems-and-sensors/lsm6dsl.html. 2021.

[76] R. Caceres M. Satyanarayanan P. bahl and N. Davies. “The Case For VM-Based Cloudlets in Mobile Computing”. In: *Pervasive Computing, Vol. 8, no. 4, pp. 14-23, oct.-dec. 2009, doi: 10.1109/mprv.2009.82*. 2020.

[77] Dimitris Margaritis et al. “Calibration of Low-cost Gas Sensors for Air Quality Monitoring”. In: *Aerosol and Air Quality Research* 21.11 (2021), p. 210073. DOI: 10.4209/aaqr.210073. URL: https://doi.org/10.4209%2Faaqr.210073.

[78] Jonathan McChesney et al. “DeFog: Fog Computing Benchmarks”. In: *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing (SEC)*. Washington, DC, Nov. 2019.

[79] Hongyu Miao et al. “StreamBox-HBM: Stream Analytics on High Bandwidth Hybrid Memory”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 167–181. ISBN: 9781450362405. DOI: 10.1145/3297858.3304031. URL: https://doi.org/10.1145/3297858.3304031.

[80] Hongyu Miao et al. “StreamBox: Modern Stream Processing on a Multicore Machine”. In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 617–629. ISBN: 978-1-931971-38-6. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/miao.

[81] Microsoft Azure Cloud Computing Services. https://azure.microsoft.com/en-ca/. 2022.

[82] Microsoft Research. *Microsoft Trill*. https://www.microsoft.com/en-us/research/project/trill/. 2021.

[83] Seyed Hossein Mortazavi et al. “CloudPath: A Multi-Tier Cloud Computing Framework”. In: *2nd ACM/IEEE Symposium on Edge Computing (SEC)*. San Jose, CA, Oct. 2017.

[84] MyoWare (AT-04-001) 3-Lead Muscle / Electromyography Sensor. https://cdn.sparkfun.com/assets/learn_tutorials/4/9/1/datasheet.pdf. 2015.

[85] Myoware Muscle Sensor AT-04-001. https://learn.sparkfun.com/tutorials/myoware-muscle-sensor-kit/all. 2022.

[86] Garima Nain, K.K. Pattanaik, and G.K. Sharma. “Towards edge computing in intelligent manufacturing: Past, present and future”. In: *Journal of Manufacturing Systems* 62 (2022), pp. 588–611. ISSN: 0278-6125. DOI: https://doi.org/10.1016/j.jmsy.2022.01.010. URL: https://www.sciencedirect.com/science/article/pii/S0278612522000103.

[87] Node MCU Amica V2 specifications. https://www.hackster.io/javagoza/nodemcu-amica-v2-road-test-2e8bff#toc-breadboard-compatibility-3. 2022.

[88] Oracle Cloud Services. https://www.oracle.com/cloud/. 2022.

[89] Oracle Internet of Things Cloud Service. https://docs.oracle.com/en/cloud/paas/iot-cloud/index.html. 2022.

[90] Jiapu Pan and Willis J. Tompkins. “A Real-Time QRS Detection Algorithm”. In: *IEEE Transactions on Biomedical Engineering* BME-32.3 (1985), pp. 230–236. DOI: 10.1109/TBME.1985.325532.

[91] JeeWoong Park et al. “Improving dynamic proximity sensing and processing for smart work-zone safety”. In: *Automation in Construction* 84 (2017), pp. 111–120. ISSN: 0926-5805. DOI: https://doi.org/10.1016/j.autcon.2017.08.025. URL: https://www.sciencedirect.com/science/article/pii/S0926580517301590.

[92] Particle Photon specifications. https://docs.particle.io/photon/. 2022.

[93] Gennady Pekhimenko et al. “TerseCades: Efficient Data Compression in Stream Processing”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 307–320. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/pekhimenko.

[94] Python programming language. https://www.python.org/. 2022.

[95] Kanishk Rai et al. “Design of an EEG Acquisition System for Embedded Edge Computing”. In: *Advances in Science, Technology and Engineering Systems Journal* 5.4 (2020), pp. 119–129. DOI: 10.25046/aj050416.

[96] Root Mean Square. https://en.wikipedia.org/wiki/Root_mean_square. 2022.

[97] Scala programming language. https://www.scala-lang.org/. 2022.

[98] Sensors for Industrial Applications by TE connectivity. https://www.te.com/usa-en/industries/sensor-solutions/applications/sensors-for-industrial.html?tab=pgp-story. 2022.

[99] Sensors for Medical Applications by TE connectivity. https://www.te.com/usa-en/industries/sensor-solutions/applications/iot-sensors/iomt.html. 2022.

[100] Hakki Soy and İbrahim Toy. “Design and implementation of smart pressure sensor for automotive applications”. In: *Measurement* 176 (2021), p. 109184. ISSN: 0263-2241. DOI: https://doi.org/10.1016/j.measurement.2021.109184. URL: https://www.sciencedirect.com/science/article/pii/S0263224121002013.

[101] Sparkfun AD8232 Heart Rate Monitor Sensor. https://www.sparkfun.com/products/12650. 2022.

[102] SparkFun Single Lead Heart Rate Monitor - AD8232. https://www.qualcomm.com/5g/what-is-5g. 2021.

[103] Ivan Stojmenovic and Sheng Wen. “The Fog computing paradigm: Scenarios and security issues”. In: *2014 Federated Conference on Computer Science and Information Systems*. 2014, pp. 1–8. DOI: 10.15439/2014F503.

[104] Stream Processing programming paradigm. https://en.wikipedia.org/wiki/Stream_processing. 2022.

[105] Adrian Ţăran-Moroşan. “The relative strength index revisited”. In: *African Journal of Business Management* 5.14 (2011), pp. 5855–5862.

[106] Teensy 3.6 specs. https://www.pjrc.com/store/teensy36.html. 2022.

[107] Douglas Terry et al. “Continuous Queries over Append-Only Databases”. In: *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’92. San Diego, California, USA: Association for Computing Machinery, 1992, pp. 321–330. ISBN: 0897915216. DOI: 10.1145/130283.130333. URL: https://doi.org/10.1145/130283.130333.

[108] Abhishek Tiwari et al. “Reconfigurable Streaming for the Mobile Edge”. In: *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*. HotMobile ’19. Santa Cruz, CA, USA: Association for Computing Machinery, 2019, pp. 153–158. ISBN: 9781450362733. DOI: 10.1145/3301293.3302355. URL: https://doi-org.myaccess.library.utoronto.ca/10.1145/3301293.3302355.

[109] B. Varghese et al. “Revisiting the Arguments for Edge Computing Research”. In: *IEEE Internet Computing* 25.05 (Sept. 2021), pp. 36–42. ISSN: 1941-0131. DOI: 10.1109/MIC.2021.3093924.

[110] What is cloud computing by Microsoft Azure. https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-cloud-computing/. 2022.

[111] Zhanguo Xia et al. “Spectral Regression Based Fault Feature Extraction for Bearing Accelerometer Sensor Signals”. In: *Sensors* 12.10 (2012), pp. 13694–13719. ISSN: 1424-8220. DOI: 10.3390/s121013694. URL: https://www.mdpi.com/1424-8220/12/10/13694.

[112] Beibei Zhang et al. “Dynamic DNN Decomposition for Lossless Synergistic Inference”. In: *CoRR* abs/2101.05952 (2021). arXiv: 2101.05952. URL: https://arxiv.org/abs/2101.05952.

[113] Beibei Zhang et al. “Dynamic DNN Decomposition for Lossless Synergistic Inference”. In: *CoRR* abs/2101.05952 (2021). arXiv: 2101.05952. URL: https://arxiv.org/abs/2101.05952.

Appendix A

Example of Source Code

In this Appendix we present an example of source code for several steps of ECG application's signal processing pipeline so that reader has an idea of how they are implemented using our SPE.

First the programmer would write a file containing implementations of all custom containers, operators, and functors they intend to use in their query. An example of source code of such file for part of ECG application:

```
/*** Sizes of windows for operators. ***/
#define PT_BUFFER_SIZE 1400
#define INT_WIN_SIZE 100

/*** Structs used as containers for different states. ***/
struct FD
{
  float fil;
  float der;
};

struct FDI
{
  float fil;
  float der;
  float integ;
};

struct FSIP
{
  float fil;
  float sq;
  float integ;
  int peak;
};

/*** Base class for all operator and receiver classes. */
template <typename IN> class Receiver
{
 public:
  Receiver() = default;
  virtual void receive(IN item) = 0;
  virtual void end() = 0;
};

/*** Base class for all operators. ***/
template <typename IN, typename OUT> class Op : public Receiver<IN>
{
 public:
  virtual void connect(Receiver<OUT> *out_operator) = 0;
  virtual void reset() = 0;
  void receive(IN item) override = 0;
  void end() override = 0;
};

/***
 * Operator that detects all peaks in the data.
 */
template <typename IN, typename OUT> class Peaks : public Op<FDI, FSIP>
{
public:
  Receiver<FSIP> *out_op{};
  int last_value_index{};
  float last_value{};
  int top_plateau_start{};
  unsigned int rising{};

  Peaks() = default;

  void connect(Receiver<FSIP> *o) override
  {
    out_op = o;
  }

  void reset() override
  {
    last_value = 1000000000.0;
    last_value_index = -1;
    top_plateau_start = 0;
    rising = 0;
  }

  void receive(FDI item)
  {
    float cur_value = item.integ;
    int peak_index = -1;

    if (last_value < cur_value)
    {
      rising = 1;
      top_plateau_start = last_value_index;
    }
    else if (last_value > cur_value)
    {
      if (rising != 0)
      {
        // we have detected a peak because we were rising, and now we are descending.
        // last value index marks the end of the plateau
        if (last_value_index == top_plateau_start)
        {
          // if our peak consists of a single point:
          peak_index = last_value_index;
        }
        else
        {
          // if our peak has a top plateau
          peak_index = (last_value_index + top_plateau_start) / 2;
        }
        // since we have descended after previous rise, we are no longer rising
        rising = 0;
      }
    }
    FSIP to_pass = {item.fil, item.der, item.integ, peak_index};
    out_op->receive(to_pass);
    last_value_index++;
    last_value = cur_value;
  }

  void end()
  {
    out_op->end();
  }
};

/***
 * Operator that performs the last stage of Pan-Tompkins algorithm (peak detection, verification and
      back search).
 */
template <typename IN, typename OUT, unsigned int buf_size> class PT_Peaks : public Op<FSIP, int>
{
public:
  FSIP buffer[buf_size];
  // int buf_size;
  int ind{};

  int Fs{};
  int rr1[8] = {0, 0, 0, 0, 0, 0, 0, 0};
  int rr2[8] = {0, 0, 0, 0, 0, 0, 0, 0};
  double rrAvg1{}, rrAvg2{}, rrLow{}, rrHigh{}, rrMiss{};
  int lastQRS{};
  double currentSlope{};
  double lastSlope{};
  double peakI{}, peakF{}, thresholdI1{}, thresholdI2{}, thresholdF1{}, thresholdF2{}, spkI{}, spkF{},
       npkI{}, npkF{};
  bool qrs{};
  bool regular{};
  bool prevRegular{};
  Receiver<int> *out_op{};

  PT_Peaks() = default;

  void connect(Receiver<int> *o) override
  {
    out_op = o;
  }

  void reset() override
  {

    Fs = 1000;
    for (int fill_ind = 0; fill_ind < 8; fill_ind++)
    {
      rr1[fill_ind] = 0;
      rr2[fill_ind] = 0;
    }
    rrAvg1 = 0, rrAvg2 = 0, rrLow = 0, rrHigh = 0, rrMiss = 0;
    lastQRS = 0;
    currentSlope = 0;
    lastSlope = 0;
    peakI = 0, peakF = 0, thresholdI1 = 0, thresholdI2 = 0, thresholdF1 = -100000000, thresholdF2 =
         -10000000, spkI = 0,
    spkF = 0, npkI = 0, npkF = 0;
    regular = true;
    ind = 0;
  }

  void receive(FSIP item) override
  {
```

```
// advance elements in a buffer.
buffer[ind] = item;
ind = (ind + 1) % buf_size;

// if the arriving event has a non -1 peak, it is a peak candidate.
if (item.peak != -1)
{
  int mark = item.peak;
  qrs = false;
  // If the current signal is above one of the thresholds (integral or filtered), it is a peak
       candidate.
  if (buffer[(mark + buf_size) % buf_size].integ > thresholdI1 ||
      buffer[(mark + buf_size) % buf_size].fil > thresholdF1)
  {
    peakI = buffer[(mark + buf_size) % buf_size].integ;
    peakF = buffer[(mark + buf_size) % buf_size].fil;
  }
  // If both the integral and the signal are above their thresholds, they're probably signal peaks.
  if ((buffer[(mark + buf_size) % buf_size].integ >= thresholdI1) &&
      (buffer[(mark + buf_size) % buf_size].fil >= thresholdF1))
  {
    // There's a 200ms latency condition. If the new peak respects the 200 ms latency condition, we
         continue
    if (mark > lastQRS + Fs / 5)
    {
      // If it respects the 200ms latency, but it doesn't respect the 360ms latency, we check the
           slope.
      if (mark < lastQRS + (int)(0.36 * Fs))
      {
        currentSlope = 0;
        for (int j = mark - 30; j <= mark; j++)
        {
          if (buffer[(j + buf_size) % buf_size].sq > currentSlope)
          {
            currentSlope = buffer[(j + buf_size) % buf_size].sq;
          }
        }
        if (currentSlope < 0.5 * lastSlope) { qrs = false; } else { qrs = true; }
      } else { qrs = true; }
    }
  }
  // Updating Averages
  // If a R-peak was detected, the RR-averages must be updated.
  if (qrs)
  {
    // If it was above both thresholds and respects both latency periods, it certainly is a R peak.
    currentSlope = 0;
    // this for loop will look back to try get maximal slope for this QRS complex in order to
         update the next one
    for (int j = mark - 30; j <= mark; j++)
    {
      if (buffer[(j + buf_size) % buf_size].sq > currentSlope)
      {
        currentSlope = buffer[(j + buf_size) % buf_size].sq;
      }
    }
    // this seems to be a real peak -> update thresholds.
    spkI = 0.125 * peakI + 0.875 * spkI;
    thresholdI1 = npkI + 0.25 * (spkI - npkI);
    thresholdI2 = 0.5 * thresholdI1;
    spkF = 0.125 * peakF + 0.875 + spkF;
    thresholdF1 = npkF + 0.25 * (spkF - npkF);
    thresholdF2 = 0.5 * thresholdF1;
    // update slope and set qrs to true
    lastSlope = currentSlope;
```

```
      // peak detected -> output it
      out_op->receive(mark);
      // Add the newest RR-interval to the buffer and get the new average.
      rrAvg1 = 0;
      for (int i = 0; i < 7; i++)
      {
        rr1[i] = rr1[i + 1];
        rrAvg1 += rr1[i];
      }
      // calculate the newest RR interval, update last QRS tracker and update average
      rr1[7] = mark - lastQRS;
      lastQRS = mark;
      rrAvg1 += rr1[7];
      rrAvg1 *= 0.125;
      // If the newly-discovered RR-average is normal, add it to the "normal" buffer and get the new
           "normal" average.
      bool rrCheck = true;
      for (int k = 0; k < 8; k++) { if (rr1[k] < rrLow && rr1[k] > rrHigh) { rrCheck = false; } }
      // at this point we know if the beat is regular or not.
      if (rrCheck)
      {
        // set new regular beat average and update the limits:
        for (int k = 0; k < 8; k++) { rr2[k] = rr1[k]; }
        rrAvg2 = rrAvg1;
        rrLow = 0.92 * rrAvg2;
        rrHigh = 1.16 * rrAvg2;
        rrMiss = 1.66 * rrAvg2;
      } else {
        thresholdI1 /= 2;
        thresholdF1 /= 2;
      }
    }
    else
    {
      // Back Search If no R-peak was detected for too long, use the lighter thresholds and do a back
            search.
      if ((mark - lastQRS > rrMiss) && (mark > lastQRS + Fs / 5))
      {
        int peak_candidate = 0;
        int i;
        for (i = lastQRS + Fs / 5; i < mark; i++)
        {
          // searching back through all points and finding maximum that is between first and second
               sets of thresholds
          // this highest peak (in integrated waveform) is the peak candidate
          if ((buffer[(i + buf_size) % buf_size].integ > thresholdI2) &&
              (buffer[(i + buf_size) % buf_size].fil > thresholdF2))
          {
            if (peak_candidate == 0) { peak_candidate = i; }
            if (buffer[(i + buf_size) % buf_size].integ > buffer[(peak_candidate + buf_size) %
                 buf_size].integ)
            {
              peak_candidate = i;
            }
          }
        }

        // if we haven't found a peak candidate, we keep looking
        if (peak_candidate != 0)
        {
          currentSlope = 0;
          i = peak_candidate;
          for (int j = i - 30; j <= i; j++)
          {
            if (buffer[(j + buf_size) % buf_size].sq > currentSlope)
```

```
              {
                currentSlope = buffer[(j + buf_size) % buf_size].sq;
              }
            }
            if ((currentSlope < (lastSlope / 2)) && i < lastQRS + 0.36 * Fs) { } else {
              // new potential qrs was found
              peakI = buffer[(i + buf_size) % buf_size].integ;
              peakF = buffer[(i + buf_size) % buf_size].fil;
              spkI = 0.25 * peakI + 0.75 * spkI;
              spkF = 0.25 * peakF + 0.75 * spkF;
              thresholdI1 = npkI + 0.25 * (spkI - npkI);
              thresholdI2 = 0.5 * thresholdI1;
              lastSlope = currentSlope;
              thresholdF1 = npkF + 0.25 * (spkF - npkF);
              thresholdF2 = 0.5 * thresholdF1;

              // If a signal peak was detected on the back search, the RR attributes must be updated.
              rrAvg1 = 0;
              for (int j = 0; j < 7; j++)
              {
                rr1[j] = rr1[j + 1];
                rrAvg1 += rr1[j];
              }
              rr1[7] = i - lastQRS;
              // peak detected -> output it
              out_op->receive(i);
              lastQRS = i;
              rrAvg1 += rr1[7];
              rrAvg1 *= 0.125;

              // RR Average 2
              if ((rr1[7] >= rrLow) && (rr1[7] <= rrHigh))
              {
                rrAvg2 = 0;
                for (int k = 0; k < 7; k++)
                {
                  rr2[k] = rr2[k + 1];
                  rrAvg2 += rr2[k];
                }
                rr2[7] = rr1[7];
                rrAvg2 += rr2[7];
                rrAvg2 *= 0.125;
                rrLow = 0.92 * rrAvg2;
                rrHigh = 1.16 * rrAvg2;
                rrMiss = 1.66 * rrAvg2;
              }

              prevRegular = regular;
              if (rrAvg1 == rrAvg2) { regular = true;
              } else {
                regular = false;
                if (prevRegular)
                {
                  thresholdI1 /= 2;
                  thresholdF1 /= 2;
                }
              }
            }
          }
        }
      }
    }
  }

  void end()
  {
    out_op->end();
  }
};


/*** Functors used in operators. ***/
// linear difference function used in low and high passes
float lin_diff_eq(int size, float *coeffs, float *x, float *y, int ind)
{
  float value = coeffs[0] * x[ind];
  for (int i = 1; i < size; i++)
  {
    value += coeffs[i] * x[(ind - i + size) % size] + coeffs[i + size - 1] * y[(ind - i + size) % size
         ];
  }
  return value / coeffs[(size * 2) - 1];
}

// functor for low pass filter
struct low_filter_functor
{
public:
  float operator()(int size, float *x, float *y, int ind) const
  {
    // low pass at 300 Hz
    float coeff[10] = {1, 4, 6, 4, 1, -4.678, -4.067, -1.093, -0.180, 5.981602912391355};
    return lin_diff_eq(size, coeff, x, y, ind);
  }
};

// functor for high pass filter
struct high_filter_functor
{
public:
  float operator()(int size, float *x, float *y, int ind) const
  {
    // high pass at 25 Hz
    float coeff[10] = {1, -4, 6, -4, 1, 4.409, -5.958, 3.591, -0.814, 1.2281171674667124};
    return lin_diff_eq(size, coeff, x, y, ind);
  }
};

// functor for derivative pass
struct deriv_functor
{
public:
  // takes array of VT of size 5 and outputs an FD
  FD operator()(float *e) const
  {
    return {e[0], (float)(1000.0 / 8) * (-e[0] - 2 * e[1] + 2 * e[3] + e[4])};
  }
};

// functor for square pass
struct square_FD
{
public:
  FD operator()(FD fdt) const
  {
    return {fdt.fil, fdt.der * fdt.der};
  }
};

// functor for integration pass
```

```
struct integr_FD
{
public:
  FDI operator()(FD *win) const
  {
    float total = 0;
    for (int i = 0; i < INT_WIN_SIZE; i++)
    {
      total += win[i].der;
    }
    return {win[INT_WIN_SIZE / 3 + 7].fil, win[INT_WIN_SIZE / 3 + 7].der, total / (float)INT_WIN_SIZE};
  }
};
```

After that the programmer only needs to declare the query using operators and functors (both custom ones and ones provided with an SPE). Example of such a declaration is below:

```
Query-BEGIN
.MapPastN(<float, float, 5, low_filter_functor>, [low_filter_functor()], {})
.MapPastN(<float, float, 5, high_filter_functor>, [high_filter_functor()], {})
.SWinFunc(<float, FD, 5, deriv_functor>, [deriv_functor()], {})
.Map(<FD, FD, square_FD>, [square_FD()], {})
.SWinFunc(<FD, FDI, INT_WIN_SIZE, integr_FD>, [integr_FD()], {})
.Peaks(<FDI, FSIP>, [], {})
.PT_Peaks(<FSIP, int, PT_BUFFER_SIZE>, [], {})
Query-END
```