



Department of Electronic & Computer Engineering

IoT and Fall Detection – cloud alerting, data logging and visualisation

Final Year Project Report

Student Name: Dylan Coffey

Student ID: 18251382

Course: Cyber Security and IT Forensics

Academic Year: 2021/2022

Project Supervisor: Dr. Jacqueline Walker

Date: 21/3/2022

Abstract

In the industry of healthcare, elderly patients can greatly benefit from the utilisation of the Internet of Things and the cloud, specifically within the application of fall detection. IoTumble aims to provide a comfortable and wearable device to wirelessly detect possible fall incidents from a user. Worn on the waist, the device monitors its acceleration data to check for reached thresholds and user inactivity. Using the storage and compute resources of Amazon Web Services, incident data is published, stored, and processed in the cloud, alerting the necessary personnel via email. Incidents can then be visualised in readable form, exported, and further analysed in a developed graphic user interface. IoTumble is designed to be user-friendly, to enhance wearable devices that interface with the cloud, and to ultimately offer a greater possibility of detecting fall incidents. IoTumble has established that it has potential to be used in the industry of healthcare, to detect and visualise possible fall incidents.

Acknowledgements

I would like to acknowledge those who helped and supported me during this project. I would like to thank my supervisor Dr. Jacqueline Walker for her advice, guidance, and suggestions throughout the project.

I would like to thank the University staff, particularly those in the ECE Department, for educating and tutoring me throughout my four years of study. Finally, I'd like to thank my family for their encouragement and moral support, as well as for providing some materials within the project.

Declaration

I declare that this report, titled “IoT and Fall Detection – cloud alerting, data logging and visualisation”, is entirely my own work, and has been completed in part fulfilment of the requirements for the Bachelor of Science in Cyber Security and IT Forensics. Where there has been made use of work of other people it has been fully acknowledged and referenced. I can confirm that this report has not been submitted to any other University, Higher Education Institution, or for any other academic award within the University of Limerick.

Signature: Dylan Coffey

Date: 21/3/2022

Table of Contents

| | |
|--|------|
| Abstract | i |
| Acknowledgements..... | ii |
| Declaration..... | iii |
| Table of Contents..... | iv |
| List of Figures | vii |
| List of Tables | viii |
| List of Code Extracts..... | viii |
| Chapter 1. Introduction and Literature Survey..... | 1 |
| 1.1 Introduction..... | 1 |
| 1.2 Project Rationale | 1 |
| 1.3 Literature Survey | 2 |
| 1.3.1 Internet of Things | 2 |
| 1.3.2 Cloud Computing | 2 |
| 1.3.3 Fall Detection..... | 3 |
| 1.3.4 Method of Fall Detection..... | 3 |
| Chapter 2. Analytical Background..... | 4 |
| 2.1. Hardware..... | 4 |
| 2.1.1 Raspberry Pi Board | 4 |
| 2.1.2 ADXL345 Accelerometer..... | 5 |
| 2.1.3 LiPo SHIM..... | 5 |
| 2.2. Software | 6 |
| 2.2.1 Python..... | 6 |
| 2.2.2 PyCharm..... | 6 |
| 2.2.3 PyLint | 6 |
| 2.2.4 Boto3 | 6 |
| 2.2.5 AWS IoT Device SDK for Python | 7 |
| 2.2.6 Adafruit CircuitPython ADXL34x Driver | 7 |
| 2.2.7 Tkinter..... | 7 |
| 2.2.8 Matplotlib | 7 |
| 2.2.9 CSV | 7 |

| | |
|---|----|
| 2.3. Cloud Service | 8 |
| 2.3.1 AWS | 8 |
| 2.3.2 AWS IoT..... | 8 |
| 2.3.3 Amazon DynamoDB..... | 9 |
| 2.3.4 AWS Lambda..... | 9 |
| 2.3.5 Amazon Simple Notification Service..... | 9 |
| Chapter 3. Specifications and Design..... | 10 |
| 3.1. Software Development Life Cycle | 10 |
| 3.1.1 Agile Model..... | 10 |
| 3.2. Use Case Diagram..... | 11 |
| 3.3. Use Case Descriptions | 11 |
| 3.3.1 Connect..... | 11 |
| 3.3.2 Disconnect | 12 |
| 3.3.3 Fill Incidents..... | 12 |
| 3.3.4 Select Incident | 13 |
| 3.3.5 Select Graph | 14 |
| 3.3.6 Read Acceleration..... | 14 |
| 3.3.7 Create Timestamp | 15 |
| 3.3.8 Check Inactivity..... | 15 |
| 3.3.9 Store Incident | 16 |
| 3.4. Project Descriptions | 17 |
| 3.4.1 IoTumble Device | 17 |
| 3.4.2 AWS Architecture | 17 |
| 3.4.3 IoTumble Program | 18 |
| 3.5. System Architecture | 19 |
| 3.5.1 Class Identification..... | 19 |
| 3.5.2 System Class Diagram | 20 |
| 3.5.3 Communication Diagram..... | 21 |
| 3.5.4 GUI Sketches | 21 |
| 3.6. Hardware Specifications..... | 23 |
| 3.6.1 Hardware Design | 23 |
| 3.6.2 Bill of Materials..... | 24 |

| | |
|---|----|
| Chapter 4. Implementation | 25 |
| 4.1. IoTumble Device..... | 25 |
| 4.1.1 Assembly..... | 25 |
| 4.1.2 Wearability | 26 |
| 4.1.3 Connecting to AWS..... | 27 |
| 4.1.4 Reading Acceleration and Creating Timestamps..... | 28 |
| 4.1.5 Checking for Thresholds | 29 |
| 4.1.6 Checking for Inactivity | 30 |
| 4.1.7 Publishing an Incident | 31 |
| 4.2. AWS Architecture | 32 |
| 4.2.1 AWS IoT..... | 32 |
| 4.2.2 DynamoDB..... | 34 |
| 4.2.3 AWS Lambda..... | 35 |
| 4.2.4 Amazon SNS..... | 37 |
| 4.3. IoTumble Program..... | 38 |
| 4.3.1 MVC Pattern | 38 |
| 4.3.2 Connecting and Disconnecting | 39 |
| 4.3.3 Filling Incidents | 40 |
| 4.3.4 Selecting an Incident | 41 |
| 4.3.5 Filling Details..... | 42 |
| 4.3.6 Selecting a Graph..... | 44 |
| 4.3.7 Exporting Graph and CSV..... | 45 |
| 4.3.7 GUI Design | 47 |
| Chapter 5. Testing and Results | 48 |
| 5.1. IoTumble Device..... | 48 |
| 5.2. AWS Architecture | 49 |
| 5.3. IoTumble Program..... | 50 |
| Chapter 6. Discussion and Conclusion | 52 |
| 6.1. Discussion | 52 |
| 6.2. Conclusion | 53 |
| References | 54 |
| Appendices..... | 57 |

| | |
|-----------------------------------|----|
| Appendix A – Source Code | 57 |
| Appendix B – Copy of Poster | 58 |

List of Figures

| | |
|---|----|
| Figure 1: Raspberry Pi Zero 2 W..... | 4 |
| Figure 2: ADXL345 Accelerometer | 5 |
| Figure 3: LiPo SHIM | 5 |
| Figure 4: Agile Model | 10 |
| Figure 5: Use Case Diagram | 11 |
| Figure 6: Package Diagram of the IoTumble Program | 19 |
| Figure 7: System Class Diagram | 20 |
| Figure 8: Communication Diagram of the IoTumble Program | 21 |
| Figure 9: GUI Sketch of HomeView | 21 |
| Figure 10: GUI Sketch of IncidentView | 22 |
| Figure 11: Pinout of Raspberry Pi Zero 2 W..... | 23 |
| Figure 12: Pinout of ADXL345 using stripped wires..... | 24 |
| Figure 13: Hardware of the IoTumble Device..... | 25 |
| Figure 14: Repurposed Smartphone Battery | 25 |
| Figure 15: Casing holding the IoTumble Device..... | 26 |
| Figure 16: Top of the casing showing the JST Connector | 26 |
| Figure 17: Camera holder storing the casing and battery | 26 |
| Figure 18: Camera holder strap for attaching to a belt | 26 |
| Figure 19: Front and side view of the attached IoTumble Device..... | 26 |
| Figure 20: File for storing the necessary AWS credentials | 27 |
| Figure 21: AWS Architecture Diagram | 32 |
| Figure 22: Created IoT Thing..... | 32 |
| Figure 23: Created IoT Policy | 33 |
| Figure 24: Created IoT Rule..... | 33 |
| Figure 25: Created DynamoDB Table..... | 34 |
| Figure 26: Timestamps in JSON Form | 34 |
| Figure 27: Created DynamoDB Stream | 35 |

| | |
|--|----|
| Figure 28: Filter criteria of the Lambda function | 35 |
| Figure 29: Created SNS Topic..... | 37 |
| Figure 30: Email subscription to the SNS Topic | 37 |
| Figure 31: HomeView before connection | 47 |
| Figure 32: HomeView after connection | 47 |
| Figure 33: IncidentView filled with details and a selected graph | 47 |
| Figure 34: Debug print statements for the IoTumble Device | 48 |
| Figure 35: Graphs visualising falling forward, backward, and to the right..... | 48 |
| Figure 36: Jumping up and down followed by inactivity | 49 |
| Figure 37: Amazon SNS email showcasing incident details | 50 |
| Figure 38: Final PyLint code rating | 50 |
| Figure 39: Details tree view displaying timestamps | 51 |
| Figure 40: Exported graph | 51 |
| Figure 41: Exported CSV..... | 51 |

List of Tables

| | |
|---|----|
| Table 1: Connect Use Case Description | 11 |
| Table 2: Disconnect Use Case Description..... | 12 |
| Table 3: Fill Incidents Use Case Description | 12 |
| Table 4: Select Incident Use Case Description | 13 |
| Table 5: Select Graph Use Case Description | 14 |
| Table 6: Read Acceleration Use Case Description | 14 |
| Table 7: Create Timestamp Use Case Description | 15 |
| Table 8: Check Inactivity Use Case Description | 15 |
| Table 9: Store Incident Use Case Description | 16 |
| Table 10: Class Identification | 19 |
| Table 11: Bill of Materials | 24 |

List of Code Extracts

| | |
|---|----|
| Code Extract 1: Connect and MQTT Configure methods..... | 27 |
|---|----|

| | |
|---|----|
| Code Extract 2: Read Accelerometer and Create Timestamp methods | 28 |
| Code Extract 3: Calculate Signal Vector Magnitude method | 28 |
| Code Extract 4: Record Timestamp method..... | 29 |
| Code Extract 5: Check Thresholds and Threshold Reached methods | 29 |
| Code Extract 6: Check Inactivity and Get Post Threshold methods | 30 |
| Code Extract 7: MQTT Publish, Create Payload, Request Incident Count methods..... | 31 |
| Code Extract 8: Lambda function obtaining the incident data | 36 |
| Code Extract 9: Lambda function sending an Amazon SNS email | 36 |
| Code Extract 10: Lambda function incrementing the incident count item | 37 |
| Code Extract 11: Structure of a Controller Class | 38 |
| Code Extract 12: Structure of a View Class..... | 38 |
| Code Extract 13: Structure of a Model Class | 38 |
| Code Extract 14: Connect and Disconnect methods in HomeView..... | 39 |
| Code Extract 15: Connect and Disconnect methods in HomeController | 39 |
| Code Extract 16: Connect, Disconnect, and Create Table methods in Session | 40 |
| Code Extract 17: Fill Incidents method in HomeController | 40 |
| Code Extract 18: Request Incident Count method in Session | 41 |
| Code Extract 19: Fill Incidents method in HomeView | 41 |
| Code Extract 20: Switch method in HomeView | 41 |
| Code Extract 21: Switch method in HomeController | 42 |
| Code Extract 22: Request Incident method in Session | 42 |
| Code Extract 23: Fill Details method in IncidentController | 43 |
| Code Extract 24: Get Max Timestamp method in Incident | 43 |
| Code Extract 25: Fill Details Labels and Tree View methods in IncidentView..... | 43 |
| Code Extract 26: Select Graph method in IncidentView | 44 |
| Code Extract 27: Fill Graph method in IncidentController | 44 |
| Code Extract 28: Plot Graph and Set Graph methods in IncidentView | 45 |
| Code Extract 29: Export Graph method in IncidentController | 45 |
| Code Extract 30: Export Graph method in IncidentView..... | 46 |
| Code Extract 31: Export Timestamps method in Incident | 46 |
| Code Extract 32: Docstring for a method | 50 |

Chapter 1. Introduction and Literature Survey

1.1 Introduction

With the ever-growing implementation of the Internet of Things (IoT) and the cloud within everyday devices, its development has now become an important and essential component to many industries. Particularly within the industry of healthcare, where new solutions are in constant development to help its patients. Healthcare and its many solutions can greatly benefit from this, specifically within wearable technology and the application of fall detection.

This is an application that could be used by elderly patients, who may live alone, and are especially prone to fall unexpectedly, due to their age or other health conditions. Detecting when a patient falls, and alerting the necessary personnel in a timely manner, could be a matter of life or death for the patient. Therefore, it can be said that fall detection is of great importance as an application of healthcare.

1.2 Project Rationale

In the project, the student will investigate the application of fall detection in wearable IoT devices, that interface with the cloud. The student will need to study IoT, cloud computing, the possible methods of fall detection, and the technologies that exist to support it. The student can then develop a wearable device for this application, using a computer board and additional hardware. The cloud architecture must be built to store the devices data, and a graphic user interface (GUI) program can be developed to interpret it.

The project will be an investigation into the application of IoT, cloud computing, and fall detection. The ultimate objective is to develop a comfortable and wearable IoT device which can detect possible fall incidents from a user. The project should interface the device with a chosen cloud service, store any collected data, alert the necessary personnel, and then visualise the data in readable form via a developed GUI program. By the end of the project, the device should enhance the application of fall detection, wearable IoT devices, and be comfortable enough to be worn and used by patients.

1.3 Literature Survey

1.3.1 Internet of Things

The Internet of Things (IoT) can be described from its name alone. It is quite literally the “things” that connect to the internet and each other. These things are typically devices such as a computer, tablet, smartphone, or just about anything that could connect to the internet. These devices could connect physically through wires, or with the use of wireless technologies such as Wi-Fi, or Bluetooth. IoT devices typically contain sensors, which obtain some form of data [1].

This data is then sent to another device, server, or cloud service to be interpreted. It could be used to enable processes from across the world, or from just across the room. Sensors today can detect and obtain tiny measurements in many different applications. Some examples include detecting movement in objects and structures or even measuring small concentrations of pollution or toxic substances in the air. Overall, sensors are essentially the “magic” or the main functionality that allows IoT devices to work [1].

1.3.2 Cloud Computing

The National Institute of Standards and Technology (NIST) defines cloud computing as *“a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.”* [2].

In simpler terms, cloud computing is an on-demand delivery of IT resources, over the Internet through a pay-as-you-go pricing model [3]. Cloud computing can be separated into two perspectives, virtualisation, and cloud services. Virtualisation is the technical perspective, dealing with the actual virtualisation of applications and servers. Whereas cloud services are the conceptual perspective, implementing the actual business process through service models [2]. There are three different service models, the first being infrastructure as a service (IaaS), which typically provides access to networking features, and data storage space [3].

The second model is platform as a service (PaaS), which provides a platform, complete with hardware and an operating system. The last model is software as a service (SaaS), which provides the software to be used by the cloud platform [2]. Each service model caters to the user of the cloud and their needs. In the context of this project, a cloud service will need to be chosen which provides the necessary features for the project.

1.3.3 Fall Detection

Falls are a major cause of injury for the elderly and is one of the most frequent causes of death for individuals over the age of 65. Therefore, the development of fall detection systems should be of major importance in the industry of healthcare. Fall detection in wearable devices could be defined as a real-time system, worn by an individual, which uses acceleration sensors to monitor changes in speed and descent. Other fall detection methods exist, such as machine learning and image-based detection, however they do not fit the context of this project. Wearable fall detection systems offer a great possibility to help elderly people. With the use of low-powered, and cost-efficient components, they can help realize an affordable but reliable wearable device [4].

1.3.4 Method of Fall Detection

There are many different methods of fall detection in wearable devices. Existing studies focus on putting fixed accelerometer sensors on the body of the user. A fall is determined by analysing the values of the x, y, and z acceleration. These values can indicate when the body of the user has moved downwards and has exceeded a certain descent limit. An existing study suggests an algorithm that analyses the Signal Vector Magnitude (SVM) to detect a fall [4]:

$$SVM = \sqrt{(A_x)^2 + (A_y)^2 + (A_z)^2}$$

Where A_x , A_y , and A_z are the acceleration in the x, y, and z-axes, respectively [4].

When the user of the wearable device moves, the SVM value changes. The value can be monitored and if it increases too quickly or goes over a certain threshold, it is possible to detect a fall incident. This can be analysed further by combining sensor data at the time of the fall [4] to check if the user has become inactive. In the context of this project, this algorithm and method could be implemented in code to detect a possible fall incident.

Chapter 2. Analytical Background

In this chapter an analytical background has been made into the available technologies and services to help the development of the project. The following sections discuss and investigate the hardware, software, and cloud service being used by the project.

2.1. Hardware

2.1.1 Raspberry Pi Board

The Raspberry Pi is a family of small, low-cost computer boards, which allow people to explore computing and learn how to program in languages such as Scratch and Python [5]. One of their latest boards available is the Raspberry Pi 4 Model B [6], however it isn't very ideal as a wearable device due to its size. There are other Raspberry Pi boards available that are smaller and more suitable for that application, while also offering wireless capabilities.

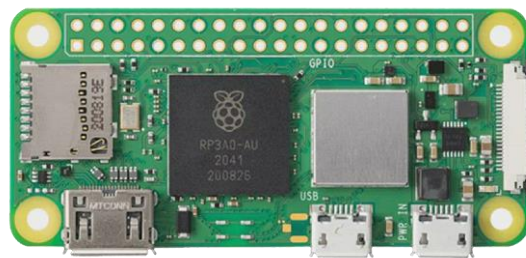


Figure 1: Raspberry Pi Zero 2 W [obtained from Raspberrypi.com]

The Raspberry Pi Zero 2 W is an ultra-compact, cheap computer board that offers wireless connectivity, such as Bluetooth, BLE, and Wi-Fi. It incorporates a single-core 1GHz processor, 512MB RAM, a mini-HDMI port, 2 micro-USB ports, 40-pin HAT-compatible header, and is powered through one of the micro-USB ports [6]. The board measures 65mm in length and 30mm in width [7], which makes it very suitable as a wearable device.

The board doesn't provide a power supply, or an operating system installed. The operating system needs to be installed in a Micro SD card, which can be done using the Raspberry Pi Imager [6]. This board incorporates all the necessary features that are needed for the project. It's suitable as a wearable device, it's low-cost, and it offers wireless connectivity.

2.1.2 ADXL345 Accelerometer

The ADXL345 is a small, thin, ultra-low power, 3-axis accelerometer, with a high-resolution measurement of up to $\pm 16g$. Digital output data is formatted as 16-bit two's complement and is accessible through either an SPI or I2C digital interface. It requires 2.0V to 3.6V in voltage supply and consumes as low as $23\mu A$ in measurement mode, and $0.1\mu A$ in standby mode. The sensor can measure the static acceleration of gravity, as well as dynamic acceleration. Its high-resolution measurement can capture inclination changes of less than 1 degree [8].



Figure 2: ADXL345 Accelerometer [edited but originally obtained from Adafruit.com]

The sensor offers several special functions, such as detecting the presence, or lack of motion by comparing the acceleration with user thresholds. Other functions include detecting single or double taps to the sensor, and if the sensor is in a free-fall. The sensor can survive up to $10000g$ of high shock and is guaranteed to operate in a temperature range of $-40^{\circ}C$ to $+85^{\circ}C$ [8].

2.1.3 LiPo SHIM

The LiPo SHIM is a small PCB that allows for the power supply of a Raspberry Pi board. It provides a 2-pin JST connector for a battery and supplies 1.5A of current. It sends a low battery warning at 3.4V and an automatic shutdown of the board at 3.0V.

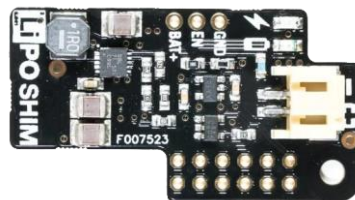


Figure 3: LiPo SHIM [edited but originally obtained from Thepihut.com]

The board is only 0.8mm thick and is designed to be ultra-compact. It is shaped to sit as low as possible on the board, which is perfectly ideal for a wearable device [13].

2.2. Software

2.2.1 Python

Python is a general-purpose, high-level, dynamic programming language that offers a lot of functionality and libraries to help develop projects. It is generally considered the more popular language used in Raspberry Pi boards. Additionally, it provides libraries such as Matplotlib and Tkinter, which can be used to develop a Graphic User Interface (GUI) to plot graphs. Python is also supported by AWS, as they offer an SDK to help users program their devices to interact with the cloud.

2.2.2 PyCharm

PyCharm is a dedicated Python Integrated Development Environment (IDE) providing a wide range of essential tools and libraries for Python developers. The IDE is tightly integrated to create a convenient environment for productive Python development. It is one of the leading IDEs for Python and offers a community edition, which is free and open source [14].

2.2.3 PyLint

PyLint is a tool that checks for errors in Python code, tries to enforce the PEP 8 coding standard, and looks for code smells. The tool can be installed as a plugin in PyCharm and can be used while developing a project. It will display messages in real time and make suggestions on ways to improve code. These suggestions can be broken down into different categories including errors, warnings, and conventions. Lastly, it will give a project an overall score which is based on the number of warnings, errors, and conventions that remain in its code [15].

2.2.4 Boto3

Boto3 is a provided AWS SDK for Python which allows for the integration of a Python program with AWS and its services, such as Amazon S3, Amazon EC2, Amazon DynamoDB, and more. Boto3 provides two distinct levels of APIs, a low-level API called Client, and a high-level API called Resource. Both APIs are driven by JSON models, as they allow for fast updates with strong consistency across all supported services on AWS [12].

2.2.5 AWS IoT Device SDK for Python

The AWS IoT Device SDK for Python allows for the development of scripts for devices. The SDK provides access to the AWS IoT platform using the MQTT protocol. Users can securely work with the rules and message broker of AWS IoT to send messages to different AWS services such as Amazon S3, Amazon DynamoDB, AWS Lambda, and more. [16].

2.2.6 Adafruit CircuitPython ADXL34x Driver

The Adafruit CircuitPython ADXL34x Driver is a library for the ADXL34x family of accelerometers. The driver allows for the quick and easy use of the ADXL345 accelerometer and its functions. The library works in tandem with CircuitPython's "board" library, which uses I2C to communicate with the Raspberry Pi board and its accelerometer [17].

2.2.7 Tkinter

Tkinter is the standard Python package to interface with the Tcl/Tk GUI toolkit. It was created as a wrapper around the Tcl interpreter and converts Python code into Tcl code. Tkinter can create GUI windows containing widgets such as frames, labels, buttons, entries, and more. These widgets can then be bound to methods within code and can be styled using the themed Tk (ttk) variations of the widgets [18].

2.2.8 Matplotlib

Matplotlib is a library which allows for the creation of static, animated, and interactive plot visualisations in Python. The library allows developers to plot a dataset and visualise it into a graph. This can then be displayed and embedded into a GUI or be exported into many different file formats [19].

2.2.9 CSV

A Comma Separated Values (CSV) file is a text file that contains a list of data in a tabular format. It can be used by applications that contain a database or some form of tabular dataset. It contains a header line consisting of its different column names and more lines consisting of the separated data, which are in line with their specific column names. Python contains a "csv" module which can be used to read and write CSV files in code.

2.3. Cloud Service

2.3.1 AWS

Amazon Web Services (AWS) is one most comprehensive and popular cloud platforms available today. It provides over 200 fully featured services from data centers globally. AWS is broken down into 25 geographic regions, and 81 availability zones, with data centers that connect to be one of the most extensive cloud infrastructures globally [9].

AWS was first launched in 2006 as an infrastructure to handle Amazon's online retail operations. Eventually, Amazon introduced a pay-as-you-go model through a mixture of the previously mentioned service models, IaaS, PaaS, and SaaS. AWS is now used by software developers, enterprises, government agencies, education institutions, and many other organisations around the world [10].

The cloud service provides a platform with services such as compute, storage, database management, networking, and many more. Some of these include, the Amazon Elastic Compute Cloud (Amazon EC2), for compute capacity. The Amazon Simple Storage Service (Amazon S3), which provides scalable object storage, and Amazon DynamoDB which offers a managed NoSQL database [10].

2.3.2 AWS IoT

AWS IoT provides services to support IoT devices. The AWS IoT core services allow a device to connect to the AWS so that other services and applications within the cloud can interact with it. The main core services are the messaging services, as this secures communication between the device and AWS [11].

A device connects to AWS IoT through the device gateway, which enables it to communicate securely and efficiently through a certain protocol. The message broker provides a mechanism between the device and AWS IoT applications to publish and receive messages from each other, which can be achieved using the MQTT protocol. Lastly, the rules engine connects data from the message broker to other services like Amazon S3 or Amazon DynamoDB [11].

AWS IoT allows devices to connect with the cloud by providing IoT device and mobile SDKs. These include open-source libraries, and developer guides to help you program your device to interact with the cloud [11].

2.3.3 Amazon DynamoDB

Amazon DynamoDB is a managed NoSQL database service provided through AWS. It is a scalable, fast, and predictable database that can store and retrieve any amount of data, at high levels of traffic. DynamoDB offers many features such as the ability to backup data, scaling capacity on demand, and the streaming of data to other AWS services [20].

DynamoDB Streams capture modifications to a DynamoDB table and stores them in a log for up to 24 hours. Whenever an item is created, updated, or deleted, a stream record is written to the log with the primary key attributes of the modified item [21]. Other services like AWS Lambda, can then perform different actions depending on these modifications.

2.3.4 AWS Lambda

AWS Lambda is a compute service that allows for the execution of code within the cloud, without the need for the provisioning or management of servers. Code can be organised into Lambda functions which will scale automatically and only run when needed, from a few requests a day to thousands per second [22].

Lambda functions could be used as data-processing triggers for other AWS services such as Amazon S3 or Amazon DynamoDB [22]. For example, when a DynamoDB Stream is triggered, a Lambda function could send an SNS notification or make even further changes to a table.

2.3.5 Amazon Simple Notification Service

Amazon Simple Notification Service (SNS) is a managed service that allows for the delivery of messages from a publisher to a subscriber. Publishers can communicate with subscribers by sending a message to a SNS topic. Subscribers can subscribe to this topic and receive published messages using a supported endpoint type, such as an email or mobile text message [23].

Chapter 3. Specifications and Design

In this chapter the projects specifications and design have been defined. The following sections look at the methodology that was chosen for development, the projects use cases and descriptions, including the device, AWS architecture, and program. Lastly, the system architecture and hardware specifications have been defined.

3.1. Software Development Life Cycle

A Software Development Life Cycle (SDLC) model is a sequence of phases that are used during the development of software. SDLC models can be described as tools to help the better delivery of a software development project. The models focus on realising a project's requirements into design, implementation, and finally testing [24]. For the development of the project, it became clear that a flexible methodology needed to be incorporated to allow for its completion before the deadline. Therefore, the correct SDLC model for the project needed to be carefully considered.

3.1.1 Agile Model

The Agile model is considered an iterative and incremental approach, focusing on the development of time critical applications [24]. The model responds to change over the course of a project, allowing for the return to previous phases where necessary. Functional requirements which may not be clear or fully realised through documentation can be developed at any time. This decreases the duration of time that is needed to complete major features of a project [24].

The model is an ideal approach to develop the project in the short time span that has been given. The model is very flexible, as requirements and changes can be implemented at any time, which will be necessary to allow the projects many features to work together properly.



Figure 4: Agile Model [obtained from Google Images]

3.2. Use Case Diagram

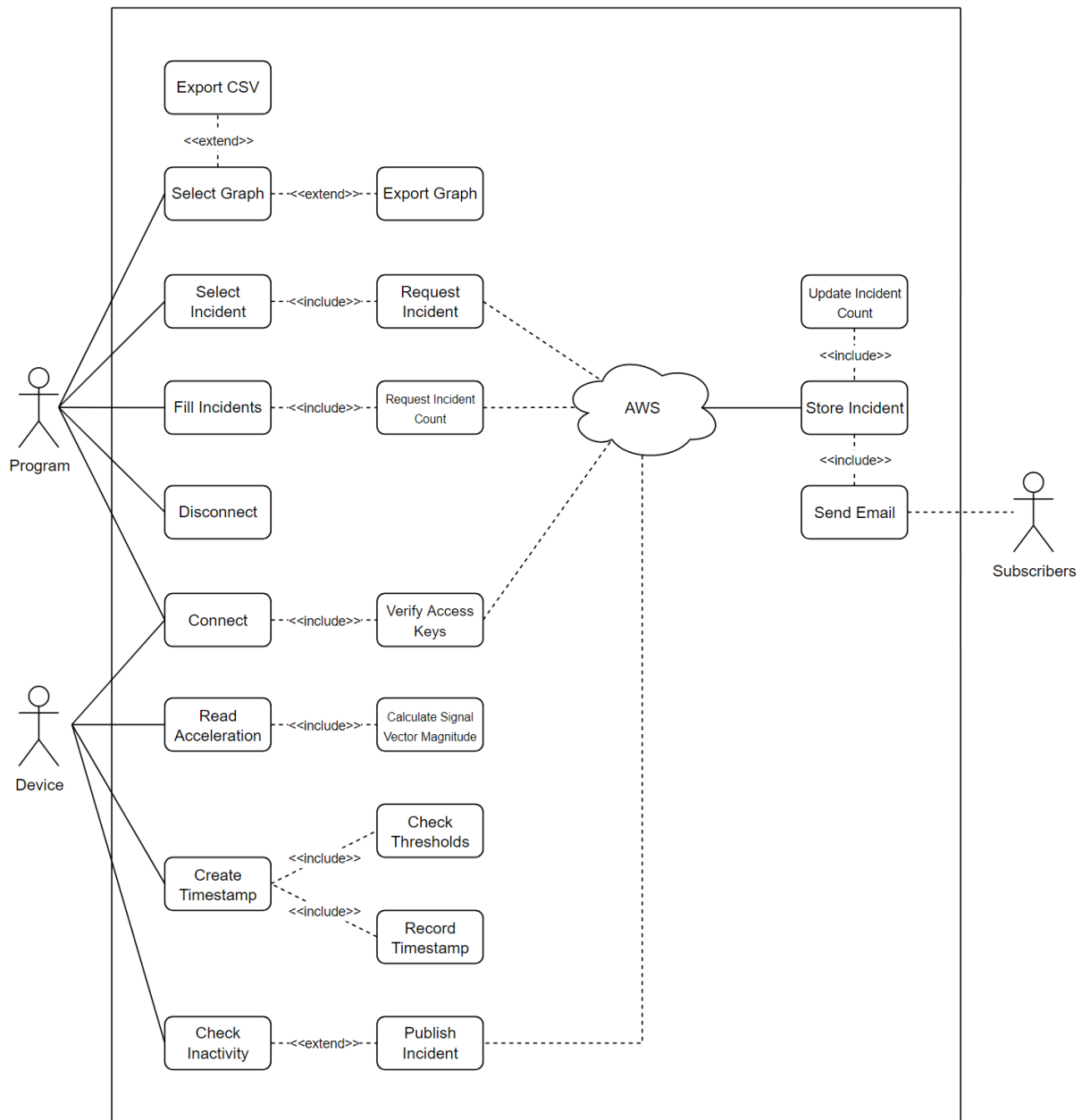


Figure 5: Use Case Diagram [created using Diagrams.net]

3.3. Use Case Descriptions

3.3.1 Connect

Table 1: Connect Use Case Description

| Use Case 1 | Connect |
|---------------------|--|
| Goal in Description | Actor connects to AWS using credentials. |
| Precondition(s) | Actor must have access to credentials. |

| | | |
|---------------------------------|--|---|
| Successful Postcondition | Actor connects to AWS and can access its permitted services. | |
| Failed Postcondition | Actor fails to connect to AWS. | |
| Actor(s) | Program and Device. | |
| Description | Step | Action |
| | 1 | For the program credentials can be inputted, for the device (and optionally for the program) credentials are read from a hidden file. |
| | 2 | Actor attempts to verify that the credentials are correct. |
| | 3 | Actor connects to AWS. |
| Extension(s) | Step | Action |
| | 2a | If the inputted credentials are incorrect the program displays an error pop-up, while the device raises an error. |

3.3.2 Disconnect

Table 2: Disconnect Use Case Description

| | | |
|---------------------------------|---|-------------------------------------|
| Use Case 2 | Disconnect | |
| Goal in Description | Actor disconnects from AWS. | |
| Precondition(s) | Actor must already be connected to AWS. | |
| Successful Postcondition | Actor disconnects from AWS and can no longer access its permitted services. | |
| Failed Postcondition | Actor fails to disconnect from AWS. | |
| Actor(s) | Program. | |
| Description | Step | Action |
| | 1 | Actor clicks the disconnect button. |
| | 2 | Actor disconnects from AWS. |

3.3.3 Fill Incidents

Table 3: Fill Incidents Use Case Description

| | | |
|----------------------------|--|--|
| Use Case 3 | Fill Incidents | |
| Goal in Description | Actor fills the incidents list of its GUI. | |
| Precondition(s) | Actor must be connected to AWS. | |

| | | |
|---------------------------------|--|--|
| Successful Postcondition | Actor fills the incidents list of its GUI by requesting the current count of incidents stored in the AWS service DynamoDB. | |
| Failed Postcondition | Actor fails to fill the incidents list or retrieve the current incident count. | |
| Actor(s) | Program. | |
| Description | Step | Action |
| | 1 | Actor retrieves the current incident count. |
| | 2 | Actor fills the incident list in its GUI with the retrieved amount. |
| Extension(s) | Step | Action |
| | 1a | The current incident count is requested from the AWS service DynamoDB. |

3.3.4 Select Incident

Table 4: Select Incident Use Case Description

| | | |
|---------------------------------|--|---|
| Use Case 4 | Select Incident | |
| Goal in Description | Actor selects an incident to view in its GUI. | |
| Precondition(s) | Actor must be connected to AWS and have filled the incidents list of its GUI. | |
| Successful Postcondition | Actor selects an incident, the incident is retrieved from AWS, and is displayed in a new GUI window. | |
| Failed Postcondition | Actor cannot select an incident from its GUI, or the incident cannot be retrieved from AWS. | |
| Actor(s) | Program. | |
| Description | Step | Action |
| | 1 | Actor scrolls through a list of incidents. |
| | 2 | Actor selects an incident. |
| | 3 | Actor retrieves the incident and displays it in a new GUI window. |
| Extension(s) | Step | Action |
| | 2a | The selected incident is requested from DynamoDB. |

3.3.5 Select Graph

Table 5: Select Graph Use Case Description

| | | |
|---------------------------------|---|---|
| Use Case 5 | Select Graph | |
| Goal in Description | Actor selects a graph to view in its GUI. | |
| Precondition(s) | Actor must have selected and requested an incident. | |
| Successful Postcondition | Actor selects a graph, which is displayed in its GUI. | |
| Failed Postcondition | Actor fails to select a graph. | |
| Actor(s) | Program. | |
| Description | Step | Action |
| | 1 | Actor selects a graph to display. |
| | 2 | The retrieved incident is used for the graphs data. |
| | 3 | The selected graph displays in the GUI. |
| Extension(s) | Step | Action |
| | 3a | The selected graph can then export its data in CSV format or be exported as a PNG file. |

3.3.6 Read Acceleration

Table 6: Read Acceleration Use Case Description

| | | |
|---------------------------------|--|---|
| Use Case 6 | Read Acceleration | |
| Goal in Description | Actor reads the acceleration data of its accelerometer. | |
| Precondition(s) | Actor must be started. | |
| Successful Postcondition | Actor retrieves the acceleration data of its accelerometer and calculates its Signal Vector Magnitude (SVM). | |
| Failed Postcondition | Actor is unable to retrieve the acceleration data or calculate its SVM. | |
| Actor(s) | Device. | |
| Description | Step | Action |
| | 1 | Actor attempts to read the acceleration data. |
| | 2 | Acceleration data is retrieved. |
| Extension(s) | Step | Action |
| | 2a | Acceleration data is used to calculate its SVM. |

3.3.7 Create Timestamp

Table 7: Create Timestamp Use Case Description

| | | |
|---------------------------------|---|---|
| Use Case 7 | Create Timestamp | |
| Goal in Description | Actor creates a timestamp. | |
| Precondition(s) | Actor must have read the acceleration data of its accelerometer and calculated its SVM. | |
| Successful Postcondition | Actor creates a timestamp, records it, and checks if it has reached a threshold. | |
| Failed Postcondition | Actor cannot create a timestamp, or record it, or is unable to check for a reached threshold. | |
| Actor(s) | Device. | |
| Description | Step | Action |
| | 1 | Actor attempts to create a timestamp using the retrieved acceleration data. |
| | 2 | A timestamp is created. |
| Extension(s) | Step | Action |
| | 2a | The timestamp is recorded. |
| | 2b | The timestamps data is checked to see if it has reached a threshold. |

3.3.8 Check Inactivity

Table 8: Check Inactivity Use Case Description

| | | |
|---------------------------------|---|---|
| Use Case 8 | Check Inactivity | |
| Goal in Description | Actor checks to see if it has become inactive. | |
| Precondition(s) | Actor must first have checked that the data of a created timestamp has reached a threshold. | |
| Successful Postcondition | Actor concludes if it has become inactive. | |
| Failed Postcondition | Actor cannot conclude if it has become inactive. | |
| Actor(s) | Device. | |
| Description | Step | Action |
| | 1 | Actor checks if it has become inactive. |

| | | |
|---------------------|-------------|--|
| | 2 | Actor makes a conclusion on if it has become inactive. |
| Extension(s) | Step | Action |
| | 2a | If the actor has become inactive, the recorded timestamps are used to publish an incident to the AWS service DynamoDB. |

3.3.9 Store Incident

Table 9: Store Incident Use Case Description

| | | |
|---------------------------------|---|--|
| Use Case 9 | Store Incident | |
| Goal in Description | Actor stores an incident in its service DynamoDB. | |
| Precondition(s) | An incident must have been published to AWS. | |
| Successful Postcondition | Actor stores an incident in its DynamoDB table, updates the incident count item, and sends an alert email to all its subscribers. | |
| Failed Postcondition | Actor is unable to store an incident in a DynamoDB table, or update the incident count item, or send an alert email to all its subscribers. | |
| Actor(s) | AWS. | |
| Description | Step | Action |
| | 1 | Actor retrieves a published incident. |
| | 2 | Actor attempts to store the incident in its DynamoDB table. |
| | 3 | Actor stores the incident. |
| Extension(s) | Step | Action |
| | 3a | A DynamoDB stream is then sent to a Lambda function to update the incident count item and send a SNS email to all subscribers of an SNS topic. |

3.4. Project Descriptions

3.4.1 IoTumble Device

The software of the IoTumble device first connects to AWS IoT using the AWS IoT Device SDK for Python. Certificates that were created within AWS IoT are used to configure access. It then begins reading the acceleration data of its accelerometer using the Adafruit CircuitPython ADXL34x Driver. The SVM is calculated from the acceleration data and a timestamp is created and recorded.

The created timestamp is checked for reached thresholds, and if the thresholds have been reached, it checks for inactivity. If inactivity is detected, a JSON payload of the incident is created from the devices recorded timestamps. This payload is then published as a MQTT message and sent to an IoT topic that was created within AWS IoT.

3.4.2 AWS Architecture

For the AWS architecture, an AWS root account must first be created. The account allows outside access to its services using AWS Identity and Access Management (IAM) roles. Each role creates credentials that can be used for connecting to AWS, consisting of an access key ID and secret access key. The role can also specify its permitted services, with the IoTumble program only having DynamoDB read access.

The architecture itself consists of an IoT rule within AWS IoT, that listens for published MQTT messages containing a certain IoT topic. When such a MQTT message is obtained, an IoT action inserts its payload into a DynamoDB table. Each item within the table holds the incidents ID, item type, and the data of its timestamps.

Once an item is inserted into the table, a DynamoDB stream is sent to a Lambda function in AWS Lambda. The function updates and increments the count item in the table and sends an Amazon SNS email to alert all subscribers of the new incident. The email contains the incident ID, date, time, and the data of the timestamp with the highest SVM.

3.4.3 IoTumble Program

The software of the IoTumble program consists of a GUI, programmed in Python using the Tkinter package. The program first displays a home view for the user to input the access key ID, secret access key, and region name to connect to AWS. With the inputted credentials, the program creates a session using the AWS SDK for Python (Boto3).

This session verifies that the credentials are valid, connects itself to AWS, and requests the current incident count from a DynamoDB table. The program then fills an incident list in the home view with the retrieved incident count. The user can scroll through this incident list and select one to view. When an incident is selected, the session requests the selected incident, hides the home view, and opens a new view for the incident.

The incident view displays its timestamp data in tabular format and as graphs. A timestamp consists of their X-Acceleration, Y-Acceleration, Z-Acceleration, and SVM. The timestamp with the highest SVM of the incident is used in the text of the incident details. This includes its date, time, and data.

The user can then select a graph to plot from a dropdown box, with the selection of each timestamp data type. The graph is displayed with the acceleration/SVM on the Y-axis, and the X-axis containing the time. The user can then export the selected graph as a PNG file or the incident timestamps in CSV format.

3.5. System Architecture

The project incorporates software that follows the object-oriented paradigm and its many principles, such as encapsulation, polymorphism, modularity, and inheritance. To achieve this, the software needed to be designed through requirement analysis. In this section, the previously defined project use cases and descriptions are used to design the system architecture, creating its class diagram, communication diagram, and GUI sketches.

As the IoTumble program was incorporating a GUI, it was decided to separate its business logic from its presentation logic by using Model-View-Controller (MVC). In the MVC design pattern, the controller classes act as a bridge between the model and view classes, to communicate any necessary information that needs to be displayed. This creates loose coupling by removing the dependency of the views, making it easier to modify and faster to develop.

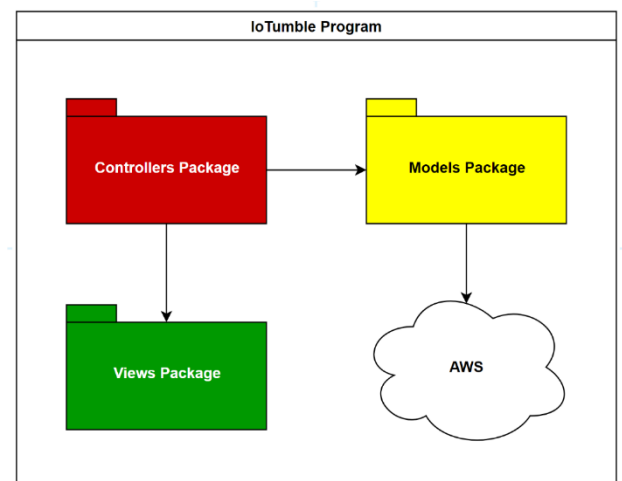


Figure 6: Package Diagram of the IoTumble Program
[created using Diagrams.net]

3.5.1 Class Identification

The classes below have been identified from the project descriptions, by using the noun identification method. Many classes were not chosen because they were either too broad, too similar, or in some cases just weren't necessary as classes.

Table 10: Class Identification

| IoTumble | Device | Certificates | Acceleration |
|---------------|------------|--------------|--------------|
| Accelerometer | Data | SVM | Timestamp |
| Threshold | Inactivity | Payload | Incident |
| Message | Program | HomeView | Credentials |
| Session | AWS | Table | IncidentView |
| Graph | Details | Date | Time |

In the case of the IoTumble device, it was decided that its software could be contained within the one Device class, because all its use cases interacted with each other. For the IoTumble program, its use cases were limited so it didn't need a lot of classes. Session, Timestamp, and Incident is all that was necessary to contain the programs necessary functionality.

3.5.2 System Class Diagram

The below diagram shows a concept of the systems structure, describing the relationship between each class, and outlining their methods and attributes. This diagram was used as a basis during the development of the project's software.

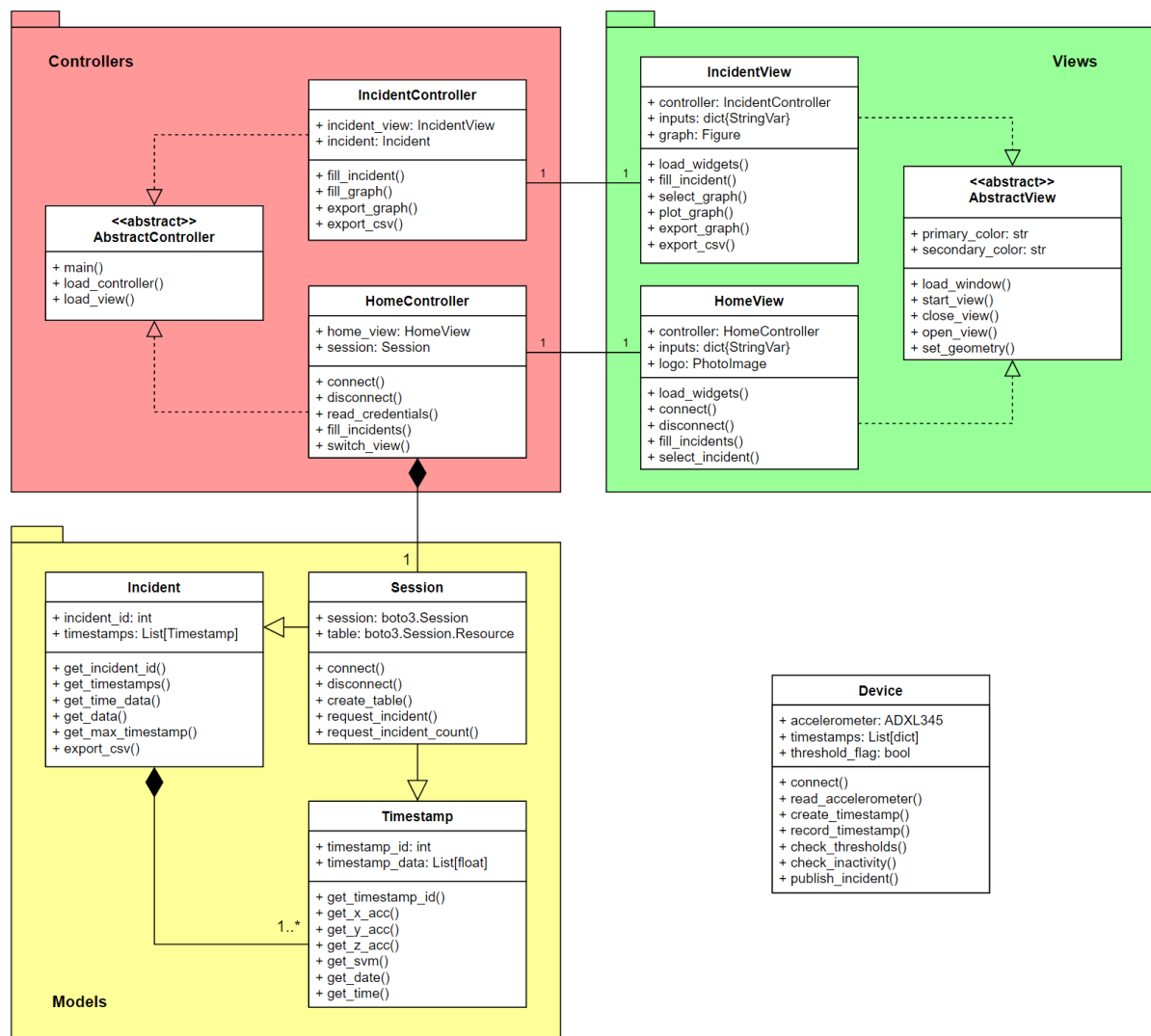


Figure 7: System Class Diagram [created using Diagrams.net]

3.5.3 Communication Diagram

The below diagram shows the interaction between the objects of the IoTumble program. A communication diagram for the IoTumble device wasn't necessary as it contains a single class that only interacts with itself.

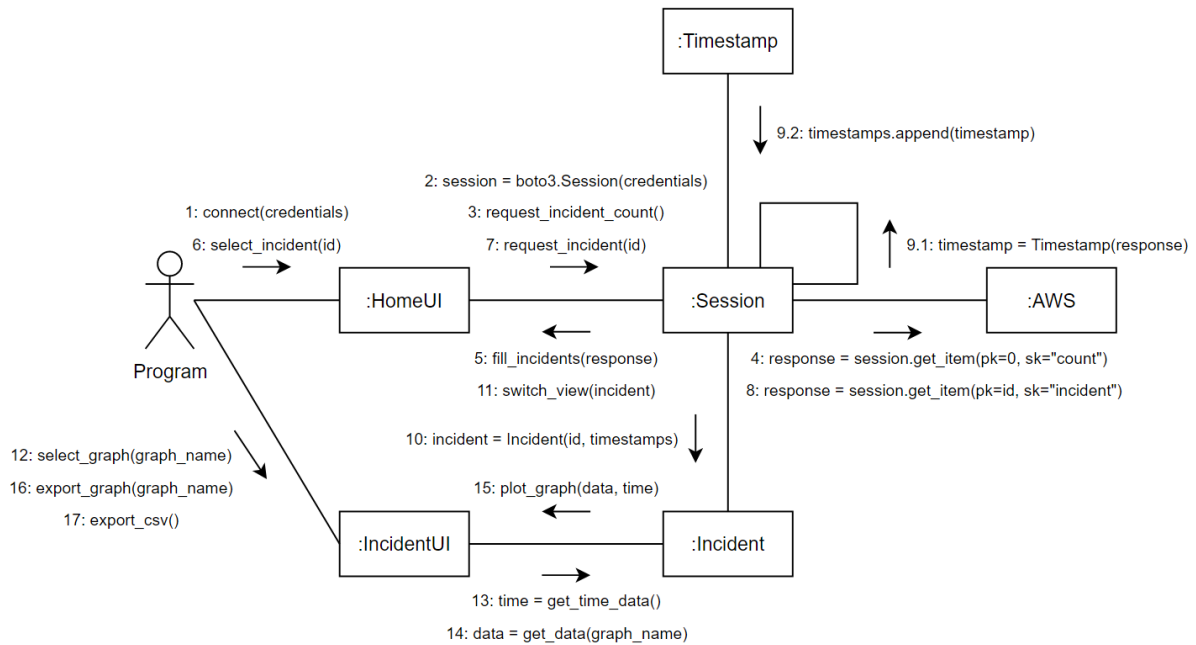


Figure 8: Communication Diagram of the IoTumble Program [created using Diagrams.net]

3.5.4 GUI Sketches

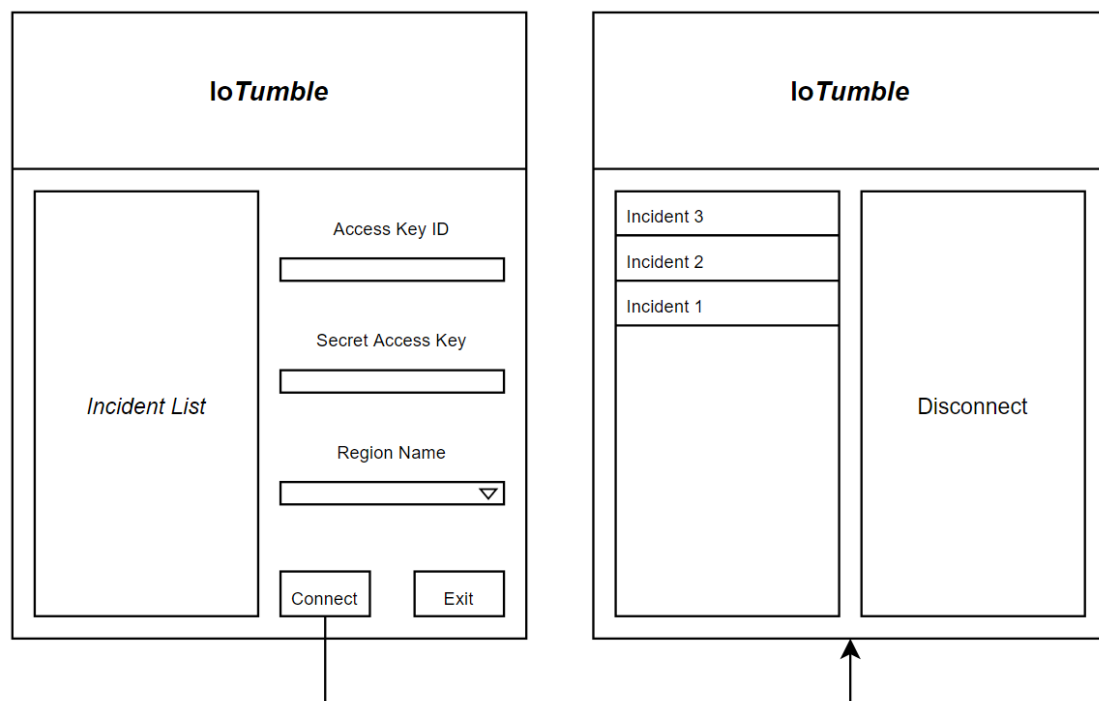


Figure 9: GUI Sketch of HomeView [created using Diagrams.net]

IoTumble

Back

Incident Details
Date =
Timestamp =
X-Acceleration =
Y-Acceleration =
Z-Acceleration =
Signal Vector Magnitude =

| Timestamp | X-Acceleration | Y-Acceleration | Z-Acceleration | SVM |
|-----------|----------------|----------------|----------------|-----|
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

Incident Graph

Select Graph

Export Graph

Export CSV

Figure 10: GUI Sketch of IncidentView [created using Diagrams.net]

The GUI of the IoTumble program is designed to be as user-friendly as possible. Every aspect is straight forward and labelled clearly for the user. The GUI contains all the necessary interactions that were established during system analysis, allowing the user to connect and disconnect from AWS, select an incident, select/export a graph, and finally export a CSV file. The final designs are much more stylised, but these sketches were a good reference to use when developing and implementing the actual GUI in code.

3.6. Hardware Specifications

3.6.1 Hardware Design

The hardware of IoTumble device consists of the Raspberry Pi Zero 2 W computer board, the ADXL345 accelerometer, and the LiPo SHIM power supply. These components are soldered using specific pinouts on the boards 40-pin header.

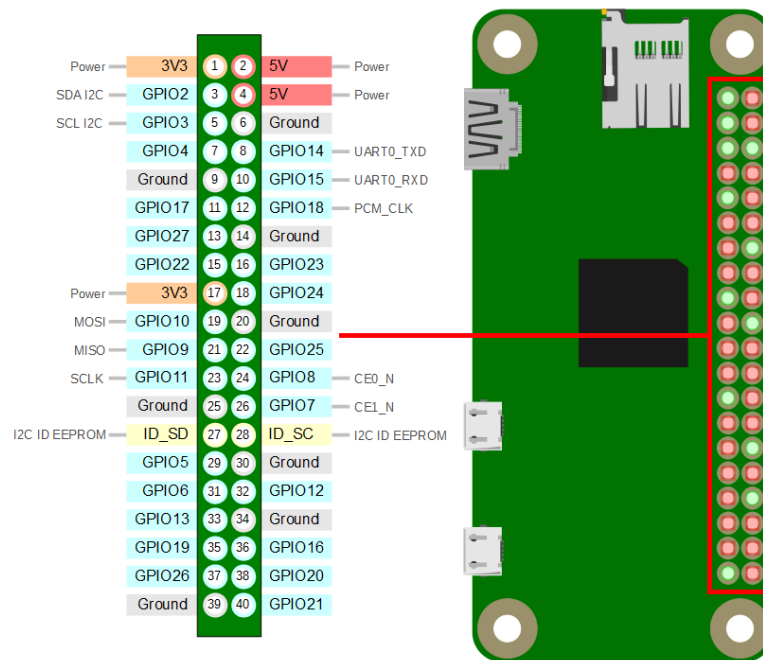


Figure 11: Pinout of Raspberry Pi Zero 2 W [edited but originally obtained from Google images]

The LiPo SHIM is designed to be soldered directly to the top of the board's header, using the following pinout:

- Pin 1 (3.3V) of the board.
- Pin 2 (5V) of the board.
- Pin 6 (Ground) of the board.
- Pin 7 (GPIO 4 – Battery Low) of the board.

The ADXL345 is connected to the board using stripped wires, which are soldered on top of the LiPo SHIM, with the following pinout:

- Pin 2 of ADXL345 (3.3V) is connected to Pin 1 (3.3V) of the board.
- Pin 3 of ADXL345 (GND) is connected to Pin 6 (Ground) of the board.
- Pin 8 of ADXL345 (SDA) is connected to Pin 3 (GPIO2 – SDA I2C) of the board.
- Pin 9 of ADXL345 (SCL) is connected to Pin 5 (GPIO3 – SCL I2C) of the board.

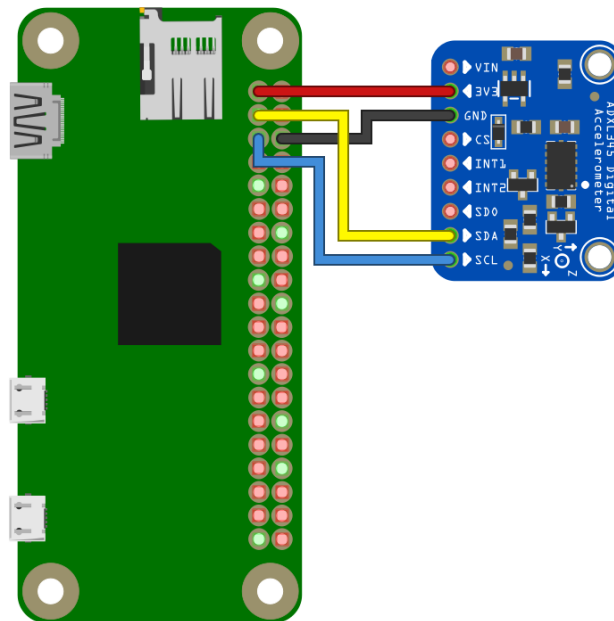


Figure 12: Pinout of ADXL345 using stripped wires [edited but originally obtained from Google images]

3.6.2 Bill of Materials

Table 11: Bill of Materials

| Material | Quantity | Cost |
|------------------------|-----------------|---------------|
| Raspberry Pi Zero 2 W | 1 | €13.23 |
| ADXL345 Accelerometer | 1 | €15.43 |
| LiPo SHIM | 1 | €11.64 |
| MicroSD Card (16GB) | 1 | €8.09 |
| 40-Pin Header | 1 | €2.36 |
| Solder | N/A | €0 |
| Stripped Wires | 4 | €0 |
| Old Smartphone Battery | 1 | €0 |
| Female JST Wire | 1 | €0 |
| Plastic Casing | 1 | €0 |
| Total | | €50.75 |

The above table shows the bill of materials for the IoTumble device. The total bill comes to €50.75, which could have been reduced if a smaller SD card was used. Some materials were not purchased as they were already in my possession such as the solder, stripped wires, old smartphone battery, female JST wire, and the plastic casing.

Chapter 4. Implementation

In this chapter the project's implementation has been explained. The following sections look at the assembly, wearability, and software of the IoTumble device, how the AWS architecture was built, and lastly the software and design of the IoTumble program.

4.1. IoTumble Device

4.1.1 Assembly

When assembling the IoTumble device, the aim was to keep it as compact as possible. This was achieved by soldering the devices components to the 40-pin header of the Raspberry Pi board. The header was first soldered to the board's pinout, allowing for the LiPo SHIM to be directly soldered on top of it.

Next, four wires were stripped and soldered to the pinout of the ADXL345 accelerometer and the top of the LiPo SHIM. The accelerometer was then held in a fixed position, by screwing a jack screw into the bottom of the Raspberry Pi board and screwing another screw into it from the top of the accelerometer.

To conserve on costs and to reduce the devices environmental impact, an old smartphone battery was repurposed. A female JST wire, recycled from the circuitry of an old printer, was soldered to its positive and negative pins. The soldering was then covered in black electrical tape to keep it from disrepair.



Figure 13: Hardware of the IoTumble Device

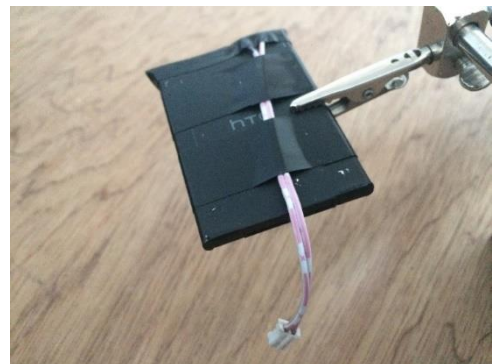


Figure 14: Repurposed Smartphone Battery

To house the hardware of the IoTumble device, a plastic casing was recycled from an old electronic device. The casing was the ideal size, as not only did the length of the Raspberry Pi board fit perfectly, but also the height of the LiPo SHIM. To gain access to the JST connector of the LiPo SHIM, a square hole was drilled and filed directly above it. To stop the hardware from moving, two-sided tape was used to stick it to the top of the casing.



Figure 15: Casing holding the IoTumble Device



Figure 16: Top of the casing showing the JST Connector

4.1.2 Wearability

To make the device wearable, a camera holder was used to store both the casing and battery. The holder was the perfect size and keeps the device rigid and in place. The holder is closed using a buttoned lid and can be attached to a belt using its strap.



Figure 17: Camera holder storing the casing and battery



Figure 18: Camera holder strap for attaching to a belt



Figure 19: Front and side view of the attached IoTumble Device

4.1.3 Connecting to AWS

The first implementation of the device's software is the method of connection to AWS. To connect to AWS, the necessary credentials and certificates need to be obtained first. These are stored within a `credentials.ini` file kept in a hidden `".aws"` directory.

```
[access]
access_key_id =
secret_access_key =
region_name =

[mqtt]
iot_thing =
iot_endpoint =

[path]
root_ca = .aws/certs/root-ca.pem
private_key = .aws/certs/private.pem.key
certificate = .aws/certs/certificate.pem.crt
```

Figure 20: File for storing the necessary AWS credentials

To read the files credentials, the method `connect()` first calls another method to create a `ConfigParser` object from the `configparser` module of Python. This object can then be used to get the value of a specified section/key pair in the file. The initialised variable `"client"` is set to an instance of a Boto3 client, imported from the AWS SDK for Python (Boto3). This is passed the parameters of `"dynamodb"` and the obtained access keys.

```
def connect(self):
    """This method connects the device to AWS and AWS IoT using its read credentials.ini."""
    credentials = self.read_credentials()
    self.client = client("dynamodb",
                        aws_access_key_id=credentials.get("access", "access_key_id"),
                        aws_secret_access_key=credentials.get("access", "secret_access_key"),
                        region_name=credentials.get("access", "region_name"))
    self.mqtt_configure(credentials)
    self.mqtt_client.connect()

def mqtt_configure(self, credentials: ConfigParser):
    """
    This method configures the devices MQTT client to publish messages to AWS IoT, using its
    read credentials.ini.

    :param credentials: ConfigParser object of the devices read credentials.ini.
    """
    self.mqtt_client = AWSIoTMQTTClient(credentials.get("mqtt", "iot_thing"))
    self.mqtt_client.configureEndpoint(credentials.get("mqtt", "iot_endpoint"), 8883)
    self.mqtt_client.configureCredentials(credentials.get("path", "root_ca"),
                                        credentials.get("path", "private_key"),
                                        credentials.get("path", "certificate"))
    self.mqtt_client.configureOfflinePublishQueueing(-1)
    self.mqtt_client.configureDrainingFrequency(2)
    self.mqtt_client.configureConnectDisconnectTimeout(10)
    self.mqtt_client.configureMQTTOperationTimeout(5)
```

Code Extract 1: Connect and MQTT Configure methods

Next, the method `mqtt_configure()` is called and passed the created `ConfigParser` object. The initialised variable `"mqtt_client"` is set as an `AWSIoTMQTTClient`, imported from the AWS IoT Device SDK for Python. This is then configured using the IoT thing, IoT endpoint, and certificate files that were created within AWS IoT. After configuration, the `"mqtt_client"` is connected to AWS IoT.

4.1.4 Reading Acceleration and Creating Timestamps

The next implementation is the reading of acceleration and creation of timestamps. The method `read_accelerometer()` is used for calling other methods. It sleeps 0.1 seconds to allow time for a new timestamp. It then creates and records one, checks if a threshold has already been met, and if not, checks if the timestamp has reached any thresholds.

To create a timestamp, the method `create_timestamp()` first reads the current acceleration of the initialised variable “accelerometer”. This variable is an instance of an ADXL345 object, imported from the Adafruit CircuitPython ADXL34x Driver.

```
def read_accelerometer(self):
    """
    This method reads the devices accelerometer, creates and records a timestamp, and if the
    threshold flag is False, it checks if its acceleration values have reached any thresholds.
    """
    sleep(0.1)
    timestamp = self.create_timestamp()
    self.record_timestamp(timestamp)
    if not self.threshold_flag:
        self.check_thresholds(timestamp)

def create_timestamp(self) -> dict:
    """
    This method uses the acceleration values of the devices accelerometer to calculate the
    Signal Vector Magnitude (SVM), and then create a timestamp with these values.

    :returns: Created timestamp.
    """
    x_acc, y_acc, z_acc = self.accelerometer.acceleration
    svm = self.calculate_svm(x_acc, y_acc, z_acc)
    timestamp = {"x": x_acc, "y": y_acc, "z": z_acc, "svm": svm, "ep": time()}
    return timestamp
```

Code Extract 2: Read Accelerometer and Create Timestamp methods

The read acceleration is used to calculate the SVM by calling `calculate_svm()`. A timestamp dictionary is then created and returned containing the acceleration values, calculated SVM, and current epoch time (the number of seconds since 01 January 1970).

```
@staticmethod
def calculate_svm(x_acc: float, y_acc: float, z_acc: float) -> float:
    """
    This method calculates the Signal Vector Magnitude from the passed acceleration values.

    :param x_acc: X-Acceleration value.
    :param y_acc: Y-Acceleration value.
    :param z_acc: Z-Acceleration value.
    :return: Signal Vector Magnitude value.
    """
    svm = sqrt((x_acc * x_acc) + (y_acc * y_acc) + (z_acc * z_acc))
    return svm
```

Code Extract 3: Calculate Signal Vector Magnitude method

After a timestamp has been created, it is recorded using the method `record_timestamp()`. The method first appends the timestamp dictionary to the initialised list "timestamps". It then checks if the length of the list is larger than 51, and if it is, removes the oldest timestamp in the list. In this way, there is only a maximum of 5 seconds of recorded timestamps since every timestamp is approximately 0.1 seconds.

```
def record_timestamp(self, timestamp: dict):  
    """  
    This method records a timestamp to the devices timestamp list. It then deletes the first  
    timestamp in the list if the list is larger than 51 (each timestamp is 0.1s, so only 5  
    seconds of previous timestamps are saved).  
  
    :param timestamp: Created Timestamp.  
    """  
    self.timestamps.append(timestamp)  
    if len(self.timestamps) > 51:  
        self.timestamps.pop(0)
```

Code Extract 4: Record Timestamp method

4.1.5 Checking for Thresholds

After a timestamp has been recorded, it is checked for reached thresholds using the method `check_thresholds()`. The method first checks the absolute value of the timestamps SVM against the threshold of 20. If it has reached this threshold, the absolute acceleration values are compared against the threshold of 18. If any of the values have reached this threshold, the method `threshold_reached()` is called.

```
def check_thresholds(self, timestamp: dict):  
    """  
    This method checks if the absolute values of a created timestamp have reached the set  
    thresholds. If its SVM reaches a value of 20, and any acceleration reaches a value of 18,  
    the thresholds have been reached.  
  
    :param timestamp: Created Timestamp.  
    """  
    svm = abs(timestamp.get("svm"))  
    if svm > 20:  
        for key in ("x", "y", "z"):  
            data = abs(timestamp.get(key))  
            if data > 18:  
                self.threshold_reached()  
                break  
  
def threshold_reached(self):  
    """  
    This method sets the threshold flag to True, gets the half length of the devices timestamp  
    list, and reads the accelerometer for this amount of timestamps. The device then checks  
    these read timestamps for inactivity, by passing half of their length to check_inactivity()  
    (as the first few timestamps will still be active due to the devices impact/movement).  
    """  
    self.threshold_flag = True  
    post_threshold_length = int(len(self.timestamps) / 2)  
    for _ in range(post_threshold_length):  
        self.read_accelerometer()  
    self.check_inactivity(int(post_threshold_length / 2))  
    self.threshold_flag = False
```

Code Extract 5: Check Thresholds and Threshold Reached methods

The method `threshold_reached()` is used to set the initialised variable “`threshold_flag`” to True, get the post threshold length of the recorded timestamps, and then call the method `read_accelerometer()` for this amount of times. This allows for the device to record more timestamps after the reached thresholds, so they can be checked for inactivity.

4.1.6 Checking for Inactivity

After the post threshold timestamps have been recorded, they are checked for inactivity using the method `check_inactivity()`. A parameter “inactivity length” is passed into it, which is half of the post threshold length. This way, it ignores the first few timestamps, as they will always still be active due to the device’s impact/movement.

```
def check_inactivity(self, inactivity_length: int):
    """
    This method gets lists of the absolute post threshold values, loops through each one,
    calculates their average, and checks if any value in the list is within -1 or 1 of their
    average. If all values are within this range, the device is inactive so it publishes an
    incident to AWS.

    :param inactivity_length: Length to check for inactivity.
    :return: None (if any value isn't within -1 or 1 of their average).
    """
    post_threshold = self.get_post_threshold(inactivity_length)
    for data_list in post_threshold.values():
        average = sum(data_list) / inactivity_length
        for data in data_list:
            comparison = data - average
            if -1 <= comparison <= 1:
                continue
        return
    self.mqtt_publish()

def get_post_threshold(self, inactivity_length: int) -> dict:
    """
    This method uses the inactivity length to get the last timestamps of its amount from the
    devices timestamp list. It then stores the absolute values for each timestamp into lists,
    and returns them in a dictionary, with their corresponding keys.

    :param inactivity_length: Length to check for inactivity.
    :return: Lists of the absolute post threshold values.
    """
    post_threshold = {"x": [], "y": [], "z": [], "svm": []}
    for timestamp in self.timestamps[-inactivity_length:]:
        for key, value in post_threshold.items():
            value.append(abs(timestamp.get(key)))
    return post_threshold
```

Code Extract 6: Check Inactivity and Get Post Threshold methods

It first calls the method `get_post_threshold()` to obtain the post threshold values, using the passed “`inactivity_length`”. The method loops through the last recorded timestamps of this amount, and then stores each absolute timestamp value into lists within a dictionary.

Using the lists of the dictionary, the method `check_inactivity()` continues and loops through each one, calculating their average value. Another loop then goes through each value within the list and checks if they are within -1 or 1 of the calculated list average. If any value is not within this range, the method returns, and does not publish an incident. But if all the values are within this range, an incident is published to AWS IoT.

4.1.7 Publishing an Incident

After inactivity is detected, an incident is published using the method `mqtt_publish()`. It first sets a string as the name of the IoT topic being used in AWS IoT, including an incremented incident ID. The method `request_incident_count()` is used for this, to request the incident count from a DynamoDB table, by using the previously connected “client” instance and its method `get_item()`.

```
def mqtt_publish(self):
    """This method publishes a created JSON payload string to AWS IoT using the MQTT client."""
    topic = f"iotumble/incident/{self.request_incident_count() + 1}"
    payload = self.create_payload()
    self.mqtt_client.publish(topic, payload, 1)

def create_payload(self) -> str:
    """
    This method creates a JSON payload string of the devices timestamp list.

    :returns: JSON payload string of the devices timestamps.
    """
    payload = "{"
    for i, timestamp in enumerate(self.timestamps):
        payload_timestamp = f'"{i}": {timestamp}'
        if timestamp != self.timestamps[-1]:
            payload += payload_timestamp + ", "
        else:
            payload += payload_timestamp + "}"
    return payload.replace("'", "")

def request_incident_count(self) -> int:
    """
    This method requests the count of incident items from the DynamoDB client, so its response
    can be incremented.

    :returns: Count of incident items.
    """
    try:
        response = self.client.get_item(TableName="iotumble_incidents",
                                         Key={"pk": {"N": "0"}, "sk": {"S": "count"}})
        count = response["Item"]["msg"]["N"]
    except ClientError as err:
        raise err
    else:
        return int(count)
```

Code Extract 7: MQTT Publish, Create Payload, and Request Incident Count methods

Then it continues and creates a JSON payload string of the recorded timestamps using the method `create_payload()`. Finally, the connected and configured “`mqtt_client`” is used to publish the incident, using its method `publish()`. The method is passed the topic, payload, and a quality-of-service value of 1 (message will be delivered at least once).

4.2. AWS Architecture

This section outlines the implemented AWS architecture, showing which services are being used and how they interact with each other.

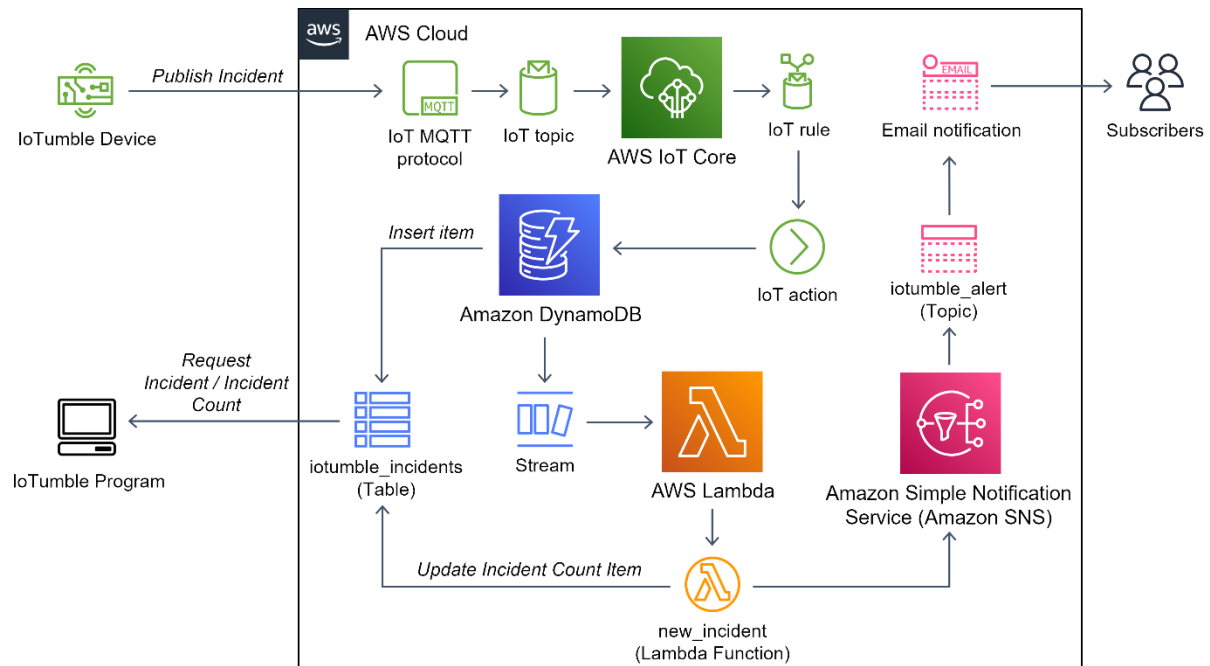


Figure 21: AWS Architecture Diagram [created using AWS Toolkit for PowerPoint]

4.2.1 AWS IoT

The first implementation in AWS was the setup of AWS IoT. For a device to connect to the service, it needs to configure itself using its IoT Thing, certificate files, and an IoT endpoint. These are created and obtained from AWS IoT. When creating an IoT Thing, the service creates three certificate files, a root certificate authority (CA) file, a private key file, and another certificate file. The name of the IoT Thing is specified, and it is given an IoT policy which allows it to connect, receive, publish, and subscribe to AWS IoT.

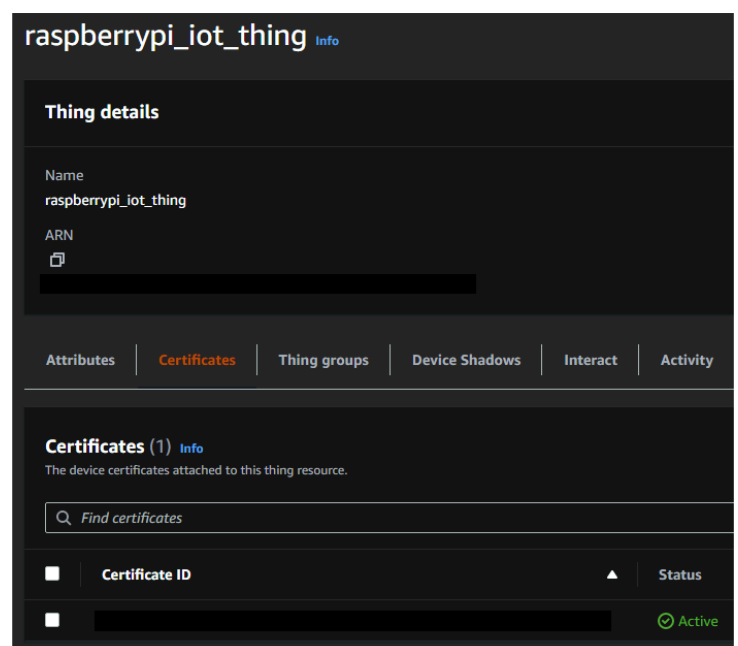


Figure 22: Created IoT Thing

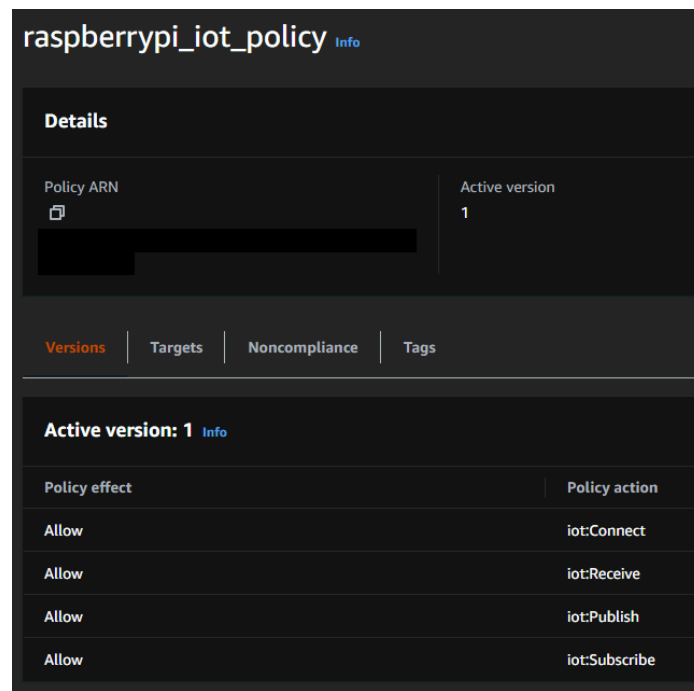


Figure 23: Created IoT Policy

The IoT endpoint can be accessed from the settings page of AWS IoT. Once all necessary credentials have been gathered, they can be used to connect the device to the service. When publishing messages to AWS IoT, an IoT rule needs to be created to set how the messages are processed. The rule is given a description, an SQL query statement selecting the data of published messages, and finally an action.

The field "iotumble/incidents/+" is being used within the SQL query statement, to select all data from published messages that have that field as their IoT topic. The '+' symbol stands for the incident ID of the published message. The action of the IoT rule inserts the message into a specified DynamoDB table. The action uses the incident ID of the IoT topic, to give the new table item a unique partition key.

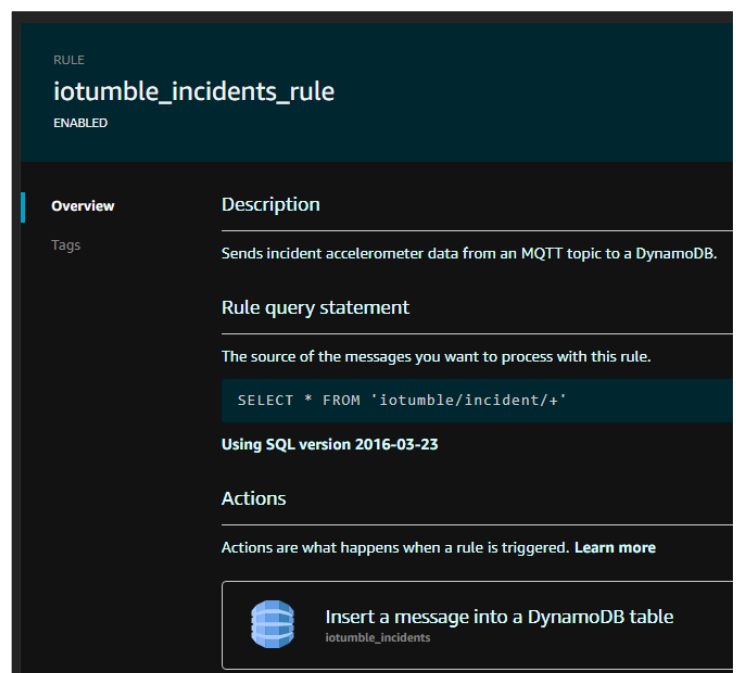
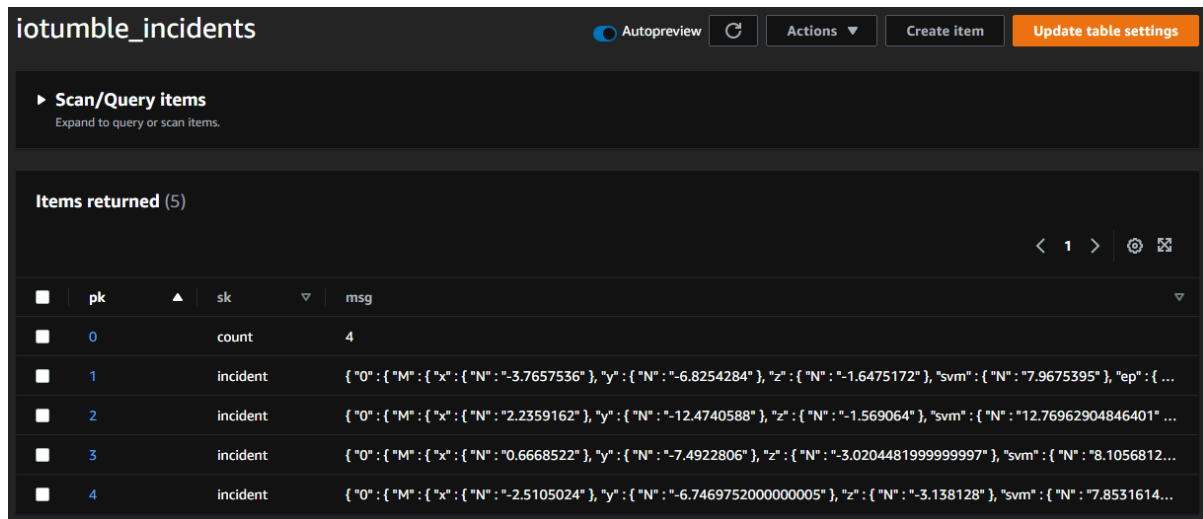


Figure 24: Created IoT Rule

4.2.2 DynamoDB

The second implementation of AWS was the creation of a DynamoDB table and a DynamoDB stream. When creating a table, it is given a name, partition key (primary key), and sort key (second part of the primary key). The partition key holds the incident ID, the sort key stores the item type, and the message field contains the data of the incident.



| | pk | sk | msg |
|--|----|----------|--|
| | 0 | count | 4 |
| | 1 | incident | { "0": { "M": { "x": { "N": "-3.7657536" }, "y": { "N": "-6.8254284" }, "z": { "N": "-1.6475172" }, "svm": { "N": "7.9675395" }, "ep": { ... |
| | 2 | incident | { "0": { "M": { "x": { "N": "2.2359162" }, "y": { "N": "-12.4740588" }, "z": { "N": "-1.569064" }, "svm": { "N": "12.76962904846401" ... |
| | 3 | incident | { "0": { "M": { "x": { "N": "0.6668522" }, "y": { "N": "-7.4922806" }, "z": { "N": "-3.0204481999999997" }, "svm": { "N": "8.1056812..." |
| | 4 | incident | { "0": { "M": { "x": { "N": "-2.5105024" }, "y": { "N": "-6.7469752000000005" }, "z": { "N": "-3.138128" }, "svm": { "N": "7.8531614..." |

Figure 25: Created DynamoDB Table

When designing a DynamoDB table, it is important to consider how a table can make efficient use of its partition and sort key. When requested for items, a table that is not optimised could use a lot of read capacity units (RCU) and produce a very high bill at the end of the month. To reduce this, a “count” item was created to store the current number of incidents in the table. This item can then be requested, which uses minimal RCU compared to what it would cost to read over every single item in the table to get an incident count.

For items containing incidents, the message field of the DynamoDB table, stores its timestamps in JSON form. The field contains key/value pairs within maps, which represent the timestamps of the incident. The data of a timestamp are all represented as float values, including the timestamp itself, which is stored as epoch time (the number of seconds since 01 January 1970). A typical incident item is a maximum of 3.1 kB in size.

```

1 {
2   "pk": 4,
3   "sk": "incident",
4   "msg": {
5     "0": {
6       "x": -2.5105024,
7       "y": -6.7469752000000005,
8       "z": -3.138128,
9       "svm": 7.853161401257764,
10      "ep": 1646929430.3366058
11    },
12    "1": {
13      "x": 2.1182364,
14      "y": -10.4342756,
15      "z": -2.7850886,
16      "svm": 11.005351028154,
17      "ep": 1646929430.4388123
18    },
19    "2": {
20      "x": -4.9425516,
21      "y": -6.276256,
22      "z": -4.0403398,
23      "svm": 8.952348931742026,
24      "ep": 1646929430.540977
25    }
  }
}
```

Figure 26: Timestamps in JSON Form

After a new item is inserted into the DynamoDB table, a DynamoDB stream captures an image of its data. The image is then used by a trigger, which in this case is an AWS Lambda function.

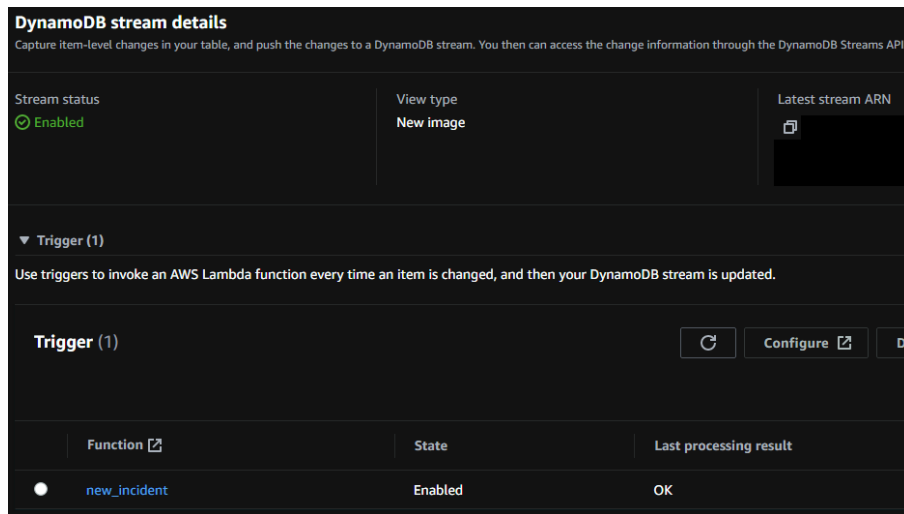


Figure 27: Created DynamoDB Stream

4.2.3 AWS Lambda

When the Lambda function is triggered by a DynamoDB stream, it first checks what type of event has called the trigger. The function only wants to be called when an item is inserted into the table, so a filter criteria is used. By using this, whenever the incident count item is updated, the Lambda function ignores the DynamoDB stream that triggers it again.

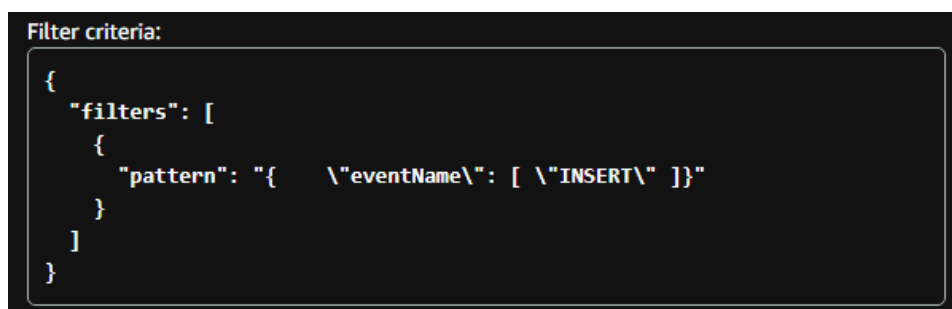


Figure 28: Filter criteria of the Lambda function

The function itself first loops through the newly inserted items of the DynamoDB table, which is only 1 in most cases. It then sets the variable "incident" as the items captured image. Using this variable it gets the incident ID, and its timestamps. The function then loops through the timestamps to append every SVM to a list, where it is then used to calculate the maximum value within it.

```

3
4 def lambda_handler(event, context):
5     for record in event["Records"]:
6         incident = record["dynamodb"]["NewImage"]
7         incident_id = incident["pk"]["N"]
8         timestamps = incident["msg"]["M"]
9
10        # Getting the maximum SVM of the incident
11        svm_list = []
12        for i in range(len(timestamps)):
13            timestamp = timestamps[str(i)]["M"]
14            svm = float(timestamp["svm"]["N"])
15            svm_list.append(svm)
16        max_svm = max(svm_list)
17
18        # Getting the timestamp data of the maximum SVM
19        timestamp_data = []
20        for i in range(len(timestamps)):
21            timestamp = timestamps[str(i)]["M"]
22            svm = float(timestamp["svm"]["N"])
23            if svm == max_svm:
24                timestamp_data.append(float(timestamp["ep"]["N"]))
25                timestamp_data.append(round(float(timestamp["x"]["N"]), 8))
26                timestamp_data.append(round(float(timestamp["y"]["N"]), 8))
27                timestamp_data.append(round(float(timestamp["z"]["N"]), 8))
28                timestamp_data.append(round(float(timestamp["svm"]["N"]), 10))
29
30        # Getting the date and time of the incident
31        date_time = datetime.fromtimestamp(timestamp_data[0])
32        date = date_time.strftime("%d %B %Y")
33        time = date_time.strftime("%H:%M:%S.%f")[:-3]
34

```

Code Extract 8: Lambda function obtaining the incident data

The timestamps are looped through again, but this time the previously obtained maximum SVM value is searched for. When the timestamp of this value is discovered, the rest of its data is appended to a list. This list is then to be used for the details of an email, that will alert subscribers of a new incident.

Once the incident data has been obtained, the Lambda function creates an instance of a SNS resource, imported from the AWS SDK for Python (Boto3). This is then used to create another instance of an SNS Topic, which is passed the Amazon Resource Name (ARN) of a created SNS topic. The function then uses it to publish an email, alerting its subscribers of a new incident, containing the obtained incident data, and explaining how to view the incident further.

```

34
35     # Publish alert to SNS topic
36     sns = boto3.resource("sns")
37     topic = sns.Topic(
38         "iotumble_alert")
39     topic.publish(
40         Subject="Incident Alert",
41         Message="IoTumble has detected inactivity after its acceleration " +
42             "threshold was reached. The following incident has been " +
43             "published to AWS:\n\nIncident ID: " + incident_id +
44             "\n\nIncident Date: " + date + " \n\nIncident Details:" +
45             "\n• Timestamp = " + time +
46             "\n• X-Acceleration = " + str(timestamp_data[1]) +
47             "\n• Y-Acceleration = " + str(timestamp_data[2]) +
48             "\n• Z-Acceleration = " + str(timestamp_data[3]) +
49             "\n• Signal Vector Magnitude = " + str(timestamp_data[4]) +
50             "\n\nConnect to AWS via the IoTumble program to view the " +
51             "incident.\n\n"
52     )

```

Code Extract 9: Lambda function sending an Amazon SNS email

Finally, after an email has been sent to all subscribers, the incident count item is updated in the DynamoDB table. It creates an instance of a DynamoDB resource and table, and uses its method `update_item()` to increment the count value.

```
52
53 # Increment table's incident count
54 dynamodb = boto3.resource("dynamodb")
55 table = dynamodb.Table("iotumble_incidents")
56 table.update_item(
57     Key={
58         "pk": 0,
59         "sk": "count"
60     },
61     UpdateExpression="SET msg = msg + :increment",
62     ExpressionAttributeValues={
63         ":increment": int(1)
64     }
65 )
66
```

Code Extract 10: Lambda function incrementing the incident count item

4.2.4 Amazon SNS

For the AWS Lambda function to send an email, an SNS topic needed to be created within Amazon SNS. The topic is given a name and a display name (the name that appears as the email sender).

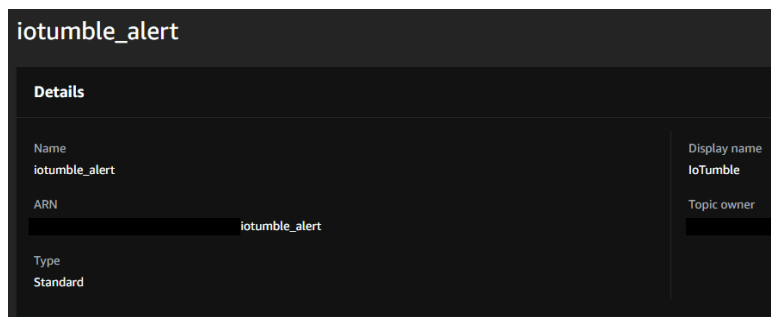


Figure 29: Created SNS Topic

Email subscriptions are created manually and need to be accepted through a verification email. Once accepted, any email that is sent to the created SNS topic will also be sent to the subscriber.

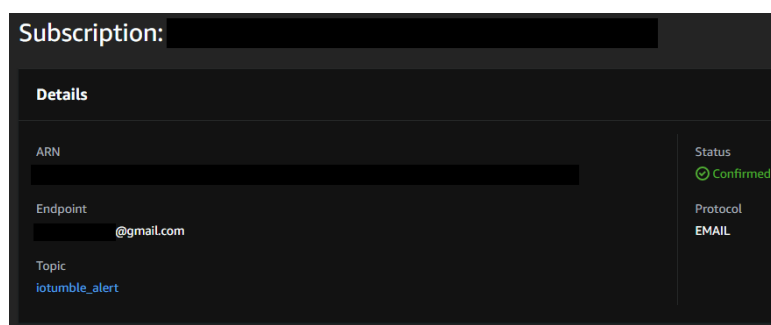


Figure 30: Email subscription to the SNS Topic

4.3. IoTumble Program

4.3.1 MVC Pattern

As previously stated in Chapter 3 Section 5, the IoTumble program implements the MVC design pattern, by separating its business logic from its presentation logic. A user interacts with the GUI of a view class, which calls a controller class to act as its bridge between the model classes. This allows for very little dependency between classes.

A controller class contains the methods to start and close its view, update it with information received from a model class, or to give functionality for some widgets within its view. A view class contains the methods to load the GUIs window and widgets, configure the widgets style, set their positions, update their information, or to give them a specific functionality such as calling a method from within its controller class.

A model class contains the methods for returning the data of its initialised variables, to perform the business logic of the program, and to interact with AWS and its services. Ultimately, the MVC structure allowed for the faster development of modifiable code. The program is also more efficient, as it leads to a reduction in the execution of code as well.

```
class IncidentController(AbstractController):
    """ ... """

    def __init__(self, home_view, incident):...

    def fill_details(self):...

    def fill_graph(self, selected_graph: str):...

    def export_csv(self):...

    def export_graph(self, selected_graph: str):...

    def back(self):...

    def main(self):...
```

Code Extract 11: Structure of a Controller Class

```
class IncidentView(AbstractView, tk.Toplevel):
    """ ... """

    def __init__(self, controller):...

    def load_root(self):...

    def load_header(self):...

    def load_details(self):...

    def load_graph(self):...

    def load_actions(self):...

    def load_graph_style(self):...

    def fill_details_labels(self, max_timestamp):...

    def fill_details_tree_view(self, timestamps: List):...

    def select_graph(self, selected_graph: str):...
```

Code Extract 12: Structure of a View Class

```
class Incident:
    """ ... """

    def __init__(self, incident_id: int, timestamps: List):...

    def get_incident_id(self) -> int:...

    def get_timestamps(self) -> List:...

    def get_timestamps_time(self) -> List[float]:...

    def get_timestamps_x(self) -> List[float]:...

    def get_timestamps_y(self) -> List[float]:...

    def get_timestamps_z(self) -> List[float]:...

    def get_timestamps_svm(self) -> List[float]:...

    def get_max_timestamp(self):...

    def export_timestamps(self, csv_path: str):...
```

Code Extract 13: Structure of a Model

4.3.2 Connecting and Disconnecting

The IoTumble program starts from the creation of a HomeController instance. The controller starts its view, which loads its GUI widgets, and puts itself in a main loop. The GUI presents entries and a button for the user to input their credentials and connect a session to AWS. The program also checks for credentials within a credentials.ini file, kept in a hidden “.aws” directory. If credentials are found, it fills the entries automatically for the user. Once the connect button is clicked, the HomeView calls its method connect().

```
def connect(self, session_frame: tk.Frame):
    """
    This method passes the session inputs to connect() from HomeController, and if it returns
    True, it covers the session section with a disconnect button.
    """
    :param session_frame: Frame of the session section.
    """
    if self.controller.connect(self.inputs["access"].get(), self.inputs["secret"].get(),
                              self.inputs["region"].get()):
        disconnect_button = ttk.Button(session_frame, takefocus=False, text="Disconnect",
                                       command=lambda: self.disconnect(disconnect_button))
        disconnect_button.place(height=40, width=293, y=22)

def disconnect(self, disconnect_button: ttk.Button):
    """
    This method calls disconnect() from HomeController, clears the incidents treeview, and
    destroys the disconnect button.
    """
    :param disconnect_button: Disconnect button to be destroyed.
    """
    self.controller.disconnect()
    self.clear_tree_view()
    disconnect_button.destroy()
```

Code Extract 14: Connect and Disconnect methods in HomeView

The method passes the inputted credentials to another method in HomeController. The controller checks if the entries are empty, and if they are it displays an error message in HomeView. Otherwise, an initialised model Session class is called to connect a Boto3 session within itself to AWS using the credentials. Lastly, a resource of a DynamoDB table is created, the incidents list is filled, and a disconnect button is placed over the GUI credential entries.

```
def connect(self, access_key_id: str, secret_access_key: str, region_name: str) -> bool:
    """
    This method connects the Session object to AWS, and creates an Amazon DynamoDB table
    resource in Session. Lastly, fill_incidents() is called and it returns its outcome.
    """
    :param access_key_id: Access Key ID of the Session.
    :param secret_access_key: Secret Access Key of the Session.
    :param region_name: Region Name of the Session.
    :returns: Boolean outcome of fill_incidents().
    """
    if access_key_id == "" or secret_access_key == "" or region_name == "":
        self.home_view.show_message("Please fill all the session entries!")
        return False
    self.session.connect(access_key_id, secret_access_key, region_name)
    self.session.create_table("iotumble_incidents")
    return self.fill_incidents()

def disconnect(self):
    """This method disconnects the Session object from AWS."""
    self.session.disconnect()
```

Code Extract 15: Connect and Disconnect methods in HomeController

```

def connect(self, access_key_id: str, secret_access_key: str, region_name: str):
    """
    This method connects to AWS by creating a boto3 session.

    :param access_key_id: Access Key ID of the Session.
    :param secret_access_key: Secret Access Key of the Session.
    :param region_name: Region Name of the Session.
    """
    self.boto_session = BotoSession(aws_access_key_id=access_key_id,
                                    aws_secret_access_key=secret_access_key,
                                    region_name=region_name)

def disconnect(self):
    """
    This method disconnects from AWS by destroying the boto3 session and DynamoDB table
    resource.
    """
    self.boto_session = None
    self.incidents_table = None

def create_table(self, table_name: str):
    """
    This method creates a DynamoDB table resource.

    :param table_name: Name of the DynamoDB table.
    """
    dynamo_db = self.boto_session.resource("dynamodb")
    self.incidents_table = dynamo_db.Table(table_name)

```

Code Extract 16: Connect, Disconnect, and Create Table methods in Session

To disconnect from AWS a similar process is implemented. Once the disconnect button is clicked, the HomeView calls the HomeController to disconnect the Boto3 session within the model Session class. It achieves this by simply setting itself and the created DynamoDB table resource as None. Finally, the filled incidents list is cleared, and the placed disconnect button is destroyed from the GUI.

4.3.3 Filling Incidents

To fill the incidents list in the GUI, the method `fill_incidents()` in HomeController is called after connecting to AWS. The method calls `request_incident_count()` from the model Session class, which uses its DynamoDB table resource to retrieve the incident count. It checks if the count has been successfully retrieved, and if not, displays an error message in HomeView saying the entered AWS session keys are not valid (and subsequently returns False to the HomeView so a disconnect button is not placed over the session entries.)

```

def fill_incidents(self) -> bool:
    """
    This method requests an incident count from Session and if it returns False, it passes an
    error message to HomeView, and returns False. If it returns True, it continues and uses its
    outcome to fill the incidents of HomeView, and return True.

    :returns: Boolean outcome of request_incident_count().
    """
    incident_count = self.session.request_incident_count()
    if not incident_count:
        self.home_view.show_message("The entered session access keys are not valid!")
        return False
    self.home_view.fill_incidents(incident_count)
    return True

```

Code Extract 17: Fill Incidents method in HomeController

```

def request_incident_count(self):
    """
    This method requests the count of incident items from the created DynamoDB resource, creates
    a count value from its response, and returns it.

    :returns: Count of incident items.
    """
    try:
        response = self.incidents_table.get_item(Key={"pk": 0, "sk": "count"})
        count = response["Item"]["msg"]
    except ClientError:
        return False
    else:
        return int(count)

```

Code Extract 18: Request Incident Count method in Session

If the incident count has been successfully retrieved, the method `fill_incidents()` in `HomeView` is called. This loops and inserts the retrieved incident count into its tree view.

```

def fill_incidents(self, incident_count: int):
    """
    This method fills the incidents treeview with a certain amount of items.

    :param incident_count: Count of incident items.
    """
    for i in range(incident_count):
        i += 1
        self.incidents_tree_view.insert("", 0, tags=str(i % 2), text="Incident " + str(i))

```

Code Extract 19: Fill Incidents method in HomeView

4.3.4 Selecting an Incident

Once the incident list has been filled, a user can select one to view. The incident tree view widget in `HomeView` is binded to call the method `switch()` whenever an item is selected. The method gets the current selected item in the tree view, obtains its incident ID, and then passes it to another method in `HomeController`.

```

def switch(self):
    """
    This method gets the selected item and its incident ID from the incidents treeview, and
    passes it to switch() in HomeController.
    """
    selected = self.incidents_tree_view.selection()[0]
    if selected != "":
        selected_text = self.incidents_tree_view.item(selected, "text")
        incident_id = int(selected_text.replace("Incident ", ""))
        self.controller.switch(incident_id)

```

Code Extract 20: Switch method in HomeView

The method `switch()` in `HomeController` calls `request_incident()` from the model `Session` class. This method uses the DynamoDB table resource to retrieve the specified incident from AWS. It then loops through its response, creates a list of model `Timestamp` objects, which are then used in a returned model `Incident` object. The `HomeController` then checks if it was retrieved successfully, and if it not, displays an error message in `HomeView` saying the selected incident does not exist.

If it is successfully retrieved, the method hides the GUI of HomeView, creates a new instance of IncidentController, and runs its main method.

```
def switch(self, incident_id: int):
    """
    This method requests an Incident object from Session and if it returns False, it passes an
    error message to HomeView. If it returns True, it hides HomeView, and passes its outcome to
    load_controller() to load an instance of IncidentController. Finally it runs the main method
    of IncidentController.

    :param incident_id: ID of the Incident.
    """
    incident = self.session.request_incident(incident_id)
    if not incident:
        self.home_view.show_message("The selected incident does not exist!")
    else:
        self.home_view.hide()
        incident_controller = self.load_controller("Incident")(self.home_view, incident)
        incident_controller.main()
```

Code Extract 21: Switch method in HomeController

```
def request_incident(self, incident_id: int):
    """
    This method requests an incident item from the created DynamoDB resource, creates an
    Incident object from its response, and returns it.

    :param incident_id: ID of the Incident.
    :returns: Instance of an Incident object.
    """
    try:
        response = self.incidents_table.get_item(Key={"pk": incident_id, "sk": "incident"})
        timestamps = response["Item"]["msg"]
    except KeyError:
        return False
    else:
        timestamps = dict(sorted(timestamps.items(), key=lambda d: int(d[0])))
        incident_timestamps = []
        for timestamp_id in timestamps:
            timestamp_data = []
            sensor_data = timestamps[timestamp_id]
            for data in sensor_data:
                timestamp_data.append(float(sensor_data[data]))
            timestamp = Timestamp(int(timestamp_id), timestamp_data)
            incident_timestamps.append(timestamp)
        return Incident(incident_id, incident_timestamps)
```

Code Extract 22: Request Incident method in Session

4.3.5 Filling Details

After an instance of IncidentController has started an IncidentView, the view loads its widgets and then calls the method `fill_details()` of IncidentController. This method calls the getter methods of the previously retrieved Incident object, which get its timestamps and maximum SVM timestamp.

The maximum SVM is obtained by creating a list comprehension for all SVM values in the Incidents timestamps, and then calling a `max()` function on it. The maximum SVM timestamp is then retrieved by looping through every timestamp in the Incident and comparing its SVM value with the maximum SVM. The obtained timestamps are then passed to the methods `fill_details_labels()` and `fill_details_tree_view()` in IncidentView.

```

def fill_details(self):
    """
    This method gets the incidents timestamps and maximum SVM timestamp and uses them to fill
    the details section of IncidentView.
    """
    max_timestamp = self.incident.get_max_timestamp()
    timestamps = self.incident.get_timestamps()
    self.incident_view.fill_details_labels(max_timestamp)
    self.incident_view.fill_details_tree_view(timestamps)

```

Code Extract 23: Fill Details method in IncidentController

```

def get_max_timestamp(self):
    """
    This method gets the Timestamp object with the maximum SVM value.

    :returns: Timestamp object with the maximum SVM value.
    """
    max_timestamp = None
    max_svm = max(timestamp.get_svm() for timestamp in self.timestamps)
    for timestamp in self.timestamps:
        if timestamp.get_svm() == max_svm:
            max_timestamp = timestamp
    return max_timestamp

```

Code Extract 24: Get Max Timestamp method in Incident

The method `fill_details_labels()` puts the data of the maximum SVM timestamp into a list, so that it can be looped through and assigned to its corresponding label in the GUI. The method `fill_details_tree_view()` loops through the list of timestamps and inserts the data of each one into a new item within the tree view.

```

def fill_details_labels(self, max_timestamp):
    """
    This method fills the details labels with the data of the maximum SVM timestamp.

    :param max_timestamp: Timestamp object with the maximum SVM value.
    """
    timestamp_data = [max_timestamp.get_date(), max_timestamp.get_time(),
                      round(max_timestamp.get_x_acc(), 8), round(max_timestamp.get_y_acc(), 8),
                      round(max_timestamp.get_z_acc(), 8), round(max_timestamp.get_svm(), 10)]
    for i, label in enumerate(self.details_widgets):
        if isinstance(label, ttk.Label):
            label_text = label.cget("text")
            self.details_widgets[i].configure(text=f"{label_text} = {str(timestamp_data[i])}")

def fill_details_tree_view(self, timestamps: List):
    """
    This method fills the details treeview with the data of the incidents timestamps.

    :param timestamps: List of Timestamp objects.
    """
    for timestamp in timestamps:
        self.details_widgets[6].insert("", 0, tags=str(timestamp.get_timestamp_id() % 2),
                                       values=(timestamp.get_time(),
                                               round(timestamp.get_x_acc(), 8),
                                               round(timestamp.get_y_acc(), 8),
                                               round(timestamp.get_z_acc(), 8),
                                               round(timestamp.get_svm(), 10)))

```

Code Extract 25: Fill Details Labels and Tree View methods in IncidentView

4.3.6 Selecting a Graph

Once the IncidentView is fully loaded and filled with details, a user can select a graph to view. The actions combobox widget in IncidentView is binded to call the method `select_graph()` whenever a graph is selected within its drop down list. The method clears the graph and then passes the name of the selected graph to `fill_graph()` in IncidentController.

```
def select_graph(self, selected_graph: str):
    """
    This method clears the graph figure and passes the selected graph to fill_graph() of
    IncidentController.

    :param selected_graph: Name of selected graph.
    """
    self.graph_widgets[1].clear()
    self.controller.fill_graph(selected_graph)
```

Code Extract 26: Select Graph method in IncidentView

The method `fill_graph()` first obtains the views graph colours and then the time data from the Incident object. Depending on the passed selected graph name, the method gets the necessary acceleration data from the Incident object, and passes it to `plot_graph()` in IncidentView.

In the case of the selected graph name being “All Acceleration”, the data of X-Acceleration, Y-Acceleration, and Z-Acceleration are put into a list. This list is then looped through, with each form of data being passed to the method `plot_graph()` with a unique colour. Finally, the method `set_graph()` is called and is passed the time data, and the selected graph name.

```
def fill_graph(self, selected_graph: str):
    """
    This method gets the acceleration data of the incidents timestamps, depending on the passed
    selected graph. It then fills the graph section of IncidentView.

    :param selected_graph: Name of the selected graph.
    """
    time_data = self.incident.get_timestamps_time()
    graph_colors = self.incident_view.get_graph_colors()
    if selected_graph == "All Acceleration":
        timestamps_data = [self.incident.get_timestamps_x(), self.incident.get_timestamps_y(),
                           self.incident.get_timestamps_z()]
        for i, acc_data in enumerate(timestamps_data):
            self.incident_view.plot_graph(time_data, acc_data, graph_colors[i])
    else:
        if selected_graph == "X-Acceleration":
            acc_data = self.incident.get_timestamps_x()
        elif selected_graph == "Y-Acceleration":
            acc_data = self.incident.get_timestamps_y()
        elif selected_graph == "Z-Acceleration":
            acc_data = self.incident.get_timestamps_z()
        else:
            acc_data = self.incident.get_timestamps_svm()
        self.incident_view.plot_graph(time_data, acc_data, graph_colors[2])
    self.incident_view.set_graph(time_data, selected_graph)
```

Code Extract 27: Fill Graph method in IncidentController

The method `plot_graph()` simply takes the passed time data, acceleration data, and graph colour to create a subplot within the GUI's graph. The method `set_graph()` sets the label names, title name, and X-axis limits of the graph.

```
def plot_graph(self, time_data: List[float], acc_data: List[float], color: str):
    """
    This method plots a colored graph using the timestamps time-data and acceleration-data.

    :param time_data: Timestamps time-data for X-axis.
    :param acc_data: Timestamps acceleration-data for Y-axis.
    :param color: Color of graph.
    """
    self.graph_widgets[1].plot(time_data, acc_data, color=color, marker=".")

def set_graph(self, time_data: List[float], selected_graph: str):
    """
    This method sets the title, labels, and legend of the graph depending on the name of the
    selected graph. It also sets the X-axis limit using the timestamps time-data.

    :param time_data: Timestamps time-data for X-axis.
    :param selected_graph: Name of selected graph.
    """
    self.graph_widgets[1].set(title=selected_graph, xlabel="Time (s)",
                              xlim=(time_data[0], time_data[-1]))
    self.graph_widgets[1].title.set_color(self.primary_fg)
    self.graph_widgets[1].grid(color=self.primary_bg)
    if selected_graph == self.texts[0]:
        self.graph_widgets[1].legend(labels=[self.columns[1], self.columns[2], self.columns[3]],
                                     labelcolor=self.primary_fg, frameon=False)
    if selected_graph == self.texts[1]:
        self.graph_widgets[1].set(ylabel=self.columns[4])
    else:
        self.graph_widgets[1].set(ylabel="Acceleration (m/s$^2$)")
    self.graph_widgets[0].canvas.draw()
```

Code Extract 28: Plot Graph and Set Graph methods in IncidentView

4.3.7 Exporting Graph and CSV

To export a graph the user must first select one and then click a button to call the method `export_graph()` in `IncidentController`. The method checks if a graph has been selected, and if not, displays an error message in `IncidentView` saying to select a graph. Otherwise, it checks if the exports directory exists, and if it not, it creates it.

Next it creates a path name for the graph PNG file, including the Incident ID, and selected graph name. It then passes it to the method `export_graph()` in `IncidentView`, and displays a message saying the graph has been successfully exported to a certain path.

```
def export_graph(self, selected_graph: str):
    """
    This method exports the selected graph in IncidentView as a PNG file to an exports
    directory.

    :param selected_graph: Name of the selected graph.
    """
    if selected_graph == "":
        self.home_view.show_message("Please select a graph to export!")
    else:
        graph_path = self.join_path("exports", "graphs")
        if not self.check_path(graph_path):
            self.create_path(graph_path)
        incident_id = self.incident.get_incident_id()
        file_path = self.join_path(graph_path, f"Incident {incident_id} - {selected_graph}")
        self.incident_view.export_graph(file_path)
        self.home_view.show_message(f"Successfully exported graph:\n'{file_path}.png'")
```

Code Extract 29: Export Graph method in IncidentController

The method `export_graph()` in `IncidentView` uses the method `savefig()` to save a PNG file of the current graph with the passed path name.

```
def export_graph(self, graph_path: str):  
    """  
    This method exports the graph figure to the passed graph path.  
  
    :param graph_path: Name of the graph path.  
    """  
    self.graph_widgets[0].savefig(graph_path)
```

Code Extract 30: Export Graph method in IncidentView

To export a CSV of the incident, the method `export_csv()` is called within `IncidentController`. This implements the exact same process as the method `export_graph()`, except it calls the method `export_timestamps()` in the model `Incident` object, instead of returning to a method within `IncidentView`.

This method uses the `csv` module of Python to open and write a new CSV file of a passed path name. It first writes a header row, containing the column names of the timestamp data. It then loops through the incident's timestamps and writes rows containing the data of each one.

```
def export_timestamps(self, csv_path: str):  
    """  
    This method exports all data from the List of Timestamp objects to a CSV file of a passed  
    path.  
  
    :param csv_path: Name of CSV path.  
    """  
    with open(f"{csv_path}.csv", "w", newline="", encoding="utf-8") as file:  
        csv = writer(file, delimiter=",")  
        csv.writerow(["Timestamp ID", "Timestamp", "X-Acceleration", "Y-Acceleration",  
                      "Z-Acceleration", "Signal Vector Magnitude"])  
        for timestamp in self.timestamps:  
            csv.writerow([str(timestamp.get_timestamp_id()), timestamp.get_time(),  
                          str(timestamp.get_x_acc()), str(timestamp.get_y_acc()),  
                          str(timestamp.get_z_acc()), str(timestamp.get_svm())])
```

Code Extract 31: Export Timestamps method in Incident

4.3.7 GUI Design

The final GUI design of the IoTumble program turned out very similar to the initial GUI sketches. However, its biggest change is its adoption in style. The GUI incorporates a dark colour scheme, with a purple accent for its text and graphs.

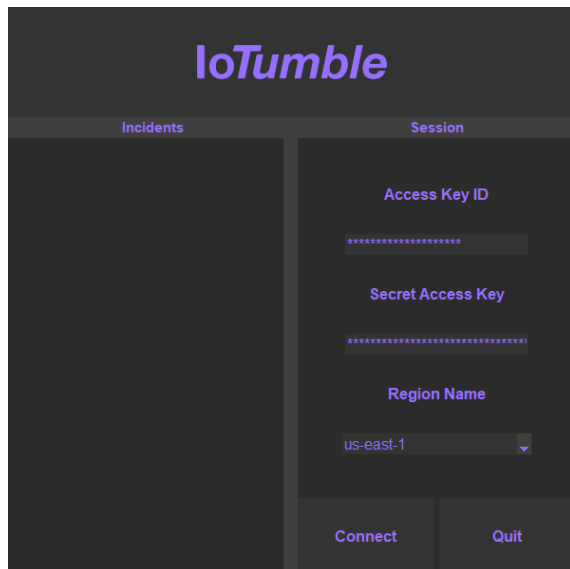


Figure 31: HomeView before connection

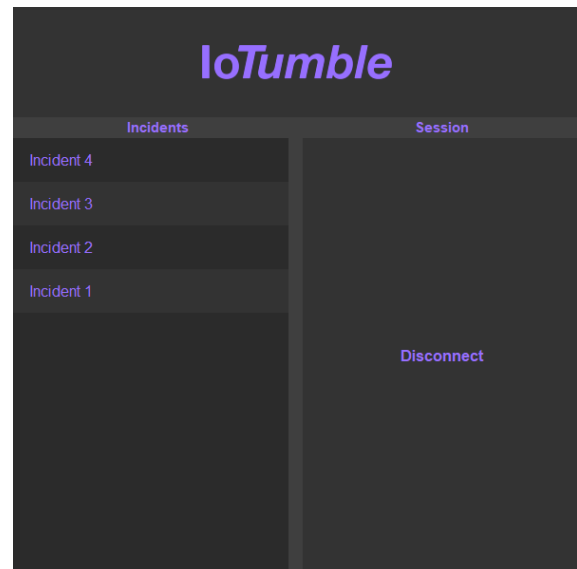


Figure 32: HomeView after connection

Both GUI views exhibit the IoTumble logo in the header sections of their windows, clearly showing what program is being used. The program removes the default Windows border for a more flat and simplistic design. Everything is clearly labelled, with titles above each section stating their purpose. The GUI views can seamlessly be switched between each other, with little to no loading time.



Figure 33: IncidentView filled with details and a selected graph

Chapter 5. Testing and Results

In this chapter the project's testing and results have been described. The following sections look at the results that were discovered, and the testing involved in the projects three aspects, including the IoTumble device, AWS architecture, and the IoTumble program.

5.1. IoTumble Device

The IoTumble device was constantly being tested throughout its development. To avoid having to continually fall just to perform testing on the device's methods, the data from a single fall was printed, and then placed into a mock list of recorded timestamps. This also meant that the device did not have to always be connected to its battery.

When the aspect of fall detection was tested, debug statements were printed into a terminal as timestamps were checked for thresholds and inactivity. In this way, incidents didn't always have to be published to AWS, and the history of timestamps could be analysed for problems through a terminal instead.

```
pi@raspberrypi:~/IoTumble/raspberrypi $ python main.py
svm | NO | Checked if 10.41250128851013 was greater than 20 | THRESHOLDS
svm | NO | Checked if 9.92448484638695 was greater than 20 | THRESHOLDS
svm | NO | Checked if 10.062980064122335 was greater than 20 | THRESHOLDS
svm | NO | Checked if 9.886426230906633 was greater than 20 | THRESHOLDS
svm | NO | Checked if 9.387131100198747 was greater than 20 | THRESHOLDS
svm | NO | Checked if 9.708801423109756 was greater than 20 | THRESHOLDS
svm | NO | Checked if 8.406398578038074 was greater than 20 | THRESHOLDS
svm | NO | Checked if 0.6373568097665059 was greater than 20 | THRESHOLDS
svm | NO | Checked if 0.26314003237857975 was greater than 20 | THRESHOLDS
svm | NO | Checked if 0.25117279321208336 was greater than 20 | THRESHOLDS
svm | YES | Checked if 34.74126146212883 was greater than 20 | THRESHOLDS
x | YES | Checked if 20.0840192 was greater than 18 | THRESHOLDS
Thresholds reached
INACTIVITY | YES | Checking if 2.6281822 - 2.4712758 = 0.1569064
INACTIVITY | YES | Checking if 2.3143694 - 2.4712758 = -0.1569064
INACTIVITY | YES | Checking if 0.0392266 - 0.2157463 = -0.1765197
INACTIVITY | YES | Checking if 0.392266 - 0.2157463 = -0.1765197
INACTIVITY | YES | Checking if 5.3359308 - 9.4536106 = -0.11767979999999995
INACTIVITY | YES | Checking if 5.571290399999999 - 9.4536106 = -0.11767979999999995
INACTIVITY | YES | Checking if 9.69890800143617 - 9.776913812764434 = -0.07802293262081683
INACTIVITY | YES | Checking if 9.854936745385253 - 9.776913812764434 = 0.07802293262081683
Incident detected
```

Figure 34: Debug print statements for the IoTumble Device

The device was tested under various situations and general everyday tasks. Focus was put into determining the threshold values that needed to be used within code. Therefore, most of the testing involved falling to ground onto a large cushion, to prevent injuries. The device was tested falling forwards, backwards, to the right, to the left, and in various other angles.

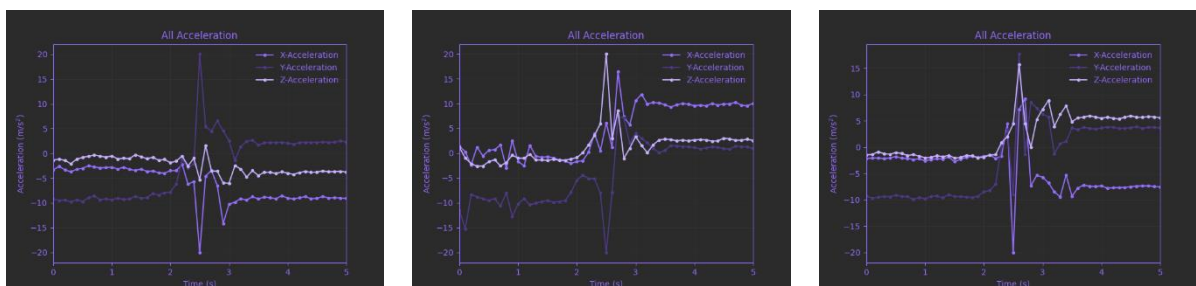


Figure 35: Graphs visualising falling forward, backward, and to the right

It was ultimately found that an absolute value of 18 to 20 was a good threshold to use, as it was never reached unless under high acceleration. Only the SVM and at least one axis needed to reach this amount, as in some cases not all axes measured high acceleration at the same time. It was also found that whenever these thresholds were reached, the check for inactivity would typically ignore non-fall incidents. However, there are situations where this isn't always the case.

If a user moves or gets up immediately after a fall, the incident will not be published. This is unlikely to happen in a real-life situation though, due to injury or shock from the user. Another discovered issue is that when thresholds are purposefully reached, and the user doesn't move afterwards, an incident will be published. Nevertheless, these issues are something to consider for possible improvements to the project.

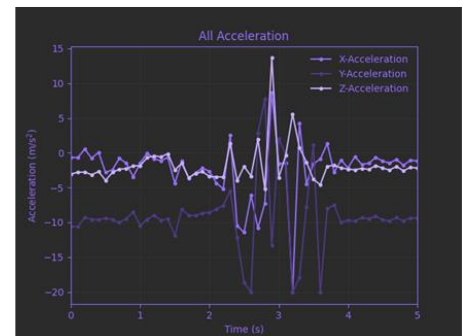


Figure 36: Jumping up and down followed by inactivity

5.2. AWS Architecture

The AWS architecture was tested by using the services provided. Firstly, the publishing of messages to AWS IoT was tested using the MQTT test client. This could subscribe to an IoT topic and monitor any MQTT messages being passed into it. This was a good way of establishing that the device could successfully publish messages to a certain IoT topic.

Secondly, the storing of incidents was tested by manually creating an item within a DynamoDB table, using the mock fall incident data that was being used by the IoTumble device. It was here that the JSON format of an item was established, as well as the tables optimisation by including an incident count item.

The DynamoDB table was further tested by publishing incidents from the IoTumble device and checking to see if they had successfully inserted into the table. This resulted in being a quite efficient form of data logging and worked with little to no issues.

The last aspect to test was the triggering of DynamoDB streams and the created AWS Lambda function. The function itself was tested against a configured test event, that was a mock structure of a DynamoDB stream. A successful test resulted in the incident count item in the table being incremented, and an email being sent to all subscribers of an SNS topic including the incidents details.

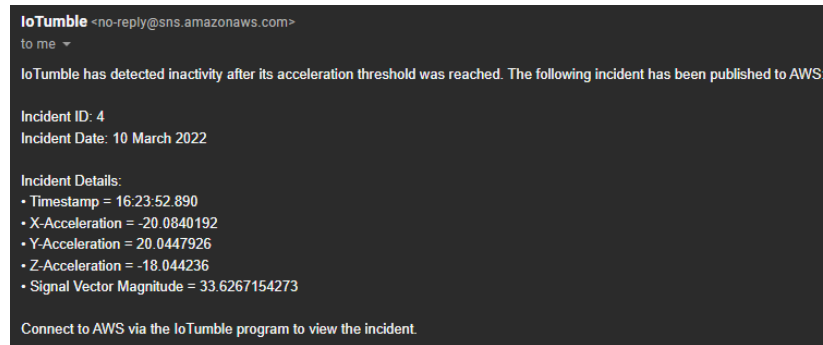


Figure 37: Amazon SNS email showcasing incident details

5.3. IoTumble Program

While developing the program, the installed plugin tool PyLint was used to check for errors in code, to enforce the PEP 8 coding standard, and to provide suggestions. One of those suggestions was the use of docstrings to help with code explanation. Docstrings were added to every module, class, constructor, and method within the project. Ultimately, the tool made it very easy to test the program, as the quality of code was kept to a high standard, with the project reaching a final code rating of 10.00/10.

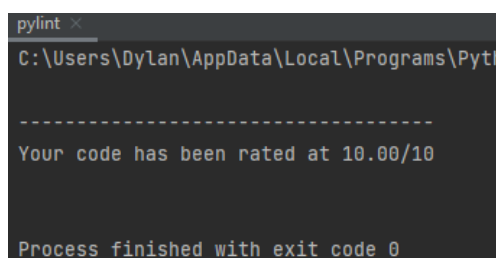
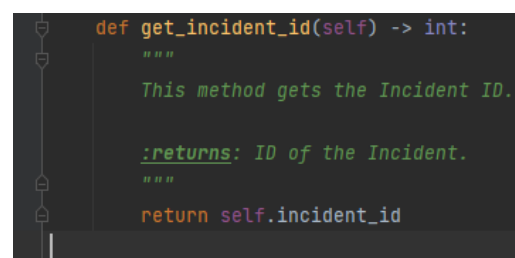


Figure 38: Final PyLint code rating



Code Extract 32: Docstring for a method

As the project incorporated an Agile approach in software development, features of the IoTumble program were tested regularly. The debugger within PyCharm was often used to troubleshoot issues during testing, by setting breakpoints and stepping through the code. In some cases, print statements were used to establish the cause of errors. However, there weren't really any major issues that hindered development or caused setbacks.

The result from testing the GUI program, is that it can successfully visualise published incidents in readable form. Graphs distinguish their subplots by using unique colours, with little markers to signify timestamps. Individual timestamps can be analysed by scrolling through the details tree view widget.

| Timestamp | X-Acceleration | Y-Acceleration | Z-Acceleration | SVM |
|--------------|----------------|----------------|----------------|---------------|
| 16:23:55.438 | 9.022118 | -0.5491724 | -4.3541526 | 10.0328883376 |
| 16:23:55.336 | 9.022118 | -0.3530394 | -4.5895122 | 10.128517772 |
| 16:23:55.234 | 8.8652116 | -0.5491724 | -4.3933792 | 9.9093565802 |
| 16:23:55.132 | 9.1790244 | -0.392266 | -4.3541526 | 10.1669565955 |
| 16:23:55.031 | 8.8652116 | -0.6668522 | -4.1580196 | 9.8145705746 |
| 16:23:54.929 | 8.825985 | -0.4314926 | -4.6287388 | 9.9754408405 |
| 16:23:54.827 | 9.022118 | -0.5099458 | -4.4326058 | 10.0651205707 |
| 16:23:54.725 | 9.0613446 | -0.6668522 | -4.2756994 | 10.0416265204 |
| 16:23:54.623 | 8.9436648 | -0.588399 | -4.1972462 | 9.8970818478 |
| 16:23:54.521 | 8.9436648 | -0.4314926 | -4.314926 | 9.9395126794 |

Figure 39: Details tree view displaying timestamps

Finally, the program can successfully export an incident as a selected graph, or as a CSV file. The advantage that this provides, is that they can be examined at any time, without connection to AWS. They could be shared to those who do not have access to the IoTumble program, for other visualisation or research purposes.

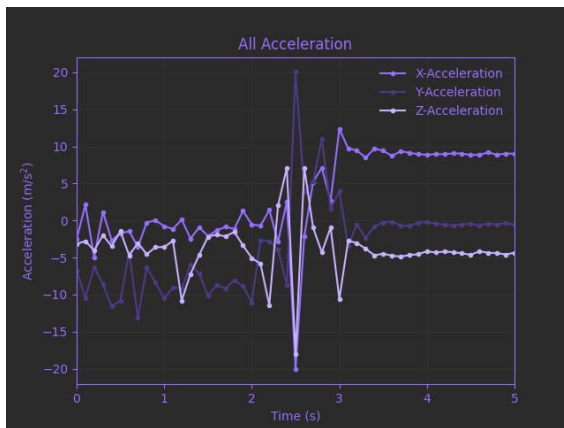


Figure 40: Exported graph

| | A | B | C | D | E | F |
|----|--------------|-----------|----------------|----------------|----------------|-------------------------|
| 1 | Timestamp ID | Timestamp | X-Acceleration | Y-Acceleration | Z-Acceleration | Signal Vector Magnitude |
| 2 | 0 | 23:50.3 | -2.5105024 | -6.7469752 | -3.138128 | 7.853161401 |
| 3 | 1 | 23:50.4 | 2.1182364 | -10.4342756 | -2.7850886 | 11.00535103 |
| 4 | 2 | 23:50.5 | -4.9425516 | -6.276256 | -4.0403398 | 8.952348932 |
| 5 | 3 | 23:50.6 | 1.1375714 | -8.5906254 | -2.0005566 | 8.893544859 |
| 6 | 4 | 23:50.7 | -2.6674088 | -11.571847 | -3.4519408 | 12.366835 |
| 7 | 5 | 23:50.8 | -1.7259704 | -10.8265416 | -1.4513842 | 11.05891012 |
| 8 | 6 | 23:50.9 | -1.4513842 | -4.2364728 | -4.707192 | 6.4970666 |
| 9 | 7 | 23:51.1 | -3.5696206 | -13.1016844 | -3.0204482 | 13.91112622 |
| 10 | 8 | 23:51.2 | -0.2745862 | -6.3154826 | -4.5502856 | 7.788826426 |
| 11 | 9 | 23:51.3 | 0.0392266 | -8.3552658 | -3.6088472 | 9.101416562 |
| 12 | 10 | 23:51.4 | -0.784532 | -10.5127288 | -3.5696206 | 11.12992132 |
| 13 | 11 | 23:51.5 | -1.1375714 | -9.022118 | -2.745862 | 9.499075746 |
| 14 | 12 | 23:51.6 | 0.1176798 | -9.1397978 | -10.787315 | 14.13916254 |
| 15 | 13 | 23:51.7 | -2.4320492 | -6.0016698 | -7.2176944 | 9.696907556 |
| 16 | 14 | 23:51.8 | -0.9414384 | -7.1392412 | -4.6287388 | 8.560390999 |
| 17 | 15 | 23:51.9 | -2.0790098 | -10.1596894 | -2.1966896 | 10.60033092 |
| 18 | 16 | 23:52.0 | -1.3337044 | -8.6690786 | -1.8828768 | 8.970892723 |
| 19 | 17 | 23:52.1 | -0.8629852 | -9.1790244 | -2.157463 | 9.468573229 |
| 20 | 18 | 23:52.2 | -1.1375714 | -8.041453 | -1.5298374 | 8.264347374 |
| 21 | 19 | 23:52.3 | 1.2944778 | -8.825985 | -3.3734876 | 9.536986032 |
| 22 | 20 | 23:52.4 | -0.5491724 | -11.0619012 | -5.0602314 | 12.17674794 |
| 23 | 21 | 23:52.5 | -0.7060788 | -2.6674088 | -5.8447634 | 6.463348682 |
| 24 | 22 | 23:52.6 | 1.4121576 | -2.8243152 | -11.4149406 | 11.84364025 |

Figure 41: Exported CSV

Chapter 6. Discussion and Conclusion

6.1. Discussion

IoTumble has successfully implemented a comfortable and wearable device, which detects and publishes possible fall incidents to AWS. Subscribers are alerted via email on the publishing of new incidents, and an accessible GUI program has been developed to visualise their data in readable form. Therefore, it can be said that the aims of the project have been achieved.

An aspect that I'm particularly proud of is the overall presentation of IoTumble. The device is compact, wireless, and fits perfectly into its casing and holder. The actual name IoTumble describes the major aspects of the project, by combining IoT and the synonym "tumble" for fall detection. The program includes the logo in its design and stays consistent by using the purple colour as its accent. Every aspect of the project ties together, making a truly presentable and marketable product. However, the project could still be further developed in its many aspects.

From testing the IoTumble device, it became quite evident that it could be improved in various ways. The first area of improvement would be to implement a more complicated method of fall detection. A gyroscope sensor could be incorporated to calculate the angular motion of the device. The roll and pitch of the device's accelerometer could have been used to calculate its orientation. The device would only then check for inactivity if the user had become horizontal.

Another area of improvement is for the IoTumble program, which could have updated its incidents list in real-time or alerted the user of new incidents within the program. The actual visualisation of data could be expanded upon also, with interactive graphs that allow the user to individually select and display its timestamps. These improvements could be integrated due to the programs object-oriented design and inclusion of the MVC design pattern.

6.2. Conclusion

In conclusion, the development of IoTumble has been a success. The project has enhanced the application of fall detection and wearable IoT devices that interface with the cloud. It can be concluded that the chosen hardware has effectively allowed for the assembly of a comfortable and wearable device to detect possible fall incidents from a user. The publishing of data to the cloud was only made possible through the implementation of the AWS IoT Device SDK for Python.

It can also be concluded that the architectural design of AWS has allowed for a fast and cost-efficient way of cloud alerting and data logging. An easy-to-use GUI program that visualises fall incident data has also been successfully developed. Without the use of the Python packages Tkinter and Matplotlib, as well as the AWS SDK for Python (Boto3), the program would not be capable of interfacing with AWS or displaying published incidents in readable form.

From my own personal reflection, I can conclude that I have improved many skills from working on this project. I have become more knowledgeable in the field of IoT and cloud applications. I have improved my creative skills, by developing a stylised GUI. Lastly, I have further developed my technical skills, particularly in AWS, embedded systems, and Python. I would like to further develop this project after I graduate, as I see great potential in IoTumble.

IoTumble ultimately offers a user-friendly experience, by wirelessly and seamlessly publishing the data of possible fall incident detections to the cloud. It alerts the necessary personnel via email and provides a platform to analyse the incident data further. IoTumble has established that it has potential to be used in the industry of healthcare, to detect and visualise possible fall incidents.

References

- [1] S. Greengard, *The Internet of Things*, 1st Edition. Cambridge, MA, USA: MIT Press, 2015.
- [2] N. Ruparelia, *Cloud Computing*, 1st Edition. Cambridge, MA, USA: MIT Press, 2016.
- [3] "What is cloud computing?" Amazon.com. [Online]. Available:
<https://aws.amazon.com/what-is-cloud-computing> [Accessed on: Sep. 30, 2021].
- [4] L. Youngmin, Y. Hongjin, K. Ki-Hyung, and C. Okkyung, "A real-time fall detection system based on the acceleration sensor of smartphone" *International Journal of Engineering Business Management*, vol. 10, Jan. 2018. Accessed on: Oct. 1, 2021. [Online]. Available:
<https://journals.sagepub.com/doi/10.1177/1847979017750669>
- [5] "What is Raspberry Pi?" Raspberrypi.org. [Online]. Available:
<https://www.raspberrypi.org/help/what-%20is-a-raspberry-pi>
[Accessed on: Oct. 10, 2021].
- [6] "Raspberry Pi Products" Raspberrypi.com. [Online]. Available:
<https://www.raspberrypi.com/products> [Accessed on: Oct. 10, 2021].
- [7] "Raspberry Pi Datasheets" Raspberrypi.com. [Online]. Available:
<https://datasheets.raspberrypi.com> [Accessed on: Oct. 10, 2021].
- [8] "ADXL345 Datasheet" Analog.com. [Online]. Available:
<https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL345.pdf>
[Accessed on: Oct. 16, 2021].
- [9] "Cloud computing with AWS" Amazon.com. [Online]. Available:
<https://aws.amazon.com/what-is-aws> [Accessed on: Oct. 17, 2021].

- [10] A. Gillis. “Amazon Web Services (AWS)” Techtarget.com. [Online]. Available:
<https://searchaws.techtarget.com/definition/Amazon-Web-Services>
[Accessed on: Oct. 17, 2021].
- [11] “How AWS IoT works” Amazon.com. [Online]. Available:
<https://docs.aws.amazon.com/iot/latest/developerguide/aws-iot-how-it-works.html>
[Accessed on: Oct. 17, 2021].
- [12] “AWS SDK for Python (Boto3)” Amazon.com. [Online]. Available:
<https://aws.amazon.com/sdk-for-python> [Accessed on: Oct. 19, 2021].
- [13] “LiPo SHIM” Thepihut.com. [Online]. Available:
<https://thepihut.com/products/lipo-shim> [Accessed on: Jan. 14, 2022].
- [14] “PyCharm Get Started” JetBrains.com. [Online]. Available:
<https://www.jetbrains.com/help/pycharm/quick-start-guide.html>
[Accessed on: Jan. 14, 2022].
- [15] “What is PyLint?” Pycqa.org. [Online]. Available:
<https://pylint.pycqa.org/en/latest/intro.html> [Accessed on: Jan. 14, 2022].
- [16] “AWS IoT Device SDK for Python” Github.com. [Online]. Available:
<https://github.com/aws/aws-iot-device-sdk-python> [Accessed on: Jan. 14, 2022].
- [17] “Adafruit CircuitPython ADXL34x” Github.com. [Online]. Available:
https://github.com/adafruit/Adafruit_CircuitPython_ADXL34x
[Accessed on: Jan. 14, 2022].
- [18] “tkinter – Python interface to Tcl/Tk” Python.org. [Online]. Available:
<https://docs.python.org/3/library/tkinter.html> [Accessed on: Jan. 14, 2022].

- [19] “*Matplotlib: Visualization with Python*” Matplotlib.org. [Online]. Available: <https://matplotlib.org> [Accessed on: Jan. 14, 2022].
- [20] “*What Is Amazon DynamoDB?*” Amazon.com. [Online]. Available: <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html> [Accessed on: Jan. 18, 2022].
- [21] “*Change Data Capture for DynamoDB Streams*” Amazon.com. [Online]. Available: <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html> [Accessed on: Jan. 18, 2022].
- [22] “*What is AWS Lambda?*” Amazon.com. [Online]. Available: <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html> [Accessed on: Jan. 18, 2022].
- [23] “*What is Amazon SNS?*” Amazon.com. [Online]. Available: <https://docs.aws.amazon.com/sns/latest/dg/welcome.html> [Accessed on: Jan. 18, 2022].
- [24] M. Sami. “*SDLC Models and Methodologies*” Medium.com. [Online]. Available: <https://medium.com/@melsatar/software-development-life-cycle-models-and-methodologies-297cfe616a3a> [Accessed on: Jan. 24, 2022].

Appendices

Appendix A – Source Code

The source code of the IoTumble program and IoTumble device have been included within the auxiliary submission alongside this report. Instructions on how to install the source code can be found within its readme.txt. The following screenshot is the entire source code of the created AWS Lambda function within the AWS architecture:

```

1  from datetime import datetime
2  import boto3
3
4  def lambda_handler(event, context):
5      for record in event["Records"]:
6          incident = record["dynamodb"]["NewImage"]
7          incident_id = incident["pk"]["N"]
8          timestamps = incident["msg"]["M"]
9
10         # Getting the maximum SVM of the incident
11         svm_list = []
12         for i in range(len(timestamps)):
13             timestamp = timestamps[str(i)]["M"]
14             svm = float(timestamp["svm"]["N"])
15             svm_list.append(svm)
16         max_svm = max(svm_list)
17
18         # Getting the timestamp data of the maximum SVM
19         timestamp_data = []
20         for i in range(len(timestamps)):
21             timestamp = timestamps[str(i)]["M"]
22             svm = float(timestamp["svm"]["N"])
23             if svm == max_svm:
24                 timestamp_data.append(float(timestamp["ep"]["N"]))
25                 timestamp_data.append(round(float(timestamp["x"]["N"]), 8))
26                 timestamp_data.append(round(float(timestamp["y"]["N"]), 8))
27                 timestamp_data.append(round(float(timestamp["z"]["N"]), 8))
28                 timestamp_data.append(round(float(timestamp["svm"]["N"]), 10))
29
30         # Getting the date and time of the incident
31         date_time = datetime.fromtimestamp(timestamp_data[0])
32         date = date_time.strftime("%d %B %Y")
33         time = date_time.strftime("%H:%M:%S.%f")[:-3]
34
35         # Publish alert to SNS topic
36         sns = boto3.resource("sns")
37         topic = sns.Topic("arn:aws:sns:us-east-1:525881020712:iotumble_alert")
38         topic.publish(
39             Subject="Incident Alert",
40             Message="IoTumble has detected inactivity after its acceleration " +
41                 "threshold was reached. The following incident has been " +
42                 "published to AWS:\n\nIncident ID: " + incident_id +
43                 "\n\nIncident Date: " + date + "\n\nIncident Details: " +
44                 "\n• Timestamp = " + time +
45                 "\n• X-Acceleration = " + str(timestamp_data[1]) +
46                 "\n• Y-Acceleration = " + str(timestamp_data[2]) +
47                 "\n• Z-Acceleration = " + str(timestamp_data[3]) +
48                 "\n• Signal Vector Magnitude = " + str(timestamp_data[4]) +
49                 "\n\nConnect to AWS via the IoTumble program to view the " +
50                 "incident.\n\n"
51         )
52
53         # Increment table's incident count
54         dynamodb = boto3.resource("dynamodb")
55         table = dynamodb.Table("iotumble_incidents")
56         table.update_item(
57             Key={
58                 "pk": 0,
59                 "sk": "count"
60             },
61             UpdateExpression="SET msg = msg + :increment",
62             ExpressionAttributeValues={
63                 ":increment": int(1)
64             }
65         )
66

```

Appendix B – Copy of Poster



**UNIVERSITY OF
LIMERICK**
OLLSCOIL LUIMNIGH



**Department of
Electronic and
Computer Engineering**

IoTumble

IoT and Fall Detection – cloud alerting, data logging and visualisation

Dylan Coffey – 18251382
Bachelor of Science in Cyber Security and IT Forensics

Introduction

The industry of healthcare can greatly benefit from the **Internet of Things (IoT)** and the **cloud**, specifically within **wearable technologies** and the application of **fall detection**. Detecting when a patient **falls**, and **alerting** the necessary personnel in a timely manner, could be a matter of life or death for a patient.

IoTumble provides a **wearable** device which can detect and publish possible **fall incidents** to **Amazon Web Services (AWS)**. Worn on the waist, the **IoTumble** device is built using the ultra-compact **Raspberry Pi Zero 2 W**, which interfaces with an accelerometer to track **3-axis acceleration**, and detect **fall incidents**.

The incidents can then be **visualised** in readable form, via the **IoTumble** program. Developed in **Python**, it provides an **easy-to-use GUI**, that can view the incidents published to **AWS**. The program can **plot** these incidents as **graphs**, or export them in **CSV** format.

Aims

The main aims of **IoTumble** were as follows:

- Develop a **comfortable** and **wearable** device that can **wirelessly** detect possible **fall incidents** from a user.
- Interface the device with **AWS** to store the accelerometer data of **fall incidents**.
- Build the necessary **AWS** architecture to hold the data, and **alert** the necessary personal in a timely manner.
- Develop an **easy-to-use GUI** program to **visualise** the data in a readable form.

Method

The method for the projects development can be separated into the following categories:

- Hardware:** The **IoTumble** device was built using the **Raspberry Pi Zero 2 W**. Running on a **Linux-based OS**, the board provides **wireless** connectivity and a 40-pin header to allow for more components. The **ADXL345** accelerometer measures its **3-axis acceleration** and communicates it through **I2C**. The **LiPo SHIM** is a small PCB with a JST connector for a power supply. To reduce the devices cost and **environmental impact**, an old smartphone battery was **repurposed** by soldering a female JST wire to its pins.

Results

The first result of the project is that a **comfortable** and **wearable** device has been built to detect and publish possible **fall incidents** to **AWS**. The device sits in an old camera holder, and is attached to the users belt.

The second result, is that when **incidents** are inserted into a **DynamoDB** table, a **Lambda** function is called to send an **Amazon SNS** email to **alert** all subscribers of a **SNS** topic.



Figure 1: IoTumble Device and repurposed battery



Figure 4: IoTumble Device attached to a belt



Figure 2: AWS Architecture Diagram



Figure 5: Amazon SNS Email for a detected incident

- Software:** The **IoTumble** program was developed in **Python**. It uses **Tkinter** for its **GUI**, **Matplotlib** to plot its graphs, and the **SDK Boto3**, to allow for integration with **AWS** and the database service **DynamoDB**. The **IoTumble** device uses the **AWS IoT Device SDK** to access **AWS IoT** and send **MQTT** messages to a **DynamoDB** table.



Figure 3: IoTumble Program displaying an incident

- Fall Detection:** The **acceleration** data is monitored and used to calculate the **Signal Vector Magnitude (SVM)**. If these go over certain thresholds, the data is combined to check if the user has become **inactive**, and if they have, it detects a possible **fall incident**.

$$SVM = \sqrt{(A_x)^2 + (A_y)^2 + (A_z)^2}$$

Equation 1: Signal Vector Magnitude (SVM)



Figure 6: Graph showing thresholds being reached, followed by inactivity

Conclusion and Reflection

In conclusion, the development of **IoTumble** has been a success. I have become more **knowledgeable** in the field of **IoT** and **cloud** applications. I have improved my **creative** skills, by developing a stylised **GUI**. I have further developed my **technical** skills, particularly in **AWS**, **embedded systems**, and **Python**. If I was to change an aspect of the project, I would incorporate a **gyroscope** sensor to calculate the angular motion of the device, and produce more accurate **fall** detections.

Acknowledgements

Jacqueline Walker - FYP Supervisor