



University
of Glasgow | School of
Computing Science

Honours Individual Project Dissertation

USING IoT EDGE COMPUTE TO MONITOR OFFICE NOISE POLLUTION

Emily Hathaway
March 1, 2020

Abstract

“With decreasing device and cloud hosting costs, the Internet of Things has expanded rapidly. This project aims to demonstrate a manageable way to process data at the point of its creation to decrease costs and ensure a higher degree of privacy. The resulting system had marginal per-device costs that scale linearly with deployment size.”

Acknowledgements

I would like to give thanks to Jeremy Singer for being a flexible and accommodating adviser, Nicholas Bailey for advice on SPL meters, Colin Cooper for debugging some LoRa issues with me and the other level 4 students for keeping me sane.

Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: Emily Hathaway Date: 25 March 2020

Contents

1	Introduction	1
2	Background	2
2.1	Digital Signal Processing (DSP)	2
2.2	Auditory perception	2
2.3	Octave bands	3
2.4	Existing solutions	3
2.5	Previous Works	4
3	Analysis/Requirements	5
3.1	Motivation for edge compute	5
3.2	What defines edge computing in this scenario?	5
3.3	Problem definition	5
3.4	Summary	6
4	Design	7
4.1	Architecture	7
4.2	Interface design	8
4.3	Application protocol	8
5	Implementation	10
5.1	High level implementation	10
5.2	Microcontroller selection	12
5.3	Peripheral selection	12
5.3.1	Bill of Materials	14
5.3.2	Programming	14
5.3.3	On-device computation	15
5.4	Service Providers	15
5.5	Web Stack design	17
5.5.1	Web Application Framework	18
5.6	Spectrogram	20
5.7	Device transmission technology	21
5.8	Data packing and unpacking	23
5.9	Development practices	23
5.10	Summary	24
6	Evaluation	25

6.1	Device testing	25
6.2	Cost analysis	28
7	Conclusion	29
7.1	Summary	29
7.2	Reflection	29
7.3	Future work	29
	Appendices	30
A	Notable device code	30
B	Google Cloud setup	31
	Bibliography	32

1 | Introduction

Loud noise has been demonstrated to be a burden on productivity (Kjellberg (1990)), especially regarding office noise types (eg: high and low frequencies (Broadbent (1957)) or “distracting” noises). Broadbent (1957), for example, demonstrated that both high and low frequencies increased the rate of errors caused by test subjects, with high frequency loud noises leading to a higher error rate than similarly loud low frequencies. Kjellberg (1990) is a meta-study of other works, and concludes that the type of disruption, as well as nature of it’s effects can vary largely- people are less annoyed by “necessary” noises, but more annoyed by noises that they have no emotional connection to. Continual loud noises can be a health risk, with an elevated risk of hypertension, and could potentially form a correlative link between people with mental health issues and those with hearing loss.

At the same time, computing power in inexpensive microcontrollers has exploded in the past 5 years, supporting a growth in the Internet of Things (IoT) industry segment, with Hunke et al. (2017) placing the global market value at \$267Bn. A significantly portion of this value is from growth in the sensor and microcontroller sectors, along with a growth in analytics and cloud infrastructure. This tightly bonds the IoT industry with other fields of study such as big data applications, as well as electrical engineering. This is demonstrated by the amount of big data tools that are specialised to offer IoT related analytics, such as SAS’ Analytics for IoT¹ tool.

This dissertation explores one potential system for monitoring office noise levels using IoT technologies to inform users of noise levels in particular rooms so they can make better informed decisions about where to work from if that is an option. For example, choosing to work in a specific shared-access lab space or from home. It would also be appropriate to store this information if any potential complaints were to be raised to relevant authorities- either the local council or building managers.

Using an ESP32 based development board and serverless cloud technologies, I will construct a system that is accurately able to record perceived loudness and a noise profile and output this data in near real time to a web dashboard. This will be a low cost solution for both initial setup and for recurring monthly costs, and will be tailored for use in urban areas with access to WiFi.

¹https://www.sas.com/en_us/software/analytics-iot.html

2 | Background

2.1 Digital Signal Processing (DSP)

Any signal collected from an external sensor needs to be processed in some way to avoid the influence of random noise. Any random source of error is known as noise, and typically comes from the random movement of electrons (either through a passive device such as a resistor as thermal noise or from the sensor as "flicker noise"). Noise is definable as a probability density, so is statistically predictable. Interference is any specific signal that is not desirable in a captured signal, such as "hum" from mains electricity. Any processing on a captured signal to remove noise or interference is known as DSP (suppression of noise and interference electronically doesn't count, as it is analogue).

DSP is used to emphasise or suppress an element or a range of elements inside a signal. It also can apply to any post-processing of a captured signal. A common form of post-processing is a Fourier transform. This type of mathematical transform converts a function defined in the time domain to a new function defined in the frequency domain. A Fast Fourier Transform (FFT) is a common discrete Fourier transform that uses captured data points to return the constituent sine waves which would, when played back, give the captured sound. The FFT requires 2^N data points that have a constant time period between them, and returns the intensity of each frequency range (or "bin"). The width of each bin is defined by the N data points and the sampling frequency f_s , such that

$$\delta f = \frac{f_s}{2^N}$$

where δf defines the width of a frequency bin. This means that precision of an FFT increases with N and decreases with f_s . Due to Nyquist's theorem, f_s is double the maximum frequency that can be distinguished in the FFT (ie. to detect an 8000Hz signal, the sampling frequency must be $> 16000\text{Hz}$.)

2.2 Auditory perception

The human ear does not have a linear audio response, meaning that the perceived volume of a sound is related to the frequency of the sound. Fletcher and Munson (1933) first published this research and it forms the basis of the equal-loudness curves used today (ISO (2003)). This means that there are multiple weighting curves to bias an unweighted (or Z-weighted) measurement. One such curve is the dBA curve, which can loosely be defined by the following polynomial

$$R_A(f) = \frac{12194^2 f^4}{(f^2 + 20.6^2)\sqrt{(f^2 + 107.7^2)(f^2 + 737.9^2)}(f^2 + 12194^2)} \quad (2.1)$$

$$dBA(f) = 20 \log_{10}(R_A) - 20 \log_{10}(R_A(1000)) \quad (2.2)$$

The purpose of $dBA(f)$ is to define the curve in dB and define the point at which no amplification occurs (1000Hz). There are other weighting functions (such as ITU-468, used widely by European broadcasters and dBC, which is a flatter curve for quick impulse sounds like gunshots), but the dBA curve is simplest to define in a polynomial while also being fairly accurate for general purpose use.

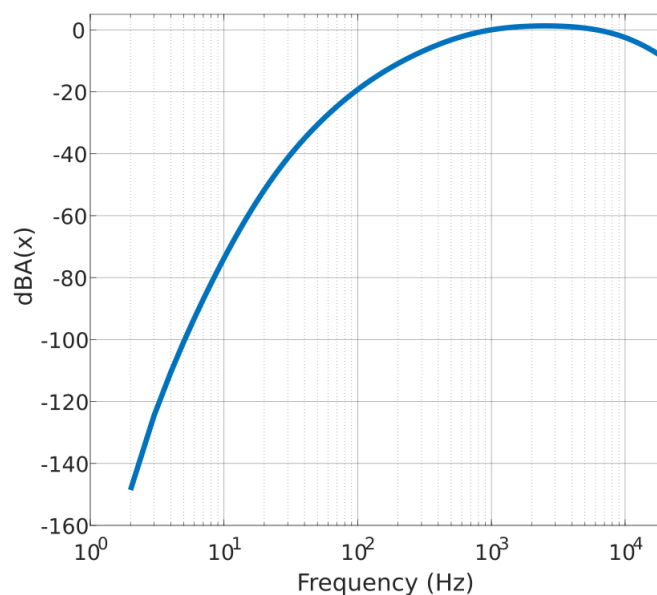


Figure 2.1: Plot of the dBA function over the audible range

2.3 Octave bands

An octave is a range of frequencies that ranges from a base frequency to double the base frequency. This means that as the octave increases, more of the smaller bins from the FFT apply (since they are of constant width). ANS (2004) defines each octave from a 31.25Hz midband frequency (ie. 63, 125, 250, 500, 1000, 2000, 4000, and 8000 Hz), so this should form the basis of sorting each FFT bin into a simpler summary of collected data.

2.4 Existing solutions

There are a few solutions for monitoring noise pollution using IoT technology, and most are engineered with the assumption that they will be deployed by councils for environmental monitoring. This gives an extremely large price tag, with some solutions such as the Libelium Plug & Sense costing thousands of pounds¹. There is also a LoRa enabled kit with a solar panel² for €1500 + VAT, which would be an unattainable price tag unless such information is contractually or legally required.

In the USA, an IoT edge-based solution has been developed to provide gunshot detection information to law enforcement agencies. “Shotspotter” used multiple sensors over a wide area to detect and correlate any potential gunshot events with nearby sensors before reporting information to a server for further analysis before reporting to the police. This solution also integrates with cameras to take advantage of the decreased latency of edge compute to point at the triangulated location of the potential shot. This is similar to this project as sonic events are processed before being reported to any sort of dashboard or cloud processor.

¹Pricing is normally stated as “on request”, but one kit with two sensors was listed at \$7,700, albeit with additional weather sensors <https://www.the-iot-marketplace.com/solutions/noise-polution>

²<https://www.iotsoundsensor.com/buy-the-iotsoundsensor/>

2.5 Previous Works

Microsoft in partnership with the University of Dublin studied the application of IoT noise level monitoring in Little et al. (2017), with a network architecture summarised in Moran (2017). This applied a wide usage of dBA noise monitors aggregated to a single “master gateway” to submit the data to the cloud, and demonstrated that relying on a highly congested WiFi network can lead to data loss. This demonstrates the need to keep the amount of data transmitted in such applications to a minimum, as more processing at the edge would require less bandwidth (while trading off the amount of data that can be investigated for other purposes later).

Samie et al. (2016) created a novel algorithm for distributing part of the workload of multiple devices while optimising the battery life of each device, in a configuration known as “fog” computing. They used an intermediary gateway device to determine if computation should occur on the device or on the cloud depending on the remaining battery life of each device on the edge and the bandwidth utilisation of the gateway. This solution achieved a 1 hour improvement in battery life over a “discrete” solution, although this baseline solution’s configuration is not specified so could be due to a reduction in transmission or in their use of a processor with a dynamic clock speed.

Gallo et al. (2018) developed a network of noise level sensors across a municipal district to keep the levels inside the legal limits. They used a smartphone and an external lavalier microphone that was calibrated with a known source, allowing for the creation of an eco-friendly and simple network while using the Android operating system.

Raj (2019) gave a tutorial in how to construct a similar circuit to the one used my solution (although mine was on a circuit board). Bird (2018) also created a similar tutorial, and formed a part of the initial code exploration for this project. Both projects use Arduino a simple op-amp to amplify and filter the microphone’s analog output before processing. They don’t connect to the internet or calculate the dBA value with reliable precision, but as simple hobby projects they allow an interested technical user to monitor the environmental noise level and it’s constituent frequencies.

3 | Analysis/Requirements

3.1 Motivation for edge compute

There are many motivations to consider performing computation at the edge device rather than using a model with more centralised computation. A major consideration is cost- in a cloud-native style, each second of CPU and RAM utilised is billable. This makes it very important to minimise the active time on the central server, and such penalties don't exist on the device that is deployed in this case. The concern for potentially using more power on the device is not accurate, especially considering the power usage of the WiFi radio.

Privacy is another crucially important concern for this project. Connecting a microphone to the internet is a seriously large potential source for accidentally collecting personal data- which would be a significant risk especially if the device or network was compromised. By processing the signal on the device, the transmitted output is anonymised - allowing the sensor to be deployed in spaces that contain people. Podesta et al. (2014) discusses briefly the concerns of even anonymised data harvesting and the consequences of gaining access to seemingly innocuous data, such as home power usage or numbers of WiFi clients on a network releasing when and where people are inside a house.

A final motivation is the required bandwidth of the system is decreased due to transmissions being both smaller and less frequent. A constant datastream of audio information requires $44100 \times 16 = 705.6\text{kbps}$ for a single sensor (uncompressed, 16 bit 44100 Hz data), whereas using edge compute this can be reduced to tens of bytes every few seconds, massively reducing the bandwidth required. This is more important for a wireless solution, as only one device can use a channel at a time, leading to a higher chance of saturating the available network link. With a smaller data transmission, other transmission technologies can be considered, such as LoRaWAN.

3.2 What defines edge computing in this scenario?

To effectively state that edge compute is actually being used in this scenario, the definition needs to be made clear. Therefore, I define running any calculations to transform a raw sound signal to an output that would be usable to a human client as the required work that must occur on the device- including any FFT runs or noise removal. Due to this requirement, the server component should be extremely simple, since all work will have been performed on any requestable object.

3.3 Problem definition

There are currently a lot of renovation work at universities across the country, including at the University of Glasgow. This has created plenty of annoyance from the loud drilling noises caused by this construction work, especially in the Boyd Orr building (where the majority of undergraduate computing science lab spaces are). This has been reported as an issue by the Staff-Student Liaison Committee as an issue to consider, but is generally treated as inevitable and unavoidable leading to the dis-satisfaction with the disruption being under-reported. If

some data about the level of disruption could be recorded and supplied to the university, then different spaces may be allocated by university timetabling or students could weight up the current disruption levels with other spaces on campus.

Obviously, this solution needs to communicate with external sources through some means and not just display it's generated summary on a screen or LED array. This therefore adds a requirement for the devices to be able to communicate with the outside world, either through WiFi, LoRa or any other wireless method. Wired communication methods aren't suitable in this use case as it would be too inconvenient to run two wires to the device and access to the wired university network is restricted, making any testing in the proposed environment impossible.

While researching the FFT library, I discovered that this would return some seemingly arbitrary numbers, and could not be related to the loudness of the input sound. This led to some further research and meant that this project would either require a pre-calibrated Sound Pressure Level (SPL) sensor or an amplifier that could be calibrated to a source or set of known sources.

Due to the near-real-time nature of this system, it was determined that an interface that updates to the current state would be very useful. The university currently has a set of web dashboards to show the utilisation of computing spaces in the library and engineering buildings. Since these systems already exist, it makes sense to use the same platform, as if a user was interested in the environment enough to check that there's a free desk in a lab then the working conditions in the lab space would also be relevant information to find.

3.4 Summary

The Final requirements for the system are as follows:

This system MUST:

- Process audio on the device
- Show recent loudness
- Show type of noise
- Be accessible to anyone

This system SHOULD:

- Update in real time
- be easy to deploy
- Be as low-cost as possible (BoM and running costs)

4 | Design

4.1 Architecture

This project's architecture is generally similar to a normal client/server pattern, with two different types of client. One type of client is "devices", which are small sensors that transmit their results to the server. The other type of client is a user, who wants to view real time or historical data that was recorded by the devices. The server will provide methods to insert and select records over a network. The transport layer between the devices and the server should not matter, and neither should the format of the user's client (either as a web, mobile or desktop application or other installation). The server should be as simple as possible, to take advantage of the edge computing capabilities as motivated in the previous section. There should be able to be multiple devices able

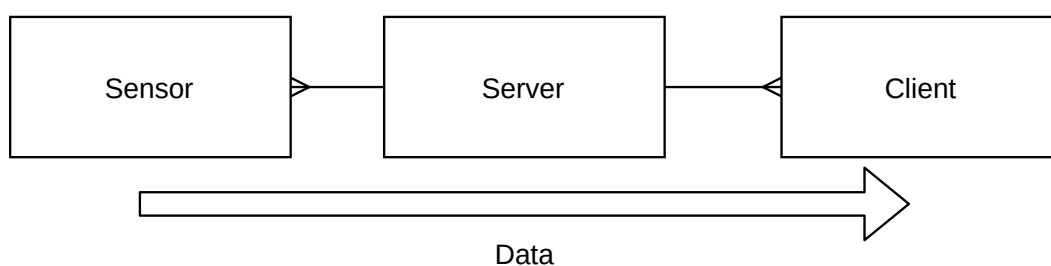


Figure 4.1: General relationship between main components

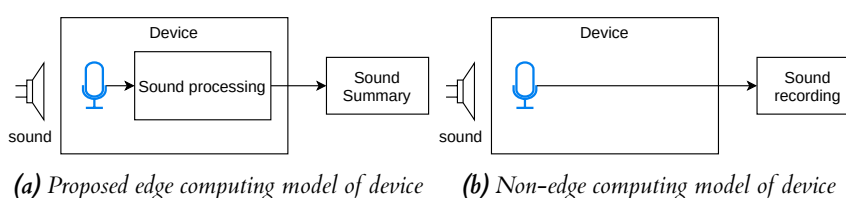


Figure 4.2: Comparison of two potential device models. This system uses 4.2a.

to be attached to the system. An IoT solution to most issues revolves around using many devices and using their combined monitoring or compute power to their advantage- so any system must be able to attach new devices. For the scope of this project, this is a secondary concern, and any provisioning matters can be considered as less important than getting an initial prototype working.

Upon analysis of the problem, there are three main components to this system: the devices to be deployed inside an office or lab space, the server to store and receive the processed data and a client for users to view the recorded data. This means that these components will need to have defined interfaces between each other.

The server has two hypothetical users: the devices and the clients. The server should authenticate the devices to only allow valid data to be uploaded, but shouldn't authenticate clients that wish to fetch data as this is public information.

The device's user are either office janitors or non-technical users. This means that the system needs to be very non-technical to use and give some visual feedback on the status of the device. Enrolling a new device can be assumed to be a bit harder to do as this could be done off-site before deployment, but it should be able to collect and send data when it's plugged into power.

The client system only needs to output data. This should be extremely simple to see the output of the entire system, and could potentially be used as signage in a lobby area to let students know if a single lab was more disruptive.

4.2 Interface design

The most important part of the interface is some form of graphical, historical output. In audio processing, this is normally done through the use of a spectrogram, which allows the visualisation of frequency and intensity over time through the use of colour (explained in background).

The interface for the device should be extremely simple- there should be some output to let a user know simple diagnostic information about it's connection status and if it's running correctly. A reset button should also be present to restart the device if it encounters any errors. Deploying a new device or an old device to a new location should be achievable by just turning it on or plugging it in.

4.3 Application protocol

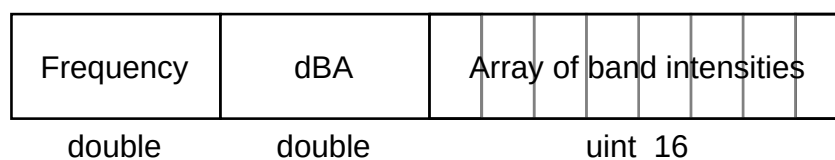


Figure 4.3: The specification of each reported time period

To minimise the amount of data transmitted, this project was designed to transmit a concrete data structure. As the structure is regular, there is no additional overhead to the transmission and since it uses the data-types natively used by the calculations on the device, reducing the amount of work required to upload the results. This is compared to an approach using a more dynamic structure where each piece of data is sent with a description or tag ie. in a JSON or XML style format. This decision increases the binding between the device and ingest server as any changes to one component must update on both at the same time or incorrect data would be decoded upon ingest. It also makes debugging transmission errors harder as the packet is not in a human-readable format (such as ASCII text).

The part from the server to the web client should use standard web technologies and protocols to reduce the amount of bespoke code used. This improves the quality of the software as libraries - especially common, well-maintained ones - are more likely to not contain defects due to an increased number of developers testing the code. Therefore this should communicate JSON over HTTP, as JavaScript handles JSON extremely well. Since the client will update regularly, there should be a simple method to fetch the latest entry and a method to fetch a larger set of previous records to populate the application at startup. The specified structure for the two JSON responses is in Figure 4.4.

```
{  
  timestamp: DateTime,  
  spectra: number[8],  
  majorPeak: number,  
  dba: number,  
  device: String,  
  documentId: String  
}
```

(a) A sound record

```
[  
  record,  
  record,  
  record...  
]
```

(b) A batch response for multiple records

Figure 4.4: The web service response that describe the sound history

5 | Implementation

5.1 High level implementation

I implemented the design in the previous chapter using common IoT technologies such as MQTT¹ and cloud tools like serverless functions. MQTT is a standard application layer protocol that relies on a long-lived TCP/IP socket to deliver binary-encoded messages between a “client” and a “server” (or broker). It is typically used in IoT scenarios as it defines a common authentication method, is low power (due to the long-lived nature of the socket and being a binary protocol, very few bytes need to be sent), and can be wrapped with TLS to provide transport security. I used an ESP32 programmed with the Arduino framework to process the data, before being decoded and stored in Google Cloud and viewed in a Vue.js-based web app hosted on Firebase².

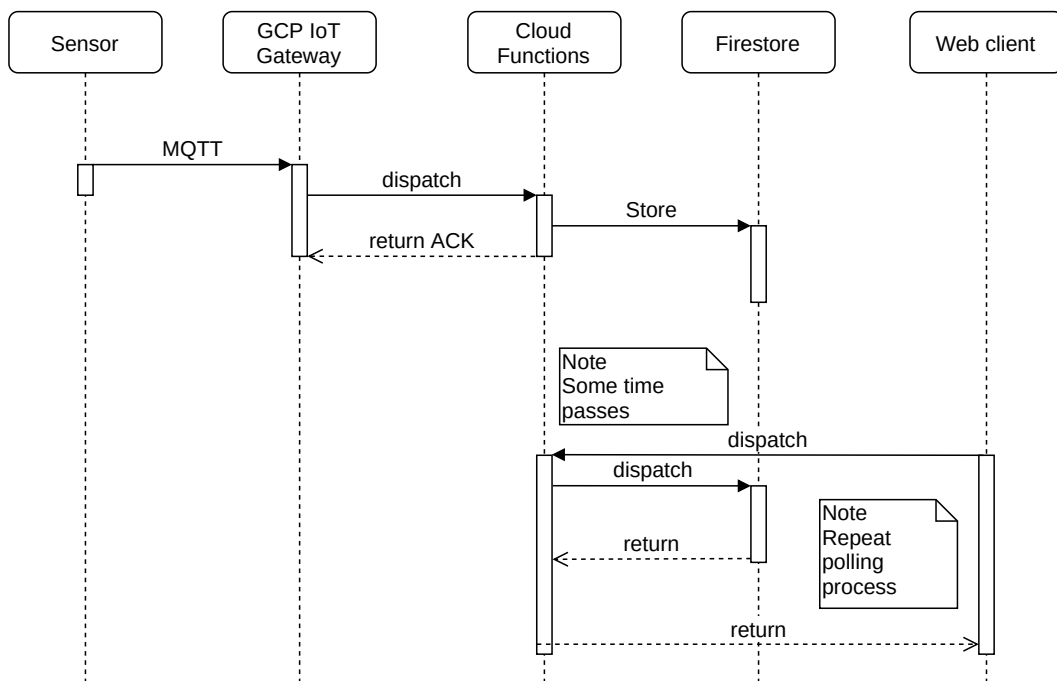


Figure 5.1: The network activity diagram of the system

5.2 Microcontroller selection

The microcontroller ecosystem for the Internet of Things has, until recently, been fairly dominated by Espressif’s ESP32 and its predecessor, the ESP8266. These are low-power WiFi-enabled

¹Specifically MQTT v3.1.1, detailed in <https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>

²The dashboard is accessible at <https://individual-project-265621.web.app>

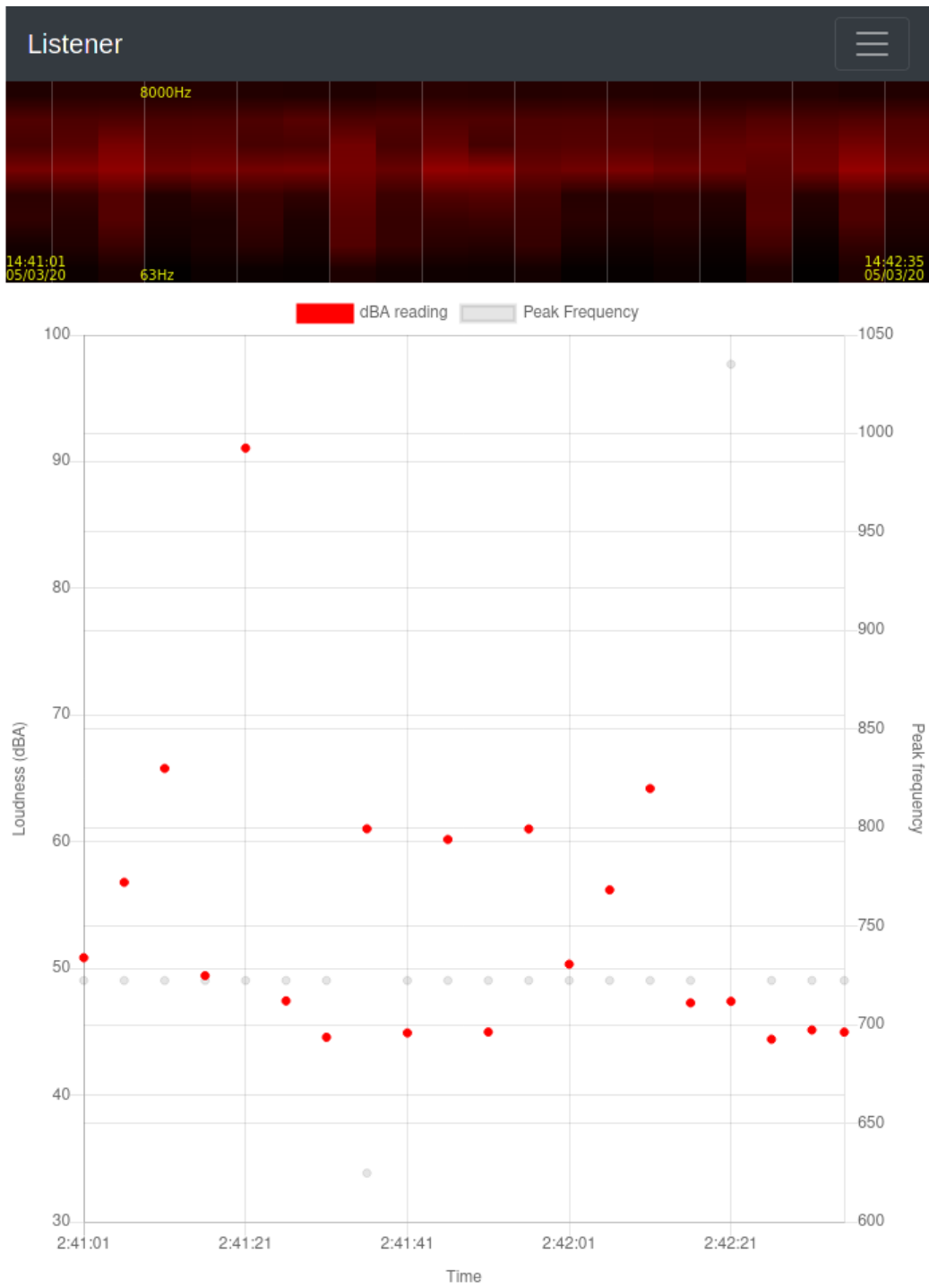


Figure 5.2: The final spectrogram page view. This updates every 5 seconds.

chips or modules that allow for rapid prototyping and modernising of existing electronic systems. There is a large hobbyist environment surrounding these products, with many development boards, each with different included peripherals (such as displays, USB/Serial adapters, voltage regulators and such). There are also a large variety of software frameworks that are compatible, from the popular Arduino to a Python implementation called MicroPython.

	Heltec LoRa WiFi 32	Adafruit Huzzah32	Aokin ESP32 OLED
Price	\$24.00	\$19.95	\$11.65
Power	microUSB, SH1.25-2 Lithium battery	microUSB, JST PH2 LiPoly battery	USB
Display	0.96-inch 128*64 dot matrix OLED	None	0.96-inch 128*64 dot matrix OLED
Additional features	LoRa radio	Featherwing compatible	None

Table 5.1: Summary of ESP32 development boards

Another possible platform would be Nordic Semiconductor's nRF 52 series of chips. These devices (normally as the nRF52832 SoC) are also very popular, but tend to be used more in professional applications, such as smart conference badges and fitness bands. There is a reasonably large development community but a slight cost premium as the development boards tend to be from western suppliers. One example of these is the Particle Argon board, which uses the nRF52840 and uses an ESP32 as a Wi-Fi co-processor. This is possibly due to the significantly better Wi-Fi support on the ESP32 platform, and increases the power consumption of the total solution.

A final potential chip was released after the majority of the work for this project was completed, but is significant so bears a mention here. Microsoft, in partnership with MediaTek, released the MT3620 SoC which is significantly different to the previous two offerings as it is designed specifically for use with Azure Sphere OS and Microsoft's IoT Edge service. Due to both the tight coupling to Microsoft services and recent release, there are not many existing solutions but there are plenty of examples supplied and allows for a more practical long-term deployment as firmware is automatically kept up-to-date and is pushed from the cloud.

A Raspberry Pi-style single board computer was not considered as these machines would be more likely to be vulnerable to potential exploitation. Running an entire Linux software stack with multithreading enabled would require more routine maintenance than a more low-level approach, and to get the required protocol (either I2C or I2S, depending on decisions elaborated on later) requires compiling a kernel module. There is an increased risk from being exploited and more probability of developing a fault (since there is many orders of magnitude more components running). Raspbian (the distribution of Linux most commonly used on a Raspberry Pi) is not designed for IoT applications and the most IoT-focused OS (Windows 10 IoT Core) doesn't support I2S, so would restrict the potential microphone choice.

5.3 Peripheral selection

I chose an analog microphone breakout board for the microphone input portion of this project. This was done using Adafruit's Max 4466 breakout to reduce the required work to implement, instead of creating my own circuit to collect and amplify the circuit. This board was chosen as it was available and provided a rail-to-rail signal - this means that the amplified signal will only "clip" when the signal would exceed the limits set by the power supply. I also had access to a

	ESP32	nRF 52	MT3620
WiFi	802.11 b/g/n	Proprietary or with coprocessor	802.11 a/b/g/n
Cores	2 LX6 cores @ 240 MHz	ARM Cortex M4 @ 64 MHz	ARM Cortex A7, 2 Cortex M4 cores
RAM	520KB	256KB	4MB
ADC	12 bit, 200ksps	12 bit, 200ksps	12 bit, 2Msps
Security	AES acceleration	ECC support, ARM TrustZone subsystem	ECDSA support, Pluton security subsystem
Frameworks	ESP-IDF, Arduino, MicroPython	Arduino, Mbed OS, CircuitPython	Azure Sphere OS

Table 5.2: Summary of Microcontroller platforms

Seed sound sensor ³ but this board only returned a 200mA peak-to-peak signal when I tested it, making it unable to accurately record loud noises. This style uses an electret microphone, which will degrade slightly over time as the internal film loses charge, but they are very expensive and tend to last about 5-10 years, so isn't a concern in this case.

Alternatively, an I2S microphone could have been used. This performs all sensing and quantisation on a single chip, and buffers the recorded data to be collected by the microcontroller as a digital signal using the I2S (Inter-IC Sound) protocol (no relation to the I2C protocol). This is preferable as it introduces much less noise into the signal between the microphone and ADC, but increases the cost of the sensing module and was not available at the time of prototyping the system. It also makes debugging harder, as an analog reading allows an external device such as an oscilloscope to 'tap' the microphone and perform the sampling and calculation with a known-good implementation.

Due to the requirement to also monitor dBA weighted levels, a calibrated noise level meter peripheral was also required. This has to be separate from the main microphone as it would have to be calibrated due to the variance in manufacturing of amplifier and microphone components, and equipment to calibrate the microphone would be more costly than other methods. I used the DFRobot dBA sound level meter board ⁴ as it is the simplest to use with a microcontroller. This is because it uses a single signal wire which holds an analog voltage that is directly proportional to the resulting decibel value.

There are other larger and standalone dBA meters, and some come with the ability to output the values they record over serial (either RS-232 or serial over USB). These are similar in functionality (they normally have a display to show the current level, and a condenser microphone capsule) but are significantly more expensive (the cheapest one on rs-online.com with an external interface is £110, and may require a level shifter to allow communication between RS-232 and UART) and would also require a 9V battery due to the use of a condenser microphone. This would decrease the theoretical lifetime the system could be deployed without intervention.

This system was built using a requirement of at least one Analogue to Digital Converter (ADC) with a 10-bit resolution. This was due to the selection of both an analog sound level meter and analog microphone.

³Available from their online store, with schematics <https://www.seeedstudio.com/Grove-Sound-Sensor-Based-on-LM386-amplifier-Arduino-Compatible.html>

⁴https://wiki.dfrobot.com/Gravity__Analog_Sound_Level_Meter_SKU_SEN0232

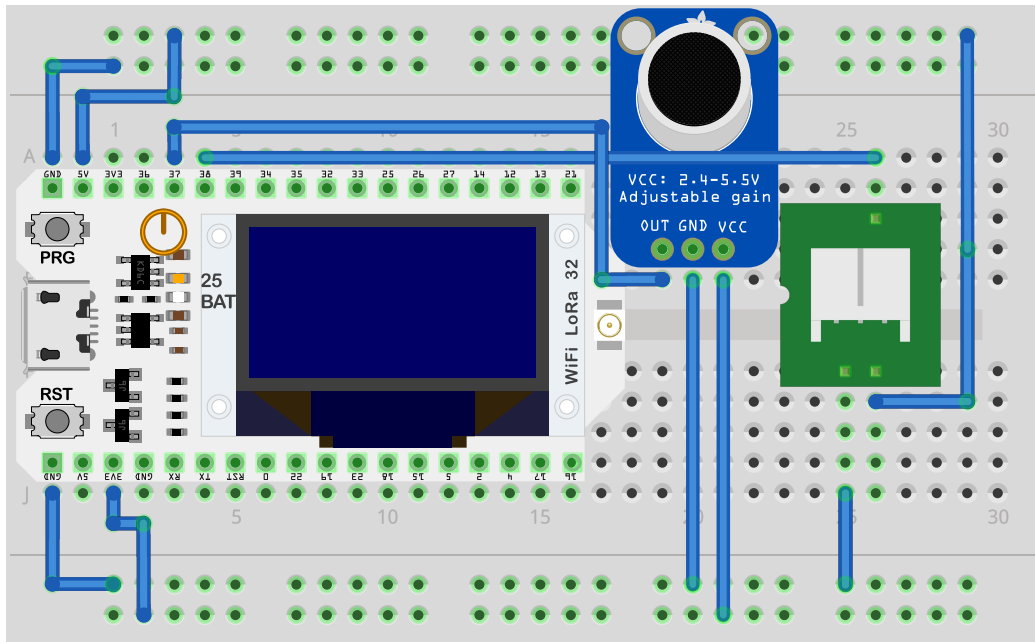


Figure 5.3: The Fritzing breadboard layout of the final circuit

5.3.1 Bill of Materials

The final Bill of Materials required to build a single module (excluding wire and a breadboard)

Component	Part Number	Cost	Count	Source
Heltec Lora 32	WiFi Lora 32 V2	£13.61	1	Aliexpress ¹
Adafruit MAX 4466	485-1063	£6.49	1	Mouser
DFRobot SENS0232	426-SENS0232	£30.57	1	Mouser
Total		£50.67		

¹: <https://www.aliexpress.com/item/32899346000.html>

5.3.2 Programming

As mentioned in the previous section, the ESP32 supports many different runtimes, such as FreeRTOS, Arduino, and ESP-IDF. I chose to use the Arduino framework as it has the widest base of support from additional libraries (including cloud service support). This allows for experimentation with LoRa using the SX1276 chip on the Heltec board, as well as simple display output and MQTT support.

I also included the Platform.io⁵(PIO) set of tools to manage the project. This set of tools allows for management of libraries, and encourages a more modular code base with the LDF (Library Dependency Finder). It employs an opinionated program structure with isolated folders for separate libraries, which encourages looser coupling between modules. PIO is also nicer to use from the perspective of a programmer as it provides access to C++ standard libraries, so plenty of common data structures such as queues and stacks are already present. As it's a separate application, there are plugins for other common IDEs (such as VS Code and JetBrains CLion) which is much nicer to use than the standard Arduino IDE.

⁵<https://platformio.org/>

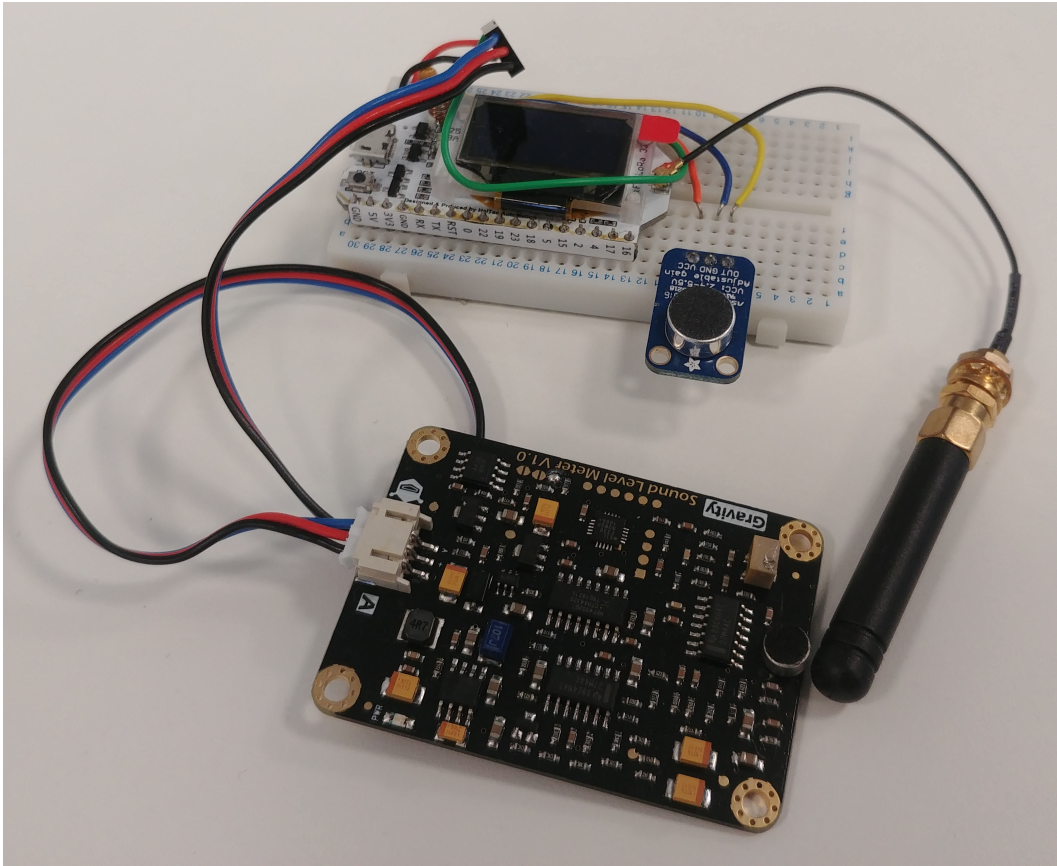


Figure 5.4: The final prototype device, with breakout boards attached

5.3.3 On-device computation

Since all computation on the signal must happen on the device, this must include any noise-removal techniques, as well as the FFT. While it is somewhat possible to minimise the effect of electrical noise through analog means (such as using capacitors and resistors to create filters), the most effective ways are digitally processing the input signal.

While experimenting with the FFT library, I discovered that there was a high amount of variance in the values over time, even in a quiet room. This indicated that there the effect of noise on the microphone was influencing the output result, which is not favourable. I therefore decided to average over the FFT values with a rolling average algorithm. This was not as easy as it potentially could have been on another device, as the device is very constrained by its RAM—there is only 390KiB available for program code and if the `fft_averages` array was composed of floats (to potentially increase accuracy from multiple samples) then the code would not run as it would use too much memory. The code that handles this can be found in Figure A.1.

5.4 Service Providers

Since this project is designed to use cloud services as optimally as possible, I assessed each provider as a single package.

Criterion:

```
[env:heltec_wifi_lora_32]
platform = espressif32
board = heltec_wifi_lora_32
build_flags =
  -D CLOUD_IOT_CORE_MQTT_PORT=443
framework = arduino
lib_deps =
  ArduinoFFT
  U8g2
  5372@^1.1.10
  617@^2.4.7
```

Figure 5.5: The project configuration file

- An Internet of Things applicable gateway (MQTT or AMQP preferred)
- A serverless function environment
- A database (NoSQL or SQL)
- (Optionally) a CDN-based hosting method

I then compared the offerings from Amazon’s AWS (Amazon Web Services), Microsoft’s Azure platform and Google’s GCP (Google Cloud Platform) environment. The largest degree of differentiation from a technical perspective is the methods that each platform implements their IoT gateways (Table 5.3). Authentication for AWS and Azure’s gateway use X.509 certificates, while GCP uses a time-based password in the form of a valid JWT (JSON Web Token). This also relies on a public-private keypair, but due to its very short expiry is more secure.

All providers only support a subset of the MQTT specification, with QoS level 2 being disabled. QoS 2 in the MQTT spec means that each message to a device will be sent once and only once. This feature does not matter in this current implementation as no messages are sent to the devices. They also all specify only certain topics as publishable by specific devices. This creates a tighter binding to each specific provider.

	Microsoft	Amazon	Google
Port	8883 or 443 over WebSockets	8883 or 443 with ALPN	8883 or 443
Supplied Library?	Non-functional	Yes	Yes
Other protocols	AMQP, HTTP	HTTP	HTTP
Price per device month	£7.454 ¹	£0.50	£0.98
Free tier limit (messages)	8000	500000	250000

¹ Azure itemises per “device unit”, so this is actually 400,000 messages per day. Each device sends approximately 17,280 messages per day, so this bulk rate is much higher. With 23 devices, this option is £0.32 per device.

Table 5.3: Comparison of cloud-based IoT gateways

With regards to pricing, each provider is very different, although it is clear to see that Azure is the most expensive for a small sample installation like this prototype. GCP’s price in the table above is deceptively low as it counts the price for data ingress but not ingress to a specific resource.

There is also cost associated with transforming the received data packet and inserting it into a database (with AWS this is largely included with the cost of sending the data). For the pricing in table 5.3, I assume a message size of 32 bytes (2×8 for the unsigned integers and 8×2 for the doubles with frequency magnitudes).

Note: this pricing estimate does not include any costs for storage and processing. This isn't much of a particularly large problem for a small prototype, since storage free tier limits are typically 1GB per month, which is significantly larger than the amount that this project will use. All providers also give free credit upon signing up (£300 for GCP, £150 for Azure) so funding this project was not an immediate concern. It was, however, kept in consideration as one of the previously stated benefits of edge computing is the reduction in cost.

I chose to use MQTT in spite of not requiring cloud to device messaging as it would allow for expansion of the system to include modifying configuration parameters (such as frequency bands). MQTT is also designed for these low-power applications (as a binary based protocol on a long-lived TCP socket), while HTTP has a greater overhead.

I selected GCP as the final host as the presence of a working Arduino library made the system much easier and reduced the amount of effort that would be duplicated. This meant that a cloud-based architecture needed to be developed using their specific services that would suit the chosen platform. An alternative to this would be a more monolithic application running on a VM or inside an application hosting system such as App Engine. This alternative is inferior to a cloud-native approach as not only does it cost more to run, but also requires more maintenance regarding scaling if the required resources for the system grow or shrink.

Data upon ingress to the system needs to be transformed from the packed, binary format to a database. This is done by a Cloud Function, Google's name for a serverless application runner. Due to Google's internal infrastructure, the first request takes a few seconds, but subsequent requests complete in roughly 60-140ms. This is more than responsive enough for this system.

The choice to use GCP led to the selection of a NoSQL-based database. Since each record is very small and there is no relationship between each record, Firestore has the highest free-tier limits of the available hosted database solutions from Google in accesses, writes, and storage size.

The final service architecture diagram for GCP (Figure 5.6) details the services and the flow of recorded information.

5.5 Web Stack design

There are two main ways to render a page with dynamic data: Server Side Rendering (SSR) and Client Side Rendering (CSR). SSR requires a server to evaluate the result from any required API queries (at least, initially) before sending the page as HTML and CSS, with a much lighter JavaScript load. CSR sends a template to every web client and requires that client to populate the page itself. SSR has many benefits, as the time to load a page fully is significantly faster (since the data isn't waiting for the template to be loaded before it's fetched). This also allows pages to be crawled much easier by search engines, as they aren't required to parse and evaluate any of the JavaScript that is vital to loading a CSR application. One main disadvantage to using a SSR is that a server is required to generate each page before it's loaded, making the server much more complex. A CSR application on the other hand can be returned from a CDN as it only returns static HTML, CSS and JavaScript.

JAMStack⁶ (JavaScript, API and Markup stack) is an approach that follows a CSR (or prerendered) behaviour, and is the approach that I chose to use for this project. This is due to the significantly reduced administration requirements (since all infrastructure is managed by a cloud provider,

⁶<https://jamstack.org/>

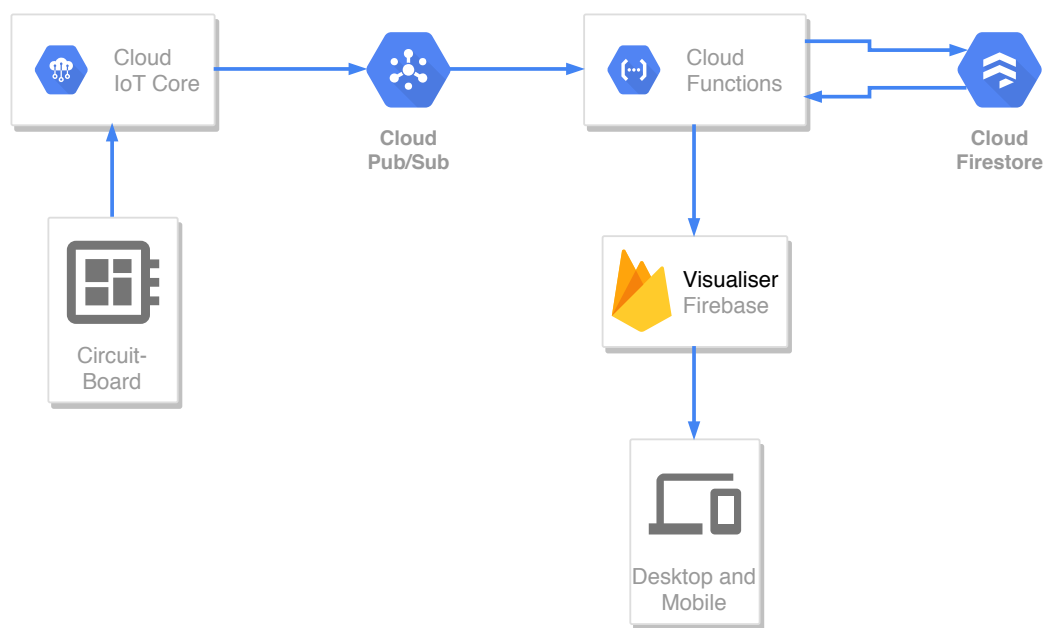


Figure 5.6: Final GCP architecture

Google in this case) and because of the potential to scale the application without any interference. JAMStack also provides much more simplicity to developing a web application, since there is no “web-server” but rather a CDN service that fulfils the web requests for static content, and serverless functions that add the live or dynamic content to the page.

So called “serverless” functions are cloud-based scripts or binaries that are executed in an isolated environment when a trigger event occurs. These triggers are commonly HTTP requests, but can also be events raised from other cloud services, such as database events or Pub/Sub messages. Google’s Cloud Functions⁷ specify either Python 3.7, Go 1.11 or 1.13, or Node.js 6, 8, or 10 as available runtime environments, and I chose to work with Node.js 10 as it reduces the number of languages required for this project while being most up-to-date version of Node available (despite being released in 2018, Google still lists this environment as a ‘beta’). The environment is created when the trigger is called, and then kept available for a short period of time to make the service more responsive. Functions are billed based on invocation count, network traffic (to off-cloud resources), RAM usage and CPU usage, timed to the nearest 100ms. This obviously makes a clear requirement for the function to be evaluated quickly.

There are two possible methods to update the client regarding additions to the database- either an active websocket which would PUBLISH new information as soon as the server is aware or the client can poll a HTTP endpoint for updated changes. I chose to use the latter method as information is regularly published by the device and would require much less runtime on the function, lowering the cost of running the service. The polling interval is 5 seconds, which is the same as the minimum interval programmed into the device.

5.5.1 Web Application Framework

For this project, I chose to use Vue.js for my JavaScript framework. A JavaScript framework allows for pages to be written in a simpler fashion than writing raw HTML and JavaScript, and allows for more consistent code due to integration with a wider ecosystem of tools such as

⁷<https://cloud.google.com/functions/>


```

window.setInterval(async () => {
  await this.fetchUpdate();
}, 5000);

```

Figure 5.7: Code snippet from the web-app’s main page, executed on load

cross-compilers (to enable wider browser support for modern language features) and linters (for consistent code style rules).

I preferred Vue over React or Angular partially due to familiarity with the framework.

The debate over which framework is “better” is endless, with plenty of reasonable arguments on both sides. React’s surrounding ecosystem is undeniably larger, with 53,806⁸ dependant packages on NPM (the most common package manager for JavaScript) compared to vue’s 24,984 at time of writing⁹. While this isn’t a particularly accurate measure, it gives a rough indication to the difference in popularity between the two frameworks (react also has approximately 5 times more downloads).

The largest difference between React and Vue is their approach to adding code. React creates new components using the class syntax in JavaScript, while Vue components are typically written as single file components. These single vue files contain the relevant HTML, CSS, and JS for each component, which makes the development of new components very simple. It also heavily encourages loose coupling between each component, since they can only interact with each other through their defined properties or the global state store (called Vuex).

Vuex stores the entire application state. This is extremely useful, as all information that is shown to the user is kept up to date. Due to the reactive behaviour of Vue, when the state is updated by one piece of code (such as the request in figure 5.7), all of the graphs are updated immediately. This reduces the amount of API requests made by the application as flicking between pages doesn’t trigger any requests to the server - only the active polling job does. It also removes any concerns for each page - their job is solely to display the known data so they don’t care about where or how that data was obtained.

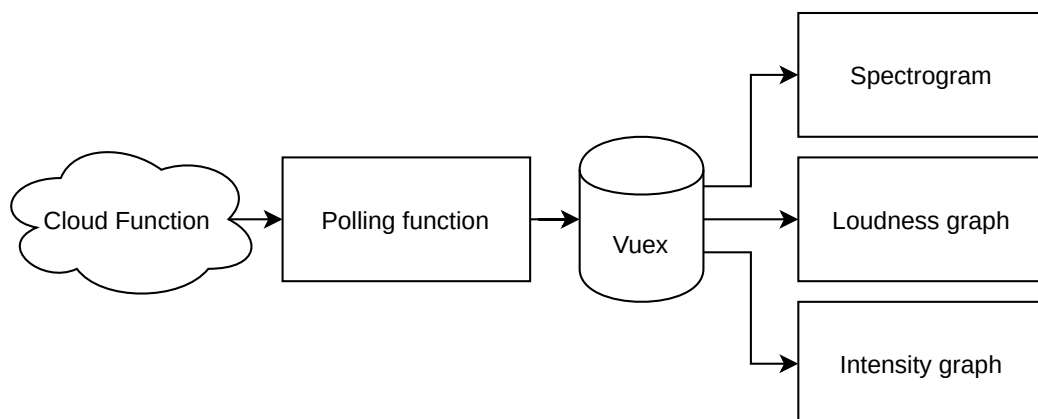


Figure 5.8: Diagram showing the flow of data through the web application

⁸<https://www.npmjs.com/package/react>

⁹<https://www.npmjs.com/package/vue>

There are plenty of vue wrappers for web-based graphing libraries, and I chose to use `chart.js`¹⁰. This is due to the existence of a vue wrapper library and compatibility with TypeScript.

```
<db-history-graph
  class="graph"
  v-if="loaded"
  :chartData="this.dbHistory"
  :options="options"
/>
```

Figure 5.9: The graph showing the dBA values and frequencies is added to the page as this component

For the web application, I chose to use TypeScript. TypeScript is a super-set of JavaScript developed by Microsoft that adds (as the name would suggest) strict typing to the language. This is evaluated at compile-time, so allows developers to detect any potential mistakes with their program before they even run the code. It also adds code-completion functionality to my IDE (Visual Studio Code), increasing development speed and decreasing the amount of typos in the code. This additional compilation step doesn't add much time to the build step as Babel (a JavaScript transpiler that allows you to write modern syntax and maintain support for older browsers) understands TypeScript as of Babel 7¹¹, along with ESLint (the most common JavaScript linter). Due to the clear advantages of strict typing and extreme amounts of developer hype (the TypeScript repository is the 39th most starred on GitHub), most frameworks and libraries support TypeScript with type definition files for the objects they contain.

Using TypeScript does, however, require a compilation step before use. This discouraged me from using for the cloud functions as the function is generated from the contents of the Google Cloud Source repository, so I'd either have to configure a longer build pipeline to be triggered by a commit hook to run the compilation or I'd have to have compiled code in a tracked source repository, which has its own issues (larger commit objects and no reliable way to prove source corresponds with output to name a couple).

Vue also had a command line interface for creating a simple new application. This allows for the creation of new projects extremely quickly, adding tooling for TypeScript, ESLint and Babel by default. This was also a motivating factor to start using Vue, as getting a simple skeleton app with the required boilerplate is extremely easy, leaving plenty of time to work on the contents of the application itself. For this project, the Airbnb rules were chosen as the linting rules—this matter is essentially just personal preference and enforces semi-colons at line endings and single quotes for strings.

5.6 Spectrogram

The spectrogram solutions that are available all take in audio (either from the microphone or a pre-recorded buffer), so were not able to solve my requirement. To solve this, I therefore created my own solution using the HTML `<canvas>` element¹². This element can be expensive to manipulate if it's visible in the DOM, and it's faster to compute the value of the buffer on an offscreen canvas element before copying the offscreen one's contents to the onscreen buffer.

¹⁰<https://www.chartjs.org/>

¹¹<https://devblogs.microsoft.com/typescript/typescript-and-babel-7/>

¹²As always the best reference manual here is MDN, due to its completeness and listed examples: <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/canvas>

Therefore my solution draws each column to an 1x8 image buffer, then scales the drawing mode of the onscreen canvas before drawing each column out. This is demonstrated in figure 5.10, where the context variable is the canvas' 2D rendering context.

```
context.save(); // save default drawing context
context.scale(canvas.width / plottable.length, canvas.height / 8);
let counter = 0;
plottable.forEach((point: record) => {
  const colors = genColumn(point.spectra);
  context.drawImage(prerenderColumn(colors), counter, 0);
  counter += 1;
});
context.restore(); // restore default drawing context
```

Figure 5.10: TypeScript that draws the records out to the visible canvas

5.7 Device transmission technology

I eventually chose to use a standard 2.4GHz WiFi connection to transmit the data from the device to the cloud. This was due to coverage across campus being more than adequate for the connection and very reliable. One downside of WiFi is that setting it up as a product requires more effort, since the SSID and authentication information (a PSK in a WPA2-PSK network or identity and password for a WPA2-Enterprise network such as eduroam) need to be loaded and stored by the device. In a final product, this could be loaded using a “configuration” mode, where the device hosts a WiFi network and a simple HTTP server allowing for collection of the network information to be stored on the local flash storage, but this was not implemented and as such is only a prototype. In the current system, the network and credential information are placed into a header file as constants and are fixed at compile time. This solution was able to use the eduroam network through the use of Espressif’s extensions to the Arduino platform.

```
WiFi.mode(WIFI_STA);
#ifdef EAP_IDENTITY
//WPA2-Enterprise (eduroam)
esp_wifi_sta_wpa2_ent_set_username((uint8_t *)EAP_IDENTITY, strlen(EAP_IDENTITY));
esp_wifi_sta_wpa2_ent_set_password((uint8_t *)EAP_PASSWORD, strlen(EAP_PASSWORD));
esp_wpa2_config_t config = WPA2_CONFIG_INIT_DEFAULT(); //set config settings to
  default
esp_wifi_sta_wpa2_ent_enable(&config); //set config settings to enable function
WiFi.begin(WIFI_SSID); //connect to wifi
#else
// WPA2-PSK
WiFi.begin(WIFI_SSID, WIFI_PSK);
#endif
```

Figure 5.11: expert from the transmission sub-component to connect to WiFi

One possible alternative would be LoRaWAN. It uses the SRD860 (Short Range Device) radio band, and has a longer point-to-point range due to the lower frequency (863-870kHz). This

lower frequency also reduces the available bandwidth per channel, which is not a problem for this project due to the low data transmission requirement. LoRa's physical layer then hosts the LoRaWAN MAC protocol, defining the network protocol between "gateways" (devices connected to LoRaWAN and the internet) and devices. Gateways then send notifications to (typically) The Things Network, which allows device key management and provides the ability to trigger a webhook upon receiving new data. This wouldn't require extensive setup as network keys are negotiated upon joining, and only an application ID (which wouldn't need to change) is required.

LoRaWAN does have some issues however (Bankov et al. (2016)), and during the testing phase it proved difficult to get acknowledgements from the local gateway on the roof of the Boyd Orr building in a reliable fashion. This failure meant that my device couldn't join the network in a reliable way. In the process of debugging this, I captured the transmission from the radio using a Software Defined Radio (Figure 5.12), and verified that the message was being sent, while nothing appeared on the dashboard for The Things Network (a very popular LoRaWAN provider). This is rather unfortunate as LoRa is a much more power efficient transmission layer (compared to WiFi) and has less chance of interfering with local working devices on a WiFi network.

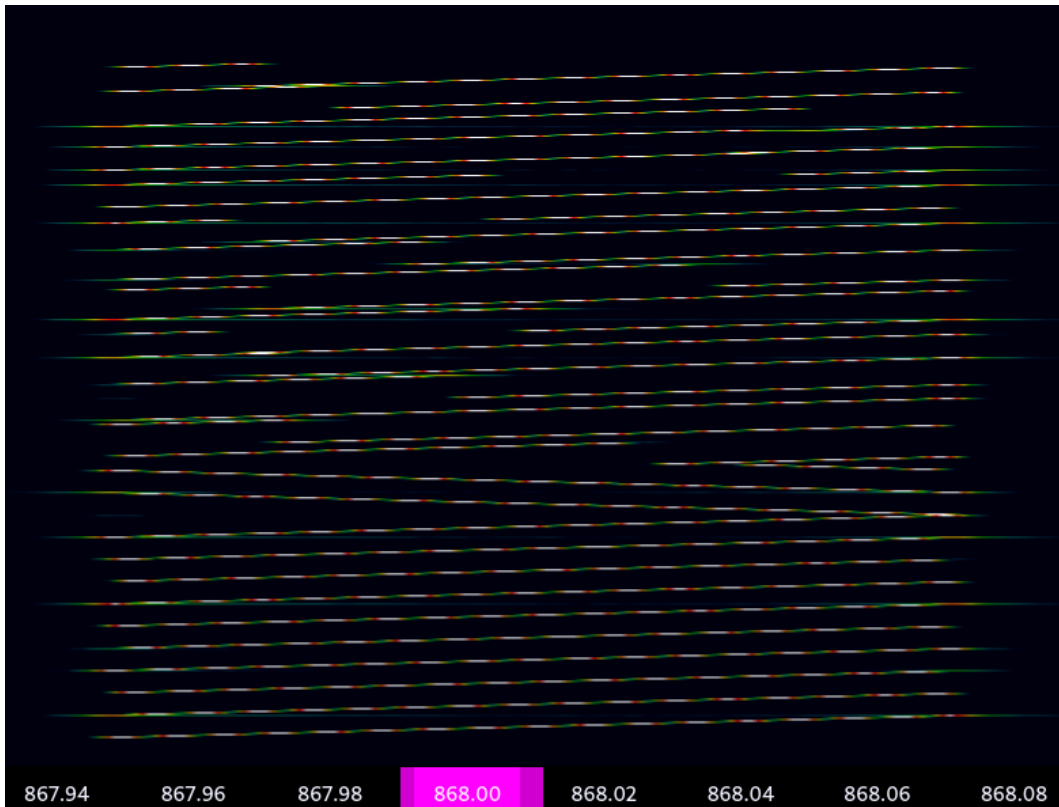


Figure 5.12: A transmitted LoRaWAN packet captured using SDRangelove

Another possible transmission method would be a 4G LTE modem. This would make setup simpler as the only configuration would be inserting an IoT SIM card. This option was not explored due to the related cost of using a data-only SIM plan and requiring an additional modem module- increasing the BoM.

5.8 Data packing and unpacking

Due to the dense and concrete nature of the MQTT payload, there is some intricacy in defining the data structure in both languages. In node.js, this required the use of the Buffer class, which allows for byte-level interpretation of each packed number. In C, this is manageable by performing pointer arithmetic and copying the memory of the target variables into the allocated memory structure.

```
const decodedData = Buffer.from(event.data, 'base64');
const majorPeak = decodedData.readDoubleLE();
const dba = decodedData.readDoubleLE(8);
const fftPeaks = [];
let i = 8*2;
while(decodedData.length>i){
  fftPeaks.push(decodedData.readUInt16LE(i));
  i+=2; // jump to the next int
}
```

(a) Decoding the received message in node.js

```
void* sendBuffer = malloc(sizeof(double)*2+sizeof(uint16_t)*8);
memcpy(sendBuffer, &p.majorPeak.frequency,
       sizeof(p.majorPeak.frequency));
memcpy(sendBuffer+sizeof(double), &dba, sizeof(dba));
const int offset = (int)sendBuffer+sizeof(double)*2;
for(int i=0;i<8;i++) {
  memcpy((void*)offset+i*sizeof(uint16_t), &p.frequencies[i],
        sizeof(uint16_t));
  p.frequencies[i] = 0;
}
```

(b) Packing the MQTT payload in Arduino C

Figure 5.13

5.9 Development practices

As can probably be observed in the previous sections, the development process for this solution used many best-practice development patterns. This includes creating components and sub-components with a loose coupling, using libraries where possible to reduce the duplication of effort, tracking source code inside an SCM (3 separate git repos). A notable absence is any form of unit tests, but while unit-testing Arduino code is *possible* with platform.io, it's listed as a premium feature and the body of code is quite hard to imagine many sections that could be tested as working in an automated way that would be easier than integration tests of the entire project. Potentially the code for applying the weighting could be tested, but everything else is connecting between external libraries and as such is less valuable to test: it would practically be just testing the library works as expected.

5.10 Summary

There are many different ways to implement the system as described in the previous chapter. This solution used a Heltec LoRa V1 ESP32 development board and Google Cloud with the motivation to keep initial and running costs low. This same motivation, as well as a desire to keep maintenance simple, was behind the choice of a JAMStack web client and the use of serverless functions to provide an API to retrieve the recorded data.

6 | Evaluation

6.1 Device testing

Testing was performed by leaving the device in a computing lab environment currently affected by renovation work. The resulting graph when an interrupting noise was measured showed both a change in major peak and a change in the dBA measured. This correlation between an observable stimulus (eg. drilling noise) and the decibel level rising on the device leads me to believe that this is at least returning data that is likely true. Further testing was planned to verify that the result was similar to that on a phone performing an FFT, but this testing was unable to be completed due to the university shutdown. As such, only artificial tests on this nature can be performed.

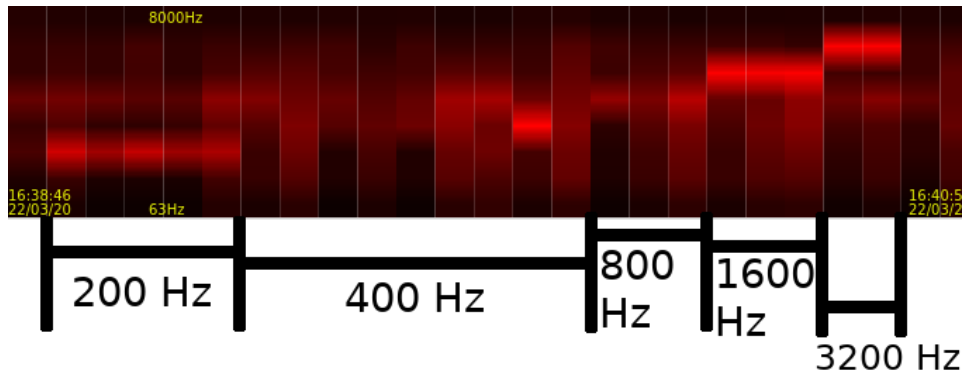
Artificial testing was also performed by playing specific sine tones at a loud volume to verify the correct frequencies were being calculated. This presented itself accurately on the web app within a few seconds, so can be assumed as working as intended. The recorded frequencies and the error associated with these mean readings are shown in table 6.1 and show that the recorded frequency is typically within 2.5% of the correct value, although some accuracy is lost at the lower frequencies. This is possibly due to the size of the bins being constant 19.53 Hz wide (and therefore amplifying potential error from using a bin that may not be perfectly aligned).

In the test output shown in figure 6.1, frequencies of 200, 400, 800, 1600, and 3200Hz were played through a PC speaker approximately 50cm from the device and monitored the output on the spectrogram page. This showed a strong ability to pick up most frequencies, but the 400 Hz input proved challenging for it to correctly accept, alternating between 400 Hz and 771 Hz (which is the most common frequency in situations with no sound or white noise). The higher frequencies show up very prominently due to being amplified by the dBA weighting function (but also possibly due to the frequency response of the speakers used to test, since this is unknown).

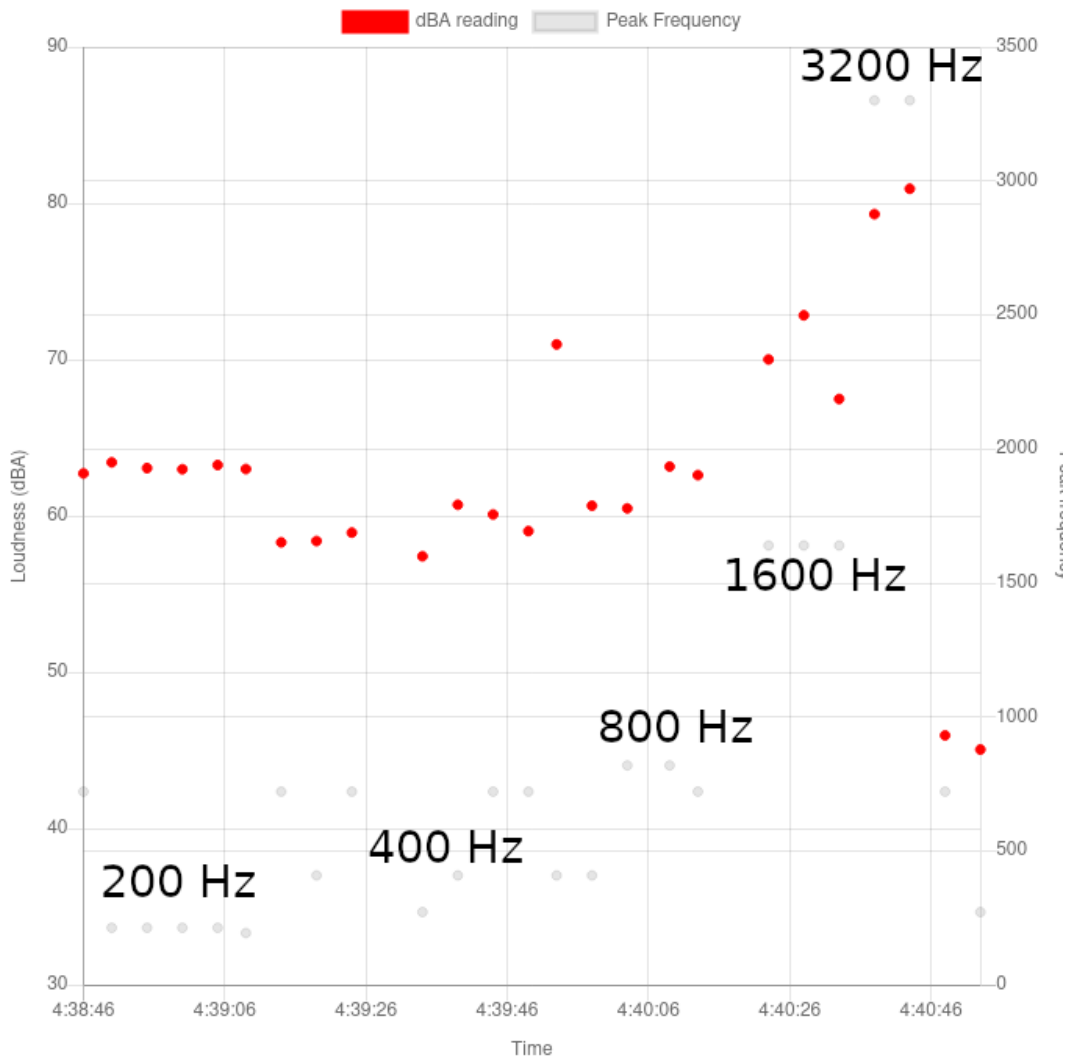
I also evaluated this project in a computer science lab space in the Boyd Orr building. This was done very simply by plugging the device into a micro-USB phone charger and leaving it there for a few days. During this test, it was connected to the aforementioned restrictive eduroam network using my personal credentials. One issue that was noticed a short while into the test was that the device would only report data for the first fifteen minutes after being turned on. This could potentially be for a number of reasons, either the TCP socket gets terminated by

test frequency (Hz)	recorded frequency (Hz)	percentage error
200	214.84375	7.4%
400	410.15625	2.5%
800	820.3125	2.5%
1600	1640.625	2.5%
3200	3300.78125	3.1%

Table 6.1: Test frequency vs the recorded 'peak value'



(a) The spectrogram output



(b) dBA and most common frequency graph

Figure 6.1: Records from a sweeping set on input tones

the network or the JWT expires and it fails to reconnect the the MQTT broker. This issue was not faced in other tests on my home network, which is a normal WPA2-PSK network so clearly would require more debugging (irritatingly with a 15 minute delay between each test). Alternatively, a watchdog timer could be set to reset the device every 10 minutes, avoiding this issue while losing a small window of data.

The results of this test demonstrated 4 records where the dBA value inside the building exceeded 90 dBA, forming an obstacle to productive work. It was also very obvious when construction work was recorded, showing a general broad-band spectrum of noise compared to the noise floor or general discussion (which was generally noticeable as a 100-200 Hz peak frequency). This shows that the data is recorded and processed in a way that is easy to understand for someone who has viewed these diagrams at the same time as the recorded noises before. One thing to note is that it would possibly be too abstract for a new user to be able to deduce a meaningful conclusion from the graphs, so could do with either some explaining or code to classify the recorded noise events.

The interface on the device is very simple, and successfully fulfils the simplicity requirement for the output. It attempts to connect to the configured WiFi network and upon success shows a confirmation and prints 'initialised!' when it's finished. The display output scrolls to always have the most recent lines on the screen like a terminal output, showing the context of the event. It ordinarily would show 5 lines of text, but the device's screen was partially cracked during development so only 3 lines are partially visible.

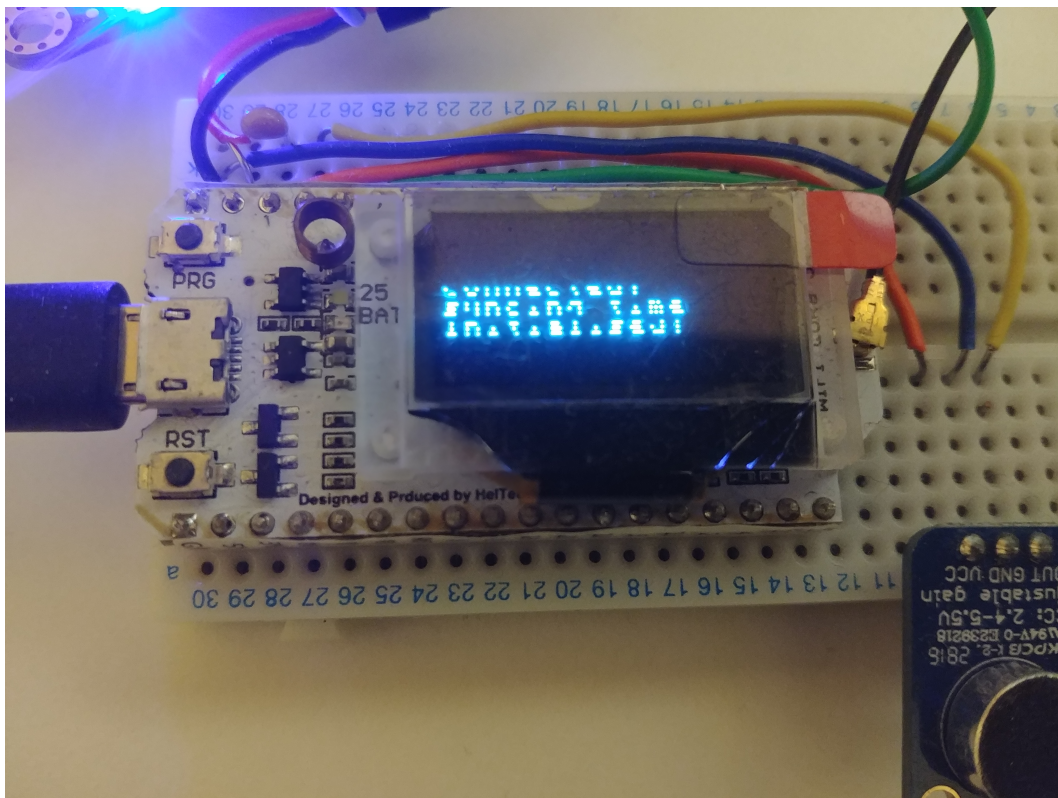


Figure 6.2: The display on the device. Unfortunately the screen was cracked at some point, but it reads “connected!
syncing time
initialised!”.

The screen breakage and long wires make the system very fragile physically. This could be

improved through design of a circuit board and an enclosure to make the solution more resilient. In a few tests, the wires were clearly loose and returned nonsensical data. It also had to be kept very still during tests to not disturb its more delicate construction.

The current usage through the USB input was between 0.02 and 0.04 amps. At approximately 5V, 0.03A would equate to 0.15W, and if used for an 8 hour work day would require a 14400mAh battery. This is significantly below the largest capacity of commercially available USB power banks. This therefore opens the possible applications up past just monitoring offices but to more mobile applications or placement away from wall sockets. It is also possible, though untested, that this could be maintained through a renewable source such as a solar panel, making the deployed running cost non-existent.

6.2 Cost analysis

The resultant system is very low cost to operate- the monthly bills for the project while in the testing phase were never higher than \$0.02, although this was only active for a fraction of a month (given the previously stated issue with eduroam causing disconnects after 15 minutes). March's usage data is listed below, with free limits included. This is only for 2628 records, which only equates to 3.65 hours of constant usage and extrapolating out (for a single device, at least) stays within the free tier and I suspect that the single cent is a rounding quantity. This fulfils the requirement to be cost effective in terms of the cloud usage.

With a single device costing approximately \$50, this is fairly cost effective, but I suspect that it would be possible to get a cheaper ESP32 development board with a screen and without the LoRa radio that remains unused in this implementation. If costs needed to be reduced further, it would be possible to replace it with a simple multi-coloured LED, as the ESP has plenty of available digital outputs.

7 | Conclusion

7.1 Summary

I have built a single device prototype to monitor the perceived volume and pitch information of an office environment, and built a cloud-based infrastructure to handle the data generated. This infrastructure is very simple to maintain, as the cloud portion is all run on managed services and the devices are very basic to leave running.

Cloud functions are very applicable for this style of architecture, and have kept the monetary cost of the project very low. This, combined with the use of static site hosting, makes the whole system very efficient and only consumes resources when it has to.

And ESP32 device client for the application was developed and performs fairly reliably, reporting every 5 seconds. It draws minimal amounts of power and is able to perform analysis of the surrounding noise within a 2-10% accuracy.

The web-app outputs visualisations of both the previous

7.2 Reflection

Upon reflection, this project spent a long period of time focused on communicating over LoRa, when it wasn't a necessary part of the specification. Due to the modular nature of the project, the shift from LoRa to MQTT over WiFi did not change the architecture massively- it only changed the communication between the device (which stayed the same) and the database (which would have still required a cloud function to provide The Things Network with a webhook to call). The ability to visualise accurately the type of noise and volume in a lab was achieved using low-cost sensors and cloud technologies. On repeating this project I would also explore other microcontroller options that are more specialised towards IoT edge applications, such as the Azure Sphere device.

7.3 Future work

This work has opened up many new possibilities for research into solutions for monitoring noise. One potential avenue to take would be using an intermediary gateway to attempt to triangulate or categorise types of noise. It would also be interesting to see if a LoRaWAN solution could be produced, or if a tunable system could be build only using a single microphone for both applications with a PCB. It may also be interesting to explore if wider conclusions could be inferred from a wider time period's worth of data, and if the noises observed correlate with people's perceptions of their productivity.

Futher work could also be done to try and determine the nature of a noise too- this would be an interesting application for Tensorflow Lite¹.

¹<https://www.tensorflow.org/lite/>

A | Notable device code

```
uint16_t fft_averages[FFT_SAMPLES/2];
memset(fft_averages, 0, sizeof(fft_averages));
for (int n=1;n<FFT_AVERAGING_WINDOW+1; n++){
    // Sample audio
    // Time taken to sample = FFT_SAMPLES*samplingPeriod = 4096 * 62us = 253952 us
    // = 0.25s
    for (int i=0;i<FFT_SAMPLES;i++){
        sampleAudio(i, samplingPeriod);
    }
    uint16_t* fft_sample = generateFFT();
    for (int i=2;i<FFT_SAMPLES/2;i++){
        uint32_t sum_bin = ((fft_averages[i]*n)+fft_sample[i]);
        uint32_t average = sum_bin/n;
        if (average > UINT16_MAX) { // clamp to maximum value
            average = UINT16_MAX;
        }
        fft_averages[i] = average;
    }
    free(fft_sample);
}
```

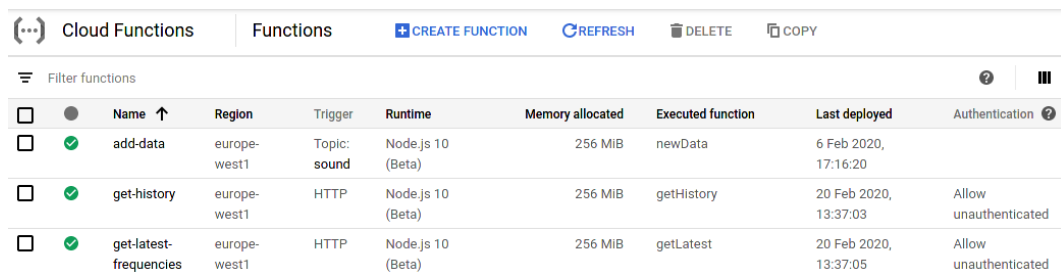
Figure A.1: The is performed multiple times and averaged before transmission

B | Google Cloud setup

This project requires the use of multiple Google cloud resources, which must be configured before running.

Inside IoT Core, a registry must be created. While creating this registry, add a Cloud Pub/Sub topic to publish MQTT events to. This should use a Google-managed key. Inside the newly created registry, add a device. Locally, generate an EC256 key and upload the public key for the device. The private key should be copied into the `lib/transmit/ciotc_config.h` file, which should also contain the name of the device registry and device id (name) for the new device.

Firestore should be enabled, and a collection called 'frequencies' should be added by the cloud function when a device first uploads some data. These functions should be configured to run the 'index.js' file in the ingress repository, with their corresponding function name. They can alternatively be configured to source their code from a Cloud Source Repo, after pushing the whole repository to a new Cloud Source Repo.



	Name ↑	Region	Trigger	Runtime	Memory allocated	Executed function	Last deployed	Authentication ?
<input type="checkbox"/>	add-data	europa-west1	Topic: sound	Node.js 10 (Beta)	256 MiB	newData	6 Feb 2020, 17:16:20	
<input type="checkbox"/>	get-history	europa-west1	HTTP	Node.js 10 (Beta)	256 MiB	getHistory	20 Feb 2020, 13:37:03	Allow unauthenticated
<input type="checkbox"/>	get-latest-frequencies	europa-west1	HTTP	Node.js 10 (Beta)	256 MiB	getLatest	20 Feb 2020, 13:37:05	Allow unauthenticated

Figure B.1: A summary of the Cloud functions

Bibliography

- ANS (2004), Specification for octave-band and fractional-octave-band analog and digital filters, Standard, American National Standards Institute Inc., NY, USA.
URL: <https://law.resource.org/pub/us/cfr/ibr/002/ansi.s1.11.2004.pdf>
- Bankov, D., Khorov, E. and Lyakhov, A. (2016), On the limits of lorawan channel access, in ‘2016 International Conference on Engineering and Telecommunication (EnT)’, pp. 10–14.
- Bird, D. (2018), ‘Esp32-8266-audio-spectrum-display’.
URL: <https://github.com/G6EJD/ESP32-8266-Audio-Spectrum-Display>
- Broadbent, D. E. (1957), ‘Effects of noises of high and low frequency on behaviour’, *Ergonomics* 1(1), 21–29.
URL: <https://doi.org/10.1080/00140135708964568>
- Fletcher, H. and Munson, W. A. (1933), ‘Loudness, its definition, measurement and calculation’, *The Journal of the Acoustical Society of America* 5(2), 82–108.
URL: <https://doi.org/10.1121/1.1915637>
- Gallo, E., Ciarlo, E., Santa, M., Sposato, E., Fogola, J., Grasso, D., Masera, S., Vincent, B. and Halbawachs, Y. (2018), Analysis of leisure noise levels and assessment of policies impact in san salvario district, torino (italy), by low-cost iot noise monitoring network, in ‘Proceedings of Euronoise’.
- Hunke, N., Yusuf, Z., Rüßmann, M., Schmiege, F., Bhatia, A. and Kalra, N. (2017), ‘Winning in iot: It’s all about the business processes’.
URL: <https://www.bcg.com/publications/2017/hardware-software-energy-environment-winning-in-iot-all-about-winning-processes.aspx>
- ISO (2003), Acoustics — normal equal-loudness-level contours, Standard, International Organization for Standardization, Geneva, CH.
- Kjellberg, A. (1990), ‘Subjective, behavioral and psychophysiological effects of noise’, *Scandinavian Journal of Work, Environment & Health* 16, 29–38.
URL: <http://www.jstor.org/stable/40965841>
- Little, S., Zhang, D., Ballas, C., O’Connor, N. E., Prendergast, D., Nolan, K., Quinn, B., Moran, N., Myers, M., Dillon, C. and et al. (2017), Understanding packet loss for sound monitoring in a smart stadium iot testbed, in ‘Proceedings of the First ACM International Workshop on the Engineering of Reliable, Robust, and Secure Embedded Wireless Sensing Systems’, FAILSAFE’17, Association for Computing Machinery, New York, NY, USA, p. 40–45.
URL: <https://doi.org/10.1145/3143337.3143341>
- Moran, N. (2017), ‘Croke park: Sound and weather data monitoring within a smart stadium’.
URL: <https://microsoft.github.io/techcasestudies/iot/2016/10/28/CrokePark.html>

Podesta, J., Pritzker, P., Moniz, E., Holdren, J. and Zients, J. (2014), 'Big data: Seizing opportunities, preserving values'.

URL: https://obamawhitehouse.archives.gov/sites/default/files/docs/big_data_privacy_report_may_1_2014.pdf

Raj, A. (2019), 'Measure sound/noise level in db with microphone and arduino'.

URL: <https://circuitdigest.com/microcontroller-projects/arduino-sound-level-measurement/>

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