VICTORIA UNIVERSITY OF WELLINGTON
TE HERENGA WAKA

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Development of an IoT System for Environmental Monitoring: Software

Benjamin Secker

Supervisor: James Quilty

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering with Honours.

Abstract

Use of the Internet of Things (IoT) is poised to be the next big advancement in environmental monitoring. We present the high-level software side of a proof-of-concept that demonstrates an end-to-end environmental monitoring system, replacing Greater Wellington Regional Council's expensive data loggers with low-cost, IoT centric embedded devices, and it's supporting cloud platform. The proof-of-concept includes a Micropython-based [1] software stack running on an ESP32 microcontroller [2]. The device software includes a built-in webserver that hosts a responsive Web App for configuration of the device. Telemetry data is sent over Vodafone's NB-IoT network [3] and stored in Azure IoT Central [4], where it can be visualised and exported.

While future development is required for a production-ready system, the proof-of-concept justifies the use of modern IoT technologies for environmental monitoring. The open source nature of the project means that the knowledge gained can be re-used and modified to suit the use-cases for other organisations.

Acknowledgments

I would like to acknowledge the hard work put in by Jolon Behrent for the hardware side of this project. I would also like to thank James Quilty for his much-needed help, encouragement and support, as well as Dave Turner and his colleagues at Greater Wellington Regional Council for being the best clients we could possibly ask for.

Contents

Figures

Chapter 1

Introduction

Environmental monitoring of the Greater Wellington region is required to be conducted by Greater Wellington Regional Council (GWRC). Groundwater quality, air quality, river levels and even early flood monitoring are some tasks undertaken by GWRC on a daily basis. This is currently done by a network of expensive, power hungry devices provided by a single supplier. The aim of this project was to explore the benefits that IoT technologies can offer, whilst maintaining the sensor accuracy required for scientific studies and regulatory functions.

GWRC, in the planning process, set out a list of principles that form the basis of the project requirements. These are:

High Accuracy. The data collected using the high-detail environmental monitoring sensors must be transformed, collected and transmitted with accurate timestamps, exhibit no data loss in transmission and be visualised and stored reliably.

High Reliability. The devices and supporting cloud software must be as reliable as possible - fault tolerant, maintainable and using well-tested code.

High Usability. Deliver an open-source system that is fully documented which will allow GWRC staff to up-skill and understand IoT concepts, as well as providing the GWRC IT department the ability to support the server and data transfer components in the future.

This report presents the high-level software side of a proof-of-concept end-to-end solution for monitoring environmental data through low-cost, custom-designed IoT Devices, connected to Microsoft Azure IoT Central over wireless networks. As a two-person project, the hardware design and low-level software is discussed in Ref. [5].

1.0.1 Project Software Requirements and Scope Reduction

A full list of initial project requirements is given in Ref. [6]. Of these, the most important and relevant requirements are:

- Device requirements
  - Ability to sleep in low power mode when not in use to maintain 3+ months battery life
  - Ability to configure the device on-site from a mobile device
  - Wireless transmission of data over wireless networks to the cloud backend

  - Ability to save the sensor recordings to an SD card.
  - Accurate time-keeping and recording, waking up at regularly scheduled intervals to take and transmit readings.

- App Requirements
  - Ability to calibrate sensor settings, boot time, and data recording and transmission parameters
  - Ability to put the device into some form of “test mode”, temporarily disabling data transmission and allowing test readings of the sensors to be taken
  - Implementation of a serial-terminal like monitor where manual SDI-12 commands can be sent to the sensors

- Cloud requirements
  - Hosted on a virtual server, with a flexible data storage format to allow future integration with GWRC databases
  - Ability to view tabular and graph-formatted data, and the ability to send an alert on unexpected values
  - Integration with or creation of a long-term ”cold” datastore for telemetry data

A number of requirements for additional features from the original list had to be dropped in favour of improving the reliability of the device. Notably, the requirement to be able to take rainfall measurements using GWRC’s rainfall sensors required additional hardware and a new software design to be able to handle the high-frequency reading requirements while also being power-efficient. Secondly, numerous technical difficulties with the NB-IoT modems encountered late in the project (section 4.3 and Ref. [5]) forced a re-prioritisation of features, dropping the device visualisation and cloud-to-device communication requirements.

Chapter 2

Background Survey and Related Work

Many governments and private companies around the world use IoT for Environmental monitoring. Measuring city air pollution, early flood warning systems or river quality measuring, are just a few types of real-world projects.

This chapter discusses Waterwatch, a commercial competitor to this project, and how their product's shortcomings ultimately caused GWRC to reject their solution. Following this is an evaluation of the different network technologies, cloud platforms, messaging protocols and embedded programming languages which influenced the design of our end-to-end solution.

2.1 Waterwatch: An Existing Proprietary Solution

Waterwatch was pitched to GWRC earlier in 2020 as a cradle-to-grave platform for environmental monitoring. The platform is built around managing multiple IoT water-monitoring nodes called *LS1 Sensors* that use Ultrasonic sensors, claimed to be accurate to 1% in the Waterwatch Datasheet [7].

The devices connect to a Software-as-a-service (SaaS) based cloud platform built on Amazon Web Services (AWS) [8] via Sigfox [9] or Narrow-band IoT (NB-IoT) on the Vodafone or Spark networks [3]. They offer a mobile app, which connects to the device via Bluetooth and allows the user to configure the device on-site.

Waterwatch's solution did not gain traction with GWRC, despite being supported by a private company. The main reason for not choosing this platform was that the sensors used on the LS1 were cheap ultrasonic sensors, and did not have the accuracy that GWRC require. Furthermore, they do not have the ability to add external SDI-12 sensors, limiting the nearly $1000 devices to the on-board measuring tools. Finally, GWRC was dissatisfied with the level of product support received from Waterwatch during the trial, and had concerns about the longevity of the company.

The main benefit of exploring this product is that it gives a real-world example of a complete end-to-end solution. It shows that cloud-based IoT monitoring is an effective model and presents opportunities for innovation in the New Zealand market. Also, the custom dashboard that Waterwatch Live implements shows some interesting UI/UX decisions that could be used as inspiration in the design of the dashboard in our project.

Furthermore, the failings of Waterwatch should indicate to us areas where we need to focus on delivering high quality work. Accuracy and reliability are the highest priority requirements of GWRC, and we should strive to have good documentation and well written code so that GWRC can continue to build the platform after this year.

2.2 Low-Power Wide Area Network Technology Comparison

Low Power Wide Area Network implementations differ in scalability, Quality of Service (QoS), network coverage and transmission range [10]. GWRC's preferred network is Vodafone's Narrowband IoT (NB-IoT) network [3] due to their existing business partnership with Vodafone, but alternate technologies Sigfox [9] and LoRa [11] were also reviewed due to their respective technical strengths [10].

LoRa is a physical layer technology that also uses unlicensed 868–928MHz bands. It's use would require GWRC to set up their own base stations, which is infeasible for this project. Sigfox is a proprietary, bi-directional, radio network using unlicensed 920MHz bands. By using an ultra-narrow 100 Hz band, Sigfox has very low noise levels and power consumption at the expense of a 100 bps maximum throughput. It is capable of only sending only 140 12–byte messages a day. NB-IoT is a state-of-the art extension of the well established LTE network infrastructure, able to send an unlimited amount of up to 1600 byte messages per day.

The choice of network is crucial for this project because it ties together the devices and the cloud backend. GWRC's high message frequency requirement means that Sigfox's data limits would prohibit the usefulness of the system. Furthermore, GWRC identified coverage gaps in the Greater Wellington region with Sigfox. These factors formed the basis of our network investigation in Ref. [12], justifying why NB-IoT was the most effective choice.

2.3 Internet-of-Things Messaging Protocols

Just as important as the choice of physical layer network technology is the application-layer message protocol for sending data to the cloud. The paper "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP" [13] compares four of the most widely-used IoT messaging protocols, comparing the technologies according to different criteria, including protocol architecture, message overhead, power consumption and quality-of-service features.

MQTT is a lightweight publish/subscribe protocol, where messages are published to "topics" in an MQTT broker which clients subscribe to. It is a binary protocol running on TCP [14], and has three levels of Quality of Service (QoS) guarantees.

CoAP is not supported in our cloud platform of choice (section 2.4) and, while its Universal Resource Identifier (URI) based design offers greater functionality than MQTT, CoAP was not investigated further.

HTTP is a text-based protocol based on a request/response architecture. Like CoAP, it sends data through the URI instead of topics. Is uses TCP as the transport protocol, but doesn't explicitly have any QoS features. AMQP, meanwhile, is an advanced messaging protocol that supports both publish/subscribe and request response architectures. It is a binary protocol, built on TCP, and adds a wide range of features for reliable messaging such as queuing, flexible routing and transactions.

Out of the three evaluated protocols, MQTT has the lowest message size vs message overhead (2 bytes per message), followed by AMQP, then HTTP. A similar pattern was found in the energy consumption of the three protocols when no QoS rules are applied. On the other hand, HTTP has the most interoperability with web systems, and its larger user base means that there is a vast amount of documentation and tutorials available online.

MQTT was ultimately chosen for this project because it's low bandwidth requirements go well with the constrained nature of NB-IoT. Furthermore, the QoS features ensure messages are sent reliably, and the protocol is well supported by all IoT cloud platforms.

2.4 Cloud Software Platform Comparison

The paper "Investigating IoT Middleware Platforms for Smart Application Development" [15] judiciously reviews the state of open source and private cloud based middleware platforms.

Out of the platforms reviewed, Microsoft Azure [16], Amazon Web Services [8], and Google Cloud [17] were considered for this project. These three platforms are the biggest in the cloud market, and are backed by large companies. This means that pricing is comparatively low due to economies of scale. Also, better community and professional support is more likely with one of these platforms.

Google Cloud and AWS are almost identical in terms of offered IoT services. For example, Google Cloud Functions [18], a serverless event-driven compute platform, is rivalled for features by Amazon AWS Lambda [19]. The paper, in the same vein, doesn't significantly differentiate AWS and Google Cloud in terms of features, deployment types, or pricing models.

Microsoft Azure has similar service-based IoT applications, but is differentiated from the other two through its Software-as-a-service (SaaS) platform *Iot Central* [4]. IoT Central is a simple yet scalable platform that merges the main IoT services under one interface.

Although all platforms offer similar services, ours and the client's top pick was Microsoft Azure, for two reasons: The first is because GWRC has existing internal infrastructure, such as Active Directory running on it, meaning that deploying and integrating our software would be more streamlined. Maintenance is also made easier because the GWRC's IT administrators only have a single cloud platform to become familiar with. Secondly, Azure IoT Central seemed to completely satisfy our cloud requirements in a self-contained platform, simplifying deployment immensely.

2.5 Micropython vs. C

Micropython [1] is an efficient implementation of the Python 3 language that is optimised to run bare-metal on a resource-constrained microcontroller. The paper "Evaluation of MicroPython as Application Layer Programming Language on CubeSats" [20] discusses the implications of using Micropython in a project that values *reliability and accuracy* over anything else, a fundamental philosophy shared by this project.

Micropython was evaluated against Arduino (essentially C), which most low-cost IoT microcontrollers support. The main benefit of C is performance. Speed (processing and IO throughput), memory usage and battery life are improved with a low level language - certain tests showed C being 90x faster than Micropython [21]. Additionally, more software libraries are available for C/Arduino as it is more mature than Micropython - for example, Micropython does not yet have a good SDI-12 [22] library, which is required for communication with the external sensors.

On the other hand, Micropython trades sheer performance for ease of implementation and rapid prototyping through readability, writability and expressivity. Given the projects small time frame, being able to rapidly test new ideas and get immediate stakeholder feedback are crucial. Micropython has excellent exception handling, allowing errors to be caught, tested and fixed. Also, modern microcontrollers are easily fast enough to run Micropython effectively. Furthermore, Micropython has support for integrating external C modules, and even supports inline assembly, so when highly optimised code is required we can get the best of both worlds. These reasons formed the basis of our decision to use Micropython on our devices.

Chapter 3

Design

3.1 Overview

This chapter concentrates on the conceptualisation of a proof of concept that is able to demonstrate the feasibility of an IoT based environmental monitoring solution. With reference to the projects core principles, the following sections present a design that encompasses the high level device software, Web App and cloud platform.

Use-cases were defined in collaboration with the customer to refine project requirements [23]. The use-cases important in informing the design of the system architecture were:

Monitoring the state of a river - The primary use case. This requires the device to have functioning sensors, a way to transmit the data and a low-powered sleep mode for extended operation.

Configuring the device - This requires a mobile-friendly application for making modifications in-situ.

Visualising and downloading device data - This requires a web-based application to look at long-term time series data, visualised with useful graphs like how their existing system allows [24].

Backed by the earlier background research, these use-cases inform the high-level overall system architecture, shown in figure 3.1. The architecture uses Vodafone's NB-IoT network to transmit telemetry data from the IoT devices to Azure, where it can be visualised and stored long term.

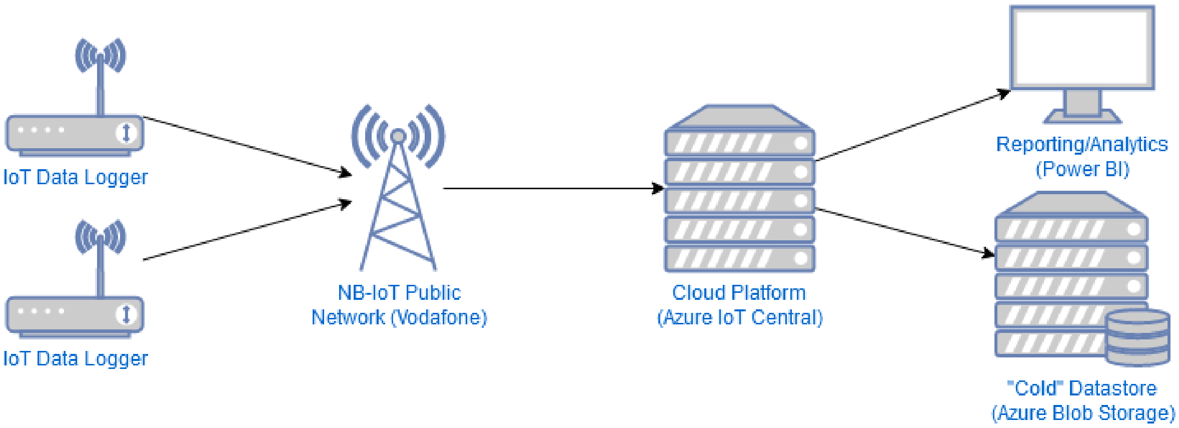


Figure 3.1: Overall system architecture, including Azure cloud services.

One limitation is the reliance of this design on Microsoft Azure. To remove this dependency, we considered a design variation which adds a ”Cloud Bridge” adapter component between the NB-IoT network and Azure. The ”Cloud Bridge” would be a self-hosted, custom-designed internet accessible server that the IoT devices would connect and send data to. It would be responsible for aggregating this data and forwarding it to the cloud platform, effectively decoupling the cloud platform from the devices. This would allow the devices to be cloud-service-provider agnostic, and using an alternative to Microsoft Azure would require no modification of the device code. However, implementing this would add an extra layer of complexity which the client did not see benefit in. There was a low chance of moving away from Azure, and the time taken to implement a cloud bridge would have limited the time available to develop other important features.

The following sections discuss in detail the design of the main components of the system shown in figure 3.1: the embedded device software, the Web App architecture and user interface, and the cloud architecture components.

3.2 Device Software Architecture

Constructing an efficient software architecture for the devices is important for reliability, testing and ease of collaboration [6]. This section focusses on the high-level software design, including the modes of operation and web server design. Readers are encouraged to refer to Ref. [5] for low-level device software design.

3.2.1 Constraints

The design of the device software is constrained to suit the low-power hardware chosen for the project. Specifically, the ESP32 microcontroller has 8MB of RAM, and 4MB of non-volatile ”disk” (flash memory) space [2]. The RAM constraints may potentially be reduced to 100KB in the future to free up some hardware pins and reduce power usage. Moreover, to obtain maximum power savings, the device needs to be in ”deep sleep” - extremely low power mode where nearly everything is turned off - for as long as possible. The software needs to be designed around this. In deep sleep mode, no code can run, so all the processing required should be done in batches before resuming deep sleep.

3.2.2 Modes of Operation

Following the different use cases presented in section 3.1, the high level software was split into two main logical ”modes”, *Configure Mode* and *Regular Mode*.

Configure Mode is initiated by the user via pressing a push-button or activating a hall-effect sensor with a magnet. It starts a WiFi access point and the webserver which allows the user to modify configuration values of the device. Once the user has finished configuring the device, a pushbutton or time-out would return the device back to sleep, or into regular mode if the user was configuring the device when a scheduled reading was meant to take place.

Regular Mode is the regular operation of the device. When the device boots up from deep sleep, it takes a reading of the SDI-12 sensor, and periodically transmits the data to IoT Central before going back to sleep.

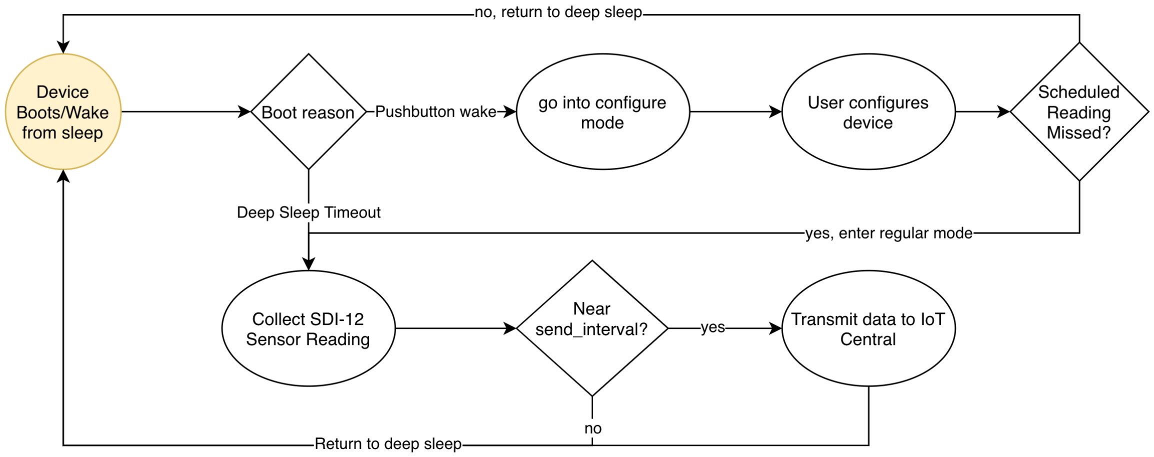


Figure 3.2: Process diagram of Configure and Regular modes, starting from when the device boots

This separation was chosen because it logically matches the two main use cases of the device - *Configuring the device* and *Monitoring the state of a river*, allowing for a workflow shown in figure 3.2.

3.2.3 Architecture Overview

Figure 3.3 shows a stack-diagram overview of our multi-tiered device-code architecture, building upon the existing community-provided micropython firmware. At the top level, the high level code is responsible for handling the ”pipeline” of regular operation as well as enabling/disabling the webserver. The high level code makes use of different ”services” - logically grouped functions and classes for particular device functionality, such as the webserver or device configuration. In turn, the services directly use community-provided libraries. They also need to communicate with the hardware. Low-level device drivers act as a translator between hardware devices and the high-level software, abstracting away the technical details and providing a common interface so that high-level code can be written independently of the specific hardware. At the bottom layers, firmware compiles high-level python code into low-level bytecode which is run by the micropython virtual machine on the ESP32 hardware. External hardware, such as the modem and SD Card reader, are communicated with via serial communication busses on the ESP32.

Designing the software in this way allows for multiple benefits:

- *Separation of concerns*: components in each layer should only communicate with adjacent layers. In theory, this decouples the components so that layers can be modified without affecting the rest of the stack.
- *Increased Collaboration*: By agreeing on shared interfaces between layers, one team member is able to work on the high-level software while the other can work on low-level code without fear of interfering with each-other.
- *Increased Testability*: Each layer can be tested in isolation, making it easier to perform unit and integration testing.

Other possible architectures were also considered: The best alternative, known as “feature oriented architecture”, takes cues from Service-Oriented Architecture (SOA), a pattern

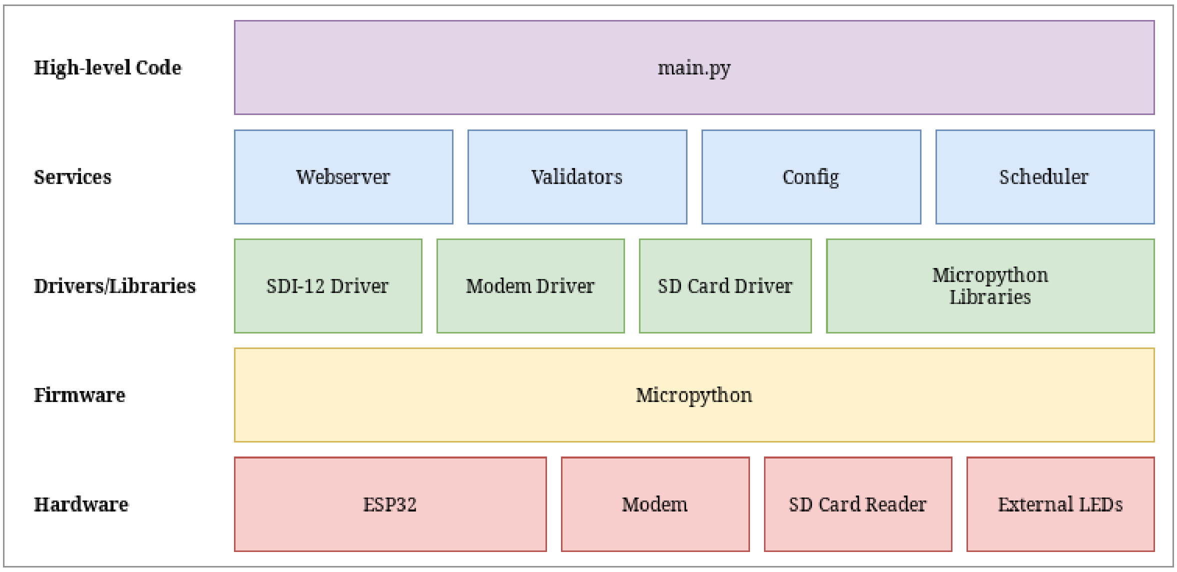


Figure 3.3: Device Architecture Stack Diagram, showing main components of the software.

commonly seen in the web-development industry [25]. In this style, each high-level logical component of the system is encapsulated into it's own "service" with it's own dependencies, effectively splitting the architecture shown in figure 3.3 into columns rather than rows. This style promotes loose coupling between the different modes of operation on the device, which allows for independent development of different features, as well as easier testing. However, a feature-oriented style doesn't work particularly well in this context due to the high amount of shared resources. Hardware device drivers, for example, must be a singleton: only one instance of the driver can exist [26]. This approach would work in a more modular hardware design - like with independent microcontrollers for separate functionality on the device - but for a proof-of-concept, this would only add needless complexity.

Web Server

The Web Server - the *backend* - is responsible for serving up the Web App to the client - the *frontend*. The web server was designed around a webserver library called TinyWeb [27], a lightweight, single-threaded HTTP server for Micropython. The primary concern was the performance of TinyWeb on the ESP32 microcontroller: *is it fast enough to provide a responsive user experience?* To answer this, we conducted a performance experiment to compare transmission time of increasingly-sized HTTP responses [28]. The results showed that the time increased linearly with response size, which was expected. A 100 KB file took 138 ms to send on average, which is just at the limit of where a user starts to notice latency [29].

The web server library follows industry-standard REST API guidelines, which allowed us to implement a simple subset of the final web-app, discussed in Ref. [30]. The proof-of-concept demonstrates that the webserver is capable of serving static content, sending the client a responsive HTML page with minified CSS. It also shows that two-way interaction (sending form data as an HTTP POST request) and basic dynamic templating is possible, functionally verifying that the webserver approach is effective for a device configuration mechanism.

Various other webserver libraries were researched and tested, including *MicroWebSrv* [31], *MicroWebSrv2* [32] and *Picoweb* [33], but each had major issues which limited their functionality [34].

Server-side vs Client Side Rendering

Choosing whether to use server-side rendering or client-side rendering for the Web App was an important design decision. Server side rendering (SSR) is when the server is responsible for creating the entire web page, and then sending that to the client. If the page requires dynamic data (for example, results from a sensor reading), the server populates the page with dynamic data using a *template engine.* This method works well for normal webservers, because they usually have high compute capacity and are able to dynamically render multiple webpages at once. However, the low CPU and RAM constraints on the microcontroller hardware mean that building websites on the device would be both slow and power consuming.

Client-side rendering (CSR) is a newer approach made possible by modern frameworks such as React [35], where the compilation of dynamic content and generation of the webpage is undertaken by the client web browser. The webserver sends the static HTML and CSS to the browser, and the browser uses the webserver's API via asynchronous javascript calls to get the dynamic data.

This approach is preferred in this situation because it keeps the webserver resource demands on the device as light as possible. The other major benefit is caching. Caching with SSR is difficult on a microcontroller due to the resource constraints, and so the server needs to send the entire page every time. A client, meanwhile, can cache repeated requests, which means subsequent configuration attempts only require getting the dynamic data. This improves page loading times significantly and reduces the load on the webserver.

REST API

With the webserver performance verified and the client-side rendering approach chosen, the concrete REST API outline was planned in Ref. [36].

An overview of the endpoints is shown in figure 3.1:

```
GET / - return static webpage
GET /data - get dynamic device data
GET /config - get dynamic device config
POST /config - update data
    POST /config/sdi12/<name> - update sensor with name <name>
    POST /config/sdi12/<name>/test - test sensor with name <name>
GET /monitor - get raw sdi-12 responses
```

Listing 3.1: Rest API Endpoint Design

3.3 Web App Design

GWRC require a way to configure certain device settings in the field [37]. This section focusses primarily on the user interface design of the Web App, discussing the design decisions that went into the app in order to make an easy-to-use, yet functional interface.

3.3.1 Mobile App vs Web App

Despite the above sections discussing a web-server based configuration method using WiFi, a feasible alternative was to configure the device using a native iOS app, connected to the device via Bluetooth. We conducted an expensive comparison in Ref. [38], finding this was a difficult decision because both options had a number of good points. Both were technically feasible, because our microcontrollers have built-in Bluetooth and WiFi on the chip, and both are strongly supported by lots of well-supported libraries [38, 34]. Furthermore, both options satisfied the client's requirements for mobile phone-based configuration.

Eventually, the choice was made to go with a Web App. The primary reason was because GWRC could not guarantee that they would get Apple Business Manager, so distribution of the app to GWRC phones would have to go through the Apple App Store [39]. This meant that the app would need to get published publicly, going through Apple's complicated app submission process and requiring lots more custom code to add authentication and security measures. Furthermore, The Web App approach simplified deployment, requiring implementing only one update stream - updating the device firmware. Without spending lots of time writing backwards compatible iOS device software, both the app and the device would have to be updated synchronously, adding extra time overhead to software delivery.

3.3.2 User Interface Design

The User Interface needed to be designed around the complex workflow associated with the sensors. Each IoT *device* is built to support an unlimited number of SDI-12 *sensors*, and each sensor can have a number of *readings*. Sensor readings can be calibrated with multipliers and offset variables.

The design of the webapp was constrained in a few ways:

- The Web App had to be totally usable on a mobile-phone sized screen - as small as 375px wide. As a result, a Mobile-First Design philosophy had to be followed to get the best results. Mobile-First Design aims to create better experiences for users by starting the design process for small screens, and then scaling the design *up* to large screens.

- The Web App needed to have a familiar workflow for it's intended users. GWRC have used HyQuest [40] - their existing monitoring software - for multiple years, and are very proficient in using it. The Web App needs to be able to be used without lots of training, and should be as intuitive as possible.

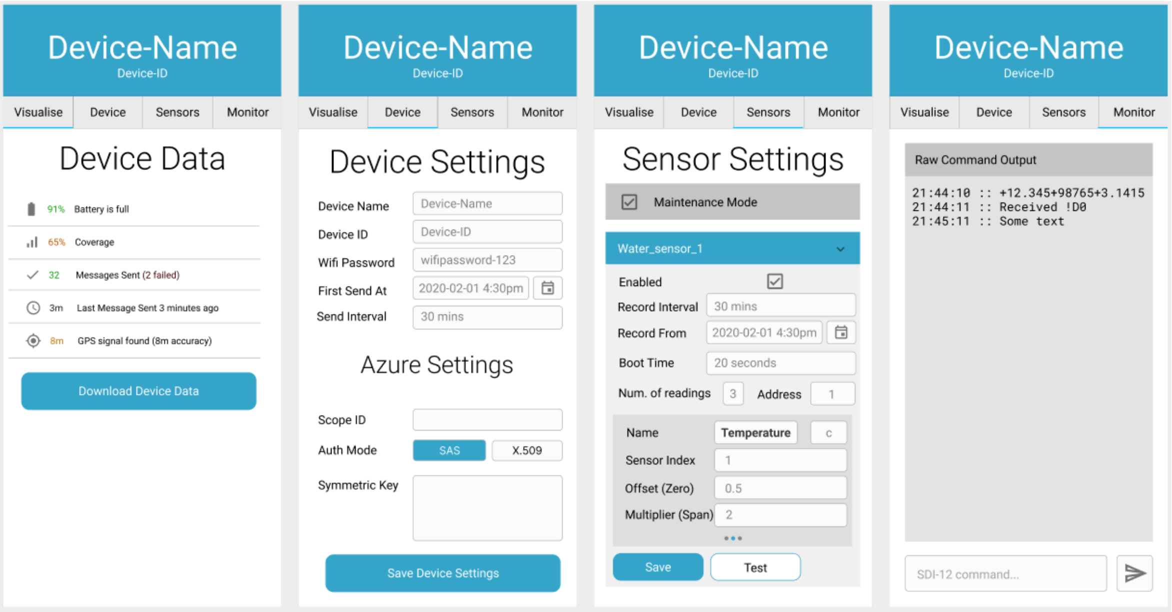
- The built Web App needed to be well under 100KB in size to ensure the ESP32 could transmit the page in a reasonable time (section 3.2.3). This means that the Web App needed a low amount of images, as little javascript as possible (limiting the amount of libraries in use) and simple page design.

- The app should have no dependencies on external servers to provide javascript libraries or styling, because there is often limited network coverage in the field.

With these constraints in mind, the design shown in figure 3.3.2 of the final mockup was created following an iterative design process involving customer feedback. The final design aims to balance functionality, usability and Web App size.

The application is laid out across four main screens. *Visualise* shows metrics collected by the device. *Device* contains forms for changing the device's settings, such as the device identifier, sleep schedule and Azure authentication settings. *Sensors* shows settings for connected SDI-12 sensors. *Monitor* shows a window that functions like a "terminal", where SDI-12 AT commands can be manually entered in.

If the application viewport - the user's visible area of a web page - is expanded horizontally, the page becomes split-screen, split vertically. The Monitor page is always displayed on the right column, and the left column can switch between the three other tabs. This is useful on laptops and tablets with lots of screen real estate because it means that manual commands can be cross-referenced with the configuration on the other side of the page.



Device Data

91% Battery is full

65% Coverage

✓ 32 Messages Sent (2 failed)

3m Last Message Sent 3 minutes ago

8m GPS signal found (8m accuracy)

Device Settings

Device Name	Device-Name
Device ID	Device-ID
Wifi Password	wifipassword-123
First Send At	2020-02-01 4:30pm
Send Interval	30 mins

Azure Settings

Scope ID	
Auth Mode	[✓ SAS] X.509
Symmetric Key	

Sensor Settings

☑ Maintenance Mode

Water_sensor_1

Enabled	☑
Record Interval	30 mins
Record From	2020-02-01 4:30pm
Boot Time	20 seconds
Num. of readings	3
Address	1
Name	Temperature c
Sensor Index	1
Offset (Zero)	0.5
Multiplier (Span)	2

Raw Command Output

```
21:44:10 :: +12.345+98765+3.1415
21:44:11 :: Received !D0
21:45:11 :: Some text
```

Figure 3.4: Final Web App design mockup, viewed on a mobile screen. Pages from left to right: *Visualise*, *Device*, *Sensors* and *Monitor.*

A multi-tabbed design was settled on to separate the different directions of interaction: visualising (device to app), and configuring (app to device). This distinction is important because it reflects the separation of the app's mutually exclusive main use-cases, discussed further in Ref. [41].

User Experience (UX) principles from the book *Mobile Usability* [42] were employed to try achieve a pleasant user experience. The four page design ensures the cognitive overhead from the abundance of configuration options is kept to a minimum, and so the user only sees what is contextually relevant. This was inspired from *Mobile Usability, Chapter 4: Defer Secondary Information to Secondary Screens*.

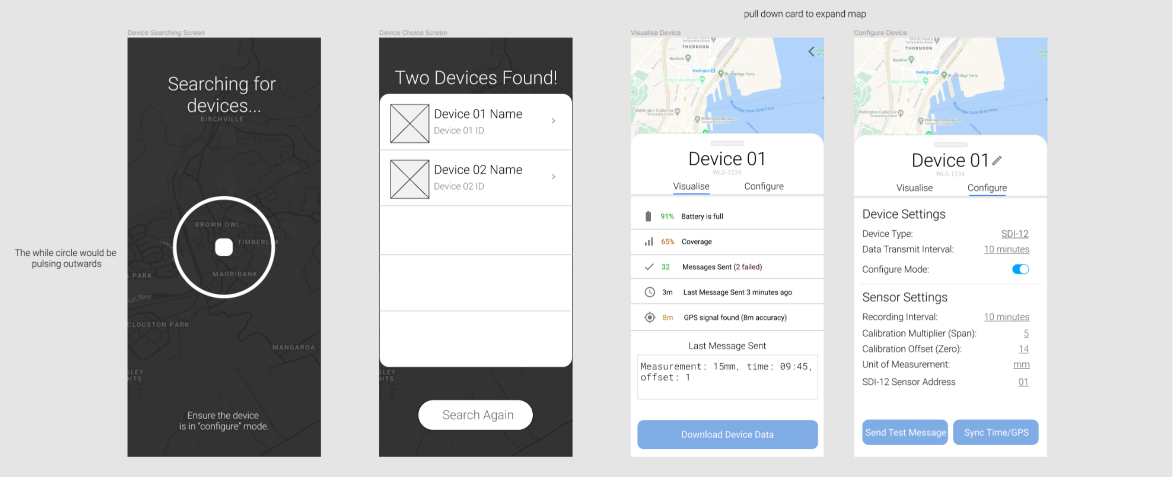
Another element is large, easy-to-press buttons that prioritise visual clarity, inspired by Apple's Human Interface Guidelines [43]. "Primary" actions are coloured blue, and "secondary" actions are white with a blue outline. This makes it obvious which button is most important to be pressed. Various alternatives were investigated [44], such as using Floating Action Buttons - fixed buttons in a corner of a screen - but the designs didn't scale well to large screens.

In the sensor settings page, there are a large amount of options for configuring an unlimited number of sensors. Because of this, the page could easily become overly information dense and confusing for the reader. The solution was to group sensors into collapsible tabs. Clicking on a tab expands the settings for that sensor and closes the rest, so that only the most relevant information is shown. Inside the tab, a carousel-style swipe-able list of sensor readings is shown. This follows the same theme of information hiding, and lets the user focus only on the sensor they are currently configuring.

3.3.3 Alternative Designs

Multiple other overall designs, and variations on the final design, are discussed in Ref. [44]. The most fleshed out design is shown in figure 3.3.3.

The first two screens are specific to a Bluetooth-based app, but the last two screens are



91% Battery is full
65% Coverage
32 Messages Sent (2 failed)
3m Last Message Sent 3 minutes ago
8m GPS signal found (8m accuracy)

Last Message Sent

```
Measurement: 15mm, time: 09:45,
offset: 1
```

Device Settings
Device Type: SDI-12
Data Transmit Interval: 10 minutes
Configure Mode: ●

Sensor Settings
Recording Interval: 10 minutes
Calibration Multiplier (Span): 5
Calibration Offset (Zero): 14
Unit of Measurement: mm
SDI-12 Sensor Address 01

Figure 3.5: Alternative bluetooth-based app design for mobile devices.

common to both app approaches. The graphic design tool Figma [45] was used to build an interactive model, which aided immensely in showing the workflow of the app.

The app takes inspiration from Uber for the device-search screen, and from Google Maps for the "card" layout [46]. The "device card" overlays a map, which would show the locations of the device as an icon. Users would be able to swipe away the card to show the device and devices around it on the map.

The main problems with this design were that the app was too heavyweight with aesthetic features, and so the time taken to load a page would be too much due to the number of images and interactive elements. Secondly, whilst the map was aesthetically pleasing, it requires network access to render (without storing the entire map of New Zealand on the client's device or webserver). For a device that is built to be placed in remote areas with limited network coverage, this means that parts of the page might not load, making for a worse user experience.

3.4 Cloud Architecture

A significant part of the project is a centralised system to store data received from the IoT devices. While GWRC initially wanted to self-host the backend systems, they later removed this requirement in favour of a cloud-hosted backend [6].

The main constraints for the cloud design was cost - while GWRC didn't have a set budget, a cloud system costing multiple thousands of dollars per year to run wouldn't make it worth switching from their existing system.

With reference to section 2.4 of the background reading, extensive comparisons between the three most popular cloud platforms - Google Cloud, Amazon AWS, and Microsoft Azure - were conducted in Ref. [47]. Microsoft Azure was chosen as the cloud platform because it's SaaS based IoT Central product appeared to satisfy all the requirements in the preliminary testing conducted [48].

This section discusses why IoT Central was chosen over the more popular Platform-as-a-service cloud architecture, as well as other cloud considerations such as security, visualisation and long-term data storage options that fed into the cloud design.

3.4.1 Software-as-a-Service vs Platform-as-a-Service

Before settling on IoT Central, the initial cloud design mimicked Azure's reference IoT Architecture[49], shown in figure 3.6 (excluding the Machine Learning service or Iot Edge Devices components). This alternative architecture is a Platform-as-a-service (PaaS) architecture, because it provides a flexible environment based around managed services. In this model, IoT Devices (provisioned with IoT DPS [50]) send data to IoT Hub [51]. Azure Function Apps [52] pull and transform this data from IoT Hub, storing data in ”warm path stores” (temporary datastores for analysis) and ”cold path stores” (long-term, reliable data storage for backups). Azure Stream Processing [53] connects IoT Hub and business intelligence services for reporting, analysing and integrating the data.

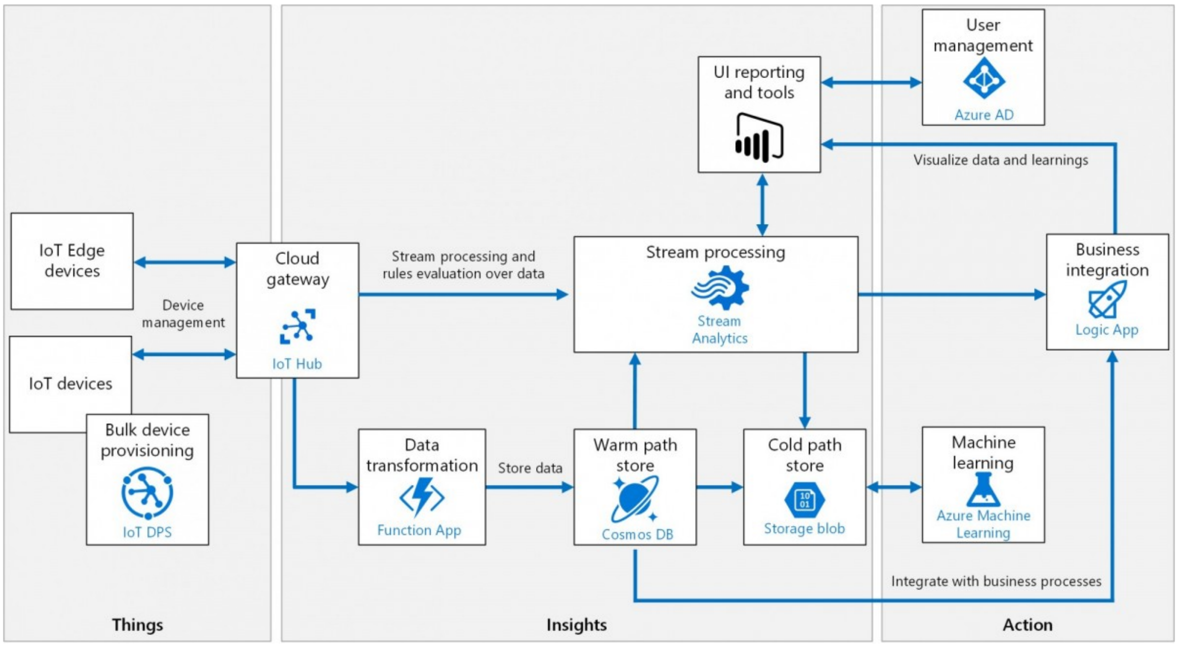


Figure 3.6: Azure Reference PaaS Architecture for IoT.

This model is good because it fulfils all of our clients requirements for storage, analytics and reporting the environmental data from our devices. It is scalable on demand, and could handle thousands of connected devices. The system is highly flexible and composable, constructed on a number of building blocks that enable the integration of existing systems and components. The ”hot” and ”cold” storage paths provide solution for handling both low-latency, real time data and long-term historical data. Furthermore, this platform is fault tolerant and highly available using failover and cross-region support features from the services.

However, with this flexibility comes a management overhead that adds time and complexity, impacting the ability to deliver a strong solution for this project. IoT Central runs most of these services under the hood, but abstracts away the management of separate services away from us, reducing the burden and cost of developing, managing and maintaining the complex IoT infrastructure.

A cost analysis of Iot Central found it to be very reasonably priced even with over 200 devices connected - the project repository contains the full details of the cost analysis [48].

3.4.2 Security Design

Security is an important consideration in this project. Data integrity is the biggest issue - the data needs to be *accurate* and not tampered with, rather than confidential (as the data is all public anyway).

Azure IoT Central enforces strict security practises for devices to be able to connect. Described further in Ref. [54], a public key certificate based authentication system was created and demonstrated using a simulated device. This involved generating a temporary X.509 root and intermediate certificate [55], and configuring our instance of IoT Central to trust the certificates. This allowed for the creation of leaf certificates and private keys, which when uploaded to the device, allowed for secured communication with the cloud software. This effectively lays the basis of a secure-by-default design.

Symmetric key based authentication was also investigated, where the device and cloud both agree on a single key for encrypting messages. While symmetric key encryption runs into issues at large scale, for this project it made more sense to use because it is much less administrative overhead to maintain an intermediary certificate store. Furthermore, GWRC do not have the resources or knowledge required to maintain an X509 certificate-based system in the long term.

3.4.3 Visualisation and Long Term Storage

IoT Central has a basic visualisation and analytics suite for viewing device telemetry. We conducted preliminary tests with the client to test it's suitability, finding it would likely be able to satisfy the client's requirement for rudimentary environmental data analytics.

To implement the requirement for "cold data stores", Azure Blob Storage was recommended by Microsoft for historical iot data storage. It is a non-structured "data-lake" service used to store large amounts of data very cheaply, with the limitation of slow data retrieval. IoT Central allows the two services to be connected in a matter of a few clicks. Azure Cosmos DB or even a self-hosted database were investigated, but Microsoft's economies of scale in their cloud platform meant that the price alone for Blob Storage vastly outweighed the alternatives.

Chapter 4

Implementation

4.1 Device Software Implementation

The main design requirements for the device software are for the device to function in *Regular Mode* when woken from deep sleep and to enter *Configure Mode* when requested.

Regular Mode was implemented using a functional design [56] that follows closely to the design set out in section 3.2.2. At a high level, *Regular Mode* is implemented as a pipeline that follows a sequence of steps after booting up. Firstly, the hardware interface drivers, such as WiFi and the SDI-12 UART, are initialised. Then, the pipeline uses the *Config Service* (section 4.1.4) to get the device config, filtering out the disabled sensors. After this, readings are taken from all enabled sensors concurrently, and the results are merged into one JSON object. These readings are logged to the SD card and then transmitted over the modem. Once all of this has finished, the device uses the *sleep scheduler* (section 4.1.5) to return to deep sleep.

Unfortunately, for reasons discussed in the hardware report [5], the NB-IoT modems were unable to achieve any reliable data transmission. As a workaround, the device code was changed to make use of the microcontroller's on-board WiFi modem, allowing the device to connect to a WiFi hotspot to send telemetry data.

Configure Mode is designed to be implemented with the TinyWeb library (section 3.2.3) and follows the Object Oriented design style of the library. It is used to update the device configuration, discussed in section 4.1.3.

4.1.1 Asyncio for cooperative multitasking

We employed multitasking to speed up processing by performing other tasks while waiting on external interfaces (such as waiting on a sensor reading). This multitasking is performed using *uasyncio*, a micropython port of Asyncio [57]. Uasyncio performs *cooperative multitasking*, which is when the code explicitly yields control of the CPU, allowing the thread scheduler to pick up a paused task [58]. This differs from the traditional method of *preemptive multitasking*, where the thread scheduler gives each thread a limited time-slice and switches between them as it sees fit. Cooperative multitasking is required for parts of the project that require explicit control of shared resources, such as UART drivers, which have strict timing and exclusivity requirements discussed in Ref. [5].

4.1.2 Transmission of Telemetry using Azure Service and Mobile Hotspot

The software code that performs the Azure communication over WiFi is contained in the *Azure Service*. It is built on the *IOTC* library [59], published by Azure for interacting with

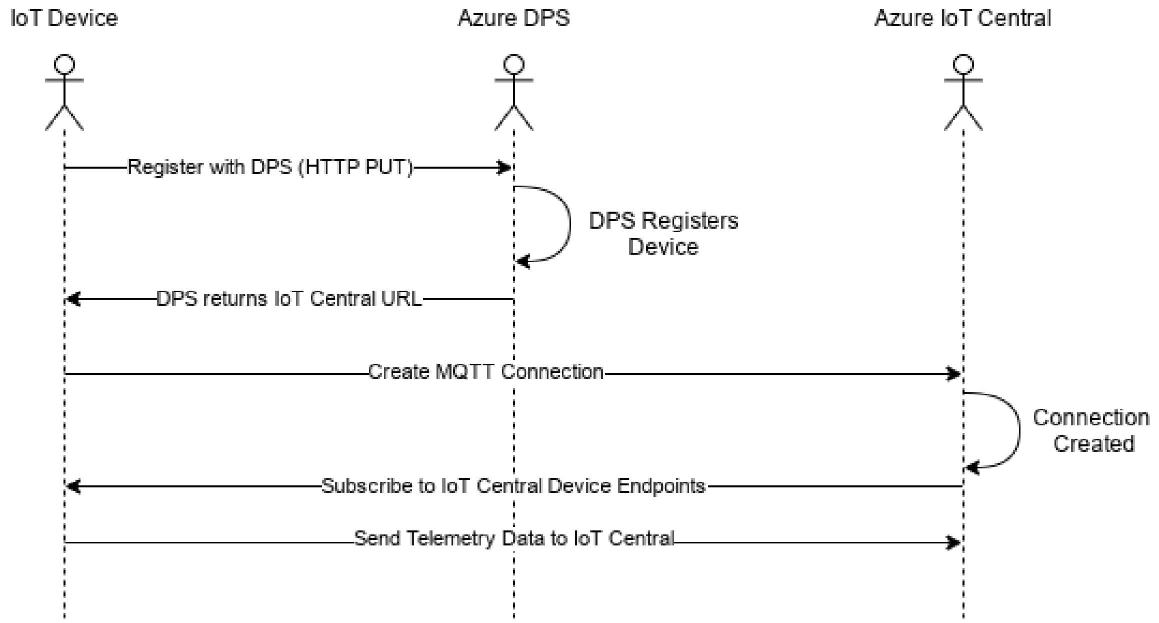

IoT Device
Azure DPS
Azure IoT Central
Register with DPS (HTTP PUT)
DPS Registers Device
DPS returns IoT Central URL
Create MQTT Connection
Connection Created
Subscribe to IoT Central Device Endpoints
Send Telemetry Data to IoT Central

Figure 4.1: Process Diagram of the steps the device goes through to connect to and send data to IoT Central

IoT Central. The library forms the basis of the communication between the device and the cloud, and as such, is a key component of the device software.

Functional verification of device to cloud communication using the library was initially performed in Ref. [60]. The "device simulator" is capable of mocking real-world scenarios, including different sensor types, locations and authentication methods.

The azure service provides functions for connecting to Azure, verifying the messages transmitting data. It is a multi-step process enforced by IoT Central, shown in figure 4.1. Firstly, the device performs an HTTP PUT request on Azure Device Provisioning Service (DPS) to register the device and get the URL of the hub that sits within IoT Central. With this information, the client can then create an MQTT connection to the IoT Central hub, and transmit the data as JSON.

The IOTC library is technically cross-platform, meaning it can run on both micropython and regular CPython. However, we encountered a bug in the library that caused micropython code to always fail to authenticate with Azure. The issue, after extensive debugging [61], was found to be an implementation difference between micropython and CPython with the unix time epoch, causing the time-based authentication token generation to be different. Micropython's epoch is `2000-01-01 00:00:00 UTC`, whilst CPython uses the Unix time epoch of `1970-01-01 00:00:00 UTC`. We made a modification to the library that offsets the time when generating the authentication token by 30 years, fixing the issue.

4.1.3 Configure Mode - Web Server Implementation

One of the requirements for configure mode was to have it cancel with a physical button press, or a timeout in case the user forgets. Cancellation via button is performed by a hardware interrupt that cancels the asyncio webserver task and closes the event loop. To cancel via timeout, a modification to the webserver was required. This was implemented by a subclass of the Tinyweb server which contains a Watchdog timer [62], and overrides the server's `_handler()` method - responsible for handling incoming requests. When a request is handled, the overridden method resets ("pats") the watchdog, and then passes control to the original parent method for actual handling. If the watchdog times out, a callback method for shutting down the server is called. Implementing the webserver this way meant that the original library code is preserved, and the modifications are visible at the "service" level.

The webserver implementation is built on the idea of *Resources* - arbitrary Python classes with methods defined for different HTTP request types, representing a logically defined object with it's own state and operations. This is a fundamental aspect of RESTful design [63]. Whilst tinyweb supports python generators [64], Micropython doesn't distinguish between generators and asyncio coroutines [57]. We implemented a modification to the library to support coroutines in resources, discussed in-detail in Ref. [65].

At the top level, resources are mapped to a URL (section 3.2.3). However, the implementation of the mapping differed from the design - TinyWeb doesn't support nested endpoints, so the dynamic part of a URI must always be at the end. For example, the resource URI `/config/sdi12/<sensor_name>/test` (where `<sensor_name>` is dynamic) would be invalid. To get around this, the API was modified to place the action - "update, rename, test" etc - before the dynamic part.

Each resource that accepts user data needs to perform validation to ensure the configuration values are set correctly. To do this, a generic validation service was written that could validate any arbitrary data, modelled on the validation functions of the Django webserver framework [66]. The service, demonstrated in Ref. [67], is capable of recursively validating nested JSON inputs, and is used extensively across the codebase to ensure data consistency.

To facilitate concurrent development of the webserver, cloud software and Web App, a "webserver simulator" was developed [60]. Built on Flask [68], the script simulates the device webserver as closely as possible with identical API endpoints. When developing the Web App, the URL of the backend can be set to the webserver simulator running on the same machine. This greatly improved developer efficiency because changes are much quicker to make on the Flask webserver. It also allowed the use of user acceptance testing (section 4.4.3).

4.1.4 Device Configuration Service and Schema

The device has a number of configuration values that are set by the user - sensor calibration values, the device ID/name, etc, all need to be stored in the SD card to be retained on each wakeup.

The device configuration is stored in a JSON file, and accessed through the *config service* [67], a service responsible for managing access to the underlying data-structure, providing functions to read and update data. JSON was chosen because it is a human-readable data storage format, making it able to be modified if the user wanted to change the configuration manually on the SD card. Furthermore, it is very good at storing nested data, making it the ideal choice for the type of configuration data stored, and it has good library support within Micropython.

The configuration file format is shown in Ref. [69]. Device-level settings are stored in the top-level properties, whilst sensor settings are nested in the 'sdi12-sensors' property as a dictionary, mapping from the sensor name to an object describing its properties. Doing it this way allows the code to easily and quickly look up individual sensors, rather than having to search through a list. It also reflects the structure of the REST API more closely, leading to less cognitive overhead in understanding the codebase.

4.1.5 Deep Sleep Scheduler Service Implementation

The device's main function is taking periodic sensor measurements at regular intervals (for example, every half an hour), and then going to low-powered deep sleep when not in active use, as shown in figure 4.2.

One major problem with the device software is that it only runs while the device is awake. Furthermore, putting the device to sleep is effectively a complete shutdown - no code can run, aside from a timer that wakes the device back up when it expires. The wakeup also causes a complete fresh restart, so the device cannot be resumed mid operation.

To address this problem, we implemented what we call the *"deep sleep scheduler"* service, responsible for calculating the time the device should go to sleep for - the `$SLEEP_TIME` variable in figure 4.2. Each device has a "record" and "send" interval, which is the respective amount of time to wait between taking a recording and sending the data. The send interval is always a multiple of the record interval to minimise the amount of time the device needs to be awake.

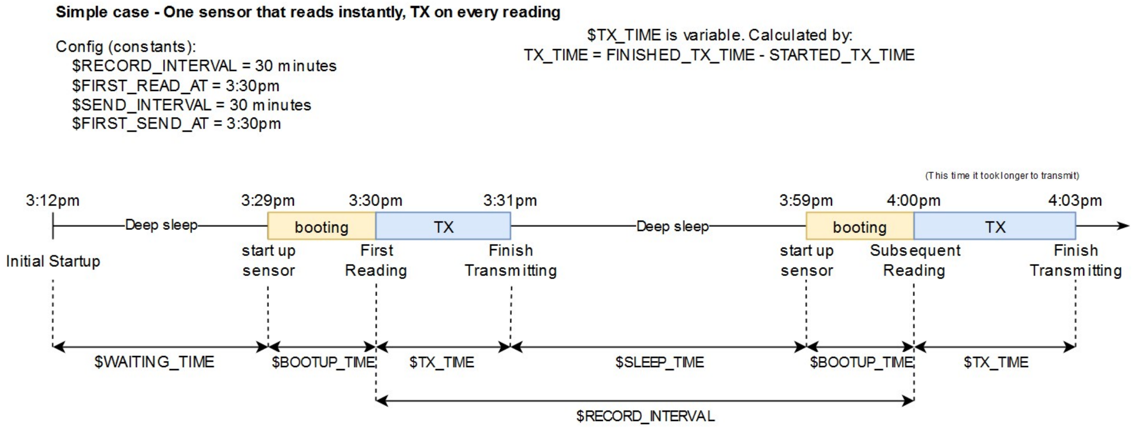


Figure 4.2: Variables involved with calculating the time to be in deep sleep for.

The devices have a significant bootup-time, and so does each SDI-12 sensor. If the user wanted the device to record every half hour with a 10 minute offset - for example: *12:10pm, 12:40pm, 1:10pm etc*, an offset date and time can be used. This is implemented in a device config variable called `record_from`. If the variable is set to a time in the future, the sleep service allows the device to delay waking up for the first time until that time has been reached.

To do this, the sleep scheduler first get the smallest time in seconds to the next scheduled sensor recording. For each sensor, this is accomplished by getting the time difference between the current time and the recording time. If the difference is positive, the first recording hasn't happened yet, and so the scheduler just returns the difference value. If it is negative, the scheduler returns the difference *mod* the recording interval time, which is the time *remaining* until the next reading.

The final sleep time is the smallest time out of all the sensors, minus the largest boot time of any sensor.

As long as the time taken for the device to boot is consistent, this design ensures that the schedule is always accurate because it adjusts as necessary just before going to sleep.

4.2 Web App Implementation

The Web App is the primary way of configuring the device and calibrating the sensor readings. This section discusses the architecture, build process and functional implementation of the frontend, built on the Preact framework [70].

4.2.1 Web App Framework Comparison

The Web App could be implemented without using an existing framework, or through the use of a web framework such as React or Angular [71] which reduce overhead by automating common activities and providing libraries for easing the development of dynamic websites.

The approach of not using an existing framework, investigated initially in Ref. [72], is beneficial because it gives us full control and knowledge over the code base, allowing us to create an optimised app with no unnecessary features. This would result in a tiny file size, an important aspect for maintaining low-latency with a resource-constrained webserver. The "classic way" of web development would consist of writing HTML, Javascript and CSS in separate files, and optionally *minifying* them (section 4.2.4) through a separate build step to produce a single artifact.

The downside of the manual approach is that dynamic rendering is harder to implement, and some complex features would take longer to implement. Libraries for asynchronous web requests - a requirement of the CSR approach discussed in section 3.2.3 - would need to be added to the webapp. Secondly, the CSR javascript code would need to modify the Document Object Model (DOM) directly. The DOM is a tree structure that represents the HTML file programmatically. Modifying the DOM directly is error prone and the complexity involved in direct DOM modification rapidly increases with app size [73].

Web app frameworks solve these downsides of the manual approach, and add more features which ease development significantly. The framework of choice for this project was Preact [70]. Preact is a "3KB alternative to React with the same modern API". React is one of the most popular web frameworks, with development supported by Facebook [71]. The compiled 3kb size means that we can get the page-loading benefits of the non-framework approach while still benefiting from the advanced features a modern web framework provides. Furthermore, unlike other frameworks, it was the only one I have had extensive experience with.

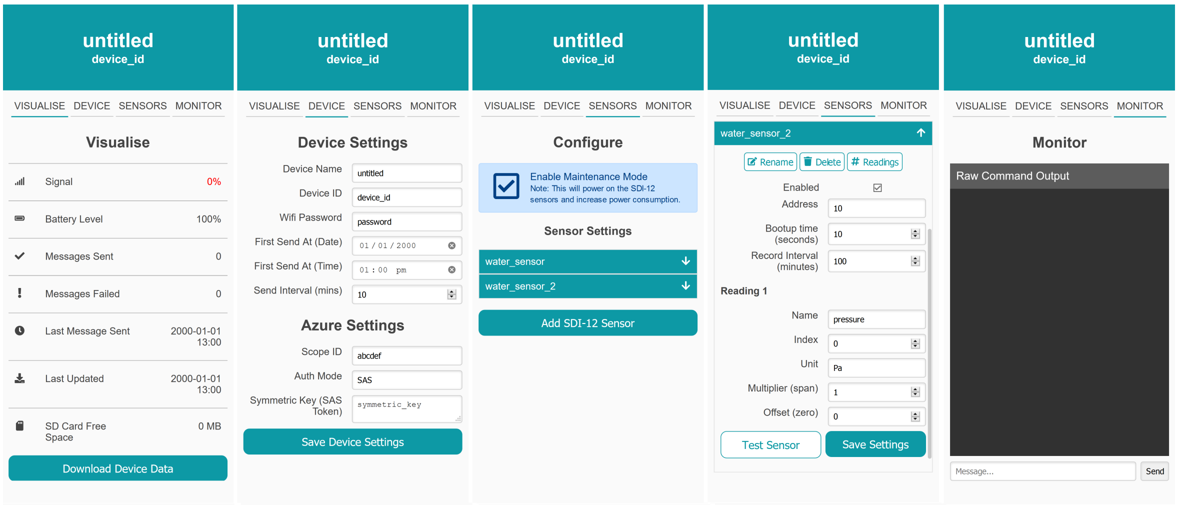
Preact is based on a virtual DOM, where an ideal, or virtual, representation of a UI is kept in memory and synced with the "real" DOM by a library such as ReactDOM [74]. The framework is designed around a component-based architecture, which allows the UI to be built from a collection of independent and reusable components. It has a modern and well documented API, which has allowed an abundance of low-size-footprint libraries for UI components to be created. Preact also bundles a series of highly-optimised tools for compressing and bundling production ready webpages, discussed in 4.2.4.

4.2.2 Web App Overview

At a high level, the *App* component is the Web App entrypoint and top-most component, being rendered first by the browser. Once the entire pre-rendered static files have been loaded by the client, the app makes requests to the backend API endpoints `/config` and `/data` . These endpoints return JSON objects which represent the state of the entire application. The *App* component passes the relevant information as "props" - read-only arbitrary inputs - to its child components. Each subcomponent contains code to render a UI component based on its internal state and its props (section 4.2.5).

4.2.3 User Interface Implementation

Screenshots of the final webapp implementation are shown in figure 4.2.3.



Signal 0%
Battery Level 100%
Messages Sent 0
Messages Failed 0
Last Message Sent 2000-01-01 13:00
Last Updated 2000-01-01 13:00
SD Card Free Space 0 MB

Device Name: untitled
Device ID: device_id
Wifi Password: password
First Send At (Date): 01 / 01 / 2000
First Send At (Time): 01 : 00 pm
Send Interval (mins): 10

Scope ID: abcdef
Auth Mode: SAS
Symmetric Key (SAS Token): symmetric_key

☑ Enable Maintenance Mode
Note: This will power on the SDI-12 sensors and increase power consumption.

water_sensor
water_sensor_2

water_sensor_2
Enabled ☑
Address: 10
Bootup time (seconds): 10
Record Interval (minutes): 100
Reading 1
Name: pressure
Index: 0
Unit: Pa
Multiplier (span): 1
Offset (zero): 0

Figure 4.3: Screens from Final Webapp. From left to right: Visualise screen, device settings screen, sensors screen, sensors screen with an open tab, and monitor screen.

As shown, the implementation closely matches the final design in section 3.3. The largest difference is the presentation of sensor readings as a list with a button to choose the reading count, rather than as a swipe-able carousel with an input box for reading count. This was for a couple of reasons - firstly, the design didn't scale well to large screens, and navigating the tabs with a mouse was difficult. Secondly, it was easier to implement, as it didn't require a javascript-based animated component.

Other differences include changes in colour and minor layout changes, made after customer feedback and manual usability testing on various devices.

4.2.4 Build Process

To turn the Web App source files and dependencies into a production-ready Web App running on the microcontroller, an optimised build process was configured.

The first stage in the build process checks for typescript errors. Typescript [75] is an extension to Javascript that adds static typing. We explicitly enforce strict static typing for the entire project - every variable needs an explicitly defined type (or infers it). For example, a portion of the global `interfaces.ts` file defines the types associated with the data returned by the backend API, shown in the following listing:

```
export type SDISensorType = { address: string; bootup_time: number; enabled: boolean }
const sensor: SDISensorType = { address: 'A', bootup_time: 1, enabled: true }
```

In this example, the type `SDISensorType` concretely implements part of the configuration defined in section 4.1.4. This allows the programmer to create a new object, *sensor*, that implements this interface. Defining this type enforces the primitive types (strings, numbers, booleans etc) of the properties of a sensor object, leading to less type errors, more resilient code and reducing development time.

The second stage *transpiles* React code into native Javascript (section 4.2.5). Transpiling is the process performed by the library *Babel* [76] that allows modern code to be backwards compatible and support all browsers. This process also involves injecting environment variables [77] into the code. Environment variables are used by the developer and Continuous Integration process (section 4.4) to differentiate between a development and production build. In a development build, the Web App connects to a webserver simulator backend [60], whilst a production build uses the webserver on the microcontroller. Environment variables enable this switch between contexts.

The third stage, performed by the *WebPack* library [78], minifies and bundles the transpiled code into HTML, CSS and Javascript files. Minification is a process that removes unnecessary or redundant data from the source code without changing the functionality - things like shortening variable names, removing formatting and comments, etc. A further optimisation we perform on top of the webpack process is to compress the bundled files with *gzip*. This had the effect of reducing the total file-size by around 50%.

4.2.5 Implementation of Architecture using Functional Components

The structure of each component in the Web App follows a functional paradigm, where components are composed from literal javascript functions, for example:

```
const component: FunctionalComponent = (props: PropType) => {
    return <div>I am a component named {props.name}!</div>
}
```

This design allows for strong composability and makes it easy to reason about the state of the component. An interesting note is that the code above is not valid javascript - the return statement returns HTML directly. This is a react-specific syntactic extension of javascript called JSX, that combines javascript and HTML into the same file [79]. Because the browser does not natively understand JSX, part of the compilation process "transpiles" this code into browser-readable javascript, as discussed in section 4.2.4.

The ideal functional component is *"pure"*, containing no side effects or mutable state. However, in a dynamic Web Application the state changes often - buttons are pressed, dropdowns toggled, and forms are modified. This is accomplished in functional components with React Hooks [80], a feature of the React framework. Hooks are a new feature of the library that offer ways to "attach" reusable behaviour to a component.

One such hook is the `useState()` hook. These hooks allow reuse of state logic without changing the component hierarchy, letting components change their output over time in response to user actions, network responses and anything else without violating the functional rule. For example, the *Monitor* component needs to track mutable state to store whether or not it can send commands to the device. This can be stored in the component with `const [canSend, setCanSend] = React.useState(true);` In this line, `useState(true)` declares a "state variable" with a default value of `true`. The function returns a pair of variables - `canSend` represents the state, and `setCanSend` is a reference to a callback function used to mutate the state.

This pattern is used throughout the application almost every component. It has allowed for simpler code that is easier to reason about and debug, and the functional design matches the design decisions of the device software.

4.2.6 Handling "Globally" shared resources

One problem faced with the architecture of the Web Application is globally-accessible state. Most of the dynamic data in the application comes from the top layer *App* component, which calls the backend API and passes the device data as props to subcomponents. This relation is one-way, and props are read-only. What happens if a sub-component, such as one that updates the device configuration, wants to update the top-level config object, so that the page doesn't have to be refreshed every time the configuration changes? To ensure two-way communication back to the top component, the top-layer *update API* method would need to be passed down to every component through props, leading to the well known and hard-to-maintain "callback hell" [81].

To get around this problem, react provides another type of hook - `useContext()`. A *context* is an object with two values - a Provider and a Consumer, created with a library function `createContext()`. the `useContext()` hook takes a context as an argument and returns the *context value* for that context. The value is determined by a specific prop - `value` - from the first Provider that sits above the the calling component in the tree.

In our app, we wrap the top layer `App` component in a context Provider, passing the `fetchConfig()` method - the method that retrieves the latest config from the webserver - into the provider as the *value* argument. Subcomponents that want to use the context now only have to call `useContext(fetchApiContext)`, which returns the callback function that updates the configuration data. Now, every sub-component can be responsible for refreshing the applications global state.

4.2.7 Implementation of User-feedback Notifications

The library *Notyf* [82] was used to implement notifications that display informational, success and error messages to the user. Notifications were implemented to provide a standard way of giving feedback to actions that hit the backend API in a centralised place. Furthermore, because API calls often took multiple seconds to run on the microcontroller, a message-based system helps assure the user that *something* is happening.

Global notifications were implemented using the same `useContext()` hooks discussed earlier. An interesting issue was that the library caused the compiler to throw errors in the production build process, halting the compilation. This is because the library relies on the DOM, which is only available in the browser. Pre-rendering runs in Node [83], which has no access to browser APIs. To get around this, a mock Notyf context was implemented that overrides the notification methods with empty stubs. A higher-order function was implemented to wrap the `useContext()` method, returning the actual Notyf context if the DOM is accessible, or the mock context otherwise.

4.3 Cloud Platform Implementation

Many aspects of the cloud implementation was straightforward due to Azure IoT Central's Software-as-a-service model - telemetry alerting, data analytics and scheduled data exports are built-in to the software package. This brief section outlines the manual work required to set up the cloud systems, including the long-term cold data exports and how we enforce data integrity in the cloud database.

4.3.1 Long-term Data storage through Azure Blob Storage

Azure has a built-in periodic data export functionality which can export directly to Azure Blob Storage [84]. This involved setting up an Azure Storage Account [85], and connecting the services with a shared authentication token as described in Ref. [86].

Azure Blob Storage stores the data as JSON, organised into sub-folders separated by day. This makes it easy to search through the long-term data, and is machine-parsable so it would be easy to integrate into GWRC's databases in the future.

4.3.2 Device Capability Models for Ensuring data integrity

IoT Central enforces incoming telemetry packets to be associated with a known device, which in turn must implement a Device Capability Model (DCM). A DCM is an interface that strictly defines the telemetry names (literally the keys in the incoming JSON-formatted data), units, datatypes and bounds, called ”capabilities”. For example, the water temperature capability sets a name - `WaterTemperature`, a unit - `m/s`, and discards any data that doesn't match a numeric datatype.

Upon consultation with the client, we defined a DCM based on all the types of telemetry GWRC collect [87], and set it as the default device template for newly-provisioned IoT Devices.

4.4 Development Practises

To work effectively in a team environment, numerous DevOps practises [88] were employed to facilitate strong collaboration between team-members, catch issues early and ensure timely delivery of features. This includes Continuous Integration (CI), which automates the integration of changes so that bugs are detected early, constant availability is guaranteed, and last-minute integration chaos is avoided. This is predominantly achieved through a strong suite of automated tests.

Readers are strongly encouraged to refer to Ref. [89], which outlines and evaluates the hardware and software DevOps practises employed in this project, such as the Continuous Integration and Deployment pipeline.

4.4.1 Embedded Unit Testing

Testing is primarily performed through *embedded unit testing*. Having confidence in good unit and system tests increases the confidence that developers are writing good, maintainable, bug-free code. Embedded Testing is performed using a custom GitLab Runner on a Raspberry Pi connected to one of our IoT devices over USB. A script was written that compiles and uploads the latest code to the device, runs the embedded tests included in the compiled code, and returns an exit code based on the results. The script uses various tools to facilitate this process, including *rshell* [90] for uploading the code, and *pyboard* [91] to run the unit tests.

The codebase contains over 90 unit tests, testing both high-level software components and low-level software drivers for the hardware. The unit testing rig ran reliably for over 3 months with nearly 100% uptime.

On the device, the micropython library *micropython-unittest-xml* [92] is responsible for running the tests. It is mostly compliant with the Python *unittest* [93] library, but some small modifications had to be made to the library to support running an arbitrary number of test suites within a subfolder.

4.4.2 Integration Tests

The biggest potential limitation to the existing test suite is that while it validates that individual parts of the program are correct, it fails to test the interfaces between different software and hardware components. This missing step is known as Integration Testing [94]. While a unit test might confirm the functionality of a software module such as a sensor driver (using mock sensor data), it does not test the interaction between the driver and the sensor itself, or the driver and the higher-level code that uses the driver. As a result, we implemented

"Bottom-up" Integration Testing - an approach that uses the bottom-layer components to facilitate testing of higher-level components, repeated until the components at the top of the hierarchy are tested.

Integration tests are built using the same library as the unit tests, but reside in their own subfolder in the device code. Currently, because they are quick to run, they are run on every commit, but the test runner script is flexible enough that the integration tests could be run on a schedule or on the master branch only.

4.4.3 User Acceptance Testing

User Acceptance Testing is performed from a manual deployment step in the CI pipeline. This step builds a production-ready version of the front-end Web App and automatically deploys it to Azure Web Apps [95]. The backend is mocked by the device simulator described in section 4.1.3. The use of this stage is so that the customer, functioning as the "user", can quickly look at changes made to verify it satisfies their business objectives. This step is important because it effectively provides a manual frontend testing framework to verify the usability of the Web App.

Chapter 5

Evaluation

As a proof-of-concept, the project demonstrates an end-to-end system that is able to justify the business case of a custom IoT-based environmental monitoring solution for GWRC. This section evaluates the effectiveness of the different software parts of the project, discussing why certain decisions were made and their effect on the quality of the final artifact.

5.1 Device Software Evaluation

The device software, running Micropython, used a mixture of Functional and Object Oriented programming patterns, leading to reliable and performant code. Despite the hardware issues experienced with the modems, communication over WiFi was adequate for this proof of concept, and the library that enabled this was fit-for-purpose despite being deprecated.

5.1.1 Use of Micropython As The Embedded Programming Language

Micropython was chosen early on in the project for it's ability to allow rapid iteration through strong readability, writability and expressivity, which held true through the project's development phase. The exception handling system ensured we can protect against unforeseen errors with the sensors, and the *Asyncio* library made asynchronous programming easy and reliable. Being mostly compatible with Python, cross-platform compatible libraries like *IOTC* made it possible to create device simulators and test scripts to validate functionality of different software components before they needed to be tested on the device. Furthermore, the interpreted nature of python meant that a complex pre-compilation build process could be avoided, and tests could be done directly in the device's serial terminal.

Whilst Micropython does not have the maturity of C or the same community size as Arduino, it's user-base is growing rapidly and is being actively maintained and developed [1]. One implication of this is that documentation is sometimes severely lacking, even for standard library code - however, Python's strong readability aspects made it possible to work around these issues.

Another limitation we found of the programming language was that it was difficult to manage memory, which has implications for future hardware iterations that plan on using a lower-memory microcontroller. Furthermore, while the (relatively) slow processing speed did not impact the current device drivers, future hardware components may require precise timing, memory management and IO throughput that micropython might struggle to provide without multiple significant optimisations.

5.1.2 Device Software Architecture and Programming Paradigms

Despite being proof of concept, the software architecture is extensible, reliable and efficient, and can be used for future development without any major restructuring. The tiered-based architecture defines a solid split between low-level drivers and high-level services, making it easy to develop and test the different components in isolation.

The decision to split the high-level code into *Regular Mode* and *Configure Mode* came from the fact that the device needs to initiate a full reboot every time it wakes from deep sleep, losing any state stored in RAM. Seeing the "wake reason" of the device allowed us to design the modes around being asleep for as long as possible. By having explicit logging of the current mode, the customer could reason about the state of the device at any time and act accordingly.

One limitation of the mutually exclusive modes is that if the device is in *Configure Mode* and a recording is scheduled to happen at that time, the data-record-transmit pipeline (section 4.3) won't be run. Future iterations could get around this by queueing readings, and then immediately switching to *Regular Mode* after configuration. The client's evaluation was that this issue could be addressed by their ensuring they would only perform configuration between recordings.

The biggest point of contention with the architecture was the mixture of programming paradigms - *Regular Mode* is designed in a functional manner, whilst *Configure Mode* follows an Object Oriented Paradigm. Functional programming forces the programmer to think about code as a series of *pure functions* - containing no side effects (modifying state outside of the function's local environment) and is deterministic (always returning the same result given a certain set of inputs). Another aspect is immutability, where variables are never modified after creation. The main benefit of this is that functions are easy to test and debug, and it is possible to reason about the result of a function if you know the inputs. A functional design for *Regular Mode* makes sense conceptually, because it is a literal pipeline of functions that put the sensor data through a series of steps to record, combine, validate and transmit telemetry to the backend. *Regular Mode* can be run at any time without resetting the state of the device because the functional approach decouples *state* from *operations*.

The push for a functional design in regular mode caused all of the low-level driver code to be functional in design as well, as discussed in Ref. [5]. Rather than initialising a singleton object for each driver, idempotent functions act on the hardware interfaces and return values, rather than modifying some external state, which made it particularly useful for unit test implementation.

Despite *Regular Mode*'s functional design, the webserver depends on the TinyWeb library, which is Object Oriented. That is, the functionality depends on the state of a *webserver* object, which listens for incoming web requests and acts based on them. This model works for the webserver because the REST API is built on the idea of "resources" (section 4.1.3), which are by nature object-oriented. However, we built the functions within each resource to be as functional as possible to allow independent testing of resources without needing to initialise the webserver, demonstrated in Ref. [96].

The two programming paradigms are used at different times (separated by the device mode), but both share the low-level driver code. While this works fine for *Regular Mode*, it added additional complexity to *Configure Mode*, requiring passing around the driver values as objects and making heavy use of callback methods. This complexity adds a mental overhead to developers to switch between ways of thinking about the code. While it didn't become a serious problem in this state of the project, future iterations should find ways of potentially modifying the webserver library to be more functionally-oriented.

5.1.3 Reliability and Code Quality

Software reliability is directly correlated to Code Quality [97]. Quality is baked in from the start in our project through a strong testing suite that runs directly on the hardware, testing everything from low-level drivers to the high level services and the interactions between them. Making testing a primary part of the continuous integration pipeline reduces the chance of bugs and defects entering "production". For example, in Ref. [98], a bug was found where the first sensor readings after a power cycle were being dropped by the SDI-12 driver. Because the development boards remained on all the time, we never saw an issue locally. The embedded testing rig performs a full reset, so the error was caught and appropriately notified.

Furthermore, the use of unit tests meant that regression errors could be captured and prevented from appearing. If a defect is found, a unit test can be made that explicitly tests for the absence of the defect after it has been fixed. An example of this is shown in Ref. [99], where regression tests were added after a bug was found that the logging library was incorrectly raising exceptions.

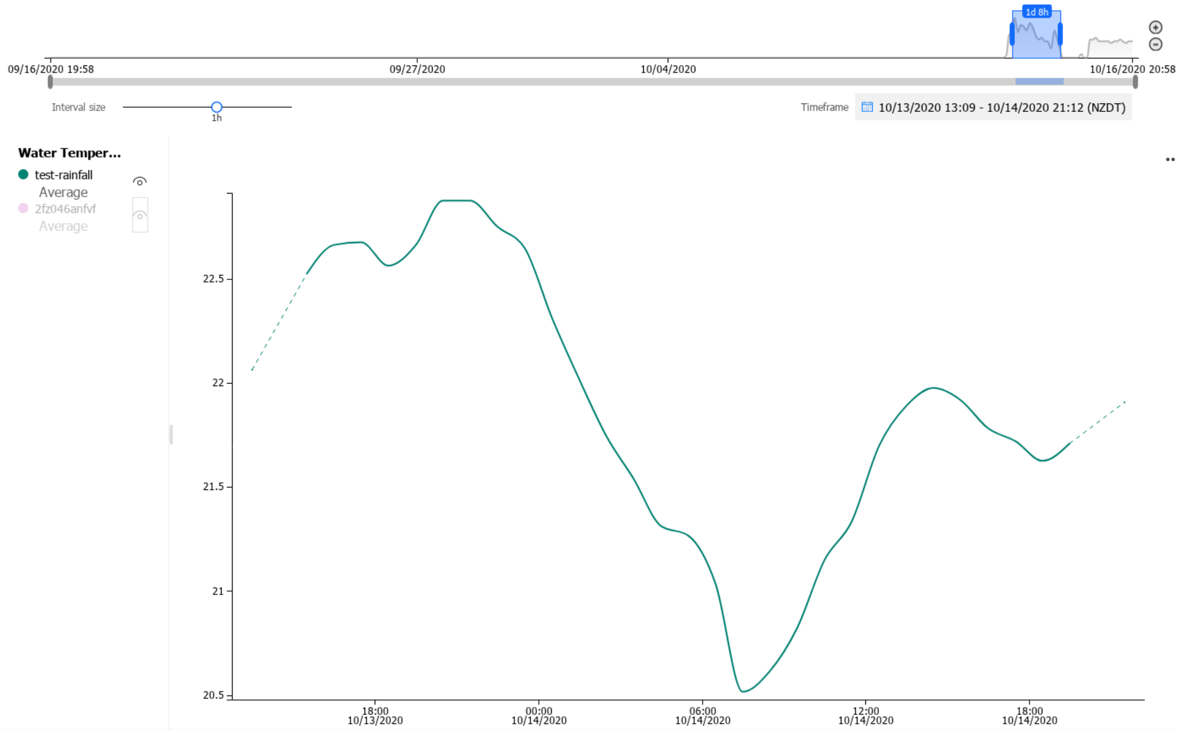
Embedded testing directly on the device has numerous improvements over running unit tests solely in software on the developer's machines. Often, embedded software can be compiled and run inside an interpreter or virtual machine (VM) on the developer's machine, but with this comes major limitations. While the VM is able to test high- level software processes, it is unable to test low-level driver code, such as code that interacts with the underlying hardware. The VM does not have access to true Analog-to-Digital converters, digital pins and serial UART ports, which are important components of the device. Testing directly on the hardware solves this problem. The exact same type of device is developed on and tested, so any failures in CI (or even when a device is deployed in the field) can usually be reproduced and fixed. This enables a high sense of confidence when writing low-level code.

Data reliability is enforced using the validation service, which is used extensively where configuration data is modified, such as the REST API resources. Unit tests pass different types of invalid data to the validation service, which catches the errors and throws user-readable exceptions. Exception handling is done in all parts of the codebase. Backed up by a good logging suite that logs to the terminal and SD card [5], hardware, configuration and runtime errors are appropriately caught and handled.

Furthermore, the code quality is enforced by a linter that validates code style, and merge request reviews which ensure all team members see the code being written.

To evaluate the long-term reliability of the device software, a test rig was set up in a University laboratory environment connected to a benchtop power supply and dedicated WiFi hotspot. The device was configured to transmit on a quick, 10 minute interval, and to record the ambient room temperature using a connected SDI-12 sensor. After running continuously for 30 hours, the sensor readings were graphed in azure, shown in figure 5.1. The logs on the device's SD card showed that the device consistently woke up at the correct time, took a reading and transmitted the data. Of the 180 readings sent, 15 packets were dropped due to a mildly unstable connection to the Universities' WiFi.

This experiment was reproduced by the client in their own laboratory setting. The telemetry data was recorded and sent reliably, with no dropped packets. From the experiments, we could determine that the device software is ready for testing in the field. While this wasn't able to be done because of a power-draw-related hardware issue, discussed in Ref. [5], the lab testing of the software proves that it is stable and reliable enough to be deployed. This experiment also verified the stability and accuracy of the deep sleep scheduler - the device woke up on average every 10 minutes, with an average error of ± 1 second.



Interval size 1h

Timeframe 10/13/2020 13:09 - 10/14/2020 21:12 (NZDT)

Figure 5.1: Temperature readings from a continuous 30 hour test in a laboratory setting.

The biggest limitation is that if a connection isn't made on boot-up to the WiFi hotspot, or the hotspot has a limited internet connection, the data won't be transmitted. However, the data is still logged to the SD card, and the device tries to re-send data if the MQTT publish step fails.

5.1.4 Cloud communication through Hotspot and IoT Central Library

While the WiFi-based workaround discussed in section 4.3 is inappropriate going forward in the project, the IoT Central communication library *IOTC* functions as a reliable replacement to make up for the lack of modem in the short term.

With a reliable source of power and network connectivity, the WiFi-based approach allows the device to be placed remotely by setting up a portable WiFi hotspot on a mobile phone. This circumvents the need for a modem, but it could mean supplying each IoT Device with a dedicated hotspot, which is costly and excessive. However, it was more than enough to satisfy the remote transmission requirements for a proof of concept project.

Regardless of the state of the hardware, the Azure service we created that builds on the IOTC library functions reliably, as shown in the long-term performance test in the previous section. Despite the observed reliability, the library is already deprecated as of version 0.3.9, and the maintainers have removed support for micropython from version 1.0.0 onwards, forcing us to remain on 0.3.9 until support is added. After asking the maintainers for an indication when this was going to happen, we received a non-committal "in the next days" [100]. We do not expect the IoT Central API to change significantly in the near future so it is likely use of this library will be able to continue in-stasis until a new version is released without any major issues.

One limitation of the library is that it's implementation is all in one file, so when the library is imported, it takes a significant amount of time for the ESP32 to compile the code. There is also a lot of extra code for unused functionality that could have been pruned from the library to reduce compilation time.

Going with IOTC in its current state was a good choice, especially considering we couldn't find any alternative libraries for IoT Central. Future development depends on the way mobile communication through the modem is implemented. If the modem is implemented in a way that transparently abstracts the modem driver as a network interface[5], the library should still be used, and updated as soon as a new release is made.

5.2 Web App Evaluation

Through the User Acceptance Testing process outlined in section 4.4.3, the Web App was continually evaluated throughout it's development by the client. This meant that rather than leaving a single large evaluation to the end of the project, the client was kept in the loop as new features were developed. An example of this is shown in Ref. [101], where the client gave feedback on the design of the visualisation screen.

Overall, the client was happy and satisfied with the design and functionality of the Web App. With a few exceptions, all of the main requirements were met, and the app is capable of fully configuring the device. This section briefly outlines some specific feedback, and also evaluates the Web App from a development perspective.

5.2.1 Client Evaluation of Web App Functionality

Towards the end of the project, we had a discussion with the client about their experience using the Web App, and asked them questions about the intuitiveness and usability of the software.

Intuitiveness The client found the app intuitive from the start, with an easy to follow layout and sensible delineation of information in the four different tabs. They set up the device for the first time without following our documented instructions, and found they had done it completely correctly. However, they did need to guess that the Symmetric Key field in the Web app referred to what Azure called "Primary Key".

Likewise, the Sensor set up was clear and easy to understand. The monitor tab worked without fault and they were able to send and receive manual SDI-12 commands seamlessly.

Usability The client reported that they were able to set up their connected SDI-12 sensor in no more than 20 seconds for the first parameters. Moreover, they made no mistakes in the setup process, which indicates that the intuitive design guides the user to make correct decisions for configuration.

When comparing our solution to their existing system - HydroTel [40] - they said the existing software has to deal with more complex devices so there are a number of unused settings that take up screen real estate, whereas our Web App is cluttered and compact. Furthermore, their software only runs on Windows based machines, so they were impressed by the fact that the Web App can run on any device, and the responsive design ensures it can be used at any screen size.

5.2.2 Web App Size and Implementation

One important aspect, as discussed in section 3.3, was that the Web App had to load in a quick enough time to not negatively impact user experience. Although the performance is mostly governed by the microcontroller, the size and complexity of the Web App needed to remain small to load quickly. The client was happy with the app loading under thirty seconds, but knew that the time could be longer due to the webpage loading from a low-powered embedded device. The most recent production Web App size was 65 KB in total, including layout, styling and code. To test the performance, we measured the time taken for various actions to occur in *Configure Mode*, as well as the time taken with caching enabled on a web browser. The results are shown in table 5.1.

Activity	**Time (s)**	**Time (with caching) (s)**
Loading Web App	25.1	15.36
Update Device Config	4.48	3.84
Send Raw SDI-12 Command	3.2	3.2
Request static file	3.5	0.1

Table 5.1: Loading time of various user activities

Caching, which is enabled by default, greatly increases loading speed of loading the Web App and static files. All of the results - most importantly the Web App loading times - were under the thirty-second bar, so the app exceeds the client's performance requirements.

The biggest limitation to the Web App functionality was the lack of automated unit testing on the frontend, so bugs and regression errors were found downstream in development. Whilst a manual CI step was used in merge requests to visually check for bugs, the occasional subtle bug was missed which meant that more time had to be spent later on.

Development of the Web App using Preact was a good decision. The use of functional components, hooks and typescript follows state-of-the-art web development standards [71], and led to readable, extensible and modular code. Despite the additional size from the framework, libraries and components, the optimised build process (section 4.2.4) meant that pages loaded quickly.

5.3 Cloud Software Evaluation

Whilst preliminary research and early testing suggested that Azure IoT Central was an effective cloud platform for this project, the specific details of the SaaS-oriented platform meant that fitness-for-purpose in the long term might not be feasible. This section evaluates IoT Central from both an end-user and development perspective, discussing the connection process, ease of use and architecture considerations.

5.3.1 High-Overhead Connection Process, Device Setup and Provisioning

At it's core, IoT Central is a user-friendly interface on top of a collection of Azure IoT services such as IoT Hub and Device Provisioning Service (DPS). For devices to connect to IoT Central, they need to go through a two-step process as outlined in section 4.1.2, which involves registering with DPS to get the IoT Hub URL, and then connecting to the MQTT broker in the Hub. This design allows for high scalability and gives Microsoft flexibility to provide a consistent method for users to connect devices. However, this connection process adds a high overhead to low-powered devices. The NB-IoT protocols low maximum bandwidth is adequate for sending very small amounts of data (section 2.2), which is why MQTT is used in this project for it's low packet overhead (section 2.3).

To test this, we used the linux program Nethogs [102] to measure the total throughput of data on the device simulator (which, aside from the data payload, behaves exactly the same as the real device). The payload was set to the following, a 43 byte JSON encoded message:

```
{"Turbidity": 8.1231231, "Flow": 0.100203}
```

Using the IOTC library, the total throughput measured by Nethogs was 6782 Bytes sent, and 16156 Bytes received. By using a debugger, we were able to break down the total throughput by action in table 5.2:

Action	Bytes Transmitted	Bytes Received
DPS Register (HTTP PUT)	2144	5273
DPS Get IOTC URL (HTTP GET)	1919	5342
MQTT Create	1762	4669
MQTT Subscribe	651	615
MQTT Set Device Settings	193	203
MQTT Publish	113	54
TOTAL	6782	16156

Table 5.2: IoT Central connection process broken down into bytes transmitted and received by IOTC library

In total, the IOTC library transmits 6.7 KB and receives 16.1 KB to send a single 43 Byte message. Analysis using *Wireshark* [103] showed a significant part of this overhead comes from TLS, which appeared to be re-initialising the TLS connection between requests and validating the server certificate, which is a high-overhead activity [104].Aside from the TLS overhead, it is evident that the DPS connection and registration process is costly.

As well as increasing the time taken to transmit such a high amount of data for a single message, NB-IoT Networks like Vodafone's also have strict usage limits (their lowest-cost plan is $3 a month for 1 MB of data). With lots of devices transmitting regularly, the data overhead for connecting to Azure IoT Central could simply make telemetry transmission too costly.

Furthermore, IoT Central only supports up to MQTT QoS Level 1 (at least once), guaranteeing that a message is delivered at least one time to the receiver. If, in the future, we added support to *Regular Mode*to retry messages if they appeared to not send correctly, the MQTT broker in IoT Central's IoT Hub wouldn't support checking duplicate messages.

In the project's current state, the devices are secured using symmetric key authentication. We found this to be reliable and easy to manage, with the only complexity being that the user needs to transfer the symmetric key to the device by entering it in the Web App.

5.3.2 IoT Central Ease-of-use and Analytics

IoT Central provides an easy-to-understand user interface that the Client could understand without any explicit training from us. It provides intuitive forms for creating new devices, displaying raw data, and visualising the data through graphs. It also has a tab for "analytics" which shows a large single graph and allows the user to dynamically modify the data, time range and interval size. We carried out an analysis in Ref. [105] to verify if the analytics tools IoT Central provided were relevant and useful for the client. The analysis found it was mostly *good enough*, but had a few crucial missing components: identifying data gaps, and performing in-depth analysis of trends and outliers. Furthermore, it was limited to the last 90 days of data - which for GWRC is too short of a time-frame for long-term analytics.

Going foward, IoT Central has a ”data export” feature which allows data to be exported into Microsoft’s dedicated Business Intelligence suite, PowerBI [106]. PowerBI is an exhaustive software suite for providing analytics, reporting and dashboards for any type of real-time time-series data. It is fairly expensive to run - over $100 per month - but its inclusion would mean that GWRC would have a fully-featured solution to get insights on their environmental data.

5.3.3 SaaS based Cloud Architecture

Overall, the SaaS-based approach was effective for this proof-of-concept. IoT Central simplified the management overhead of lots of different Azure services. The downside, as discussed above, was the necessity to conform to it’s particular way of connecting devices as well as ensuring data conformed to the device templates. This essentially enforced a strong coupling between the device software and Azure, making it more difficult to move away from Azure if it became too expensive or unsuitable for GWRC.

Going forward, the platform-as-a-service option would enable a greater degree of flexibility in the design of the cloud backend. A Microsoft Cloud Architect even suggested we move away from IoT Central for production for our project, instead making use of Microsoft’s “Solution accelerators” - essentially custom scripts for quickly bootstrapping a PaaS based infrastructure solution.

Chapter 6

Conclusions

6.1 Conclusion

Overall, the proof-of-concept sufficiently demonstrates a working end-to-end environmental monitoring system which is reliable, scalable and easy to use.

The high-level micropython-based device software coordinates the low-level hardware drivers, reads sensor data and manages configuration data through a webserver. The Web App facilitates visualisation, updating the device configuration and calibration of sensors. In the cloud, Azure IoT Central handles the incoming device telemetry and displays the environmental data in easy-to-read graphs.

The success of this proof-of-concept raises the following conclusions:

- Micropython, although being relatively new and having a small community, is suitable for low-level embedded development for IoT applications. It's expressivity and readability allows for implementation of complex application logic in an understandable way, enabling rapid development of new features for a proof-of-concept. Despite this, it is highly reliable and robust, and can be optimised for performance when needed.

- Azure offer a range of IoT-focussed services suiting a wide number of use cases. IoT Central, their SaaS offering, is good for initial development, but is lacking features that make it feasible long term, meaning a shift to a PaaS solution would be required in the future. Overall, a cloud based backend system meets the long-term scalability, flexibility and interoperability goals for the client. It is secure enough for GWRC's needs and integrates with their existing infrastructure.

- The design of an IoT based monitoring system is coupled to the availability of the wireless networks. As shown with the modem hardware issues [5], the network is a critical component of this solution. Because of this, IoT devices need fault-tolerance and fallback measures to ensure important environmental data doesn't get missed.

- A custom-developed solution for the client, with the intention of open sourcing the work, has led to a highly-efficient and specialised product which satisfies the main requirements for replacing their existing systems. In comparison to the vendor-provided Hydrotel [40] software, which adds bloat through unused features, a custom solution promotes a lean workflow tailored to GWRC's specific use-cases. Furthermore, the open-source nature of the project means that other regional councils can modify the software and hardware for their own uses.

6.2 Future Work

As mentioned in chapter 5, Additional work is required to move from a proof-of-concept to a production-ready system.

6.2.1 Device Software

The device software is mostly feature-complete, but still needs additional reliability measures built in to improve fault-tolerance and resiliency in different operation conditions. In *Regular Mode*, telemetry readings should be batched up in a queue, and transmitted on a different time interval to the recording interval. This would reduce power usage because the modem wouldn't need to be booted every time the device wakes. Furthermore, the devices should resend messages if they failed to be received. This is partially reliant on the QoS support by the cloud backend.

Another improvement could be to improve *Configure Mode* so that if a scheduled reading happens when the device is being configured, the telemetry reading process is queued up and executed after the user has finished setting up the device.

As discussed in the requirements document [6], a stretch goal is to support GWRC's *Rainfall Sensors* [5]. Because these sensors have shorter required recording intervals, a restructuring of the device software would be required to reduce the boot time and minimise power drain from the more frequent device wake-ups.

Another stretch goal is to support Over-The-Air (OTA) updates, where new firmware is sent by the cloud to the device, which is able to automatically deploy the new firmware without human interaction. This is particularly useful when the devices are deployed at scale - because they are often deployed to remote locations, going around to each device to do a manual upgrade is an expensive and time-consuming process.

Some development process upgrades would also be beneficial. With a plan documented in Ref. [107], Continuous Delivery (CD) [88] of device firmware would be an improvement over the manual process of *pulling latest changes, rebuilding the code and uploading changes to the device*. This could involve generating "nightly" unstable builds, as well as "stable" release builds.

6.2.2 Web App

The Web App satisfies most of the original requirements, but some extra features were deferred as they weren't required by the customer for the proof-of-concept.

Firstly, the device visualisation page (*"Visualise"*) should be implemented to completion so that users can visualise device data when configuring the device on-site. This also involves implementing a button that allows the user to download the device data to their phone without needing to get it off the SD card.

Secondly, Another improvement would be to add a loading screen to the device, preventing the user from interacting with the page until it has fully loaded.

Thirdly, Implementing a "captive portal" [108] on the webserver would mean that the user's device would be able to connect directly to the webapp when joining the WiFi access point the device creates, rather than having to navigate to an IP address.

Lastly, some process improvements would be to add unit testing to the Web App code to improve reliability, and linting to improve code quality.

6.2.3 Cloud Platform

Going forward, the cloud platform needs a significant amount of work to make the system ready for production and be able to scale to a high number of devices.

As discussed in section 5.3, Azure IoT Central imposes constraints that impact the connection process, network bandwidth, and analytics options. We recommend a complete rewrite of the cloud backend to use the PaaS solution offered by microsoft for IoT [49].

One consequence of a complete rebuild would mean the device provisioning process could be improved. It is currently a manual process to set up the device in IoT Central and copy the symmetric keys and other data to each device when they are initially configured. With the introduction of IoT Hub, devices could be set up to start sending data without an explicit registration step through Device Provisioning Service [50].

To be able to provision IoT devices at scale, a change in security model is required, moving away from the symmetric key method described in section 3.4.2 to *X.509* based asymmetric-key authentication. This would require setting up and managing a Certificate Authority [55], which adds additional management overhead, but allows an unlimited number of devices to connect to the cloud without managing individual keys.

Another improvement with the cloud that is missing from this proof-of-concept is the integration with GWRCs existing systems. This would require writing a script that is able to periodically collect data and transmit it over FTP in the custom XML format used by their monitoring software [6].

Bibliography

[1] Micropython, "Micropython - python for microcontrollers." https://micropython.org/.

[2] Espressif, "ESP32 Overview — Espressif Systems." https://www.espressif.com/en/products/socs/esp32.

[3] Vodafone, "Your overview of NB-IoT and LTE-M," tech. rep., 2019. Available: https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/blob/master/docs/research/networks/NBIoT%20and%20Cat-M%20June%202019.pdf.

[4] Microsoft, "Microsoft azure iot central." https://azure.microsoft.com/en-us/services/iot-central/, May 2020. Accessed: 2020-05-30.

[5] J. Behrent, "Hardware final report." https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/tree/master/docs/reports/final-report/hardware, 2020. Accessed: 2020-10-16.

[6] J. Behrent and B. Secker, "Requirements." https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/blob/master/docs/requirements/requirements.md, May 2020. Accessed: 2020-05-30.

[7] Waterwatch, "Waterwatch ls1 datasheet." https://7c6999ea-f0ce-4771-adf8-84cf41824879.filesusr.com/ugd/4e3b76_ba7c34fddadc4e8abcffca22ea7ba4f0.pdf, May 2020. Accessed: 2020-05-30.

[8] Amazon, "Amazon web services." https://aws.amazon.com/, May 2020. Accessed: 2020-05-30.

[9] Sigfox, "Sigfox - the worlds leading service provider for internet of things (iot)." https://sigfox.com/, May 2020. Accessed: 2020-05-30.

[10] K. Mekki, E. Bajic, F. Chaxel, and F. Meyer, "A comparative study of lpwan technologies for large-scale iot deployment," *ICT Express*, vol. 5, no. 1, pp. 1 – 7, 2019.

[11] Lora Alliance, "Lora alliance." https://lora-alliance.org/, May 2020. Accessed: 2020-05-30.

[12] B. Secker, "Network evaluation." https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/blob/master/docs/research/networks/networks.md, May 2020. Accessed: 2020-05-30.

[13] N. Naik, "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP," in *2017 IEEE International Systems Engineering Symposium (ISSE)*, pp. 1–7, Oct. 2017.

[14] V. Cerf and R. Kahn, "A Protocol for Packet Network Intercommunication," *IEEE Transactions on Communications*, vol. 22, pp. 637–648, May 1974.

[15] P. Agarwal and M. Alam, "Investigating iot middleware platforms for smart application development," 2018.

[16] Microsoft, "Microsoft azure." `https://azure.microsoft.com/`.

[17] Google, "Google cloud platform." `https://cloud.google.com/`, May 2020. Accessed: 2020-05-30.

[18] Google, "Google cloud functions." `https://cloud.google.com/functions`, May 2020. Accessed: 2020-05-30.

[19] Amazon, "Amazon web services - lambda service." `https://aws.amazon.com/lambda`, May 2020. Accessed: 2020-05-30.

[20] S. Plamauer and M. Langer, "Evaluation of micropython as application layer programming language on cubesats," in *ARCS 2017; 30th International Conference on Architecture of Computing Systems*, pp. 1–9, 2017.

[21] Micropython, "Micropython vs c performance." `https://github.com/micropython/micropython/wiki/Performance`, May 2014. Accessed: 2020-05-30.

[22] J. Behrent, "Hardware preliminary report." `https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/tree/master/docs/reports/preliminary-report/hardware`, May 2020. Accessed: 2020-05-30.

[23] B. Secker, "Web App Use Cases." https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Enviro
Monitoring-2020/engr489-project/-/blob/master/docs/research/software/mobile-app/use-cases.md, 2020.

[24] GWRC, "Environmental Monitoring and Research." http://graphs.gw.govt.nz/, 2020.

[25] D. Krafzig, K. Banke, and D. Slama, *Enterprise SOA: Service-Oriented Architecture Best Practices*. Prentice Hall Professional, 2005.

[26] W. Pree, "Design Patterns for Object-Oriented Software Development," in *ACM Press, Addison-Wesley*, CUMINCAD, 1995.

[27] K. Belyalov, "Tinyweb repository." `https://github.com/belyalov/tinyweb/`, May 2020. Accessed: 2020-05-30.

[28] B. Secker, "Web app transfer speed tests." `https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/tree/master/docs/research/software/mobile-app/transfer-test`, May 2020. Accessed: 2020-05-30.

[29] I. Arapakis, X. Bai, and B. B. Cambazoglu, "Impact of response latency on user behavior in web search," in *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR '14, (New York, NY, USA), pp. 103–112, Association for Computing Machinery, July 2014.

[30] B. Secker, "Tinyweb tests." `https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/tree/master/docs/research/software/mobile-app/tinyweb-test`, May 2020. Accessed: 2020-05-30.

[31] J.-C. Bos, "Jczic/MicroWebSrv," Oct. 2020.

[32] J.-C. Bos, "Jczic/MicroWebSrv2," Oct. 2020.

[33] P. Sokolovsky, "Pfalcon/picoweb," Sept. 2020.

[34] B. Secker, "Investigate web app frameworks." `https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/issues/66`, May 2020. Accessed: 2020-05-30.

[35] "React – A JavaScript library for building user interfaces." https://reactjs.org/.

[36] B. Secker, "Web server/rest api documentation." `https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/blob/master/docs/development/webserver.md`, May 2020. Accessed: 2020-05-30.

[37] B. Secker, "Mobile app requirements." `https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/blob/master/docs/research/software/mobile-app/README.md`, May 2020. Accessed: 2020-05-30.

[38] B. Secker, "Evaluate app (bluetooth) versus device (wifi) for on-device configuration." `https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/issues/67`, May 2020. Accessed: 2020-05-30.

[39] Apple, "Apple app store publishing guidelines." `https://developer.apple.com/app-store/review/guidelines/`, May 2020. Accessed: 2020-05-30.

[40] HyQuest, "HydroTel- Telemetry System." http://www.hyquestsolutions.com/products/software/mor
software/hydrotel-telemetry-system, 2020.

[41] B. Secker, "Mobile app use cases." `https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/blob/master/docs/research/software/mobile-app/use-cases.md`, May 2020. Accessed: 2020-05-30.

[42] J. Nielsen and R. Budiu, *Mobile Usability*. Berkeley, CA: New Riders, 1st edition ed., Oct. 2012.

[43] "Human Interface Guidelines - Design - Apple Developer." https://developer.apple.com/design/human-interface-guidelines/, 2020.

[44] B. Secker, "App design mockups." https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/tree/master/docs/research/software/mobile-app/design, May 2020. Accessed: 2020-05-30.

[45] Figma, "Figma - the collaborative interface design tool." https://figma.com/, May 2020. Accessed: 2020-05-30.

[46] B. Secker, "App inspiration screenshots." https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/tree/master/docs/research/software/mobile-app/design/inspiration, May 2020. Accessed: 2020-05-30.

[47] B. Secker, "Cloud platform evaluation." https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/blob/master/docs/research/software/cloud-platforms.md, May 2020. Accessed: 2020-05-30.

[48] B. Secker, "Iot central evaluation." https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/blob/master/docs/research/software/cloud/azure-iot-central.md, May 2020. Accessed: 2020-05-30.

[49] Microsoft, "Azure IoT reference architecture - Azure Reference Architectures." https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/iot.

[50] Microsoft, "Overview of Azure IoT Hub Device Provisioning Service." https://docs.microsoft.com/en-us/azure/iot-dps/about-iot-dps.

[51] Microsoft, "IoT Hub — Microsoft Azure." https://azure.microsoft.com/en-us/services/iot-hub/, 2020.

[52] Matt Chenderson, "Azure Functions Overview." https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview, 2020.

[53] Microsoft, "Overview of Azure Stream Analytics." https://docs.microsoft.com/en-us/azure/stream-analytics/stream-analytics-introduction, 2020.

[54] B. Secker, "Security research." https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/tree/master/docs/research/software/security, May 2020. Accessed: 2020-05-30.

[55] Y. Dzambasow, S. Joseph, R. Nicholas, P. Hesse, and M. Cooper, "Internet X.509 Public Key Infrastructure: Certification Path Building." https://tools.ietf.org/html/rfc4158.

[56] J. Hughes, "Why Functional Programming Matters," *The Computer Journal*, vol. 32, pp. 98–107, Jan. 1989.

[57] "Uasyncio — asynchronous I/O scheduler — MicroPython 1.13 documentation." https://docs.micropython.org/en/latest/library/uasyncio.html, 2020.

[58] E. Lubbers and M. Platzner, "Cooperative multithreading in dynamically reconfigurable systems," in *2009 International Conference on Field Programmable Logic and Applications*, pp. 551–554, Aug. 2009.

[59] L. Druda, "Iotc: Azure IoT Central device client for Python."

[60] B. Secker, “Iotc device simulator.” https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/tree/master/device-simulator, May 2020. Accessed: 2020-05-30.

[61] B. Secker, “Implement device to cloud communication.” https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/issues/87, 2020. Accessed: 2020-08-30.

[62] “Peterhinch/micropython-async.” https://github.com/peterhinch/micropython-async/blob/master/asyn.py, 2020.

[63] Geert Jansen, “RESTful API Design - Resources.” https://restful-api-design.readthedocs.io/en/latest/resources.html, 2011.

[64] “Generators - Python Wiki.” https://wiki.python.org/moin/Generators, 2020.

[65] B. Secker, “Implement backend functionality for handling monitor inputs.” https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/merge_requests/152, 2020. Accessed: 2020-10-16.

[66] A. Holovaty and J. Kaplan-Moss, *The Definitive Guide to Django: Web Development Done Right*. Apress, Aug. 2009.

[67] B. Secker, “High level device software - services implementation.” https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/blob/master/device/src/services, 2020. Accessed: 2020-10-16.

[68] M. Grinberg, *Flask Web Development: Developing Web Applications with Python*. ”O’Reilly Media, Inc.”, Mar. 2018.

[69] B. Secker, “Device configuration schema.” https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/blob/master/docs/development/configuration.md, 2020. Accessed: 2020-10-16.

[70] “Preact - Fast 3kB alternative to React with the same modern API.” https://preactjs.com/, 2020.

[71] E. Wohlgethan, *SupportingWeb Development Decisions by Comparing Three Major JavaScript Frameworks: Angular, React and Vue.Js*. Thesis, Hochschule für angewandte Wissenschaften Hamburg, Aug. 2018.

[72] B. Secker, “Static files test.” https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/tree/master/docs/research/software/mobile-app/static-site, 2020. Accessed: 2020-10-16.

[73] M. Persson, “Javascript dom manipulation performance: Comparing vanilla javascript and leading javascript front-end frameworks,” 2020.

[74] React, “Virtual DOM and Internals – React.” https://reactjs.org/docs/faq-internals.html, 2020.

[75] G. Bierman, M. Abadi, and M. Torgersen, "Understanding TypeScript," in *ECOOP 2014 – Object-Oriented Programming* (R. Jones, ed.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 257–281, Springer, 2014.

[76] "Babel · The compiler for next generation JavaScript." https://babeljs.io/, 2020.

[77] S. Motte, "Motdotla/dotenv," Oct. 2020.

[78] "Webpack." https://webpack.js.org/, 2020.

[79] "Introducing JSX – React." https://reactjs.org/docs/introducing-jsx.html, 2020.

[80] "Introducing Hooks – React." https://reactjs.org/docs/hooks-intro.html, 2020.

[81] Max Ogden, "Callback Hell - A guide to writing asynchronous JavaScript programs." http://callbackhell.com/, 2016.

[82] C. Roso, "Caroso1222/notyf - a minimalistic JavaScript library for toast notifications.," Oct. 2020.

[83] Node.js, "Node.js - JavaScript runtime built on Chrome's V8 JavaScript engine.." https://nodejs.org/en/.

[84] viv-liu, "Export data from Azure IoT Central." https://docs.microsoft.com/en-us/azure/iot-central/core/howto-export-data, 2020.

[85] tamram, "Storage account overview - Azure Storage." https://docs.microsoft.com/en-us/azure/storage/common/storage-account-overview.

[86] B. Secker, "Azure iot central setup notes." `https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/blob/master/docs/development/cloud/azure_setup.md`, 2020. Accessed: 2020-10-16.

[87] B. Secker, "Azure iot central device capability model for iot devices." `https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/issues/241`, 2020. Accessed: 2020-10-16.

[88] G. Kim, P. Debois, J. Willis, J. Humble, and J. Allspaw, *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. Portland, OR: IT Revolution Press, illustrated edition ed., Oct. 2016.

[89] B. Secker, "Devops practises in hardware-centric student projects." `https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/tree/master/docs/reports/embedded-report.pdf`, 2020. Accessed: 2020-10-16.

[90] D. Hylands, "Dhylands/rshell - Remote MicroPython shell," Oct. 2020.

[91] "The pyboard.py tool — MicroPython 1.13 documentation." https://docs.micropython.org/en/latest/reference/pyboard.py.html.

[92] B. Beyer, "Junit-xml: Creates JUnit XML test result documents that can be read by tools such as Jenkins."

[93] "Unittest — Unit testing framework — Python 3.9.0 documentation." https://docs.python.org/3/library/unittest.html.

[94] H. Leung and L. White, "A study of integration testing and software regression at the integration level," in *Proceedings. Conference on Software Maintenance 1990*, pp. 290–301, Nov. 1990.

[95] "Web App Service — Microsoft Azure." https://azure.microsoft.com/en-us/services/app-service/web/.

[96] B. Secker, "Rest api implementation." https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/tree/master/device/src/services/restapi, 2020. Accessed: 2020-10-16.

[97] H. Pham, *Software Reliability*. Springer Science & Business Media, Feb. 2000.

[98] B. Secker, "Job failed 119789." https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/issues/324, 2020. Accessed: 2020-10-16.

[99] J. Behrent, "Regression tests for logging driver." https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/blob/master/device/src/test/test_logging.py, 2020.

[100] Benjamin Secker, "Differences with older version & micropython support · Issue #1 · iot-for-all/iotc-python-client." https://github.com/iot-for-all/iotc-python-client/issues/1.

[101] B. Secker, "Redesign and implement visualise page." https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/merge_requests/143, 2020. Accessed: 2020-10-16.

[102] A. Engelen, "Raboof/nethogs," Oct. 2020.

[103] A. Orebaugh, G. Ramirez, and J. Beale, *Wireshark & Ethereal Network Protocol Analyzer Toolkit*. Elsevier, Dec. 2006.

[104] Nasko Oskov, "TLS overhead - netsekure rng." http://netsekure.org/2010/03/tls-overhead/, 2020.

[105] B. Secker, "Iot central visualisation and analytics exploration." https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/blob/master/docs/research/software/cloud/iotc-visualisation.md, May 2020. Accessed: 2020-05-30.

[106] "Power BI on Azure - unified self-service and enterprise analytics solution — Microsoft Azure." https://azure.microsoft.com/en-in/services/developer-tools/power-bi/, 2020.

[107] B. Secker, "Redesign and implement visualise page." https://gitlab.ecs.vuw.ac.nz/QuiltyGroup/Research/ENGR489/Environmental-Monitoring-2020/engr489-project/-/issues/115, 2020. Accessed: 2020-10-16.

[108] D. Wilson, C. Esdaile, D. Setia, and P. Iyer, "Captive portal redirection using display layout information," May 2016.