

No part of the candidate's evidence in this exemplar material may be presented in an external assessment for the purpose of gaining an NZQA qualification or award.



Scholarship Technology 2025

93601

EXEMPLAR

Outstanding Scholarship

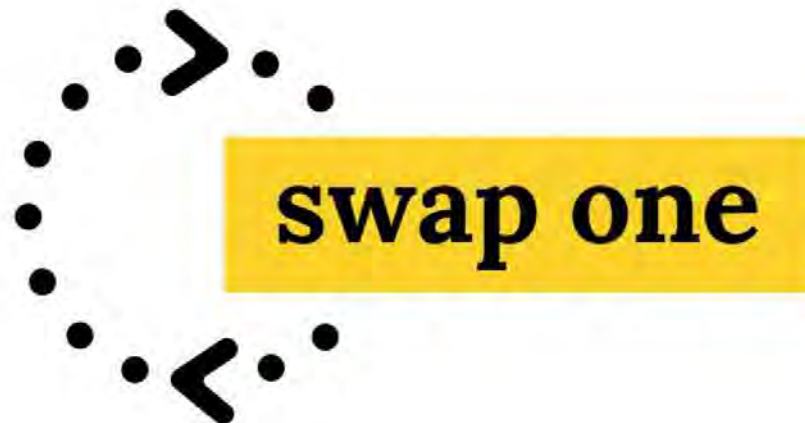


Table of Contents

Introduction

Abstract.....	2
Declarations.....	2

Part 1: Brief Development and Research

1.1. Problem Statement.....	2
1.2. Market Research.....	3
1.3. User Research.....	5
1.4. Development of the Brief.....	6

Part 2: Planning and Design

2.1. Project Planning.....	9
2.2. Architecture and Technology.....	11
2.3. Conceptual Design of the Application.....	16
2.4. User Interface Design.....	18

Part 3: Software Development

3.1. Development of the MVP.....	26
3.2. Development of the Second Iteration.....	33
3.3. User Interface Testing.....	41
3.4. Finalisation and Productionisation.....	43

Part 4: Evaluation

4.1. Final Testing.....	48
4.2. Fitness for Purpose.....	50
4.3. Real-World Deployment and Impact.....	52
4.4. Reflection and End Note.....	53

Abstract

This report showcases my development process for the Swap One app. Swap One is a community initiative in Nelson, encouraging participants to reduce their carbon emissions from commuting to work by using sustainable means like walking, biking, or using an electric vehicle. The Swap One programme wanted an application to encourage this, to allow users to log their sustainable trips, and to facilitate communication from the programme to users, and I was asked to develop this. Using an iterative Agile methodology, I created a magic-link style email-based application, which met the needs of the programme and was deployed into production, after being evaluated against the brief and found to be fit for purpose.

Declarations

No generative AI was used to program this application or write this report.

The Swap One application as a whole was my contribution to the wider Swap One programme, and was my work, except when I explicitly state that I use libraries or modules. The research and design process was also my work, apart from the initial mock-up designs which were provided to me.

Part 1: Brief Development and Research

1.1. Problem Statement

The Problem

In the Nelson region, 61% of greenhouse gas emissions are from transportation. To meet our 1.5°C Paris Agreement obligations (the most warming the earth can have to avoid drastic consequences), the rate at which we decarbonise must proceed at about 8% per year¹. According to the Nelson-Tasman Climate Forum, if every person in Nelson cut a fifth of their transport emissions, their household emissions would be reduced by 9%, meaning they are doing their part to meet Nelson's 8.3% emissions reduction target.

Stakeholder Needs

My primary stakeholder for this project was the Nelson-Tasman Climate Forum. They had recently received a small amount of funding from the Nelson City Council to run a programme named Swap One, with the goal of encouraging people in Nelson to cut their commuting emissions. The Forum, along with other organisations, planned to encourage swapping a trip a week with two methods. Firstly, producing an advertising campaign together with the local council, and then by getting an app set up where users can log their sustainable trips – providing valuable statistics to organisations and allowing a prize draw to be arranged to reward people who travel sustainably. They were initially planning to become a member of an initiative called the Wednesday Challenge, which provides similar software and resources.

¹https://www.nelsontasmanclimateforum.nz/wp-content/uploads/sites/353/2024/08/Target-Explainer_July-2024-w-2022-data.pdf

However, to purchase a city membership, the Wednesday Challenge would charge a fee of \$65,000 in total! The Climate Forum, a volunteer-led organisation, cannot afford to subscribe to this initiative, and even if they could, it would not fully meet their needs. It was clear that they needed a bespoke application for the local programme to succeed. This created an opportunity for me to develop a custom-built app that was cost-effective and met the needs of the project.

To summarise, they needed me to develop a piece of software to (a) allow organisations or individuals in the Nelson-Tasman region to register their commutes to school or work, (b) incentivise use of the app and therefore sustainable commutes, and (c) support a long term goal so that, even after the project ends, users will have formed strong habits and will continue commuting sustainably.

1.2. Market Research

I began by studying two other initiatives with similar goals – the Wednesday Challenge and Love to Ride's Aotearoa Bike Challenge – to see if there were any important lessons I could take away from them.

The Wednesday Challenge is a programme working with businesses and councils across New Zealand, incentivising participants to use sustainable means to commute to work or school on Wednesdays. As mentioned before, Swap One initially intended to use their web app, but they charge \$25,000 for access in schools and \$20,000 for access for businesses. Furthermore, if the project was to be rolled out to Tasman as well as Nelson City, there would be a total cost of \$65,000 to the programme, which was not within the budget. Additionally, the Wednesday Challenge system did not quite meet the requirements for the programme – the Swap One vision was to allow people to swap trips at any point, not just Wednesdays, and the organisers of the Wednesday Challenge would not be able to share information about the participants, so Swap One would be unable to report back to organisations about progress.

The other similar initiative, the Aotearoa Bike Challenge, is run by the global bike advocacy organisation Love to Ride, aiming to get New Zealanders on to bikes for the month of February each year. Whilst a more polished and better-known platform, the same issues making the Wednesday Challenge infeasible to use affect the Bike Challenge – costs around \$50,000 and no access to data. Additionally, this initiative is just for bikes, meaning swaps in other forms such as public transport or walking would not be counted.

A welcoming user interface will make the app much more usable

Both apps have interfaces which were sometimes difficult to use, which made it harder for me to log trips, and thus will reduce the users' motivation. For example, both the Wednesday Challenge and the Bike Challenge are really cluttered, with confusing buttons without obvious purposes, which will make it harder for users to complete the

task they want. Wednesday has a modern-style interface, but it looks bland and empty despite the clutter, with uneven spacing being the cause. Because of this, the webpage seems uninviting and unprofessional.

On the other hand, the Bike Challenge has a more professional and well balanced aesthetic, with less blank space, although it feels dated. The warmer colours also contribute to a feeling of homeliness, which will make the Bike Challenge page much more pleasant to use. The takeaway from these interfaces is that it's important for the Swap One interface to look and feel welcoming, while getting out of the way so users can complete their desired task.

Sign up must be as simple as possible

The Wednesday Challenge had a difficult sign-up process, unlike the Bike Challenge. For Wednesday, the user has to go through pages of various questions, with loading times between each, and annoying lists of massive icons to scroll through. The user must find themselves on Google Maps, decide what motivates them to participate, and provide various other information during the sign up process, all designed in a spread out and thus slow manner. Whilst this system might look better than a simple form, it is a usability nightmare and no doubt has dissuaded many potential users from signing up.

In comparison, the Bike Challenge allows the user to do a single-click sign up with Google, instantly creating their account. After sign up, the user can add other information and engage with various mechanics only if they wish, unlike the forcible system Wednesday uses. Sign up is one of the most common places to lose interested people, so it is very clear that the sign up system must be as simple as possible, with only the most needed questions. If there are lots of questions that must be asked of the user, they should be enterable in the most generic way, so that users can quickly get through the sign up.

Feature bloat will confuse and alienate users

Both the apps I studied seemed to be affected by feature bloat. There are many features on the app, that while useful and engaging for some people, are almost certainly ignored by most of the users. For example, both apps have a prominent badge system. While these maybe great for some people, they clutter the app up and detract from the main purpose. The Wednesday app has three separate systems to incentivise – badges, points, and reminders, all of which feel incomplete and unpolished. Users clearly will not bother with remembering all three, so one fully functional system would boost engagement much more than the current setup. Similarly, the Bike Challenge has four different incentive systems – personal goals, workplace challenges, badges, and streaks – and all these motivators make it hard for users to know which is important. I suspect in these apps stakeholders have requested features to be added without understanding the concept of “less is more” – something that has to be key to my app.

My app must be more usable, otherwise uptake will suffer

The two different design approaches seem to have made a tangible difference in user retention. While the Bike Challenge does have flaws, it is easy enough to use and log your trips, showing in the fact that there is a high amount of active users. Meanwhile, Wednesday has high sign up rates but low trips logged, showing that people have signed up, logged a couple of trips, and never touched the app again. A key part of

this is potentially that the Wednesday Challenge uses calendar-based reminders, which seem ineffective, probably because of the difficulty of getting from the reminder to the webapp. A key takeaway is that to improve retention, we need a way to remind people to take their trip while allowing them to easily take action from the reminder.

Both of these programmes, despite their flaws, are good inspiration for my app. Neither app tells the users what to do, and expect the users to be willing to spend ten minutes figuring it out. My app must be intuitive or explained. My biggest take away from these platforms is that it must be as easy as possible to log your trips, or people won't bother and the app will die.

1.3. User Research

To complement the market research, I also distributed a survey to members of the public to help understand the project's primary target audience – employed people in Nelson-Tasman who care about the environment but feel like they're powerless to do anything about it. The survey had the goal to test ideas and identify promising features from the market research while understanding what potential users actually want. I received 12 responses from varied demographics, which seemed to have no correlation with the responses. The survey asked respondents about their commute and what they thought would motivate them to use the app or commute sustainably. If respondents indicated that they had participated in the Aotearoa Bike Challenge, I followed up with questions asking what they thought worked well and what could have been done better. I decided to leave out questions about the Wednesday Challenge due to the low usage in Nelson.

I had three key findings from the survey which made a big impact on the design direction of my app.

Users are primarily motivated by clear, simple statistics about their impact

When asked which features would motivate them, respondents were overwhelmingly enthusiastic (10/12 respondents wanting the feature) about statistics explaining what they've done. Gamification and prizes were also popular (7/12 respondents each), but competitive features like leaderboards or competing against contacts seemed only useful to a small minority of people (4/12 respondents, with 5 others saying it would be actively unhelpful). This clearly shows that personal and intrinsic motivation is a key motivator here. While I considered adding competitive features into the app, and had them on my feature lists, I decided to tone the competitiveness down a lot based on this, making these very much optional and not disturbing people who don't want them.

A frictionless user experience is critical for success

Bike Challenge participants praised the easy to enter trips and the easy registering system, but criticised it for being cluttered, busy looking, and the difficulty to reset a password. Other features from the bike challenge that users enjoyed were linking to Strava, which unfortunately is not feasible due to difficulties around the API, and the ability to quickly see your group's progress. Based on the feedback around this platform, it seems clear that for users to enjoy their experience and thus stick around, trip logging and registration should be as easy as the bike challenge, while the rest of the app should be as simplified and user-friendly as possible.

To be successful for the whole community, accessibility barriers must be removed

A respondent to the question asking if they thought the app would be useful said “The app will only reach the audience already captured on sustainability. The smaller kids won’t have access... Young families are usually time poor... Senior residents won’t engage...” This is an excellent point. Rather than being discouraged, I think prioritising design to engage these groups is necessary for success. Whilst it is not possible to provide phones to smaller kids, by using reminders and making the process as quick as possible younger families can be engaged, and with a very simple system seniors can be engaged. Additionally, the Swap One programme targets primarily working age people through their businesses, so most will be in the suitable demographic. To get full success, however, simplicity and accessibility must be key design principles, above everything else.

Remember the target demographic

Whilst all of this is very useful, it is important to note that this survey has inherent bias. Because the people who were already interested in sustainability were the ones who chose to answer the survey, these responses are from people who are already motivated, and would potentially not find external motivation like leaderboards or competition useful, whereas other groups of people who aren’t already engaged may.

In conclusion, respondents seem to be mostly interested in the app, and the responses match up well with what I’ve gathered from the market research. Simplicity, speed, and accessibility are key to the wide appeal of the app and must be prioritised in the design philosophy. The lack of desire for competitive features was interesting, since personally, I thought that would motivate me. However, if people from my target demographic believe they would negatively impact the experience, these features may not be worth it.

1.4. Development of the Brief

The brief process started with an initial feature request list from [REDACTED], my primary stakeholder and the Swap One project lead. This was comprehensive, but it was also in the form of a wish-list of features. I knew that I needed to focus this down, so I produced a brief in the form of Agile user stories based on feedback from my stakeholders. To get to that point, I went through a four step process with [REDACTED] based off of all my research: Vision → Outcomes → Non-functional Requirements → User Stories. This allowed me to narrow down the set of feature requirements to a Minimum Viable Product, which identified the basic features for a working app.

Vision and Outcomes

The overarching vision of the Swap One programme is to “Reduce carbon emissions in Nelson-Tasman”. Based on this vision, we agreed on four outcomes that the app must meet to be successful.

1. App must incentivise users to commute sustainably in Nelson-Tasman

This is the primary outcome of the programme, and directly leads to reduced emissions in Nelson-Tasman.

2. App must collect data for stakeholders to use, both in- and out-programme

This outcome meets one of the critical needs for my stakeholders, and is a key reason that other off-the-shelf applications were not suitable. This data is needed both for the prize strategy and for informing future initiatives.

3. App must ensure a frictionless and trustworthy user experience

Both market and user research indicated that user drop-off is caused by user interface difficulties. For long term user retention, a great interface is required.

4. App must have a scalable and maintainable technical foundation

For the app to be suitable to be run by non-profit organisations with limited budgets, the app must be able to be used and adapted by other individuals if I am not available.

Non-Functional Requirements (NFRs)

NFRs are quality standards for an application that describe how well it must do certain things. I derived these from the outcomes, the user and market research, and discussions with the stakeholders. NFRs usually have target measures, but for this project I am only using NFRs to guide my planning.

App must be easy to use and accessible for users with a range of technical literacy and abilities (Usability & Accessibility)

This links to the outcome of a frictionless and trustworthy user experience, since, based on my research, for a good experience the app must be easy to use.

App communications with users must be reliable and trustworthy (Reliability & Security)

This links to the outcome of a frictionless and trustworthy user experience and the outcome of collecting data for stakeholders, since trustworthy communications are needed to build user trust so that they will feel confident in submitting their data.

App must utilise responsive design (Portability)

This links to a frictionless and trustworthy user experience, since users may find an improperly responsive application doesn't work on their device, adding friction.

App must handle concurrent user activity without crashing or lagging (Stability & Performance)

This links to the outcomes of supporting a quality user experience, incentivising users to commute sustainably, and a scalable and maintainable technical foundation, since technical errors will make the user experience difficult and thus drive away users, as well as causing maintainability issues in the future.

App must be built and operated with minimal ongoing costs (Cost Effectiveness)

This links to the outcome of a scalable and maintainable technical foundation, since if it is expensive to operate the stakeholders will be unable to afford running and scaling the app.

User Stories

For the final layer of the brief, I used the Agile methodology's user stories concept, where every requirement is described in context, explaining what a user needs. Based on the features requested by my stakeholders, I developed an initial list of 20 stories, which was then cut down to eight MVP (Minimum Viable Product) stories. These are the features that the app cannot launch without, otherwise it will not meet the outcomes. They are what formed the formal brief and final specifications. These stories also have acceptance criteria, which are what must be done to implement the request. In Part 4, I used these, along with outcomes and NFRs, to judge the application's fitness for purpose.

However, it's important to note that these stories were just the start of the process. Throughout the development process, more stories were added, either as feature requests from my stakeholders or things I realised were needed, since iteration is an important part of the software development process and the Agile methodology. To preserve the flow of the work, I have recorded those additional stories chronologically in Section 3.2, when they actually came up during development.

As a prospective user I want to join Swap One so I can reduce my emissions

Acceptance criteria:

- Sign up system asks for user information
- Collected information stored so it is accessible to other system components
- System confirms sign up

As an app user I want to log my sustainable trips so I can participate in the programme

Acceptance criteria:

- System records appropriate trip information
- User response saved to database

As a programme coordinator I need to be able to access commute data in an Excel compatible format

Acceptance criteria:

- Database exported to user-readable Excel compatible file
- Spreadsheet uploaded to cloud storage daily

As an app user I want to be able to view emissions saved so I feel motivated to keep commuting sustainably

Acceptance criteria:

- Emissions saved calculated based on data from databases
- Emissions saved visible in weekly email

- Emissions saved visible on logging webpage

As an app user I want to be reminded to do a sustainable trip so that I don't forget to commute sustainably

Acceptance criteria:

- User sent reminder notification weekly
- User able to access logging system from reminder

As an app user I want the design to be simple and intuitive so I don't give up on using the app

Acceptance criteria:

- Page has consistent user interface
- Page UX meets Nielsen's heuristics
- Page UX meets WCAG accessibility guidelines

As an app user I want to be able to unsubscribe from the system so I can retain control of my information and stop participating

Acceptance criteria:

- Unsubscribe subsystem easily accessible
- All identifiable user data removed from database when unsubscribed

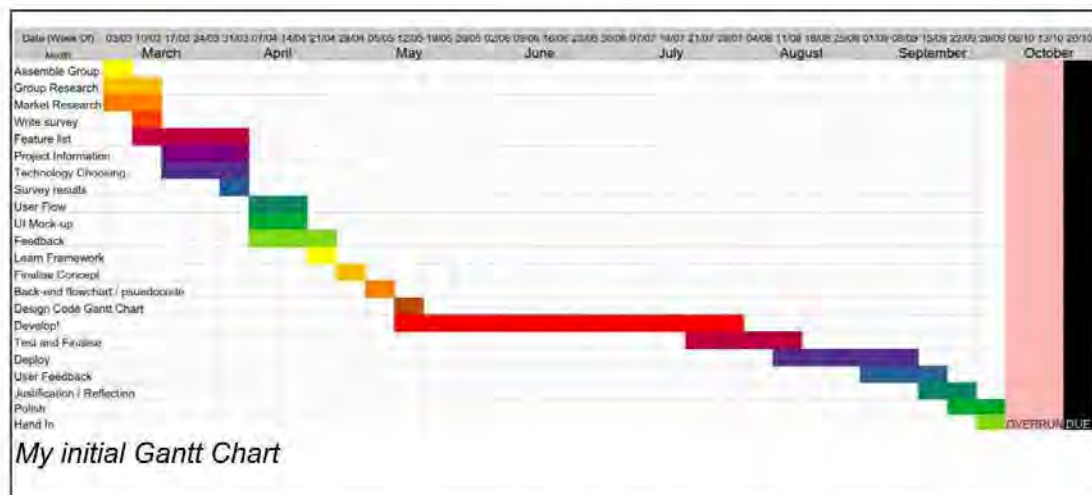
Part 2: Planning and Design

2.1. Project Planning

Planning Approach

Before starting this project, I planned to use a traditional waterfall approach – do thing one, then thing two, then thing three. I originally started making a Gantt Chart to plan out when I would work on different parts of the project, leaving time for things to go wrong and basing the rest off of my initial to-do list. However, a few weeks in I realised that the Gantt Chart really was not a suitable planning methodology for my project.

When I started this project, I had no idea how long things would take. I made reasonable estimates, but obviously things took different amounts of time. For example, I did not spend a whole week “finalising the concept”, but needed more than a quarter of a week just to analyse and understand the survey results. Additionally, my availability was different week-to-week, and I ended up just doing as much as I could every week, instead of stopping when I had finished that week's goal.



Real project management and development is often not linear like my chart made it out to be. It turned out not to be possible to design the user flow, followed by the UI, then get feedback, before doing code design, and then writing all of the code in one go, perfectly. In the real world with project management, stakeholders ask for more features, or you realise that something doesn't work, and you end up circling back four or five times through all the stages to achieve real iterative development.

I didn't actually end up doing everything on my Gantt chart, and I ended up reordering the way things were done. For example, I did not end up meticulously writing out pseudocode and planning out the whole program in advance, like you're supposed to do in NCEA assignments. When you go through and develop, and realise something isn't quite right and must be changed tens of times, or that a major feature needs to be added, it isn't feasible to re-plan everything. Furthermore, I realised mid-project that it would be far more effective to design my final interface once I'd settled on what had to be in the actual app, rather than designing beautiful interfaces that weren't fit for purpose.

Due to this, I decided to go all in on the Agile methodology, which I was already borrowing the concept of user stories from. Here, the project is broken down into little chunks which are then prioritised according to the stakeholders' needs. Since the chunks are created relatively independently, planned elements can be changed or replaced whenever, and new features that come up can easily be added if that's what the stakeholders want. With Agile, I built a prototype, shared it with stakeholders for feedback, and then could quickly and easily modify it and repeat the process again.

Since I still needed a way to plan my project, I ended up using lists in an equivalent manner to a Kanban board, having a *to-do* list (my backlog), an *in progress* list, and a *completed* list. As I went through development, I would move user stories between these phases, and if something needed to change I could easily adapt one feature without worrying about the others.

By changing from the waterfall to the Agile methodology, I could actually match the chaos of real-world development, and had a planning structure that helped me achieve what I wanted, rather than holding me to a strict and rigid path.

Risk Management

At the beginning of the project, I recognised that there were inherent risks that could stop me finishing the project on time. Due to this, I identified everything I could think of and came up with strategies to avoid them.

Scope creep

I knew that I would either be tempted to add unnecessary features, or that my stakeholders may want more things than could be delivered. This could have caused me to run out of time, so I decided to identify the MVP features first, and complete them to a satisfactory level before implementing anything else.

Programming difficulty

I was also worried that I would find the programming I had to do new and difficult, and so I would run out of time to implement everything properly. So just like professional developers, I decided to use off-the-shelf technologies and frameworks that had “batteries included” whenever available. This meant I could focus on programming features that were unique to my app, not stuff that had already been done.

Getting burned out

I was also worried that I would take things too fast at the beginning and get “burned out” – feeling tired and demotivated, meaning I’d struggle to complete the project. I decided to pace myself and not spend all my free time on this project, instead doing other relaxing things when I could.

Unpredictable stakeholder availability

My primary stakeholders were volunteers with their own commitments, so I was concerned that they would sometimes be unavailable when I needed feedback. Due to this, I planned ahead when I asked for feedback, making sure I had other work to do whilst I waited.

2.2. Architecture and Technology

App Type and Authentication

The first and most fundamental decision was whether I should build a web app (accessed through a browser) or a mobile app (downloaded from an app store). Based on that, I then needed to decide how users would interface with the app because users have to be able to identify themselves so that they can submit information. When reviewing my user stories, I came up with three options for this, centred around how to authenticate the user.

I knew that developing a mobile app is difficult, with technologies I’ve never used before. Even if I could figure it out, it would take a long time, and it would be a massive hassle to get onto app stores. I had read past Technology Scholarship entries, and app submission always seems to be an issue, taking months to get builds approved and being consistently rejected if you’re under 18.

Because of this, I felt like the app had to be a web app, running entirely in a browser and using technologies that I am familiar with.

Option 1 – a traditional username/password model

The first option is a traditional model where users input a username and password. Next, they get taken to some sort of home page, where they can choose to log a trip. Every week, they can be sent reminders by email.

When thinking about this model, I found some inherent flaws. It seemed really difficult to develop a good authentication method. The traditional username and password system is really easy to mess up, and since I had no experience in this space, it seemed possible that I could accidentally leak people's passwords. I could certainly hack together a working system, but if a malicious actor tried to scrape people's information they would almost certainly be able to get passwords. Since people often reuse passwords, this could be catastrophic. Additionally, having to remember and enter passwords isn't a great user experience, and since I wanted to make the logging as easy as possible, this system seemed to have some significant downsides.

Option 2 – oAuth model

The alternative to a bespoke username/password method is using a third party authentication provider like Google's oAuth, which lets users login with their Google account and sends me a user token to use in lieu. While this is much more secure, it will be far more difficult to integrate, and will cost the stakeholders every time a user logs in, which seems unfeasible for this project. At this point I felt lost, but after reviewing my user stories, requirements, and talking with people, I came up with a third option.

Option 3 – email Magic Link system

Since web apps couldn't send notifications to phones, if I wanted to prompt users to log trips, I'd have to use a system like email, since users would never remember to go to a specific website once a week. Setting up email would be a pain, but completely doable compared to everything else. But if I was already setting up email, why not use the email reminders as the way to authenticate users, and do away with passwords and auth providers all together?

The "Magic Link" concept is already in use by many different organisations, and everyone has been sent a link to reset their password. Why not take it a step further and make email the way to access the system? In this model, users sign up simply by inputting their email, and every week are sent an email asking if they had swapped a trip, acting as a reminder. In this email, there are links or buttons they can press that take them to a web app already logged in, allowing them to input their swapped trip information.

This email system has incredible ease of use, which is a key design priority. Doing something the users already do – checking their inbox – they see an email and can log a trip in just a couple of clicks. Everyone knows how to use email, so users won't just sign up and forget how to use the system. They'll be reminded naturally every week as they check their emails, and since it's so easy users will actually do it.

Sending emails is also absurdly cheap, accessible, and reliable, so the email system seems to meet all of my criteria. This is clearly the better option.

This model is, however, less secure than the others. The only security in this system revolves around the user's UUID (identification string). While this string will be random and thus unguessable, if a person forwards their email or leaves the page in their browser history, anyone accessing these resources will be able to view the user's account.

I thought this was a worthwhile trade off, but we needed to make sure only to collect non-sensitive information and warn users of the risks.

Technology Stack

Now that I knew what I was making, I had to figure out what technologies I would build my app on.

Choosing an affordable and maintainable back-end architecture

My next choice was whether to use Back-end as a Service (BaaS) systems like Google's Firebase or to script something myself. BaaS systems work as a series of functions that the front-end application interfaces with, compared to the traditional back-end components.

Although I knew that a BaaS approach would probably be easier, I decided not to use one. The biggest advantage of using my own back-end is cost. BaaS costs are incredibly variable depending on traffic, whereas renting a server from a cloud provider like Azure costs only around \$10 a month. The custom build is clearly more inline with my Cost-Effectiveness NFR.

While getting the very basics of a web app working properly would be achievable with BaaS, the second I had to do something more complex, I would have to learn their system in depth, along with probably a new programming language. By using my own back-end, I would have access to modules that abstract complex elements anyway, while retaining control of the parts I choose to keep. Having more control would support the Maintainability NFR.

Another important point in favour of my own back-end is that I would learn a lot more. Rather than having difficult elements like databases abstracted away, I would have to learn to work with these directly, which are skills that I want to have and will be useful in the future.

Additionally, most BaaS providers have servers in Australia, which use 74% fossil fuels to generate their power. In New Zealand, only 16% of our power comes from fossil fuels, so hosting my back-end here will reduce emissions. I recognised that the total server emissions will be minimal, but it still felt against the ethos of Swap One.

A big flaw with not using a BaaS is the lack of security. A BaaS would have protected user data better than any app I could program, so the Swap One team decided to mitigate this by collecting as little user information as possible.

This means I can rent a low-powered server running Linux.

Using Python as a back-end language

I then had to decide which scripting language to use on the server. There were two good options – Python and JavaScript.

Python is a popular and modern programming language that, while not incredibly performant, is flexible, incredibly modular, runs on a wide variety of devices, and has wide community support with many usable modules that will make development a bit easier. Python has many modules that support hosting a web page from a script.

The other possible option is JavaScript with a hosting framework called Node.js. It has pretty much all the same advantages as Python, is faster, as well as being more of a web-oriented language, which is useful for this project.

However, my Python understanding and skill level was significantly better than my JavaScript. I had written projects in both, but I had never used back-end JS, while I had used Python as a back end for my projects many times. Even though the JS system would've been slightly more efficient to run, the ease of programming in a language I already knew seemed to be more inline with what I needed, since the app had to be done quickly to be ready for launch.

So, I decided to write my back-end scripting in Python.

Django and SQLite / PostgreSQL

I also had to decide between using Flask or Django as the Python framework running the website. The core difference between these two is that Django does a lot of things automatically for you whereas Flask expects you to do things yourself. Flask's philosophy was certainly more lightweight, since only the needed features were included, but I thought in this case, Django's features were worth it. Django is overkill for my purposes and has a lot of features I didn't use, but the templating engine for HTML pages, the built in security features protecting against XSS and SQL injections, and an admin control page seemed worth it. Since I wasn't experienced with security at all, strong protections on the back end from Django were very much worthwhile, supporting the Security NFR.

For most of the development I used Django's development server since it was easy and included. Once I was moving to production, I switched to Gunicorn, which is a single liner install and is very fast.

For most of the process of development, I used the SQLite database built into Django, which is notorious for being slow and locks the database when it is being written to. As I was testing the production server, I was having read/write concurrency issues, which I will go more in depth to later. Because of this, I made a last-minute pivot to PostgreSQL, which is much much faster and supports read and write concurrency, while being much harder to set up.

Programming the front-end without a styling framework

I decided to use HTML, CSS, and JS for my front-end programming. Web frameworks like React or Bootstrap look good for specific styles, but I wanted maximum control over how the page looked, so I could style it how I wanted, instead of how the React developers wanted it. Additionally, I wanted my page to be fast and simple to support my Usability and Performance NFRs, and simple CSS is still the fastest way to style a website. I also had less experience with other frameworks.

Supporting services for emails and file hosting

I needed a way to style emails beyond plain text, so I decided to use HTML emails. The issue was that different inboxes have different levels of HTML support, from full HTML5, to using the rendering engine from Microsoft Word 2007. It's almost impossible to write HTML to make an email that looks good in all inboxes, but it had to be usable in all inboxes to meet my Accessibility NFR. Thankfully, there is a markup language called MJML, which gets compiled to email-friendly HTML. I used this to generate my email templates.

Both Microsoft's Azure Communication Services (ACS) and Amazon's Simple Email Service (SES) have reasonably cheap pay-as-you-go email sending services. SES was slightly cheaper, so I initially decided to use that. However, when I tried to get approval to increase the rate limit from 10 emails a day to 1000, our project was rejected for being a "security risk", without further explanation. Because of this, we switched to ACS, who thankfully accepted us. ACS costs around \$0.10 USD per 1000 emails sent, which is completely affordable for the project and meets the Cost-Effectiveness NFR.

There was also the alternative of sending emails directly from the server, however these would absolutely not be trusted by inboxes and would certainly be sent to spam folders.

I needed to be able store exported data securely, so that stakeholders could access it. Both Microsoft and Amazon offer cheap, API based file uploads. Initially, I chose to use Amazon, since I was already using their services, however I switched to Microsoft to keep all of my cloud services with one company.

Tooling Decisions

I did all of my programming and developing on a computer running Linux. This is because the server I will be running the app on will be on Linux, so developing on the same kernel will make it much easier to move the code from my desktop to the server.

I used the Phoenix Code IDE for all of my development. Phoenix Code is a simple text editor which gets out of my way and had a sidebar for me to navigate easily through my codebase, which I find a must for productivity. It also had useful features such as Live HTML Preview so I could see how a page looked as I changed it, automatic text formatting so my HTML is readable, and a built in Git client to make version control easy.

I used Git for my version control, with GitHub as my repo host, because it's the industry standard and I already had an account. With Git, my work was always stored in a safe place so that if I made a mistake I could easily revert any changes. Every time I finished something, I committed my code, resulting in over 100 commits over a five-

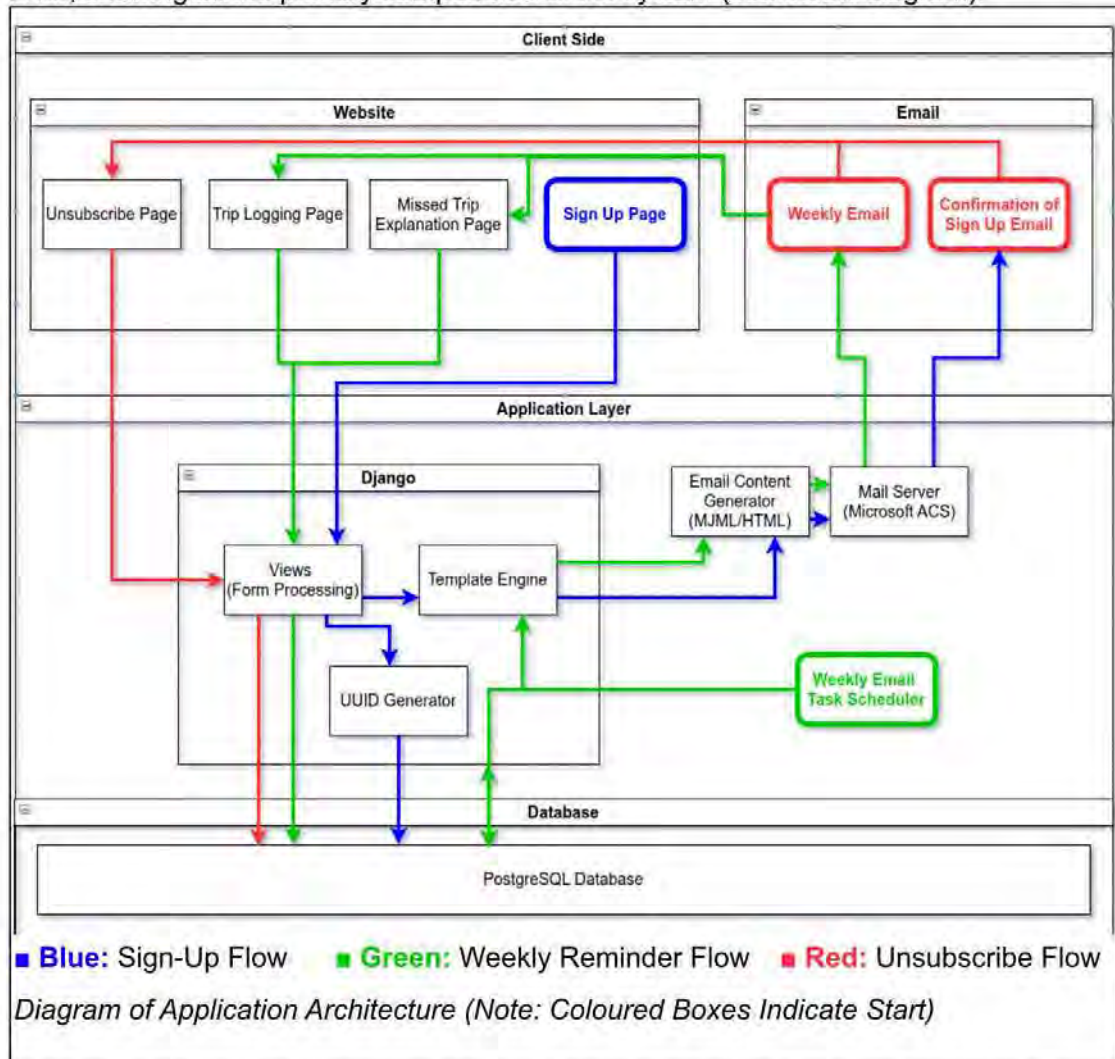


Phoenix Code on the Swap One repo

month timespan. Git was also incredibly useful when moving to production, since I could make a change on my desktop and run a single command to apply my changes to the server.

2.3. Conceptual Design of the Application

Right before development, I designed the architecture for the MVP system at a high level, showing all the primary components of the system (see below diagram).

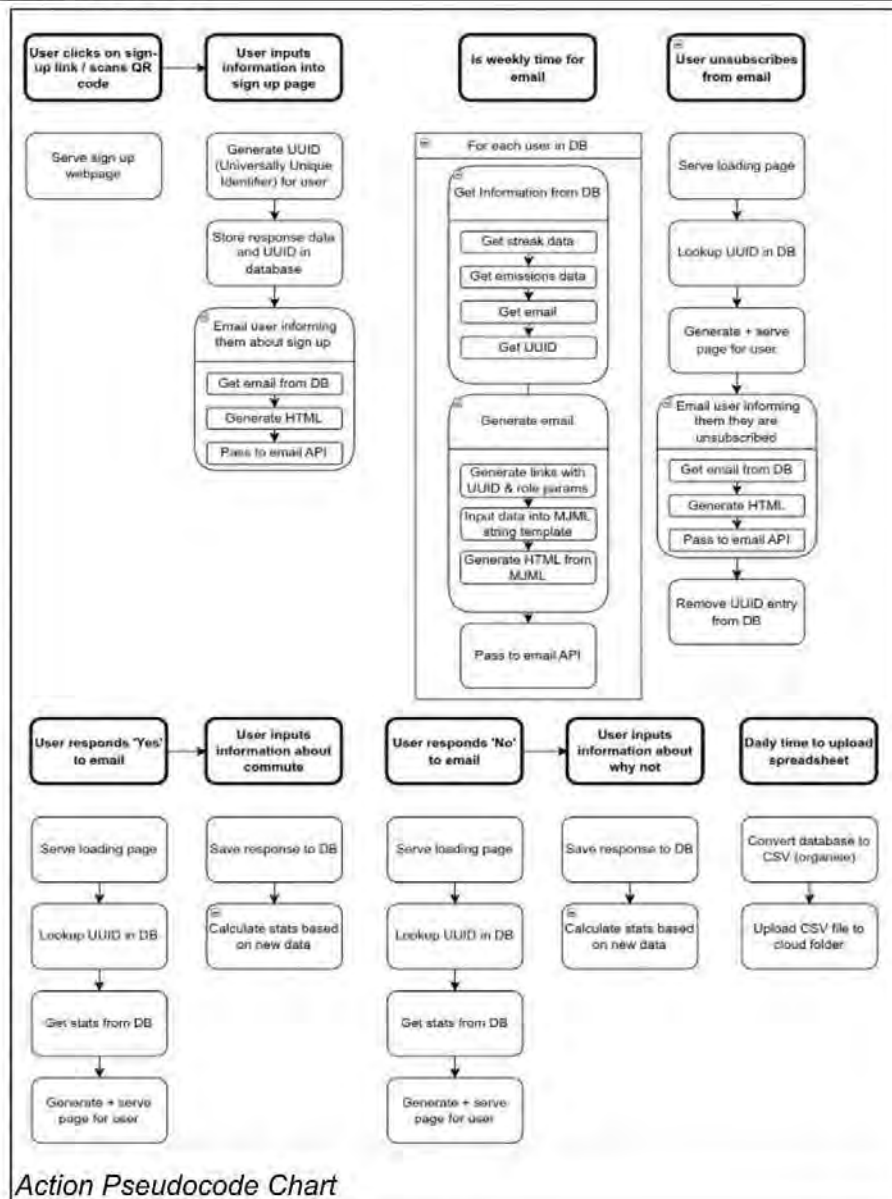
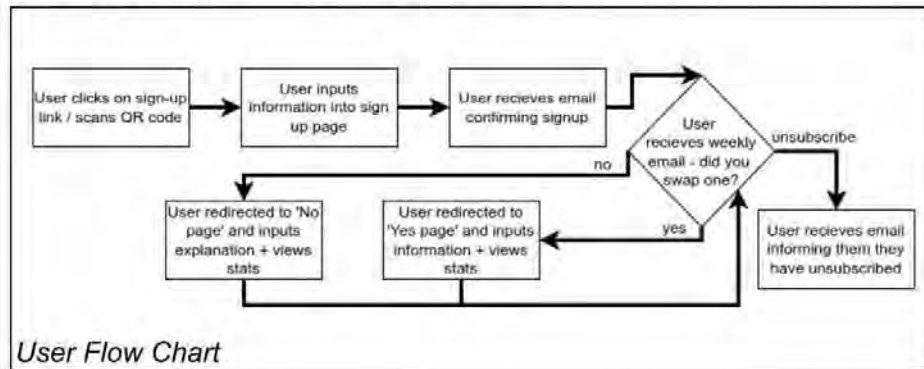


The app architecture is split into three tiers – the client side (made up of the website and the email interface), the application layer (consisting of Django, a task scheduler, and the email sending system), and the database (consisting of PostgreSQL).

This architecture was a useful description of the foundation of the application, but doesn't include any features beyond MVP or any application resilience work to make it a viable, production ready system.

Next I made a a user flow diagram with actions in flow-chart pseudocode. Here, an action can be triggered by either the user throughout the flow, or alternatively by the

system. Whilst this also does not include anything beyond MVP, it was a good way to understand my thinking at this phase of the project.



At this stage I hadn't fully grasped the interconnectedness of my application, and still thought the programming process would be relatively linear.

2.4. User Interface Design

Design Inspiration and Research

I ended up designing my final interface after building a functional MVP prototype that I then iterated on. Following Jakob Nielsen's ten usability heuristics² throughout the process, this meant that I knew the general way the app would be laid out and organised. Part of that research included the user testing that I explain in Section 3.3.

The Swap One team provided me with a branding guidelines document, showing that I was supposed to use the Lora font for headings and the Inter font for body text.

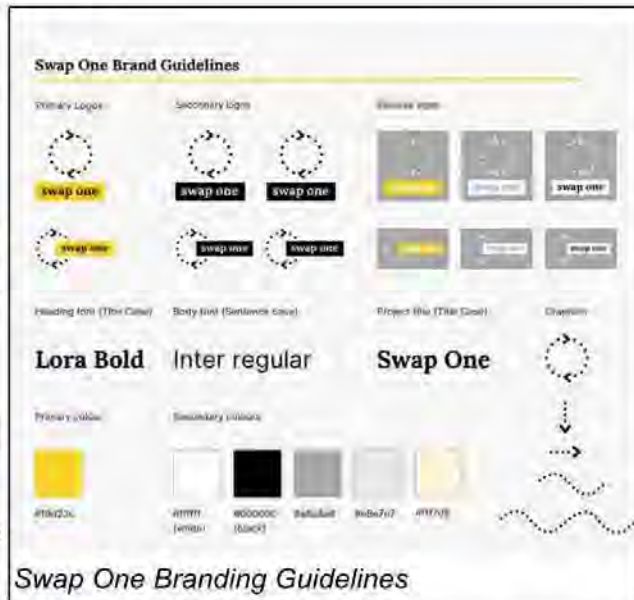
The general colour scheme was based around yellows. I was additionally provided some mock-up designs that created to inspire my design work.

When looking at the mock-ups, my immediate opinion was positive, although I did notice some issues.

Considering accessibility and usability of initial design

Firstly, I thought that the grey text on the text inputs was contrasting enough to meet accessibility guidelines, which is a NFR of the app. But using a contrast checker, I found this text did not meet the Web Content Accessibility Guidelines (WCAG) international standard, which meant it was not accessible enough. Using an online checker, I darkened the dark gray until it met requirements, where the hex colour code was #4A4A4A.

Another thing I noticed is that it was very difficult to distinguish headers and text from inputs and buttons. This can make the app



² <https://www.nngroup.com/articles/ten-usability-heuristics/>

less accessible and harder to use, against NFRs. In this version, it is not obvious whether the 'next' element is supposed to be a button or just encouraging scrolling.

Disregarding technical things, my gut reaction is that the very square and rectangular styling does not match the motif of circular and smooth movement as seen in the logo. It's a very jarring feel, and took me out of my flow.

Focusing on specific page designs

I chose to make detailed designs for four of my 'pages' since designing every page in high detail would be unnecessary – the sign up page, the welcome and the weekly emails, and the dashboard (explained more later).

Since I needed to make both a mobile design and a desktop design, I chose to design the sign up page and the emails because they would all look different to the rest of the app, and the dashboard, because it was the most complicated of my 'main' pages. The other pages, like the 'yes' and 'no' pages, just had the same stylesheet as the dashboard applied to them, so weren't worth the design time.

The dashboard is a page I decided to develop after I had finished the MVP but before I started design work, as stakeholders had requested a place to show more detailed statistics and communications from the team (the development process of the page will be explained in Section 3). This is good proof of my choice to finalise the design after MVP development as, if I had gone through the design process before realising I needed a dashboard, I would have had to redo the whole design to account for a more complex page.

Design research

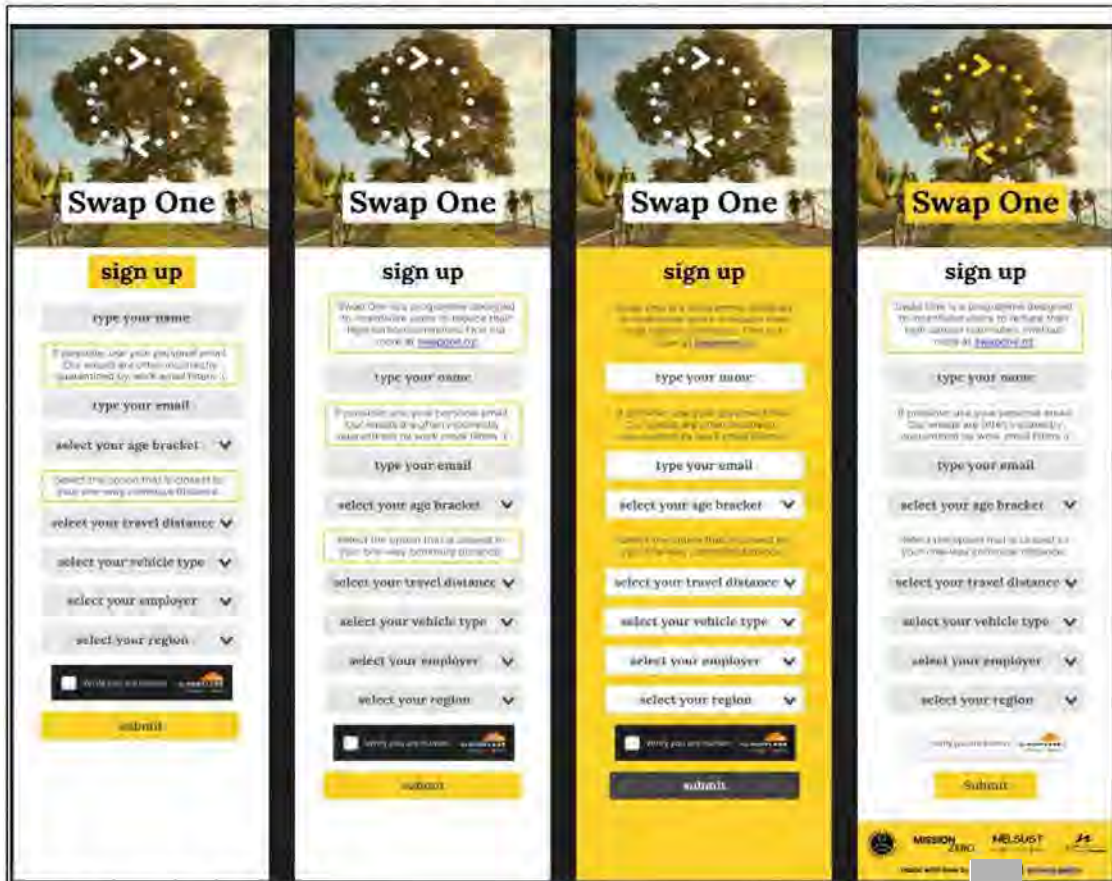
Before starting work, I did some design research. The project is sponsored by the Nelson City Council, so the app had to have a professional and official feel. On the other hand, the app also had to feel fun, modern, fresh, and welcoming, since we wanted users to enjoy using the website so that they kept coming back, inline with the outcome to incentivise users to use the application and thus commute sustainably. After some research, I found that there weren't many government or council adjacent websites that were designed for people who choose to be there, instead of people who have to be there.

The closest I could find were websites like the Lowdown, a government-supported mental health website, but the theme of this was far too serious for our app. Other than that, the Wednesday Challenge and Love to Ride's Aotearoa Bike Challenge seemed like the closest match. Reading through my notes from the start of the project, I found that the Wednesday Project's less cluttered aesthetic is more in line with the project requirements than the busy Love to Ride.

I also looked at the FluTracking survey, which works similarly with a weekly email allowing users to log in and report if they have the flu. Unfortunately, this has a boring, dated, and utilitarian design with plaintext emails, and does not seem to encourage users to engage.

Sign Up Page Design

Using UI design software named Figma, I designed four initial iterations of the sign up page, loosely inspired by the Wednesday Challenge and [redacted] work. I decided to take a mobile-first design approach (and come back to the desktop design later), because I expected that most users would access the page through their phone.



Initial Sign Up Page Iterations

For my first iteration, on the left, I mostly copied [redacted] work, only changing the aforementioned details such as input text colour and roundedness, as well as adding help text above specific options that needed them. However, I decided that this wasn't suitable since there was no space for programme information, and the sign up header looked too much like a button, violating the consistency and standards heuristic.

To remedy this, I designed the second iteration, where I removed the box around the header and added an infobox at the top describing the program, applying the help and documentation heuristic. Unfortunately, this created new issues. There was not enough of the brand colour so the website looked bland and boring. Moreover, the yellow infoboxes drew my attention first, ruining the UI hierarchy. Users would completely miss the title of the page and just look at the input information. In the third iteration, I tried making the whole page yellow in an attempt to make it more interesting. Getting rid of the infobox highlights made it even less branded and dull, but something had to be done. Unfortunately, the yellow was far too bright, and may

have scared away users. It made the page feel like a genuinely unpleasant place to be.

So, in the fourth iteration I restored the white background and focused on making as many other elements yellow as possible, including adding a yellow footer to tie the page together, while making the Cloudflare widget white to blend into the page better. This seemed like the first revision that looked good and might be fit for purpose.

After making these, I did more research into accessibility standards to make sure I was meeting my NFR. Whilst the design looks cleaner without labels and instead using placeholder text, as seen above, every other website³ I have found uses dedicated labels to allow for screen readers to understand the system, which aligns with the recognition rather than recall heuristic, since users now no longer need to remember the context.

As a result I added in labels, as seen in the image above. Since the placeholder is no longer critical, I reduced the input font size, allowing thinner inputs to fit the same content in less space. To achieve this I had to decide between question style labels ("How old are you?"), or simple labels ("Age"). Other similar applications use simple labels, but the question style labels initially felt more welcoming. In saying that, as the questions went on, they started to feel interrogatory, and began to have the opposite effect.

I wasn't sure which of these, if either were good, so I asked my friend. Firstly, it was suggested that that since name and age feel more personal than the rest they should be in question format, whereas the rest of the labels can be in the simple format. Because of that, they also suggested switching up the order so these two are together. That was really great feedback, and I agreed that it hit the right balance between professional and casual.

Secondly, it was also suggested that I make the label smaller and aligned to the right. They pointed out that this matches up with the other websites I have seen, and

What's your name?
type your name

What's your email address?
if possible, use your personal email
Our emails are often incorrectly quarantined by work email filters
type your email

How old are you?
select your age bracket

How far do you live from work?
Select the option that is closest to your one-way regular commute
select your travel distance

What type of car do you have?
select your vehicle type

Who do you work for?
select your employer

Which region do you live in?
select your region

Name:
type your name

Email:
if possible, use your personal email
Our emails are often incorrectly quarantined by work email filters
type your email

Age:
select your age bracket

Commute Distance:
Select the option that is closest to your one-way regular commute
select your travel distance

Vehicle Type:
select your vehicle type

Employer:
select your employer

Region:
select your region

Two Types of Label

Sign Up

Swire Dri is a program designed to incentivise users to reduce their high carbon commutes. Find out more at swiredri.nz

What's your name?
type your name

How old are you?
select your age bracket

Email
if possible, use your personal email
Our emails are often incorrectly quarantined by work email filters
type your email

Commute Distance
Select the option that is closest to your one-way regular commute
select your travel distance

Vehicle Type
select your vehicle type

Employer
select your employer

Region
select your region

Verify your email & location [+0649 400 0000](#)
ready to go

Submit

Form after Feedback from Friend

³ NZ Govt. standards also require this, saying *All inputs must have visible labels. Placeholder text is not an acceptable replacement for a label because it vanishes when users start typing.*

makes the interface less cluttered. When I tried this, it immediately made the interface look better, to the degree that I immediately decided I wanted to keep it.

They also did suggest that I change the capitalisation from *sign up* to *Sign up*. I thought this made the page look unbalanced, and does not match with conventions using title case for headings. However, during this, I also tried capitalising like *Sign Up*, which fit much better.

After this, I presented the design for this page to my stakeholders, including [redacted] the team's [redacted]

Aesthetic changes

Many stakeholder suggestions were made to improve the aesthetics of the design. For example, elements such as the header were requested to have yellow-on-white text, the labels were requested to be removed to make the page cleaner, and the link was requested to be made black to fit in better.

However, these all went against the NFRs of accessibility and usability. While these changes would give a cleaner visual appearance, all three changes would make the page harder to use and less accessible for people in need, so I could not make the changes.

One piece of feedback that I did accept was that I should remove the grey instruction boxes above the inputs while italicising the text. This made the page feel cleaner and less cluttered. However, it was also suggested that I put these under the input, which isn't great for usability, since users will read the text after they've answered the question. As a result, I kept the input label while changing the styling.

Refining the visual hierarchy

It was suggested that I remove the yellow box around the introduction text to make the page more readable. I agreed that this cluttered the page without adding anything, so I removed it. I also was asked to add a subtitle under the Swap One name, but since I felt that re-cluttered the page I decided not to keep it.

Another recommendation that was made was to reserve the Lora font for higher level headings, and use Inter for labels and inputs. Since [redacted] was far more experienced in graphic design, I deferred to



The image shows a final design for a 'Sign Up' page. At the top is a banner image of a tree with yellow arrows pointing in a circular path, with the text 'Swap One' overlaid. Below the banner is the heading 'Sign Up' in a large, bold font. The main text reads: 'Real change can start with just one trip a week. Ditch the car and walk, bike, bus, or work from home instead. Swap One is a programme to help you do this. It's easy. It matters. Find out more: [swapone.nz]'. Below this is a form with several input fields: 'What's your name?' (text input), 'How old are you?' (dropdown menu), 'Email:' (text input with a note: 'Please use your personal email. Work emails can block our messages and your rewards!'), 'Commute Distance:' (dropdown menu with a note: 'Pick the option that is closest to your one-way regular commute.'), 'Vehicle Type:' (dropdown menu), 'Employer:' (dropdown menu), and 'Region:' (dropdown menu). At the bottom of the form is a 'Submit' button. Below the form is a footer section with logos for 'MISSION ZERO', 'NELSUST', and '21', and text: 'Made with love by [redacted] Privacy Policy'.

her on that, and agreed that the page was more understandable as a result, matching up with the usability NFR.

Also, upon stakeholder request, I removed the yellow filter from the top image. While I thought the yellow made the page feel more cohesive, that was a matter of personal taste, and everyone else found it off-putting.

Cultural consideration

I also received feedback that the header image could sometimes cut off at people's heads. I was told that in Māori culture this is considered tapu, and as such I can't have it included – a critical oversight on my part. So, I decided to manually check before launch that at no screen size heads will be cut off by adapting any adaptive cropping systems. Cultural reviews should often be factored into design processes.

Desktop design

Next, I designed the desktop version of this page. I recognised that just stretching the page would look bad after a certain extent, so I decided to add margins in the lighter yellow colour.



Revisions of Desktop Header

However, I still needed to decide what to do with the header image. I tried stretching it or leaving the lighter yellow margin on the sides, but neither looked cohesive. I tried the darker yellow on the sides of the image and liked how that looked.

Email Design

A major constraint of the email system is simplicity, since most inboxes cannot handle complex styling. Because of this, I made a simple header with a logo in it, simple text with the information needed from the email, and then the footer from the sign up page. Since I need to be able to navigate from this email into the app, I also added links taking me to different parts of the website.

For the email that would invite people to log their trips, I added some stacked buttons in the area, but I kept the rest of the email the same.



Logging and Welcome Emails

I originally designed these with images, but my stakeholders thought they took up too much space when emails are meant to focus on just communication. I understand that while the images looked cool, they probably distracted the user more, and as working age adults, my stakeholders had better understanding of what they expect to see in an email.

Feedback on Email Design

Apart from a few phrasing changes, it was suggested that I put the bottom section in the footer in sentence case (*Made with love by [My Name] | Privacy Policy*). This made it more readable, and the only reason it was that way was because of the old lower-case styling of other elements, which were removed. I then also updated the sign up page design.

I was also asked to remove the sad faces from the no button and the unsubscribe link. While I felt strongly that the faces added character to the app, we also didn't want to shame users who decided the programme wasn't right for them, so in the end I removed them.

The last piece of feedback on these was that I needed some text informing users not to forward the emails. This was because of the trade-off we made to keep things simple by including links to their personal travel log from their email. Even though this was non-sensitive information I added a block at the end to explain this clearly.



Final Welcome Email Design

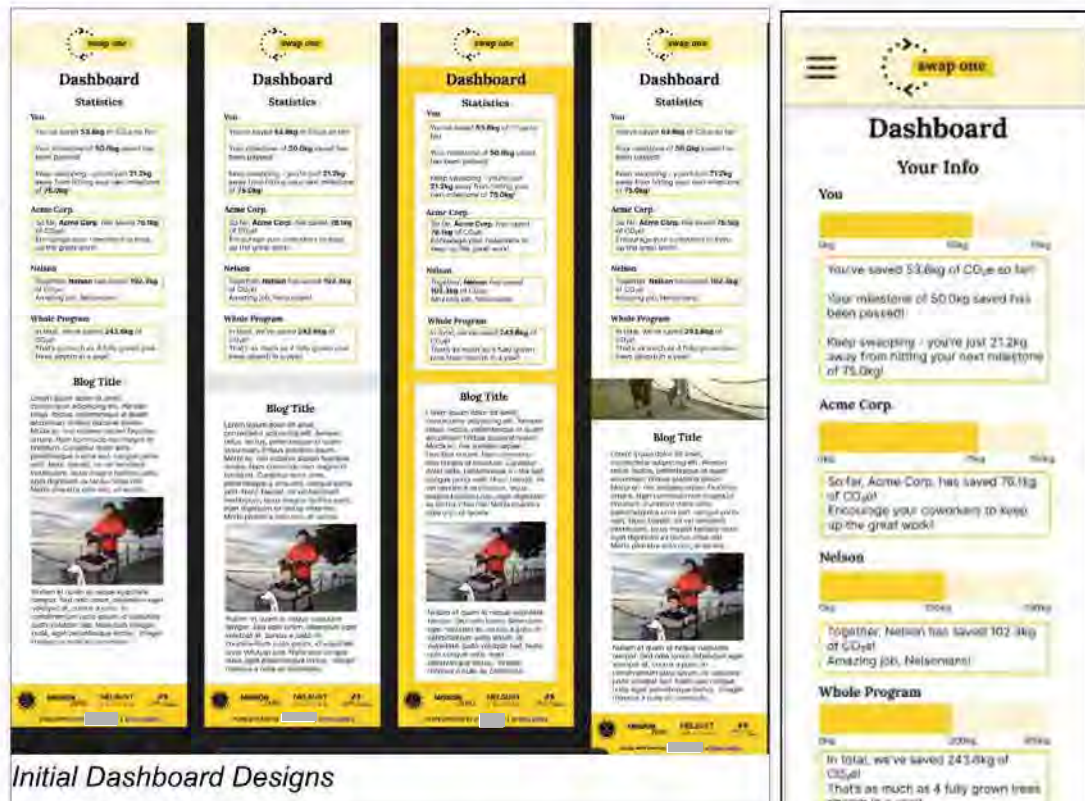
Dashboard Design

On the dashboard, I decided to reuse the header and footer structure from the emails, since I thought they looked good and would fit throughout the rest of the site. I needed to figure out how to show statistics and messages from the programme. I tried to separate the two parts, but I couldn't make it look good. Additionally, the statistics section specifically looked bland and boring, with just text to tell the user what they had achieved. So, I showed the designs to my father and asked him for advice on how to make the page more interesting. After talking for a while, we realised that the page needed some kind of visual element. Since we were already providing users with goals, I wondered how it would feel with graphs showing how much carbon they needed to save to get to their next goal. I thought this looked great, felt more interesting, and also made the numbers provided actually mean something to the users.

While doing that, I realised I could just separate the sections with a coloured rule, which also looked excellent.

One last decision I made on this page was to have navigation in the top bar rather than the footer. While having the links in the footer would match up with the email better, I thought there might be different usage patterns for email versus a web

experience. Email users expect navigation in the footer, whereas web page users expect a navigation bar, usually in the form of a hamburger bar. So, I added a hamburger in, which opens up to buttons linking to various other pages, supporting the aesthetic and minimalist design heuristic, focusing the interface on the primary content.



After presenting the page to my stakeholders, I got a lot of positive feedback and only two things that they thought I should change.

Firstly, it was suggested that there was too much bold on the page and it was distracting. I agreed with the distraction, so I decided to un-bold all the text in the statistics boxes.

Secondly, some stakeholders were worried that users would miss the blog article. I did consider putting the blog above the statistics, but ultimately decided that the statistics were more important to users, and if they saw cut off statistics, they would probably scroll down anyway.



Part 3: Software Development

3.1. Development of the MVP

In this section, I completed all eight MVP user stories, meaning I had a Minimal Viable Product and a functioning application.

Setup, Database Design, and Sign Up

Django, HTML, and CSS setup

At this stage of the project, I had never used Django. I first spent an hour reading through Mozilla's Developer Network tutorial on Django. This tutorial was based around building a library website, which explained Django concepts while showing me how to program my application well. Throughout the project, I referred back to this documentation whenever I was stuck.

After reading, I decided that I would attempt to get an HTML page rendering through Django, with CSS working. Firstly, I installed a dependency manager called Poetry, which allowed me to specify which packages I needed. Based on this, I could run a single command to install those on any other system, which paid off when moving to the production server.

After installing Django to my system, I created a blank project and launched it to test if everything had worked. I then programmed a view (Django code to interact with what the user sees) to render a test HTML page, and connected it to the URL `/signup` to make it work.

Welcome:

to the sign up page! Here, you can sign up to the commute program!

As proven by the red text, HTML and (external) CSS are functioning on this webpage!!!

First Page on the Website!

Database

The next step was to define a user in Django's object relational model (ORM), a way to work with the database through Django. Using the ORM meant that, even though I was using SQLite at this time, it was possible to switch database later.

```
class User(models.Model):
    """Model storing users."""
    uuid = models.UUIDField(primary_key=True, editable=False, default=uuid.uuid4)
    email = models.EmailField(max_length=120, help_text='Enter your email')
    sign_up_time = models.DateTimeField(editable=False, default=(datetime.datetime.now))
    age_group = models.CharField(choices=[("<13", "Less than 13"), ("13-17", "13 - 17"),
    gender = models.CharField(choices=[("male", "Male"), ("female", "Female"), ("other",
```

Initial User Class

I defined the User class with five fields: a UUID (Identifying code for the user), an email address, the time the user signed up, their age group, and their gender. I decided to make age and gender multiple choice options instead of allowing them to be typed in. By grouping age into broad categories, I thought users would be more comfortable with sharing this information. Additionally, it's much easier to provide error checking and data validation with only a few options rather than checking strings or numbers.

Based on this, Django generated a new SQLite database file, which had the correct fields.

Sign up

By implementing sign up, I would be completing the user story: **As a prospective user, I want to join Swap One so I can reduce my emissions.**

Using Django's form system, I defined the questions to ask, making sure to limit the responses to the acceptable database inputs. I then linked this form to the HTML markup, before collecting the response from my view and creating a new user through the ORM. After some brief testing, I determined that this feature was fully functional.

I also added a confirmation page after the sign up form is submitted, so that the user knows that the system has succeeded.

All acceptance criteria for this story were met now – the sign up system asks for user information, the information is stored so it is accessible to other system components, and the system confirms sign up to the user.

Welcome:

to the sign up page! Here, you can sign up to the commute program!

As proven by the red text, HTML and (external) CSS are functioning on this webpage!!!

Input your email:
Select your age group:
Select your gender:

First Functioning Sign Up Page

Security and error handling

Before moving on, I wanted to make sure the form worked. I submitted a sign up request, viewed the database in a database viewer, and saw a new user was created! After a bit of testing, I found that if the database was in use or if the same email was used twice the system would display a 500 error code. I implemented logic to catch that and used the Django form error system to provide a non-catastrophic error for the user, allowing them to try again without crashing the app.

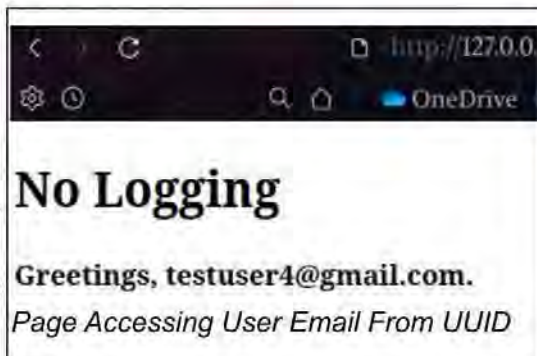
I then wanted to make sure the Django form system was secure, so I used a tool called Burp Suite to override client side protections built into the HTML and test the server side handling. For example, using the regular UI, the user would be unable to submit a form with a malformed email. Using Burp Suite, I could send that malformed email into the system, and was pleased to find that the form responded with a readable error message and prompted the user to try again.

Core User Journey Features

Trip Logging

By implementing trip logging, I would be completing the user story: **As an app user, I want to log my sustainable trips so I can participate in the programme.**

To implement this system, I decided there should be two URLs to access the pages



– `/yes/[UUID]` and `/no/[UUID]`. These URLs would be available in the emails as buttons, and when clicked would take the user to either the 'yes' or the 'no' page. Then, the system could read the UUID from the end of the URL, identifying the user and providing the correct content on the page.

On the back end, the system accessed the UUID and created an object with the user's information, based on that. This information was then passed to the HTML templating engine, which put the correct information (in this example, the user's email address) into the markup.

To get logging working, I made a second table in the database for logged trips through the ORM, linked to a user via a many-to-one relationship. This means that every trip must have exactly one user associated with it, whereas a user can have any amount of trips. This means that trips are their own objects, so they can store more information and be changed without affecting the user object. When testing this, I found that if I deleted a user but their trips remained, the whole app would break. Due to this, I implemented a database rule to cascade deletions, meaning when a user was deleted their trips would be too.

After implementing a basic web form in the same manner as the sign up page for 'yes' logging, asking the user approximately how far they had travelled and what mode they used, I realised I needed to change my trip model to support 'no' logging. There are three states for responses – responded yes, responded no, or didn't respond. If the user responded yes, I wanted to collect trip information, if they responded no, I wanted to find out why not, and if they didn't respond, I can't collect anything. I then made changes to the trip database model, allowing the mode of transport to be blank, the distance to be zero, and adding a new text field that can be blank. This way, if the response was yes, there would be a value for mode of transport, and if the response was no, the form will not show any of the other questions, and automatically fill them in.

I then implemented the 'no' logging page form with a text input, and added links between the yes and no pages.

No Logging

Greetings, fhjsahdklfhkj@dfs.fd.

[Oops! I meant to press yes!](#)

yea i was a bit silly and decided i actually wanted to pollute sorry :(

also i hate trees

Why didn't you do your trip this week?

No Logging Page

At this point I had met both acceptance criteria for this story – the system recorded appropriate trip information and the user response was added to the database.

Unsubscribe

By implementing unsubscribing, I would be completing the user story: **As an app user, I want to be able to unsubscribe from the system so I can retain control of my information and stop participating.**

I decided that I wanted to retain some data for reporting when a user unsubscribes, especially their trip information so that I can still get an accurate measure of carbon savings. So, I decided to duplicate the Users and Trips tables in the ORM to DeletedUsers and DeletedTrips, while removing identifying information such as email addresses. I quickly made a new page linked from the others on the site for this feature.

I next made another form asking the user to confirm they wanted to unsubscribe. The view handling that page was then set up to copy the user and any trip objects to the new tables. I found that since cascading delete was active, I had to ensure to copy both the user and the trips over before deleting the original.

I had met the acceptance criteria for this story – the unsubscribe subsystem was easily accessible, and all identifiable data was removed from the database when unsubscribed, meaning this feature was complete.

Unsubscribe

Greetings, ba@balls.com.

If you unsubscribe, all your progress will be lost.

Do you want to unsubscribe from the program?

Unsubscribe Page

Statistics, Admin, and Integration

Emission statistics

By implementing the feature of user statistics, I would be completing the user story:
As an app user, I want to be able to view emissions saved so I feel motivated to keep commuting sustainably.

I needed to show users how much carbon they had saved, but I also knew that calculating it would be computationally intensive. To calculate the emissions from trip objects would require iterating through every trip in the database and running a calculation, locking the database in the process. Once the trips logged reached high numbers, the slowdowns would become intolerable. As a result, I decided to store the user's carbon savings separately.

I created a new field in the users table storing their emissions saved. Each time a user logged a trip, their carbon saved from that trip would be calculated and then added to the value already saved. This was less precise and more prone to error. For example, if for some reason there's an error in the script when the response data is saved and then the user is prompted to submit again, there could be double-counting. However, the immense savings in database access were worth it, since the statistics are already just an approximation due to inaccurate distances. This was something the stakeholders agreed didn't need to be perfect.

I stored these values as grams of CO₂e saved, to avoid using imprecise floats. In the MVP version, I assumed that every trip is human powered with no carbon output and that otherwise the user would've driven a petrol car, using a value of 0.243kg of carbon emitted per km driven. This is based on the Ministry for the Environment emission factors for 2024. Conveniently, this converts to 0.243g of carbon per metre travelled in an average petrol car.

When showing the value to the user, I wanted to divide the value in grams by 1000 to show it in kilograms. However, Django does not natively support division in HTML templates, so I installed a module named `django-mathfilters` that allowed me to perform my division. The emissions saved value is shown on both the yes and no pages above the logging form.

This meant that I had met most of the criteria for this story, with emissions saved calculated based on data from the database and the emissions saved visible on the logging webpage. I had not yet met the requirements of the emissions being visible in the weekly email, as these had not been implemented yet.

CAPTCHA

I knew that bots automatically filling out forms are common on the internet, so some sort of CAPTCHA was needed. I wanted to avoid difficult CAPTCHAs like Google's ReCaptcha, where the user clicks on signs or vehicles, because they are a genuinely poor user experience. After doing some research, I found a provider named Altcha, who has an open-source CAPTCHA with a module designed for Django. This captcha worked on the Proof Of Work methodology, where the user's computer was sent a cryptographic puzzle to solve. It only takes a couple of seconds to complete, but uses enough computing power to make large-scale attacks infeasible.

Whilst it would not work if a malicious actor decided to spam the sign ups of the programme specifically, automated web attacks just pick random websites, and this CAPTCHA will serve well enough for the attacker to move onto an easier target.

However, when I implemented this, I realised that the few-second wait after the box had been ticked was annoying to users. So, I set the CAPTCHA up to start as soon as the page was opened, so by the time the user had completed the form, the CAPTCHA would have already been verified.

I then tested this using Burp Suite by changing the provided CAPTCHA token, and the client was provided with an appropriate error message.

Email system

By implementing the email integration feature, I would be completing the user story: **As an app user, I want to be reminded to do a sustainable trip so that I don't forget to commute sustainably**, as well as meeting the final criteria from the statistics story.

I first made and configured an AWS account and set up my test domain to accept sent emails, before writing a test Python script to verify that I could send emails. I then decided that I wanted to get my styled emails working. Since I was using MJML for my email markup, I decided it would be more efficient to pre-compile MJML to HTML templates, so that I did not need to include an MJML



compiler in my app. I developed a basic email message, which was received correctly in my inbox.

The logo on the top of the email did not properly render, but I realised that this was because Gmail caches and reduces the quality of images, and the server hosting the image was not connected to the public internet yet.

Next, I linked the template into the sign up system to send a confirmation email. I built a set of functions to send email through the API configured properly, with only subject, email, and content parameters needed, which saved a lot of repetition for every email sent.

```
def send_email(recipient, subject, html_body, uuid):
    aws_email()
    sender = 'no-reply@testing123.my'
    unsubscribe_url = "http://127.0.0.1:8080/unsubscribe/" + uuid
    client = boto3.client('ses', region_name='ap-southeast-2')
```

Start of send_email Function

Initially, I was going to build my own templating system to fill in information like the user's UUID in links. I built a quick prototype of this, but found that parsing the HTML tree myself to replace <variable> tags, while elegant, was slow and inefficient. I then realised that Django's templating system which I was already using for web pages could be repurposed, which ended up much more stable and maintainable. This just required an input of the variables needed, and provided me with an HTML string which could be passed to my email functions.

I then built two email message designs in MJML – one for logging trips weekly, and one for confirming that the user had signed up. The logging email featured two buttons, which linked to the yes and no pages made previously, an unsubscribe link, and statistics about how much carbon the user had saved.

The final element of the email system was the task scheduler. I installed a module called django-apscheduler, which allowed me to specify a function to be run at specific times, and also let me trigger on demand through Django's error console for testing. This was scheduled for Monday at 7:30 AM at this stage. For now, this was asynchronous, meaning only one email could be sent at a time with around a 3-second wait for each email. This was fine for the MVP stage of the project.

By completing this, I had completed the acceptance criteria for my reminder story. Users were sent a reminder notification weekly, and could access the logging system from the reminder. I had also finished the requirements for the statistics story, since emissions saved were visible in the weekly email.

File export

By implementing the file export feature, I would be completing the user story: **As a programme coordinator, I need to be able to access commute data in an Excel compatible format.**

When I asked my stakeholders how they wanted this file formatted, they said that every line should be a new trip, with the user's information in the columns of this. I decided to use the open CSV spreadsheet file format for this, since it was very easy to interact with from Python.

Using Python's csv module, I built a function to iterate through every trip in the database and write the data to a row in the spreadsheet. During this, I set up the

system to look up the user and add the applicable information to each row, primarily so that winners could be identified. The system also iterated through the deleted trips table, showing 'DELETED' for any removed user information. I also implemented error handling to this system, so that if for some reason something didn't work, rather than crashing the whole spreadsheet it would just print the error message into the current cell of the spreadsheet and move on.

Since I needed to automatically send this file to stakeholders, and the file could become too large to be sent as an email attachment, I decided to upload it to a cloud storage system. I initially picked AWS's S3 storage, although this was changed to Azure when I later moved email providers.

By configuring my AWS account to allow uploads, I implemented the upload logic to the end of my function. With this, I received a URL to access the data, which I had configured to expire in 24 hours to prevent the data being leaked. This URL was then set up to be emailed to the stakeholders who requested the data.

I then set up another APScheduler job to run this function once a day, making sure that stakeholders always had an active link to access the spreadsheet database.

With that, I had met the acceptance criteria for the two relevant stories – the database was exported to a user-readable Excel compatible file, the spreadsheet was uploaded to cloud storage daily, and the exported Excel file was easily able to pick a random user based on trip numbers.

Beta User Interface (UI)

By implementing the next iteration of my UI, I would be completing the following feature: **As an app user, I want the design to be simple and intuitive so I don't give up on using the app.**

The initial UI was purely for testing (an alpha version), and was not suitable to be shown to stakeholders and testers. Despite this, the scope of the app was not set, so I didn't want to implement the final UI. As a result, I decided to clean up the interface and markup to set a good foundation for the final UI. This resulted in a beta version of the interface that allowed stakeholders to use the app in a testing manner.

I decided to use CSS's Flexbox system to build the UI. Flexbox is a system that makes it easy to build a responsive website, meaning it adapts to different screen sizes. To improve maintainability, I decided to have all my CSS in one file, so one set of styles are applied to the whole page.

While standardising my markup with CSS classes for consistent styling, I realised that Django's forms system didn't natively give me a way to style the form elements. When a form is defined in the Python file and then selected to be embedded, it does not have the classes needed to appropriately style different inputs of the form. So, in the forms file, I manually overrode the class system and applied three of my own classes to each – the name of the input, the type of input, and just 'input'. For example, the email input had the classes *email short-text input*. This allowed me to target any type of input element in the forms, while not requiring long chains of different classes in the stylesheets.

Another issue I had to deal with was that every input was wrapped in a div element. This was a problem, since I wanted to use the 'width: 100%' style to make the

element take up the whole width. Since this is relative to the parent element, which itself had a set width, I could not responsively achieve this. Thankfully, I could style its parent, so I could use a CSS child override to set the width to 100%.

During the styling, I also refactored my markup to use semantic tags like <header> and <main> to specify the types of content in various areas, since this helps improve accessibility for screen readers, which is an NFR.

Based on this, I had met the acceptance criteria for the UI story – the page had a consistent user interface, the page UX met Nielsen's heuristics, and the page UX met WCAG accessibility guidelines, so this feature is complete.

3.2. Development of the Second Iteration

I met with stakeholders to discuss the MVP of the app. It was very positive, but I was provided some feedback and feature requests in three different areas – User Onboarding, Logging, and Dashboard.

I converted this feedback into a new set of user stories so that, when implemented, the app would be feature complete, only requiring testing, the final interface, and to be loaded onto a server.

User Onboarding

As an app user, the system should ask me how far my commute is when I sign up so that I don't have to put it in every week.

I was asked if we could move the travel distance from the logging page to the sign up. By moving this, users would have a much quicker logging experience, and since their commuting distance would be the same every week, nothing would be lost. So, I added a dropdown on the sign up page and removed the one on the logging page, while keeping the same distance options, before storing the data in the user object to allow it to be accessible by other subsystems.

When trips were being logged, I then modified the system to look up the distance value from the user object before storing it in the trip object. The advantage of storing the distance each time is that if the user later changes their commute distance their old trips will stay the same, improving accuracy of data. Additionally, when updating the carbon saved value, I configured the system to use the new distance value.

We also discussed changing the dropdown with ranges of distance to a free-input text box, so users could specify the exact distance. I decided that keeping ranges would be better, since users often don't know exactly how far away they live from their workplace. While more accurate, the free input would induce decision anxiety in some users, and this could cause them to skip signing up altogether. The collected data does not need to be extremely accurate, so ranges work just fine.

The acceptance criteria for this story were the user being asked for their commute length on sign up, that length was saved to the database, and that the length was being used to calculate saved carbon, so this story was complete.

As an app user, I want the system to refer to me by name, not my email, so I feel welcomed by the app.

It was pointed out that I was referring to users by their email addresses quite frequently throughout the app. This could be unwelcoming, so I decided to ask users what they would like to be called when they sign up and display that wherever possible, primarily in emails. This is also shown at the top of the logging page.

This has an additional deliverability benefit, since email inboxes are much more likely to allow messages to pass spam filters if they use the user's real name, as it shows that they are trusted enough to have this information

The acceptance criteria for this story were the user being asked their name, the name being saved to the database, and the user being referred to by that name throughout the app, so this story was complete.

As an app user, I want my carbon savings to be calculated correctly so that I am informed about my impact.

I was asked to collect information on what kind of car people used, so that I can accurately calculate the carbon that they've saved. This makes sense, since diesel cars emit far more than electric vehicles, so knowing this information would drastically improve the accuracy. Because of this, I added this question to another dropdown on the sign up page, through the views, and stored the data in the user object.

On the back end, I decided to store the emission factor of the user's vehicle rather than a string describing it. The emission factor of a vehicle is a number representing the number of

grams of carbon that the average vehicle in that class emits per kilometre travelled. These numbers are set by the Ministry for the Environment every year, and I used the 2024 numbers, as the 2025 numbers had not been released. For example, a petrol car outputs 243g/km, a diesel car outputs 265g/km, and an electric car only outputs 19g/km.

When logging a trip, I changed the view, so that instead of assuming the vehicle is a petrol car and using 243g/km, it multiplied the distance value by the stored emission factor to provide an accurate carbon saved value.

The acceptance criteria for this story were the user being asked about their mode of usual transport, the emission factor being saved to the database, and that value being used to calculate trip savings, so this story was complete.

As a programme coordinator, I want to know the workplace the users work at so I can report to businesses how much carbon they have saved and As a programme coordinator, I want to know what regions the users live in so I can report to specific organisations.

Programme coordinators wanted to be able to identify the users' workplace and region while being able to add new ones if needed.

Unlike my other dropdowns, I knew that these would be changing frequently since new regions could be added to the programme, or new businesses could sign up. As a result, I came up with the idea of defining two new models for employers and regions, and reading the options for the dropdowns from these models.

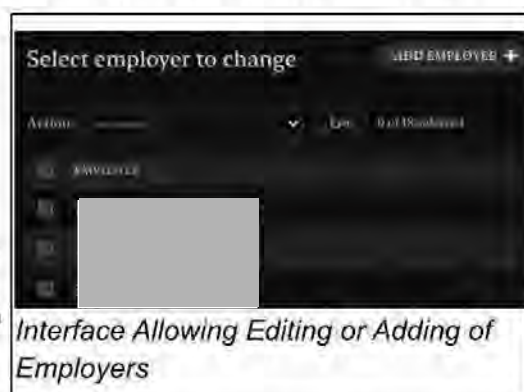
When defining the sign up form, I could then set the *queryset* to all objects in the respective model. One usability issue that occurred here was that by default the employers or regions were ordered alphabetically. Since there were many options, the 'None / Other / Prefer Not To Say' option ended up in the middle. As a result, I created an override in the Django form, manually setting the priority of the *Other* option to 0, while leaving the others at 1. This meant that 'None / Other / Prefer Not To Say' was at the top, preventing users from having to hunt for it.

Now that the options were linked through the model, they could be edited through the web-based Django Admin Console (accessible at */admin/*), meaning the Django UI could be used to add options to the dropdowns.

The acceptance criteria for these stories were the user being asked to select a region/workplace, the region/workplace being saved to the database, and the

```
vehicle = models.PositiveIntegerField(choices=[(243, "Petrol"), (265, "Diesel"), (19, "Hybrid"), (98, "Plug-in Hybrid"), (19, "Electric"), (243, "Other / Don't know")])
```

Code Defining the Vehicle Model



regions/workplaces being editable by programme coordinators, so these stories were complete.

Logging

As an app user, I'd like to be able to log my trips in other sustainable forms of transport, such as an EV or carpooling so I don't miss out on the programme.

The stakeholders reminded me that EV travel and carpooling were now classified as sustainable for the purposes of the programme. As a result of this, I now needed to implement these as options for logging a sustainable trip.

However, since both of these modes emit carbon emissions, I needed to subtract those from the savings that were being calculated. After adding the options to the dropdown, I made a dictionary, allowing me to look up the emission factor of each mode, adding EVs, carpooling, and also buses, since the emissions from those were previously not counted. Since there is no way of getting enough information about carpooling's savings, I decided to cut the user's vehicle emissions in half, assuming they were getting a ride from one person, thus splitting the total emissions between the two.

Once I had an emission factor, I could then subtract that from the user's car emission factor, to get an emissions saved factor. This was then multiplied with the distance in the same manner the original factor was, meaning these modes were taken into consideration.

I then did some brief testing to check if my carpool calculations were correct. I logged a carpool trip on a test account, with the distance at 50km and the car emission factor for a hybrid: 98. $98 \times 50 = 4900$, but since this was a carpool, the value was halved to 2450g of carbon saved, which is exactly what the system said!

The acceptance criteria for this story were carpooling and EVs added to the logging options and the carbon emissions of the trip calculated and subtracted from the regular trip savings, so this story is complete.

As an app user, I want to log multiple trips in a week at once so I don't have to go back to my emails and re-click on the link.

Another piece of feedback is that if a user was swapping trips multiple times a week, it was a pain to go back to the yes link from the email and log trips all over.

I decided to add a number input asking the user how many trips they had completed, with a 14 trip limit. This number was chosen, since with 7 days in a week, this was the most that could be achieved. I also put an information box on top of this explaining that a trip meant one way, to avoid any confusion.

This value submitted was then added to the trip object. With this change, a trip object was no longer the same thing as a trip, but rather any number of trips between 0 and 14. This was unfortunate, but with the existing architecture of the system, there was no easy way to add multiple trip objects in this situation. However, this did make the trips export spreadsheet more compact and understandable, since identical trips were now grouped together.

The trip quantity value was then multiplied by the trip emissions before being added to the total emissions.

The acceptance criteria for this story were having a way to select the quantity of trips, the user being informed that a trip means one way, the carbon saved value multiplied by the number of trips, and the quantity of trips saved in the database, so this story is complete.

Dashboard

Creation of the dashboard

The stakeholders had asked for multiple new features where information is shown to users. While I could have put this information at the bottom of the logging pages, this page was already becoming cluttered.

So, I decided to make a new page, the dashboard. This page was set up to be a hub for information. I set up a redirect, so users would be taken there automatically when they completed their logging instead of the old confirmation page. By requiring the user's UUID in the URL, the server could authenticate the user and send the client relevant user information, such as statistics or their personal links to other pages.

As an app user I'd like to be able to see how much greenhouse gas emission has been saved and what that means so I can understand my impact

In the dashboard, I wanted to show the user how much carbon they had saved, how much their employer had saved, how much their region had saved, and how much the whole programme had saved.

I decided to reuse the models I had made for employers and regions, and added a carbon saved value to each object. I then modified the logging views so that each time a user saved carbon, it was added to their region and their employer's carbon variable. I also made another model, 'All', which was used to store the whole programme's emissions.

Since I wanted this to be flexible and easy to change later, and it was much easier to have fine control through JavaScript compared to the Django Templating Engine, I decided to display the statistics through client-side JS code. I firstly configured Django to pass all the relevant variables into a script tag in the markup, including the names of the user's region and employer, as well as the carbon all four entities had saved.

Next, the JS code filtered through these and displayed text for each, along with a motivational message. If the user had selected other for either their region or their employer, these would automatically not be shown.

For the whole programme message, I decided to use a tree analogy. Since a single pine tree in New Zealand absorbs, on average, 54.3 kilograms of carbon per year, I divided the emissions to show how many trees worth had been saved. While I did consider showing this for the other entities as well, I felt like this would be discouraging if a user had skipped a trip a few times, and only saw half a tree.

The acceptance criteria for this story were showing carbon statistics for user, employer, region, and programme, and showing an analogy for carbon emissions, so this story has been completed.

As an app user, I want to be recognised when I meet my commute goal so I feel motivated to keep participating

To recognise users for meeting goals, firstly goals had to be assigned. I considered allowing users to set their own goals, but realised that they would struggle to set realistic numbers if they had no knowledge about emission rates from different vehicles. So, I decided to set these numbers myself, with goals going up roughly exponentially from five kilograms to 100 tonnes, to make sure everyone is always working towards a goal. These goals are stored in an array, and the program iterates through to find the user's next goal.

You've saved 32.2kg of CO2e so far!
You've hit the milestone of saving 30.0kg!
Keep swapping - you're just 7.8 kilograms away from hitting your next milestone of 40.0kg!
So far, Acme Corp has saved 785kg of CO2e!
Encourage your coworkers to keep up the great work!
Together, Nelson has saved 2495kg of CO2e!
Great job, Nelsonians!
In total, we've saved 3456kg of CO2e!
That's as much as 64 trees absorb in a year!

Statistics Section of Dashboard

Then, text is displayed showing what the user's last goal was and what their next goal is.

The acceptance criteria for this story were users being presented with a goal, and the system recognising users who have achieved their goal, so this story has been completed.

As a programme coordinator, I want to be able to insert blog content into the app so I can communicate information to users.

Coordinators wanted to be able to communicate blog posts to users, and I decided the dashboard, under the statistics page was the best place for this. Users will naturally end up here, and will see the post.

I needed a way to allow the coordinators to edit the blog posts, and after consideration I decided to use the Admin Console. By making a new model for posts, I could then access the most recent and display it at the bottom of the page.

The only issue was that the coordinators needed to be able to format the text, add links, and include images. So, I decided to use markdown. Markdown is a standard where characters such as asterisks or hashtags are used to define text formatting.

After researching, I found that there is a markdown formatting module for Django, where a markdown widget is put into a form, and then appropriate HTML is returned. As a result, I added the markdown widget into the blog model, and created a *formatted_markdown()* function which returns the formatted HTML. I could then embed this HTML into the

HELLO PEOPLE!!!!

This is an **epick** post by [REDACTED]

we have content here like lists

- *Italicized item (cool)*
- **Bold item (scary)**
- *Third item (somehow scary but cool)*

cool quote by a cool person (the cool person is [REDACTED])

also you can link to things like [this page](#) (a very cool page) and...

A Blog Post Showing Text Formatting

main HTML template for the page, giving me coordinator-controllable blog post functionality.

I also implemented image functionality. The markdown module had automatic drag-and-drop image support, so I configured Django's media directory to also be served by the development server and passed the path through to the module configuration. This meant that a coordinator could drop an image into the markdown box, which gets uploaded to the media directory and is then automatically added to the HTML of the dashboard.

The acceptance criteria for this story were coordinators being able to add formatted text to a blog post and this post being shown to users when using the app, so this story is complete.

Bug Fixes and Tweaks

While the additional user story features were now complete, I also encountered various issues and things that needed to be changed throughout this process.

For example, the dividers I was using in my email apparently are commonly used by spam emails, so I removed them to improve email deliverability. Another example is that I changed the format of dates in the generated spreadsheets to *YY-MM-DD HH:MM*, since I'd found that Excel could not interpret the existing format.

Mobile compatibility

While the app worked perfectly on my PC, I decided to take a look at how it worked on mobile phones. On an Android device there were no issues, but when I used an iPhone running iOS, the page was rendering incorrectly, with an unexpected zooming effect when clicking on an input, as well as all input text being blue. Moreover, my Altcha CAPTCHA was not working at all on iOS, with it refusing to pass.

I thought that this might be an issue with WebKit because this was occurring on all browsers on my iPhone, and I knew that on iOS all browsers use the WebKit engine under the hood, as opposed to Chromium or Gecko on Android and most desktop browsers. However, when loading up the page on a desktop WebKit browser, neither of these issues occurred.

After some research, I realised that the blue text and zoom were intentional features on iOS input text defaults to the system accent colour and input fields smaller than 16px zoom in to improve readability. Since every other browser defaults to black input text, I hadn't assigned it with CSS, and doing this fixed the issue. Likewise, increasing the input font size from 14px to 16px resolved the zooming issue.

No matter how much research I did, I couldn't identify the issue with the Altcha CAPTCHA option, which seemingly had no mention on the internet but occurred on every iPhone I tested. My conclusion was that it was an issue with the third-party Django module I was using, and since I did not want to write a compatibility module from scratch, I decided to switch to another CAPTCHA provider.

After researching, I decided to use Cloudflare's Turnstile captcha system. The only alternatives were click-based CAPTCHAs like ReCaptcha or hCaptcha, and Cloudflare allows the usage of up to 20 CAPTCHAs on a website for free. Additionally, Turnstile has a Django module written by a Cloudflare employee, so it should be more reliable.

The module was a single-line swap per form. I also configured it to operate in invisible mode, meaning that users don't even have to know there's a CAPTCHA – it just solves itself automatically.

```
captcha = AitcheField(debug=False, auto='onload')
captcha = TurnstileField()
```

Single Line CAPTCHA change

Email system

At this point, I was preparing the application for testing, which meant the system needed to be able to email non-approved users. With AWS, I had to pre-verify every address through the portal while on the trial tier, and I had to be accepted to the paid tier.

I sent a request to join the paid tier, introducing the programme and stating that the app was in alpha testing. Sadly, the programme was rejected, since according to them, "your use of Amazon SES could have a negative effect on our service". I'm still not sure why this happened, but we couldn't appeal it and they refused to tell us what the issue was.

This meant the programme was in big trouble, as the app couldn't function without email. Thankfully, an alternative from Microsoft's Azure was still available. Doing some research, I also found that on the free tier, a hundred emails are able to be sent to non-verified emails every day, which is more than enough for testing.

To ensure we would not be rejected again, I decided to wait until the last minute to apply for the Azure paid tier, since by then email mechanisms such as bounce protection would be running, and I could talk about them in more certain terms.

Free tier Azure accounts don't support custom email addresses, so I couldn't connect the *swapone.nz* domain to the email. However, I found that they had only implemented this block through the GUI, and by using the PowerShell interface this could be overridden.

On the back-end side, to implement Azure services I just replaced the boilerplate SES email sending code with boilerplate ACS email sending code and replaced the variables, which was surprisingly easy. As a result, I also decided to move my spreadsheet storage from AWS to Azure so that all my services were on the same platform, with similar results.

Edit profile

While I was developing the other features, I was also asked to make an easy way to edit the user's profile. I created a new page and a new view linked to the user's UUID. I then made a new form based entirely off of the user model, defining widgets for the inputs I would like changed. This means that when the form is opened the user's pre-existing data is there. They can then change whatever they want before submitting, and the new information is directly applied to the user object because of the link to the model.

To access this, I added a link in the footer of the email.

Twice-a-week log alert

I also wanted to warn people if they tried to log a trip twice in a week. So, I added a boolean to the user model storing whether the user had logged a trip that week or not, and passed that to the front end JavaScript. If the user had already logged a trip, I displayed a native JavaScript alert warning them, while allowing them to continue to log another if they want.



3.3. User Interface Testing

At this point, the app was now feature complete, but I wanted to see whether users understood the interface before I designed the final UI. I knew my final designs would be based off of the placement of elements which themselves were based on user feedback, so I wanted to make sure the way the app worked made sense to real people who knew nothing about the programme.

I developed a user testing script that introduced a scenario, imagining people had heard about Swap One and wanted to join the programme, and then identified the tasks I wanted to test. I found 8 people who represented the types of people we were targeting and read out the script to them. I prompted them with tasks like *Sign up to Swap One* or *From your inbox, record that you didn't swap a commute this week*, inviting them to verbalise their thinking so that I could understand what was working and what wasn't. Based on the way the users achieved the tasks, I then made the following changes to the app to increase usability – one of my NFRs.

Language and Phrasing

I learned that my choice of words and text throughout the app confused many of the testers, especially older users or people who had English as a second language.

As there were no instructions on how to use the app, a tester thought the Sign Up page was just informational and wasn't where they should sign up. Additionally, the question form inputs were that were labelled (*What would you like us to call you?*) confused some testers, especially non-native English speakers, meaning that this wasn't accessible. A tester also found the 'Sign Up!' button confusing, as well as being unsure about the meaning of the 'Commute Vehicle' field (asking what vehicle the user commutes with). Another tester, who didn't drive, was unsure about what vehicle to pick.

As a result of this, I decided to improve the language on this page. I added a 'Register' subtitle above the sign up form to improve clarity, renamed the button to 'Register' since it's more widely recognised, and rephrased Commute Vehicle to 'Vehicle Type'. I also changed the name question field to just 'Name'. However, when testing this, I found that users were now unsure of whether to use their first or full name. Either would work, but because they couldn't decide they ended up doing

neither. Since there were several situations where more clarity was needed, I introduced infoboxes in multiple places – at the start of the page explaining about the programme and how to sign up, above the name clarifying that they can say whatever they want, and above the vehicle informing users that if they don't drive, the programme is not for them.

Another point of confusion with phrasing was the screen after sign up, informing users that they'll *'receive an email within the next 5 minutes'*. A significant minority of testers believed that they had to click on a verification link in that email, as that is a typical user pattern. Since users of my app did not have to do this, I corrected the text to say *'You'll receive an email within the next 5 minutes, but this is all you need to do for now,'* which prevented user confusion in further testing.

Another issue that a tester ran into was being unsure about the distance input. Since the distances are fixed values and not ranges, they were unsure what to do when their exact commute distance was not on the list. I decided to use another infobox, since users ended up only reading them if they were stuck, otherwise ignoring them. The infobox informed them that they should just pick the closest value.

CAPTCHA Error

The testing also helped me find a major issue with my CAPTCHA implementation. Since I had hidden the CAPTCHAs in an attempt to make the experience smoother, I found that if a CAPTCHA failed, it would only provide a cryptic error message which testers could not handle, meaning that they could not use the rest of the app.

After some research and testing, I found that there were two issues causing this to occur. When the CAPTCHA failed the message was confusing, since the users did not even know there was a CAPTCHA.

But additionally, I found that these failures were occurring when a user pressed the register button twice while waiting. The root cause of this was that there was no visual feedback that the button press had done anything, so in the second-long wait time for the server to process, the user had pressed the button again.

To fix this, I made two changes. Firstly, I made the CAPTCHA visible by default. It would still automatically solve, but now the users had the required context to solve the problem themselves if an error was to happen. Secondly, I added some JavaScript code to block the button from being pressed while the response was loading, and made it change the text to *'Processing...'*. Now that users knew that their input had worked, the root cause had been addressed, and for any other errors the visible CAPTCHA allowed users to try again.

Change Sign Up Information

Consistently, testers could not figure out how to change their sign up information. I had a default link at the top of the dashboard, and a second link in the footer of the email, but they could not find these without being guided. Despite this, with unsubscribe links in the same positions, users always knew to look in the email footer to unsubscribe.

This is no doubt due to usage patterns people are familiar with. Unsubscribe is frequently a function that is accessible from links in the bottom of emails, but editing your profile is done from the main app. Furthermore, on the web, users are not used

to using plain-text links to get to settings, and expect a navigation menu like a hamburger.

As a result, I left the links in the email footer the same, while implementing a hamburger menu in the dashboard with unsubscribe and edit functions to make the most of user patterns.

3.4. Finalisation and Productionisation

Final User Interface Implementation

This is the stage where chronologically I went through the 2.4. Interface Design process. After that was completed, I needed to turn my finalised Figma mockups into actual web pages that matched my completed designs.

To do this, I firstly re-implemented the Sign Up page design. I decided that since this page looked significantly different from the rest, I would create a second stylesheet just for it. I set the hero image at the top to dynamically crop to support different screen sizes, while testing to make sure that no heads were severed. I used a media query to add margins to the page only if it was wider than 900px, so that the central interface part would never be more wide than that. After implementing the rest of the page, I also added the footer, which is just images inside `<a>` tags, using flexbox for spacing and sizing.

In my designs, the placeholders for both dropdowns and text inputs were a shade of grey, whereas the final input was black. This was easy to do with a single line of CSS for the text input, but turned out to be very difficult for the dropdowns.

Dropdowns don't have placeholders by default, but can be provided an unselectable option that is set by default. This could be set with JavaScript on page load, along with a class, 'placeholder'. To achieve the dynamic colouring, I had to iterate through all dropdowns whenever one was changed, and check if it had been changed to a non-placeholder option, changing the class. I could then set a CSS override rule to change the colour to gray only when the placeholder class was present.

The screenshot shows a web page titled 'Sign Up'. At the top is a hero image of a person climbing a tree, with a yellow 'Swap One' button overlaid. Below the image is the 'Sign Up' heading. The form contains the following fields: 'What's your name?' (text input), 'How old are you?' (text input), 'Email' (text input), 'Current Distance' (dropdown menu), 'Vehicle Type' (dropdown menu), 'Employer' (dropdown menu), and 'Region' (dropdown menu). Below the form is a 'Sign Up' button. At the bottom is a footer with logos for 'MISSION ZERO', 'NELSON', and 'NELSON'.

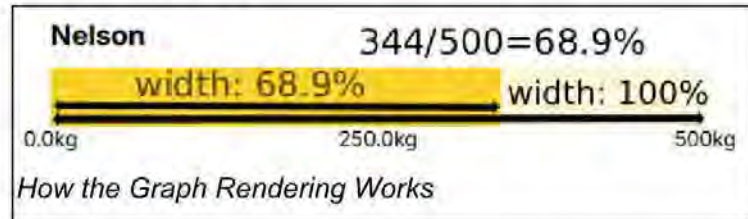
Finalised Sign Up Page

I then built the stylesheet for the rest of the pages, using the same margin and footer system. I added the hamburger menu to all of the pages, and used flexbox to space out the menu options. These changes worked flawlessly on all of the pages thanks to the markup standardisation I did in Section 3.2.

The last change to make was to implement the emissions savings graphs that showed progress on the dashboard. I wanted to avoid using a library or images to improve page

loading times and reduce dependencies, so I invented a unique method. I made the two graph parts div elements inside each other, with the progress bar aligned left. I worked out how much of the goal was saved by dividing the values of the savings by the goal, and used that percentage to set how wide the bar should be.

I gave each marker on the graph a set size and used that same percentage to place them in the right spot along the bar.



Server Setup

Installation of software

To move into production, I first had to load the program onto a server. Since I was already using Azure for email, I decided to use their \$12 per month micro server, since it was affordable and powerful enough. By logging into the service and configuring the server, I was then able to access it through SSH, a protocol to connect to a computer via the terminal. I chose to use the Ubuntu Linux distribution. While another distro called RHEL was closer to my development environment, this cost an extra dollar per month and I decided the cost was not worth it. Since at the time of development the Python version 3.13, which I had written my whole program for, was not available for Ubuntu, I had to download the Python binary files and build it myself. After that, I installed Git and Poetry.

Once that was complete, I was able to use Git to download the codebase on the server. I filled in the keys to access services I was using, and found that the app was working on the IP address I was using. I next installed Gunicorn, an alternate serving script to Django's development server, since it was a lot faster and more secure. I then wrote a bash script to start and stop both Gunicorn and the task scheduler, meaning only a single command was needed to start and stop the app.

Connection of domain

To finish the server setup, I needed to connect the domain to the server so I connected the IP address, using Cloudflare to route the system. I chose the subdomain *app.swapone.nz* to access the app because stakeholders also wanted a secondary website at *swapone.nz* for general information about the programme.

For the app to be secure and allowed on various networks, I needed to use HTTPS, which is an encrypted standard for website transmission. To do this, I decided to configure the routing tool Caddy to automatically provide me with a certificate,

allowing me to use HTTPS with a single line of code.

Email in Production

Rate limit handling

Azure had rate limits on emails, meaning I could only send a certain amount per hour. That meant I had to implement code to store messages when the rate limit was reached. When an email did not send because of a rate limit, I detected the error message and stored the email information in a new Django model, Backed Emails, along with a stored priority value. Since I wanted sign up confirmation emails to be sent as soon as possible, I gave these a priority score of 1, with logging emails at priority 2, and database dumps at 3. I then setup a scheduler that checked if there were any backed up emails every minute, and if so, ran an exponential backoff algorithm, waiting 2 seconds and trying again, then 4, then 8, and so on. This allowed me to test if the rate limit had been lifted or not without unnecessarily spamming Azure's servers. If it was detected that there was no rate limit, the emails would, in order of priority, be run one by one through the original send email function. This means that if the rate limit reappeared before all emails were cleared, the remaining emails would be put onto the backed up list again.

List hygiene

For Azure to increase our email send limit above 100 per hour, I had to prove that we had list hygiene measures in place. This means that if invalid email addresses signed up or an inbox stopped working, known as a bounce, they are unsubscribed automatically.

The first step towards this was detecting if emails had bounced. After research, I found that I could configure Azure to send a message to the server if an email had bounced. I set up a new view at the URL `/wh/bounce` and got the server to print out the message sent, since there was no documentation about how this was formatted. Since these messages were only sendable through HTTPS, I had to do this testing on the actual development server, as obviously my desktop and home IP didn't have a domain nor HTTPS. This meant that every change had to be committed to Git and viewed through the server command line.

Firstly, I had to return an authentication code to Azure to prove that I owned the server. But after that, when I tried to bounce an email, there was no response from Azure. After some research, I found that sending an email to a domain without email set up didn't count as a bounce so it was not logged.

Next, I tried with an invalid email on a domain with email, but it still didn't work. After hours and hours of testing and debugging, I finally figured out what had gone wrong. The server had successfully returned the information, but unrelated messages had overwritten this. The next time I attempted this, Azure automatically suppressed the email since it had previously bounced, which meant there was no server response. After hours of trial and error, I tried a different address, and it worked, helping me figure out what was going on.

Now that Azure was sending parseable messages to the server, I realised that there were two types of bounce – a 4xx error, meaning temporary issue, and 5xx, meaning permanent issue. I decided to unsubscribe emails that bounced with a 5xx error code, or ones that for whatever reason had not included a code, while keeping 4xx

email since they may work in the future. This was achieved with a simple email lookup in the system, and transferring the user object to the deleted users model.

There was one last issue – I had no way of knowing that the bounce message came from Azure and not some third party, meaning a bad actor could forcibly unsubscribe users if their email address was known. Reading Microsoft documentation, there did not seem to be a free way to authenticate this, and their key based solution was expensive and difficult to implement. However, since I can choose the URL for Azure to notify, I can add a parameter to the end. Since the message is sent through HTTPS, this is encrypted, so I can add a secret token to the end and check against it in the server. For example, the URL could be *https://app.swapone.nz/wh/bounce?token=secretcode*, and the message would only be listened to if the token was correct.

Rate limit increase

Since hygiene was implemented, I could then request a rate limit increase. 100 emails/hour was not enough, since if more than 100 users were signed up, the system would back up on the weekly send. Because of this, I requested an increase to 1000 emails/hour, providing much more detail when compared to the SES request that had previously been rejected, such as a detailed explanation of the hygiene system, images of the emails sent, and the unsubscribing system, as well as linking to the main initiative website.

Initially, the programme was rejected, due to a high bounce rate, which made me think the whole initiative was over – there was no other provider who I could go to. Thankfully, after a long and in depth discussion with multiple levels of service representatives, I managed to explain that the bounces all happened on the same day, and these were from the testing of our bounce rejection system. After a terrifyingly long consideration on Azure's part, a rate limit increase to 1000 emails/hour was accepted under the condition that future bounce rates remain low.

Sendability

Whilst the messages could now be sent at a rapid rate, they were often being sent to spam inboxes. To fix this, I used a mail tester, which ranked our deliverability score at 3/10 because of no proof-of-identity records such as DKIM and MX, broken links, no alt text on images, and no one-click unsubscribe.

I used Cloudflare to add DKIM and MX records, meaning that inboxes know that the server is authorised to send emails to prove they are legitimate. I also added missing extensions to links such as *https://* and *www.* when applicable to fix the links in the eyes of the system, and added alt text to all images in the messages, which brought the deliverability score to an 8.5/10.

To get to 10/10, one-click unsubscribe needed to be included in the email header, so that inboxes can display a button. However, I wanted the unsubscribe experience from the main link to display a message asking if the user was sure. Because of this, I checked if the POST request from client had *oneclick* in the header. If it did, they would be instantly unsubscribed, whereas if it didn't, they would be taken through the GUI.

These changes made the deliverability perfect, with no more emails going to spam from my testing.

Monitoring Systems

Sentry

Sentry is a web system that plugs into the app and records any errors, sending me an emails. Since the app was moving to production, I needed to know whenever anything went wrong, so that I could fix it. I plugged it into both the front-end JavaScript and the back end Django system, by simply entering a boilerplate script into the respective scripts, including the head of every HTML page. This meant that when I disabled Django's debug mode which provided error messages, I could still see when anything went wrong with the system.

Analytics

I also wanted an analytics tool in my system, so I could see how many users had accessed the page and from where. The two common services that I knew about were Google Analytics or Simple Analytics. While Google allowed me more information, including viewing users' exact actions, it used tracking cookies, meaning I would have to set up a disclaimer and disable it if users reject cookies. This seemed like it would interrupt user flow and lose trust, as well as being technically challenging, so I decided to go with Simple Analytics. By simply adding a snippet of code to the head of HTML pages, I could track users through a friendly dashboard while respecting their privacy.

Database Production

Crashes

A week before the app launch, I found that the task scheduler had crashed. Starting it up again, I realised that on average it was crashing around once per day. Using Sentry to view the tracebacks, I could decipher that the root cause was that both the website and the scheduler were trying to read the database at the same time, increased in occurrence by the bi-minutely check for backed up emails. Since Django prioritised the website, the scheduler timed out and crashed. This was because SQLite could only handle one read or write at a time. I changed some timeout settings and changed the mode to Write-Ahead Logging (allowing reads to happen at the same time as writes), hoping this would work as a quick fix. However, when checking Sentry in the morning, I realised that it had crashed within hours of the fix.

After researching and asking people I knew, I came to the conclusion that while using SQLite, I would have consistent stability issues. As this was a real application people were going to use, I decided to bite the bullet and spend hours and hours porting my entire system to PostgreSQL, which supported concurrent reads and writes, and would fix the issue. Since I had used Django's ORM to interface with the database, this was possible without major code rewrites.

I had two options – either subscribe to an expensive Postgres host, or run Postgres locally on the Azure server I was renting. Since I had only days before the launch, and no time to get a budget increase approval, I decided to run it on the server. I installed Postgres on the Azure server and added it to my startup and quit shell script.

I then changed settings deep in the Django configuration system and spend hours configuring permissions through the Linux command line, finally managing to connect the new database system.

I then ran into a snag where I needed to port existing users who had signed up for testing to the new database. After lots of research, I realised that the only way to do this was through Django's ORM, so I had to undo all my work with installing the new database to export the data. Once I reconnected SQLite and exported the data file, I then reconnected Postgres in a much quicker manner, and imported the file, adding the users back in.

I found that the crashes had stopped, so the port was worth the difficulty.

Backups

I wanted a way to backup the database, so that if something went wrong with the server, all the user data would be stored in a safe manner. Initially, I wrote this for the SQLite database, but with the port to Postgres, I had to scrap most of my work and write the system again.

I decided that I wanted to store backups both on the server, and in an Azure blob in the same way that spreadsheets are stored, so that if something went wrong I would still have a backup.

I found a command that made a copy of a Postgres database, and used Python's subprocess.run function to run the command and make a copy in the backups directory, naming it in the format `yyyy-mm-dd_backup.sql`. I then parsed all the other files' names, identifying which were older than 30 days and deleting them, before using the file upload system to store a ZIP of this directory in Azure's blob storage.

I linked this function to the daily make spreadsheet task, meaning that it would occur at the same time without any overhead.

With that, the app was ready for final testing before launch.

Part 4: Evaluation

4.1. Final Testing

Email Render Testing

To test my NFRs of **Portability** and **Reliability**, I wanted to test how my emails rendered with different email providers. To achieve this, I sent test emails of both kinds to *emailpreviewservices*, a provider that tests emails in 49 different inboxes, choosing this platform because of the availability of a free trial. By visually reviewing these email previews, I discerned that in most inboxes the emails rendered perfectly, with a few dark mode interfaces modifying the colours. However, these were still perfectly usable. In many inboxes, like Gmail, the incorrect font was used, as these services do not support custom fonts. However, an alternate serif or sans-



Email Render Test Results

serif font is used as appropriate, and as such this is acceptable. The only inbox that completely failed was Zoho Mail, but this appears not to support HTML at all, only has a fraction of a percentage point of users, and displays a fallback message asking the user to utilise another inbox.

Accessibility Audit

To test the NFR of **Accessibility**, I wanted to check that the app met the WCAG accessibility standards. To do this, I used the browser extension, WAVE, to test every page. I was given points for the semantic structuring of the markup achieved in section 2.1, text contrasts, and the responsiveness of the page, but the system found an oversight – I had included 'alt' text on my email images to improve deliverability, but had forgotten to add it to any of my images on the actual website!

This means that users with vision issues could not understand the content of images, a critical accessibility score. So, I went through every icon and image, and labelled them with relevant information, describing the content. For example, the logo at the top of most pages was labelled *Swap One logo (link to main website)*. After completing this, the accessibility requirements were met.



Performance and Load Testing

To make sure the website met my NFRs of **Stability** and **Performance**, I used an online load tester to spam the website, including Sign Up requests. The page could handle the requests all the way up to a speed of 30 requests per second, which far exceeds any expected user traffic limits. I determined that the infrastructure configuration was fit for purpose.

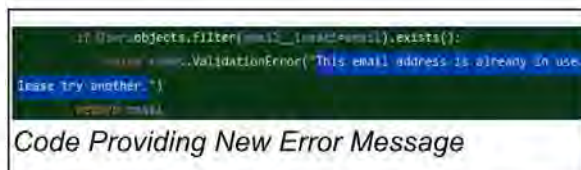
Security Validation

To ensure that the **Security** NFR had been met, I used the tool Burp Suite to ensure that malformed data could not be passed to the back end. I ensured that Django's built in protections for Cross-Site Scripting (XSS) and SQL injection attacks were functional, and that user friendly error messages were presented whenever anything went wrong.

I also attempted to override the Turnstile CAPTCHA by passing a false CAPTCHA token into the system. The server identified this and rejected the request, showing that the system was protected against bot attacks, meaning the stakeholder data has integrity.

Functional Regression Testing

I completed a final end-to-end test against all user stories to make sure no functionality had regressed throughout the final development process. Features like email sending,



blog updating, and data export worked flawlessly during this test. When testing sign up, I found that if an already registered email was used, a system error message would be displayed rather than a human readable version – a regression. After adding code fixing this, I completed the rest of the testing routine without any further regressions.

4.2. Fitness for Purpose

Evaluation Against Non-Functional Requirements

My final outcome, the completed Swap One web app met all Non-Functional Requirements that were set out at the start of the design process. The technical testing completed in Section 4.1 showed that the **Portability, Reliability, Accessibility, Stability, Performance**, and **Security** NFRs had all been met

The **Cost Effectiveness** NFR, while unable to be technically tested, was met due to the selections of open-source technologies and low-cost or free cloud hosting and services. To hire a professional developer to create this app would have cost tens of thousands of dollars, however I built the app for free while keeping the monthly cost below \$20.

The **Usability** NFR, was also successfully met, as seen in the user testing section and throughout the design process.

Evaluation Against MVP User Stories

User Story	Met?	Evidence
As a prospective user I want to join Swap One so I can reduce my emissions.	YES	Application features a single-page sign up page allowing users to sign up, as proven by post-launch non-tester sign ups.
As an app user I want to log my sustainable trips so I can participate in the programme.	YES	Logging page accessible via URL from emails, as proven by successfully logged trips from outside users.
As a programme coordinator I need to be able to access commute data in an Excel compatible format.	YES	System emails coordinators Excel-compatible spreadsheets daily, storing both user and trip data, in a secure manner. Stakeholders have tested export, and format is approved.
As an app user I want to be able to view emissions saved so I feel motivated to keep commuting sustainably.	YES	System calculates user emissions based on their inputted personal data, and shows users emissions saved on dashboard and emails, with graphs comparing their organisations to others to improve motivation.

As an app user I want to be reminded to do a sustainable trip so that I don't forget to commute sustainably.	YES	System sends a weekly reminder email from the automated <i>django-apscheduler</i> library. Reliability of email is high due to the queuing system and bounce handling, and has been verified with render testing on 49 clients.
As an app user I want the design to be simple and intuitive so I don't give up on using the app.	YES	Page has a nice looking interface which is easy to use, as proven by previous user testing.
As an app user I want to be able to unsubscribe from the system so I can retain control of my information and stop participating.	YES	Users can unsubscribe through unsubscribe links in the emails or the unsubscribe button on the main website. This is proven since outside users have unsubscribed.

Evaluation Against Post-MVP Enhancements

The project exceeded the initial brief, with additional enhancements requested by stakeholders (as noted in Section 3.2.). This is a direct result of the Agile methodology used for this project, since I was able to adapt to these major changes without going through another complex planning process, thus delivering an app more fit for purpose for my stakeholders.

Area	Met?	Impact on Fitness for Purpose
User Onboarding – refer to user by and collect name, collect distance, car type, workplace, and region.	YES	Made overall user experience more friendly for users, and significantly removed friction with logging system.
Logging – support EVs and carpooling as sustainable alternatives, allow multiple trips logged at once.	YES	Made application more useful for a wider audience, allowing people with varied commutes to use easier.
Dashboard – see more statistics in context, congratulate user, blog posts	YES	Place for user engagement leading to better communication and retention.

4.3. Real-World Deployment and Impact

The Launch

On the 17th of September, 2025, at an event with the local media, the Swap One programme was officially launched, alongside my app. It has been adopted by the Nelson-Tasman Climate Forum as the official digital platform for the initiative, with [REDACTED] saying the development was “*rigorous, professional and creative, and you delivered the software solution on time and on budget!*” *Swap One literally would not have been possible in its present form without your assistance*”. Going from the initial 10 businesses signed up, we now have 24 registered organisations, including the [REDACTED], and [REDACTED], and [REDACTED].

Programme coordinators from all of these organisations have signed up and are already swapping trips, and as they begin advertising the program to employees in their company, I expect user numbers to rapidly increase. The programme made the front page of the newspaper, as well as a team member being interviewed about the programme on Radio New Zealand's Morning Report.



Meeting of Desired Outcomes

A month after launch, I can safely say that the Swap One app has met all of the set outcomes that it's supposed to achieve.

The app is incentivising users to commute sustainably in Nelson-Tasman. The primary goal of the programme was to reduce Nelson's emissions by incentivising a sustainable commute, and this has been achieved, with over 180 people signed up and actively swapping so far. This is only through word-of-mouth communication and the newspaper article, without any promotion to employees yet, which is expected to drastically increase sign up numbers. As mentioned earlier, the app has incentivised users to save over 600kg of carbon, which is already a substantial contribution.

The app is collecting data for stakeholders to use, both in- and out-programme. Stakeholders are receiving daily data exports, and have already reported back initial program emissions savings directly from the exported data. At the end of the month, a winner will be drawn to win the first programme prize, and the provided data will be used. Additionally, the emissions savings data of any registered employees is being shared with businesses, allowing them to adapt their own

sustainable commute strategies based how their employees are getting to work.

The app is ensuring a frictionless and trustworthy user experience. Since users are willing to sign up and share their information, we are obviously a trusted platform. Due to the high rates of logged trips, we also know the core user path of receiving an email and logging a trip is frictionless, since otherwise users would not bother. The biggest sign that this app is trusted is that the Swap One programme coordinators are confident enough to promote it to all their users, risking their reputation on my work and proving that it is fit for their long term purpose.

The app has a scalable and maintainable technical foundation. All the code is openly licensed on GitHub to be adapted and changed by anyone who's doing anything similar. A professional developer, JP, was asked to do a pre-launch architectural review, verifying that the app will scale to the extent of the programme and can be modified by other parties if needed. Since Django is so modular, it is genuinely easy to adapt and add new features if required for the programme.

4.4. Reflection and End Note

Reflection on Process

Overall, the process I used for this application worked really well, as seen by the quality of the outcome. In saying that, if starting this whole endeavour again, I would absolutely make some changes.

I would start with a much greater focus on adaptability. My initial Waterfall methodology failed almost immediately, and the core structuring of the application is a remnant of this. Due to rigid and non-adaptable constraints like the way trips were stored, making last-minute changes such as predefined options for 'No' logging were very difficult without breaking the rest of the codebase. If I'd leaned into Agile from the start and designed the database structure with the expectation these core things could change, the app would have been more flexible and much easier to program.

Likewise, I would try to be more clear with my stakeholders about what they actually wanted from the app. It all worked out in the end, but the final scope was much bigger than we agreed at the beginning. Having a better idea of what to make from the start would have helped me make better base architectural decisions. But perhaps, this reflects the reality of working with stakeholders whose own ideas change through a development experience.

I would also have used a different authentication system. While the pure magic-link based system was entirely appropriate for the MVP, since the edit information feature allows the user to view their information, there is now the possibility that a malicious actor could gain access to a user's UUID via, for example, browser history, and find their information. While the stakeholders are comfortable with this approach for now, if starting again, I would use Django's password system to properly authenticate users before allowing them to view their information, to prevent any data breaches.

The technology choices I initially made worked out very well, especially the decision to use Django. The 'batteries included' philosophy of this service as well as the wealth of third party modules made so many additional changes, such as the admin panel or the blog system super easy. If I had used Flask instead, I simply would not have been able to implement these features, as they would have taken far more development time than I had available. Choosing Django was the best decision of the whole development process.

Learnings and Future Iterations

I learned so much from this development process. At the start, I had not ever built a production application, nor done any server-side development work. What a first proper programming project! I now understand so much more about how the web works, and after this year's work, I feel like I'm finally getting the hang of Django. The documentation and forum posts from all the various communities whose work I referred to was immensely helpful, and I really appreciate how simple everything was. If a novice like me could build a reliable and powerful web application from scratch, anyone could.

My most valuable lessons from this, however, were about how to work in a team. Good communication between everyone working on a project is so important, and I learned that I couldn't just go build an app that worked for myself – I needed to work with my team mates so that it would meet their needs and work with everything else they had designed.

I will be maintaining the Swap One web app for the duration of the programme, until the end of next year. I've already been asked to make some changes based on initial user feedback, such as allowing these users to input a distance each trip if they want instead of forcing all trips to be the same lengths. I'm definitely going to do this, but sadly there's no time to include it in this version.

End Note

The Swap One app is clearly still a work in progress, but it's a *working* work in progress, in production and being actively used by my stakeholders.

This has by far been the most complex and difficult thing I have ever achieved, taking up most of my free time for most of a year, far from the initial 50 hour programming time estimate. In saying that, I am so proud to have completed this project. I've learned how to program and work with people, but more importantly I have built something that is helping to make the world a better place.

I'm excited to see Nelson swapping away from the car.

Outstanding Scholarship

Subject: Technology

Standard: 93601

Total score: 22

Q	Score	Marker commentary
1	08	The candidate developed a sophisticated website to meet an authentic issue, providing a means for Nelson residents to reduce their carbon emissions from commuting to work. The requirements and attributes are well researched to establish comprehensive parameters of the brief in the broadest sense. The project exhibits seamless progression, demonstrating excellent design thinking throughout the technological process. The final outcome reflects deep, complex knowledge and a high level of refinement, achieved through highly competent technological practice. The efficient testing and knowledge that has been integrated within the project demonstrated the required elements of synthesis, polish, and optimisation that led to an outcome that is seen as a viable solution to the issue. The website is live and being used by many businesses in the Nelson-Tasman region. The development of the website is innovative and original, enabling the candidate to demonstrate a very high level of skills and creativity.

2	07	The candidate has provided a well-structured and compelling justification for their technological practice and outcome. A combination of very well written evidence and imagery provides visual connections to support the ongoing commentary. A short video clearly demonstrates how the various aspects of the website work in practice. Evidence throughout the report is consistently relevant, with minimal repetition, ensuring clarity and precision. A high level of technological modelling is evident, enabling the candidate to respond effectively to challenges during the website's development. The candidate has been discerning with their work to ensure the scholarship criteria has been met. Communication is systematic and persuasive, supporting the technological practice underpinning the outcome. This includes clear explanations of how complexities and constraints were resolved and reviewed throughout the iterative process. The candidate convincingly justifies their decision-making, technological practice, and the effectiveness of the final outcome. The narrative is coherent and compelling, further enhancing the perception of the product as an innovative and viable solution.
3	07	In this report, it is evident that the candidate has critically reflected on their own technological practice. The candidate selected and worked closely with members of the Nelson Tasman Climate Forum, who were genuinely invested in the success of the outcome. These stakeholders were consulted at key moments to provide important feedback. The candidate then critically reflected on the information gained to inform and guide their practice, resulting in a high level of skilled execution. Evidence clearly shows how ideas were progressed and decisions made independently, with a strong focus on ensuring the website was fit for purpose. This reflective approach underpinned a high level of skilled execution and contributed to the overall quality and effectiveness of the final outcome.