

B.Sc. (Hons) in Software Development



Ollscoil
Teicneolaíochta
an Atlantaigh

Atlantic
Technological
University

Green Count

By
BRENDAN MCSHANE (G00410478)

May 2, 2025

Minor Dissertation

**Department of Computer Science & Applied Physics,
School of Science & Computing,
Atlantic Technological University (ATU), Galway.**

Contents

1	Introduction	2
2	Methodology	4
2.1	Agile Development Approach	4
2.1.1	Scrum Framework in Green Count	4
2.2	Technical Research Approach	5
2.2.1	Technology and Framework Analysis	5
2.2.2	Existing Solutions and Best Practices	5
2.2.3	Iterative Development and Testing	5
2.2.4	Alignment with Agile Development	6
2.3	Planning and Feature Definition	6
2.3.1	Project Planning Approach	6
2.3.2	Feature Identification and Prioritisation	6
2.3.3	Tools and Documentation	7
2.4	Implementation of Development Technologies	8
2.5	Supervisory Meetings and Feedback	8
2.6	Development Tools	9
2.7	Testing	9
2.7.1	Unit Testing in Angular	9
2.7.2	Manual Testing and Debugging	10
2.7.3	Test Automation and Stability	10
3	Technology Review	12
3.1	Development of Modern Web Technologies	12
3.1.1	Evolution of Web Frameworks	12
3.2	Impact of Component-Based Architectures on Modern Web Development	13
3.3	Choosing Angular, Flask, and Supabase for Green Count	14
3.3.1	Choosing Angular for the Frontend	14
3.3.2	Using Flask for the Backend	15
3.3.3	Using Supabase for Database Management	15

3.4	Framework Comparison for Modern Web Development	16
3.4.1	Evaluation of Performance	16
3.4.2	Community Adoption and Framework Longevity	18
3.4.3	Scalability and Long-Term Maintainability	19
3.4.4	User Interface Design and User Experience	21
3.5	Justification for Technology Choices	23
3.5.1	Frontend Technology Selection: Angular	23
3.5.2	Backend Technology Selection: Flask	24
3.5.3	Database Technology Selection: Supabase	24
4	System Design	26
4.1	Application Architecture	26
4.1.1	System Architecture Overview	26
4.1.2	Component Roles and Interactions	26
4.1.3	User Interface Structure	27
4.1.4	Data Flow	28
5	System Evaluation	29
5.1	Frontend Structure and Integration	29
5.1.1	Reusable Angular Components	29
5.1.2	Dynamic Updates and Routing	30
5.1.3	Integration with Flask API	30
5.2	Data Handling and Real-Time Updates	30
5.2.1	Supabase Integration and Sync	31
5.2.2	API Performance and Reliability	32
5.3	UX and Visual Design	33
5.3.1	User Interface and Navigation	33
5.3.2	Emissions Visualisation	34
5.3.3	Accessibility and Clarity	35
5.4	Evaluation Summary	36
5.4.1	System Strengths	36
5.4.2	Limitations and Areas for Improvement	36
5.5	Alignment with Modern Development Standards	36
5.5.1	Use of Angular and TypeScript Best Practices	37
5.5.2	Data Security and Supabase Authentication	38
5.5.3	Considerations for Future AI Integration	39
5.6	Summary of Evaluation Findings	40
6	Conclusion	42
6.1	Summary of System Goals	42
6.2	Outcomes and Forward-Looking Enhancements	42

6.3	Insights from Development Practice	42
6.4	Final Reflections	43
Appendix		46

List of Figures

2.1	GitHub Project Board showing task management and sprint organisation.	7
2.2	Frontend code coverage results using Angular’s test suite.	11
3.1	Overview of Angular’s architecture highlighting core features such as components, services, dependency injection, and templates. . . .	14
3.2	Flask logo representing its lightweight and Pythonic design, ideal for scalable backend web development.	15
3.3	Supabase logo representing its open-source, PostgreSQL-backed platform used for scalable backend services in Green Count.	15
3.4	Supabase dashboard showing recent database and auth activity. Demonstrates real-time usage, authentication, and client library support.	16
3.5	GitHub Actions workflow demonstrating automated testing and continuous integration for the Green Count frontend.	25
4.1	Supabase database table structure illustrating stored emission records and associated metadata.	27
4.2	Green Count application’s main dashboard interface, highlighting key emission summaries and interactive features.	28
5.1	User-friendly emission entry form within the Green Count frontend, where users input data such as miles driven, food consumption, and work hours.	31
5.2	Angular unit testing summary via Karma/Jasmine, confirming successful execution and comprehensive test coverage.	33
5.3	Bar chart visualisation displaying estimated carbon emissions by day within the Green Count application.	34
5.4	Information resource page showcasing external environmental resources, designed for clear visual appeal and easy navigation.	35

5.5	Step Line Chart visualisation displaying user-entered carbon emissions over selected dates, highlighting peaks and patterns clearly. . .	38
5.6	Screenshot of OpenAI API integration showing sample chatbot response for carbon emissions advice.	40

Chapter 1

Introduction

In recent years, the growth of digital tools has led to increased opportunities for individuals to engage with and respond to environmental concerns through technology. Among these, web applications provide easily accessible places for users to track and reflect on their everyday actions that contribute to their carbon footprint. With environmental awareness becoming a more important concern, software solutions can help facilitate individual change through data openness and easy design.

This dissertation discusses the development of *Green Count*, a web-based emissions tracking application designed to help users measure their personal carbon footprint through daily activities. Users can log a variety of indicators, such as travel, work, and food intake, which the system converts into carbon emission estimates. Green Count also integrates the OpenAI API to provide users with suggestions on how to reduce their environmental impact. Data is securely stored and presented through dynamic visualisations, encouraging long-term awareness of personal carbon output.

The project was built using a modern web technology stack, including Angular as the frontend framework, Flask for backend functionality, API management, and Supabase as the backend-as-a-service platform. These technologies were chosen for their ability to offer a modular, scalable, and maintainable system design.

This application was built using an Agile process, with work divided into development sprints and maintained via GitHub Projects. GitHub Actions was used for automated testing and continuous integration, which ensured consistency and dependability in deployment. Unit tests were built for core frontend components, and backend endpoints were validated using both manual and automated tests. The usage of current development standards meant that usability and performance objectives were accomplished.

The objective of *Green Count* is not only to provide a working emissions tracker, but also to produce a stable and expandable software product that can serve as

the platform for future development, including predictive analytics or AI-based capabilities. The solution also integrates with the OpenAI API, demonstrating the possibility for conversational feedback and environmentally orientated support.

This dissertation is structured as follows:

- Chapter 2: **Methodology** - describes the Agile development process, planning methodology, research strategy, and testing procedures employed throughout the project.
- Chapter 3: **Technology Review** - analyses modern web development trends, compares relevant frameworks, and justifies the technologies chosen for this project.
- Chapter 4: **System Design** - outlines Green Count's architecture, which includes component roles, data flow, and integration methods.
- Chapter 5: **System Evaluation** - assesses the application's functional goals, usability criteria, and technological performance.
- Chapter 6: **Conclusion** - outlines the project's findings and makes recommendations for future development and improvement.

Green Count strives to encourage environmental accountability through software innovation by providing an accessible interface, accurate data handling, and scalable architecture.

Chapter 2

Methodology

2.1 Agile Development Approach

Agile is a flexible and iterative approach that was popularized in the 2000's. This methodology focuses on incremental progress, continuous feedback and adaptability. It is far more flexible than other older methodologies such as Waterfall. Agile encourages collaboration between developers, testers, higher-ups, and users.

2.1.1 Scrum Framework in Green Count

The project took a **Sprint-Based Development** approach, implementing features in short iterations. This iterative methodology enabled continuous feedback-driven improvements, ensuring that the application progressed efficiently. By breaking down development into defined sprints, progress could be regularly assessed, insights gathered, and features refined in response to testing results and changing project needs.

Daily Standups were conducted as self-check-ins to ensure steady progress and alignment with project goals. These regular progress reviews helped track completed tasks, identify potential roadblocks, and ensure development consistency. This approach ensured that work stayed on track and that any necessary adjustments could be made quickly.

A structured approach to **Backlog and Task Management** was crucial for keeping development organized. **GitHub Projects** was used to track tasks and coordinate development sprints, creating a clear visual representation of ongoing and completed work. Additionally, **GitHub Issues** was a valuable tool for documenting reported bugs, feature requests, and development discussions. The use of these tools increased transparency in task management while also streamlining the project's overall workflow.

This practical implementation of Scrum in a classroom-based project setting aligns with findings by Matthies et al. (2016), who highlight the effectiveness of combining surveys, tutor feedback, and artifact analysis to assess and reinforce agile adoption in student-led development environments [1]. The approach taken in Green Count mirrors this model by integrating structured feedback loops and tool-supported progress tracking to guide iteration and ensure alignment with core Scrum principles.

2.2 Technical Research Approach

This project's approach to technology selection and development was guided by a structured research methodology. This research phase involved comparative analysis, best practices review, and user-centred research to make informed decisions.

2.2.1 Technology and Framework Analysis

Choosing the appropriate technology stack was critical to the project's success. Various frontend, backend, and database technologies were evaluated in terms of performance, scalability, integration ease, and community support. Angular was chosen as the front-end framework due to its structured approach and widespread industry adoption. Flask was chosen for the backend due to its lightweight nature and flexibility, and Supabase was chosen as the database due to its PostgreSQL foundation and real-time capabilities. In addition, research was conducted into potential machine learning integrations for emissions data analysis, with the goal of determining the feasibility of predictive models in future iterations.

2.2.2 Existing Solutions and Best Practices

Existing solutions were reviewed to gain a better understanding of how similar applications track sustainability. This entailed examining their features, UI/UX design decisions, and common user pain points. Green Count was created using best practices in emissions tracking tools, with an intuitive user experience and a focus on usability.

2.2.3 Iterative Development and Testing

Throughout development, a continuous research-driven approach was employed. Testing various tools and methods ensured that the application could adapt to new insights. Scenario-based testing helped refine workflows, ensuring that Green

Count provided a consistent experience. Regular usability assessments guided advancements, allowing for practical refinements based on real-world experiences

2.2.4 Alignment with Agile Development

Research was not a one-time event, but a continuous process throughout development. Technology comparisons and best practices influenced feature prioritization, and usability testing was used to guide iterative improvements. This adaptable approach enabled the project to effectively respond to new discoveries and changing requirements.

2.3 Planning and Feature Definition

Green Count was developed with a structured planning process to ensure efficient implementation and iterative improvements. The planning phase was centred on identifying key features, prioritizing development tasks, and maintaining flexibility to accommodate refinements based on testing and feedback.

2.3.1 Project Planning Approach

Green Count was designed utilising an Agile methodology that emphasised incremental development and flexibility. The planning method was based on setting defined milestones, with each sprint delivering a functional part of the project. **GitHub Projects** was used to monitor tasks and visualize development progress, upcoming features, and awaiting refinements. Furthermore, **GitHub Issues** was utilized to log feature requests, document bugs, and handle conversations regarding development improvements.

2.3.2 Feature Identification and Prioritisation

Green Count's core features were determined during an initial research and analysis phase, with a focus on developing an intuitive and efficient emissions tracking tool. User authentication, data input and storage, emissions summarisation, and real-time database synchronisation via Supabase were among the key functionalities. The application also included a user dashboard, which served as a centralised interface for tracking progress and viewing trends.

Feature development was prioritised according to necessity and feasibility. The initial sprints focused on implementing critical functionalities like authentication and data storage, which served as the application's foundation. Once the core system was in place, the focus shifted to enhancements such as data visualisation

and insight generation. The prioritisation strategy ensured that critical features were developed first, with improvements and new functionality added in subsequent iterations. This approach struck a balance between providing a minimum viable product and allowing for incremental refinement.

2.3.3 Tools and Documentation

To keep the workflow organised, several tools for task tracking and documentation were used. **GitHub Projects** was the primary tool for managing sprints, visualising progress, and tracking completed and pending tasks. GitHub Issues was used to document development progress, feature requests, and reported issues, ensuring a systematic approach to each task. Throughout the project, documentation was created to describe the implementation of key features, allowing future scalability and improvements.

Figure 2.1 illustrates a screenshot of the GitHub Project Board used to manage and visualise development sprints and tasks throughout the project.

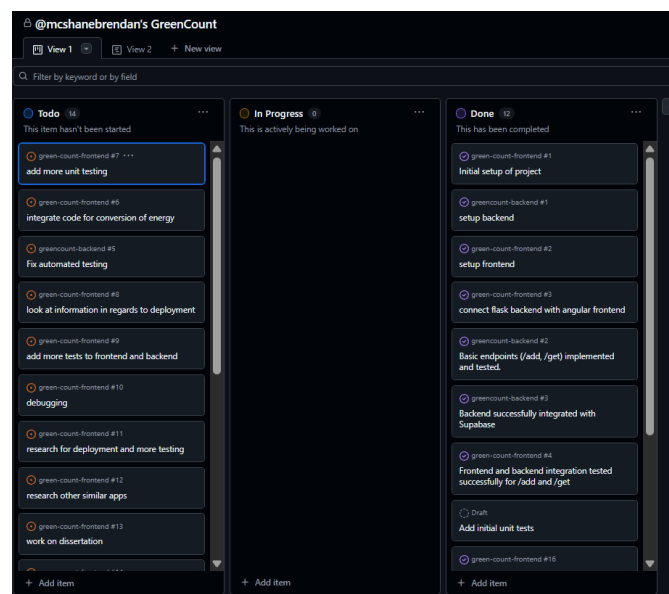


Figure 2.1: GitHub Project Board showing task management and sprint organisation.

2.4 Implementation of Development Technologies

Following the choices of Angular, Flask, and Supabase, the project's focus shifted to successfully integrating these technologies. The frontend was structured using Angular's component-based architecture, ensuring modularity and maintainability. Services were created to handle API requests, enabling seamless connectivity with the backend.

The backend, built with Flask, featured RESTful API endpoints for processing user data, calculating emissions, and managing authentication. Supabase was incorporated as the backend-as-a-service, allowing for real-time database transactions and user authentication. API queries were secured with authentication tokens, allowing users to only access their own data.

Unit tests, manual validation, and integrated logging tools were used for testing and debugging. API calls and answers were tracked using the browser's developer console, and server-side activity was examined using Flask debug mode. In order to ensure that upgrades would not bring about any disruptive changes, GitHub Actions was also incorporated for continuous testing. The project's development process remained seamless, scalable, and maintainable by utilising these technologies.

A vital part of the backend infrastructure, Supabase offered API-driven data management, real-time database interactions, and authentication. Utilising Supabase's integrated authentication services, user authentication was put into practice, guaranteeing safe access via access tokens. Through API calls, user emissions data could be efficiently retrieved and modified thanks to the database's storage. Supabase's real-time features made it possible for the frontend to update instantly, guaranteeing a flawless user experience. Complex backend setups were no longer necessary thanks to this integration, which streamlined development while preserving data security and performance.

2.5 Supervisory Meetings and Feedback

Regular supervision meetings were held throughout the project's development to review progress, handle any technical or design issues, and plan future tasks. These consultations served as checkpoints to ensure that the project remained on track with its objectives and adhered to the anticipated timetable.

Each session gave an opportunity to demonstrate completed features, receive specific feedback, and improve portions of the application based on the supervisor's recommendations. These reviews were similar to sprint evaluations in that they allowed for iterative improvement and aided in the prioritisation of essential development activities.

Meetings also helped to ensure constant development and time management. They helped towards the project's success by promoting continuous thought, decision-making, and adaptability to new difficulties.

2.6 Development Tools

Green Count was developed using a simple but effective set of tools to aid in the writing, testing, and maintenance of the program. These tools helped to ensure a uniform and effective workflow on both the frontend and backend.

Visual Studio Code was utilised as the primary development environment. Its built-in terminal, robust extension ecosystem, and smooth connection with Git made it ideal for working with Angular and Flask. Real-time error detection, syntax highlighting, and code formatting all contributed to a more efficient coding process and decreased the likelihood of common errors.

GitHub was utilised for version control and code management. Separate repositories for the frontend and backend were maintained, with consistent commits to ensure that progress could be tracked throughout the project. This enabled clear change tracking, which aided debugging and feature management.

GitHub Projects and GitHub Actions were used together to help with task management and continuous integration. GitHub Projects featured a kanban-style board to assist managing sprints, prioritise work, and graphically track progress. GitHub Actions was utilised to perform automated tests on each push, assuring codebase stability and speedy identification and resolution of bugs. Together, these tools gave structure and automation to the project's agile workflow.

2.7 Testing

To ensure the application's dependability, accuracy, and consistency, a combination of automated unit testing and manual testing was used. Testing was integrated into the development process from the start, allowing features to be validated incrementally and reducing the likelihood of regressions.

2.7.1 Unit Testing in Angular

Unit tests for frontend components were carried out using Angular's built-in testing framework. These tests were written in TypeScript using Jasmine and ran with Karma through the Angular CLI's `ng test` command. Unit tests were created in `.spec.ts` files to ensure that individual components performed as intended in isolation.

This configuration was used to test core features such component logic, data rendering, and user input processing. It was possible to get quick feedback on whether implemented features were correct by writing unit tests concurrently with component development. Priority was given to components that dealt with user interface and emissions data presentation, even if not all of them had complete test coverage.

2.7.2 Manual Testing and Debugging

Throughout the development process, manual testing was done to confirm functionality and user experience across various processes. This involved testing important functions like logging in, sending data, getting emissions summary, and changing views inside the program.

Tracking API calls and confirming that data was being delivered and received successfully between the frontend and backend were done using the developer console of the browser and network inspection tools. During development, Flask's debug mode helped find and fix problems by providing server-side logging and error messages.

2.7.3 Test Automation and Stability

Automated tests were performed on every push to the frontend repository using GitHub Actions. This ensured that any code changes passed all current tests prior to being merged, adding another layer of reliability. In addition to keeping the codebase stable, continuous integration stopped undetected errors from spreading to later stages of development.

The use of continuous integration services such as GitHub Actions contributed significantly to development reliability in Green Count. This approach aligns with findings by Matthies et al. (2017), who demonstrated the effectiveness of GitHub-based CI workflows in reinforcing automated testing and Test-Driven Development practices. Their study demonstrated that real-time feedback and enforced test execution can enhance consistency and encourage more thoughtful code contributions — an effect that was also evident throughout this project's development [2].

As shown in Figure 2.2, the Angular test suite provided visual feedback on code coverage, ensuring the robustness of key components.

```
==== Coverage summary =====  
Statements : 81.21% ( 320/394 )  
Branches   : 59.67% ( 74/124 )  
Functions  : 81.6% ( 102/125 )  
Lines      : 81.62% ( 311/381 )  
=====
```

Figure 2.2: Frontend code coverage results using Angular's test suite.

Chapter 3

Technology Review

This chapter provides a critical analysis of the core technologies chosen for the Green Count project—Angular, Flask, and Supabase—highlighting the rationale for their selection, comparative advantages, and how they support the project’s scalability, maintainability, and user-centric design objectives.

3.1 Development of Modern Web Technologies

Over time, web development has evolved substantially, with many different frameworks emerging to make web development a smoother process. The evolution of these various frameworks and the shift towards component-based architectures, such as Angular, which has become a leader in the space.

Modern web development has evolved significantly, transitioning from simple HTML-based pages to complex, scalable web applications. According to Dzhangarov et al. (2021), this transformation has been driven by the need for structured interfaces, enhanced interactivity, and seamless integration with backend systems. Their research highlights the importance of using frameworks that support modularity and real-time responsiveness—principles that underpin the selection of Angular, Flask, and Supabase in this project [3].

3.1.1 Evolution of Web Frameworks

Some of the mainstays in early web development were HTML, CSS and JavaScript. Using these together during the early days was challenging because developers were required to manually organize and update these web pages without interactivity. Later, jQuery came along and offered streamlined event handling and DOM manipulation, particularly as applications grew more complex. Although jQuery fixed many issues with web development at that time, it did not provide a systematic

method for creating large web applications.

Over time, there became a demand for a more expansive way to develop web applications. Modern JavaScript frameworks and libraries emerged to tackle these issues. These frameworks allowed developers to have better productivity in creating large and scalable applications. These frameworks introduced features such as state management, modular development and component-based architecture.

3.2 Impact of Component-Based Architectures on Modern Web Development

In recent years, the demand for organised, maintainable codebases has increased as web applications have become more sophisticated. Traditional development techniques often resulted in monolithic applications that were challenging to scale and modify. Component-based architecture provided a solution by enabling developers to break applications into smaller, reusable components.

Google's Angular framework has greatly influenced contemporary web development. In contrast to conventional JavaScript libraries, Angular is a complete framework with integrated modular design, dependency injection, and state management, making it ideal for large-scale business applications. The framework enforces a clear structure, allowing developers to build scalable applications while maintaining separation of concerns.

Recent research by Shah (2023) highlights how HTML5 Web Components and customized built-in elements are shaping the future of Component-Based Software Engineering. These modern approaches support the development of modular and reusable design systems, which align closely with Angular's architecture. The emphasis on encapsulated, interoperable elements reinforces the value of Angular's component model in building scalable and maintainable applications like Green Count [4].

Angular's **two-way data binding** is a major benefit that minimizes the need for manual updates by guaranteeing synchronization between the model and the view. The **Ahead-of-Time (AOT) compilation** feature of Angular also enhances application performance by converting TypeScript code into optimized JavaScript prior to runtime.

Although component-based architectures have also been embraced by frameworks such as React and Vue. Angular's structured methodology and integrated tools make it a solid option for projects requiring enterprise-level features, scalability, and maintainability. This has led to the widespread adoption in industries where stability and long-term support are critical.

3.3 Choosing Angular, Flask, and Supabase for Green Count

For Green Count, choosing the appropriate framework was essential to guaranteeing a scalable and maintainable application. Following an evaluation of a number of frameworks, such as React and Vue, Angular was selected due to its well-organised methodology and integrated features that complement the project's goals.

3.3.1 Choosing Angular for the Frontend

Code maintainability is improved by Angular's robust **TypeScript support** which helps developers create organised and type-safe applications. Its **two-way data binding** simplifies UI interactions, reducing the need for manual DOM manipulation, which is crucial for an application handling dynamic environmental data. Additionally, Angular encourages modularity through its **built-in dependency injections**, which enhances modularity, increasing the project's scalability and manageability as new features are added.

An additional key consideration in choosing Angular was its **full ecosystem**, which includes NgRx for state management and Angular Material for UI elements, guaranteeing a seamless and professional user experience. Furthermore, Angular's **security features** offer a solid foundation for handling sensitive environmental data, including defense against **cross-site scripting (XSS)** and **request forgery attacks**.

Angular's architecture is built around key concepts like components, services, templates, and dependency injection, all of which are visualised in Figure 3.1.

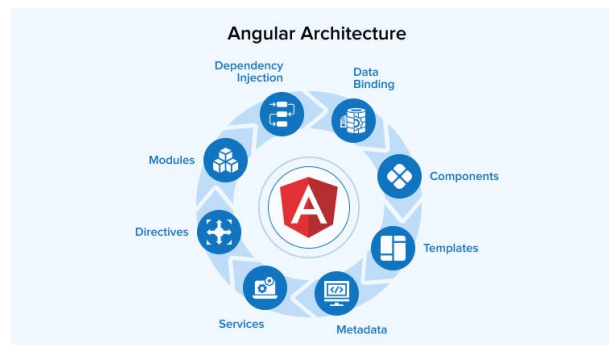


Figure 3.1: Overview of Angular's architecture highlighting core features such as components, services, dependency injection, and templates.

3.3.2 Using Flask for the Backend

Flask was selected as the backend framework because it is a lightweight and adaptable, which makes it ideal for managing database interactions and API requests. More customisation is possible thanks to Flask's micro-framework architecture, which only provides the project's necessary components while remaining scalable as new features are added.

An additional benefit of Flask is its **integration with Python's extensive ecosystem**, which enables Green Count's functionality to be enhanced through the use of data processing libraries. Additionally, Flask has built-in security features for input validation, authentication, and defence against common online threats.



Figure 3.2: Flask logo representing its lightweight and Pythonic design, ideal for scalable backend web development.

3.3.3 Using Supabase for Database Management

Supabase was chosen as the Green Count database solution due to its **PostgreSQL-based infrastructure**, which provides a very scalable and efficient environment for data storage and retrieval. With built-in authentication, real-time data synchronisation, and API generation, Supabase shortens development time without sacrificing a reliable and secure backend.

The choice of Supabase aligns well with the project's need for cloud-hosted, serverless database that allows for easy scalability. Its integration with Flask enables efficient data handling, allowing Green Count to store and retrieve environmental impact records dynamically.



Figure 3.3: Supabase logo representing its open-source, PostgreSQL-backed platform used for scalable backend services in Green Count.

3.4 Framework Comparison for Modern Web Development

This section will focus on analysing different modern web frameworks in terms of performance, scalability, UI design and support. This section will also compare multiple frameworks and highlight why Angular was chosen as the preferred framework.

Supabase's dashboard provides a real-time overview of backend services such as database requests, authentication events, and available client libraries, as shown in Figure 3.4.

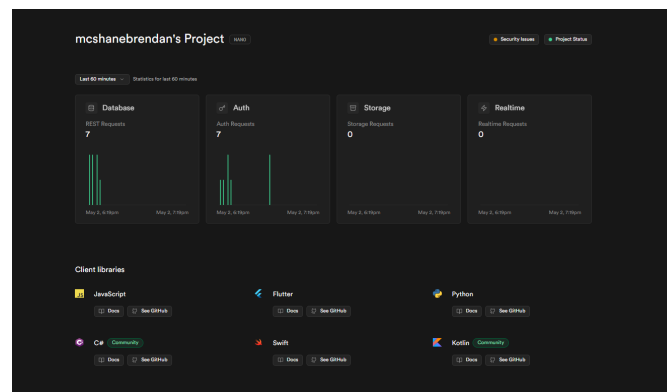


Figure 3.4: Supabase dashboard showing recent database and auth activity. Demonstrates real-time usage, authentication, and client library support.

3.4.1 Evaluation of Performance

- **Angular Performance Analysis**

Angular is known for being optimised for large-scale applications with a structured architecture. This framework uses Ahead-of-Time (AOT) compilation, this boosts performance by pre-compiling TypeScript before runtime. This leads to faster runtime and improved efficiency. Furthermore, Angular's built-in **Change Detection Mechanism** helps optimise updates, which then reduces unnecessary rendering cycles.

Recent academic studies have evaluated the rendering performance of popular frontend frameworks, highlighting trade-offs between their architectures. Ollila et al. [5] conducted a comprehensive comparison of Angular, React, Vue, Svelte, and Blazor, noting that frameworks with optimized change detection mechanisms — like Angular — tend to perform better in large-scale applications where real-time updates are frequent. Their findings reinforce

the choice of Angular for this project, especially given its Ahead-of-Time (AOT) compilation and built-in change detection, which proved beneficial for handling dynamic emissions data visualisations within Green Count's dashboard interface.

Compared to React, which mainly relies on a Virtual DOM to improve the speed of rendering by updating only the changed components, Angular's method guarantees consistency but may have a higher initial costs compared to React. In contrast, Ionic which is based on Angular itself and is focused on cross platform development, heavily relies on Web View Rendering, so browser capability determines its performance. Although Ionic is adaptable across many devices, it might not be as responsive as native or fully web optimised frameworks such as Angular.

Angular was chosen for this project due to its ability to manage challenging, large-scale projects. Its structured architecture, TypeScript support, built-in features including dependency injection and AOT compilation provide a strong foundation for scalability and maintainability. These capabilities are native to Angular, unlike React which needs extra frameworks for routing and state management. The choice of framework can significantly affect application performance, particularly when it comes to rendering efficiency. Studies comparing, Angular, React, Vue, Svelte and Blazor [5] indicated that frameworks that utilise reactive programming and optimised change detection mechanisms perform better, particularly in large-scale applications.

This shift is discussed in Mohd et al. (2022), who outline how modern JavaScript frameworks like Angular have evolved to address jQuery's limitations [6].

- **Ionic Performance Analysis**

Ionic was created for cross-platform development, this allows applications to run on iOS, Android and the web, from a single codebase. This framework is built on Angular and it gains its structured approach and features such as modular development and two-way data binding. However, unlike Angular's rendering that is native, Ionic relies on WebView rendering, meaning its speed and performance is based on browser capability instead of native execution.

In comparison, React offers better performance for mobile applications than Ionic. React does this by utilising a bridge to interact with native components. Ionic is reliant on WebView which can cause performance bottlenecks, particularly for applications that are resource intensive. While Ionic development provides flexibility because of its multi-platform deployment, it does not have the same efficiency as Angular or React.

Due to these performance trade-offs, Angular was selected over Ionic. This project's focus is to build a web application and Ionic would be more suited if the project was mobile focused, as it is designed for cross-platform development with a single codebase.

- **React Performance Analysis** React is a popular JavaScript library that focuses on component-based development. React often depends on the **Virtual DOM** to efficiently update UI components. This method creates a lightweight in-memory UI representation and updates only the necessary elements.

According to Ollila et al. [5], while the Virtual DOM in React can improve performance in various scenarios, it also adds another processing layer. Rather than directly modifying the DOM, React creates a virtual representation and compares it to the previous state to determine the minimal changes required. In large-scale systems with deeply nested components, the reconciliation process can cause significant performance bottlenecks. In contrast, Angular uses direct DOM updates, which can be more efficient in scenarios with frequent and complex UI changes.

React's performance often can be influenced by its state-management approach. Unlike Angular, which contains built-in state handling mechanisms, React can require additional libraries such as Redux or Context API to handle the application state efficiently. This modularity increases complexity in extensive applications even if it offers flexibility. Due to these reasons, React was not chosen for this project, as Angular's structured framework, integrated tooling, and efficient change detection are better suited for scalable web applications.

3.4.2 Community Adoption and Framework Longevity

Selecting a framework for a web application relies on understanding community acceptance and framework longevity. These factors influence the availability of resources, third-party libraries, and long-term support, which directly impact the maintainability of a project.

- **Community Adoption**
 - **NPM Downloads:** React leads in downloads, while Angular maintains a stable but smaller share.
 - **GitHub:** Angular has a strong community driven by enterprise, while React and Vue both have significantly high engagement.

- **Job Market Demand:** Angular remains a top choice in the world of enterprise-level applications, with high demand in the software industry that prioritises long-term maintainability and large-scale architecture.

- **Framework Longevity**

- **Corporate Backing:** Angular benefits from long-term support from Google. This ensures continuous updates, security patches, and industry-standard best practices.
- **Ecosystem and Updates:** Angular contains built-in tools, this includes dependency injection and modular architecture. These tools make Angular a preferred choice for large-scale enterprise solutions. This structured nature ensures stability and long-term support.

Considering these factors, the structured framework of Angular and its corporate backing from Google ensures longevity and stability, making it the most suited framework for this chosen project, due to Angular's maintainability, security, scalability are key priorities.

3.4.3 Scalability and Long-Term Maintainability

- **Frontend Scalability: Angular vs. React**

- By means of organized development and separation of concerns, **Angular's modular architecture** supports scalability.
- Blair et al. (2009) highlight that modular design allows for clearer separation of concerns and improved maintainability, supporting Angular's component structure [7].
- Lazy loading delays module loading until necessary, hence improving speed.
- **Built-in dependency injection** improves code reusability and tests simplicity.
- Blair et al. [7] highlight that comparative studies of Angular and React found that:
 - * React's adaptability helps with fast prototyping but creates long-term maintainability problems
 - * Large projects often suffer fragmentation if teams use various state management techniques, therefore influencing scalability.

- **State Management and Maintainability**

- Khati et al. (2024) emphasise the importance of state management in achieving scalability [8].
- React has to utilise third-party solutions such as Redux or Context API, which adds an extra layer of complexity that Angular does not have.
- Angular offers a built-in solution such as NgRx, this reduces fragmentation over time.
- Due to Angular's structured nature, it ensures maintainability for the developer. Making it a superior choice for certain applications that may be sizeable.

- **Backend Scalability: Flask vs. Django**

- Flask and Django are both similar frameworks that work with Python but they differ when it comes to scalability.
- Flask is a micro-framework, meaning it offers just the essentials, allowing developers to customise their architecture as needed.
- Django follows a monolithic structure, meaning it enforces standards with built in features like ORM, admin panel and authentication system.
- Flask's flexibility and lightweight nature make it a more scalable for API and database-driven applications such as Green Count.
- Django would be a more suited selection for a project that would benefit from its strict structures and built-in features.
- While applications often need scaling the entire system, Flask's support of **microservices architecture** lets certain parts of the program scale separately.

- **Database Scalability: Supabase vs. Firebase**

- Both Supabase and Firebase are both **back-end-as-a-service (BaaS)** platforms, but they differ in scalability and architecture.
- Supabase is built on **PostgreSQL**. This offers a relational database structure. For this project which relies on using a user table, Supabase became the preferred option due to it having a more suitable structure for queries.
- Firebase uses NoSQL which provides high-speed access but lacks the same amount of relational data integrity as Supabase.

- Supabase allows **server-side SQL querying and advanced joins**. This makes the technology more suitable for applications requiring structured data relationships.
- Firebase is more optimised for real time event-driven applications but it often can become inefficient when handling challenging searches.
- This project's demand for organised emissions data and effective querying made it that Supabase was the preferred technology over Supabase.
- Li and Manoharan (2013) found that SQL databases generally offer stronger data consistency and structured querying capabilities, making them well-suited for applications like Green Count [9].

By selecting Angular, Flask and Supabase, this project ensures a scalable and maintainable system that can maintain its performance and reliability.

3.4.4 User Interface Design and User Experience

The choice of technologies for a web application greatly impacts its user experience (UX) and user interface (UI). The choice of frontend framework, backend architectures and database all contribute to determine how effective an application delivers **visually appealing, responsive and interactive experiences**.

- **Component-Based UI Development**

- Angular enforces a structured framework that has a component-based architecture, which promotes maintainability and code reusability.
- Angular employs **Templates and Directives** which allows developers to create dynamic UIs with declarative syntax.
- React also follows a component-based approach, its JSX (JavaScript XML) syntax combines UI and mixes logic, offering adaptability but requiring additional styling decisions.

- **Built-in UI Libraries and Styling**

- Angular provides **Angular Material**, an official UI component library that adheres to *Google's Material Design principles*.
- React depends on third-party libraries such as Chakra UI or Material-UI, which provides developers flexibility but requiring external dependencies.
- In order to prevent styles from inadvertently leaking between components, Angular has built-in styling encapsulation.

- **Performance and UX Responsiveness**

- Although Angular’s **change detection mechanism** guarantees that user interface updates are managed effectively, it may result in performance overhead in intricate applications.
- React’s **Virtual DOM** optimises UI updates by minimises the manipulation of direct DOM. This often results in smoother animations and transitions.
- Both Angular and React support **lazy loading**, which helps improve initial load times by loading components only when needed.

- **Backend UI Considerations: Flask vs. Django**

- Flask and Django both support backend-rendered UI templates but they vary in flexibility.
- Flask allows for more customisation with its **Jinja2 templating**, making it ideal for lightweight applications that need a flexible backend with an API-driven frontend.
- Django offers a pre-made UI for handling applications with an extensive amount of content, but it adds needless complexity for projects looking to prioritise an API-driven design in the frontend.
- Django’s full-stack approach is better for monolithic systems that require backend UI administration, whereas Flask’s micro-framework structure therefore is **better suited for modern frontend heavy applications**.

- **Database UX Enhancements: Supabase vs. Firebase**

- Supabase is built on a **PostgreSQL** database which offers relational integrity and structured querying, enhancing data organisation and retrieval.
- Firebase is built on a **NoSQL** model, which enables real-time syncing, which is beneficial for applications requiring instant data updates.
- Firebase employs **document-based storage**, which can be more flexible but less effective for sophisticated queries, whereas Supabase provides **traditional SQL querying**, which facilitates the implementation of structured data relations.
- While Firebase excels in real-time data streaming whereas Supabase provides extra **advanced query control** and higher-quality support for server-side integrations.

- Grolinger et al. (2013) provide a detailed comparison of NoSQL and NewSQL systems in cloud environments, reinforcing the suitability of relational models like Supabase for scalable, cloud-based data management [10].

Overall, Angular offers a **structured, enterprise-ready UI framework** with built-in styling and state management, making it ideal for applications requiring consistency and long-term, maintenance. Flask was chosen over Django due to its **API-first flexibility**, which makes it a better match for a frontend-driven UI by allowing lightweight, scalable backend development without requiring a monolithic structure. Supabase was selected over Firebase due to its **relational database support**, which aligns with corresponds with Green Count’s structured data requirements, providing enhanced querying capabilities and improved data integrity for long-term scalability.

3.5 Justification for Technology Choices

For this project, selecting the appropriate technologies was essential to ensure maintainability, scalability, and performance. The following subsections outline reasoning behind choosing Angular for the frontend, Flask for the backend and Supabase for database management.

3.5.1 Frontend Technology Selection: Angular

- **Enterprise-level scalability:** Angular’s built-in state management and structured architecture make it suitable for large applications.
- **Integrated tooling:** A complete development environment is offered with features like dependency injection, CLI, and Angular Material.
- **Two-way data binding** streamlines dynamic updates by effectively synchronising UI elements.
- Angular has greater **built-in functionality** than React, which lessens the need for third-party libraries.
- Saks (2019) suggests that Angular is better suited for large-scale applications that require long-term support [11].

3.5.2 Backend Technology Selection: Flask

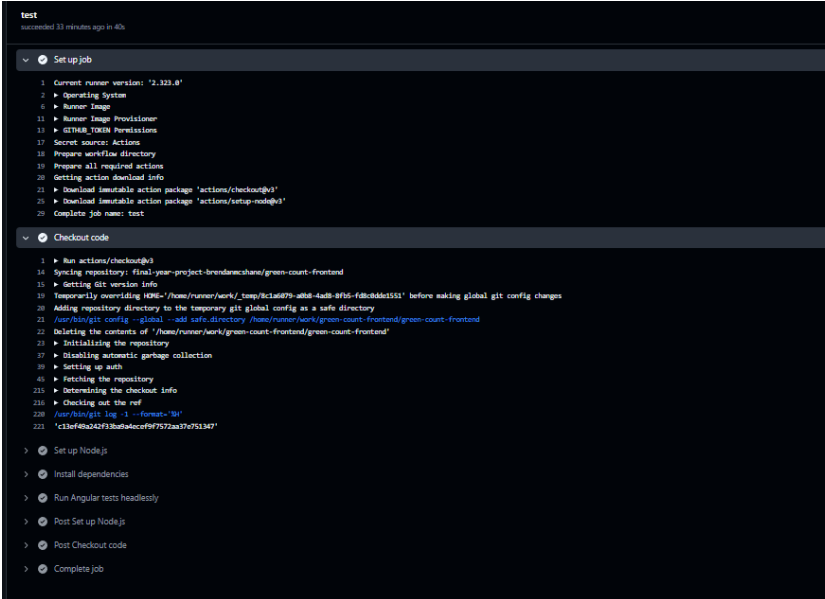
- **API-first** and lightweight: Flask's micro-framework structure gives designers freedom when creating RESTful APIs.
- **Python ecosystem**: facilitates the integration of data processing and machine learning tools.
- **Scalability**: Unlike Django's monolithic architecture, Flask's modular design is well-suited for microservices.

3.5.3 Database Technology Selection: Supabase

- **SQL-based storage**: Supabase uses PostgreSQL, which supports complex queries and structured data.
- **Server-side execution** is more efficient for handling relational data compared to Firebase's NoSQL approach.
- **Authentication and real-time sync**: minimises the backend workload, providing built-in authentication and live data updates.

For this project, the combination of Angular, Flask and Supabase provides an organised, scalable and maintainable architecture, that ensures long-term reliability and performance.

As shown in Figure 3.5, GitHub Actions was configured to automate the front-end test suite, ensuring that continuous integration processes were enforced with each code push.



```
test
succeeded 13 minutes ago in 40s

Set up job
1 Current runner version: '2.321.0'
2 Operating System
6 Runner Image
11 Runner Image Provisioner
11 GitHub Token Permissions
17 Secret source: Actions
18 Prepare workflow directory
19 Prepare all required actions
20 Getting action download info
21 Download immutable action package 'actions/checkout@v3'
23 Download immutable action package 'actions/setup-node@v3'
25 Complete job name: test

Checkout code
1 Run actions/checkout@v3
14 Syncing repository: final-year-project-brendanmcschane/green-count-frontend
15 Getting git version info
16 Temporarily overriding SSH key: /home/runner/work/_temp/1c1a0979-9d8d-4a8b-8f6d-f6b0b6b1551 before making global git config changes
20 Adding repository directory to the temporary git global config as a safe directory
21 /usr/bin/git config --global --add safe.directory /home/runner/work/green-count-frontend/green-count-frontend
22 Deleting the contents of /home/runner/work/green-count-frontend/green-count-frontend
23 Initializing the repository
27 Disabling automatic garbage collection
30 Setting up auth
40 Fetching the repository
215 Determining the checkout info
216 Checking out the ref
220 /usr/bin/git log -1 --format=%h
221 'c13ef6a24f33b0b0ace9f97572aa37e751347'

Set up Node.js
Install dependencies
Run Angular tests headlessly
Post Set up Node.js
Post Checkout code
Complete job
```

Figure 3.5: GitHub Actions workflow demonstrating automated testing and continuous integration for the Green Count frontend.

Chapter 4

System Design

4.1 Application Architecture

Green Count's system design was structured to ensure maintainability, scalability, and efficient communication between its components. This section will outline the component roles, high level architecture and the flow of data throughout the system.

4.1.1 System Architecture Overview

The database, frontend, and backend of Green Count operate as separate but connected modules according to a **client-server architecture**.

- **Frontend (Angular)** Handles data visualisation, user interactions, and form submission. Angular was chosen due to its MVC architecture and built-in dependency injection, which make it suitable for scalable web applications, as noted by Jain et al. [12].
- **Backend (Flask)**: Manages requests, authentication, and business logic as a RESTful API.
- **Database (Supabase)**: this stores user generated data and environmental impact records.

4.1.2 Component Roles and Interactions

- **Angular**: renders dynamic content, sends HTTP requests to Flask API, and updates UI components.

- **Flask:** Communicates with Supabase database, processes requests, and returns JSON responses to frontend.
- **Supabase:** Stores environmental impact records and user-generated data.
- **External APIs (OpenAI):** OpenAI API was integrated into the application to create a chatbot that is prompted to focus on giving advice for carbon emissions related queries.

Figure 4.1 shows an example of how emission data and related metadata are stored within the Supabase database schema used by Green Count.

id	name	email	created_at	user_id	unit	val	name	email	category	type
194054-8131-4348-e837-1b0dc4e4d4d			2025-03-03 18:03:32.870609+0		km	12			Transport	Rail
163008-ea08-4805-8194-9ab576a5822			2025-04-05 11:41:26.539271+0		km	21			Transport	Rail
3ab33405-4698-4305-8325-c8bdc0a4e4d			2025-04-05 18:41:58.467161+0		night	2			Accommodation	Hotel-UK
48e54493-e599-4916-95a3-e4869187a7b			2025-05-02 12:56:34.340259+0		mile	3		0.8000000000000001	Car	Diesel
4965483-79ac-4a3b-af70-7a0a671a67c			2025-04-05 12:12:53.583691+0		mile	3		0.54668	Car	Diesel
59e3a339-c639-4a14-8479-7d16a5716ee			2025-05-02 10:28:45.388905+0		mile	2		0.52946	Car	Petrol
6a465977-ea46-433b-c7d0-2195a3a7ac1			2025-05-02 10:38:24.503335+0		mile	10		2.6473000000000004	Car	Petrol
68b184e-e48f-4f6d-b4fb-d8c08183854			2025-05-02 12:39:21.778481+0		hour	2		0.60756	Working	Home
6613940b-c8ac-466d-9ea3-c16b000a6d			2025-02-17 20:41:59.541941+0		mile	67		1773691	Car	Petrol
662387a3-30a0-4204-aaa0-2263202532f			2025-04-05 15:10:15.348781+0		mile	2		0.54668	Car	Diesel
791ae059-9a39-4a3f-a597-f77821a6709			2025-05-02 12:38:53.818086+0		mile	5		1.3136500000000002	Car	Petrol
7a37f04e-6181-4136-977c-7a3d31a6c1a			2025-05-02 13:10:49.980776+0		mile	1		0.27334	Car	Diesel
8a4543a0-7794-4a25-94ee-8a8bba483a			2025-04-05 10:58:21.938251+0		km	0.66		0.0716636	Transport	Bus
8a32a59f-ba32-47ee-87ee4b48a67			2025-04-05 18:41:36.137581+0		mile	13		0.3006780000000000	Car	Diesel
8a85459f-7a8f-4aee-9a8f-827228a4b7a			2025-04-05 10:57:38.422343+0		km	1		0.03546	Transport	Rail
8b32256-cab7-47a7-81ec-8a67333a4a8			2025-02-17 20:39:27.779154+0		mile	66		1720761	Car	Petrol
87daeeb-e1b4-470d-9f6e-ba8b728a7a			2025-02-03 16:37:46.357286+0		mile	3		0.8000000000000001	Car	Diesel
b59a140-3214-4717-883f-877a48a4322			2025-04-05 18:41:49.836341+0		hour	1.9		0.04716000000000001	Working	Office
b6a389f-ba0d-4856-896f-1a919a7a5a			2025-04-05 10:46:48.477543+0		km	21		3.36228	Transport	Bus
ccc0924-980f-4953-e885-c270a6c3a6c			2025-04-30 19:12:59.316584+0		mile	1.2		0.37676	Car	Petrol
cea8744e-a277-498f-e9a3-e647a1a481a			2025-05-02 11:25:24.000036+0		mile	1		0.27334	Car	Diesel
e6979a7e-5d4c-4a04-a56e-8f08000a63f			2025-04-05 12:36:11.548072+0		mile	1		0.27334	Car	Diesel
ea45e407-6193-47ee-7ee959958ac			2025-03-04 13:16:15.472098+0		km	21		2.27766	Transport	Bus

Figure 4.1: Supabase database table structure illustrating stored emission records and associated metadata.

4.1.3 User Interface Structure

The user interface (UI) adheres to a structured layout designed for usability and efficiency:

- **Navigation Bar:** Provides easy actions to different pages of the web application.
- **Dashboard:** Displays analytics and user-inputted data.
- **Form Components:** allows user to enter their environmental impact data such as miles driven, hours worked, food eaten, etc.

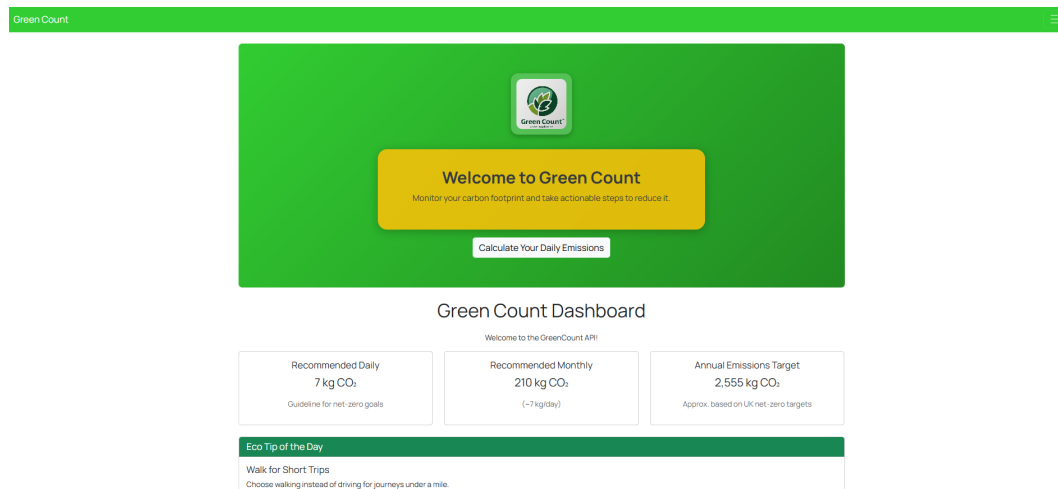


Figure 4.2: Green Count application's main dashboard interface, highlighting key emission summaries and interactive features.

4.1.4 Data Flow

Data travels in a systematic manner across the system:

- Data is submitted through **Angular frontend**.
- This data is then sent via HTTP requests to the **Flask backend**.
- Flask then processes and validates this data before sending it to **Supabase**.
- Supabase then stores this data and provides retrieval endpoints for reporting and analytics.
- The frontend then retrieves this processed data and modifies the UI components dynamically.

This system design ensures that Green Count's architecture is modular, maintainable, and scalable allowing it to efficiently handle environmental data tracking and visualisation.

Chapter 5

System Evaluation

This chapter assesses the Green Count web application against the objectives described in Chapter 1. The evaluation focusses on critical areas of the application, such as architecture, data processing, user experience, and conformance to development standards. The system's strengths and shortcomings are critically evaluated, and potential areas for development are identified. The purpose is to reflect on the project's practical outcomes and demonstrate how successfully the system achieved its intended objectives.

5.1 Frontend Structure and Integration

Angular was chosen for the frontend of Green Count due to its organised, component-based architecture and widespread industry usage. The solution was designed to be modular, reusable, and integrate seamlessly with backend services. This section assesses the effectiveness of the frontend implementation, namely component design, routing, and data flow.

5.1.1 Reusable Angular Components

This application was created with Angular's component-based architecture, which enabled a flexible and maintainable frontend framework. Throughout the development process, reusable components for critical aspects such as data entry forms, emissions summaries, and interactive elements were produced. This method minimised code duplication while allowing for consistent styling and behaviour throughout the application. Reusing these components across various views made the program easy to grow and update in subsequent development sprints. However, there is still potential for more abstraction, as certain code may have been placed into other shared components or services to boost scalability even more.

5.1.2 Dynamic Updates and Routing

Green Count uses Angular’s built-in routing system to provide seamless navigation between views such as the homepage, data input forms, and settings. Using standalone components with Angular’s router-outlet enables each view to be rendered dynamically, eliminating the need for full page reloads. This results in a fast, responsive user experience with smooth transitions throughout the application.

State changes caused by user input, such as form submissions or changing display choices, were reflected in real time. This was accomplished with Angular’s two-way data binding and reactive programming methods. The usage of these strategies kept UI components in sync with application logic and backend answers.

Client-side routing also supported deep linking, which allowed users to visit certain areas of the program directly using URLs. This increased usability and made the program easier to navigate, particularly during testing and development. The routing system was kept lightweight and controllable, with well-defined paths and little redundancy.

5.1.3 Integration with Flask API

Through a set of RESTful API endpoints created with Flask, the frontend of Green Count connects to the backend, enabling real-time communication between the Angular client and the backend services that handle and store emissions data.

To submit and retrieve user data, Angular services send HTTP requests to endpoints like ‘/add’, ‘/get’, and ‘/summarize’. The application maintains responsiveness even during data-intensive operations because these interactions are managed asynchronously. The API’s structure was kept straightforward and effective, with distinct routes for each functionality.

Flask and Supabase work together to manage authentication as well. Secure tokens are transferred from the frontend to the backend during user registration or login, enabling the system to verify and link data to specific accounts. To handle invalid requests or token difficulties, error handling was added, making the experience more reliable and intuitive.

Overall, the integration of Angular with Flask was successful, with dependable data transport and low latency. The separation of concerns between frontend presentation and backend processing allowed for cleaner code and easier maintenance.

5.2 Data Handling and Real-Time Updates

Data management was an important feature of Green Count, especially given its emphasis on user-submitted emissions records and summarised environmental

insights. The system was developed to provide accurate data storage, secure user identification, and real-time updates.

The backend validated and processed incoming data before saving it to the Supabase database. Each contribution was evaluated for structure, type, and value consistency, which helped to ensure data integrity across all records. Supabase's PostgreSQL foundation allowed for efficient querying and filtering, which was critical for producing accurate emissions summaries and facilitating user-specific data retrieval.

Supabase's event-driven architecture enabled real-time functionality, allowing the frontend to quickly access newly added or updated data. This was especially useful for ensuring that changes to the interface were reflected immediately after user input. The Angular frontend, Flask backend, and Supabase all communicated seamlessly, ensuring that user interactions were accurately reflected in the system's state. These mechanisms contributed to a more dependable and interactive experience, thereby increasing the application's overall usability and performance.

User-submitted data is a key aspect of Green Count's functionality. The structured form for entering emissions (Figure 5.1) is crucial to ensuring accurate carbon impact tracking.

The screenshot shows a web form titled "Add Emission". It contains three main input areas: a "Category:" dropdown menu with "Car" selected, a "Type:" dropdown menu with "Diesel" selected, and a "Value:" text input field with the placeholder text "Enter Value". Below the "Value:" field is a smaller text label that reads "Enter distance driven (miles)". At the bottom left of the form is a blue button labeled "Add Emission". Below the entire form is a green button labeled "Display My Emissions" with a small icon of a document with a list.

Figure 5.1: User-friendly emission entry form within the Green Count frontend, where users input data such as miles driven, food consumption, and work hours.

5.2.1 Supabase Integration and Sync

Supabase served as Green Count's primary backend-as-a-service, managing both user authentication and emissions data storage. Its integration resulted in a robust and scalable solution with minimal backend setup, allowing for rapid development and reliable data management.

Authentication was handled by Supabase's built-in auth service, which used secure access tokens to validate users and protect endpoints. This ensured that data operations were securely linked to individual user accounts, allowing each user to only see and manage their own emissions data.

The data was stored in a structured PostgreSQL database provided by Supabase. Insertions, queries, and summaries were done using Flask endpoints that communicated with Supabase via its Python client. This setup provided a straightforward but powerful interface for adding, retrieving, and processing user data, with strong consistency guarantees.

Real-time syncing capabilities improved the overall user experience. When emissions records were added or modified, the frontend reflected the changes without requiring a page reload. This was accomplished through Supabase's support for real-time subscriptions and instant query execution, which helped to keep the interface responsive and up to date.

Supabase became a critical component of Green Count's architecture due to its secure authentication and responsive data syncing, which streamlined development while maintaining performance and data reliability.

5.2.2 API Performance and Reliability

Green Count's Flask backend handled core application logic, such as data validation, emission calculations, and communication with the Supabase database. Throughout development, special care was taken to ensure that the API remained responsive, stable, and reliable under a variety of usage scenarios.

Endpoints were designed in a RESTful manner, with clear paths for adding, retrieving, summarising, and converting data. Each endpoint included validation checks to prevent invalid input from reaching the database, lowering the risk of data corruption and increasing the overall system robustness.

In terms of performance, the API consistently provided low-latency responses throughout testing. Even with multiple data entries or summarisation queries, the endpoints performed efficiently, thanks to Flask's lightweight nature and Supabase's optimised query capabilities. Logging was also enabled to track request patterns, identify problems, and monitor usage without causing delays.

Unit testing and the integration of GitHub Actions helped to improve reliability even further. Automated test suites were used to ensure that each update did not introduce regressions, thereby promoting consistent behaviour across deployments. These practices helped to build a stable API foundation capable of supporting real-time interactions and data-driven features on the frontend.

As part of quality assurance, automated unit testing was integrated using Karma and Jasmine. These tests were executed via GitHub Actions to validate functionality with every code change, ensuring consistent application behaviour (Figure 5.2).

```

    at UserContext.apply (http://localhost:9876/_karma_webpack_/webpack/src/app/home/home.component.spec.ts:274:15)
    at _ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/webpack/node_modules/zone.js/fesm2015/zone.js:369:28)
    at ProxyZoneSpec.onInvoke (http://localhost:9876/_karma_webpack_/webpack/node_modules/zone.js/fesm2015/zone-testing/zone-testing-ALERT: 'Error adding emission: fail')
Chrome 136.0.0.0 (Windows 10): Executed 83 of 104 SUCCESS (0 secs / 0.743 secs)
LOG: 'Fetched data:', Object{message: 'Welcome!'}
Chrome 136.0.0.0 (Windows 10): Executed 84 of 104 SUCCESS (0 secs / 0.756 secs)
LOG: 'Fetched data:', Object{message: 'Welcome!'}
Chrome 136.0.0.0 (Windows 10): Executed 85 of 104 SUCCESS (0 secs / 0.769 secs)
LOG: 'Current newEmission state:', Object{category: 'Transport', type: 'Train', value: 25}
Chrome 136.0.0.0 (Windows 10): Executed 85 of 104 SUCCESS (0 secs / 0.769 secs)
LOG: 'Sending emission data:', Object{category: 'Transport', type: 'Train', value: 25}
Chrome 136.0.0.0 (Windows 10): Executed 85 of 104 SUCCESS (0 secs / 0.769 secs)
LOG: 'Emission added successfully:', Object{success: true}
Chrome 136.0.0.0 (Windows 10): Executed 85 of 104 SUCCESS (0 secs / 0.769 secs)
LOG: 'User emissions loaded:', []
Chrome 136.0.0.0 (Windows 10): Executed 85 of 104 SUCCESS (0 secs / 0.769 secs)
LOG: 'Fetched data:', Object{message: 'Welcome!'}
Chrome 136.0.0.0 (Windows 10): Executed 86 of 104 SUCCESS (0 secs / 0.782 secs)
LOG: 'Fetched data:', Object{message: 'Welcome!'}
Chrome 136.0.0.0 (Windows 10): Executed 87 of 104 SUCCESS (0 secs / 0.795 secs)
LOG: 'Current newEmission state:', Object{category: 'Transport', type: 'Train', value: 25}
Chrome 136.0.0.0 (Windows 10): Executed 87 of 104 SUCCESS (0 secs / 0.795 secs)
LOG: 'Sending emission data:', Object{category: 'Transport', type: 'Train', value: 25}
Chrome 136.0.0.0 (Windows 10): Executed 87 of 104 SUCCESS (0 secs / 0.795 secs)
LOG: 'Emission added successfully:', Object{success: true}
Chrome 136.0.0.0 (Windows 10): Executed 87 of 104 SUCCESS (0 secs / 0.795 secs)
LOG: 'User emissions loaded:', []
Chrome 136.0.0.0 (Windows 10): Executed 87 of 104 SUCCESS (0 secs / 0.795 secs)
LOG: 'Fetched data:', Object{message: 'Welcome!'}
Chrome 136.0.0.0 (Windows 10): Executed 88 of 104 SUCCESS (0 secs / 0.888 secs)
LOG: 'Payload sent to backend:', Object{Metric: 'electricity', Value: 100, TargetUnit: 'kg'}
Chrome 136.0.0.0 (Windows 10): Executed 88 of 104 SUCCESS (0 secs / 0.888 secs)
LOG: 'Conversion successful:', Object{Emissions: 42}
Chrome 136.0.0.0 (Windows 10): Executed 88 of 104 SUCCESS (0 secs / 0.888 secs)
LOG: 'Full response:', Object{Emissions: 42}
Chrome 136.0.0.0 (Windows 10): Executed 88 of 104 SUCCESS (0 secs / 0.888 secs)
LOG: 'Energy emissions:', 42
Chrome 136.0.0.0 (Windows 10): Executed 88 of 104 SUCCESS (0 secs / 0.888 secs)
LOG: 'Fetched data:', Object{message: 'Welcome!'}
Chrome 136.0.0.0 (Windows 10): Executed 89 of 104 SUCCESS (0 secs / 0.821 secs)
LOG: 'Settings saved:', Object{darkMode: true, animations: false}
Chrome 136.0.0.0 (Windows 10): Executed 90 of 104 SUCCESS (0 secs / 0.834 secs)
Chrome 136.0.0.0 (Windows 10): Executed 104 of 104 SUCCESS (0.984 secs / 0.878 secs)
TOTAL: 104 SUCCESS
02:05:20:58.858:WARN [Chrome 136.0.0.0 (Windows 10)]: Disconnected (0 times) Client disconnected from CONNECTED Chrome 136.0.0.0 (Windows 10) ERROR
Disconnected Client disconnected from CONNECTED state (transport close)
Chrome 136.0.0.0 (Windows 10): Executed 104 of 104 SUCCESS (0.984 secs / 0.878 secs)
Chrome 136.0.0.0 (Windows 10) ERROR
Disconnected Client disconnected from CONNECTED state (transport close)

```

Figure 5.2: Angular unit testing summary via Karma/Jasmine, confirming successful execution and comprehensive test coverage.

5.3 UX and Visual Design

Green Count’s development required careful consideration of both user experience and visual design. The interface was designed with simplicity and clarity in mind, with a minimalist visual style that reduces distractions and emphasises important information. A consistent layout, clear icons, and responsive design contributed to the application’s accessibility across devices and screen sizes, allowing it to be used by a larger audience.

Angular’s component-based architecture allowed for a modular and consistent design across all views. Common UI elements, such as buttons and forms, used a consistent visual language, which improved usability and familiarity. By emphasising functionality and user engagement, these design decisions contributed to the project’s overall goal of increasing environmental awareness.

5.3.1 User Interface and Navigation

The user interface of this application is designed to be simple and straightforward. A simple, intuitive layout was implemented, allowing users to easily navigate key sections such as emissions input, settings, and other features. Angular’s routing

system allowed for seamless transitions between views without reloading the page, resulting in a fluid and responsive experience.

To avoid overwhelming the user, navigation elements were kept to a minimum and clearly labelled. Buttons, forms, and interactive components all used consistent design patterns, making it easier for users to understand how to interact with the system. Visual cues such as loading indicators and confirmation messages were used to provide feedback, allowing users to feel more confident while using the application.

5.3.2 Emissions Visualisation

The front page of Green Count serves as a user-friendly starting point, providing direct access to the application's main features. Its straightforward design prioritises simplicity and accessibility, allowing users to easily navigate and interact with the system.

A variety of charts were used to visually represent data trends and comparisons. These included bar charts, pie charts, and line graphs that were generated dynamically in response to user input. The use of multiple visual formats allowed for a more flexible and engaging representation of information. These visualisations improved both user understanding and interaction. Selected examples of these charts may be included in this document to show their design and functionality.

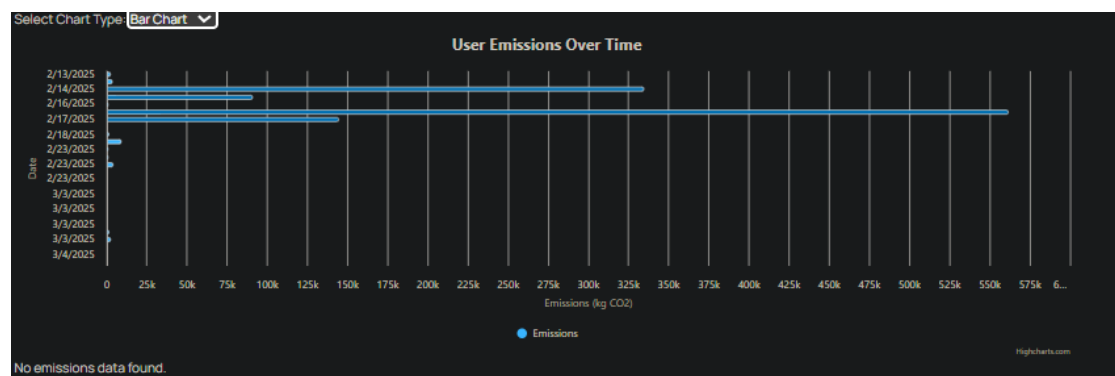


Figure 5.3: Bar chart visualisation displaying estimated carbon emissions by day within the Green Count application.

5.3.3 Accessibility and Clarity

The Green Count user interface was designed with accessibility and clarity in mind. The layout had a clean and minimal structure, with high contrast between text and background to improve readability. Font sizes and spacing were chosen to accommodate a wide range of users and keep content legible across devices.

To guide user interaction, the application featured clear labelling and consistent iconography. Form fields included placeholder text and error handling to reduce user confusion while entering data. The navigation was intended to be intuitive, with each page having a clear heading and purpose. While no formal accessibility testing was conducted, efforts were made to adhere to general web accessibility principles, making the system more accessible to a wider audience.

The application also includes an Information page that presents external environmental resources in a visually accessible layout. This page supports the application's goal of raising awareness by offering users clear, engaging links and summaries (Figure 5.4).

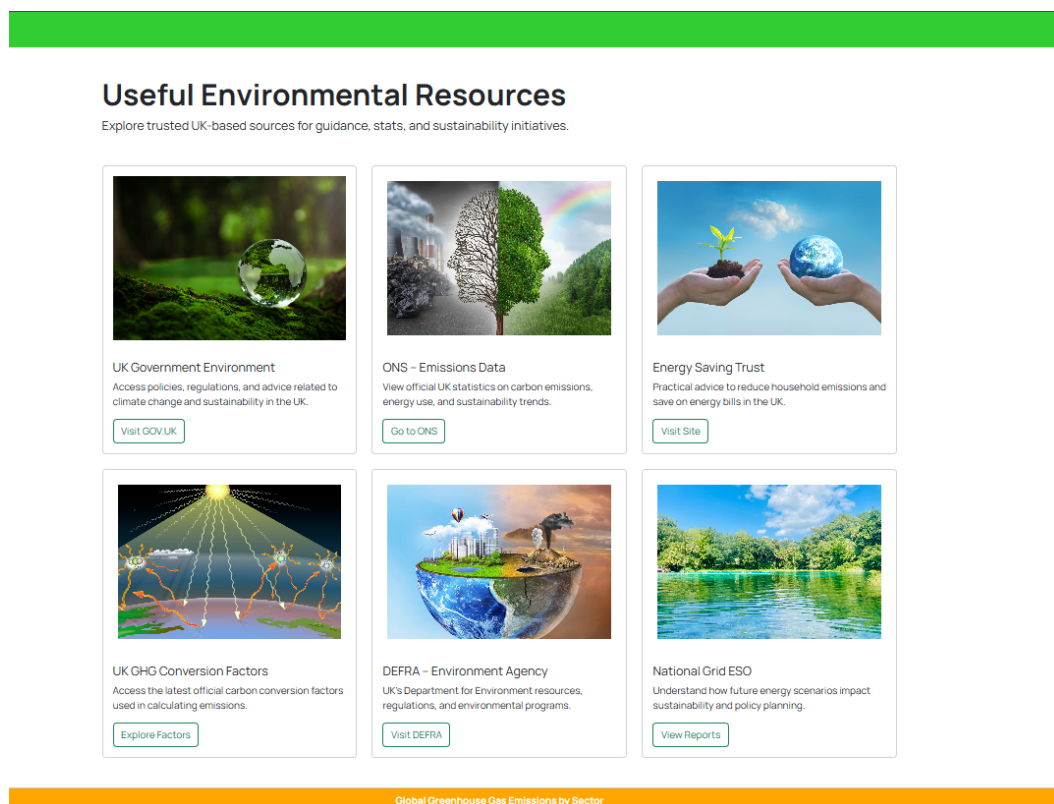


Figure 5.4: Information resource page showcasing external environmental resources, designed for clear visual appeal and easy navigation.

5.4 Evaluation Summary

This section provides a concise summary of the system's overall performance, taking into account its implementation, design, and real-world usage. It outlines Green Count's strengths as well as areas where future enhancements could improve functionality, scalability, or user experience.

5.4.1 System Strengths

Green Count demonstrated a range of strengths during development and testing. The use of Angular's modular architecture enabled the creation of a flexible and maintainable frontend. Supabase offered real-time database syncing and secure user authentication without the need for extensive backend configuration, which helped streamline development. The integration of Flask and Supabase was effective, allowing for smooth API performance and accurate data handling. Furthermore, the user experience benefitted from responsive design choices, fast navigation, and interactive visual elements such as charts and summaries.

5.4.2 Limitations and Areas for Improvement

While Green Count accomplished its major goals, a few drawbacks were discovered during the evaluation process. Some components may benefit from further abstraction or reworking to increase long-term scalability. To prevent duplication, certain logic could be relocated to shared services.

The system's data validation is effective, but it might be improved with more robust error signals and client-side checks for greater usability.

From a design perspective, accessibility elements could be improved, especially for individuals with visual or movement impairments. Enhancing user preferences, including language and theme customisation, could further elevate engagement. Ultimately, although charts are useful, the addition of downloadable reports or export functionalities could enhance the utility for customers monitoring their emissions longitudinally.

5.5 Alignment with Modern Development Standards

Green Count was planned and developed in accordance with current web development best practices, guaranteeing that the system is stable, scalable, and secure. This section assesses the project's adherence to best practices for frontend and

backend development, data security, and future-proofing through prospective AI integration.

5.5.1 Use of Angular and TypeScript Best Practices

The frontend implementation adhered to Angular's recommended practices, utilising its modular structure, standalone components, and services to generate a clean and maintainable codebase. TypeScript was utilised to ensure stringent type checking, enhance code reliability, and reduce runtime errors. Angular's built-in capabilities, such as two-way data binding and dependency injection, helped to speed up development and simplify debugging.

Throughout development, emphasis was placed on maintaining code readability and consistency. The components were arranged logically, and services were used to segregate concerns like HTTP requests and state management. This adherence to Angular rules improves scalability and makes the application easier to maintain or pass on to future developers.

A key output of the system's data flow is the generation of emission visualisations for the user. The step line chart, shown in Figure 5.5, provides a clear and continuous view of carbon emission trends across multiple entries.

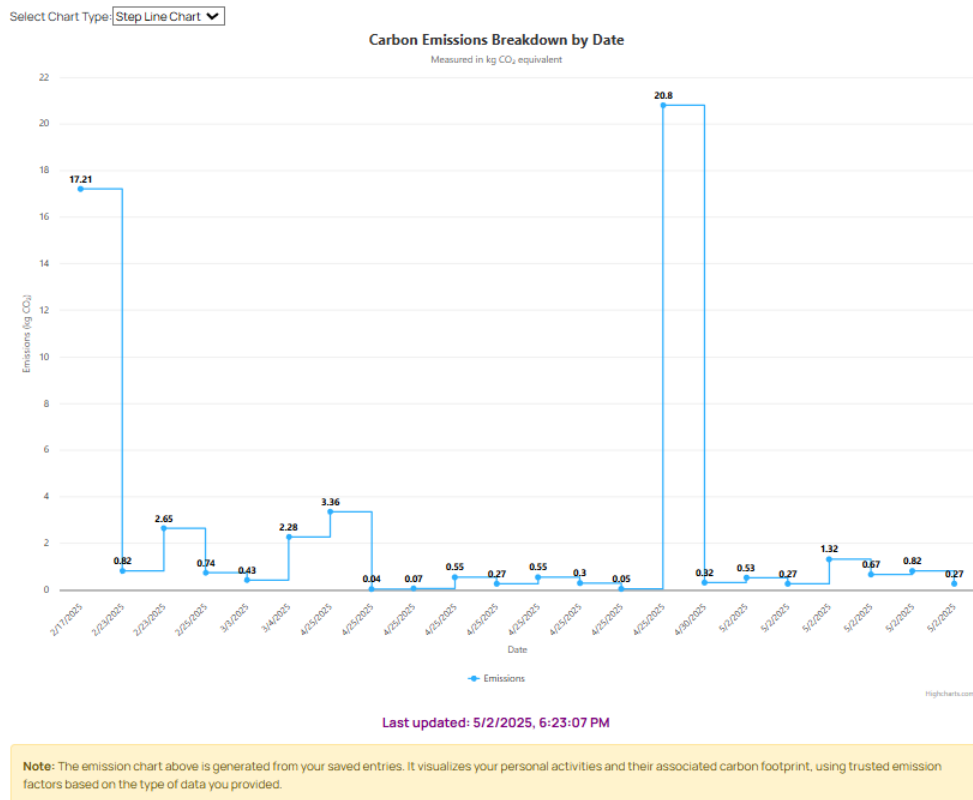


Figure 5.5: Step Line Chart visualisation displaying user-entered carbon emissions over selected dates, highlighting peaks and patterns clearly.

5.5.2 Data Security and Supabase Authentication

The security of user data was a top priority throughout the Green Count development process. Supabase was chosen not only for its real-time data capabilities, but also for its integrated authentication and authorisation features, which ensured secure access to user-specific data.

Token-based access control was used for authentication, which involved issuing secure JWT tokens upon login and verifying each API request. This safeguarded user emissions records and ensured that data could only be viewed and edited by authorised individuals. Supabase's connection with Flask allowed for smooth token validation, lowering the likelihood of unauthorised access.

In addition, backend endpoints featured validation layers to protect against malformed requests and possible injection threats. Although more complex security mechanisms, such as role-based permissions and rate restriction, could be added in future iterations, the current arrangement meets the minimum security

requirements for web apps that handle personal data.

5.5.3 Considerations for Future AI Integration

During the creation of Green Count, exploratory work was done to use machine learning to improve the system's capabilities. Early experiments with training models were carried out using Python tools such as Scikit-learn and TensorFlow to examine emissions data and make simple predictions. These tests revealed the viability of employing AI to deliver insights such as projecting carbon output or recommending lifestyle adjustments.

Furthermore, the OpenAI API was successfully incorporated into the backend, enabling for future growth into natural language processing or AI-powered feedback. This connection opens the door to AI-generated conversational interfaces, intelligent summaries, and environmentally sensitive suggestions.

While early machine learning experiments were conducted, it became clear that such techniques did not naturally align with the core goals of Green Count. The project focuses on factual, user-generated data and precise emissions tracking using verified coefficients. Introducing predictive models could have introduced ambiguity, potentially undermining the system's emphasis on transparency and real-world accuracy. As a result, the decision was made to exclude machine learning from the final implementation.

AI could play an important role in enhancing user engagement and providing smarter environmental insights in future iterations, as long as concerns about transparency, ethics, and user control are adequately addressed.

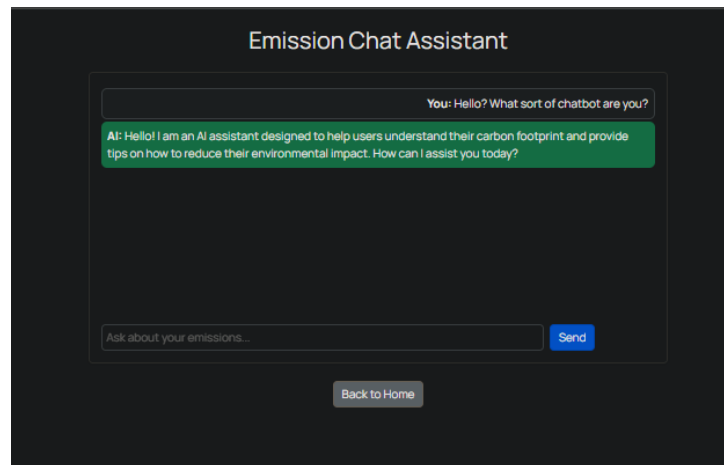


Figure 5.6: Screenshot of OpenAI API integration showing sample chatbot response for carbon emissions advice.

5.6 Summary of Evaluation Findings

Green Count’s evaluation indicates that the system satisfies its basic objectives, as defined in Chapter 1, by providing users with a streamlined and accessible tool to track their carbon emissions. The application’s architecture exemplifies sound technical decisions, with Angular enabling flexible and responsive user interface design, Flask facilitating quick backend connection, and Supabase providing secure, real-time data management.

From a user perspective, Green Count offers intuitive navigation, fast interactions, and informative visual feedback. The combination of reusable components, real-time updates, and interactive charts ensures a smooth experience across the application. Key performance and reliability goals were also met through the careful design of backend routes, automated unit testing with GitHub Actions, and a clear separation of concerns between the frontend and backend systems.

While the system has significant potential, areas for future development have been identified. These include increasing test coverage, improving accessibility, and providing exportable insights or downloadable reports. The foundation laid for AI integration—through machine learning experiments and the inclusion of the OpenAI API—opens up exciting possibilities for future iterations.

Overall, Green Count serves as a solid foundation for an emissions tracking platform, designed with modern technologies and future scalability in mind. It

strikes a balance between functionality, performance, and user experience, creating a useful tool for raising environmental awareness and tracking personal impact.

Chapter 6

Conclusion

6.1 Summary of System Goals

The Green Count project aimed to create a responsive, accessible, and scalable web application that allows users to track and reflect on their own carbon emissions. These objectives were achieved by creating a system that converts user-submitted activity data—such as travel, work, and food consumption—into projected carbon output. AI-generated recommendations and structured visualisations provide insights. The system was designed with adaptability in mind, providing a solid base for potential future improvements such as predictive analytics, exportable reports, and deeper AI integration.

6.2 Outcomes and Forward-Looking Enhancements

The completed system successfully demonstrates how modern web technologies can be combined with environmental data to increase awareness of personal carbon footprints. Angular, Flask, and Supabase were used to build a modular and maintainable solution, and the OpenAI API was used to add conversational feedback and environmentally conscious suggestions. Future iterations could improve the user interface for greater accessibility, expand the emissions coefficient datasets, and include APIs for external lifestyle and transportation data, all of which would increase the system’s relevance and responsiveness.

6.3 Insights from Development Practice

Development followed Agile principles, with sprint-based planning and task tracking through GitHub Projects. Automated testing and continuous integration via

GitHub Actions helped to ensure consistent progress. Unit tests for core components supported frontend functionality, while manual and programmatic endpoint validation ensured backend reliability. These practices emphasised the importance of iterative, test-driven development in producing stable and scalable applications.

6.4 Final Reflections

Green Count provides a practical application of current software development standards to support sustainability-oriented tools. The system combines usability, security, and extensibility into a single platform, providing utility in its current state while also serving as a model for future development. This project demonstrates the potential of well-structured, easily accessible digital tools to help individuals engage in environmentally responsible behaviour.

Bibliography

- [1] Christoph Matthies, Thomas Kowark, Keven Richly, Matthias Uflacker, and Hasso Plattner. How surveys, tutors, and software help to assess scrum adoption in a classroom software engineering project. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, page 313–322, New York, NY, USA, 2016. Association for Computing Machinery.
- [2] Christoph Matthies, Arian Treffer, and Matthias Uflacker. Prof. ci: Employing continuous integration services and github workflows to teach test-driven development. In *2017 IEEE Frontiers in Education Conference (FIE)*, pages 1–8. IEEE, 2017.
- [3] AI Dzhangarov, KK Pakhaev, and NV Potapova. Modern web application development technologies. In *IOP Conference Series: Materials Science and Engineering*, volume 1155, page 012100. IOP Publishing, 2021.
- [4] Hardik Shah. Harnessing customized built-in elements—empowering component-based software engineering and design systems with html5 web components. *arXiv preprint arXiv:2311.16601*, 2023.
- [5] Risto Ollila, Niko Mäkitalo, and Tommi Mikkonen. Modern web frameworks: A comparison of rendering performance. *Journal of Web Engineering*, 21(3):789–813, 2022.
- [6] Tauheed Khan Mohd, Jordan Thompson, Amedeo Carmine, and Grant Reuter. Comparative analysis on various css and javascript frameworks. *J. Softw.*, 17(6):282–291, 2022.
- [7] Gordon Blair, Thierry Coupaye, and Jean-Bernard Stefani. Component-based architecture: the fractal initiative. *annals of telecommunications-Annales des telecommunications*, 64:1–4, 2009.
- [8] Sanjay Khatri Chhetri. Comparative study of front-end frameworks: React and angular. 2024.

- [9] Yishan Li and Sathiamoorthy Manoharan. A performance comparison of sql and nosql databases. In *2013 IEEE Pacific Rim conference on communications, computers and signal processing (PACRIM)*, pages 15–19. IEEE, 2013.
- [10] Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari, and Miriam AM Capretz. Data management in cloud environments: Nosql and newsql data stores. *Journal of Cloud Computing: advances, systems and applications*, 2:1–24, 2013.
- [11] Elar Saks. Javascript frameworks: Angular vs react vs vue. 2019.
- [12] Nilesh Jain, Priyanka Mangal, and Deepak Mehta. Angularjs: A modern mvc framework in javascript. *Journal of Global Research in Computer Science*, 5(12):17–23, 2014.

Appendix

GitHub Repository

The full source code for the Green Count project is available at the following GitHub repository:

<https://github.com/final-year-project-brendanmcshane>

Project Screencast

A screencast demonstrating the core functionality of Green Count is available here:

Green Count Screencast (YouTube):