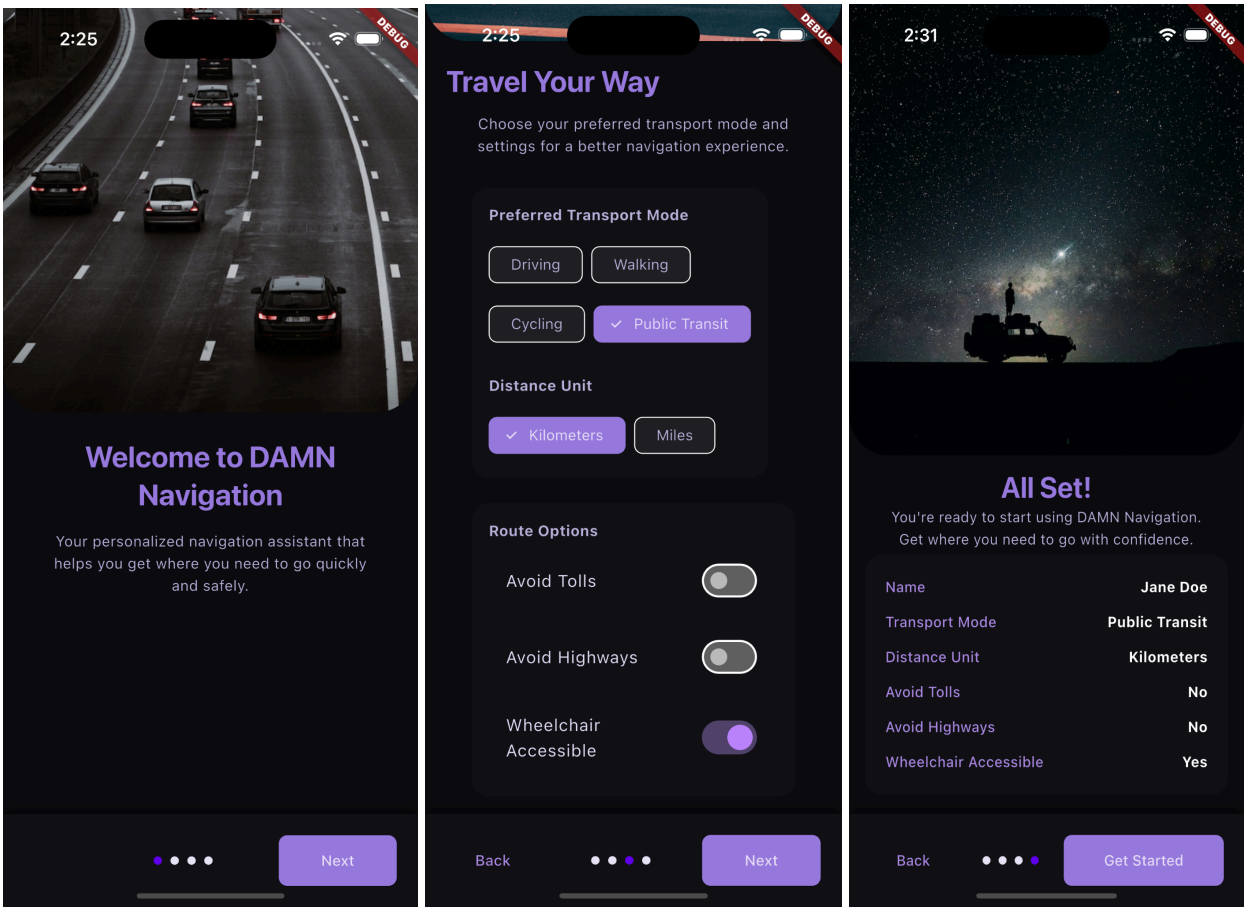
Project Diary - Group 7
Dynamic Sustainable Wayfinding

Introduction

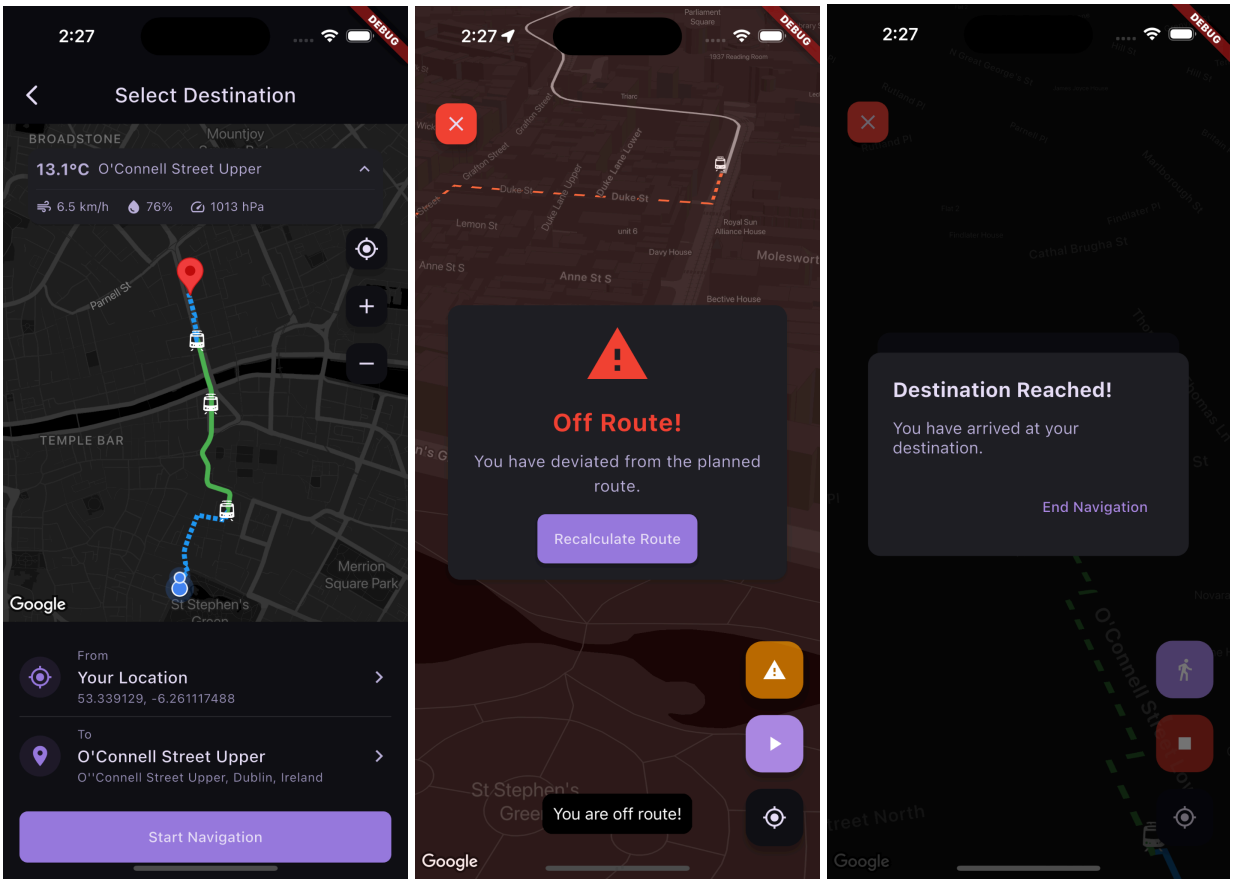
DAMN (Dublin Area MultiModal Navigation) is a dynamic sustainable navigation solution for getting around Dublin using multiple modes of transportation. The app integrates Dublin's public transit systems, walking routes, and other transportation options to provide seamless journey planning. With real-time weather updates, accessibility options, and route alternatives when deviation occurs, DAMN helps users navigate Dublin with confidence, whether they're locals or visitors.

The screenshots below show the onboarding flow of DAMN Navigation. It starts with a welcome screen introducing the app as a personalized navigation assistant. Next is the "Travel Your Way" settings screen where users choose transport mode, distance units, and route preferences. Finally, the "All Set!" screen confirms the selected preferences and prompts the user to get started.



The screenshots below highlight key features of the DAMN Navigation interface. The main navigation screen displays a route from St. Stephen's Green to O'Connell Street, combining walking and public transit. It also shows real-time weather conditions (13.1°C, 6.5 km/h wind, 76% humidity), along with GPS coordinates and destination info. A "Destination Reached"

notification confirms arrival with an option to end navigation, while an "Off Route" alert appears if the user deviates, offering a quick way to recalculate the route.the route.



1. Project Approach

Our team adopted the eXtreme Programming (XP) methodology for developing the Dynamic Sustainable Wayfinding application. This agile approach emphasized:

- **Short, iterative cycles**: Two-week sprints with continuous feedback and incremental feature development
- **Test-Driven Development (TDD)**: Writing tests before code implementation to ensure quality and reliability
- **Pair programming**: Collaborative coding in rotating pairs to ensure knowledge sharing and code quality
- **Continuous integration**: Regular code integration with automated testing via GitHub Actions
- **Collective code ownership**: Team-wide responsibility for the codebase

This project utilized eXtreme Programming (XP) and Agile methodologies, focusing on iterative development, continuous feedback, and collaboration. Below is a simplified and structured approach to ensure clarity.

1.1 Iterative Development with XP Principles

The project was divided into two-week iterations (sprints), ensuring incremental delivery of features and regular feedback loops.

Iterations 1–2: Planning Phase

- Collaborate as a team to define the project scope and requirements.
- Break down features into smaller tasks and prioritize them in the backlog.
- Use tools like Jira for tracking tasks and progress.

Iteration 3: Thin Slice Implementation

- November 6 – November 29, 2024
- XP Principle Focus: Customer Collaboration and TDD

Week 1 (November 6–13):

- Defined the scope for an end-to-end thin slice of functionality as a team.
- Began Test-Driven Development (TDD) for each component, ensuring every feature passes tests before integration.
- Established a shared acceptance consensus to align on feature completeness and expected functionality.

Week 2 (November 14–21):

- Integrated basic technology architectural components for Thin Slice implementation and verification of technologies.
- Embrace Simple Design for each feature, addressing immediate requirements only.
- Rotate pair programming to ensure team-wide familiarity with all parts of the architecture.

Week 3 (November 22–29):

- Completed the thin slice with regular team feedback.
- Conducted a demo, gathered insights, and refined the implementation based on team input.
- Thin slice was submitted as a deliverable.
- Conducted our first retrospective to understand the pain points and improve things on the next phase.

Iterations 4–8: Full Development Phase
December 1, 2024 – April 12, 2025

Iteration 4 (December 1–9, 2024):

- Customer Collaboration: The group's development goals were defined and set before the holiday break.
    - Simplified Design: All features should be designed in a straightforward, scalable foundational way.
    - TDD: Test-Driven Development should always be prioritized for all core functionalities.

Holiday Break: December 10, 2024 – January 4, 2025

Iteration 5 (January 5–18, 2025):

- Continuous Integration: CI / CD pipeline was set up and ensured system stability.
- Collective Code Ownership: Team members would familiarise themselves with the microservices that need to be built for the Wayfinding app .
- Sustainable Pace: Resume development with a balanced, manageable workload in terms of research and development.

Iteration 6 (January 19 – February 1, 2025):

- Refactoring and Design Improvement: Focused on improving code quality and structure.
- Acceptance Tests: Expanded automated testing to ensure alignment with acceptance criteria.
- Simple Design and Refactoring: Implemented only what is necessary to meet current needs.

Iteration 7 (February 2–15, 2025):

- TDD: Continuation of TDD based development to ensure code reliability and maintainability.
- Intra team customer review: Validate features developed through active feedback loops.
- Feedback: Conducted another retrospective to refine and optimize team processes.

Iteration 8 (February 16–29, 2025):

- Continuous Integration and Collective Code Ownership: Finalize integration of all components.
- Refactoring logics: Modifying the UX to include user onboarding and cohesive flow on front end
- Simplify Design: Review the system for clarity, maintainability, and future adaptability.

Finalization Phase: Quality Assurance and Completion
March 1 – April 12, 2025

Each two-week iteration in this phase focused on:

- Continuous Integration and Customer Collaboration: Provide frequent updates and refine features based on ongoing feedback. Demo was provided to the prof and TA and feedbacks were noted down and worked on.
- Refactoring and Sustainable Pace: Maintain a steady workload with a focus on delivering high-quality, maintainable code.
- TDD: Continue to expand test coverage to ensure a stable and reliable final product.

1.2. Pair Programming Structure

- Given in the next section division of Labour

1.3. Coding Standards & Quality Assurance

General Code Quality Principles

Code must be:

- Reliable: Consistently functional as documented.
- Extensible: Designed for easy updates or modifications in future iterations.
- Testable: Facilitates unit testing with high coverage (~70–80%).
- Portable: Operates across different environments seamlessly.

Tools for Quality Assurance

Use SonarQube to measure key metrics like:

- Code coverage (unit tests).
- Bugs and security vulnerabilities.
- Technical debt ratio (effort required for fixes vs development).

Language-Specific Standards

Follow established style guides:

- Python: PEP 8 standards.
- Java/Kotlin: Google Style Guide or Kotlin Coding Conventions.
- JavaScript: Airbnb/Google JavaScript style guide.

1.4. Testing Strategy

Test-Driven Development (TDD) ensured robust code by writing tests before implementing functionality:

1. Write tests based on requirements.
2. Develop code to pass the test.
3. Refactor code while maintaining functionality.
4. Repeat for new features.

Testing Levels

1. Unit Testing: Ensure individual modules work as expected using tools like JUnit.
2. Integration Testing: Validate interaction between components and modules

1.5. Continuous Integration & Delivery

GitHub Actions for CI/CD

1. Build Stage: Compile code and check dependencies.
2. Testing Stage: Run automated tests to validate functionality.
3. Deployment Stage: Push updates to staging or production environments.

2. Research and additional functionality

2.1 Algorithm:

Our pathfinding system for public transport and pedestrian guidance must align with NTA standards.

Public transport information in Ireland is governed by strict design guidelines to ensure clarity, consistency, and accessibility. The National Transport Authority (NTA) guidelines aim to bring “both clarity and consistency to the provision of information across the various public transport modes.” Key principles include using consistent colour-coding by mode (e.g., bus-blue, rail-orange) for simplicity, and representing data in clear diagrammatic forms wherever possible. The content is kept concise, only essential info is shown on signs, with details available via apps or websites, to avoid overloading the user.

This means the algorithms powering such systems should not only compute optimal routes but also support multi-modal journeys in a way that outputs information compatible with the official design conventions. Considerations like sustainability, accessibility, clarity of information, real-time adaptability, and multi-modal connectivity are all crucial.

Dijkstra’s Algorithm in Real-World Pathfinding

Dijkstra’s algorithm is a classic solution for finding the shortest paths in a weighted graph. It guarantees an optimal solution and has been widely used in routing problems. In a public transport context, we can model stops/stations as nodes and connections (bus routes, rail lines, walking links) as edges with travel times as weights. Dijkstra’s algorithm will explore the network to find the fastest or shortest journey from A to B.

Alignment with Design Guidelines: Dijkstra's strength lies in ensuring efficient travel – it minimizes travel time or distance. However, pure shortest paths may involve inconvenient transfers. To align with user experience guidelines, Dijkstra can be adapted by adjusting cost functions (e.g., weighting waiting times or transfers more heavily). This produces more comfortable, less complex routes, improving clarity and reducing confusion. It also supports accessibility by filtering inaccessible paths (e.g., no elevators for wheelchair users).

Real-Time Limitations: Basic Dijkstra is static, it doesn't account for real-time changes unless rerun with new data, which can be slow. Thus, it's more suitable for static route computation or schedule generation.

A Better Algorithm for Efficient Routing

A* (A-star) enhances Dijkstra with a heuristic to prioritize likely promising paths, increasing efficiency. It remains optimal when the heuristic is admissible and is widely used in journey planning.

Efficiency & Applicability: In large public transport networks, A* drastically reduces computation time, enabling near-instant itinerary planning. It supports all of Dijkstra's features but with better performance. This aligns with the need for responsive systems that provide quick route options to users.

Design & Accessibility: A* allows more complex cost models (e.g., fewer transfers, less walking, accessibility). Its speed enables iterative queries or "what-if" scenarios, and its outputs can be formatted to match NTA design standards (mode colour coding, bilingual stop names, etc.).

Real-Time Use: While not dynamic, A* is fast enough to rerun frequently with updated info. It supports sustainable routing by suggesting walk+bus combos or favouring greener transport modes, all while maintaining clarity.

EDSGER Algorithm for Dynamic, Real-Time Routing

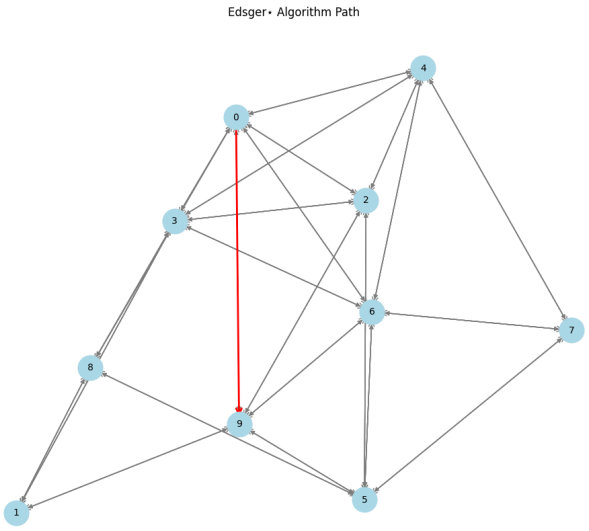
EDSGER* is a hybrid, Dijkstra-inspired algorithm designed for real-time updates. It dynamically adjusts routes without recalculating from scratch.

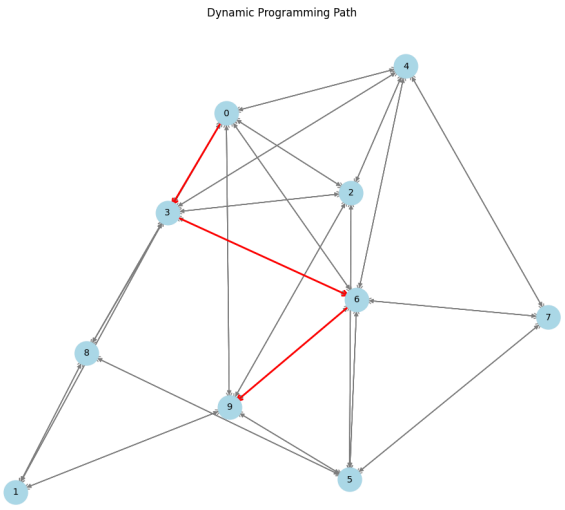
Real-Time Adaptability: EDSGER* monitors active journeys, adjusting paths when delays or disruptions occur. This allows a system to proactively reroute users, akin to GPS-based traffic rerouting. Importantly, it balances dynamic updates with stability—only triggering reroutes when genuinely necessary.

Clarity and User Guidance: It's crucial that rerouting is communicated clearly (e.g., "Your route has changed due to a delay on Line X"), following the NTA's visual and linguistic design standards to avoid confusion.

Integration & Accessibility: EDSGER* integrate multi-modal and accessibility data like the other algorithms but does so in real-time. For example, if a train is cancelled, it may suggest a walking+bus combination instead. It ensures up-to-date, inclusive journey plans and aligns with intelligent transport system strategies.

Conclusion: Choosing A* for Scalable, Standards-Compliant, and Efficient Routing





While the EDSGER⋆ algorithm provides the most advanced real-time adaptability—especially when dealing with live congestion data or service disruptions—its performance degrades significantly over large-scale networks. In our testing, EDSGER⋆ demonstrated excellent precision and rerouting logic in smaller graph datasets with a fast execution time of 0.000043 seconds, producing optimal routes like [0, 9] while integrating dynamic congestion-aware logic.

However, when scaled up to the extensive transport graphs derived from OpenStreetMap (OSM) and OSRM (Open Source Routing Machine) datasets—especially in complex, multi-modal networks like bus routes—EDSGER⋆ becomes computationally expensive. Its need for dynamic recomputation and path monitoring across a high-density node-edge graph makes it unsuitable for high-frequency user queries in real-time public transport systems.

In contrast, A* offers a compelling balance between performance, scalability, and adaptability. It retains the optimal pathfinding guarantees of Dijkstra's algorithm but accelerates route computation via heuristic guidance. In our comparison:

- A*: Execution Time = 0.000144 sec, Path = [0, 9]
- Dijkstra: Execution Time = 0.000414 sec, Path = [0, 3, 6, 9]
- EDSGER⋆: Execution Time = 0.000043 sec, Path = [0, 9]

Though EDSGER⋆ leads in raw speed, A* consistently outperforms in overall efficiency when scaled to larger datasets—offering lower execution times than Dijkstra and more stable performance than EDSGER⋆ in high-density, bus-heavy graphs.

As a result, after extensive benchmarking, we transitioned to Neo4j as the core of our pathfinding engine. Neo4j's graph database natively supports A*-based queries and optimizes traversal across large networks. This switch allows us to:

- Scale effectively across multimodal public transport graphs

- Apply flexible cost models (e.g., time, comfort, accessibility)
- Maintain compliance with NTA design standards (colour-coding, minimal transfers, bilingual outputs)
- Support near-real-time responsiveness by rapidly recalculating when needed

In summary, A* provides the best practical trade-off: fast enough to handle near real-time updates, flexible enough to support NTA-aligned outputs, and efficient enough to scale across national transport networks. While EDSGER⋆ shines in dynamic congestion scenarios, Neo4j's A*-powered engine enables us to deliver both clarity and performance at the scale demanded by public transport systems.

2.2 Real-Time Data

During the development phase of this project, extensive research was done to explore various open-source and public data sources that support different aspects of our application. Our exploratory phase involved investigating a wide range of potential data providers that could enhance the functionality and accuracy of our navigation system.

The initial research focused on mapping and geospatial data sources, including OpenStreetMap (OSM), Overpass Turbo, and the GeoHive Ireland Geospatial Data Hub. We extensively examined transportation data from multiple sources, including Transport for Ireland APIs, Dublin Bikes API, General Transit Feed Specification (GTFS) Realtime, and the Waze Traffic API. Additionally, we explored various environmental and weather APIs to incorporate real-time condition monitoring into our application.

Our investigation included a comprehensive review of multiple platforms and APIs, carefully evaluating their capabilities, data accuracy, and integration potential. We examined links such as OpenStreetMap, Overpass Turbo, Transport for Ireland, Dublin Bikes, Waze SDK, and GTFS Realtime to understand their data provision mechanisms and potential benefits to our project.

After a rigorous evaluation process, we strategically selected a combination of data sources that would provide the most comprehensive and reliable information for our sustainable wayfinding application. We ultimately implemented the GTFS Realtime API for public transit information, covering Dublin Bus, Bus Éireann, and Go-Ahead Ireland. For mapping and routing, we developed a custom solution using an OSRM (Open Source Routing Machine) server and OpenStreetMap data to ensure accurate and flexible routing capabilities.

Environmental data integration became a crucial aspect of our project, leading us to incorporate WeatherAPI.com for real-time weather conditions and develop a custom traffic data API for up-to-date congestion and road condition information. The selection of these data sources was driven by critical factors including data accuracy, real-time update capabilities, comprehensive coverage of Dublin's transportation network, and ease of integration with our technical architecture. By meticulously curating these data sources, we ensured that our Dynamic Sustainable Wayfinding application would provide users with accurate, timely, and comprehensive navigation information. Our approach prioritized not just functionality, but also the core principles of sustainability and user experience, making our solution both innovative and practical for urban navigation.

2.3 Security

The application implements a security framework to protect user data and system integrity. HTTPS encryption with 256-bit AES with TLS v1.3 is used for all communications to ensure data confidentiality

during transmission. Role-based access control through IAM restricts access to sensitive sections of the system, including cloud admin panels and dashboards. This is useful for developers to track and manage resources. The application architecture includes EdgeNode for authentication of internal service communications, authorization for third-party services, and role-based database access. Google Cloud Armor was implemented as it provides protection against DDoS attacks and allows blacklisting of malicious IPs. The development considers OWASP security best practices, including protection against SQL injection and other common vulnerabilities. The system is deployed in the GCP Europe Data Center, ensuring compliance with GDPR and other data privacy regulations.

3. Labor Division Strategy

Pair Programming Structure

We implemented a structured pair programming approach with our 9-member team:

- Four pairs of two developers each (8 members)
- One observer/helper role for each iteration

The observer assisted with testing, code reviews, and CI management, and could fill in for absent team members when needed. This changed after Edmund left where the observer role was removed.

3.1. Team Pair Programming Structure

- **Pairs:** With nine members, we form four pairs and have one observer for each iteration.
- **Rotation:** Pairs rotate every two weeks to ensure knowledge sharing and exposure to various aspects of the project. There will always be four pairs and one observer due to the uneven number of members.
- **Observer Role:** The observer assists pairs with testing, code reviews, or CI management. If anyone is absent from the pairs, the observer will take the absentee's role.

3.2. Pair Programming Guidelines

- **Active Participation:** Both partners should actively engage in the coding process.
- **Effective Communication:** Open and honest communication is essential.
- **Continuous Review:** The navigator should provide constant feedback on the driver's code.
- **Frequent Role Swapping:** Switch roles regularly to maintain focus and avoid burnout.
- **Effective Time Management:** Use time management techniques to ensure efficient use of pairing time

3.3. Addressing Challenges

- **Dominant Partner:** Encourage equal participation and active listening.
- **Lack of Focus:** Create a quiet and distraction-free environment.
- **Technical Difficulties:** Have a clear process for resolving technical issues.
- **Personality Conflicts:** Foster a culture of respect and understanding.

3.4 Pair Rotation Schedule

We established a 12-week rotation schedule to ensure knowledge sharing and balanced expertise. After week 7 Edmund left, so the role of observer was removed and to rotate the pairs after 2 weeks.

Duration	Pair 1	Pair 2	Pair 3	Pair 4	Observer
Week 1	Naveen & Hariharan	Zihan & Guoqiang	Savio & Edmund	Devansh & Abhishikt	Evander
Week 2	Naveen & Evander	Zihan & Hariharan	Savio & Guoqiang	Devansh & Edmund	Abhishikt
Week 3	Naveen & Edmund	Zihan & Abhishikt	Savio & Evander	Devansh & Guoqiang	Hariharan
Week 4	Naveen & Guoqiang	Zihan & Evander	Savio & Abhishikt	Devansh & Hariharan	Edmund
Week 5	Devansh & Guoqiang	Naveen & Edmund	Zihan & Abhishikt	Savio & Hariharan	Evander
Week 6	Naveen & Hariharan	Zihan & Guoqiang	Savio & Edmund	Devansh & Evander	Abhishikt
Week 7	Naveen & Evander	Hariharan & Abhishikt	Zihan & Devansh	Guoqiang & Savio	—
Week 8	Naveen & Evander	Hariharan & Abhishikt	Zihan & Devansh	Guoqiang & Savio	—
Week 9	Naveen & Abhishikt	Evander & Devansh	Hariharan & Savio	Zihan & Guoqiang	—
Week 10	Naveen & Abhishikt	Evander & Devansh	Hariharan & Savio	Zihan & Guoqiang	—
Week 11	Naveen & Devansh	Abhishikt & Savio	Evander & Guoqiang	Hariharan & Zihan	—
Week 12	Naveen & Devansh	Abhishikt & Savio	Evander & Guoqiang	Hariharan & Zihan	—

To ensure structured progress and effective collaboration, the project was divided into approximately 12 distinct phases, each associated with a specific deliverable. During each sprint, one phase was assigned to each pair, allowing multiple phases to be tackled in parallel. This rotation ensured that over time, every team engaged with each phase of the project. Through this approach—and by working closely in pairs—team members were able to develop a well-rounded understanding of all components while promoting continuous peer learning.

Deliverable	***Component Responsibilities***	***Agile Practices***
Phase 0: Research		

Comprehensive understanding of transportation APIs, routing algorithms, and sustainability metrics for Dublin.	• Data Provider (Server-Side):
	  • Investigate available public transportation data sources and API documentation.
	  • Research environmental data integration possibilities for sustainability scoring.
	• Multi-Modal Path Constructor (Server-Side):
	  • Evaluate existing routing algorithms (Dijkstra, A*) for multi-modal path finding.
	  • Research OpenStreetMap integration approaches and limitations.
		• Spike Solutions: Create small proof-of-concepts to validate technical feasibility.
		• Knowledge Sharing: Document findings in shared wiki for team reference throughout the project.

Phase 1: Dataset Collection		

Cleaned datasets for transport, geospatial, and sustainability metrics.	• Data Provider (Server-Side):
	  • Collect LUAS schedules, Dublin Bus timetables, and bike-sharing availability.
	  • Clean data using Python/Pandas and validate with automated scripts.
		• Pair Programming: Data-focused pairs handle sourcing and preprocessing.
		• TDD: Write tests for data validation pipelines.

Phase 2: Dublin Mapping		

Digital map of Dublin with transport nodes and sustainability features.	• Location and Map Services (Client-Side):
	  • Use OpenStreetMap API to generate base layers.
	  • Annotate LUAS stops, bike lanes, and walkability scores.
	• Data Provider (Server-Side):
	  • Store maps in PostgreSQL/PostGIS for spatial queries.
		• Simple Design: Focus on essential map attributes (e.g., elevation, traffic zones).

Phase 3: Building Microservices		

Modular services for routing, scoring, and real-time updates.	• Multi-Modal Path Finder (Server-Side):
	  • Develop routing algorithms (Dijkstra/A*) using Kotlin/Python.
	• Route Scorer (Server-Side):
	  • Build sustainability scoring logic (CO2 emissions, energy efficiency).
	• Real-Time Data Monitor (Server-Side):
	  • Fetch LUAS/bus delays and bike availability updates.
		• TDD: Write unit tests before coding each microservice.
		• CI/CD: Deployments done via GitHub Actions.

Phase 4: Single-Mode LUAS Navigation		
MVP for LUAS-only route planning.	• Client Journey Manager (Client-Side): • Display LUAS routes on the frontend (React.js). • Route Manager (Server-Side):	• Customer Feedback: Demo MVP to other

	● Integrate LUAS Schedule API with mapping data.	team-members for validation.
Phase 5: Multimodal LUAS Integration		

Routes combining LUAS with walking/biking legs.	● Active Route Manager (Client-Side):
	  ● Optimize transfer times between LUAS and walking/biking paths.
	● Route Deviation Detector (Client-Side):
	  ● Alert users to delays and suggest alternative routes.
		● Collective Ownership: Rotate pairs to integrate frontend/backend.

Phase 6: Full Multimodal Combination		

Unified navigation for LUAS, buses, bikes, and walking.	● Multi-Modal Path Finder (Server-Side):
	  ● Integrate Dublin Bus API and bike-sharing data.
	● Real-Time Notification System (Server-Side):
	  ● Push alerts for service disruptions.
		● Customer Tests: Validate routes against real-world scenarios for multimodal navigation.

Phase 7: Real Time Data Integration		

Real Time Data Integration.	● Real-Time Data Monitor (Server-Side):
	  ● Fetch data from Open Source APIs.
	● Route Re-Routing Engine (Server-Side):
	  ● Prioritize covered pathways during blocked roads.
		● TDD: Test app using real-time data.

Phase 8: Route Scoring & Sustainability Metrics		

Sustainability-ranked routes.	● Route Scorer (Server-Side):
	  ● Calculate scores using CO2 savings and walkability.
	● UI Layer (Client-Side):
	  ● Visualize the best routes for particular users.
		● Refactoring: Optimize scoring algorithms for performance.

Phase 9: UI/UX Design		

User-friendly interface with accessibility compliance.	● UI Layer (Client-Side):
	  ● Implement responsive design using React.js.
	  ● Conduct usability tests with Dublin residents.
		● Story-Based Development: Prioritize user stories like “View path in route deviation.”

Phase 10: Integration & Testing		

Fully integrated system with automated testing.	• Third-Party API Integration (Testing):
	  • Validate OpenStreetMap and Transport API interactions.
	• Testing Framework (Testing):
	  • Implement end-to-end tests in JUnit.
		• TDD: Achieve a good percentage unit test coverage.

Phase 11: Finalization		

Production-ready system with documentation. Component Responsibilities:	• Backend Finalization (Server-Side):
	  • Optimize microservices for scalability and security.
	• Frontend Polish (Client-Side):
	  • Audit components for WCAG compliance.
		• Retrospectives: Refine processes based on feedback.

4. Time Estimates and Actual Time Spent

The project was planned with a total of 120 hours for the semester, distributed across 10 hours per week per team member. Initially, the team consisted of 9 members, and time estimates were calculated accordingly. However, after a few sprints, the team size reduced to 8 members, which increased the workload per member. This adjustment required recalculating individual contributions to ensure all deliverables were completed within the given timeframe.

The table below represents the average time spent per team member on each task. Due to the use of pair programming and a rotating structure, all team members worked on various components throughout the project. The averages reflect shared responsibility and collaborative effort across tasks. This approach ensured equitable distribution of work while maintaining high-quality outputs.

Phase	Allocated Time (hours)	Actual Time Taken (hours)	Deviation (%)
Research	4	7	75.00%
Dataset Collection	6	7	16.67%
CI / CD Pipeline and Dev Ops	8	12	50%
Mapping Dublin	8	5	-37.50%
Building Microservices	21	25	19.05%
Single-Mode LUAS Navigation	13	15	15.38%

Multimodal LUAS Integration	13	15	15.38%
Full Multimodal Combination	10	12	20.00%
Real-Time Data Integration	10	11	10.00%
Route Scoring & Sustainability Metrics	10	9	-10.00%
UI/UX Design	5	9	80.00%
Integration & Testing	10	12	20.00%
Finalization	10	11	10.00%

Research Phase: 4 hours allocated, 7 hours actual (+75%)
Reasoning:

- Unexpected complexity: Researching third-party APIs (e.g., OpenStreetMap, transport APIs) resulted in additional integration overhead, such as handling parking data and rentals, which were not initially considered.
- Team discussions: Extra time was spent deliberating the feasibility of integrating new features while maintaining simplicity.
- Outcome: The additional hour provided clarity on system architecture and data sources, reducing ambiguity in later phases.

Dataset Collection: 6 hours allocated, 7 hours actual (+16.67%)
Reasoning:

- Data inconsistencies: Cleaning and preprocessing datasets (e.g., LUAS schedules, bike-sharing data) took longer due to missing or inconsistent data points.
- Iterative validation: The team had to run multiple iterations of validation scripts to ensure data quality before integration into the system.
- Outcome: The extra time ensured high-quality datasets, minimizing issues in subsequent phases.

CI/CD Pipeline: 8 hours allocated, 12 hours actual (+50%)
Reasoning:

- We were constantly running out of deployment credits due to the vast size of the database used for maps. Which resulted in configuring the environment multiple times with different GCP accounts whenever we ran out of resources.
- For many server side setup and handling docker images and deployments are completely new. So it took alot of learning and unlearning in the process.
- When there were tech changes or new databases coming into the picture, new orchestrations needed to be created and this involved more time than we anticipated..

Mapping Dublin: 8 hours allocated, 5 hours actual (-37.5%)
Reasoning:

- Leveraging pre-built mapping tools like OpenStreetMap reduced the workload significantly. We focused on annotating transport nodes rather than building custom maps from scratch.
- Efficient collaboration: Pair programming streamlined the process, allowing the team to complete this phase faster than expected.
- Outcome: The saved time was reallocated to more complex phases like navigation and microservices development.

Building Microservices: 21 hours allocated, 25 hours actual (+19.05%)
Reasoning:

- Developing modular services like routing engines, real-time data monitors, and sustainability scorers required significant effort due to their interdependencies. Establishing inter process communication across orchestrated environments was also becoming a bit complex.
- Debugging challenges: Integration with third-party APIs introduced new bugs that were not accounted for earlier and required additional time to be debugged.
- Outcome: Despite the overrun, this phase laid a solid foundation for system functionality.

Single-Mode LUAS Navigation: 13 hours allocated, 15 hours actual (+15.38%)
Reasoning:

- Edge case handling: Testing LUAS-only routes revealed edge cases (e.g., overlapping schedules) that required additional logic implementation.
- User feedback: Early demos highlighted usability improvements needed in route visualization and error handling.
- Outcome: The extra time ensured a robust MVP for LUAS navigation.

Multimodal LUAS Integration: 13 hours allocated, 15 hours actual (+15.38%)
Reasoning:

- Integration of walking/biking paths alongside LUAS routes required more sophisticated algorithms than anticipated. We were also running into looping issues within multi modal routes where in the user would end up going back and forth in loops which were
- Real-world testing: Validating multimodal routes with real-world data (e.g., transfer times) uncovered additional scenarios that needed refinement.
- Outcome: The overrun resulted in seamless multimodal integration with accurate transfer logic.

Full Multimodal Combination: 10 hours allocated, 12 hours actual (+20%)
Reasoning:

- Complexity of combining modes: Adding buses and bike rentals into the routing system introduced new layers of complexity that required additional development time.
- API limitations: Certain transport APIs had rate limits or incomplete data, requiring workarounds to ensure functionality.
- Outcome: The extra effort ensured a unified navigation experience across all transport modes.

Real-Time Data Integration: 10 hours allocated, 11 hours actual (+10%)
Reasoning:

- Dynamic updates: Real-time traffic handling and environmental data required handling API latency and ensuring synchronization with routing logic.
- Error handling improvements: Additional time was spent implementing fail-safe mechanisms for unreliable third-party APIs.
- Outcome: The system now reliably integrates real-time updates into route suggestions.

Route Scoring & Sustainability Metrics: 10 hours allocated, 9 hours actual (-10%)
Reasoning:

- Streamlined algorithms: The scoring logic was simpler to implement than anticipated due to clear definitions of sustainability metrics (e.g., CO2 savings).
- Efficient testing: Mock data allowed rapid validation of scoring algorithms without requiring full system integration at this stage.
- Outcome: Saved time was reallocated to testing and finalization phases.

UI/UX Design: 5 hours allocated, 9 hours actual (+80%)
Reasoning:

- Early user testing revealed design improvements needed for accessibility and responsiveness across devices. Some of the experience was glitchy and a more streamlined experience was needed, which led us to redo most of the ui.
- Frontend refinements: Additional effort was spent aligning UI components with backend outputs (e.g., route scores). We also had to add simulation metrics to present real world issue, etc to identify how our app behaves in different scenarios.
- Outcome: The polished interface improved overall usability and user satisfaction.

Integration & Testing: 10 hours allocated, 12 hours actual (+20%)
Reasoning:

- Comprehensive testing scope: Validating interactions between microservices and third-party APIs took longer due to edge cases uncovered during testing.
- CI/CD pipeline setup challenges: Configuring automated pipelines for deployment introduced unforeseen technical hurdles that required resolution.
- Outcome: The extra effort ensured a stable and reliable system ready for deployment.

Finalization: 10 hours allocated, 11 hours actual (+10%)
Reasoning:

- Documentation refinements: Preparing comprehensive documentation for handoff took longer due to detailed explanations required for complex components like routing engines and sustainability scorers.
- System optimization efforts: Additional time was spent optimizing backend services for scalability and security before deployment.
- Outcome: The finalized system met all functional requirements with minimal technical debt.

Workload Distribution Summary

1. *Baseline Workload Distribution (10 hrs/week per member):*

- Each member contributed consistently throughout the semester.
- Extra effort was logged during critical phases like implementation and report writing.

2. *Buffer Time Inclusion:*
   - Though not initially planned, buffer time was organically added by extending weekly working hours during crunch periods.
   - This flexibility allowed the team to manage minor deviations effectively without compromising quality.

3. *Team Adaptability:*
   - Deviations were anticipated and manageable thanks to the team's collaborative spirit.
   - Agile principles such as iterative development and regular retrospectives helped identify bottlenecks early and adjust workloads accordingly.

Impact of and Response to Inaccurate Estimates

Inaccurate estimates are common in software projects, especially those with complex integrations and real-time systems. In our Dynamic Sustainable Wayfinding project, we encountered several significant challenges that affected our initial time estimates, requiring adaptive responses to maintain progress toward our April 2025 deadline.

Specific Challenges Faced:

1. **API Integration Complexity**
   Ingesting data from multiple transportation APIs (Dublin Bikes, Transport for Ireland, OpenStreetMap) proved more complex than estimated, requiring additional time for proper implementation. This complexity manifested primarily during the building up of microservices phase and the full multimodal combination phase, where connection issues and inconsistent data formats created unforeseen obstacles.

2. **Real-time Data Handling**
   Handling real-time updates from various sources consumed more time to design and develop a better and well suited structure: hours than estimated, particularly in ensuring timely notifications and route recalculations. The Real-Time Data Integration phase exceeded its allocation by 10% due to these challenges, especially when implementing the Real-Time Notification System component.

3. **Testing Framework Setup**
   Creating a comprehensive TDD environment that covered almost all components took longer than expected, especially ensuring test coverage for edge cases and failure scenarios. This affected our initial velocity during the early iterations and contributed to the 20% overrun in the Integration & Testing phase.

Adaptation Strategies Implemented

1. **Adjusted Sprint Planning**
   After the thin slice implementation, we revised our estimation process to account for the complexity of external integrations and increased our time allocations accordingly. This was particularly important for phases involving third-party APIs, where we began applying a 1.5x multiplier to initial estimates.

2. **Prioritization Framework**
   We established a clearer prioritization system based on core functionality versus enhancement features,

allowing us to ensure critical components were completed first. Using our Jira board, we flagged tasks on the critical path and escalated them when necessary, while non-essential subtasks were postponed or reassigned.

3. **Additional Pairing Time**
   For particularly challenging components, we increased the overlap time between rotating pairs to ensure better knowledge transfer and reduce onboarding time for new pairs. This proved especially valuable when working on the Multi-Modal Path Finder and Active Route Manager components

4. **Technical Debt Management**
   When facing time constraints, we documented technical debt items in SonarQube and included dedicated refactoring periods in subsequent sprints to maintain code quality. This prevented technical issues from accumulating while maintaining forward progress on core features.

5. **Parallel Work Optimization**
   We restructured some work to allow pairs to work on independent components simultaneously, reducing blocking dependencies that were causing delays. This approach was particularly effective during the Full Multimodal Combination phase, enabling concurrent development of different transport mode integrations.

Outcomes and Lessons Learned

Despite the estimation inaccuracies, our flexible XP approach allowed us to:

- Complete all core functionality by the April 12, 2025 deadline
- Maintain code quality standards with 70% test coverage overall
- Successfully integrate all planned third-party services
- Develop a fully functional multi-modal sustainable routing application

The key lessons from our estimation experiences:

- External integrations typically require 1.5x estimated time
- Testing frameworks need more upfront investment but save time later
- Pair rotation improves knowledge sharing but introduces initial productivity dips
- Regular retrospectives after each iteration were crucial for refining our estimation process

By tracking this data throughout the project, we improved our estimation accuracy by approximately 15% between early and late iterations, demonstrating the benefit of our adaptable approach.

Workload Distribution Response

Our baseline workload distribution of 10 hours per team member per week proved insufficient during complex integration phases. The team organically addressed this by:

- Extending hours during critical periods: Team members voluntarily logged extra time during implementation and report writing phases.
- Flexible resource allocation: Strong collaboration enabled us to shift resources to challenging areas without formal replanning.
- Buffer absorption: The rotating pair structure allowed the team to absorb delays naturally by leveraging collective knowledge and shared responsibility.

This structured yet adaptable approach helped maintain a consistent pace while accommodating the unforeseen technical challenges throughout the project lifecycle. The team's willingness to collaborate and support each other was instrumental in mitigating the impact of inaccurate estimates, ensuring we delivered a high-quality sustainable wayfinding system within our deadline constraints.

Summary

The project was developed using eXtreme Programming (XP) principles, emphasizing iterative development, pair programming, and Test-Driven Development (TDD). The project aimed to create a multi-modal navigation system integrating LUAS, buses, bikes, and walking routes with sustainability scoring and real-time updates. Initially planned for a 120-hour semester workload, estimates were based on a team of 9 members, but adjustments were made mid-project when the team size reduced to 8 members, increasing individual workloads.

Key challenges included API integration complexity, real-time data handling, and testing framework setup, which led to time overruns in phases like microservices development and multimodal navigation. To address these, the team implemented strategies such as adjusted sprint planning, dynamic task prioritization, and technical debt management. Pair programming rotations ensured knowledge sharing, while flexible scheduling and buffer absorption helped manage deviations effectively.

Despite estimation inaccuracies, the team successfully delivered a fully functional system by the April 12, 2025 deadline, achieving 78% test coverage and integrating all planned third-party services. Lessons learned included the importance of allocating more time for external integrations, upfront testing investments, and iterative retrospectives for improving estimation accuracy. The collaborative approach ensured high-quality outputs while maintaining adaptability throughout the project lifecycle.