Problem Statement

Good afternoon, we are group 2, I am xxx and here is our presentation on Dream Neighbourhood, an app that helps users find their dream home easily. Firstly, the problem. Finding the right home and neighbourhood in Singapore is difficult as existing platforms, like Property Guru, focus mainly on listings and prices, not lifestyle fit. As you can see here, the filters only include searching by areas or MRT stops, and do not integrate other lifestyle features and amenities. Home seekers hence struggle to compare areas based on amenities, accessibility, and community vibe. On the other hand, dream neighbourhood is a centralised platform that helps users discover neighbourhoods suited to their daily routines, priorities, and long-term needs through our innovative amenity ranking system.

Target Users

Our target audience, Home seekers in Singapore such as individuals or families looking for homes that match their lifestyle and budget, simply have to input the amenities they prioritise. For young professionals and expats, they might eat out regularly or take the MRT. For families, schools and supermarkets for cooking at home. And for retirees, healthcare services. Users just have to click these 6 amenities — MRT station, Healthcare, Sports & Recreation, Hawker Centres, Schools and Supermarkets — in the order of importance to them, and set a maximum walking distance. Our app then assigns the properties in our database a score, and displays the results. This way, users can find a neighbourhood that fits their daily life more conveniently.

Use Case Diagram

The use-case diagram illustrates the interactions between our user roles and system functions. We have one user which can view the property listings and its related actions. They will only have access to our public interface. We will expand on these different interactions, users and systems in our live demo.

External APIs

For our APIs, we rely on the Resale Flat Prices Dataset API for property listings, and Geoapify Places API to identify nearby amenities and their distances from the properties. Next we use OpenStreetMap for our map display, rendered via Leaflet.js. Our backend services connect via Supabase's REST API to store property data, user authentication and each users' saved listings.

Tech Stack

Our frontend is built with React and Vite in TypeScript, and styled with Tailwind CSS. We use React Router and modular components to structure the user flow from amenity ranking to property results and details. On the backend, we use Node.js and Express, with Supabase providing a managed PostgreSQL database for properties, users, and favourites. Business logic such as property ranking and authentication is implemented in our Express API, while Supabase handles data storage.

Live Demo:

Presenter (P) — introducing and explaining the web app features
Target User (U) — asking realistic questions and interacting naturally

P: Hi everyone, welcome to the live demo of *Dream Neighbourhood*!
This is a web application designed to help home seekers find a property that best fits their lifestyle — based on the amenities that matter most to them.

U: Sounds great! So this app basically finds me a home that matches my preferences?

P: Exactly. Instead of just filtering by price or size, Dream Neighbourhood lets you rank amenities — like schools, hospitals, MRT stations, or supermarkets — according to how important they are to you. Then, it scores all properties in the area and shows you the best matches.

Starting the Search

P: Let's start by searching for a property.
Here on the homepage, you can find properties by the amenities section.

U: Okay, I'll select Bukit Merah since I want to stay near central Singapore.

P: Perfect. Now, after selecting Bishan, you can choose the amenities that matter most to you.
Let's say you pick:
1. MRT Station
2. Supermarket
3. School

U: Got it. So the order I pick them actually matters, right?

P: Yes, exactly. Each amenity gets a weight based on its ranking. For example, your top priority, the MRT Station, will have the highest weight, while the last one, recreation, will have the lowest.
You can also set the maximum distance for each amenity — for example, MRT within 500 meters, supermarket within 1000 meters, and so on.

Property Ranking & Results

U: (Clicks Search) It's loading… oh nice, I can see the results now.

P: Yup! Behind the scenes, the system goes through all the properties in Bishan, evaluates them based on your ranked amenities, and assigns a weighted score to each.
The list is sorted so the property that best matches your lifestyle shows up first.

U: The top one says Blk 44 TELOK BLANGAH DR with a score of *1.44*. So that means it matches my priorities the most?

P: Exactly. The higher the score, the better it fits your preferences.
Each property card shows the name, price, number of rooms, and the overall match score.

Filtering After the Search

U: What if I only want to see 3-room units or properties under a certain budget?

P: Good question. You can use the **filters** right here above the results.
After the search, you can refine your list by:

- **Number of rooms**,
- **Maximum budget**, and
- **Minimum area**.

U: Okay, I'll set 3 rooms, budget up to *$1.5 million*, and at least *80 square meters*.

P: Perfect — the list now updates automatically to only show properties that match both your lifestyle ranking *and* your chosen filters.

Viewing Property Details

U: I'll click on Blk 44 TELOK BLANGAH DR to see more details.

P: On this page, you'll find the full property info — price, number of rooms, floor area, and now, because you've done a ranked search, you'll also see the **Nearest Amenities** section.

U: Oh, it shows the distance to Telok Blangah MRT and the nearest supermarket — that's really useful!

P: Exactly. This section only appears after you've completed a ranked search. If you visit a property without searching first, that section won't show yet.

Favouriting a Property (Login & Verification)

U: I want to save this property — can I click the heart icon?

P: You can, but you'll need to log in as a verified user first. Try clicking it now.

U: (Clicks) It says *"Please log in to favorite properties."*

P: That's right. Go ahead and register an account — enter your name, email, and password.

U: Done. It says I need to verify my email.

P: Check your inbox and click the verification link. Once you're verified, you can log in and start favoriting properties.

U: Okay, verified and logged in! Now I'll click the heart again.

P: Perfect — now it's saved! You can view all your favorite properties anytime from your **Favorites** tab.

Wrap-Up

P: So that's *Dream Neighbourhood*!
It combines lifestyle-based ranking with flexible filtering, helping you discover homes that truly suit your daily needs.

U: That's awesome — it's so much more personalized than traditional property websites.

P: Exactly. With ranked amenities, scoring, filtering, and secure user verification, *Dream Neighbourhood* makes home searching smarter, simpler, and tailored to you.

Optional Add-on (If time allows)

U: Does it work for other areas too?

P: Yes — you can search anywhere in Singapore, and the system automatically evaluates properties in that area using live data from Geoapify and OneMap APIs.

SWE practices
In our project, we followed several good software engineering practices to ensure our codebase is maintainable, scalable, and easy to understand.
First, we emphasised documentation.
We included a detailed README file, which explains how to set up, run, and use the project.
This acts as a reference for future developers, ensuring consistency and maintainability over time.
We also included API documentation and the Postman collection, so others can easily test all available endpoints.
This helps new contributors quickly understand the project and reduces time spent on onboarding or debugging.
Our project follows a modular structure, where each module or controller has a specific responsibility.
This improves code reusability and makes the system easier to maintain.
We also used Shadcn UI components, which helped us keep a consistent and clean design across the entire application.
Overall, these practices not only improved the quality of our software but also made collaboration much smoother and more efficient.
Next, for our system architecture. At the top, we have the User Interface, which interacts through the frontend to register, log in, and view properties.

When the user performs an action, the Express.js backend API handles authentication, property ranking logic, and data retrieval.
For authentication, the backend sends a verification email via the Gmail API, and user credentials are securely stored in Supabase PostgreSQL.
From our class diagram,
Our project follows a 3-tier architecture.
The top layer handles the user interface and routes, the middle contains all business logic and authentication consisting of our backend controllers and frontend services, and the bottom manages persistent data and external API requests. This structure ensures modularity, scalability, and security.

VEDA
We introduced the Facade Pattern through UserService, which hides all that complexity.
Now, controllers only need to call a single method like register(), and the UserService handles everything behind the scenes. This made our code cleaner, more modular, and far easier to maintain

Our Repository Pattern keeps all database interactions centralized.
Before, controllers and services directly called Supabase queries, which made maintenance hard.
With UserRepository, we isolate all database logic.
This means if our DataBase schema or technology changes, we only modify one file.

We applied the Factory Pattern through our TokenFactory class.
Previously, every service created its own JWTs or verification tokens, which led to inconsistencies.
The TokenFactory centralizes token generation ensuring consistent structure and easy maintenance.
Now, if we ever change token duration or type, we modify one place instead of several files.

We use the Observer Pattern to handle notifications in our system.
The UserService acts as the Subject, whenever a new user registers, it triggers observers such as EmailObserver to send verification emails automatically.
This keeps our logic modular — if we later add SMS or analytics notifications, we can just attach more observers without changing our core code

To ensure consistency, our UserService follows the Singleton Pattern.
Only one instance is created and reused across all controllers.
This ensures shared observers and configurations, prevents redundant setup, and keeps our system efficient.
It also complements our Facade Pattern, since the same unified service instance handles all user operations throughout the backend.

Now, let’s move on to one of the most important foundations of good software design, the SOLID principles.

First, we have the Single Responsibility Principle (SRP).
This means that every class or module should have only one reason to change. In our project, each package in the folder structure has a distinct responsibility.
For example, our controllers handle request logic, while middleware focuses on request validation or authentication.
This separation makes debugging and scaling much easier.”

(Top-right)
“Next, the Open-Closed Principle (OCP) — open for extension, but closed for modification.
That means instead of editing existing code, we extend it through new modules or patterns.
We use techniques like factory and singleton patterns to add new features without breaking existing functionality.
This helps prevent introducing bugs when new requirements come in.”

(Bottom-left)
“The third one is the Dependency Inversion Principle (DIP).
Here, high-level modules shouldn’t depend directly on low-level modules — both should depend on abstractions.
For instance, our controllers depend on interfaces for the database or email services, not concrete implementations.
This keeps the system loosely coupled, easier to maintain, and allows us to mock or swap dependencies during testing.”

(Bottom-right)
“Finally, the Interface Segregation Principle (ISP).
Rather than having one large interface for everything, we split them into smaller, more specific ones.
So, our authentication, favourites, and property services each have their own interfaces.
That way, classes only depend on what they actually use keeping the codebase modular and efficient.”

EN HSIN Traceability for “Search, Filter & Rank Properties”

Next, I’ll trace one feature end-to-end: **Use Case 1 — Search, Filter & Rank Properties**.

From the use case into the dialog map, the UI flow is simple: user logs in to **Home**, taps **Search** to open Search **Filters**, and hits the **Search button** to get **Search Results**.

This class diagram shows how our Search, Filter, and Rank feature is implemented in a clear, traceable flow. A user’s request goes through `SearchRoutes`, which connects to `SearchController`. The controller then calls `SearchService`, where the main business logic: validation, scoring, and sorting takes place. To fetch data, the service interacts with

`PropertyRepository`. By separating these roles we can easily trace every step of this use case from the user action on the interface all the way down to the database and back.

Here is our sequence diagram that shows the runtime interactions and how the controller validates via filters, queries the repository and invokes the ranking and sorting functions, otherwise throwing an error for invalid inputs.

Our **good designs** show up clearly in code:

- The controller is a thin façade. It parses `town` and `rankings`, delegates to the service, and maps errors to HTTP codes. No SQL, no business logic—just orchestration.
- The service holds the business rules. It does defensive validation up front, calls a **pure scoring** function to compute a total score per property, then **sorts descending** and paginates, keeping it unit-testable and easy to evolve.
- Our repository uses the **Pool** client with parameter binding, so queries are safe and predictable, and data access stays isolated from business logic. Changing scoring touches only service, while changing databases only touches the repository. This shows low coupling and single responsibility.

We also wrote **black-box tests** for ranking and scoring. We cover valid searches across towns and rankings, **boundary** cases like zero amenities or extreme budgets, and **invalid** inputs such as out-of-range ranks. In each test we assert the HTTP status, verify results are **sorted by score (desc)**, and confirm invalid inputs return **clear error messages** rather than crashing.

Lastly, for our future plans, we want to work on expanding to Real-time property updates and an AI-powered recommendation system. For technical improvements, we will split modules and cache frequent queries. We will also beef up our UI with interactive maps with visual heat-zones, and provide accessibility with dark-mode and multi-language support. Finally, we also want to adopt Observer and Factory design Patterns, for real time updates and creating different user types respectively.

Thank you!