Project Submission: Student Commute Optimizer

1. Project Overview & Problem Interpretation

The **Student Commute Optimizer** is a full-stack web application designed to address the inefficiency, cost, and environmental impact of solo commuting for students. The core problem is the absence of a dedicated, secure, and easy-to-use platform for students of the same institution to find and connect with peers traveling along similar routes.

This solution is a map-based carpooling application built around these core features:

- **Anonymous & Secure Registration:** Users sign up with a unique, non-duplicatable username to protect their identity. All interactions on the map are anonymous until a user decides to chat.
- **Intuitive Route Submission:** A simple, interactive map interface allows students to visually pinpoint their start (home) and end (college) locations.
- **Real-Time Match Visualization:** The application displays other students traveling along similar paths as anonymous icons on the map, providing instant visual feedback on potential carpool opportunities.
- **Secure In-App Chat:** Students can initiate a secure, direct chat with potential matches by clicking their icon, allowing for safe and private coordination.

The final product will be a user-friendly, privacy-centric tool that promotes cost-saving, reduces local traffic congestion, and helps build a stronger, more connected student community.

2. Technology Stack

The technology stack was chosen for rapid development, scalability, and strong support for real-time, location-based features, making it ideal for building a robust Minimum Viable Product (MVP).

Component	Technology	Justification & Trade-offs
Frontend	React.js + Leaflet.js	**React** is perfect for a dynamic, component-based Single Page Application. **Leaflet.js** is a lightweight, open-source mapping library. *Trade-off:* Leaflet

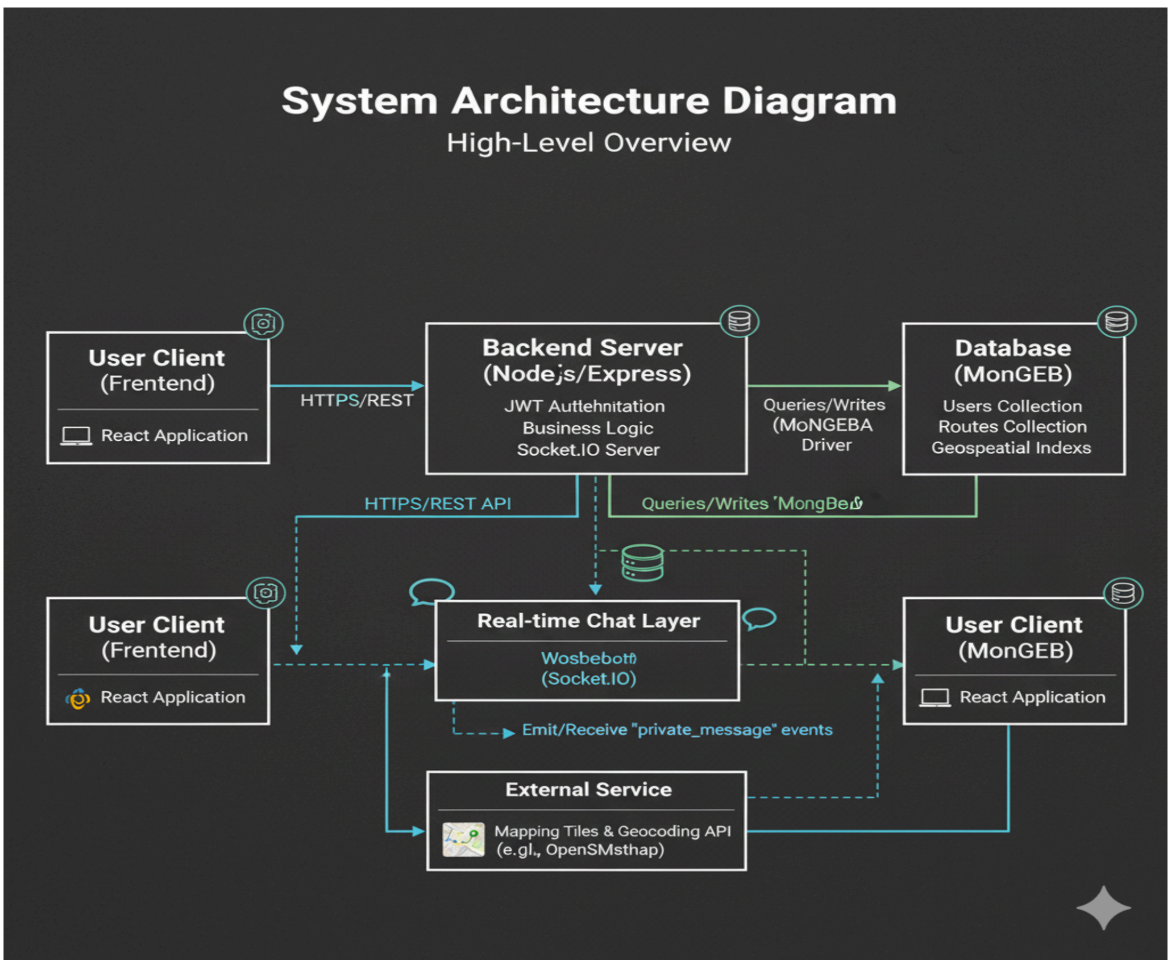
		was chosen over Google Maps API to avoid API key management and focus on core features.
Backend	Node.js + Express.js	**Node.js** provides a fast, non-blocking architecture ideal for handling real-time data like chat and location updates. **Express.js** allows for the quick creation of robust APIs.
Database	MongoDB with Mongoose	**Crucial Decision:** MongoDB's native support for **GeoJSON** data types and efficient geospatial queries ($near, $geoIntersects) is the cornerstone of the route-matching logic, making it far superior to SQL for this use case.
Real-time Chat	Socket.IO	The industry standard for enabling low-latency, bidirectional communication between the client and a Node.js server. It is the ideal choice for implementing the live, anonymous chat.
Geospatial API	OpenStreetMap / Mapbox	An external service is needed to convert addresses to coordinates (geocoding) and generate route polylines. A free service like OpenStreetMap provides this data without needing to build complex routing algorithms from

		scratch.

3. High-Level System Architecture

The system follows a classic 3-tier architecture. The client (browser) communicates with the backend server via a RESTful API for standard operations (auth, route submission) and maintains a persistent WebSocket connection for real-time chat.

- **Client (Presentation Tier):** A React.js single-page application running in the user's browser. It handles UI rendering, map interactions, and communication with the backend.
- **Server (Application Tier):** A Node.js/Express.js server that houses the core business logic, including user authentication, the route-matching algorithm, and API endpoints. It also manages real-time chat connections with Socket.IO.
- **Database (Data Tier):** A MongoDB instance that persists user and route data, leveraging geospatial indexing for efficient queries.



4. Detailed User & Data Flow

This section details the step-by-step journey of a user, outlining their actions and the corresponding data flow through the system.

Step 1: New User Registration

- **User Action:** Navigates to the sign-up page and submits a unique username and a password.
- **Data Flow:**
  1. The React frontend sends a POST request to /api/auth/register with the { username, password } payload.
  2. The backend server validates that the username is unique by **reading** the Users collection.
  3. If unique, the server securely hashes the password.
  4. A new document is **created** in the Users collection with the username and hashed password.
  5. A success response is sent back to the client.

Step 2: User Login

- **User Action:** Enters their username and password on the login page.
- **Data Flow:**
  1. The frontend sends a POST request to /api/auth/login with credentials.
  2. The server **reads** the Users collection to find the user by their username.
  3. It compares the submitted password with the stored hash.
  4. Upon successful authentication, a JSON Web Token (JWT) is generated and sent to the client. The client stores this token for authenticating future requests.

Step 3: Submitting a Daily Commute Route

- **User Action:** Clicks two points on the Leaflet map: a start location (home) and an end location (college).
- **Data Flow:**
  1. The frontend captures the latitude and longitude for both points.
  2. It sends an authenticated POST request to /api/routes with a GeoJSON payload, e.g., { startPoint: { type: 'Point', coordinates: [lon, lat] }, ... }.
  3. The backend validates the data and **creates** a new document in the Routes collection, associating it with the authenticated userId. This route is now active for matching.

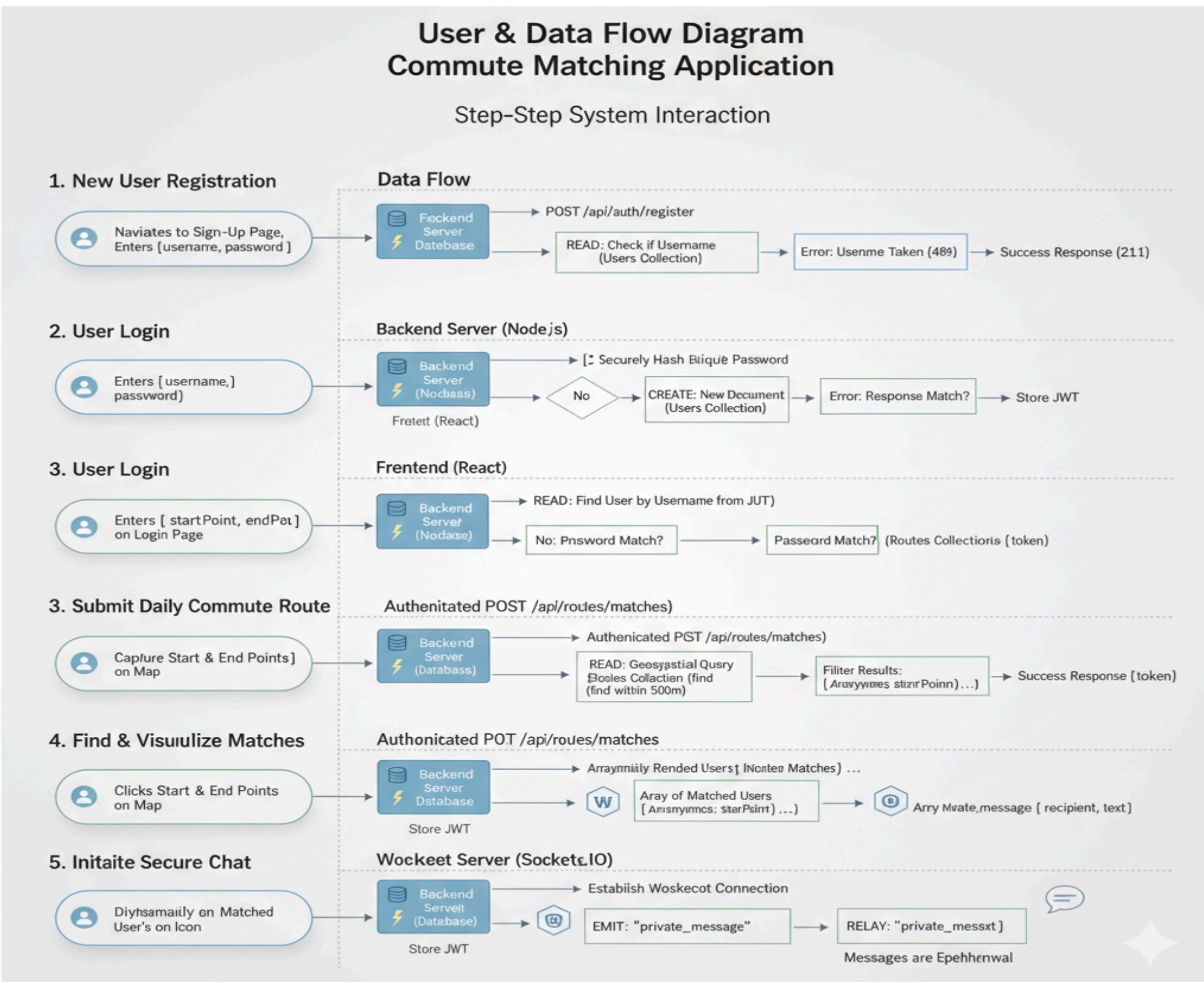
Step 4: Finding and Visualizing Matches

- **User Action:** This process happens automatically after the user submits their route.
- **Data Flow:**
  1. The frontend sends an authenticated GET request to /api/routes/matches.
  2. The backend executes the core matching algorithm: it performs a complex **read** on the Routes collection using an efficient geospatial query ($near) to find other routes with start and end points within a predefined radius (e.g., 500 meters).

3. The server returns an array of matched users, containing only their anonymous username and start point coordinates.
4. The frontend receives this data and dynamically renders anonymous icons on the map for each match.

Step 5: Initiating a Secure Chat

- **User Action:** Clicks on a matched user's anonymous icon on the map.
- **Data Flow:**
  1. A chat window opens in the UI. The client establishes a persistent, real-time WebSocket connection to the server using Socket.IO.
  2. The user sends a message. The client emits a private_message event over the WebSocket, containing the message text and the recipient's username.
  3. The backend server receives the message and relays it instantly to the correct recipient over their WebSocket connection. (Note: For the MVP, chat messages are ephemeral and not persisted in the database).



5. Database Schema (MongoDB)

Two core collections are used. This schema is optimized for performance by denormalizing the username and using specialized indexes.

Users Collection:

```
{
  _id: ObjectId,
  // Unique, indexed username to prevent duplicates as per requirements.
  username: { type: String, required: true, unique: true },
  // Password will be stored as a secure hash using a library like bcrypt.
  password: { type: String, required: true },
  createdAt: { type: Date, default: Date.now }
}
```

Routes Collection:

```
{
  _id: ObjectId,
  // Reference to the user who created the route.
  userId: { type: ObjectId, ref: 'User', required: true },
  // Denormalized for quick display on the map without extra lookups.
  username: { type: String, required: true },
  // GeoJSON format is essential for geospatial queries. A 2dsphere index will be applied here.
  startPoint: {
    type: { type: String, enum: ['Point'], required: true },
    coordinates: { type: [Number], required: true } // [longitude, latitude]
  },
  endPoint: {
    type: { type: String, enum: ['Point'], required: true },
    coordinates: { type: [Number], required: true }
  },
  // A TTL (Time-To-Live) index automatically deletes routes older than 4 hours to keep data
fresh.
  createdAt: { type: Date, default: Date.now, expires: '4h' }
}
```

6. Core Logic: Route Matching Algorithm

The matching logic uses a **Proximity-Based Buffering** algorithm. It checks if a student's start and end points fall within a specified radius of another student's points. This is a deliberate **trade-off** favoring implementation speed and lower computational cost over a perfectly accurate path intersection analysis, which is unnecessary for an MVP.

Pseudocode:

```
FUNCTION find_matches(current_user_route):
  // Define a search radius for matching (e.g., 500 meters)
  SEARCH_RADIUS = 500

  // 1. Find routes with nearby start points using MongoDB's efficient geospatial '$near' query.
  // This is highly optimized by the 2dsphere index.
  potential_matches = DB.Routes.find({
    userId: { $ne: current_user_route.userId }, // Exclude the user's own route
    startPoint: {
      $near: {
        $geometry: current_user_route.startPoint,
        $maxDistance: SEARCH_RADIUS
      }
    }
  }).limit(20) // Limit results for performance

  matched_students = []

  // 2. Further filter the results in memory for destination proximity.
  FOR each other_route in potential_matches:
    // Calculate the distance between the two destinations.
    end_point_distance = calculate_distance(current_user_route.endPoint,
other_route.endPoint)

    // If both start and end points are close, it's a good match.
    IF end_point_distance < SEARCH_RADIUS:
      matched_students.add({
        username: other_route.username,
        location: other_route.startPoint.coordinates
      })

  RETURN matched_students
```

7. Key API Endpoints

The backend will expose the following RESTful endpoints.

Authentication

- POST /api/auth/register
  - **Body:** { "username": "student1", "password": "secure_password" }
  - **Response (201):** { "message": "User created successfully" }
- POST /api/auth/login
  - **Body:** { "username": "student1", "password": "secure_password" }
  - **Response (200):** { "token": "JWT_TOKEN_HERE" }

Routes

- POST /api/routes (Authenticated)
  - **Body:** { "startPoint": { "type": "Point", "coordinates": [72.8, 19.0] }, "endPoint": { "type": "Point", "coordinates": [72.9, 19.1] } }
  - **Response (201):** { "message": "Route created successfully" }
- GET /api/routes/matches (Authenticated)
  - **Response (200):** [{ "username": "student2", "location": [72.81, 19.01] }, ...]

8. Future Enhancements

While the MVP focuses on core functionality, the application is designed for future expansion:

- **Recurring Trips:** Allow students to save and schedule their daily or weekly commutes.
- **User Ratings & Verification:** Implement a simple rating system and optional .edu email verification to build trust within the community.
- **Push Notifications:** Notify users in real-time about new matches or chat messages.
- **Cost-Splitting Calculator:** Integrate a feature to help students easily split fuel costs.
- **Route Polyline Matching:** Move beyond simple proximity to check for significant overlap in the actual route paths for more accurate matches.