Technical Approach

Our technical approach for the Student Commute Optimizer is built on a modern, robust, and scalable foundation. We will leverage a **microservices architecture**, which is a strategic decision to ensure the platform remains flexible, resilient, and highly scalable. By decoupling the core business functionalities into smaller, independent services, our development teams can work in parallel, deploy features independently, and scale specific components based on real-time user demand without affecting the entire system.

The application will operate on a standardized and highly efficient technology stack. The **frontend** will be developed using **React Native**, enabling a single codebase for both iOS and Android applications, reducing development time and maintenance overhead. On the **backend**, we will use **Node.js with the Express.js framework**. This choice is optimal for our application's I/O-heavy nature, including API requests and real-time communication. All data will be stored in a **PostgreSQL** database, specifically chosen for its **PostGIS** extension, which provides powerful geospatial functions essential for our core ride-matching algorithm. The entire system will be deployed on **Amazon Web Services (AWS)**, utilizing managed services for improved security, reliability, and automated scaling.

This architecture consists of several distinct services managed by a central **API Gateway**. The **User Service** handles all authentication and secure data storage, including hashing passwords and verifying student IDs. The core business logic resides within the **Ride Service**, which manages ride data and processes all matching requests. Real-time, anonymous chat functionality is handled by a separate **Chat Service**, likely built on a WebSocket protocol to ensure low-latency communication.

*To visualize this, imagine an image of a **microservices architecture diagram**. It would show the Mobile App and various services (User, Ride, Chat) as separate blocks, all connected through an API Gateway. It would also show the connection to a central database and external services like the Google Maps API. This provides a clear, high-level overview of our system's structure.*

The most critical component of our system is the **route-matching algorithm**. This is not a simple distance calculation but a precise geospatial query leveraging PostGIS capabilities. When a user requests a match, the Ride Service will use a function like ST_Intersection to calculate the exact geographical overlap between the user's route (stored as a polyline) and other active rides in the database. The algorithm will then sort matches by the percentage of route overlap, returning only the most relevant rides. This ensures high-quality matches and an efficient user experience.