# Student Commute Optimizer — Project Plan

**& Design**

**Format requested:** ideation + solutions, high-level diagrams, pseudocode, thought process, choices & trade-offs. This document follows that exact format so you can submit it.

# Project summary

**What:** A carpooling / route-sharing app that matches students traveling similar routes so they can share rides.

**Why:** Reduce single-occupancy commutes, save cost/time, encourage social/eco-friendly commuting while keeping privacy and safety for students.

#### Core constraints:

* + Keep identity private (public-facing unique username that cannot be duplicated).
  + Accurate route overlap matching (road-aware routing).
  + Low cost and reasonably real-time suggestions.

# Goals & deliverables (what to hand in)

Deliverables (for submission):

1. A short **project report** (this document). ✅
2. **High-level diagrams** (architecture, sequence, ER) — included below. ✅
3. **Pseudocode** for the key algorithms (matching, username generation, contact consent) — included below. ✅
4. A **clear step-by-step development plan** and milestones (what to implement for an MVP).

✅

1. Optionally: small code snippets or README pointing to a minimal prototype (optional).

# Minimum Viable Product (MVP)

Core features for the MVP:

* + User registration & login (email, password) — generate a public, unique username.
  + Create a commute (home -> school) with preferred departure time window and seats offered/needed.
  + Snap route to roads (Mapbox/OSRM) to obtain a LineString polyline.
  + Backend matching service that finds candidate routes with route overlap + time proximity.
  + Map UI showing your route and candidate matches.
  + In-app chat between matched students (username only). Contact sharing only after mutual consent.
  + Safety: block/report flow; store minimal PII.

Nice-to-have (phase 2+): Pickup suggestions, cost split, campus timetable integration, dynamic pooling.

# High-level architecture

[Student Mobile/Web UI] <--HTTPS--> [API Gateway] ---> [Auth Service (JWT / OTP)]

OSRM)]

PubSub)] (Push/Email)] Redis Cache

/ RabbitMQ)]

|--> [Routing Service (Mapbox /

|--> [Matching Service (worker)]

|--> [Chat Service (WebSocket /

|--> [Notification Service

|--> [DB: Postgres + PostGIS] +

|--> [Message Queue (Redis Streams

External: Map Provider (Mapbox / OSRM / OSM) for snapping and directions

#### Notes:

* + Matching Service can be synchronous for small scale (on route create) + asynchronous worker for heavier scoring.
  + Use PostGIS for accurate geometry ops and GiST index for spatial queries. For high scale, consider H3/S2 index.

# Data model (conceptual + SQL sketches)

Entities (brief): users, routes, matches, messages, preferences.

## Simplified SQL schema (Postgres + PostGIS)

CREATE EXTENSION IF NOT EXISTS postgis; CREATE TABLE users (

id UUID PRIMARY KEY,

email TEXT UNIQUE, password\_hash TEXT,

username TEXT UNIQUE, -- public name created\_at TIMESTAMP DEFAULT now()

);

CREATE TABLE routes ( id UUID PRIMARY KEY,

user\_id UUID REFERENCES users(id), start\_geom geometry(Point,4326), end\_geom geometry(Point,4326), route\_geom geometry(LineString,4326), start\_time TIMESTAMP,

end\_time TIMESTAMP, seats\_available INT DEFAULT 1, active BOOLEAN DEFAULT true, created\_at TIMESTAMP DEFAULT now()

);

CREATE INDEX idx\_routes\_geom ON routes USING GIST (route\_geom); CREATE TABLE matches (

id UUID PRIMARY KEY,

route\_a UUID REFERENCES routes(id), route\_b UUID REFERENCES routes(id), initiator\_user UUID,

status TEXT DEFAULT 'pending', created\_at TIMESTAMP DEFAULT now()

);

CREATE TABLE messages ( id UUID PRIMARY KEY,

match\_id UUID REFERENCES matches(id), sender\_user UUID,

content TEXT,

created\_at TIMESTAMP DEFAULT now()

);

#### ER (ASCII)

users 1---\* routes \*---\* matches \*---\* messages

| \

| \ (match links route pairs)

# Matching algorithm — design, pseudocode, complexity

## Goals for matching

* + Identify candidate routes whose path overlaps meaningfully with user route.
  + Respect time windows and seat constraints.
  + Return ranked candidates with a score.

## High-level approach (practical & scalable)

1. Snap both users' origins/destinations to road network and get full route\_geom

(LineString). Use Mapbox Directions or OSRM.

1. Sample or keep full LineString. Use a spatial index (PostGIS GiST) to find candidate routes whose bounding boxes intersect a buffered version of the origin route (buffer in meters).
2. For each candidate compute geometric overlap length: overlap\_len = ST\_Length(ST\_Intersection(routeA, buffer(routeB))) in meters (geography cast for meters).
3. Compute time proximity score (difference between preferred departure midpoints).
4. Combine overlap ratio, time proximity, seats, and user preferences into a weighted score.

## Pseudocode (detailed)

function match\_for\_route(routeA\_id, buffer\_meters=200, top\_k=10):

sel = SELECT route\_geom, start\_time, end\_time, seats\_available FROM routes WHERE id=routeA\_id

buffer\_geom = ST\_Buffer(sel.route\_geom::geography, buffer\_meters)::geometry candidates = SELECT id, user\_id, route\_geom, start\_time, end\_time,

seats\_available

FROM routes

WHERE id != routeA\_id AND active = true

AND ST\_Intersects(route\_geom, buffer\_geom)

results = []

for each candidate in candidates:

clipped = ST\_Intersection(candidate.route\_geom, buffer\_geom) overlap\_m = ST\_Length(clipped::geography)

overlap\_ratio = overlap\_m / ST\_Length(sel.route\_geom::geography) time\_diff\_minutes = abs(midpoint(sel.start\_time,end\_time) -

midpoint(candidate.start\_time,end\_time)) in minutes

time\_score = max(0, 1 - time\_diff\_minutes / allowed\_window\_minutes) seats\_score = 1 if seats\_available >= needed else 0

score = w\_overlap\*overlap\_ratio + w\_time\*time\_score + w\_seats\*seats\_score if score >= SCORE\_THRESHOLD: add to results

sort results by score desc return top\_k results

### Complexity

* + Candidate selection uses spatial index, so initial filtering is O(log n + m) where m is number of intersecting candidates.
  + Exact ST\_Intersection is heavier; do it only on filtered candidates.

### Implementation notes & optimizations

* + Use geography casts in PostGIS for meter-accurate length computations:

route\_geom::geography.

* + For large scale: use H3/S2 cell overlays to filter candidates quickly (approximate). Then apply accurate PostGIS intersection.
  + Cache frequent origin-destination snaps.
  + Add thresholds (min overlap meters or min overlap ratio) to reduce false positives (e.g., overlap >= 400m and ratio >= 0.25).

# Pseudocode: username generation & privacy flows

## Unique username generation pseudocode

function make\_unique\_username(desired):

base = normalize(desired) # lowercase, remove spaces, only alphanumerics and underscores

candidate = base i = 0

while db.exists(users WHERE username = candidate): i += 1

candidate = base + str(i) return candidate

**Privacy policy:** only show username and public metadata (rating, seats) to other users. PII (email, phone) remains private until mutual consent.

## Contact sharing / mutual consent flow

1. After a match is accepted, each user sees a Share Contact button.
2. When user A taps Share Contact, the backend marks contact\_shared\_by\_A = true for that match.
3. Contacts are only revealed to the other party when both contact\_shared\_by\_A and

contact\_shared\_by\_B are true.

Pseudocode:

POST /matches/:match\_id/share\_contact # requires auth if current\_user not in match participants: return 403 set contact\_shared\_by\_current\_user = true

if contact\_shared\_by\_both: reveal encrypted contact to both participants via API

# API Design (core endpoints)

All endpoints under /api and use JWT for auth (Bearer token).

#### Auth

* + POST /api/auth/register { email, password, desiredUsername? } -> { user, token }
  + POST /api/auth/login { email, password } -> { user, token }

#### Routes (commutes)

* + POST /api/commutes { start: {lat,lng}, end:{lat,lng}, start\_time, end\_time, seats } -> { route\_id }
  + GET /api/commutes/mine -> list user's routes

#### Matching

* + GET /api/matches/route/:route\_id -> list candidate matches and scores
  + POST /api/matches/request { routeA, routeB } -> create match (initiates chat)
  + POST /api/matches/:id/accept -> accept match
  + POST /api/matches/:id/share\_contact -> mark contact share

#### Chat

* + WebSocket /ws with rooms keyed by match\_id (messages persisted via REST if needed)
  + GET /api/chat/:match\_id/messages -> list messages

#### Admin / safety

* + POST /api/report { match\_id | user\_id, reason }
  + POST /api/block { user\_id }

Examples are in the previous code sketch from the provided MVP.

# Frontend UI & UX (student-facing)

## Key screens (wireframe ASCII)

[Login / Register]

[Create Route]

* Map (pick origin/destination)
* Time window slider
* Seats needed / offered
* [Save]

[Map View] (main)

* Your route polyline (blue)
* Candidate dots / polylines (green)
* Tap candidate -> open Match Card

Match Card:

username: student\_8321

overlap: 68% | estimated shared distance: 3.2 km departure: 08:10 - 08:25

[Chat] [Request] [Save] [Chat]

* Room per match
* Option: [Share Contact] (mutual consent required)

[Profile & Safety]

* Report user
* Block user
* View past rides & rating

UX notes:

* + Show overlap percentage and visual snippet of shared section on map.
  + Suggested pickup point: midpoint of overlapping segment or nearest public stop (bus stop); show ETA.

# Chat & real-time design

* + Use WebSocket server (Socket.IO or native websockets) for low-latency chat.
  + Each match has a room (match:<id>). Only match participants can join.
  + Messages are persisted in DB for history and moderation. Store encrypted at rest if required.
  + Implement rate limiting and content filtering (basic profanity filter / moderation queue).

# Security & privacy checklist

* + HTTPS only in production (TLS).
  + JWT for session tokens; use short lived access tokens + refresh tokens if desired.
  + Passwords hashed with bcrypt/argon2.
  + Minimal PII exposure: show only username until consent.
  + Secure DB access, encrypt backups and credentials in secrets manager.
  + Audit logs for contact sharing and report actions.
  + Allow account deletion (and data erasure) in line with GDPR.
  + CSRF protections on front-end forms where relevant.

# Testing & quality assurance

#### Unit tests

* + Matching function edge cases: nested/looping routes, very short/very long routes.
  + Username creation conflict resolution.

#### Integration tests

* + DB PostGIS geometry operations (intersection & length computations).
  + End-to-end route creation -> matching -> request -> chat flow.

#### Load testing

* + Simulate thousands of route inserts and match queries to validate worker throughput.

**Acceptance criteria** (example for submission)

* + Creating route stores a snapped LineString.
  + Matching returns at least one candidate for known overlapping test cases.
  + Chat messages persist and both users can see them.

# Deployment & scaling (suggested)

#### Local / Student setup

* + Docker Compose with Postgres+PostGIS, app backend, and optionally OSRM (if you avoid Mapbox costs).

#### Production

* + Containerize services and deploy to a managed Kubernetes (GKE/EKS) or PaaS (Fly/Render).
  + Managed Postgres with PostGIS extension.
  + Redis for caching & short-term state.
  + Message queue (Redis Streams / RabbitMQ) for background matching tasks.
  + Autoscaling for match workers.

#### Scaling notes

* + Spatial queries are heavy: use a hybrid approach: H3/S2 cell-based pre-filtering → PostGIS precise computation.
  + Cache snapped routes & common OD pairs.

# Suggested step-by-step development plan (checkpoint-driven)

Implement features in small deliverable checkpoints. Each checkpoint should be demonstrable.

#### Step 1 — Project setup & research

* + Choose stack (Node/Express + Postgres+PostGIS; React + Leaflet for UI).
  + Prepare Docker Compose for Postgres+PostGIS. Run quick proof-of-concept for

ST\_Length & ST\_Intersection operations.

#### Step 2 — Auth & user model

* + Implement register/login, unique username generation, DB schema for users.
  + Tests: register multiple users with same desired username.

#### Step 3 — Route creation

* + Add routes table. Integrate Mapbox/OSRM to snap routes; store route\_geom.
  + Test: manually add two overlapping routes and assert ST\_Intersects.

#### Step 4 — Basic matching

* + Implement matches/route/:id endpoint that returns candidates using buffer & intersection.
  + UI: create route & show matches list.

#### Step 5 — Chat & consent

* + Build WebSocket chat and persist messages.
  + Implement contact share consent.

#### Step 6 — Polishing

* + Add safety features, reporting, block.
  + Add time-window scoring & seats matching.
  + Add map visuals for overlap segment and suggested pickup.

#### Step 7 — Testing & deployment

* + Run integration tests, prepare README + submission artifacts.

# Key choices & trade-offs (thought process)

#### Routing (Mapbox vs OSRM vs Google)

* + Mapbox: easy to use, generous free tier for development, good routing accuracy.
  + OSRM (self-hosted): free and fast but requires hosting resources and maintenance.
  + Google: feature-rich but costlier. Recommendation: start with Mapbox for student project; switch to OSRM if you want fully free stack.

#### Geospatial approach (PostGIS vs H3)

* + PostGIS gives precise geometry ops (intersection length) — ideal for correctness.
  + H3/S2 provides fast cell-based approximate lookup for large-scale matching.
  + Trade-off: use H3 for quick candidate filtering at scale, PostGIS for final accurate scoring.

#### Realtime vs Batch matching

* + Realtime matching on route creation is good UX but can be compute-heavy at scale.
  + Use hybrid: quick candidate filter synchronous + deeper scoring asynchronous (background worker) to update suggestions.

#### Privacy vs matching quality

* + More data yields better matches. But store PII server-side and only reveal after mutual consent.
  + Publicly display only username & necessary metadata.

#### Chat implementation

* + Socket.IO or Firebase for quick implementation; both are valid. For production, a dedicated messaging service scales better.

# Example test case (walkthrough)

1. Alice registers as alice\_1 and creates route: Home A -> College X (08:00-08:15). Route saved and snapped.
2. Bob registers as bob\_2 and creates route: Home B -> College X (07:50-08:20). Route saved.
3. Backend matches Bob as candidate for Alice with overlap\_m = 3.2km, overlap\_ratio = 0.62, time\_diff = 5 minutes.
4. Alice requests match -> match record created -> chat room opens.
5. Both accept; both press "Share Contact"; when both do, contacts are revealed.

# Appendix — useful SQL & functions

#### Compute overlap (example query)

WITH sel AS (SELECT route\_geom FROM routes WHERE id = $1) SELECT

r.id,

ST\_Length(ST\_Intersection(r.route\_geom, ST\_Buffer(sel.route\_geom::geography,

$2)::geometry)::geography) AS overlap\_m, ST\_Length(r.route\_geom::geography) as route\_len\_m

FROM routes r, sel WHERE r.id != $1

AND r.active

AND ST\_Intersects(r.route\_geom, ST\_Buffer(sel.route\_geom::geography,

$2)::geometry)

ORDER BY overlap\_m DESC LIMIT $3;

#### Compute suggested pickup (simple)

* + pickup\_point = ST\_LineInterpolatePoint(shared\_segment, 0.5) — midpoint of shared geometry.

# What to submit (format checklist for graders)

* + This **project document** (PDF or Markdown). ✅
  + Optional: short video/screenshots of the UI and map interactions. ✅/optional
  + Pseudocode & SQL snippets (embedded in this doc). ✅
  + Optionally: a link to a code repo or a zipped MVP (if you implemented one). ✅/optional

# Final notes & next steps I can take for you now

I prepared this full-format project plan exactly as requested: ideation, diagrams, pseudocode, thought process, and trade-offs.

If you want, I can immediately do one of the following (I will produce the content in this chat right away):

* **A.** Convert any section here into runnable code snippets (e.g., matching worker in Python/Node).
* **B.** Produce a one-page slide (PDF) summarizing the project for submission.
* **C.** Create a docker-compose.yml that includes Postgres+PostGIS + OSRM so you can run fully offline.
* **D.** Expand the matching pseudocode into actual SQL+Node code (service implementation). Pick A/B/C/D and I’ll produce it immediately.

*End of document.*