Mastering System Design

Design a Taxi Hailing App (aka Uber)

Understanding the Problem

- Goal: Build a scalable taxi-hailing platform
- Core focus areas:
    - Real-time user-driver matching
    - Geo-location tracking
    - Payment processing
    - High concurrency support
- Why this is hard:
    - Requires low-latency, high-availability systems
    - Involves mobile, backend, and external integrations

Functional Requirements (MVP)

- Rider:
  - Sign up / login
  - Request a ride: source → destination
  - Track driver in real time
  - View ETA & trip progress
  - Pay via app
- Driver:
  - Sign up / login
  - Go online/offline
  - Accept/reject ride requests
  - Navigate to pickup & drop-off
- System:
  - Match riders with nearby available drivers
  - Handle real-time location updates
  - Update ride status (e.g., assigned, en route, completed)
  - Calculate fares and process payments

Non-Functional Requirements

- Scalability: Handle a large number of daily active users and concurrent ride sessions.
- Availability: Ensure high availability, especially during peak usage.
- Low Latency: Quick real-time matching and fast location updates.
- Data Consistency: Eventually consistent location data for real-time accuracy.
- Security: Prioritize user data privacy and secure payment processing.

Assumptions and Constraints

- 📌 Assumptions
  - Users and drivers use GPS-enabled smartphones
  - External APIs (e.g., Google Maps, Stripe) are available and reliable
  - The system launches in a single city/region initially
  - Payment is handled via in-app digital methods only
  - Real-time communication is supported via WebSockets or MQTT
- ⛓️ Constraints
  - Third-party APIs (e.g., Maps, Payments) have rate limits and latency
  - Mobile connectivity is unreliable in some areas (e.g., tunnels, rural zones)
  - Drivers may go offline or change location suddenly
  - Limited compute/storage on mobile devices
  - Strict expectations for low-latency and high availability during peak usage

Real-Time, Mapping & System Challenges

- 🔄 Real-Time System Complexity
  - Match riders and drivers within <2 seconds
  - Handle frequent location updates (every 2–3 seconds)
  - Maintain millions of concurrent connections reliably
- 🗺️ Map & Geolocation Challenges
  - Show live user/driver positions and dynamic routes
  - Perform geocoding + nearby driver search using spatial indexing
  - Deal with GPS inaccuracies and map API rate limits
- ⚠️ System-Level Edge Cases
  - Prevent race conditions in ride assignment
  - Handle stale/missing location data gracefully
  - Design for fallbacks when map/payment APIs fail

Estimating Users & Usage Metrics

- Assume MVP launch in a single metro city.
- Estimated Daily Usage:
  - 🧑 Users: 10 million registered, 1 million DAU
  - 🚗 Drivers: 200,000 active drivers/day
  - 📲 Peak concurrent sessions: ~150K (riders + drivers)
- Activity Estimates:
  - 500,000 ride requests/day → ~6 rides/sec
  - 3x that for location updates → ~18 location updates/sec
  - 1M map tile views/hour → ~280/sec
  - 100K payment transactions/day → ~1.2/sec

Key Bottlenecks & Scaling Challenges

- 🧭 Real-Time Location & Matching
  - Frequent driver updates → write-heavy, low-latency needs
  - Matching engine performance degrades as driver pool scales
  - Must push updates instantly to riders + drivers
- 🗺️ Third-Party Dependency Risks
  - Map APIs: Rate limits, cost, external latency
  - Payments: Timeouts, retries, fraud detection delays
- ⚠️ Platform-Wide Scaling Challenges
  - Push notifications to millions (iOS + Android)
  - Synchronizing state across services → eventual consistency
  - Cost-performance trade-offs at scale (compute, APIs, storage)

Core Microservices Overview

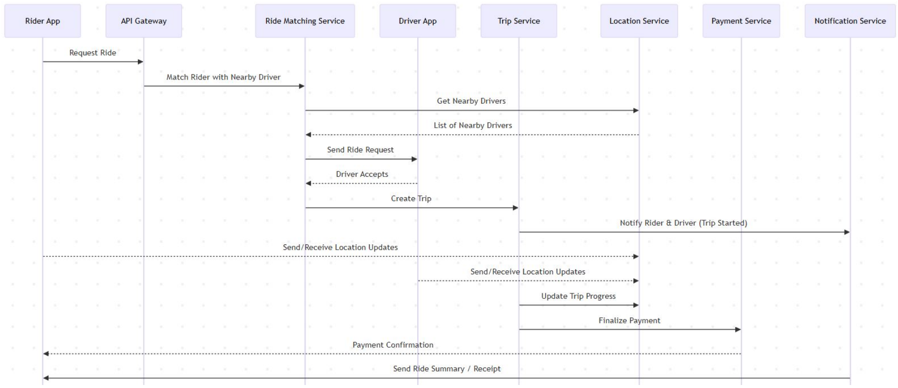
- 👥 User Service   User: onboarding, auth, profile mgmt
- 🚗 Driver Service: Driver onboarding, vehicle info, status
- 📍 Location Service: Real-time location ingestion & geo-indexing
- 📦 Ride Matching Service: Matching riders ↔ drivers based on proximity
- 💳 Payment Service: Ride fare calculation, payment, refunds
- 📊 Trip Management: Trip lifecycle: start, progress, end
- 📬 Notification Service: Push, SMS, and in-app alerts

API Gateway & Communication Patterns

- Patterns Used:
  - 🔁 Internal service-to-service: gRPC or async messaging (Kafka, NATS)
  - 🌐 External APIs exposed via API Gateway (REST/GraphQL)
  - 🧭 WebSockets or MQTT for real-time location/events
- API Gateway Responsibilities:
  - Auth & rate limiting
  - Routing requests to appropriate microservices
  - Aggregating responses for frontend

Client-Backend Interaction Flow

1. 📲 Rider requests a ride → via API Gateway
2. 🧠 Ride Matching Service selects nearby driver
3. 🔁 Driver gets notified → Accepts
4. 📍 Location Service updates rider/driver positions
5. 🚕 Trip Service tracks ride progress
6. 💳 Payment Service handles post-trip payment
7. 📬 Notification Service sends trip updates



Real-Time Communication Design

- Tech Stack:
  - Use WebSockets or MQTT for persistent rider-driver comms
  - Publish-subscribe model (e.g., Redis Pub/Sub, Kafka) for location/trip events
  - Fallback: Polling for unreliable networks
- Scenarios Handled:
  - Live driver movement on rider map
  - Driver arriving → pickup status → in-progress → completed
  - Trip cancellation, surge update, ETA changes

Strategic Tech & Infra Decisions

- Real-Time Communication
  - WebSockets vs MQTT for low-latency, persistent connections
  - gRPC vs REST for service-to-service communication (considering latency and payload size)
- Data Storage
  - SQL (PostgreSQL/MySQL) vs NoSQL (MongoDB, Cassandra) for different use cases (e.g., user data vs ride logs)
  - Redis or Kafka for caching and event streaming
- Geospatial Indexing
  - Geohashing vs H3 for location-based queries and matching (efficiency vs accuracy)
- Scalability
  - Horizontal scaling via cloud services (AWS, GCP, Azure) vs containerization (Kubernetes) for microservices
  - Auto-scaling strategies and load balancing techniques (e.g., round-robin, least-connections)
- Fault Tolerance & High Availability
  - Multi-region replication for disaster recovery (geo-redundancy)
  - Eventual consistency vs strong consistency in trip status and payments

The Final Design - Taxi Hailing App