Ride Hailing Application: GoComet DAW

You are tasked with designing a multi-tenant, multi-region ride-hailing system (think Uber/Ola). The platform must handle driver–rider matching, dynamic surge pricing, trip lifecycle management, and payments at scale with strict latency requirements.

1. Business Logic

Overview

Design a ride-hailing system (like Uber/Ola) supporting:

- Real-time driver location updates (1–2 per second)
- Rider requests (pickup, destination, tier, payment method)
- Driver–rider matching within 1s p95
- Trip lifecycle (start, pause, end, fare, receipts)
- Payments via external PSPs
- Notifications for key ride events

Core APIs (minimal expectation)

`POST /v1/rides` — Create a ride request

`GET /v1/rides/{id}` — Get ride status

`POST /v1/drivers/{id}/location` — Send driver location updates

`POST /v1/drivers/{id}/accept` — Accept ride assignment

`POST /v1/trips/{id}/end` — End trip and trigger fare calculation

`POST /v1/payments` — Trigger payment flow

The developer can decide whether to implement asynchronous messaging (queues, topics) or keep synchronous APIs for simplicity. Use Postgres/ MySQL for Transactional data, Redis for Caching

- Implement APIs with validation and idempotency.
- Manage trip and assignment state transitions cleanly.
- Handle common edge cases (timeouts, declined offers, retries).

2. Scalability & Reliability

Guidelines

- Design the system to scale for ~100k drivers, ~10k ride requests/min, and ~200k location updates/sec.
- Use caching or local indexing for fast driver lookups.
- Keep writes region-local; avoid blocking on cross-region sync.
- Ensure APIs and components are stateless and can scale horizontally.

3. New Relic for Monitoring

Integrate New Relic (100GB free for new accounts) to monitor API performance. They should:

- Track API latencies under real load.
- Identify bottlenecks and slow database queries.
- Set up alerts for slow response times.

4. Optimize API Latency

Refactor all 3 apis to Reduce latency. Few things to consider are:

- Using Database Indexing
- Implementing Caching
- Optimising Queries
- Handling Concurrency
- Ensuring data consistency

5. Ensure Atomicity and Consistency

Use transactions to handle concurrent writes without race conditions.

- Implement cache invalidation strategies to ensure up-to-date rankings.
- Guarantee that driver allocations remain consistent under high traffic.

6. Build a Simple Frontend UI with Live Updates

Candidates must create a basic frontend interface to display

- Real time updates for ride/ driver allocation

Evaluation Criteria

- Bug free working

- Code Quality & Efficiency
- PRs and change managementUnit tests
- Performance Optimization
- Data Consistency
- Monitoring & Analysis
- Basic API Security
- Problem-Solving & Ownership
- Documentation (HLD/LLD)
- Demo

Final Deliverables

- Backend code.
- Frontend code.
- Performance report with New Relic (dashboard or screenshots).
- Documentation.