Plus Points in Implementation (Overall Evaluation Criteria)

1. Authentication:
- Implement robust user authentication protocols to ensure secure access.

2. Cost Estimation - Time and Space:
- Conduct a thorough analysis of time and space complexity in the system.
- Utilize efficient algorithms and data structures to optimize both time and space requirements.

3. Handling System Failure Cases:
- Implement fault-tolerant mechanisms to address system failures.
- Employ backup and recovery strategies for data integrity.
- Develop comprehensive error recovery procedures to minimize downtime.

4. Object-Oriented Programming Language (OOPS):
- Choose a robust OOPS language for structured and modular code.
- Leverage OOPS principles such as encapsulation, inheritance, and polymorphism for maintainability and extensibility.

5. Trade-offs in the System:
- Clearly define and document trade-offs made during system design.
- Evaluate and communicate the rationale behind architectural and design decisions.
- Consider trade-offs in terms of performance, scalability, and maintainability.

6. System Monitoring:
- Implement comprehensive monitoring tools to track system performance.
- Utilize real-time dashboards and logging mechanisms to promptly identify and address issues.

7. Caching:
- Integrate caching mechanisms to enhance system response times.
- Utilize caching for frequently accessed data to reduce database load.
- Implement cache eviction policies for optimal resource utilization.

8. Error and Exception Handling:
- Develop a robust error and exception handling framework.
- Provide meaningful error messages for effective debugging.
- Regularly review and update error-handling strategies based on system usage patterns.

Instructions:

1. Read and Understand the Problem Statement:

- Carefully read the problem statement provided. Understand the requirements, inputs, expected outputs, and any constraints mentioned.

2. Choose a Programming Language:
- Select a programming language you are comfortable with and that is suitable for solving the problem described in the case study.

3. Design Your Solution:
- Plan the overall structure of your solution. Consider the algorithms, data structures, and any potential optimizations needed.

4. Write the Code:
- Implement your solution in code. Follow best practices for coding standards, such as meaningful variable names, proper indentation, and comments where necessary.
- Break down the problem into smaller functions or modules to improve code readability and maintainability.

5. Test Your Code:
- Test your code thoroughly with different sets of input data, including edge cases and boundary conditions.
- Ensure that your code produces the expected outputs for all test cases.

7. Document Your Code :
- Consider adding documentation or comments to explain the logic and purpose of your code, especially for complex parts or algorithms.

8. Submit Your Solution:
- Once you're satisfied with your code and it meets all the requirements, submit your solution on GitHub and share the GitHub link.

9. Demonstration:
- Include a demonstration video showcasing key features of the ride-sharing platform.
- Alternatively, use screenshots to visually highlight the user interface and functionality.

Intelligent Alert Escalation & Resolution System

Context

MoveInSync operates multiple fleet-monitoring modules (Safety, Compliance, Feedback). Each generates alerts — like overspeeding, expiring documents, or poor driver feedback.
Currently, these alerts are static and require manual review. The operations team wants a **smart alert engine** that can *automatically escalate, de-escalate, and close alerts* based on dynamic rules, while providing a **Dashboard** to visualize alert trends.

Problem Statement

Design and implement an **Intelligent Alert Escalation & Resolution System** that does the following:

1. **Centralized Alert Management**

    - Design the system to **ingest alerts from multiple source modules** (e.g., Overspeeding, Document Expiry, Negative Feedback) through a **central API**.

For this case study, the candidate should implement an API endpoint that can accept and create alerts for various event types, normalizing them into a common structure.

    - Stores them in a unified format with key fields: (Below are mandatory)
      {alertId, sourceType, severity, timestamp, status, metadata}.

    - Alerts can have one of the following states:
      OPEN ⟶ ESCALATED ⟶ AUTO-CLOSED ⟶ RESOLVED.

2. **Lightweight Rule Engine**

    - Implement a configurable rule system that defines **escalation and closure thresholds** per alert type.

    - Example rules:

        - **Overspeeding**: escalate to *Critical* if repeated 3 times within 1 hour.
        - **Compliance**: auto-close if document renewed.
        - **Negative Feedback**: escalate if two bad feedbacks are received within 24 hours.

    - Rules should be stored in memory or a simple JSON/YAML file and evaluated dynamically, not hardcoded.

👉 *Bonus*: The rule engine should allow simple DSL-like syntax, e.g.

```
{
 "overspeed": { "escalate_if_count": 3, "window_mins": 60 },
 "feedback_negative": { "escalate_if_count": 2, "window_mins": 1440 },
 "compliance": { "auto_close_if": "document_valid" }
}
```

3. **Auto-Close Background Job**

    - A periodic background worker scans alerts every few minutes to:

        - Check if conditions to auto-close are met.
        - Change alert state from OPEN/ESCALATED ⟶ AUTO-CLOSED.

    - Store auto-closure events.

4. **Dashboard View (Minimal UI)**
    - Design a simple dashboard or API that aggregates alerts by severity and category:

        - Show counts of Critical, Warning, Info.
        - List top 5 drivers with most open alerts.
        - Show recent auto-closed alerts for transparency.

Functional Requirements

- Alerts and rules must be **decoupled** — updating a rule should automatically affect future alert evaluations.
- Background jobs should be **idempotent** (safe to re-run).
- Escalation should trigger only once per condition (avoid loops).
- The system must allow **manual resolution** if needed, but focus is on **auto-driven behavior**.
- Alerts expire after a defined time window (configurable).

Rule-based escalation demo
Post 3 overspeed alerts for one driver within an hour → system escalates to Critical.

Auto-close demo
Post a compliance alert → after "document renewed" event or time window passes, alert auto-closes.

Design a **Dashboard (web UI or API)** that provides **real-time visibility and analytics** for all alerts.

1. **Top Offenders / Entities**
    - Table or leaderboard listing:
        - Top 5 drivers with the most open or escalated alerts.
    - Stream or table of recent alert lifecycle events (created, escalated, auto-closed, resolved).
    - Display timestamps, source type, and current state.
2. **Auto-Closed Alerts Transparency**
    - Section showing recently auto-closed alerts with reason (e.g., "Document renewed", "Time window expired").
    - Include filters like "Show last 24 hours / 7 days".
3. **Trend Over Time**
    - Line graph showing how total alerts, escalations, and auto-closures have evolved daily/weekly.
    - Helps visualize impact of rule tuning or operational changes.
4. **Alert Drill-Down**
    - Click any alert to view:
        - History of state transitions (OPEN → ESCALATED → AUTO-CLOSED/RESOLVED).
        - Associated metadata (driver ID, vehicle, event count, rule triggered).
        - Option for manual resolution.
5. **Configuration Overview (Bonus)**
    - Optional admin view displaying currently active rule thresholds and alert configuration.
    - Helps visualize which rules are driving current escalations.