

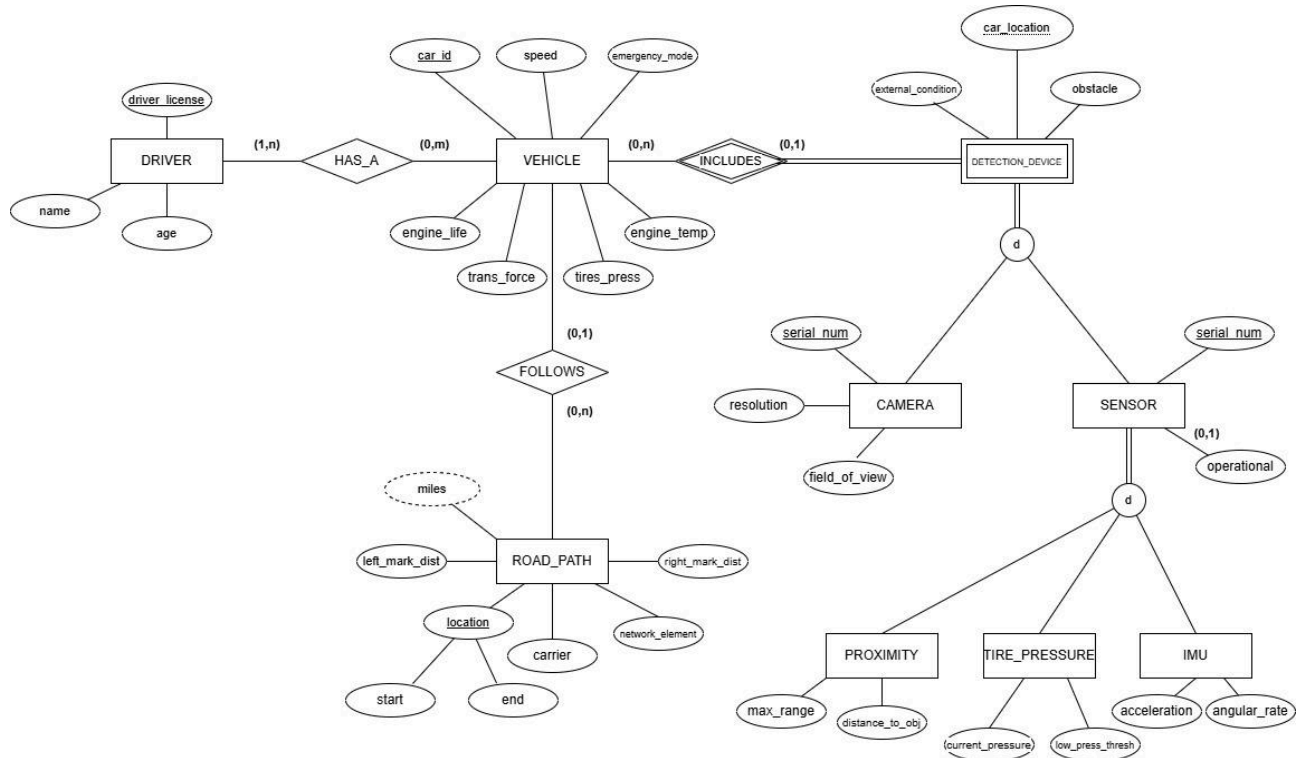
# Autonomous Vehicle System

Anwar, John, Daniel, Gabe, Rodolfo, Joseph

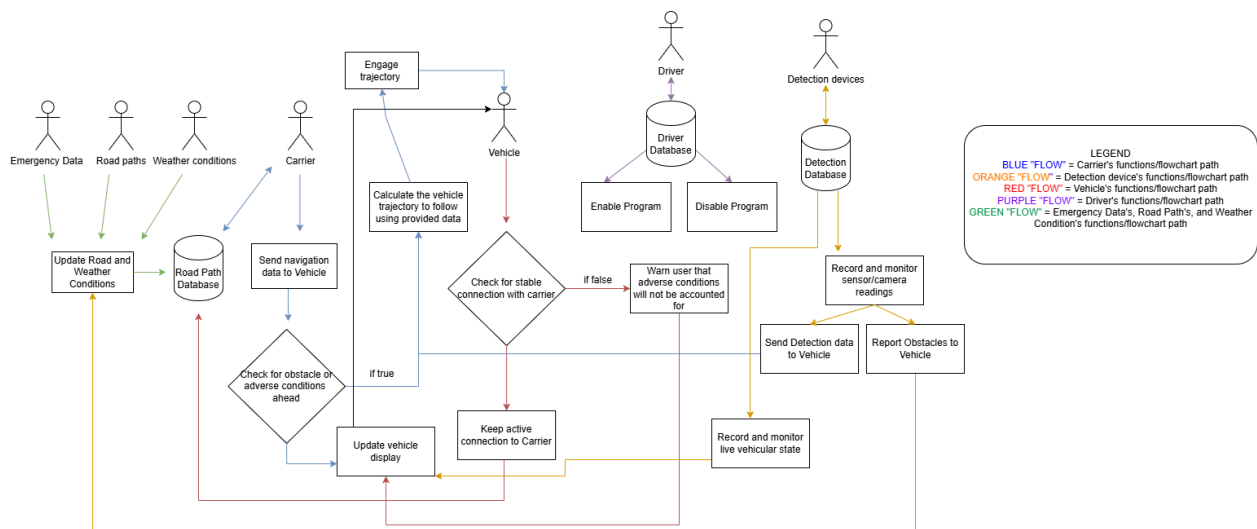
CS 250: Introduction to Software Systems

Professor Umut Can Cabuk

## Database Diagram:



## Updated SWA Diagram:



**UML Class Diagram for the Car System**

```

classDiagram
    class Driver {
        driverLicense: string
        age: int
        name: string
        activateEmergencyMode(): bool
        activateAutonomousDriving(): bool
    }
    class Vehicle {
        carID: string
        path: RoadPath
        speed: int
        emergencyMode: bool = false
        engineLife: float
        engineTemperature: float
        transmissionOnce: float
        tirePressure: float
        turnLeft(float degrees): void
        turnRight(float degrees): void
        activateEmergencyMode(string disaster): void
        changeTrajectory(Obstacle Object): void
        followRoad(RoadPath): void
        getNetworkInfo(NetworkElement antenna): RoadPath
        saveAnalytics(): void
        throttle(float intensity): void
        break(float intensity): void
    }
    class NetworkElement {
        serialNum: string
        carrierInfo: Carrier
        updatedRoad: RoadPath
        communicatePath(RoadPath path): RoadPath
        communicatePath(RoadPath path, string emergency): RoadPath
        getCarrierInfo(): Carrier
        getRoadInfo(): RoadPath
    }
    class RoadPath {
        start: Location
        end: Location
        leftMarkDist: float
        rightMarkDist: float
        miles: float
        getMarks(): float[] marks
        getMiles(): float
    }
    class Carrier {
        carrierID: string
        currentRoad: RoadPath
        disaster: string
        providePath(Location origin, Location destiny): RoadPath
        communicateDisaster(Location currentLocation): string
    }
    class DetectionDevice {
        carLocation: string
        scannedObstacle: Obstacle
        sensors: Sensor
        getScannedObstacle(): Obstacle
        isDanger(Obstacle scannedObstacle): bool
        communicateDanger(): Obstacle
        scanEnvironment(): void
    }
    class Obstacle {
        width: float
        height: float
        distance: float
        getWidth(): float
        setWidth(float width): void
        getHeight(): float
        setHeight(float height): void
    }
    class Camera {
        serialNum: string
        resolution: string
        fieldOfView: float
        conditions: ExternalCondition
        detectSurroundings(): Obstacle
        captureFrame(): ImageData
    }
    class Sensor {
        serialNum: string
        isOperational: bool
        conditions: ExternalCondition
        method/type: type
        detectSurroundings(): Obstacle
        checkStatus(): bool
    }
    class ProximitySensor {
        distanceToObject: float
        maxRange: float
        detectSurroundings(): Obstacle
        getDistanceToObject(): float
    }
    class IMUSensor {
        acceleration: Vector3D
        angularRate: Vector3D
        detectSurroundings(): Obstacle
        getVehicleAltitude(): Vector3D
    }
    class ExternalCondition {
        weatherConditions: string[]
        visibility: float
        weather: string
        getWeather(): string
        setWeather(string weather): void
        getVisibility(): float
        setVisibility(): void
    }
    class TirePressureSensor {
        currentPressure: float
        psiPressureThreshold: float
        detectSurroundings(): Obstacle
        isPressureLow(): bool
    }

    Driver --> Vehicle
    Vehicle --> NetworkElement
    NetworkElement --> RoadPath
    RoadPath --> Carrier
    Carrier --> RoadPath
    Carrier --> DetectionDevice
    DetectionDevice --> Obstacle
    DetectionDevice --> Sensor
    Sensor --> ProximitySensor
    Sensor --> IMUSensor
    Sensor --> TirePressureSensor
    Camera --> DetectionDevice
    Camera --> Sensor
    ExternalCondition --> DetectionDevice
    ExternalCondition --> Sensor
    ExternalCondition --> TirePressureSensor
  
```

**Class Details:**

- Driver**
  - Attributes: `driverLicense: string`, `age: int`, `name: string`
  - Operations: `activateEmergencyMode(): bool`, `activateAutonomousDriving(): bool`
- Vehicle**
  - Attributes: `carID: string`, `path: RoadPath`, `speed: int`, `emergencyMode: bool = false`, `engineLife: float`, `engineTemperature: float`, `transmissionOnce: float`, `tirePressure: float`
  - Operations: `turnLeft(float degrees): void`, `turnRight(float degrees): void`, `activateEmergencyMode(string disaster): void`, `changeTrajectory(Obstacle Object): void`, `followRoad(RoadPath): void`, `getNetworkInfo(NetworkElement antenna): RoadPath`, `saveAnalytics(): void`, `throttle(float intensity): void`, `break(float intensity): void`
- NetworkElement**
  - Attributes: `serialNum: string`, `carrierInfo: Carrier`, `updatedRoad: RoadPath`
  - Operations: `communicatePath(RoadPath path): RoadPath`, `communicatePath(RoadPath path, string emergency): RoadPath`, `getCarrierInfo(): Carrier`, `getRoadInfo(): RoadPath`
- RoadPath**
  - Attributes: `start: Location`, `end: Location`, `leftMarkDist: float`, `rightMarkDist: float`, `miles: float`
  - Operations: `getMarks(): float[] marks`, `getMiles(): float`
- Carrier**
  - Attributes: `carrierID: string`, `currentRoad: RoadPath`, `disaster: string`
  - Operations: `providePath(Location origin, Location destiny): RoadPath`, `communicateDisaster(Location currentLocation): string`
- DetectionDevice**
  - Attributes: `carLocation: string`, `scannedObstacle: Obstacle`, `sensors: Sensor`
  - Operations: `getScannedObstacle(): Obstacle`, `isDanger(Obstacle scannedObstacle): bool`, `communicateDanger(): Obstacle`, `scanEnvironment(): void`
- Obstacle**
  - Attributes: `width: float`, `height: float`, `distance: float`
  - Operations: `getWidth(): float`, `setWidth(float width): void`, `getHeight(): float`, `setHeight(float height): void`
- Camera**
  - Attributes: `serialNum: string`, `resolution: string`, `fieldOfView: float`, `conditions: ExternalCondition`
  - Operations: `detectSurroundings(): Obstacle`, `captureFrame(): ImageData`
- Sensor**
  - Attributes: `serialNum: string`, `isOperational: bool`, `conditions: ExternalCondition`
  - Operations: `method/type: type`, `detectSurroundings(): Obstacle`, `checkStatus(): bool`
- ProximitySensor**
  - Attributes: `distanceToObject: float`, `maxRange: float`
  - Operations: `detectSurroundings(): Obstacle`, `getDistanceToObject(): float`
- IMUSensor**
  - Attributes: `acceleration: Vector3D`, `angularRate: Vector3D`
  - Operations: `detectSurroundings(): Obstacle`, `getVehicleAltitude(): Vector3D`
- ExternalCondition**
  - Attributes: `weatherConditions: string[]`, `visibility: float`, `weather: string`
  - Operations: `getWeather(): string`, `setWeather(string weather): void`, `getVisibility(): float`, `setVisibility(): void`
- TirePressureSensor**
  - Attributes: `currentPressure: float`, `psiPressureThreshold: float`
  - Operations: `detectSurroundings(): Obstacle`, `isPressureLow(): bool`

**Associations and Dependencies:**

- Driver** is associated with **Vehicle**.
- Vehicle** is associated with **NetworkElement**.
- NetworkElement** is associated with **RoadPath**.
- RoadPath** is associated with **Carrier**.
- Carrier** is associated with **DetectionDevice**.
- DetectionDevice** is associated with **Obstacle**.
- DetectionDevice** is associated with **Sensor**.
- Sensor** is associated with **ProximitySensor**.
- Sensor** is associated with **IMUSensor**.
- Sensor** is associated with **TirePressureSensor**.
- Camera** is associated with **DetectionDevice**.
- Camera** is associated with **Sensor**.
- ExternalCondition** is associated with **DetectionDevice**.
- ExternalCondition** is associated with **Sensor**.
- ExternalCondition** is associated with **TirePressureSensor**.

**Notes:**

- While `emergencyMode == false` the system is activated.
- The emergency mode can be activated automatically when receiving a natural disaster.
- `saveAnalytics` will store the information of the car in a database.
- The emergency is received through the carrier.
- The attribute `currentRoad` stores the `RoadPath` that the car has to follow. It is extracted from will have different values, and those values are the one we will send the vehicle.
- The path is adjusted here in the `NetworkElement` if there's an emergency.
- Through getting the satellite info.
- Location is an external class that we don't need to define in our system.

Following our past versions of the model, we've made edits to reflect how we will store and manage all our data in order to have a functional and efficient program. We've rendered this new diagram that reflects our databases after revising our past design. We decided that our "Navigation Database" was too abstract and vague, so we elected to separate it into "DRIVER", "DETECTION\_DEVICE" and "ROAD\_PATH". These communicate with each other using an SQL approach in the relational database and store different information that is nicely organized. For example, the "VEHICLE" database follows the "ROAD\_PATH" database and is able to retrieve information from it without the queries overloading "ROAD\_PATH". Also, "VEHICLE" includes "DETECTION\_DEVICE", meaning it uses the information from the database to operate since our program wouldn't be able to navigate without it. This is a conscious decision, as we've decided that dividing the database into separate ones allows for more abstraction which is easier to understand and follow. Additionally, these changes are reflected in our UML diagram and SWA diagram. These additions and alterations to our previous designs get us one step closer to implementing a program that will operate seamlessly and fulfill all the benchmarks from our design specification.

**Data Management Strategy:**

The Autonomous Vehicle System relies on a single relational database (SQL) to manage all critical data. This centralized approach ensures data consistency and integrity across the tightly coupled entities, which is essential for safe operation.

The database is structured around three primary tables/conceptual entities:

Entry	Purpose	Key Data stored	Usage and Updated Frequency
ROAD_PATH	Stores foundational, geo-spatial, and regulatory data for navigation.	Route segments, GPS coordinates, speed limits, lane geometry, intersection rules, pre-defined maps.	Primarily Read operations by the Vehicle Navigation Logic; updated infrequently by the mapping service.
DETECTION_DEVICE	Stores high-frequency, real-time data from all vehicle sensors.	LiDAR readings, camera object detection (type and position), radar velocity measurements, timestamped event logs.	Primarily Write/Update operations by the sensor fusion component (high volume); Read operations for immediate obstacle avoidance.
DRIVER	Stores persistent data related to the driver, vehicle configuration, and historical performance.	User preferences, emergency contact details, vehicle configuration settings, historical trip logs, maintenance records, authentication tokens.	Balanced Read/Write operations; used during startup, trip planning, and shutdown.

**Data Flow Strategy:**

1. Guidance: The VEHICLE system initiates a query to the ROAD\_PATH table to retrieve the necessary path segments and regulatory information.

2. Perception: Simultaneously, the DETECTION\_DEVICE entity is continuously updated with sensor data. The Vehicle Perception Component executes highly localized queries against this real-time data to determine immediate threats and safe movement vectors.
3. Logging and State: All system actions, errors, and trip metadata are persisted to the DRIVER table's log fields, ensuring an immutable record for post-trip analysis or diagnostics.

This relational structure allows for clear foreign key relationships (e.g., the sensor data in DETECTION\_DEVICE references the specific vehicle/driver context), enforcing the required strong consistency crucial for safety-critical operations.

**Tradeoffs And Discussion For Design Choices:** The chosen data management strategy utilizes a single SQL database due to the system's core requirements for strong transactional consistency and handling of complex, tightly coupled relationships between entities including the "VEHICLE", "DRIVER", and various "DETECTION\_DEVICE" components. SQL is superior to NoSQL alternatives in this context because the integrity of safety-critical vehicle data is more crucial than massive, distributed scalability. The decision to use a single overarching relational database over multiple, distributed databases simplifies operations, guarantees data integrity across related tables within a single transaction, and avoids the complexity and potential synchronization issues associated with distributed data and microservices, which are unnecessary for this system's defined scope.