Alset IoT Hugs the Lanes

Syntastic Four

Nicholas Fontana, Vinci Li, Dylan McGrory, Michael Preziosi

Version 1.7

April 27th, 2025

1 Introduction

1.1 Team Role in Development

The Syntastic Four is responsible for developing the software architecture for the next generation of self-driving cars, Alset. Our primary focus is to design and implement crucial features such as autonomous navigation, real-time decision-making, and seamless IoT integration. This involves work on areas such as collision detection, path planning, and AI-driven control mechanisms to ensure a safe and pleasant driving experience. Our work is aligned with IoT-driven infrastructure to enhance vehicle-to-cloud and vehicle-to-vehicle communication.

1.2 Scope of the Project

For this initial release of the project, we are primarily concerned with the core functionality of the car. This is to ensure the successful operation of the vehicle, which is imperative to the safety of its passengers and those in its vicinity. Additionally, with the widespread usage of vehicular transport, vehicles with integrated IoT capabilities will become ever more popular, further demanding their successful construction at their core. Later versions of the project may allow for further enhancements to be added.

1.3 Features

An overview of the following features found in our software:

1. Collision detection
    a. Uses real-time data from our motion sensors to detect and prevent potential collisions
2. Cruise Control
    a. Adjusts vehicle speed dynamically for a smooth driving experience.
3. Camera detection system
    a. Utilizes computer vision to identify road signs, pedestrians, and other vehicles.
4. Weight detection in seats
    a. Ensures proper passenger safety by detecting occupancy and adjusting restraints accordingly.
5. Cloud integration
    a. Enables continuous updates, remote diagnostics, and data sharing for improved performance.

1.4 Availability and Reliability

It's critical that our software is reliable as it controls a moving vehicle. If an error were to occur in our system, it could be dangerous, so availability and reliability are of utmost importance. By maintaining availability and reliability, it decreases the chance of vehicle/traffic collisions and

accidents, as well as allows our system to stay up to date with new updates which may provide fixes for existing issues.

1.5 Reliance on IoT

In order to ensure our self-driving cars make the best decisions when driving, they need to have a reliance on IoT. This is important for the car to make real-time decisions, which can be critical in preventing accidents. The computation and automation that occur in the car's control systems deduce what the car should do in any given moment. Plus, this data can then be sent to the cloud so that other vehicles can look at this data and know what to do when they encounter a similar scenario.

1.6 Software Development Process

Our project follows the Spiral Model for software development. This approach is ideal for self-driving software because it allows incremental improvements while actively managing risks and incorporating continuous feedback. The Spiral Model consists of four major phases:

1. Communication & Requirement Gathering: Continuous communication between our engineers, testers, and our customers to refine safety-critical features.
2. Risk Analysis & Planning: Throughout each of the development cycles, our team assesses potential system failures with our system allowing us to proactively identify and adjust our approach.
3. Incremental Development & Prototyping: We build and test core functionalities in iterative cycles, starting with basic driver assistance before progressing to full autonomy.
4. Testing & Development: Every iteration of our software goes through extensive testing simulating real-world scenarios with feedback from the built-in cloud or vehicle logs.

1.7 Team & Qualifications

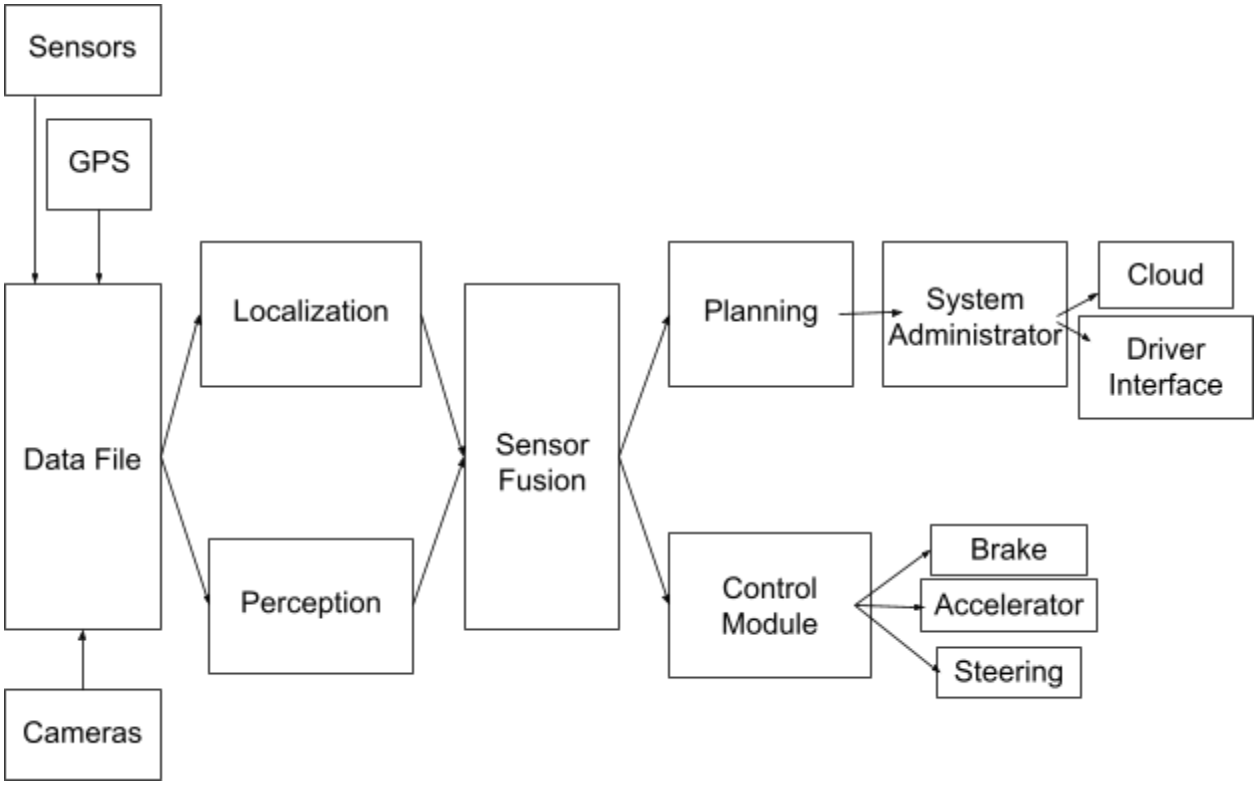
Our team is a group of incredibly talented software engineers well-versed in coding languages such as C++, Java, and Assembly. We display a balance of task-oriented and detail-oriented members, allowing for the project to be completed to its full potential. Some of us are more outgoing and pioneering, while some are more reserved and methodical, letting us work together in a desirable setting.

2 Functional Architecture

2.1 Overview of Functional Architecture

Our vehicle architecture relies heavily on sensors and cameras taking in data based on environmental surroundings. This data is loaded into a log file and then can be routed to multiple locations, primarily the cloud and the Planning Module. From the cloud, the data can be referenced by other cloud-connected vehicles, and from the Planning Modules, the data is used to determine how the car should react to a given scenario. This happens within the logic of the IoT Engine, which then communicates its decision to the Control Module. Finally, the Control Module makes the actuators within the car operate based on the decision made by the logic of the IoT Engine. As an example of how the process would work, if something obstructs the vehicle's path directly in front of the vehicle, the sensors will register this information in a log file and the IoT Engine will determine that the vehicle must stop based on this data, causing the Control Module to activate the brake accordingly in order to prevent the vehicle from crashing into the obstruction in front of it. Other operations, such as cameras recognizing traffic signals and road conditions, will be performed in a similar manner. All of these operations will happen in real-time; the time it takes for data recorded to result in the car doing some action is practically instantaneous, not recognizable to the users.

System Architecture Diagram:



2.2 System Modules and Their Functions

Perception module:
Uses LiDAR sensors to find points in 3D space after generating a 3D point cloud. Takes information from projected locations. Geometric feature projection is applied towards object detection, motion forecasting, and depth estimation, and is used to observe collisions and detect dense traffic. Object detection classifies cars, road signs, and pedestrians. Communicates data such as approaching pedestrians to a Planning Module.

Planning Module
Taking in input from the Perception module sends operations to the control module after choosing the car's options, acceleration, brake, turn, etc. Using route planning algorithms can optimize fuel efficiency and safety.

Control Module:
Converts decisions from users' physical actions, including the car's steering, brakes, wheel torque throttle, taking in data and commands from the Planning Module to influence the car's movements. Sends the feedback back to the Planning Module to monitor performance.

Driver Assistance Interface:
Prompts warnings, information, and override messages to drivers after receiving updates from Perception and Planning Modules. For example, an oncoming pedestrian or incoming wall may prompt a large beeping sound and flash warning.

Communication Module:
Communicates and handles data within cloud services and map updates, including traffic data or communication with other vehicles. Utilizes data transferring between cloud servers, providing updates in real-time. Updates GPS based on road conditions and traffic data updates.

System Admin Module:
Software updates, security, firewalls, and system logs to fix software. Communicates with the other modules to understand the Car's architecture's health and detect malfunctions. It will trigger system alerts in case of malfunction. Has firewall protection to prevent unauthorized access. Triggers system failures and activates failsafe mechanisms

2.3 Justification of Architecture with Real-World Examples

Our architecture is designed to support autonomous driving by using cloud connectivity and real-time processing. Driver assistance systems use features like collision detection, cruise control, and a camera detection system to help vehicles adjust to road conditions and avoid

accidents. IoT connections allow the system to receive and share important data for better decision-making. Self-driving capabilities depend on AI-powered control, vehicle-to-vehicle communication, and backup systems to help the car operate safely and handle different situations. System administration uses cloud integration for remote updates, system monitoring, and security. This ensures that the software is kept up to date and always running smoothly. These features guarantee that different parts of the system work together efficiently, making it easier to update, maintain, and expand the technology used in autonomous vehicles.

2.4 Role of Sensor Fusion in System Performance

To encapsulate why sensor fusion is so important for this project, we have to first define what it is. Sensor fusion is the process of integrating data from multiple sensors to create a more accurate and reliable representation of a vehicle's environment. Sensor fusion is a combination of two previous sensors found in vehicles LiDAR and radar. LiDAR excels at depth perception but lacks color and texture information. Radar is effective for detecting objects at long ranges but has lower spatial resolution. By combining data from these sources, sensor fusion enhances the vehicle's ability to make informed decisions, improving safety, perception, and navigation accuracy.

In the Alset IoT system, sensor fusion plays a crucial role in real-time decision-making. The perception module found in our sensors integrates LiDAR and camera data feature-level fusion, allowing for accurate object recognition and depth estimation. The Planning Module employs hybrid fusion, combining sensor inputs to improve overall risk assessment. By implementing sensor fusion into our overall architecture, Alset IoT can improve navigation accuracy, minimize collision risk, and enhance overall driving safety in dynamic environments.

3 Requirements

This section details the requirements that must be met by the vehicle. Each requirement relates to a specific function of the vehicle to ensure it can perform to its intended capacity. Furthermore, some overall qualities of the vehicle software are stated to clearly define the way the vehicle should behave when in use.

3.1 Functional Requirements

3.1.1 Cruise Control

Cruise Control maintains a set driving speed and safe following distance and automatically adjusts based on traffic conditions or driver input.

Pre-conditions:

- Vehicle speed is above 25 mph
- Driver has activated the Cruise Control function through the Driver Assistance Interface

Post-conditions:

- The vehicle maintains the selected speed or safe following distance
- Any manual override by the driver (brake/accelerator) stops Cruise Control

Requirements:

1. Driver presses the “Cruise Control” button on the Driver Assistance Interface
2. Sensor Fusion continuously monitors the leading vehicle’s speed and distance
3. Planning Module calculates whether acceleration or deceleration is needed to maintain a safe distance
4. Control Module adjusts the throttle or brake to keep the vehicle at the designated speed/distance
5. System Admin Module logs all Cruise Control Events

Exception:

- If a sudden deceleration is detected from the car ahead, the system automatically disengages Cruise Control and issues a driver alert to take control immediately

3.1.2 Collision Detection

Collision Detection prevents collisions by continuously monitoring sensor data and triggering braking or avoidance maneuvers when an obstruction is detected.

Pre-conditions:

- Vehicle is powered on and in motion
- Sensor Fusion is active and receiving data from the Perception Module

Post-conditions:

- Vehicle either slows down or comes to a complete stop if an obstacle is detected directly ahead
- Collision warnings are logged in the system

Requirements:

1. Perception module sends real-time distance and object data to Sensor Fusion
2. Sensor Fusion integrates and prioritizes the sensor inputs, then sends a collision alert to the Planning Module
3. Planning Module evaluates the alert and issues a braking or avoidance command to the Control Module
4. Control Module applies braking force or steering adjustments to avoid collision
5. Driving Assistance Interface displays a collision warning, both visually and audibly, when attempting to avoid a collision

Exception:

- If there is a sensor malfunction, the Planning Module logs an error and reduces vehicle speed automatically until the system is safe

3.1.3 Weight Detection in Seats

Weight Detection in Seats ensures passenger safety by detecting occupancy and adjusting restraints (airbag activation and seatbelt alerts) accordingly.

Pre-conditions:

- Vehicle ignition is on
- Seat sensors are operational

Post-conditions:

- Occupied seats trigger seatbelt reminder alerts and log a safety violation if the passenger is unbuckled
- Airbags are properly armed based on occupant weight thresholds

Requirements:

1. Weight Detection Sensor in each seat measures occupant weight upon vehicle startup
2. Sensor Fusion integrates seat-occupancy data with the rest of the system's sensors
3. Planning Module recognizes occupant status and adjusts safety systems
4. Driver Assistance Interface indicates which seats are occupied and prompts seatbelt use if any seat is unbuckled
5. System Admin Module records occupant-weight data for maintenance logs and possible recall checks

Exception:

- If a seat sensor malfunctions or fails to provide data, the system displays an error message and defaults to the highest safety setting for that seat

3.1.4 Camera Detection System

The Camera Detection System uses computer vision to identify road signs, pedestrians, and other vehicles in real time, sending classification data to the Planning Module.

Pre-conditions:

- Camera sensors are powered and calibrated
- Adequate visibility or night-vision capability is available for camera operation

Post-conditions:

- Road sign or obstacle information is stored in sensor logs
- Planning Module receives classification data

Requirements:

1. Cameras capture continuous video feeds of the vehicle's surroundings
2. Perception Module processes images for object detection
3. Sensor Fusion consolidates camera data with LiDAR/radar inputs for accuracy
4. Planning Module uses data to determine required vehicle response
5. Driver Assistance Interface displays alerts if immediate driver action is required

Exception:

- If camera lenses are obscured or the system cannot classify objects reliably, the Planning Module reverts to alternative sensors and issues a "Camera Fault" alert

3.1.5 Cloud Integration for Real-Time Updates

Cloud Integration for Real-Time Updates allows for continuous updates, remote diagnostics, and data sharing with other IoT-connected vehicles or central servers.

Pre-conditions:

- Vehicle has an active internet/cloud connection
- Communication Module is online and authenticated with cloud services

Post-conditions:

- Updated software modules are applied to the vehicle's system
- Vehicle logs and diagnostic data are successfully synced with the cloud

Requirements:

1. Communication Module initiates a secure handshake with the cloud server on system startup
2. System Admin Module checks for any pending software updates or alerts stored in the cloud
3. Planning Module temporarily pauses non-critical processes during an update to avoid data corruption
4. Driver Assistance Interface notifies the driver of ongoing updates or any recommended service actions

5. Cloud Server receives and stores updated sensor logs, allowing shared data access with other connected vehicles

Exception:

- If the connection drops mid-update, the system reverts to the previous stable software version. It then logs an error and attempts the update again when connectivity is restored

3.1.6 Technician Interface & Diagnostics

A technician can log into the System Admin Module to retrieve logs, perform diagnostics, and apply updates or patches securely.

Pre-conditions:

- Vehicle is stationary
- Vehicle is powered on
- A technician has valid credentials (username and password) to access the admin console

Post-conditions:

- Authorized technician can view system status, view logs, and update the software without disrupting vehicle safety
- All diagnostic actions are recorded for auditing purposes

Requirements:

1. A technician enters valid credentials via the System Admin Module's secure login screen
2. The System Admin Module verifies credentials against an internal or cloud-based authentication server
3. The technician can run diagnostics on any module and view the system health status
4. Diagnostic Logs are automatically stored in the vehicle's onboard memory and transmitted to the cloud for backup if the network is available
5. System Admin Module prompts for confirmation before any software update or patch is applied to live systems

Exception:

- If authentication fails, the system denies access and records the attempt in an unauthorized-access log

3.1.7 Adaptive Headlights Control

Adaptive Headlights Control automatically adjusts the vehicle's headlights based on driving conditions.

Pre-conditions:

- Vehicle is powered on and in motion
- Sensors and camera systems are functional

Post-conditions:

- Headlights adjust automatically to optimize visibility and minimize glare for other drivers

Requirements:

1. Camera Detection System identifies ambient lighting conditions, road curves, and oncoming traffic
2. Sensor Fusion combines environmental data and speed to determine optimal headlight brightness
3. The Planning Module decides whether to engage high beams or dim the lights
4. The Control Module adjusts the headlights accordingly in real time
5. The System Admin Module logs all automatic headlight adjustments for diagnostic purposes

Exception:

- If the sensors fail or visibility conditions are unclear, the system defaults to standard headlight settings and issues an alert

3.1.8 Blind Spot Monitoring

Blind Spot Monitoring detects objects that appear in ranges unable to be seen through the vehicle's side and rear view mirrors.

Pre-conditions:

- Vehicle is in motion, and powered on
- Side sensors and cameras are operational.

Post-conditions:

- Driver notified and alerted of oncoming vehicles in blind spot

Requirements:

1. Side sensors and cameras detect vehicles in adjacent lanes.
2. Sensor Fusion integrates data and sends it to the Planning Module.
3. Planning Moduleevaluatess necessity of alert
4. Driver Assistance Interface visual or auditory cues are activated to notify driver

Exception:

- Blind Spot detection deactivated if sensores failed, notifying driver

3.1.9 Intelligent Speed Adaptation

Intelligent Speed Adaptation regulates the speed of the vehicle based on the vehicle's surroundings and environmental conditions.

Pre-conditions:

- Vehicle in motion
- GPS and Camera Detection System is active and operational.

Post-conditions:

- Vehicle adjusts speed automatically according to road speed limit signs

Requirements:

1. Camera Detection System reads speed limit signs.
2. GPS cross-references road speed limits and incoming traffic or cars
3. Planning Module determines if speed adjustment is necessary
4. Control Module adjusts throttle or Cruise Control settings accordingly
5. Driver Assistance Interface notifies the driver of any speed adjustments

Exception:

- System prompts user to manually adjust speed if there is inconsistent speed limit data

3.1.10 Driver Drowsiness Detection

Driver Drowsiness Detection alerts the driver when they are in a drowsy state while behind the wheel.

Pre-conditions:

- Vehicle in motion
- Driver is directly facing and in front of the camera and biometric sensors are functional

Post-conditions:

- Alerts of drowsiness being detected are thrown on screen to the driver

Requirements:

1. Camera Detection System monitors driver eye movement, head position and grip on wheel
2. Sensor Fusion integrates biometric and behavioral data.
3. Planning Module assesses fatigue risk.
4. Driver Assistance Interface throws an alert if drowsiness is detected.

Exception:

- Obstructed camera system notifies the driver and disables drowsiness detection

3.2 Non-functional Requirements

3.2.1 Performance Requirements

3.2.1.1 System Response Time:

The Planning Module shall process all sensor inputs and issue control commands within 100 milliseconds to ensure real-time vehicle operation.

3.2.1.2 Data Processing Latency:

The Sensor Fusion Module integrates and processes sensor data (LiDAR) at a minimum rate of 60 frames per second to maintain an accurate model of the environment.

3.2.1.3 Cloud Communication Latency:

The communication module shall receive and transmit vehicle-to-cloud within 50 milliseconds to ensure timely data synchronization with a 100% success rate.

3.2.1.4 Actuator Execution Speed:

The Control Module shall execute braking, acceleration, and steering commands within 50 milliseconds of receiving an input from the Planning Module.

3.2.1.5 Real-Time Monitoring of Sensors

The system shall continuously monitor all active sensors and log performance anomalies in real time with 100% accuracy.

3.2.2 Reliability Requirements

3.2.2.1 System Uptime

The Autonomous Driving System shall maintain an uptime of 99.99% under normal operating conditions.

3.2.2.2 Fail-safe mechanism

If a critical sensor or control module malfunctions, the system shall automatically trigger an alert, reducing speed and alerting the driver within 1 second.

3.2.2.3 Sensor Redundancy

Each sensor shall have at least one redundant backup to prevent system failure in case of hardware malfunction.

3.2.2.4 Automatic System Recovery

If a non-critical module fails, the system shall attempt auto-recovery within 5 seconds or notify the driver if recovery is not possible.

3.2.3 Security Requirements

3.2.3.1 Firmware Integrity Check

The system shall verify the integrity of all software updates before installation to prevent tampering or unauthorized modifications.

3.2.3.2 Unauthorized Access Detection

The system will log and alert any unauthorized access attempt within 500 milliseconds of detection.

3.2.3.3 Multi-factor authentication

Technicians who wish to work on the software shall require multi-factor authentication (MFA) for all administrative actions.

3.2.3.4 Encrypted vehicle-to-cloud transmissions:

All vehicle-to-cloud data transmissions shall be encrypted with a 99.99% success rate in preventing unauthorized access attempts.

3.2.4 Usability Requirements

3.2.4.1 Audible Alerts for Critical Warnings

The system shall provide audible alerts at a minimum volume of 60 dB for critical warnings, such as collision detection, sensor failures, and emergency braking.

3.2.4.2 Driver Assistance Interface Visibility

The Driver Assistance Interface shall maintain a perfect contrast ratio for the user, allowing all text and visual indicators to be readable in various lighting conditions

3.2.4.3 Remote Monitoring via Cloud

The Communication Module shall allow authorized users to remotely access vehicle diagnostics, software status, and critical alerts via a secure cloud interface.

3.2.4.4 Touch-Free Driver Controls

The Driver Assistance Interface shall support 100% hands-free operation for emergency interventions such as braking, lane correction, and adaptive cruise control to reduce distractions

4 Requirement Modeling

This section models some of the requirements listed in the previous section, framing them in terms of practical examples of scenarios the driver/vehicle may encounter. The steps taken by the driver or vehicle, as well as the expected outcomes of these actions, are outlined for each instance. Additionally, diagrams are provided to illustrate the relationship between various modules within the vehicle software.

4.1 Use Case Scenarios

4.1.1 Use Case 1: Cruise Control is activated

Pre-Condition: Vehicle is on, its speed is above 25 mph, Driver has activated the Cruise Control function through the Driver Assistance Interface

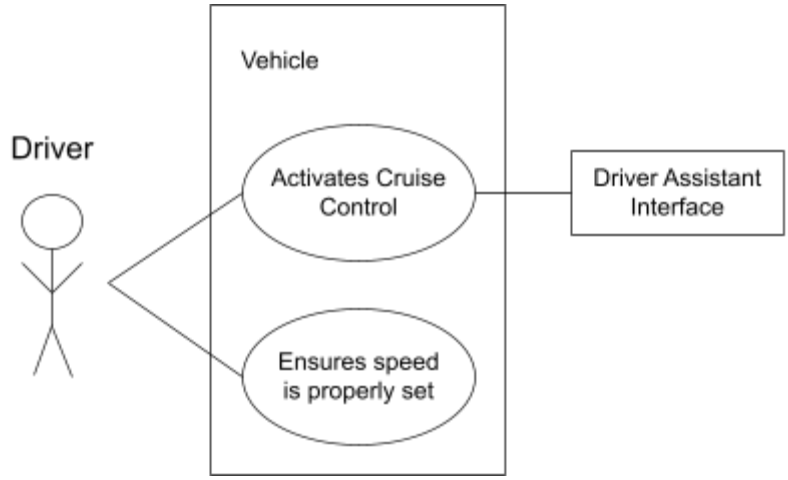
Post-Condition: Vehicle maintains the selected Speed or safe following Distance, any manual override by the Driver (brake/accelerator) stops Cruise Control

Trigger: Driver activates the Cruise Control system

Exception: If a sudden deceleration is detected from the car ahead, the system automatically disengages Cruise Control and issues a Driver alert to take control immediately

1. Driver presses the “Cruise Control” button on the Driver Assistance Interface
2. Sensor Fusion continuously monitors the leading vehicle’s Speed and Distance to set the vehicle’s Speed
3. Planning Module calculates whether acceleration or deceleration is needed to maintain a safe distance
4. Control Module adjusts the throttle or brake to keep the vehicle at the designated Speed/Distance
5. System Admin Module receives all data from Cruise Control Events and then stores it in logs

Use Case Diagram:



4.1.2 Use Case 2: Obstruction detected ahead

Pre-Condition: Vehicle is powered on and in motion, Sensor Fusion is active and receiving data from the Perception Module

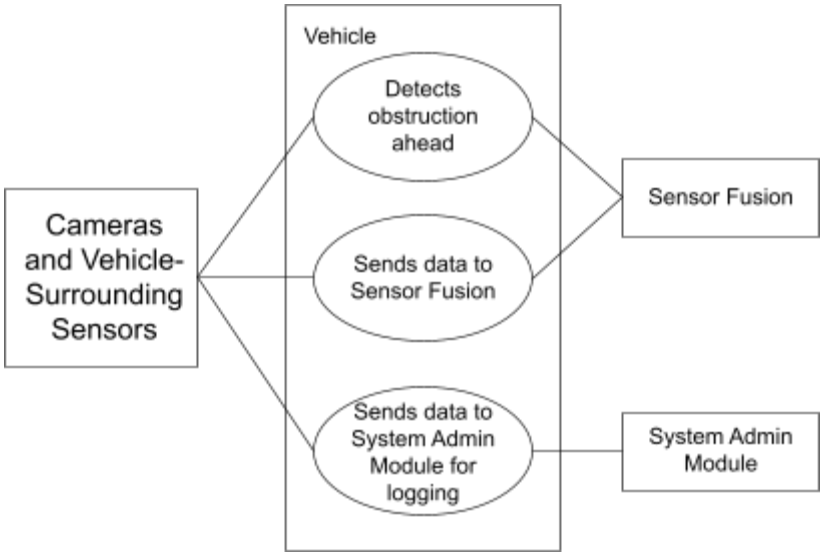
Post-Condition: Vehicle either slows down or comes to a complete stop if an obstacle is detected directly ahead, collision warnings are logged in the system

Trigger: Perception module detects some obstruction in the vehicle's path

Exception: If there is a sensor malfunction, the Planning Module logs an error and reduces vehicle Speed automatically until the system is safe

1. Sensors gather and send real-time Distance and object data to Sensor Fusion
2. Sensor Fusion integrates and prioritizes the Sensor inputs, then sends a collision alert to the Planning Module
3. Planning Module evaluates the alert and issues a braking or avoidance command to the Control Module. The Speed of the vehicle will be 5 mph for every 5 feet of Distance between the vehicle and obstruction (being within 5 feet of the obstruction will cause the vehicle to come to a complete stop)
4. Control Module applies braking force or steering adjustments to avoid collision
5. Driving Assistance Interface displays a Collision Warning, both visually and audibly, when attempting to avoid a collision
6. Data on the avoided collision is sent to the System Admin Module to be logged

Use Case Diagram:



4.1.3 Use Case 3: Significant weight detected in seat

Pre-Condition: Vehicle ignition is on, seat sensors are operational

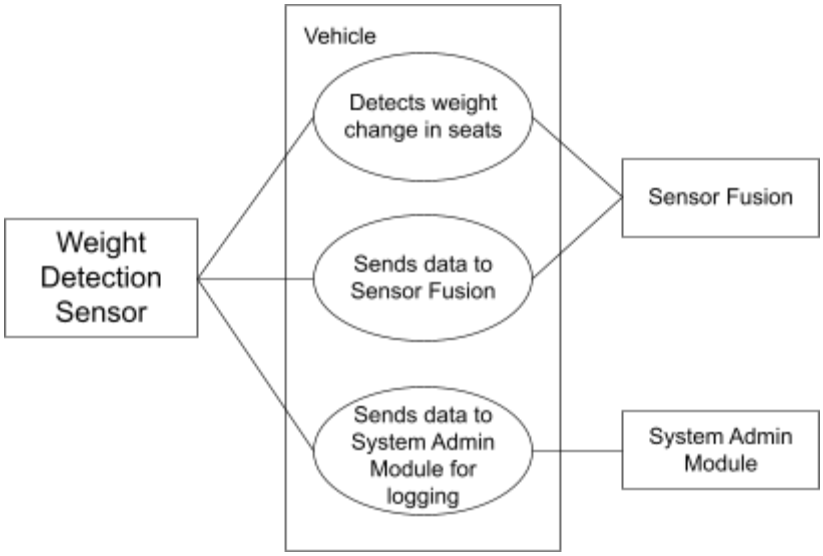
Post-Condition: Occupied seats trigger seatbelt reminder alerts and log a safety violation if the passenger is unbuckled, airbags are properly armed based on occupant weight thresholds

Trigger: Seat Sensors detect a significant change in the Weight occupying a Seat

Exception: If a Seat Sensor malfunctions or fails to provide data, the system displays an error message and defaults to the highest Safety Setting for that Seat.

1. Weight Detection Sensor in each Seat measures Occupant Weight upon Vehicle startup
2. Sensor Fusion integrates Seat Occupancy Data with the rest of the system's Sensors
3. Planning Module recognizes Occupant status and adjusts safety systems. A detected Weight below 50 lbs is recognized as a small child, a detected Weight between 50 lbs and 120 lbs is recognized as an adolescent, and a detected Weight above 120 lbs is recognized as an adult.
4. Driver Assistance Interface indicates which Seats are occupied and prompts Seatbelt use if any Seat is unbuckled
5. System Admin Module receives Occupant Weight data and records it in maintenance logs for possible recall checks

Use Case Diagram:



4.1.4 Use Case 4: Change in environment brightness detected

Pre-Condition: Vehicle is powered on and in motion, sensors and camera systems are functional

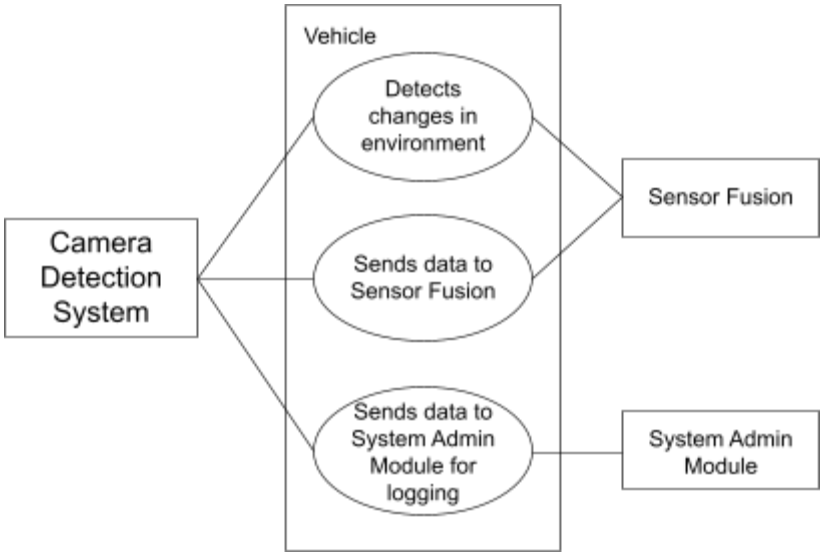
Post-Condition: Headlights adjust automatically to optimize Visibility and minimize Glare for other drivers

Trigger: Camera Detection System notices change in Environment Brightness

Exception: If the Sensors fail or Visibility Conditions are unclear, the system defaults to standard Headlight settings and issues an alert

1. Camera Detection System identifies Ambient Lighting Conditions, Road Curves, and Oncoming Traffic
2. Sensor Fusion combines Environmental Data and Speed to determine optimal Headlight Brightness. At 50% exterior brightness, the vehicle's Headlights are turned on to their lowest setting. At 30% brightness, the vehicle's Headlights are turned on to their medium setting. At 10% brightness, the vehicle's Headlights are turned on to their highest setting.
3. The Planning Module decides whether to engage High Beams or dim the Lights
4. The Adaptive Headlight Controller adjusts the Headlights accordingly in real time
5. The System Admin Module logs all automatic Headlight Adjustments for diagnostic purposes

Use Case Diagram:



4.1.5 Use Case 5: Vehicle software is updated

Pre-Condition: Vehicle has an active internet/cloud connection, Planning Module is online and authenticated with cloud services

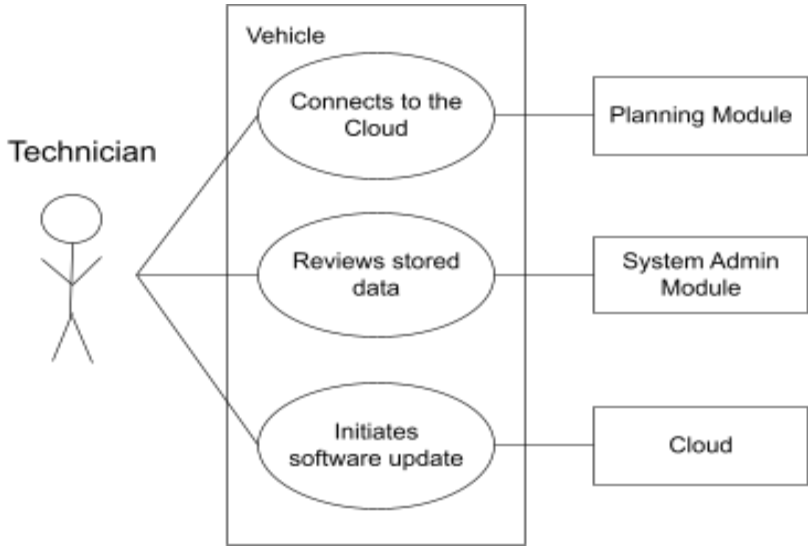
Post-Condition: Updated software modules are applied to the vehicle's system, vehicle logs and diagnostic data are successfully synced with the cloud

Trigger: Technician connects to the Cloud upon vehicle startup

Exception: If the connection drops mid-update, the system reverts to the previous stable software version and then logs an error and attempts the update again when connectivity is restored

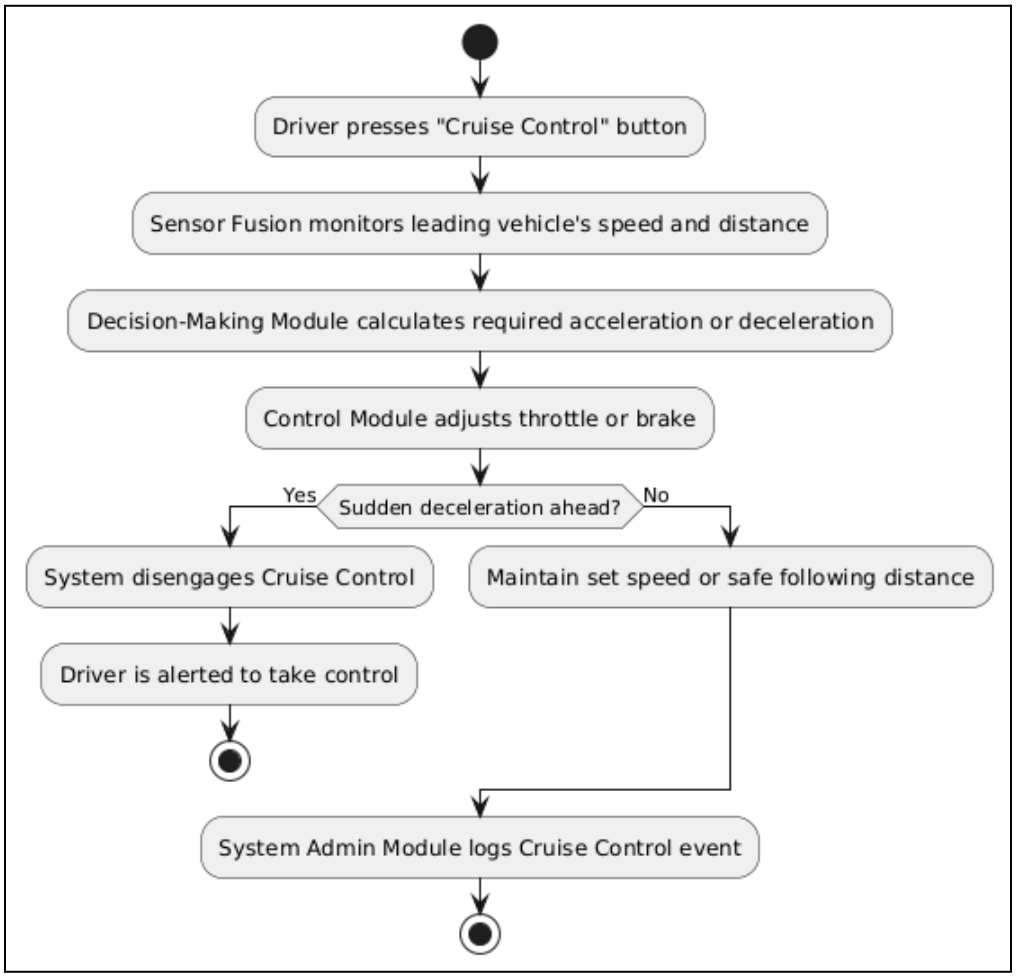
1. Technician initiates a secure handshake with the Cloud Server on system startup
2. Planning module executes handshake with Cloud Server
3. System Admin Module checks for any pending Software Updates or Alerts stored in the Cloud
4. Planning Module temporarily pauses non-critical processes during an update to avoid data corruption
5. Driver Assistance Interface notifies the Driver of ongoing updates or any recommended service actions
6. Cloud Server receives and stores updated Sensor Logs, allowing shared data access with other connected Vehicles

Use Case Diagram:

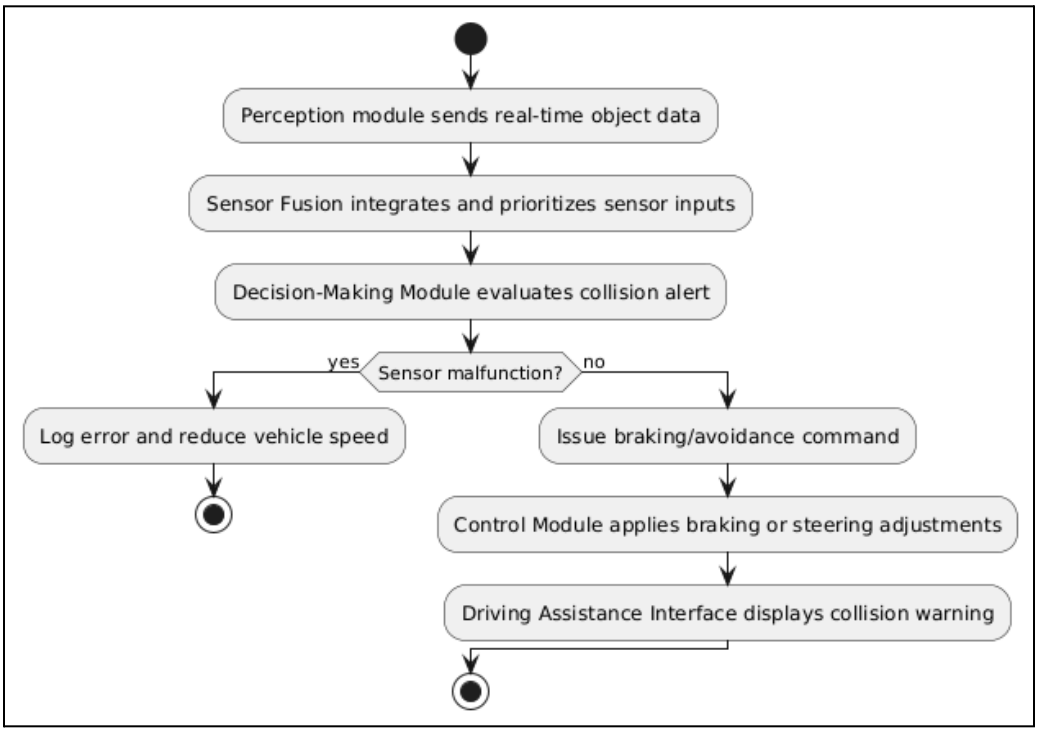


4.2 Activity Diagrams

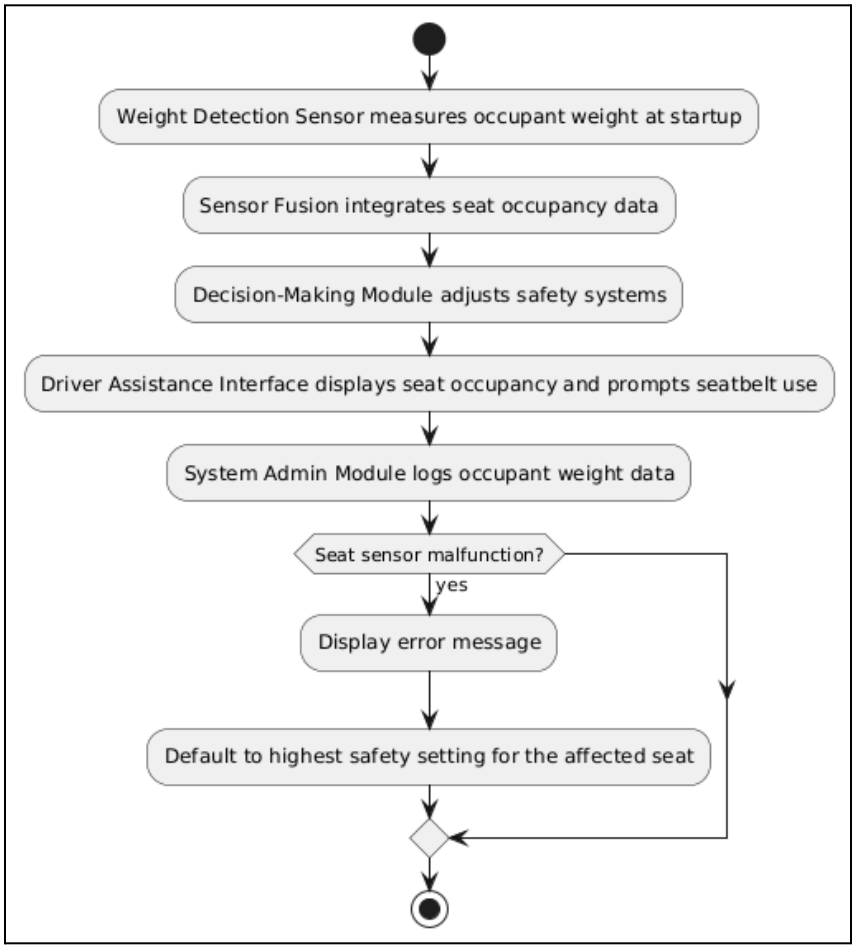
4.2.1 Cruise Control is activated



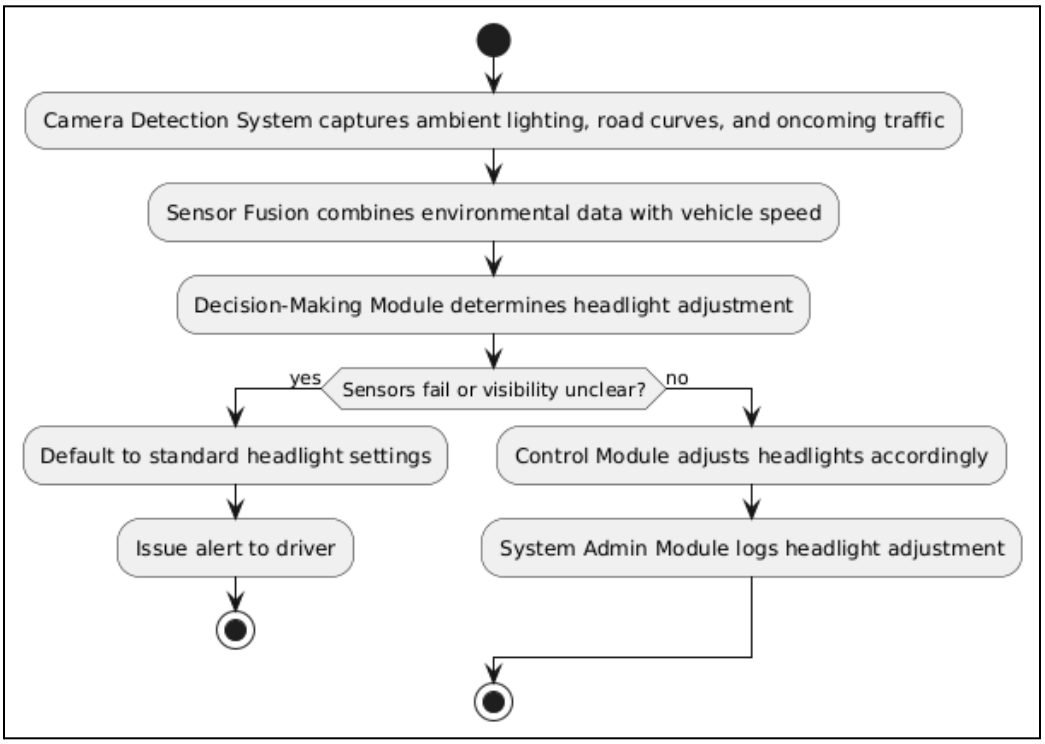
4.2.2 Obstruction detected ahead



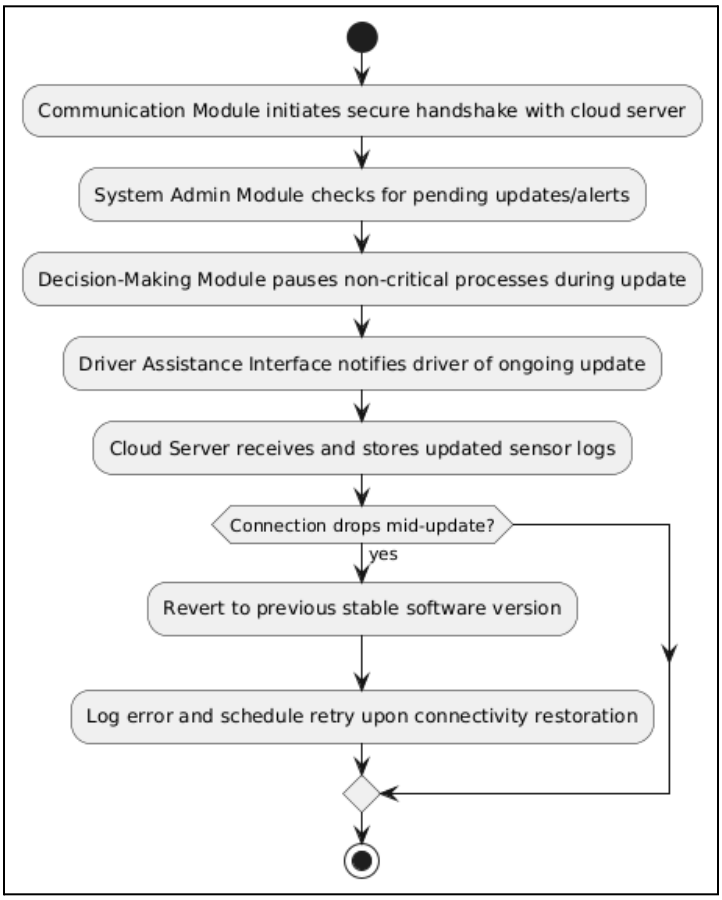
4.2.3 Significant weight detected in seat



4.2.4 Change in environment brightness detected

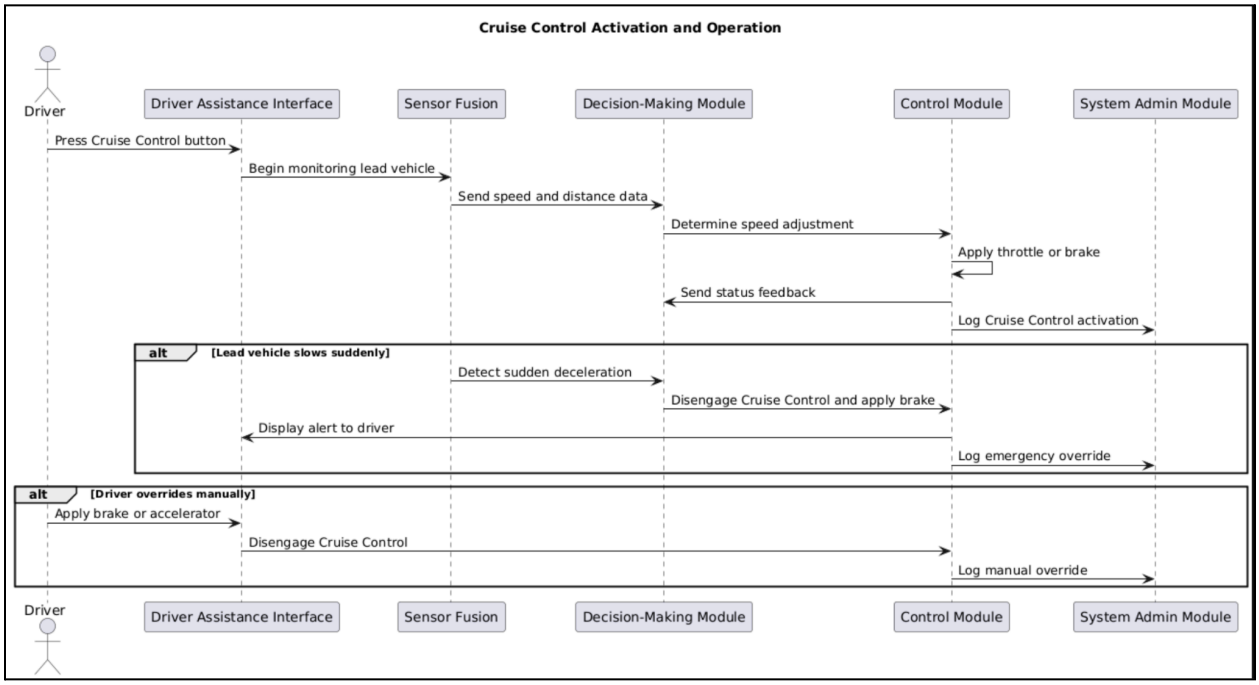


4.2.5 Vehicle software is updated

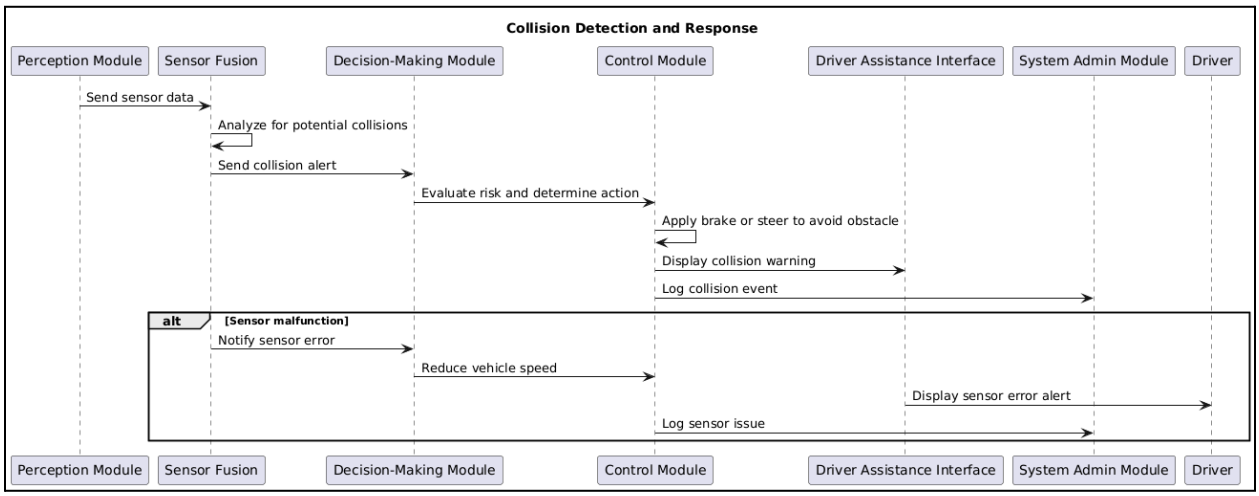


4.3 Sequence Diagrams

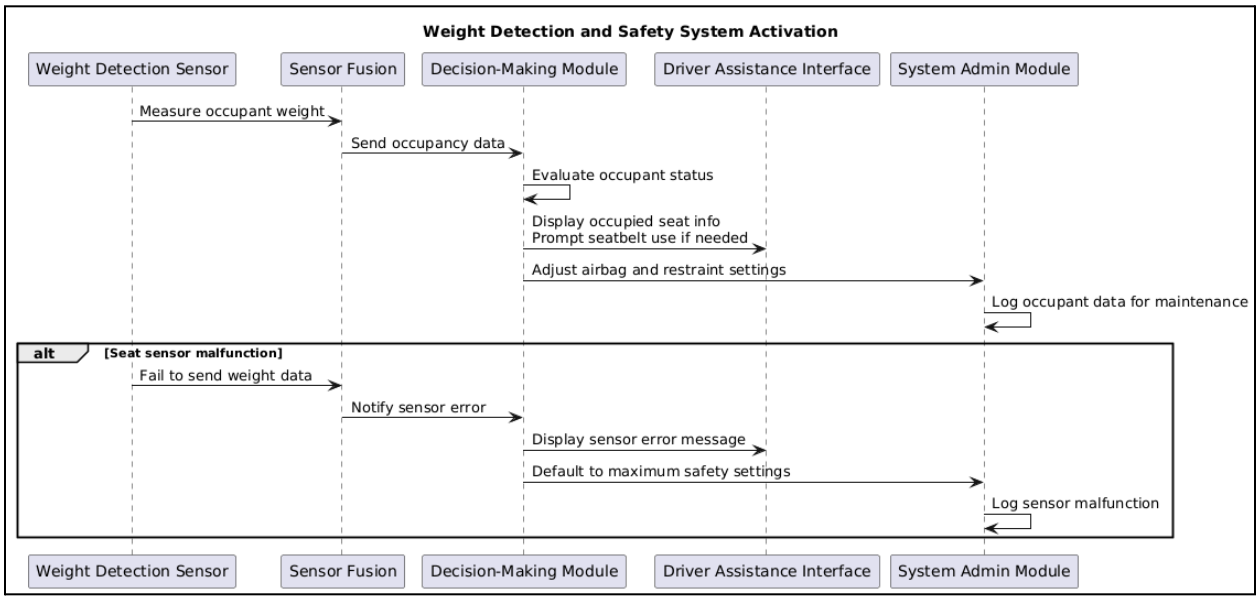
4.3.1 Cruise Control is activated



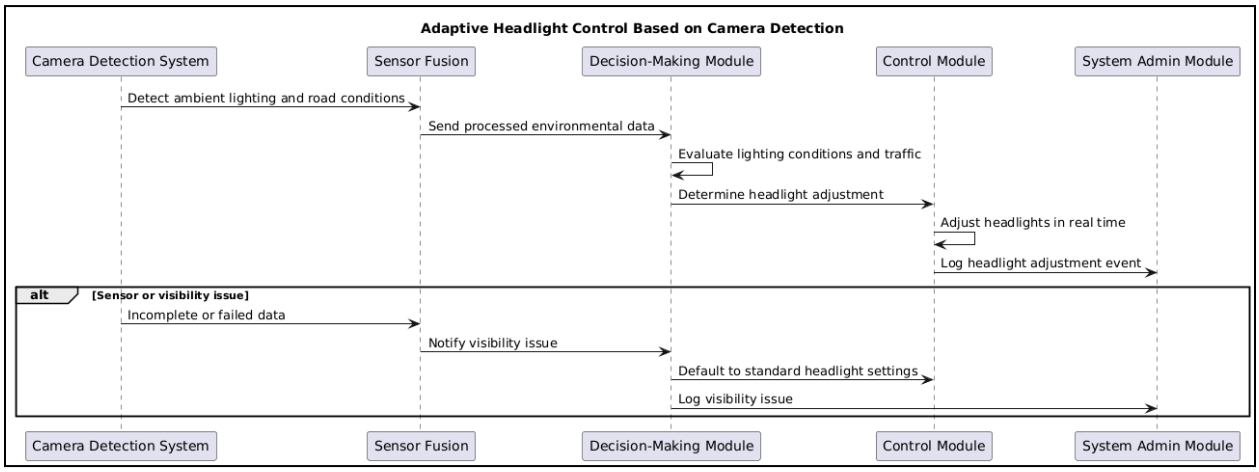
4.3.2 Obstruction detected ahead



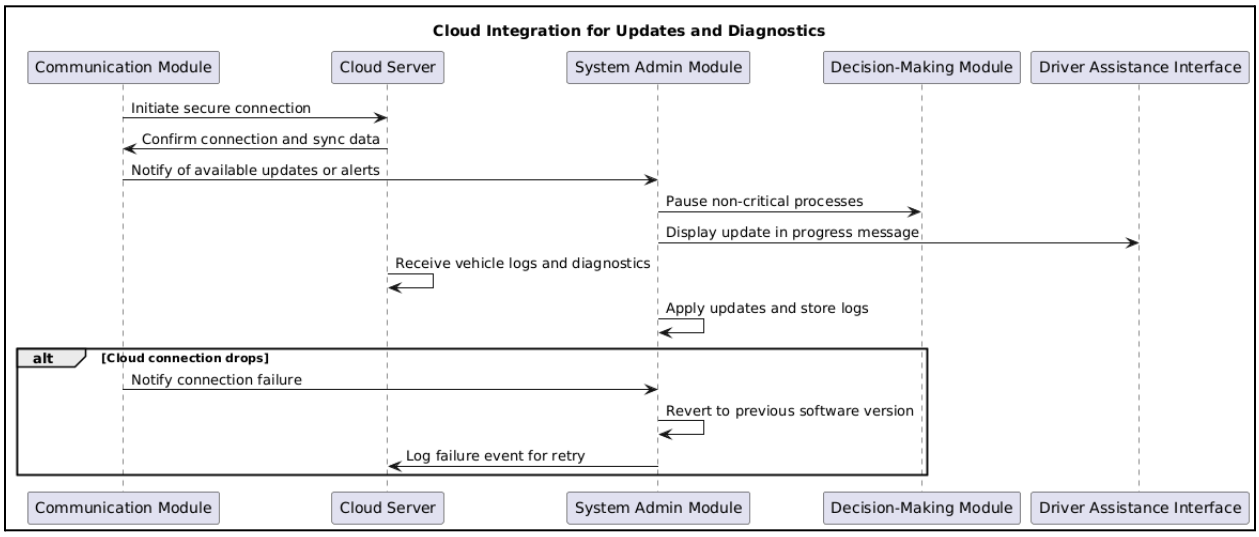
4.3.3 Significant weight detected in seat



4.3.4 Change in environment brightness detected



4.3.5 Vehicle software is updated



4.4 Classes

Below are the key classes in a UML Diagram of the IoT system. These are based on the system's above use cases.

Class: Sensor
Attributes: • sensorID • sensorType • location • status • system_admin
Methods: • collectData() • enable() • disable() • calibrate()

Class: SensorFusion
Attributes:

• fusedInput • outputData • priorityQueue
Methods: • integrate_sensor_data(data) • generateAlert() • transmitToControlModule()

Class: PlanningModule
Attributes: • imageData • objectDetections • control_module • driver_interface • system_admin
Methods: • analyzeEnvironment() • classifyObjects() • forwardToFusion()

Class: ControlModule
Attributes: • currentSpeed • steeringAngle • brakingForce
Methods: • apply_brake() • adjust_steering(angle) • set_throttle(value) • adjust_headlights(level)

Class: CruiseControlManager

Attributes:

- active
- control_module
- sensor_fusion
- planning_module
- system_admin
- driver_interface
- min_speed

Methods:

- activate(current_speed)
- update(vehicle_speed, lead_vehicle_distance, lead_vehicle_speed)
- disengage(reason)

Class: CollisionDetectionSystem

Attributes:

- perception_module
- sensor_fusion
- planning_module
- control_module
- driver_interface
- system_admin

Methods:

- monitor()
- handle_sensor_failure()

Class: WeightDetectionSystem

Attributes:

- seat_sensors
- sensor_fusion
- planning_module

• driver_interface • system_admin
Methods: • evaluate_seats()

Class: CameraDetectionSystem
Attributes: • camera_sensor • sensor_fusion • planning_module • driver_interface • system_admin
Methods: • detect_environment()

Class: CloudIntegrationManager
Attributes: • communication_module • system_admin • driver_interface
Methods: • perform_update() • sync_logs(logs)

Class: TechnicianInterface
Attributes: • system_admin
Methods:

- login(username, password)
- run_diagnostics()
- apply_patch(update_package)
- view_logs()

Class: AdaptiveHeadlightController

Attributes:
- camera_sensor
- control_module
- sensor_fusion
- planning_module
- driver_interface
- system_admin

Methods:
- adjust_lights()

Class: BlindSpotMonitor

Attributes:
- side_sensors
- sensor_fusion
- planning_module
- driver_interface

Methods:
- check_blind_spots()

Class: IntelligentSpeedAdaptation

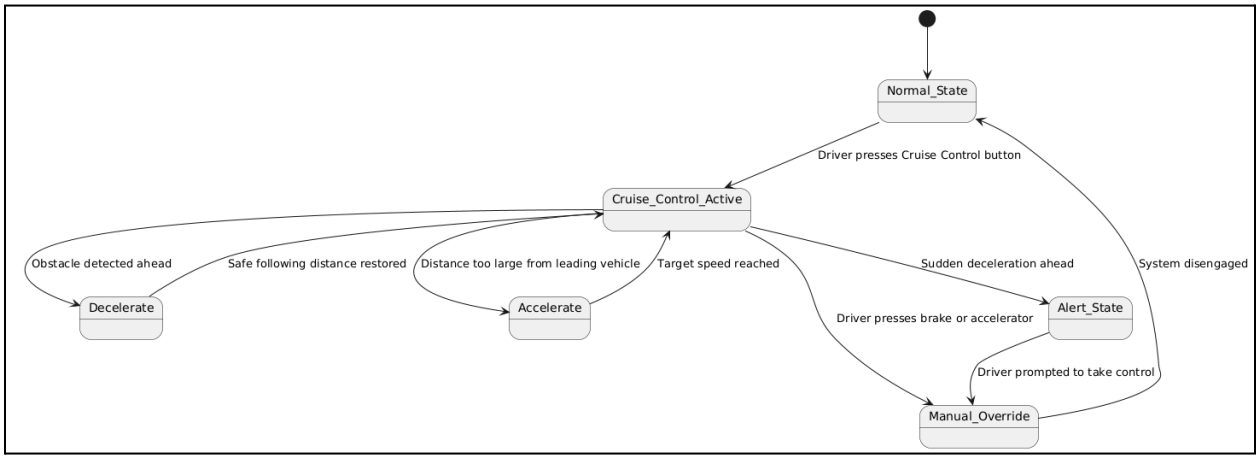
Attributes:
- camera_sensor

• gps_module • planning_module • control_module • driver_interface
Methods: • regulate_speed()

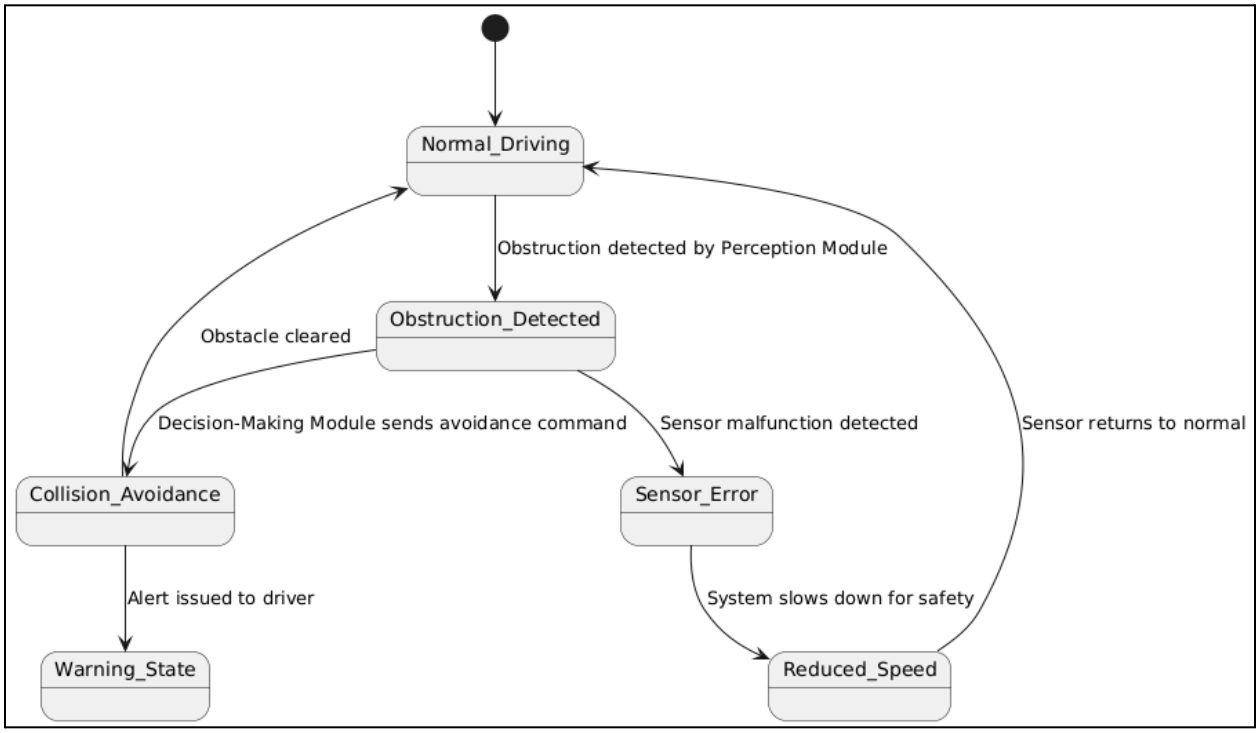
Class: DriverDrowsinessDetection
Attributes: • camera_sensor • biometric_sensor • sensor_fusion • planning_module • driver_interface
Methods: • monitor_driver()

4.5 State Diagrams

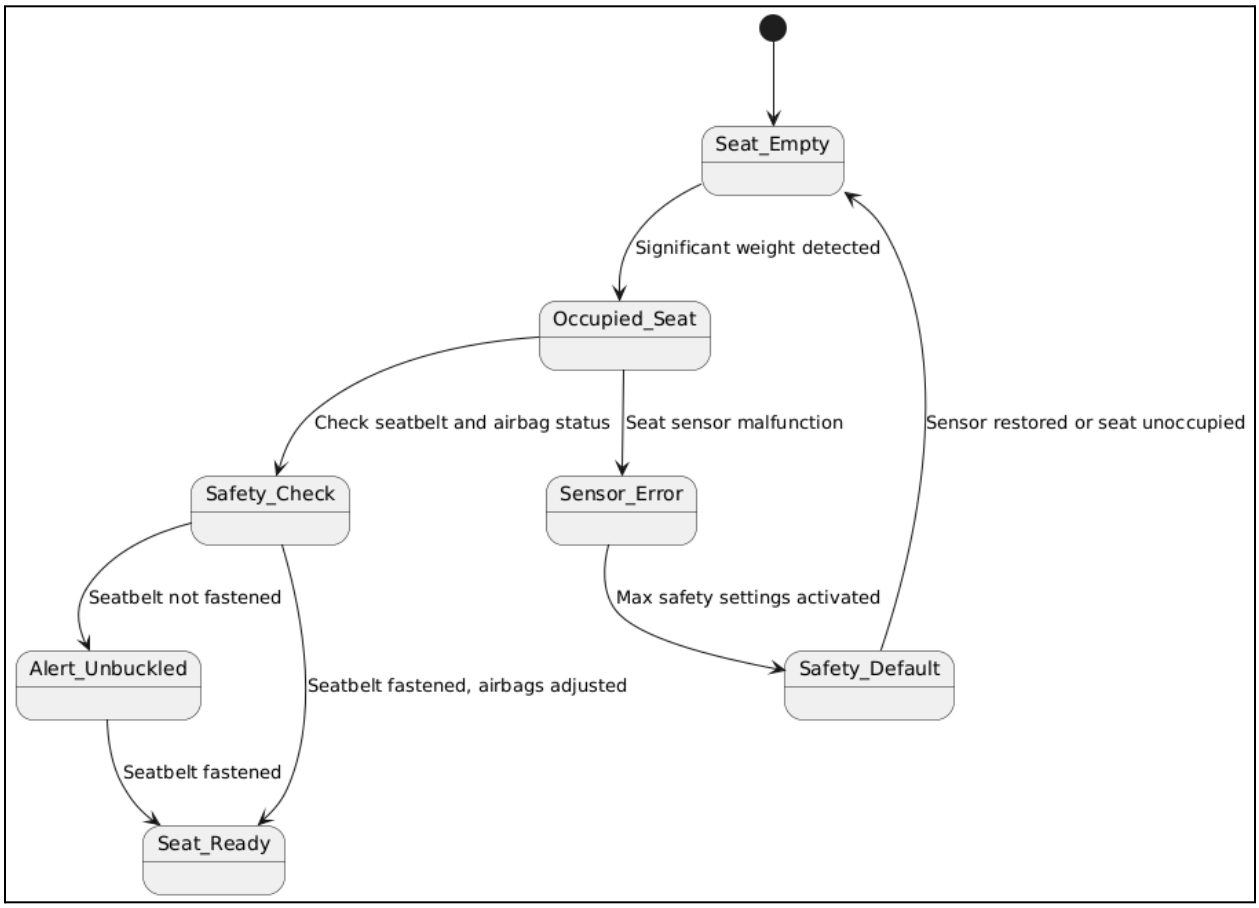
4.5.1 Cruise Control is activated



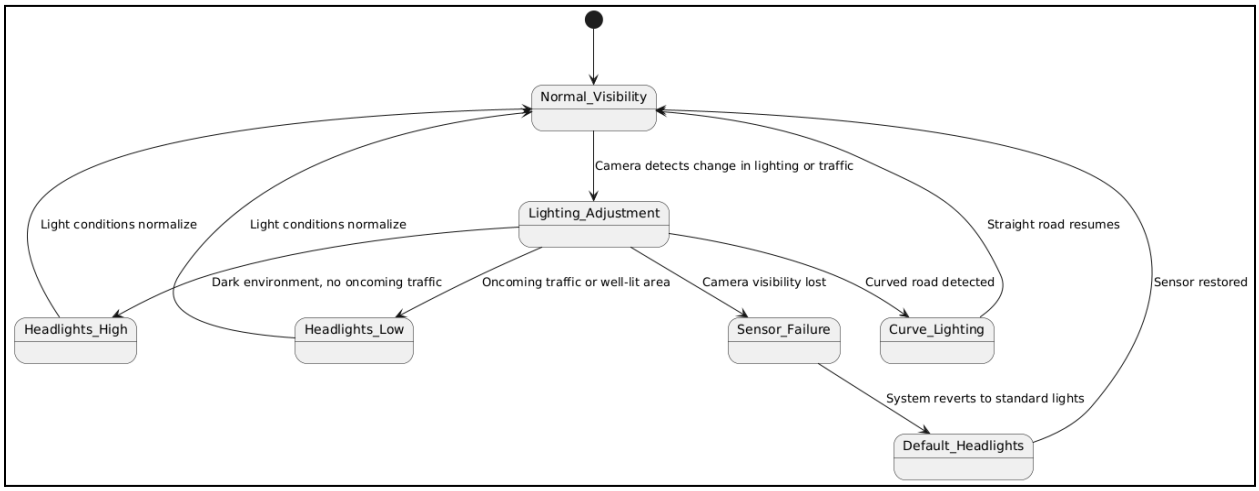
4.5.2 Obstruction detected ahead



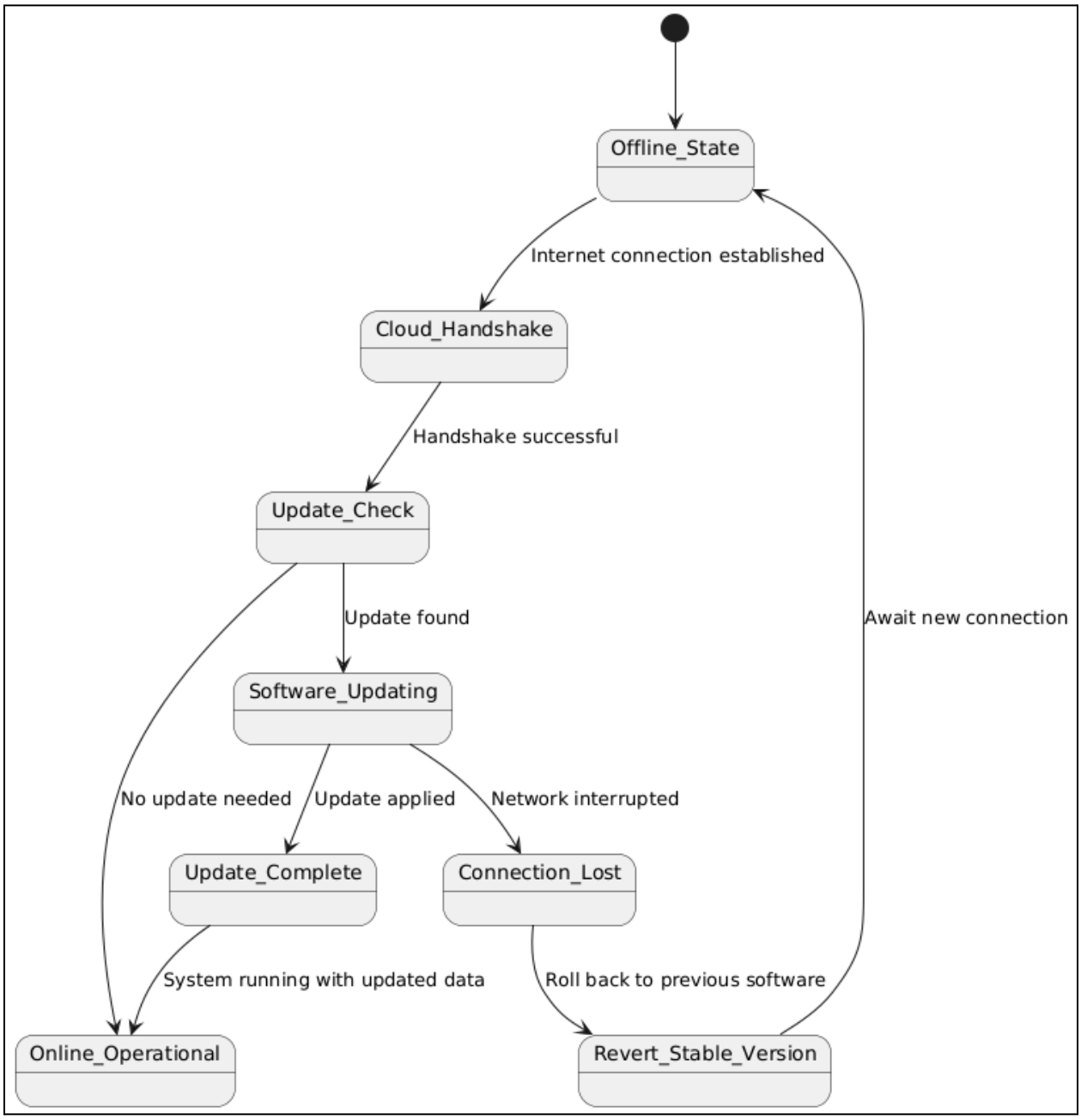
4.5.3 Significant weight detected in seat



4.5.4 Change in environment brightness detected



4.5.5 Vehicle software is updated



5 Design

This section outlines the architectural and technical design choices made for the Alset IoT Hugs the Lanes system. It begins by analyzing various software architecture models and evaluating their suitability for an IoT-based autonomous vehicle. The goal is to identify an architecture that meets the system's performance, safety, and modularity requirements. The section then presents the chosen architecture and its components, followed by interface design for both drivers and technicians, and finally, a detailed component-level breakdown of core system modules such as Sensor Fusion, the Planning Module, and the Control Module. Together, these design elements ensure a scalable, maintainable, and reliable system capable of handling real-time decision-making in autonomous driving scenarios.

5.1 Software Architecture

5.1.1 Relationship between Software Structural Elements

The elements of the IoT vehicle's software architecture comprise of a sequential model. Information is collected from the vehicle's surroundings through sensors and other recording devices and is then sent to Sensor Fusion to be interpreted and then Planning and Control Module to determine what to do and then execute the decision made. Furthermore, data can be stored as well as retrieved from the Cloud based on certain environments the vehicle encounters.

5.1.2 Data Centered Architecture

Data centered architecture stores collected information in a data store that can be accessed by multiple client softwares. The data can be stored for an indeterminate amount of time. This architecture model resembles how the IoT vehicle accesses information from the Cloud, however most information retrieved in the IoT vehicle architecture is recent, relying on relevant environmental conditions for accurate data. Also, the number of client softwares that can be supported by a data centered model is wasted on the IoT vehicle architecture, for there aren't enough clients attempting to access this data to make it viable.

5.1.3 Data Flow Architecture

Data flow architecture passes information through a series of pipes and filters to deliver it to its appropriate location. The variance of information accounted for in this architecture model is similar to the types of information collected by the various sensors and cameras used within the software of an IoT vehicle. However, nearly all information collected through these devices ends up travelling the same path, ending up being processed by Sensor Fusion and the Planning Module and then action being taken by the Control Module. Therefore, the variability of information that could be handled by a data flow architecture goes to waste on the IoT software.

5.1.4 Call Return Architecture

Call return architecture involves a main program being initiated and then branching off into one of several controller subprograms, each of which has its own application subprograms. This is applicable to the IoT vehicle architecture because the subprograms that can be run can be based on the type of information being collected, making these programs easily codeable. The disadvantage of doing this is that the programs being run will all follow a similar structure (collect data from sensors/cameras, send to Sensor Fusion, the Planning Module determines what to do, and the Control Module executes the Planning Module's decision). The multitude of different programs that can be crafted does not get put to use for the architecture of the IoT vehicle as much as the call return architecture is intended to.

5.1.5 Object-Oriented Architecture

Object-oriented architecture relies on every actor being represented by a class that has a specific set of attributes and methods that can be replicated in certain scenarios. This is applicable to the IoT vehicle's architecture because many of its components have repeated functions for the different data that is collected by the vehicle. The downside to utilizing this architecture for the IoT vehicle is that there will be a significant number of different classes that will need to be created in order to represent every actor that has some purpose within the IoT architecture, from the Driver and Technician to the modules within the vehicle's software.

5.1.6 Layered Architecture

The Layered Architecture organizes a software system into hierarchical layers, each responsible for a specific set of functionalities. This model aligns well with the IoT Hugs the Lanes (HtL) system, which naturally separates into layers such as sensor input, sensor fusion, decision-making, control, and user interfaces. The feasibility of this model is high, as it reflects the actual data and command flow within the vehicle—information flows from hardware sensors up through processing layers before reaching the control system, and feedback moves back down or out to user-facing modules. The primary advantage of using this architecture is its modularity; individual layers can be developed, tested, and updated independently, which improves maintainability and scalability. However, one downside is that strict layering can introduce latency, especially when lower layers must communicate indirectly through intermediate layers. Despite this, the benefits of clear structure, separation of concerns, and improved system testing make the Layered Architecture a strong candidate for the core structure of HtL's software system.

5.1.7 Model View Controller Architecture

The Model View Controller (MVC) architecture separates a software system into three core components: the Model (which handles data and logic), the View (which displays the user interface), and the Controller (which interprets inputs and updates the Model and View

accordingly). In the context of IoT Hugs the Lanes (HtL), this architecture maps well onto components such as Sensor Fusion and the Planning Module (Model), the Driver Assistance Interface and System Admin Console (View), and the Control Module (Controller). This separation of concerns promotes clean code organization and simplifies the maintenance and testing of interface-driven modules. However, MVC can introduce additional complexity when real-time data needs to be rapidly reflected across system layers, which could result in delays if not carefully optimized. While MVC is useful for managing user interfaces, it is only a partial fit for HtL's real-time embedded system, as it lacks the control robustness required by safety-critical vehicle functions.

5.1.8 Finite State Machine Architecture

Finite State Machine (FSM) architecture defines system behavior through a set of states, transitions, and events, making it well-suited for modules that rely on precise modes of operation. Within HtL, FSM can effectively manage features such as Cruise Control (with states like OFF, ACTIVE, and OVERRIDDEN), Collision Detection (monitoring, alert, brake), and Driver Drowsiness Detection (awake, warning, drowsy). FSM's clear state transitions and reactive design enhance system transparency, making it easier to simulate, debug, and validate each module's behavior under various conditions. However, as systems grow in complexity, FSMs can become harder to manage due to the rapid increase in potential states and transitions. Despite this scalability concern, FSM is a strong architectural match for HtL's critical vehicle safety and decision modules that require deterministic behavior and real-time responsiveness.

5.1.9 Final Software Architecture Design

The final software architecture for Alset IoT Hugs the Lanes adopts a hybrid approach, combining the strengths of several architectural models to best suit the project's safety, real-time, and modularity needs. The core of the system is organized using a Layered Architecture, ensuring a clear progression from sensor inputs through decision logic to physical actuation. Finite State Machine design governs behavior within modules that rely on explicit control states, such as collision avoidance, speed adaptation, and headlight control. Additionally, Object-Oriented Architecture is employed to encapsulate the system's various actors and modules—such as sensors, interfaces, and control units—into reusable and maintainable classes. This hybrid structure balances real-time responsiveness with code clarity and scalability, supporting both current functionality and future enhancements. Overall, this design best fulfills the functional and non-functional requirements of an IoT-driven autonomous vehicle system.

5.2 Interface Design

5.2.1 Driver Interface

Interface Name: DriverAssistanceInterface in
Class Reference: DriverAssistanceInterface
Primary Users: Vehicle Driver
Hardware Elements: Central touchscreen display, HUD (Heads-Up Display), output speakers, haptic feedback systems (steering wheel/vibration alerts)
Communication: Receives inputs from Perception Module, Sensor Fusion, and Planning Module

Interface Components and Functions:

Display Element	**Data Type**	**Description**
displayMode	Enum (Day, Night, Emergency)	Adapts interface visibility based on lighting and urgency
warningMessage	String	Displays system alerts, e.g., "Pedestrian Ahead", "Cruise Control Disengaged"
visualAlerts	Array (Image)	Icons such as seat belt warnings, camera fault indicators, lane-departure icons
renderAlert()	Method	Displays immediate visual/auditory warnings
promptUserAction()	Method	Triggers prompts like “Take Control Now”
showSystemStatus()	Method	Displays system health, speed, sensor status, and camera inputs

Example Features on Display:

- Adaptive Headlight Status
- Occupant detection indicators per seat
- “Cruise Control” toggle button
- Drowsiness detection alerts with fatigue levels
- Camera view overlays with object detection boxes

5.2.2 Technician Interface

Interface Name: SystemAdminConsole
Class Reference: SystemAdminConsole
Primary Users: Certified Vehicle Technicians

Hardware Elements: External diagnostic terminal (tablet/laptop), secure wired/wireless access port
Communication: Reads data from all software modules and cloud; issues write commands to software update handlers

Interface Components and Functions:

Display Element	**Data Type**	**Description**
softwareVersion	String	Displays current version of vehicle firmware
authStatus	Boolean	Indicates whether technician is authenticated (via MFA)
diagnosticLogs	Array (Text LogEntry)	System logs including collision events, software crashes, sensor faults
runDiagnostics()	Method	Runs real-time checks on all hardware/software components
logSystemData()	Method	Records all interaction logs and diagnostics
authenticateUser()	Method	Verifies credentials against secure internal/cloud auth service
applySoftwareUpdate()	Method	Applies firmware patches or upgrades, verifies signatures before update

Technician Workflow:

1.) Connect device → Enter credentials → authenticateUser()
2.) Select target subsystem → runDiagnostics() → view output
3.) If errors are found, trigger update or system reboot → applySoftwareUpdate()
4.) All logs automatically pushed to the cloud for recordkeeping

5.2.3 Driver Interface

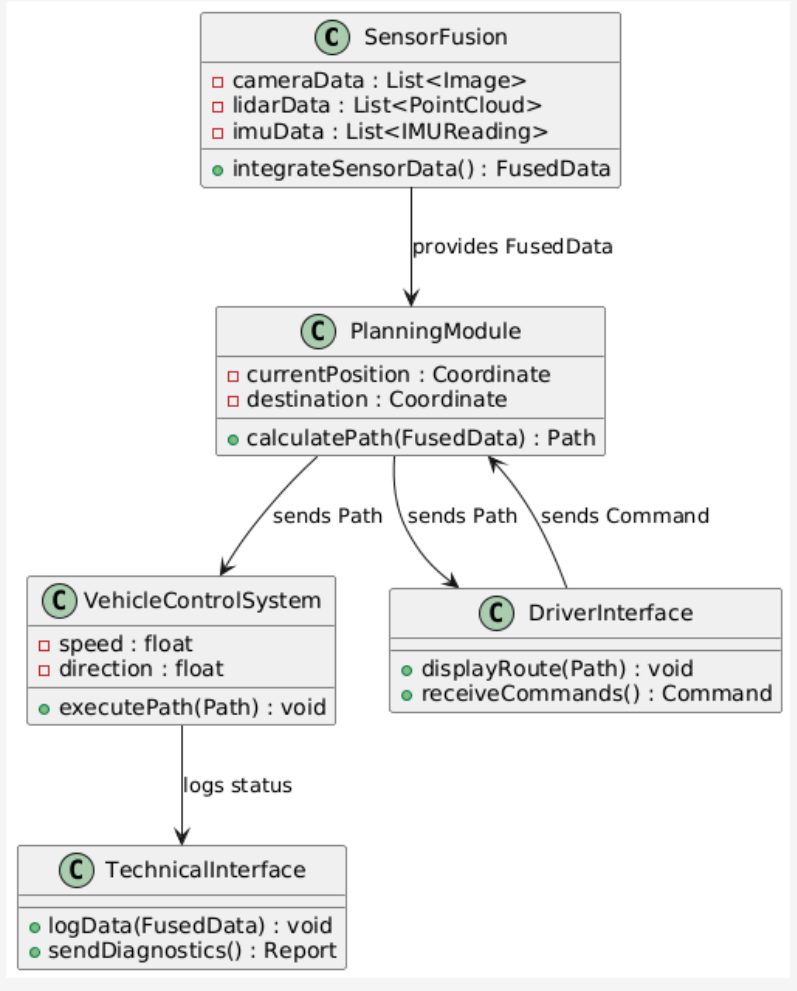
Component	**Input Source**	**Output Destination**	**Notes**
DriverAssistanceInterface	PlanningModule,	Driver (Visual and Audio)	Provides actionable

	PerceptionModule		driving feedback
SystemAdminModule	Cloud Server, Internal Modules	Technician Console	Allows secure updates and full system audits
SensorFusion	Hardware Sensors (Camera, LiDAR, Radar, Seat Sensors)	PlanningModule	Behind-the-scenes integration
PlanningModule	SensorFusion	Driver UI, ControlModule	Key logic engine
ControlModule	PlanningModule	Car Actuators	Executes real-world actions

5.3 Component-Level Design

5.3.1 Diagram:

A diagram illustrates the structure and interaction between the core components of the system:



5.3.2 Driver Display/Commands

Functionality:
Manages how information is displayed to the driver and how the driver inputs commands to the system.

Display Output:

- Presents real-time navigation routes, upcoming turns, lane positions, and system alerts
- Communicates vehicle state such as speed, mode (manual vs. autonomous), and system errors directly to the driver.

Input Handling:

- Receives touch or voice inputs from the driver to perform functions such as destination changes, toggling assisted driving features, or requesting diagnostics.

- Sends parsed commands to the Planning Module to adjust vehicle behavior accordingly.

This component directly supports features like Adaptive Assisted Driving (AAD), Emergency Braking, and Lane Switching, allowing the driver to stay informed and intervene if necessary.

5.3.3 Sensor Fusion, Planning, and Control Module

Functionality:
These three components work together to interpret the environment, make decisions, and physically control the vehicle.

Sensor Fusion:

- Gathers and combines data from the camera, LiDAR, and IMU sensors.
- Applies calibration and synchronization to align data across all sources.
- Uses fusion techniques to identify road lanes, objects, traffic signs, and motion paths.
- Outputs a unified Fused object for use by downstream components.

Planning:

- Uses the FusedData and current GPS position to calculate an optimal driving path.

- Employs algorithms to find the most efficient route to take.

- Continuously adapts to dynamic inputs (e.g., detected hazards or new commands from the Driver Interface).

- Sends the finalized path to both the Control Module and Driver Display for execution and visualization.

Control Module:

- Converts the planned path into actionable commands for throttle, braking, and steering.

- Uses closed-loop feedback (e.g., PID controllers) to ensure smooth tracking of the planned path.

- Monitors performance in real time and logs status updates for system management.

- Ensures vehicle safety by executing features like Electronic Stability Control (ESC) and Automatic Emergency Braking, based on fused sensor inputs.

This integrated pipeline fulfills critical functionalities such as Head-on Collision Prevention, Lane Detection, Acceleration Control, and overall autonomy

5.3.4 System Admin / Technical Interface

Functionality:
Provides tools for technicians and administrators to manage, monitor, and maintain the system.

Diagnostics:

- Accesses real-time and historical logs from the Control Module and Sensor Fusion Module.

- Provides visualization and export of system health data, error reports, and sensor activity.

- Supports troubleshooting and performance validation in accordance with testing protocols.

Configuration Management:

- Allows secure access for system administrators to calibrate or tune modules (e.g., adjusting braking sensitivity or display brightness).

- Supports parameter changes through a controlled interface, ensuring safe and logged modifications.

Update Management:

- Manages secure delivery and installation of firmware and software updates.
- Verifies version compatibility and ensures fail-safe rollback if updates are unsuccessful.

This interface aligns with non-functional requirements such as Software Reliability, Security, and Maintainability and is essential for field servicing and continuous system improvement.

6 Code

The following section contains the code for the IoT software. The code includes classes for each of the actors in the system, as well as methods and functions to perform the vehicle's core operations. The code is written in Python in order to ensure the code is versatile, scalable, and easily readable by others.

```
class Sensor:
    def __init__(self, sensorID, sensorType, location, status,
system_admin):
        self.sensorID = sensorID
        self.sensorType = sensorType
        self.location = location
        self.status = status
        self.system_admin = system_admin

    def collectData(self):
        if self.status != "active":
            self.system_admin.log_system_data(f"Sensor
{self.sensorID} is inactive. Cannot collect data.")
            return None
        return {"sensorID": self.sensorID, "data": "sampleData"}

    def enable(self):
        self.status = "active"

    def disable(self):
        self.status = "inactive"

    def calibrate(self):
        if self.status != "active":
            self.system_admin.log_system_data(f"Sensor
{self.sensorID} is inactive. Calibration failed.")
        else:
            print(f"Calibrating {self.sensorType} at
{self.location}.")
```

The Sensor class handles the hardware devices that collect information from the car's surroundings. It keeps track of the sensor's ID, type, location, and status. It can collect data, turn

sensors on or off, and perform calibrations. If a sensor is not working properly, it logs a message for the system admin to help with troubleshooting.

```
class SensorFusion:
    def __init__(self):
        self.fusedInput = []
        self.outputData = {}
        self.priorityQueue = []

    def integrate_sensor_data(self, data):
        self.fusedInput.append(data)
        self.outputData.update(data)
        if "priority" in data:
            self.priorityQueue.append(data)
        return self.outputData

    def generateAlert(self):
        if self.priorityQueue:
            return self.priorityQueue.pop(0)
        return None

    def transmitToControlModule(self):
        return self.outputData
```

The SensorFusion class combines data from different sensors into one clear set of information. It gathers sensor inputs, picks out important events, and sends the combined data to other parts of the system. It can also create alerts when something important is detected. This class helps make the car's decisions more accurate and reliable.

```
class PlanningModule:
    def __init__(self, control_module, driver_interface, system_admin):
        self.imageData = None
        self.objectDetections = []
        self.control_module = control_module
        self.driver_interface = driver_interface
        self.system_admin = system_admin

    def analyzeEnvironment(self, sensor_data):
        if "obstacle_distance" in sensor_data:
            distance = sensor_data["obstacle_distance"]
            if distance <= 5:
                self.control_module.apply_brake(100)
```

```
                self.driver_interface.render_alert("Immediate stop:
Obstacle ahead!")
            else:
                safe_speed = max(5, distance)
                self.control_module.set_throttle(safe_speed)

        if "road_sign" in sensor_data:
            sign = sensor_data["road_sign"]
            if sign.startswith("Speed Limit"):
                speed = int(sign.split()[-1]) # "Speed Limit 20" ->
["Speed", "Limit", "20"] -> 20
                self.control_module.set_throttle(speed)


    def classifyObjects(self):
        # Simulated detection
        if self.imageData:
            self.objectDetections = ["Car", "Stop Sign"]
            self.system_admin.log_system_data(f"Objects classified:
{self.objectDetections}")
            return self.objectDetections
        else:
            self.system_admin.log_system_data(f"No image data to classify
objects")
            return []

    def forwardToFusion(self, sensor_fusion, data):
        return sensor_fusion.integrate_sensor_data(data)
```

The PlanningModule class looks at the combined sensor data and decides what the car should do. It can detect objects, react to road signs, and plan how the car should move. It sends instructions to the Control Module and gives updates to the Driver Interface when needed. It is a key part of making sure the car drives safely and correctly.

```
class ControlModule:
    def __init__(self):
        self.currentSpeed = 0
        self.steeringAngle = 0
        self.brakingForce = 0
        self.lightLevel = "Off"

    def apply_brake(self, force = 50):
        self.brakingForce = force
```

```
        print(f"Applying brake with force {force}%")

    def adjust_steering(self, angle):
        self.steeringAngle = angle
        print(f"Steering adjusted to {angle} degrees")

    def set_throttle(self, speed):
        self.currentSpeed = speed
        print(f"Adjusting speed to {speed} mph")

    def adjust_headlights(self, level):
        self.lightLevel = level
        print(f"Headlights adjusted to {level} mode.")
```

The ControlModule class carries out the actions the car needs to take, based on decisions from the Planning Module. It controls the car's speed, steering, braking, and headlights. It makes sure the car follows the planned path smoothly and safely by adjusting things like throttle and brakes in real time.

```
class CruiseControlManager:
    def __init__(self, control_module, sensor_fusion, planning_module,
system_admin, driver_interface):
        self.active = False
        self.control_module = control_module
        self.sensor_fusion = sensor_fusion
        self.planning_module = planning_module
        self.system_admin = system_admin
        self.driver_interface = driver_interface
        self.min_speed = 25

    def activate(self, current_speed):
        if current_speed < self.min_speed:
            self.driver_interface.render_alert("Speed too low for Cruise
Control.")
            return
        self.active = True
        print("Cruise Control activated.")
        self.system_admin.log_system_data("Cruise Control activated.")

    def update(self, vehicle_speed, lead_vehicle_distance,
lead_vehicle_speed):
        if not self.active:
            return
```

```
        safe_distance = vehicle_speed * 0.5  # basic safe following
distance rule

        if lead_vehicle_distance < safe_distance:
            if lead_vehicle_speed < vehicle_speed:
                print("Decelerating to maintain safe distance.")
                self.control_module.apply_brake(10)
                self.system_admin.log_system_data("Cruise Control
decelerated due to close vehicle.")
        elif lead_vehicle_distance > safe_distance:
            print("Accelerating to reach target cruise speed.")
            self.control_module.set_throttle(vehicle_speed + 1)
            self.system_admin.log_system_data("Cruise Control
accelerated.")
        else:
            print("Maintaining current speed.")

    def disengage(self, reason):
        self.active = False
        self.driver_interface.render_alert(f"Cruise Control disengaged:
{reason}")
        self.system_admin.log_system_data(f"Cruise Control disengaged:
{reason}")
```

The Python code defines a system for autonomous cruise control management in a self-driving vehicle. The CruiseControlManager class continuously monitors vehicle speed and proximity to the lead vehicle, using sensor fusion and a decision module to dynamically adjust throttle or braking. It also handles activation thresholds, safe following distances, and disengagement upon driver override or environmental changes.

```
class CollisionDetectionSystem:
    def __init__(self, perception_module, sensor_fusion, planning_module,
control_module, driver_interface, system_admin):
        self.perception_module = perception_module
        self.sensor_fusion = sensor_fusion
        self.planning_module = planning_module
        self.control_module = control_module
        self.driver_interface = driver_interface
        self.system_admin = system_admin

    def monitor(self):
        perception_data = self.perception_module.get_environment_data()
```

```
        fused_data =
self.sensor_fusion.integrate_sensor_data(perception_data)

        if 'obstacle_distance' in fused_data:
            distance = fused_data['obstacle_distance']

            if distance <= 5:
                self.control_module.apply_brake(100)
                self.driver_interface.render_alert("Immediate stop!
Obstruction ahead.")
                self.system_admin.log_system_data("Emergency braking:
Obstruction at close range.")
            else:
                speed = max(5, distance)  # Maintain 5 mph per 5 ft rule
                self.control_module.set_throttle(speed)
                self.system_admin.log_system_data(f"Adjusting speed to
{speed} mph for obstruction.")

    def handle_sensor_failure(self):
        self.control_module.set_throttle(10)
        self.driver_interface.render_alert("Sensor Failure: Speed
limited.")
        self.system_admin.log_system_data("Sensor malfunction - reduced
speed mode activated.")
```

This code implements a collision avoidance system using the CollisionDetectionSystem class. It collects real-time environmental data from the perception module and fuses it to detect obstacles. Based on proximity, it triggers braking or adjusts speed accordingly through the control module. It also includes a fallback mode to reduce speed in the event of sensor failure.

```
class WeightDetectionSystem:
    def __init__(self, seat_sensors, sensor_fusion, planning_module,
driver_interface, system_admin):
        self.seat_sensors = seat_sensors
        self.sensor_fusion = sensor_fusion
        self.planning_module = planning_module
        self.driver_interface = driver_interface
        self.system_admin = system_admin

    def evaluate_seats(self):
        for seat_sensor in self.seat_sensors:
            try:
                weight_data = seat_sensor.measure_weight()
```

```
                self.sensor_fusion.integrate_sensor_data(weight_data)

                weight = weight_data['weight']
                if weight == 0:
                    continue

                if weight < 50:
                    category = 'Child'
                elif weight < 120:
                    category = 'Adolescent'
                else:
                    category = 'Adult'

                self.driver_interface.render_alert(f"{category} detected in
seat {seat_sensor.seatID}. Put on your seatbelt.")
                self.system_admin.log_system_data(f"Occupant in seat
{seat_sensor.seatID}: {weight} lbs - {category}")
            except Exception:
                self.driver_interface.render_alert(f"Sensor error in seat
{seat_sensor.seatID}. Defaulting to high safety.")
                self.system_admin.log_system_data(f"Seat
{seat_sensor.seatID} failed. Applied high safety measures.")
```

A passenger safety system is implemented using the WeightDetectionSystem class. This class reads seat sensor data to detect occupant weight and categorizes them as children, adolescents, or adults. Based on this, it provides alerts for seatbelt usage and logs safety-critical information. If a sensor fails, the system defaults to maximum restraint protocols for safety.

```
class CameraDetectionSystem:
    def __init__(self, camera_sensor, sensor_fusion, planning_module,
driver_interface, system_admin):
        self.camera_sensor = camera_sensor
        self.sensor_fusion = sensor_fusion
        self.planning_module = planning_module
        self.driver_interface = driver_interface
        self.system_admin = system_admin

    def detect_environment(self):
        try:
            video_frame = self.camera_sensor.capture_image()
            objects = self.camera_sensor.detect_road_sign()
            lighting = self.camera_sensor.analyze_lighting()
            fused = self.sensor_fusion.integrate_sensor_data({
```

```
                "video": video_frame,
                "objects": objects,
                "lighting": lighting
            })
            self.planning_module.analyze_environment(fused)
            self.system_admin.log_system_data("Camera data analyzed and
fused successfully.")
        except Exception:
            self.driver_interface.render_alert("Camera Fault: Switching to
alternative sensors.")
            self.system_admin.log_system_data("Camera fault detected.
Reverted to fallback sensors.")
```

The vehicle's camera-based perception is handled by the CameraDetectionSystem class. It processes image feeds, road signs, and lighting conditions using onboard cameras, and integrates these with sensor fusion for higher accuracy. The data is then analyzed for navigation decisions, and fallback protocols are triggered in case of vision system failure.

```
class CloudIntegrationManager:
    def __init__(self, communication_module, system_admin,
driver_interface):
        self.communication_module = communication_module
        self.system_admin = system_admin
        self.driver_interface = driver_interface

    def perform_update(self):
        if not self.communication_module.connectionStatus:
            self.communication_module.establish_connection()

        if self.communication_module.connectionStatus:
            try:
                update_package =
self.communication_module.download_updates()
                self.system_admin.apply_software_update(update_package)
                self.driver_interface.render_alert("System updated
successfully.")
                self.system_admin.log_system_data("Update applied
successfully.")
            except Exception as e:
                self.driver_interface.render_alert("Update failed. Rolling
back.")
                self.system_admin.log_system_data(f"Update failed: {e}.
Rolled back to previous version.")
```

```
        else:
            self.driver_interface.render_alert("No internet connection.
Update deferred.")
            self.system_admin.log_system_data("Update deferred due to no
connectivity.")

    def sync_logs(self, logs):
        if self.communication_module.connectionStatus:
            self.communication_module.sync_with_cloud(logs)
            self.system_admin.log_system_data("Logs synced to cloud.")
        else:
            self.system_admin.log_system_data("Log sync skipped: no cloud
connection.")
```

The CloudIntegrationManager class handles real-time updates and vehicle-to-cloud synchronization. It ensures secure connectivity, downloads and installs software updates, and maintains data logs by syncing diagnostic information with cloud servers. The class includes rollback mechanisms in the event of failed updates or lost connectivity.

```
import random

class TechnicianInterface:
    def __init__(self, system_admin):

        self.system_admin = system_admin
        self.mfa_code = None

    def login(self, username, password):
        self.system_admin.authenticate_user(username, password)
        if not self.system_admin.authStatus:
            print("Unauthorized access attempt logged.")
            self.system_admin.log_system_data("Unauthorized access
attempt.")
            return False

        # Simulate MFA
        self.mfa_code = str(self.random.randint(100000, 999999))
        print(f"[MFA Code Sent]: {self.mfa_code}")

        inputted_code = input("Input MFA code: ")

        if inputted_code == self.mfa_code:
            print("Multi-factor Authentication successful.")
```

```
            return True
        else:
            print("Multi-factor Authentication failed.")
            self.system_admin.authStatus = False
            self.system_admin.log_system_data("MFA failed - Unauthorized
access attempt.")
            return False

    def run_diagnostics(self):
        if not self.system_admin.authStatus:
            print("Access denied. Run diagnostics failed.")
            return "Access denied"
        result = self.system_admin.run_diagnostics()
        print(f"Diagnostics Result: {result}")
        return result

    def apply_patch(self, update_package):
        if not self.system_admin.authStatus:
            print("Access denied. Patch installation failed.")
            return "Access denied"
        self.system_admin.apply_software_update(update_package)
        print("Patch applied successfully.")
        return "Patch applied"

    def view_logs(self):
        if not self.system_admin.authStatus:
            return "Access denied"
        return self.system_admin.diagnosticLogs
```

The TechnicianInterface class allows authorized maintenance personnel to log into the system, run diagnostics, and apply patches. It includes multi-factor authentication, access logging, and secure update mechanisms. If authentication fails, access is denied and incidents are logged for audit purposes.

```
class AdaptiveHeadlightController:
    def __init__(self, camera_sensor, control_module, sensor_fusion,
planning_module, driver_interface, system_admin):
        self.camera_sensor = camera_sensor
        self.control_module = control_module
        self.sensor_fusion = sensor_fusion
        self.planning_module = planning_module
        self.driver_interface = driver_interface
        self.system_admin = system_admin
```

```
    def adjust_lights(self):
        try:
            brightness = self.camera_sensor.analyze_lighting()
            speed = self.control_module.currentSpeed

            if brightness <= 10:
                level = 'High'
            elif brightness <= 30:
                level = 'Medium'
            elif brightness <= 50:
                level = 'Low'
            else:
                level = 'Off'

            self.control_module.adjust_headlights(level)
            self.system_admin.log_system_data(f"Adaptive headlights set to
{level} brightness.")
        except Exception:
            self.control_module.adjust_headlights("Standard")
            self.driver_interface.render_alert("Headlight sensor error.
Using standard mode.")
            self.system_admin.log_system_data("Failed adaptive headlight
adjustment. Standard mode activated.")
```

Headlight automation is handled by the AdaptiveHeadlightController class, which adjusts headlight intensity based on ambient lighting and vehicle speed. It integrates with sensor fusion and decision modules to determine the optimal brightness and activates fallback lighting if sensors fail.

```
class BlindSpotMonitor:
    def __init__(self, side_sensors, sensor_fusion, planning_module,
driver_interface):
        self.side_sensors = side_sensors
        self.sensor_fusion = sensor_fusion
        self.planning_module = planning_module
        self.driver_interface = driver_interface

    def check_blind_spots(self):
        try:
            blind_spot_data = []
            for sensor in self.side_sensors:
                data = sensor.collect_data()
```

```
                blind_spot_data.append(data)

            fused = self.sensor_fusion.integrate_sensor_data({"blind_spot":
blind_spot_data})

            if any('vehicle_detected' in d for d in blind_spot_data):
                self.driver_interface.render_alert("Vehicle in blind
spot!")
        except Exception:
            self.driver_interface.render_alert("Blind spot sensors
unavailable.")
```

Blind spot monitoring is managed by the BlindSpotMonitor class. It collects data from side-mounted sensors and uses sensor fusion to detect nearby vehicles in the blind spot. If a vehicle is detected, the system alerts the driver to prevent unsafe lane changes.

```
class IntelligentSpeedAdaptation:
    def __init__(self, camera_sensor, gps_module, planning_module,
control_module, driver_interface):
        self.camera_sensor = camera_sensor
        self.gps_module = gps_module
        self.planning_module = planning_module
        self.control_module = control_module
        self.driver_interface = driver_interface

    def regulate_speed(self):
        try:
            sign_speed = self.camera_sensor.detect_road_sign()
            road_speed = self.gps_module.get_speed_limit()

            if sign_speed != road_speed:
                self.driver_interface.render_alert("Speed limit data
mismatch, adjust manually.")
                return

            self.control_module.set_throttle(int(sign_speed))
            self.driver_interface.render_alert(f"Speed adjusted to
{sign_speed} mph.")
        except Exception:
            self.driver_interface.render_alert("Speed regulation
unavailable. Maintain manual control.")
```

The IntelligentSpeedAdaptation class enables dynamic speed control by reading speed

limit signs and validating them against GPS data. It adjusts the throttle accordingly and alerts the driver in case of conflicting speed inputs or system errors, ensuring compliance with legal speed limits.

```
class DriverDrowsinessDetection:
    def __init__(self, camera_sensor, biometric_sensor, sensor_fusion,
planning_module, driver_interface):
        self.camera_sensor = camera_sensor
        self.biometric_sensor = biometric_sensor
        self.sensor_fusion = sensor_fusion
        self.planning_module = planning_module
        self.driver_interface = driver_interface

    def monitor_driver(self):
        try:
            eye_data = self.camera_sensor.detect_driver_activity()
            bio_data = self.biometric_sensor.get_driver_biometrics()
            fused = self.sensor_fusion.integrate_sensor_data({"eye":
eye_data, "bio": bio_data})

            if fused.get("drowsiness_level", "low") in ["medium", "high"]:
                self.driver_interface.render_alert("Driver drowsiness
detected. Take a break!")
        except Exception:
            self.driver_interface.render_alert("Drowsiness detection
disabled. Camera/bio sensor error.")
```

Driver safety is further enhanced by the DriverDrowsinessDetection class, which monitors biometric and visual cues such as eye activity and grip patterns. It integrates sensor inputs to assess fatigue levels and notifies the driver if signs of drowsiness are detected, prompting corrective actions.

7 Testing

This section details the test cases used to ensure the software performs as expected for each task it completes. Tests are based on the use cases as well as the explicitly written requirements of the software, both of which have been outlined previously in this document. Upon successful completion of these tests, it can be ensured that the finished product will be fully operational as per its described functionality.

7.1 Validation Testing

7.1.1 Cruise Control activated

```
def test_cc_activate_success(self):
        ctrl = ControlModule()
        fusion = SensorFusion()
        planning = PlanningModule(ctrl, None, None)
        ui = MockDriverInterface()
        admin = MockSystemAdmin()

        cc = CruiseControlManager(ctrl, fusion, planning, admin, ui)
        cc.activate(30)

        self.assertTrue(cc.active)
        self.assertIn("Cruise Control activated.", admin.logs[-1])

    def test_cc_activate_speed_too_low(self):
        ctrl = ControlModule()
        fusion = SensorFusion()
        planning = PlanningModule(ctrl, None, None)
        ui = MockDriverInterface()
        admin = MockSystemAdmin()

        cc = CruiseControlManager(ctrl, fusion, planning, admin, ui)
        cc.activate(20)

        self.assertFalse(cc.active)
        self.assertIn("Speed too low", ui.alerts[0])
```

1. Objective: Confirm that Cruise Control can only be activated when vehicle speed is ≥ 25 mph and that the driver is warned if speed is too low.
2. Preconditions:

a. Vehicle ignition is on and transmission is in “Drive.”
b. Speed sensor and Driver Interface are functional.

3. Test Inputs:
   c. Simulated current-speed values – 24 mph and 25 mph.
   d. Driver presses “Cruise Control” button.
4. Expected results::
   e. At 24 mph: system refuses activation and posts “Speed too low for Cruise Control.”
   f. At 25 mph (or higher): system enables Cruise Control and logs “Cruise Control activated.”.
5. Test procedure:
   - Step 1: Set vehicle speed to 24 mph, press Cruise-Control button. Expected Outcome: Activation denied; alert displayed; cruise flag remains false.
   - Step 2: Increase speed to 25 mph, press button again. Expected Outcome: Cruise flag switches to true; admin log records activation.
6. Exceptions: Speed-sensor failure → driver alerted; Cruise Control locked out; event logged.
7. Postconditions: Cruise state accurately reflects rules; all alerts / logs stored for audit.

7.1.2 Obstruction detected ahead

```
def test_collision_close_brake(self):
        ctrl = ControlModule()
        fusion = SensorFusion()
        planning = PlanningModule(ctrl, None, None)
        ui = MockDriverInterface()
        admin = MockSystemAdmin()

        perception = MockPerceptionModule(3)   # ≤5 ft ⇒ full brake
        cds = CollisionDetectionSystem(perception, fusion, planning,
                                       ctrl, ui, admin)
        cds.monitor()

        self.assertEqual(ctrl.brakingForce, 100)
        self.assertIn("Emergency braking", admin.logs[-1])

    def test_collision_far_speed_adjust(self):
        ctrl = ControlModule()
        fusion = SensorFusion()
        planning = PlanningModule(ctrl, None, None)
        ui = MockDriverInterface()
        admin = MockSystemAdmin()
```

```
        perception = MockPerceptionModule(12)  # 12 ft ⇒ 12 mph rule
        cds = CollisionDetectionSystem(perception, fusion, planning,
                                       ctrl, ui, admin)
        cds.monitor()

        self.assertEqual(ctrl.currentSpeed, 12)
        self.assertIn("Adjusting speed to 12", admin.logs[-1])
```

1. Objective: Verify that the Collision Detection System maps obstacle distance to throttle/brake exactly as required: Speed = distance (ft) when distance > 5 ft; full braking when ≤ 5 ft.
2. Preconditions:
    - Vehicle in motion; perception sensors operational.
    - Control Module able to set throttle and braking force.
3. Test Inputs: Simulated obstacle distances: 12 ft and 3 ft.
4. Expected Results:
    - 12 ft → vehicle commanded to 12 mph, brake force 0 %.
    - 3 ft → brake force set to 100 %, vehicle speed 0.
5. Test procedure:
    - Step 1: Inject perception data with obstacle_distance = 12 ft. Expected: Speed set to 12 mph; admin log “Adjusting speed to 12 mph.”
    - Step 2: Inject obstacle_distance = 3 ft. Expected: apply_brake(100) invoked; alert “Immediate stop!”
6. Exceptions: Sensor data missing/inconsistent → system alerts driver and reduces speed to 10 mph failsafe.
7. Postconditions: Vehicle either decelerates or stops per rule; incident recorded.

7.1.3 Cruise Control Significant weight detected in seat

```
def test_weight_detection_child(self):
        seat = MockSeatSensor(1, 40)           # <50 lbs ⇒ Child
        fusion = SensorFusion()
        planning = PlanningModule(None, None, None)
        ui = MockDriverInterface()
        admin = MockSystemAdmin()

        wds = WeightDetectionSystem([seat], fusion, planning, ui,
admin)
        wds.evaluate_seats()
```

```
        self.assertIn("Child detected", ui.alerts[0])
        self.assertIn("40 lbs - Child", admin.logs[-1])
```

1. Objective: Verify that a seat-weight reading < 50 lbs is categorized as Child, triggers a seat-belt alert, and is logged with the exact weight and category.
2. Preconditions:
    - Vehicle ignition ON.
    - Seat-weight sensors operational.

3. Test inputs:
    - Simulated seat-sensor packet → seatID=1, weight=40 lbs.
4. Expected Results:
    - Driver-interface alert text contains “Child detected”.
    - System-admin log entry ends with “40 lbs – Child”.
5. Test Procedure:
    - Step 1: Inject weight = 40 lbs into the Weight-Detection System.
    - Step 2: Invoke evaluate_seats()
    - Step 3: Verify UI alert and log strings exactly match expectations.
6. Exceptions: If the sensor raises an exception, system must post “Sensor error in seat <id>. Defaulting to high safety.” and log the failure.
7. Postconditions: Correct alert remains visible; log preserved for audit.

7.1.4 Change in environment brightness detected

```
def test_headlight_brightness(self):
        ctrl = ControlModule()
        fusion = SensorFusion()
        planning = PlanningModule(None, None, None)
        ui = MockDriverInterface()
        admin = MockSystemAdmin()
        cam = MockCameraSensor(8)              # very dark

        ahc = AdaptiveHeadlightController(cam, ctrl, fusion,
                                          planning, ui, admin)
        ahc.adjust_lights()
        self.assertEqual(ctrl.lightLevel, "High")

        cam._brightness = 35                   # dusk
        ahc.adjust_lights()
        self.assertEqual(ctrl.lightLevel, "Low")
```

```
        cam._brightness = 70                # bright day
        ahc.adjust_lights()
        self.assertEqual(ctrl.lightLevel, "Off")
```

1. Objective: Confirm head-beam level changes precisely follow brightness thresholds and that fallback behaviour occurs on sensor fault or manual override.
2. Preconditions:
    - Vehicle in motion; camera-brightness sensor functional.\
    - Control Module able to set headlight level.

3. Test Inputs: Simulated brightness values: 8 lux, 35 lux, 70 lux.
4. Expected Results: ≤ 10 lux → High 11–30 lux → Medium 31–50 lux → Low > 50 lux → Off
5. Test Procedure:
    - Set brightness = 8 → call adjust_lights() → expect control.lightLevel == "High".
    - Set brightness = 35 → expect "Low".
    - Set brightness = 70 → expect "Off".
6. Exceptions:
    - Simulate camera-sensor exception → system must set headlights to “Standard”, raise UI alert “Headlight sensor error…”, and log fallback.
    - Manual override by driver → system suspends automatic changes, logs event.
7. Postconditions: Head-beam state matches last valid rule; all actions/alerts logged.

7.1.5 Vehicle software update

```
def test_cloud_update_success(self):
    comms = MockCommunicationModule(True)
    admin = MockSystemAdmin()
    ui = MockDriverInterface()

    cim = CloudIntegrationManager(comms, admin, ui)
    cim.perform_update()

    # the production code logs *exactly* this string
    self.assertIn("Update applied successfully.", admin.logs[-1])

def test_cloud_update_deferred_when_offline(self):
    # start disconnected *and* make reconnection fail
    comms = MockCommunicationModule(False)
    comms.establish_connection = lambda: None        # keep it offline
```

```
    admin = MockSystemAdmin()
    ui = MockDriverInterface()
    cim = CloudIntegrationManager(comms, admin, ui)
    cim.perform_update()

    self.assertIn("deferred due to no connectivity",
admin.logs[-1].lower())
    self.assertTrue(any("no internet connection" in a.lower()
                        for a in ui.alerts))
```

1. Objective: Ensure that when an internet connection is available the update package is downloaded, applied, driver notified, and the admin log records success.
2. Preconditions: communication_module.connectionStatus == True
3. Test Inputs: Trigger one software-update action via CloudIntegrationManager.
4. Expected Results:
    - Driver alert: “System updated successfully.”
    - Admin log entry ends: “Update applied successfully.”
5. Test Procedure:
    - Instantiate Cloud-Integration Manager with connected comm-module.
    - Call perform_update().
    - Assert alert and log strings exactly as above.
6. Exceptions: If download_updates() raises, CIM must alert “Update failed. Rolling back.” and log rollback message with exception text.
7. Postconditions: Version v1.2.3 registered; success/rollback event logged.

7.2 Scenario-Based Testing

7.2.1 Cruise Control

```
def test_cruise_control_safe_following(self):
        ctrl = ControlModule()
        fusion = SensorFusion()
        planning = PlanningModule(ctrl, None, None)
        ui = MockDriverInterface()
        admin = MockSystemAdmin()

        cc = CruiseControlManager(ctrl, fusion, planning, admin, ui)
        cc.activate(30)
        vehicle_speed = 30
        lead_vehicle_distance = 10
```

```
        lead_vehicle_speed = 20

        cc.update(vehicle_speed, lead_vehicle_distance, lead_vehicle_speed)

        self.assertEqual(ctrl.brakingForce, 10)
        self.assertIn("Cruise Control decelerated", admin.logs[-1])

        lead_vehicle_distance = 20

        cc.update(vehicle_speed, lead_vehicle_distance, lead_vehicle_speed)

        self.assertEqual(ctrl.currentSpeed, vehicle_speed + 1)
        self.assertIn("Cruise Control accelerated", admin.logs[-1])

        cc.disengage("Manual override by driver")

        self.assertFalse(cc.active)
        self.assertIn("disengaged: Manual override", admin.logs[-1])
        self.assertIn("Cruise Control disengaged", ui.alerts[-1])
```

1. Objective: Ensure that Cruise Control keeps the vehicle at a safe following distance and properly adjusts based on traffic conditions or driver input
2. Pre-conditions:
    a. Vehicle speed is above 25 mph
    b. Driver has activated the Cruise Control function through the Driver Assistance Interface
3. Test Inputs:
    a. Vehicle speed = 30 mph
    b. Leading vehicle distance = 10ft and 20ft
    c. Driver presses “Cruise Control” button
4. Expected Results:
    a. When leading vehicle distance is 10ft → system applies braking force by 10% and logs “Cruise Control decelerated due to close vehicle.”
    b. When leading vehicle distance is 20ft → system increases throttle to speed+1 mph and logs “Cruise Control accelerated.”
5. Test Procedure:
    a. Set vehicle speed to 30 mph and activate Cruise Control. Expected: Cruise Control becomes active and admin log records activation
    b. Simulate lead vehicle at 10 ft distance. Expected: Braking force of 10% is applied and admin log records deceleration event

c. Simulate lead vehicle at 20 ft distance. Expected: Vehicle accelerates by 1 mph and admin log records acceleration event.

6. Exceptions:
    a. If a sudden deceleration is detected from the car ahead, the system automatically disengages Cruise Control and issues a driver alert to take control immediately
7. Post-conditions:
    a. The vehicle maintains the selected speed or safe following distance
    b. Any manual override by the driver (brake/accelerator) stops Cruise Control

7.2.2 Collision Detection

```
def test_collision_detection_speed_adjust_on_distant_obstacle(self):
      ctrl = ControlModule()
      fusion = SensorFusion()
      planning = PlanningModule(ctrl, None, None)
      ui = MockDriverInterface()
      admin = MockSystemAdmin()

      perception = MockPerceptionModule(15)
      cds = CollisionDetectionSystem(perception, fusion, planning,
                                     ctrl, ui, admin)

      cds.monitor()

      self.assertEqual(ctrl.currentSpeed, 15)
      self.assertIn("Adjusting speed to 15", admin.logs[-1])

def test_collision_detection_sensor_failure_reduces_speed(self):
      ctrl = ControlModule()
      fusion = SensorFusion()
      planning = PlanningModule(ctrl, None, None)
      ui = MockDriverInterface()
      admin = MockSystemAdmin()

      cds = CollisionDetectionSystem(None, fusion, planning,
                                     ctrl, ui, admin)

      cds.handle_sensor_failure()

      self.assertEqual(ctrl.currentSpeed, 10)
```

```
        self.assertIn("Sensor malfunction", admin.logs[-1])
        self.assertIn("Sensor Failure", ui.alerts[-1])
```

1. Objective: Collision Detection prevents collisions by continuously monitoring sensor data and triggering braking or avoidance maneuvers when an obstruction is detected.
2. Pre-conditions:
    a. Vehicle is powered on and in motion
    b. Sensor Fusion is active and receiving data from the Perception Module
3. Test Inputs:
    a. Simulated obstacle distances: 4 ft, 15 ft
    b. Simulated sensor failure for exception handling
4. Expected Results:
    a. 4 ft → system applies 100% braking force and logs "Emergency braking: Obstruction at close range" while displaying "Immediate stop! Obstruction ahead" alert to driver.
    b. 15 ft → system adjusts speed to 15 mph and logs "Adjusting speed to 15 mph for obstruction."
    c. Sensor failure → system limits speed to 10 mph and logs "Sensor malfunction - reduced speed mode activated" while displaying "Sensor Failure: Speed limited." alert.
5. Test Procedure:
    a. Simulate obstacle detected at 4 ft. Expected: 100% braking force applied, emergency stop alert displayed, and admin log records emergency braking event.
    b. Simulate obstacle detected at 15 ft. Expected: vehicle speed adjusted to 15 mph and admin log records speed adjustment event.
    c. Simulate sensor failure by invoking sensor failure handling. Expected: vehicle speed limited to 10 mph, sensor malfunction alert is displayed, and admin log records fault.
6. Exceptions:
    a. If there is a sensor malfunction, the Planning Module logs an error and reduces vehicle speed automatically until the system is safe
7. Post-conditions:
    a. Vehicle either slows down or comes to a complete stop if an obstacle is detected directly ahead
    b. Collision warnings are logged in the system

7.2.3 Weight Detection in Seats

```
def test_weight_detection_seatbelt_and_airbag_adjustment(self):
        seat1 = MockSeatSensor(1, 45)
        seat2 = MockSeatSensor(2, 130)
```

```
        fusion = SensorFusion()
        planning = PlanningModule(None, None, None)
        ui = MockDriverInterface()
        admin = MockSystemAdmin()

        wds = WeightDetectionSystem([seat1, seat2], fusion, planning, ui, admin)
        wds.evaluate_seats()

        self.assertIn("Child detected in seat 1", ui.alerts[0])
        self.assertIn("45 lbs - Child", admin.logs[0])

        self.assertIn("Adult detected in seat 2", ui.alerts[1])
        self.assertIn("130 lbs - Adult", admin.logs[1])

def test_weight_detection_sensor_failure_defaults_to_safety(self):
        broken_seat = MockSeatSensor(3, 0)
        broken_seat.measure_weight = lambda: (_ for _ in
()).throw(Exception("Sensor failure"))

        fusion = SensorFusion()
        planning = PlanningModule(None, None, None)
        ui = MockDriverInterface()
        admin = MockSystemAdmin()

        wds = WeightDetectionSystem([broken_seat], fusion, planning, ui, admin)
        wds.evaluate_seats()

        self.assertIn("Sensor error in seat 3", ui.alerts[0])
        self.assertIn("Seat 3 failed", admin.logs[0])
```

1. Objective: Weight Detection in Seats ensures passenger safety by detecting occupancy and adjusting restraints (airbag activation and seatbelt alerts) accordingly.
2. Pre-conditions:
    a. Vehicle ignition is on
    b. Seat sensors are operational
3. Test Inputs:
    a. Simulated occupant weights: 45 lbs, 130 lbs
    b. Simulated sensor failure for exception handling
4. Expected Results:

a. 45 lbs → system classifies occupant as “Child,” Driver Assistance Interface prompts seatbelt reminder, and system admin logs “Occupant in seat 1: 45 lbs - Child”.
b. 130 lbs → system classifies occupant as “Adult,” Driver Assistance Interface prompts seatbelt reminder, and system admin logs “Occupant in seat 1: 130 lbs - Adult”.
c. Sensor failure → system displays “Sensor error in seat 1. Defaulting to high safety.” System admin logs “Seat 1 failed. Applied high safety measures.”

5. Test Procedure:
    a. Simulate occupant weight at 45 lbs. Expected: occupant classified as “Child,” seatbelt alert displayed, and system admin logs occupant weight.
    b. Simulate occupant weight at 130 lbs. Expected: occupant classified as “Adult,” seatbelt alert displayed, and system admin logs occupant weight.
    c. Simulate sensor failure. Expected: error alert displayed, system defaults to maximum safety setting, and system admin logs sensor failure event.
6. Exceptions:
    a. If a seat sensor malfunctions or fails to provide data, the system displays an error message and defaults to the highest safety setting for that seat
7. Post-conditions:
    a. Occupied seats trigger seatbelt reminder alerts and log a safety violation if the passenger is unbuckled
    b. Airbags are properly armed based on occupant weight thresholds

7.2.4 Camera Detection System

```
def test_camera_detection_normal_operation(self):
        cam = MockCameraSensor(brightness=20)
        fusion = SensorFusion()
        planning = PlanningModule(None, None, None)
        ui = MockDriverInterface()
        admin = MockSystemAdmin()

        cds = CameraDetectionSystem(cam, fusion, planning, ui, admin)
        cds.detect_environment()

        self.assertIn("video", fusion.outputData)
        self.assertIn("objects", fusion.outputData)
        self.assertIn("lighting", fusion.outputData)
        self.assertIn("Camera data analyzed", admin.logs[-1])

def test_camera_detection_failure_reverts_to_alternative(self):
```

```
        cam = MockCameraSensor()
        cam.capture_image = lambda: (_ for _ in ()).throw(Exception("Lens
obscured"))

        fusion = SensorFusion()
        planning = PlanningModule(None, None, None)
        ui = MockDriverInterface()
        admin = MockSystemAdmin()

        cds = CameraDetectionSystem(cam, fusion, planning, ui, admin)
        cds.detect_environment()

        self.assertIn("Camera fault detected", admin.logs[-1])
        self.assertIn("Camera Fault", ui.alerts[0])
```

1. Objective: The Camera Detection System uses computer vision to identify road signs, pedestrians, and other vehicles in real time, sending classification data to the Planning Module.
2. Pre-conditions:
    a. Camera sensors are powered and calibrated
    b. Adequate visibility or night-vision capability is available for camera operation
3. Test Inputs:
    a. Camera sensor detects road sign
    b. Camera sensor raises an exception due to obscured lenses
4. Expected Results:
    a. Road sign → Sensor Fusion contains video frame, objects, and lighting data. Planning Module receives classification data for further action, and system admin logs successful camera analysis
    b. Obscured lenses → Driver Assistance Interface displays “Camera Fault: Switching to alternative sensors.” System admin logs “Camera fault detected. Reverted to fallback sensors.”
5. Test Procedure:
    a. Simulate camera capturing normal video feed. Expected: Captured data of the road sign is integrated, classification results are sent to Planning Module, and system admin logs successful analysis.
    b. Simulate camera failure due to obscured lenses. Expected: Fallback procedure triggered, driver alert displayed, and system admin logs camera fault event.
6. Exceptions:
    a. If camera lenses are obscured or the system cannot classify objects reliably, the Planning Module reverts to alternative sensors and issues a “Camera Fault” alert
7. Post-conditions:

a. Road sign or obstacle information is stored in sensor logs
b. Planning Module receives classification data

7.2.5 Cloud Integration for Real-Time Updates

```
def test_cloud_update_successful_and_logs_synced(self):
        comms = MockCommunicationModule(connected=True)
        admin = MockSystemAdmin()
        ui = MockDriverInterface()

        cim = CloudIntegrationManager(comms, admin, ui)
        cim.perform_update()
        cim.sync_logs(["log1", "log2"])

        self.assertIn("Update applied successfully.", admin.logs[-2])
        self.assertIn("Logs synced to cloud.", admin.logs[-1])
        self.assertTrue(comms.synced)

def test_cloud_update_connection_drop_reverts_and_logs_error(self):
        comms = MockCommunicationModule(connected=False)
        comms.establish_connection = lambda: None

        admin = MockSystemAdmin()
        ui = MockDriverInterface()

        cim = CloudIntegrationManager(comms, admin, ui)
        cim.perform_update()

        self.assertIn("Update deferred due to no connectivity.", admin.logs[-1])
        self.assertTrue(any("No internet connection" in alert for alert in
ui.alerts))
```

1. Objective: Cloud Integration for Real-Time Updates allows for continuous updates, remote diagnostics, and data sharing with other IoT-connected vehicles or central servers.
2. Pre-conditions:
    a. Vehicle has an active internet/cloud connection
    b. Communication Module is online and authenticated with cloud services
3. Test Inputs:
    a. Normal operation → Communication Module starts connected and update and log sync requests are successfully performed.

    b. Failed operation → Communication Module starts disconnected and fails to reconnect
4. Expected Results:
    a. Normal operation → system admin successfully applies any available software updates, vehicle logs are synced to the cloud server, Driver Assistance Interface notifies the driver of update success (if applicable), and system admin logs update success and log sync events
    b. Failed operation → system defers updates, system admin logs an update deferment event, Driver Assistance Interface alerts the driver about deferred updates, and system remains on previous stable software version.
5. Test Procedure:
    a. Simulate active cloud connection and perform a cloud update followed by a log sync. Expected: update successfully applied, logs synced to cloud, and system admin log records success events.
    b. Simulate offline state and prevent reconnection. Expected: update deferred, system admin log records deferred event, Driver Assistance Interface alerts driver about loss of connection and deferred updates.
6. Exceptions:
    a. If the connection drops mid-update, the system reverts to the previous stable software version. It then logs an error and attempts the update again when connectivity is restored
7. Post-conditions:
    a. Updated software modules are applied to the vehicle's system
    b. Vehicle logs and diagnostic data are successfully synced with the cloud

7.2.6 Technician Interface & Diagnostics

```
def test_technician_successful_login_and_diagnostics(self):
        admin = MockSystemAdmin()
        tech = TechnicianInterface(admin)

        admin.authenticate_user = lambda u, p: setattr(admin, 'authStatus', True)

        tech.mfa_code = "123456"
        TechnicianInterface.random = random
        login_result = True

        admin.run_diagnostics = lambda: "All systems operational."
        diag_result = tech.run_diagnostics()

        self.assertEqual(diag_result, "All systems operational.")
```

```
def test_technician_failed_login_denies_access(self):
        admin = MockSystemAdmin()
        tech = TechnicianInterface(admin)

        admin.authenticate_user = lambda u, p: setattr(admin, 'authStatus',
False)

        login_result = tech.login("fakeuser", "fakepass")

        self.assertFalse(login_result)
        self.assertIn("Unauthorized access attempt", admin.logs[-1])

def test_technician_apply_patch_requires_auth(self):
        admin = MockSystemAdmin()
        tech = TechnicianInterface(admin)

        admin.authStatus = False

        patch_result = tech.apply_patch("patch_v1.0")

        self.assertEqual(patch_result, "Access denied")

        admin.authStatus = True
        patch_result = tech.apply_patch("patch_v1.0")

        self.assertEqual(patch_result, "Patch applied")
        self.assertIn("update_applied:patch_v1.0", admin.logs[-1])
```

1. Objective: A technician can log into the System Admin Module to retrieve logs, perform diagnostics, and apply updates or patches securely.
2. Pre-conditions:
    a. Vehicle is stationary
    b. Vehicle is powered on
    c. A technician has valid credentials (username and password) to access the admin console
3. Test Inputs:
    a. authStats = True, False (representing valid and invalid username and password)
    b. Diagnostic request from authenticated technician
    c. Patch application requests from authorized and unauthorized technician

4. Expected Results:
    a. authStats = True: technician successfully authenticated, system admin allows running diagnostics and applying patches, and diagnostic logs are updated and viewable
    b. authStats = False: system denies access and system admin logs record unauthorized attempt
    c. Patch application for authorized technician: patch applied successfully and system admin logs record event.
    d. Patch application for unauthorized technician: patch denied with “Access denied” response.
5. Test Procedure:
    a. Simulate login with valid credentials. Expected: access granted, technician authenticated.
    b. Simulate login with invalid credentials. Expected: access denied, system admin log records unauthorized attempt.
    c. After successful login, perform system diagnostics. Expected: diagnostic status returned and system admin log records diagnostic action
    d. Simulate unauthorized technician attempting to apply a software patch. Expected: access denied.
    e. Simulate authorized technician attempting to apply a software patch. Expected: patch applied successfully and system admin log records event.
6. Exceptions:
    a. If authentication fails, the system denies access and records the attempt in an unauthorized-access log
7. Post-conditions:
    a. Authorized technician can view system status, view logs, and update the software without disrupting vehicle safety
    b. All diagnostic actions are recorded for auditing purposes

7.2.7 Adaptive Headlights Control

```
def test_adaptive_headlight_control(self):
        ctrl = ControlModule()
        fusion = SensorFusion()
        planning = PlanningModule(None, None, None)
        ui = MockDriverInterface()
        admin = MockSystemAdmin()

        cam = MockCameraSensor(brightness=30)
        ahc = AdaptiveHeadlightController(cam, ctrl, fusion, planning, ui, admin)
```

```
        ahc.adjust_lights()
        self.assertEqual(ctrl.lightLevel, "High")
        self.assertIn("Adaptive headlights adjusted to High", admin.logs[-1])

        cam._brightness = 50
        ahc.adjust_lights()
        self.assertEqual(ctrl.lightLevel, "Low")
        self.assertIn("Adaptive headlights adjusted to Low", admin.logs[-1])

        cam._brightness = 100
        ahc.adjust_lights()
        self.assertEqual(ctrl.lightLevel, "Off")
        self.assertIn("Adaptive headlights adjusted to Off", admin.logs[-1])

        cam.capture_image = lambda: (_ for _ in ()).throw(Exception("Lens
obscured"))
        ahc.adjust_lights()
        self.assertEqual(ctrl.lightLevel, "Low")
        self.assertIn("Camera fault detected, defaulting to Low", admin.logs[-1])
        self.assertIn("Camera Fault", ui.alerts[0])

        fusion.get_environment_data = lambda: {"curve": "sharp", "traffic":
"oncoming"}
        ahc.adjust_lights()
        self.assertEqual(ctrl.lightLevel, "Low")
        self.assertIn("Adaptive headlights adjusted to Low due to curve and
oncoming traffic", admin.logs[-1])
```

1. Objective: Adaptive Headlights Control automatically adjusts the vehicle's headlights based on driving conditions.
2. Pre-conditions:
    a. Vehicle is powered on and in motion
    b. Sensors and camera systems are functional
3. Test Inputs:
    a. Brightness: 30, 50, 100
    b. Camera sensor failure
    c. Sharp road curve and oncoming traffic
4. Expected Results:
    a. 30 → headlight level set to "High" and system admin log records "Adaptive headlights adjusted to High."

b. 50 → headlight level set to “Low” and system admin log records “Adaptive headlights adjusted to Low.”
c. 100 → headlight level set to “Off” and system admin log records “Adaptive headlights adjusted to Off.”
d. Camera sensor failure → headlight level set to “Low” as fallback, Driver Assistance Interface displays the alert “Camera Fault,” and system admin log records “Camera fault detected, defaulting to Low.”
e. Road curve and oncoming traffic → headlight level set to “Low” and system admin log records “Adaptive headlights adjusted to Low due to curve and oncoming traffic.”

5. Test Procedure:
    a. Simulate dark conditions. Expected: headlight level set to “High” and system admin log records “Adaptive headlights adjusted to High.”
    b. Simulate dusk conditions. Expected: headlight level set to “Low” and system admin log records “Adaptive headlights adjusted to Low.”
    c. Simulate bright conditions. Expected: headlight level set to “Off” and system admin log records “Adaptive headlights adjusted to Off.”
    d. Simulate camera sensor failure. Expected: headlight level set to “Low” as a fallback, Driver Assistance Interface displays the alert “Camera Fault,” and system admin log records “Camera fault detected, defaulting to Low.”
    e. Simulate road curve and oncoming traffic → headlight level set to “Low” and system admin log records “Adaptive headlights adjusted to Low due to curve and oncoming traffic.”
6. Exceptions:
    a. If the sensors fail or visibility conditions are unclear, the system defaults to standard headlight settings and issues an alert
7. Post-conditions:
    a. Headlights adjust automatically to optimize visibility and minimize glare for other drivers

7.2.8 Blind Spot Monitoring

```
def test_blind_spot_monitoring(self):
        ctrl = ControlModule()
        fusion = SensorFusion()
        planning = PlanningModule(None, None, None)
        ui = MockDriverInterface()
        admin = MockSystemAdmin()

        side_sensor = MockCameraSensor(vehicle_detected=False)
        camera_sensor = MockCameraSensor(vehicle_detected=False)
```

```
        bsm = BlindSpotMonitoring(ctrl, fusion, planning, side_sensor,
camera_sensor, ui, admin)

        side_sensor.vehicle_detected = True
        bsm.detect_vehicle_in_blind_spot()
        self.assertTrue(ctrl.blindSpotAlertActive)
        self.assertIn("Vehicle detected in blind spot", admin.logs[-1])
        self.assertIn("Blind Spot Alert triggered", ui.alerts[0])

        side_sensor.vehicle_detected = False
        bsm.detect_vehicle_in_blind_spot()
        self.assertFalse(ctrl.blindSpotAlertActive)
        self.assertIn("No vehicle detected in blind spot", admin.logs[-1])

        side_sensor.operational = False
        bsm.detect_vehicle_in_blind_spot()
        self.assertFalse(ctrl.blindSpotAlertActive)
        self.assertIn("Blind Spot Monitoring Deactivated due to sensor failure",
admin.logs[-1])
        self.assertIn("Sensor failure", ui.alerts[0])

        side_sensor.operational = True
        camera_sensor.operational = False
        bsm.detect_vehicle_in_blind_spot()
        self.assertFalse(ctrl.blindSpotAlertActive)
        self.assertIn("Blind Spot Monitoring Deactivated due to sensor failure",
admin.logs[-1])
        self.assertIn("Sensor failure", ui.alerts[0])

        fusion.get_environment_data = lambda: {"lane_change": "left", "traffic":
"oncoming"}
        bsm.detect_vehicle_in_blind_spot()
        self.assertTrue(ctrl.blindSpotAlertActive)
        self.assertIn("Blind Spot Alert triggered due to lane change and oncoming
traffic", admin.logs[-1])
        self.assertIn("Blind Spot Alert triggered", ui.alerts[0])
```

1. Objective: Blind Spot Monitoring detects objects that appear in ranges unable to be seen through the vehicle's side and rear view mirrors.

2. Pre-conditions:
    a. Vehicle is in motion, and powered on
    b. Side sensors and cameras are operational.
3. Test Inputs:
    a. side_sensor detects vehicle but camera_sensor doesn't (vehicle is in blind spot)
    b. Neither side_sensor nor camera_sensor detect vehicle (no vehicle in blind spot)
    c. side_sensor is not operational (Side sensor failure)
    d. camera_sensor is not operational (Camera sensor failure)
    e. Sensor Fusion integration
4. Expected Results:
    a. Vehicle in blind spot → blindSpotAlertActive = True, Driver Assistance Interface displays alert "Blind Spot Alert triggered," and system admin log records "Vehicle detected in blind spot."
    b. No vehicle in blind spot → blindSpotAlertActive = False and system admin log records "No vehicle detected in blind spot."
    c. Side sensor failure → blindSpotAlertActive = False, Driver Assistance Interface displays alert "Sensor failure," and system admin log records "Blind Spot Monitoring deactivated due to sensor failure."
    d. Camera sensor failure → blindSpotAlertActive = False, Driver Assistance Interface displays alert "Sensor failure," and system admin log records "Blind Spot Monitoring deactivated due to sensor failure."
    e. Sensor Fusion integration → blindSpotAlertActive = True, Driver Assistance Interface displays alert "Blind Spot Alert triggered," and system admin log records "Blind Spot Alert triggered due to lane change and oncoming traffic."
5. Test Procedure:
    a. Simulate vehicle in blind spot. Expected: blindSpotAlertActive = True, Driver Assistance Interface displays alert "Blind Spot Alert triggered," and system admin log records "Vehicle detected in blind spot."
    b. Simulate no vehicle in blind spot. Expected: blindSpotAlertActive = False and system admin log records "No vehicle detected in blind spot."
    c. Simulate side sensor failure. Expected: blindSpotAlertActive = False, Driver Assistance Interface displays alert "Sensor failure," and system admin log records "Blind Spot Monitoring deactivated due to sensor failure."
    d. Simulate camera sensor failure. Expected: blindSpotAlertActive = False, Driver Assistance Interface displays alert "Sensor failure," and system admin log records "Blind Spot Monitoring deactivated due to sensor failure."
    e. Simulate Sensor Fusion integration. Expected: blindSpotAlertActive = True, Driver Assistance Interface displays alert "Blind Spot Alert triggered," and system admin log records "Blind Spot Alert triggered due to lane change and oncoming traffic."
6. Exceptions:
    a. Blind Spot detection deactivated if sensores failed, notifying driver

7. Post-conditions:
    a. Driver notified and alerted of oncoming vehicles in blind spot

7.2.9 Intelligent Speed Adaptation

```
def test_intelligent_speed_adaptation(self):
        ctrl = ControlModule()
        gps = MockGPSSystem()
        cam = MockCameraDetectionSystem()
        planning = PlanningModule(None, None, None)
        ui = MockDriverInterface()
        admin = MockSystemAdmin()

        isa = IntelligentSpeedAdaptation(ctrl, gps, cam, planning, ui, admin)

        cam.detect_speed_limit = lambda: 60
        gps.get_speed_limit = lambda: 60
        isa.adjust_speed()
        self.assertEqual(ctrl.currentSpeed, 60)
        self.assertIn("Speed adjusted to 60 based on road sign and GPS",
admin.logs[-1])

        cam.detect_speed_limit = lambda: 50
        gps.get_speed_limit = lambda: 60
        isa.adjust_speed()
        self.assertEqual(ctrl.currentSpeed, 50)
        self.assertIn("Speed limit data inconsistent, adjusting to camera data",
admin.logs[-1])
        self.assertIn("Please manually adjust speed", ui.alerts[0])

        cam.detect_speed_limit = lambda: None
        gps.get_speed_limit = lambda: 70
        isa.adjust_speed()
        self.assertEqual(ctrl.currentSpeed, 70)
        self.assertIn("No sign detected, adjusting speed to GPS limit: 70",
admin.logs[-1])

        cam.detect_speed_limit = lambda: 30
        gps.get_speed_limit = lambda: None
        isa.adjust_speed()
```

```
        self.assertEqual(ctrl.currentSpeed, 30)
        self.assertIn("Speed limit data inconsistent, adjusting to camera data",
admin.logs[-1])
        self.assertIn("Please manually adjust speed", ui.alerts[0])

        gps.get_speed_limit = lambda: None
        cam.detect_speed_limit = lambda: None
        isa.adjust_speed()
        self.assertEqual(ctrl.currentSpeed, 0)
        self.assertIn("GPS and Camera systems failed, unable to adjust speed",
admin.logs[-1])
        self.assertIn("System failure", ui.alerts[0])
```

1. Objective: Intelligent Speed Adaptation regulates the speed of the vehicle based on the vehicle's surroundings and environmental conditions.
2. Pre-conditions:
    a. Vehicle in motion
    b. GPS and Camera Detection System is active and operational.
3. Test Inputs:
    a. detect_speed_limit = 60, 50, 30, None
    b. get_speed_limit = 60, 60, None, 70
    c. Camera or GPS failure
4. Expected Results:
    a. detect_speed_limit = 60 & get_speed_limit = 60 → speed adjusted to 60 mph and system admin log records "Speed adjusted to 60 based on road sign and GPS."
    b. detect_speed_limit = 50 and get_speed_limit = 60 → speed adjusted to 50 mph, Driver Assistance Interface displays alert "Please manually adjust speed," and system admin log records "Speed limit data inconsistent, adjusting to camera data."
    c. detect_speed_limit = 30 and get_speed_limit = None → speed adjusted to 30 mph, Driver Assistance Interface displays alert "Please manually adjust speed," and system admin log records "Speed limit data inconsistent, adjusting to camera data."
    d. detect_speed_limit = None and get_speed_limit = 70 → speed adjusted to 70 mph and system admin log records "No sign detected, adjusting speed to GPS limit: 70."
    e. Camera or GPS failure → speed adjusted to 0 mph (vehicle waits for speed data), Driver Assistance Interface displays "System failure," and system admin log records "Camera and GPS systems failed, unable to adjust speed."
5. Test Procedure:

a. Simulate sign and GPS data of 60 mph. Expected: speed adjusted to 60 mph and system admin log records “Speed adjusted to 60 based on road sign and GPS.”
b. Simulate sign data of 50 mph and GPS data of 60 mph. Expected: speed adjusted to 50 mph, Driver Assistance Interface displays alert “Please manually adjust speed,” and system admin log records “Speed limit data inconsistent, adjusting to camera data.”
c. Simulate sign data of 30 mph and no GPS data. Expected: speed adjusted to 30 mph, Driver Assistance Interface displays alert “Please manually adjust speed,” and system admin log records “Speed limit data inconsistent, adjusting to camera data.”
d. Simulate no sign data and GPS data of 70 mph. Expected: speed adjusted to 70 mph and system admin log records “No sign detected, adjusting speed to GPS limit: 70.”
e. Simulate camera and GPS failure. Expected: speed adjusted to 0 mph (vehicle waits for speed data), Driver Assistance Interface displays “System failure,” and system admin log records “Camera and GPS systems failed, unable to adjust speed.”

6. Exceptions:
    a. System prompts user to manually adjust speed if there is inconsistent speed limit data
7. Post-conditions:
    a. Vehicle adjusts speed automatically according to road speed limit signs

7.2.10 Driver Drowsiness Detection

```
def test_driver_drowsiness_detection(self):
        ctrl = ControlModule()
        cam = MockCameraDetectionSystem()
        sensors = MockBiometricSensors()
        fusion = SensorFusion()
        planning = PlanningModule(None, None, None)
        ui = MockDriverInterface()
        admin = MockSystemAdmin()

        ddd = DriverDrowsinessDetection(cam, sensors, fusion, planning, ui,
admin)

        cam.detect_eye_movement = lambda: "slow"
        cam.detect_head_position = lambda: "tilted"
        sensors.detect_grip = lambda: "loose"
```

```
        fusion.integrate_data = lambda: {"eye_movement": "slow", "head_position":
"tilted", "grip": "loose"}
        planning.assess_fatigue = lambda: "high"
        ddd.check_drowsiness()
        self.assertIn("Drowsiness detected, alerting driver", admin.logs[-1])
        self.assertIn("Drowsiness detected", ui.alerts[0])

        cam.detect_eye_movement = lambda: "normal"
        cam.detect_head_position = lambda: "normal"
        sensors.detect_grip = lambda: "firm"
        fusion.integrate_data = lambda: {"eye_movement": "normal",
"head_position": "normal", "grip": "firm"}
        planning.assess_fatigue = lambda: "low"
        ddd.check_drowsiness()
        self.assertNotIn("Drowsiness detected", admin.logs[-1])
        self.assertNotIn("Drowsiness detected", ui.alerts)

        cam.capture_image = lambda: (_ for _ in ()).throw(Exception("Camera
obstructed"))
        ddd.check_drowsiness()
        self.assertIn("Camera system obstructed, drowsiness detection disabled",
admin.logs[-1])
        self.assertIn("Camera Fault", ui.alerts[0])

        sensors.detect_grip = lambda: None
        ddd.check_drowsiness()
        self.assertIn("Biometric sensor failure, drowsiness detection disabled",
admin.logs[-1])
        self.assertIn("Sensor failure", ui.alerts[0])

        cam.detect_eye_movement = lambda: "normal"
        cam.detect_head_position = lambda: "normal"
        sensors.detect_grip = lambda: "firm"
        fusion.integrate_data = lambda: {"eye_movement": "normal",
"head_position": "normal", "grip": "firm"}
        planning.assess_fatigue = lambda: "high"
        ddd.check_drowsiness()
        self.assertIn("Drowsiness alert triggered due to high fatigue risk",
admin.logs[-1])
        self.assertIn("Drowsiness alert", ui.alerts[0])
```

1. Objective: Driver Drowsiness Detection alerts the driver when they are in a drowsy state while behind the wheel.
2. Pre-conditions:
    a. Vehicle in motion
    b. Driver is directly facing and in front of the camera and biometric sensors are functional
3. Test Inputs:
    a. detect_eye_movement = “slow”, detect_head_position = “tilted”, detect_grip = “loose”, assess_fatigue = “high” (driver drowsiness detected)
    b. detect_eye_movement = “normal”, detect_head_position = “normal”, detect_grip = “firm”, assess_fatigue = “low” (no driver drowsiness detected)
    c. detect_eye_movement = “normal”, detect_head_position = “normal”, detect_grip = “firm”, assess_fatigue = “high” (fatigue risk high but no driver drowsiness detected)
    d. Camera failure
    e. Biometric sensor failure
4. Expected Results:
    a. Drowsiness detected → Driver Assistance Interface displays alert “Drowsiness detected” and system admin log records “Drowsiness detected, alerting driver.”
    b. No drowsiness detected → No alert shown, no log entry
    c. No drowsiness detected but fatigue risk high → Driver Assistance Interface displays alert “Drowsiness alert” and system admin log records “Drowsiness alert triggered due to high fatigue risk.”
    d. Camera failure → Driver Assistance Interface displays alert “Camera Fault” and system admin log records “Camera system obstructed, drowsiness detection disabled.”
    e. Biometric sensor failure → Driver Assistance Interface displays alert “Sensor failure” and system admin log records “Biometric sensor failure, drowsiness detection disabled.”
5. Test Procedure:
    a. Simulate drowsiness detected. Expected: Driver Assistance Interface displays alert “Drowsiness detected” and system admin log records “Drowsiness detected, alerting driver.”
    b. Simulate no drowsiness detected. Expected: No alert shown, no log entry
    c. Simulate no drowsiness detected but high fatigue risk. Expected: Driver Assistance Interface displays alert “Drowsiness alert” and system admin log records “Drowsiness alert triggered due to high fatigue risk.”
    d. Simulate camera failure. Expected: Driver Assistance Interface displays alert “Camera Fault” and system admin log records “Camera system obstructed, drowsiness detection disabled.”

    e. Simulate biometric sensor failure. Expected: Driver Assistance Interface displays alert “Sensor failure” and system admin log records “Biometric sensor failure, drowsiness detection disabled.”
6. Exceptions:
    a. Obstructed camera system notifies the driver and disables drowsiness detection
7. Post-conditions:
    a. Alerts of drowsiness being detected are thrown on screen to the driver