Autonomous Vehicle System

Team Members

Samuel Garcia

Parsa Sedighi

Priyesh Patel

Christian Cate

Brandon Sosa

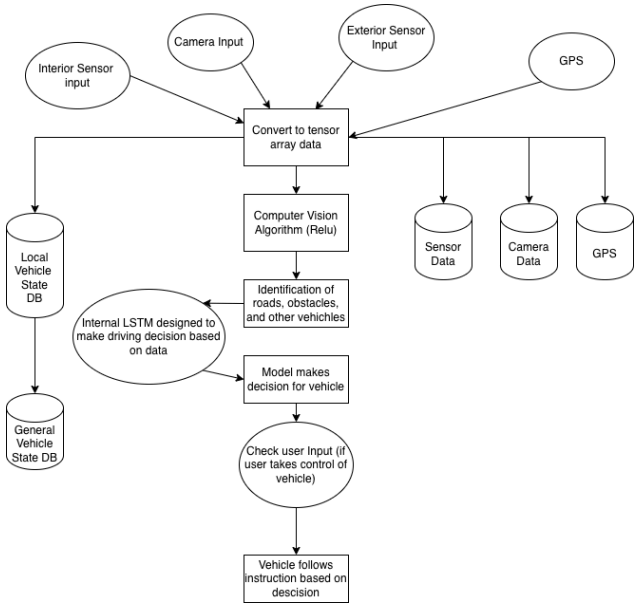
System Description

In this iteration of Autonomous systems, the commercial four wheeled vehicle aims to drive passengers to desired destinations with safety being top priority and no input from the user. Real time data through satellite feed will provide the system with the information required for navigation. Sensors such as camera, global positioning system (GPS) navigation etc are fused to detect objects, adjust driving mode per weather condition and navigate on different roads. All sensor data are collected in real time and monitored for sensor tuning and system safety checks.

Software Architecture Overview

- Architectural diagram of all major components
- UML Class Diagram
- Description of classes
- Description of attributes
- Description of operations

SWA



SWA Description

This system is designed to enable autonomous vehicle control through real-time data processing, computer vision, and decision-making using machine learning models. This system is designed to use 2 machine learning models, one to identify and one to drive. In the process of getting from data to action the system will go through a couple different phases.

1. **Data Acquisition**

The system collects data from multiple cameras, LiDAR, GPS and interior sensors that collect vehicle state information. This combines to bring enough data to paint a picture of the vehicle's surroundings, location, and status. All inputs are where they are preprocessed and converted into structured tensor arrays suitable for AI model consumption. At the same time all the data is saved to the local database to be sent to the general database when internet connection is available.

2. **Computer Vision and Environment Understanding**

The tensorized data is processed by a Computer Vision Algorithm utilizing ReLU (Rectified Linear Unit) activation functions. This actively allows the model to identify roads, lane markings, obstacles, other vehicles, traffic signs, and signals.

3. **Decision-Making**

The Internal LSTM (Long Short-Term Memory) uses the categorized output given by the Computer vision model to predict and determine the optimal driving action such as acceleration, braking, or steering.

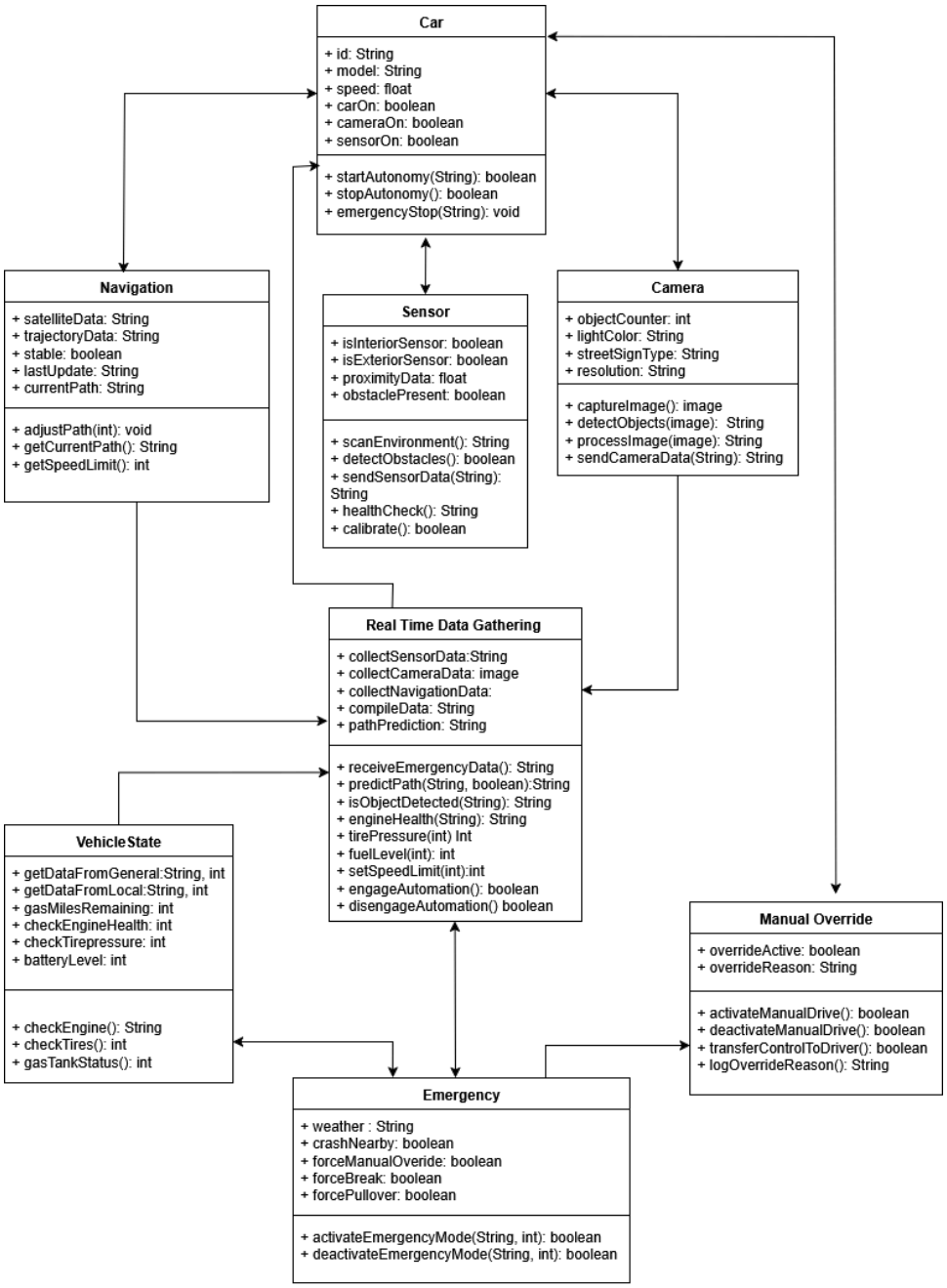
4. **User Control Verification**

Before execution, the system checks for User Input to determine whether the human driver has taken manual control. If no user override is detected, the Model’s Decision is executed.

5. Action Execution

Finally, the Vehicle Control Module executes driving commands, ensuring that the vehicle follows the chosen path safely and efficiently.

UML Class Diagram



Class Descriptions

1. Navigation

Receives data from satellite GPS system to help gather current path data and potential changes in path. Sends data to Real Time Data Gathering to ensure the current path is

safe, as well as gather the current path speed limit before sending to Real Time Data Gathering. Provides functionality to adjust path based on changes in navigation data.

Class Attributes:

- **trajectoryData: String** Stores data about the car's planned movement path, such as route coordinates or direction patterns. Used to determine how the car should navigate to its destination.
- **stable: boolean** Indicates whether the navigation system currently has a stable and reliable connection or path (true) or if it's unstable/interrupted (false).
- **lastUpdate: String** Records the timestamp or time of the most recent navigation data update. Useful for ensuring the system uses current route information.
- **currentPath: String** Represents the car's currently active navigation path, such as the road or segment it's following.
- **satelliteData:** String Contains information received from GPS or satellite systems, used to calculate positioning and routing accuracy.

Class Operations

- **getCurrentPath(): String** receives the current path from satellite GPS and converts data into a readable String for further use
- **adjustPath(int): void:** takes an integer value based off of satellite data to adjust the current path
- **getSpeedLimit(): int** receives an integer value that dictates speed limit of current path from satellite GPS

2. Camera

Receives imaging information from the front facing camera in the vehicle. The camera counts objects and differentiates between street signs and current stoplight status. The class captures and processes images, allowing for detection of objects in camera view. Data from camera is then sent to Real Time Data Gathering

Class Attributes

- **objectCounter: int** Keeps track of the number of objects detected by the camera, such as vehicles, pedestrians, or obstacles. Useful for object recognition and environmental awareness.
- **lightColor: String** Stores the current color detected from traffic lights. Helps determine vehicle behavior at intersections.
- **streetSignType: String** Identifies the type of street sign recognized by the camera. Used for decision making and navigation assistance.
- **resolution:** String Specifies the camera's image resolution. Higher resolutions provide more detailed visual data for processing.

Class Operations

- **captureImage(): image** Starts the camera recording function to capture images in front of the vehicle and its frontal surroundings as an image file

- **processImage(image): String** Takes the image file created by the captureImage function and converts the readings of the image file into a readable String for other functions
- **detectObjects(image): String** Reads the image file passed by the captureImage function and determines if there is a detectable object, and returns the result as a string for functions in the Real Time Data Gathering class
- **sendCameraData(String): String** Function to be able to log and send the String data received by the processImage function to the Vehicle State class

3. Sensor

The sensor class detects whether current sensor readings are coming from the interior or exterior sensors and differentiates between. The sensor class provides functionality to calibrate sensors, and if the sensor is interior, is able to detect engine health and stability. The exterior sensors scan for any objects in the proximity of the vehicle. The sensor class then sends data to Real Time Data Gathering

Class Attributes

- **proximityData: float** Measures the distance between the car and nearby objects, usually in meters or centimeters. Used for detecting how close the car is to obstacles.
- **obstaclePresent:** boolean Indicates whether an obstacle is currently detected by the sensor (true) or not (false). Helps trigger safety or avoidance mechanisms.
- **isExteriorSensor: boolean** Specifies if the sensor is located outside the car (true), such as on bumpers or mirrors, for detecting external objects or other vehicles.
- **isInteriorSensor: boolean** Specifies if the sensor is located inside the car (true), used for monitoring interior conditions like driver alertness or passenger presence.

Class Operations

- **calibrate(): void** Function to initialize or reset the sensor
- **scanEnvironment(): String** Scan surrounding environment of both interior and exterior sensors and converts current readings into a readable String for other functions
- **detectObstacles(String): String** Reads the String data from the scanEnvironment function and detects if there are any potential obstacles from the exterior sensor and converts readings into a String for further use
- **healthCheck(): String** Reads the String data from the interior sensor and detects and potential issues with engine health and converts readings into a String for further use
- **sendSensorData(String): String** Receives String data from the healthCheck, detectObstacles, and scanEnvironment functions and sends the String data to be used in the Vehicle State and Real Time Data Gathering classes

4. Real Time Data Gathering

The Real Time Data Gathering class handles all incoming information from the Sensor, Camera, and Navigation classes, as well as current Vehicle State. This data is then compiled and compared to check for safety concerns such as engine health, tire pressure, and current fuel, as well as check if any obstacles have been detected by both sensor and camera. The class also confirms, based on navigation data, that the current path is stable and viable, as well as the current speed limit of the current path. Once data is compared, the Real Time Data Gathering class then uses all information to predict the current safest path and then engage or disengage the autonomous driving, or force manual override in case of emergency.

Class Attributes

- **pathPrediction: String** Represents predicted route or trajectory data based on current navigation and sensor inputs. Used for forecasting the vehicle's next movements.
- **compileData: String** Stores processed or combined data collected from multiple subsystems. Useful for generating a comprehensive vehicle status overview.
- **collectNavigationData: String** Gathers real time information from the navigation system, such as GPS coordinates and route updates.
- **collectCameraData: image** Captures image or video data from the vehicle's camera system for object recognition, obstacle detection, and scene analysis.
- **collectSensorData: String** Retrieves data from the car's sensors to assist in environmental awareness and safety monitoring.

Class Operations

- **receiveEmergencyData():String** Receives data from the Emergency class and converts the data into a readable String for logging purposes
- **predictPath(String, boolean): String** Receives String data from Navigation class, as well as a boolean value from the isObjectDetected function to accurately predict the path for the vehicle before sending String data of path to Car class
- **isObjectDetected(String): boolean** Reads String data from Sensor and Camera classes and returns a boolean value of true if any objects are detected, or false otherwise
- **engineHealth(String): String** Receives String data from Vehicle State and Sensor class to determine the current status and health of the engine, and returns result as a String
- **tirePressure(int): int** Receives an integer value from the checkTires function in the Vehicle State class and returns an integer value for further use in Realtime Data Gathering
- **setSpeedLimit(int): int** Receives an integer value of the current path's posted speed limit from the Navigation class and returns an integer value to set the speed of the car during autonomous driving

- **fuelLevel(int): int** Receives an integer value from the Vehicle State class and returns the current level of fuel for Real time Data Gathering
- **engageAutomation(): boolean** Returns a boolean value of true if autonomous driving is currently engaged, false if otherwise
- **disengageAutomation(): boolean** Returns a boolean value of true if autonomous driving is disengaged, false if otherwise

5. Vehicle State

This class handles data held in both localized and generalized databases, as well as information from interior sensors. It relays information related to car safety such as engine health, tire pressure, and current fuel in the car to Real Time Data Gathering for further processing.

Class Attributes

- **checkTirePressure: int** Indicates the current tire pressure level in PSI or a percentage value. Used to monitor and alert when tire pressure is too low or too high.
- **batteryLevel: int** Represents the current battery charge percentage or voltage level of the vehicle. Important for electric or hybrid vehicle monitoring.
- **checkEngineHealth: int** Reflects the status of the engine's overall health, often represented as a diagnostic value or percentage. Helps identify maintenance needs.
- **gasMilesRemaining: int** Shows the estimated number of miles the car can travel with the remaining fuel. Useful for range estimation and trip planning.
- **getDataFromLocal: String, int** Stores or retrieves locally collected data along with an associated integer value, possibly representing data quantity or status code.
- **getDataFromGeneral: String, int** Stores or retrieves general vehicle system data along with an associated integer value for analysis or synchronization.

Class Operations

- **checkEngine(String): String** Reads String data form the interior sensor and stores the current engine status, which can then be sent to the Real Time Data Gathering class for further use
- **checkTires(): int** Checks current tire pressure and returns an integer value of the current tire pressure for use with Real Time Data Gathering
- **gasTankStatus(): int** Returns an integer value of the current fuel level of vehicle for use with Real Time Data Gathering

6. Emergency

This class handles emergency functions of the vehicle. Reads data from sensor and camera to determine current weather conditions, as well as communicates with vehicle

state to determine if the car is safely autonomously operable. The class provides methods to both activate and deactivate emergency mode, as well as force the vehicle to either pull over or manual override in the case of dangerous conditions or vehicle malfunctions based on data from Vehicle State.

Class Attributes

- **forcePullover: boolean** Indicates whether the system should command the vehicle to pull over immediately (true) due to an emergency or unsafe condition.
- **forceManualOverride: boolean** Determines if manual control should be forced (true) to let the driver take control during an emergency situation.
- forceBrake: boolean
- **weather: String** Describes the current weather conditions. Used to adjust driving behavior and safety measures.
- **crashNearby: boolean** Indicates whether an accident or collision has been detected in close proximity (true). Helps the system take precautionary or evasive actions.

Class Operations

- **activateEmergencyMode(String, int); boolean** Receives both String and integer inputs from the Vehicle State and Real Time Data Gathering classes to determine if there is an issue that requires either an immediate pullover or manual override, and if so passes true to the Manual Override class, otherwise remains false
- **deactivateEmergencyMode(String, int): boolean** Receives both String and integer values from Vehicle State and Real Time Gathering classes to determine if emergency situation has been resolved, and returns True, otherwise stays false.

7. Manual Override

This class allows manual driving, and in the case of an emergency, will force termination of the autonomous driving and give control of the vehicle back to the operator. This class has functionality to both activate and deactivate the manual control, as well as log the reason or conditions that manual override was engaged in the case of forced override by Emergency class.

Class Attributes

- **overrideActive: boolean** Indicates whether manual control of the vehicle is currently active (true) or if the autonomous system is in control (false).
- **overrideReason: String** Describes the reason manual override was engaged. Useful for logging and safety analysis.

Class Operations

- **activateManualDrive(): boolean** turns on manual operation and Returns a boolean value of true if the car is currently manually operable, or false if otherwise
- **deactivateManualDrive(): boolean** Returns a boolean value of true when the car is currently engaged in autonomous driving, or false if otherwise
- **transferControlToDriver(): boolean** When Emergency class declares that a manual override is necessary, this class with force manual operation and return a boolean value of true, or false if otherwise
- **logOverrideReason(): String** In the case of a forced manual override, this function will return a String value of the reason the override was caused for use in the Vehicle State class

8. Car

Handles information about car including make and model, current speed, and if the car is currently operating. The car can then either begin or end autonomous driving based on information received from Real Time Data Gathering. Also provides capability to emergency break in the case of unsafe driving conditions or car health based on Real Time Data Gathering.

Class Attributes

- **id: String** A unique identifier for the car (e.g., license plate number or system-assigned ID). It helps distinguish one car from another in the system.
- **model: String** The car's make and model name. It provides descriptive information about the vehicle type.
- **speed: float** The current speed of the car, typically measured in miles per hour (mph). Used for tracking or controlling car motion.
- **carOn: boolean** Indicates whether the car is currently turned on (true) or off (false).
- **cameraOn: boolean** Represents whether the car's camera system is active (true) or inactive (false). Useful for enabling visual data collection or safety features.
- **sensorOn: boolean** Shows whether the car's sensor system is active (true) or not (false). Used for navigation, collision detection, or automation.

Class Operations

- **startAutonomy(String): boolean** Function to start autonomous driving, will receive a String value from the predictPath function in Real Time Data Gathering and engage autonomous driving, returning a boolean value of true, and false once autonomous driving ends
- **stopAutonomy(): boolean** Function to stop autonomous driving, returning a boolean value when car is manually operable, and false once autonomous driving begins

- **emergencyStop(String): void** Receive a String input from the Emergency class when Real Time Data Gathering has detected an issue that requires an immediate stop, and will force car to end autonomy and come to a halt

Development plan and timeline

Task	Due Date	Members Responsible
Requirement specification	due by Oct 16, 202	Parsa Sedighi
Research of similar autonomous systems	due by Oct 30, 2025	Samuel Garcia Parsa Sedighi Priyesh Patel Christian Cate Brandon Sosa
First draft design of software development	due by Nov 20, 2025	Priyesh Patel Christian Cate Brandon Sosa
Unit Testing GPS, Computer vision and sensors	due by Nov 22, 2025	Samuel Garcia Parsa Sedighi
Second draft design of software development	due by Nov 27, 2025	Priyesh Patel Christian Cate Brandon Sosa

Unit Testing GPS, Computer vision and sensors	due by Nov 29, 2025	Samuel Garcia Parsa Sedighi
System Integration	due by Dec 4, 2025	Samuel Garcia Parsa Sedighi Priyesh Patel Christian Cate Brandon Sosa
System Integration Test	due by Dec 11, 2025	Priyesh Patel Christian Cate Brandon Sosa
System Test	due by Dec 18, 2025	Samuel Garcia Parsa Sedighi Priyesh Patel Christian Cate Brandon Sosa

Verification & Test Plan

Test Case 1 Detection Tests:

Description: Overall test cases for our detection systems. The tests will focus on the detection devices and services in the vehicle process. We will begin with a unit test of the camera perception classes and methods. Then do an integrated test checking that what is detected by the camera perception is able to be read and processed. Finally the system test for our detection systems will include an urban intersection where a takeover is applied and the driver is successfully able to take control of the vehicle and then return safely. Everything in this process would be logged in a successful test.

Unit Testing

Unit Set U1 Camera Perception

Description: This is the first step of the camera stream check. The goal of this test is to validate the functionality of the onboard camera. Object detection and classification are the first steps of this test, as they ensure traffic signal lights, objects and humans on the road are detected properly within 1% error. There is more complexity introduced in the next step, the type of object is tested as more attention to the content of the object such as yield or stop signs are required. Upon validation of all mentioned steps, the unit test is passed and this sub-system is ready to move forward to integration testing, where it will be part of the overall system.

Target Classes / Methods	Camera.captureImage, processImage, detectObjects, sendCameraData
Objective	Validate image classification, object counting, and message formatting.
Test 1 – Traffic Signal Classification	Input: images with red/yellow/green lights. Expected: Correct lightColor detection; no obstacle. Failures Covered: Wrong color mapping.
Test 2 – Object Density & Sign Type	Input: scenes with 0–20 objects; signs STOP, SPEED_LIMIT_45, YIELD. Expected: Correct counts and valid message. Failures Covered: Miscounts, malformed payloads.
Pass Criteria	≥99% correct color, ≥98% correct count, 100% valid schema.

Integration Testing

Integration Set I1 Perception -> RTDG Path Gating

Description: This test ensures that the perception from the camera is correctly taken, interpreted, and reacted to in an appropriate and efficient manner. This test will validate the fusion controls to ensure the automatic driving is safe and secure by testing when the road is clear and when the road has a sudden obstacle. The only way the program passes is if it makes the correct safest reaction to the situation.

Target Components	Camera, Sensor, RTDG.isObjectDetected, predictPath, engageAutomation
Objective	Validate perception fusion controls automation.
Scenario A – Clear Road	Input: no objects, obstaclePresent=false. Expected: isObjectDetected=false; automation on.
Scenario B – Sudden Obstacle	Input: pedestrian 0.5m. Expected: isObjectDetected=true; slow/stop or disengage.
Pass Criteria	Correct fusion logic and safe automation.

System Testing

System Set S1 Urban Intersection with Takeover

Description: The goal of this test is to ensure that the full system pipeline from the beginning of the instructions to the driver takeover is followed correctly, safely, and efficiently. The program has a set of instructions it must follow correctly where the driver takes control of the vehicle, the program will not pass unless all traffic laws are followed and it will also not pass if the transition to a human driver is not smooth and safe.

Objective	Validate full pipeline and manual takeover.
Steps	Start at 45 mph ->red light -> stop -> green -> driver takes control.
Expected Results	Stops correctly, resumes safely, takeover logged.
Pass Criteria	No red-light violation, smooth handoff to driver.

Test Case 2 Navigation and Emergency Tests:

Description: Overall test cases for our navigation systems and emergency systems. The tests will focus on the functions of the navigation class providing accurate readings of both speed limits and potential path stability. We will also be testing the integration of the navigation systems with our emergency systems by ensuring emergency systems engage properly, and in the case of path instability based on readings from navigation systems. We will also be testing how these systems behave in a more complex driving scenario of driving on a closed highway while also engaging in an emergency stop.

Unit Testing

Unit Set U2 Navigation & Speed Limits

Description: These tests check the functionality of our navigation and emergency systems by ensuring the readings based on navigation data for speed limits are accurate, and that the current path the

autonomous vehicle on is both safe and stable. It also tests to ensure that if the readings on speed limit are accurate, that they will translate into the proper speed for the autonomous vehicle. If the tests indicate there is an error in path safety and stability, that the functions can correctly alter path

Target Classes / Methods	Navigation.getCurrentPath, adjustPath, getSpeedLimit, RTDG.setSpeedLimit
Objective	Ensure accurate path stability and speed-limit management.
Test 1 – Speed Limit Extraction	Input: GPS limits {25, 45, 65, 70, 15}. Expected: Correct limit propagation. Failures Covered: Stale limit, wrong units.
Test 2 – Path Stability & Adjustment	Input: Stable vs. unstable paths; reroute adds waypoints. Expected: Updated path; correct stability flag. Failures Covered: Missed reroutes.
Pass Criteria	Correct limit propagation and valid path updates.

Integration Testing

Integration Set I2 VehicleState / Emergency -> ManualOverride

Description: This test checks if the Vehicle state changes into a dangerous state. If the engine overheats then automation would need to turn off and manual control will need to be given to the driver. Also if the environment is in a dangerous state like heavy snow then the automations turns off and emergency mode would be switched on.

Target Components	VehicleState, Emergency, ManualOverride
Objective	Verify hazards trigger driver control and logging.
Scenario A – Engine Overheat	Input: checkEngine=OVERHEAT. Expected: Manual override true; reason logged.
Scenario B – Adverse Weather	Input: weather=HEAVY_SNOW; low tire PSI. Expected: Emergency on, automation off.
Pass Criteria	Override triggers and logs correctly.

System Testing

System Set S2 Highway Obstacle & Emergency Pullover

Description: This test case makes sure the vehicle can successfully handle an obstacle and emergency pull over. In this test the vehicle is cruising at 65 miles per hour, the vehicle detects debris in the road 20 meters ahead and at the exact same time gets an emergency engine overheat. The system needs to detect the item successfully, navigate an emergency pull over, decellerate, and make sure it applies what is

needed to ensure the engine does not overheat. When the vehicle stops it needs to successfully stop, park, put the hazards on, and finally log the reason the engine overheated and log the exact sequence of events that occurred to trigger the emergency pull over. The criteria for this test to pass would be that the vehicle comes to a safe stop, no collision occurred, and all the data is logged.

Objective	Validate collision avoidance and safe pullover.
Steps	Cruise 65 mph -> debris 20m ->engine overheat ->emergency.
Expected Results	Decelerates, pulls over, hazards on, log reason.
Pass Criteria	Safe stop, no collision, full logging.

Data Management Strategy

The database strategy consists of two separate distinct parts, a non SQL database using mongoDB to handle all of our visual, sensor, and navigation data, and a SQL database to hold any data related to the state of the car both locally and generally.

The SQL part consists of Vehicle state, both local and General. Storing info about battery percentage, car condition and overall safety score of the vehicle.
We have chosen these to be SQL since they are relational in nature and the tables have data related to each other. For instance the local and general vehicle states are going to share some common columns and attributes. Since each individual piece of data does not need a huge amount of table space since most data points can be described with a couple numerical values over columns, it is a much more efficient way of storing, and finding data for later.

The non-SQL part consists of all of the parts that have to do with camera and sensor data. These types of data are highly inefficient to store in a table like SQL database. So these would be instead stored in a mongo DB database using GridFS. GridFS allows data to be stored that exceeds the BSON file size limit. This means that the GridFS architecture is specifically designed for large pieces of data, and it can more efficiently store large amounts of it, which is perfect for all of the camera and sensor data points.