HUG THE LANES

Software Documentation

Group 8 - *Justerini Mejia, Justus Neumeister, Philip Russo, TaeSeo Um*

Photo by Ahmed ツ, @mutecevvil. *Cars on a Highway* [Photograph]. Pexels.

Table of Contents

Section 1: Introduction

1.1 Roles

Our role in this project consists of developing software that will handle a self-driving vehicle's decision-making. In essence, our task boils down to receiving information about a vehicle's current state and environment through a variety of sensors and cameras, and then deciding how the vehicle will respond. As such, we are responsible for translating the vehicle's sensor input into usable information in the software module's code, and then developing different use cases and responses to deal with this input.

While we are not responsible for deciding exactly which types of sensors, cameras, or other features go into the vehicle, we will be determining the appropriate responses to the stimuli that the vehicle provides.

1.2 Scope

The scope of the software's first release will consist of only a few basic features that are integral to the development of the rest of the software. This includes being able to properly receive and process the input that the vehicle receives from its sensors, and some simple yet critical features that can be made possible using this information - such as lane-keeping, cruise control, and collision detection. Once these features are implemented, it will be possible to release further increments in the development like parallel parking, obstacle avoidance, traffic-light reading, and full self-driving.

1.2.1 Features

Software that enables a vehicle to fully drive without the need for human assistance requires a set of features that takes in every possible scenario that a vehicle might run into while driving. However, as mentioned above, our software relies a lot on the type of information that can be retrieved from the variety of sensors and cameras on the vehicle. The list of features that the customers will be able to benefit from the software can be divided into three very basic ways.

The first set of features is already present in a lot of vehicles on the market:

- Adaptive cruise control
- Lane-keeping assistance
- Collision detection

The second set of features is far more advanced:

- Self-parking
- Traffic light detection

The third set of features is a complete package of the aforementioned qualities where the vehicle is fully capable of self-driving:

- Navigation from point A to point B without human interaction

1.2.2 Availability and Reliability

The error margin for the software must be aimed to be zero. This software will eventually allow vehicle to perform a large number of calculations in real-time as it is moving around. If anything goes wrong, there is a possibility that the vehicle and the people inside (and outside) will get hurt. The availability of such software will certainly help a lot of people during their drive, but increasing its reliability to complete the drive without consequence is going to be the focus of this software. This means a lot of testing will need to be done, and a lot of simulations in order to ensure that the customers are satisfied with the final product for the entire duration they use it.

1.2.3 Architecture: IoT

The Internet of Things, or IoT, is crucial for the software's functionality and success. The sensors present will take in data on the vehicle's current condition and surrounding environment, which will then feed into our software. Our software will process the data and make decisions accordingly. These decisions will then be sent to the Vehicle's Control System, which will physically make the adjustments. The IoT will also be used to inform the driver of the vehicle's status:

- Gauges (such as gas and speed)
- Maintenance issues
- Gear selected

It will also allow the customer to input commands into the system such as:

- Shutting down and turning on various features
- Detailed maintenance information

Software updates will automatically be received during periods of extended downtime, such as over one hour, or when the customer specifies. The vehicle's status will also be uploaded to the company cloud every hour.

1.3 The Team

Courses Taken												
Member	101	115	135	284	306	334	382	385	392	396	522	559
Justus Neumeister	☑	☑	☑	☑	☐	☑	☑	☑	☑	☑	☑	☑
TaeSeo Um	☑	☑	☑	☑	☐	☑	☑	☑	☑	☑	☑	☑
Justerini Mejia	☑	☑	☑	☑	☐	☐	☑	☑	☑	☑	☐	☐
Philip Russo	☑	☑	☑	☑	☑	☐	☐	☐	☐	☐	☐	☐

Languages Known					
Member	C#	C	C++	Java	Python
Justus Neumeister	☑	☑	☑	☑	☑
TaeSeo Um	☑	☑	☑	☑	☑
Justerini Mejia	☐	☑	☑	☐	☐
Philip Russo	☐	☐	☐	☑	☑

Qualifications:

Justus Neumeister

- Coding
- Problem Solving
- Patient

Philip Russo

- Reserved
- Detail-Oriented
- Patient

TaeSeo Um

- Task-Oriented
- Project Management
- Analysis

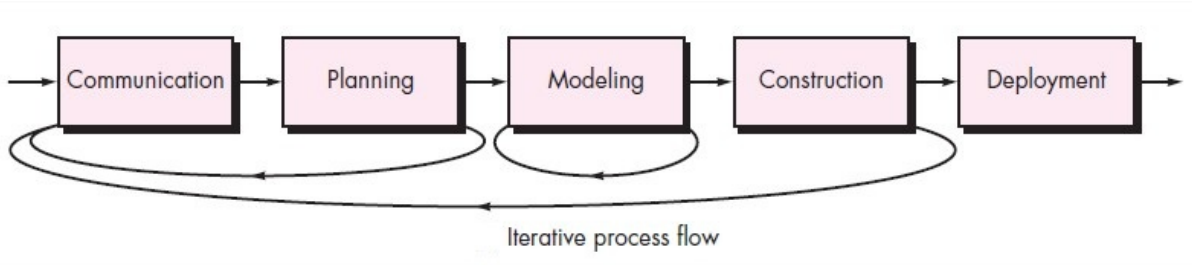
Justerini Mejia

- Writing
- Problem Solving
- Analysis

Forming a good team requires having people that differ from each other. It avoids the recycling of ideas and instead encourages the sharing of unique ones to make more effective solutions. The varying skill levels that are present allow *every* member of the team to learn from one another to enhance the team's collective problem-solving ability.

1.3.1 Development Process

We will be following an iterative process for the development of this software.



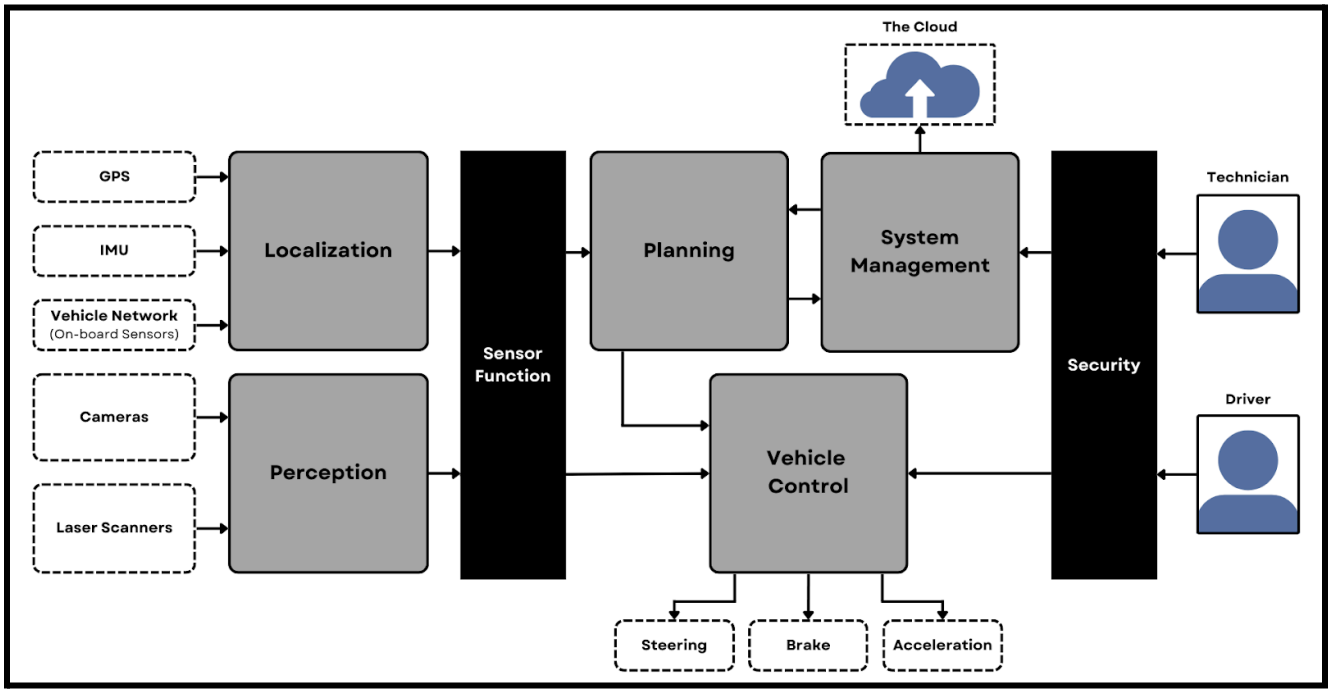
Iterative process model (Arumugam 2017)

As we are an inexperienced team we picked a simple model to avoid complications. It was decided that the development of this software needed a certain degree of flexibility, therefore the more linear processes, such as the waterfall process, were discarded. Our team believed that this process would work best for us.

1.3.2 Discipline During Documentation

Maintaining proper discipline when documenting allows both us—the developers—and the customers to track progress. Knowing exactly what steps we take to achieve our goal will let us backtrack in order to make dealing with any issues easier. As such, we promise to ensure that this document will be clear and concise, with a high level of consistency throughout the development of this software.

Section 2: Functional Architecture



The logical decomposition of the system into components and subcomponents, as well as the data-flows between them.

Functional Architecture, unmodified version by Guo (2020)

2.1 Perception

The Perception module is responsible for perceiving the environment around the vehicle. This module receives all sensor data pertaining to the environment around the vehicle. These sensors include:

- Camera for RGB imaging
- LiDAR for 3D input
- RADAR for object detection

They enable the vehicle to track:

- Moving objects
- Road condition
- Weather
- Current trajectory
- Various other visual cues (signage, traffic lights)

This module has no direct interface, as all of the information is transmitted directly to the Sensor Fusion module (2.3).

2.2 Localization

The Localization module is responsible for understanding the vehicle's current location and condition. This module receives wireless data and all sensor data pertaining to the vehicle's condition. These sensors include:

- The IMU for angular rate and specific force
- The vehicle network for understanding the state of the vehicle's systems.

This module has no direct interface, as all of the information is transmitted directly to the Sensor Fusion module (2.3).

2.3 Sensor Fusion

The Sensor Fusion module takes the information received by the Perception (2.1) and Localization (2.2) modules, and standardizes it into a uniform and constant-time format that will be much easier to interpret when passed along to the Planning module (2.4) and Vehicle Control module (2.5). Since many different types of sensors are used in the Perception and Localization modules, from possibly several different manufacturers, these sensors often relay their information in inconsistent intervals of time. Thus, the Sensor Fusion module is necessary in order to store this information for fixed intervals of time and relay it to the Planning and Vehicle Control modules for a more reliable interpretation. For example, the IMU sensor in the Perception module may send the vehicle's inertial information every 5ms, while the linear scanners in the Localization module may send their information every 3ms, so in order to make sure the Planning and Vehicle Control modules are able to make accurate decisions based on all aspects of the vehicle at the time, the Sensor Fusion module is necessary to facilitate the transfer of this information.

The figure on the next page shows the different types of sensors.

Types of Sensors

These sensors collect the information that dictates the actions that the vehicle will take.

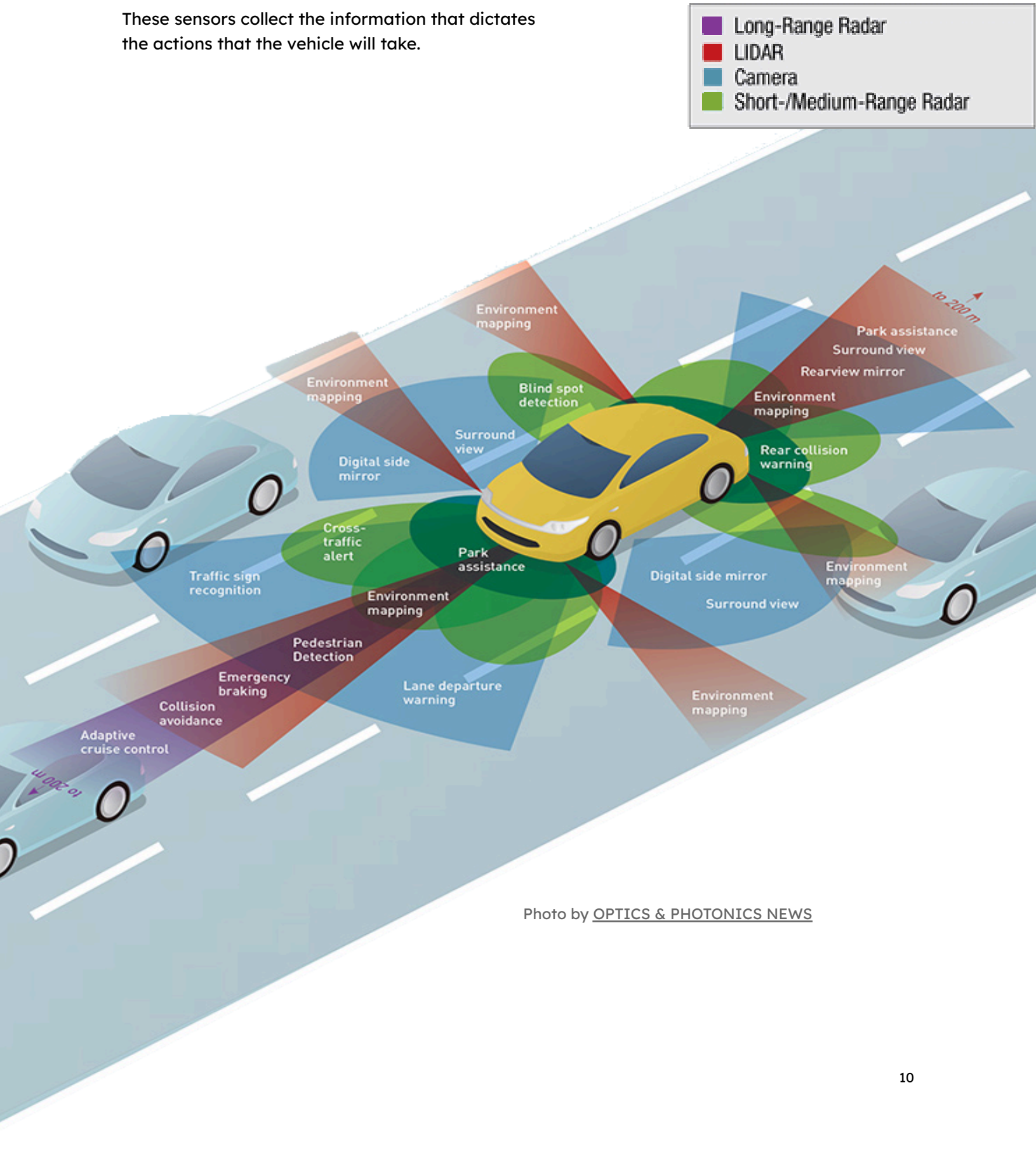


Photo by OPTICS & PHOTONICS NEWS

2.4 Planning

The Planning module is responsible for receiving information from the Sensor Fusion module (2.3) about where the vehicle is located, how fast it is going, what objects are nearby, etc., and making decisions about how the vehicle will respond. Once the Planning module receives this information from the Sensor Fusion module (2.3) in a standardized format, it must then perform logical evaluations using this information to formulate a decisive response that will then be passed to the Vehicle Control module (2.5) for physical execution. For example, if the Planning module receives information that an object was detected 500 feet in front of the vehicle, and the vehicle is only traveling 5 miles per hour, the Planning module may decide to tell the Vehicle Control module (2.5) to brake lightly. However, if the Sensor Fusion module (2.3) had relayed that the vehicle was instead traveling at 30 miles per hour, the Planning module would tell the Vehicle Control module (2.5) to brake at maximum capacity.

2.5 Vehicle Control

The Vehicle Control module is the part of the functional architecture where all the information received from the Sensor Fusion module (2.3) and Planning module (2.4) is used to physically control the vehicle. The Vehicle Control module takes the planned course of action provided by the Planning module and executes it in real-time. The calculations that this module performs is when it should start braking, when it should start accelerating and by how much to reach the desired velocity, and also when it should move the steering wheel and also by how much to turn the vehicle. That is what the movement the Vehicle Control module is responsible for can be divided into two big categories; latitudinal (controls the steering) and longitudinal (braking and acceleration). Also, this module should also account for sudden reactions such as collision detection so it can immediately apply the brakes and/or steer away, where information is being provided by the Sensor Fusion module. In the case of where driver decides to turn off any self-driving functionality, then the only input that the vehicle control module needs is the driver physically steering the wheel, stepping on the accelerator and the brakes.

2.6 System Management

The System Management module is the central unit that makes sure that all the components are working seamlessly and properly together. It is also this module that is responsible for performing tasks that constantly checks the condition of the vehicle.
It performs tasks such as:

- checking if all the sensors are working
- manages power distribution
- manages energy efficiency
- makes sure data being inputted/outputted through the entire system is done so efficiently
- Oversees the emergency response such as sudden brakes and any fail-safe system that is in the vehicle

All of this data and controls are done in this module, and that is why the security layer is crucial to be integrated with the System Management module. The technician and the driver will need to pass the security layer if they want to see the system conditions of the vehicle to check for any errors or past performance data.

Section 3: Requirements

3.1 Functional Requirements

For the purposes of this document, it is assumed that all parts and modules of the vehicle are fully functional unless otherwise specified.

3.1.1 Front Collision Detection

Detect an object in front of the vehicle and alert the driver / apply brake to avoid collision

- Preconditions
  - The vehicle is on
- Postconditions
  - If the object is still present, the vehicle stops within 5 feet of it
  - If the object is not present still, the vehicle will not brake for this event

Requirements	
3.1.1.1	The Perception Module detects an object that is L feet away from the vehicle, and sends data (distance, size, relative velocity) about this object to the Sensor Fusion Module
3.1.1.2	The Sensor Fusion Module normalizes this data and sends it to the Planning Module
3.1.1.3	If the Planning Module determines that the detected object is traveling towards the vehicle, and the vehicle is not moving, the Planning Module alerts the driver of the incoming collision. If the object is determined to be within 50 feet, determine an evasive action based on recent data from the Sensor Fusion Module, and send this plan to the Vehicle Control System.
3.1.1.4	If the Planning Module instead determines that the vehicle is moving and traveling towards the object, send a brake request to the Vehicle Control System if the object is within 200 feet.
3.1.1.5	The Vehicle Control System will apply the brake at level 5 (maximum) if the object is within 25 feet, level 4 within 25 and 50 feet, level 3 within 50 and 100 feet, level 2 within 100 and 150 feet, and level 1 within 150 and 200 feet

3.1.2 Rear Collision Detection

Detect an object behind the vehicle and alert the driver / apply brake to avoid collision

- Preconditions
  - The vehicle is on
- Postconditions
  - If the object is still present and coming closer, the vehicle moves out of the way
  - If the object is no longer present, or the object is present but not coming closer, alert the driver

Requirements	
3.1.2.1	The Perception Module detects an object that is L feet away from the vehicle, and sends data (distance, size, relative velocity) about this object to the Sensor Fusion Module.
3.1.2.2	The Sensor Fusion Module normalizes this data and sends it to the Planning Module.
3.1.2.3	If the Planning Module determines that the detected object is traveling towards the vehicle, and the vehicle is not moving, the Planning Module alerts the driver of the incoming collision. If the object is determined to be within 50 feet, determine an evasive action based on recent data from the Sensor Fusion Module, and send this plan to the Vehicle Control System.
3.1.2.4	If the Planning Module instead determines that the vehicle is moving and traveling towards the object, send a brake request to the Vehicle Control System if the object is within 200 feet.
3.1.2.5	The Vehicle Control System will apply the brake at level 5 (maximum) if the object is within 25 feet, level 4 within 25 and 50 feet, level 3 within 50 and 100 feet, level 2 within 100 and 150 feet, and level 1 within 150 and 200 feet

3.1.3 Engage Cruise Control

The driver pushes a button and the vehicle will automatically maintain the current speed.

- Preconditions
    - The vehicle is on
    - The driver is present
    - Cruise control is not currently on
    - The vehicle is moving (speed > 0)
    - The driver does not have their foot on brake

- Postconditions
    - The vehicle will maintain the current speed, which can also be changed by the driver by press nearby buttons, until the driver presses the main button once again to disengage cruise control
    - If the driver presses the gas or brake pedal, disengage

Requirements	
3.1.3.1	The driver presses the cruise control main button, which causes the Vehicle Control System to send a message to the Planning Module that cruise control has been engaged
3.1.3.2	Information about the vehicle's current speed is sent from the Localization Module to the Sensor Fusion module, where it is normalized and passed along to the Planning Module.
3.1.3.3	Information about nearby objects is sent from the Perception Module to the Sensor Fusion module, where it is normalized and passed along to the Planning Module.
3.1.3.4	The Planning Module then uses all of this information in order to decide how to proceed. If obstacles are detected within 50 feet of the front of the vehicle, the vehicle will not accelerate until it is gone. Otherwise, if the Planning Module determines that the vehicle's speed is less than the desired value, it will send a message to the Vehicle Control System to increase the throttle. Likewise, if the speed is determined to be greater than the desired value, it will tell the Vehicle Control System to decrease the throttle.

3.1.4 Disengage Cruise Control

The driver pushes the main cruise control button while it is engaged to turn cruise control off

- Preconditions
    - The vehicle is on
    - The driver is present
    - Cruise control is currently engaged
    - The vehicle is moving (speed > 0)
- Postconditions
    - The vehicle disengages cruise control

Requirements	
3.1.4.1	The driver presses the cruise control main button, which causes the Vehicle Control System to send a message to the Planning Module that cruise control has been disengaged.
3.1.4.2	The Planning Module receives this message and stops managing the speed of the vehicle

3.1.5 Lane Correction

- Preconditions
    - The vehicle is on
    - Vehicle is moving (speed > 0)
    - Driver is not steering the vehicle
    - The vehicle is not parallel to the lane marking
- Postconditions
    - The vehicle has not crossed over the lane marking
    - The vehicle is parallel to the lane marking

Requirements	
3.1.5.1	A camera sends the angle of the lane markings on both sides to the Perception module, which then sends that information to the Sensor Fusion module.
3.1.5.2	Whether or not the vehicle is being actively steered by a driver is sent by the Vehicle Control module to the Sensor Fusion module.
3.1.5.3	The Sensor Fusion module normalizes the data and sends it to the Planning module.
3.1.5.4	If: ● The vehicle is not being actively steered. ● The vehicle is not parallel to the lane marking. Then: The module calculates the angle needed to steer for the vehicle to be parallel to the lane markings. This angle is then sent to the Vehicle Control module. Otherwise: No further action is taken.
3.1.5.5	The Vehicle Control module steers the vehicle.
3.1.5.6	A log of the time, data, and decisions made is sent by the Planning module to the System Management module.
3.1.5.7	The data received by the System Management module is logged.

3.1.6 Activating Headlights

- Preconditions
    - The vehicle is on
    - Average lux from all light sensors is less than four hundred (light < 400)
- Postconditions
    - Headlights are on

Requirements	
3.1.6.1	Multiple light sensors send their lux levels to the Perception module, which then sends that information to the Sensor Fusion module.
3.1.6.2	The Sensor Fusion module normalizes the data and sends it to the Planning module.
3.1.6.3	If: • The average lux from all light sensors is less than four hundred. Then: A command to turn off the headlights is sent to the Vehicle Control module. Otherwise: No further actions taken.
3.1.6.4	The Vehicle Control module turns on the headlights.
3.1.6.5	A log of the time, data, and decisions made is sent by the Planning module to the System Management module.
3.1.6.6	The data received by the System Management module is logged.

3.1.7 Deactivating Headlights

- Preconditions
    - The vehicle is on
    - Average lux from all light sensors is at or above four hundred and fifty (light >= 450)
- Postconditions
    - Headlights are off

Requirements	
3.1.7.1	Multiple light sensors send their lux levels to the Perception module, which then sends that information to the Sensor Fusion module.
3.1.7.2	The Sensor Fusion module normalizes the data and sends it to the Planning module.
3.1.7.3	If: • The average lux of all light sensors is at or above four hundred and fifty (450). Then: A command to turn off the headlights is sent to the Vehicle Control System. Otherwise: No further actions are taken.
3.1.7.4	The Vehicle Control module turns off the headlights.
3.1.7.5	A log of the time, data, and decisions made is sent by the Planning module to the System Management module.
3.1.7.6	The data received by the System Management module is logged.

3.1.8 Automatic Windshield Wipers

- Preconditions
    - The vehicle is on.
    - System enabled by driver.
    - Rain or snow is detected.
- Postconditions
    - If there is rain/snow, the vehicle activates the wipers.
    - If there are large amounts of rain/snow on the windshield even after activating the wipers, it will increase the wiper speed.

Requirements	
3.1.8.1	The Perception Module reads from both the onboard cameras and laser scanners that there are objects that it detects as rain/snow falling nearby and on the vehicle.
3.1.8.2	Then this fact of rain or snow falling is sent to the Planning Module where it makes a decision if there is enough rain/snow falling to activate the windshield wipers.
3.1.8.3	The decision to turn on windshield wipers is outputted to the Vehicle Control module. The Vehicle Control module activates the windshield wipers.
3.1.8.4	The Planning Module continuously receives input from the Perception Module of how much rain/snow is falling and makes a decision to either increase or decrease the level of the wiper speed. This is done continuously until there is no more rain/snow or the preconditions are not met.

3.1.9 Automatic Temperature Regulation

- Preconditions
    - The vehicle is on.
    - System enabled by driver.
    - Driver has their preferred temperature set in their user preferences of the vehicle.
- Postconditions
    - If the vehicle's interior temperature is lower than the preferred temperature, heater will automatically turn on and regulated to keep the temperature
    - If the vehicle's interior temperature is higher than the preferred temperature, air conditioning will automatically turn on and regulated to keep the temperature

Requirements	
3.1.9.1	The onboard sensors will provide the Sensor Fusion module the current temperature readings of the interior of the vehicle. This will depend on the outside temperature, but what is important is the interior temperature.
3.1.9.2	The Sensor Fusion module will provide the temperature readings to the Planning module where it will decide whether to turn the heater on or the cooler on depending on the driver's preferred temperature.
3.1.9.3	The Planning module will output its decision to the Vehicle Control module to either turn the heater on or the cooler on.
3.1.9.4	The Sensor Fusion module and Planning module will need to continuously communicate with each other in order to maintain the desired temperature because the interior temperature will change throughout the drive.

3.1.10 Traffic Light Recognition

- Preconditions

    - The vehicle is on.
    - A traffic light is detected.
- Postconditions
    - The vehicle has reacted appropriately to the corresponding light.
        - If red, then the vehicle will come to a stop and wait for a change of light.
        - If yellow, then the vehicle will slow down and wait for a change of light.
        - If green, then the vehicle will proceed through the intersection if there is no obstruction.
        - If there are any other variations of lights, then slow down and alert the driver of an unidentifiable light.

Requirements	
3.1.10.1	The Perception module will detect any incoming traffic lights and what signal is currently being displayed.
3.1.10.2	The status of the traffic light is sent to the Planning module, where it will decide which action corresponds to the current state of the light. The decision is then sent to the Vehicle Control System.
3.1.10.3	The Vehicle Control System will: • Come to a stop if the light is red. • Slow down if the light is yellow. • Proceed through the intersection if the light is green. • Slow down and alert the driver if there is an unidentified light.
3.1.10.4	This process will cycle as long as there is a traffic light in range of the car, as it anticipates any possible changes.

3.1.11 Emergency Vehicle Detection

- Preconditions

    - The vehicle is on.
    - An emergency vehicle whose siren is on is detected.
- Postconditions
    - The vehicle has alerted the driver that it has detected an approaching emergency vehicle and has safely moved to an empty lane.

Requirements	
3.1.11.1	The Perception module will detect any nearby emergency vehicles and determine if it is “active”. An emergency vehicle is considered to be an ambulance, a police car, or a fire truck. A vehicle is “active” if it has its siren on, or is blaring its horn.
3.1.11.2	If the emergency vehicle is active, then send this to the Planning module. The planning module will then send this to the Vehicle Control System to display as a warning: “An Emergency Vehicle is Approaching!”
3.1.11.3	The Perception module will detect if the emergency vehicle is to the left or the right of the vehicle. 1. If it is to the left or on the oncoming side, then the car will check the lane to the right. 2. If it is on the right side, then the car will check the lane to the left. Either way, this is sent to the Planning module to determine if a collision will occur.
3.1.11.4	If the Planning module determines that there will be no collision and there is an empty lane in the corresponding scenario, then the car will move out of the way of the incoming emergency vehicle.
3.1.11.5	If the detected emergency vehicle is not active, then treat it as a regular vehicle.

3.1.12 Automatic Door Locks

- Preconditions
    - The vehicle is on

    - The doors are closed
    - The vehicle is moving at or above twelve mph (speed ≥ 12 mph)
- Postconditions
    - All doors are locked

Requirements	
3.1.12.1	The accelerometer sends the current speed to the Localization module, along with the door state, which then sends this information to the Sensor Fusion module.
3.1.12.2	The Sensor Fusion module normalizes the data and sends it to the Planning module.
3.1.12.3	If: • The doors are closed • The vehicle is moving at or above twelve mph. Then: A command to lock the doors is sent to the Vehicle Control module. Otherwise: No further action is taken.
3.1.12.4	The Vehicle Control Module locks the doors.
3.1.12.5	A log of the time, data, and decisions made is sent by the Planning module to the System Management module.
3.1.12.7	The data received by the System Management module is logged.

3.2 Non-Functional Requirements

3.2.1 Performance Time

Requirements

3.2.1.1	For any decision, the system must analyze the input and produce an output faster than the average human reaction time of 0.15 seconds.
3.2.1.2	This requirement will be tested by looking at the difference in time between input received and action taken, as recorded in the system log.
3.2.1.3	For every functional requirement the test will be repeated 1000 times, in order to ensure the consistency of the performance.

3.2.2 Reliability

Requirements	
3.2.2.1	The Planning Modules should come to the same conclusion across all vehicles given they receive the same data from the Sensor Fusion Module - i.e. two different vehicles should not react differently to the same scenario.
3.2.2.2	The driver should always retain the ability to override any autonomous functionality in case of emergencies or errors.
3.2.2.3	No features or functions listed should experience more than 1 error in 200,000 hours or operation.
3.2.2.4	The Sensor Fusion Module should update the Planning Module with the fully-detailed state of the vehicle at fixed intervals of time

3.2.3 Security

Requirements	
3.2.3.1	The Security Module should block any external attempts to activate the Vehicle Control Module through wireless communication such as Wifi, Bluetooth, etc.

3.2.3.2	The Security Module should only allow Technicians with the correct authorization to gain access to access the data of the vehicle.
3.2.3.3	The Security Module should only allow Technicians with the correct authorization to make changes to system software.
3.2.3.4	The Security Module should only accept the correct 'key' to communicate with the Vehicle Control module to lock/unlock the vehicle.
3.2.3.5	Any data communication between the Vehicle Control module and the Planning module should be encrypted so ensure it is not vulnerable to possible attacks.

3.2.4 User Interface

Requirements	
3.2.4.1	The Driver should be able to easily control the vehicle's Autonomous and non-Autonomous features that communicates with the Vehicle Control module and
3.2.4.2	The Driver should be able to see the current status of the vehicle - speed, RPM (revolutions per minute), fuel, headlight, wipers, gear, oil temperature, etc.
3.2.4.3	The Technician should be able to see every data of the vehicle from the System Management module on their device once connected to the vehicle's software or in the cloud once the System Management module uploads it.

Section 4: Requirements Modeling

4.1 Use Case Scenarios

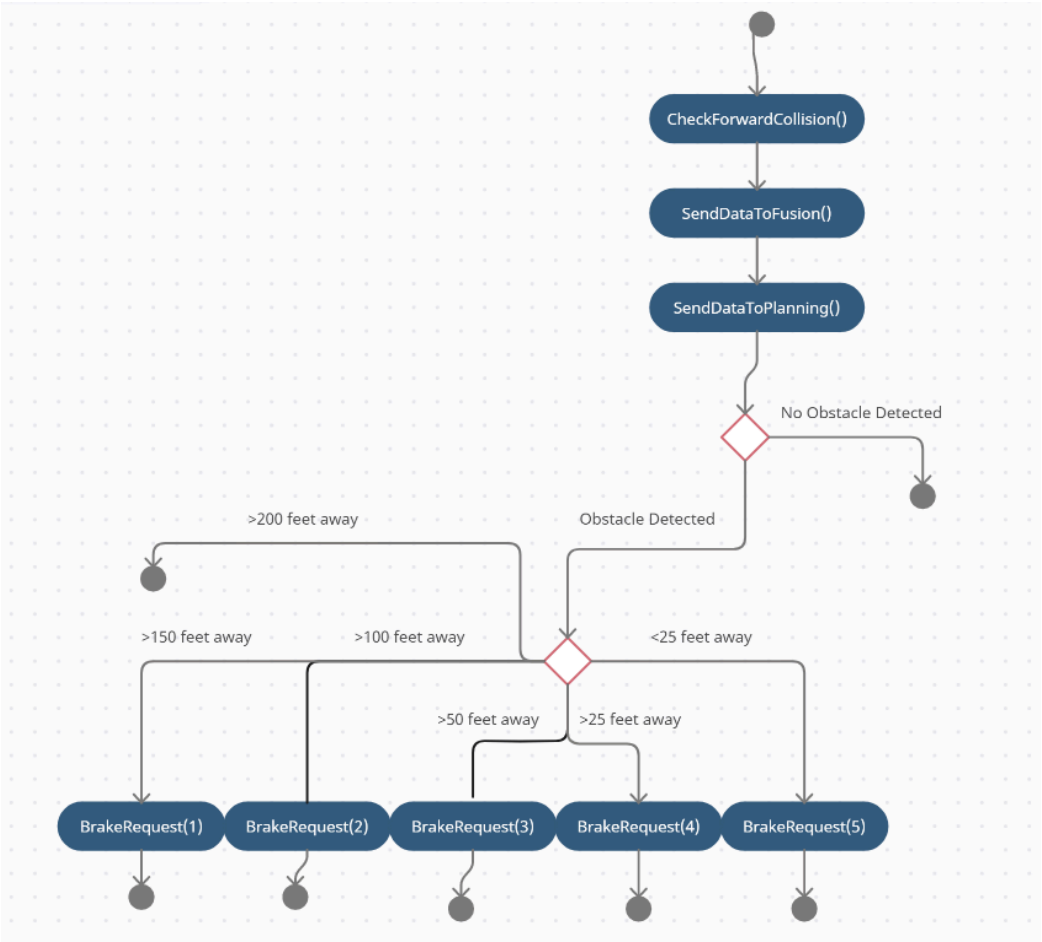
4.1.1 Collision Detection

- Preconditions

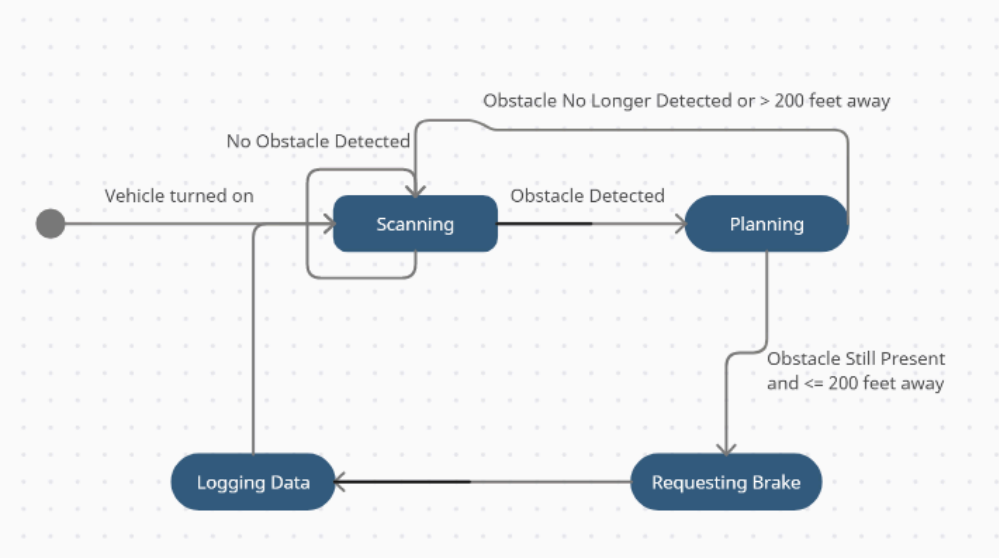
    - The vehicle is on
    - The vehicle is moving
- Postconditions
    - If the obstacle is still present, the vehicle stops at least 5 feet away.
    - If the obstacle is no longer present, the vehicle does not brake for this event.
- Trigger
    - An object is detected within 200 feet in front of the vehicle

Steps	
4.1.1.1	The forward collision sensor detects an object in front of the car, and sends its collected data about the size and distance of the object to Sensor Fusion.
4.1.1.2	Sensor Fusion normalizes the size and distance data that it receives, and sends this to Planning.
4.1.1.3	Planning sends a brake request to VCS at Brake level 5 (max) if the obstacle is <= 25 feet away, level 4 <= 50 feet away, level 3 <= 150 feet away, or level 2 <=200 feet away.

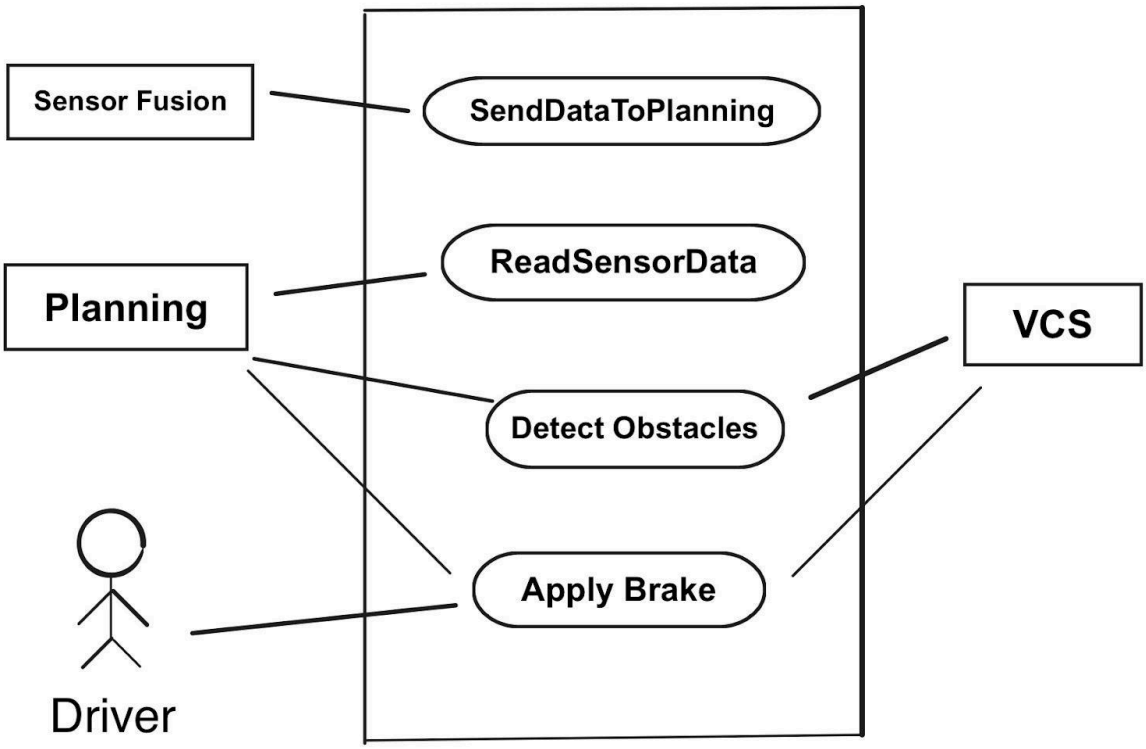
Activity Diagram



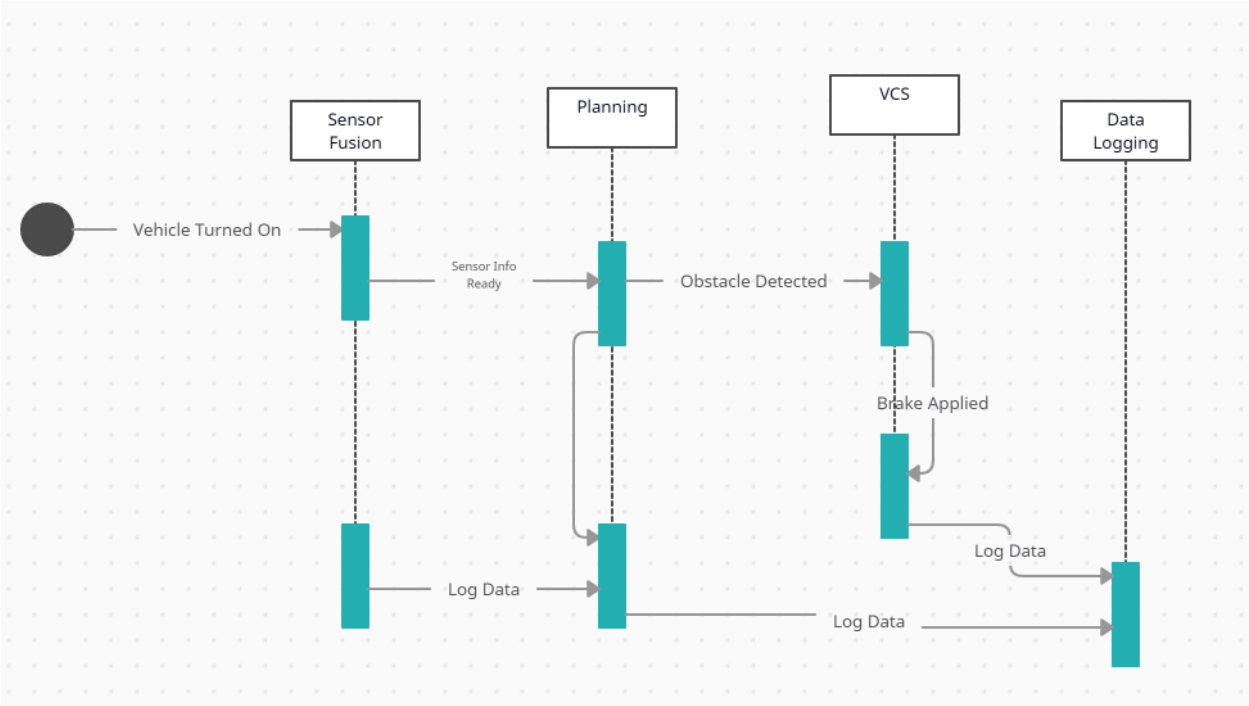
State Diagram



Use Case Diagram



Sequence Diagram



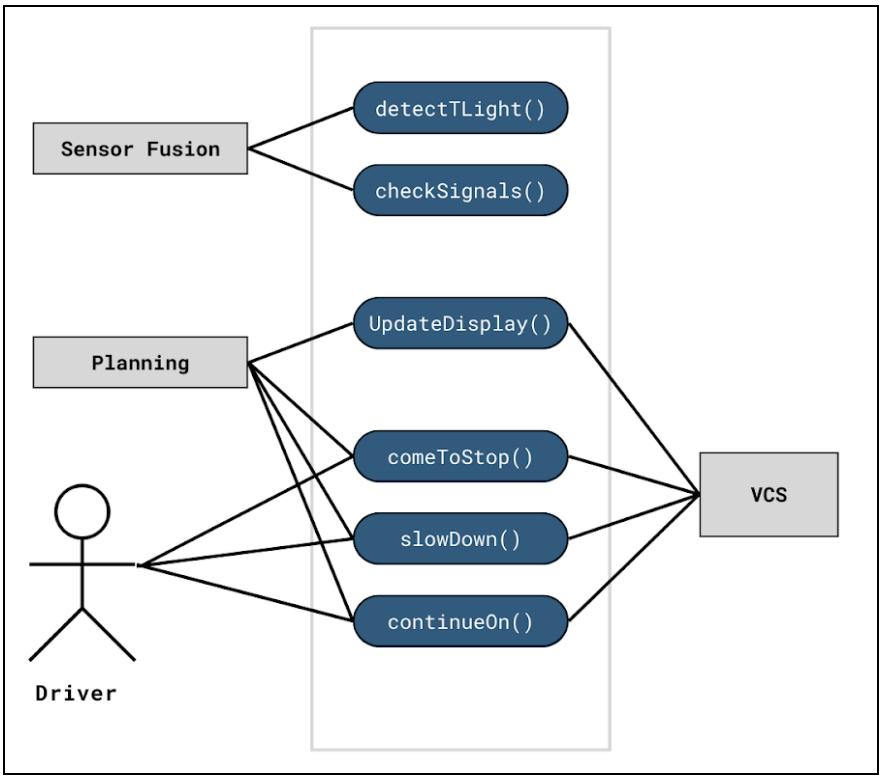
4.1.2 Traffic Light Recognition

- Preconditions

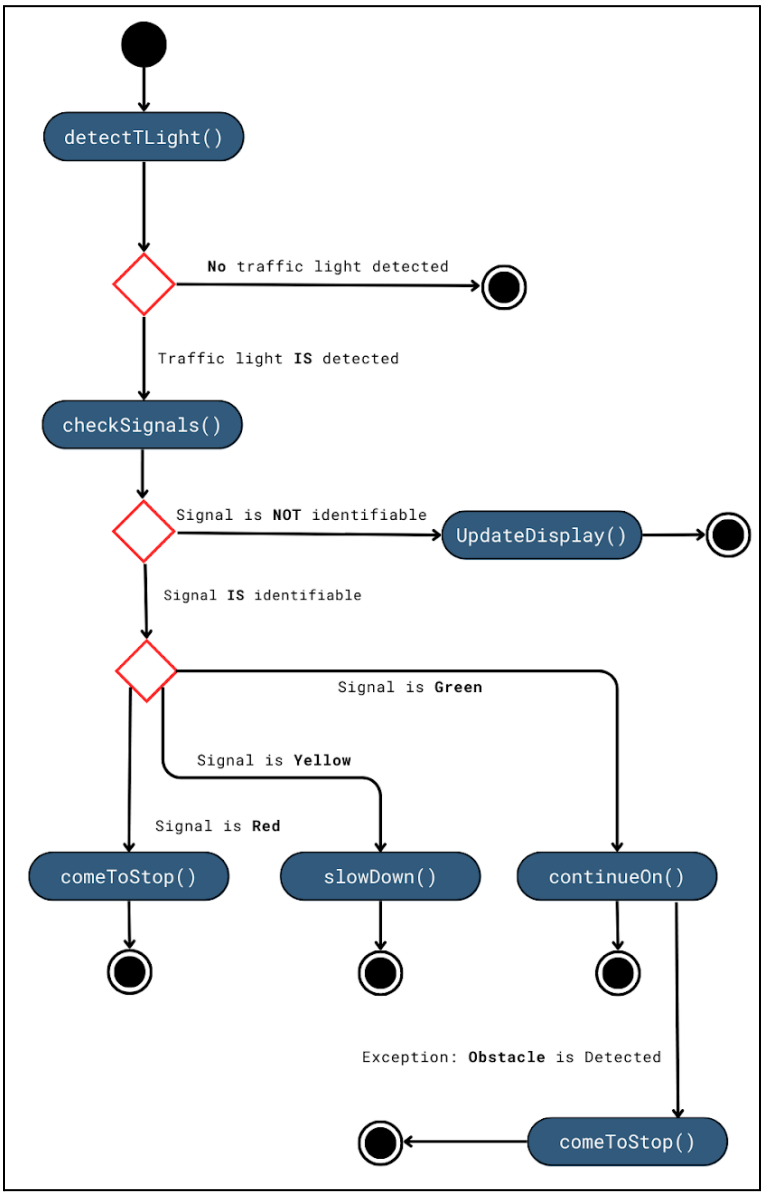
    - The vehicle is on
- Postconditions
    - The vehicle has reacted appropriately to the corresponding light.
- Trigger
    - A traffic light is detected
    - The signal is identifiable

Steps	
4.1.2.1	A traffic light is detected.
4.1.2.2	The sensor detects what type of signal is being displayed. An identifiable signal is either: Red, Yellow, or Green.
Exception	If the signal displayed is unidentifiable, send a warning to the user through the Display module.
4.1.2.3	If the signal is red: The car will come to a stop.
4.1.2.4	If the signal is yellow: The car will slow down.
4.1.2.5	If the signal is green: The car will proceed through the intersection. **Exception:** If there is an obstruction detected, then the car will come to a stop.

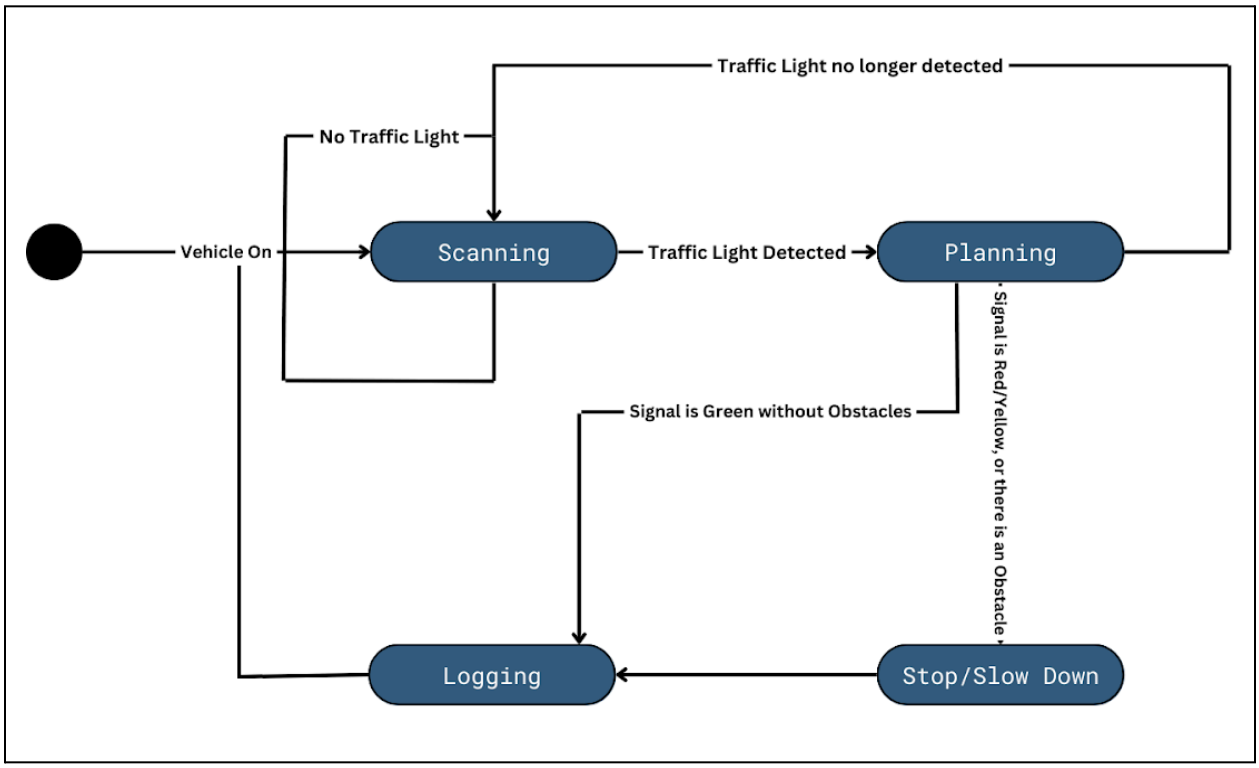
Use Case


Sensor Fusion
detectTLight()
checkSignals()
Planning
UpdateDisplay()
comeToStop()
slowDown()
continueOn()
VCS
Driver

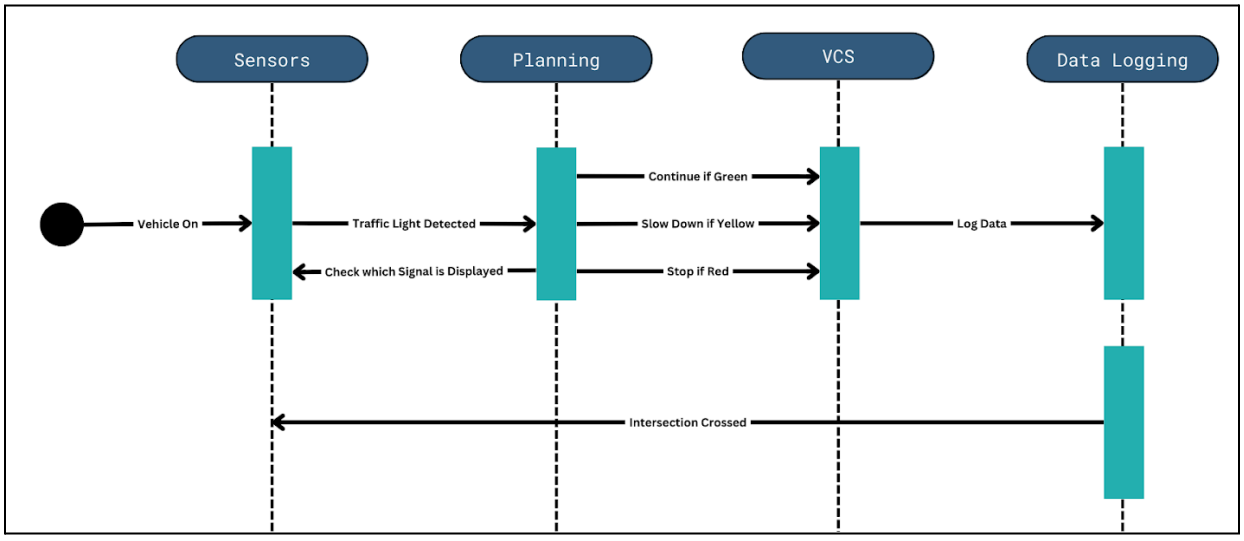
Activity Diagram



State Diagram



Sequence Diagram



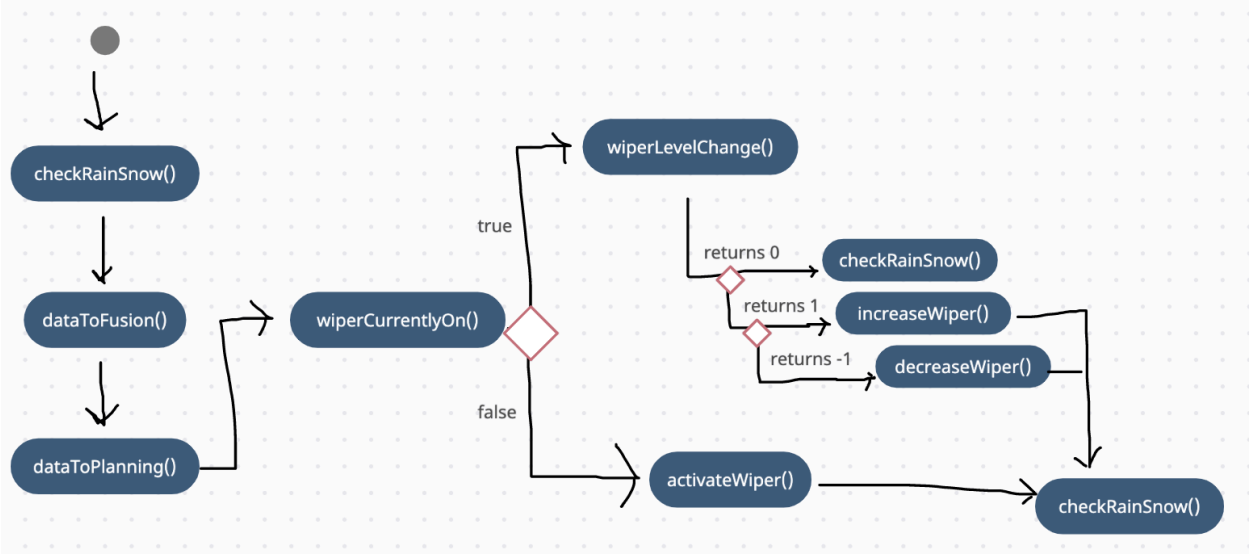
4.1.3 Automatic Windshield Wipers

- Preconditions

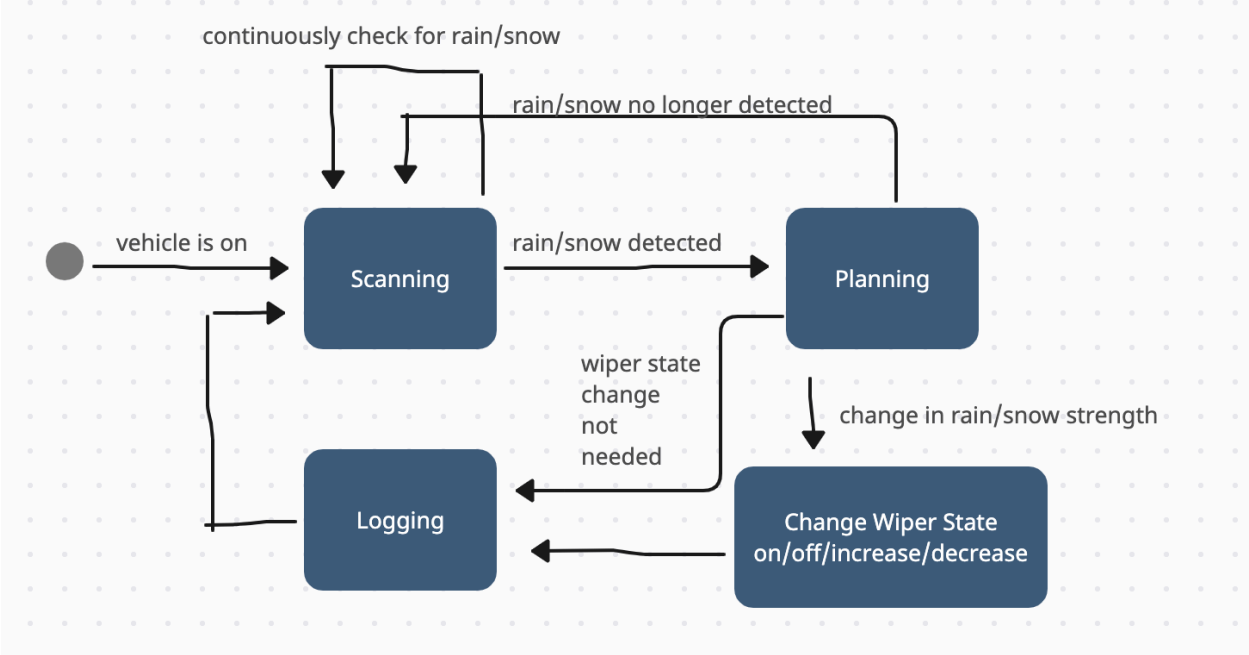
    - The vehicle is on
    - The driver is present
    - Automatic windshield system is activated by the driver
    - It is currently snowing and/or raining outside the vehicle
- Postconditions
    - If there is rain/snow, the vehicle activates the wipers.
    - If there are large amounts of rain/snow on the windshield even after activating the wipers, it will increase the wiper speed.
- Trigger
    - Rain and/or snow is detected
    -

Steps	
4.1.3.1	Rain and/or snow is detected by Cameras and Laser Scanners
4.1.3.2	The information is then sent to the Perception module where it is then transferred to the Sensor Fusion
4.1.3.3	The Sensor Fusion sends the information to the Planning module
4.1.3.4	The Planning module retrieves information on the current state of the windshield wipers, whether it is already activated
If not currently activated	**4.1.3.4** The Planning module will tell the Vehicle Control module to turn it on **4.1.3.5** Vehicle Control module will activate wipers to level 1 **4.1.3.6** The wipers will remain activated in level 1 until further notice
If it is currently activated	**4.1.3.7** Cameras and Laser Scanners detects different strength of rain/snow **4.1.3.8** The information is sent to the Perception module, then the Sensor Fusion **4.1.3.9** The Sensor Fusion sends the information to the Planning module **4.1.3.10** Planning module tells Vehicle Control module to increase/decrease wiper level **4.1.3.11** The wipers level changes based on the decision of the Planning module
Exception	If the automatic wiper feature is not on, then the display shows that the feature is off
Exception	If the user manually increases/decreases the level of the wiper, the automatic wiper feature turns off
4.1.3.12	Show the current wiper status on the display on the dashboard

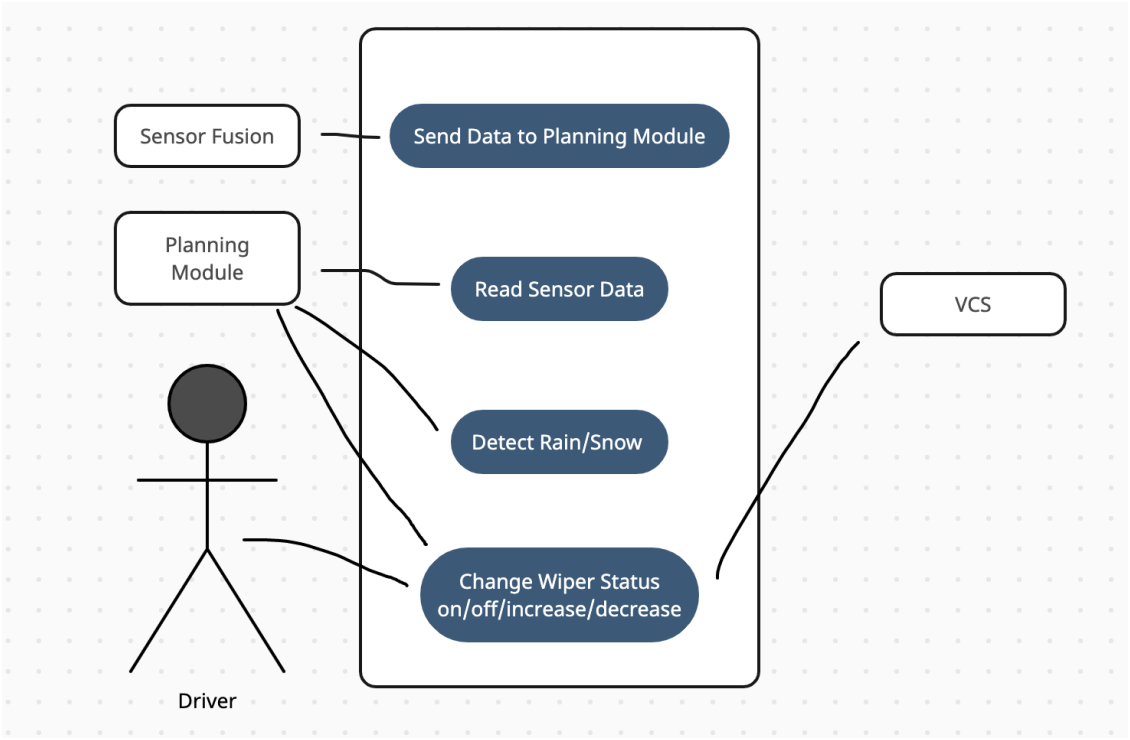
Activity Diagram



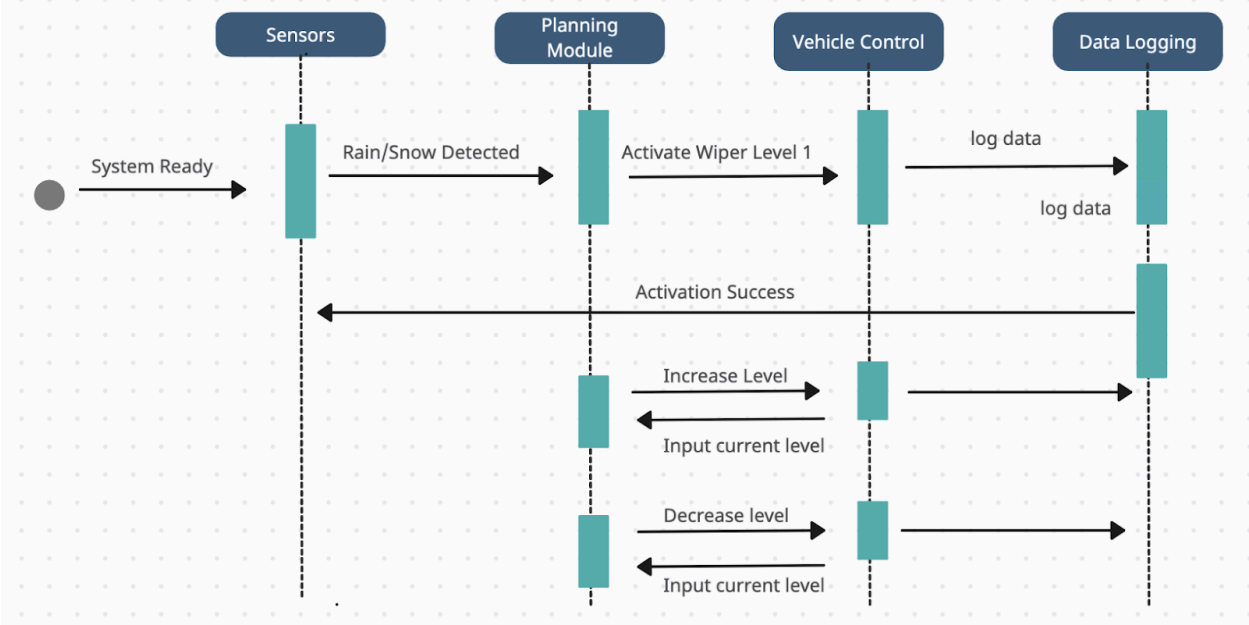
State Diagram



Use Case Diagram



Sequence Diagram



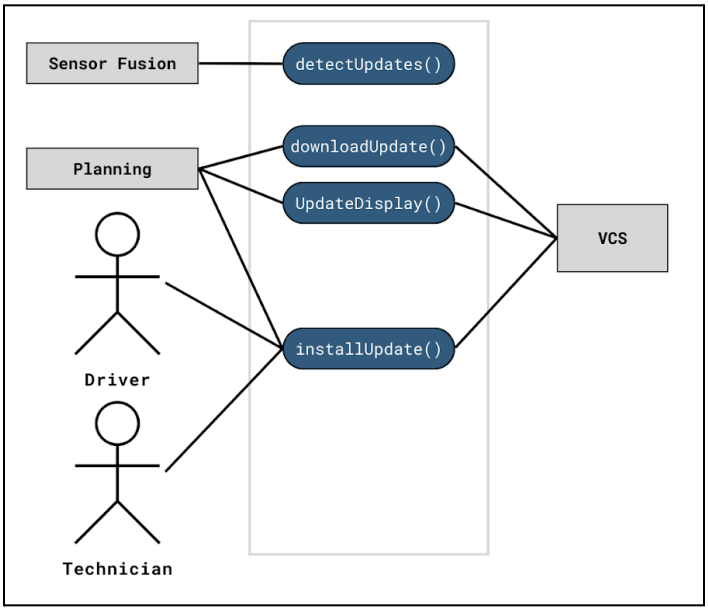
4.1.4 Software Update

- Preconditions

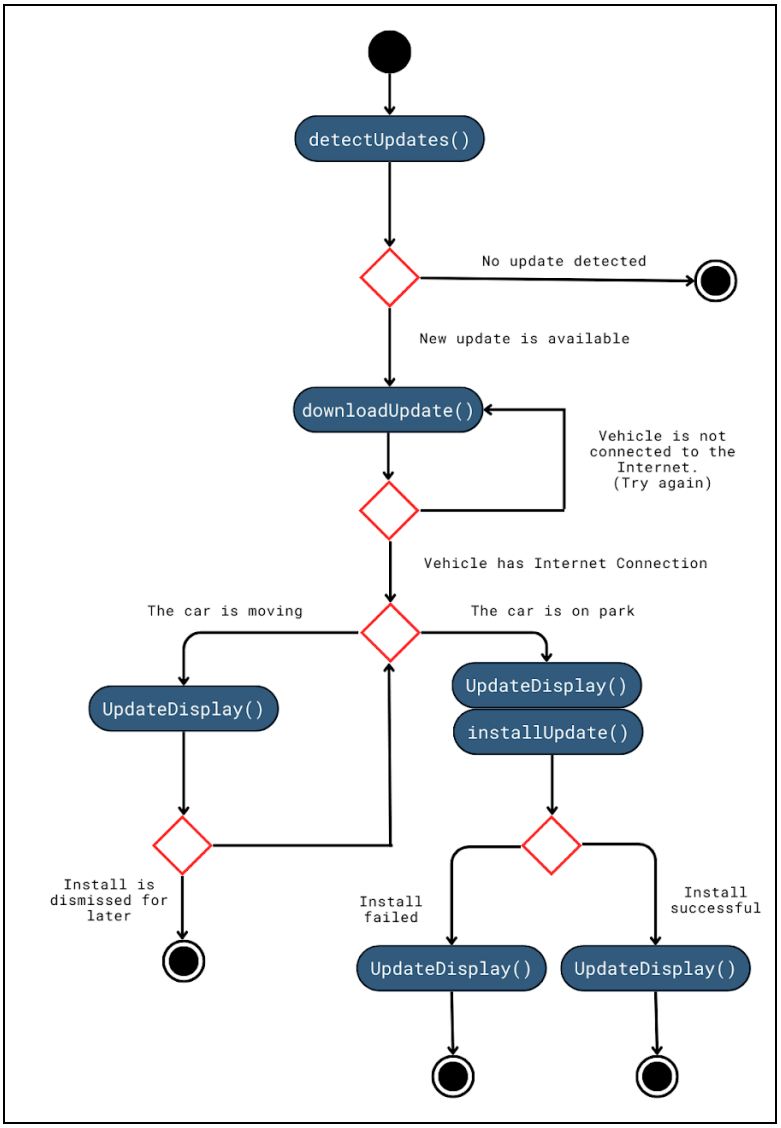
    - The vehicle is on.
    - The vehicle's software is out of date.
- Postconditions
    - The vehicle's software has been updated, or ignored.
- Trigger
    - A new software update is available on startup.

Steps	
4.1.4.1	A new software update is available.
4.1.4.2	Download the update.
Exception	If there is no internet connection, pause the download.
4.1.4.3	If the car is on park, start the installation and show the progress on the display.
Exception	If the car is moving, prompt the driver to start installation now or dismiss.
4.1.4.4	If the installation fails, tell the driver that it must be brought to a technician on the Display. If the installation is successful, tell the driver that the update is complete on the Display.

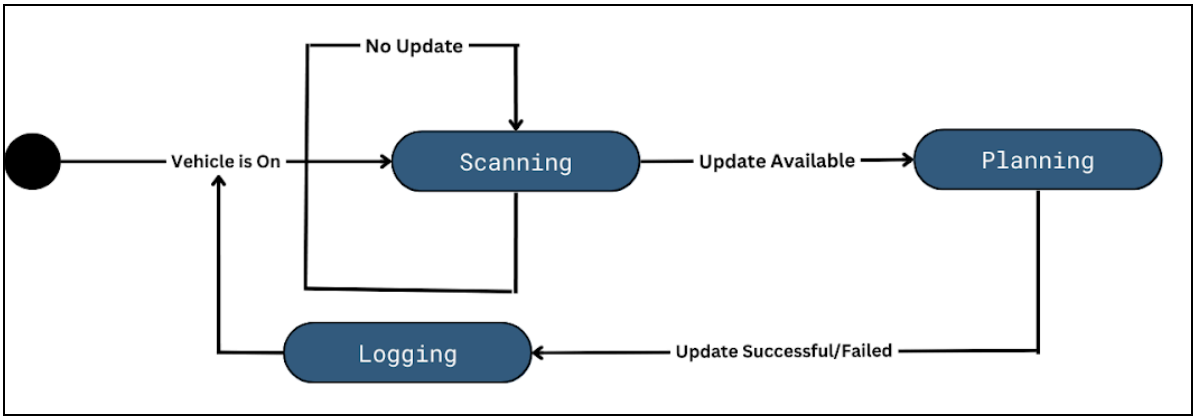
Use Case



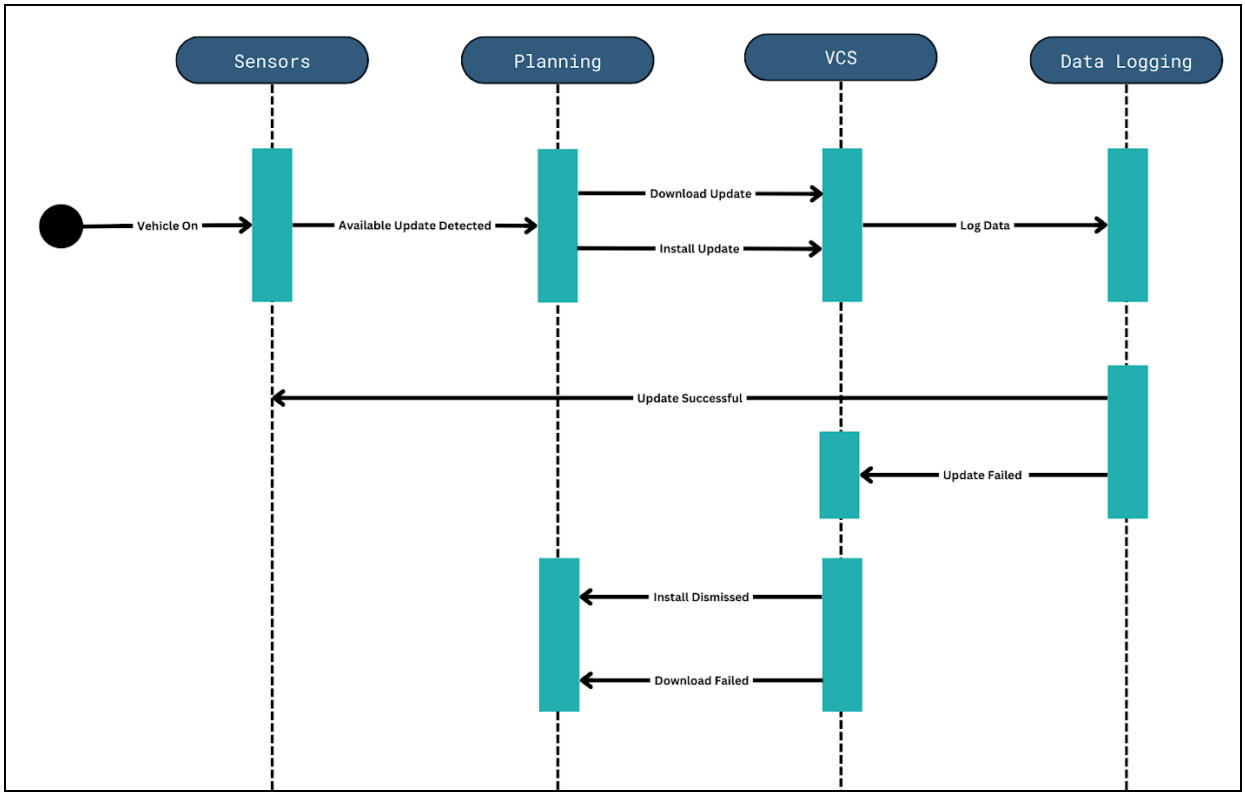
Activity Diagram



State Diagram



Sequence Diagram



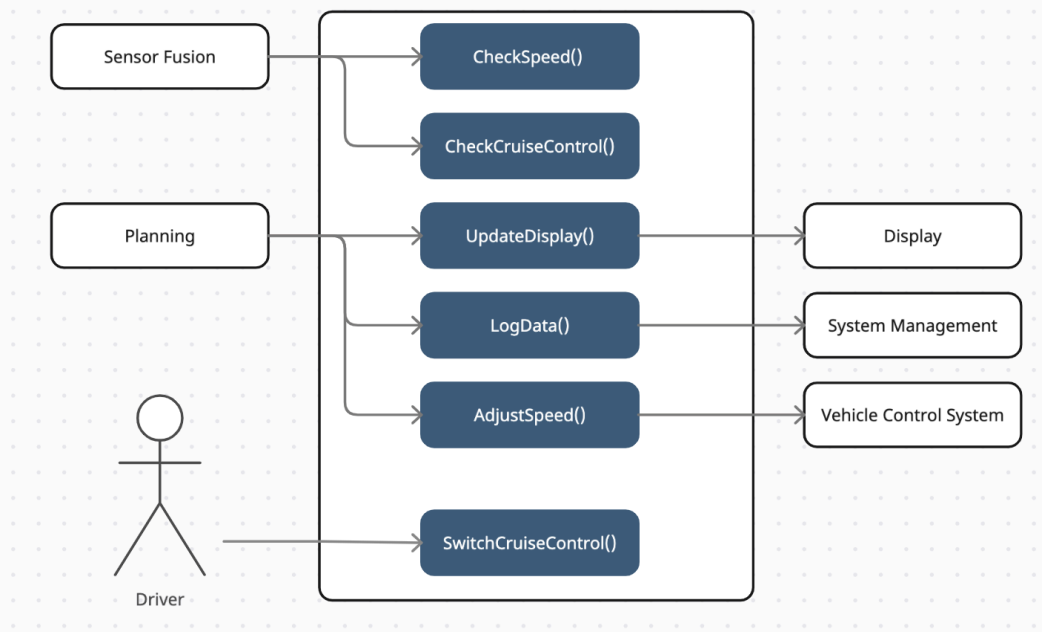
4.1.5 Engaging Cruise Control

- Preconditions

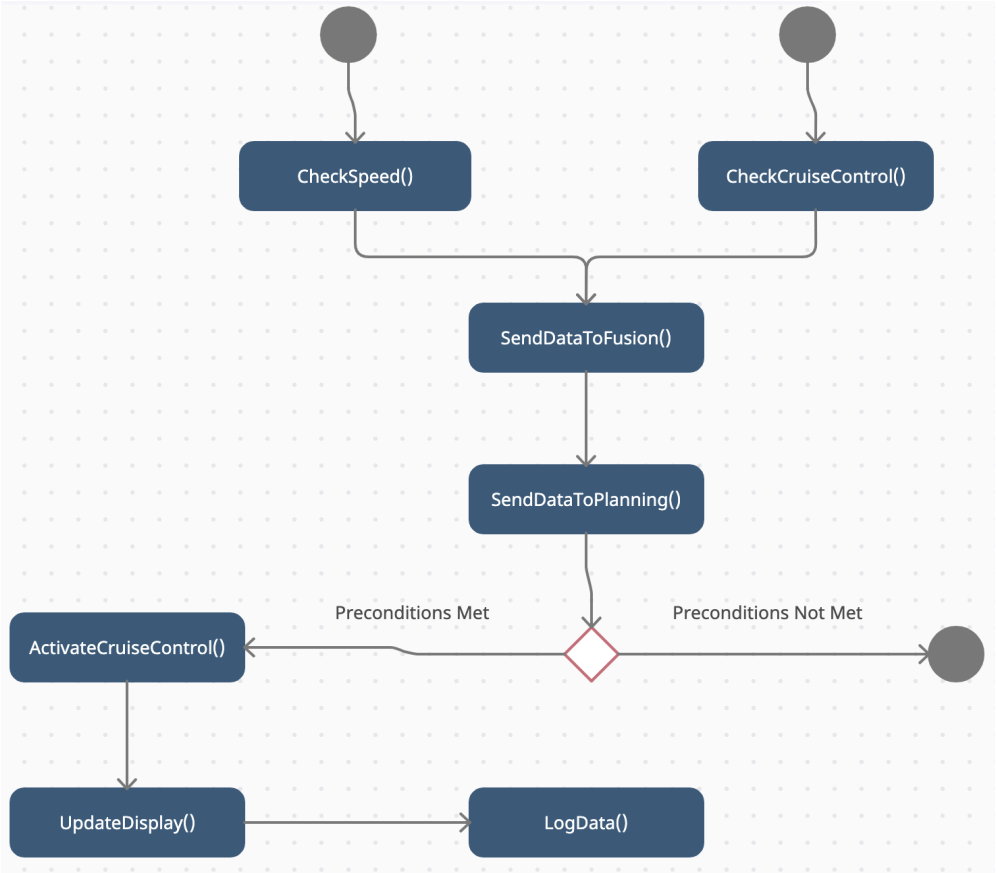
    - The vehicle is on
    - The driver is present
    - Cruise control is not currently on
    - The vehicle is moving (speed > 0)
    - The driver does not have their foot on the brake or gas
- Postconditions
    - The vehicle will maintain the current speed
- Trigger
    - The driver presses the cruise control button

Steps	
4.1.5.1	The driver presses the cruise control button.
4.1.5.2	Cruise control, gas pedal, and brake pedal send their state to the Localization module.
4.1.5.3	The accelerometer sends the vehicle's current speed to the Localization module.
4.1.5.4	The Localization module sends its data to the Sensor Fusion module.
4.1.5.5	The Sensor Fusion module normalizes the data and sends it to the Planning module.
4.1.5.6	The Planning module determines if the preconditions are met.
Exception	Preconditions are not met, the Planning module tells the Display module to display that cruise control cannot be activated. (Go to 4.1.5.8).
4.1.5.7	The Planning module activates cruise control, keeping the vehicle's speed consistent.
4.1.5.8	The Planning module tells the Display module to display that cruise control is active and the set speed.
4.1.5.9	A log of the time, data, and decisions made is sent by the Planning module to the System Management module.
4.1.5.10	The data received by the System Management module is logged.

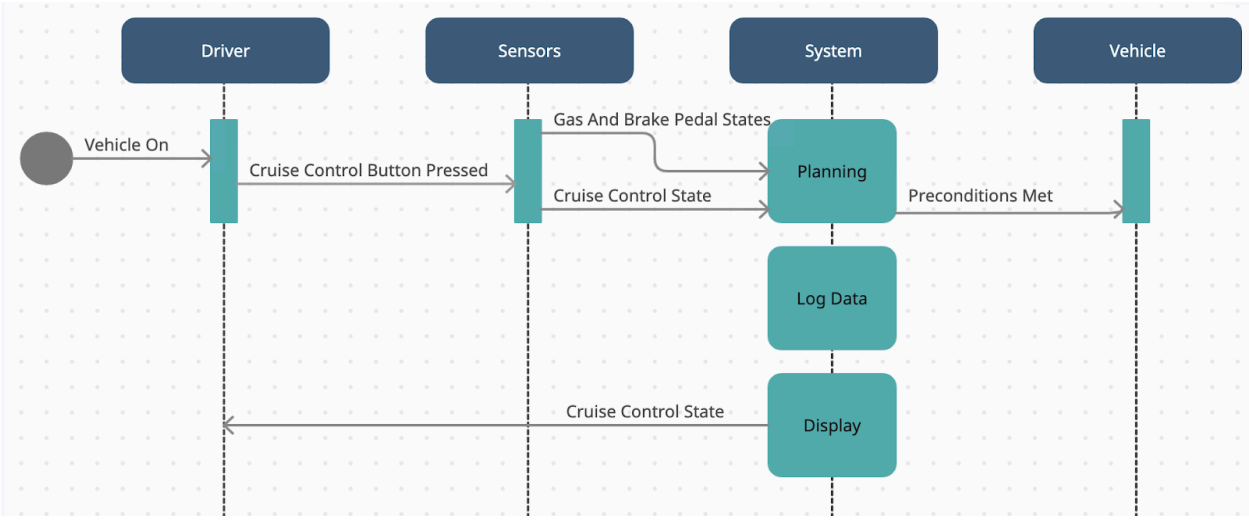
Use Case Diagram



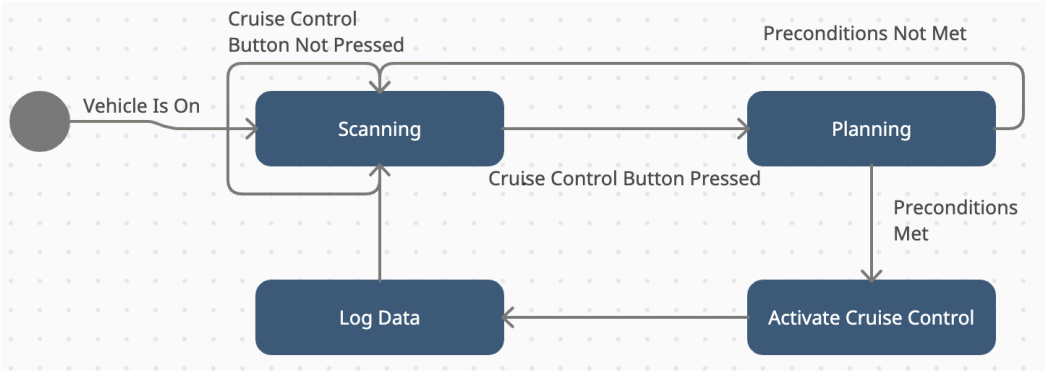
Activity Diagram



Sequence Diagram



State Diagram



4.1.6 Class Responsibility Collaborator (CRC) Cards

4.1.6.1

Class: CollisionDetection	
Description: The vehicle will detect anything in this trajectory within 200 feet. The vehicle will slow down, continue, or come to a stop.	
Responsibilities	Collaborator
Check forward collision	Sensors (sensor fusion)
Slow down	Sensors (sensor fusion), Planning Module, Vehicle Control Module
Continue moving forward	Sensors (sensor fusion), Planning Module, Vehicle Control Module
Come to a complete stop within 5 feet of the object	Sensors (sensor fusion), Planning Module, Vehicle Control Module

4.1.6.2

Class: TrafficLightDetection	
Description: The vehicle will detect the current traffic light status, and will also recognize any traffic signs. The vehicle will react correspondingly.	
Responsibilities	Collaborator
Detect Traffic Light	Sensors (sensor fusion), Planning Module
Check Signals	Sensors (sensor fusion), Planning Module
Come to a stop	Sensors (sensor fusion), Planning Module, Vehicle Control Module
Slow vehicle down	Sensors (sensor fusion), Planning Module, Vehicle Control Module
Continue moving	Sensors (sensor fusion), Planning Module, Vehicle Control Module
Update Display	Display

4.1.6.3

Class: AutomaticWipers

Description: Windshield wipers will be activated when the sensors detect rain and/or snow. The wipers' level will be adjusted with constant communication with the sensors and the different modules.	
Responsibilities	Collaborator
Activate wipers	Sensors (sensor fusion), Planning Module
Increase wiper level	Sensors (sensor fusion), Planning Module, Vehicle Control Module
Decrease wiper level	Sensors (sensor fusion), Planning Module, Vehicle Control Module
Deactivate wipers (decrease until level 0)	Sensors (sensor fusion), Planning Module, Vehicle Control Module

4.1.6.4

Class: SoftwareUpdate	
Description: The vehicle will update its software if there is a software update available upon startup. The user can either update the software or ignore the update.	
Responsibilities	Collaborator
Detect Update	Internet Module (hardware)
Download Update	System Management Module
Update Display	Display

4.1.6.5

Class: EngageCruiseControl	
Description: Vehicle will remain at the current speed	
Responsibilities	Collaborator
Maintain current velocity of the vehicle	Accelerometer
Update the display	Display

Section 5: Design

5.1 Software Architecture

In this section we will consider multiple design options for our software architecture.

5.1.1 Data Centered

Feasibility: This architecture is suitable for IoT (Internet of Things) HTL (Home, Transportation, and Logistics) since it is an architecture that revolves around a central data repository that is capable of managing a high volume of data generated by multiple devices.	
Pros	**Cons**
Efficient data management	Potential high data traffic issues at data repository
Secure data consistency and availability	Difficult to handle and process real-time data

5.1.2 Data Flow:

Feasibility: This architecture is suitable for IoT HTL since it is an architecture that focuses on the flow of data between the multiple processing elements. This architecture can help manage such flows effectively since IoT systems use complex data pipelines.	
Pros	**Cons**
Easy to manage complex data processing pipelines	Hard to manage and monitor the amount of complex flow of data between devices
Can handle asynchronous and parallel data processing	Difficult to design a system with low latency and protection from possible data loss

5.1.3 Call Return:

Feasibility: This architecture is not suitable for IoT HTL since it is an architecture that does not support the distribution and asynchronous nature of how IoT systems work with multiple devices. Using this type of architecture will make the IoT system very difficult to scale and manage with more devices and data.	
Pros	**Cons**
Simple and easy to use for a small centralized system	Does not support asynchronous nature of IoT systems
	Cannot handle real-time data processing

5.1.4 Object Oriented:

Feasibility: This architecture is suitable for IoT HTL since it is an architecture that promotes modular design, encapsulation, and the reuse of code. A modular design and having reusable code leads to a more manageable system, so it would be much easier to maintain and scale the system.	
Pros	**Cons**
The modularity of this model makes it easier to handle the complex nature of IoT systems	Challenging to learn and code a well structured object oriented system
System is easily to maintain	Requires more planning before coding

5.1.5 Layered:

Feasibility: This architecture is suitable for IoT HTL since it is an architecture that separates the concerns and functionalities into individual layers. The separation would better allow for scaling, maintaining, and developing the entire IoT system. The layers could be for data management, processing, storage, etc.	
Pros	**Cons**
System is easy to scale and maintain	Performance overhead due to the communication between multiple layers
Each layer's development is independent	Too many layers can lead to a complex system

5.1.6 Model View Controller:

Feasibility: This architecture is suitable for IoT HTL since it is an architecture that heavily involves the users' interactions. This architecture separates the system's data, user interface, and the controller to increase maintainability and code modularity.	
Pros	**Cons**
Separation promotes modularity and maintainability	It is too focused in user interactions
Can easily develop and evolve the user interface, systems, and controller individually	Requires a good design to separate the three components

5.1.7 Finite State Machine:

Feasibility: This architecture is suitable for IoT HTL since it is an architecture that models the behavior of individual devices in the IoT system, given that the devices have a limited number of states and possible transitions. However, managing and maintaining this system would become difficult as more devices are connected to the system.	
Pros	**Cons**
Good for modeling a system with small amount of devices with limited states	This architecture alone is not enough to maintain an entire IoT system
Easy to predict component behavior(s)	Does not handle the asynchronous nature of IoT systems

5.1.8 Final Decision

After careful review of all of our architecture options, we have decided that an Object Oriented Architecture is the best choice for the software. It synergizes well with our selected coding language: C++. It also allows for work to be done individually in parallel.

5.2 Interface Design

5.2.1 Technician Interface

- showSystemLog()
    - Shows the technician every line of system log, showing history of both software and any interaction the car had.
- checkSensorHealth(sensorList)
    - When running this function, the sensorList can be any type of list. For example it could be Localization Sensors, Perception Sensors, etc. The

function will output the status of all the sensors in the list showing "HEALTHY" or "FAILURE"

- checkSoftwareHealth()
    - Shows if there are any issues with the software, history of software crashes or malfunction.
- installUpdate()
    - Technician can run this function to install the most up-to-date software to the vehicle
- uploadLogToServer()
    - The most recent system log will be uploaded to the server that the technician authorized

5.2.2 Driver Interface

- DisplaySpeed()
    - Continuously shows the current speed of the vehicle
- ToggleCruiseControl()
    - Engages / disengages cruise control
    - Displays current cruise control state
- ToggleAutonomousDriving()
    - Engages / disengages autonomous driving mode
    - Continuously displays current driving mode, vehicle destination
- DisplayFuel()
    - Continuously shows the amount of fuel remaining
    - Displays alerts when fuel level is low
- DisplayTemperature()
    - Continuously shows the temperature of both the inside of the vehicle as well as the outside environment
- CollisionAlert()
    - Display visual cues to inform the driver of nearby vehicles passing on either side
    - Display a collision warning as well as play an audio cue to alert the driver of an incoming collision

5.2.3 Testing Interface

- test()
    - Tests the software by reading an input file of sensors and their states. This will test every action that the software can take multiple times.
    - It will then display whether the software took the expected action(s) or not. If it did not, then it will display the system log for the action(s).

- showSystemLog()
    - Shows the technician every line of system log, showing history of both software and any interaction the car had.

5.3 Component-Level-Design

5.3.1 Sensors
Converts raw data into usable data which is given to either localization or perception.

5.3.2 Sensor Fusion
Takes in data from localization and perception, then normalizes it. This allows the Planning module to read.

5.3.3 Planning
Takes in data from sensor fusion, Decides what to do with the data, sends logs to sys management, orders/actions to VCS.

5.3.4 System Management
Gathers data from Planning in order to upload it to the cloud as logs, data to display

5.3.5 Security
Takes in a key/password, sets designated security level (driver/tech).

5.3.6 Display
Takes in data from sys management, displays data for the user to see.

5.3.7 Vehicle Control
Takes inputs from the Planning module or the driver and does the actions (steering, braking, accelerating)

5.3.8 Localization
Takes in data from sensors pertaining to the vehicle itself and sends it to data fusion.

5.3.9 Perception
Takes in data from cameras and laser scanners

Section 6: Project Code

```
#include <iostream>
#include <sstream>
#include <fstream>
#include <string>
#include <tuple>
#include <chrono>
#include <ctime>
#include <cmath>
#include <vector>
#include <stdio.h>
#include <string.h>
#include <iomanip>
using namespace std;
/*
THIS IS THE ENTIRE SYSTEM
*/

////////////////////////// SYSTEM MANAGEMENT MODULE //////////////////////////
class systemManagementModule{
        public:
        static vector<string> sysLog;
        void receiveLog(string entry){//Records the sent log of data
        sysLog.push_back(entry);
        }

        /* old displayLog
        void displayLog(){//Displays the entire log one entry at a time
        for(int i = 0; i < (long)sysLog.size(); i++){
                cout << sysLog[i] << endl;
        }
        } */

        /*
        sends the log after 1 exec of the code to a file
        takes a filename as an input */
        void logToFile(string filename) {
        ofstream outFile(filename, ios::app); // appends log to end of previous one if it exists
        if (outFile.is_open()) {
                outFile << "\nNew entry:\n----------------------\n";
                for (int i = 0; i < (long)sysLog.size(); i++) {
                outFile << sysLog[i] << endl;
                }
                outFile.close();
                cout << "Log written to file: " << filename << endl;
        }
        else {
                cout << "Error: could not open file " << filename << " for writing" << endl;
        }
        }

};
vector<string> systemManagementModule::sysLog;// Enables static vector (compile error without)

////////////////////////// VEHICLE CONTROL SYSTEM MODULE //////////////////////////
class vehicleControlModule{
```

```
        public:
        void adjustWiperLevel(int wiperStatus, bool increaseWiper, bool decreaseWiper){
        if (increaseWiper) {
                wiperStatus++;
                std::cout << "Wiper level increased" << std::endl;
        }
        else if (decreaseWiper) {
                wiperStatus--;
                std::cout << "Wiper level decreased" << std::endl;
        }
        else {
                std::cout << "Wiper level stayed the same" << std::endl;
        }
        }

        void adjustTemperature(int internalTemperature, bool increaseTemperature, bool
decreaseTemperature){
        if (increaseTemperature) {
                internalTemperature++;
                std::cout << "Temperature increased" << std::endl;
        }
        else if (decreaseTemperature) {
                internalTemperature--;
                std::cout << "Temperature decreased" << std::endl;
        }
        else {
                std::cout << "Temperature stayed the same" << std::endl;
        }
        }

        void adjustHeadlights(bool state){
        if (state){//Turns headlights on
                //headlightsState = true;
                std::cout << "Headlights on" << std::endl;
        }
        else{//Turns headlights off
                //headlightsState = false;
                std::cout << "Headlights off" << std::endl;
        }
        }


        void activateLocks(){
        //door1 = locked;
        //door2 = locked;
        //door3 = locked;
        //door4 = locked;
        std::cout << "Doors locked" << std::endl;
        }


        void laneCorrection(float steeringAngle){
        //steeringWheelStatus = steeringAngle
        std::cout << "Steering changed to: " << steeringAngle << std::endl;
        }

        void applyBrake(int brakeLevel){
        //brakeStatus = brakeLevel;
        std::cout << "Brake level applied: " << brakeLevel << std::endl;
```

```
        }

        void trafficAction(string action) {
        if(action == "continueOn") {
                cout << "Light was Green, Continue!" << endl;
        }
        else if(action == "slowDown") {
                cout << "Light was Yellow, Slow Down!" << endl;
        }
        else if(action == "comeToStop") {
                cout << "Light was Red, Come to a Stop!" << endl;
        }
        else {
                cout << "Light unidentifiable. Requires driver action." << endl;
        }
        return;
        }
};

////////////////////////// PLANNING MODULE //////////////////////////
class PlanningModule{

        public:
        vehicleControlModule vcm;
        systemManagementModule smm;
        float forwardCollision, backCollision, lightSensor, accelerometer, breakStatus, gasPedal,
interiorThermometer, exteriorThermometer, desiredThermometer, rainSensor, wiperStatus,
steeringWheelStatus, roadAngleCamera1, roadAngleCamera2;
        string trafficCamera;
        bool cruiseControlButtonPressed, autoTemperatureStatus, autoWiperStatus, autoTrafficDetection,
driverSteering, emergencyVehicleCamera1, emergencyVehicleCamera2, doorLockButtonPressed1,
doorLockButtonPressed2, doorLockButtonPressed3, doorLockButtonPressed4;
        bool headlightState = false;
        float time;

        void ReceiveData(float _time, float _forwardCollision, float _backCollision, float
_lightSensor, float _accelerometer, float _breakStatus, float _gasPedal, float _interiorThermometer,
float _exteriorThermometer, float _desiredThermometer, float _rainSensor, float _wiperStatus, float
_steeringWheelStatus, float _roadAngleCamera1, float _roadAngleCamera2, string _trafficCamera, bool
_cruiseControlButtonPressed, bool _autoTemperatureStatus, bool _autoWiperStatus, bool
_autoTrafficDetection, bool _driverSteering, bool _emergencyVehicleCamera1, bool
_emergencyVehicleCamera2, bool _doorLockButtonPressed1, bool _doorLockButtonPressed2, bool
_doorLockButtonPressed3, bool _doorLockButtonPressed4 ){
        time = _time;
        forwardCollision = _forwardCollision;
        backCollision = _backCollision;
        lightSensor = _lightSensor;
        accelerometer = _accelerometer;
        breakStatus = _breakStatus;
        gasPedal = _gasPedal;
        interiorThermometer = _interiorThermometer;
        exteriorThermometer = _exteriorThermometer;
        desiredThermometer = _desiredThermometer;
        rainSensor = _rainSensor;
        wiperStatus = _wiperStatus;
        steeringWheelStatus = _steeringWheelStatus;
        roadAngleCamera1 = _roadAngleCamera1;
        roadAngleCamera2 = _roadAngleCamera2;
        trafficCamera = _trafficCamera;
```

```
        cruiseControlButtonPressed = _cruiseControlButtonPressed;
        autoTemperatureStatus = _autoTemperatureStatus;
        autoWiperStatus = _autoWiperStatus;
        autoTrafficDetection = _autoTrafficDetection;
        driverSteering = _driverSteering;
        emergencyVehicleCamera1 = _emergencyVehicleCamera1;
        emergencyVehicleCamera2 = _emergencyVehicleCamera2;
        doorLockButtonPressed1 = _doorLockButtonPressed1;
        doorLockButtonPressed2 = _doorLockButtonPressed2;
        doorLockButtonPressed3 = _doorLockButtonPressed3;
        doorLockButtonPressed4 = _doorLockButtonPressed4;


        Plan();
        }

        // function that calls the actual functions
        void Plan(){
        automaticHeadlights(lightSensor, headlightState);
        automaticDoorLocks(accelerometer, (doorLockButtonPressed1 && doorLockButtonPressed2 &&
doorLockButtonPressed3 && doorLockButtonPressed4));
        laneCorrection(accelerometer, roadAngleCamera1, roadAngleCamera2, driverSteering);
        ForwardCollision(forwardCollision);
        RearCollision(backCollision);
        automaticWindshield(rainSensor, wiperStatus, autoWiperStatus, cruiseControlButtonPressed,
autoTemperatureStatus, autoTrafficDetection);
        automaticTemperature(interiorThermometer, exteriorThermometer, desiredThermometer,
autoTemperatureStatus);
        trafficLightRecognition(trafficCamera, autoTrafficDetection);
        emergencyVehicleDetection(emergencyVehicleCamera1, emergencyVehicleCamera2);
        }

        /*
        string getTime(){
        char buff[70];
        std::chrono::system_clock::time_point now = std::chrono::system_clock::now();
        std::time_t now_c = std::chrono::system_clock::to_time_t(now);
        std::tm now_tm = *std::localtime(&now_c);
        strftime(buff, sizeof buff, "%c", &now_tm);
        return buff;
        }*/

        /*
        testing out this new getTime() that includes milliseconds, as well as changes format to:
        year-mon-day hour:minute:seconds.milliseconds */
        string getTime() {
        chrono::system_clock::time_point now = chrono::system_clock::now();
        auto now_ms = chrono::duration_cast<chrono::milliseconds>(now.time_since_epoch()) % 1000;
        time_t now_c = chrono::system_clock::to_time_t(now);
        tm now_tm = *localtime(&now_c);
        ostringstream oss;
        oss << put_time(&now_tm, "%Y-%m-%d %H:%M:%S.") << setfill('0') << setw(3) << now_ms.count();
        return oss.str();
        }


        std::tuple<bool, bool, bool> automaticWindshield(int rainSensor, int wiperStatus, bool
autoWiperStatus, bool increaseWiper, bool decreaseWiper, bool stayWiper){
        // If automaticWiperStatus is false, automatically print "Automatic wipers are off"
```

```
        ostringstream log;
        if (autoWiperStatus == false){
                log << getTime() << " - " << "Wipers off";
                smm.receiveLog(log.str());
                return std::make_tuple(false, false, false);
        }
        else{
                // depending on the rainSensor value, the wiperStatus has to increase or decrease to
match it
                if (rainSensor > wiperStatus){
                increaseWiper = true;
                log << getTime() << ": " << "Wiper status increased to" << wiperStatus;
                smm.receiveLog(log.str());
                vcm.adjustWiperLevel(2, true, false);
                return std::make_tuple(true, false, false);
                }
                else if (rainSensor < wiperStatus){
                decreaseWiper = true;
                log << getTime() << ": " << "Wiper status decreased to" << wiperStatus;
                smm.receiveLog(log.str());
                vcm.adjustWiperLevel(2, false, true);
                return std::make_tuple(false, true, false);
                }else if (rainSensor == wiperStatus){
                stayWiper = true;
                log << getTime() << ": " << "Wiper status stayed the same" << wiperStatus;
                smm.receiveLog(log.str());
                vcm.adjustWiperLevel(4, false, false);
                return std::make_tuple(false, false, true);
                }
        }

        // If none of the flags match, return "stayWiper" flag
        return std::make_tuple(false, false, true);
        }


        std::tuple<bool, bool, bool> automaticTemperature(int interiorTemperature, int
exteriorTemperature, int desiredTemperature, bool autoTemperatureStatus) {
        // If autoTemperatureStatus is false, automatically print "Automatic temperature is off"
        ostringstream log;
        bool increaseTemperature = false;
        bool decreaseTemperature = false;
        bool stayTemperature = false;
        if (autoTemperatureStatus == false) {
                std::cout << "Automatic temperature is off";
                log << getTime() << ": " << "Automatic temperature is off";
                smm.receiveLog(log.str());

                return std::make_tuple(false, false, false);
        }
        else {
                // increase or decrease the interiorTemperature to match the desiredTemperature
                if (interiorTemperature < desiredTemperature) {
                increaseTemperature = true;
                if (increaseTemperature) {
                        log << getTime() << ": " << "Interior temperature increased" <<
interiorTemperature;
                        smm.receiveLog(log.str());
                        vcm.adjustTemperature(67, true, false);
```

```
                        return std::make_tuple(true, false, false);
                }
                }
                else if (interiorTemperature > desiredTemperature) {
                decreaseTemperature = true;
                if (decreaseTemperature) {
                        log << getTime() << ": " << "Interior temperature decreased" <<
interiorTemperature;
                        smm.receiveLog(log.str());
                        vcm.adjustTemperature(70, false, true);
                        return std::make_tuple(false, true, false);
                }
                }
        }

        // If the interiorTemperature and desiredTemperature match, return a "stayTemperature" flag
        if (stayTemperature) {
                log << getTime() << ": " << "Interior temperature stayed the same";
                smm.receiveLog(log.str());
                vcm.adjustTemperature(65, false, false);
                return std::make_tuple(false, false, true);
        }

        // If none of the flags match, return "stayTemperature" flag
        return std::make_tuple(false, false, true);
        }

        void automaticHeadlights(float lightLevel, bool state){
        ostringstream log;
        if (lightLevel < 400){//Tells VCS to turn on headlights.
                log << getTime() << ": automaticHeadLights{lightLevel: " << lightLevel << " state: " <<
state << " Decision: " << "on}";
                smm.receiveLog(log.str());
                vcm.adjustHeadlights(true);
                return;
        }
        else if (lightLevel >= 450){//Tells VCS to turn off headlights.
                log << getTime() << ": automaticHeadLights{lightLevel: " << lightLevel << " state: " <<
state << " Decision: " << "off}";
                smm.receiveLog(log.str());
                vcm.adjustHeadlights(false);
                return;
        }
        //Tells VCS to keep the headlights as they are.
        log << getTime() << ": automaticHeadLights{lightLevel: " << lightLevel << " state: " << state
<< " Decision: " << "none}";
        smm.receiveLog(log.str());
        vcm.adjustHeadlights(state);
        return;
        }


        void automaticDoorLocks(float speed, bool closeDoors){
        ostringstream log;
        if (closeDoors && speed >= 12){//Locks all doors
                log << getTime() << ": automaticDoorLocks{speed: " << speed << " closeDoors: " <<
closeDoors << " Decision: " << "lock}";
                smm.receiveLog(log.str());
                vcm.activateLocks();
```

```
                return;
        }
        log << getTime() << ": automaticDoorLocks{speed: " << speed << " closeDoors: " << closeDoors <<
" Decision: " << "none}";
        smm.receiveLog(log.str());
        return;
        }


        //Keeps the vehicle as centered in the lane as possible
        void laneCorrection(float speed, float leftLaneAngle, float rightLaneAngle, bool beingSteered){
        ostringstream log;
        if (beingSteered || speed == 0 || (leftLaneAngle + rightLaneAngle) == 0){
                log << getTime() << ": laneCorrection{speed: " << speed << " leftLaneAngle: " <<
leftLaneAngle << " rightLaneAngle: " << rightLaneAngle << " beingSteered: " << beingSteered << "
Decision: " << "none}";
                smm.receiveLog(log.str());
                return;
        }
        //Assumes that a positive angle means intersection if the vehicle continues forward
        float steerAngle = ((rightLaneAngle - leftLaneAngle)/2) * 4;//Gets the direction needed to
steer then multiplies by 4 for full 360 wheel rotation
        log << getTime() << ": laneCorrection{speed: " << speed << " leftLaneAngle: " << leftLaneAngle
<< " rightLaneAngle: " << rightLaneAngle << " beingSteered: " << beingSteered << " Decision: " << "Steer
" << steerAngle << "}";
        smm.receiveLog(log.str());
        vcm.laneCorrection(steerAngle);
        return;
        }

        void ForwardCollision(double dist){
        ostringstream log;
        if ( dist < 0 ){
                std::cout << "ERROR:  Distance is negative" << std::endl;
                log << getTime() << ": ForwardCollision ERROR: Distance is negative " << dist;
                smm.receiveLog(log.str());
                return;
        }
        if ( dist  < 250 ){
                if ( dist  < 25 ){
                log << getTime() << ": ForwardCollision: Distance : " << dist << " - Applying Brake
level 5" ;
                vcm.applyBrake(5);
                }
                else if ( dist  < 50 ){
                log << getTime() << ": ForwardCollision: Distance : " << dist << " - Applying Brake
level 4" ;
                vcm.applyBrake(4);
                }
                else if ( dist  < 100 ){
                log << getTime() << ": ForwardCollision: Distance : " << dist << " - Applying Brake
level 3" ;
                vcm.applyBrake(3);
                }
                else if ( dist  < 200 ){
                log << getTime() << ": ForwardCollision: Distance : " << dist << " - Applying Brake
level 2" ;
                vcm.applyBrake(2);
                }
```

```
                else{
                log << getTime() << ": ForwardCollision: Distance : " << dist << " - Applying Brake
level 1" ;
                vcm.applyBrake(1);
                }
                smm.receiveLog(log.str());
        }
        }
        void RearCollision(double dist){
        ostringstream log;
        if ( dist < 0 ){
                std::cout << "RearCollision ERROR:  Distance is negative" << std::endl;
                log << getTime() << ": RearCollision ERROR: Distance is negative " << dist;
                smm.receiveLog(log.str());
                return;
        }
        if ( dist  < 250 ){
                if ( dist  < 25 ){
                log << getTime() << ": RearCollision: Distance : " << dist << " - Applying Brake level
5" ;
                vcm.applyBrake(5);
                }
                else if ( dist  < 50 ){
                log << getTime() << ": RearCollision: Distance : " << dist << " - Applying Brake level
4" ;
                vcm.applyBrake(4);
                }
                else if ( dist  < 100 ){
                log << getTime() << ": RearCollision: Distance : " << dist << " - Applying Brake level
3" ;
                vcm.applyBrake(3);
                }
                else if ( dist  < 200 ){
                log << getTime() << ": RearCollision: Distance : " << dist << " - Applying Brake level
2" ;
                vcm.applyBrake(2);
                }
                else{
                log << getTime() << ": RearCollision: Distance : " << dist << " - Applying Brake level
1" ;
                vcm.applyBrake(1);
                }
                smm.receiveLog(log.str());
        }
        }

        void trafficLightRecognition(string color, bool autoTrafficDetection) {
        ostringstream log;
        //bool obstacle = false;
        string action;

        // If the driver has this function disabled:
        if(autoTrafficDetection == 0) {
                log << getTime() << ": trafficLightRecognition{ Traffic Light Recognition is off }";
                smm.receiveLog(log.str());
                return;
        }
        // using the camera, the type of signal displayed determines the action
        if (color == " green") {
```

```
                action = "continueOn";
                log << getTime() << ": trafficLightRecognition{ GREEN traffic light detected! }";
                smm.receiveLog(log.str());
                /*
                if(obstacle) {
                comeToStop()
                cout << "There is an obstacle blocking the intersection!"
                }*/
        }
        else if (color == " yellow") {
                action = "slowDown";
                log << getTime() << ": trafficLightRecognition{ YELLOW traffic light detected! }";
                smm.receiveLog(log.str());
                return;
        }
        else if(color == " red") {
                action = "comeToStop";
                log << getTime() << ": trafficLightRecognition{ RED traffic light detected! }";
                smm.receiveLog(log.str());
                return;
        }
        else if (color == " unidentifiable") {
                // updateDisplay
                log << getTime() << ": trafficLightRecognition{ UNIDENTIFIABLE traffic light detected!
}";
                smm.receiveLog(log.str());

        }

        cout << action << endl;
        vcm.trafficAction(action); // decides which action to take depending on the light color
        return;
        }

        void emergencyVehicleDetection(bool emergencyVehicleCamera1, bool emergencyVehicleCamera2) {
        // if either camera detects an active emergency vehicle,
        ostringstream log;
        if(emergencyVehicleCamera1 || emergencyVehicleCamera2) {
                log << getTime() << ": emergencyVehicleDetection{ Emergency Vehicle is Approaching! }";
                smm.receiveLog(log.str());
                return;
        }
        else if (!emergencyVehicleCamera1 && !emergencyVehicleCamera2) {
                log << getTime() << ": emergencyVehicleDetection{ No Emergency Vehicle Present. }";
                smm.receiveLog(log.str());
                return;
        }
        return;
        }

};

////////////////////////// SENSOR FUSION //////////////////////////
class SensorFusion{
        public:

        float forwardCollision1, forwardCollision2;
        float backCollision1, backCollision2;
        float lightSensor1, lightSensor2, lightSensor3, lightSensor4;
```

```
        bool cruiseControlButtonPressed;
        float accelerometer, breakStatus, gasPedal;
        float interiorThermometer, exteriorThermometer, desiredThermometer;
        bool autoTemperatureStatus, autoWiperStatus, autoTrafficDetection, driverSteering;
        float rainSensor, wiperStatus, steeringWheelStatus;
        string trafficCamera;
        float roadAngleCamera1, roadAngleCamera2;
        bool emergencyVehicleCamera1, emergencyVehicleCamera2, doorLockButtonPressed1,
doorLockButtonPressed2, doorLockButtonPressed3, doorLockButtonPressed4;
        float time;

        float forwardCollision, backCollision, lightSensor;
        PlanningModule planningModule;


        void normalizeData(){
        if((isnan(forwardCollision1) || forwardCollision1 <0) ){
                if(!(isnan(forwardCollision2)) && forwardCollision2 > 0){
                forwardCollision1 = forwardCollision2;
                }
                else{
                forwardCollision1 = 250;
                }
        }
        if((isnan(forwardCollision2) || forwardCollision2 <0) ){
                if(!(isnan(forwardCollision1)) && forwardCollision1 > 0){
                forwardCollision2 = forwardCollision1;
                }
                else{
                forwardCollision2 = 250;
                }
        }
        if((isnan(backCollision1) || backCollision1 <0) ){
                if(!(isnan(backCollision2)) && backCollision2 > 0){
                backCollision1 = backCollision2;
                }
                else{
                backCollision1 = 250;
                }
        }
        if((isnan(backCollision2) || backCollision2 <0) ){
                if(!(isnan(backCollision1)) && backCollision1 > 0){
                backCollision2 = backCollision1;
                }
                else{
                backCollision2 = 250;
                }
        }

        autoWiperStatus = true;
        autoTemperatureStatus = true;
        forwardCollision = (forwardCollision1+forwardCollision2)/2;
        backCollision = (backCollision1+backCollision2)/2;
        lightSensor = (lightSensor1 + lightSensor2 + lightSensor3 + lightSensor4)/4;

        sendData();

        }
```

```
        void sendData(){
        planningModule.ReceiveData(time, forwardCollision,backCollision, lightSensor, accelerometer,
breakStatus, gasPedal, interiorThermometer, exteriorThermometer, desiredThermometer, rainSensor,
wiperStatus, steeringWheelStatus, roadAngleCamera1, roadAngleCamera2,trafficCamera,
cruiseControlButtonPressed, autoTemperatureStatus, autoWiperStatus, autoTrafficDetection,
driverSteering, emergencyVehicleCamera1, emergencyVehicleCamera2, doorLockButtonPressed1,
doorLockButtonPressed2, doorLockButtonPressed3, doorLockButtonPressed4);
        }


};




////////////////////////// TECHNICIAN INTERFACE //////////////////////////
class technicianInterface{ /* Just printing all the data that is necessary*/
        public:
        //a function showSystemLog()
        void showSystemLog(){
        std::cout << "System Log" << std::endl;
        }

        //a function checkSensorHealth(sensorList)
        void checkSensorHealth(std::string sensorList){
        std::cout << "Sensor Health" << std::endl;
        }

        //a function checkSoftwareHealth() shows if the there are any issues
        void checkSoftwareHealth(){
        std::cout << "Software Health" << std::endl;
        }

        //a function installUpdate() that just prints "Update currently up to date: " and the current
time and date
        void installUpdate(){
        auto currentTime = std::chrono::system_clock::now();
        std::time_t time = std::chrono::system_clock::to_time_t(currentTime);
        std::cout << "Update currently up to date: " << std::ctime(&time) << std::endl;
        }

        //a function uploadLogToServer() that just prints "Log uploaded to server: " and the current
time and date
        void uploadLogToServer(){
        auto currentTime = std::chrono::system_clock::now();
        std::time_t time = std::chrono::system_clock::to_time_t(currentTime);
        std::cout << "Log uploaded to server: " << std::ctime(&time) << std::endl;
        }
};



////////////////////////// DRIVER INTERFACE //////////////////////////
class driverInterface{ /* This is where we can create a function like if the password is equal then they
can create a log output*/
        public:
        //a function displaySpeed(speed)
        void displaySpeed(int speed){
        cout << "Speed: " << speed << endl;
```

```
        }

        //a function that toggles cruise control
        void toggleCruiseControl(bool cruiseControlStatus){
        if (cruiseControlStatus) {
                cout << "Cruise control is on" << endl;
        }
        else {
                cout << "Cruise control is off" << endl;
        }
        }

        //a function that shows the fuel level
        void displayFuelLevel(int fuelLevel){
        cout << "Fuel level: " << fuelLevel << endl;
        }

        //a function that shows the temperature
        void displayTemperature(int temperature){
        cout << "Temperature: " << temperature << endl;
        }

        //a function collisionAlert
        void collisionAlert(int collisionAlert){
        if (collisionAlert > 200){
                cout << "Collision Alert: " << collisionAlert << endl;
        } else {
                cout << "No collision detected" << endl;
        }
        }
};


////////////////////////// TESTING INTERFACE //////////////////////////
class testingInterface{
        public:
        //test()
        void test(){
        cout << "Test" << endl;
        }

        //showSystemLog()
        void showSystemLog(){
        cout << "System Log" << endl;
        }
};

////////////////////////// MAIN //////////////////////////
int main() {
        SensorFusion sensorFusion;
        sensorFusion.time = 0;
        PlanningModule planningModule;
        sensorFusion.planningModule = planningModule;
        systemManagementModule sysMan;
        planningModule.smm = sysMan;
        vehicleControlModule vcm;
        planningModule.vcm = vcm;
        string filename = "rawData.txt";
        ifstream file(filename);
```

```
if (!file) {
cerr << "Error: Cannot open file " << filename << endl;
return 1;
}

vector<string> sensor_data;

string line;

while (getline(file, line)) {
char *c = new char[line.length() + 1];
std::copy(line.begin(), line.end(), c);
c[line.length()] = '\0';
string sensorName = strtok(c,",");
string sensorValue = strtok(NULL, ",");



if(sensorName == "forwardCollision1"){
sensorFusion.forwardCollision1 = stof(sensorValue);
}
else if(sensorName == "forwardCollision2"){
sensorFusion.forwardCollision2 = stof(sensorValue);
}
else if(sensorName == "backCollision1"){
sensorFusion.backCollision1 = stof(sensorValue);
}
else if(sensorName == "backCollision2"){
sensorFusion.backCollision2 = stof(sensorValue);
}
else if(sensorName == "lightSensor1"){
sensorFusion.lightSensor1 = stof(sensorValue);
}
else if(sensorName == "lightSensor2"){
sensorFusion.lightSensor2 = stof(sensorValue);
}
else if(sensorName == "lightSensor3"){
sensorFusion.lightSensor3 = stof(sensorValue);
}
else if(sensorName == "lightSensor4"){
sensorFusion.lightSensor4 = stof(sensorValue);
}
else if(sensorName == "cruiseControlButtonPressed"){
if(strcmp(sensorValue.c_str(),"true") == 0){
        sensorFusion.cruiseControlButtonPressed = true;
}
else{
        sensorFusion.cruiseControlButtonPressed = false;
}
}
else if(sensorName == "accelerometer"){
sensorFusion.accelerometer = stof(sensorValue);
}
else if(sensorName == "breakStatus"){
sensorFusion.breakStatus = stof(sensorValue);
}
else if(sensorName == "gasPedal"){
sensorFusion.gasPedal = stof(sensorValue);
```

```
}
else if(sensorName == "interiorThermometer"){
sensorFusion.interiorThermometer = stof(sensorValue);
}
else if(sensorName == "exteriorThermometer"){
sensorFusion.exteriorThermometer = stof(sensorValue);
}
else if(sensorName == "desiredThermometer"){
sensorFusion.desiredThermometer = stof(sensorValue);
}
else if(sensorName == "autoTemperatureStatus"){
if(strcmp(sensorValue.c_str(),"true") == 0){
        sensorFusion.autoTemperatureStatus = true;
}
else{
        sensorFusion.autoTemperatureStatus = false;
}
}
else if(sensorName == "rainSensor"){
sensorFusion.rainSensor = stof(sensorValue);
}
else if(sensorName == "wiperStatus"){
sensorFusion.wiperStatus = stof(sensorValue);
}
else if(sensorName == "autoWiperStatus"){
if(strcmp(sensorValue.c_str(),"true") == 0){
        sensorFusion.autoWiperStatus = true;
}
else{
        sensorFusion.autoWiperStatus = false;
}
}
else if(sensorName == "trafficCamera"){
sensorFusion.trafficCamera = sensorValue;
}
// changed if(strcmp(sensorValue.c_str(),"true") == 0)
// THIS might be why some variables aren't read correctly
else if(sensorName == "autoTrafficDetection"){
if(strcmp(sensorValue.c_str()," True") == 0){
        sensorFusion.autoTrafficDetection = true;
}
else{
        sensorFusion.autoTrafficDetection = false;
}
}
else if(sensorName == "driverSteering"){
if(strcmp(sensorValue.c_str(),"true") == 0){
        sensorFusion.driverSteering = true;
}
else{
        sensorFusion.driverSteering = false;
}
}
else if(sensorName == "steeringWheelStatus"){
sensorFusion.steeringWheelStatus = stof(sensorValue);
}
else if(sensorName == "roadAngleCamera1"){
sensorFusion.roadAngleCamera1 = stof(sensorValue);
}
```

```
else if(sensorName == "roadAngleCamera2"){
sensorFusion.roadAngleCamera2 = stof(sensorValue);
}
else if(sensorName == "emergencyVehicleCamera1"){
if(strcmp(sensorValue.c_str()," True") == 0){
        sensorFusion.emergencyVehicleCamera1 = true;
}
else{
        sensorFusion.emergencyVehicleCamera1 = false;
}
}
else if(sensorName == "emergencyVehicleCamera2"){
if(strcmp(sensorValue.c_str()," True") == 0){
        sensorFusion.emergencyVehicleCamera2 = true;
}
else{
        sensorFusion.emergencyVehicleCamera2 = false;
}
}
else if(sensorName == "doorLockButtonPressed1"){
if(strcmp(sensorValue.c_str(),"true") == 0){
        sensorFusion.doorLockButtonPressed1 = true;
}
else{
        sensorFusion.doorLockButtonPressed1 = false;
}
}
else if(sensorName == "doorLockButtonPressed2"){
if(strcmp(sensorValue.c_str(),"true") == 0){
        sensorFusion.doorLockButtonPressed2 = true;
}
else{
        sensorFusion.doorLockButtonPressed2 = false;
}
}
else if(sensorName == "doorLockButtonPressed3"){
if(strcmp(sensorValue.c_str(),"true") == 0){
        sensorFusion.doorLockButtonPressed3 = true;
}
else{
        sensorFusion.doorLockButtonPressed3 = false;
}
}
else if(sensorName == "doorLockButtonPressed4"){
if(strcmp(sensorValue.c_str(),"true") == 0){
        sensorFusion.doorLockButtonPressed4 = true;
}
else{
        sensorFusion.doorLockButtonPressed4 = false;
}


sensorFusion.normalizeData();
sensorFusion.time += .05;
}

sensor_data.push_back(line);
}
```

```
        file.close();
        /* old displayLog() which only put it in the terminal
        planningModule.smm.displayLog();
        */
        planningModule.smm.logToFile("log.txt");
        return 0;
}
```

Section 7: Testing

7.1 Raw Data

This data is a mockup of all the vehicle sensors' outputs. Each batch of data is then taken in as input for all the modules and functions from Section 6.

```
forwardCollision1, 137
forwardCollision2, 89
backCollision1, 172
backCollision2, 201
lightSensor1, 322
lightSensor2, 768
lightSensor3, 452
lightSensor4, 199
cruiseControlButtonPressed, False
accelerometer, 121
breakStatus, 3
gasPedal, 2
interiorThermometer, 22
exteriorThermometer, 48
desiredThermometer, 68
autoTemperatureStatus, False
rainSensor, 1
wiperStatus, 0
autoWiperStatus, False
trafficCamera, green
autoTrafficDetection, False
driverSteering, True
steeringWheelStatus, -180
roadAngleCamera1, -36
roadAngleCamera2, 63
emergencyVehicleCamera1, False
emergencyVehicleCamera2, False
doorLockButtonPressed1, False
doorLockButtonPressed2, False
doorLockButtonPressed3, False
doorLockButtonPressed4, False
forwardCollision1, 16
forwardCollision2, 234
backCollision1, 95
backCollision2, 53
lightSensor1, 910
lightSensor2, 602
lightSensor3, 315
lightSensor4, 874
cruiseControlButtonPressed, True
accelerometer, 178
breakStatus, 1
gasPedal, 5
interiorThermometer, 35
exteriorThermometer, 83
desiredThermometer, 72
autoTemperatureStatus, True
rainSensor, 2
wiperStatus, 2
autoWiperStatus, True
trafficCamera, yellow
```

```
autoTrafficDetection, True
driverSteering, False
steeringWheelStatus, 270
roadAngleCamera1, 45
roadAngleCamera2, -54
emergencyVehicleCamera1, True
emergencyVehicleCamera2, True
doorLockButtonPressed1, True
doorLockButtonPressed2, False
doorLockButtonPressed3, True
doorLockButtonPressed4, True
forwardCollision1, 219
forwardCollision2, 24
backCollision1, 45
backCollision2, 149
lightSensor1, 121
lightSensor2, 398
lightSensor3, 790
lightSensor4, 582
cruiseControlButtonPressed, False
accelerometer, 46
breakStatus, 4
gasPedal, 3
interiorThermometer, 55
exteriorThermometer, -2
desiredThermometer, 62
autoTemperatureStatus, False
rainSensor, 0
wiperStatus, 1
autoWiperStatus, False
trafficCamera, red
autoTrafficDetection, False
driverSteering, True
steeringWheelStatus, 90
roadAngleCamera1, 20
roadAngleCamera2, 5
emergencyVehicleCamera1, False
emergencyVehicleCamera2, True
doorLockButtonPressed1, False
doorLockButtonPressed2, True
doorLockButtonPressed3, False
doorLockButtonPressed4, False
forwardCollision1, 205
forwardCollision2, 110
backCollision1, 38
backCollision2, 180
lightSensor1, 738
lightSensor2, 433
lightSensor3, 867
lightSensor4, 109
cruiseControlButtonPressed, False
accelerometer, 96
breakStatus, 0
gasPedal, 1
interiorThermometer, 44
exteriorThermometer, 29
desiredThermometer, 75
autoTemperatureStatus, True
rainSensor, 1
```

```
wiperStatus, 0
autoWiperStatus, True
trafficCamera, green
autoTrafficDetection, True
driverSteering, False
steeringWheelStatus, -45
roadAngleCamera1, -10
roadAngleCamera2, 47
emergencyVehicleCamera1, True
emergencyVehicleCamera2, False
doorLockButtonPressed1, True
doorLockButtonPressed2, True
doorLockButtonPressed3, False
doorLockButtonPressed4, False
forwardCollision1, 62
forwardCollision2, 218
backCollision1, 76
backCollision2, 235
lightSensor1, 292
lightSensor2, 97
lightSensor3, 951
lightSensor4, 512
cruiseControlButtonPressed, True
accelerometer, 150
breakStatus, 5
gasPedal, 0
interiorThermometer, 30
exteriorThermometer, 60
desiredThermometer, 80
autoTemperatureStatus, False
rainSensor, 3
wiperStatus, 3
autoWiperStatus, False
trafficCamera, yellow
autoTrafficDetection, False
driverSteering, True
steeringWheelStatus, 180
roadAngleCamera1, 85
roadAngleCamera2, -73
emergencyVehicleCamera1, False
emergencyVehicleCamera2, True
doorLockButtonPressed1, False
doorLockButtonPressed2, False
doorLockButtonPressed3, True
doorLockButtonPressed4, True
forwardCollision1, 48
forwardCollision2, 175
backCollision1, 152
backCollision2, 21
lightSensor1, 804
lightSensor2, 609
lightSensor3, 239
lightSensor4, 657
cruiseControlButtonPressed, False
accelerometer, 189
breakStatus, 2
gasPedal, 4
interiorThermometer, 78
exteriorThermometer, 15
```

```
desiredThermometer, 65
autoTemperatureStatus, True
rainSensor, 0
wiperStatus, 1
autoWiperStatus, True
trafficCamera, red
autoTrafficDetection, True
driverSteering, False
steeringWheelStatus, -270
roadAngleCamera1, -50
roadAngleCamera2, 12
emergencyVehicleCamera1, True
emergencyVehicleCamera2, False
doorLockButtonPressed1, True
doorLockButtonPressed2, True
doorLockButtonPressed3, True
doorLockButtonPressed4, False
forwardCollision1, 130
forwardCollision2, 71
backCollision1, 203
backCollision2, 93
lightSensor1, 350
lightSensor2, 992
lightSensor3, 508
lightSensor4, 722
cruiseControlButtonPressed, True
accelerometer, 34
breakStatus, 1
gasPedal, 3
interiorThermometer, 40
exteriorThermometer, 92
desiredThermometer, 61
autoTemperatureStatus, False
rainSensor, 2
wiperStatus, 2
autoWiperStatus, False
trafficCamera, green
autoTrafficDetection, False
driverSteering, True
steeringWheelStatus, -135
roadAngleCamera1, 65
roadAngleCamera2, -88
emergencyVehicleCamera1, False
emergencyVehicleCamera2, True
doorLockButtonPressed1, False
doorLockButtonPressed2, False
doorLockButtonPressed3, False
doorLockButtonPressed4, True
forwardCollision1, 244
forwardCollision2, 155
backCollision1, 88
backCollision2, 65
lightSensor1, 141
lightSensor2, 258
lightSensor3, 894
lightSensor4, 301
cruiseControlButtonPressed, False
accelerometer, 111
breakStatus, 4
```

```
gasPedal, 5
interiorThermometer, 67
exteriorThermometer, -5
desiredThermometer, 74
autoTemperatureStatus, True
rainSensor, 1
wiperStatus, 0
autoWiperStatus, True
trafficCamera, yellow
autoTrafficDetection, True
driverSteering, False
steeringWheelStatus, 315
roadAngleCamera1, 0
roadAngleCamera2, 30
emergencyVehicleCamera1, True
emergencyVehicleCamera2, False
doorLockButtonPressed1, True
doorLockButtonPressed2, True
doorLockButtonPressed3, False
doorLockButtonPressed4, False
forwardCollision1, 99
forwardCollision2, 220
backCollision1, 190
backCollision2, 245
lightSensor1, 675
lightSensor2, 831
lightSensor3, 412
lightSensor4, 955
cruiseControlButtonPressed, True
accelerometer, 159
breakStatus, 3
gasPedal, 2
interiorThermometer, 84
exteriorThermometer, 70
desiredThermometer, 77
autoTemperatureStatus, False
rainSensor, 3
wiperStatus, 1
autoWiperStatus, False
trafficCamera, red
autoTrafficDetection, False
driverSteering, True
steeringWheelStatus, 45
roadAngleCamera1, 75
roadAngleCamera2, -60
emergencyVehicleCamera1, False
emergencyVehicleCamera2, True
doorLockButtonPressed1, False
doorLockButtonPressed2, False
doorLockButtonPressed3, True
doorLockButtonPressed4, True
forwardCollision1, 183
forwardCollision2, 144
backCollision1, 118
backCollision2, 39
lightSensor1, 456
lightSensor2, 789
lightSensor3, 628
lightSensor4, 215
```

```
cruiseControlButtonPressed, False
accelerometer, 67
breakStatus, 5
gasPedal, 1
interiorThermometer, 32
exteriorThermometer, 54
desiredThermometer, 81
autoTemperatureStatus, True
rainSensor, 0
wiperStatus, 3
autoWiperStatus, True
trafficCamera, green
autoTrafficDetection, True
driverSteering, False
steeringWheelStatus, -225
roadAngleCamera1, -15
roadAngleCamera2, 80
emergencyVehicleCamera1, True
emergencyVehicleCamera2, False
doorLockButtonPressed1, True
doorLockButtonPressed2, True
doorLockButtonPressed3, True
doorLockButtonPressed4, False
```

7.2 Main Function

This is used to run, and therefore, test the code:

```
int main() {
    SensorFusion sensorFusion;
    sensorFusion.time = 0;
    PlanningModule planningModule;
    sensorFusion.planningModule = planningModule;
    systemManagementModule sysMan;
    planningModule.smm = sysMan;
    vehicleControlModule vcm;
    planningModule.vcm = vcm;
    string filename = "rawData.txt";


    ifstream file(filename);

    if (!file) {
        cerr << "Error: Cannot open file " << filename << endl;
        return 1;
    }

    vector<string> sensor_data;

    string line;

    while (getline(file, line)) {
        char *c = new char[line.length() + 1];
        std::copy(line.begin(), line.end(), c);
        c[line.length()] = '\0';
        string sensorName = strtok(c,",");
        string sensorValue = strtok(NULL, ",");
```

```
if(sensorName == "forwardCollision1"){
    sensorFusion.forwardCollision1 = stof(sensorValue);
}
else if(sensorName == "forwardCollision2"){
    sensorFusion.forwardCollision2 = stof(sensorValue);
}
else if(sensorName == "backCollision1"){
    sensorFusion.backCollision1 = stof(sensorValue);
}
else if(sensorName == "backCollision2"){
    sensorFusion.backCollision2 = stof(sensorValue);
}
else if(sensorName == "lightSensor1"){
    sensorFusion.lightSensor1 = stof(sensorValue);
}
else if(sensorName == "lightSensor2"){
    sensorFusion.lightSensor2 = stof(sensorValue);
}
else if(sensorName == "lightSensor3"){
    sensorFusion.lightSensor3 = stof(sensorValue);
}
else if(sensorName == "lightSensor4"){
    sensorFusion.lightSensor4 = stof(sensorValue);
}
else if(sensorName == "cruiseControlButtonPressed"){
    if(strcmp(sensorValue.c_str(),"true") == 0){
        sensorFusion.cruiseControlButtonPressed = true;
    }
    else{
        sensorFusion.cruiseControlButtonPressed = false;
    }
}
else if(sensorName == "accelerometer"){
    sensorFusion.accelerometer = stof(sensorValue);
}
else if(sensorName == "breakStatus"){
    sensorFusion.breakStatus = stof(sensorValue);
}
else if(sensorName == "gasPedal"){
    sensorFusion.gasPedal = stof(sensorValue);
}
else if(sensorName == "interiorThermometer"){
    sensorFusion.interiorThermometer = stof(sensorValue);
}
else if(sensorName == "exteriorThermometer"){
    sensorFusion.exteriorThermometer = stof(sensorValue);
}
else if(sensorName == "desiredThermometer"){
    sensorFusion.desiredThermometer = stof(sensorValue);
}
else if(sensorName == "autoTemperatureStatus"){
    if(strcmp(sensorValue.c_str(),"true") == 0){
        sensorFusion.autoTemperatureStatus = true;
    }
    else{
        sensorFusion.autoTemperatureStatus = false;
```

```
    }
}
else if(sensorName == "rainSensor"){
    sensorFusion.rainSensor = stof(sensorValue);
}
else if(sensorName == "wiperStatus"){
    sensorFusion.wiperStatus = stof(sensorValue);
}
else if(sensorName == "autoWiperStatus"){
    if(strcmp(sensorValue.c_str(),"true") == 0){
        sensorFusion.autoWiperStatus = true;
    }
    else{
        sensorFusion.autoWiperStatus = false;
    }
}
else if(sensorName == "trafficCamera"){
    sensorFusion.trafficCamera = sensorValue;
}
// changed if(strcmp(sensorValue.c_str(),"true") == 0)
// THIS might be why some variables aren't read correctly
else if(sensorName == "autoTrafficDetection"){
    if(strcmp(sensorValue.c_str()," True") == 0){
        sensorFusion.autoTrafficDetection = true;
    }
    else{
        sensorFusion.autoTrafficDetection = false;
    }
}
else if(sensorName == "driverSteering"){
    if(strcmp(sensorValue.c_str(),"true") == 0){
        sensorFusion.driverSteering = true;
    }
    else{
        sensorFusion.driverSteering = false;
    }
}
else if(sensorName == "steeringWheelStatus"){
    sensorFusion.steeringWheelStatus = stof(sensorValue);
}
else if(sensorName == "roadAngleCamera1"){
    sensorFusion.roadAngleCamera1 = stof(sensorValue);
}
else if(sensorName == "roadAngleCamera2"){
    sensorFusion.roadAngleCamera2 = stof(sensorValue);
}
else if(sensorName == "emergencyVehicleCamera1"){
    if(strcmp(sensorValue.c_str()," True") == 0){
        sensorFusion.emergencyVehicleCamera1 = true;
    }
    else{
        sensorFusion.emergencyVehicleCamera1 = false;
    }
}
else if(sensorName == "emergencyVehicleCamera2"){
    if(strcmp(sensorValue.c_str()," True") == 0){
        sensorFusion.emergencyVehicleCamera2 = true;
    }
    else{
```

```
                sensorFusion.emergencyVehicleCamera2 = false;
            }
        }
        else if(sensorName == "doorLockButtonPressed1"){
            if(strcmp(sensorValue.c_str(),"true") == 0){
                sensorFusion.doorLockButtonPressed1 = true;
            }
            else{
                sensorFusion.doorLockButtonPressed1 = false;
            }
        }
        else if(sensorName == "doorLockButtonPressed2"){
            if(strcmp(sensorValue.c_str(),"true") == 0){
                sensorFusion.doorLockButtonPressed2 = true;
            }
            else{
                sensorFusion.doorLockButtonPressed2 = false;
            }
        }
        else if(sensorName == "doorLockButtonPressed3"){
            if(strcmp(sensorValue.c_str(),"true") == 0){
                sensorFusion.doorLockButtonPressed3 = true;
            }
            else{
                sensorFusion.doorLockButtonPressed3 = false;
            }
        }
        else if(sensorName == "doorLockButtonPressed4"){
            if(strcmp(sensorValue.c_str(),"true") == 0){
                sensorFusion.doorLockButtonPressed4 = true;
            }
            else{
                sensorFusion.doorLockButtonPressed4 = false;
            }


            sensorFusion.normalizeData();
            sensorFusion.time += .05;
        }

        sensor_data.push_back(line);
    }

    file.close();
    /* old displayLog() which only put it in the terminal
    planningModule.smm.displayLog();
    */
    planningModule.smm.logToFile("log.txt");
    return 0;
}
```