Team 13: Peyrovian's Posse

Milind Kathiari, Angel Todorov,

Andrew Schomber, Matthew Bernardon

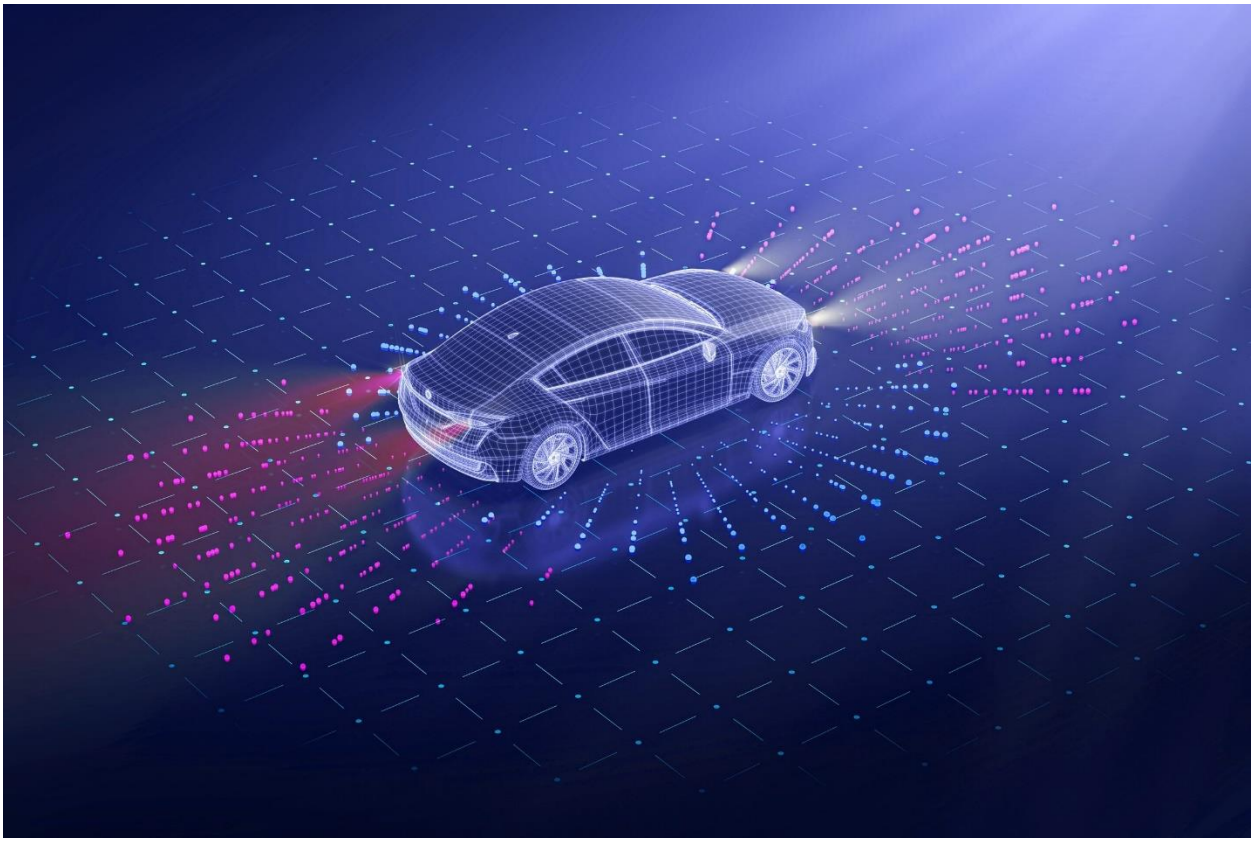


Table of Contents

Section 1: Introduction

As a team, we are responsible for designing the overarching infrastructure behind a self-driving vehicle. While we are not responsible for creating the individual components that will be used inside the vehicle, we are responsible for measuring their effectiveness and regulating the vehicle in real-time based on the responses provided by the various sensors and such that will inevitably be included within this vehicle. This project will culminate into an official "first release," but will also improve past this point in iterative releases which improve upon the initial release with feedback provided and data collected. This first release will hopefully include some of the following features: automatic break support, automatic steering, automatic acceleration, as well as manual versions of these features.

Additionally, our software supports automatic emergency handling. This project is uniquely classified, as it is forced to operate using mission critical real-time embedded systems. It needs rapid response time to change based on updated information from the roads surrounding it. Thus, it must be exceptionally reliable and consistent. We do not have the luxury of failure, as falling short could lead to catastrophe. The most effective way for us to have this speed and consistency is by basing our systems on IoT architecture. Creating such a system of intertwined devices communicating rapidly is only possible with this architecture. This allows us to access the information from the sensors and transport that data to the relevant components of the vehicle. This is only possible with IoT architecture.

To create such an advanced product, we must be able to work cohesively as a team, with a powerful sense of direction. This sense of direction can only be provided with a polished software development process. Thus, we will be utilizing the Prototyping Process Model. Our team is professionally qualified, with all of us being proficient with general programming. Milind is uniquely proficient in project management. Angel is uniquely proficient in generating ideas. Andrew is uniquely proficient in low-level programming. Matt is uniquely proficient in problem analysis. With these combined traits, we can effectively follow the Prototyping Process Model to build this product successfully.

Section 2: Functional Architecture

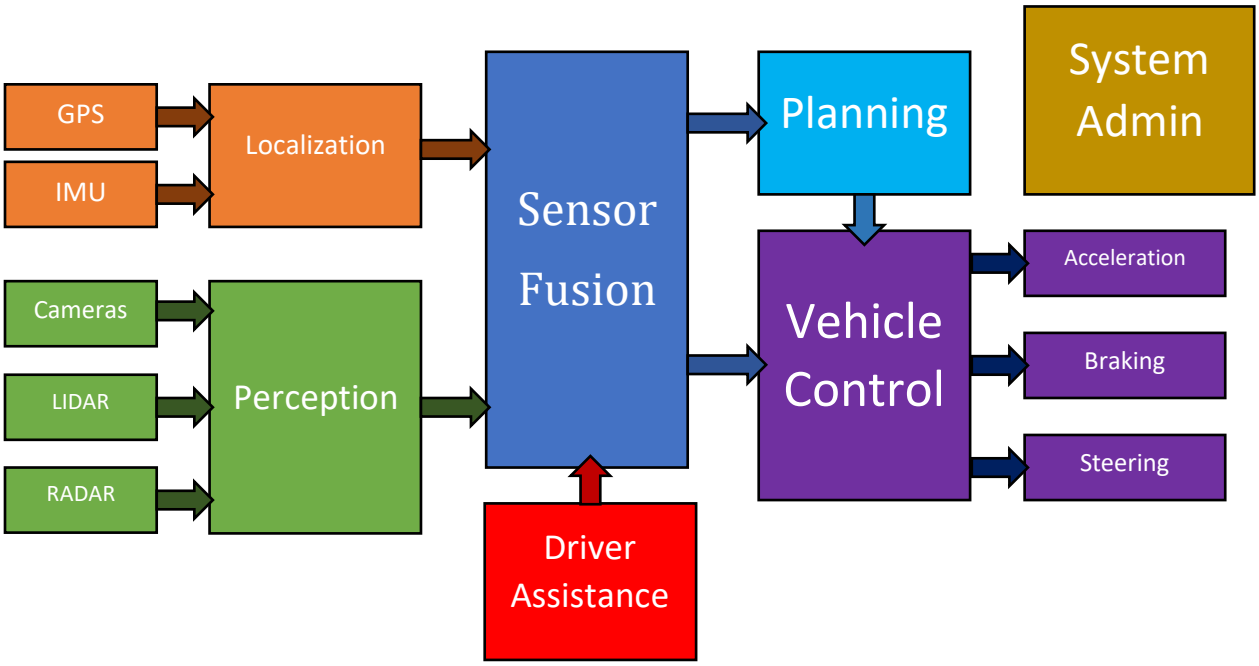


Figure 1: Functional Architecture

In designing the functional architecture for our self-driving vehicle, we must consider various components and how they interact to achieve seamless autonomous operation. This project is uniquely classified, as it is forced to operate using mission critical real-time embedded systems. Everything must work in unison and seamlessly to not put the lives of passengers and others at risk. Our vehicle will use a diverse array of sensors and technologies built on IoT architecture that will each play a crucial role in its operation.

2.1 Perception:

The sensors on the car need to make up for everything that would normally be handled by the driver. LIDAR helps identify the motion of objects and obstructions around the car, such as pedestrians and other vehicles. RADAR is used primarily to help determine the distances between the car and objects around it. RADAR works well over long distances in most weather and can also be used for blind spot detection and knowing when to switch

lanes. As will be discussed later, the fusing of LIDAR and RADAR tells the car when to accelerate, brake, and change lanes. Cameras are used for object identification and pathfinding through methods of image segmentation and machine learning. The cameras help distinguish between objects for decision making, such as identifying cars, stop signs, streetlights, pedestrians and so on. The is a considerable amount of overlap in the function of these components that allows for a prominent level of fault tolerance and a more seamless integration of them.

2.2 Localization:

It is vital for our car to always know exactly where it is on a large scale, and small scale. The IMU is very precise, allowing the car to know where it is on the road as well as its speed and direction. GPS data from Google Maps is used on a large scale for recognizing local speed limits or laws, such as the legality of right turns through red lights. The car will also receive local weather information and current traffic patterns for intelligent navigation.

2.3 Sensor Fusion & Planning:

Information from the sensors and localization will be condensed into a log file, where the processor can read from and make decisions based on that information. An accurate representation of the world around the car is constructed by fusing all the data collected through these methods. This allows the car to plan its course of action based on its environment and surroundings. As with localization, planning is done on both a large and small scale. The car needs to plan the overall route it will take as well as every small detail in controlling the vehicle, such as acceleration, braking, and steering. Warnings will also be given to the driver about weather and traffic data, along with prompts asking the driver if they would like their route to be further optimized to avoid traffic and inclement weather. Planning will be done by the machine learning model that will be trained by human driving data.

2.4 Vehicle Control:

Both the Sensor Fusion module and the Planning module are connected to the Vehicle Control module. This allows the car to be controlled in a more methodical way from planning, such as when to speed up, change lanes, or make turns, as well as quickly and directly by the sensor data when speed is vital, such as unexpected braking. In the latter case, a simple program will be used to detect when braking or serving is required and will apply those actions to the brakes and steering. In the former case, the machine learning model will make more complex decisions and actions, like smoother braking, acceleration, and steering.

2.5 Driver Assistance:

It is important for our car to have human control as an option for several reasons. Unforeseen circumstances such as damaged hardware, severe weather conditions, or unique obstructions would put the passengers and others at risk when in self-driving mode. In these emergencies, the driver could take the wheel and acquire control of the vehicle. The process of the driver taking control is seamless so that they can take control quickly without the interference of the self-driving functionality. While being controlled by the driver, the car will still take in information from all its sensors to inform the driver of obstructions, such as blind spot and lane departure warnings. Additionally, standard automatic safety features would still be employed when necessary, such as brake assist. Because all the actions performed by the driver are measured through sensors, driver assistance is connected to the Sensor Fusion module.

2.6 System Admin:

As is recommended by the University of Pennsylvania, our system admin component will handle “fault management, logging facility and the ‘human-machine interface’(HMI).” This is where we will locally store the log files, and when an internet connection is available, upload them to the cloud so our team of engineers can implement data-driven

improvements to the model. Finally, we will implement an HMI (Human Machine Interface) for drivers to access the navigation systems provided by the Google Maps API (Application Programming Interface) so that they can choose where to go to. The API will connect to the cloud for real-time navigation and updates as well as a network of neighboring vehicles with the software to increase awareness of nearby vehicles.

Section 3: Requirements

After planning our functional architecture, we must proceed on to the requirements necessary to materialize the features we want. With thorough requirements, we can collectively be more effective in the creation of our vehicle. Furthermore, clear requirements give us standards which we can test for, allowing us to ensure we can deliver an excellent product. Clear input and intended output will later be provided for these requirements, which will allow us to test our vehicle. Functional requirements are placed on the functional architecture, while non-functional requirements are more general. Any data that reaches the Planning Module will automatically be sent to the System Admin for logging purposes.

3.1 Functional Requirements

1. Automatic Braking
    - 1.1. If the Car detects that it will hit an object in front of it, the Car will automatically apply the brakes to avoid this collision.
    - 1.2. Pre-Conditions
        - 1.2.1. The speedometer detects that the Car is moving.
        - 1.2.2. An object that is detected in front of the Car using the Car's radar sensors.
        - 1.2.3. The Car is moving at a speed that would hit the object in 3 seconds or less if it continued at the same speed.
    - 1.3. Post-Conditions
        - 1.3.1. The brakes are applied to slow the Car to a stop.
        - 1.3.2. If the object stops being detected, the Car's speed reverts to what it was prior to braking.
    - 1.4. Requirements
        - 1.4.1. The Front Radar Sensor detects an object and sends the distance to Sensor Fusion.
        - 1.4.2. The Vehicle Speed Sensor tracks the speed of the Car and sends the data to Sensor Fusion.
        - 1.4.3. The Weather Sensor sends data about the current road conditions to Sensor Fusion.
        - 1.4.4. Sensor Fusion sends the data to Planning module.

1.4.5. If Planning calculates that the pre-conditions are met, it sends a brake signal to Vehicle Control with strength depending on the closeness of the object and the conditions of the road.

1.4.6. Planning sends a signal is also sent to the Console to let the Driver know that automatic brakes are being applied.

1.4.7. If the object is no longer detected by the Front Radar Sensor, the Car will accelerate to its original speed to avoid a collision from behind.

2. Driver-Assisted Steering Correction

2.1. In an area with clearly defined lanes (i.e., highways), if the Car is in Assisted-Driving mode and the Driver is beginning to veer from lane unintentionally, the Car will notify the driver and hold the Vehicle steady.

2.2. Pre-Conditions

2.2.1. The Speedometer detects that the Car is moving.

2.2.2. Cameras detect that the Vehicle is creeping out of its lane.

2.2.3. No turn signal or other indicator of lane change is presented.

2.3. Post-Conditions

2.3.1. A visual and auditory notification is presented to the Console.

2.3.2. Steering is corrected to have the Car remain within lane.

2.4. Requirements

2.4.1. Cameras detect nearby lanes and send data to perception.

2.4.2. IMU detects Car's current movement patterns and sends to localization.

2.4.3. Localization & Perception forward data to Sensor Fusion.

2.4.4. Sensor Fusion passes data to Planning.

2.4.5. If Planning detects that the pre-conditions are met, it sends a Lane-Departure signal to Vehicle Control.

2.4.6. Vehicle Control sends signals to the Steering Wheel to help the Vehicle back into the lanes.

2.4.7. Planning sends a signal to the Console to alert the Driver that the Vehicle's steering is being corrected.

3. Charging Station Navigation

3.1. No matter what charge the Car Battery has, the Car will always be able to locate all the charging stations that can be reached with that battery level for reference whenever the Driver wishes to recharge.

3.2. Pre-Conditions

3.2.1. The Car can reach a charging station with its current battery level.

3.2.2. The Car is at 15% or less charge or the Driver asks the Console to display charging stations.

3.3. Post-Conditions

3.3.1. The Console will display all the available charging stations that can be reached.

3.3.2. Charging stations will be displayed in order of distance to the Car when no destination is set, or by distance to the fastest route when a destination is set.

3.4. Requirements

3.4.1. The GPS sends positioning data to Sensor Fusion.

3.4.2. The Battery Sensor sends the battery level to Sensor Fusion.

3.4.3. The Weather Sensor sends temperature data to Sensor Fusion.

3.4.4. Sensor Fusion sends all the data to Planning.

3.4.5. Planning calculates the routes to all charging stations that can be reached on the current battery level given the current weather conditions.

3.4.6. When the pre-conditions are met, Planning sends this information to the Console, where the Driver can select a charging station to navigate to.

4. Assisted-Driving to Self-Driving Transition

4.1. When prompted by the Driver, the Vehicle should automatically gather necessary navigation data and transition to Self-Driving Mode within one second.

4.2. Pre-Conditions

4.2.1. The Vehicle is in Assisted-Driving Mode.

4.2.2. A destination is set in the Console.

4.2.3. The Driver presses and holds the Self-Driving Button for two seconds.

4.3. Post-Conditions

4.3.1. The Console displays a visual/auditory notification that indicates Self-Driving was prompted.

4.3.2. Car transitions to Self-Driving Mode.

4.4. Requirements

4.4.1. The Driver presses and holds the Self-Driving Button for two seconds.

4.4.2. Self-Driving Button sends push information to Sensor Fusion.

4.4.3. Sensor Fusion collects all data from Localization and Perception.

4.4.4. Sensor Fusion sends data to the Planning module.

4.4.5. If the Planning module is unable to activate Self-Driving for any reason, a message is sent to the Console to alert the Driver.

4.4.6. If the Planning module activates Self-Driving, it sends data to Vehicle Control.

4.4.7. Vehicle Control resumes control of Vehicle.

5. Parking Assistance

5.1. The Car will detect when the Driver wants to park and can park by itself if the Driver wishes.

5.2. Pre-Conditions

5.2.1. The speedometer detects that the Car is moving at 10 mph or less.

5.2.2. The Car's sensors detect that there are other parked cars in the vicinity.

5.3. Post-Conditions

5.3.1. The Car's steering and acceleration will be automatically controlled by the Car to park in a parking space or on the side of a street where parallel parking is allowed.

5.4. Requirements

5.4.1. The Radar Sensors send data to Sensor Fusion.

5.4.2. The LIDAR Sensor sends data to Sensor Fusion.

5.4.3. Sensor Fusion sends data to the Planning module.

5.4.4. If the pre-conditions are met, Planning sends a signal to the Console to ask the Driver if they wish to park.

5.4.5. If the Driver confirms, then Planning will send data to Vehicle Control to automatically park the Car.

5.4.6. Vehicle Control sends signals to the actuators and steering to maneuver the Car into the nearest empty space.

5.4.7. When in the parking space, the Car will put itself in Park.

6. Route Plotting

6.1. When a location is put into the Console, the Car will plot an optimal route to the location given GPS data & local traffic data.

6.2. Pre-Conditions

6.2.1. Driver inputs a destination into the Console.

6.2.2. Driver presses "Plot Route."

6.3. Post-Conditions

6.3.1. Console provides numerous route options, suggesting the most optimal one that arrives the quickest.

6.3.2. Console also displays traffic information on the route, and other pieces of relevant information (tolls, highways).

6.4. Requirements

6.4.1. GPS sends location data of the Vehicle and destination to Localization.

6.4.2. Localization passes data to Sensor Fusion.

6.4.3. Sensor Fusion passes data to Planning.

6.4.4. Planning displays the route options to the Console.

6.4.5. After the Driver selects a route, the Console will display GPS directions to the destination.

6.4.6. If in Self-Driving mode, Planning will send GPS directions to Vehicle Control to drive the Vehicle to the destination.

7. Emergency Pull-Over

7.1. Should the Driver press a designated "Emergency" button on the Console, the Vehicle will immediately take control and attempt to pull-over at a safe location.

7.2. Pre-Conditions

7.2.1. Driver presses "Emergency" button on Console.

7.3. Post-Conditions

7.3.1. Sensors locate a safe pull-over location.

7.3.2. Planning provides routing information to Vehicle Control.

7.3.3. Vehicle Control pulls Vehicle over at designated location.

7.4. Requirements

7.4.1. Console detects "Emergency" button press.

7.4.2. Sensor Fusion sends information (including "Emergency" button press) to Planning.

7.4.3. Planning sees "Emergency" button press, immediately locates a safe pull-over location, and sends routing information to Vehicle Control.

7.4.4. Vehicle Control routes the Vehicle to the location and parks.

8. Emergency Vehicle Detection & Response

8.1. Upon detection of emergency vehicles, if Vehicle is in Self-Driving mode, it will immediately notify the Driver and attempt to pull over.

8.2. Pre-Conditions

8.2.1. The speedometer detects the Car is moving.

8.2.2. Emergency Vehicles with sirens active are detected using the Car's Cameras.

8.3. Post-Conditions

8.3.1. Car notifies the Driver via visual and auditory notifications from the Console.

8.3.2. If Vehicle is in Self-Driving Mode, Vehicle Control immediately routes Car to safe location on side of road.

8.4. Requirements

8.4.1. Cameras send visual information to Perception.

8.4.2. Perception passes information to Sensor Fusion.

8.4.3. Sensor Fusion passes information to Planning.

8.4.4. Planning notices active-response Emergency Vehicles and checks if Vehicle is in Self-Driving Mode.

8.4.5. Planning notifies the Driver via the Console.

8.4.6. If in Self-Driving mode, Planning sends routing information to Vehicle Control.

8.4.7. Vehicle Control signals the actuators and steering wheel to pull the Car to the side of the road.

9. Crash Detection

9.1. If the Car detects that it has been in an accident, it will stop and contact emergency services.

9.2. Pre-Conditions

9.2.1. Vehicle detects a crash via Cameras & IMU.

9.3. Post-Conditions

9.3.1. Planning tells Vehicle Control to halt all operations.

9.3.2. Planning sends call to emergency services.

9.4. Requirements

9.4.1. Speedometer & IMU send locational information to Localization.

9.4.2. Cameras send visual information to Perception.

9.4.3. Localization & Perception passes information to Sensor Fusion.

9.4.4. Sensor Fusion passes information to Planning.

9.4.5. Planning notices a Car crash has occurred with information from Cameras and the sudden stop of movement from IMU and Speedometer.

9.4.6. Planning tells Vehicle Control to stop.

9.4.7. Planning calls emergency services.

10. Self-Driving to Assisted-Driving

10.1. Vehicle correctly interprets all Driver steering commands and disables Self-Driving if it is active.

10.2. Pre-Conditions

10.2.1. The Vehicle detects Steering, Braking, or Acceleration from Driver Input.

10.3. Post-Condition

10.3.1. Self-Driving mode is disabled.

10.3.2. Vehicle correctly adjusts movement based on Driver Input.

10.4. Requirements

10.4.1. Driver Assistance reads input from Driver, sends data to Sensor Fusion.

10.4.2. Sensor Fusion sends data to Planning module.

10.4.3. Planning reads input from Sensor Fusion, notices Driver Input, and sends the corresponding signals to Vehicle Control with high priority.

10.4.4. Planning disables Self-Driving mode if it was active.

10.4.5. Vehicle Control sends signals to the actuators and steering wheel to adjust Steering, Braking, and Acceleration based on driver input.

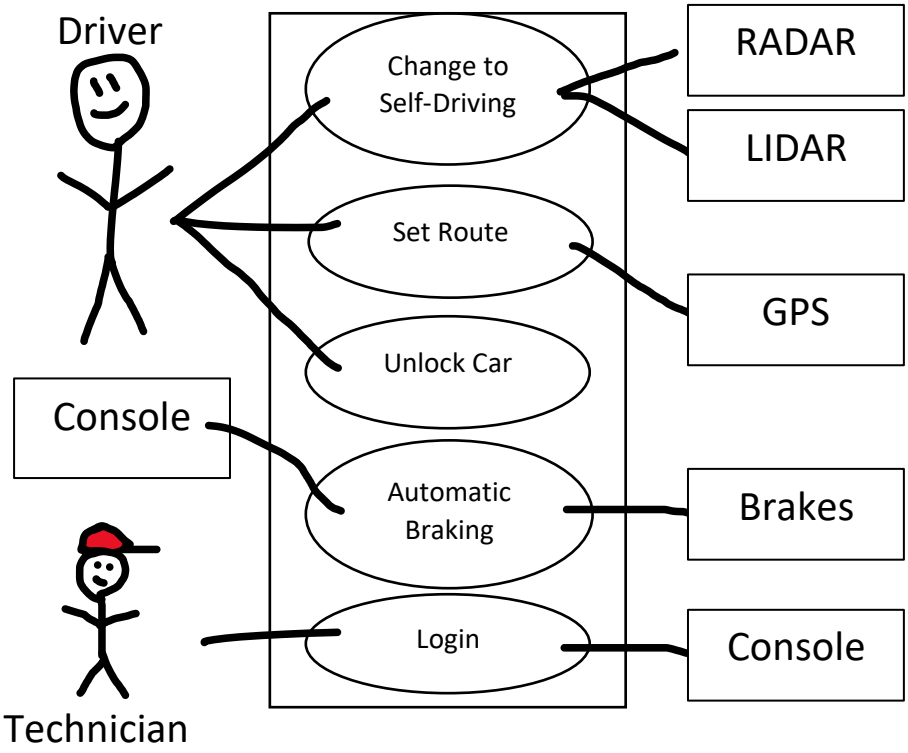
3.2 Non-Functional Requirements

1. Quality of the Car
    a. Materials
        i. The insides of the Car are made from the highest quality leather and fabric.
        ii. The frame is made of high strength and durable aluminum.
    b. Structure
        i. The Crumple Zones of the Car have been proven to work for crashes up to 80 mph and crumple 1-3 feet depending on crash speed.
2. Reliability
    a. Battery Length
        i. The Battery lasts up to 400 miles on a single charge
    b. Motor Life
        i. Each of the motors has a guaranteed lifespan of 25 years with regular servicing.
    c. Hardware
        i. All physical functions of the Car have at least six-nine reliability.
    d. Software
        i. Software has long-term support and will continue to receive updates for at least 10 years after production finishes.
        ii. Software has at least five-nine reliability.
3. Performance
    a. Motors
        i. Each of the 4 wheels have two electric motors dedicated to it for 8 motors in total.
    b. Architecture
        i. The time for sensors to receive data to time the Vehicle Control System acts shall be less than 10ms.
4. Security
    a. Key Fob
        i. The Car Doors automatically lock and unlock when the fob is five feet or less from the Car.
    b. Fingerprint Reader
        i. The Steering Wheel has a Fingerprint Reader such that the Car will only start and function when select people scan their finger.
    c. Software
        i. The Car's Operating System is a hardened RTOS such that no unauthorized connections can be made, and the User is incapable of modifying Security Settings.

ii. Only the Owner or a Technician can update the software or retrieve Log Files through the Fingerprint Reader or ID and password.

5. Software Updates
    a. Technician
        i. Larger software updates are installed by the technicians when brought in for servicing.
    b. Cloud
        i. Users are prompted to schedule a time for software updates when they do not need to use the Car.
        ii. If the Car is needed while it is being updated, the manual controls will be available to the Driver. However, the software of the Car that is being updated will not be available.
    c. Support
        i. Software has long-term support and will continue to receive updates for at least 10 years after production finishes.
6. Accessibility
    a. Visual Cues
        i. Software has easily readable font size with clearly labeled buttons for different settings and functionalities.
    b. Audio Cues
        i. Software incorporates voice activated voice-over features to assist the visually impaired.
    c. Wheelchair Accessible
        i. Software incorporates hardware drivers for extendible rails for people in wheelchairs to easily enter the vehicle.
7. Console Features
    a. Dimensions
        i. The Console screen is sized at 9 inches by 12 inches.
        ii. Console screen will have a refresh rate of 30hz.
    b. Functions
        i. The Console screen has numerous display modes, such as music, GPS mapping, phone notification alerts, and vehicle camera display.

Section 4: Requirement Modeling



4.1 Use Case Scenarios

1. Assisted Driving to Self-Driving
    a. When prompted by the Driver, the Vehicle should automatically gather necessary navigation data and transition to Self-Driving Mode within one second.
    b. Pre-Conditions
        i. The Vehicle is in Assisted-Driving Mode.
        ii. A destination is set in the Console.
    c. Post-Conditions
        i. The Console displays a visual/auditory notification that indicates Self-Driving was prompted.
        ii. Car transitions to Self-Driving Mode.
    d. Trigger
        i. The Driver presses and holds the "Self-Driving" Button for two seconds.
    e. Use Case Statements
        i. The Driver presses and holds the Self-Driving Button for two seconds.

ii. Self-Driving Button sends push information to Sensor Fusion.
iii. Sensor Fusion collects all data from Localization and Perception.
iv. Sensor Fusion sends data to the Planning module.
v. If the Planning module is unable to activate Self-Driving for any reason, a message is sent to the Console to alert the Driver.
vi. If the Planning module activates Self-Driving, it sends data to Vehicle Control.
vii. Vehicle Control resumes control of Vehicle.

2. GPS Route Planning
    a. When a location is inputted into the Console, the Car will plot an optimal route to the location given GPS data & local traffic data.
    b. Pre-Conditions
        i. Driver inputs a destination into the Console.
    c. Post-Conditions
        i. Console provides numerous route options, suggesting the most optimal one that arrives the quickest.
        ii. Console also displays traffic information on the route, and other pieces of relevant information (tolls, highways).
    d. Trigger
        i. Driver presses "Plot Route" after entering a destination.
    e. Use Case Statements
        i. GPS sends location data of the Vehicle and destination to Localization.
        ii. Localization passes data to Sensor Fusion.
        iii. Sensor Fusion passes data to Planning.
        iv. Planning displays the route options to the Console.
        v. After the Driver selects a route, the Console will display GPS directions to the destination.
        vi. If in Self-Driving mode, Planning will send GPS directions to Vehicle Control to drive the Vehicle to the destination.
3. Automatic Braking
    a. If the Car detects that it will hit an object in front of it, the Car will automatically apply the brakes to avoid this collision.
    b. Pre-Conditions
        i. The speedometer detects that the Car is moving.
        ii. An object that is detected in front of the Car using the Car's radar sensors.
        iii. The Car is moving at a speed that would hit the object in 3 seconds or less if it continued at the same speed.

c. Post-Conditions
    i. The brakes are applied to slow the Car to a stop.
    ii. If the object stops being detected, the Car's speed reverts to what it was prior to braking.
d. Trigger
    i. The Car's Front Radar Sensor detects it will collide with an object.
e. Use Case Statements
    i. The Front Radar Sensor detects an object and sends the distance to Sensor Fusion.
    ii. The Vehicle Speed Sensor tracks the speed of the Car and sends the data to Sensor Fusion.
    iii. The Weather Sensor sends data about the current road conditions to Sensor Fusion.
    iv. Sensor Fusion sends the data to Planning module.
    v. If Planning calculates that the pre-conditions are met, it sends a brake signal to Vehicle Control with strength depending on the closeness of the object and the conditions of the road.
    vi. Planning sends a signal is also sent to the Console to let the Driver know that automatic brakes are being applied.
    vii. If the object is no longer detected by the Front Radar Sensor, the Car will accelerate to its original speed to avoid a collision from behind.

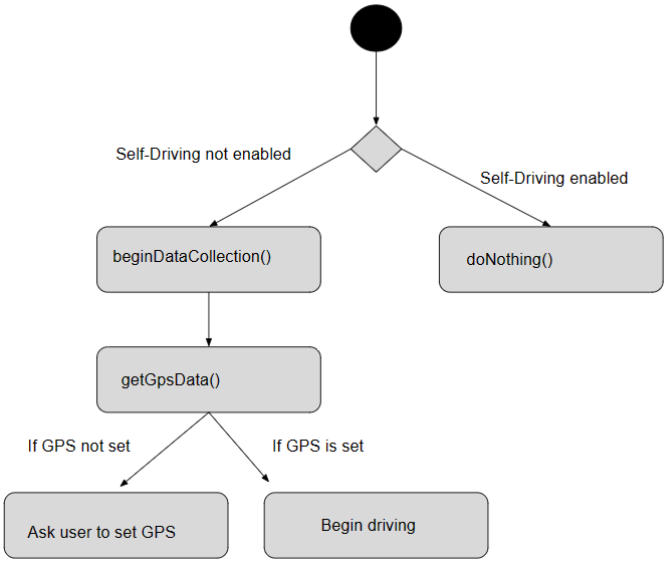
4. Key Fob Auto Lock & Unlock
    a. Preconditions
        i. The Key Fob is detected within five feet of the Key Fob Sensor.
        ii. The Key Fob is no longer detected within five feet of the Key Fob Sensor.
    b. Postconditions
        i. The Vehicle is locked or unlocked, depending on the distance to the Key Fob.
    c. Trigger
        i. The Key Fob is moved from greater than five feet to less than five feet from the Key Fob Sensor or vice versa.
    d. Use Case Statements
        i. The Key Fob Sensor pings the Key Fob every second.
        ii. If the Key Fob's distance is found, Key Fob sensor sends distance to Sensor Fusion.
        iii. Sensor Fusion sends distance data to Planning module.

iv. If Planning detects that the Key Fob's original distance was greater than five feet but is now less than five feet, then it sends a signal to the doors to unlock.
v. If the Planning detects that the Key Fob's original distance was less than five feet but is now greater than five feet, then it sends a signal to the doors to lock.

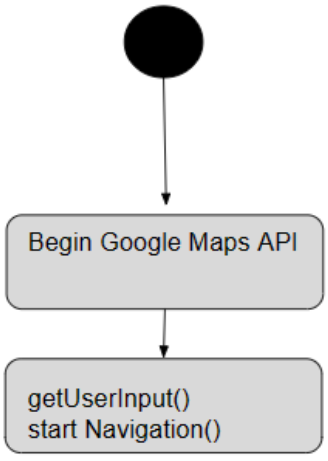
5. Technician Login
    a. Preconditions
        i. Driver brings Car to Technician.
        ii. Technician interacts with the Console to attempt to login.
    b. Postconditions
        i. The Car's software becomes unlocked so that the Technician can install Software Updates to the Car or modify existing settings.
    c. Trigger
        i. Technician enters a password in attempt to login.
    d. Use Case Statements
        i. Technician tells the Console that they want to login.
        ii. The Console displays a prompt asking for a password.
        iii. The Technician enters a password.
        iv. If the password is valid, the Car's software becomes unlocked.
        v. If the password is not valid, the prompt will display again.
        vi. If the password is not valid three times in a row, then the prompt will lock for one minute.
        vii. After the lock time has expired, further invalid passwords will lock the car for an additional five minutes per invalid password.

4.2 Activity Diagrams

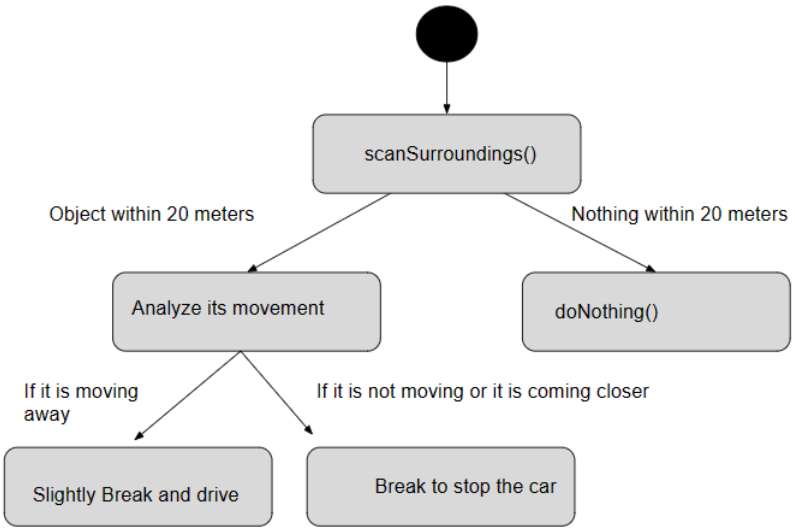
4.2.1 - Assisted Driving to Self-Driving



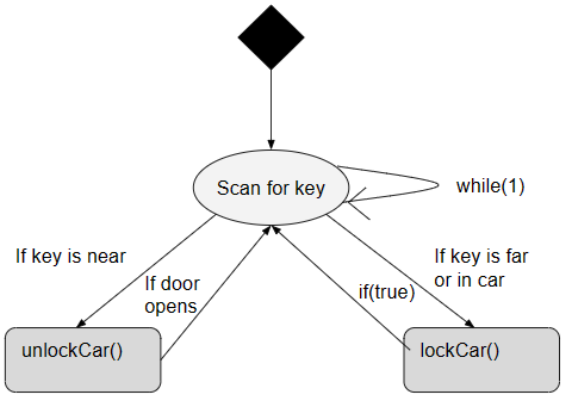
4.2.2 - GPS Route Planning



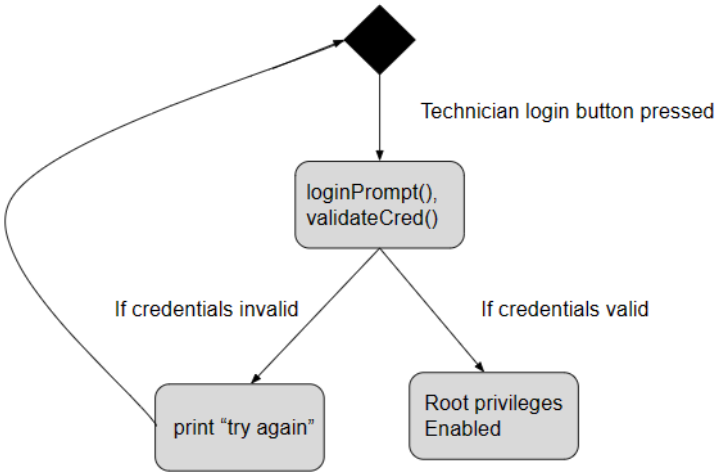
4.2.3 - Automatic Braking



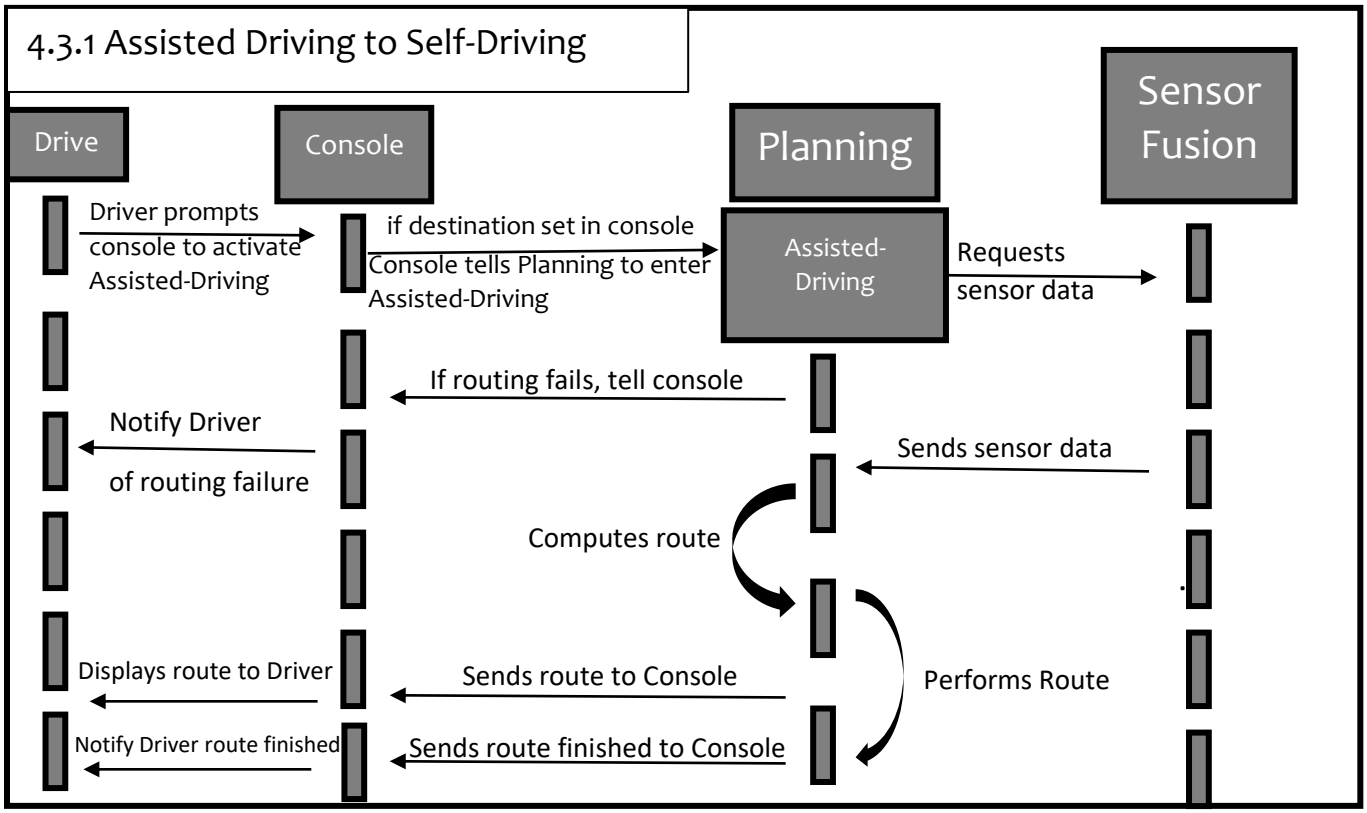
4.2.4 - Key Fob Auto Lock & Unlock

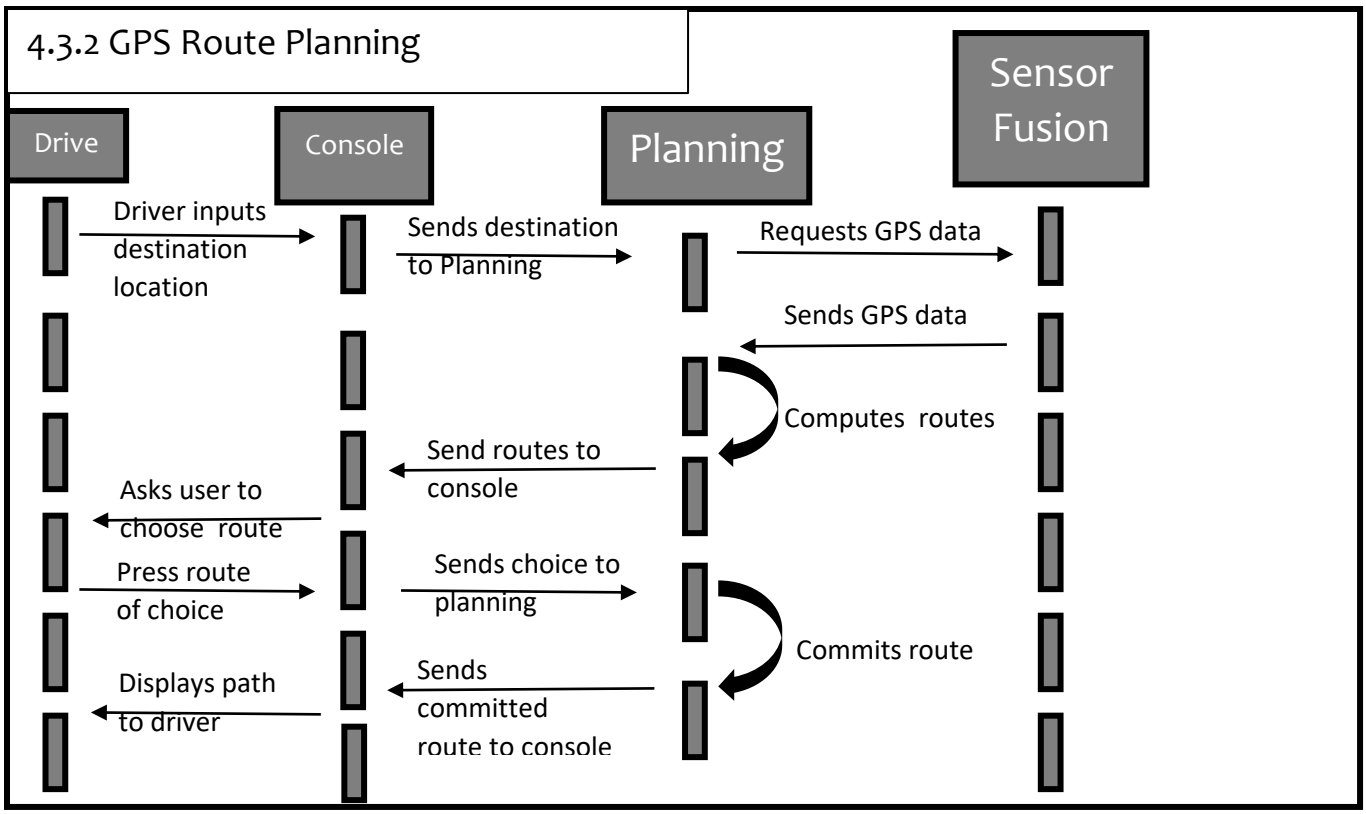


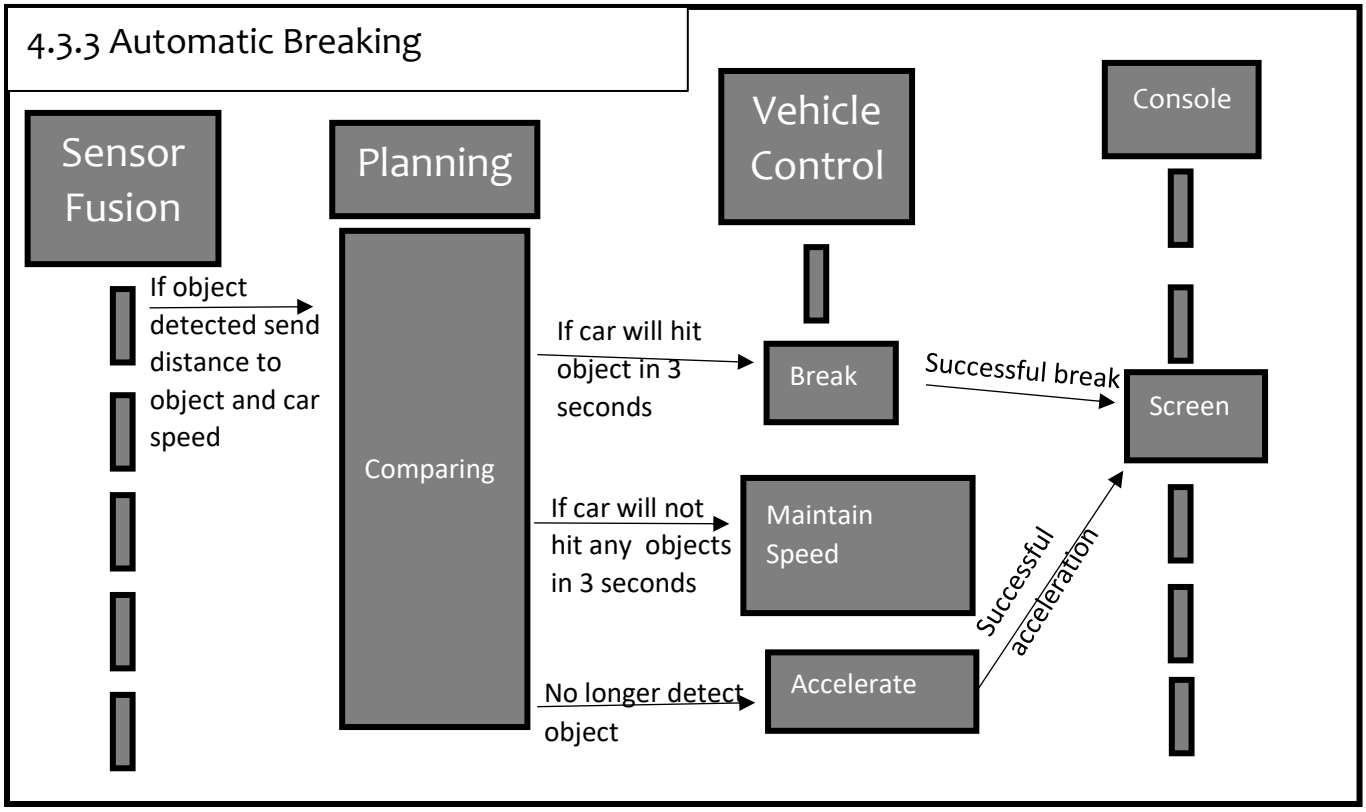
4.2.5 - Technician Login

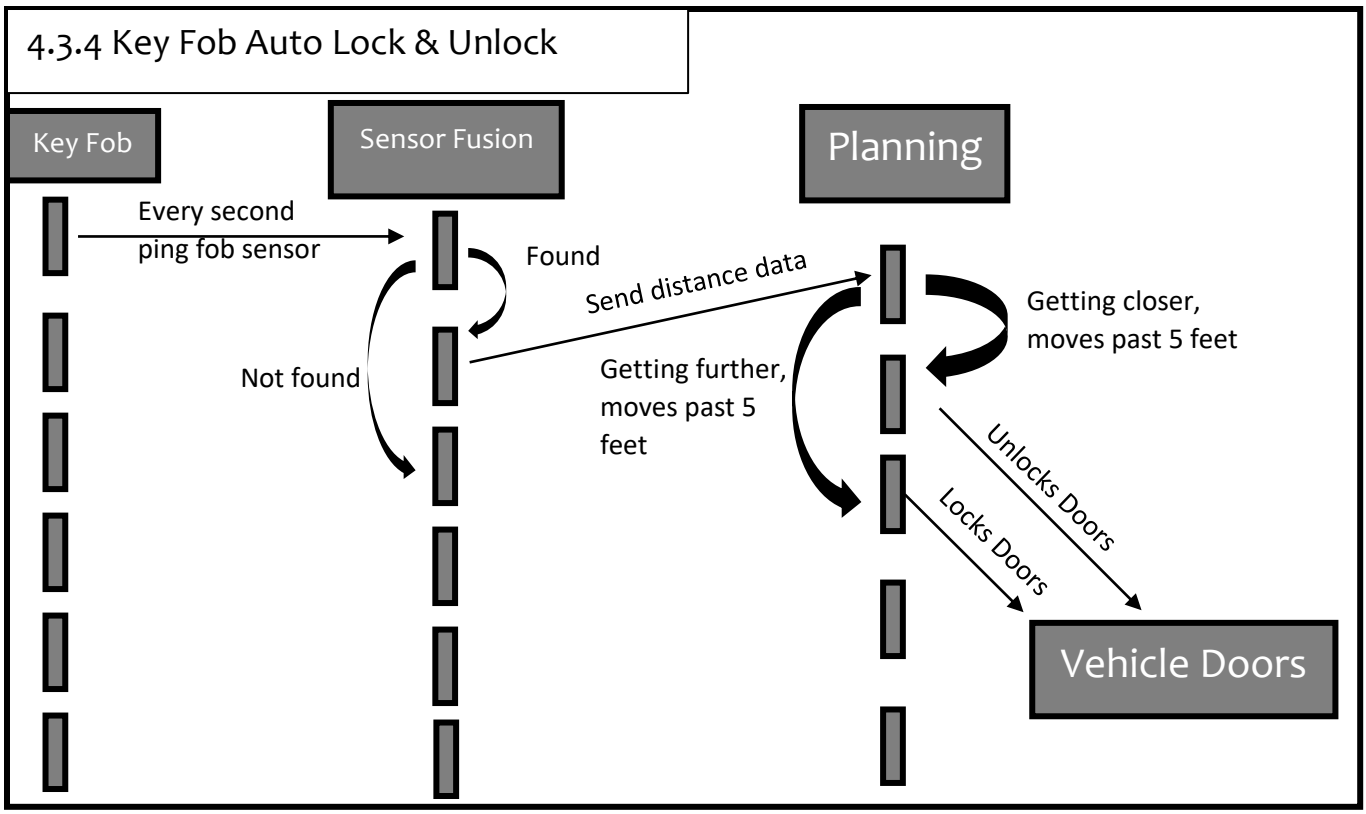


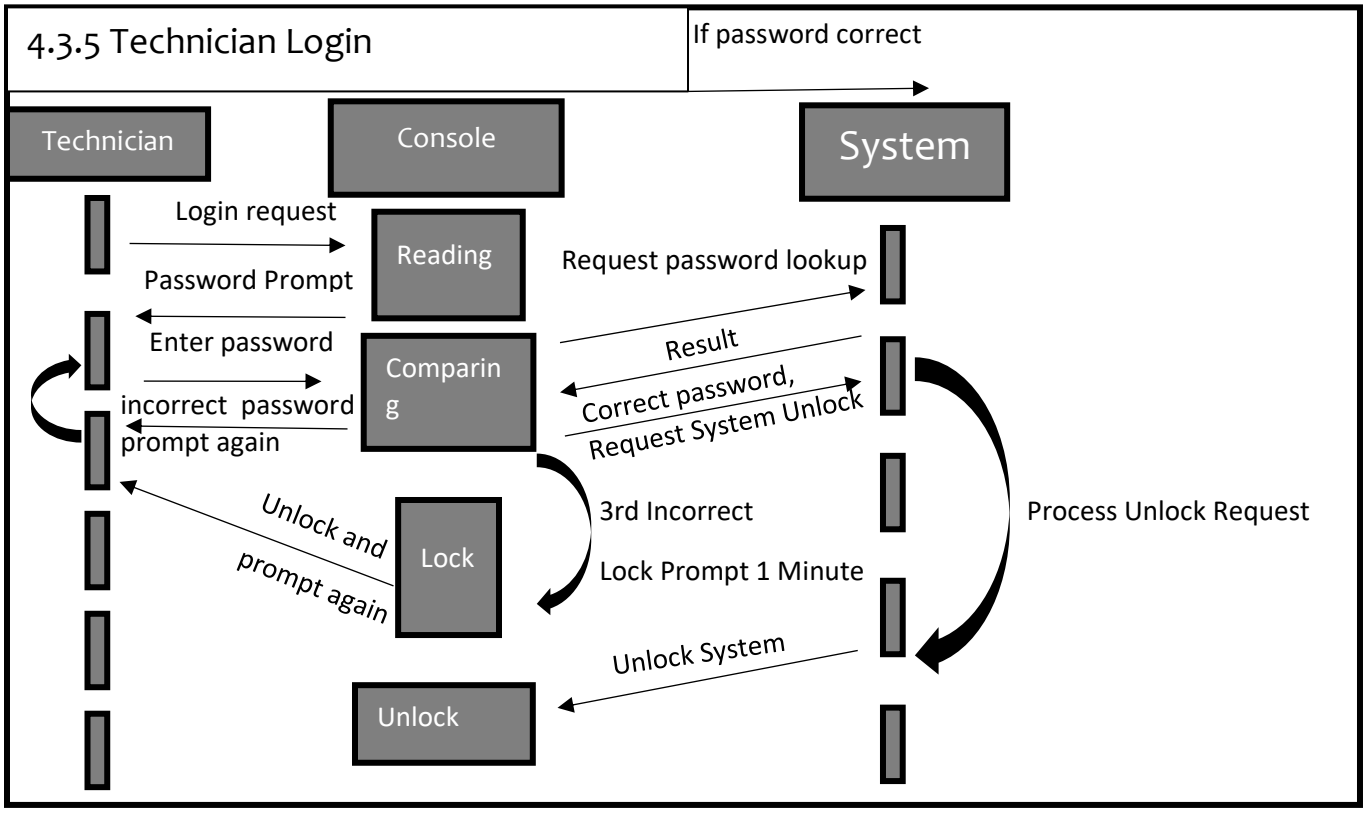
4.3 Sequence Diagrams




4.3.2 GPS Route Planning
Drive
Console
Planning
Sensor Fusion
Driver inputs destination location
Sends destination to Planning
Requests GPS data
Sends GPS data
Computes routes
Send routes to console
Asks user to choose route
Press route of choice
Sends choice to planning
Commits route
Displays path to driver
Sends committed route to console


4.3.3 Automatic Breaking
Sensor Fusion
Planning
Vehicle Control
Console
If object detected send distance to object and car speed
Comparing
If car will hit object in 3 seconds
Break
Successful break
Screen
If car will not hit any objects in 3 seconds
Maintain Speed
Successful acceleration
No longer detect object
Accelerate


4.3.4 Key Fob Auto Lock & Unlock
Key Fob
Sensor Fusion
Planning
Every second
ping fob sensor
Found
Send distance data
Getting closer,
moves past 5 feet
Not found
Getting further,
moves past 5
feet
Unlocks Doors
Locks Doors
Vehicle Doors


4.3.5 Technician Login
If password correct
Technician
Console
System
Login request
Reading
Request password lookup
Password Prompt
Enter password
Result
Comparin
g
Correct password,
Request System Unlock
incorrect password
prompt again
3rd Incorrect
Process Unlock Request
Unlock and
prompt again
Lock
Lock Prompt 1 Minute
Unlock System
Unlock

4.4 Classes

4.4.1 - GPS

GPS	Explanations
bool activeDisplay int[] destinations int[] currentLocation int[] nextDestination Route route	activeDisplay – boolean representing whether or not routing is displayed on console destinations – sequential array of destinations to be plotted currentLocation – current location nextDestination – the upcoming destination, could be thought of as destinations[0] in a zero-indexed language route – the current route to get to nextDestination
getETA() getRemainingDistance() updateMap() endRoute() pauseRoute() resumeRoute() addDestination() plotRoute() reRoute() notify()	getETA() – get estimated time of arrival to nextDestination getRemainingDistance() – get the remaining distance to nextDestination updateMap() – called when the car moves to update the map endRoute() – ends routing, either driver manually ends or destination is reached pauseRoute() – driver manually pauses routing resumeRoute() – driver manually resumes routing addDestination() – appends a destination to the array. Overloaded method should be available incase driver wants to add it to a specific location within the array plotRoute() – calculate the route to the next destination reRoute() – called either when driver wants to take an alternate route or deviates from the provided one – calculates the next best optimal pathing to destination notify() – when an upcoming milestone in the route is approaching (i.e. relevant turn or exit), notify driver

4.4.2 - Controls

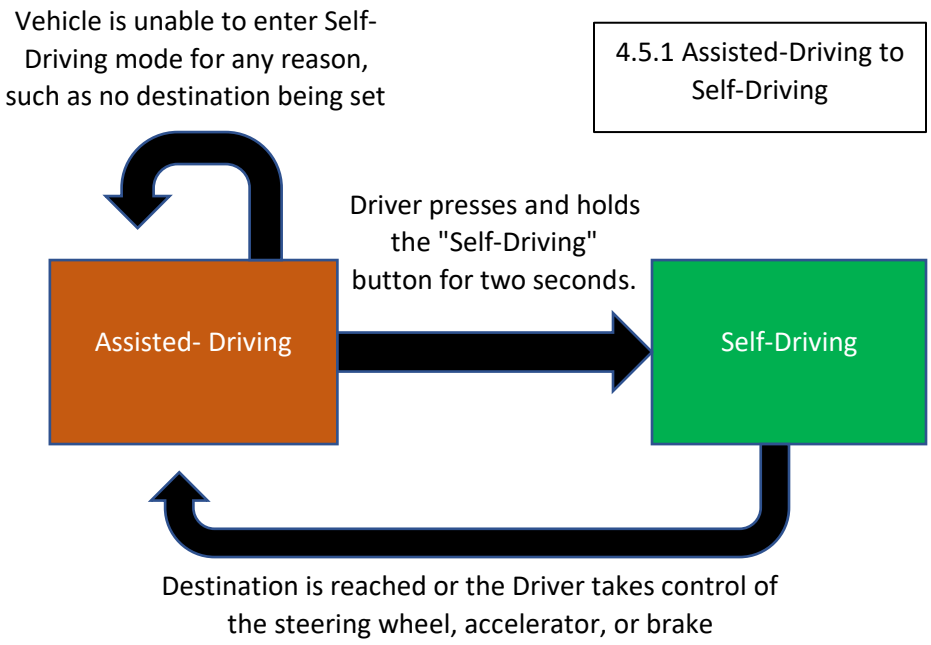
Controls	**Explanations**
int currentSpeed int[] currentLocation int currentAcceleration bool brakeStatus bool selfDriving bool cruiseControl int steeringAngle	currentSpeed – uses IMU to calculate current speed of car currentLocation – uses GPS to find current location of car currentAcceleration – uses IMU to calculate current acceleration of car brakeStatus – degree to which breaks are applied selfDriving – Boolean value representing if car is self-driving or assisted driving cruiseControl - boolean indicating whether or not cruise control is active steeringAngle – value representing the angle at which the steering wheel is facing
break() accelerate() setSpeed() toggleCC() toggleSelfDriving() turnWheel()	break() – modularly applies breaks to slow car down dependent upon driver pedal pressure or desired amount should car be in self-driving mode accelerate() - modularly applies acceleration to slow car down dependent upon driver pedal pressure or desired amount should car be in self-driving mode setSpeed() – sets the desired speed of vehicle. Used in either self-driving mode or when cruise control is active toggleCC() – toggles cruise control toggleSelfDriving – toggles self-driving mode turnWheel() – turns the wheel to the specified angle

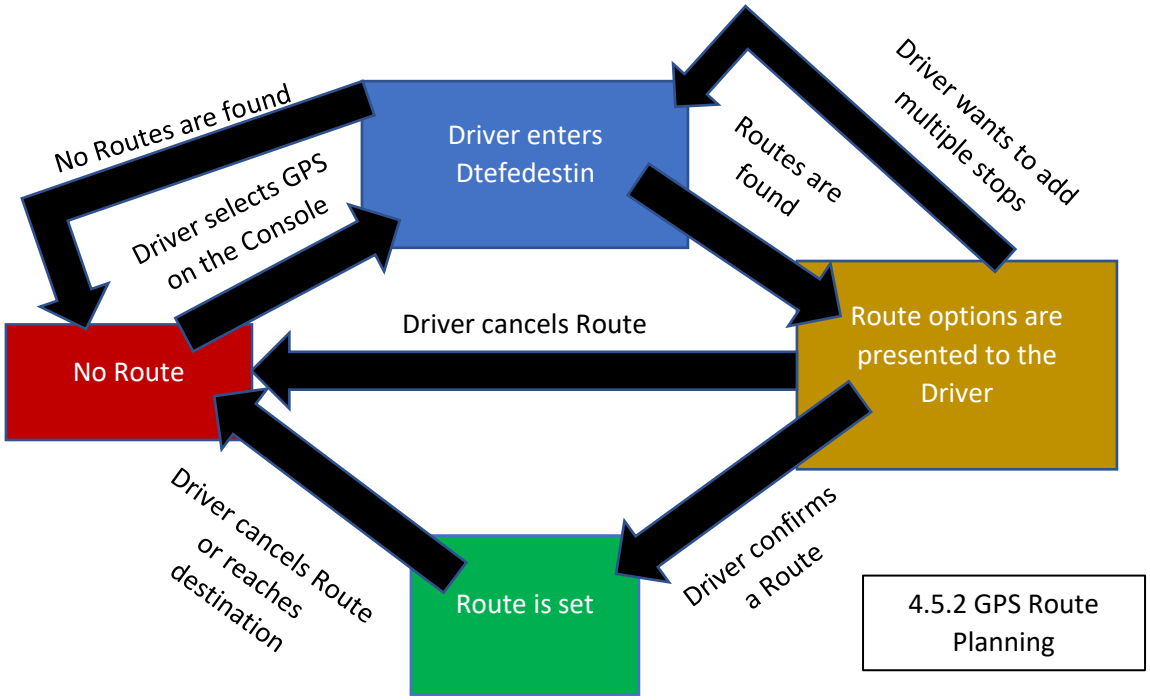
4.4.3 - Security

Security	**Explanations**
bool locked Stringp[] validUsers int[] enabledSensors int[] disabledSensors int[] systemIntegrity String issueLog String[] checkupItems	locked – Boolean value if the vehicle is locked or not validUsers – array of valid users enabledSensors – array of enabled sensors disabledSensors – array of disabled sensors systemIntegrity – percentage value representing the health of the system issueLog – record of all errors the car experiences checkupIssues – self-diagnosed issues provided for when maintenance is done
alarm() notifyAuthorities() sensorScan() integrityCheck() disableSensor() enableSensor() fobScan() toggleLock() addUser() removeUser() recommendMaintenance()	Alarm() – car alarm goes off notifyAuthorities() – sends a call to local authorities stating location and make/model of vehicle sensorScan() – checks health of sensors integrityCheck() – checks the physical integrity of the vehicle disableSensor() – if a sensor is detected to be faulty or otherwise impaired, it could be disabled enableSensor() – when a problematic sensor is fixed, it is re-enabled fobScan() – checks for approaching or leaving scan, respectively unlocks or locks toggleLock() – locks or unlocks addUser() – add valid user to array

	removeUser() – remove user from array recommendMainenance() – when health falls below certain level, vehicle will prompt user to take vehicle to technician

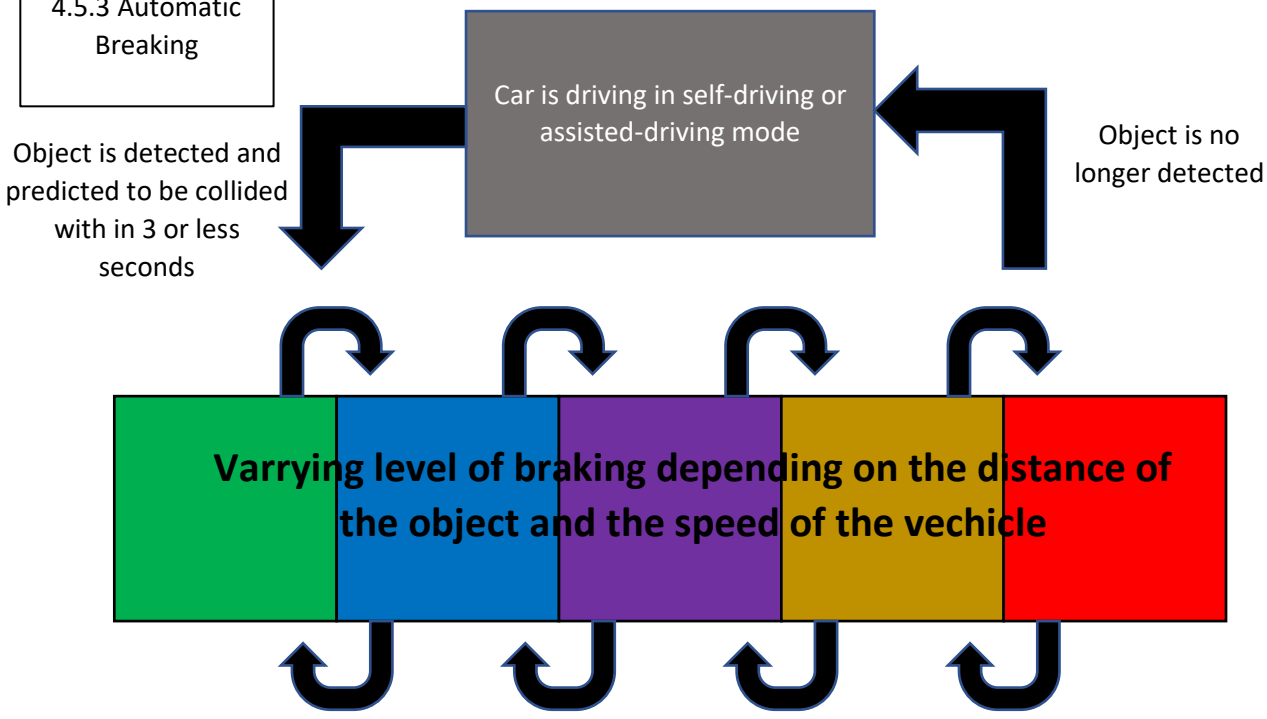
4.5 State Diagrams



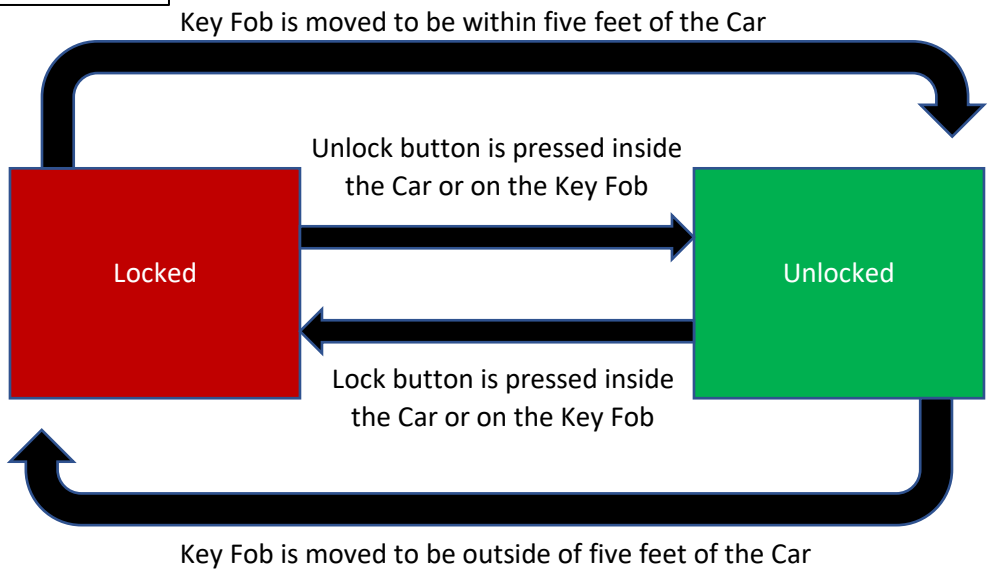


4.5.2 GPS Route Planning

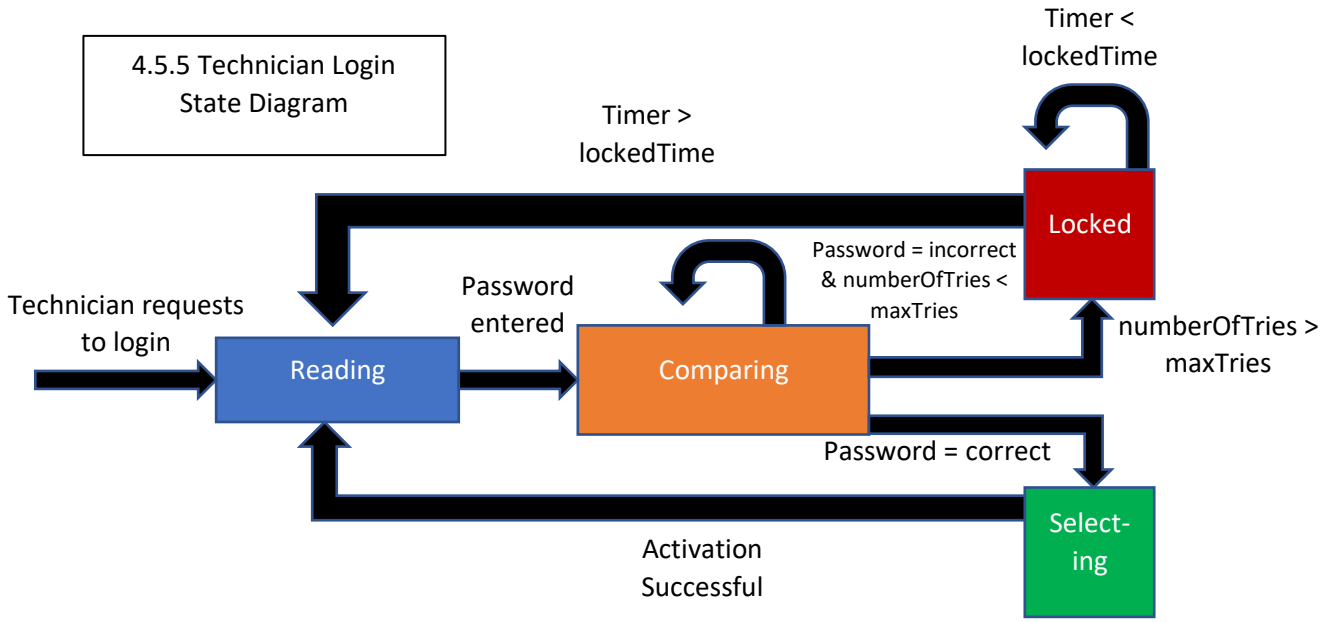
4.5.3 Automatic Breaking



4.5.4 Key Fob Auto Lock & Unlock



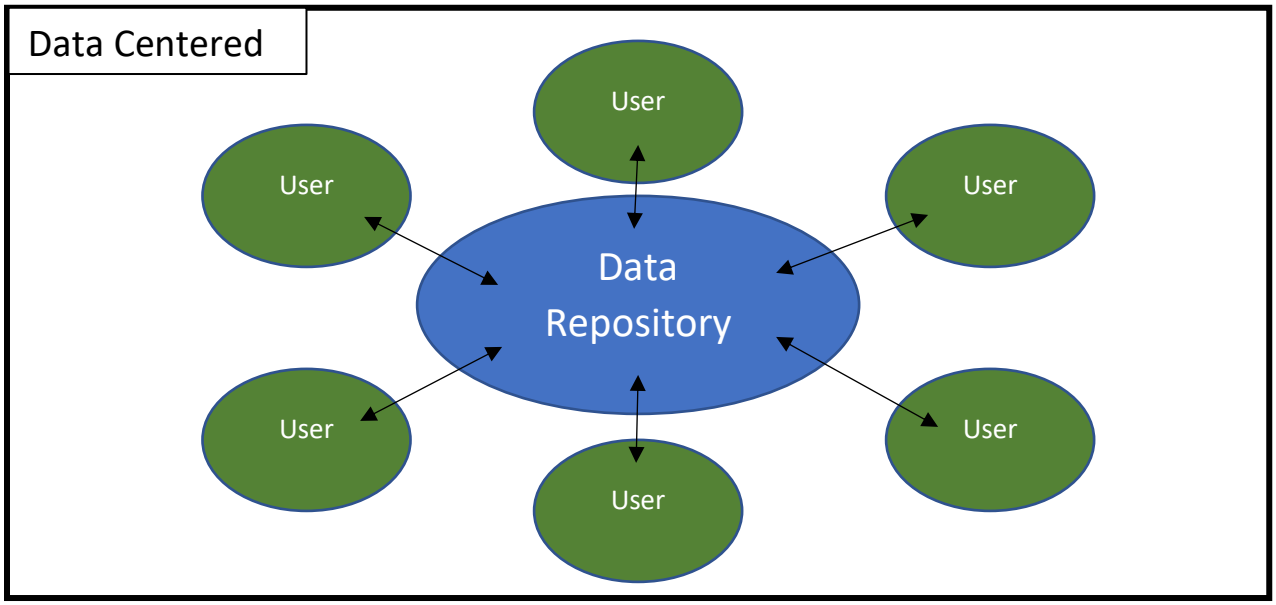
4.5.5 Technician Login State Diagram



Section 5: Design

5.1 Software Architecture

1. Data Center
    a. Pros
        i. Components can be changed or added without concern for other components being affected
        ii. Data can be easily passed among clients using blackboard or passive methods.
        iii. Clients can execute processes.
    b. Cons
        i. Valuable only when the system has a data repository and many clients.
        ii. Does not have any decision making.
        iii. Single point of failure.
    c. Overall
        i. Would not be a valid fit for IoT HTL because of its focus on a repository of data, not data in the moment.
        ii. However, it could be utilized for admin logging of sensor data.

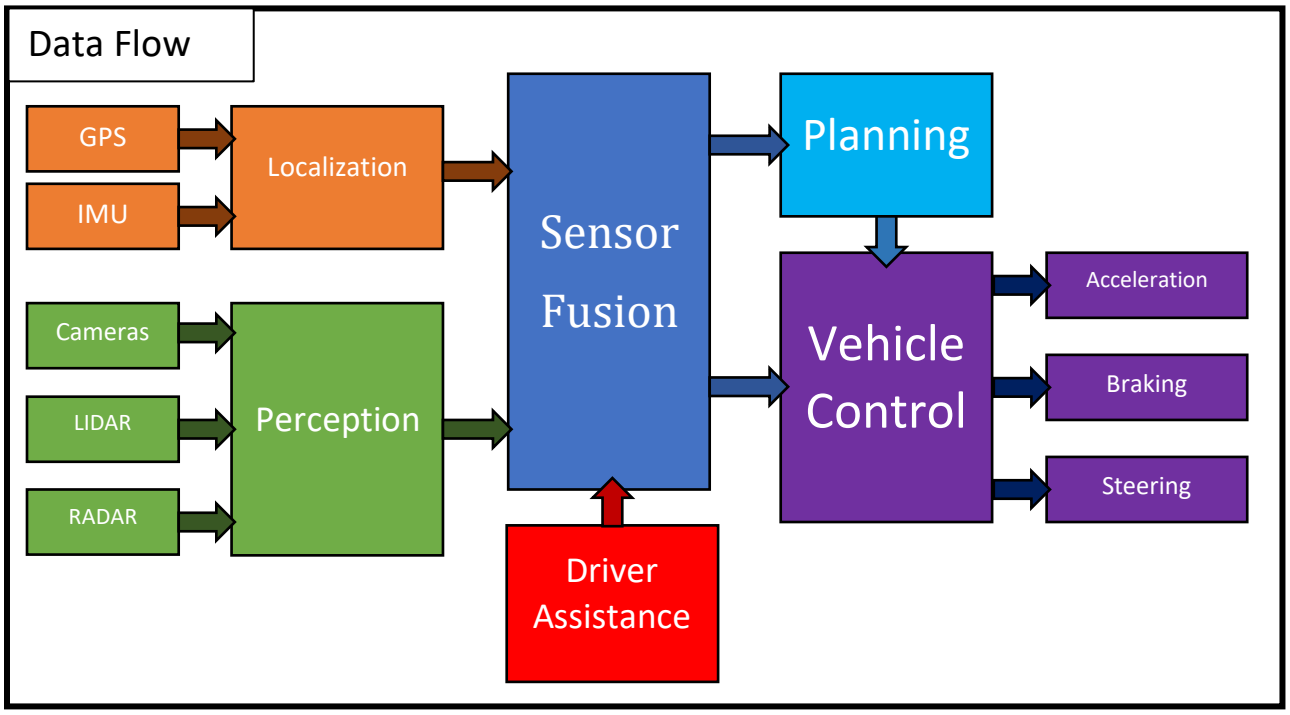


2. Data Flow
    a. Pros
        i. Can transform input into output through a series of components called filters.
        ii. Filters work independently of others and do not need to have knowledge of the workings of other filters.

iii. More modular and broad than some other architectures.

b. Cons
   i. It can become overly complex with larger systems.

c. Overall
   i. Data flow is a strong option for IoT HTL because of its flexibility and decision-making capabilities with respect to data management. Its representation mimics that of the functional architecture.

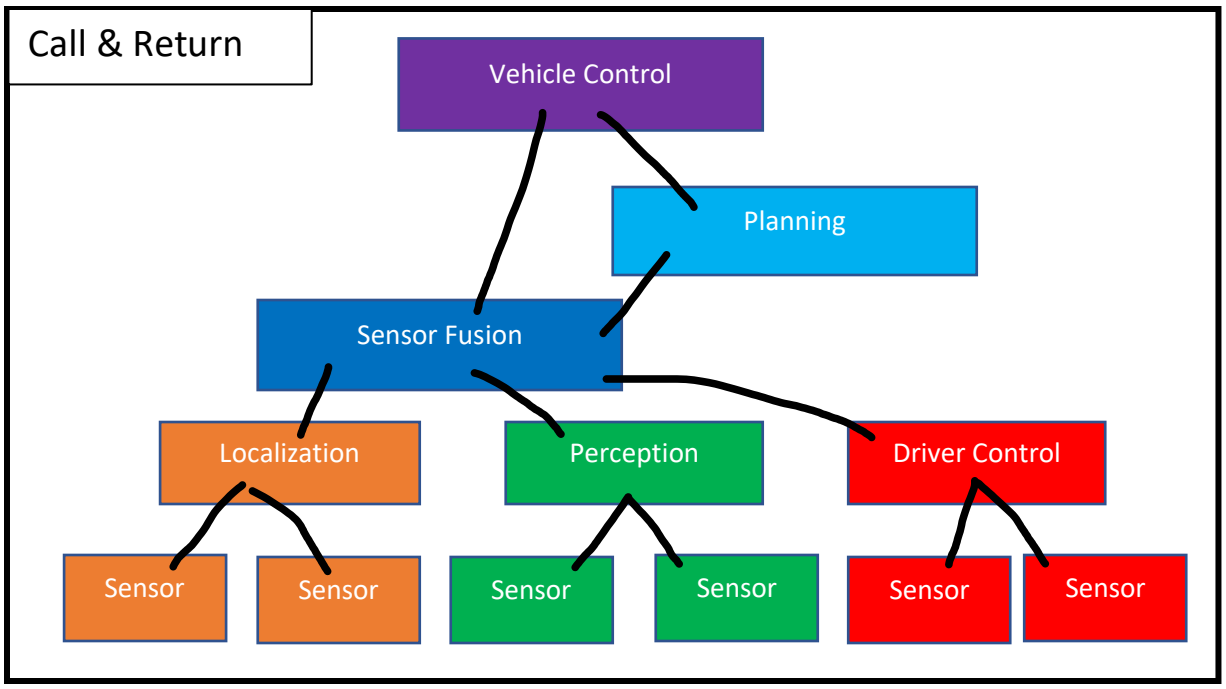


3. Call & Return

a. Pros
   i. Easy to modify, scale, and segment.
   ii. Program blocks are reuseable and structured.
   iii. Easier to debug and update.

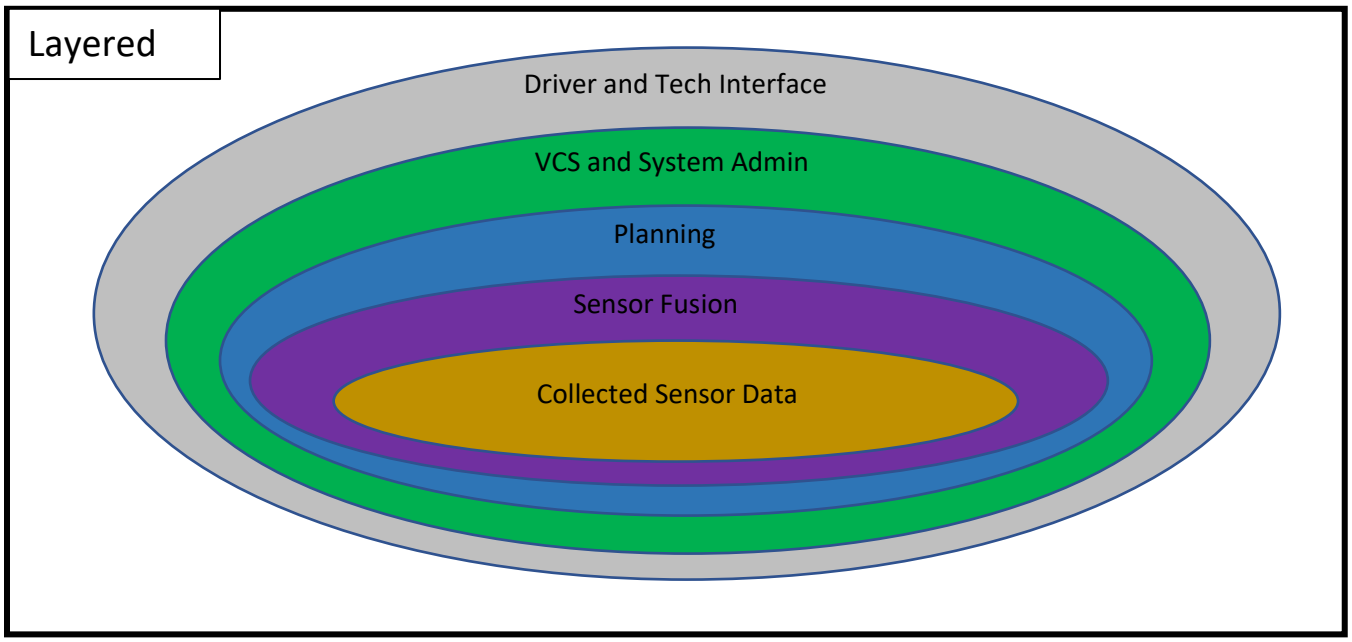
b. Cons
   i. Overly rigid structure can make it difficult to slot in new procedures.
   ii. Time efficiency is not on par with other architectures for real-time systems.
   iii. Limits parallel execution.

c. Overall
   i. Call & Return could be utilized for software such as the console user interface, however it would not be viable for the real-time nature of IoT HTL.

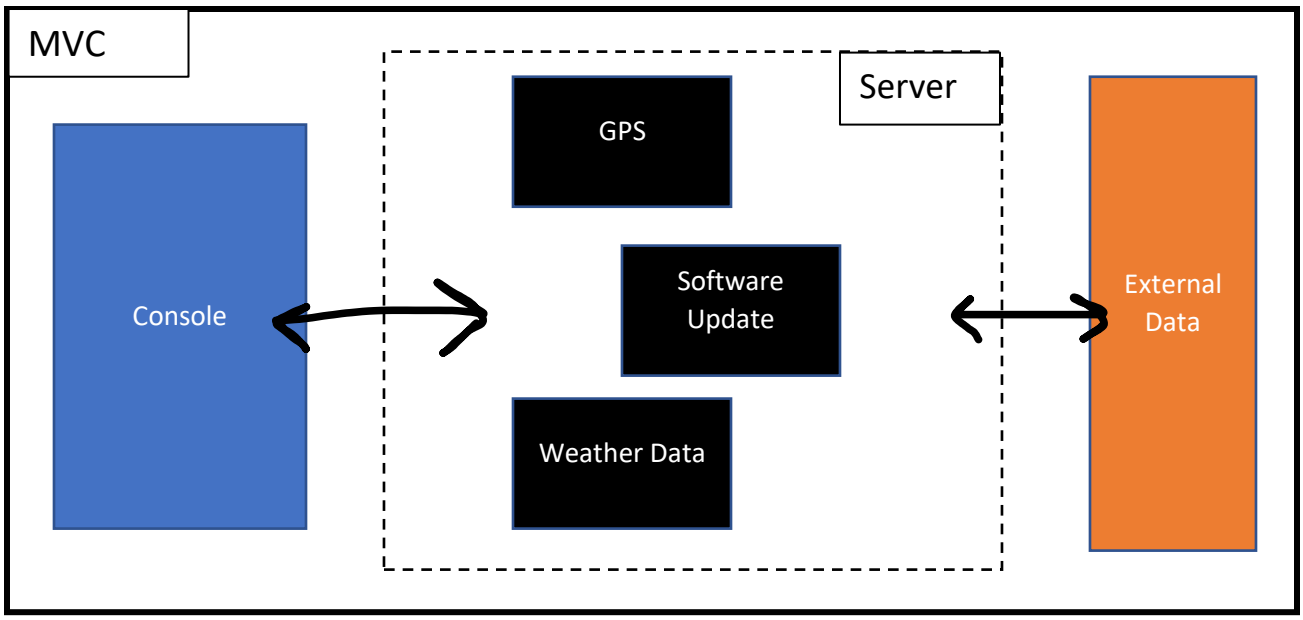


4. Object Oriented
    a. Pros
        i. Modular and easy to update.
        ii. Code is reusable across different components.
        iii. Message passing to communicate and coordinate between components.
    b. Cons
        i. Performance and time efficiency issues due to object overhead.
        ii. Overly complex compared to other architectures.
    c. Overall
        i. Could be useful in lowering the amount of coding needed but might not be viable for IoT HTL due to its complexity and performance.
        ii. Could be used in technician login system.
5. Layered
    a. Pros
        i. Separates different responsibilities into different regions of the architecture.
        ii. Code is flexible and reuseable as core systems can be used by many other applications.
        iii. Can make decisions quickly.
    b. Cons
        i. Complex to navigate between layers of the system.
        ii. Too many layers can cause performance or complexity problems.

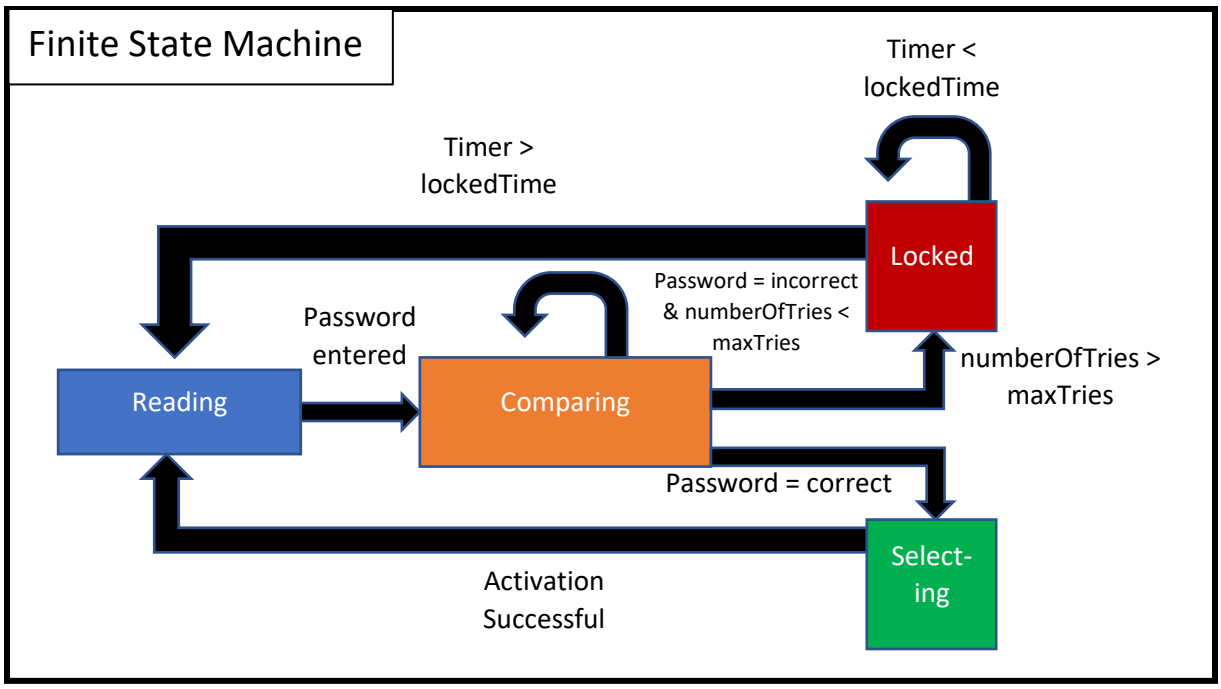
c. Overall
    i. Layered architecture is not as viable as other options due to the lack of the need for layered processes.



6. Model View Controller
    a. Pros
        i. Great for web-development and user-interface.
        ii. Allows for concurrency of programs.
    b. Cons
        i. Large amount of complexity and overhead.
        ii. Time efficiency is not on par with other architectures for real-time systems.
        iii. Limited use outside of user-interface and server interaction
    c. Overall
        i. Not viable for IoT HTL due to its focus on server interaction and time efficiency.
        ii. Viable for the Console interface.



7. Finite State Machine
    a. Pros
        i. Simple and predictable logic that is easy to visualize.
        ii. Fault tolerant and easy to debug or update.
    b. Cons
        i. Can become overly complex when on a large scale.
        ii. Not viable for dynamic changes in behavior that is required for driving a car.
    c. Overall
        i. Viable for IoT HTL technician login.
        ii. Not viable for self-driving due to the lack of concrete states.



8. Final Verdicts
    a. The core functionality for IoT HTL will use a Data Flow architecture.
    b. The Technician Login will use a finite state machine architecture.
    c. The Console will use a Model View Controller architecture, specifically for features that require the internet like software updates and GPS.

5.2: Interface Design

The Driver will interact with the Car through various methods, turning their interface into a command-and-control center. Interfaces like the Instrument Panel, Console, Brake Pedal, Accelerator, Steering Wheel and Parking Brake will function as the Driver's main method of interacting with the Car.

1. Instrument Panel
    a. Requests information from sensor fusion:
        i. Signals from sensors
        ii. Speed information
    b. Provides information:
        i. Speed
        ii. Battery Level
        iii. Basic Navigation Instructions
        iv. Hazard Indicators (motor malfunctions, battery level warnings, sensor malfunctions)

2. Console:
    a. Requests information from Sensor Fusion:
        i. Camera logs
        ii. GPS data
    b. Provides comprehensive information:
        i. Camera feeds
        ii. Media
        iii. Weather hazards and reports
        iv. More complete Navigation Data
        v. Drive mode strip for switching "gears" (park, drive, reverse)
    c. Special employee login for Technician:
        i. Lower-level control of Car
        ii. Access to operating system for repairs
        iii. Provide system updates and patches
3. Traditional Controls:
    a. Steering Wheel
    b. Brake Pedal
    c. Accelerator
    d. Parking Brake

5.3: Component-Level Design

1. Movement Component
    a. Description
        i. Always active when the car is in motion, responsible for storing/modifying vehicle's trajectory. Not responsible for any high-level decisions, just low-level actions to modify the car's movement. Constantly waiting for other components to call relevant functions.
    b. Functions
        i. changeMode(int mode) – changes car to drive, reverse, park, etc.
        ii. brake(int amt) – modularly applies breaks to slow car down dependent upon driver pedal pressure or desired amount should car be in self-driving mode
        iii. accelerate(int amt) - modularly applies acceleration to slow car down dependent upon driver pedal pressure or desired amount should car be in self-driving mode
        iv. turnWheel(int degree) - turns the wheel to the specified angle
2. Self-Driving Component
    a. Description

i. Created/Active when user presses button on console to activate self-driving mode. Is called by Planning Component, and calls Movement Component.

b. Functions

i. setSpeed(int desired) – checks if current speed (pulled from Sensor Component) is at desired, if not accelerates/breaks respectively to arrive at desired speed.

ii. adjustLane(int target, int current) – checks if vehicle is in target lane, if not calls upon Movement Component to transition into desired lane

iii. autoPark(int spotID, bool rearFacing) – Planning component identifies a spot and calls this method to do needed movements to park. rearFacing determines if the car back into the spot or goes in forwards.

3. Assisted-Driving Component

a. Description

i. Created/Active when user deactivates Self-Driving mode, and by default when the car starts. Calls upon Movement Component to navigate vehicle.

b. Functions

i. toggleCruiseControl(int speed) – toggles cruise control, if activating then a desired speed is inputted. Else -1 is inputted for speed. Calls upon it's own setSpeed method which calls upon the Movement Component

ii. setSpeed(int desired) – only activated by the toggleCruiseControl method, or when user changes cruise control speed on the Console Component. Checks if current speed (pulled from Sensor Component) is at desired, else calls Movement Component to reach desired.

iii. Maneuver() – called whenever pedals/steering wheel are touched, calls Movement Component to change trajectory of vehicle

4. Console Component

a. Description

i. Always active in the vehicle, responsible for interfacing with the humans inside the vehicle. Pulls from Planning Component to display relevant information.

b. Functions

i. inputDestination(String addr) – user inputs address and console sends information to Planning Component to calculate the route for the vehicle

ii. addStop(String addr) – user inputs additional address to console, which sends information to Planning so route can be modified as needed.

iii. displayRoute() – pulls from Planning every tenth of a second to actively display the route the vehicle is traveling on.
iv. ccControls(int speed) – when cruise control is active, display a setting to change speed as needed. When it's pressed, calls upon Assisted Driving Component to change speed.
v. displayETA() - Retrieves estimated time of arrival (ETA) from the Planning Component and displays it on the console, providing passengers with real-time updates on when they will reach their destination.

5. Planning Component
    a. Description
        i. The planning part of self-driving cars decides the route to take, making sure to avoid hitting anything, staying in the right lane, and following traffic rules. It adjusts plans as needed to stay safe and get where it's going efficiently
    b. Functions
        i. getGoogleMapsData(location_t location) - sends the coordinates or address of the place the driver inputs into the GPS and returns the most optimal route
        ii. aggregateSensorData(sensordata_t sd) - processes the data recieved from the sensors and makes adjustments to the current route
            1. This could include changing lanes, handling intersections, complying with traffic laws, adapting to dynamic environments, and reacting to emergency situations
6. Security Component
    a. Description
        i. Protects the vehicle's systems and data from unauthorized access, tampering, or malicious attacks.
    b. Functions
        i. enableLockdownMode(void) - enables lockdown mode, which hardens the operating system and requires two factor authentication for drivers to access the vehicle
        ii. encrypt(key_t key) - encrypts the data that is sent to the servers
        iii. decrypt(key_t key) - decrypts the data that is received from the servers
        iv. is_locked(void) – Checks the boolean value "locked" if the vehicle is locked or not
        v. validateUsers(user_t user) – Checks if user is in the array of valid users
7. Sensor Component
    a. Description

i. These sensors capture information from the surrounding environment and the vehicle itself, gathering the information needed for the vehicle to make informed decisions about navigation, obstacle avoidance, and overall vehicle control.

b. Functions
   i. captureImage(image[][]) - Captures an image from the vehicle's cameras.
   ii. CaptureLidarData(lidarData_t LidarData) - Captures lidar data, providing information about the surrounding environment
   iii. CaptureRadarData(radarData_t RadarData) - Captures radar data, which includes velocity, distance, and angle information about objects near the vehicle.
   iv. ReadGPS(gpsData_t GPSData) - Reads GPS coordinates, providing information about the vehicle's latitude and longitude
   v. ReadTrafficLightRecognition(image[][]) - Reads data from the traffic light recognition system, providing information about detected traffic lights such as color and state.
   vi. ReadLaneMarkings(image[][]) - Reads data about lane markings, providing information about their position and curvature

8. Redundancy and Fail-Safe Component
   a. Description
      i. To ensure safety, our vehicle will incorporate redundant systems and fail-safe mechanisms. These systems include backup sensors, redundant control systems, and strategies for safely stopping or pulling over in the event of a malfunction.
   b. Functions
      i. verifySensorOperations(void) - checks if there are any sensors that are not working. If there are sensors that do not work, the backup sensors will be activated
      ii. getOperationScore(void) - analyzes all the working sensors and other safety components in the vehicle and returns a score out of 100, indicating the level of operationality of the safety mechanisms
      iii. panic(int op_score) - accepts the output of the above function and pulls over it the score is less than 40; sets the maximum speed to 40 mph if the score is between 40 and 70; and displays a warning if the score is between 70 and 90

Section 6: Code

1.6.1 Requirement Code

1.6.1.1 Automatic Braking

```python
#If the car detects that it will hit an object in front of it,
#the car will automatically apply the brakes to avoid the collision.
#Speeds are in m/s, distances are in meters, and time is in seconds.
def automatic_breaking(car_speed: float, object_detected_distance: float, road_is_slick: bool) -> bool:
    if car_speed > 0:
        time_to_collision = object_detected_distance / car_speed
        if road_is_slick:
            time_to_collision = time_to_collision / 1.5
        if time_to_collision <= 3:
            print("Potential collision detected! Automatic breaking initiated.")
            return True
    return False
```

1.6.1.2 Driver Assisted Steering Correction

```python
# In an area with clearly defined lanes, if the car is in Assisted-Driving
# mode and the driver is beginning to veer from the lane unintentionally,
# the car will notify the driver and hold the Vechicle steady.
# direction_of_creep is a float between -1 and 1 (-1 is left, 0 is straight, 1 is right)
def driver_assisted_steering_correction(car_speed: float, turn_signal: bool, out_of_lane: bool, direction_of_creep: float) -> float:
    if car_speed > 0 and not turn_signal and out_of_lane:
        if direction_of_creep < -0.25:
            print("Driver assisted steering correction initiated: steering right.")
            return -direction_of_creep
        elif direction_of_creep > 0.25:
            print("Driver assisted steering correction initiated: steering left.")
            return -direction_of_creep
    return 0
```

1.6.1.3 Charging Station Navigation

```python
# The car will always be able to locate all the charging stations that can be reached with its current battery level
def charging_station_navigation(battery_percentage: float, driver_prompt: bool, \
                                miles_remaining: float, distances_to_charging_stations: list) -> list:
    if battery_percentage < 15 or driver_prompt:
        vdistances = []
        for distance in distances_to_charging_stations:
            if distance <= miles_remaining:
                vdistances.append(distance)
        vdistances.sort()
        return vdistances
    return []
```

1.6.1.4 Assisted Driving to Self-Driving Transition

```python
# When prompted by the driver, the vehicle should automatically gather necessary
# navigation data and transistion to Self-Driving mode if a destination is set.
def assisted_driving_to_self_driving_transition(mode: str, destination_set: bool, can_transition: bool) -> bool:
    if mode == "assisted-driving" and destination_set and can_transition:
        mode = "self-driving"
        print("Transitioning from assisted driving to self-driving.")
        return True
    if not destination_set:
        print("Destination not set, cannot transition.")
    elif not can_transition:
        print("Transition currently not possible.")
    return False
```

1.6.1.5 Parking Assistance

```python
# The Car will detect when the driver wants to park
# and can park by itself if the driver wishes.
def parking_assistance(speed: float, nearby_parking_spaces: bool, user_authorization: bool, mode: str) -> bool:
    if speed <= 10 and nearby_parking_spaces and user_authorization and mode == "assisted-driving":
        print("Parking assistance initiated.")
        mode = "self-parking"
        return True
    return False
```

1.6.1.6 Route Plotting

```python
# When a location is put into the console, the car will plot an optimal route
# to the location given GPS data and local traffic data.
def route_plotting(destination: str, gps: list, traffic: str) -> list:
    # find_routes simulates the car calculating all possible routes to a
    # destination and returns a list of times in minutes to display to the driver
    if destination != "":
        routes = find_routes(destination, gps, traffic)
        routes.sort()
        for i in range(3):
            print("Route ", i+1, ": ", routes[i], " minutes.")
        return routes[:3]
    return []
```

1.6.1.7 Emergency Pullover

```python
# Should the driver press a designated "Emergency" button on the console, the
# vehicle will immediately take control and attempt to pull-over at a safe location
def emergency_pullover(mode: str, distance_to_pullover: float) -> bool:
    mode = "self-driving"
    print("Attempting to pull over to location in ", distance_to_pullover, " meters.")
    return True
```

1.6.1.8 Emergency Vehicle Detection and Response

```python
# Upon detection of emergency vehicles, if the car is in Self-Driving mode
# it will immediately notify the driver and attempt to pull over
def emergency_vehicle_detection_and_response(speed: float, mode: str, emergency_vehicle_detected: bool) -> bool:
    if speed > 0 and mode == "self-driving" and emergency_vehicle_detected:
        print("Emergency vehicle detected! Initiating pull over.")
        return True
    return False
```

1.6.1.9 Crash Detection

```python
# If the car detects that it has been in an accident,
# it will stop and contact emergency services.
def crash_detection(potential_collision_detected: bool, speed: float, acceleration: float) -> bool:
    if potential_collision_detected and speed == 0 and acceleration < -20:
        print("Crash detected! Halting all operations and contacting emergency services.")
        return True
    return False
```

1.6.1.10 Self-Driving to Assisted Driving Transition

```python
# Car correctly interprets all driver steering commands
# and disables Self-Driving if it is active.
def self_driving_to_assisted_driving_transition(speed: float, mode: str, driver_input: bool):
    if speed > 0 and mode == "self-driving" and driver_input:
        mode = "assisted-driving"
        print("Driver control resumed. Turning off Self-Driving.")
        return True
    return False
```

1.6.2 Use Case Code

1.6.2.1 Assisted Driving to Self-Driving

```python
# When prompted by the driver, the vehicle should automatically gather necessary
# navigation data and transistion to Self-Driving mode if a destination is set.
def assisted_driving_to_self_driving_transition(mode: str, destination_set: bool, can_transition: bool) -> bool:
    if mode == "assisted-driving" and destination_set and can_transition:
        mode = "self-driving"
        print("Transitioning from assisted driving to self-driving.")
        return True
    if not destination_set:
        print("Destination not set, cannot transition.")
    elif not can_transition:
        print("Transition currently not possible.")
    return False
```

1.6.2.2 Route Planning

```
# When a location is put into the console, the car will plot an optimal route
# to the location given GPS data and local traffic data.
def route_plotting(destination: str, gps: list, traffic: str) -> list:
    # find_routes simulates the car calculating all possible routes to a
    # destination and returns a list of times in minutes to display to the driver
    if destination != "":
        routes = find_routes(destination, gps, traffic)
        routes.sort()
        for i in range(3):
            print("Route ", i+1, ": ", routes[i], " minutes.")
        return routes[:3]
    return []
```

1.6.2.3 Automatic Braking

```
#If the car detects that it will hit an object in front of it,
#the car will automatically apply the brakes to avoid the collision.
#Speeds are in m/s, distances are in meters, and time is in seconds.
def automatic_breaking(car_speed: float, object_detected_distance: float, road_is_slick: bool) -> bool:
    if car_speed > 0:
        time_to_collision = object_detected_distance / car_speed
        if road_is_slick:
            time_to_collision = time_to_collision / 1.5
        if time_to_collision <= 3:
            print("Potential collision detected! Automatic breaking initiated.")
            return True
    return False
```

1.6.2.4 Key Fob Auto Lock and Unlock

```
# If the car is locked and the key fob is brought within 5 feet of the car, the car will unlock.
# If the car is unlocked and the key fob is taken more than 5 feet away, the car will lock.
def key_fob_auto_lock_unlock(new_key_distance: float, old_key_distance: float, locked: bool) -> bool:
    if locked and new_key_distance <= 5 and old_key_distance > 5:
        locked = False
        return False
    elif not locked and new_key_distance > 5 and old_key_distance <= 5:
        locked = True
        return True
    return locked
```

1.6.2.5 Technician Login

```
# Technichian login. If an incorrect username or password is entered 3 times, the login will lock.
# Returns true on unlock, false on failed attempt.
# Note: default login, password is Milind, KDog2004
def technichian_login(username: str, password: str, trys: int = 0):
    if trys >= 3:
        print("Login locked.")
        return False
    if username == CONST_USERNAME and password == CONST_PASSWORD:
        return True
    return False
```

Section 7: Testing

1.7.1 Validation Testing

1.7.1.1 Automatic Breaking

```
def test_automatic_breaking():
    # If the car is moving at 50 m/s, there is an object 100 meters in front of it,
    # and the road is not slick, the car should break.
    assert automatic_breaking(50, 100, False) == True
    # If the car is moving at 30 m/s, there is an object 100 meters in front of it,
    # and the road is slick, the car should break.
    assert automatic_breaking(30, 100, True) == True
    # If the car is moving at 50 m/s, there is an object 300 meters in front of it,
    # and the road is not slick, the car should not break.
    assert automatic_breaking(50, 300, False) == False
    # If the car is not moving, there is an object 100 meters in front of it,
    # and the road is not slick, the car should not break.
    assert automatic_breaking(0, 100, False) == False
    # If the car is moving at 50 m/s, there is an object 10 meters in front of it,
    # and the road is not slick, the car should break.
    assert automatic_breaking(50, 10, False) == True
```

1.7.1.2 Driver Assisted Steering Correction

```
def test_driver_assisted_steering_correction():
    # If the car is moving at 50 m/s, the driver is not using the turn signal,
    # the car is out of lane, and the driver is veering left, the car should steer right.
    assert driver_assisted_steering_correction(50, False, True, -0.5) == 0.5
    # If the car is moving at 50 m/s, the driver is not using the turn signal,
    # the car is out of lane, and the driver is veering right, the car should steer left.
    assert driver_assisted_steering_correction(50, False, True, 0.5) == -0.5
    # If the car is moving at 50 m/s, the driver is not using the turn signal,
    # the car is not out of lane, and the driver is not veering, the car should not steer.
    assert driver_assisted_steering_correction(50, False, False, 0) == 0
    # If the car is moving at 0 m/s, the driver is not using the turn signal,
    # the car is not out of lane, and the driver is not veering, the car should not steer.
    assert driver_assisted_steering_correction(0, False, False, 0) == 0
    # If the car is moving at 50 m/s, the driver is using the turn signal,
    # the car is out of lane, and the driver is veering right, the car should not steer.
    assert driver_assisted_steering_correction(50, True, True, 0.5) == 0
```

1.7.1.3 Charging Station Navigation

```
def test_charging_station_navigation():
    # If the battery is at 50%, the driver prompts the console, the car can go 250 miles,
    # and the distances to the charging stations are [10, 20, 30, 40, 50], the car should return all the distances.
    assert charging_station_navigation(50, True, 250, [10, 20, 30, 40, 50]) == [10, 20, 30, 40, 50]
    # If the battery is at 5%, the driver does not prompt the console, the car can go 25 miles,
    # and the distances to the charging stations are [10, 20, 30, 50, 40], the car should return [10, 20].
    assert charging_station_navigation(5, False, 25, [20, 10, 30, 50, 40]) == [10, 20]
    # If the battery is at 50%, the driver does not prompt the console, the car can go 250 miles,
    # and the distances to the charging stations are [20, 30, 22, 10, 55], the car should return [].
    assert charging_station_navigation(50, False, 250, [20, 30, 22, 10, 55]) == []
    # If the battery is at 14%, the driver prompts the console, the car can go 70 miles,
    # and the distances to the charging stations are [10, 22, 33, 44, 77, 88], the car should return [10, 22, 33, 44].
    assert charging_station_navigation(14, True, 70, [10, 22, 33, 44, 77, 88]) == [10, 22, 33, 44]
    # If the battery is at 8%, the driver does not prompt the console, the car can go 40 miles,
    # and the distances to the charging stations are [11, 35, 22, 41, 56, 70], the car should return [11, 22, 35].
    assert charging_station_navigation(8, False, 40, [11, 35, 22, 41, 56, 70]) == [11, 22, 35]
```

1.7.1.4 Assisted Driving to Self-Driving Transition

```python
def test_assisted_driving_to_self_driving_transition():
    # If the car is in Assisted-Driving mode, the destination is set,
    # and the car can transition, the car should transition to Self-Driving.
    assert assisted_driving_to_self_driving_transition("assisted-driving", True, True) == True
    # If the car is in Assisted-Driving mode, the destination is not set,
    # and the car can transition, the car should not transition to Self-Driving.
    assert assisted_driving_to_self_driving_transition("assisted-driving", False, True) == False
    # If the car is in Assisted-Driving mode, the destination is set,
    # and the car can not transition, the car should not transition to Self-Driving.
    assert assisted_driving_to_self_driving_transition("assisted-driving", True, False) == False
    # If the car is in Self-Driving mode, the destination is set,
    # and the car can transition, the car should not transition to Self-Driving.
    assert assisted_driving_to_self_driving_transition("self-driving", True, True) == False
    # If the car is in Self-Driving mode, the destination is not set,
    # and the car can transition, the car should not transition to Self-Driving.
    assert assisted_driving_to_self_driving_transition("self-driving", False, False) == False
```

1.7.1.5 Parking Assistance

```python
def test_parking_assistance():
    # If the car is moving at 10 m/s, there are nearby parking spaces, the user has authorized parking,
    # and the car is in Assisted-Driving mode, the car should park itself.
    assert parking_assistance(10, True, True, "assisted-driving") == True
    # If the car is moving at 7 m/s, there are nearby parking spaces, the user has not authorized parking,
    # and the car is in Self-Driving mode, the car should not park itself.
    assert parking_assistance(7, True, False, "self-driving") == False
    # If the car is moving at 8 m/s, there are nearby parking spaces, the user has not authorized parking,
    # and the car is in Assisted-Driving mode, the car should not park itself.
    assert parking_assistance(8, True, False, "assisted-driving") == False
    # If the car is moving at 9 m/s, there are no nearby parking spaces, the user has authorized parking,
    # and the car is in Assisted-Driving mode, the car should not park itself.
    assert parking_assistance(9, False, True, "assisted-driving") == False
    # If the car is moving at 20 m/s, there are nearby parking spaces, the user has authorized parking,
    # and the car is in Assisted-Driving mode, the car should not park itself.
    assert parking_assistance(20, True, True, "assisted-driving") == False
```

1.7.1.6 Route Plotting

```python
def test_route_plotting():
    # If the destination is "Home", the car should return [4, 6, 9].
    assert route_plotting("Home", "GPSData", "WeatherData") == [4, 6, 9]
    # If the destination is "Work", the car should return [22, 24, 28].
    assert route_plotting("Work", "GPSData", "WeatherData") == [22, 24, 28]
    # If the destination is "School", the car should return [13, 14, 17].
    assert route_plotting("School", "GPSData", "WeatherData") == [13, 14, 17]
    # If the destination is "Store", the car should return [34, 44, 48].
    assert route_plotting("Store", "GPSData", "WeatherData") == [34, 44, 48]
    # If the destination is not set, the car should return [].
    assert route_plotting("", "GPSData", "WeatherData") == []
```

1.7.1.7 Emergency Pullover

```
def test_emergency_pullover():
    # If the car is in Assisted-Driving mode and the distance to pull over is 100 meters,
    # the car should pull over.
    assert emergency_pullover("assisted-driving", 100) == True
    # If the car is in Self-Driving mode and the distance to pull over is 100 meters,
    # the car should pull over.
    assert emergency_pullover("self-driving", 100) == True
    # If the car is in Assisted-Driving mode and the distance to pull over is 10 meters,
    # the car should pull over.
    assert emergency_pullover("assisted-driving", 10) == True
    # If the car is in Self-Driving mode and the distance to pull over is 20 meters,
    # the car should pull over.
    assert emergency_pullover("self-driving", 20) == True
    # If the car is in Assisted-Driving mode and the distance to pull over is 50 meters,
    # the car should pull over.
    assert emergency_pullover("assisted-driving", 50) == True
```

1.7.1.8 Emergency Vehicle Detection and Response

```
def test_emergency_vehicle_detection_and_response():
    # If the car is moving at 55 m/s, is in Self-Driving mode,
    # and an emergency vehicle is detected, the car should pull over.
    assert emergency_vehicle_detection_and_response(55, "self-driving", True) == True
    # If the car is moving at 69 m/s, is in Assisted-Driving mode,
    # and an emergency vehicle is detected, the car should not pull over.
    assert emergency_vehicle_detection_and_response(69, "assisted-driving", True) == False
    # If the car is not moving, is in Self-Driving mode,
    # and an emergency vehicle is detected, the car should not pull over.
    assert emergency_vehicle_detection_and_response(0, "self-driving", True) == False
    # If the car is moving at 13 m/s, is in Self-Driving mode,
    # and an emergency vehicle is not detected, the car should not pull over.
    assert emergency_vehicle_detection_and_response(13, "self-driving", False) == False
    # If the car is moving at 24 m/s, is in Assisted-Driving mode,
    # and an emergency vehicle is not detected, the car should not pull over.
    assert emergency_vehicle_detection_and_response(24, "assisted-driving", False) == False
```

1.7.1.9 Crash Detection

```
def test_crash_detection():
    # If the car detects a potential collision, is moving at 10 m/s,
    # and has an acceleration of 0m/s^2, the car should not alert.
    assert crash_detection(True, 10, 0) == False
    # If the car detects a potential collision, is moving at 0 m/s,
    # and has an acceleration of -4m/s^2, the car should not alert.
    assert crash_detection(True, 0, -4) == False
    # If the car detects a potential collision, is moving at 0 m/s,
    # and has an acceleration of -100m/s^2, the car should alert.
    assert crash_detection(True, 0, -100) == True
    # If the car does not detect a potential collision, is moving at 10 m/s,
    # and has an acceleration of -21m/s^2, the car should not alert.
    assert crash_detection(True, 10, -21) == False
    # If the car does not detect a potential collision, is moving at 0 m/s,
    # and has an acceleration of -21m/s^2, the car should not alert.
    assert crash_detection(False, 0, -21) == False
```

1.7.1.10 Self Driving to Assisted Driving Transition

```
def test_self_driving_to_assisted_driving_transition():
    # If the car is moving at 30 m/s, is in Self-Driving mode,
    # and the driver prompts the console, the car should disable Self-Driving.
    assert self_driving_to_assisted_driving_transition(30, "self-driving", True) == True
    # If the car is moving at 60 m/s, is in Self-Driving mode,
    # and the driver does not prompt the console, the car should not disable Self-Driving.
    assert self_driving_to_assisted_driving_transition(60, "self-driving", False) == False
    # If the car is moving at 0 m/s, is in Self-Driving mode,
    # and the driver prompts the console, the car should not disable Self-Driving.
    assert self_driving_to_assisted_driving_transition(0, "self-driving", True) == False
    # If the car is moving at 20 m/s, is in Assisted-Driving mode,
    # and the driver prompts the console, the car should not disable Self-Driving.
    assert self_driving_to_assisted_driving_transition(20, "assisted-driving", True) == False
    # If the car is moving at 50 m/s, is in Assisted-Driving mode,
    # and the driver does not prompt the console, the car should not disable Self-Driving.
    assert self_driving_to_assisted_driving_transition(50, "assisted-driving", False) == False
```

1.7.2 Scenario-Based Testing

1.7.2.1 Assisted Driving to Self-Driving

```
def test_assisted_driving_to_self_driving_transition():
    # If the car is in Assisted-Driving mode, the destination is set,
    # and the car can transition, the car should transition to Self-Driving.
    assert assisted_driving_to_self_driving_transition("assisted-driving", True, True) == True
    # If the car is in Assisted-Driving mode, the destination is not set,
    # and the car can transition, the car should not transition to Self-Driving.
    assert assisted_driving_to_self_driving_transition("assisted-driving", False, True) == False
    # If the car is in Assisted-Driving mode, the destination is set,
    # and the car can not transition, the car should not transition to Self-Driving.
    assert assisted_driving_to_self_driving_transition("assisted-driving", True, False) == False
    # If the car is in Self-Driving mode, the destination is set,
    # and the car can transition, the car should not transition to Self-Driving.
    assert assisted_driving_to_self_driving_transition("self-driving", True, True) == False
    # If the car is in Self-Driving mode, the destination is not set,
    # and the car can transition, the car should not transition to Self-Driving.
    assert assisted_driving_to_self_driving_transition("self-driving", False, False) == False
```

1.7.2.2 Route Planning

```
def test_route_plotting():
    # If the destination is "Home", the car should return [4, 6, 9].
    assert route_plotting("Home", "GPSData", "WeatherData") == [4, 6, 9]
    # If the destination is "Work", the car should return [22, 24, 28].
    assert route_plotting("Work", "GPSData", "WeatherData") == [22, 24, 28]
    # If the destination is "School", the car should return [13, 14, 17].
    assert route_plotting("School", "GPSData", "WeatherData") == [13, 14, 17]
    # If the destination is "Store", the car should return [34, 44, 48].
    assert route_plotting("Store", "GPSData", "WeatherData") == [34, 44, 48]
    # If the destination is not set, the car should return [].
    assert route_plotting("", "GPSData", "WeatherData") == []
```

1.7.2.3 Automatic Breaking

```
def test_automatic_breaking():
    # If the car is moving at 50 m/s, there is an object 100 meters in front of it,
    # and the road is not slick, the car should break.
    assert automatic_breaking(50, 100, False) == True
    # If the car is moving at 30 m/s, there is an object 100 meters in front of it,
    # and the road is slick, the car should break.
    assert automatic_breaking(30, 100, True) == True
    # If the car is moving at 50 m/s, there is an object 300 meters in front of it,
    # and the road is not slick, the car should not break.
    assert automatic_breaking(50, 300, False) == False
    # If the car is not moving, there is an object 100 meters in front of it,
    # and the road is not slick, the car should not break.
    assert automatic_breaking(0, 100, False) == False
    # If the car is moving at 50 m/s, there is an object 10 meters in front of it,
    # and the road is not slick, the car should break.
    assert automatic_breaking(50, 10, False) == True
```

1.7.2.4 Key Fob Auto Lock and Unlock

```
def test_key_fob_auto_lock_unlock():
    # If the new key distance is 4 feet, the old key distance is 12 feet,
    # and the car is locked, the car should be unlocked.
    assert key_fob_auto_lock_unlock(4, 12, True) == False
    # If the new key distance is 18 feet, the old key distance is 9 feet,
    # and the car is unlocked, the car should stay unlocked.
    assert key_fob_auto_lock_unlock(18, 9, False) == False
    # If the new key distance is 3 feet, the old key distance is 11 feet,
    # and the car is unlocked, the car should stay unlocked.
    assert key_fob_auto_lock_unlock(3, 11, False) == False
    # If the new key distance is 12 feet, the old key distance is 5 feet,
    # and the car is locked, the car should stay locked.
    assert key_fob_auto_lock_unlock(12, 5, True) == True
    # If the new key distance is 10 feet, the old key distance is 10 feet,
    # and the car is locked, the car should stay locked.
    assert key_fob_auto_lock_unlock(10, 10, True) == True
```

1.7.2.5 Technician Login

```
def test_technichian_login():
    # If the correct username and password are entered, the car should unlock.
    assert technichian_login("Milind", "KDog2004") == True
    # If more than 3 incorrect attempts are made, the console should not unlock.
    assert technichian_login("Milind", "KDog2004", 4) == False
    # If an incorrect username and password are entered, the console should not unlock.
    assert technichian_login("Nilind", "KCat2004") == False
    # If the correct username and password are entered, the car should unlock.
    assert technichian_login("Milind", "KDog2004", 1) == True
    # If an incorrect username or password are entered, the console should not unlock.
    assert technichian_login("Milind", "KDog2003", 2) == False
```