York University

EECS4080 Capstone Project

**Design Document**

**A black and white machine with wheels

Description automatically generated**

| Version: | 1.0 |
| --- | --- |
| Print Date: | October 4, 2023 |
| Release Date: | October 10, 2023 |
| Release State: | D1 |
| Approval State: |  |
| Approved by: |  |
| Prepared by: | Samir, Chirayu |
| Reviewed by: |  |
| Path Name: |  |
| File Name: |  |
| Document No: |  |

**Document Change Control**

| **Version** | **Date** | **Authors** | **Summary of Changes** |
| --- | --- | --- | --- |
| 1.0 | October 4, 2023 | All members | Insert Gantt, sequence, UML, use case, and component diagrams |
| 1.1 | October 8, 2023 | All members | Revised UML diagrams and updated sequence diagrams |
| 1.2 | October 12, 2023 | All members | Added detailed descriptions to use case diagrams |
| 2.0 | October 18, 2023 | All members | Major revision of all diagrams and introduction of new component diagrams |
| 2.1 | October 24, 2023 | All members | Implemented adapter design pattern for exposing car data via api |
| 2.2 | October 28, 2023 | All members | Implemented rest api for car data |
| 2.3 | November 5th, 2023 | All members | Implemented state design pattern and factory for edge computing |
| 2.4 | November 8th, 2023 | All members | Implemented edge computing servers and other design patterns to complete the code |
| 2.5 & 2.6 | November 10th, 2023 | All members | Updated document with sequence, component, uml, design patterns, modules, interfaces, and research |

**Document Sign-Off**

| **Name (Position)** | **Signature** | **Date** |
| --- | --- | --- |
| Samir Gandhi | SG | November 15th, 2023 |
| Chirayu Parmar | CP | November 105h, 2023 |

**Contents**

**1** **Introduction 4**

1.1 Purpose 4

1.2 What is edge computing and how does it work? 4

1.3 What is Vehicle-to-vehicle communication: 7

1.4 What is an outlier and how it works? 8

1.5 How can other people access the data of the car? 9

**2** **Major Design Decisions 10**

**3** **UML Diagram 12**

**4** **Sequence Diagrams 14**

**5** **Architecture 14**

**6** **Activities Plan 21**

6.1 Group Meeting Logs 21

**7** **Bibliography 22**

# Introduction

## Purpose

This EECS4080 project involves creating a software layer for the Quanser QCar. The main objective of this project is to leverage the capabilities of edge computing for vehicle-to-vehicle communication and expose data recorded by the QCar to the public using a well-defined API for efficient data processing and transmission. Any anomaly (outlier data) recorded by QCar due to malfunction of the sensors removed by the anomaly detector and only trusted data will be transferred to the public.

The design of this capstone project leverages the Quanser QCars’s data handling capabilities. The objectives are to lower latency, improve data processing rates and ensure dependable data transfer by incorporating edge computing and trust. The accessible API will offer a standard way of exchanging data and enabling smooth interaction with the public (external users).

Our initial research led us to realize Quanser’s Qcar, was in its early stages of development, Quanser's technology relies on internet-based connectivity, which limits the operational abilities, allowing for data transfer only within a small range.

The scope of this project includes a detailed overview of the system, highlighting its functionalities, architectural and software design and the interaction of its various components.

The main goal of this project was to implement the architectural framework for this car to allow autonomous vehicle-to-vehicle communication using edge computing protocol and maintain trust by sharing data recorded by the car with the public and removing outliers. Therefore, we started the project by implementing the mock-up data on JAVA IDE for development and testing purposes. Due to time constraints, we didn’t receive much information from Quasar regarding the QCar API connection. As a result, we don’t have a code running on QCar and Quasar Server. Instead, we have a client-server application where the client is a car, and the server is a Quasar Server.

## What is edge computing and how does it work?

Edge computing is a distributed computing model that brings the processing and storage of data storage closer to its data generator. Edge computing allows us to analyze the data generated at the edge of the network. This reduces the latency and energy consumption in comparison to the scenario where data is sent to distant cloud servers for processing. [1]

Edge computing can also be intertwined with Artificial Intelligence technologies. Using AI with edge computing would give us the ability to have adaptive, efficient handling of tasks such as resource allocation, and predictive analytics on the data being generated. [2]

Moreover, this framework helps to increase response time, and bandwidth availability, and reduce the need for large data from different resources to travel over the network to reach one server. Edge Computing and 5G Mobile Edge Computing provide wide-ranging data analysis, faster response time, and better customer experiences. [1]

In edge computing, vendors can also be assigned to the edge and can extend the services by increasing the traffic handling capacity, better performance, and improving security.

A diagram of a cloud server

Description automatically generated

Source: Adapted from [1]

Since edge computing significantly reduces the data stress on each server, the servers' performance is significantly better. When one central server is used, and every car sends data to that server, there is heavy stress on the server. Splitting the servers based on location and optimizing edge computing allows certain cars to be connected only to certain servers. This reduces the data load and makes processing times faster. Edge computing can be scaled largely without many changes to the current system. In our code, we have implemented 5 servers and can add many more edges by just adding them to the database and initializing them. No changes would be required to be made to the client side.

Edge computers also provide security if there is any Distributed Denial of Service (DDos). In the case of DDoS there is a chance that only one edge server will be affected. To overcome this issue, each edge server can be allocated to separate agents. These agents can provide security as well as maintenance.

A computer screen shot of a program code

Description automatically generated

Our Algorithm and design patterns allow our car to receive the best edge to connect to in addition to its nearby edges. This allows our car to send data to a few servers and only receive a response from the best edge to connect to. This also creates a failsafe in the event that one edge or server is not working correctly. The car will always receive a response from an edge and always be connected to the network. The car maintains the currently connected edge and its adjacent edges and sends the data to those servers and waits for a response from only 1 edge. .  
  
Our server has a method that returns the best edge to connect to based on the car's current edge and location. This algorithm can be modified based on some formula to return a more ideal edge. Currently, we return based on location. Actors such as signal can be included in the future. Since we implemented a robust design with many design patterns and adapters, a change to the algorithm will not impact the existing code, and everything will still function correctly.

A screen shot of a computer program

Description automatically generated

## What is Vehicle-to-vehicle communication:

Vehicle-to-vehicle communication refers to the sharing of data between multiple vehicles. These vehicles can use systems such as networks to share the data. Often times these communication systems allow vehicles to transmit and receive information about their speed, position, direction, and other relevant data to nearby vehicles [1]

**Vehicle to Vehicle communication has many benefits:**

1. **Safety:** Vehicle-to-vehicle communication can present many safety benefits as it is able to prevent accidents and collisions. Cars can communicate with each other to let other vehicles know they are coming and their surroundings, this data can be used and processed to prevent accidents and create awareness of the surroundings.
2. **Traffic Management:** Vehicle-to-vehicle communication can reduce traffic congestion and improve traffic flow by providing information on traffic and conditions surrounding the vehicles. This data can be collected and processed to create better vehicle routes, causing them to bypass traffic.
3. **Emergency Services:** Vehicle-to-vehicle communication can be used to communicate with emergency services and provide real-time and accurate data, which can be more helpful for emergency responders. This could lead to better and faster response times, which can be life-saving.

All autonomous vehicles that are currently deployed on the roads use vehicle-to-vehicle communication. Without this communication, people would not be safe on the roads.  
  
Our algorithm implements this technology by exposing an API that provides data about the vehicle. This API can be accessed and analyzed by anyone considering the car's id. In the future, algorithms can be developed that use the API between nearby cars and share the data with them. The cars can then process the data and make decisions accordingly.  
  
Future uses of this API with Quanser can be used to help make decisions and train the ML algorithm that the car will use to drive autonomously. This could lead to significant improvements to the vehicle. What we have implemented is not new but uses a reusable and robust architectural style. This can be useful to Quanser as they do not have this implemented yet.  
  
Below is an image of how our rest API server is set up to handle temperature and traffic data requests from the car. This, in the future, can be modified to include the unique ID of the car so that the same link can be used for different vehicles.

A screen shot of a computer program

Description automatically generated

## What is an outlier and how it works?

The car will constantly record and share certain data such as temperature, and traffic intensity with the public. This data will be accessed via different APIs. However, there is an odd chance of skewed data being sent to the server due to the malfunction of the sensors or genuinely rare and extreme events. To avoid this situation, we have implemented a trust system within our project. It uses an outlier algorithm to prevent exposing skewed data to the server.

This outlier algorithm keeps track of data using a rolling dataset. If there is a drastic difference at any point in the dataset compared to the rest of the data, then it considers this data as an error and doesn’t share it with the public. The outlier is still added to the rolling data set in case it is not an error and is used to determine future outliers. Identifying outliers is important in data to avoid errors in data collection and statistical analysis.

There are different methods and techniques for detecting outliers. The choice of outlier methods and techniques varies based on the nature of the data. Machine learning algorithms can be implemented to identify the outliers to determine whether the recorded data is accurate or not by doing some prediction-based analysis. On the other hand, outliers can be computed using statistical data analysis where data will be stored in the database and recorded data will be considered as accurate or error data based on the standard deviation.

For this project, we have used the rolling-based data set where data are saved in a queue. Every piece of recorded data gets compared with the previously recorded data. However, if a drastic change occurs (error data) in the recorded data due to any of the reasons, this data is considered as an outlier.

A diagram of a diagram of a number of dots

Description automatically generated  
Source: Adapted from [3]

Outlier detection using trust in data mining is the process of identifying and analyzing data points that significantly differ from the rest of the dataset and not passing them to the client. Detecting these outliers is critical as it creates trust between the source and receiver Ensuring data is accurate allows us to better process the dating leading to more valuable insights. [3]

## How can other people access the data of the car?

We have implemented a unique API for all the cars, this allows external users to access data provided by the car without being connected to the car. Additionally, each car has its own unique ID allowing for any car data to be collected provided they have been given the ID of the car. This data can be used by externally approved users to essentially collect data and use it for urban planning, safety precautions, traffic management, regulatory oversight, and customer service. To keep the data error-free, we have used the outlier protocol. With the help of his outlier, any data recorded due to malfunction of the sensor, or any other reason will not be shared with the public.

In our implementation, we have used the Interquartile Range algorithm to determine outliers. We have defined a max reading variable which sets the maximum number of readings to consider for calculating the IRQ. In a real-world example, this number would be much larger. We constantly add the data received to a queue and remove the oldest data if we have reached the maximum readings. This enables a rolling dataset. We then sort the data and calculate the median values of the first and last halves of the sorted list, Q1 and Q3, respectively. We then calculate the IRQ as well as the lower and upper bounds. The method returns true if the given temperature is less than the lower bound or greater than the upper bound, indicating it is an outlier.  
  
The IQR algorithm is not susceptible to extreme outliers as it only considers the middle 50% of the data. This makes it much more reliable.

A screen shot of a computer program

Description automatically generated

# Major Design Decisions

**Design Patterns and Architectural Choices**

Our project's core design incorporates several classical design patterns, each chosen for their specific benefits and suitability to the project's needs:

1. **Singleton Pattern:** Ensures a single instance of critical management components, promoting consistency and preventing issues related to multiple instantiations.
2. **Strategy Pattern:** This pattern allows for the dynamic alteration of the car's operational algorithms. It's instrumental in adapting the car's behavior based on real-time data and conditions. It is used to determine which type of data to pass back to the users based on their API access.
3. **Adapter Pattern:** Facilitates seamless integration, allowing other cars to utilize the same API for external data sharing while ensuring our design remains flexible and adaptable to various external interfaces.
4. **Factory Pattern:** Used for creating objects in our system, this pattern provides a way to encapsulate the instantiation logic, making the system more modular and easier to manage when more edges are added to the network.

**Edge Computing Implementation:**

A fundamental aspect of our project is implementing a basic edge computing design. This allows the Quanser Qcar to interact intelligently with different servers based on geographical location. The benefits of this approach include:

1. **Reduced Latency:** We significantly decrease response times by processing data closer to the source.
2. **Geographical Optimization:** The car communicates with the nearest server, ensuring faster data transfer and efficient use of network resources.

**Springboot Server and API Exposure:**

We have developed a Springboot server to manage the data interactions of the Qcar. The server's main role is to expose the car's data through a custom-built API. This API is the gateway for external parties to access the data generated by the car. Key features of this API include:

1. **Data Accessibility:** External members can access the car's data anywhere, ensuring global connectivity and data sharing.
2. **Command and Control Capabilities:** Future upgrades might let outside users with the vehicle or ask for data sets to improve the car's interactive capabilities.

**Trust Algorithm and Outlier Detection:**

An innovative aspect of our design is the implementation of a trust algorithm, leveraging outlier detection methodologies. This algorithm plays a crucial role in the security and integrity of the data exposed through our API. Its main functions are:

1. **Identifying Anomalies:** The algorithm scrutinizes data for inconsistencies or irregularities, flagging potential security threats.
2. **Data Integrity Assurance:** By filtering out outlier data, the algorithm helps maintain the authenticity and reliability of shared data.

# UML Diagram

Car, API, & Trust: A diagram of a network

Description automatically generated with medium confidence

Server & Edge Computing: A diagram of a diagram

Description automatically generated

# Sequence Diagrams

Car retrieves the nearest edge and its adjacent edges: *A diagram of a computer program

Description automatically generated*

Car implements trust and outlier detection to expose its data via API: A screenshot of a computer

Description automatically generated

# Architecture

Overview of Component Diagram:

A diagram of a website

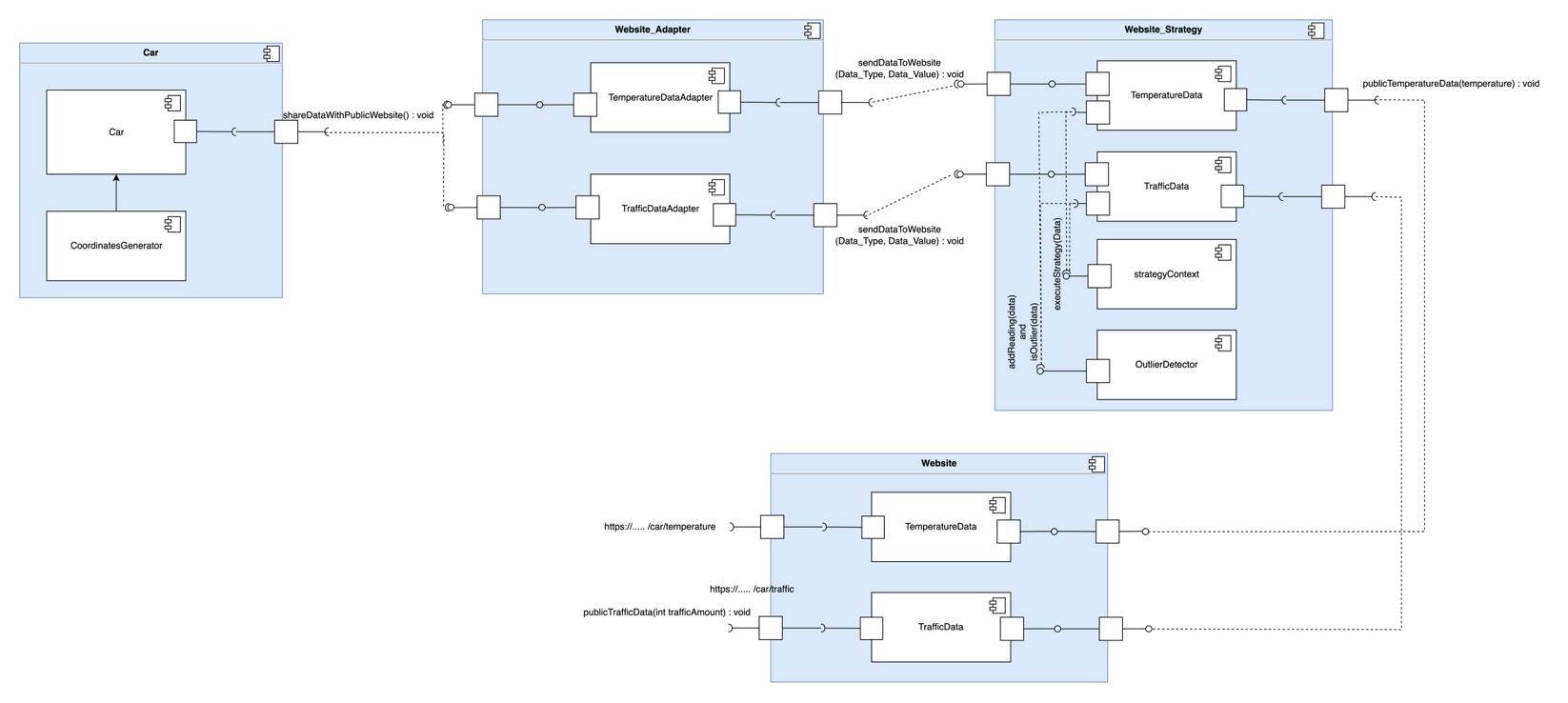
Description automatically generated

Car sends data to edge and its adjacent edges Component Diagram:

*A diagram of a computer

Description automatically generated*

Car exposes its data via API and uses trust Component Diagram:



Combined Component Diagram:

A diagram of a computer network

Description automatically generated

**Client & Trust:**

| **Module Name** | **Description** | **Exposed Interface Names** | **Interface Description** |
| --- | --- | --- | --- |
| Car | Represents a mock-up of a real Qcar, handling data processing and transfer. | Car:getLatitude, Car:getLongitude, Car:sendClientData, Car:shareDataWithPublicWebsite | getLatitude: Gets the latitude of the car. getLongitude: Gets the longitude of the car. sendClientData: Sends data to a specified server. shareDataWithPublicWebsite: Shares car data with a public website. |
| Coordinates  Generator | Generates mock coordinates for the car. | CoordinatesGenerator:generateRandomLatitude, CoordinatesGenerator:generateRandomLongitude | generateRandomLatitude: Generates a random latitude value. generateRandomLongitude: Generates a random longitude value. |
| Outlier  Detector | Detects outliers in temperature readings. | OutlierDetector:isOutlier | isOutlier: Determines if a temperature reading is an outlier. |
| Part3  Application | Main class for running the car application. | - | - |
| SelectStrategy | Selects and executes strategies for sending different types of data. | SelectStrategy:sendDataToWebsite | sendDataToWebsite: Sends data to the website based on the type of data. |
| SendStrategy | Interface for sending strategies. | SendStrategy:send | send: Abstract method for sending data. |
| Send  Temperature  Data | Adapter for sending temperature data to the website. | TemperatureDataAdapter:tempData | tempData: Sends temperature data to the website. |
| SendTraffic  Data | Adapter for sending traffic data to the website. | TrafficDataAdapter:trafficData | trafficData: Sends traffic data to the website. |
| Strategy  Context | Context for executing different sending strategies. | StrategyContext:executeStrategy | executeStrategy: Executes a specific sending strategy. |
| Temperature  Data | Implements sending strategy for temperature data. | SendStrategy:send | send: Sends temperature data after outlier detection. |
| Temperature  DataAdapter | Interface for temperature data adapter. | - | - |
| TrafficData | Implements sending strategy for traffic data. | SendStrategy:send | send: Sends traffic data after outlier detection. |
| TrafficData  Adapter | Interface for traffic data adapter. | - | - |
| Website | Exposes REST API endpoints for temperature and traffic data. | Website:temperatureAPI, Website:trafficAPI | temperatureAPI: Returns temperature data.<br>trafficAPI: Returns traffic data. |

| **Interface Name** | **Operations** | **Operation Descriptions** |
| --- | --- | --- |
| Car:getLatitude | double getLatitude() | getLatitude: Returns the latitude of the car. |
| Car:getLongitude | double getLongitude() | getLongitude: Returns the longitude of the car. |
| Car:sendClientData | void sendClientData(String data) | sendClientData: Sends data to a specified server. Uses a list of servers and sends the data to each of them. |
| Car:shareDataWithPublicWebsite | void shareDataWithPublicWebsite() | shareDataWithPublicWebsite: Shares car data with a public website. Uses adapters to format the data. |
| CoordinatesGenerator:generateRandomLatitude | double generateRandomLatitude() | generateRandomLatitude: Generates a random latitude value based on the car's current location. |
| CoordinatesGenerator:  generateRandom  Longitude | double generateRandomLongitude() | generateRandomLongitude: Generates a random longitude value based on the car's current location. |
| OutlierDetector:isOutlier | boolean isOutlier(Double temperature) | isOutlier: Determines if a temperature reading is an outlier based on IQR method. |
| SelectStrategy:sendData  ToWebsite | void sendDataToWebsite(String typeOfData, Object... data) | sendDataToWebsite: Sends data to the website based on the type of data. Uses StrategyContext for execution. |
| SendStrategy:send | void send(Object... data) | send: Abstract method for sending strategies. Implemented by TemperatureData and TrafficData. |
| TemperatureDataAdapter:tempData | void tempData(Double temperature) | tempData: Sends temperature data to the website. Uses SelectStrategy to determine the sending mechanism. |
| TrafficDataAdapter:trafficData | void trafficData(int trafficAmount) | trafficData: Sends traffic data to the website. Uses SelectStrategy to determine the sending mechanism. |
| Website:temperatureAPI | String temperatureAPI() | temperatureAPI: REST API endpoint for accessing temperature data. |
| Website:trafficAPI | String trafficAPI() | trafficAPI: REST API endpoint for accessing traffic data. |

**Server & Edge:**

| **Module Name** | **Description** | **Exposed Interface Names** | **Interface Description** |
| --- | --- | --- | --- |
| DistanceCalculator | Calculates the distance between two geographical coordinates. | DistanceCalculator:getDistance | getDistance: Calculates distance between edge and car coordinates. |
| Edge | Represents a network node with geographical coordinates and manages connected cars. | Edge:getEdgeId, Edge:addAdjacentEdge, Edge:findNearestEdge, Edge:connectCarToEdge, Edge:removeCarFromEdge | getEdgeId: Returns the ID of the edge.  addAdjacentEdge: Adds an adjacent edge to this edge.  findNearestEdge: Finds nearest edge based on car coordinates  connectCarToEdge: Adds a car to this edge.  removeCarFromEdge: Removes a car from this edge. |
| EdgeDAO | Accesses Edge instances from EdgeDatabase. | EdgeDAO:getEdge | getEdge: Retrieves a specific edge instance based on ID. |
| EdgeDatabase | Maintains and provides access to Edge instances. | EdgeDatabase:getInstance, EdgeDatabase:edgeOne, ... edgeFive | getInstance: Returns singleton instance of EdgeDatabase edgeOne, ... edgeFive: Retrieves specific Edge instances. |
| EdgeServer  Application | Entry point for the Spring Boot application. | - | - |
| ServerController | Processes HTTP requests and manages edge server operations. | ServerController:receiveDataFromClient1, ..., receiveDataFromClient5, ServerController:sendServerResponse | receiveDataFromClient1, ..., receiveDataFromClient5: Handles POST requests from clients.  sendServerResponse: Handles GET request and sends server response. |

| **Interface Name** | **Operations** | **Operation Descriptions** |
| --- | --- | --- |
| DistanceCalculator:  getDistance | double getDistance(double edgeLatitude, double edgeLongitude, double carLatitude, double carLongitude) | getDistance: Calculates the distance between edge and car coordinates. |
| Edge:getEdgeId | int getEdgeId() | getEdgeId: Returns the unique ID of the edge. |
| Edge:addAdjacentEdge | void addAdjacentEdge(Edge edge) | addAdjacentEdge: Adds an adjacent edge to the edge's adjacency list. |
| Edge:findNearestEdge | Edge findNearestEdge(String carID, double latitude, double longitude) | findNearestEdge: Determines the nearest edge to the provided car coordinates and updates car connections. |
| Edge:connectCarToEdge | void connectCarToEdge(String carID) | connectCarToEdge: Connects a car to the edge by adding the car's ID to the edge's list. |
| Edge:removeCarFromEdge | void removeCarFromEdge(String carID) | removeCarFromEdge: Removes a car from the edge by removing the car's ID from the edge's list. |
| EdgeDAO:getEdge | Edge getEdge(int edgeId) | getEdge: Retrieves a specific Edge instance from the EdgeDatabase based on the provided edge ID. |
| ServerController:receiveDataFromClient1, ..., receiveDataFromClient5 | String receiveDataFromClient1(String data), ..., receiveDataFromClient5(String data) | receiveDataFromClient1, ..., receiveDataFromClient5: Processes POST requests from clients and updates edge connections based on data. |
| ServerController:sendServerResponse | String sendServerResponse() | sendServerResponse: Responds to GET requests with a list of server IDs adjacent to the closest edge. |

# Activities Plan

## Group Meeting Logs

| **Version** | **Date** | **Authors** | **Summary of Changes** |
| --- | --- | --- | --- |
| 1.0 | October 4, 2023 | All members | Insert Gantt, sequence, UML, use case, and component diagrams |
| 1.1 | October 8, 2023 | All members | Revised UML diagrams and updated sequence diagrams |
| 1.2 | October 12, 2023 | All members | Added detailed descriptions to use case diagrams |
| 2.0 | October 18, 2023 | All members | Major revision of all diagrams and introduction of new component diagrams |
| 2.1 | October 24, 2023 | All members | Implemented adapter design pattern for exposing car data via api |
| 2.2 | October 28, 2023 | All members | Implemented rest api for car data |
| 2.3 | November 5th, 2023 | All members | Implemented state design pattern and factory for edge computing |
| 2.4 | November 8th, 2023 | All members | Implemented edge computing servers and other design patterns to complete the code |
| 2.5 | November 10th, 2023 | All members | Create sequence, component, and uml diagrams |
| 2.6 | November 12th, 2023 | All members | Created component diagrams and filled out module/interfaces table |
| 2.7 | November 13th, 2023 | All members | Added research about trust, outliers, edge computing, external access via api |
| 2.8 | November 15th, 2023 | All members | Completed design and architectural section of report and formatted document |

# Bibliography

[1] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, “Edge Computing for Autonomous Driving: Opportunities and Challenges | IEEE Journals & Magazine | IEEE Xplore,” *ieeexplore.ieee.org*, Jun. 24, 2019. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8744265 (accessed Dec. 07, 2023).

[2] H. Ji, O. Alfarraj, and A. Tolba, “Artificial Intelligence-Empowered Edge of Vehicles: Architecture, Enabling Technologies, and Applications | IEEE Journals & Magazine | IEEE Xplore,” *ieeexplore.ieee.org*, Mar. 26, 2020. https://ieeexplore.ieee.org/document/9047892 (accessed Dec. 07, 2023).

[3] R. Jodha, “Outlier Analysis in Data Mining,” *Scaler Topics*, May 26, 2023. https://www.scaler.com/topics/data-mining-tutorial/outlier-analysis-in-data-mining/

[4]X. Xu, Y. Xue, X. Li, L. Qi, and S. Wan, “A Computation Offloading Method for Edge Computing With Vehicle-to-Everything | IEEE Journals & Magazine | IEEE Xplore,” *ieeexplore.ieee.org*, Aug. 10, 2019. https://ieeexplore.ieee.org/abstract/document/8830373 (accessed Dec. 07, 2023).