# ADAS Test Scenarios in Carla Simulator

Smart Vehicular System

Davide Di Marco - 0001136612 {davide.dimarco5e@studio.unibo.it}

January 20, 2025

# Contents


**1 Introduction** **3**
- 1.1 Detection of a Vehicle in the Ego Vehicle's Blind Spot . . . . . 3
  - 1.1.1 Non-Functional Requirements . . . . . 4
- 1.2 Detection of a Pedestrian Suddenly Crossing in Front of the Ego Vehicle . . . . . 4
  - 1.2.1 Functional Requirements . . . . . 4
  - 1.2.2 Non-Functional Requirements . . . . . 4
- 1.3 Detection of a Bicycle Overtaking the Ego Vehicle from the Right at an Intersection . . . . . 4
  - 1.3.1 Functional Requirements . . . . . 5
  - 1.3.2 Non-Functional Requirements . . . . . 5

**2 Design** **5**
- 2.1 System Architecture . . . . . 5
  - 2.1.1 Rear Blind Spot and Lateral Right Sensors . . . . . 5
  - 2.1.2 Pedestrian Detection System . . . . . 7
  - 2.1.3 Test Suite menù - User interface . . . . . 9
  - 2.1.4 Debugging visualization . . . . . 9
- 2.2 Software Architecture . . . . . 11
  - 2.2.1 Structure . . . . . 12
  - 2.2.2 Blind Spot Detection . . . . . 14
  - 2.2.3 Pedestrian Detection . . . . . 16
  - 2.2.4 Bycicle Detection - Vehicle Turn Assist . . . . . 18

**3 Technologies** **19**
- 3.1 CARLA Simulator and Python API . . . . . 19
- 3.2 YOLOv8 . . . . . 19
- 3.3 HiveMQ - MQTT Service . . . . . 20
- 3.4 Conda . . . . . 21

**4 Code** **21**
- 4.1 Blind Spot Detection . . . . . 21
- 4.2 Pedestrian Detection . . . . . 23
- 4.3 Bicycle Overtaking . . . . . 25

**5 Testing** **26**
- 5.1 Test Scenarios . . . . . 26

**6 Deployment** **29**
- 6.1 Prerequisites . . . . . 29
- 6.2 Execution . . . . . 29

**7 Conclusions** **30**

# 1 Introduction

In recent years, the car industry has changed a lot because of new technologies like Advanced Driver Assistance Systems (ADAS) and autonomous driving. These systems help vehicles behave better on the road, making driving safer, easier, and more efficient. Since these systems are important for the safety of drivers, passengers, and other road users like pedestrians and cyclists, they must be tested carefully. Testing on real roads can be expensive, risky, and hard to manage. That's why simulation platforms are very useful. They allow us to create many different and repeatable situations in a safe and controlled way. CARLA is an open-source simulator made for research in autonomous driving. It provides a flexible and realistic environment, with urban and suburban roads, traffic situations, and support for different sensors like cameras, Lidars, and Radars. This makes it possible to simulate complex and dangerous situations that would be difficult or unsafe to test in real life. The goal of this project is to use CARLA to test ADAS features by simulating three important scenarios: blind spot detection, pedestrian detection, and bicycle overtaking at intersections. The project checks if the system can detect objects correctly, react in time, and make the right decision. The scenarios are designed to be modular and easy to reuse, so that more tests can be added later.
This report describes the project requirements, the design, the implementation, and the testing of the ADAS scenarios created in CARLA. It also explains the methods we used and the challenges we faced.

The project aims to develop and test three specific ADAS scenarios:

- Detection of a vehicle in the ego vehicle's blind spot
- Detection of a pedestrian suddenly crossing in front of the ego vehicle
- Detection of a bicycle overtaking the ego vehicle from the right at an intersection

The requirements for each are detailed in the following subsections.

## 1.1 Detection of a Vehicle in the Ego Vehicle's Blind Spot

Blind Spot Detection must reliably identify objects in the ego vehicle's blind spot, including vehicles, pedestrians, and bicycles.

**Functional Requirements**

- Trigger an auditory alert when an object is detected in the blind spot, warning the driver of potential hazards.
- Connect each sensor to an online MQTT broker for data transmission.

### 1.1.1 Non-Functional Requirements

- The detection system must accurately identify objects within the predefined blind spot area, ensuring consistency with the simulation environment.
- Real-time detection is required to provide immediate feedback to the driver.
- Sensors must be placed on the rear-right, rear-left and lateral right side of the vehicle to monitor the respective blind spot areas.

## 1.2 Detection of a Pedestrian Suddenly Crossing in Front of the Ego Vehicle

The pedestrian detection system must accurately identify pedestrians appearing in front or laterally in front of the ego vehicle, particularly in situations where a pedestrian suddenly crosses the road.

### 1.2.1 Functional Requirements

- The ego vehicle must detect pedestrians and, in case of potential collision, take corrective action to avoid a collision.
- The ego vehicle must either slow down or apply full braking, depending on the confidence level of the pedestrian detection system.

### 1.2.2 Non-Functional Requirements

- The pedestrian detection system must operate with minimal latency to ensure rapid response.
- The system must use a forward RGB camera for pedestrian detection.
- An MQTT message must be sent to the broker to communicate detected hazards.

## 1.3 Detection of a Bicycle Overtaking the Ego Vehicle from the Right at an Intersection

This scenario involves a bicycle overtaking the ego vehicle from the right at an intersection, particularly when the ego vehicle is executing a right turn, posing a collision risk.

### 1.3.1 Functional Requirements

- The blind spot detection system must identify objects within the blind spot area.
- The ADAS must emit an auditory alert in case of a potential collision with a bicycle.
- When the ego vehicle is turning right, the ADAS must take corrective action to prevent a collision with the bicycle. The main goal is save the cyclist.

### 1.3.2 Non-Functional Requirements

- A radar sensor must be placed on the right side of the vehicle to monitor the lateral blind spot area very near the vehicle.
- The radar's detection range must be optimized for close-range coverage, specifically for bicycle overtaking scenarios.
- An MQTT message must be sent to the broker to inform of detected events.

# 2 Design

## 2.1 System Architecture

In this section, the sensors used for detection are described along with the rationale behind their placement and orientation on the vehicle, also a graphical representation is explained to correct understand where the sensors are located and how the sensors are expected to work.

### 2.1.1 Rear Blind Spot and Lateral Right Sensors

Two radar sensors are installed on the rear sides of the vehicle, one on the right and one on the left. Each radar is positioned **2.0 meters behind** the vehicle's origin along the $x$-axis, **0.9 meters laterally** ($y$-axis), and at a height of **0.7 meters** ($z$-axis). The right radar is oriented at **160°**, and the left radar at **200°**, effectively covering the blind spot zones. Both have a **55° horizontal FOV** and a **7-meter range**, which are suitable for detecting approaching vehicles in adjacent lanes.

For lateral bicycle detection, a radar sensor is mounted on the **right side** of the vehicle, positioned **0.9 meters** laterally ($y$-axis) and **0.7 meters** in height ($z$-axis), with an orientation of **90°**. This sensor has a **150° horizontal FOV** and a shorter **range of 3 meters**, optimized for detecting bicycles approaching from the side during right turns at intersections or overtaking from right.

After defining the sensor, we define the **blind spot area** where the object could be immediately not recognized by eyes. In this project we also need a

| Radar | Position | Yaw | Hor.Fov | Vert. Fov | Range |
|---|---|---|---|---|---|
| Blind Spot | Rear-Right | 160° | 55 | 10 | 7 |
| Blind Spot | Rear-Left | 200° | 55 | 10 | 7 |
| Bicycle Radar | Right-Side | 90° | 150 | 10 | 3 |

Table 1: Rear Blind Spot and Lateral Blind Spot

**lateral blind spot** as shown in the image 1 (area in red color). This blind spot is related to the driver that look at front and doesn't see the lateral side of the vehicle. It's useful to identify bike as shown in one of the scenario study in deep in following section.

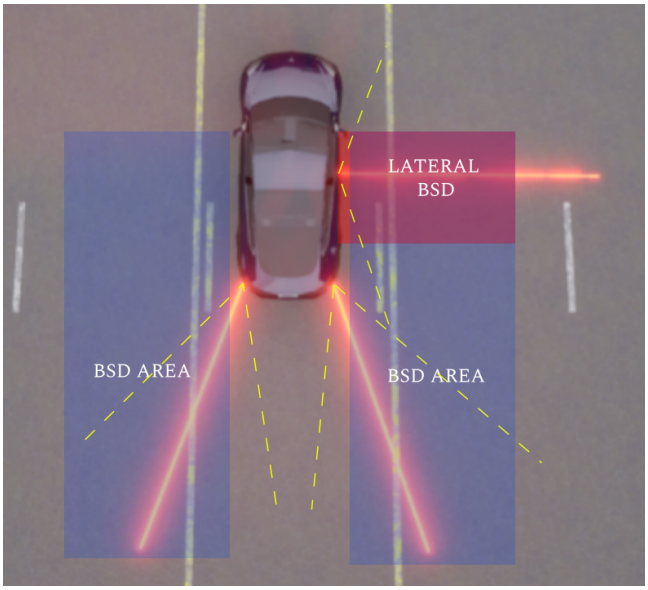


Figure 1: Defined Blind spot Area

After defining the positioning of the sensor on the vehicle we understand the behaviour of the ADAS. Blind Spot Detection must be coherent with the position of the object in the area of detection, so we define the areas as shown in Figure 1. In particular we define the coordinate accordingly with the radar sensor position. The values of x and y shown in Table 2 refer to the *Local frame*

| Radar | Position | x-axis (m) | y-axis (m) |
|---|---|---|---|
| Blind Spot | Rear-Right | $0 < x < 10$ | $0.5 < y < 3.0$ |
| Blind Spot | Rear-Left | $0 < x < 10$ | $-3.0 < y < -0.5$ |
| Lateral Blind Spot | Right-Side | $-1.0 < x < 3.5$ | $0 < y < 2.0$ |

Table 2: Rear Blind Spot and Lateral Blind Spot Area

of each radar sensor, not to the world coordinate system. The x-axis points in the direction the radar is facing (forward for the lateral radar, rearward for rear radars). The y-axis represents lateral movement relative to the radar: positive values point to the right and negative to the left when facing the radar's direction. These coordinates are used to define a detection zone around the vehicle for each sensor, ensuring the ADAS only reacts to objects within the intended blind spot area. This approach allows the system to react based on the object's location relative to the vehicle, ensuring accurate alerts.

### 2.1.2 Pedestrian Detection System

Pedestrian detection is performed using an **RGB camera** mounted at the **front-center of the vehicle** at **0.5 m** along the $x$-axis and **1.5 m** in height ($z$-axis). The camera faces forward with a **90° field of view** and captures images at a resolution of **800×600 pixels**. The followd system schema for this approach is referred in 2 The images are processed using a YOLOv8s model, as described in technologies section. Depending on the confidence of detection, actions like ***Slowing down*** or ***Braking*** are triggered. Effective detection is achieved within approximately **100 meters**, suitable for urban driving environments.

| Camera | Position | Yaw | FOV | Range | Resolution |
|---|---|---|---|---|---|
| RGB Camera | Rear-Center | 0° | 90° | 100m | 800×600 |

Table 3: Camera parameters for pedestrian detection

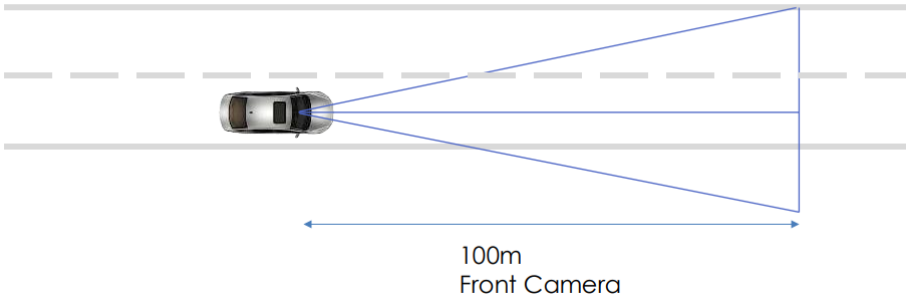


Figure 2: Schema of the Camera Positioning

As shown in Figure 3 in order to understand what the camera sees, an **OpenCV** is implemented to display the scene in real time. This live feed al-

lows for visual feedback during the simulation, helping to verify the correct placement of the camera, the effectiveness of the detection system, and ensure debug potential issues with the YOLOv8s inference. The real-time visualization also useful in performance evaluation by allowing direct observation of pedestrian detection accuracy and the corresponding vehicle response, such as slowing down or braking.

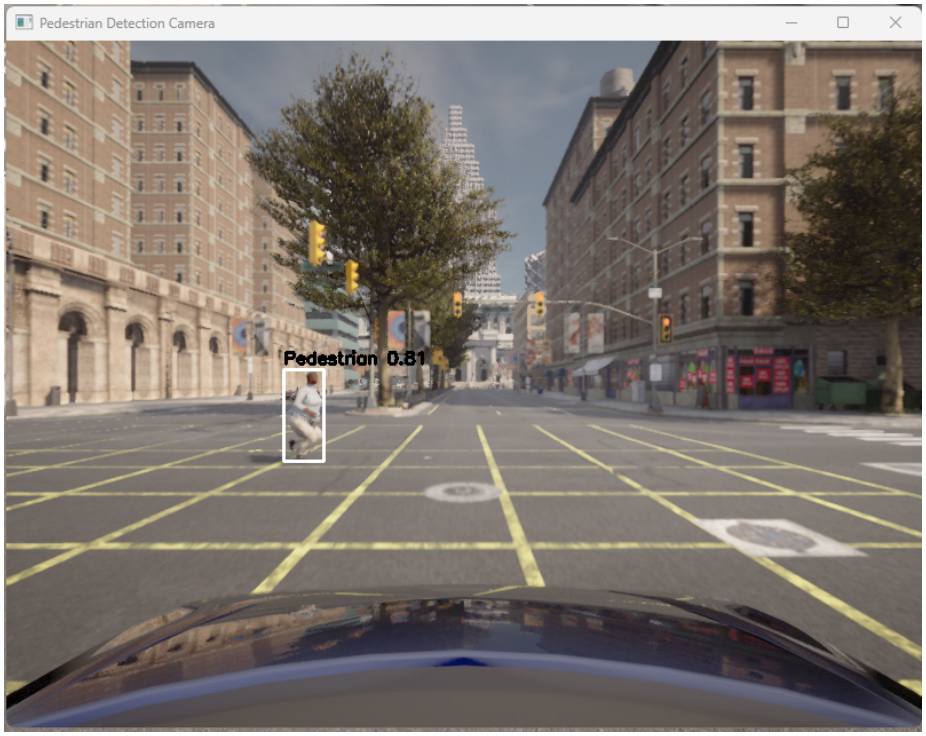


Figure 3: OpenCV Windows that shows Camera's Frame

### 2.1.3 Test Suite menù - User interface

In order to interact with the simulator and the entire system and select the test the user wishes to execute, a CLI (Command Line Interface) has been implemented. As shown in Figure 4, the interface is structured as a hierarchical menu that first allows the user to choose a macro test category (Blind Spot Detection, Pedestrian Detection, Bicycle Overtaking). The user can then select the specific type of test within that category (for instance, a static test or a dynamic test), and finally the exact test scenario (Overtake Scenario, Traffic Scenario, Sudden Braking in Blind Spot). This structure has been designed to ensure modularity and scalability: the system can be easily extended with new test categories or scenarios, and users can add their own menu entries and tests with minimal effort, without modifying the core logic of the interface.

```
===== Select a Macro Test Category =====
1. Blind Spot Detection (BSD)
2. Pedestrian Detection
3. Bicycle Overtaking
0. Exit
Enter the number of the test category: 1

===== Select a BSD Test Type =====
1. Static BSD Test
2. Dynamic BSD Test
0. Back to Main Menu
Enter the number of the BSD test type: 1
Subscribed: 1, QoS: [ReasonCode(Suback, 'Granted QoS 1')]
Radar sinistro e destro e laterale spawnati correttamente!

===== Select a Specific BSD Test =====
1. Overtake Scenario
2. Traffic Scenario
3. Sudden Braking in Blind Spot
0. Back to BSD Menu
Enter the number of the specific test:
```

Figure 4: CLI for the test selection

In figure 5 CLI is also used to allow for logging during simulation. Printing the simulation output is costly in term of performance. During final tests most of the log are silenced.

### 2.1.4 Debugging visualization

To facilitate test debugging and provide better insight into how each scenario is functioning, as shown in Figure 6 **??** several visual elements have been introduced into the simulator environment. The radar sensor's orientation is represented by a directional ray and any detected objects or points are highlighted in

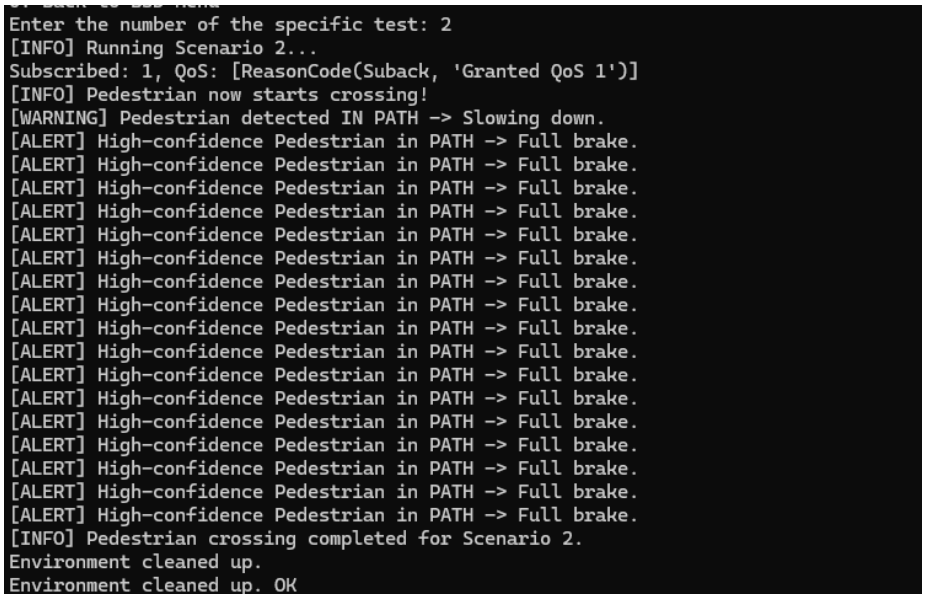


Figure 5: Logs in case of pedestrian detection with action

yellow. This allows users to confirm that the radar coverage is oriented correctly and that detections are triggered as expected.

In addition, for certain tests, as in case of dynamic tests, the vehicle's path is make evident through a series of waypoints, giving a visual indication of the intended route and helping to verify if the vehicle follows it correctly. These debug elements are useful for users to understand and validate the behavior of both the sensor system and the test scenario.

The visual debug elements introduced are:

- **Radar sensor orientation ray:** shows the current heading of the radar, helping verify coverage and detection angles.
- **Detection points in yellow:** highlights where the radar has detected objects, ensuring that detection is working as intended.
- **Path waypoints (in dynamic tests):** depicts the vehicle's planned trajectory, enabling the user to confirm that the vehicle navigates correctly.
- **Rectangle to identify pedestrian:** a first person camera visualization is implemented, a rectangle cover the pedestrian and confidence in shown above the rectangle.

Figure 6: Detection points in yellow, radar sensor orientation ray and path waypoints

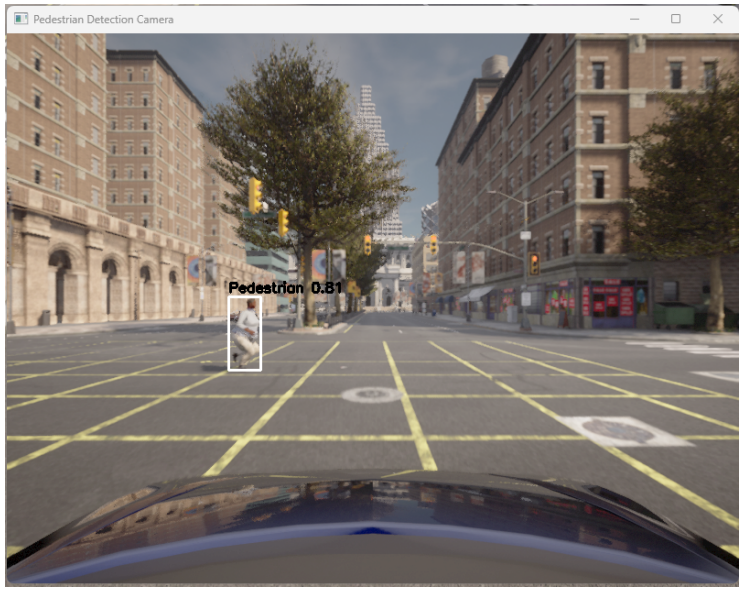


Figure 7: First camera view and pedestrian detection rectangle

## 2.2 Software Architecture

In this section is explained the software architecture implemented to interact with the simulator. These choices are very important because permit reusability,

scalability and additions of new test scenarios in few lines of code. For each scenario, the high level logic of the software is explained. For the main technical part of the explaination refer to Section 4.

### 2.2.1 Structure

The project is organized in modules and classes explained in following paragraph.

**Package Diagram** The project is organized into packages, each containing components such as Python classes or audio file. As shown in Figure 8, the project structure follows principles of maintainability, separation of concerns and modularity. Each package is designed with a specific responsibility. The structure of the packages is divided as follows:

- **ADAS**: the adas module is the main folder where all the implementation of the test is.
- **Networking**: this package contains the MQTT Client and the related features.
- **Managers**: this package contains all the implementation of the managers. The sense of the entity *Manager* is explained in the Class Diagram paragraph.
- **Pedestrian Detection**: this package contains all the implementation of the pedestrian detection systems.
- **BlindSpotDetection**: this package contains all the implementation for testing blind spot detection.
- **BicycleOvertaking**: this package contains all the implementation to test the bicycle overtaking test.

This design promotes a clean separation of concerns, allowing the system to be extended and maintained.

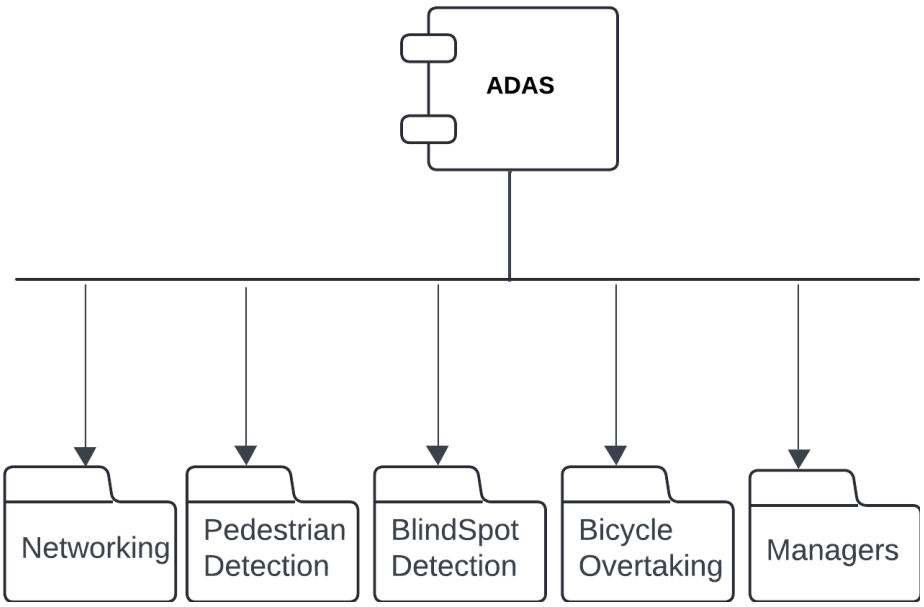


Figure 8: Package Diagram

**Class Diagram** The figure 9 shows the structure of the modules components through a UML class diagram, in which only the most significant fields, methods and relations are shown for each entity.

Central to the system is the *EnvironmentManager*, which oversees the spawning and management of vehicles, pedestrians, bicycles, and radar sensors in the virtual world. Sensor data is acquired and processed by the *CameraManager*, which utilizes a YOLO model for real-time image analysis, and the *RadarManager*, which handles radar data from lateral and rear sensors. Both managers uses *MQTTClient*, responsible for sending and receiving alerts through the MQTT protocol, enabling integration with external systems for monitoring or actuation.

The *RouteManager* controls vehicle navigation by setting destinations, drawing paths, and commanding the vehicle to follow specific routes using a *BasicAgent*. Test classes including *TestBlindSpot*, *TestBicycleOvertaking*, and *TestPedestrianDetection*, implement driving scenarios, such as dynamic blind spot monitoring, safe overtaking, and pedestrian detection. These tests use sensor feedback and environmental control to simulate realistic interactions.

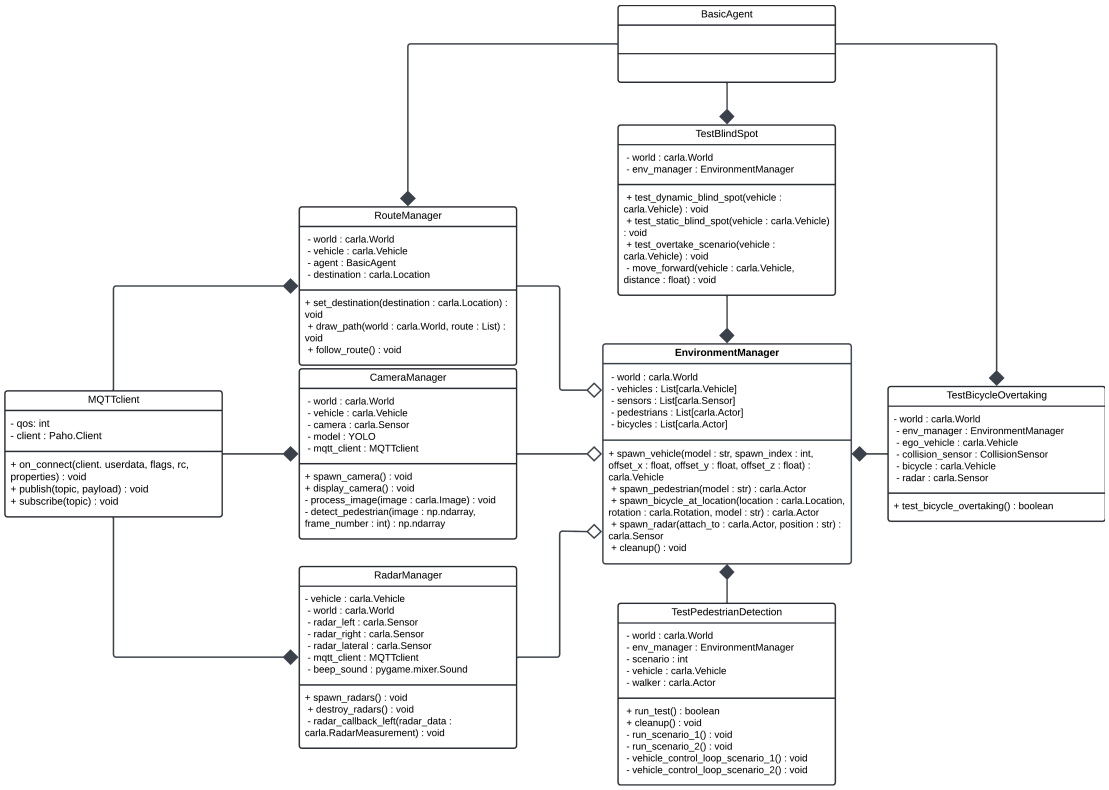


Figure 9: Class Diagram

### 2.2.2 Blind Spot Detection

In this project, Blind Spot Detection (BSD) is implemented to monitor the areas adjacent to and behind the vehicle, commonly referred to as blind spots, which are not visible through standard mirrors.

As shown in the System Architecture section, two mid-range radar sensors are mounted on the rear-left and rear-right of the vehicle, defining the blind spot zones. The system detects objects entering these zones and determines their speed relative to the ego vehicle.

The figure 10 illustrates the functional architecture of the BSD system. It follows a standard ADAS data flow structure:

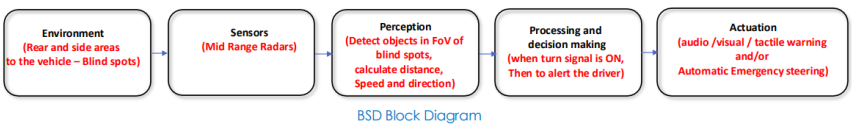


Figure 10: Blind Spot Detection Block Diagram

The process begins with the **Environment**, which refers to the rear and lateral areas around the vehicle. *Mid-range radars sensors*, collect real-time data about objects entering the blind spots. The **Perception** part, which is the radar callback, processes this data to detect objects within the radar's field of view, calculating their distance and detection. The **Processing and Decision Making** part evaluates if the detected objects pose a risk and decides whether to alert the driver. Finally, the **Actuation** system provides appropriate audio warnings. This modular approach allows for real-time detection and responsive driver assistance, enhancing vehicle safety during lane changes or turns. The BSD system was integrated into a simulation environment to test and validate its performance under various traffic conditions, in both **static** and **dynamic** contexts:

- **Static Context:** the ego vehicle is stopped while another vehicle enters or remains in the blind spot (see Figure 11).
- **Dynamic Context:** the ego vehicle is moving, and a follower vehicle continuously remains in its blind spot as shown in Figure 6

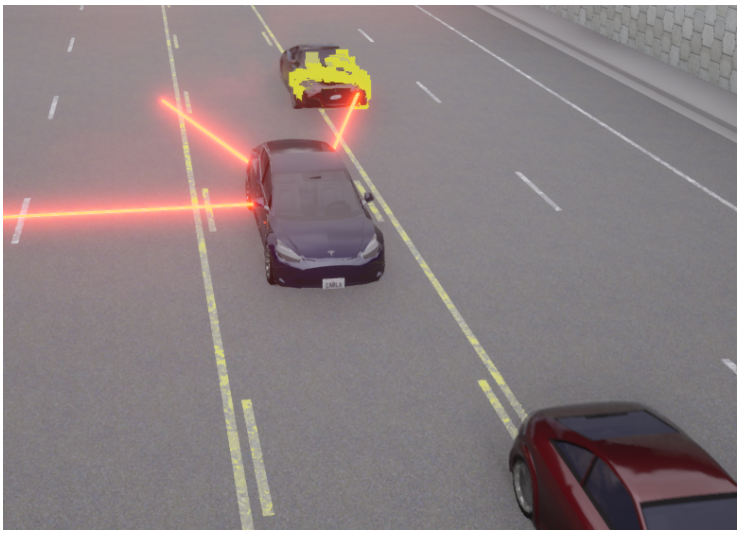

Figure 11: Example of static context in blind spot detection simulation

To ensure robust performance, multiple static test scenarios were developed. These include the ***overtake scenario***, where a vehicle overtakes the ego vehicle from the left; the traffic scenario, with multiple vehicles entering the blind spot sequentially; and the sudden braking scenario, where a follower vehicle sudden stops in the blind spot. In the ***dynamic ego scenario***, the ego vehicle follows a planned route while being followed in its blind spot by another vehicle.

### 2.2.3 Pedestrian Detection

In this project, the Pedestrian Detection system is implemented to identify pedestrians in the vehicle's path and react accordingly to avoid collisions, especially in urban environments where pedestrian interaction is frequent.

As illustrated in Figure 12, the system follows a classical ADAS (Advanced Driver Assistance Systems) pipeline. In Figure 12 is shown the classical Forward Collision Warning because a pedestrian collision avoidance can be interpreted as a specialization of this ADAS.

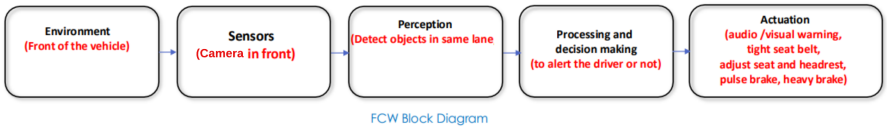


Figure 12: Pedestrian Detection Block Diagram

The process begins with the **Environment**, which refers to the area in front of the ego vehicle where pedestrians might appear. The system uses **Sensors**, specifically a front-facing camera, to continuously capture visual data of the road ahead. The **Perception** part processes the video stream to detect and classify objects, specifically identifying pedestrians in the same lane as the vehicle. This detection leverages confidence levels, distinguishing between standard detections and high-confidence detections, which indicate a higher probability that a pedestrian is in the direct path of the vehicle.

Next, the **Processing and Decision Making** alghoritm evaluates the pedestrian's position relative to the vehicle. Based on this analysis, it decides whether a warning or braking action is necessary. For instance, if a pedestrian is detected directly in front of the car and within a predefined safety distance, the system decides to trigger a full brake; otherwise, it might simply slow down the vehicle.

Finally, the **Actuation** phase translates the decision into action: applying controlled braking (gradual deceleration or full braking). The system dynamically adapts the response depending on the pedestrian's location and safety distance to stop the vehicle. In **Scenario 1** in Figure 13, multiple pedestrians randomly cross the vehicle's path, creating a tunnel-like environment. it simulate the case of crowded pedestrian area.


Figure 13: Scenario 1 of pedestrian detection

In **Scenario 2** in Figure 14, a single pedestrian crosses the street while the ego vehicle approaches. In both scenarios, real-time pedestrian detection is used to trigger appropriate vehicle responses, ensuring safety of the systems.

Figure 14: Scenario 2 of pedestrian detection

### 2.2.4 Bycicle Detection - Vehicle Turn Assist

This ADAS feature is a combination of Blind Spot Detection (BSD) and Vehicle Turn Assist (VTA).

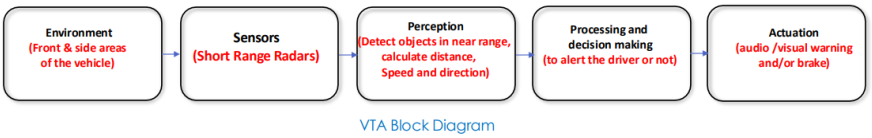


Figure 15: Vehicle Turn Assist Block Diagram

As shown in Figure 15, the system monitors the **Environment**, focusing on the front and lateral areas of the vehicle where cyclists may approach or ride alongside. It uses **Short Range Radars**, mounted on the side of the ego vehicle, to detect objects in close proximity. The **Perception** part analyzes the radar data to identify objects within the near range, calculating their distance. Based on this analysis, the **Processing and Decision Making** part determines if the detected object, in this case 16 a ciclyst, poses a collision risk, particularly during a right turn. If necessary, it triggers the **Actuation** system, which issues warnings with audio alert and activate automatic braking and steering to avoid the cyclist.

Figure 16: Turn Assist Simulation Frame

In the implemented scenario, the ego vehicle is given a random destination

that requires it to turn right, while a bicycle is spawned on its right side. The system uses radar-based detection to monitor the bicycle's position during the maneuver. If the bicycle is detected within the vehicle's turn path, the system can prevent the turn applying a small change in the trajectory of the vehicle and apply brakes to avoid the bicycle.

The simulation integrates **real-time detection and autonomous control**: the ego vehicle is driven by a navigation agent toward its target, while the bicycle moves in parallel. A radar manager continuously evaluates the environment to decide whether to allow the maneuver or take action. The effectiveness of this detection and response system is validated using a ***collision sensor.***

This combined approach of BSD and VTA provides comprehensive safety support for complex urban maneuvers, particularly those involving vulnerable road users like cyclists.

# 3 Technologies

In this section we present the technologies involved in these project.

## 3.1 CARLA Simulator and Python API

The CARLA Simulator is an open-source platform designed for autonomous driving research. It provides a realistic urban environment for testing and validating self-driving algorithms, including perception, planning, and control. CARLA supports flexible configurations of sensors and scenarios, making it an essential tool for developing and evaluating autonomous vehicle systems [2]. In this project, we used many of CARLA's components such as radar sensors, cameras, debugging visualization tools, and object blueprints (vehicles, pedestrians, bicycles, etc). CARLA also allows for dynamic weather conditions, enabling simulation testing under various environmental scenarios. The Python API provided by CARLA was used to control actors, configure sensors, and manage the simulation environment programmatically. This API allowed for control over the simulation, including the spawning and destruction of actors, real-time data acquisition from sensors, and custom behavior implementation. We also use elements like the `BasicAgent` [1].
The `BasicAgent` provides autonomous navigation capabilities using a built-in local planner, allowing vehicles to follow a set of waypoints or reach specific destinations while avoiding collisions.

## 3.2 YOLOv8

YOLOv8 offers state-of-the-art object detection and image segmentation capabilities, framed as a single regression problem from image pixels to bounding box coordinates and class probabilities. YOLOv8 is renowned for its speed and accuracy, making it suitable for applications requiring real-time processing, such

as autonomous driving. YOLOv8 introduces architectural improvements over its predecessor, YOLOv5. These enhancements contribute to improve accuracy and efficiency across various tasks. Performance metrics indicate that YOLOv8 achieves a higher mean Average Precision (mAP) with comparable or improved inference speeds compared to YOLOv5 [7].

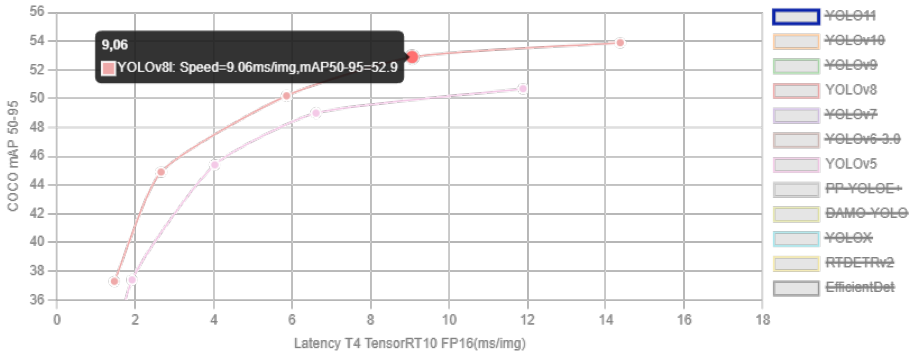


Figure 17: Graphic of perfomance comparison between yolov5 and yolov8

In this project, the YOLOv8s model was chosen over YOLOv5s due to its superior performance in terms of accuracy shown in Figure 17 and its ability to handle a broader range of vision tasks beyond object detection, such as instance segmentation and pose estimation.

| Modello | dimensione(pixel) | mAPval50-95 | VelocitàCPU ONNX(ms) | VelocitàT4TensoreRT10(ms) | params(M) | FLOP(B) |
|---|---|---|---|---|---|---|
| YOLOv8n | 640 | 37.3 | 80.4 | 1.47 | 3.2 | 8.7 |
| YOLOv8s | 640 | 44.9 | 128.4 | 2.66 | 11.2 | 28.6 |
| YOLOv8m | 640 | 50.2 | 234.7 | 5.86 | 25.9 | 78.9 |
| YOLOv8l | 640 | 52.9 | 375.2 | 9.06 | 43.7 | 165.2 |
| YOLOv8x | 640 | 53.9 | 479.1 | 14.37 | 68.2 | 257.8 |

Figure 18: Different model of yolov8

In figure 18 [8] are shown the charateristics of different type of model of yolov8. In our case we use yolov8s to have the right compromise between performance and model size.

## 3.3 HiveMQ - MQTT Service

HiveMQ is a scalable and reliable MQTT broker designed for IoT applications. It facilitates efficient, bidirectional communication between devices and cloud, ensuring data transfer in real-time systems. HiveMQ's service is particularly useful in scenarios requiring low latency and high throughput. In this project

it is used to enhance the communication with MQTT protocol garantee the publish/subscribe pattern and payload in JSON format. In our project we use JSON Format to transfer payload from the ego vehicle to the broker. Messages in a standard format make the system extendible in such way in which a data manipulation is needed or MQTT can be used to communicate in real time with other sensors inside the car or V2X Infrastructure. In Figure 19 an example of JSON payload used during tests scenarios.

```
message = {
    "event": "Blind Spot Detection",
    "side": side,
    "coordinates": {"x": round(x, 2), "y": round(y, 2)},
    "velocity": round(velocity, 2),
    "timestamp": time.time()
}
```

Figure 19: JSON Payload of BSD Simulation

### 3.4 Conda

Conda is an open source package management system and environment management system for installing multiple versions of software packages and their dependencies and switching easily between them. It works on Linux, OS X and Windows, and was created for Python programs but can package and distribute any software [3]. In this project Conda is used to create the virual environment in which execute the simulation. In the environment all the software must be installed in order to be executed inside. The requirements and installation are explained in Section 6.

## 4 Code

In this section are explain the main parts of the implementation showing the relevant code parts.

### 4.1 Blind Spot Detection

This scenario evaluates the detection of a vehicle located in the left-side blind spot using radar data. The ego vehicle follows a set of waypoints, while a second vehicle (the leader) is followed with a safe distance. A radar placed on the left side detects potential obstacles that may not be visible through eyes.

- **Waypoint Generation and Following:**
  - The ego vehicle follows a route by adjusting throttle and steering based on angle difference with the next waypoint [5].

– Control logic adapts throttle and brake to reach and maintain a target speed.

– The waypoints calculation is obtained calculating a traslation of the main path.

To compute the angular difference between the vehicle's current orientation and the direction towards the next waypoint, the main point is the following formula:

$$\theta = \text{atan2}(\text{x}_\text{p}, \text{y}_\text{p})$$

Where:

– $\theta$ is the angle between the vehicle and next waypoint.

– $x_p$ and $y_p$ are the coordinate of next waypoint.

With this formula we can obtain the angle between the cartesian axis and the point. Knowing the angle we can calculate the angular difference between the waypoint and the current orientation of the vehicle and set the steer value accordingly.

```
    def generate_parallel_waypoints(self, waypoints, offset):
        parallel_waypoints = []
        for wp in waypoints:
            location = wp.transform.location
            right_vector = wp.transform.get_right_vector()

            parallel_location = carla.Location(
                location.x + offset * right_vector.x,
                location.y + offset * right_vector.y,
                location.z
            )
            parallel_transform = carla.Transform(parallel_location, wp.
                transform.rotation)
            parallel_waypoints.append(parallel_transform)
        return parallel_waypoints


        def follow_waypoints(self, vehicle, waypoints, target_speed=20.0):
            . . .
            while True:
                vehicle_location = vehicle.get_location()
                distance = vehicle_location.distance(location)

                control = carla.VehicleControl()
                vector = location - vehicle_location
                yaw = math.atan2(vector.y, vector.x)
                current_yaw = math.radians(vehicle.get_transform().rotation.
                    yaw)

                #angle difference + normalization
                angle_diff = (yaw - current_yaw + math.pi) % (2 * math.pi) -
                    math.pi
                steer = max(-1.0, min(1.0, angle_diff))

                #actual velocity
                velocity = vehicle.get_velocity()
```

```
                current_speed = math.sqrt(velocity.x**2 + velocity.y**2 +
                    velocity.z**2)

                #throttle and braking
                if current_speed < target_speed:
                    control.throttle = 0.5
                    control.brake = 0.0
                elif current_speed > target_speed:
                    control.throttle = 0.0
                    control.brake = 0.5
                else:
                    control.throttle = 0.2

                control.steer = steer
                vehicle.apply_control(control)

                if distance < 1.0:
                    break
                time.sleep(0.1)
```

## 4.2 Pedestrian Detection

This section focuses on detecting and reacting to pedestrians that suddenly cross the road in front of the ego vehicle.

- **Safety Distance Calculation:** The system dynamically computes a safety distance based on the vehicle's speed using the rule:

$$\text{safety\_distance} = \left(\frac{v_{km/h}}{10}\right)^2$$

- **Path Checking:** The pedestrian's position is transformed into the vehicle's local frame [9] to determine if they are in the path. In this implementation of simulation the pedestrian location is computed using the Carla.Location property considering the distance between coordinates between the pedestrian and the vehicle.

$$\Delta x = x_p - x_v, \quad \Delta y = y_p - y_v$$

and the vehicle's yaw angle $\theta$ (in radians), the pedestrian's position in the local reference frame of the vehicle is:

$$\begin{bmatrix} x_{\text{local}} \\ y_{\text{local}} \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

$$x_{\text{local}} = \Delta x \cdot \cos\theta + \Delta y \cdot \sin\theta$$

$$y_{\text{local}} = -\Delta x \cdot \sin\theta + \Delta y \cdot \cos\theta$$

- **Control Logic Loop:**

  - If a high-confidence pedestrian is in trajectory path apply **Full-Brake**.
  - If confidence is lower apply **Partial brake**.
  - If no detection apply **Normal Cruise**.

- **Pedestrian Detection:**
  - Uses a YOLO-based model to detect class "person" in the camera feed [6].
  - Detections are highlighted and classified based on confidence.

- **MQTT Alerts:** For each detection, a JSON message is sent throught MQTT Protocol in `vehicle/bsd` topic with the action, confidence and other useful info.

```python
def compute_safety_distance(self, vehicle):
    velocity = vehicle.get_velocity().x
    velocity_km = velocity * 3.6
    return (velocity_km / 10)**2

def vehicle_control_loop_scenario_2(self):
    while not self.stop_display:
        safety_distance = self.compute_safety_distance(self.vehicle)
        safety_offset = 3
        safety_total = safety_distance + safety_offset
        if self.camera_manager.high_confidence_pedestrian and self.
            is_pedestrian_in_path(safety_total):
            self.vehicle.apply_control(carla.VehicleControl(throttle=0.0,
                brake=1.0))
        elif self.camera_manager.detected_pedestrian and self.
            is_pedestrian_in_path(safety_total):
            self.vehicle.apply_control(carla.VehicleControl(throttle=0.0,
                brake=0.2))
        else:
            self.vehicle.apply_control(carla.VehicleControl(throttle=0.7,
                brake=0.0))
        time.sleep(0.1)

def detect_pedestrian(self, image, frame_number):
    results = self.model(image)[0]
    self.detected_pedestrian = False
    self.high_confidence_pedestrian = False
    for box in results.boxes.data:
        x1, y1, x2, y2, conf, cls = box.cpu().numpy()
        if int(cls) == 0:
            self.detected_pedestrian = True
            if conf > 0.60:
                self.high_confidence_pedestrian = True
            message = self.format_mqtt_message(x1, y1, conf, frame_number)
            self.mqtt_client.publish("vehicle/pedestrian", message)

def format_mqtt_message(self, x, y, confidence, frame_number):
    action = "Braking" if self.high_confidence_pedestrian else (
             "Slowing down" if self.detected_pedestrian else "No action")
    return json.dumps({
        "event": "Pedestrian Detection",
        "action": action,
        "confidence": confidence
    })
```

## 4.3 Bicycle Overtaking

This test simulates a bicycle overtaking the ego vehicle from the right side. The goal is to detect the bicycle using a lateral radar and apply a light avoidance steer to prevent a side collision.

**Main Features:**

- **Environment Setup:**
  - The ego vehicle is spawned and directed to a predefined destination using a BasicAgent.
  - A bicycle is spawned slightly behind and to the right of the ego vehicle using:
- **Lateral Radar:**
  - A right-facing radar is attached to the ego vehicle to monitor the lateral area.
  - The radar detects if an object is within the "danger zone" on the right:

    $$-1 < x < 3.5, \quad 0 < y < 2$$

    where $(x, y)$ are coordinates of the detected object in the vehicle's local frame.
- **Collision Avoidance:**
  - If the ego vehicle is turning right and a bicycle is detected laterally, an avoidance maneuver is triggered the following action are activated: Light brake, steering left, throttle reduction
  - After the maneuver, the control is reset to the normal driving state.
- **MQTT Communication:**
  - A message is published to `vehicle/bsd` topic containing the detection type, coordinates, and object velocity.
  - A warning sound is also played.

```
ego_transform = self.ego_vehicle.get_transform()
bicycle_spawn_location = carla.Location(
    ego_transform.location.x - ego_transform.get_forward_vector().x * 5.0 +
    ego_transform.get_right_vector().x * 4.0,
    ego_transform.location.y - ego_transform.get_forward_vector().y * 5.0 +
    ego_transform.get_right_vector().y * 1.45,
    ego_transform.location.z
)
self.bicycle = self.env_manager.spawn_bicycle_at_location(
     bicycle_spawn_location)
lateral_right_spot = (-1 < x < 3.5) and (0 < y < 2)
if lateral_right_spot:
    if control.steer > 0.1:  #vehicle is turning right?
        control.throttle = 0.1
        control.brake = 0.5
```

```
        control.steer = -0.1
        self.vehicle.apply_control(control)
        time.sleep(1)
        control.steer = 0.0
        control.brake = 0.0
        self.vehicle.apply_control(control)
    message_json = self.format_mqtt_message("LATERAL_RIGHT", x, y,
        object_velocity)
    self.mqtt_client.publish("vehicle/bsd", message_json)
    self.beep_sound.play()
```

# 5 Testing

In this section, we describe the approach adopted to test the ADAS scenarios developed in the CARLA simulator, including the methodology, specific test scenarios, and obtained results. In this project, we implemented automated tests for Bicycle Detection and Pedestrian Detection.

**Methodology** For Pedestrian Detection (Sudden Pedestrian Crossing) and Bycicle Overtaking (Bicycle Overtaking at Intersection) to evaluate the ADAS functionalities in different scenarios, we implemented an automated system that repeatedly executes each test case. Each run is classified as a success or failure based on collision of the ego vehicle between the ego vehicle and the actor of the test. For the Blind Spot Detection (BSD) scenario instead, testing was conducted manually since it primarily involves tracking vehicles in adjacent lanes rather than critical real-time interventions.

For the success of the test we consider, for each scenario, the collision with the actors of the test. This sensor is a built-in sensor of CARLA simulator. The collision sensor is installed on the ego vehicle and percept all the collision. In case of collision the test is considered failed.

## 5.1 Test Scenarios

For test scenarios we create different conditions of wheater. In particular with the following weather:

- CloudyNoon
- MidRainSunset
- HardRainNoon
- ClearSunset
- SoftRainNight
- Default

For each weather the execution is started and data of success or failure are collected.

**Sudden Pedestrian Crossing** In this scenario, a pedestrian suddenly crosses the road in front of the ego vehicle. The objective is to evaluate the pedestrian detection system's responsiveness and assess whether, under varying weather conditions, the ADAS system can timely detect the pedestrian and avoid a collision. The focus is on the effectiveness of automatic braking and the system's capacity to mitigate risks in critical real-time situations.

| Test | Weather | $n$ | Success | Failure | % of Success |
|---|---|---|---|---|---|
| Pedestrian | CloudyNoon | 70 | 51 | 19 | 72.85% |
| Pedestrian | MidRainSunset | 70 | 57 | 13 | 81.42% |
| Pedestrian | HardRainNoon | 70 | 56 | 14 | 80.00% |
| Pedestrian | ClearSunset | 70 | 52 | 18 | 74.28% |
| Pedestrian | SoftRainNight | 70 | 34 | 36 | 48.57% |
| Pedestrian | Default | 70 | 40 | 30 | 57.14% |

Table 4: Sudden Pedestrian Crossing – Test Results by Weather Condition

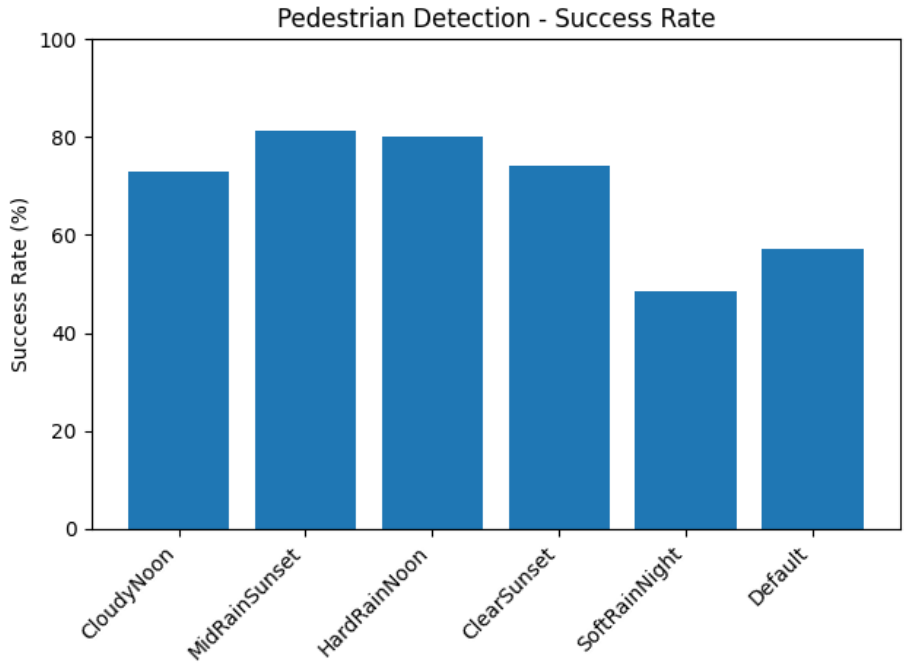


Figure 20: Graph of the test results

**Bicycle Overtaking at Intersection** In this scenario, a bicycle overtakes the ego vehicle from the right at an intersection. The test assesses whether the

ADAS system can correctly detect the bicycle and, if the ego vehicle attempts to turn right, whether it performs the necessary braking and applies *corrective steering* to avoid a collision. The aim is to verify both detection accuracy and the ability of the system to execute preventive maneuvers in dynamic traffic environments.

| **Test** | **Weather** | $n$ | **Success** | **Failure** | **% of Success** |
|---|---|---|---|---|---|
| Bicycle | CloudyNoon | 50 | 37 | 13 | 74.00% |
| Bicycle | MidRainSunset | 50 | 30 | 20 | 60.00% |
| Bicycle | HardRainNoon | 50 | 18 | 32 | 36.00% |
| Bicycle | ClearSunset | 50 | 37 | 13 | 74.00% |
| Bicycle | SoftRainNight | 50 | 21 | 29 | 42.00% |
| Bicycle | Default | 50 | 33 | 17 | 66.00% |

Table 5: Bicycle Overtaking at Intersection – Test Results by Weather Condition

The data shows that the ADAS system maintains high performance in standard weather conditions, while performance degrades in the presence of heavy rain. This indicates a need for further optimization of sensor fusion and detection algorithms under reduced visibility and adverse environmental factors.

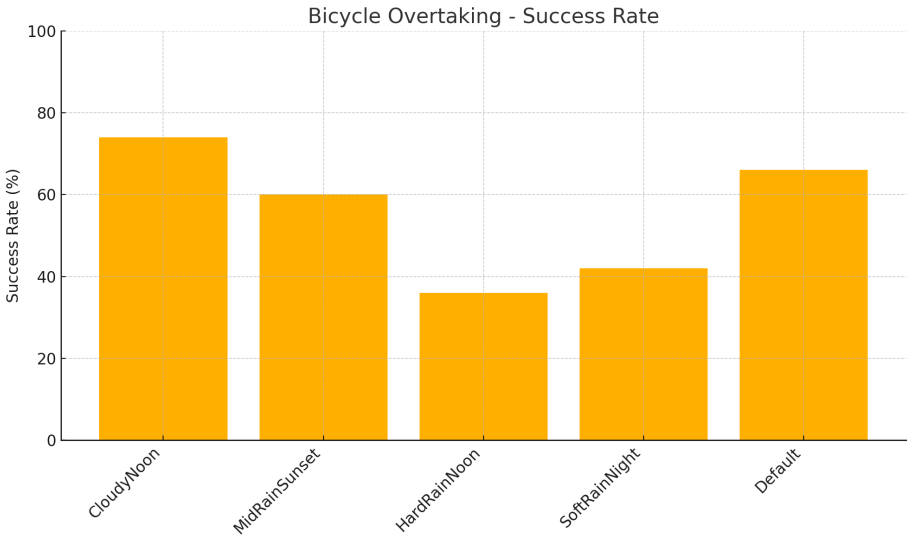


Figure 21: Graph of the test results

# 6 Deployment

Here is the instructions to execute the software.

## 6.1 Prerequisites

- Anaconda
- Carla Simulator

For prerequisites installation refer to the official documentation in Carla and Anaconda sites.

## 6.2 Execution

1. **Create a new Conda environment with Python 3.8**

   ```
   conda create -n svs-env-test python=3.8
   ```

2. **Clone the repository from GitHub**

   ```
   git clone https://github.com/davidedimarco00/SVS-Project
   ```

3. **Activate the Conda environment**

   ```
   conda activate svs-env-test
   ```

4. **Execute the script to install dependencies**

   ```
   ./install_dependencies.sh
   ```

5. **Start Carla Simulator**

   ```
   <Carla Path>\CarlaUE4.exe
   ```

6. **Execute the main script**

   ```
   python main.py
   ```

# 7 Conclusions

In this project, we designed, developed, and tested different ADAS (Advanced Driver Assistance Systems) scenarios using the CARLA simulator. We focused on important functions like blind spot detection, pedestrian detection, and bicycle overtaking at intersections. The software we built is modular and easy to expand. This means we can add new features or test more scenarios in the future without major changes. The CARLA simulator helped us create realistic urban driving situations, including different weather conditions, moving vehicles, and complex situations on the road. We used radar sensors, RGB cameras, and a YOLOv8s object detection model to detect important events in real time. For example, we were able to detect pedestrians crossing the road, bicycles passing the car while turning, and vehicles appearing in the blind spot. We used MQTT with JSON messages to send alerts about possible dangers. This communication system could also be used to connect with other vehicles or external systems (V2X). Overall, this project showed that simulation is a useful and safe way to test ADAS systems. In the future, we could add new functions like traffic sign recognition or lane keeping assist, connect the system with real sensor data, and improve the vehicle's automatic control.

# References


[1] Carla agent. `https://carla.readthedocs.io/en/0.9.12/adv_agents/`. Accessed: 2025-03-21.

[2] Carla simulator. `https://carla.org/`. Accessed: 2025-03-21.

[3] Conda. `https://anaconda.org/anaconda/conda`. Accessed: 2025-03-21.

[4] Hivemq - mqtt broker. `https://www.hivemq.com/`. Accessed: 2025-03-21.

[5] Math.atan2 for angle calculation. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/atan2`. Accessed: 2025-03-21.

[6] Pedestrian detection in yolo8v. `https://blog.stackademic.com/step-by-step-to-surveillance-innovation-pedestrian-detection-with-yolov8-and-python-oper` Accessed: 2025-03-21.

[7] Yolo by ultralytics. `https://ultralytics.com/yolo`. Accessed: 2025-03-21.

[8] Yolov8-yolov5 comparison. `https://docs.ultralytics.com/it/compare/yolov5-vs-yolov8/`. Accessed: 2025-03-21.

[9] Lynn Seaman. Rotation transformations for two-dimensional calculations. *International Journal of Solids and Structures*.