# User Manual

Running the CARLA Map:

1. On your keyboard, hit the windows key, and type in ‘cmd’.
2. Once command prompt has opened, navigate to the CARLA 0.9.13 folder.
    a. This can be done by typing ‘cd [path]
        i. An example path would be “C:\Users\John\Desktop\CARLA_0.9.13\WindowsNoEditor”
3. Once in the folder’s directory, type “CARLAUE4.exe” and press enter.
    a. This should open the background map needed to run the external python scripts.

Running External Python Scripts:

1. Once the world has opened, navigate to the “PythonAPI” and then “examples” folder.
2. Open another command prompt by following step 2 in “Running the CARLA map” steps, and navigate the command prompt to the examples folder.
3. Once in the examples folder, you may run any python script desired within.
    a. In order to do this, the user must type “python3 [script name]”
        i. For example, “python3 collab_control.py”
            1. Including the “.py” at the end of the file is extremely important, the file will not start without it.

## Scenario Runner

### Introduction

**1.1 Overview:** Scenario Runner with Carla is a simulation framework designed for testing and evaluating autonomous driving algorithms. It allows users to define complex scenarios and execute them within the Carla simulator environment. By simulating various driving conditions and scenarios, developers can assess the performance and reliability of their autonomous driving systems.

**1.2 Purpose:** The purpose of this user manual is to provide comprehensive guidance on using Scenario Runner with Carla. It covers everything from installation and setup to defining scenarios, running simulations, and troubleshooting common issues.

**1.3 Scope:** This manual is intended for developers, researchers, and enthusiasts interested in autonomous driving technology. It assumes a basic understanding of Python programming and familiarity with the Carla simulator. Users will learn how to define custom scenarios, execute simulations, and analyze results using Scenario Runner.

**Installation and Setup:** Before using Scenario Runner with Carla, ensure that you have both tools installed and configured properly on your system. Follow the installation instructions provided by the respective projects to set up Scenario Runner and Carla. Ensure that all dependencies are installed and any necessary environment variables are set. It is recommended that you create a virtual environment before you begin the installation process.

**2.2 Requirements:**

- Python 3.8
- Carla Simulator
- Scenario Runner
- Operating System: Windows or Linux

**2.3 Compatibility:** Scenario Runner and Carla are regularly updated with new features and improvements. Ensure that you are using compatible versions of both tools to avoid compatibility issues. Check the official documentation and release notes for any version-specific requirements or updates

**Defining Scenarios**

**3.1 Scenario Structure:** Scenarios in Scenario Runner are defined using Python scripts. These scripts describe the initial state of the simulation, the sequence of events to be executed, and the conditions for success or failure. The python scripts should be saved in the ***scenario_runner/srunner/scenarios*** folder. A scenario typically consists of:

- Initialization: Setting up the simulation environment.
- Behavior: Defining the sequence of actions and events.
- Test Criteria: Specifying conditions for success or failure.

**3.2 Scenario Classes:**

Scenario classes are fundamental components in Scenario Runner that define the structure, logic, and objectives of a simulation scenario. These classes encapsulate the behavior, events, and conditions that drive the simulation forward and determine its outcome. Understanding scenario classes is crucial for designing, implementing, and executing scenarios effectively in Scenario Runner.

**Key Components of Scenario Classes:**

1. **Inheritance from ScenarioConfiguration:**
    - All scenario classes must inherit from the **BasicScenario** base class provided by Scenario Runner. This base class provides essential methods and attributes for defining and executing scenarios.
2. **Initialization:**
    - The **__init__** method of the scenario class is used for initializing the scenario configuration. It typically accepts parameters such as the simulation world, traffic manager, and any additional configuration options.
3. **Scenario Behavior:**

    - The **_create_behavior()** method is where the main scenario logic is defined. This method specifies the sequence of actions, events, and behaviors that occur during the simulation. Actions such as spawning actors, setting their initial state, defining their movement, and triggering events are defined here.
4. **Test Criteria:**
    - The **_create_test_criteria()** method is responsible for specifying the conditions that determine whether the scenario is successful or not. Test criteria define the objectives, goals, or requirements that the scenario must meet for it to be considered successful. These criteria are evaluated during the simulation to determine the scenario outcome.

**Example Scenario Class:**

```
from srunner.scenarioconfigs.scenario_configuration import
ScenarioConfiguration

class MyScenario(BasicScenario):
    def __init__(self, world, traffic_manager,
debug_mode=False):
        super(MyScenario, self).__init__(world, traffic_manager,
debug_mode)

    def _create_behavior(self):
        # Define the behavior logic here
        self.add_action(SpawnActor(actor_type="vehicle", ...)

self.add_action(ActorTransformSetter(actor_id="vehicle_1",
transform=...))
        # Add more actions as needed

    def _create_test_criteria(self):
        # Define the test criteria for success or failure

self.add_test_criteria(ReachPositionCriteria(actor_id="vehicle_1
", target_position=...))
```

**3.3 Actors:**

**XML Files for Actors:**

Scenarios in Scenario Runner often depend on XML files that contain information about the type of actor to be spawned and their positions in the simulation. These XML files serve as blueprints for configuring the initial state of the scenario. They define parameters such as the actor type, initial position, velocity, and other relevant attributes. The xml files for each scenario should be saved in ***scenario_runner/srunner/examples*** folder.

**Defining Actors in XML Files:**
The XML files typically include tags or elements for each actor to be spawned in the simulation. These elements specify various properties of the actor, such as:

- **Type:** The type of actor to be spawned, such as vehicle, pedestrian, or obstacle.
- **ID:** A unique identifier for the actor within the simulation.
- **Position:** The initial position of the actor in the simulation environment.
- **Velocity:** The initial velocity of the actor, if applicable.
- **Other Attributes:** Additional attributes specific to the type of actor, such as color, size, behavior, etc.

**Example XML File:**
```
<?xml version="1.0"?>
<scenarios>
    <scenario name="ChangeLane_1" type="ChangeLane"
town="Town04">
        <ego_vehicle x="284.4" y="16.4" z="2.5" yaw="-173"
model="vehicle.lincoln.mkz_2017" />
                 <!-- spawn actors-->
                 <other_actor x="264.4" y="16.3" z="-500"
yaw="-179" model="vehicle.tesla.model3" />
                 <other_actor x="184.4" y="14.9" z="-500"
yaw="-176" model="vehicle.volkswagen.t2" />
        <weather cloudiness="0" precipitation="0"
precipitation_deposits="0" wind_intensity="0"
sun_azimuth_angle="0" sun_altitude_angle="75" />
    </scenario>
```

**Key Components of Test Criteria:**

1. **Definition:**
    - Test criteria are typically defined within the **_create_test_criteria()** method of the scenario class. This method is responsible for specifying the test criteria based on the scenario's objectives and requirements.
2. **Evaluation:**
    - Test criteria are evaluated continuously during the simulation to determine whether they are met or not. Each test criterion is monitored and assessed based on the behavior and state of the simulation entities (actors, environment, etc.).
3. **Success/Failure Conditions:**

- Test criteria specify the conditions that constitute success or failure for the scenario. These conditions may include reaching a specific waypoint, avoiding collisions, adhering to traffic rules, or achieving a certain level of performance.

4. **Multiple Criteria:**
    - A scenario can have multiple test criteria, each representing a different aspect or requirement of the scenario. By defining multiple criteria, developers can assess various aspects of the algorithm's performance and behavior in different situations.

**Example Test Criteria:**
```
from srunner.scenariomanager.scenario_criteria import
CollisionTest, ReachedRegionTest

def _create_test_criteria(self):
    # Test criteria for avoiding collisions
    collision_test = CollisionTest(self.ego_actor)
    self.add_test_criteria(collision_test)

    # Test criteria for reaching a specific region
    region_test = ReachedRegionTest(self.ego_actor,
region_id="target_region")
    self.add_test_criteria(region_test)
```

**Running Scenarios:**
To execute scenarios in Scenario Runner, you'll typically use a Python script to initiate the simulation. Here's a step-by-step guide on how to execute scenarios:

1. **Start Carla:**
    - Before running scenarios, ensure that the Carla simulator is running with the desired map and settings. Launch Carla and wait for the simulation environment to load completely.
2. **Prepare Scenario Setup:**
    - Ensure that the scenario setup, including the scenario class definition, XML files for actor configuration, and any additional dependencies, is ready and properly configured.
3. **Execute Scenario Script:**
    - Open a command prompt or terminal window and navigate to the Scenario Runner root directory.
    - Use the following command to execute the scenario: ***python scenario_runner.py --scenario scenario_name –reloadWorld***. Replace **scenario_name** with the name of the scenario you want to execute, as specified in the xml file.

   - This command initiates the execution of the specified scenario within the Carla simulator environment. The **--reloadWorld** flag ensures that the world is reloaded before running the scenario, which may be necessary to reset the simulation environment.
4. **Controlling the Ego Vehicle:**
   - In a separate command prompt or terminal window, navigate to the Scenario Runner root directory.
   - Run the collaborative control Python file to control the car. *python collaborative_control.py.*
   - This command initiates the collaborative control mechanism, allowing you to control the car's behavior or intervene in the simulation as needed.

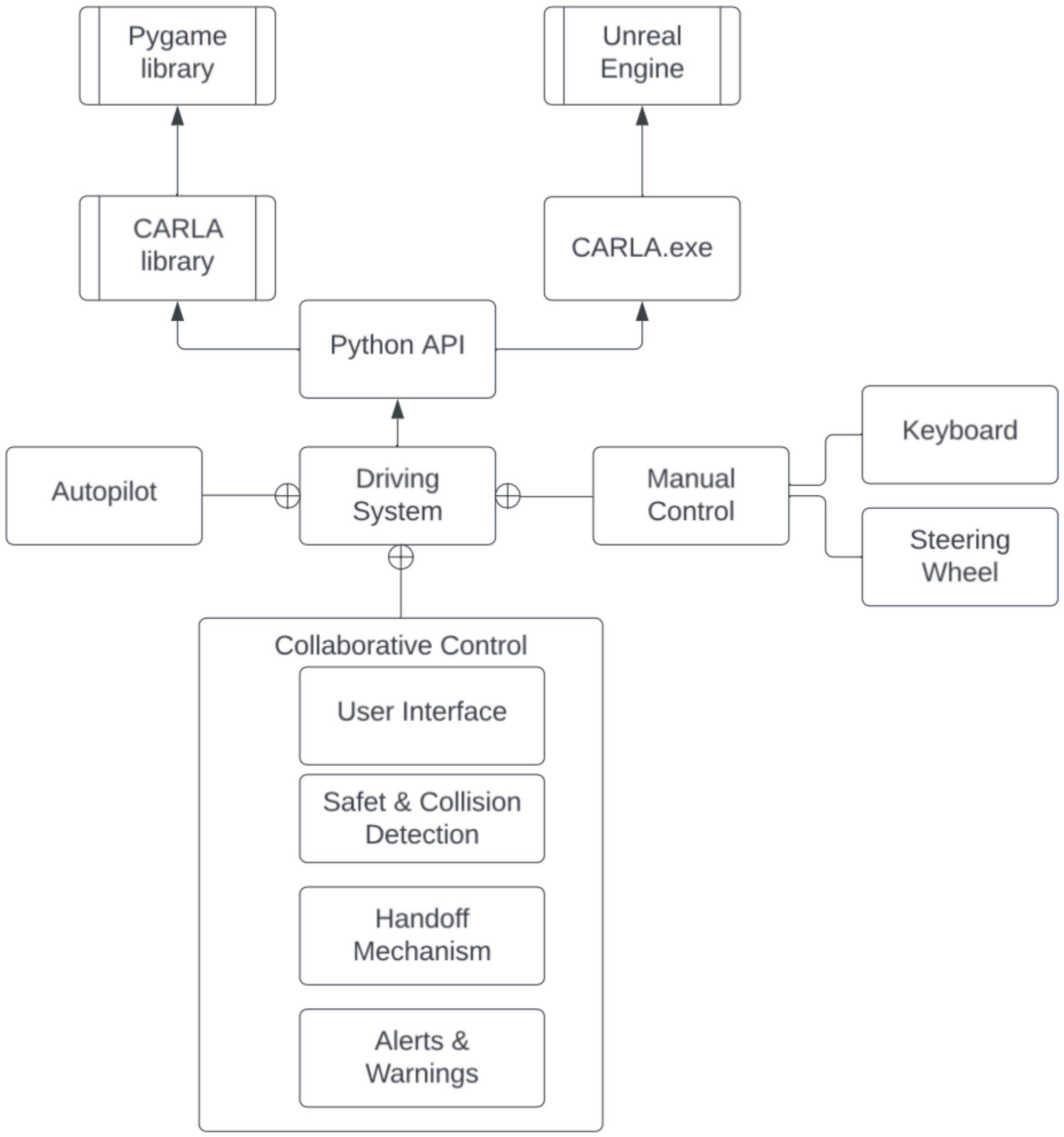
By following these steps, you can execute scenarios in Scenario Runner, control the car using collaborative control, and analyze simulation results effectively.

**Common Issues:**

1. **Scenario Not Executing:**
   - If the scenario does not execute or terminates prematurely, check the scenario definition for errors or inconsistencies. Ensure that all necessary dependencies are installed and properly configured.
   - Verify that Carla is running and accessible, and the scenario setup is correct.
   - Check the logs for any error messages or exceptions that may provide clues to the cause of the issue.
2. **Actor Placement Errors:**
   - If actors are not placed correctly or do not behave as expected, double-check the XML files containing actor configurations. Ensure that actor positions, orientations, and other attributes are specified accurately.
   - Verify that the actor spawning logic in the scenario script is functioning correctly and that actors are spawned in the desired locations and states.
3. **TensorFlow and NumPy Compatibility:**
   - In some cases, using incompatible versions of TensorFlow and NumPy may result in compilation errors or runtime issues when running Scenario Runner with Carla. These errors can manifest as module import errors or incompatible library versions during scenario execution.
   - Before running Scenario Runner with Carla, check the requirements section of the Carla and Scenario Runner documentation to determine the recommended versions of TensorFlow and NumPy for your setup.
   - Use package managers such as pip or conda to install specific versions of TensorFlow and NumPy, ensuring compatibility with Carla and Scenario Runner.

**Developer Manual**

Design Diagram:



Above is the design diagram for the system. This was created during the early stages of the project, and the idea has not changed. Unreal Engine is the engine used to create map, actors, vehicles, etc, which can then be open by running the executable file. Pygame is the library implementation

**Creating a Map in RoadRunner:**
RoadRunner is a powerful mapping tool designed to create detailed maps for various applications. This manual provides a comprehensive guide to creating a map on RoadRunner and exporting it for use in CARLA, a popular open-source simulator for autonomous driving research.

***System Requirements:***

- RoadRunner software installed
- Computer with minimum specifications (check RoadRunner documentation)
- Data sources for map creation (satellite imagery, GPS data, etc.)

***Installation:***

1. Download the RoadRunner installer from the official website.
2. Run the installer and follow the on-screen instructions to complete the installation process.

***Step 1: Launching RoadRunner:***

1. Double-click on the RoadRunner icon to launch the application.
2. If prompted, log in with your credentials.

***Step 2: Setting Up a New Project:***

1. Click on "New Project" or navigate to File > New Project.
2. Name your project and choose a location to save it.
3. Set the project parameters such as map size and coordinate system.

***Step 3: Editing the Map:***

1. Use the editing tools provided by RoadRunner to modify the map according to your requirements.
2. Add roads, landmarks, buildings, and other features as needed.
3. Ensure accuracy and detail in your map for optimal performance.

***Step 4: Exporting the Map to CARLA***

1. *Made a documentation of the entire process on the lab pc, will paste it here tomorrow morning.*

Creating New Python Examples:

Creating new Python examples is quite difficult to write a step-by-step guide for, as it is different per each example. However, you may follow these steps as a baseline:

1. If looking to edit an already existing program, all the user must do is open the program in their preferred editor (Notepad++, Visual Studio Code, PyCharm, etc)
2. From there, the user may look through the entire program, and edit/add as needed.
    a. In the case of the Collaborative Control of Autonomous Cars team, they started with the "manual_control.py" file, and edited/added components from there in order to achieve the mixture of self and user driving.
3. If the user wishes to create their own code from scratch, they must make sure that they are importing all necessary libraries:
    a. Pygame, numpy, argparse, collections, datetime, logging, math, random, re, weakref, etc.
        i. The user can look through other python examples already within the folder and import accordingly.

HUD class:

The HUD class located within the collaborative control is the information center of the program. Besides the dashboard that has been implemented, this HUD was already created and displayed on the left side of the screen when CARLA was downloaded. This HUD shows things like the current vehicle, vehicle speed, vehicle gear, and much more.

Dashboard:

The dashboard itself is not defined in a separate class, instead it is initialized in the beginning of the program. The dashboard consists of a black background and a steering wheel that is reactive based on the Logitech steering wheel input. It can turn at a total of 540 degrees in either direction (1:1 with a real vehicle).