

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Fakultät für Informatik

CSR-26-10

Integration of Deep learning Algorithms in a Cloud Server-Based Carla Simulation Environment

Anh Tu Nguyen · Julkar Nine · Wolfram Hardt

November 2026

Chemnitzer Informatik-Berichte



TECHNISCHE UNIVERSITÄT
CHEMNITZ

Integration of Deep learning Algorithms in a Cloud Server-Based Carla Simulation Environment

Master Thesis

Submitted in Fulfilment of the
Requirements for the Academic Degree
M.Sc.

Dept. of Computer Science
Chair of Computer Engineering

Submitted by: Anh Tu Nguyen
Student ID: 732738
Date: 02.04.2026

Supervising tutor: Prof. Dr. W. Hardt
(further supervisors)
M. Sc Julkar Nine

Abstract

The integration of computer vision and artificial intelligence into vehicles has given rise to a new research direction: autonomous vehicles. To ensure the new algorithms operate stably and efficiently, a safe environment is required for testing, which is the primary reason for the emergence of simulation systems. CARLA provides a flexible and safe environment to test perception and control algorithms, reducing costs and risks compared to testing vehicles in the real world. However, training models, launching simulations, and implementing machine learning algorithms within simulations require huge hardware resources and computing power, which is often challenging to meet on local machines. This research will develop a solution using a client-server model with TCP/Socket connection protocol to communicate and offload computation to powerful GPU servers (cloud-based). This system allows for large-scale experiments on the CARLA simulation environment (including integrating deep learning models) without hardware limitations on local computers. With this research, deep learning models can be integrated onto GPU servers, enabling multiple training runs to be conducted in parallel, facilitating flexible model updates, and accelerating the development cycle.

Keywords: CARLA, Cloud-Based System, YOLO, Hardware Acceleration, Object Detection.

Contents

Contents . . . 3

List of Figures . . . 5

List of Tables . . . 7

List of Abbreviations . . . 8

1 Introduction . . . 10
1.1 Problem Statement . . . 10
1.2 Motivation . . . 11

2 State of the Art . . . 12
2.1 Simulation Systems . . . 12
2.2 Integrate Machine Learning Algorithms into the Simulator . . . 17
2.2.1 Object Detection Algorithms . . . 17
2.2.2 Integrate Object Detection to Simulator . . . 24
2.3 Offloading Computational Task to Cloud . . . 27

3 Methodology . . . 32
3.1 System Architecture . . . 32
3.2 System Workflow . . . 35
3.3 Client–Server Connectivity . . . 38
3.4 Integrate Deep-learning Models . . . 42
3.5 Stream Data . . . 45

4 Implementation . . . 48
4.1 Tools and Techniques . . . 48
4.2 Socket Communication and TCP Protocol . . . 51
4.3 Manage Session and Data . . . 56
4.4 Communication with Simulation Environment . . . 58
4.5 Integrate Deep-learning Models and Execute Computational Task . . . 61
4.6 Stream Data to Local PC Workstations . . . 63
4.7 User Interface (GUI) for the System . . . 66

5 Result and Evaluation **69**
5.1 Result 69
5.1.1 Overall system implementation 69
5.1.2 Communication result 70
5.1.3 File upload and execution result 71
5.1.4 Streaming result 73
5.1.5 GUI result 75
5.2 Evaluation 76
5.2.1 System Performance 76
5.2.2 Multi-User Capability and Scalability 80
5.2.3 Security and Data Validation 81
5.2.4 Reliability and Fault Tolerance 82
5.2.5 UI/UX Design 84

6 Conclusion **86**

Bibliography **88**

List of Figures

2.1 Levels of Driving Automation . . . 12
2.2 NVIDIA DRIVE Sim . . . 13
2.3 Microsoft AirSim Simulator . . . 14
2.4 LGSVL Simulator . . . 15
2.5 CARLA Simulator . . . 16
2.6 Classical traditional and DCNNs object detection . . . 18
2.7 Images from database with different pedestrian . . . 18
2.8 Object detection algorithm using HOG features . . . 18
2.9 SIFT keypoint descriptor . . . 19
2.10 Process of R-CNN: Regions with CNN features . . . 21
2.11 Single Shot MultiBox Detecto (SSD) framework . . . 22
2.12 Process of YOLO methods. . . . 23
2.13 Block diagram of implementation. . . . 24
2.14 Simulation results. . . . 25
2.15 System Overview. . . . 25
2.16 Mobile edge computing for autonomous driving environment. . . . 27
2.17 Autonomous driving edge computing architecture. . . . 29
2.18 Design overview of ECSeg. . . . 31

3.1 Design of System . . . 32
3.2 Docker CARLA Architecture . . . 33
3.3 GPU Server with Computational Server added . . . 34
3.4 The basic architecture of CARLA . . . 34
3.5 Detailed model of the entire system . . . 35
3.6 Workflow of system . . . 36
3.7 Detail of Local PC Workstation . . . 37
3.8 Details of Cloud Server cluster . . . 38
3.9 Overview of Socket . . . 39
3.10 Describe the connection process and compare the two protocols . . . 41
3.11 Python client API connects . . . 42
3.12 Compare BGR and RGB color space . . . 43
3.13 Visualize object detection in CARLA . . . 44
3.14 Describe a packet (frame) using the 4-byte prefix method . . . 45

4.1 Describe the design of the main thread and sub-threads in the system 57
4.2 Describes the server's session directory tree . . . 58

4.3 Describe the communication between the simulation server and the computational server 59
4.4 Basic design of a graphical user interface (GUI) 67
4.5 Compare the workflow of CLI and GUI. 68

5.1 Connect from the client side and establish a session 70
5.2 Client-side operations and session verification 71
5.3 Syntax for uploading files and receiving responses from the server . . 72
5.4 The server receives requests from the client 73
5.5 Start the Python file execution function from the client 73
5.6 This signal initiates the execution of the Python file on the server . . 73
5.7 Visualize server-side frames using Pygame on the client 74
5.8 Graphical User Interface (GUI) visualization 75
5.9 Upload time of Python files and YOLO models from the client 77
5.10 Illustrates the detailed timing breakdown of the system workflow . . . 79
5.11 Handling multiple connections on the server 80
5.12 Describes the process of validating the file type uploaded by the client 82

List of Tables

List of Abbreviations

CARLA Car Learning to Act

TCP Transmission Control Protocol

GPU Graphics Processing Unit

CPU Central Processing Unit

YOLO You Only Look Once

SAE Society of Automotive Engineers

UAV Unmanned Aerial Vehicle

USD Universal Scene Descriptors

LiDAR Light Detection and Ranging

RADAR Radio Detection and Ranging

ADAS Advanced Driver Assistance Systems

AI Artificial Intelligence

IMU Inertial Measurement Unit

GPS Global Positioning System

JSON JavaScript Object Notation

CAN Controller Area Network

ROS Robot Operating System

ROS2 Robot Operating System 2

GNSS Global Navigation Satellite System

HOG Histogram of Oriented Gradients

SIFT Scale-Invariant Feature Transform

SVM Support Vector Machine

DCNN Deep Convolutional Neural Network

CNN Convolutional Neural Network

Faster RCNN Faster Region-Based Convolutional Neural Network

SSD Single Shot Detector

RGB Red, Green, Blue

NMS Non-Maximum Suppression

IoU Intersection over Union

mAP mean Average Precision

VOC 2012 PASCAL Visual Object Classes 2012

PASCAL VOC PASCAL Visual Object Classes

COCO Common Objects in Context

ILSVRC ImageNet Large Scale Visual Recognition Challenge

SSD300 Single Shot Detector with 300 $\times$ 300 input image size

VOC 2007 PASCAL Visual Object Classes 2007

API Application Programming Interface

DRL Deep Reinforcement Learning

RMSE Root Mean Square Error

LSTM Long Short-Term Memory

AWS Amazon Web Services

V2X Vehicle-to-Everything

DoS Denial of Service

DDoS Distributed Denial of Service

MEC Multi-access Edge Computing

AP Average Precision

RSU Roadside Unit

DSRC Dedicated Short-Range Communications

5G Fifth Generation

MPC Model Predictive Control

PC Personal Computer

UNIX UNIX

LINUX Linux

IP Internet Protocol

IPv4 Internet Protocol version 4

LAN Local Area Network

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

REST API Representational State Transfer Application Programming Interface

AMQP Advanced Message Queuing Protocol

MQTT Message Queuing Telemetry Transport

UDP User Datagram Protocol

SYN Synchronize

ACK Acknowledgment

BGRA Blue, Green, Red, Alpha

FPS Frames Per Second

JPEG Joint Photographic Experts Group

ML Machine Learning

GIL Global Interpreter Lock

IO Input / Output

FOV Field of View

UI User Interface

UX User Experience

DHCP Dynamic Host Configuration Protocol

UTC Coordinated Universal Time

EOF End of Frame

GUI Graphic User Interface

CLI Command-Line Interface

1 Introduction

1.1 Problem Statement

The first and second decades of the 21st century witnessed an explosion of information technology, particularly in the fields of computer vision and artificial intelligence. With the integration of computer vision and artificial intelligence into vehicles, a new research direction has emerged: autonomous vehicles. To safely test and refine new algorithms and approaches before deployment or application in practice, researchers need a safe and reliable environment to conduct large-scale experiments with various scenarios. This is the main reason for the emergence of simulation systems; there are many different simulation systems supporting all fields, especially in the automotive and autonomous vehicle systems sector, with CARLA being a prominent and noteworthy example.

CARLA provides a flexible and safe environment to test perception and control algorithms, reducing costs and risks compared to testing vehicles in the real world. However, training models, launching simulations, and implementing machine learning algorithms within simulations require huge hardware resources and computing power, which is often challenging to meet on local machines. For these reasons, cloud computing is a feasible solution, offering a scalable and flexible infrastructure to address the limitations above by enabling resource-intensive models to be trained, deployed, and executed in a distributed manner.

In early traditional research, deep learning has mainly been implemented on local hardware for inference, prediction, and decision-making tasks. While their effectiveness is undeniable, these methods struggle to handle real-time operations with large datasets or complex simulation scenarios. Simulation systems can run on servers and connect to local devices; however, local devices often fail to handle large computational tasks or new algorithms due to hardware limitations. Therefore, the need for a platform that can connect to the simulation environment and execute computational tasks and algorithms entirely on servers is entirely justified. Cloud-based environments offer an effective solution to overcome these challenges by providing high-performance computing resources, parallel processing, and seamless integration with large-scale simulation frameworks.

1.2 Motivation

The rapid development in autonomous driving research has created an urgent need for a consistent and centralized cloud platform that can support diverse testing requirements. Such a platform enables researchers and developers to remotely access the simulation and testing environment from any computer, thereby eliminating the reliance on high-end on-site hardware and physical location. By centralizing computing resources in the cloud, multiple users can simultaneously conduct experiments while sharing GPUs, memory, and simulation infrastructure in a controlled manner. Through this approach, resources are not only improved in terms of cost efficiency but also allow computing resources to be shared among multiple users, ensuring system stability and reliability in a multi-user environment.

To meet the high computational demands of autonomous driving algorithms, cloud servers hold a primary position in enabling the execution of massive computational tasks and the integration of complex deep learning within simulation environments. These servers provide the computing power needed to train, infer, test, and deploy advanced models such as object detection, perception, and decision-making networks at scale. By standardizing the software stack and dependencies on the server side, researchers can create a consistent testing environment, eliminating common problems associated with library conflicts and hardware incompatibilities. Furthermore, seamless model deployment allows deep learning models to be executed directly in remote simulations, enabling efficient testing and validation without the need for repeated local setup. This approach significantly accelerates development cycles and improves reproducibility between experiments.

A scalable cloud-based simulation framework also contributes to bridging the gap between simulation testing and real-world autonomous vehicle performance. By supporting large-scale testing with diverse models and simulation scenarios, researchers can better evaluate system behavior under varying conditions, including rare or safety-critical events. The scalability of the system allows for continuous improvement as new models, sensors, and algorithms are introduced over time. This extensive and systematic testing process enhances the robustness and generalizability of autonomous driving systems, helping to identify weaknesses early in development, improve safety and reliability, and ensuring that autonomous vehicles are better prepared for real-world deployment.

This research will focus on integrating deep learning algorithms within a cloud server–based simulation environment. By leveraging the cloud, the goal is to enhance the scalability, adaptability, and efficiency of simulation-driven experimentation. This approach enables real-time testing and evaluation of intelligent systems under dynamic and diverse conditions, making it particularly relevant for applications in autonomous driving, robotics, and smart infrastructure.

2 State of the Art

2.1 Simulation Systems

The development of testing procedures for autonomous vehicles has been evolving since the advent of autonomous vehicles; however, legal concerns that arise during autonomous vehicle testing, or accidents caused by autonomous vehicle decisions, have posed new challenges, requiring a radical and appropriate solution to address the problem. As defined by Thrun [1], an autonomous vehicle is "[...] an unmanned vehicle that is capable of sensing its environment and navigating without human intervention". Researchers and the Society of Automotive Engineers (SAE) have agreed to develop a standard that divides the "level of autonomy" of autonomous vehicles into six levels, ranging from 0 to 100% [2]. Figure 2.4 details the levels of autonomous driving agreed upon by researchers and the SAE, corresponding to whether an autonomous vehicle can drive partially or fully.



Figure 2.1: Levels of Driving Automation.[1]

Autonomous vehicles need an environment with full conditions similar to reality so that researchers can test new solutions, new algorithms or simply test the vehicle's response to the environment, which is why virtual environments (simulations) for testing were born. In fact, researchers can completely use real environments with real vehicles for testing, but legal and humanitarian issues absolutely do not allow it. Besides, the cost for a single test is not low, imagine when an algorithm fails,

[1] Source: SAE May 3, 2021

it will completely destroy the system, including vehicles, facilities, and if repeated, the cost will create a burden that affects the project. Simulation environments have completely solved this problem, researchers can come up with scenarios suitable for their projects at low cost, in cycles and completely safe.

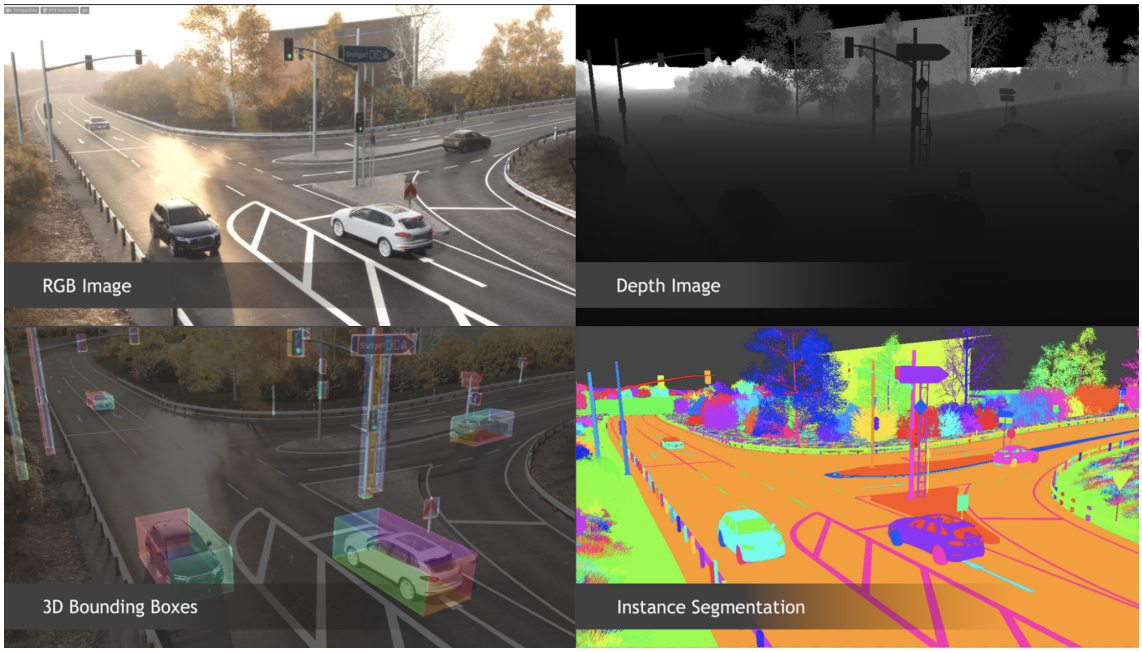


Figure 2.2: NVIDIA DRIVE Sim.[2]

Current simulation environments are divided into many fields with different specialized tasks such as drones/UAVs, robotics, or automotive. In the automotive field, simulation systems are mainly developed by large corporations such as NVIDIA, a giant in the graphics chip industry with NVIDIA Drive Simulator; Microsoft, a major software corporation with AirSim (Car mode); and LG with SVL Simulator. NVIDIA developed NVIDIA DRIVE Sim to address a core challenge of current autonomous vehicle systems: the impossibility of safely, affordably, and comprehensively testing autonomous systems in the real world. Real-world road testing is expensive, time-consuming to set up, and unsafe for rare or hazardous scenarios (e.g., accidents, extreme weather, sensor failures). DRIVE Sim is created based on the NVIDIA Omniverse platform, enabling the creation of physically accurate, photographic-like environments using USD (Universal Scene Descriptors). The simulator is designed not only for testing but also for training deep learning models, generating labeled data for cameras, LiDAR, radar, and sonar sensors. By leveraging its industry-leading GPU capabilities, NVIDIA DRIVE Sim simulates complex physical phenomena and sensor models, including physics-based models of automotive sensors (camera noise, LiDAR detection, radar reflections), bridging the gap between simulation and reality, as shown in Figure 2.2. Furthermore, it's designed

[2] Source: NVIDIA April 12, 2021

to run on cloud-based GPU data centers and clusters, enabling large-scale parallel testing across thousands of scenarios, supporting ADAS certification processes and autonomous driving, and aligning with automotive safety standards. However, NVIDIA DRIVE Sim is not focused on academic or research purposes; this simulator is designed for industrial research labs, large-scale commercial autonomous vehicle programs, or automotive manufacturers (e.g., manufacturers of self-driving cars). Additionally, DRIVE Sim is not an open source platform, limiting its transparency, customization capabilities, and accessibility for academic research. Furthermore, the NVIDIA GPU requirements are extremely high; unsuitable for low-resource systems, small servers, or standard student workstations. This limits flexibility for quick academic experiments and makes it less convenient for rapid prototyping, algorithm testing, or custom scripting compared to research-focused simulators.

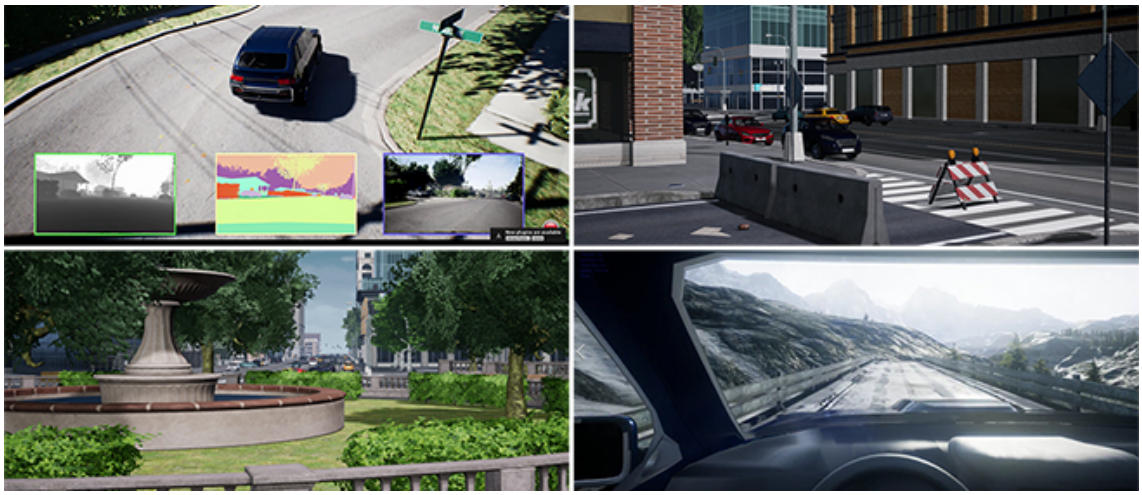


Figure 2.3: Microsoft AirSim Simulator.[3]

AirSim was developed by Microsoft as an open-source simulation platform to accelerate research in autonomous systems and artificial intelligence, as shown in Figure 2.3, which simulates the world and vehicles. The goal of Microsoft when building AirSim was to provide researchers with a safe, cost-effective, and flexible environment to develop and test algorithms that rely heavily on computer vision, control, and reinforcement learning, without the risks and costs of real-world deployment. Initially, AirSim was created for testing unmanned aerial vehicles (UAVs); however, the introduction of an automotive mode has expanded its scope to ground vehicles, addressing the growing need for autonomous driving research platforms that emphasise cognitive-based AI models rather than industry-standard certification processes. AirSim (Car Mode) is designed based on Unreal Engine, enabling it to deliver high-quality, realistic rendering, lighting, shadows, and weather effects, making it suitable for robust testing of vision-based deep learning models. It also supports automotive sensors such as RGB cameras, depth cameras, LiDAR (basic),

[3] Source: Microsoft November 13, 2017

IMU, GPS, and wheel encoders, configurable via JSON files. In contrast to industrial simulators, AirSim (Car Mode) is designed for research and offers an open-source license that encourages experimentation, modification, and academic collaboration. This allows researchers to quickly test new cognitive and control algorithms without complex system integration. Furthermore, AirSim is a single platform supporting both airborne and ground vehicles, fostering reusable AI research. Despite its many advantages, AirSim still faces some drawbacks when used in projects. These drawbacks include limited traffic complexity and scenarios due to a lack of realistic urban traffic behavior, and less accurate physical models for high-speed or advanced control tasks. Furthermore, the less realistic LiDAR and radar models significantly impact research requiring high precision in sensitive fields such as autonomous vehicles or unmanned aerial vehicles (UAVs).

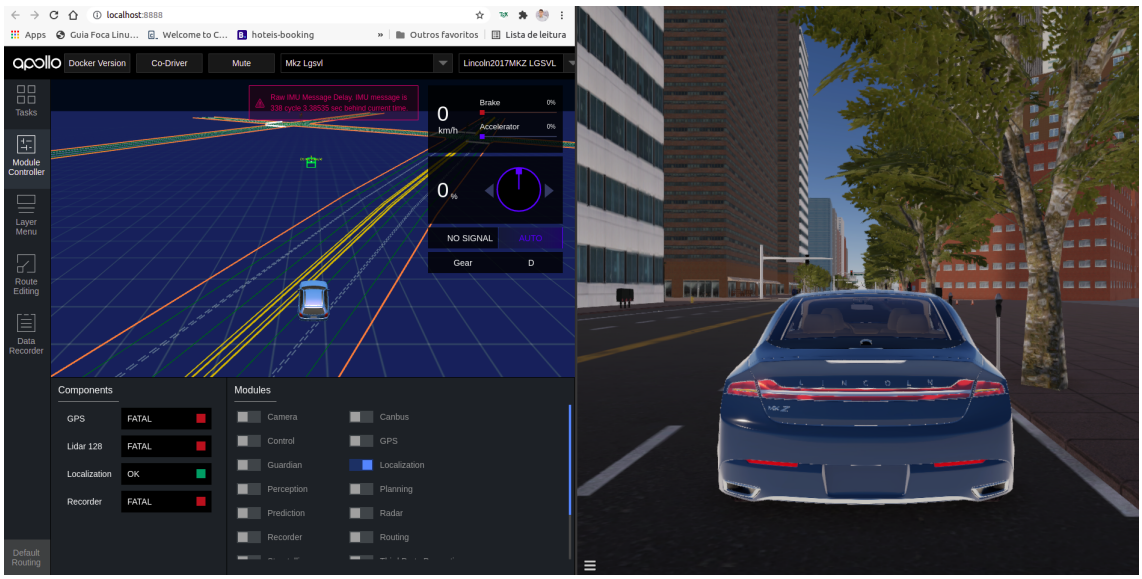


Figure 2.4: LGSVL Simulator.

The SVL simulator system was initially developed by LG Electronics through autonomous driving research ideas (originally called LGSVL Simulator) to address the essential needs in autonomous vehicle development: closed-loop simulation, high accuracy, and tight integration with the entire autonomous driving software, this simulator is illustrated in Figure 2.4. As autonomous driving systems evolved, validating the cognitive, positioning, planning, and control modules together became increasingly difficult using only discrete simulators. LG created the SVL Simulator to solve this problem, allowing end-to-end system-level testing of autonomous driving software operating in a realistic virtual environment before deployment on real vehicles. Initially, LG retained the copyright and did not share this simulator system; however, to accelerate exploitation and collaboration, LG later released the simulator system as open source, subsequently renaming it the SVL Simulator. The idea of the SVL system is built based on the core Unity Engine, which was chosen for

its balance between visual realism, physics performance, and real-time simulation stability. The platform design of the SVL simulator reflects realism through accurate sensor simulation, including configurable camera, LiDAR, radar, GPS, IMU, and CAN bus interfaces, matching real-world vehicle setups. This allows the SVL simulator to model vehicle kinematics and dynamics more accurately than other research simulators. Despite its advantageous design philosophy, the SVL Simulator still suffers from several drawbacks, such as limited large-scale traffic modeling and higher setup complexity due to the need for careful configuration of ROS/ROS2, maps, sensors, and autonomous stacks, making it difficult to access. Therefore, it is not ideal for rapidly prototyping cognitive models, reinforcement learning, or simple student experiments.



Figure 2.5: CARLA Simulator [3].

The simulation systems listed above are all good systems, meeting various aspects when applied to research and production. However, these systems still have many limitations when applied to a single field, namely autonomous vehicles, such as not sharing source code, inaccurate simulation of vehicle sensors like LiDAR, RADAR, etc., or requiring too much hardware and other settings to run the simulation. Among the numerous simulation systems available today, one stands out that meets most of the requirements for autonomous vehicle research: CARLA. CARLA (Car Learning to Act) was created by researchers at the Barcelona Supercomputing Center in collaboration with Intel Labs to address the challenges in autonomous driving research: the lack of open, realistic, and scalable simulation environments for training and evaluating autonomous driving systems. CARLA was introduced to the

research community in 2017, as illustrated in Figure 2.5, at a time when many simulation systems were either commercially available and closed-source (limiting transparency and reproducibility), or too simplistic for modern cognitive deep learning methods. Therefore, CARLA was designed to provide an open-source, research-first alternative capable of supporting end-to-end autonomous driving, from raw sensor input to vehicle control. Built on Unreal Engine, CARLA can provide realistic urban environments, lighting, weather conditions, and material properties necessary for vision-based AI. Researchers designed CARLA based on a client-server architecture that separates simulation execution from control logic, allowing external Python or C++ clients to interact with the simulator—an architecture well-suited for cloud and distributed setups. Furthermore, it not only supports realistic vehicle dynamics and a rich sensor suite including RGB cameras, LiDAR, radar, depth cameras, IMU, GNSS, and semantic segmentation sensors, but also enables control of traffic behavior, pedestrians, and programmed scenarios for systematic testing. In the field of autonomous vehicle simulation, CARLA plays a central academic and research role; it is more flexible and open than industrial simulation systems like NVIDIA DRIVE Sim, offers more diverse traffic data and scenarios than AirSim (Car Mode), and is more accessible than SVL Simulator. Despite its limitations in vehicle dynamics accuracy and initial accessibility for beginners, its advantages outweigh these drawbacks, making CARLA particularly suitable for cloud-based deep learning integration, directly aligning with modern autonomous driving research processes.

2.2 Integrate Machine Learning Algorithms into the Simulator

2.2.1 Object Detection Algorithms

Object detection is a component of computer vision that involves identifying and locating objects in an image or video frame. Although the idea is quite similar to image classification in that it labels images after analysis using a model, object detection provides bounding boxes around detected objects, enabling real-time recognition in automated systems.

In the early days of machine learning, systems often applied traditional approaches of hand-crafted features (e.g., Haar cascade, HOG features, SIFT) combined with classical machine learning algorithms (e.g., SVM). Figure 2.6 shows the classification of classical traditional object detection and DCNN-based object detection. Amr Suleiman et al. [5] proposed a method that combines both HOG feature and Linear SVM classification algorithms to detect objects in images, their idea is shown in Figures 2.7 and 2.8. This algorithm will place the image into an 8x8 pixel grid, corresponding to a cell, where each pixel will have its gradient calculated accordingly [5]. The histogram will describe the gradient direction with 9 bins for each cell and

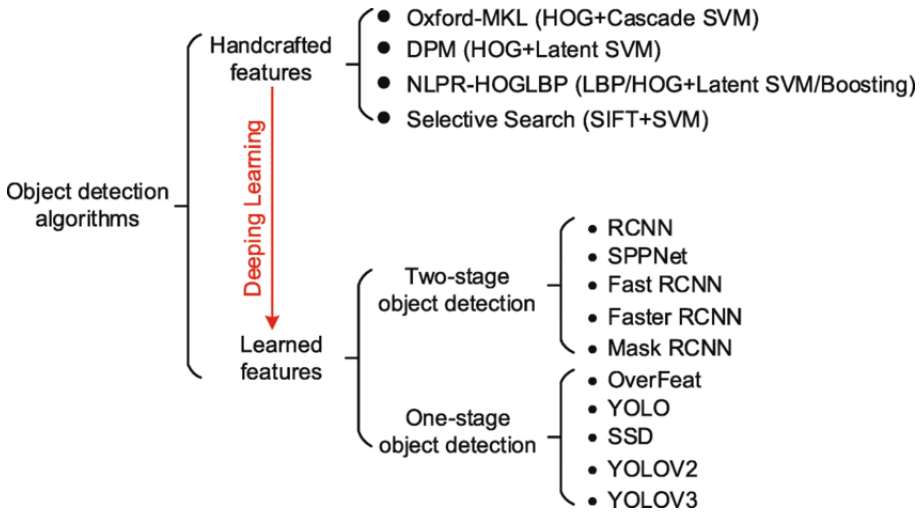


Figure 2.6: Classical traditional and DCNNs object detection [4].

is normalized to increase resistance to noise through neighboring cells; this process is performed on 2x2 cell blocks, and the final feature will have 36 dimensions [5]. The pre-trained SVM classifier performs the final task of multi-scale detection based on a classifier sized to the detection window [5]. For each image pyramid generated, it will consist of zoomed-out sections containing frames [5]. Finally, all levels of the aforementioned image pyramid will be processed by the SVM classifier [5]. Although the method is computationally efficient, it is not adaptable to real-world variations and is not suitable for highly dynamic environments such as transportation.



Figure 2.7: Images from database with different pedestrian [5].

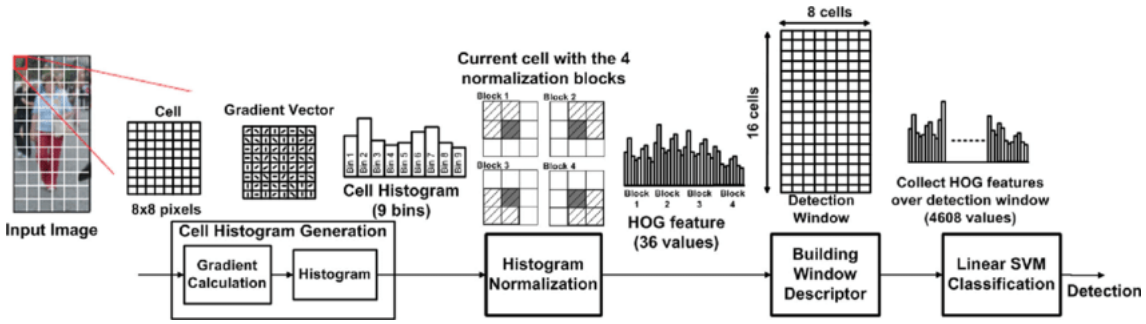


Figure 2.8: Object detection algorithm using HOG features [5].

Another method within the group of hand-crafted feature transformations is SIFT. In his research, D. G. Lowe presented a method using invariant and unique features from the image to be recognized, thereby enabling reliable matching between different viewpoints of an object [6]. This method is named Scale Invariant Feature Transform (SIFT) because the coordinates are scale-invariant relative to the local features transformed from the image data [6]. This method will generate features that completely cover the image; with this method, a 500x500 pixel image will have approximately 2000 features, depending on the image content or the choice of the parameter [6]. These features contribute significantly to object recognition and detection, with unclear objects in complex contexts, a large number of features need to be matched precisely [6]. To match and recognize objects, SIFT features are first extracted from training images and stored [6]. The image to be recognized is then matched individually with the previously extracted database based on the Euclidean distance of the feature vectors [6]. Figure 2.9 illustrates the SIFT feature descriptor; the image on the left depicts "image gradients", where SIFT calculates the magnitude and direction of the gradient at multiple pixels for a region around a detected feature point; the image on the right shows the "keypoint descriptor," with regions divided into a grid of cells (usually 4x4 cells). In each cell, SIFT builds a directional histogram (typically 8 direction bins) from the gradients within that cell. These histograms are then concatenated to form the SIFT descriptor (feature vector) [6]:

$$4 \times 4 \text{ cells} \times 8 \text{ direction bins} = 128\text{-dimensional feature vector.}$$

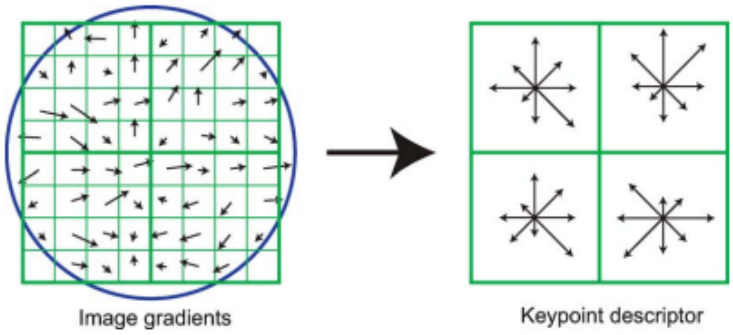


Figure 2.9: SIFT keypoint descriptor [6].

However, while SIFT (Scale-Invariant Feature Transform) is excellent for matching textured, rigid objects between images, it has some fundamental drawbacks when used for object detection (finding + locating objects in complex scenes). SIFT finds and describes feature points, and you typically detect an object by matching these feature points to a pattern/model, making it more suitable for specific object recognition tasks (e.g., this exact logo/book cover) rather than category detection (e.g., any car). Manually generated local features don't capture high-level concepts well;

a "car" can look very different between models, colors, shapes, and viewing angles—SIFT cannot generalize the whole. Additionally, because SIFT uses features to identify objects, objects with low texture or repetitive textures (car bodies, traffic signs with large uniform areas) are often difficult to accurately identify. While SIFT is quite robust to moderate lighting changes, it can fail to handle glare, shadows, reflections, and strong exposure shifts, concluding that it is very sensitive to changes in lighting. With these limitations, SIFT is not yet suitable for real-time requirements and complex, constantly changing environments like in the traffic field; it is only suitable for static image classification tasks or features for object or face classification.

Deep learning has made a huge breakthrough for object detection in computer vision. With the advent of deep learning, Convolutional Neural Network (CNN)-based detectors such as Faster R-CNN, SSD (Single Shot Multi-Box Detector) and YOLO (You Only Look Once) family have revolutionized object detection in general and in the automotive field in particular, enabling highly accurate and efficient real-time models [7]. R. Girshick and his colleagues developed an object recognition process using CNNs that combine region recommendations, and named it R-CNN (Regions with CNN features) [8]. The object recognition process begins by using category-independent recommendation methods on images (Selective Search) to find approximately 2000 candidate regions for each image. Because the CNN network requires fixed-size input, image normalization is a mandatory step, where each recommendation is placed within a bounding box and has added contextual padding (p = 16 pixels), then resized to 227×227 RGB [8]. The CNN network uses a 4096-dimensional feature vector to extract feature vectors; for each recommended region, the CNN generates a fixed-length feature vector [8]. A separate linear SVM algorithm for each class evaluates those feature vectors; higher scores mean that region is more likely to contain the classes being searched for. Finally, after multiple boxes with high scores for the same class (often overlapping), the Non-Maximum Suppression (NMS) algorithm is run independently for each class to sort the boxes by SVM score (in descending order), retaining the box with the highest score, and eliminating any remaining boxes with an IoU (> 0.5) with the retained box exceeding the threshold. The entire process is summarized and described in Figure 2.10. The results obtained from this study are very promising, improving the mean accuracy (mAP) by more than 30% compared to the previous best result on VOC 2012—achieving an mAP of 53.3%. With the results of this study, it can be seen that deep learning algorithms have made a leap forward for object detection tasks; however, despite the significant increase in model accuracy, it still faces some problems when applied in practice. Firstly, it is too slow for real-time practical use (a core limitation). R-CNN runs Selective Search (2000 suggestions) and then transforms and forwards each suggestion through CNN before classification, making it difficult to achieve real-time object detection. Secondly, the multi-stage process (not end-to-end), with a 3-module system: region suggestion + CNN feature extraction + linear SVM specific to each class, plus NMS for each class and optional box regression, makes this

method more difficult to train, fine-tune, and deploy. Finally, regarding accuracy, it depends on the quality of the region proposal algorithm, and the proposals after transformation (227×227 with contextual padding) can distort the object shape, leading to reduced accuracy when placed in rapidly changing environments. This is a major limitation when applying this method, especially in rapidly changing environments such as traffic and vehicles, where real-time recognition and computational cost are paramount.

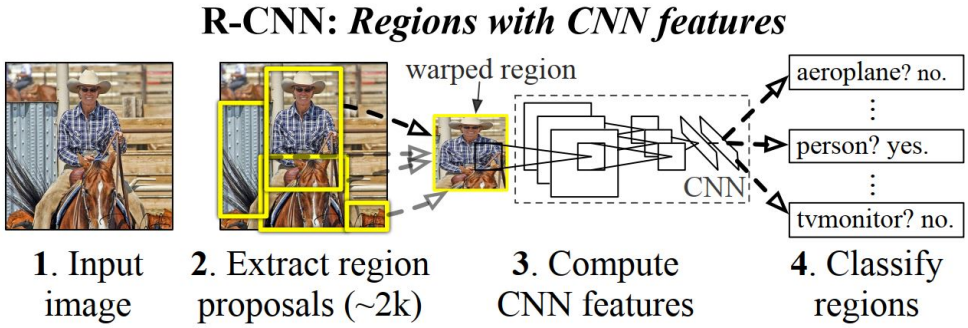
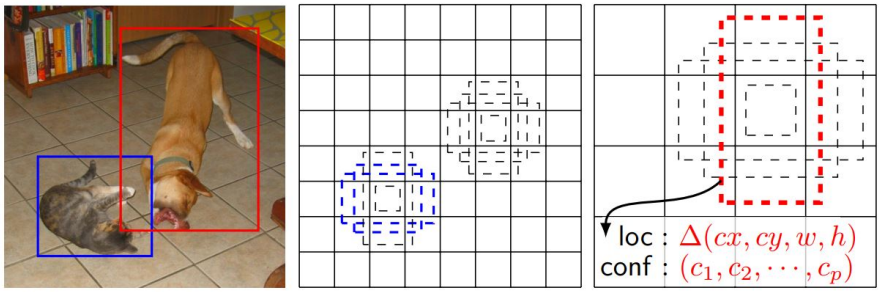


Figure 2.10: Process of R-CNN: Regions with CNN features [8].

For the two-step method, training images need to be sampled and proposed by the algorithm, as mentioned above, which makes them slow and not real-time, although their accuracy is relatively stable. To overcome this drawback, a one-step method is proposed as an improved solution with all processes encapsulated in a single step, typically the Single Shot MultiBox Detector (SSD) and YOLO. W. Liu et al. [9] introduced a new approach to object detection in images using a single deep neural network and called it the Single Shot MultiBox Detector (SSD). The main idea of the algorithm is to use predictions from multiple feature maps at different resolutions (multi-scale detection). To process different objects, when the algorithm begins its prediction, it scores the objects in each default box and adjusts its size to [9]. It then combines predictions from multiple feature maps with different resolutions to produce the prediction [9]. The image processing process begins by processing the input image to a fixed model size (typically 300×300 or 512×512). In the next stage, being a one-step method (without region suggestions), it directly outputs the class confidence score for each "default box" (anchor), then uses box bias (regression) to adjust each default box to the predicted bounding box. These predicted bounding boxes are then decoded (offsets to default boxes). SSD predicts the offset for each default box with (c_x,c_y,w,h) relative to that default box. Before making a final prediction, the algorithm performs noise filtering by applying confidence thresholds to eliminate most unsatisfactory boxes. The Non-Maximum Suppression (NMS) algorithm is applied to each class, retaining the top 200 detections per image after applying the NMS. The final result is a set of boxes with class labels, confidence scores, and refined bounding box coordinates. Whole process above is summarized and illustrated in Figure 2.11. The SSD method has shown that it is simpler than

the two-step methods, as it completely eliminates the need to propose a region and sample pixels or features within that region, making it easy to train and integrate into object detection systems with limited computational resources. Experimental results from studies on PASCAL VOC, COCO, and ILSVRC datasets have shown that SSD is indeed more accurate and faster than the two-step method [9]. With input data standardized to 300x300 and 512x512, SSD achieved mAP1 of 74.3% and mAP of 76.9%, outperforming the equivalent Faster R-CNN model [9]. However, despite its superior computational resources and speed, it still has some minor issues. First, small object detection is a weakness; as a one-step method without a suggested region, classifying small objects is relatively difficult even when increasing the input size from 300x300 to 512x512. Secondly, there's the trade-off between speed and accuracy; if you compensate for the accuracy of small objects by increasing the input size, latency will be the trade-off. Finally, multiple prediction boxes will introduce reliance on post-processing (threshold + NMS). SSD methods output thousands of boxes (e.g., 8732 for SSD300), so they rely on filtering (confidence threshold) and NMS on each layer to make the inference process efficient. This can become a bottleneck or source of missed detections in crowded scenes (because NMS can suppress nearby objects).



(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

Figure 2.11: Single Shot MultiBox Detecto (SSD) framework [9].

As introduced in the one-step method above, besides SSD, YOLO is also a prominent model of this method. In 2016, J. Redmon et al. [10] proposed a new model for object detection called You Only Look Once (YOLO). Similar to SSD, YOLO also uses a one-stage method and views object detection as a regression problem with bounding boxes in space and associated probability classes [10]. The basic idea of YOLO is to use a single neural network to process the bounding boxes and directly evaluate the probabilities on the image classes in one-stage, so the entire process is a unified whole optimized for inference performance [10]. The model also begins with preliminary processing of the input image similar to SSD; however, YOLO uses a 448×448 size for the input image [10]. Secondly, because YOLO is a "single-shot" network, one network evaluation produces all predictions. The algorithm places a grid of cells onto the image and makes predictions on those cells. Each predicted cell provides (x, y, w, h, $confidence$), where confidence is defined as:

$$\Pr(\text{Object}) \times \text{IoU}(\text{pred, truth})$$

with each grid cell also predicting the probability of the conditional class Pr(Class | Object) [10]. In the third step, the model will transform the raw data into a class-specific confidence score for each predicted box with the formula 2.1 and 2.2 [10]:

$$\begin{aligned} \text{score}(\text{class}_i,\ \text{box}) &= \Pr(\text{Class}_i \mid \text{Object}) \times\ \Pr(\text{Object}) \times \text{IoU}(\text{pred},\ \text{truth}) && (2.1) \\ &= \Pr(\text{Class}_i) \times \text{IoU}(\text{pred},\ \text{truth}) && (2.2) \end{aligned}$$

This gives the class-specific confidence score, combining the class probability and the box fit (via the IoU term). To arrive at the final result, the algorithm will perform threshold-based noise filtering by retaining only boxes with class-specific confidence scores exceeding the chosen threshold, and simultaneously apply Non-Maximum Suppression (NMS) to remove duplicate boxes. Finally, the remaining boxes after removing Non-Maximum Suppression (NMS) are the final result, including (class label, confidence, bounding box coordinates). The entire process is illustrated in Figure 2.12. The results of this method are extremely promising, achieving real-time detection on PASCAL VOC 2007 (speed + accuracy) with 63.4% mAP at 45 FPS [10]. While YOLO's accuracy cannot be directly compared to R-CNN, its detection speed, as announced by a smaller version of the network is Fast YOLO, processes up to 155 frames per second while still achieving double the mAP compared to other real-time detectors [10]. Although there are still some minor issues, such as higher positioning errors, the trade-off of slightly reduced positioning accuracy for faster inference speed is acceptable.

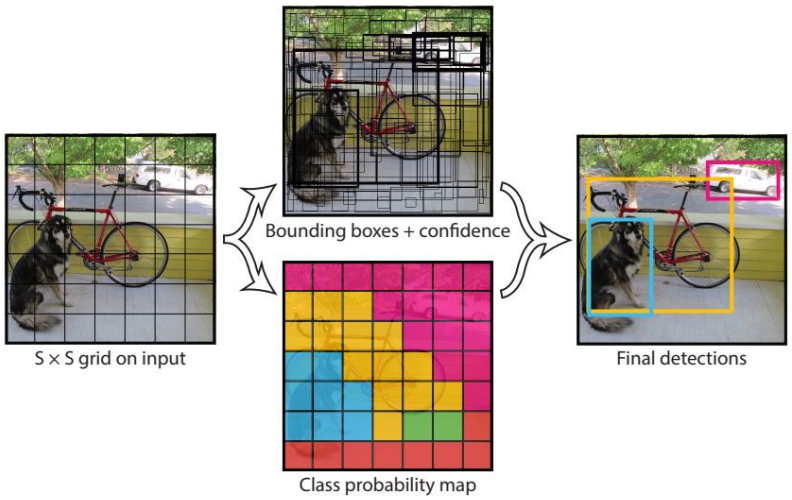


Figure 2.12: Process of YOLO methods [10].

Based on all the research mentioned above, it can be concluded that two-stage deep learning methods like R-CNN are suitable for recognition tasks requiring high accuracy where real-time performance is not a major concern. Conversely, one-stage methods, typified by SSD and YOLO, have demonstrated their versatility in achieving fast object recognition with acceptable accuracy. With nearly reaching real-time object detection, one-stage methods are highly suitable for applications in transportation, particularly in the field of autonomous vehicles with constantly changing environments. Architectures like YOLO and SSD, when applied to the practical constraints of modern transportation systems, can meet the specific low-latency requirements of automotive applications, balancing high detection accuracy with real-time inference capabilities on embedded hardware.

2.2.2 Integrate Object Detection to Simulator

Autonomous vehicles are currently a field receiving a great deal of attention from researchers, with many resources and studies focused on this concept. Sensors have long been applied to vehicles to assist drivers, and with advancements in technology, more modern sensor technologies such as LiDAR and RaDAR have changed how vehicles perceive their surroundings.

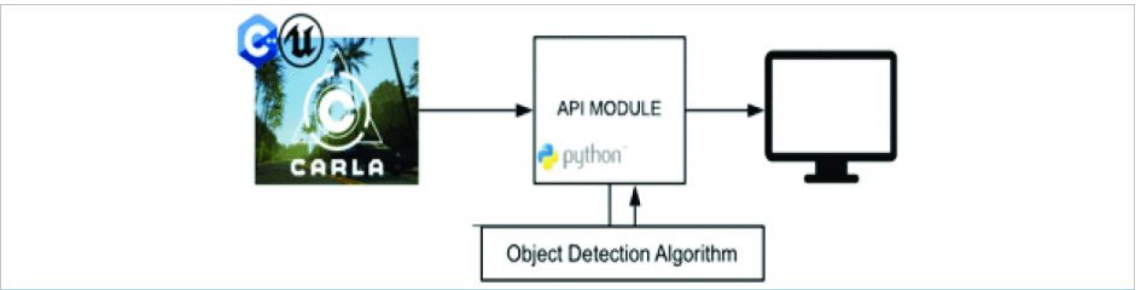


Figure 2.13: Block diagram of implementation [11].

Autonomous vehicles represent an even bigger step forward, combining RGB camera systems and machine learning algorithms with older systems to locate obstacles, thereby providing timely actions to prevent collisions or warn drivers of traffic situations, thus improving road safety and significantly reducing costs. Therefore, it can be concluded that computer vision and the application of machine learning and deep learning in autonomous vehicles are extremely important. As mentioned earlier, in order to test these algorithms and provide evaluations before applying them to real-world applications, researchers need a testing environment, and CARLA is a perfectly suitable candidate for these requirements. D.R. Niranjan et al. [11] proposed a method of integrating machine learning and deep learning algorithms into CARLA for testing via Python APIs. Figure 2.13 illustrates their system; they initially used CARLA to generate a dataset and train recognition models based on it, then used these models to perform the recognition task [11]. This group of authors used SSD MobileNet (Single Shot Multibox Detector) as the main recognition

algorithm. Figure 2.14 shows the research results of the group of authors, indicating that the SSD algorithm classified the object groups as Vehicles (cars and trucks), Bikes, Motorbikes, Traffic Lights, and Traffic signs in the CARLA simulator's test images with an average accuracy of 82.81% [11].

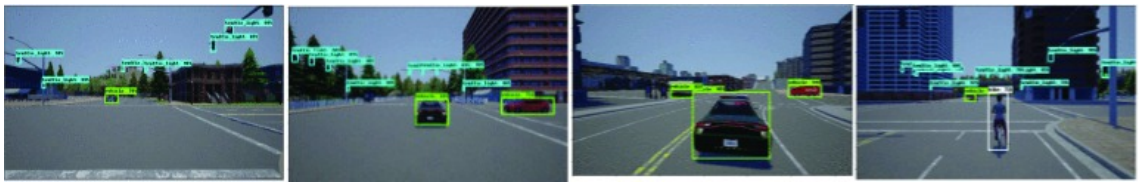


Figure 2.14: Simulation results [11].

Ó. Pérez-Gil and his research team also presented a study using Deep Reinforcement Learning (DRL) for autonomous vehicle control and tested it in a simulated environment. The idea of this research is illustrated in Figure 2.15. The research and algorithm aim to find the fastest routes while avoiding collisions and lane deviations in a crowded and complex urban environment [12]. To achieve this goal, the research team proposed a software architecture for training and validation based on DRL, using Robot Operating System (ROS) communication standards, packaging them in Docker, and using CARLA as a platform to test this new approach [12]. The performance of this model is evaluated based on indicators such as [12] "[...] Root Mean Square Error (RMSE) between the executed trajectory and the ideal trajectory, Maximum Error (ME) when driving on routes, and the time the vehicle spends completing the navigation". From the two studies above, it can be seen that CARLA offers many advantages when integrating machine learning algorithms into it. Besides CARLA, many other simulation systems are also being studied to integrate machine learning algorithms for testing.

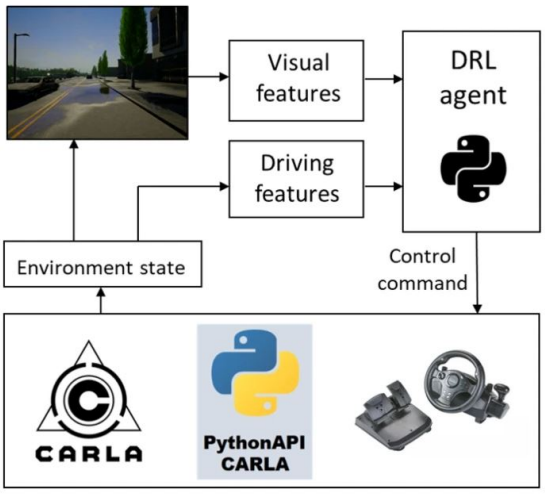


Figure 2.15: System Overview [12].

To demonstrate that an autonomous vehicle can achieve real-time control (steering, acceleration, braking), the research team led by A. Navarro and colleagues studied how to use end-to-end deep neural networks with only a forward-facing camera [13]. Their implementation method is based on two main steps: first, real-world data collection + data enhancement. Since models trained on simulators do not perform well on real racetracks (glare, shadows, etc.), they collected over 36,000 real-world images and performed data enhancement (shifting, turning, steering correction) [13]. They then modified Udacity's simulator by recreating the racetrack in Unity, then collecting driving data by driving laps themselves and recording the images and driving actions. The models integrated and tested on this system included CNN + dense layers for steering/throttle prediction, CNN + LSTM (RNN) for smoother control (but sometimes too dependent on previous states), and a modified CNN that also takes current speed and predicts a normalized position (they refer to this as a PoseNet-style idea) [13]. Although the testing was relatively successful, it also revealed some drawbacks when integrating these models into Udacity's simulator. First, the world model has low accuracy (doesn't match reality). The research team rebuilt the map in Unity using satellite textures + terrain elevation maps, but they clearly noted the low resolution (elevation maps + color) and that even with further improvements, it would never accurately simulate the real world. Second, there's the weak coverage of real-world visual variations (the gap between simulation and reality), an inherent problem with simulated image data: real-world lighting creates glare, shadows, reflections, etc. Finally, the sensor setup and simulation update rate are very limited in Udacity, which makes the simulation tests less comparable to modern self-driving car systems that rely on multi-sensor setups. Based on the above shortcomings, it can be tentatively concluded that, apart from closed-source simulation systems developed by large corporations, CARLA is the most suitable open-source simulation environment for researching and testing features and integrating new algorithms into transportation, especially autonomous vehicles. Another research led by P. Ahire also used CARLA as a simulation system to test object recognition tasks. Their system model was similar to D.R. Niranjan's [11], however, instead of using a Single Shot Multi-Box Detector (SSD), they chose YOLO as the object recognition model [14]. The research team proposed and built a CARLA architecture with a sensor layer: a simulated camera, LiDAR, RADAR, GPS, and dummy sensors such as depth and semantic segmentation. Then, based on this data, they used models for object detection, lane detection, and traffic sign/sign recognition through the YOLOv7 model. This study does not provide parameters regarding model accuracy or metrics such as mAP, accuracy/recall, so it is not possible to assess the accuracy of the YOLO application. However, this study shows that integrating deep learning models into CARLA for testing is extremely common, and they are used in research worldwide.

2.3 Offloading Computational Task to Cloud

The application of deep learning algorithms and other computational tasks to vehicles to create autonomous models is increasingly on the rise. This creates a major problem as the hardware at workstations or on-board vehicles cannot fully meet the demands of these computational tasks. These tasks include processing signals from sensors, inferring and applying deep learning models to images transmitted from cameras, perception, decision-making, and interacting with cloud platforms to create high-definition (HD) maps and store data [15]. Furthermore, the energy consumption required to process the massive amount of data from sensors and cameras, and maintain it throughout the vehicle's operation, has presented a new challenge for autonomous vehicles: offloading these computational tasks to cloud servers.

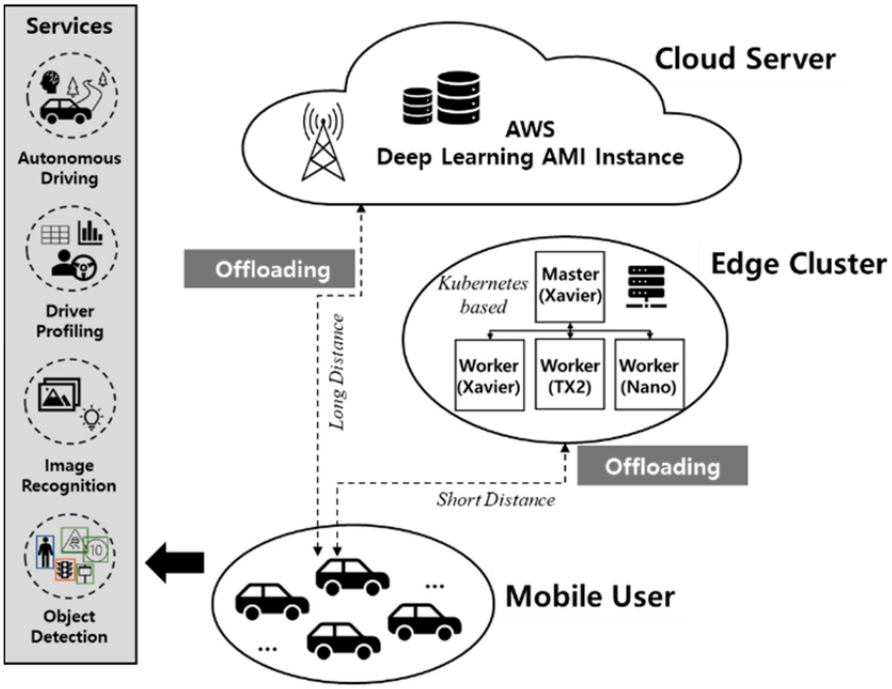


Figure 2.16: Mobile edge computing for autonomous driving environment [16].

J. Jang and colleagues proposed [16] "[...] a new task allocation algorithm model based on Lyapunov optimization to maintain system stability and minimize task processing latency," this system is described in Figure 2.16. The goal of the research is to flexibly decide whether each deep learning task should run on the edge server cluster (nearby Jetson/Kubernetes) or on the cloud (AWS) to keep the system stable (reduce queue load) and minimize latency. This research group created a system that uses cloud computing resources and analyzes them, from which it selects where tasks will be executed. The workload delivered mainly consists of predefined inference services (deep learning services requested as tasks). They send inference input to the selected server and receive inference results (using WebSocket). The

cloud's role is that of a remote server with abundant resources, but also a remote location, so transmission latency/network cost is a concern; that's why they used edge computing as the primary solution and the cloud as part of a trade-off between edge computing and the cloud. The results show that this system stably handles computational tasks in average times of 0.4231 seconds, 0.7095 seconds, and 0.9017 seconds for object detection, driver profiling, and image recognition, respectively [16].

S. Liu's research group conducted a survey on the design of an edge computing ecosystem system with the computing, redundancy, and security capabilities for large volumes of heterogeneous sensor data streams in real time under strict power and mobility constraints, while also collaborating with cloud + V2X infrastructure [15]. Their proposed general model is described in Figure 2.17. They built a structure starting from identifying data volume and latency constraints as the two main issues to address. Their argument was that autonomous driving generates massive sensor data streams and requires very low latency for decision-making, making entirely cloud-based processing unfeasible [15]. To address this, a role-division reference architecture with the following components was proposed: on-vehicle edge computing, edge servers, and the cloud [15].

- *On-vehicle edge computing*: real-time cognitive/control tasks requiring immediate response [15].
- *Edge servers (roadside/near-vehicle computing)*: support offloading and collaborative cognitive/decision-making [15].
- *Cloud computing*: heavy offline tasks such as large-scale training, HD map building, simulation, and archiving [15].

In addition, security and safety issues are also addressed as they significantly impact safety when migrating tasks to the cloud. Security is carefully considered in this research because in an edge computing-assisted autonomous driving system, any successful attack or incident could quickly lead to a physical accident [15]. Autonomous vehicle decisions directly control steering/braking, so attacks could manipulate what the vehicle perceives or decides, potentially resulting in unsafe driving behavior. For V2X communication, the transition to cloud computing means more access points than a standalone vehicle: wireless links, roadside devices, shared servers and services [15]. Therefore, an attack here (e.g., DoS/DDoS, spoofing) will cause damage not only to a single vehicle but also to the entire information sharing system [15]. This leads to the general conclusion that, in addition to researching optimal methods for moving computing to the cloud or large servers, security and safety are also crucial aspects that cannot be overlooked.

Another research by F. Hawlader [17] solves the issue of migrating computations from local hardware to edge or cloud servers. The problem posed in this study is

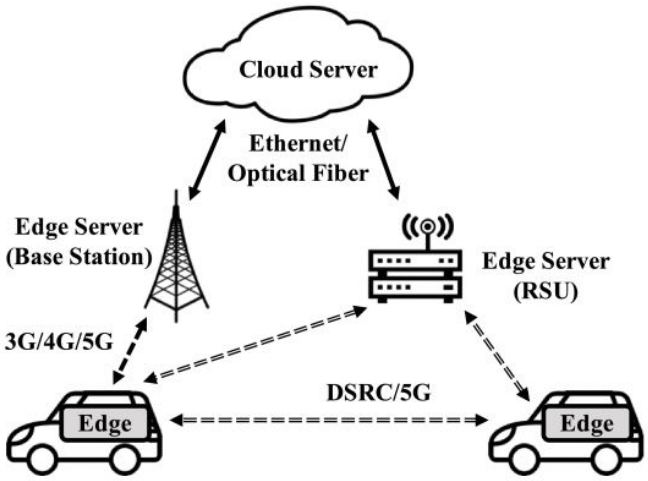


Figure 2.17: Autonomous driving edge computing architecture [15].

how to meet the requirement of real-time object detection (20 Hz) while maintaining high detection quality, in the context of vehicles with limited computing power and energy [17]. From this question, the research team proposed an idea to migrate object detection from the vehicle to the edge device (MEC) or cloud, but with a controlled balance between detection quality (mAP/AP) and end-to-end latency (compression + network + decompression + inference). They implemented a test procedure with three different deployment strategies for comparison: local inference (on-vehicle), transport to the edge device (MEC), and transport to the cloud [17]. For the local inference method (on-vehicle), the process involves running a small detector on embedded hardware; the advantage is that there is no transmission delay, so the end-to-end latency will be almost equal to the model inference time [17]. For the transport to the edge device (MEC) method, data frames are first compressed via hardware, this data is transmitted to the edge server, where it is decoded and object detection models are applied, and finally redirected back to the vehicle [17]. The final method tested was cloud-to-cloud transmission, which is quite similar to the MEC (Medium-Engineered Engineer) transmission process, however it uses higher computing power and larger/higher-resolution models [17]. Camera frames are sent to a remote cloud server, the cloud runs a detector (usually larger), and then only sends back the detection results (boxes/layers/scores) [17]. Although the process is similar to edge-to-edge transmission, the cloud is more remote, so network latency is often the main issue with this method. Despite being affected by transmission latency, the benefits of this method are clear: better detection accuracy for small objects, which is often improved with higher resolution and more powerful models, especially useful when using the YOLOv5 model, which has the disadvantage of difficulty accurately identifying objects in complex environments [17]. This research has developed a method to improve object recognition in autonomous vehicles and an idea to reduce the computational load on local computers. This involves offloading the recognition task to a cloud server and transmitting only the processed results.

This is a worthwhile idea because object detection tasks in real-world conditions consume a lot of resources and are difficult to handle with local hardware on the vehicle. Furthermore, it's also an idea for testing new algorithms with simulation systems, where cloud servers handle running simulation environments and processing computational tasks, then transmitting only the resulting frames to the user.

A new research recently published, also based on the idea of moving computational tasks—in this case, image semantic segmentation—to edge devices and the cloud, comes from a research group led by S. Zhou [18]. Their research is driven by the argument that current autonomous vehicle research, while incorporating complex deep learning algorithms, still faces limitations due to hardware inadequacies [18]. The new approach is to move these tasks to edge devices or the cloud, but this is hampered by significant latency in communication between the vehicle and the cloud (especially during movement), meaning results from the cloud may arrive too late for safe use in frame-by-frame processing [18]. On the other hand, using only edge devices may be insufficient due to physical limitations and the computational demands of the task requiring substantial hardware power and energy [18]. When on-premise computing capabilities are limited, self-driving cars often have to run lightweight segmented models to meet real-time demands, but at the cost of accuracy [18]. Through practical experiments, S. Zhou and his team proposed a solution using a fifth-generation (5G) mobile network with low data transmission latency to establish a cloud computing model for offloading computational tasks to the cloud, which they call ECSeg [18]. The model and idea of the ECSeg method are illustrated in Figure 2.18. This method was compared with four basic methods: Edge-only, Knowledge distillation lightweight model (KD), Offline threshold switching, and MPC-based switching (with prediction models). The results of this study, as published, showed that ECSeg achieved the highest latency reduction in mIoU among all methods, reporting an improvement of up to 48.8% compared to KD on Cityscapes [18]. When compared to MPC, the decision time was approximately 10 ms for ECSeg compared to approximately 400 ms for MPC [18]. For Edge-only, in the next case (boundary detection), ECSeg showed the lowest Hausdorff distance and was reported to be more than 60.5% shorter than Edge-only [18]. However, despite the significant latency improvement compared to older methods, ECSeg still faces a difficult-to-overcome limitation. The cloud's dependence on distance causes performance to drop sharply when the cloud server is far away (increased latency), as shown in the author's published study. With two cities tested, one nearby and one approximately 1000km away, the average latency ranged from about 110 ms to 205 ms [18]. Nevertheless, this is still a significant step forward and a tested idea, sufficient to demonstrate the potential for migrating computational tasks to the cloud and its effectiveness for autonomous vehicles.

From the research mentioned above, it can be concluded that the method of transferring computational tasks to more powerful servers for execution and return of results is feasible. Furthermore, S. Zhou's research demonstrated that it is possi-

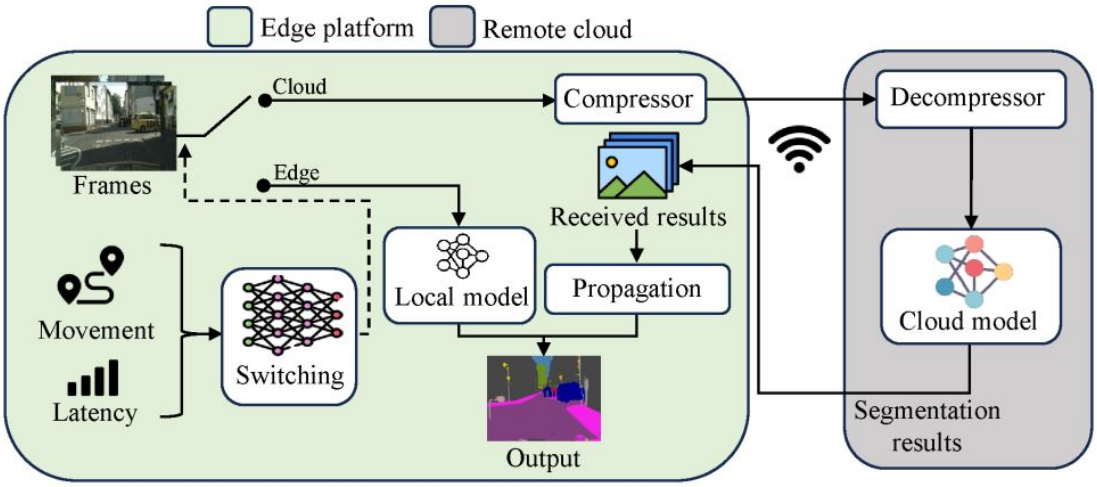


Figure 2.18: Design overview of ECSeg [18].

ble to transmit processed results from the cloud to workstations via the Internet or similar connection protocols with low latency. This leads to the conclusion that designing a system to transfer entire computational tasks from and execute them, launch and test them within a fully simulated environment on the server, and then transmit the results back to the workstations is feasible.

3 Methodology

3.1 System Architecture

The entire system is built on the Client-Server concept, comprising two main elements: the GPU Server and the Local PC Workstation. The GPU Server is built from the actual GPU Server of the Department of Computer Engineering, and the Local PC Workstations are the actual computers in the research lab used for testing purposes. The system design begins with establishing a connection from the Local PC Workstation to the GPU Server via a standard protocol. To control the connection, these connections need to be authenticated before being approved by the server. An overview of the system design is illustrated in Figure 3.1.

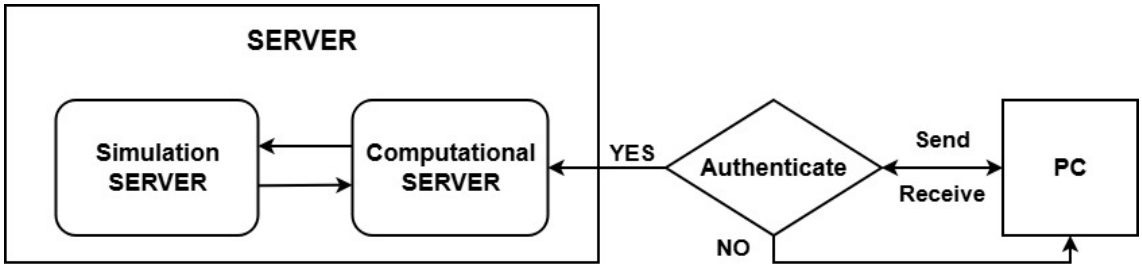


Figure 3.1: Design of System.

The GPU Server is built on a UNIX/LINUX operating system with a command-line interface for control. The system will build a server cluster consisting of two components working together internally: a Simulation Server and a Computational Server, which will perform distinct roles in operating the server cluster. The Simulation Server is the server that runs the CARLA simulation. It is built on a pre-existing Docker-based server model to launch the CARLA simulation with a fixed IP address and port. This setup uses a web interface to control the Simulation Server, turning on or off different clusters of Docker containers. Docker turns individual containers on or off; it can launch an image with multiple containers on different ports on the same IP address. This creates countless different containers with fixed ports, enabling multiple clients to connect and experiment. This operation can be visualized in Figure 3.2, which shows how the simulation server is set up within the GPU Server. This is also the old setup for a server system using CARLA, with the connection made directly from the GPU server to the client.

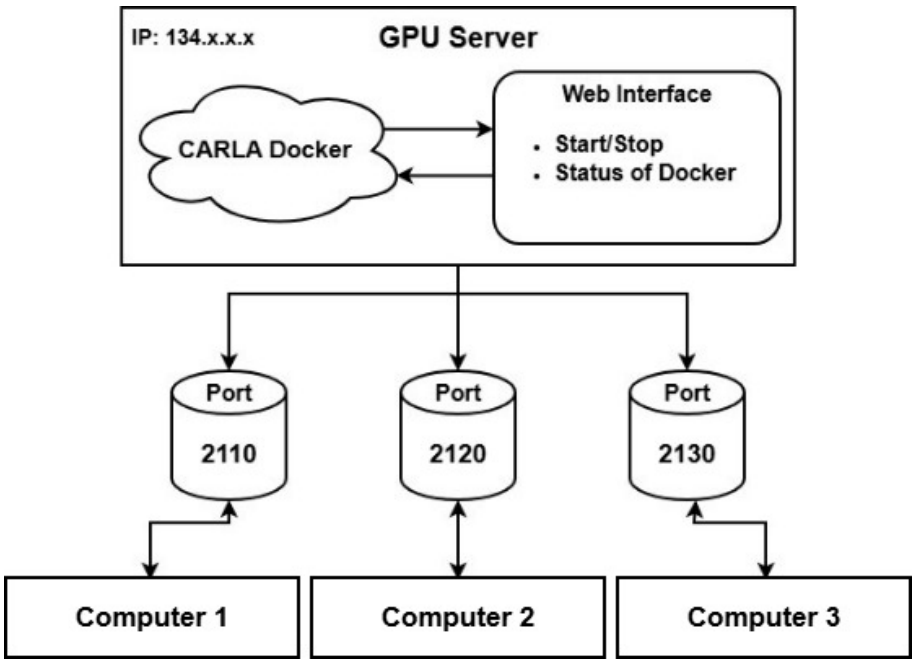


Figure 3.2: Docker CARLA Architecture.

Docker CARLA can be launched in both public and internal IP modes, which is why building a Computational Server to connect to them is the idea. The Computational Server is also a component of the system's servers; it establishes connections to the Simulation Server via an internal IP address and various ports, and acts as an intermediary to establish connections to the Local PC Workstation and transmit signals and control commands to the Simulation Server. The main idea behind creating the Computational Server is that it will act as an intermediary point, positioned between the existing system as mentioned above, processing signals from both sides and performing computational tasks. Placing such an intermediary point leverages the advantages of a high-performance server for processing computational tasks. The conceptual model for the Computational Server is described and illustrated in Figure 3.3, with the GPU Server cluster now added to the Simulation Server as an intermediary point.

In the old model, the Local PC Workstation would connect directly to the Simulation Server to send low-level vehicle control commands (steering, acceleration, braking) and meta commands to reset the simulation, change environmental attributes (e.g., weather/lighting), adjust traffic density, and modify sensors. It would then directly receive sensor data from the simulation server, decode it, and convert it into visual signals or feedback data for the algorithm. This is the basic client-server model proposed by Alexey Dosovitskiy [3] and his colleagues when publishing CARLA. This model isolates the two parts—the server for simulation and the client—through Python APIs that control objects in the world, allowing the client to restart without

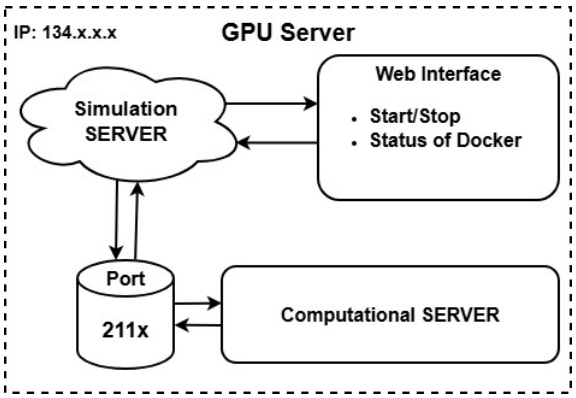


Figure 3.3: GPU Server with Computational Server added.

necessarily shutting down the simulator, and vice versa. This was a groundbreaking idea for a simulation system, as it separated the "simulation" from the "autonomy code" (planning, control, and deep learning inference) and ran code as an external program, while the server focused on simulating and rendering the world, making experiments cleaner and easier to maintain. Additionally, a stable network API enabled multiple clients (Python/C++ and possibly other languages) to control the same simulator without embedding everything in Unreal. This older model is illustrated in Figure 3.4, with the CARLA simulator launched with an internal IP address and a fixed port, communicating via socket connections. However, as demand for large-scale experiments and larger models increased, workstations with limited hardware configurations and power were unable to run simulations and deep learning inference simultaneously. The proposed solution to this problem was to run CARLA simulations using Docker and deploy them to more powerful servers. Initially, this solution proved effective in partially addressing the hardware issue, as it reduced the load on the local PC workstation. However, the remaining machine learning models, when inferred on local PC workstations, still encountered slower inference due to the increased number of parameters. Adding a Computational Server, as mentioned above, suggested a more complete solution: moving all computation to the Computational Server, leaving the local PC workstation to display the experimental results.

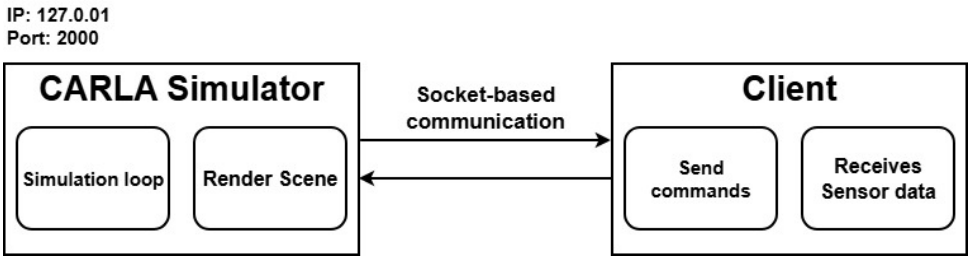


Figure 3.4: The basic architecture of CARLA.

With this system model, as described in Figure 3.5, all heavy computational tasks and simulations are handled entirely on the GPU server. The focus of the entire research will be on building a Client-Server model with a GPU Server, which will include building a Computational Server, connecting it to a Simulation Server, and a Client that sends basic control commands to the server, then receives data and displays it on the Local PC Workstation.

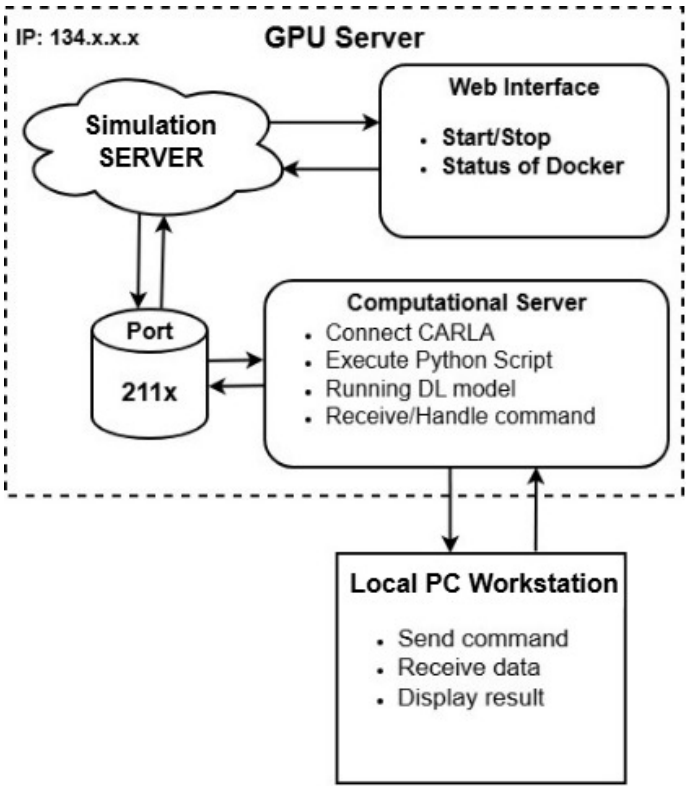


Figure 3.5: Detailed model of the entire system.

3.2 System Workflow

The entire system's workflow will consist of three main components, containing the processing logic that forms the foundation of the entire system, as illustrated in Figure 3.6:

- Administrator validates.
- Processing at the Local PC Workstations.
- Processing on the Cloud server (including the simulation server and the computational server).

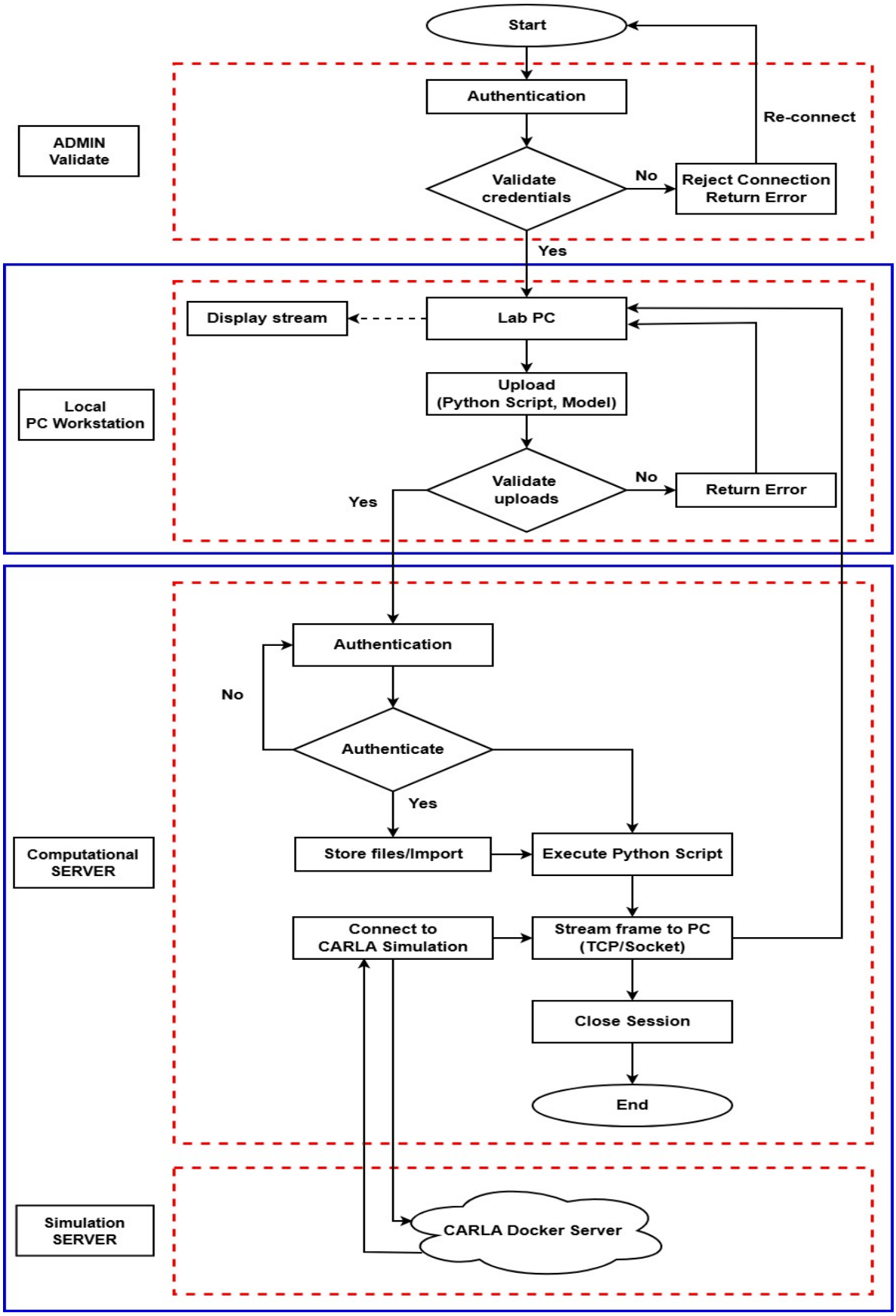


Figure 3.6: Workflow of system.

In every system today, when accessing a server, users must go through an important step of authentication to control traffic and verify validity. This verification is essential for this research as it involves direct access to a server containing real data, so every connection must verify identity and obtain permission from the administrator. However, this will be a separate research area to develop a robust, reliable control mechanism, so it will not be discussed in detail in this study but will serve only as a starting point for the system.

After the authentication mechanism determines that the entity connecting to the server is legitimate, the client machines will have access to the server. Figure 3.7 illustrates the client machine cluster in detail. When a client machine is granted access, it can upload Python files or machine learning model files. However, as mentioned above, since this server contains a lot of data, the security of the entire server is a top priority, so it must undergo basic authentication. These initial authentication steps will be performed on the client machine to determine what the client is uploading. The system will determine the file size, file type, and classification, which will also support the archiving process when files are uploaded to the server. In addition, once access to the client machine has been granted, it can receive streaming data from the server and display it. The data stream consists of

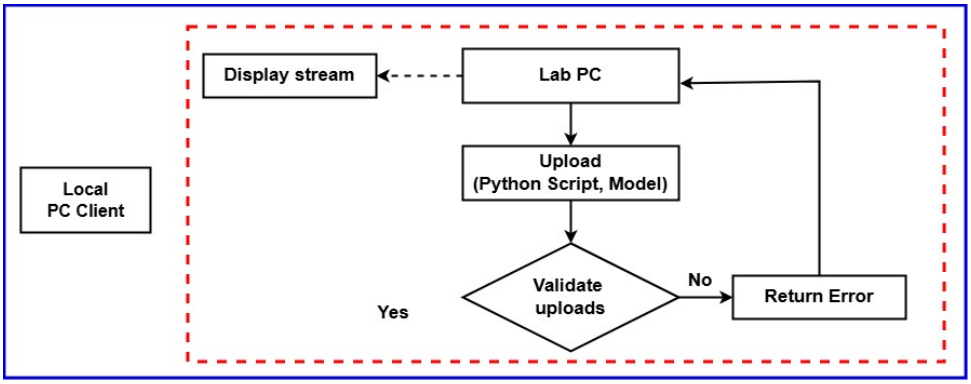


Figure 3.7: Detail of Local PC Workstation.

Python files or machine learning model files that, once deemed valid, are sent to the server. There, they undergo further verification; however, as mentioned above, this is a separate mechanism that requires filtering and identifying potential risks to the server and will be discussed in a separate study. The system assumes that all files have passed verification and are deemed safe. They are then stored on the server in two separate directories: one for the Python file and one for the machine learning model file. These directories are managed by a larger directory identified by its IP address, port, and connection time to the server. Each directory is unique and inaccessible to a single client session. This design allows for future expansion, enabling multiple connections from various users. When the server receives control commands from the client, for example, to execute a Python file, the system checks

the requesting session and executes the uploaded files as requested by the control command. This module also connects to another module that handles communication with the simulation server; both modules then send data to the streaming module, which transmits it back to the client. Finally, upon receiving a disconnect command, the workflow terminates and deletes the entire session directory to ensure space and "cleanliness" for the server. The entire process involves two servers, the simulation and the computing server, belonging to the cloud server cluster as described in Figure 3.8.

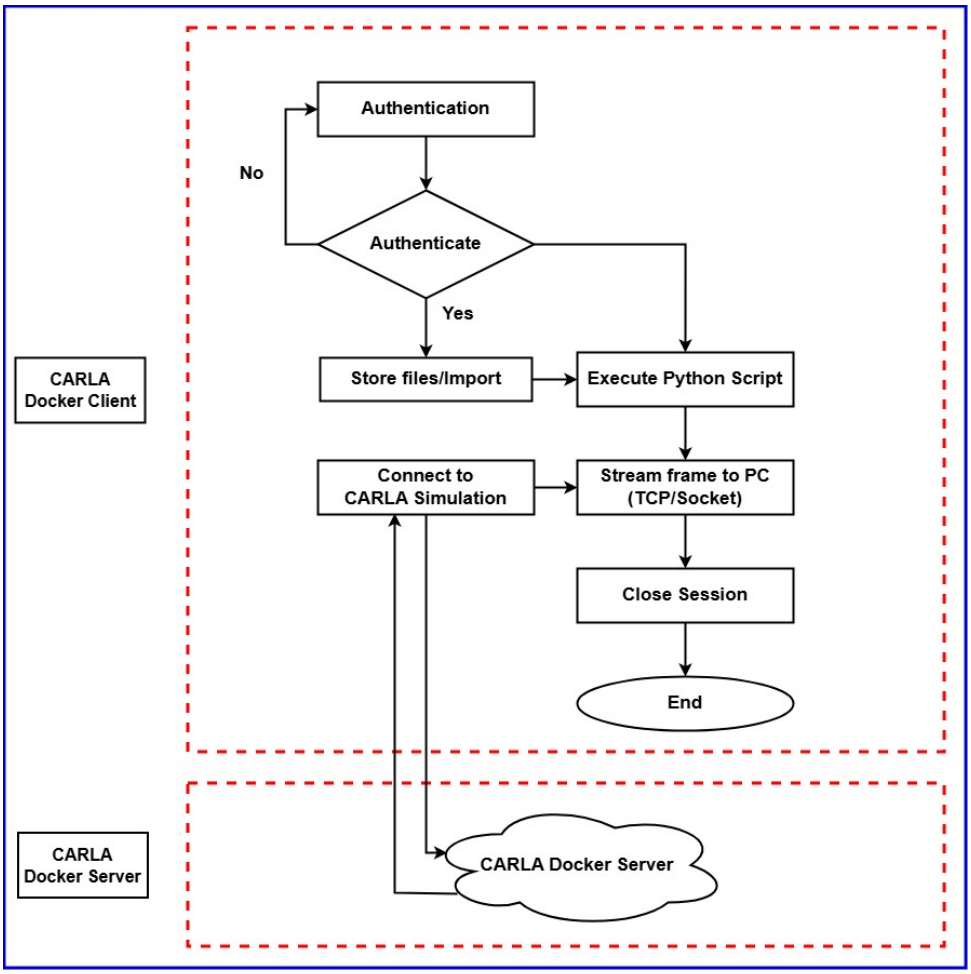


Figure 3.8: Details of Cloud Server cluster.

3.3 Client–Server Connectivity

The client-server model was first mentioned in the 1960s and 1970s, initially consisting only of terminal devices on mainframe computers. This model only truly gained attention with the rapid development of personal computers and LANs, where

personal computers became true "clients" (performing some local computations) and servers were capable of providing shared services such as files, printers, and databases. The modern client-server architecture is defined as a distributed computing model. This architecture includes a client (a user-facing device/application) that initiates data or service requests, and a server (central system) that processes those requests, manages resources (data, computing, security), and returns responses over the network. Its characteristics are based on three main principles: request/response communication, centralized server control, and multiple connections. This architecture offers numerous advantages in deployment, firstly the concentration of resources and the ability to share them across multiple users. This is a crucial characteristic, as the server will share resources—in this case, emulating CARLA—among multiple users and execute user files in a consistent environment, without encountering dependency issues. Secondly, it offers scalability and ease of maintenance or upgrades. With this capability, a server (or a cluster of servers) can serve many clients; it can be scaled by making the server more powerful or expanding with more servers. The centralized nature of the architecture is demonstrated by the ease of maintenance or upgrades; changes only require a single update on the server, instead of updating the software on each client device. Finally, there are issues related to security and client hardware configuration. Client-server architecture effectively manages security issues because when input data is sent to the server, it needs to be centrally authenticated/authorized before service access is granted. This allows the client to maintain a lightweight configuration (user interface + communication) while the server handles the heavy processing and storage.

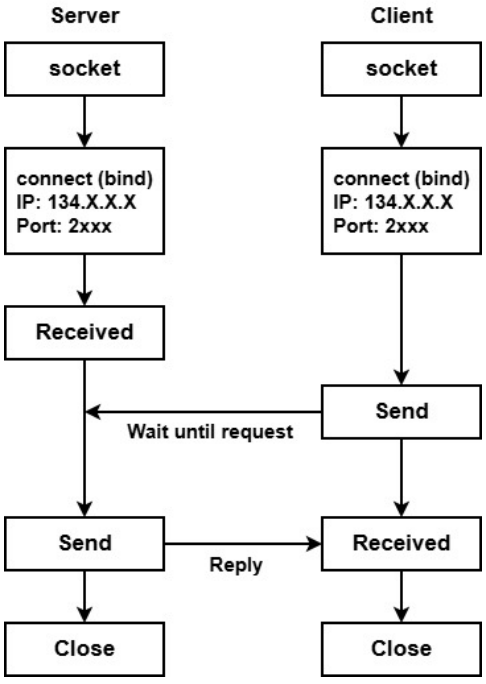


Figure 3.9: Overview of Socket.

To build a client-server architecture, the most fundamental requirement is a method for communication between these two entities. Many methods exist for connecting these two objects, including HTTP/HTTPS (REST API), WebSockets, Message queues, or Pub-Sub (e.g., AMQP, MQTT, Kafka-style). While these methods work well, they have drawbacks, including not being truly real-time, high transmission costs, greater infrastructure complexity, limited data transmission capabilities, and difficulty with debugging. Pure socket communication, illustrated in Figure 3.9, is a solution to these problems. In network communication, a socket is defined as a "pipe" that allows the client and server to exchange data directly over the network. For a socket to function, the server needs to open (bind) a connection with an IP address and port, and then listen for incoming connections. Clients do the same thing, opening a socket connection and connecting to the server's IP address and port. Each connection to the server from a client is unique because each client will have a different IP address and port (from the client), so this is an ideal method for managing server connections. Another reason sockets are suitable and meet the research requirements is real-time streaming. Sockets are a low-level, high-performance method, making them ideal for continuously sending sensor frames/data, enabling real-time connections and sensor data streaming. Additionally, sockets are customizable and support user-defined message formats (e.g., header + payload), making them extremely flexible as the amount of data transmitted from the server can now be adjusted, and control and marking of data to be transmitted can be applied.

The server-client model was built using socket communication between the two objects; however, this model still lacks a protocol for data transmission. To transmit data between two objects, the preferred protocols are UDP and TCP. UDP (User Datagram Protocol) is a transport protocol used in client-server systems when low latency and low cost are prioritized, while data loss is acceptable. This protocol transmits data in packets (datagrams), with each packet being independent; however, it does not guarantee that packets will arrive at their destination, and their order may vary. Another significant and complex issue with the UDP protocol that makes it unsuitable for the research requirements is its lack of congestion control, which means it cannot automatically reduce speed when the network is congested. Furthermore, due to the characteristic mechanism of UDP not resending lost packets, leading to packets transmitted between two entities can be lost when traffic to the server increases, thereby reducing to a loss of system reliability. Finally, the UDP protocol lacks authentication and security mechanisms, allowing any packet and data to be sent from any client to the server without an authentication or IP address verification step before they are received. This limitation makes the system weaker and more vulnerable to attacks such as injecting malicious code into packets, direct attacks from compromised clients, and DDoS attacks.

The system requires reliable transmission of packets and data, while also ensuring the security and confidentiality of the server. Given these stringent requirements,

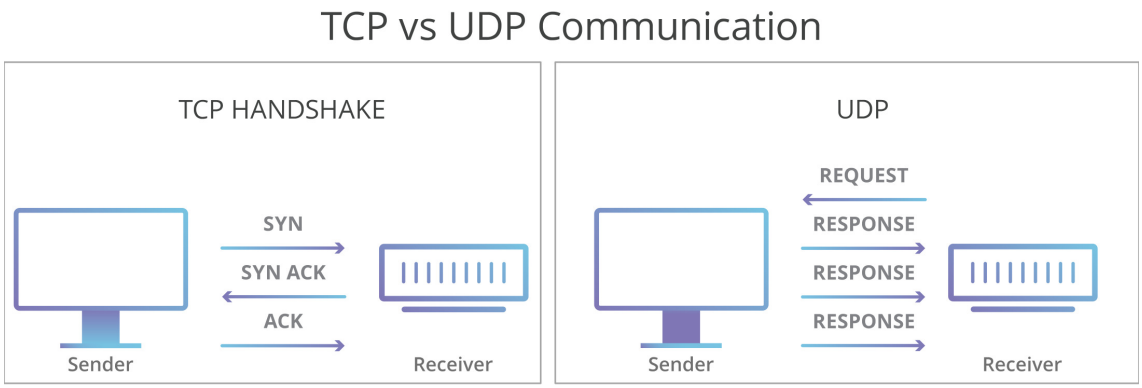


Figure 3.10: Describe the connection process and compare the two protocols.

the TCP protocol is a perfect choice in fully meeting the system's technical requirements. TCP (Transmission Control Protocol) is the most common transport protocol for client-server systems when data transmission requires reliability and ordering. The TCP connection mechanism uses a three-step handshake to establish a connection between the client and the server. Figure 3.10 illustrates the connection process and comparison between the two protocols TCP and UDP. In the first step, the client sends a "SYN" message to the server. Simply put, this message indicates the client's intention to connect to the server, and it also sends an initial sequence number (ISN), called x, which allows TCP to track bytes, detect loss, reorder, and retransmit in case of failure. After the server receives the client's "SYN" message and ISN(x), the server sends back two things: a "SYN" indicating that the server also wants to connect with the client, which also includes its own ISN (let's call it "y"), and an "ACK" containing the server's response that it has received the client's "SYN". Finally, the client sends back an "ACK" message to the server confirming receipt of the server's "SYN" packet. Now that both sides agree on each other's starting sequence numbers (x and y), and the connection has been established, both sides can send and receive bytes until one side closes the connection. The 3-step TCP handshake process (SYN → SYN-ACK → ACK) exists to make the connection reliable and secure before any actual data is exchanged. The main benefits of this process include confirming bidirectional access and synchronizing sequence numbers when both sides agree on the starting sequence number, which is fundamental to TCP reliability (ordering, packet-loss detection, retransmission). Additionally, using this process also prevents duplicate old packets from corrupting new sessions with new sequence numbers, helping TCP distinguish between new connections and delayed packets from previous connections. With this mechanism, the TCP protocol effectively controls the data flow and resists congestion, ensuring that if a packet is lost, it will retransmit the data to the application in the correct order. The only drawbacks of the TCP protocol are the cost of sending packets and its latency compared to the UDP protocol. Retransmitting lost packets and congestion control can cause streaming delays ("freezing"), and since TCP is a byte stream,

framing must be implemented during data transmission. A data frame transmitted using the TCP protocol includes a length prefix (4 bytes and followed by data), a separator (newline/JSON line, e.g., "\n"), and finally, a fixed-size message, making this method significantly increases the cost of data transmission between the server and the client. While these disadvantages are significant compared to the UDP protocol, the system established in this research prioritizes security and data integrity, making it a preferred protocol.

Based on the characteristics and benefits analyzed above, the combination of socket communication to connect two computers and the TCP protocol to transmit data between the server and client as the foundation for the entire system is entirely feasible. It meets the system's requirements by prioritizing data integrity and security for the server system, while still ensuring that data is transmitted in near real-time.

3.4 Integrate Deep-learning Models

CARLA is built on the idea of using Python APIs to connect to a server, as illustrated in Figure 3.11 by Alexey Dosovitskiy [3], serving the purpose of controlling the CARLA world and its objects using built-in functions, libraries, or basic commands. This is a significant advantage when using CARLA simulation as a foundation for building a system and integrating machine learning and deep learning models, because the Python programming language supports many computer vision libraries such as Tensorflow, PyTorch, and Ultralytics (YOLO), making it extremely easy to integrate seamlessly during programming.

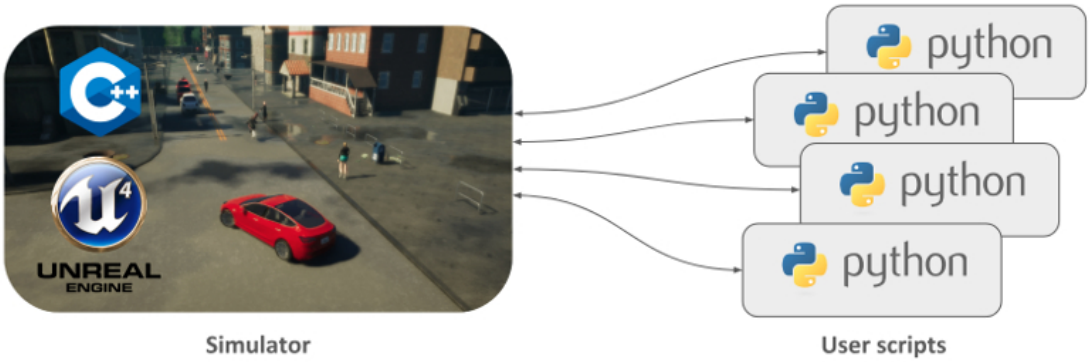


Figure 3.11: Python client API connects [3].

Integrating machine learning and deep learning models into CARLA begins with processing image frames. The nature of machine learning models when processing video or detecting objects in real time is to process each image (frame) and apply the model to each image independently. The images processed by the model are continuously output, giving the impression of directly incorporating the model into the

videos in real time. With this idea, embedding these models into CARLA also begins with processing data generated by CARLA, which is image data in the BGRA (raw) color space format. The data used by machine learning and deep learning models is typically RGB; although the RGB camera sensor inside the CARLA simulator also uses RGB, when rendering images, CARLA uses an image buffer displayed as BGRA with primary color channels: Blue (B), Green (G), Red (R), and Alpha (A). With three primary BGR channels, which are the most important components for image rendering, the remaining Alpha channel serves as a constant (typically 255) for each pixel (completely opaque). It is primarily used for compatibility, buffering, and alignment (4 bytes per pixel) and facilitates easy interaction with graphics APIs and image libraries. CARLA utilizes this color space primarily because it is a very common packaged pixel format in rendering workflows (Unreal/GPU textures) and convenient for alignment. When images are rendered as BGRA by CARLA, the system places them into buffers, which are then stored and retrieved by machine learning algorithms through the extraction of each image (frame) in the array.

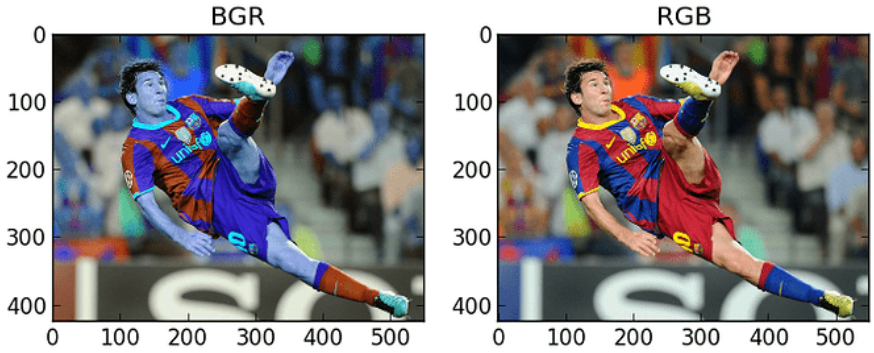


Figure 3.12: Compare BGR and RGB color space.

After the image is rendered and the system receives signals from the buffers, object recognition will be performed. First, these buffers will be extracted element by element (frame by frame). These images are rendered in BGRA format as mentioned above, so immediately after the extraction step, the system will apply image transformations to the images. Each BGRA color channel consists of 4 bytes; however, depending on the machine learning model used, there will be an option to remove the Alpha channel or not. In this study, the YOLO model is the primary model for recognition, as YOLO only requires three main channels: RGB, so the Alpha channel will be completely removed. Another issue to note is that the order of the color channels when removing the Alpha (A) channel remains BGR; therefore, for the YOLO model to be able to read them, the order must be reversed to RGB. The problem with the BGRA color space, which needs to be converted to RGB, is not only applicable to machine learning models, but also relates to the issue of displaying image tonal values when they are rendered. Figure 3.12 shows that, with the BGRA color channel, the tonal values are completely inaccurate compared to the standard

RGB color channel. In the second step, once the frames have been normalized, the YOLO model is applied to each frame consecutively. The network makes predictions for multiple candidate boxes in the image. Conceptually, it predicts for each candidate box including the coordinates (x, y, w, h) or (x_1, y_1, x_2, y_2), the object score ("is there an object?"), and the probability of the class (car, person, etc.). This occurs at various feature map scales to detect small/medium/large objects. The objects within the CARLA simulation are almost identical to the real world, although they are only simulated images, their realism reaches over 80%. This allows the pre-built YOLO model to be directly applied to the simulation without needing to retrain the model. However, CARLA is designed based on standard traffic, so some objects will not be included in the trained object list. In the final step, after filtering out low-confidence predictions (confidence threshold) and running the Non-Max Suppression (NMS) algorithm to remove overlapping duplicate boxes, a list of the final detected objects is produced with bounding box parameters (x_1, y_1, x_2, y_2), confidence score, and class identifier/label. These parameters are continuously output on the image (frames), because the nature of videos is that they are sequential frames, as mentioned earlier, which makes them appear to be performing object detection in real time. The results of this process are shown in Figure 3.13, where the detected objects appear with bounding boxes along with their class names and confidence levels. For this study, the model integrated into the system is YOLO with version 8 (YOLOv8), and the task is object detection version "n". In practice, YOLOv8n is the perfect choice for object detection tasks on autonomous vehicles because of its small size (few parameters), lightweight nature, and fast object detection, which are suitable for the limited resources on these vehicles. Given this practical requirement, this will be the fixed model for the system; however, it will still be able to use models from the same series, retrained with new classes from the user. In the future, this research will aim to enable the use of multiple models for testing in CARLA simulations.

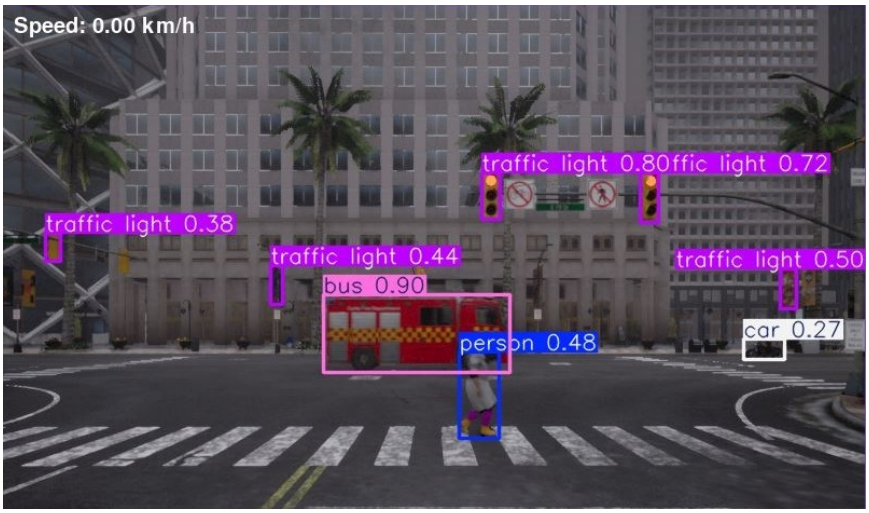


Figure 3.13: Visualize object detection in CARLA.

3.5 Stream Data

After the integration of machine learning and deep learning models into the simulation is complete, as described in section 3.4, these results need to be displayed to assess the accuracy and reliability of the algorithm. The traditional method for displaying these results is to render them directly on the client machines via the Python Pygame library. In this method, after the frames are applied to the machine learning model and the results are generated, they are stored in buffers and remain in a waiting state. These buffers are then read sequentially by the Pygame library, and the frames within this memory are displayed in the RGB color space. The advantage of this traditional method is that there is no latency, as they are transmitted almost directly from the CARLA simulation server, without any intermediate points. However, while this method eliminates server latency, it introduces issues with frames per second (FPS). This problem stems from using machine learning models to perform inference directly on the client side and display the results, whether in the purest way—running the simulation on the client—or by migrating the simulation server to the cloud. These bottlenecks arise from insufficient hardware power to simultaneously run the simulation, machine learning inference, and display, or to simultaneously run inference and display while receiving frames from the simulation server transmitted from the cloud. When all tasks—starting the simulation server, running the machine learning models (even the smallest ones), and displaying the results—are done simultaneously on the same client side, it pushes the hardware to its limits due to the enormous computational workload, resulting in a sharp drop in frames per second (sometimes as low as 3 to 4 FPS). Migrating the simulation server to the cloud to reduce the load might improve the situation somewhat, but the problem still arises from the machine learning models as they perform inference. When using larger models (e.g., YOLOv8x) for a more comprehensive and accurate overview, the frame rate will still be limited to 10 frames per second. This frame rate bottleneck can disrupt large-scale experiments due to delays; while these delays do not affect the results, they prolong the experiment time, causing unnecessary losses.

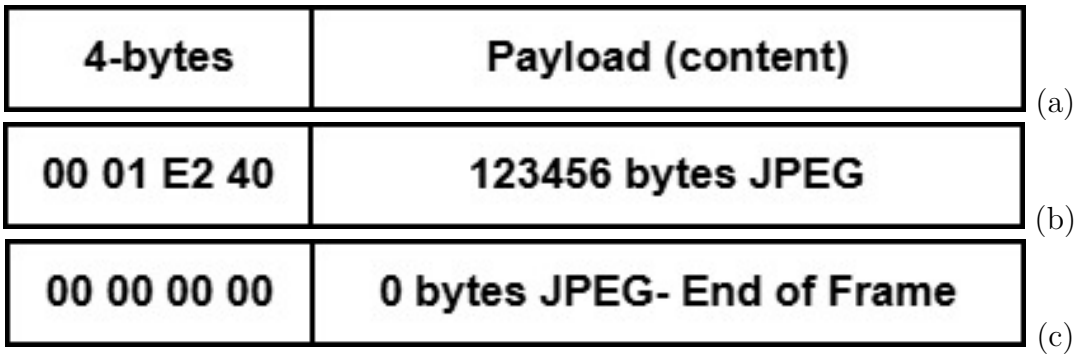


Figure 3.14: Describe a packet (frame) using the 4-byte prefix method.

This research aims to address this issue thoroughly. In addition to moving the simulation server to the cloud, it creates an intermediate point as described in section 3.1. At this intermediate point (named the computational server), it processes the frames received from the simulation server, handles the Python files and models sent from the client, and then applies machine learning models to it. This is a key difference from traditional methods; instead of displaying the frames on the server, these frames will be transmitted back and displayed on the client, completely eliminating any computational tasks on the client. To realize this idea, the research will use the TCP protocol as a transport layer to transmit data between the two objects. To transmit files (including Python files and models), the system first reads the files to be sent and calculates their size on the client. After the reading and computation process is complete, a send command is generated, transmitting the entire packet as binary code to the server according to a pre-established structure. At the server, when the packet is received and decoded, it creates a file with the same name it received in the packet, then overwrites the content to match the size it received during decoding. This process is complete when the client receives a response from the server and repeats for other files. Based on the idea of transmitting files to the server, the method for streaming frames from the server to the client also uses the TCP protocol. However, unlike using binary variables and filenames and pre-calculating the file size, streaming requires some technical changes for transmission. After the model is applied, data frames are continuously pushed into a buffer, which consists of arrays, and their size changes continuously until the simulation stops, making it impossible to pre-calculate the size. Additionally, streaming is continuous, so it is not known in advance how many frames the server will send (it could run indefinitely as mentioned above), plus the size of each JPEG changes from frame to frame (compression variations), so the client must know exactly how many bytes belong to the next frame. The proposed method to address this problem is to use a 4-byte prefix to mark the beginning of each packet. These bytes are big-edians (network order), meaning the most significant byte is at the beginning of the byte sequence (from left to right). Each packet sent from the server will have these big-edians calculated based on the size of the content being sent. Without this 4-byte prefix method, streaming would face a significant problem: leakage of data bytes within frames, causing them to freeze or, worse, crash. When frames are transmitted over TCP, the receiver only sees a continuous stream of bytes, and TCP does not preserve "message boundaries" (frames), which leads to the question: What could go wrong without lengths or separators? Suppose the sender sends a JPEG frame A (50,000 bytes), then continues to send a JPEG frame B (48,000 bytes). On the receiver side, decoding functions might return 12,000 bytes (part of A), then 30,000 bytes (the rest of A), and finally 20,000 bytes (the end of A + the beginning of B mixed). This is a significant problem if there is no way to determine the endpoint of A; the receiver cannot reliably split the data stream back into frames. A fixed 4-byte prefix is very small and fast to parse, it works perfectly for repeating this cycle, and avoids problems where JPEG bytes might accidentally contain characters like "\n". Figure 3.14 (a) illustrates a packet (frame) sent by adding a 4-byte prefix to the beginning

when transmitting JPEG bytes. A specific example illustrating this cycle is shown in Figure 3.14 (b), a JPEG with a size of 123456 bytes, when converted to 32-bit binary code, becomes the string "00000000 00000001 11100010 01000000". This is then converted to hexadecimal as "0x00 0x01 0xE2 0x40", so the 4-byte prefix will be "00 01 E2 40", which indicates the size of the JPEG content that follows it, as illustrated in Figure 3.14 (c).

On the client side, the decoding process of these JPEG files begins by searching for a system-defined flag (e.g., START_STREAM). This operation signals to the client that this is a streaming stream with JPEG data and only stops upon receiving an end of frame (EOF) signal. Upon receiving the signal flag, the client switches from receiving binary data to decoding JPEG data. At this point, the client sequentially reads and decodes the incoming signals and displays them via the Pygame library. These frames are continuously decoded and combined, creating the illusion of being streamed directly from the server (videos are essentially a series of consecutive frames). The transmission ends when either of two conditions occurs: the server experiences a problem that causes the stream to crash, or the client receives a signal to stop the transmission. When the client sends a signal to stop the stream, the server will send a final 4-byte message "00 00 00 00" (as mentioned above) to interrupt the stream and also send an additional flag (e.g., STOP_STREAM) to trigger the client to switch back to receiving binary data.

Through data transmission methods via the TCP protocol and image streaming from the server to the client, the foundation for the system model described in section 3.1 has been established, while also addressing the hardware resource issue as all tasks can now be performed entirely on the cloud server. Combining all the methods presented in sections 3.3, 3.4, and 3.5 minimizes the computational load on the client, using the client only as a place to display results and control server operations.

4 Implementation

4.1 Tools and Techniques

Throughout this research, Python was chosen as the foundational programming language for building the entire system, including modules that contact to the simulation server, implement the client-server architecture, and integrate deep learning algorithms into the simulation environment. Python is not a new programming language; it even predates the very popular Java, which has many applications today, when it was first released in 1991, but received little attention until the explosion of the computer vision field, which propelled it to become the most popular language (according to TIOBE [1]). Python's popularity stems from its simple, user-friendly syntax and few special syntax constructs, making it particularly suitable for beginners and for prototyping or automation tasks. However, do not confuse the syntax's simplicity with its power; it is extremely powerful, with a vast library that supports a wide range of modern programming tasks, including AI/ML, data analysis, scripting, DevOps, and testing. Furthermore, it also supports multiple platforms, including Windows, Linux, or a UNIX-based platform like macOS. Python's compilation and execution times are slower than compiled languages (C/C++/Java/C#) for CPU-intensive tasks, and conflicts regarding packaging/dependencies (venv/conda, native libraries, due to differences between operating systems) can occur. Another issue with Python when used for multi-threaded or multi-core tasks is the GIL, which limits the ability to parallelize multi-threaded CPUs (although multi-threading, native extensions, and asynchronous I/O can help). Despite its technical limitations, it still has advantages over Java or C/C++ due to its faster code writing, simpler syntax, excellence for data/automation, and a vast library ecosystem. As mentioned in section 3.4, CARLA uses Python APIs entirely for simulation control, and most popular machine learning models today are supported by libraries that run on Python. Furthermore, Python libraries provide functions for establishing Socket and TCP connections, making Python a perfect choice for building systems.

As analyzed in the methodology, there are many different ways for a server-client architecture to establish a connection and communication. However, according to the system requirements and the purpose of the research, the entire system runs on an internal LAN (belonging to the Computer Engineering department, Faculty of Computer Science, TU Chemnitz), so the priority will be to establish a connection that is fast, simple, secure, and allows for near real-time communication. For this

[1] Source: TIOBE Index, Feb 2026 (tiobe.com)

reason, socket communication based on the Python language was chosen as an optimal method for communication. The specificity of this research is the development of a platform that can directly transmit signals and data from the cloud server to clients, so the data type during transmission is not fixed. Socket communication enables determining the data format (binary or JSON), message semantics, and flow (request/response or broadcast). Furthermore, they work well for custom protocols, real-time updates, IoT device linking, telemetry streams, and can handle multiple connections efficiently. Given these characteristics and benefits, adopting socket communication for the system is a perfectly suitable approach. In socket communication, TCP and UDP are two popular protocols for transporting data streams. UDP is more suitable for streaming only data streams and does not require reliability on both sides. Conversely, TCP is a safer choice because the system needs to send packets containing sensor signals and model accuracy during streaming, and these need to be sent in order. Combining socket communication with the TCP protocol is a cornerstone for establishing client-server communication and data transmission in the research.

This research, in addition to building connections in a client-server architecture, aims to integrate machine learning and deep learning models into simulations for testing. The You Only Look Once (YOLO) algorithm will be the main algorithm embedded in the simulation for testing, using version 8 (YOLOv8). To be able to use and embed the model into the simulation environment via Python, the model needs to have the relevant dependencies and libraries installed. These libraries include Numpy, OpenCV2, and PyTorch, which are entirely packaged within the larger Ultralytics library. The OpenCV2 library is primarily used around the model and acts as a preprocessor for input data. Its basic tasks typically involve reading data streams, loading and saving images, performing basic image manipulations used in pipelines (resizing, converting BGR to RGB colors, drawing), and finally, visualization through drawing frames/masks and displaying frames (cv2.rectangle, cv2.imshow). The Numpy library, on the other hand, is primarily used for representing and manipulating image data, especially in pipelines built around OpenCV. The input image, as an array after being output by OpenCV2, is passed into NumPy, where common operations such as resizing, adding borders/black frames, normalization, data type conversion (uint8 to float32), and color processing are naturally performed using NumPy arrays within CV processes. NumPy can be seen as an intermediary point in Ultralytics—connecting OpenCV frames, preprocessing, and outputting model outputs into standard arrays for further use. Ultimately, the brain of the entire Ultralytics library is PyTorch, where YOLO truly operates. The PyTorch library defines the model architecture (layers, forward pass), loads model weights from model files (".pt" files), runs inferences, and handles the training process with loss functions, backpropagation, optimizers, and derivatives.

For CARLA simulations to work, they need the CARLA libraries from Python. The CARLA library (often imported with "import carla") is the client-side API

used to connect to a running CARLA server (a CARLA/Unreal simulation) and control the simulation and read sensor data. In practice, this library performs the following operations: connecting to the CARLA server, creating a client, setting a timeout, and accessing the simulated "world". Besides connecting to the server, it also loads maps, configures weather, time of day, traffic settings, and resets scenarios, while simultaneously reading the world state (actors, objects, current settings). The blueprint libraries in CARLA are also managed by the CARLA library in Python, through tasks such as creating vehicles, pedestrians, and other agents, and controlling vehicles using the throttle/steering wheel/brakes (manual or automatic control). A key function of this library for autonomous vehicle research is the configuration of simulation timing and sensor configuration. This function uses synchronous mode (deterministic step) versus asynchronous mode, then runs the simulation frame by frame for repeatable experiments (important for ML evaluation). The library also initialises sensors such as RGB cameras, LiDAR, Radar, IMU, and GNSS, and sets sensor parameters (resolution, FOV, range, speed, and binding transformation). After configuring the sensors in the object, it processes the sensor data stream and measurements, and receives sensor callback functions to retrieve frames/dot clouds/measurements as Python objects. A library always included with Carla is PyGame, which is primarily used on the client side to build a simple real-time window and control layer. The primary goal of PyGame is to create a window for displaying and visualizing the CARLA simulation by rendering RGB camera frames (or a composite HUD) in a desktop window. Additionally, this library can receive keyboard/controller input signals to capture user input (WASD, arrow keys, steering wheel) and convert it into CARLA vehicle controls (accelerator/brake/steering) when in manual control mode. PyGame manages event loops to retrieve frames, update the screen, and smoothly process input events, while displaying FPS, speed, number, collision warnings, sensor on/off, etc.

The initial system development relied entirely on a command-line interface, with two main components: the server and the client. While the command-line interface clearly and intuitively demonstrated how these two entities interacted and was easy for the server to execute, it had a weakness: it was difficult for those without command-line server control experience to access and was not user-friendly. This led to the need to build a UI/UX that would allow users to interact easily and accelerate the testing process, as the system was now controlled via buttons rather than command lines. Qt5 was chosen to build this UI/UX. The main task of the Qt5 library is to allocate fixed-size areas within a window, then place function buttons in those areas. Additionally, it provides a space to display sensor data transmitted from the server. Qt5 was chosen not only because it is highly compatible with Python, but also because it supports multi-threading, keeping the user interface responsive while the inference/networking process runs in the background.

4.2 Socket Communication and TCP Protocol

The communication between the client and the server must be defined before establishing a connection. Communication within a connection is defined as the exchange of messages and data from end to end. As analyzed in the Methodology section, communication based on the Python Socket API is the most suitable choice for the system. To establish communication using this method, the "socket" library in Python will be used to set up a server with the basic function:

server

```
1:# Create socket connection
2:server = socket(socket.AF_INET, socket.SOCK_STREAM)
3:# Bind the IP and Port
4:socket.bind((IPv4, Port))
5:# Listen on IP and Port
6:server.listen()
7:# Accept the connect from client
8:client, addr = server.accept()
```

The first command initializes a server with two important parameters: AF_INET and SOCK_STREAM. AF_INET stands for "Address Family = Internet (IPv4)", which tells Python that the socket will use an IPv4 address, for example, 192.168.1.10, and ports like 8080. With this configuration, IPv6 addresses can also be used if a change is necessary, simply by changing the address family to AF_INET6. The second parameter, SOCK_STREAM, specifies the protocol for data transmission, indicating that the system is using TCP. The data transmission protocol can be changed flexibly by modifying this parameter; for example, the UDP protocol would be denoted as SOCK_DGRAM. With just this command, a server is configured with two options: IP address type (v4 or v6) and protocol type (TCP or UDP). The next, equally important step, is to inform the server of the range of IP addresses and ports it will listen on using the "bind" function. IPv4 addresses are typically used in different ways, the first one is a local address ("localhost" or 127.0.0.1), which instructs the server to listen only for programs on the same computer, preventing others on the network from connecting. The second method allows direct access to the server's IP address (e.g., 192.168.1.10), allowing the entire range of IPs within the same subnet (e.g., 192.168.1.x) to connect. This method limits the number of clients allowed to access the subnet because, if the subnet's IP range is 192.168.1.0/24, usable addresses have accessibility to connect to the server, limited to only 192.168.1.1 to 192.168.1.254, making it more manageable and secure. The final method allows it to listen on the entire IP range on the server via the address 0.0.0.0. This setup allows the server to accept all connections sent to any IPv4

address of the server or any other address the server might use. This setup is more commonly used in practice because it allows clients to connect using any IPv4 address assigned to the server (e.g., a LAN IP address). Additionally, it works even when the server's IP address changes (DHCP) or when adding/removing network interfaces, and the connection includes localhost + LAN without further setup. When using this method, the server is configured as a real server, with access controlled by a firewall or authentication, so this will be the primary server configuration. An important point to note is that the client will not connect to 0.0.0.0, but will actually connect to a real IP address (such as 192.168.1.50) or a hostname that resolves to that address.

client

```
# Create socket connection
client = socket(socket.AF_INET, socket.SOCK_STREAM)
# Connect with the IP and Port
client.connect((IPv4, Port))
# Send something
client.sendall(request.encode())
# Close the connection
client.close()
```

Conversely, the client's setup is almost identical to the server's. It will also establish a socket connection with two parameters: AF_INET and SOCK_STREAM. However, one difference is that the client will not "bind" to a specific IP address and port; instead, it will connect directly to the server's IP address and port. After a successful connection, the server receives a signal from the client, enabling communication. For example, the client can send a "sendall" command to send data messages to the server. Finally, when communication is complete, the connection must be closed using the "close()" command. Closing the connection is extremely important, as it serves as a "command flag" to let the system know the connection has ended and that the operating system resources (file descriptors or ports) used by the socket client should be released, thereby preventing resource leaks.

The two pseudocode sections above represent the fundamental concepts of the system when two client-server objects connect via a socket. In this study, once the server and client connect, they perform a series of tasks before disconnecting. The first basic tasks include sending and receiving control commands from the client to the server. When a command from the client needs to be sent to the server, it is initially in text form and needs to be converted into data that the TCP protocol can transmit – streams of bytes. These bytes are sent to the server via the "sendall()" function; however, the length of the bytes is not fixed, thus creating several new

problems. The first problem is that data parsing will be inaccurate (message truncation) because TCP does not preserve message boundaries. If the protocol is "one request = one message", but the system reads a long message continuously or stops when overloaded, it can result in decoding or parsing failures or incomplete data processing. Another problem is that two messages might be merged in a single reception; for example, the client sends "CMD1\n" followed by "CMD2\n", but the server returns "CMD1\nCMD2\n" in a single read. To address this, the system uses a built-in function that allows the client to send a fixed-length message, and the server reads only that same length. Additionally, when packets containing text are sent, they are marked with "\n", which signals to the server that it needs to read until the next line.

read fixed-length bytes and use delimiter on server pseudo code

```
1:   def recv_more()
2:      data = recv(fixed_length_bytes)
3:      if timeout:
4:         return None
5:
6:   def read_line()
7:      cmd = find delimiter "\n"
8:      if cmd == None:
9:         return None
10:     raw = bytes(read cmd)
11:     return decode(raw)
```

Another function of the system that also uses socket communication and the TCP protocol is uploading Python files or machine learning model files. The purpose of this is to transfer computational tasks to the cloud server and execute them on those servers. Unlike the text messages mentioned above, these files usually have a fixed size and change only when there are updates within them, so transmitting these objects to the server requires only basic socket communication and the TCP protocol. First, a command from the client is needed for the server to recognize that this is a data upload request. The structure of this command is as follows:

upload _ <type_of_file> <path_to_file>

This command, with the beginning "upload", lets the client understand that this is a file upload request. The next part is "type", which changes depending on the type of file the user needs to upload (e.g., py or yolo). The next part of the structure is the file path, which shows the system the current location of the file to be uploaded; it must be an absolute path. Once the system has all the information, it will open the chosen file in binary mode, read its contents, and calculate its size.

From this information, a complete command will be generated to upload these files to the server as follows:

UPLOAD <py | yolo> <name> <size>

With this structure, when the server receives the packet from the client, it will read sequentially from left to right. First is the UPLOAD command flag, which allows the server to understand that this is a file upload request from the client. For files categorized by the tags "py" or "yolo", place them in the designated directory. Continuing with the file name, it will initialize a file on the server with the same name as in the packet sent from the client and open this file. In the final step, with the size included in the command, the server reads the exact number of bytes, which is also the file's size, and writes it to the newly created file. In this way, uploading a file to the server involves converting the file's contents to binary data, sending the data stream to the server, then creating a file and writing the data to it. To illustrate this more clearly, the pseudo-code below will visualize the flow of ideas for uploading files.

upload file algorithm pseudo code

```
1:   def upload_file() % <---- Client side prepare and send
2:      file_name = open file with path (remove \n and \r)
3:      size = get size of this file
4:      header = UPLOAD {file type} {file_name} {size}
5:      send {header} to server
6:      while open {file}:
7:        data = read the content (read up to 65,536 bytes from the file
each loop)
8:        if not data:
9:           break
10:       send data to server
11:
12:   def handle_upload_file() % <---- Server side handle upload
13:      total = read the buffer
14:      if total < 0 or total > 2_000_000_000:  % <---- Max size is set
to 2GB
15:        return Error
16:      data = read (total)
17:      create, and open file with name sent from client
18:      write (data) to file
```

For non-text data, as analyzed in the methodology, a different mechanism is needed to transmit data integrity without causing errors or freezing due to transmission leaks or errors. This mechanism uses a 4-byte prefix to encapsulate the data (in this case, images or frames), then transmits them to the client via TCP sockets as byte streams. This 4-byte prefix mechanism consists of two fundamental parts: first, it encodes the frames in the JPEG format, along with parameters for frame quality. This encoding is done through functions in the "cv2" library, which are then converted to bytes. The second part is to pack the data using the 4-byte prefix, followed by the data, using built-in system functions. These packets are transmitted to the client in the structure described above. The client then reads the first four bytes of each packet, and based on that information, unpacks the packets and reads the data within the size indicated by the four bytes of the prefix. In programming practice, sending with a 4-byte prefix is represented by the initial string "!I", which is a format string for Python's struct module. Analyzing the "!I", the "!" character represents the network byte order (big-endian), also known as the "standard size", without platform-dependent padding. The next character "I" represents an unsigned 32-bit integer (4 bytes) ranging from 0 to 4,294,967,295. With this introductory section, the server defines how data is sent, and the client builds functions based on this principle to extract the data. A pseudocode snippet describing the behavior and idea of this mechanism is described below, covering both the server-side sending and client-side extraction.

4-bytes prefix method pseudo code

```
1:  def emit_frame() % <---- Emit the frame on server to client
2:     frame = encode frame with cv2(type, color space, quality)
3:     data = convert to bytes(frame)
4:     send the bytes with struct ("!I", data)
5:
6:  def read_pack() % <---- Read the package on the client side
7:     header = reading first 4-bytes in package
8:     if header < 4:
9:        return Error
10:    frame = unpack the package("!I", header)
11:    read until frame == 0
```

In summary, the use of socket communication and the TCP protocol for data transmission depends on the type of data being sent. For plain-text control commands sent from the client, they will be sent as byte streams, with the delimiter ("\n") separating them. The TCP protocol does not preserve message boundaries, so the system needs to use delimiters to separate control commands sent from the client to avoid confusion or errors on the server. For client file uploads, since these files

do not change in size and are sent as binary byte streams, they will be sent in the usual way, along with necessary information such as the file name and file size. The included file size helps the server know exactly how many bytes to read, preventing data leakage. Finally, there's the transmission of special data types (frames or images). These data types are essentially bytes, but their size is not fixed because the quality and object vary between frames; therefore, the 4-byte prefix technique is applied to this data type as the simplest solution. This 4-byte prefix allows the system to accurately read each frame, resulting in smoother data transmission and preventing data leakage between packets.

4.3 Manage Session and Data

After the connection establishment steps and the TCP three-way handshake are complete, the client will be connected to the server as described in the previous section. Connections would be relatively simple to manage if they were one-to-one connections, where a server would only connect to a single client and share data with only that client. However, this study focuses on multiple connections between a server and multiple clients, so building a mechanism to manage these connections is a crucial system requirement. Connection requests to the server typically vary in timing, even when they are nearly simultaneous, as connection requests are read sequentially from the buffer; therefore, timing is the first fundamental factor in managing a client connection session. However, time varies across geographical regions and between server setups. Therefore, for consistency and ease of programming and management, the system will use Unix time (also known as a Unix timestamp). The precise meaning of Unix time is the number of seconds that have elapsed since 00:00:00 UTC on January 1, 1970. Unix time is characterized by counting in seconds using UTC and does not include leap seconds. The advantage of using Unix timestamps is simplicity, as it is just an integer value, making it easy to store, compare, and transmit, which facilitates easy time checking, as well as logging and sorting. Furthermore, it does not require complex date formats (e.g., 2026-03-08 14:30:00), making it compatible with many systems. The second element added after the timestamp to make it unique for each connection session is the IP address and port of the client requesting the connection. Each connection to the server uses the client's IP address, which is unique to each connection because the system design dictates that each client can have only one connection to the server (meaning the same IP address can have only one connection). Following that is the client's port; these are also unique because each port serves a specific purpose according to the operating system's settings, and two services cannot use the same port. Combining the IP address and port, a second label for session management is formed. Along with the Unix timestamp analyzed above, a complete label for managing client connection sessions would be presented as:

Unix-timestamp _ IP _ Port

After the labels containing session information for the client connection are created, the server uses them to manage existing connections. Server tasks are managed by threads; the system has two thread concepts that need to be defined: main threads and sub-threads. The main thread is responsible for the continuous operation of the server, helping it maintain and process all signals sent from clients by creating sub-threads to handle these requests. Based on this concept, the system assigns each connection to the server as a sub-thread to process without affecting the main thread. When executing task control commands from the client, these threads continue to create sub-threads, for example, to upload files or stream sensor data. This design completely isolates a connection, separating it from the server's main operational thread, thereby minimizing the possibility of server crashes when a connection fails. This organization is illustrated in Figure 4.1 below, which depicts the layers of threads on the server as they create sub-threads to execute tasks.

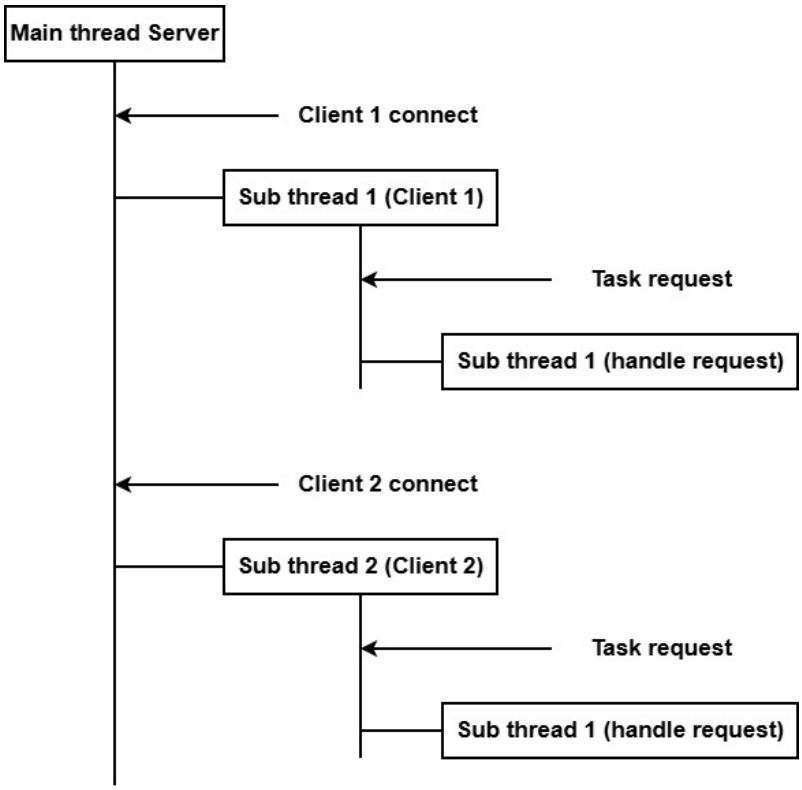


Figure 4.1: Describe the design of the main thread and sub-threads in the system.

In parallel with assigning sub-threads to client connection sessions, the system also initializes directories for each connection session. These directories are created within the server's session directory to manage and store files uploaded from the client. Each session has a separate directory named after the session labels mentioned above, making them easily accessible by simply comparing the session label requesting the task with the directory name. Each directory contains subdirectories

named “py/” and “yolo/”, corresponding to the two types of files uploaded from the client: Python files and model files. When the client attempts to upload files, the server first compares the upload session label with the existing session directories on the server. After matching is complete, based on the classification label in the upload statement (here, “py” or “yolo”, as described in section 4.2), they will be placed in the designated classified directories. The complete directory tree for a connection session is described in Figure 4.2, which illustrates the location of the session directories and their subdirectories.

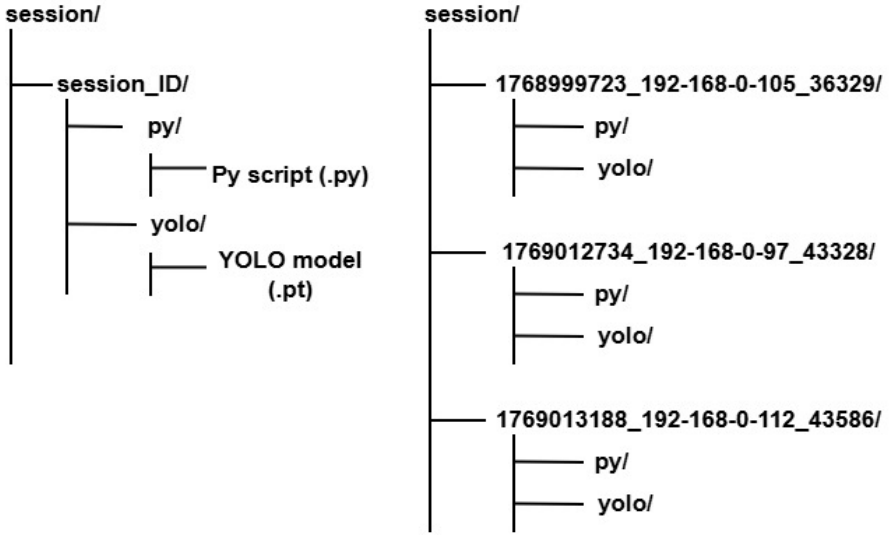


Figure 4.2: Describes the server’s session directory tree.

In summary, each connection is assigned a unique label by the server, consisting of two main components: a Unix timestamp and an IP address, along with the port. Each label is responsible for managing the connection session to the server and creating separate directories to manage the files uploaded to these sessions. Each client connection session is unique and is allocated a sub-thread from the main thread, and tasks, when requested, continue to create sub-threads from these sub-threads. The main threads are responsible for keeping the server running and distributing task requests to the correct sub-threads within the sessions. The benefit of this threaded design is that it isolates and separates the activity of each individual connection session, enabling the system to handle multi-threading and multi-tasking with multiple connections. It also helps protect against attacks and errors that could cause the server to crash.

4.4 Communication with Simulation Environment

The CARLA Simulator is typically installed in two different ways. The first is a pre-built version with labels spanning different versions. This method is lightweight

and easy to install, requiring minimal library dependencies and complex server settings. While map and world customization are limited, it's the most common way to set up a CARLA simulator. The second method involves building CARLA from scratch, giving users complete control over map customization, construction, and various constraints. However, this method consumes a lot of memory and is difficult to implement on a server due to the numerous libraries and parameters that need to be adjusted. To simplify installation and enable global customization via modified map files, GPU servers often use Docker to install the CARLA simulator environment. CARLA Docker is a packaged version of CARLA that runs inside a Docker container, allowing the emulator to run without manually installing all dependencies directly on the host operating system. CARLA Docker is used because of its easy installation instead of manually setting everything up, and it's good for both the host and the remote machine (the client in this study). CARLA documentation shows the use of Docker both in display and offline mode, which is especially useful for GPU servers.

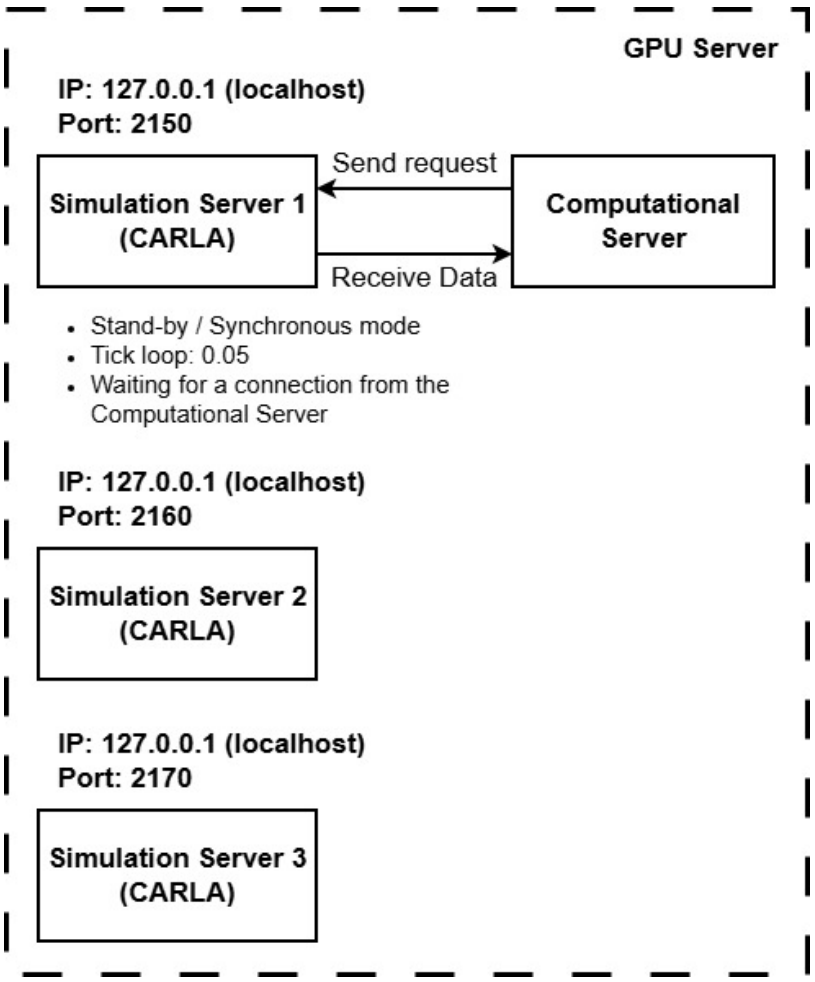


Figure 4.3: Describe the communication between the simulation server and the computational server.

The simulation server and the computational server are two separate entities set up on the same cloud server. These two entities communicate directly with each other through basic settings as outlined in the CARLA documentation. The communication between the two objects, the computational server and the simulation server, is briefly described in Figure 4.3. In normal operation, when the CARLA simulation server is started via Docker, it initializes the world using the default map specified in the configuration files used to build the Docker images. At this point, the simulation server is waiting because it has not yet received any connections from the computational server, and it will operate continuously until it receives a shutdown signal from Docker. When the computational server makes a connection request, it uses the specified IP and port to connect to the simulator server, and all of CARLA's Python APIs are activated when they are called. These connections are established using an internal IP address ("localhost" or 127.0.0.1), because in reality these two objects are communicating on the same platform (the GPU server) and all information is directly connected. The only difference is the ports used for connection; the GPU server is set up with multiple CARLA Docker branches running simultaneously. These branches operate independently on different ports (e.g., 2150/2160/2170), when computational servers established the connect, each computational server on the GPU Server can specify the port to connect to the appropriate branch. The system configuration and connection to the simulation server in this study will operate in synchronous mode of the CARLA simulation environment with the "fix_delta_second" parameter set to 0.05. To further explain the synchronous mode of CARLA simulation, synchronous mode in CARLA means the simulator does not run continuously automatically; instead, it waits for an explicit "tick" command before proceeding with the simulation of the world. In synchronous mode, the server can keep everything synchronized across the same simulation step. This means that every object on the simulation server is consistent, as it has to wait for the CARLA server to request an update at each beat (a beat corresponds to the "fix_delta_second" value). This is important because the project needs the computational server to coordinate the simulation processes as they read camera/sensor frames and perform object detection processing and finally transmit the results back to the client. The "fix_delta_second" parameter defines the amount of simulation time that elapses when the "world.tick()" function of the Python API belonging to the simulation server is called once. This means that each tick will advance the CARLA world by exactly "fixed_delta_seconds" (0.05 = 50ms) of simulation time. At the same time, this number also means: 1/0.05 = 20. This means 20 simulation steps per second, corresponding to 20Hz, which is also roughly equivalent to the scan frequency of the installed sensors. The value 0.05 was chosen because it reduces the load compared to higher speeds like 30 Hz or 60 Hz. In practice, for this study, the computational server not only connects to the simulation server but also collects frames, compresses JPEGs, sends them over TCP, and can perform object detection. A base rate of 20 Hz is often the practical balance between smoothness in terms of timing and computational/network cost.

In short, the communication between the simulation server and the computational server running on the GPU server will occur via a local connection ("localhost") on a specific port. The simulation server will be in indefinite standby mode and operate synchronously with a "fix_delta_second" parameter of 0.05 (corresponding to 20Hz). Information processing and communication will begin upon receiving a request signal from the client.

4.5 Integrate Deep-learning Models and Execute Computational Task

After Python files are uploaded from the client side, they are stored in separate directories within that client's session and are ready for execution. These Python files, once uploaded, do not execute automatically but wait for a trigger command from the client with the syntax:

create <vehicle _ name> <spawn _ point> <flag = yolo>

The main idea for executing a Python file completely independently and only transmitting data is that instead of hardcoding the computation process inside the main server, the computational server creates a dedicated session object to execute the Python file uploaded from the client. The computational server only provides the environment, along with parameters specifying the location of the deep learning model file and permissions to use resources from the CARLA simulation environment for these Python files to operate. Each of these objects is a separate thread, completely isolated from the server, solely responsible for performing the computation task specified in the Python file, and with absolutely no involvement in other server tasks. After the logic in this file executes, it returns the simulation server data and the sensor data as BGRA images via an "emit()" function. The processing loop continues indefinitely until it receives a stop command from the client without blocking the main socket server. Further analysis of the "create" syntax: when the computation server receives a request from the client, it executes the Python file stored on the server with the session directory corresponding to the client. The "create" statement, with its three subsequent parameters, provides the computation server with information, including the name of the vehicle type to be created, the object's location within the CARLA world, and whether to use the model. These parameters will be passed directly into these Python files via APIs when they are executed. The pseudocode snippet below further visualizes the idea of executing a Python script on a computational server.

Algorithm: Computational Server Python script execution

```
Receive create command
Create ClientYoloSession
Create SessionAPI for this session
Start session thread
Execute the uploaded Python script with the API injected
Select entry style:
   - control_run(api), or
   - main(api), or
   - run(api), or
User script processes CARLA data
User script sends result frames through emit_frame()
Stop the session when requested
END
```

Integrating machine learning models into the CARLA simulation environment is defined as using models to infer frames generated from the simulation environment to detect or identify objects within them for the purpose of controlling or testing autonomous vehicle algorithms. As mentioned and analyzed in section 3.4, embedding models into the simulation (which can be understood as reading a video) essentially means the system reads each frame sequentially and applies deep learning inference to it. Similarly, the CARLA simulation environment, when visualizing sensor data into images, is also a form of video with consecutive frames, so the idea is equivalent. After the computational server interacts with the simulation server, this data is output in the format of frames stored in buffers with the BGRA color space. Based on the theory discussed in sections 2.2.1 and 3.4 regarding the processing of machine learning models (here, YOLO), this data needs to be normalized before applying inference. The system reads a buffer containing this data; from an image with BGRA color space, the Alpha (A) channel is removed, and then it is converted to RGB. This normalized data is then applied to each frame sequentially using the deep learning model (YOLO) and stored in a separate buffer. These buffers, which contain processed data, include information about objects within the frame and the image with bounding boxes drawn by the deep learning model. Part of the data is returned to the executing computation tasks to continue the algorithms' processing and control cycle, while the remaining frames are processed and transmitted to the client for display, as described in section 4.2 for non-text data. For deep learning models, the server will always maintain a default model, YOLO (version 8), denoted by "n" and with the full name "YOLOv8n.pt," because version 8 of YOLO meets most object detection tasks. Additionally, this model is relatively lightweight, so it won't consume too many server resources or cause frame rate lag due to a heavy model. Furthermore, the system allows uploading user-created models, but they must also belong to the YOLO group, and their usage will be similar to that of the

default model on the server. By default, the system will not automatically apply deep learning models when executing Python tasks or launching simulations. The use of the model is enabled when a "yolo" command flag is added to the end of the statement sent from the client, as described in the syntax above. When the "yolo" flag is enabled, the server will use the uploaded model or the model stored on the server; otherwise, YOLO will be explicitly disabled. The pseudocode below illustrates the system's processing flow for the frames.

Algorithm: Load model and process frame inside the Python script

```
1:  read model path from yolo_path
2:  load YOLO model from file
3:  run one dummy prediction for warm-up
4:
5:  WHILE session is active DO
6:       capture frame from CARLA camera
7:       preprocess frame
8:       run YOLO inference on frame
9:       extract bounding boxes, classes, and confidence
10:       draw detection results on frame
11:       combine processed views into one output frame
12:       send frame to client with emit_frame()
13:  END WHILE
```

The idea of executing Python scripts on a computational server can be summarized as encapsulating the computation task within files uploaded from each client within a dedicated session object. These scripts run in their own threads via an embedded API, while the main TCP server handles only communication and session control. Integrating the deep learning model into the CARLA simulation environment involves applying it to individual frames and returning the results. These frames are then continuously streamed back to the client, creating the feeling of streaming a sensor stream with the model embedded within.

4.6 Stream Data to Local PC Workstations

Data transmission, consisting of frames, from the computational server back to the client, occurs after Python files are executed, and the "emit()" function pushes these frames back into the computational server's buffer. The data stored in this cache is then sequentially extracted and compressed into JPEG format by the OpenCV library, along with relevant image quality parameters. This data stream is then transmitted to the client via socket communication and the TCP protocol. It's important to note that although this data stream has been converted to JPEG, it remains essentially encoded binary bytes so that it can be sent directly to the

client without further conversion. However, the size of each frame varies because they encode different colors, objects, and scenes, requiring a special mechanism to prevent confusion at the client end. Based on the analysis in section 3.5 and the data transmission technique in section 4.2, the technique using a 4-byte prefix for this data type was chosen because it meets the technical requirements for algorithmic simplicity, consumes few resources, and is easily applicable to the existing TCP protocol. Immediately after the data is compressed into JPEG format, the system sequentially sends all the data in the buffer to the client as packets, each packet containing a 4-byte prefix indicating the size of the data to be read, followed by the corresponding amount of data. The pseudocode below will visualize the idea, approach, and server-side processing flow for this task.

Algorithm: Handle data and send frame on Server side

```
1:  def stream_frames():
2:      send "STREAM_START\n"
3:
4:      while not stop_flag:
5:          frame = get_camera_frame()
6:          bgr = convert_raw_to_bgr(frame)
7:          jpg = encode_to_jpeg(bgr)
8:          data = convert jpg to bytes
9:
10:          send 4-byte frame length
11:          send JPEG bytes
12:
13:      send 4-byte value 0
```

On the client side, when the text stream signal appears (e.g., "START_STREAM\n"), the client immediately understands that a stream is being sent and switches to binary reception mode, meaning it switches to streaming the data stream and stops receiving text data. This mode was changed because, in the normal text data reading state, the system is configured to search for points containing the "\n" character as a delimiter, which is interpreted as the end of a command. If this search method were applied to non-text data, it would cause serious errors when the client streams, because the "\n" character could appear within the encoded data string in packets sent by the server. When the client reads these characters, according to the old logic, it would interpret them as a stop command, which could cause frame freezing or, worse, incorrect frame size reading, leading to stream crashes. The client continuously decodes packets sent from the server; first, it reads the 4-byte prefix and identifies the data to be read, then it decodes the JPEG byte data into an image in BGR color space. At this point, a Pygame window will appear to display

these frames. However, these frames need to be converted to the RGB color space beforehand, as Pygame can only display them correctly when the default frame color space is RGB. This stream remains continuous and uninterrupted until the user requests a stop via a "stop()" command. It is worth explaining that, similar to the server setup, client-side execution tasks are also built on the idea of a main thread and parallel child threads. From this concept, when the client's main thread receives a streaming message from the server, it creates a child thread to handle the entire task, completely separate from the main thread. However, when the client is in streaming mode, the main thread locks down all other tasks, including querying client information, uploading files, or creating a new stream, leaving only the functions to stop streaming. The purpose of this locking is due to the system's design and requirements. When running a streaming service, uploading model files or other Python files violates logic because they need to have their new paths and related parameters updated to create the simulation environment. Additionally, creating a new stream in parallel with the current one can confuse the server, overload it, or cause the simulation server to crash due to unclear client control commands directed at specific objects within it. The main stream will continue to operate independently to maintain a connection to the server and receive the "stop()" command from the user, then send a signal to stop the stream to the server. The server will respond to the client with a packet containing an End of Frame (EOF) signal with a value of 0. The pseudocode below summarizes and clarifies the approach to handling the streaming signal at the client level.

Algorithm: Handle data and display frame on Client side

```
1:  def receive_and_display():
2:      connect to server
3:      wait for "STREAM_START\n"
4:      switch to binary receive mode
5:
6:      while streaming:
7:      size = read 4-byte frame length
8:      if size == 0:
9:          stop stream
10:
11:      data = read exact size bytes
12:      bgr = decode JPEG data
13:      rgb = convert BGR to RGB
14:      display RGB frame with pygame
15:
16:      send "stop()" to server
```

4.7 User Interface (GUI) for the System

The initial idea behind the system was simply to build a system that could migrate all computing tasks to a cloud server via a command-line interface. With a command-line interface, task commands are clearly and fully expressed within a predefined structure to perform the tasks described in the pseudocode below.

CLI commands

```
# Upload Python Script
upload_py <path_to_file>
# Upload YOLO Model
upload_yolo <path_to_file>
# Execute Python Script and Streaming
create <vehicle_name> <spawn_point> <flag=yolo>
```

This setup makes requirements clear to both the system and the user, making management easier. Furthermore, the command-line interface is relatively simple and lightweight because it lacks any graphical displays, making it easy to deploy and install on the client side. However, during testing, the command-line interface revealed several limitations and difficulties for users without extensive experience in controlling the system via commands (especially in UNIX operating systems). This limitation stems from the fact that a single control command sent from the client is often too long. While this design makes task requests clearer, it also creates some inconvenience when sending requests to the server. Because of these limitations, creating a user interface is necessary to simplify system usage, allowing clients to complete task requests via established buttons. The graphical user interface (GUI) will be built using Python's Qt5 library, while the actual network logic and data transmission will still utilize Python sockets, TCP, OpenCV, and NumPy underneath. The basic design of this graphical user interface will be described in Figure 4.4. The UI concept will be built around three main issues: simplifying task requests, directly displaying the streaming stream within it, and finally, being able to interact with the server and display logs. To simplify client task requests, the initial designs will remain as analyzed in the sections above, but the command parameters will be assigned sequentially to input fields, and these events will be set to buttons. The first task is to simplify changing the port or IP address for connecting from the client to the server. Previously, with the command-line interface (CLI), when wanting to switch to a different server branch with a changed IP and port, the user needed to type the "disconnect" command to stop the client, then restart the client application with the IP and port parameters changed in the configuration file. However, with a graphical user interface (GUI), the interface will always be maintained, and changing the IP and port will be entered in input fields, with a "Connect" button designed to trigger the connection to the server. For uploading Python files and model files, in the command-line interface (CLI), the user needs to

enter the syntax:

upload _ <type_of_file> <path_to_file>

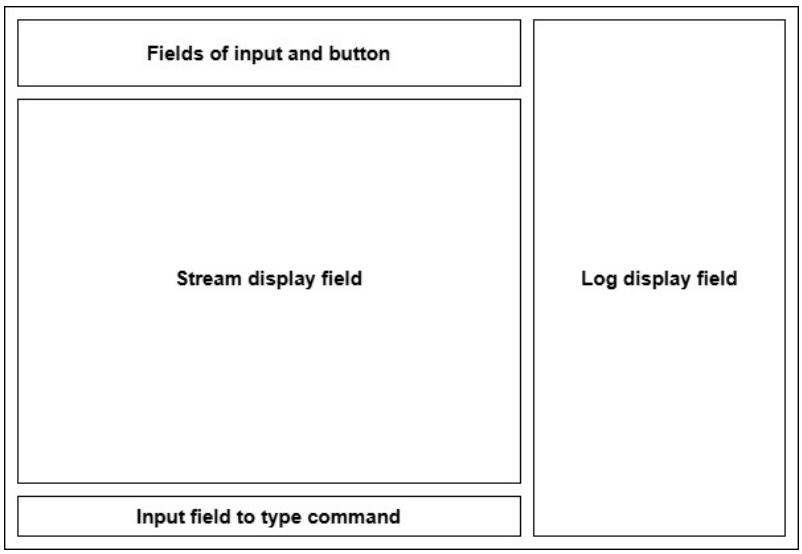


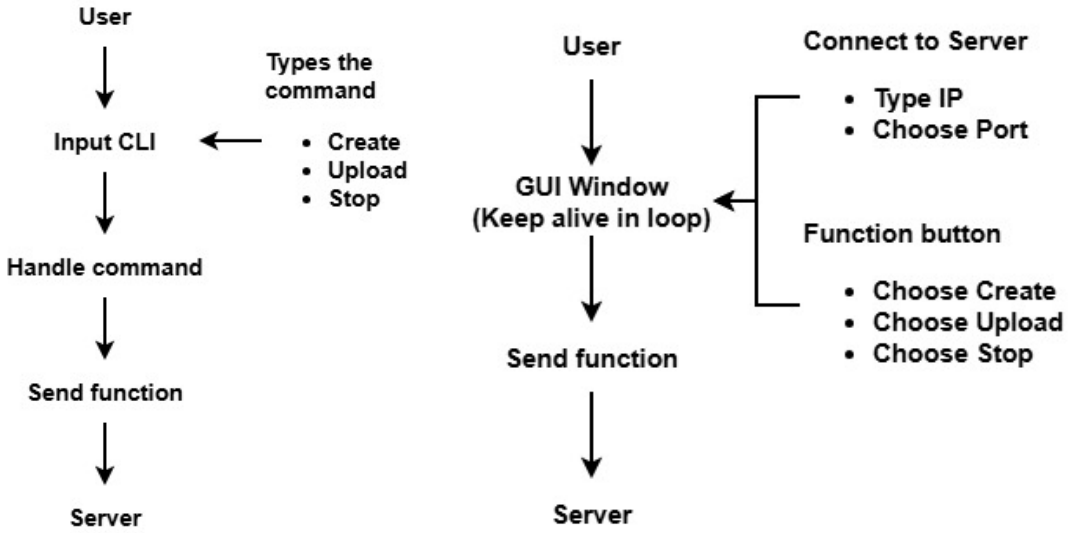
Figure 4.4: Basic design of a graphical user interface (GUI).

This syntax will remain the same; however, instead of manually typing the entire path or copying it from the clipboard, the system will be designed with a button to open a window to the location of the file to be uploaded. Once the file is selected, the upload event will start, sending a similar syntax to the system without the user having to manually enter a long path to the file location. Alternatively, for the task of executing the uploaded Python file and starting a streaming stream, it also needs to send the following syntax to the server:

create <vehicle> <spawn_point> <flag=yolo>

When applying the GUI, the parameters <vehicle> and <spawn_point> will be entered into input fields and remain fixed until they are changed by the user. For the parameter <flag=yolo>, it will be converted to a checkbox, implicitly turning on or off the use of the YOLO model file. A "Start stream" button will be created; when this event is triggered, it will retrieve the values from the input fields <vehicle> and <spawn_point>, then check if the parameter <flag=yolo> is selected, and finally send the "create" syntax with all the above parameters to the server. The second issue to address when building a graphical user interface (GUI) is streaming the data stream directly into the interface window. Unlike the method described earlier, which used Pygame to display decoded frames, a Qt5-based GUI cannot embed Pygame directly. Therefore, the streaming frame will be displayed using a GUI widget within the same application window. The entire image processing logic will be similar to the original design; however, when receiving a frame, the system will

create a QImage from the NumPy RGB data, then convert it to a QPixmap, and finally display it within the existing window using setPixmap(). The final issue to address is the interaction and display of communication logs between the server and the client. This task is relatively simple; the user interface window will still retain an input field so that the user can manually enter all commands. A display field will be added to the current interface; when the client interacts with the server, the entire communication log will be displayed within it. This display field makes communication clearer when receiving responses from the server and interactive information, such as file upload or execution commands. Similar to the initial design for the command-line interface, the graphical user interface will also be designed based on the division of main and sub-threads. Tasks, when executed, will be pushed into sub-threads, completely separate from the main thread, keeping the interface window always active until it is closed. Additionally, as analyzed in section 4.6, when the streaming task starts, all other tasks will be locked; similarly, the graphical user interface will lock all input fields and buttons, adhering to the system's design logic. Figure 4.5 illustrates the workflow and compares the command-line and graphical user interfaces.



(a) CLI Flow (b) GUI Flow

Figure 4.5: Compare the workflow of CLI and GUI.

In short, designing a user interface makes it easier to interact with the system. The interface does not change the system's structure or logic; it simply simplifies usage by replacing manual entry of long commands with function buttons that send those commands to the server. These function buttons are assigned events that trigger when pressed, reducing the need for lengthy commands and saving time and improving work efficiency.

5 Result and Evaluation

5.1 Result

5.1.1 Overall system implementation

The development and experimentation process has shown that the system model used to integrate deep learning models into the cloud-based CARLA simulation environment has been successfully built. In the implemented design, users first establish a client-server connection via Python API Socket communication and the TCP protocol. Once the connection has undergone sufficient security procedures and is accepted, the server creates a separate and unique session for that client. Each client session includes a session identifier with a unique structure (Unix-timestamp_IP_Port), including directories for storing Python scripts, and the YOLO model files are also based on this structure. The session-based system design not only allows the server to manage each client connection independently but also makes the server's operational organization clearer. The result of the first deployment step is the establishment of a session management framework and system communication structure, serving as the foundation for all subsequent system operations and server functions.

After the connection has been established and a session created individually for each client, the next stage of the process allows the client to upload the necessary files to the server. In this project, the client can upload a Python script containing user-defined simulation logic, along with a YOLO model file for object detection. When the server receives the upload command, it validates the transmitted information, stores the received files in the corresponding session directory, and performs syntax checking on the uploaded Python script before accepting it for later execution. These files are only stored in the client's corresponding session directory and are not executed until a request from the client is sent. This is an important part of the deployment process because it separates file transfer from execution, making the system more structured and reducing errors during uploads. Once the necessary files are available in each session's directory, the client can send a command to the server to begin the execution process. During this processing phase, the server initializes an object to run the session (a subthread) and executes the uploaded Python script on the server side with environment parameters and the path to the inserted Python file and model file, providing controlled access to the CARLA environment and streaming functions. Through this mechanism, the uploaded script can interact

with the simulator without a fixed program structure, requiring only a function to push frames back to the server. This deployment result demonstrates that the developed platform is capable of supporting remote execution of user-provided Python code in a controlled server-side environment, which is one of the core objectives of the project. In parallel with remote script execution, the system provides real-time visual feedback through frame transmission. While the simulation is running, frames generated from the CARLA environment can be encoded on the server side as JPEG data and continuously transmitted to the client. Before starting binary frame transmission, the server sends a stream start signal, then each frame is transmitted with a 4-byte prefix, followed by the corresponding data in JPEG format. On the client side, the received binary data is decoded, converted into a viewable image format, and then displayed on the user interface. This confirms that the implementation is not limited to remote command exchange and execution but also includes a complete transmission process for monitoring simulation results in near real-time.

Overall, the project's implementation demonstrated a complete end-to-end workflow and was successfully integrated into a unified client-server platform. The results demonstrated that the proposed architecture is indeed feasible, not only conceptually designed but also functionally realized within a single operating framework.

5.1.2 Communication result

The client-server architecture of the system is built on the idea of combining Python API Sockets and the TCP protocol, providing a basic foundation for communication, message transmission, and signal processing in the data flow between the server and the client. When the client connects to the selected server port, communication begins with the exchange of control messages that determine the next action to be performed. The interaction between the server and client when establishing a connection via Python API Sockets is shown in Figure 5.1, which represents the server side acknowledging receipt of the connection from the client, and Figure 5.2 shows the client beginning operations while waiting for user input.

```
(.venv) PS E:\test> python -m CARLA_Stream_all_py.server.server
Server listening on 0.0.0.0:2150.
Commands: connect [MapName] --port <PORT> | disconnect | status | down | set vehicle … | set spawn …
> [Session] 1773605752_192-168-0-105_3088 created at E:\test\sessions\1773605752_192-168-0-105_3088
Client connected from ('192.168.0.105', 3088).
```

Figure 5.1: Connect from the client side and establish a session.

The problem of isolating and separating the activity stream to manage sessions on the server through sub-streams has also been successfully established. Through the

connection description in Figure 5.1, it can be seen that the server created a sub-stream and used the Unix-timestamp_IP_Port structure to assign to this sub-stream (1773605752_192-168-0-105_3088). On the client side, Figure 5.2 also shows the result of verifying the connection using the control command "info", with the server's response corresponding to the structure mentioned above. The successful establish-

```
(.venv) PS E:\test> python -m CARLA_Stream_all_py.client.client
pygame 2.6.1 (SDL 2.28.4, Python 3.8.10)
Hello from the pygame community. https://www.pygame.org/contribute.html
Commands:
  upload_py <local_path>
  upload_yolo <local_path>
  demo [vehicle] [spawn]              (server camera demo)
  create <vehicle> <spawn> [yolo]     (run your uploaded script; add 'yolo' to enable model)
  stop
  info
  exit
> info
Session ID: 1773605752_192-168-0-105_3088
>
```

Figure 5.2: Client-side operations and session verification.

ment of this communication structure is a significant result because it provides a simple yet flexible method to control server activity from the client side. Instead of placing processing logic on the client side, the system simply transmits plain text commands, while the actual processing is entirely handled on the server.

Another important outcome of the communication implementation is the separation between text-based command transmission and binary data transmission. Typical control commands are sent to the server and processed by the server as text, as analyzed in section 4.2, via the "\n" delimiter mechanism, making them easy to read and manage in the communication process. For non-text data, a more specific mechanism, a 4-byte prefix, is established, allowing the client to identify the begin and end points of each frame exactly. The results from these experiments lead to the conclusion that the platform is suitable for realistic simulation control, where users may need to upload files, start execution, stop a running process, or request data streaming within a connected workflow. Furthermore, the communication mechanism between the server and client not only functions as a simple transport channel but also acts as the backbone of the system, connecting all the main functional modules of the system and indirectly confirming that the communication architecture has been successfully implemented and is operating correctly as designed.

5.1.3 File upload and execution result

The migration of computational tasks from local workstations to cloud servers is a central part of the entire research. This is achieved by developing a process that

enables local workstations to upload Python files and YOLO models to the server, where they are stored and executed. This functionality is one of the main research components of the entire system; it allows users to control the CARLA simulation environment from local workstations without directly manipulating the GPU server. From this mechanism, the system has evolved into a practical server-client communication model for controlling the CARLA simulation environment and executing computational tasks.

The file upload process begins after the client session has been established. At this stage, the client selects the necessary Python files and YOLO model files from the local system and sends an upload request to the server along with relevant information. To illustrate this stage, Figure 5.3 shows the syntax for uploading a file from the client and receiving a response from the server. After receiving the request, the

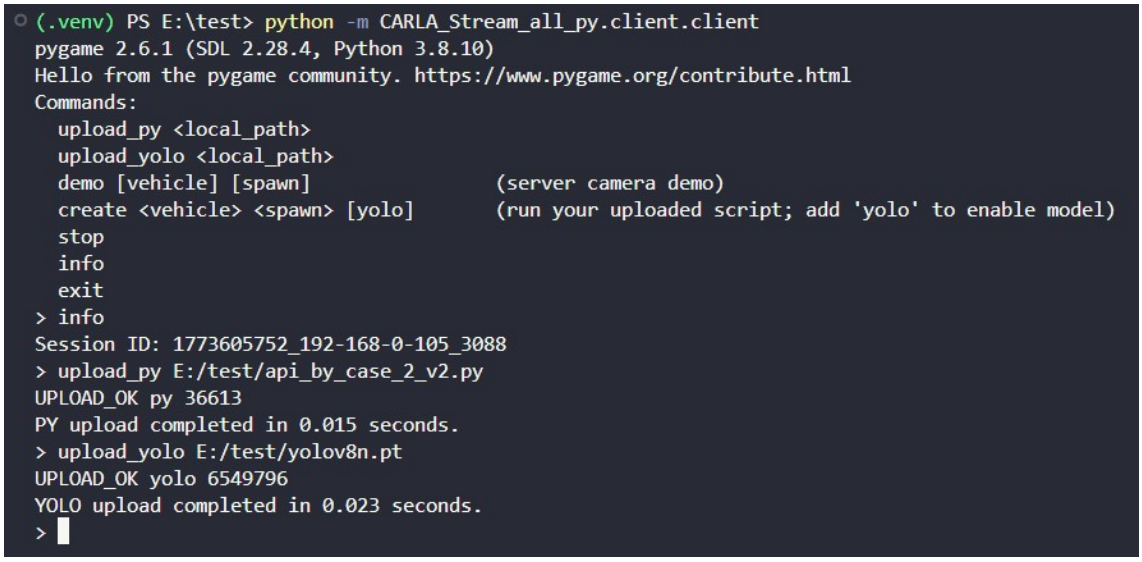


Figure 5.3: Syntax for uploading files and receiving responses from the server.

server prepares the corresponding session directory, accepts the incoming file data, and stores it in the specific location of the corresponding session. Figure 5.4 shows the processing of client requests on the server with the corresponding session and uploaded file. The above results show that the system can organize uploaded resources separately for each connected client, which is crucial for maintaining a clear workflow and separating each session. Furthermore, when these files are uploaded to the server, they are stored only in the session directory for each session and are not executed immediately. The uploading and execution operations are performed in two independent stages, a mechanism designed to help control the workflow clearly, ensuring that files from the client are fully uploaded before execution begins. Execution of a Python file on the server only begins when it receives a request from the client to trigger the execution function. The triggering of the Python file execution function on the client is illustrated in Figure 5.5. Upon receiving this request, the

```
(.venv) PS E:\test> python -m CARLA_Stream_all_py.server.server
Server listening on 0.0.0.0:2150.
Commands: connect [MapName] --port <PORT> | disconnect | status | down | set vehicle … | set spawn …
> [Session] 1773605752_192-168-0-105_3088 created at E:\test\sessions\1773605752_192-168-0-105_3088
Client connected from ('192.168.0.105', 3088).
Received from [('192.168.0.105', 3088)] commands: info
Received from [('192.168.0.105', 3088)] commands: upload py api_by_case_2_v2.py 36613
Received from [('192.168.0.105', 3088)] commands: upload yolo yolov8n.pt 6549796
```

Figure 5.4: The server receives requests from the client.

```
Session ID: 1773605752_192-168-0-105_3088
> upload_py E:/test/api_by_case_2_v2.py
UPLOAD_OK py 36613
PY upload completed in 0.015 seconds.
> upload_yolo E:/test/yolov8n.pt
UPLOAD_OK yolo 6549796
YOLO upload completed in 0.023 seconds.
> create cybertruck 0 yolo
NOT_CONNECTED: ask admin to 'connect' first.
> create cybertruck 0 yolo
STREAM: user-module starting...
>
```

Figure 5.5: Start the Python file execution function from the client.

server creates a session (a subthread), pushes all parameters related to the file location, establishes the CARLA emulation environment, and grants access to resources within that environment. At this stage, the Python file on the server does not function as a standalone program; it only receives access permissions, interacts with the CARLA simulation environment, and performs some image processing. Figure 5.6 illustrates the server receiving the client's request to execute the client's Python file, including parameters for the CARLA emulation environment. By designing the

```
connect
INFO:  Found the required file in cache!  Carla/Maps/Nav/Town10HD_Opt.bin
INFO:  Found the required file in cache!  Carla/Maps/TM/Town10HD_Opt.bin
CARLA connected. Current map: Town10HD_Opt
> Received from [('192.168.0.105', 3088)] commands: create cybertruck 0 yolo
```

Figure 5.6: This signal initiates the execution of the Python file on the server.

Python file uploaded from the client to have access to server resources while remaining within the server's control structure, it indirectly confirmed that the transfer of computational tasks to the server and their execution worked as designed.

5.1.4 Streaming result

In parallel with the task of executing Python files on the server, transmitting frames generated by the sensors from the CARLA simulation server to the client is also part

of the task. Streaming the sensor data stream is crucial for the system because it visualizes the results of executing the logic within the Python files, while also allowing users to remotely monitor the simulation without directly executing the graphics on CARLA on their local machine.

After the server executes the Python files and connects to the CARLA server to conduct the simulation, the sensors generate images to display the results. These frames are transferred to buffers in preparation for compression into JPEG format for transmission to the client via TCP socket communication. The images need to be converted to JPEG format because, in the BGR color space, their sizes are relatively large; sending them directly would be costly to transmit (in terms of payload and resources). Through JPEG compression, the system reduces the payload of these frames while retaining all the frame information. Once the frames are ready on the server, a control signal to start the stream is sent to the client first. The purpose of this signal is to inform the client that the stream is about to begin and that the client needs to switch to receiving binary data. The transmission of the frames begins with the server reading the buffers and sending each frame sequentially with a 4-byte prefix structure, followed by the data in JPEG format. On the client side, upon receiving the data stream signal, it begins the decoding process. The client reads the bytes in the packet sequentially, then decodes the compressed frames into image data, continues with color format conversion for display, and uses Pygame to visualize them. This streaming stream is illustrated in Figure 5.7, with a Pygame window open, containing the CARLA simulation image streamed directly from the server. Through this processing stream, the client can continuously display the im-

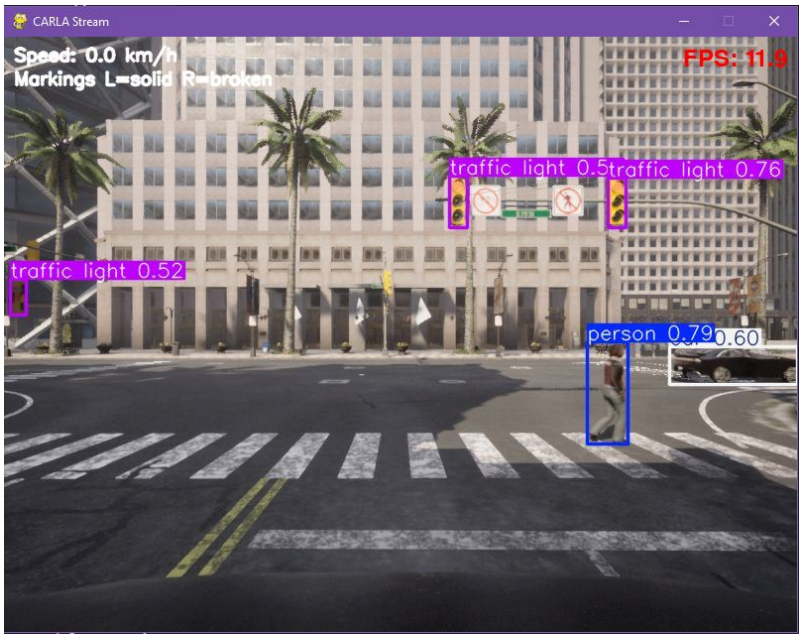


Figure 5.7: Visualize server-side frames using Pygame on the client.

age generated by the CARLA simulation sent from the server in near real-time. The result of this processing stream also allows users to directly observe the real-time behavior of the simulation through the transmitted image data.

5.1.5 GUI result

In practice, when conducting research, the client side acts not only as the communication endpoint but also as the primary interface through which users access and control the CARLA simulation environment. A basic command-line interface is sufficient for users to interact with the server. However, from a user's perspective, a command-line interface does not visually represent the system's functionality and is relatively difficult to use. In the context of this project, it is particularly important because users must be able to observe whether the uploaded script has been accepted, whether the execution process has started correctly, and whether the CARLA frames have been successfully transmitted. This makes the interaction more realistic because the same client application acts as both the command interface and the point of view for returned results. With a graphical user interface, it not only sends requests to the server but also receives responses in the form of status messages, execution responses, and visual stream output. In the graphical client version, this interaction is further enhanced through interface elements that simplify access to available functions. Figure 5.8 shows the graphical user interface (GUI) with functional clusters as designed in section 4.7. Instead of relying on manual

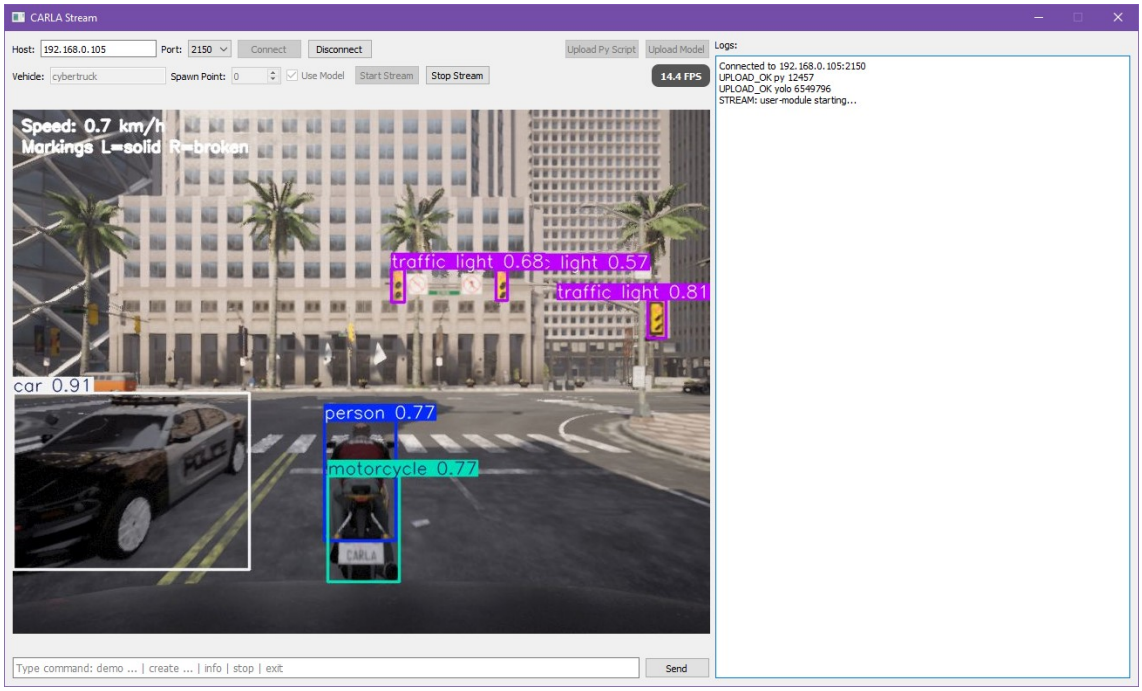


Figure 5.8: Graphical User Interface (GUI) visualization.

input via the command line, the graphical user interface (GUI) provides a simpler

and more intuitive way to easily perform operations such as establishing connections, selecting files, triggering Python file execution, and visualizing data streams from the server. This not only improves the usability of the system by reducing the complexity of repetitive manual input but also makes the workflow easier to follow. Simultaneously, although operations have shifted to interactive graphical interfaces, the command-based logic is maintained and consistent with the underlying design, meaning the graphical interface functions as a higher-level control layer rather than a separate system.

5.2 Evaluation

5.2.1 System Performance

The architecture of the entire system is designed to support remote interaction between the client and the server. Therefore, system performance is determined not only by whether the system operates correctly, but also by the efficiency of data transmission, user request response, and client task execution. In this project, performance evaluation focuses primarily on two criteria: latency and upload and execution times, as these directly affect system responsiveness and the overall usability of the remote simulation process.

Latency is a critical performance metric as it determines the system's response speed to remote user interactions. In the proposed platform, latency can be observed in several parts of the workflow, such as the delay between sending a command and receiving a response from the server, the delay between triggering an execution request and the actual start time of the uploaded script, and the delay between frame generation on the server and frame display on the client. Regarding the delay between sending a command and receiving a response, since the server and client are set up on the same network (not localhost) and communicate via relatively simple socket communication, the communication delay when transmitting commands between the server and client is very low, almost instantaneous, with a latency ranging from approximately 1.37 to a maximum of 50 milliseconds (ms). Regarding the delay between triggering the execution request and the time the file actually starts, since the delay when transmitting text commands, as mentioned, ranges only from 1.37 to a maximum of 50 ms, and the files execute directly on the server, the delay is negligible and depends on the server's power. Finally, the delay occurs when frames are pushed from the CARLA simulator to the server and transmitted back to the client. In reality, this processing flow on the server is instantaneous because the simulator server and the compute server are located on the same cloud server platform. When the compute server compresses and transmits the frames, it takes some time, and decoding them on the client side also takes a certain amount of time, so this creates a delay. However, this delay occurs only once, when the first frames are sent to the client and begin to display. Experimental data show that this delay is

approximately 0.6 to a maximum of 1 second for the graphical user interface. In the command-line interface, this delay is negligible; the difference stems from the different architectures used to display frames. The graphical user interface using the Qt5 library does not directly embed Pygame into the processing stream but uses QImage functions for conversion before displaying it on the interface. So, this approach using the graphical user interface (GUI) makes it slightly slower than the default command-line interface. Besides, the system is designed to meet low-latency and stability requirements, thereby improving the interactive user experience. Although delays are unavoidable in network architectures based on Socket communication and TCP protocols, especially for binary transmission. However, if the system's actual performance remains within an acceptable range for monitoring and controlling the CARLA environment, then the trade-off is still worthwhile.

The second evaluation criterion is upload time, which reflects the system's efficiency in processing user-uploaded files. The files from the client need to be executed, must be uploaded to the server, then stored in the client's session directory, and prepared for execution before the Python file execution process. The client's processing flow when handling file uploads is a binary stream of the underlying TCP protocol transmitted over a socket. Therefore, the upload time of these scripts depends entirely on the size of the Python files. Figure 5.9 illustrates the upload time of a Python file and a YOLO model file in experimental data. Experimental results show that

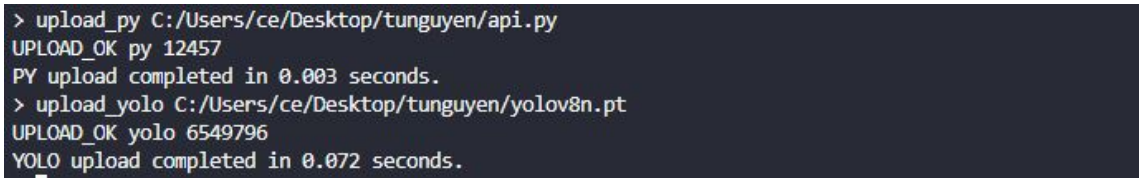


Figure 5.9: Upload time of Python files and YOLO models from the client.

for a Python file of 12457 bytes (equivalent to 0.012 MB), the client only takes 0.003 (equivalent to 3 ms) seconds to upload, and for a YOLO model file of 6549796 bytes (equivalent to 6.25 MB), the upload time is also only about 0.072 seconds (equivalent to 72 ms). The upload time is calculated from the moment the upload command begins sending data until the server finishes reading the data stream and successfully saves the file to the corresponding session directory. The time spent uploading plays an important role because it affects the speed at which users can transition from the preparation phase to actual simulation control. If uploading and execution take too much time, the feasibility of the remote method will be reduced, even if the system remains exactly functional. For this reason, evaluating the time required to upload a file provides an important measure of the platform's operational efficiency.

To provide a more detailed view of system performance, Figure 5.10 presents a detailed time analysis of the main workflow, including connection establishment, file upload, frame transmission, and client-side rendering. In real-world testing on

the system, the average time required to establish a connection between the local workstation and the computing server after every ten attempts was 10.19 ms. This result indicates that establishing a connection via the TCP three-step handshake and socket connection is minimal in the tested network environment, consistent with the lightweight communication logic implemented by the client and server. Through the current implementation, client-side upload time is measured from the moment a request to upload Python script files or YOLO models is made and the binary data stream begins to be sent until the client receives confirmation of upload completion or an error response from the server. This means the measured value includes both transmission time and server completion confirmation. A more detailed analysis of uploading files from the client also reveals the expected relationship between file size and upload time. As illustrated in Figure 5.10, after every ten attempts, the results show that uploading a Python script of 12,457 bytes (approximately 0.012 MB) takes an average of about 3.92 ms, while uploading a YOLO model file of 6,549,796 bytes (approximately 6.25 MB) takes an average of about 75.69 ms. This confirms that the upload process remains efficient even for significantly larger model files, although the transmission time clearly increases proportionally with the amount of data transmitted. In the system being deployed, file uploads are performed as binary data stream transmissions using the TCP protocol after a text-based upload command is sent, and the server stores the received file in its corresponding session directory before returning a completion status to the client. Therefore, the measured upload time reflects not only the raw transmission over the socket, but also the actual end-to-end latency experienced by the user before the file is fully available on the server for later execution.

Regarding the Python script file execution and streaming functionality, the measured process shows that the connection from the compute server to the simulator server and the processing within the CARLA environment are negligible compared to other phases. This is reasonable since both services are deployed on the same cloud platform, thus the communication path between them is much shorter and does not cause significant network latency in the observed setup. The most significant latency in the continuous operation of the system occurs during the frame transmission and display process. After the transmission mode begins, the server sends a "STREAM_START" signal and then transmits each compressed JPEG frame with a 4-byte prefix, followed by a binary payload. On the client side, the frame is reconstructed by reading the pre-specified dimensions, accurately retrieving the frame bytes, decoding the JPEG image, converting from BGR to RGB, and then displaying it on the client interface. Measurements in Figure 5.10 show that the average time of the entire process to send a frame from the computing server back to the local workstation, excluding the encoding time on the server, is approximately 18.42 ms when the processing includes drawing bounding boxes and approximately 8.34 ms when there is no bounding box processing. Considering only the system's transmission process, packets containing encoded frames take an average of approximately 17,546 ms when additional bounding boxes are processed, and about 7,476

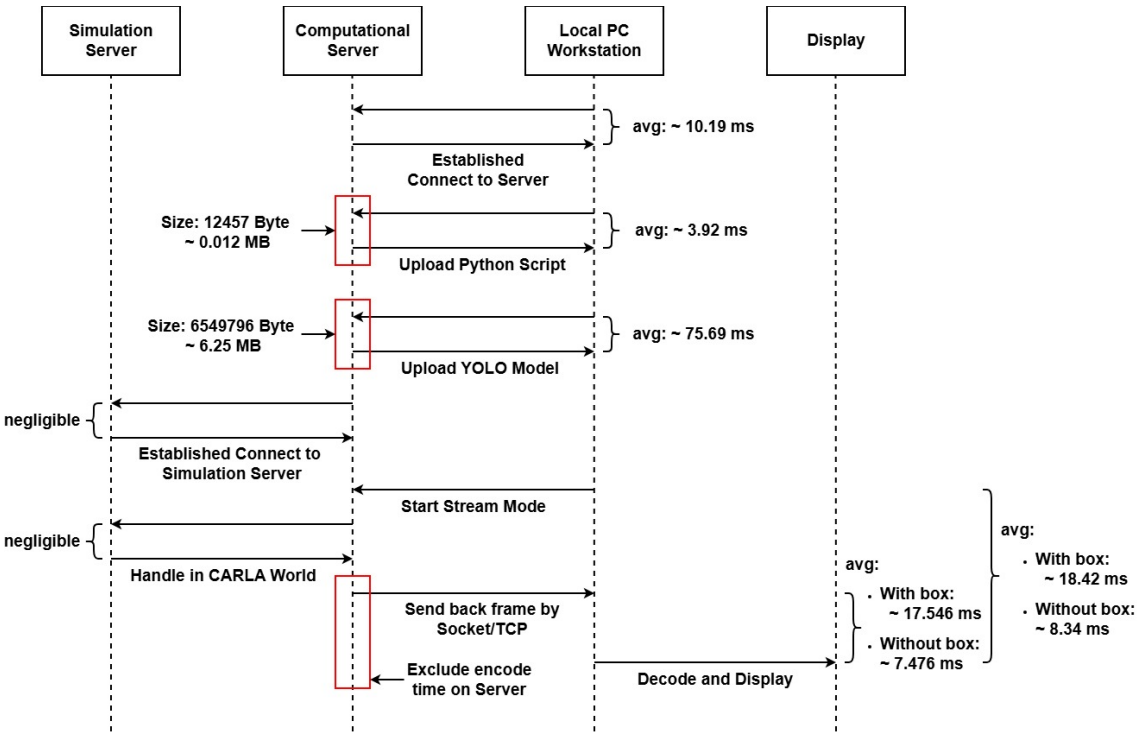


Figure 5.10: Illustrates the detailed timing breakdown of the system workflow.

ms when these frames do not include object recognition. This large difference is explained by the fact that normal frames do not include bounding boxes, class labels, and confidence values, so the compressed binary payload in each packet is very low. Conversely, when a frame carries too much information, the compressed and transmitted packet becomes much larger, thus taking much longer, potentially up to three times longer in some cases if too many objects are recognized in a single frame. These values show that the key performance differences stem not only from network transmission, but also from the amount of image processing that needs to be completed before a final frame is displayed.

Detailed real-time data analysis of all functions and overall system evaluation shows that the proposed platform maintains a low-latency workflow across all key operational phases. Connection setup and command-related interactions are very fast, file uploads remain efficient even for larger model files, and the most significant time overhead is concentrated on frame transmission, decoding, and client-side rendering. From a practical standpoint and considering the requirements of the system design, this result is acceptable as the system is designed for remote monitoring and control of the CARLA environment, not for local rendering with extremely low latency.

5.2.2 Multi-User Capability and Scalability

In the context of this project, the two aspects of multi-user capability and scalability are crucial, as this research is not only expected to work for a single client connection but also to provide a structural basis for scaling to support use cases with multiple users and multiple connections simultaneously. The entire system is based on a client-server architecture, so the ability to manage multiple connections as well as allocate resources, and maintain stable operation becomes an indispensable part of the overall evaluation.

From a multi-user capability perspective, the implemented architecture possesses a crucial foundation: session-based design. When a client connects to the server, a sub-thread (session) is created, including separate identifiers and storage locations for uploaded files and execution-related data. This separates the connections of one user from another and reduces the risk of conflicts between the two connections on the server. In practice, this means the system design has been prepared for controlled separation for multi-user support operations. Therefore, the existence of this session mechanism is a positive outcome, because it shows that the system is not only built as a monolithic execution thread supporting only one connection, but also as a modular system capable of organizing multiple independent interactions from multiple users. Figure 5.11 shows that the server has achieved the ability to establish and allocate resources simultaneously for multiple connections; it can be seen that two independent directories with different IP address labels (1773755056_192-168-0-100_49526 and 1773755071_192-168-0-105_63282) are coexisting on the server. Although sessions can be created separately, the current implementation also has

```
Server listening on 0.0.0.0:2150.
Commands: connect [MapName] --port <PORT> | disconnect | status | down | set vehicle … | set spawn …
> [Session] 1773755056_192-168-0-100_49526 created at E:\test\sessions\1773755056_192-168-0-100_49526
Client connected from ('192.168.0.100', 49526).
[Session] 1773755071_192-168-0-105_63282 created at E:\test\sessions\1773755071_192-168-0-105_63282
Client connected from ('192.168.0.105', 63282).
status
========= SERVER INFO =========
 Bind: 0.0.0.0:2150
 CARLA: 127.0.0.1:2000
 Connected to CARLA: False
 Current CARLA Map: (unknown)
 Connect target map: (not specified; keep current)
 Clients connected: 2
 Vehicle default (demo): (random car)
 Spawn index (demo): (random)
 TM port: 8000
 Started at: 2026-03-17 14:35:06
================================
> Received from [('192.168.0.100', 49526)] commands: upload py client_v2.py 19062
Received from [('192.168.0.100', 49526)] commands: upload yolo best.pt 6238435
Received from [('192.168.0.105', 63282)] commands: upload py api.py 12457
Received from [('192.168.0.105', 63282)] commands: upload yolo yolov8n.pt 6549796
```

Figure 5.11: Handling multiple connections on the server.

practical limitations regarding fully simultaneous multi-user operation. The reason for this problem stems from the system's actual scalability, which depends entirely on available server resources and how internally the emulation and execution services are managed. When the CARLA emulation server begins performing emulation tasks, as well as framing and processing related to deep learning, it places significant pressure on CPU, GPU, memory, and network bandwidth allocation. This forces the server to divide these resources among sessions, resulting in a sharp drop in server performance unless additional resource scheduling or load-balancing mechanisms are implemented. With limitations and reasons above, although the architecture conceptually supports multiple-session workflows, however, the current implementation is better suited for controlled, small-scale use rather than for large-scale, and unlimited concurrent access.

An important point to consider is the system's scalability, which can be examined from the platform's communication and service architecture. Because the system is designed and built to handle multiple connections via a session-based architecture on the server side and defined communication ports, it provides a fundamental framework for extending the connection delivery beyond a single client. This is an advantage for the whole system because it allows the system to be scaled by adding service instances or implementing more advanced connection management in future development. In other words, the proposed design includes features that support theoretical scalability, even if the current implementation does not fully optimize for a large number of users simultaneously. Another scalability consideration in this study is that scalability is not limited to the number of users connected at a time, but also includes the system's ability to support more complex tasks over time. An example of these extensions in the future could include handling more uploaded scripts, supporting more simulation scenarios, integrating additional object detection models beyond YOLO, as well as extending the graphical interface for more advanced user interaction. Therefore, scalability also refers to the flexibility of the system architecture to accommodate additional features without a complete redesign.

5.2.3 Security and Data Validation

In a system where users can remotely connect, upload Python files, and trigger server-side execution, security is a critical concern because the server must not only provide functionality but also maintain controlled and predictable operation. So, this is the reason why it is necessary to evaluate how command validation, uploaded file verification, as well as the extent to which these mechanisms contribute to protecting the system from invalid input and unwanted behavior.

By design, before execution can begin, the client must first establish a valid session and upload the necessary files via a defined protocol. This means user input is processed in a way that goes through a controlled sequence of processing and stor-

age steps. Furthermore, the mechanism for executing Python files from the client is not designed as a system with completely unrestricted direct access to the server. Instead, the uploaded scripts are executed through an environment that is a subthread created by the server, where it interacts with the system via a provided API structure. This design gives the system a certain degree of control because files and code snippets uploaded from the client are integrated into an execution environment defined by the server instead of being treated as an arbitrary, unmanaged process. From a security perspective, it offers quite a few benefits as it creates a clear boundary between the running server and the logic of files uploaded by users. While this in itself does not guarantee complete protection, it shows that the system has been implemented with a certain level of operational control rather than allowing unrestricted external manipulation of the server workflow. In parallel, the research also applies a format validation procedure to evaluate the reliability of files uploaded for each session and uses a structured transport mechanism to reduce errors caused by misformatted input or incorrect communication flows, thereby improving system stability and indirectly contributing to security. Figure 5. illustrates the first layer of protection when the client side verifies the file type of the uploaded file; for Python file uploads, if the file extension is not ".py", it will not be allowed to upload, along with a response to the user, also same idea applies to YOLO model files (".pt"). The protection against malicious input from targeted attacks requires

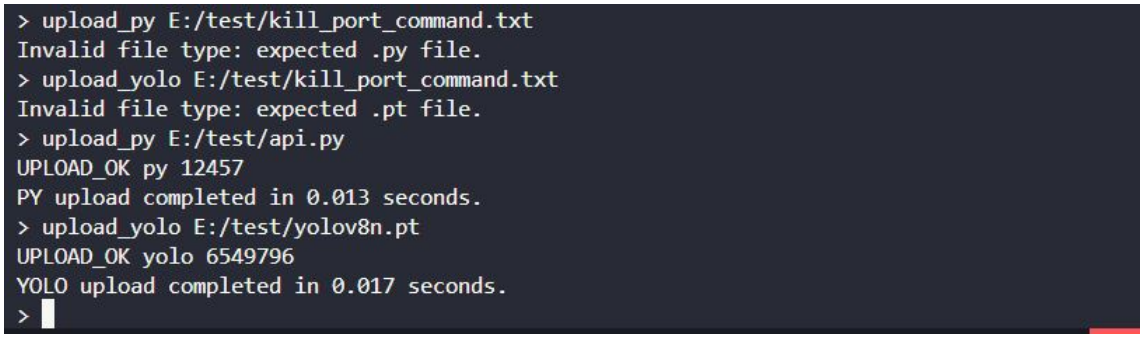


Figure 5.12: Describes the process of validating the file type uploaded by the client.

not only additional defence mechanisms beyond conventional validation, such as stricter command restrictions, but also adding more deeper inspection of uploaded content, and more robust runtime isolation. Therefore, while the current data validation method improves robustness, it should not be understood as a complete replacement for comprehensive security controls.

5.2.4 Reliability and Fault Tolerance

Within the scope and context of this study, system reliability refers to its ability to maintain accurate and stable operation under normal operating conditions. Conversely, fault tolerance refers to how a system reacts to unexpected conditions, such

as invalid commands, communication interruptions, failed uploads, execution errors, or thread termination. These two aspects are particularly important because, in practice, a system is considered to be performing well not only based on whether its core functions can work, but also on its ability to handle errors in a controlled manner without causing failures throughout the entire system structure.

Server-client communication is built using Python API Sockets and the TCP protocol; with each connection established, sub-threads are created to establish a session for each connection on the server. Furthermore, uploaded files undergo input validation, are then stored in directories belonging to each connection, and execution is only triggered after the necessary preparatory stages are completed. The process of the system mentioned above was designed to ensure that the operations of the entire system are performed in a predictable order. Therefore, conflicts during normal system operation will decrease because tasks such as communication, uploading, as well as execution are independent of one another and follow a controlled process, thereby directly contributing to the platform's operational reliability. Besides, the design of separating the system's different stages and functions reduces the risk that one type of operation will directly disrupt another part of the workflow. For example, keeping the upload and execution processes separate allows the server to verify that the file has been received and stored before attempting to run it. Alternatively, when streaming is active, it locks down all other functions; this type of design ensures both data integrity and increases system reliability because, logically, if a different Python script is to be used, the system needs to stop the stream and update again. Similarly, using a length-prefixed frame protocol in the streaming stage improves reliability by helping the client accurately reproduce frame boundaries from the TCP byte stream.

When considering the fault tolerance aspect of the entire system, the design, which isolates and separates each sub-thread (session) from the main thread and allows them to operate independently, helps to localize connection-related errors rather than immediately impacting the entire system workflow. In streaming tasks, the use of control signals such as stream start notifications and 4-byte prefix-based frame markers allows the client to know when streaming begins and ends, supporting more controlled handling of stream interruptions or termination. Additionally, the system incorporates protection mechanisms that ensure that when Python files are executed and connect to the emulation server, if a failure occurs due to internal file logic, the sub-thread disconnects and deletes the entire object generated by that sub-thread without crashing the entire emulation server. The constraint from the mechanism demonstrated that the system is not only designed for ideal operating conditions but also includes fundamental constraints to handle system failures. However, fault tolerance in current system implementations remains limited to the research level rather than a complete production standard readily applicable in practice. For example, unexpected exceptions on the server related to resource exhaustion, or deeper execution-level errors may require additional recovery mechanisms to prevent disruption or automatically restore sessions, so even more complex errors can still affect

the whole system operation. Therefore, in general view, fault tolerance in practice still depends on server resources and additional mechanisms, meaning that critical infrastructure-level errors can still affect multiple sessions simultaneously.

In summary, regarding reliability and fault tolerance, the proposed system design demonstrates well-structured, clearly separated operational phases and fundamental mechanisms for the controlled handling of communication, upload, execution, and transmission errors. The designed mechanisms contribute positively to the stable operation of the system under normal conditions, although there are still limitations in advanced fault recovery and large-scale recovery capabilities. Therefore, the system can be considered sufficiently reliable for controlled testing, while future improvements are needed to achieve stronger fault tolerance for more demanding real-world deployment scenarios.

5.2.5 UI/UX Design

In this project, the graphical user interface (GUI) is an important part of the system because it visually represents the system functions that users use to connect to the server. Therefore, the quality of the interface affects not only the system's appearance but also the practicality, clarity, and ease of use of the entire platform.

From a user-interaction perspective, the implemented interface reflects the system's workflow logic. The first workflow starts by the user establishing a connection to the server via an input field containing the IP and port, then uploading the necessary files by clicking a button to open a window directly to the file location, beginning execution, and finally observing the returned outputs or visual streams. Instead of forcing users to manage server-side processes directly, the user interface organizes access to these functions in a more manageable format. As a result, the platform becomes easier to operate, especially compared to a completely manual interaction method based on a command-line interface (CLI). Besides, the graphical user interface (GUI) not only provides the methods to input control commands and data but also serves as a tool to display results, status monitoring, and visualize sensors data from the simulation server, contributing to improving usability and making the system more user-friendly, as users can perform control operations and observe their effects within the same window. Through experimentation and practical implementation, the graphical user interface (GUI) has proven effective in allowing users to easily see whether a connection has been successfully established, file uploads have been accepted, as well as whether script execution and frame streaming is active, all without switching between multiple tools or windows, directly contributing to maintaining workflow continuity and improving the overall user experience. The design of the graphical user interface (GUI) delivers a completely bring a new experience to the system when users interact, but maintains the original logic and coherence of communication, meaning usability improvements are achieved without altering the core system architecture. However, the primary focus of this research is lean towards

on implementing the communication architecture, execution, and data transmission rather than building an application with pretty interfaces, so the GUI is primarily geared towards simplifying the process of interacting with the system rather than being a highly refined graphical interface. So, while the client-side GUI may offer sufficient control for testing and demonstration, it may still lack advanced features in a complete system, such as more detailed feedback error displays. Regarding the graphical user interface (GUI) in this research, it can be summarized that although the current design is more functional than purely optimized from a user experience perspective, it has provided a clear and usable foundation for the practical operation of the system in this project.

6 Conclusion

This thesis presented the design, implementation, and evaluation of a cloud-based client–server platform for CARLA simulation with support for remote script execution and real-time frame streaming. The main objective of this research was to develop a system that allows users to interact with a CARLA simulation environment hosted on a remote server and integrate the deep learning model into the simulator while reducing the computational task on the local PC workstation. To achieve this, the proposed architecture combined TCP-based communication, session-based server management, file upload handling, controlled execution of user-provided Python scripts, and JPEG-based frame streaming into a unified operational workflow. The final implementation demonstrated that these components could be successfully integrated into one functional platform for remote simulation control and monitoring.

The results of the implementation showed that the system could establish a structured workflow comprising client connection, session creation, file upload, server-side execution, and real-time visual feedback. Through this workflow, the client can serve as an access point for interacting with the CARLA simulation server, while the computational server handles the main computational task and pushes the results back to the client. This confirmed that the proposed platform is capable of supporting practical remote operation without requiring the user to execute the full simulation on a local workstation. In this way, the research achieved its main technical goal of separating user interaction from heavy computational processing through a distributed architecture.

The evaluation further showed that the developed system provides a reasonable level of performance and functionality for a research-oriented prototype. The platform supported continuous communication, handled uploaded scripts in a controlled, session-based structure, and streamed simulation frames from the server to the client in near real time. In addition, the modular separation between communication, file handling, execution control, and streaming provides a solid structural basis for future extension. However, the evaluation also identified several limitations, which include the dependence on network quality, the computational constraints of the server under increased workload, limited multi-user support in the current implementation, and the absence of advanced security, recovery, and large-scale resource management mechanisms. Therefore, while the platform is functionally successful within the intended research scope, but further development would be necessary before deployment in a real-world environment.

In conclusion, this thesis demonstrates that integrating the deep learning algorithm into the CARLA simulation based on a cloud-based client–server architecture and offloading all computational tasks to a computational server is both feasible and effective. The work provides a practical contribution by showing how communication, remote execution, and frame streaming can be combined into one operational system tailored to simulation-based experimentation. Beyond the implementation itself, the thesis also establishes a foundation for future research in areas such as multi-user scheduling, stronger security and isolation, more advanced GUI support, performance optimization, and tighter integration of deep-learning models such as YOLO for server-side perception tasks. As a result, the proposed system can be considered a meaningful step toward more flexible, scalable, and accessible simulation platforms for autonomous driving and related intelligent transportation research.

Bibliography

[1] Sebastian Thrun. Toward robotic cars. *Communications of the ACM*, 53(4):99–106, 3 2010.

[2] Francisca Rosique, Pedro J. Navarro, Carlos Fernández, and Antonio Padilla. A Systematic Review of Perception system and Simulators for Autonomous Vehicles research. *Sensors*, 19(3):648, 2 2019.

[3] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: an open urban driving simulator. *arXiv (Cornell University)*, pages 1–16, 11 2017.

[4] Youzi Xiao, Zhiqiang Tian, Jiachen Yu, Yinshu Zhang, Shuai Liu, Shaoyi Du, and Xuguang Lan. A review of object detection based on deep learning. *Multimedia Tools and Applications*, 79(33-34):23729–23791, 6 2020.

[5] Amr Sulaiman and Vivienne Sze. Energy-Efficient HOG-based Object Detection at 1080HD 60 fps with Multi-Scale Support. 10 2014.

[6] David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, 6 2004.

[7] Yanwei Pang and Jiale Cao. *Deep learning in object detection*. 1 2019.

[8] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *arXiv (Cornell University)*, 11 2013.

[9] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single Shot MultiBox Detector. *arXiv (Cornell University)*, 12 2015.

[10] Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi, University of Washington, Allen Institute for AI, and Facebook AI Research. You only look once: Unified, Real-Time Object Detection. *arXiv:1506.02640v5 [cs.CV] 9 May 2016*, 2016.

[11] Niranjan, B C VinayKarthik, and Mohana. Deep Learning based Object Detection Model for Autonomous Driving Research using CARLA Simulator. *2021 2nd International Conference on Smart Electronics and Communication (ICOSEC)*, pages 1251–1258, 10 2021.

[12] Óscar Pérez-Gil, Rafael Barea, Elena López-Guillén, Luis M. Bergasa, Carlos Gómez-Huélamo, Rodrigo Gutiérrez, and Alejandro Díaz-Díaz. Deep reinforcement learning based control for Autonomous Vehicles in CARLA. *Multimedia Tools and Applications*, 81(3):3553–3576, 1 2022.

[13] Anthony Navarro, Jendrik Joerdening, Rana Khalil, Aaron Brown, and Zachary Asher. Development of an autonomous vehicle control strategy using a single camera and deep neural networks. *SAE technical papers on CD-ROM/SAE technical paper series*, 1, 4 2018.

[14] Prashant Ahire, S M M Naidu, Sandeep Varpe, Sakshi Nadarge, and Anushree Patil. Simulating vehicle driving using CARLA. *Journal of Electrical Systems*, 20:44–52, 2024.

[15] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 6 2019.

[16] Jihye Jang, Khikmatullo Tulkinbekov, and Deok-Hwan Kim. Task offloading of deep learning services for autonomous driving in mobile edge computing. *Electronics*, 12(15):3223, 7 2023.

[17] Faisal Hawlader, François Robinet, and Raphaël Frank. Leveraging the edge and cloud for V2X-Based Real-Time object detection in autonomous driving. *arXiv (Cornell University)*, 8 2023.

[18] Siyuan Zhou, Duc Van Le, and Rui Tan. ECSeg: Edge-Cloud Switched Image Segmentation for Autonomous Vehicles. Technical report, 12 2024.

TECHNISCHE UNIVERSITÄT CHEMNITZ

Zentrales Prüfungsamt

Selbstständigkeitserklärung

Name: Nguyen Vorname: Anh Tu geb. am: 10.01.1998 Matr.-Nr.: 732738	**Bitte beachten:** 1. Bitte binden Sie dieses Blatt am Ende Ihrer Arbeit ein.

Selbstständigkeitserklärung*

Ich erkläre gegenüber der Technischen Universität Chemnitz, dass ich die vorliegende Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch nicht als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

Datum: 02.04.2026 Unterschrift: ……

* Statement of Authorship

I hereby certify to the Technische Universität Chemnitz that this thesis is all my own work and uses no external material other than that acknowledged in the text.

This work contains no plagiarism and all sentences or passages directly quoted from other people's work or including content derived from such work have been specifically credited to the authors and sources.

This paper has neither been submitted in the same or a similar form to any other examiner nor for the award of any other degree, nor has it previously been published.



This report - except logo Chemnitz University of Technology - is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made. The images or other third party material in this report are included in the report's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the report's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/.

Chemnitzer Informatik-Berichte

In der Reihe der Chemnitzer Informatik-Berichte sind folgende Berichte erschienen:

CSR-25-01 Md. Ali Awlad, Hasan Saadi Jaber Aljzaere, Wolfram Hardt, AUTOSAR Software Component for Atomic Straight Driving Patterns, März 2025, Chemnitz

CSR-25-02 Billava Vasantha Monisha, Hasan Saadi Jaber Aljzaere, Wolfram Hardt, Automotive Software Component for QT Based Car Status Visualization, März 2025, Chemnitz

CSR-25-03 Zahra Khadivi, Batbayar Battseren, Wolfram Hardt, Acoustic-Based MAV Propeller Inspection, Mai 2025, Chemnitz

CSR-25-04 Tripti Kumari Shukla, Ummay Ubaida Shegupta, Wolfram Hardt, Time Management Tool Development to Support Self-regulated Learning, August 2025, Chemnitz

CSR-25-05 Ambu Babu, Ummay Ubaida Shegupta, Wolfram Hardt, Development of a Retrieval Model based Backend of a Tutoring Agent, August 2025, Chemnitz

CSR-25-06 Shahid Ismail, Ummay Ubaida Shegupta, Wolfram Hardt, Development of a Generative Model based Backend of Tutoring Agent, August 2025, Chemnitz

CSR-25-07 Chaitanya Sravanthi Akula, Ummay Ubaida Shegupta, Wolfram Hardt, Integration of Learning Analytics into the ARC-Tutoring Workbench, August 2025, Chemnitz

CSR-25-08 Jörn Roth, Reda Harradi, Wolfram Hardt, Implementation of a Path Planning Algorithm for UAV Navigation, Dezember 2025, Chemnitz

CSR-25-09 Alhassan Khalil, Reda Harradi, Stephan Rupf, Wolfram Hardt, Development of an Automation Framework for 1D Measurement, Dezember 2025, Chemnitz

CSR-26-01 Vismay Gunda, Shadi Saleh, Wolfram Hardt, Cloud-Based AI Solutions for Ensuring Data Quality in Predictive Models, Februar 2026, Chemnitz

CSR-26-02 Sami Mansoor Alavi, Shadi Saleh, Wolfram Hardt, Continuous Integration for Cloud-Based Swarm Farming Applications, Februar 2026, Chemnitz

Chemnitzer Informatik-Berichte

CSR-26-03 Sarah Onyinyechi Obasi, Shadi Saleh, Wolfram Hardt, Cloud-Based AI for Data Completeness Analysis and Improvement in Predictive Modeling, März 2026, Chemnitz

CSR-26-04 Atul Chandra Nath, Reda Harradi, Wolfram Hardt, GPS-based UAV Precision Landing, April 2026, Chemnitz

CSR-26-05 Josey Mol Sibi, Batbayar Battseren, Wolfram Hardt, Real Time Multi Stream Video Transmission in Autonomous UAV, Mai 2026, Chemnitz

CSR-26-06 Silpa Geetha Harshakumar, Batbayar Battseren, Wolfram Hardt, Simultaneous Video Streaming and Recording with Geotagging in Autonomous UAV, Mai 2026, Chemnitz

CSR-26-07 Monica Chambial, Batbayar Battseren, Wolfram Hardt, Implementation and Enhancement of a Mission Scheduling Component for Autonomous UAV, Mai 2026, Chemnitz

CSR-26-08 Ganapathi Teja Chirravuri, Batbayar Battseren, Wolfram Hardt, Dynamic Model Switching for Real-Time UAV Inspection, Oktober 2026, Chemnitz

CSR-26-09 Hariharan Chandrasekaran, Batbayar Battseren, Wolfram Hardt, LLM-Powered Novel Scenario Extraction and Reasoning in Autonomous Driving, Oktober 2026, Chemnitz

CSR-26-10 Anh Tu Nguyen, Julkar Nine, Wolfram Hardt, Integration of Deep learning Algorithms in a Cloud Server-Based Carla Simulation Environment, November 2026, Chemnitz

Chemnitzer Informatik-Berichte

ISSN 0947-5125

Herausgeber: Fakultät für Informatik, TU Chemnitz
Straße der Nationen 62, D-09111 Chemnitz