Architecture Overview for Self-driving Car

# High level components

1. **ROS 2 Framework**: Modular nodes for sensor processing, fusion, control, and actuation.
2. **Simulation**: Train in Gazebo/ROS 2 for safe, scalable RL training.
3. **Reinforcement Learning (RL)**: Replace regression with RL for dynamic decision-making.
4. **Sensor Fusion**: Combine LiDAR and camera data via separate ML models.
5. **Controller Unit**: ML-based decision node to fuse predictions and send actuator commands.

**Step-by-Step Implementation**

**1. Simulation Setup with ROS 2 and Gazebo**

* **Tools**:
  + **ROS 2 Humble** (optimized for Raspberry Pi 5).
  + **Gazebo Fortress** (ROS 2-compatible simulator).
  + **TurtleBot4** or **AWS DeepRacer** as a simulated car model.
* **Workflow**:
  1. Model the car, LiDAR, and camera in Gazebo.
  2. Simulate environments (tracks, obstacles) for training.
  3. Integrate ROS 2 nodes to mirror real-world hardware.

**2. Reinforcement Learning (RL) Training**

* **Environment Design**:
  + **State Space**: LiDAR point cloud + camera frames (resized to 64x64 RGB) + IMU data.
  + **Action Space**: Continuous outputs for speed (0–1) and steering angle (-30° to +30°).
  + **Reward Function**:
    - +10 for forward progress.
    - -100 for collisions.
    - -5 for near misses (LiDAR detects obstacles < 30cm).
    - -2 for excessive steering (smoothness penalty).
* **Algorithm**: Use **PPO (Stable Baselines3)** for continuous action spaces.
* **Training Pipeline**:
  + Train in simulation until the agent achieves > 80% success rate on test tracks.
  + Transfer learning to the physical car with domain adaptation.

**3. Sensor Fusion with LiDAR and Camera**

* **LiDAR Model**:
  + **Task**: Obstacle distance prediction (0–5m).
  + **Model**: Lightweight CNN (e.g., **MobileNetV3** with regression head).
  + **Input**: 360° LiDAR scan (1D array of 360 distance values).
  + **Output**: Closest obstacle distance in 8 angular sectors (e.g., 45° each).
* **Camera Model**:
  + **Task**: Object detection (cars, pedestrians) and lane segmentation.
  + **Model**: **YOLOv8n** (Nano version) for real-time inference.
  + **Output**: Bounding boxes + lane boundaries (binary mask).
* **Fusion Strategy**:
  + **ROS 2 Node**: Subscribes to LiDAR and camera topics.
  + **Fusion Logic**:
    - If LiDAR detects obstacles < 50cm, prioritize stopping.
    - If camera detects lane drift, adjust steering.
    - Use **confidence scores** from both models to resolve conflicts.

**4. Controller Unit (ML Decision Node)**

* **Model**: Small neural network (2-layer MLP) for real-time inference.
  + **Input**:
    - LiDAR sector distances (8 values).
    - Camera lane deviation score (0–1) + object proximity (0–1).
  + **Output**: Speed (0–1) and steering angle (-1 to +1).
* **Training**:
  + Use behavioral cloning from simulation data (supervised learning).
  + Refine with RL for edge cases (e.g., overtaking).

**5. ROS 2 Node Structure**

* **Nodes**:
  1. **/lidar\_node**: Processes RPLIDAR scans, runs obstacle detection model.
  2. **/camera\_node**: Captures images, runs YOLOv8n and lane detection.
  3. **/fusion\_node**: Aggregates LiDAR/camera predictions into a custom message.
  4. **/rl\_controller**: Runs the RL policy or MLP controller, sends motor commands.
  5. **/motor\_driver**: Converts speed/steering to PWM signals for L298N.
* **Custom Messages**:
  1. SensorFusion.msg: float32[8] lidar\_distances, float32 lane\_deviation, float32 object\_risk.

**6. Real-Time Optimization**

* **Raspberry Pi Tweaks**:
  + Overclock CPU to 2.0 GHz.
  + Use **TensorFlow Lite** or **ONNX Runtime** for model inference.
  + Assign CPU cores to critical nodes (e.g., taskset for /rl\_controller).
* **Edge TPU Acceleration**: Offload YOLOv8n to Coral USB Accelerator.

**7. Testing & Deployment**

1. **Simulation Validation**: Test RL policy in varied Gazebo environments.
2. **Hardware-in-the-Loop (HIL)**: Connect ROS 2 to real sensors/motors while using simulated LiDAR/camera.
3. **Field Testing**:
   * Start in controlled environments (e.g., empty parking lot).
   * Gradually introduce dynamic obstacles (e.g., moving boxes).

**ROS 2 Custom Message**:

# SensorFusion.msg

float32[8] lidar\_distances

float32 lane\_deviation

float32 object\_risk

**RL Controller Node**:

import rclpy

from rclpy.node import Node

from sensor\_fusion.msg import SensorFusion

from stable\_baselines3 import PPO

class RLController(Node):

def \_\_init\_\_(self):

super().\_\_init\_\_('rl\_controller')

self.sub = self.create\_subscription(SensorFusion, '/fusion\_data', self.callback, 10)

self.model = PPO.load("rl\_policy\_sim.zip") # Load trained policy

def callback(self, msg):

state = np.concatenate([msg.lidar\_distances, [msg.lane\_deviation, msg.object\_risk]])

action, \_ = self.model.predict(state)

self.publish\_motor\_command(action)

**Sensor Fusion Logic**:

def fuse\_data(lidar\_distances, camera\_data):

if np.min(lidar\_distances) < 0.3: # 30cm threshold

return (0.0, 0.0) # Stop immediately

else:

# Blend steering based on lane deviation

steer = -0.5 \* camera\_data.lane\_deviation

speed = 0.7 if camera\_data.object\_risk < 0.2 else 0.3

return (speed, steer)

**Challenges & Mitigation**

* **Latency**: Use ROS 2 QoS settings to prioritize motor commands.
* **Sim-to-Real Gap**: Add noise to simulation LiDAR/camera data during training.
* **Compute Limits**: Offload inference to a secondary Pi or edge device.

**Resources**

* **ROS 2 Tutorials**: [ROS 2 Documentation](https://docs.ros.org/en/humble/)
* **Gazebo Setup**: [Gazebo ROS 2 Plugins](https://gazebosim.org/docs/fortress/ros2_installation)
* **RL Training**: [Stable Baselines3 Guide](https://stable-baselines3.readthedocs.io/)