



Development Of An Advanced Road Safety System Based On V2V And V2C Technologies

Team Members

**Alaa Gaber Hamada Ahmed
Ali AbdelMenam Mohamed
Bassant Ehab Saber Esmail
Toqa Sameh Salah Swilam
Rawan Mohamed Fathy Mohamed
Mariam Reda Ibrahim Kamel
Nermeen Ahmed Fouad Ali**

Project Advisor

Dr. Essam Mostafa Ibrahim Aloleimi

Presented to

**Electrical Engineering Department
Computer Systems Engineering**

2024-2025

Faculty Of Engineering At Shoubra

Benha University

ELECTRICAL ENGINEERING DEPARTMENT

COMPUTER SYSTEMS ENGINEERING

GRADUATION PROJECT

SPONSORED AND MENTORED BY



Development of an advanced road safety system based on V2V and V2C technologies



ACKNOWLEDGEMENT

We would like to express our sincere gratitude and deepest appreciation to those who supported and guided us throughout the journey of our graduation project.

First and foremost, we would like to extend our heartfelt thanks to **Dr. Essam Aboleimi**, our project supervisor, for his continuous support, expert guidance, and valuable insights. His encouragement and mentorship played a pivotal role in shaping the direction and success of this work.

We are also especially grateful to **Eng. Abdelrahman El Dessouky** for his outstanding technical support and mentorship through the **Made in Egypt (MIE)** initiative. His experience, patience, and dedication greatly enhanced the quality of our project, both in theory and practice.

Special thanks also go to **Eng. Mohamed Khaled** from Brightskies for his valuable guidance and support. His input and encouragement were instrumental in refining our ideas and pushing us toward real innovation.

We would also like to sincerely thank the **Brightskies** team and the **Egypt IoT** League for providing us with the platform, tools, and motivation to transform our ideas into real-world applications

Their belief in student innovation and support for embedded systems research has truly inspired us.

This project wouldn't have reached this level without the collaborative spirit, technical exposure, and continuous motivation we received from everyone mentioned above.

To all those who believed in us — **thank you** for being a part of our journey.

ABSTRACT

Traditional methods of monitoring road conditions are often slow and inefficient, relying on periodic inspections or driver reports. This reactive approach results in delayed responses to hazardous conditions. As urban traffic increases, there is a pressing need for innovative solutions that harness technology to improve road safety and maintenance.

This project proposes the development of a comprehensive road damage detection system that utilizes Vehicle-to-Vehicle (V2V) and Vehicle-to-Cloud (V2C) communication and integrates with OpenStreetMap (OSM) to display real-time geolocated damage reports on an interactive map interface, combined with the YOLOv12 object detection model. Smart vehicles equipped with cameras will use YOLOv12 to identify and classify road damage, including cracks, manholes, faded road markings, potholes, and garbage on street in real-time. This information will be shared among vehicles via V2V communication, sent to the cloud to generate a report about road damages, and then forwarded to government agencies to accelerate road repair operations via V2C communication. These road damages will also be visualized on an interactive map interface and summarizing the project's achievements and potential impact:

Accurate Road Damage Detection: The YOLOv12 model, trained on a diverse dataset, demonstrates high accuracy in detecting and classifying different types of road damage, such as cracks, manholes, faded road markings, potholes, and garbage on street.

Real-Time Data Sharing: The integration of communication framework enables real-time communication between vehicles and government agencies, allowing for immediate sharing of road damage information,

Enhanced Road Safety: By identifying and reporting road damage promptly, the system can help prevent accidents and improve road safety for drivers and pedestrians

Cost-Effective Maintenance: Early detection of road damage can lead to more efficient and cost-effective maintenance.

TABLE OF CONTENTS

Contents

CHAPTER 1	10
INTRODUCTION	10
1.1 INTRODUCTION	11
1.2 PROBLEM STATEMENT	11
1.3 STATISTICS	12
1.4 PROJECT OBJECTIVES	14
1.5 PROPOSED SOLUTION	14
CHAPTER 2	17
LITERATURE REVIEW	17
2.1 INTRODUCTION	18
2.2 REVIEW BODY	19
2.3 CONCLUSION	37
CHAPTER 3	39
SOFTWARE DESIGN AND COMPUTER VISION	39
3.1 INTRODUCTION TO COMPUTER VISION	40
3.2 HOW COMPUTER VISION WORKS	40
3.2.1 IMAGE ACQUISITION	40
3.2.2 PREPROCESSING	40
3.2.3. FEATURE EXTRACTION	41
3.2.4. INTERPRETATION AND DECISION-MAKING	41
3.3 KEY TECHNIQUES IN COMPUTER VISION	42
3.3.1. OBJECT DETECTION	42
3.4 OBJECT DETECTION	42
3.4.1 KEY COMPONENTS OF OBJECT DETECTION	42
3.4.2 TECHNIQUES FOR OBJECT DETECTION:	43
3.5 ROAD HAZARD DETECTION	44
3.6 YOU ONLY LOOK ONCE (YOLO)	44
3.6.1 WHY YOLO FOR ROAD SAFETY?	44
3.6.2 How YOLO WORKS?	45

3.6.3 WHY YOLOv12 WAS CHOSEN FOR THIS PROJECT?	46
3.7 DATASETS.....	47
3.7.1 ROAD DAMAGE DETECTOR RDD2022	47
<i>RDD2022: The Multi-National Road Damage Dataset 2022</i>	48
3.7.2 PREPROCESSING	49
3.7.3 ROAD DAMAGE DATASET FROM ROBOFLOW	52
3.7.4 PAVEMENT DATASET	53
3.7.5 DATASET PREPARATION	54
3.8 MODEL TRAINING	55
3.9 MODEL RESULTS.....	57
3.9.2 BENCHMARK DATASET	58
3.10 OBJECT TRACKING	59
3.10.1 BYTE TRACK: STATE-OF-THE-ART MULTI-OBJECT TRACKING.....	59
3.10.2 IMPLEMENTATION IN THIS PROJECT	59
3.11 MODEL QUANTIZATION	60
3.11.1 WHAT IS MODEL QUANTIZATION?	60
3.11.2 WHY QUANTIZATION IS NEEDED FOR RASPBERRY PI 5	61
3.11.3 MODEL PERFORMANCE AND QUANTIZATION COMPARISON - RPI 5	62
3.11.4 IMPLEMENTATION AND FORMAT CONVERSION.....	63
3.12 MODEL DEPLOYMENT	64
3.12.1 COMPARISON BETWEEN RASPBERRY PI 5 AND JETSON	64
3.13 AI OUTPUT INTEGRATION	66
CHAPTER 4	67
EMBEDDED LINUX	67
4.1 WHAT IS EMBEDDED LINUX?	68
4.2 EMBEDDED LINUX ARCHITECTURE	69
4.2.1 Bootloader	69
4.2.2 Kernel	70
4.2.3 Root filesystem	70
4.2.4 Services	72
4.2.5 Application/Program	72
4.3 EMBEDDED LINUX VS. DESKTOP LINUX.....	73
4.4 EMBEDDED LINUX VS. BARE METAL VS. RTOS.....	73

4.5 INTRODUCTION TO EMBEDDED LINUX DEVELOPMENT	74
4.5.1 SD card structure in Embedded Linux	74
4.5.2 The Traditional Method for building a Customized Linux Image	75
4.5.3 Using Yocto to build The Image.....	80
4.6 INSTALLING YOCTO AND BUILDING THE IMAGE.....	87
4.6.1 Setting up the Yocto Environment	87
4.6.2 Initialize the Build Environment	88
4.6.3 Creating our own Layer	96
4.6.4 Creating our own Linux Distribution	98
4.6.5 Configuring our Layer	99
4.7 BITBAKE.....	100
4.7.1 Introduction	100
4.7.2 Concepts.....	102
4.7.3 The BitBake Command.....	105
4.7.4 Execution.....	105
4.7.5 Parsing the Base Configuration Metadata	105
4.7.6 Locating and Parsing Recipes.....	107
4.7.7 Dependencies.....	108
CHAPTER 5.....	109
COMMUNICATION AND INTEGRATION SYSTEMS.....	109
5.1 INTRODUCTION.....	110
5.2 VEHICLE-TO-VEHICLE (V2V) COMMUNICATION	110
5.2.1 ESP-NOW & ESP32 Integration.....	111
5.2.2 V2V Alert Scenarios.....	112
5.3 VEHICLE-TO-CLOUD (V2C) COMMUNICATION	112
5.3.1 Cloud Platforms and APIs.....	112
5.3.2 Integration with Government Agencies.....	113
5.3.3 MQTT Protocol	114
5.3.4 AWS iot core broker.....	115
5.4 OPENSTREETMAP INTEGRATION.....	117
5.4.1 Live Road Condition Updates	117
5.4.2 Visualization for Users.....	118
CHAPTER 6.....	120
HARDWARE AND	120
SOFTWARE DESIGN.....	120
6.1 INTRODUCTION.....	121
6.2 HARDWARE OVERVIEW	122

6.2.1	<i>Raspberry Pi</i>	122
6.2.2	<i>Raspberry Pi Camera</i>	129
6.2.3	<i>ESP32 Modules</i>	133
6.2.4	<i>Arduino Uno</i>	139
6.2.5	<i>GPS Module</i>	144
6.2.6	<i>LCD Display</i>	147
6.2.7	<i>RC Car Setup</i>	148
6.3	SOFTWARE	154
6.3.1	<i>ESP-NOW</i>	154
6.3.2	<i>Flask Web Application for Road Damage Reporting and Visualization</i>	155
6.3.3	<i>MQTT</i>	159
6.3.4	<i>Uart</i>	163
CHAPTER 7		168
CONCLUSION AND		168
FUTURE WORK		168
6.1	CONCLUSION	169
6.2	SYSTEM PERFORMANCE SUMMARY	169
6.3	BENEFITS TO SMART , NON-SMART VEHICLES AND GOVERNMENT AGENCIES	170
6.4	FUTURE ENHANCEMENTS	170
6.4.1	<i>Integration with Emergency Services</i>	170
6.4.2	<i>Expanded Dataset and Multi-Country Deployment:</i>	170
6.4.3	<i>Implement Vehicle Action Response:</i>	170
REFERENCES		172

LIST OF FIGURES

Fig.1	Number of deaths due to open manhole between the years 2015 and 2019	13
Fig.2	impact of repair delays on user costs and agency costs	13
Fig.3	Proposed Solution	15
Fig.4	Techniques for Object Detection	43
Fig.5	How YOLO Works	46
Fig.6	Preprocessing.....	49
Fig.7	Dataset.....	50
Fig.8	Longitudinal	50
Fig.9	Alligator	50
Fig.10	Pothole.....	51
Fig.11	Transverse Crack.....	51
Fig.12	Other Corruption.....	51
Fig.14	Road Damage Dataset from Roboflow.....	52
Fig.13	Road Damage Dataset from Roboflow.....	52
Fig.15	Pavement Dataset 2.....	53
Fig.16	Pavement Dataset 1.....	53
Fig.17	Model training.....	56
Fig.18	Model training2.....	56
Figure 19	MODEL PERFORMANCE AND QUANTIZATION COMPARISON -CPU	Error!
Bookmark not defined.		
Figure 20	MODEL PERFORMANCE AND QUANTIZATION COMPARISON - RPI 5.....	62
Fig.21	Model Deployment	64
Fig.22	Comparison between Raspberry pi 5 and jetson	65
Fig.23	Comparison between Raspberry pi 5 and jetson.	65
Figure 24	output log.....	66
Fig.25	Embedded Linux Architecture	69
Fig.26	Yocto Project	80
Fig.27	Yocto Project Environment	82
Fig.28	Poky	82
Fig.29	Poky File Structure	86
Fig.30	Poky Installation Files	87
Fig.31	bblayer.conf file.....	89

Fig.32	local.conf file	91
Fig.33	local.conf file	92
Fig.34	Content of our meta-ai layer	96
Fig.35	Content of our meta-grad-distro layer	97
Fig.36	Content of our meta-apps layer	97
Fig.37	Bitbake	101
Fig.38	Our image	101
Fig.39	V2V	110
Fig.40	ESP-NOW	111
Fig.41	V2C	113
Fig.42	MQTT Overview	115
Fig.43	OSM	118
Fig.44	OSM map	119
Fig.45	System Architecture	121
Fig.46	Raspberry pi 5 specifications	125
Fig.47	Raspberry pi 5 GPIO	126
Fig.48	Raspberry pi camera	129
Fig.49	Schematic of Raspberry pi camera	130
Fig.50	Raspberry pi 5 with Camera	131
Fig.51	ESP-32 BOARD	134
Fig.52	ESP32 GPIO COMPONENTS	136
Fig.53	ARDUINO UNO BOARD	139
Fig.54	ARDUINO UNO COMPONENTS	142
Fig.55	GPS NEO-6M	144
Fig.56	Raspberry Pi Connected to GPS Module	146
Fig.57	LCD with ESP32	147
Fig.58	RC Car	148
Fig.59	RC Car Connection Diagram	148
Fig.60	HC-05 Bluetooth Module	150
Fig.61	Battery Operated motors	151
Fig.62	L298N Motor Driver	151
Fig.63	Ultrasonic Sensor	153

LIST OF TABELS

Table.1	MODEL TRAINING	56
Table.2	MODEL RESULT 1	56
Table.3	MODEL RESULT 2	56
Table.3	Embedded Linux vs. Desktop Linux	73
Table.4	Embedded Linux vs. Bare metal vs. RTOS	73
Table.5	Raspberry Pi Camera Specification	130
Table.6	GPS NEO-7M Parameters	146

CHAPTER 1

INTRODUCTION



1.1 Introduction

Roads are an essential part of our daily lives, and as a result, their condition can significantly impact the safety and efficiency of travel. Damaged roads, characterized by cracks, manholes, faded road markings, potholes, garbage on street and lack of proper signs, pose a substantial risk to drivers, cyclists, and pedestrians alike. The issue of responsibility for accidents on these damaged roads is complex because there is an interplay of maintenance by the municipalities, driver awareness, as well as the condition of the vehicle. [1]

1.2 Problem Statement

Damage to roads can significantly contribute to accidents in several ways:

- cracks, manholes, faded road markings, potholes, and garbage on street can cause drivers to lose control of their vehicles, especially at high speeds. This can lead to swerving, and collisions with other vehicles or objects.
- Damaged roads can reduce the traction between the tire and the road surface, increasing the risk of accidents. This is particularly hazardous in wet or icy conditions, where the grip on the road is already compromised.
- Damage such as cracks, manholes, faded road markings, potholes, and garbage on street can create unexpected obstacles for drivers. Swerving to avoid these can lead to accidents, particularly if the maneuver is sudden or if it takes the vehicle into another lane.
- Consistent driving on damaged roads can lead to increased wear and tear on vehicles, potentially causing mechanical failures that can result in accidents. For example, a damaged suspension system or tire blowouts can make a vehicle difficult to control.
- Damage to the road surface can also affect the braking distance of a vehicle. If a driver is not aware of the road's condition, they may not brake in time to avoid an obstacle or a sudden stop in traffic.

In order to reduce the risks of accidents occurring due to road conditions, regular road maintenance and prompt repairs of any damage are crucial. Additionally, drivers should be aware of the condition of the roads they are using and adjust their driving style accordingly. [1]

1.3 Statistics

As we navigate the statistical landscape, the data underscores the significance of our advanced road safety project, highlighting the challenges posed by existing communication gaps between vehicles, road infrastructure, and emergency response systems. Here are the statistics we have gathered:

- The Independent Transport Commission found that pothole-related incidents cause about 1% of all road accidents. For motorcyclists and cyclists, the risks are even higher. A survey by Cycling UK revealed that 31% of its members had been involved in accidents or near misses due to poor road surfaces, including potholes. [2]
- In the most tragic cases, potholes have been directly linked to fatalities on the road. In the UK, the Road Angel organization warns that the risk to life from potholes is now severe, with nearly 30,000 people killed or seriously injured on UK roads last year. [3]
- According to International Journal of Engineering and Technology Innovation, road repair and maintenance actions are often delayed due to a lack of awareness of road users and agencies, even though these actions can cause damage to a section of the road to worsen and can directly

increase the economic losses experienced by road users which is clearly depicted in fig.10.[34]

- According to the article released by Times of India on September 11-2020, the number of persons died by accidental fall due to improper closing of manhole has not reduced over the last 5 years which is clearly depicted in Fig. 1.1.

According to the survey report of NCRB (National Crime Record Bureau), the number of persons died due to accident caused by open manhole is found to be 102 which are clearly explained in Fig.1.2 [35]

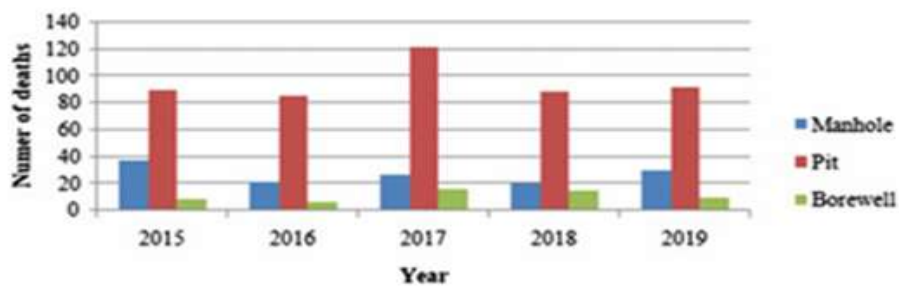


Fig.1 Number of deaths due to open manhole between the years 2015 and 2019

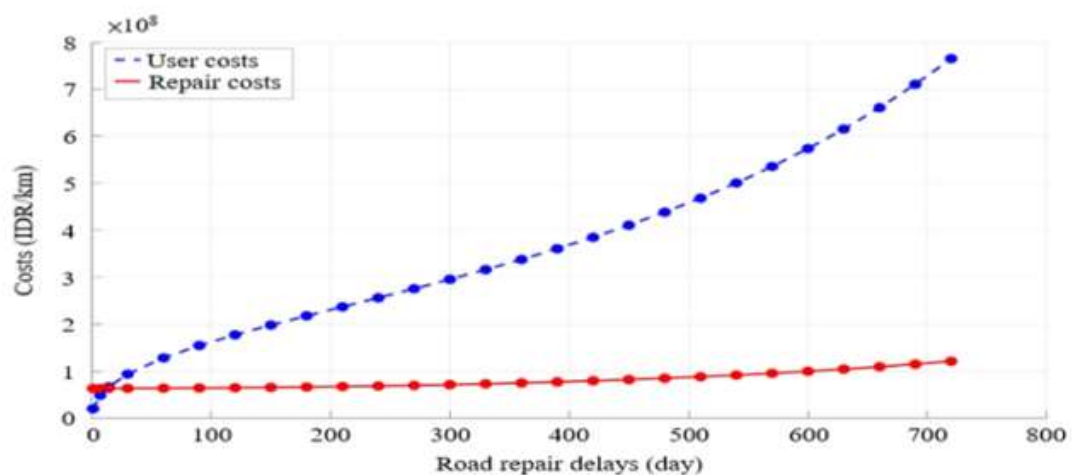


Fig.2 impact of repair delays on user costs and agency costs

1.4 Project Objectives

In this project, we developed a comprehensive system for road damage detection using advanced technologies such as Camera and the YOLOv12 model, integrated with communication frameworks like V2V and V2C, this system will ensure:

- **Improved Safety:** By alerting vehicles about road hazards in real-time, the system helps prevent accidents and enhances overall road safety.
- **Efficient Maintenance:** Real-time data transmission to government agencies allows for prompt assessment and prioritization of maintenance tasks, reducing response times for repairs.
- **Data-Driven Infrastructure Management:** The project provides valuable data to local governments, improving planning and resource allocation for road maintenance initiatives.
- **Enhanced Communication:** V2V and V2C technologies facilitate seamless information exchange between vehicles and infrastructure, creating a more responsive and integrated transport system.

1.5 Proposed Solution:

The Advanced Road Safety Project leverages cutting-edge technologies, including computer vision, machine learning, and advanced communication protocols such as Vehicle-to-Vehicle (V2V) and Vehicle-to-Cloud (V2C). The system utilizes high-resolution images captured from vehicles equipped with cameras to identify road

surface damage such as cracks, manholes, faded road markings, potholes, and garbage on the street. Detected issues are first communicated to nearby vehicles via V2V to alert them about the road damage. Additionally, the system communicates with government agencies to accelerate repair responses by municipal authorities, then to visualize the detected road issues, the system integrates with OpenStreetMap (OSM)

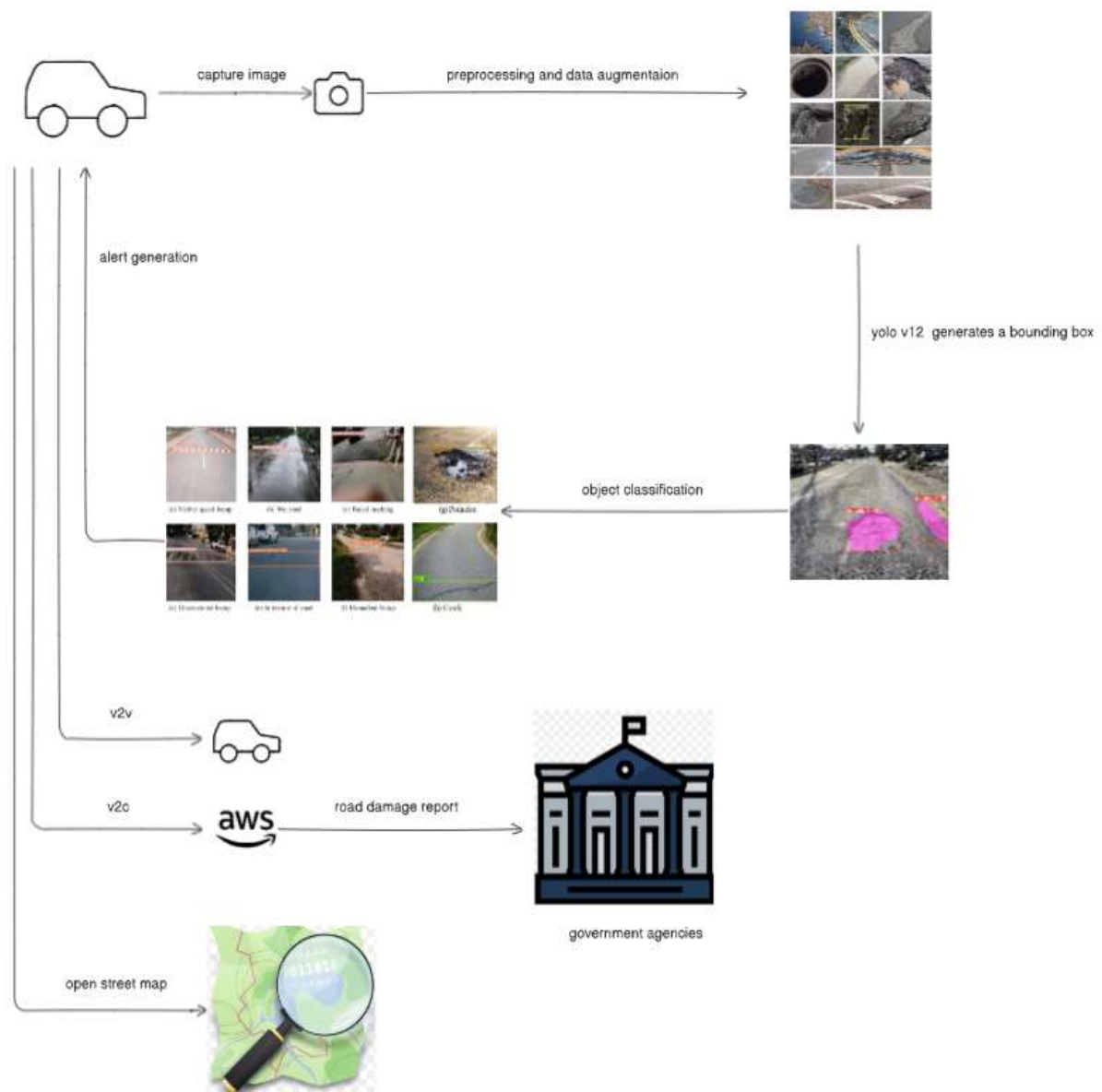


Fig.3 Proposed Solution

This project not only addresses immediate safety concerns but also contributes to more effective urban planning and resource allocation for road maintenance. By fostering a safer driving environment and promoting proactive infrastructure management, the Road Damage Detection Project paves the way for smarter, more resilient transportation systems.

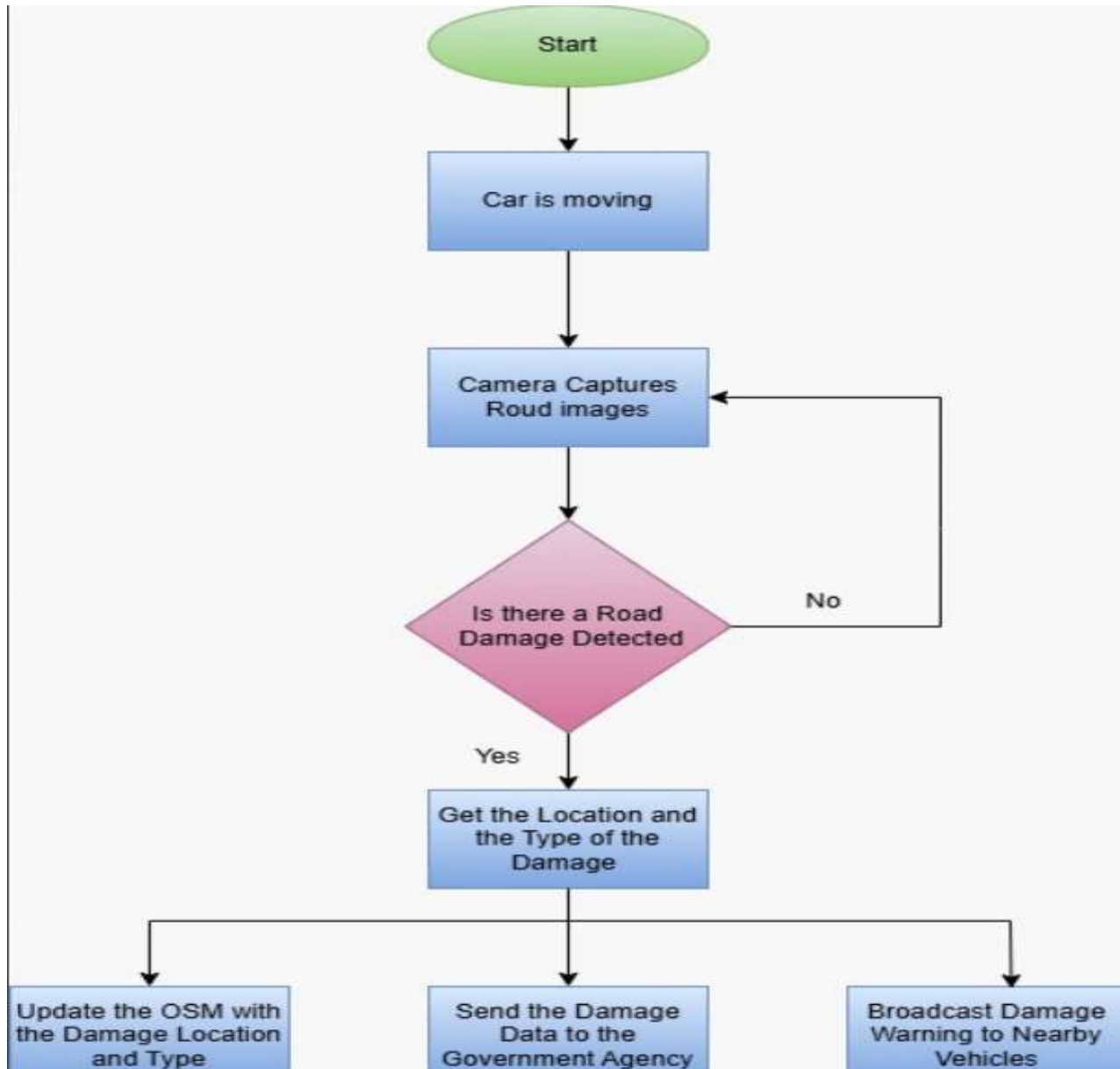


Fig.64 flow diagram

CHAPTER 2

LITERATURE REVIEW



2.1 Introduction

In modern transportation systems, maintaining road infrastructure is crucial for ensuring vehicle safety, reducing accident risks, and optimizing traffic flow. However, road damage, such as cracks, manholes, faded road markings, potholes, and garbage on the street, poses significant challenges to both drivers and transportation authorities. The absence of a real-time, automated detection system leads to delayed maintenance, increased vehicle damage, and a higher likelihood of accidents.

Traditional road inspection methods rely heavily on manual surveys, which are time-consuming, labor-intensive, and prone to human error. As road networks continue to expand, these methods become increasingly inefficient. To address this issue, advanced technologies such as artificial intelligence, vehicle-to-vehicle (V2V) communication, vehicle-to-cloud (V2C) and integrating with open street map offer promising solutions for automated road damage detection and reporting.

This literature review explores key research themes integral to enhancing the efficiency of road damage detection in real-world driving environments. It focuses on critical components such as deep learning-based detection systems (particularly YOLOv12), the role of V2V communication in sharing road hazard information, and the integration of C-V2X technology for cloud-based reporting to government agencies.

The purpose of this literature review is to thoroughly examine and synthesize existing research on these topics, identifying trends, gaps, and potential improvements. By analyzing the advancements in computer vision, V2X communication, road hazard warning systems, and cloud-based reporting, the review aims to extract key insights that will contribute to the development of an integrated system for real-time road damage detection and alert dissemination.

Such a system would not only enhance road safety by providing timely warnings to vehicles but also facilitate efficient maintenance planning by enabling government agencies to act swiftly on reported road conditions. Ultimately, the integration of AI-driven detection, V2V communication, and cloud-based infrastructure monitoring will pave the way for a more intelligent and responsive road management system in urban and highway environments.

2.2 Review Body

Paper 1:

Title: A Fused Deep Learning Expert System for Road Damage Detection and Size Analysis Using YOLO9tr and Depth Estimation

Date: 2024

conference/Event: AI Research Group, Department of Civil Engineering Faculty of Engineering, King Mongkut's University of Technology Thonburi Bangkok, Thailand

This paper introduces a fused deep learning system for automated road damage detection and size analysis, combining the YOLO9tr object detection model with a novel grayscale-based depth estimation technique. YOLO9tr enables real-time identification of various road damage types such as cracks, potholes, and surface deterioration, while the depth estimation module extracts dimensional

information from standard grayscale images using an innovative calibration approach. This fusion eliminates the need for expensive depth sensors or stereo cameras, making the system cost-effective and deployable with conventional imaging equipment.

The system leverages pre-calibrated grayscale depth mapping to convert pixel-based measurements into real-world dimensions, allowing precise quantification of damage size. With a calibration range spanning 3.5 to 7.0 meters, it achieves consistent measurement accuracy across diverse road conditions. Experimental results demonstrate a Mean Absolute Percentage Error (MAPE) of 19.44% in damage size estimation, outperforming existing methods. Despite a slight tendency to overestimate damage areas due to conservative annotation practices, the model achieves robust performance in both detection and dimensional analysis.

The study highlights the practical advantages of the system, including its ability to operate in real-time, accurately measure damage size, and function in varying environments without specialized hardware. The approach significantly advances road condition monitoring by integrating advanced object detection and depth estimation techniques, providing a scalable and efficient tool for infrastructure maintenance. Future research

directions include addressing overestimation biases, refining detection accuracy for linear defects, and extending the system's applicability to non-planar surfaces. [4]

Paper 2:

Title: RDD-YOLO: Road Damage Detection Algorithm Based on Improved You Only Look Once Version 8

Date: 2024

conference/Event: School of Computer Science and Technology, Dong Hua University

This research introduces RDD-YOLO, an improved version of the YOLOv8 algorithm for detecting road damage. The model integrates a simple attention mechanism (SimAM) to enhance focus on critical image features, GhostConv for reduced computational complexity, and bilinear interpolation to improve image detail during up-sampling. Evaluated on the RDD2022 dataset, RDD-YOLO demonstrates improved accuracy, achieving an mAP50 of 62.5% and an F1 score of 69.6%,

outperforming baseline models. This method addresses challenges like diverse damage types and environmental variations, offering a precise and efficient solution for road maintenance and traffic safety. [5]

Paper 3:

Title: Road Damage Detection for Autonomous Driving Vehicles using YOLOv8 and Salp Swarm Algorithm

Date: 2024

conference/Event: Faculty of Computing, Universiti Malaysia Pahang Al-Sultan Abdullah

This study presents an improved road damage detection system for autonomous vehicles by integrating the YOLOv8 object detection model with the Salp Swarm Algorithm (SSA) for hyperparameter optimization. The method enhances the model's performance in identifying four types of road damage—longitudinal cracks, transverse cracks, alligator cracks, and potholes—using the Czech subset of the RDD2022 dataset. By optimizing key parameters like learning rate, momentum, and optimizer selection, SSA-YOLOv8 achieves a 3.5% improvement in mAP@50 compared to the default YOLOv8n, as well as superior precision and recall metrics. This automated approach simplifies tuning while enhancing real-time performance, making it highly effective for autonomous driving and road safety applications. [6]

Paper 4:

Title: A Lightweight Method for Road Damage Detection Based on Improved YOLOv8n

Date: 2024

conference/Event: Intelligent Construction Internet of Things Application Technology Key Laboratory of Liaoning Province, China

This study introduces BSE-YOLO, a lightweight road damage detection model designed to operate efficiently on mobile edge devices. Built on the YOLOv8n framework, the model incorporates several enhancements to balance accuracy and computational efficiency. Key improvements include the BiFPN feature fusion network for better feature extraction, the SC2f module for reducing parameter count and computational overhead, and the SEAHead detection module for enhanced localization accuracy and small object detection. Evaluated on the RDD2022 dataset, BSE-YOLO achieves a 40%

reduction in parameters, 2.2 GFLOPs less computation, and 3 additional FPS, with only a minimal 0.1% decline in mAP@0.5 compared to the original YOLOv8n. These advancements make BSE-YOLO an effective solution for real-time road damage detection, particularly on resource-constrained devices. [7]

Paper 5:

Title: Improved Road Damage Detection Algorithm Based on YOLOv8

Date: 2024

conference/Event: Intelligent Construction Internet of Things Application Technology Key Laboratory of Liaoning Province, China

This study introduces BSE-YOLO, a lightweight road damage detection model designed to operate efficiently on mobile edge devices. Built on the YOLOv8n framework, the model incorporates several enhancements to balance accuracy and computational efficiency. Key improvements include the BiFPN feature fusion network for better feature extraction, the SC2f module for reducing parameter count and computational overhead, and the SEAHead detection module for enhanced localization accuracy and small object detection. Evaluated on the RDD2022 dataset, BSE-YOLO achieves a 40% reduction in parameters, 2.2 GFLOPs less computation, and 3 additional FPS, with only a minimal 0.1% decline in mAP@0.5 compared to the original YOLOv8n. These advancements make BSE-YOLO an effective solution for real-time road damage detection, particularly on resource-constrained devices. [8]

Paper 6:

Title: YOLOv8-PD: an improved road damage detection algorithm based on YOLOv8n model

Date: 2024

conference/Event: College of information and Network Safety, People's Public Security University of China

The study proposes YOLOv8-PD, an enhanced lightweight algorithm specifically designed for road damage detection, building upon the YOLOv8n framework. The model incorporates four key improvements: the BOT Transformer module, which captures long-range dependencies and global features; the Large Separable Kernel Attention (LSKA) mechanism, enhancing the detection of complex cracks while reducing computational demands; the C2fGhost block in the neck network,

improving feature extraction and reducing computational costs; and the LSCD-Head, a lightweight detection head module that decreases model parameters while enhancing multi-scale detection accuracy. Extensive testing on the RDD2022 dataset demonstrated a 1.4% improvement in mAP50, with a 27.6% reduction in parameters and a 25% reduction in computational complexity, making the model efficient for real-time applications. The algorithm also showed strong generalization on the RoadDamage dataset, achieving a 4.1% mAP50 improvement, particularly excelling at detecting longitudinal cracks. Despite challenges with small targets like potholes, the YOLOv8-PD model outperformed baseline models in both detection accuracy and computational efficiency, making it suitable for deployment on resource-constrained devices and practical scenarios. [9]

Paper 7:

Title: An Efficient Approach to Monocular Depth Estimation for Autonomous Vehicle Perception Systems

Date: 2023

conference/Event: Department of Computer Science and Engineering, Sharif University of Technology, Tehran

The paper presents a novel and efficient approach to monocular depth estimation designed specifically for autonomous vehicle perception systems, addressing the high costs and complexity associated with LiDAR and other sensor-based methods. By utilizing RGB cameras, the research proposes a two-stage framework: first, the YOLOv7 algorithm is employed to detect vehicles and their front and rear lights; second, a nonlinear model maps these detections to estimate radial depth. The integration of a Convolutional Block Attention Module (CBAM) within the YOLOv7 architecture enhances detection precision by improving feature extraction capabilities. The methodology was tested using simulations in the CARLA environment and evaluated on real-world datasets, such as KITTI, showcasing a strong balance between accuracy and processing speed, with a mean squared error (MSE) of 0.1. This approach demonstrated superior performance compared to existing models, particularly in mid-range distance estimations, while maintaining cost efficiency and scalability by leveraging affordable monocular cameras. The results highlight the model's robustness in various lighting and weather conditions, as well as its reliability in complex real-world traffic scenarios, offering a practical solution for advancing autonomous vehicle safety and collision avoidance systems. [10]

Paper 8:

Title: A Dual Attention and Partially Overparameterized Network for Real-Time Road Damage Detection

Date: 2024

conference/Event: _

This study presents DAPONet, a novel deep learning model designed for real-time road damage detection using street view image data (SVRDD). Traditional road damage detection methods, such as manual inspections and sensor-mounted vehicles, are inefficient, costly, and often inaccurate, particularly for minor damages. To address these challenges, DAPONet incorporates three key innovations: a dual attention mechanism, a multi-scale partial over-parameterization module, and an efficient down sampling module. These components enhance feature extraction, improve detection accuracy, and optimize computational efficiency. Experimental results on the SVRDD dataset show that DAPONet achieves an mAP50 of 70.1%, surpassing YOLOv10n by 10.4%, while significantly reducing model parameters and computational complexity. Additionally, on the MS COCO2017 validation dataset, DAPONet outperforms EfficientDet-D1 with an mAP50-95 of 33.4%, despite having 74% fewer parameters and FLOPs. These findings demonstrate DAPONet's effectiveness in accurately detecting road damages across various scales while maintaining low computational costs, making it highly suitable for real-world deployment in transportation infrastructure maintenance and smart city applications. [12]

Paper 9:

Title: iRodd (intelligent-road damage detection) for real-time infrastructure preservation in detection, classification, calculation, and visualization

Date: 2024

conference/Event: Journal of Infrastructure, Policy and Development is published by EnPress Publisher

This research presents iRodd (Intelligent-Road Damage Detection), an advanced artificial intelligence-based system designed for real-time detection, classification, quantification, and visualization of road damage. Traditional road inspection methods, such as manual surveys and high-cost specialized

vehicles, are inefficient, costly, and often inaccurate, particularly for large-scale road networks. To address these challenges, iRodd utilizes a single Convolutional Neural Network (CNN) for object detection, enabling precise identification of different types of road damage. For potholes, the system integrates LiDAR technology on smartphones to measure damage volume and generate 3D visualizations. The model was tested in 38 provinces across Indonesia, achieving a precision of 95% and a mean absolute error (MAE) of 5.35% in pothole volume estimation, with an RMSE of 6.47 mm. The iRodd system is cost-effective, accurate, and efficient, significantly improving road maintenance planning and infrastructure management. By combining real-time detection with advanced visualization, this model enhances road safety and optimizes resource allocation for infrastructure preservation. [13]

Paper 10:

Title: A Fused Deep Learning Expert System for Road Damage Detection and Size Analysis Using YOLO9tr and Depth Estimation

Date: 2024

conference/Event: Faculty of Engineering, King Mongkut's University of Technology Thonburi Bangkok, Thailand

This study presents an advanced road damage detection and measurement system combining the YOLO9tr object detection model with a grayscale-based depth estimation technique. YOLO9tr enables real-time detection of various road damages, including potholes, cracks, and surface deterioration, while the depth estimation module extracts dimensional data from grayscale images. The system achieves a Mean Absolute Percentage Error (MAPE) of 19.4% in damage size estimation, demonstrating consistent performance across different damage types. By eliminating the need for specialized depth sensors and operating with standard imaging equipment, this approach offers a cost-effective and scalable solution for road infrastructure monitoring. The integrated methodology provides precise three-dimensional assessments, enabling more effective road maintenance and safety planning. [14]

Paper 11:

Title: 4D Attention-Guided Road Damage Detection And Classification

Date: 2025

conference/Event: __

This research introduces RDD4D, an advanced road damage detection model that integrates a novel 4D Attention mechanism to improve feature extraction across multiple scales. The study also presents the Diverse Road Damage Dataset (DRDD), which captures various damage types under different conditions, enhancing model training and evaluation. RDD4D leverages the Attention4D module to refine feature maps using positional encoding and "Talking Head" components, leading to superior performance in detecting large road cracks with an Average Precision (AP) of 0.458. The model achieves a significant improvement of 0.21 AP over previous approaches on the CrackTinyNet dataset. These advancements make RDD4D a powerful tool for automated road inspection, contributing to efficient infrastructure maintenance and safety planning. [15]

Paper 12:

Title: YOLOv11: An Overview of the Key Architectural Enhancements

Date: 2024

Cited as: [arXiv:2410.17725](https://arxiv.org/abs/2410.17725)

conference/Event: arXiv preprint

This paper provides a comprehensive analysis of YOLOv11's architectural innovations, focusing on its novel C3k2 block and C2PSA (Convolutional block with Parallel Spatial Attention). The C3k2 block enhances feature extraction by integrating cross-stage partial connections and kernel-wise convolutions, while C2PSA employs parallel spatial attention mechanisms to prioritize critical regions in images. These advancements enable YOLOv11 to achieve a 22% reduction in parameters compared to YOLOv8 while improving mean Average Precision (mAP) by 3.2% on the COCO dataset. The model's streamlined design ensures real-time inference speeds of 18 ms per frame, making it ideal for applications like road hazard detection, where balancing accuracy and latency is critical. [16]

Paper 13:

Title: YOLOv11 for Vehicle Detection: Advancements, Performance, and Applications in Intelligent Transportation Systems

Date: 2024

Cited as: [arXiv:2410.22898](https://arxiv.org/abs/2410.22898)

conference/Event: arXiv preprint

This study evaluates YOLOv11's performance in vehicle detection, particularly in scenarios involving occluded or geometrically complex vehicles. Leveraging the UA-DETRAC dataset, the model achieves a precision of 92.5% and recall of 89.8%, outperforming YOLOv8 by 4.7% in F1-score. Key innovations include adaptive anchor box scaling and dynamic feature pyramid networks, which enhance detection of small or overlapping vehicles. With an inference speed of 21 ms per frame on the NVIDIA Jetson Xavier, YOLOv11 proves highly effective for real-time traffic monitoring and autonomous driving systems, addressing challenges like urban congestion and highway safety. [17]

Paper 14:

Title: Research and Application of YOLOv11-Based Object Segmentation in Intelligent Recognition at Construction Sites

Date: 2024

Cited as: [MDPI:2075-5309](https://doi.org/10.3390/buildings140505309)

conference/Event: MDPI Buildings

Focused on construction site safety, this paper applies YOLOv11 to segment and detect objects such as debris, machinery, and workers under challenging conditions (e.g., low light, dust, and occlusions). The model integrates instance segmentation heads and multi-scale feature fusion, achieving a 92.3% mAP@0.5 on a proprietary dataset. Comparative tests show a 15% improvement in recall over Mask R-CNN in detecting small objects like nails or cracks. The study underscores YOLOv11's adaptability to harsh environments, with direct implications for road safety systems requiring reliable hazard detection in fog, rain, or uneven terrain. [18]

Paper 15:

Title: YOLOv11 Optimization for Efficient Resource Utilization

Date: 2024

Cited as: [arXiv:2412.14790](https://arxiv.org/abs/2412.14790)

Conference/Event: arXiv preprint

This work optimizes YOLOv11 for edge deployment by implementing layer pruning, FP16 quantization, and dynamic computation allocation. The pruned model reduces parameters by 35% while retaining 98% of the original mAP@0.5 on the COCO dataset. Quantization further decreases memory usage by 40%, enabling deployment on devices like the Raspberry Pi 4 with inference speeds of 28 ms per frame. The paper also introduces size-specific variants (e.g., YOLOv11-nano for low-power devices and YOLOv11-XL for cloud servers), ensuring scalability for road safety systems operating across diverse hardware platforms. [19]

Paper 16:

Title: Research on the Directional Bounding Box Algorithm of YOLOv11 in Tailings Pond Identification

Date: 2024

Cited as: [SSRN:5055415](#)

Conference/Event: SSRN preprint

Addressing irregularly shaped hazards, this study modifies YOLOv11 with oriented bounding boxes (OBBs) to detect tailings ponds in mining areas. The OBB mechanism uses angle prediction heads to fit rotated rectangles around oblique or elongated objects, achieving a mAP@0.5 of 0.919 on a dataset of 15,000 satellite images. The model's precision in identifying geometrically complex hazards—such as landslides or eroded road edges—demonstrates its potential for road safety applications, particularly in rural or mountainous regions where traditional bounding boxes fail to capture damage accurately. [20]

Paper 17:

Title: You Only Look Once: Unified, Real-Time Object Detection

Date: 2016

Cited as: [arXiv:1506.02640](#)

Conference/Event: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)

This paper introduces YOLO, a novel object detection framework that reframes detection as a single regression problem, directly from image pixels to bounding box coordinates and class probabilities. Unlike traditional methods that repurpose classifiers for detection, YOLO applies a single neural network to the full image, enabling end-to-end optimization and real-time processing at 45 frames per

second. A smaller variant, Fast YOLO, achieves an impressive 155 frames per second. While YOLO may exhibit more localization errors compared to state-of-the-art detectors, it significantly reduces false positives and demonstrates strong generalization to unexpected inputs, outperforming other methods in cross-domain generalization tasks. [21]

Paper 18:

Title: A Review Paper on Computer Vision

Date: March 2023

Cited as: [IJARSCT-8901](#)

Conference/Event: International Journal of Advanced Research of Science, Communication and Technology (IJARSCT)

This review paper provides a comprehensive overview of computer vision, covering its evolution, fundamental techniques, and applications. It highlights the integration of digital image processing, pattern recognition, machine learning, and artificial intelligence in solving complex visual tasks. The paper discusses key areas such as object detection, segmentation, and pattern recognition, emphasizing the role of deep learning in advancing the field. It also explores challenges like parameter sensitivity, algorithm robustness, and computational complexity, while outlining future directions such as 3D vision, edge computing, and ethical AI. The review underscores the transformative potential of computer vision in industries like healthcare, autonomous vehicles, and agriculture, making it a foundational resource for understanding the field's current state and future trajectory. [22]

Paper 19:

Title: A tutorial survey on vehicle-to-vehicle communications

Date: 4 December 2019

Published in: Telecommunication Systems (Springer Nature)

In this paper, we presented a tutorial overview of V2V communication. We focused on various aspects of V2V; these include the architectural requirements for V2V communication, V2V applications, and protocols. While the NHTSA, as a branch of the U.S. Department of Transportation (DOT), has mandated the use of DSRC [79], several car manufacturers have already started exploring alternative wireless technologies (such as Cellular-V2X), which they argue have better advantages than DSRC.

Both DSRC and Cellular-V2X have their own benefits. For example, DSRC is efficient for a short-range application (approximately 300 m), whereas cellular technologies are more suitable for applications such as traffic control and infotainment that require longer coverage ranges. Yet 5G technology provides ultra-low latency, high data rates, guaranteed jitter, and low rates of packet loss, all of which will meet many of the most stringent requirements of VANET applications. As 5G becomes ubiquitous and cars with DSRC support flow into the market, we must ensure that these two technologies can coexist and interoperate. [23]

Paper 20:

Title: Vehicle to Vehicle (V2V) Communication Protocol: Components, Benefits, Challenges, Safety and Machine Learning Applications

Date: Feb. 2021

Cited as: [arXiv:2102.07306](https://arxiv.org/abs/2102.07306)

The Vehicle-to-vehicle communication protocol and security systems are in the developing stage. It is currently able to send and receive warning messages to and from drivers. In the future, this Vehicle-to-vehicle communication protocol and security systems can be used with automated vehicles that are manufactured using artificial intelligence and machine learning algorithms. In future automated vehicles become common and there will be no need for drivers to drive. The distraction from driving and the security of data concerns will be eliminated. The Vehicle-to-vehicle communication protocol and security systems will increase driving efficiency, protect lives from accidents, and gives organizations to increase their productivity. It is also used to reduce the emission of carbon by reducing congestion in metropolitan cities. The paper has given a brief explanation about Vehicle-to-vehicle communication protocol and security systems, its benefits, limitations, and future aspects.[24]

Paper 21:

Title: Communication Module for V2X Applications using Embedded Systems

Date: 2021

Conference/Event: Virtual Conference on Engineering, Science and Technology (ViCEST)

In this paper, an affordable communication device is implemented to enable all the vehicles that support OBDII port to have accessibility to cellular V2X services and features, which are becoming

the future of the vehicle industry, because the most newly developed vehicles are equipped with similar communication modules. Cellular V2X requires all vehicles to have this module to transmit vehicle data and to receive feedback, so that V2X services can be implemented promisingly. Furthermore, the device can be upgraded to support other cellular communication systems such as LTE, and 5G by replacing the transceiver with a new upgraded one without affecting other components of the system. [25]

Paper 22:

Title: Secure vehicle to vehicle voice chat based MQTT and coap internet of things protocol

Date: 2020

Conference/Event: Indonesian Journal of Electrical Engineering and Computer Science

In this work, a secure message between ambulance and central emergency monitoring and tracking unit based MQTT IoT protocol and other vehicles in the road using CoAP IoT protocol. The message has been encrypted using OTP and DNA computing. The proposed work shows fast encryption/decryption process where the main point in this work is to not affect the overall process of the system and guaranteed emergency services on time. Key generation based on LFSR is not good enough since it suffers from key repetition which has been solved in this work by combining different key to generate a single key. [26]

Paper 23:

Title: Using the MQTT Protocol to Transmit Vehicle Telemetry Data.

Date: 2022

Conference/Event: XII International Conference on Transport Infrastructure: Territory Development and Sustainability

As a result of the study, the network protocol MQTT was considered for the transmission of short messages containing the indicators of sensors to monitor the condition of the vehicle during the execution of work on the transport of goods. The advantage of this protocol is its simple structure of control messages, the possibility of use on low-resource systems, and the possibility of easy integration, and the information system of the enterprise, through software libraries. This protocol will be widespread in the field of IoT and machine to-machine communication, in cooperation with network

protocols such as 5G, Zigbee, and will be analogous to the web protocol HTTP, in the Internet of Things. The flexibility and freedom from semantics of the data format of the protocol allows you to choose the data format that will provide the necessary business requirements of the information system of the transport company. [27]

Paper 24:

Title: VANETs Cloud: Architecture, Applications, Challenges, and Issues

Date: 2021

Conference/Event: -

VANETs Cloud formed by making use of cloud computing paradigms along with the VANETs is believed to enhance and strengthen the capabilities and features of existing VANETs in improving the state of art of the current transportation system by providing plethora of applications related to healthcare, congestion control, disaster management, parking space management, and forming of datacenters from the vehicles in parking lot. In this review article on VANETs Cloud, attempts has been made to analyze the different architectures supported in VANETs Cloud on the basis of different parameters like communication type, cloud state, and services provided etc. Various applications supported by VANETs Cloud is also thoroughly discussed to find out the capabilities of VANETs Cloud. [28]

Paper 25:

Title: The Vehicle as a Mobile Sensor Network base IoT and Big Data for Pothole Detection Caused by Flood Disaster.

Date: 2019

Conference/Event: IOP Conf. Series: Earth and Environmental Science.

In this study, we built vehicles as a mobile sensor network (VaaMSN) and intelligent and analytic pothole detection in Real-Time Systems (SEMAR) for pothole monitoring systems. VaaMSN consists of modem accelerometer, gyro, GPS, 4G WiFi which is connected to the Head Unit and connected to the vehicle. Data from VaaMSN is sent via the MQTT protocol to the Big Data platform and analyzed by decision trees (DT) and machine support vector algorithms (SVM). Data is sent and visualized in real time. We conducted experiments by combining the functions of analytic data using the SVM

Linear algorithm and also the decision tree algorithm. MSE values from the decision tree can be 6.2% and 5.5% for the SVM algorithm. [29]

Paper 26:

Title: A Secured Manhole Management System Using IoT and Machine Learning.

Date: 2022

Our secured manhole management system monitors the lid of the manhole continuously and sends the alert information to the Municipal Corporation whenever the position of the lid is changed. Similarly the condition of the sewers are monitored periodically and an alert information regarding the issue in the sewer system along with its location details are sent to Municipal Corporation to take necessary actions whenever there is an emergency situation. This system also monitors and intimates the higher authorities in the Municipal Corporation about the availability of the corporation workers in attending the issue reported spot. This system is very much useful in maintaining a proper hygiene in our society. [2]

Paper 27:

Title: AIoT-CitySense: AI and IoT-Driven City-Scale Sensing for Roadside Infrastructure Maintenance

Date: 2023

In this paper, we presented an AI and IoT-driven city-scale sensing framework, AIoT-CitySense and a tailored solution of this, Mobile IoT-Roadbot. Mobile IoT-Roadbot is an innovative first-of-its-kind solution for city-scale sensing that has been deployed and piloted on 11 waste collection service trucks in Melbourne, Australia. The solution uses IoT devices to capture data about roadside infrastructure and advanced AI models to automatically detect and report issues with roadside infrastructure. A 6-month pilot of the solution validates its ability to offer city-scale sensing scalability while delivering significant benefits in supporting cities via real-time identification and reporting of road infrastructure issues. AIoT-CitySense has the potential to revolutionise the way cities manage their infrastructure and services, making them more efficient and responsive to the needs of their citizens. Future work includes further optimization and enhancement of our AI models to improve the accuracy of automatic issue detection and to develop and deploy the models on the edge. [30]

Paper 28:

Title: Overview of V2V and V2I Wireless Communication for Cooperative Vehicle Infrastructure Systems.

Date: 2019

conference/Event: IEEE 4th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC 2019).

V2V and V2I communication technology is one of the key technologies of CVIS, also known as CVIS information exchange technology. It guarantees real-time and efficient links between people, vehicles and infrastructure. With the advancement of communication technology, CVIS information exchange has evolved from the traditional central station forwarding mode to the vehicle self organizing forwarding mode. That is to say, the vehicle ad hoc network mode is used for information exchange. The moving vehicle can communicate directly with the nearby vehicle and roadside unit to form a single-hop mode. DSRC and LTE technologies can meet the high demand for real-time transmission of vehicle communications. On the contrary, they can be realized by global microwave interconnection access technology such as WiMAX, Wi-Fi, 3G and other communication technologies. Broadly speaking, wireless communication technology can be used for long distance point-to-point voice and data interaction. In addition to the communication technology, Bluetooth, broadcasting and other technologies are also used in the CVIS [39]. Among them, DSRC/WAVE is an extension of the IEEE 802.11 standard, which is supported on the general 5.8/5.9 GHz band of information exchange protocol. It can meet the needs of information exchange between vehicles and infrastructures. WAVE has the characteristics of self organization, low delay, long distance transmission and fast transmission rate. The IEEE 802.11p adopts the Wi-Fi standard tailored to the vehicle environment and can cover up to 400 meters near the vehicle. The Physical Layer (PHY) of 802.11p in IEEE contains eight different data rates, ranging from 3 Mbps to 54 Mbps. However, because of its highest packet transfer rate performance [40], the standard defines 6Mbps as the default data rate. WLAN technology uses IEEE 802.11a/b/g as the protocol standard, works in the general 2.4 GHz ISM band, the communication distance is about 200m, and the theoretical maximum transmission rate is 54 Mb/s. With the development of WLAN technology, higher speed, wider coverage and more stable links make it a powerful candidate for CVIS communication. [31]

Paper 29:

Title: Evolutionary V2X Technologies Toward the Internet of Vehicles: Challenges and Opportunities**Date:** 2, February 2020

To support the recently emerged demands from the mobility world, e.g., ITS, advanced driving, and autonomous vehicles, conventional connected vehicles are evolving from V2X communications to the IoV. V2X CT has emerged to enhance road safety and transportation efficiency by enabling vehicles to communicate with their external environments, e.g., other vehicle's motion states, traffic conditions, etc. The original V2X radio technology, i.e., the DSRC is based on the IEEE Standard (Std.) 802.11p, which is an amendment version from the IEEE Std. 802.11a to take advantage of the simplicity and capability of distributed operation of 802.11 networks, such as dynamic spectrum access, quick deployment, and effective network access. Ever since the initial release of DSRC in 1999, various V2X technologies have been developed to support ubiquitous, wide-scale, and high performance communication methods for vehicle users, including both the IEEE 802.11 V2X and cellular V2X (C-V2X). The evolution of V2X applications includes three stages. The first and second stages focus on the fields of ITS telematics and advanced auxiliary driving, respectively. As the age of 5G is approaching, V2X technology is evolving to the third stage that can support a wider range of advanced automotive applications, such as autonomous vehicles, remote and cooperative driving, and real-time ITS environmental perception and control. In this article, we first provide an overview of the initial generation of V2X technology, i.e., the DSRC. Then, we investigate the main camps of V2X technology from two different routes, i.e., 802.11 V2X and cellular V2X, and elaborate an in-depth technical comparison between them. Finally, considering the advent of big data-enabled paradigms and cloud-edge computing regime, we study two promising future trends for IoV technologies, i.e., big data-driven IoV and CIOV, which can provide the urgently needed information for both academia and industry. [32]

Paper 30:

Title: V2X Communication Technology: Field Experience and Comparative Analysis **Date:** 23 October 2012

The exploration is built upon Delphi's, Nissan's, Cohda Wireless' and Savari's experiences in Asia, Europe and U.S.A. It describes and derives lessons from all four companies' contributions in projects such as SMARTWAY in Japan, Drive C2X and in Europe, as well as the Connected Vehicle Safety Pilot in the U.S.A. All the above programs were implemented by means of the Dedicated Short Range Communication (DSRC) technology in the SHF spectrum based on the IEEE 802.11p/Wireless Access in Vehicular Environments (WAVE) standard. The study is supplemented with insights regarding complementary technologies such as DSRC in the lower UHF frequency band (i.e. 700 MHz) as well as a V2X implementation through the 4G LTE (Long Term Evolution) cellular telecommunication technology. This paper addresses issues regarding the physical layer (PHY) of the DSRC system. The combination of the delay profile caused by multipath propagation along with the motion-based Doppler spread leads to time and frequency dispersion. This limits the number of bytes acceptable for reliable communication or requires a solution at the receiver end. The analysis of the Doppler spread shows that DSRC implemented at 700 MHz is more immune from data packet length issues as opposed to 5 GHz DSRC. On the other hand, 700 MHz DSRC exhibits a much longer delay spread. Thus, guard time interval specified in ASTM E2213 03 cannot be applied as is to 700 MHz DSRC. This paper refers to the German project CoCarX and the Japanese SKY for pedestrian for studying the feasibility a V2X system built on the 4G/LTE technology and its infrastructure. It provides on a vision for an accelerated V2X deployment based on a heterogeneous system. Last, we recommend the ITS stakeholders to carry out extensive research and validation works on DSRC capacity for ensuring a large scale deployment. [33]

2.3 Conclusion

In reviewing the existing literature, it is evident that each study has successfully addressed specific challenges within the domain of road damage detection and vehicle communication. However, a critical observation reveals a discernible gap—there is a need for a comprehensive, real-time system that seamlessly integrates road damage detection, V2V and V2C-based cloud reporting.

The envisaged solution should function in real-time, ensuring an effective framework for detecting road hazards, communicating alerts to vehicles, reporting damage to authorities for timely maintenance and visualizing road damages on interactive map. Additionally, it is noteworthy that some of the technologies discussed in the literature, such as fully developed C-V2X infrastructure, may not yet be readily available in the specific context of Egypt and similar regions. Hence, there arises the imperative to tailor our project to the unique technological landscape, ensuring practical implementation within local road networks.

Proposed solutions:

Our project aims to bridge this gap by developing a real-time, AI-powered system that integrates all essential subsystems. Specifically, the project seeks to address the following key aspects:

- Real-Time Road Damage Detection Using AI:

- Implementation of YOLOv12-based deep learning models for real-time and high-accuracy detection of road damage, including potholes, cracks, manholes, garbage on the street.
- Integration of image processing and machine learning to ensure detection under various environmental conditions (lighting, weather, road types).
- Use of onboard cameras and edge computing for instant processing and classification of road hazards without requiring high computational resources.

- Efficient Hazard Communication Using V2X Technology:

V2V Communication:

- Vehicles will exchange real-time hazard alerts to warn nearby drivers about detected road damage.
- Ensures proactive decision-making, such as lane changes and speed adjustments, to enhance safety.

V2C Communication (Cloud-Based Reporting):

- Detected road damage data is uploaded to a cloud-based system, allowing authorities to receive real-time reports.
- Enables automated road maintenance scheduling based on severity and location.
- Ensures long-term road condition monitoring for infrastructure improvements.

Integration with OpenStreetMap to:

- Visualize detected road damage on an interactive map.
- Enable authorities and users to access live road condition updates through a user-friendly interface.

By integrating AI-based road damage detection, real-time V2V and V2I hazard communication, cloud-based V2C reporting, and OpenStreetMap-based damage visualization, our project offers a comprehensive, real-time solution that addresses key gaps in existing literature. This approach not only enhances road safety but also supports efficient road maintenance strategies, ultimately advancing intelligent transportation systems in Egypt .

CHAPTER 3

SOFTWARE DESIGN AND COMPUTER VISION



3.1 INTRODUCTION TO COMPUTER VISION

Computer vision is an interdisciplinary field of artificial intelligence (AI) that enables machines to interpret and analyze visual data—such as images, videos, and live camera feeds—to replicate or augment human visual perception. Rooted in digital image processing, pattern recognition, machine learning (ML), and computer graphics, computer vision bridges the gap between raw visual data and actionable insights. Its evolution has been driven by advancements in sensor technology, computational power, and algorithmic innovation, making it indispensable across industries like healthcare, autonomous driving, agriculture, and road safety.

3.2 HOW COMPUTER VISION WORKS

Computer vision systems follow a structured pipeline to transform raw visual data into meaningful information. This process involves three primary stages: **image acquisition**, **preprocessing**, and **interpretation**, supported by techniques from image processing, pattern recognition, and machine learning.

3.2.1 Image Acquisition

The first step involves capturing visual data using imaging devices like cameras, sensors, or drones. For example, in road safety applications, vehicle-mounted cameras collect high-resolution images of road surfaces to identify hazards like potholes or cracks. The quality of this data is critical, as factors like resolution, lighting, and sensor noise directly impact downstream analysis.

3.2.2 Preprocessing

Raw images often contain noise, distortions, or irrelevant details that must be standardized for analysis. Key preprocessing steps include:

- **Resizing:** Scaling images to a uniform resolution (e.g., 640x640 pixels).
- **Normalization:** Adjusting pixel values to a standardized range (e.g., 0–1) for consistency.
- **Noise Reduction:** Applying filters like Gaussian blur or median filters to remove artifacts.
- **Color Correction:** Enhancing contrast or converting images to grayscale to emphasize features.

For instance, preprocessing foggy road images might involve dehazing algorithms to improve visibility for hazard detection.

3.2.3. Feature Extraction

This stage identifies patterns or structures within the image that are relevant to the task. Features can include edges, textures, shapes, or color gradients. Techniques are divided into two categories:

- **Traditional Methods:**
 - **Edge Detection:** Algorithms like Canny or Sobel highlight boundaries between objects (e.g., the outline of a pothole).
 - **Keypoint Detection:** Identifying unique points (e.g., corners) using methods like SIFT (Scale-Invariant Feature Transform) or SURF (Speeded-Up Robust Features).
- **Deep Learning:**
 - **Convolutional Neural Networks (CNNs):** Automatically learn hierarchical features through layers like convolutions and pooling. For example, a CNN might learn to distinguish cracks from shadows by analyzing texture patterns.

3.2.4. Interpretation and Decision-Making

The final stage involves deriving actionable insights from extracted features. This includes:

- **Classification:** Assigning labels to objects (e.g., “pothole” or “oil spill”).
- **Localization:** Drawing bounding boxes around detected objects.
- **Segmentation:** Dividing the image into regions (e.g., separating road surfaces from non-road areas).
- **3D Reconstruction:** Estimating depth or spatial relationships (e.g., calculating pothole depth).

In road safety systems, this stage triggers actions like alerting drivers or prioritizing repairs for municipal authorities.

3.3 KEY TECHNIQUES IN COMPUTER VISION

3.3.1. Object Detection

Object detection identifies and localizes objects within an image. Traditional methods like SIFT and SURF rely on handcrafted features, while modern approaches leverage deep learning:

- **Convolutional Neural Networks (CNNs):** Use convolutional layers to extract spatial features and fully connected layers for classification. YOLO (You Only Look Once) is a popular CNN-based framework for real-time detection.
- **Region-Based Methods:** Models like R-CNN and Faster R-CNN propose regions of interest before classification, achieving high accuracy at the cost of speed.

3.4 OBJECT DETECTION

Object detection is a critical task in computer vision that involves identifying, localizing, and classifying objects within an image or video frame. It serves as the backbone for numerous real-world applications, including autonomous driving, surveillance, robotics, and, most notably, **road safety systems**. The primary goal of object detection is to enable machines to perceive and interpret their environment by accurately outlining objects of interest and assigning them appropriate labels. All object detection algorithms share three fundamental components.

3.4.1 Key Components of Object Detection

- **Localization:**
Localization involves determining the precise location of objects within an image by drawing **bounding boxes** around them. For road safety applications, this means identifying hazards such as potholes, cracks, or oil spills and marking their exact positions on the road surface.
- **Classification:**
Classification assigns a label or category to each detected object. In the context of road safety, this step distinguishes between different types of hazards (e.g., “pothole,” “crack,” or “oil spill”) to provide actionable insights for drivers and maintenance teams.
- **Object Recognition:**
Object recognition goes beyond detection and classification by associating

detected objects with specific classes or categories. This step enables the system to understand the context of the detected hazards, such as their severity or potential impact on road safety.

3.4.2 Techniques for Object Detection:

Object detection encompasses two primary techniques: Traditional Computer Vision and Deep Learning. Traditional methods rely on handcrafted features and traditional algorithms, while Deep Learning utilizes neural networks to automatically learn features. Deep Learning, particularly with Convolutional Neural Networks, has revolutionized object detection, achieving high accuracy and efficiency. Both approaches have strengths and weaknesses, and choosing the appropriate technique depends on the specific application requirements.

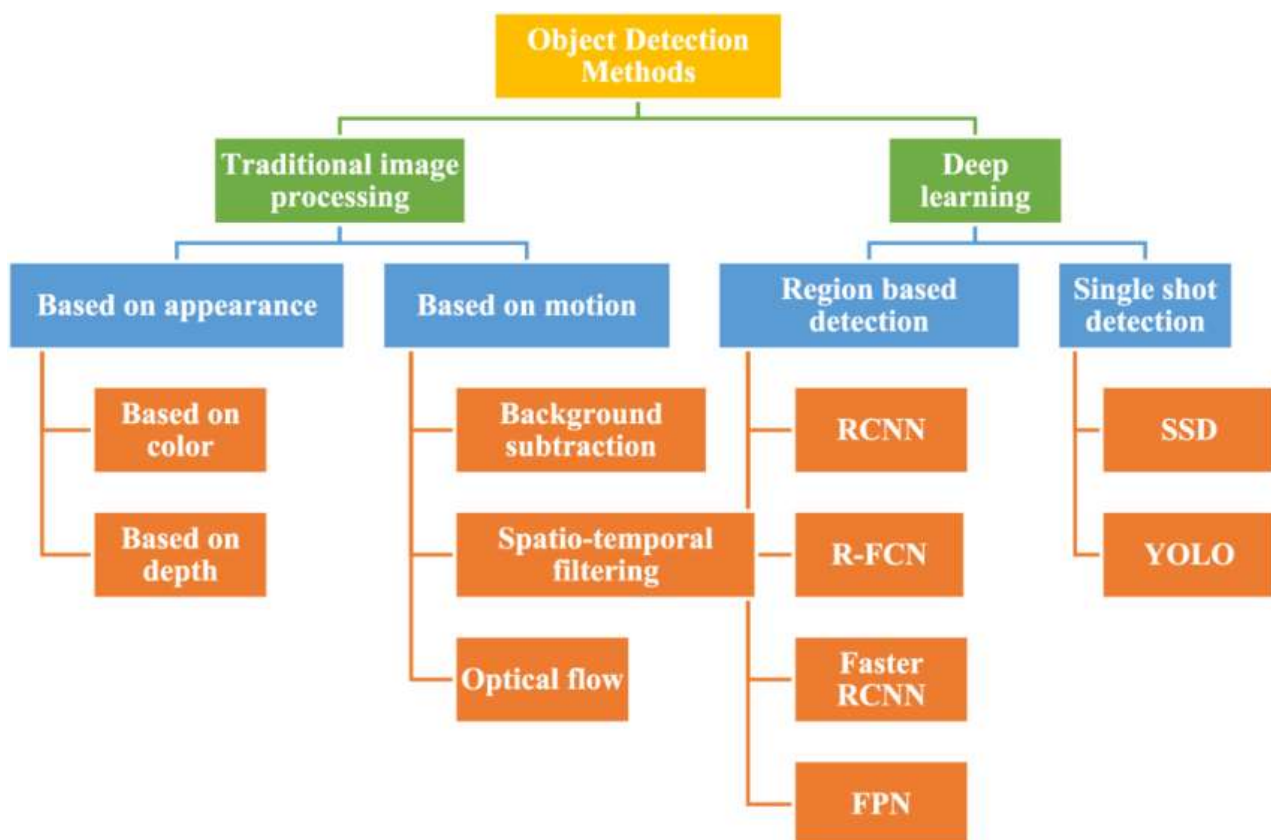


Fig.4 Techniques for Object Detection

3.5 ROAD HAZARD DETECTION

Road hazard detection is a specialized application of object detection that focuses on identifying and classifying road surface anomalies, such as potholes, cracks, oil spills, and other unexpected obstacles. These hazards pose significant risks to road safety, vehicle integrity, and overall traffic efficiency. Traditional methods of road inspection, such as manual surveys or sensor-based systems, are often time-consuming, labor-intensive, and prone to human error. In contrast, AI-driven solutions, particularly those leveraging deep learning, offer a scalable, accurate, and real-time alternative for hazard detection.

The success of road hazard detection systems hinges on their ability to:

1. **Detect Hazards in Real-Time:** Identify hazards as they appear, enabling immediate alerts to drivers and authorities.
2. **Classify Hazards Accurately:** Distinguish between different types of hazards to provide context-specific responses (e.g., icy patches vs. oil spills).
3. **Operate Under Diverse Conditions:** Maintain robustness across varying lighting, weather, and road conditions.

3.6 YOU ONLY LOOK ONCE (YOLO)

To meet these requirements, the project employs **YOLO (You Only Look Once)**, a state-of-the-art object detection framework, and specifically its latest iteration, **YOLOv12**.

3.6.1 Why YOLO for Road Safety?

YOLO has emerged as a leading choice for real-time object detection due to its unique architecture and performance advantages:

1. **Speed:**
 - YOLO processes an entire image in a single forward pass through the neural network, unlike region-based methods (e.g., R-CNN) that require multiple passes.
 - This single-pass approach enables **real-time inference**, making YOLO ideal for time-sensitive applications like road hazard detection.

2. Accuracy:

- YOLO achieves high precision by leveraging advanced convolutional layers and anchor-based detection mechanisms.
- It balances localization and classification tasks effectively, ensuring accurate bounding boxes and labels.

3. Efficiency:

- YOLO's lightweight architecture allows it to run efficiently on edge devices, such as vehicle-mounted systems, without requiring extensive computational resources.

4. Scalability:

- YOLO's modular design supports customization for specific use cases, such as detecting rare or complex road hazards.

3.6.2 How YOLO Works?

The "**You Only Look Once**" (YOLO) object detection system introduces a novel approach by reframing object detection as a single regression problem, directly mapping image pixels to bounding box coordinates and class probabilities. This method eliminates the need for complex pipelines used in traditional detection systems. **Figure 5** illustrates this process, where the input image is divided into a grid, bounding boxes and confidence scores are predicted, and final detections are obtained.

As shown in **Figure 5**, YOLO divides the input image into an $S \times S$ **grid**. Each grid cell is responsible for predicting **B bounding boxes** along with their confidence scores, which indicates the probability of an object being present and the accuracy of the bounding box prediction. Additionally, each cell predicts **C class probabilities** for the object within that cell. These predictions are encoded as an $S \times S \times (B * 5 + C)$ **tensor**.

The model employs a **single convolutional neural network (CNN)** that processes the entire image to generate these predictions simultaneously. This unified architecture allows YOLO to reason **globally** about the image context, leading to more informed predictions. At test time, the network processes an image in a single evaluation, making it **highly efficient** and suitable for real-time applications.

By training on full images and optimizing directly for detection performance, YOLO achieves a balance between speed and accuracy, making it a powerful tool for real-time object detection tasks.

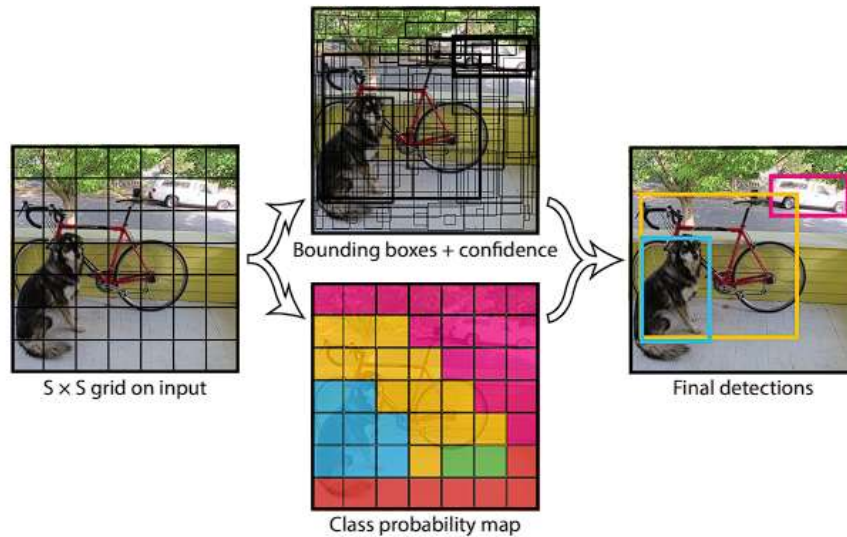


Fig.5 How YOLO Works

3.6.3 Why YOLOv12 Was Chosen for This Project?

YOLOv12, the latest iteration of the YOLO series, was selected for its unparalleled advancements in real-time object detection, efficiency, and adaptability to diverse road safety scenarios. Below is the rationale, supported by recent research:

1. Architectural Innovations for Enhanced Precision

YOLOv12 introduces key architectural enhancements, including the R-ELAN backbone combined with *area* (A^2) *attention* modules and optimized separable convolutions. Benchmarks on MS-COCO show YOLOv12-N achieving **40.6 % mAP** with just **1.64 ms per image** on a T4 GPU—a **+2.1 %** improvement over YOLOv10-N and **+1.2 %** over YOLOv11-N at comparable inference speeds

2. Outstanding Speed–Accuracy Trade-off

Across all scales (N/S/M/L/X), YOLOv12 consistently sets a new performance frontier—offering either better accuracy at the same latency or matching accuracy with faster inference

3. Efficiency through Attention

YOLOv12's attention-centric modules drive significant efficiency: For example, YOLOv12-S runs **42 %** faster than RT-DETR-R18 while using **36 %** of the computing power and **45 %** fewer parameters

4. Scalability and Variant Flexibility

YOLOv12 is available in five size variants—N, S, M, L, X—allowing flexible

deployment across hardware platforms. The larger 'X' variant achieves **55.2 % mAP** on COCO, surpassing YOLOv10-X and YOLOv11-X by 0.6–0.8 %

5. Real-Time Suitability on Modern Hardware

With Turing/Ampere GPUs (e.g., T4), YOLOv12 fully leverages Flash Attention to minimize latency. Without it, performance may drop—but modern hardware enables the model to consistently achieve sub-2 ms latency on small models

6. Support for Advanced Vision Tasks

The inclusion of attention modules and the R-ELAN backbone enhance support for advanced imaging tasks like instance segmentation and pose estimation—aligning with possible future directions (e.g., detecting different road hazards)

3.7 DATASETS

A dataset refers to a collection of data that is organized and grouped based on a specific topic. These datasets can encompass a variety of information, including numerical data, text, images, audio, and videos. Datasets serve as the foundation for training, validating, and benchmarking object detection models.

They provide the necessary data to develop algorithms that can distinguish between different Road damages based on visual characteristics such as shape and size. A robust dataset ensures that the models are exposed to a wide variety of Road damage types, lighting conditions, and viewpoints, making them more generalizable and effective in real-world scenarios.

3.7.1 Road Damage Detector RDD2022

The Road Damage Detector dataset RDD2022, is designed for the task of object detection, specifically targeting the identification and classification of road surface damages. This dataset is particularly valuable for applications in the automotive and utilities industries, enabling the development of deep learning-based methods for automated road damage detection and classification. It is a comprehensive resource for researchers and engineers working on infrastructure maintenance and smart city solutions.

RDD2022: The Multi-National Road Damage Dataset 2022

comprises 47,420 road images from six countries, Japan, India, the Czech Republic, Norway, the United States, and China. The images have been annotated with more than 55,000 instances of road damage. Five types of road damage, namely longitudinal crack, transverse crack, alligator crack, pothole and other corruption are captured in the dataset. The annotated dataset is envisioned for developing deep learning-based methods to detect and classify road damage automatically. The dataset has been released as a part of the Crowd sensing-based Road Damage Detection Challenge (**CRDDC'2022**). The challenge CRDDC'2022 invites researchers from across the globe to propose solutions for automatic road damage detection in multiple countries.

The dataset comprises **41,816 road images** collected from six countries: **Japan, India, the Czech Republic, Norway, the United States, and China**. These images are annotated with **over 55,000 instances of road damage**, categorized into five primary types:

- **Longitudinal Crack (D00):** Linear cracks running parallel to the road's direction.
- **Transverse Crack:** Cracks perpendicular to the road's direction.
- **Alligator Crack:** Interconnected cracks resembling the pattern of an alligator's skin.
- **Pothole:** Depressions in the road surface.
- **Other Corruption:** Miscellaneous damages, including block cracks and repairs.

The images in RDD2022 have been acquired using different methods for different countries. For India, Japan, and the Czech Republic, smartphone-mounted vehicles (cars) were utilized to capture road images. In some cases, the setup with the smartphone mounted on the windshield (inside the car) was also used. Images of resolution 600x600 are captured for Japan and the Czech Republic. For India, images are captured at a resolution of 960x720 and later resized to 720x720 to maintain uniformity with the data from Japan and the Czech Republic.

- For Norway, instead of smartphones, high-resolution cameras mounted inside the windshield of a specialized vehicle, ViaPPS, were used for data collection.
- iaPPS System employs two Basler_Ace2040gc cameras with Complementary metal oxide semiconductor (CMOS) sensor to capture images and then stitches them into one wide view image of a typical resolution 3650x2044.

3.7. 2 Preprocessing

The original dataset annotations were provided in **XML format** using the (xmin, ymin, xmax, ymax) bounding box convention, commonly associated with the **Pascal VOC** format. These annotations were later converted to **TXT format** following the (center_x, center_y, width, height) format used by **YOLO**. This conversion was necessary to ensure compatibility with the training framework and model architecture used in the project.

```
def convert_xml_to_yolo(xml_dir, output_dir, img_width=512, img_height=512):
    """
    Convert XML annotations to YOLO format text files
    :param xml_dir: Directory containing XML files
    :param output_dir: Directory to save YOLO format text files
    """
    os.makedirs(output_dir, exist_ok=True)

    for xml_file in os.listdir(xml_dir):
        if not xml_file.endswith('.xml'):
            continue

        try:
            tree = ET.parse(os.path.join(xml_dir, xml_file))
            root = tree.getroot()

            # Get image dimensions
            size = root.find('size')
            if size is not None:
                img_width = float(size.find('width').text)
                img_height = float(size.find('height').text)

            txt_filename = os.path.splitext(xml_file)[0] + '.txt'
            with open(os.path.join(output_dir, txt_filename), 'w') as f:
                for obj in root.findall('object'):
                    # Class handling
                    name = obj.find('name').text.strip().upper()
                    class_id = class_mapping.get(name, 3) # default to 'other corruption'

                    # Bounding box handling with float conversion
                    bndbox = obj.find('bndbox')
                    xmin = float(bndbox.find('xmin').text)
                    ymin = float(bndbox.find('ymin').text)
                    xmax = float(bndbox.find('xmax').text)
                    ymax = float(bndbox.find('ymax').text)

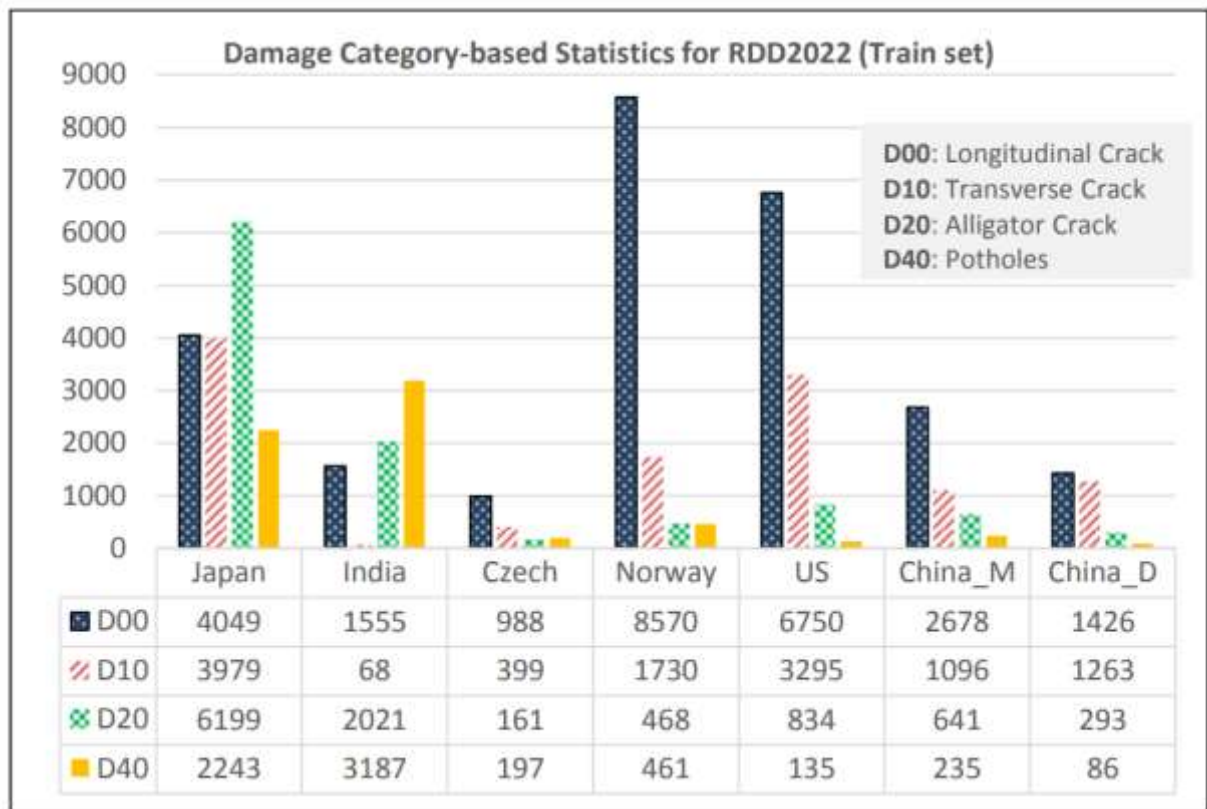
                    # Validate coordinates
                    xmin = max(0.0, xmin)
                    ymin = max(0.0, ymin)
                    xmax = min(img_width, xmax)
                    ymax = min(img_height, ymax)

                    # Convert to YOLO format
                    x_center = (xmin + xmax) / 2 / img_width
                    y_center = (ymin + ymax) / 2 / img_height
                    width = (xmax - xmin) / img_width
                    height = (ymax - ymin) / img_height

                    # Write to file
                    f.write(f"{class_id} {x_center:.6f} {y_center:.6f} {width:.6f} {height:.6f}\n")

        except Exception as e:
            print(f"Error processing {xml_file}: {str(e)}")
```

Fig.6 Preprocessing



Alligator



Fig.9 Alligator
Fig.7 Dataset

Longitudinal

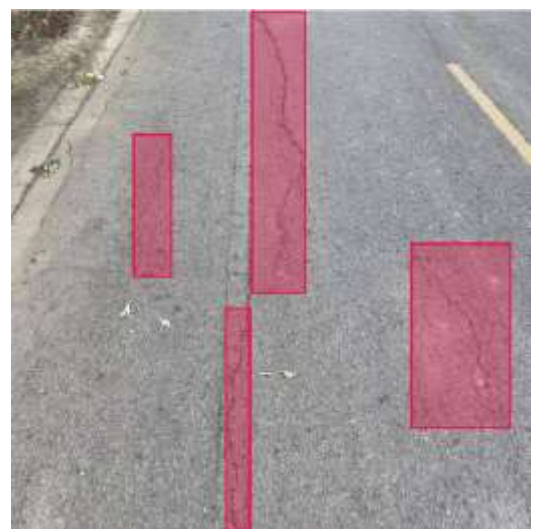


Fig.8 Longitudinal

Transverse Crack



Fig.11 Transverse Crack

Pothole



Fig.10 Pothole

Other Corruption



Fig.12 Other Corruption

3.7.3 Road Damage Dataset from Roboflow

The dataset employed for training the road damage object detection model was sourced from **Roboflow**, a widely used platform for managing and distributing computer vision datasets. Roboflow offers a diverse collection of pre-annotated datasets and supports seamless export in formats compatible with major deep learning frameworks, including YOLO, COCO, and Pascal VOC. Its tools facilitate efficient dataset preprocessing, annotation, augmentation, and version control, making it a reliable solution for developing high-quality object detection models.

Total images: 5503

Resolution: 640x640

Number of Classes: 4

Class: Pothole, Longitudinal Crack, Other Corruption, Alligator Crack

Cameras: Images are captured by multiple surveillance cameras in a real-world environment, primarily during daytime conditions.

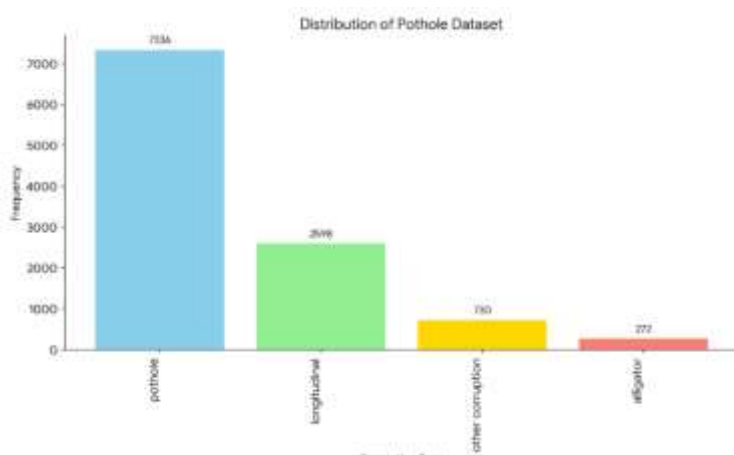


Fig.14 Road Damage Dataset from Roboflow

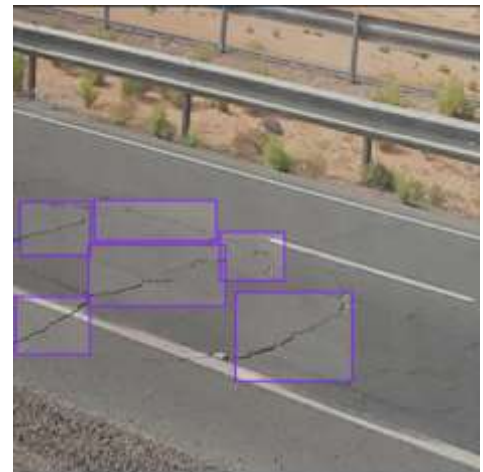


Fig.13 Road Damage Dataset from Roboflow

3.7.4 Pavement Dataset

The Pavement Dataset is a curated collection of images depicting various types of pavement surfaces. It was used as one of the datasets in this project to support object detection tasks related to road and infrastructure analysis.

The dataset was sourced from **Roboflow**, a platform that hosts a wide range of annotated datasets for computer vision applications. Roboflow provides tools for dataset preprocessing, augmentation, and export in formats compatible with machine learning frameworks such as YOLO and COCO.

Description

Total images: 6219

Number of Classes: 5

Class: Pothole, Longitudinal Crack, Other Corruption, Alligator Crack, Transverse Crack

Cameras: Varies across the dataset; images are of different sizes depending on their source and capture conditions, but mostly

Resolution was 640x640

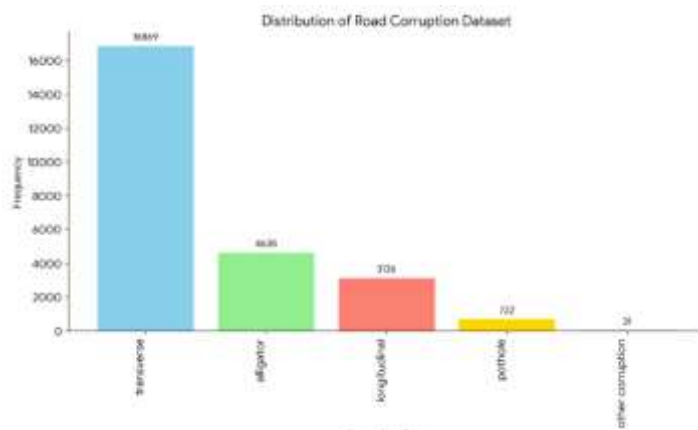


Fig.15 Pavement Dataset 2

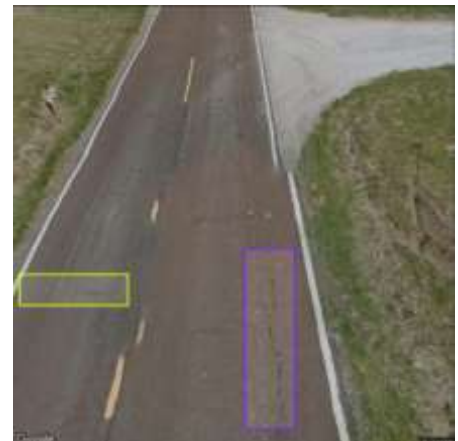


Fig.16 Pavement Dataset 1

3.7.5 Dataset Preparation

To ensure comprehensive coverage of all target classes and to mitigate issues such as **class imbalance** and **data scarcity**, multiple datasets were combined into a single unified dataset. This integration strategy was adopted to increase the variability of image scenes, environmental conditions, and annotation patterns—ultimately improving the model’s performance across real-world scenarios.

Challenges

Several challenges were encountered during dataset preparation, particularly due to differences in annotation formats, class definitions, and data quality across the source datasets:

- **Annotation Format Standardization:** For example, the RDD2022 dataset used XML-based annotations (Pascal VOC format), which had to be carefully converted to the YOLO format. This required parsing and re-structuring each annotation file to ensure compatibility with the unified model training pipeline.
- **Class Index Harmonization:** Each dataset defined damage classes differently, both in naming and index assignment. A general class mapping scheme was developed to unify all class labels, ensuring that semantically similar damage types (e.g., “crack_long” vs. “longitudinal crack”) were mapped to a consistent index across all datasets.
- **Duplicate Image Detection and Removal:** Merging multiple datasets introduced redundancy. A comprehensive duplicate-checking process was implemented to detect and remove repeated images that could bias model training and evaluation.
- **Annotation Quality Control:** Low-quality samples and incorrectly labeled images were identified and either corrected or removed. Some annotations were missing, misaligned, or contained incorrect bounding boxes—these issues were resolved through manual inspection and validation tools.
- **Label Correction and Reindexing:** Before merging, it was necessary to ensure that each dataset’s internal indexing matched the unified class scheme.

Where misalignments were found, class indices were remapped to maintain a consistent structure.

A key challenge in this process was managing **class index conflicts and inconsistencies across datasets**. Each source used different label indices and naming conventions, which required a thorough remapping to a unified label set. Additionally, the distribution of classes in the merged dataset had to be **evaluated to ensure it preserved the diversity and balance** present in the individual datasets. In some cases, class distributions were rebalanced through selective sampling or augmentation.

Another critical step involved identifying and removing **duplicate images**, which could introduce training bias or inflate evaluation scores. Furthermore, several images lacked valid annotations or contained improperly formatted labels, which had to be manually corrected or excluded to maintain annotation quality.

After rigorous curation and validation, the resulting dataset contained **50,107 annotated images**, offering a comprehensive and well-distributed foundation for training. It was split into the following subsets to support effective learning and unbiased evaluation:

- **Training set (70%):** Used to update model weights and optimize performance 40177 image
- **Validation set (20%):** Used to tune hyperparameters and assess overfitting 9930 image
- **Test set (10%):** Held out during training to provide an unbiased evaluation of final model performance 9035 images .

3.8 MODEL TRAINING

The chosen model **YOLOV12n**, The model selected for this project was **YOLOv12n**, the smallest variant in the YOLOv12 family. This choice was made to ensure compatibility with the hardware limitations of the Raspberry Pi 5, where lightweight models are crucial for achieving real-time performance. YOLOv12n offers a significantly reduced model size and computational load compared to larger variants, while still maintaining reasonable detection accuracy and delivering acceptable inference speed (FPS) during deployment on edge devices.

Model	size (pixels)	mAP _{val} 50-95	Speed CPU ONNX (ms)	Speed T4 TensorRT (ms)	params (M)	FLOPs (B)	Comparison (mAP/Speed)
YOLO12n	640	40.6	-	1.64	2.6	6.5	+2.1%/-9% (vs. YOLOv10n)
YOLO12s	640	48.0	-	2.61	9.3	21.4	+0.1%/+42% (vs. RT-DETRv2)
YOLO12m	640	52.5	-	4.86	20.2	67.5	+1.0%/-3% (vs. YOLO11m)
YOLO12l	640	53.7	-	6.77	26.4	88.9	+0.4%/-8% (vs. YOLO11l)
YOLO12x	640	55.2	-	11.79	59.1	199.0	+0.6%/-4% (vs. YOLO11x)

Fig.17 Model training

In this section, we focus on the training phase of the YOLOv12 model on our customized data set . Utilizing the Roboflow platform, we employ labeled image to train the model using the Ultralytics YOLO library,

Training Configuration and Hyperparameters The training was conducted using the Ultralytics implementation of YOLOv12. The key training arguments and hyperparameters were carefully selected based on recommended defaults and empirical tuning. The following configuration was used:

Image size:	<i>640 x 640</i>
Epochs	<i>400</i>
Optimizer	<i>Adam</i>
Early stopping	<i>50</i>
Batch worker	<i>16</i>
	<i>8</i>

Table.1 MODEL TRAINING

```
model.train(data="/kaggle/working/yaml_file.yaml",
            epochs=400,
            pretrained=True,
            batch=16,
            workers=8,
            optimizer="Adam",
            patience=50)
```

Fig.18 Model training2

Transfer learning was employed by initializing the model with pretrained weights from the COCO dataset. This accelerated convergence and improved performance, especially in earlier epochs.

Model training was performed using Kaggle’s free cloud GPU environment, which provides high-performance resources and seamless integration with custom notebooks. Kaggle kernels offered the flexibility to:

- Use preinstalled packages and GPU support (e.g., Tesla P100)
- Integrate directly with Roboflow via API for dataset import
- Utilize experiment tracking and logging

3.9 MODEL RESULTS

3.9.1 To assess the performance of the trained YOLOv12n object detection model, evaluation metrics were calculated across all object classes using a validation dataset consisting of 9,930 images and 20,037 labeled instances. The metrics include **Precision, Recall, mean Average Precision at IoU threshold 0.5 (mAP@50)**, and **mean Average Precision across IoU thresholds from 0.5 to 0.95 (mAP@50:95)**—standard benchmarks in object detection tasks.

Class	Images	Instances	Precision	Recall	mAP@50	mAP@50:95
All	9930	20037	.638	.555	.592	.324
Longitudinal Crack	3318	6403	.606	.464	.5	.269
Transverse Crack	2519	5553	.631	.619	.619	.273
Alligator Crack	1963	3076	.68	.432	.552	.296
Other Corruption	1574	2233	.633	.752	.734	.469
Pothole	1463	2772	.638	.507	.555	.314

Table.1 MODEL RESULTS

3.9.2 Benchmark Dataset

A benchmark dataset was used to evaluate the model’s performance in a standardized setting. This dataset provides only the F1-score as a metric, and the YOLOv12n model achieved an F1-score of **58**, indicating a balanced performance between precision and recall under the benchmark’s evaluation criteria.

3.9.3 Performance Improvement Through Dataset Expansion

To improve the model’s overall performance and address specific shortcomings observed during evaluation—particularly the limited class diversity in the benchmark dataset—we expanded the training data by incorporating additional annotated images from various **Roboflow**-hosted datasets. These datasets included diverse road damage types and were selected to complement the RDD2022 dataset by introducing examples of underrepresented hazards.

This dataset augmentation strategy enhanced the model’s ability to generalize across different road damage classes and environmental conditions. As shown in Table 2, notable improvements were observed in precision and mAP scores for several classes such as *Transverse Crack* and *Pothole*, demonstrating the value of leveraging a broader and more varied dataset. This step was essential for adapting the model to real-world scenarios that go beyond the scope of the original benchmark.

Class	Images	Instances	Precision	Recall	mAP@50	mAP@50:95
All	9930	20037	0.641	0.562	0.598	0.348
Longitudinal Crack	3318	6403	0.613	0.487	0.512	0.301
Transverse Crack	2519	5553	0.648	0.601	0.625	0.296
Alligator Crack	1963	3076	0.665	0.459	0.564	0.296
Other Corruption	1574	2233	0.639	0.755	0.741	0.469
Pothole	1463	2772	0.644	0.526	0.562	0.353

Table.2 MODEL RESULTS

3.10 OBJECT TRACKING

Object tracking is a fundamental task in computer vision that involves continuously identifying and following objects across video frames while maintaining consistent identities. Unlike object detection, which processes each frame independently, tracking associates unique IDs with detected objects over time, enabling motion analysis, behavioral understanding, and trajectory prediction. This capability is critical for applications such as surveillance, autonomous vehicles, and video analytics, where temporal consistency is essential.

3.10.1 Byte Track: State-of-the-Art Multi-Object Tracking

Byte Track (Zhang et al., 2021) is a cutting-edge multi-object tracking (MOT) algorithm that addresses key limitations of earlier methods like SORT and DeepSORT. Traditional trackers discard low-confidence detections, often losing occluded or ambiguous objects. In contrast, Byte Track employs a **two-stage association strategy**:

- **High-score detections** are first matched to existing tracks using IoU (Intersection over Union).
- **Low-score detections** are reused to recover missed objects (e.g., due to occlusion or motion blur).

This approach significantly reduces identity switches (ID switches) and improves robustness in crowded scenes while operating efficiently in real-time. Unlike DeepSORT, ByteTrack avoids computationally expensive appearance models, relying instead on motion cues and detection quality. Its performance on benchmarks like MOT17 and MOT20 has made it a popular choice for practical MOT systems.

3.10.2 Implementation in This Project

For this project, Byte Track was integrated with the **YOLOv12n** object detection model using the Ultralytics framework. The workflow consisted of:

1. **Detection:** YOLOv12n localized objects in each frame with bounding boxes and confidence scores.
2. **Tracking:** Byte Track associated detections across frames, assigning and maintaining unique IDs for each object.
3. **Performance Optimization:** The system was deployed on a **Raspberry Pi 5**, demonstrating efficient real-time tracking despite hardware constraints.

Although object detection alone is sufficient to localize objects in individual frames, it lacks **temporal awareness**. Without tracking, the same object could be treated as a new detection in every frame, leading to redundant or noisy triggers. **Object tracking was essential to ensure that objects were only acted upon when they first appeared**, avoiding repeated responses for the same object. Furthermore, tracking allowed the system to determine when an object had **left the frame**, enabling more stable and intelligent decision-making in real-world scenarios.

By leveraging **Ultralytics' integrated support for ByteTrack**, the system achieved accurate and efficient object tracking suitable for road safety applications as in Fig.

```
model.track(frame, tracker="bytetrack.yaml")
```

3.9.11 MODEL QUANTIZATION

As deep learning models advance in complexity and capability, their **computational and memory requirements** have grown substantially. While powerful GPUs can accommodate these demands during development and training, **deployment to edge devices**—such as the **Raspberry Pi 5**—requires optimization to ensure acceptable performance within strict resource constraints, including limited CPU power, memory bandwidth, and energy efficiency. To bridge this gap between model performance and hardware limitations, **quantization** is a critical step.

3.11.1 What is Model Quantization?

Model quantization is a technique used to reduce the size and computational cost of a neural network by converting its floating-point (e.g., FP32) weights and activations into lower-precision representations, typically INT8 (8-bit integers). This process decreases the model's memory footprint and allows for faster arithmetic operations during inference.

There are several types of quantization:

- **Post-Training Quantization (PTQ):** Applied after training, with minimal or no fine-tuning.
- **Quantization-Aware Training (QAT):** Simulates quantization effects during training, leading to better performance at low precision.
- **Dynamic and Static Quantization:** Based on how and when quantization parameters (e.g., scale and zero-point) are calculated.

3.11.2 Why Quantization Is Needed for Raspberry Pi 5

Although YOLOv12n is one of the most lightweight and efficient variants in the YOLO family, its default FP32 format still presents challenges when deployed on **resource-constrained hardware** like the Raspberry Pi 5. Running the model in FP32 would result in **high latency** and **limited responsiveness**, undermining its real-time detection potential.

To mitigate these issues, the model was **quantized to INT8** and **converted to other low-precision formats like FP16** using Ultralytics' **built-in export tools**. This process yielded several benefits:

- **Reduced model size** → Less memory consumption and faster loading time
- **power usage** → Critical for embedded and battery-powered systems
- **Faster inference** → Enables real-time or near real-time object detection performance on edge hardware

These optimizations ensure that the model remains both **accurate and efficient**, enabling real-time road hazard detection directly on the Pi without offloading to external servers.

3.11.3 MODEL PERFORMANCE AND QUANTIZATION COMPARISON - RPi 5

To enable real-time inference on edge hardware, we tested multiple optimized versions of the YOLOv12n model on a Raspberry Pi 5. These models were exported using different toolchains including ONNX Runtime, OpenVINO, and MNN (Mobile Neural Network), each with varying levels of precision (FP16, INT8) and framework support.

Model Type	Average Inference Time (ms)
ONNX Runtime CPUExecutionProvider	251.4 ms
OpenVINO LATENCY mode for OpenVINO Default Model	175.28 ms
OpenVINO INT8 Model with YOLO framework (batch=1)	155.91 ms
Default Model for MNN Inference	183.15 ms
NN FP16 Model with YOLO framework	183.13 ms
NN INT8 Model with YOLO framework	183.01 ms

Figure 19 MODEL PERFORMANCE AND QUANTIZATION COMPARISON - RPi 5

Among all tested configurations, the **OpenVINO INT8 model** achieved the best performance with an average inference time of **155.91 ms** per frame (**≈6.4 FPS**), making it the most suitable option for **real-time deployment** on Raspberry Pi 5. Compared to the baseline ONNX Runtime (251.4 ms), quantization and framework optimization reduced inference time by over **30%**.

These results demonstrate that the system can operate within real-time constraints on low-power devices, ensuring fast detection and communication of road hazards without reliance on cloud resources.

3.11.4 IMPLEMENTATION AND FORMAT CONVERSION

After training the YOLOv12n model, it was necessary to prepare it for **efficient edge deployment**. Using **Ultralytics' export functionality**, the model was converted into multiple deployment-friendly formats:

- **ONNX:** A widely supported, interoperable format for inference across platforms
- **OpenVINO IR (Intermediate Representation):** Optimized for Intel-based hardware
- **MNN (Mobile Neural Network):** Tailored for lightweight inference on ARM-based devices like the Raspberry Pi

Both **FP16** and **INT8** quantized versions were generated where applicable. The **INT8 version** provided the best trade-off between performance and efficiency, especially when used with OpenVINO for edge inference. The **MNN format** was also evaluated due to its lightweight runtime and fast execution on ARM CPUs, demonstrating potential for extremely low-power environments.

These efforts enabled the **YOLOv12n model to run smoothly on the Raspberry Pi 5**, maintaining sufficient accuracy for road damage detection while achieving performance suitable for real-time operation.

These optimizations were performed using the **Ultralytics framework**, which provides built-in tools for exporting trained YOLO models to multiple formats such as **ONNX**, **OpenVINO**, and **MNN**, along with support for **post-training quantization** as shown in fig

```
model.export(format="openvino",int8=True,batch=1,nms=True,device="cpu")
```

3.12 MODEL DEPLOYMENT

The model's inference performance was evaluated across two environments to assess its suitability for real-time applications. When running the **OpenVINO INT8 quantized model** on a video stream using **Kaggle's GPU-enabled environment**, it achieved an average inference speed of approximately **23 frames per second (FPS)**, demonstrating smooth and responsive real-time processing under high-performance conditions.

In contrast, when deployed on a **Raspberry Pi 5**, the same quantized model—optimized specifically for edge inference—achieved an average of **6 FPS** during standard object detection. When extended to include **object tracking**, the system maintained an average of approximately **3 FPS**. While the frame rate on the Raspberry Pi is lower due to hardware constraints, it remains acceptable for many practical applications where extremely high FPS is not a strict requirement.



Fig.20 Model Deployment

3.12.1 Comparison between Raspberry pi 5 and jetson

While the Raspberry Pi 5 offers a cost-effective and accessible platform for deploying lightweight deep learning models, it has notable limitations compared to more specialized edge AI hardware such as the NVIDIA Jetson series. The Jetson boards—particularly models like the Jetson Nano, Xavier NX, and Jetson Orin—are equipped with dedicated CUDA-enabled GPUs and Tensor Cores optimized for AI workloads, providing significantly higher inference throughput and better support for GPU-accelerated deep learning frameworks. In contrast, the Raspberry Pi 5 relies on a

general-purpose ARM CPU and, while it features improved performance over its predecessors, it lacks dedicated hardware acceleration for deep learning. As a result, models like YOLOv12n that achieve real-time FPS on Jetson platforms may run at lower frame rates on the Pi 5 due to the absence of specialized inference engines. Nevertheless, the Pi 5 remains a practical option for deployment when cost, power efficiency, and portability are key considerations, especially when using optimized models and quantization techniques to maximize performance within its hardware limits as you will see the huge saving in inference time for both

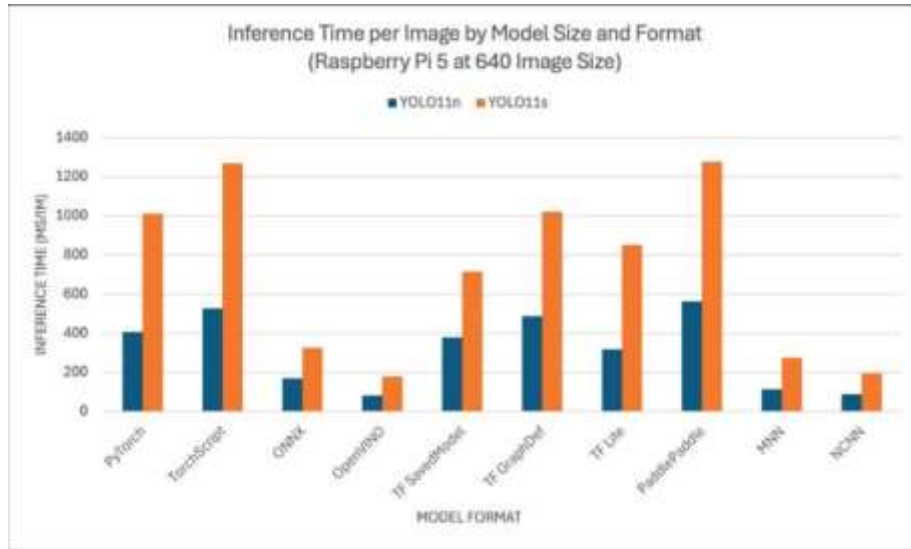


Fig.21 Comparison between Raspberry pi 5 and jetson

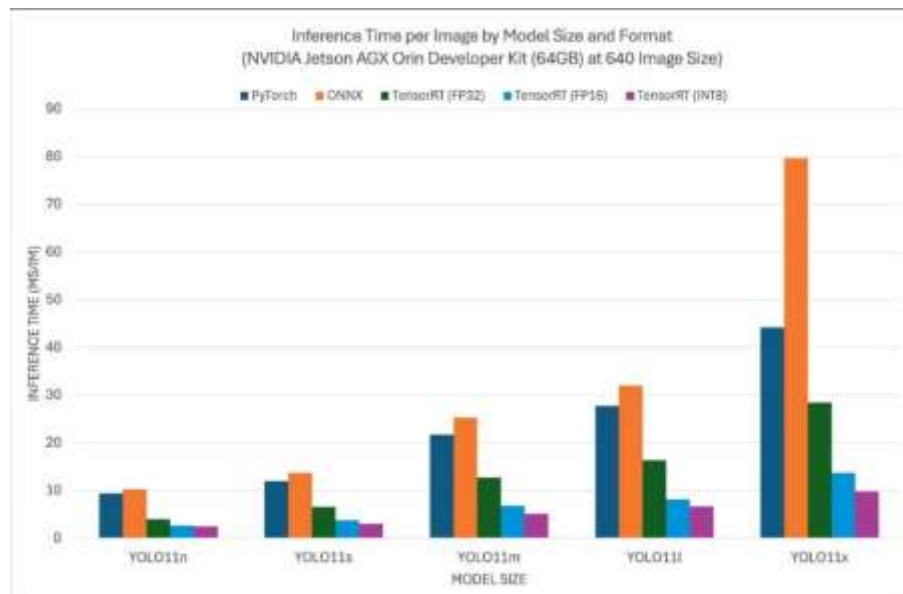
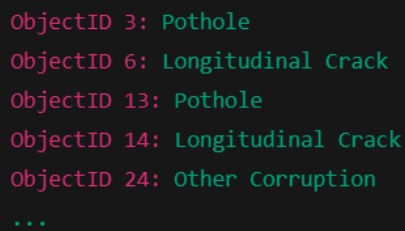


Fig.22 Comparison between Raspberry pi 5 and jetson.

3.13 AI OUTPUT INTEGRATION

At the final stage of the AI pipeline, each detected road hazard is logged in a structured format containing a unique **object ID** and its corresponding **class label**. This log serves as the communication interface between the AI module and other system components (e.g., ESP32 and OSM mapping logic).

An example of the recorded output is shown below:



```
ObjectID 3: Pothole
ObjectID 6: Longitudinal Crack
ObjectID 13: Pothole
ObjectID 14: Longitudinal Crack
ObjectID 24: Other Corruption
...
```

Figure 23 output log

Each line represents a tracked object detected by the system, where:

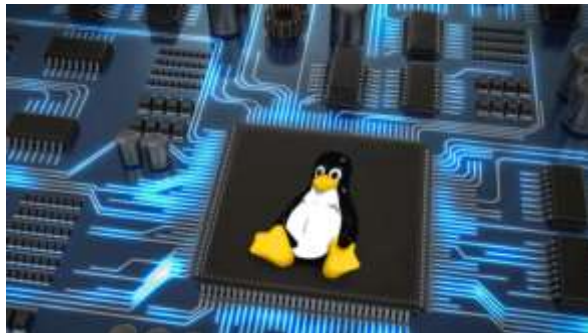
- ObjectID refers to the persistent identifier assigned by the tracking algorithm (ByteTrack).
- The label specifies the classified type of road damage.

These structured detections are transmitted to the **ESP32 module via UART**, enabling **V2V communication using ESP-NOW**, while GPS-tagged entries are published to **OpenStreetMap (OSM)** for real-time hazard mapping.

This logging format ensures accurate, deduplicated tracking and seamless integration with the system's communication and visualization modules.

CHAPTER 4

EMBEDDED LINUX



4.1 What is Embedded Linux?

Embedded Linux is a type of Linux operating system/kernel that is designed to be installed and used within embedded devices and appliances. In other words, it's a compact version of Linux that offers features and services in line with the operating and application requirement of the embedded system. Although it uses the same kernel, Embedded Linux is quite different from the standard operating system. First of all, it gets tailored for embedded systems and, therefore, is much smaller in size, requires less processing power, and has minimal features compared to that of a fully-fledged operating system. The Linux Kernel is modified and optimized as an embedded Linux version. Such a Linux instance can only run applications created specifically for the device.

Embedded Linux also offers its developers with several advantages over other systems such as:

- 1- Cross-Compilation for any supported platform.
- 2- Community reflection of common vulnerabilities and exposures (CVE) fixes in updated releases.
- 3- Deployment to commonly used Linux infrastructure and tools.
- 4- Modern, cloud-native environment.
- 5- Broad hardware support.
- 6- Productive lifecycle through community LTS.

4.2 Embedded Linux Architecture

At the most basic level, an Embedded Linux system is one that uses Linux as the operating system that sits between hardware and the application of an embedded device. There are five key components to an Embedded Linux system which we will go

through, and these components are:

- 1- Bootloader.
- 2- Kernel.
- 3- Root filesystem.
- 4- Services.
- 5- Applications/Programs.

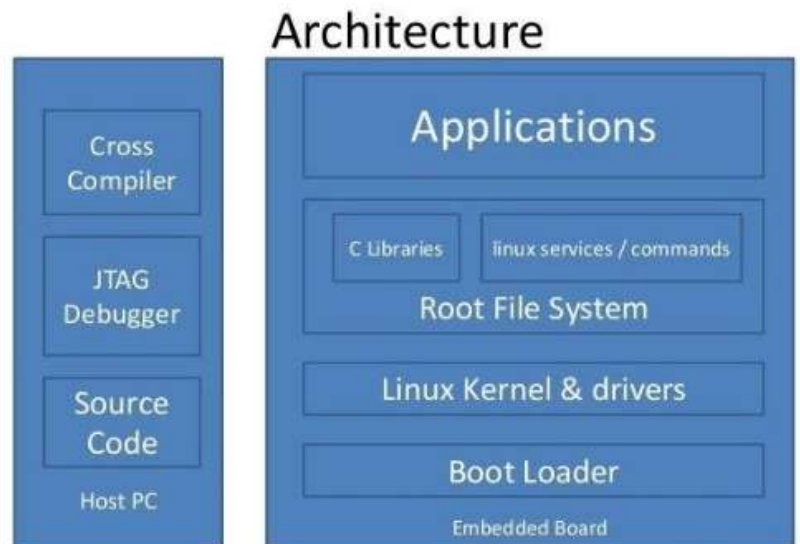


Fig.24

Embedded Linux Architecture

4.2.1 Bootloader

When the computer is powered on, after performing some initial setup, it will load a bootloader into memory and run that code. The bootloader's main job is to find the operating system's binary program, load that binary into memory, and run the operating system, which in our case is the Linux kernel.

The bootloader is done at this point, and all of its code and data in RAM are usually overwritten by the operating system. The bootloader won't run again until the computer is reset or power-cycled again.

The bootloader in embedded systems is different from a typical laptop, desktop, or server computer.

A typical PC usually boots into what we call the BIOS first and then runs GRUB as the bootloader.

Embedded Linux systems boot using Das-UBoot or U-Boot for short as the bootloader. Once the bootloader loads the Linux kernel into memory and runs it, the kernel will begin running its startup code. This code will be responsible for the initialization of the hardware, the system critical data structures, the scheduler, all the hardware drivers, the filesystem drivers, mount the first filesystem, and launch the first program, and more.

4.2.2 Kernel

The Linux kernel's main job is to start applications and provide coordination among these applications (or programs, as they are commonly called in Linux). The Linux kernel cannot identify all the programs that are supposed to run, so the Linux kernel starts only one program and lets that program launch all the other needed programs. And this very first program is none other than the init program, or sometimes referred to as just 'init'. Note that this first program doesn't need to be in a file called 'init', but it often is.

If the kernel for any reason cannot find the init program, the kernel's purpose is gone and the kernel crashes.

The main takeaway in the Linux kernel for embedded systems is that it is built to run on a different CPU architecture. Otherwise, the way the kernel operates is the same as the typical PC, which is one of its main advantages.

4.2.3 Root filesystem

In Linux, the kernel loads the programs into memory separately, and the kernel expects these programs to be stored on some medium organized into files and directories. This organization is what is known as a filesystem. Linux akin to many operating systems has filesystems on media, which is the data stored on a storage medium, and filesystem drivers, which is the code that knows how to interrupt and update the filesystem data on the medium, which is often a hardware device like SD cards, or even flash memory.

The Linux kernel works hand in hand with what is called the root filesystem. This is the filesystem upon which the root directory can be mounted, and which contains the files necessary to bring the system to a state where other filesystems can be mounted and user space daemons and applications started. The directory structure for a root filesystem can be extremely minimal, as we'll see in a moment, or it can contain the usual set of directories including `/dev`, `/bin`, `/etc`, and `/sbin`, among others that you see in any desktop Linux distribution. The kernel boot process concludes with the

5 `init` code (see `init/main.c`) whose primary purpose is to create and populate an initial root filesystem with a set of directories and files. It then tries to launch the first user mode process to run an executable file found on this initial filesystem. This first process ("`init`") is always given process ID 1. There are three ways for the kernel to find the file that will be run by the `init` process. The first method is to use a file specified at boot time with the `init=` kernel parameter. If this parameter is not set, the kernel tries a series of locations to find a file named "`init`". These include `/sbin/init`, `/etc/init`, and `/bin/init`. If all these fail, the kernel tries to run any shell it finds at `/bin/sh`. If this last fallback is not found, the kernel will print an error saying that no `init` could be found.

Unlike Windows, Linux filesystems do not get associated with drive letters, they do get associated with a directory. More so, filesystems can be associated with any directory, even ones that are several layers down in a path, and this association is called 'mounting'. Linux first starts with an empty directory called `/` or slash, then during startup, the top most filesystem gets associated with (or in other words, mounted to) this directory, and all the contents of that filesystem appear under `/` or slash. This topmost filesystem is called the root filesystem.

Linux systems expect the root filesystem to be laid out a certain way. So, this filesystem is special and can't just be some random set of directories and files. This is where directories like `bin`, `sbin` and more come from.

Because embedded systems have different hardware constraints, often Linux embedded systems use special filesystem formats rather than the typical EXT3, EXT4, btrfs, or xfs used on desktop or laptop computers.

4.2.4 Services

When the kernel finds, loads and runs the init program, that program then is responsible for bringing up the rest of the system. At this point, the kernel is no longer actively running and remains to coordinate the sharing of hardware among all the running programs.

There are several init programs available. Regardless of which init program is chosen, this program will launch all of the necessary services and applications that are needed for the system to be useful. This set of services includes setting up networking, mounting additional filesystems, setting up graphical environment, and more.

Under Linux, services are just programs that run in the background. These services were once known as daemon or daemon program, but recently this terminology became less popular.

4.2.5 Application/Program

Embedded Linux enables us to run programs in higher level language than that of bare metal embedded. Languages like python, C/C++, and Rust are the most common. The init program is responsible for starting these programs.

Embedded Linux is used to develop core software, and many other examples such as network equipment, machine control, industrial automation, navigation equipment, spacecraft flight software, and medical instruments in general. Even Microsoft Windows has Linux components as part of the Windows Subsystem for Linux or WSL. But perhaps the best example of an Embedded Linux application is Android, developed by Google

4.3 Embedded Linux vs. Desktop Linux

Embedded Linux	Desktop Linux
Linux kernel running in the embedded system product/single board computer/development board.	Linux kernel running on Desktop/Laptop.
Real time Linux kernel is used, making the response time real time or deterministic.	Linux kernel running in the desktop or laptop is not real time, the kernel response is not deterministic for response against events.
Kernel used in Embedded Linux is the customized version of the original kernel. User configures the kernel as per target processor, components present on the board, need of driver, etc.	Complete version of the kernel is used with all possible drivers and libraries. Whenever any new device or protocol is released then its driver patch is provided by either Linux community or by vendor.
Embedded Linux kernel footprint is less, around 1MBs.	Desktop Linux kernel footprint is more, around 100 MBs.

Table.2 Embedded Linux vs. Desktop Linux

4.4 Embedded Linux vs. Bare metal vs. RTOS

Embedded Linux	Bare-metal	RTOS
Large overhead compared to other technologies due to scheduler, memory management, background tasks, etc.	Little to no software overhead.	Scheduler overhead.
Requires a microprocessor with a memory management unit (MMU) and an external RAM.	Low power requirement.	Requires a more powerful microcontroller.
Low direct control of hardware (files or abstraction layers).	High control of hardware.	High control of hardware
Multiple threads and processes.	Single-purpose or simple applications, hardware-dependent.	Multi-threading.
Multiple complex tasks, like networking, filesystem, graphical interface, etc.	Strict timing.	Multiple tasks, like networking, user interface, etc.

Table.3 Embedded Linux vs. Bare metal vs. RTOS

4.5 Introduction to Embedded Linux Development

Embedded Linux is a wide field, hence an Embedded Linux developer has many goals, targets, and responsibilities, all of which can be divided into:

- Develop and maintain C/C++, Python, or Rust codes.
- Develop and maintain Linux device drivers
- Develop and maintain a Linux environment either through Yocto (most common), or build root, or any other method.

4.5.1 SD card structure in Embedded Linux

Before we dive any deeper, let's first talk about the structure of the SD card we will be using on our target device.

Recall the Embedded Linux elements we discussed earlier:

- Toolchain.
- Bootloader.
- Kernel + Device tree binary (DTB).
- Rootfs
- Application.

Each element should be placed in a specific manner, but first we need to part out our SD card into two partitions:

- Boot, which is a small FAT partition, will contain our bootloader, the kernel, device tree binary (DTB), and the configurations.
- Rootfs, which is a large EXT4 partition, will contain our directories (/user, /bin, etc.).

Now, we need to understand a very important term that we'll see very often not only when dealing with Embedded Linux, but with Linux in general, and that is 'Mount'.

The data we place on our SD card is stored in binary, so how are these ones and zeros transformed into actual information? The kernel contains a virtual file system (VFS) responsible for mapping binaries into readable data (.txt, .img ... etc.). This process is called 'Mounting', where a row of data is transformed into readable data. If we were to try to access or read the data without mounting, the output data would be garbage values.

4.5.2 The Traditional Method for building a Customized Linux Image

Rather than the traditional bare metal embedded software development, where we write a C code, build, and produce a hex file, then burn the hex file on our target hardware, we create a Linux image that is flashed on the development board through an SD card.

The traditional method for building a Linux image is as it may sound, more of a primitive approach where we must, as they say, roll up our sleeves and do all the work by ourselves. The development process can be broken down to the following steps:

- 1- Study the board of choice, and specifically the booting process.
- 2- Build or download a toolchain.
- 3- Download a bootloader, which is a special piece of software with one sole purpose and that is to load the kernel and hand over the control to it (U-boot is the most popular one).
- 4- Build the kernel, which is the heart of the operating system.
- 5- Build the filesystem, depending on our configurations.
- 6- Build our application.

7- Plug and play.

4.5.2.1 Studying the booting sequence

In the previous chapters, we covered the booting sequence of Raspberry Pi with all its steps and elements.

4.5.2.2 Build or download a Toolchain

First, what is meant by a toolchain, a toolchain is a set of tools that compiles source code into an executable that can run on our target device (includes a compiler, kernel headers, binutils, a linker, and run-time libraries).

In order to discuss the importance of the toolchain, we must first understand an important concept, and that is the difference between a cross compiler and a native compiler.

Let's say that we have a C file called test.c, we compile it using the GCC compiler to output test.exe which we run on our host machine. In this case, GCC is called a native compiler because we used it to generate code for the same platform on which it runs.

Say that for the same c file called test.c, we used arm.gcc to compile and output an executable that will run on an ARM target device. In this case, arm.gcc is called cross compiler because it was used to generate an executable code for a platform other than the one on which the compiler is running.

Building a toolchain can be done using crosstool-ng , Let's see how

this process will go:

- 1- Clone the repository.
- 2- Search for suitable configurations.
- 3- Set the default configurations according to the board we're using.

4- Apply all the necessary edits through menuconfig (Kconfig, which is a GUI that will provide us with helpful tools to output the .config file).

5- Build.

4.5.2.3 Building a bootloader (U-boot)

Let's first take a look at U-boot or the Universal Boot Loader and understand what it is. It supports a wide range of microprocessors like MIPS, ARM, PPC, Blackfin, AVR32 and x86. It also supports different methods of booting, which is neat, it can support booting from USB, SD card, NOR and NAND flash (non-volatile memory). It can also boot Linux kernel from the network using TFTP. U-boot also provides a command line interface which gives us very easy access to it and many different things.

U-boot is the second-stage bootloader, responsible for loading the kernel and giving it control. When U-boot is compiled, we get u-boot.img.

To use U-boot on our target device, we will follow the same steps as in the toolchain. First, we will download the source code, which is of course open source and available on GitHub. We will then identify then specify the architecture we will be using, which can be found either through the target device's name or the SOC in the config file. Now that we have specified the architecture, we also need to identify what family it's under (for example, if we were to use this method, our architecture goes under the ARM family).

Running the make command followed by the architecture and family will output the .config file.

The next step is to configure the U-boot configurations through menuconfig. After applying all necessary configurations, all that is left now is to build. So, we simply run the build command followed by the number of cores to utilize through the build process.

After building, we should now have the u-boot.bin file

4.5.2.4 Building a Linux Kernel

First of all, a kernel can be a zimage or a uimage but it shouldn't really matter for us as all of them are just different ways to store.

Recall the function of a kernel in Embedded Linux:

- Runs and schedules different processes.
- Manages resources.
- Manages memory.
- Handles interrupts and multiusers.
- Provides filesystem, networking and security.

The kernel is, as one could imagine, a bunch of code (written in C) compiled to binary. This binary output which resembles the kernel doesn't have any information about the hardware configurations.

Instead, these configurations can be found in the Device Tree Binary (DTB). Note that DTB files were at first stored as Device Tree Source files (DTS) which were then compiled using a Device Tree Compile (DTC).

We will follow the same sequence we used in the toolchain and bootloader. First, we will download the source code through Github, as usual. Then we will specify the target device architecture, the family, and enable the cross compiler, all through the build command. All that is left now is to apply all the necessary configurations through menuconfig, as we did before, and to finally build.

4.5.2.5 Building the Filesystem

Up till now, we have discussed building a toolchain, bootloader (U-boot), Linux kernel, and now

we will talk about the filesystem. We will be following the same sequence we used to build the prior elements.

For the first step, instead of downloading some source code through Github, we will be installing a tool called BusyBox that will aid us in generating a root file system.

BusyBox is open-source and licensed under the GPL. BusyBox is a collection of core UNIX utilities packaged as a single binary. This makes it ideal for resource-constrained environments, which is just a fancy way to describe an embedded system.

The complete distribution has almost 400 of the most common commands.

BusyBox input and output can be used in a standard manner across different distributions and versions of Linux. This helps keep the system configuration files compatible between different versions of Linux and allows easy updating of the system without having to learn new commands or make modifications.

In addition, it can also help automate tedious tasks so less time is spent maintaining the system, saving valuable resources and time.

BusyBox provides the everyday convenience commands that often feel like they're part of your shell.

Although userland tools like `ls` and `cat` are ubiquitous, they actually reside in a separate utility package that's independent of your shell. Many Linux distributions deliver these commands via GNU's `coreutils` but others ship BusyBox instead.

4.5.2.6 Building the Application

Embedded Linux can run a wide variety of programming languages, it can run Python, C/C++, rust, etc. It all comes down to what application we want to develop.

Let's for example look at our case, where we had used the traditional method, we would be running our code on a cross compiler to produce an executable that we'll run on our target device. Of course, the cross compiler is going

to differ according to the target device, as for Raspberry Pi, it will be arm cross compiler.

4.5.3 Using Yocto to build The Image

After exploring the traditional approach to customizing a Linux image, we will delve into the details of The Yocto Project, an alternative method of customizing a Linux image that we employed in our project.



Fig.25 Yocto Project

The Yocto Project is a comprehensive open-source embedded Linux build system that enables the rapid development of custom embedded Linux distributions and related components.

It provides an intuitive user interface to easily configure, create, and deploy these distributions to multiple target platforms.

With its wide array of tools and libraries, it simplifies the implementation of complex multi-level layers for product specific customization and board support packages (BSPs).

Furthermore, its adaptability helps manufacturers reduce time-to-market while increasing efficiency and extensibility in products with integrated Yocto components. By using packages such as Poky and OpenEmbedded Core (OE-Core), developers have access to preconfigured as well as customizable software stacks that can be used across multiple architectures including ARM, PowerPC, MIPS, x86 or x86_64 processors. That makes Yocto one of the most accepted open-source embedded build systems around.

The Yocto Project provides a compliant framework for creating customized Linux distributions for embedded and IoT devices. Its components include the OpenEmbedded-Core layer, which is a set of core recipes and classes to support complex image customization operations; the Poky layer, which offers technology to construct any type of Linux distribution; the BitBake build system which is responsible for the integration process of all other layers; Configurations files, used to interact effectively with multiple host systems to ensure easy cross-compilation, and Templates, providing helpful utilities and hooks enabling users to create very specific customizations. All said components allow the Yocto Project to offer developers advanced flexibility in building custom solutions for their applications in the embedded market. In the following paragraphs, we will delve deeper into the structure of the Yocto Project and all of its components.

Using reference templates, layers, build tools and machine configurations, it facilitates software integration activities across different hardware architectures. The project utilizes a shared metadata that allows developers to easily manage and customize the more than 3,000 electronic components supported by its ecosystem. Uniquely among Linux embedded platforms, Yocto provides developers with unparalleled flexibility across I/O devices, peripherals, and boards. In addition to established drivers and frameworks such as Linux kernel modules, multimedia codecs and hardware adaptation

capabilities relying on board support packages features from 3rd party vendors; all aspects of the development environment can be tailored or extended according to specific requirements.

The following figure illustrates the Yocto Project Environment.

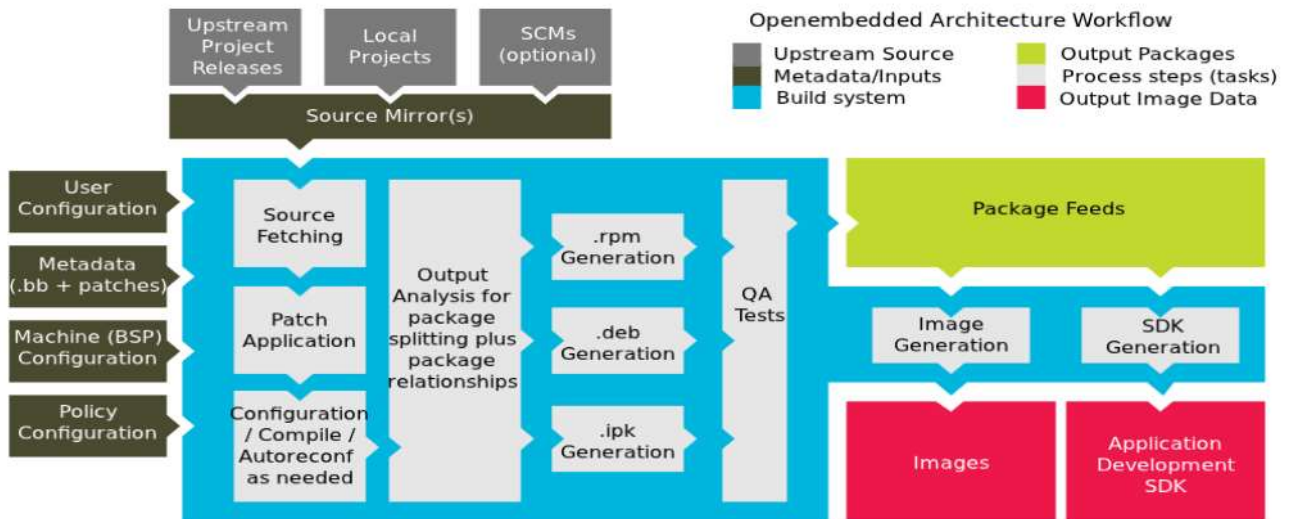


Fig.26 *Yocto Project Environment*

Now that we have untangled what the Yocto Project is, let's demonstrate the components and tools it's built upon.

4.5.3.1 Poky

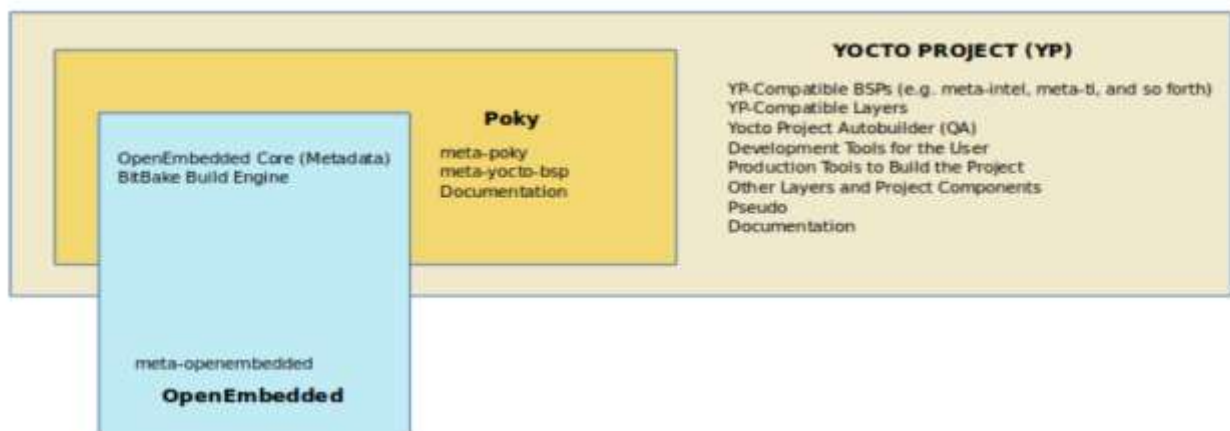


Fig.27 *Poky*

It is the reference distribution or the reference OS kit of the Yocto Project. It provides a standard set of metadata and tooling that serves as a foundation layer for other distributions and products in the project ecosystem.

Poky includes routines to build Embedded Linux systems with cross-compiling support, as well as applications tailored to embedded environments such as BitBake and OE Core. While Poky is a "complete" distribution specification and is tested and put through QA, we cannot use it as a product "out of the box" in its current form.

4.5.3.2 Board Support Package (BSP)

It is a collection of packages and recipes that allow operating systems to be customized for embedded systems. The BSP package includes all necessary files for building a customized operating system for an embedded board, such as board-specific device support, graphics drivers, startup scripts, bootloaders, and kernel modules. Poky supports multiple BSPs by default like x86, Beaglebone, and Freescale platform.

4.5.3.3 Meta Data

It is basically the layers about a specific hardware or a software package. It abstracts away all the cross-compilation complexity and makes it easy for users to simply identify what packages are needed. This reduces development time by streamlining component selection, enabling faster delivery of products built around the Yocto Project technology stack. Meta data can be:

- Configurations Files (.conf) like Machine type, packages to be installed, etc.
- Recipes are the most common form of metadata. A recipe contains a list of settings and tasks (i.e. instructions) for building packages that are then used to build the binary image. A recipe describes where you get source code and which patches to apply. Recipes describe dependencies for libraries or for other recipes as well as configuration and compilation options.

Related recipes are consolidated into a layer. Recipes (.bb or .bbappend) (BitBake) files are makefile-like recipes used within the Yocto Project to build packages. Each recipe file contains all the important information to build a component, including what source code is needed for the component, how it should be configured, and which dependencies must be satisfied before building it. Furthermore, bb files also contain instructions on how to package and install a component once it has been built. Additionally, since source code typically used in Yocto projects is often very complicated or infrequently used, bb files help developers focus on only the specific parts that really matter when dealing with them.

- Classes (.bbclass) files enable developers to create their own recipes using classes. Class files provide a means to share common code and configuration options between packages. This allows developers to reduce the number of lines of code they need to write, while also avoiding the duplication of code, which can lead to error-prone applications. Additionally, with class files, developers can quickly access settings and variables shared across multiple packages without having to look up each setting separately.
- Includes (.inc) files can be used to configure the kernel, add software packages, or modify system settings. Additionally, they can also store recipes in order to define how a specific package would be built during image generation.

4.5.3.4 Open-Embedded Core (OE-Core)

It provides developers with the necessary tools to build their own customized Linux distributions for embedded targets. OE-Core takes care of dependency management and cross compilation support. This ensures that platforms such as ARM can run sophisticated applications while still utilizing minimal system requirements. With its powerful yet easy to use feature set, Yocto's Open- Embedded Core makes using Linux on embedded devices easier than ever.

4.5.3.4 BitBake

It is a powerful tool used by the Yocto Project to support cross-platform, embedded Linux development. At its core, it takes project metadata and recipes as inputs, executes user commands and tasks, and outputs binary packages. Furthermore, it is a sophisticated build system that can handle dynamic runtime dependencies between tasks during execution via an internal dependency engine.

4.5.3.5 Layer

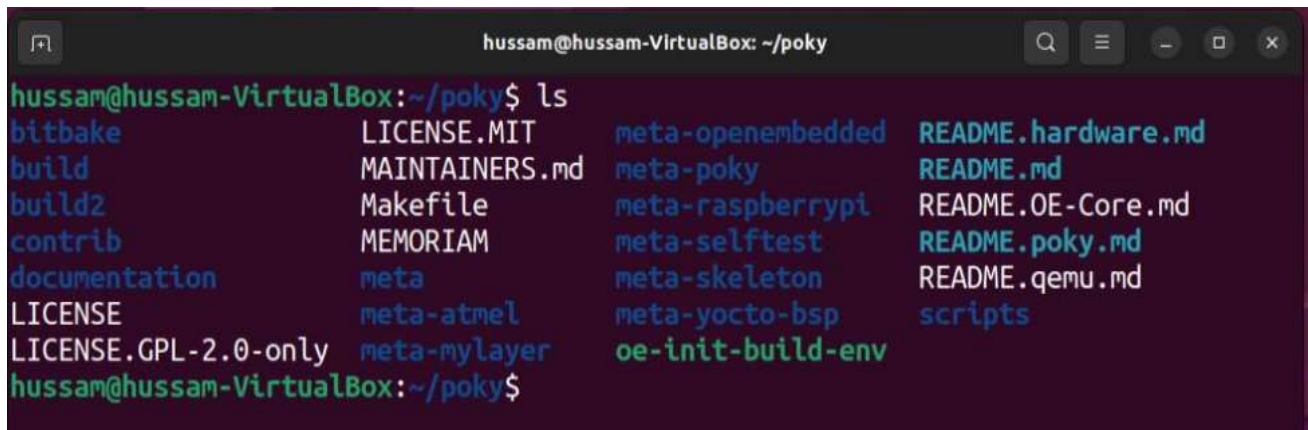
A collection of related recipes. Layers allow us to consolidate related metadata to customize our build. Layers also isolate information used when building for multiple architectures. Moreover, Layers are hierarchical in their ability to override previous specifications. We can include any number of available layers from the Yocto Project and customize the build by adding your layers after them. We can search the Layer Index for layers used within Yocto Project.

4.5.3.6 Packages

In the context of the Yocto Project, this term refers to a recipe's packaged output produced by BitBake (i.e., a "baked recipe"). A package is generally the compiled binaries produced from the recipe's sources. We "bake" something by running it through BitBake.

4.5.3.7 QEMU

It is an open-source virtual machine developed by the Linux Foundation in cooperation with Intel and is well-suited for use with the Yocto Project. By utilizing QEMU in an embedded system, it allows systems engineers a safe environment to conduct experiments without interfering with actual hardware or its real-time behavior. Simply, it acts as a simulator to run an image to test it and to know whether it is working or not and to know if there are any bugs. To be able to use Yocto Project, Poky is downloaded to the Host Machine and its file structure contains:

A terminal window titled 'hussam@hussam-VirtualBox: ~/poky' showing the output of the 'ls' command. The files are listed in four columns. The first column contains: bitbake, build, build2, contrib, documentation, LICENSE, LICENSE.GPL-2.0-only, and hussam@hussam-VirtualBox:~/poky\$. The second column contains: LICENSE.MIT, MAINTAINERS.md, Makefile, MEMORIAM, meta, meta-atmel, meta-mylayer, and hussam@hussam-VirtualBox:~/poky\$. The third column contains: meta-openembedded, meta-poky, meta-raspberrypi, meta-selftest, meta-skeleton, meta-yocto-bsp, oe-init-build-env, and hussam@hussam-VirtualBox:~/poky\$. The fourth column contains: README.hardware.md, README.md, README.OE-Core.md, README.poky.md, README.qemu.md, scripts, and hussam@hussam-VirtualBox:~/poky\$.

```
hussam@hussam-VirtualBox: ~/poky
hussam@hussam-VirtualBox:~/poky$ ls
bitbake          LICENSE.MIT      meta-openembedded  README.hardware.md
build           MAINTAINERS.md  meta-poky          README.md
build2          Makefile        meta-raspberrypi   README.OE-Core.md
contrib         MEMORIAM        meta-selftest      README.poky.md
documentation   meta            meta-skeleton      README.qemu.md
LICENSE         meta-atmel      meta-yocto-bsp     scripts
LICENSE.GPL-2.0-only meta-mylayer    oe-init-build-env
hussam@hussam-VirtualBox:~/poky$
```

Fig.28 Poky File Structure

- meta

It is the Open-Embedded meta data.

- meta-poky

It is the meta data of Yocto Project.

- meta-selftest

Meta data used while building and testing the image, but it is not included in the image.

- meta-skeleton

A template used as a layer example.

- meta-yocto-bsp

It contains the BSPs supported by poky like x86, BeagleBone and FreeScale.

4.6 Installing Yocto And Building The Image

4.6.1 Setting up the Yocto Environment

To use Yocto Project on our host machine, we must first make sure that the following requirements are met:

1. A host system with a minimum of 50 GB of free disk space running a supported Linux distribution (Ubuntu, Debian, Fedora, etc...)
2. The like git, gcc, python3, and more on the host which are essential for Yocto to generate the image.

For our project, we will be using Ubuntu 22.04.

The following command will install said packages based on an Ubuntu distribution:

```
$ sudo apt install gawk wget git diffstat unzip texinfo gcc build-essential chrpath socat cpio python3 python3-pip python3-pexpect xz-utils debianutils iputils-ping python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev pylint3 xterm python3-subunit mesa-common-dev zstd liblz4-tool
```

Once the setup is complete, the next step is to install a copy of the Poky repository (which is a reference distribution of the Yocto Project that contains the OpenEmbedded Build System, BitBake, and OpenEmbedded Core, as well as a set of metadata to get you started building our own distro) on our build host using the following command:

```
$ git clone git://git.yoctoproject.org/poky
```



```
ubuntu@ubuntu: ~/EmbeddedLinux/YOCTO/poky$ ls
Graduation_rpi5  MAINTAINERS.md  README.hardware.md  SECURITY.md  corrupted.db  meta-ui  meta-features  meta-poky  meta-selftest  raspi_images
LICENSE          MEMORIAM        README.md           bitbake      documentation  meta-apps  meta-freescale  meta-python2  meta-skeleton  scripts
LICENSE.GPL-2.0-only  Makefile        README.poky.md      clean_new.db  dump.sql      meta-atmel  meta-grad-distro  meta-qt5  meta-yocto-bsp  shared_yocto_space
LICENSE.MIT      README.OE-Core.md  README.qemu.md      contrib      meta          meta-digi  meta-openembedded  meta-raspberrypi  oe-init-build-env
```

Fig.29 Poky Installation Files

The last step in the installation process is to choose the release, which we will be working on, through their corresponding code name, each representing a branch from the Poky repository with its own support lifetime. For our project, we are using Kirkstone.

By running the above commands, we now have set up our Yocto Project environment and are ready to create our own distro and layer.

4.6.2 Initialize the Build Environment

As mentioned above, Yocto can support any number of targeted builds for different embedded devices. Each of these builds however had to reside inside its own build directory. Yocto provides a script to set up and/or a user-specific build directory.

From within the poky directory, we run the `oe-init-build-env` environment setup script to define Yocto Project's build environment on our build host.

```
$ cd ~/yocto/poky
```

```
$ source oe-init-build-env Graduation-rpi5
```

`Oe-init-build-env` Is the environment setup script that will set up the local path and other variables for your build directory.

`Graduation-rpi5` will be the name of the build directory. Note that hadn't we specified the build directory name, it would have been set to build by default. Also note that we can source the script by also running `$. oe-init-build-env Graduation-rpi5`

The first call to the script with a new directory name will create the directory and its contents, all subsequent calls will only set the environment variables accordingly. We have to call `source oe-init-build-env` once per terminal session, otherwise building and many other tools will not work.

The `source` command runs the contents of the script as if typed onto the current shell. This includes directory changes, thus running the `source oe-init-build-env` script will

leave you inside the build directory. Looking at the build directory we have, we'll find the following files:

1) **Conf:** it contains the following 3 files:

- **Bblayers.conf:** before the OpenEmbedded build system can use our new layer, we need to enable it. And to enable it, we simply add our layer's path to the **BBLAYERS** variable in our `conf/bblayers.conf` file.

```
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
/home/ubuntu/EmbeddedLinux/YOCTO/poky/meta \
/home/ubuntu/EmbeddedLinux/YOCTO/poky/meta-poky \
/home/ubuntu/EmbeddedLinux/YOCTO/poky/meta-yocto-bsp \
/home/ubuntu/EmbeddedLinux/YOCTO/poky/meta-raspberrypi \
/home/ubuntu/EmbeddedLinux/YOCTO/poky/meta-grad-distro \
/home/ubuntu/EmbeddedLinux/YOCTO/poky/meta-ai \
/home/ubuntu/EmbeddedLinux/YOCTO/poky/meta-openembedded/meta-oe \
/home/ubuntu/EmbeddedLinux/YOCTO/poky/meta-openembedded/meta-python \
/home/ubuntu/EmbeddedLinux/YOCTO/poky/meta-openembedded/meta-networking \
/home/ubuntu/EmbeddedLinux/YOCTO/poky/meta-features \
/home/ubuntu/EmbeddedLinux/YOCTO/poky/meta-openembedded/meta-multimedia \
/home/ubuntu/EmbeddedLinux/YOCTO/poky/meta-apps \
/home/ubuntu/EmbeddedLinux/YOCTO/poky/meta-python2 \
"
```

Fig.30 *bblayer.conf file*

Layers can also be added through a bitbake command:

```
$ bitbake-layers add-layer
```

- **Local.conf:** is a powerful tool that can configure almost every aspect of the building process.

Through `local.conf` we can set the following

it contains the used bit-bake layers in `bblayers.conf` and the build configurations like machine type, package classes, configuration version and many more in `local.conf`.

- a. Set the machine which we will be working on (raspberrypi5 in our case). There is a various selection of emulated machine available beside the QEMU emulator.
- b. Set where to place downloads directory. As during the first build the system will download many different source code tarballs from various upstream projects. This can take a while, particularly if the network connection is slow. These are all stored in `DL_DIR`. When wiping and rebuilding we can preserve this directory to speed up this part of subsequent builds. This directory is safe to share between multiple builds on the same machine too.
- c. Set where to place the shared-state files directory. Since BitBake has the capability to accelerate builds based on previously built output. This is done using "shared state" files which can be thought of as cache objects and this option determines where those files are placed. We can wipe out `TMPDIR` leaving this directory intact and the build would regenerate from these files if no changes were made to the configuration. If changes were made to the configuration, only shared state files where the state was still valid would be used (done using checksums).
- d. Set where to place the build output (tmp file). This option specifies where the bulk of the building work should be done and where BitBake should place its temporary files and output.

Keep in mind that this includes the extraction and compilation of many applications and the toolchain which can use Gigabytes of hard disk space.

- e. Set the distribution settings. The distribution setting controls which policy settings are used as defaults. The default value (Poky) is fine for general Yocto project use, at least initially.

Ultimately when creating custom policy, people will likely end up subclassing these defaults.

- f. Enable/Disable package management configurations.
- g. Add extra packages to the generated images through EXTRA_IMAGE_FEATURES.
- h. QEMU configurations in case we're using QEMU.
- i. Optimization options for maximum build time.

In addition to many other features, these were the main ones.

The following figures shows what's inside the local.conf file:

```
#MACHINE ?= "beaglebone-yocto"
#MACHINE ?= "genericx86"
#MACHINE ?= "genericx86-64"
#MACHINE ?= "edgerouter"
#
# This sets the default machine to be qemux86-64 if no other machine is selected:
MACHINE ??= "raspberrypi5"

#
# Where to place downloads
#
# During a first build the system will download many different source code tarballs
# from various upstream projects. This can take a while, particularly if your network
# connection is slow. These are all stored in DL_DIR. When wiping and rebuilding you
# can preserve this directory to speed up this part of subsequent builds. This directory
# is safe to share between multiple builds on the same machine too.
#
# The default is a downloads directory under TOPDIR which is the build directory.
#
#DL_DIR ?= "${TOPDIR}/downloads"
DL_DIR ?= "${TOPDIR}/../shared_yocto_space/downloads"

# Where to place shared-state files
#
# BitBake has the capability to accelerate builds based on previously built output.
# This is done using "shared state" files which can be thought of as cache objects
# and this option determines where those files are placed.
#
# You can wipe out TMPDIR leaving this directory intact and the build would regenerate
# from these files if no changes were made to the configuration. If changes were made
# to the configuration, only shared state files where the state was still valid would
# be used (done using checksums).
#
# The default is a sstate-cache directory under TOPDIR.
#
#SSTATE_DIR ?= "${TOPDIR}/sstate-cache"
SSTATE_DIR ?= "${TOPDIR}/../shared_yocto_space/state-cache"
```

Fig.31 local.conf file


```

"
# Default policy config
#
# The distribution setting controls which policy settings are used as defaults.
# The default value is fine for general Yocto project use, at least initially.
# Ultimately when creating custom policy, people will likely end up subclassing
# these defaults.
#
DISTRO ?= "grad"
# As an example of a subclass there is a "bleeding" edge policy configuration
# where many versions are set to the absolute latest code from the upstream
# source control systems. This is just mentioned here as an example, its not
# useful to most new users.
# DISTRO ?= "poky-bleeding"

#
# Package Management configuration
#
# This variable lists which packaging formats to enable. Multiple package backends
# can be enabled at once and the first item listed in the variable will be used
# to generate the root filesystems.
# Options are:
# - 'package_deb' for debian style deb files
# - 'package_ipk' for ipk files are used by opkg (a debian style embedded package manager)
# - 'package_rpm' for rpm style packages
# E.g.: PACKAGE_CLASSES ?= "package_rpm package_deb package_ipk"
# We default to rpm:
PACKAGE_CLASSES ?= "package_rpm"

# CONF_VERSION is increased each time build/conf/ changes incompatibly and is used to
# track the version of this file when it was generated. This can safely be ignored if
# this doesn't mean anything to you.
CONF_VERSION = "2"

BB_NUMBER_THREADS="4"
PARALLEL_MAKE="-j 4"
#####3 Rasberry pi configurations #####

RPI_KERNEL_DEVICETREE_OVERLAYS+= "overlays/uart2-pi5.dtbo"
RPI_KERNEL_DEVICETREE_OVERLAYS+= "overlays/uart3-pi5.dtbo"
RPI_KERNEL_DEVICETREE_OVERLAYS+= "overlays/uart4-pi5.dtbo"

# GPU memory (minimal for USB cam)
GPU_MEM = "64"

# Auto-load USB camera driver
KERNEL_MODULE_AUTOLOAD += "uvcvideo"
MACHINE_ESSENTIAL_EXTRA_RECOMMENDS += "kernel-module-uvcvideo"

INHIBIT_PACKAGE_STRIP = "1"
INHIBIT_PACKAGE_DEBUG_SPLIT = "1"
IMAGE_LOCALES_ARCHIVE = "0"

LICENSE_FLAGS_ACCEPTED = "commercial"

```

Fig.32 local.conf file

2) Downloads: it contains the downloaded files and packages that will be installed to the custom image.

3) Sstate-cache: it contains the shared state cache. It will be used later to act as a base for multiple builds. This directory can be moved to another location and specify in the build configuration of the image where it is through the `SSTATE_DIR` variable.

4) Tmp: The OpenEmbedded build system creates and uses this directory for all build system's output. BitBake creates this directory if it does not exist. As a last resort, to clean up a build and start it from scratch (other than the downloads), we can remove everything in the tmp directory or get rid of the directory completely. Note that if we do so, we should also completely remove the state-cache. tmp also contains the following sub directories:

- **buildstats:** stores the build statistics.
- **cache:** after bitbake parses the metadata (recipes and configuration files), it caches the results tmp/cache to speed up future builds. During subsequent builds, Bitbake checks each recipe (together with, for example, any files included or appended to it) to see if they have been modified. Changes can be detected, for example, through file modification time (mtime) changes and hashing of file contents. If no changes to the file are detected, then the parsed result stored in the cache is reused. If the file has changed, it is reparsed.
- **deploy:** This directory contains any “end result” output from the OpenEmbedded build process. The `DEPLOY_DIR` variable points to this directory.
- **deb:** This directory receives any .deb packages produced by the build process. The packages are sorted into feeds for different architecture types.
- **rpm:** This directory receives any .rpm packages produced by the build process. The packages are sorted into feeds for different architecture types.

- `ipk`: This directory receives `.ipk` packages produced by the build process.
- `licenses`: This directory receives package licensing information. For example, the directory contains sub-directories for `bash`, `busybox`, and `glibc` (among others) that in turn contain appropriate `COPYING` license files with other licensing information.
- `Images`: This directory is populated with the basic output objects of the build (think of them as the “generated artifacts” of the build process), including things like the boot loader image, kernel, root filesystem, and more. If we want to flash the resulting image from a build onto a device, look here for the necessary components. Note that when deleting files in this directory.

We can safely delete old images from this directory (e.g. `core-image-*`). However, the kernel (`*zImage*`, `*uImage*`, etc.), bootloader, and other supplementary files might be deployed here prior to building an image. Because these files are not directly produced from the image, if we delete them they will not be automatically re-created when we build the image again.

- `sdk`: The OpenEmbedded build system creates this directory to hold toolchain installer scripts which, when executed, install the sysroot that matches your target hardware.
- `sstate-control`: The OpenEmbedded build system uses this directory for the shared state manifest files. The shared state code uses these files to record the files installed by each `sstate` task so that the files can be removed when cleaning the recipe or when a newer version is about to be installed. The build system also uses the manifests to detect and produce a warning when files from one task are overwriting those from another.
- `sysroots-components`: This directory is the location of the sysroot contents that the task `do_prepare_recipe_sysroot` links or copies into the recipe-specific sysroot for each recipe listed in `DEPENDS`. Population of this directory is handled through shared state, while the path is specified by the `COMPONENTS_DIR` variable. Apart from a few

unusual circumstances, handling of the `sysroots-components` directory should be automatic, and recipes should not directly reference `build/tmp/sysroots-components`.

- **sysroots:** Previous versions of the OpenEmbedded build system used to create a global shared sysroot per machine along with a native sysroot.
- **stamps:** This directory holds information that BitBake uses for accounting purposes to track what tasks have run and when they have run. The directory is sub-divided by architecture, package name, and version.
- **work:** This directory contains architecture-specific work sub-directories for packages built by BitBake. All tasks are executed from the appropriate work directory.

For example, the source for a particular package is unpacked, patched, configured, and compiled all within its own work directory.

Within the work directory, organization is based on the package group and version for which the source is being compiled as defined by the `WORKDIR`.

- **work-shared:** For efficiency, the OpenEmbedded build system creates and uses this directory to hold recipes that share a work directory with other recipes. In practice, this is only used for `gcc` and its variants (e.g. `gcc-cross`, `libgcc`, `gcc-runtime`, and so forth).

4.6.3 Creating our own Layer

Instead of adding the required packages and applications to our image in `local.conf` file, present the build directory, which is not considered a good practice. As by doing so, any image will be built with these added packages which is not what we want, we will create a layer of our own to be able to add specific packages and applications to it. Whenever an image is built, we can add this layer to the `bblayers.conf` file which includes all the added layers to the image. This way the added packages and applications are abstracted so that whenever they are needed, we can just include it in the `bblayers.conf` file.

There are various ways to create a layer:

- The manual way is to use the meta-skeleton folder which acts like an example of a new layer

and provides some examples of the recipes to be used.

- The automated way is to use the bitbake to automatically create a layer:

```
$ bitbake-layers create-layer meta-mylayer
```

```
ubuntu@ubuntu:~/EmbeddedLinux/YOCTO/poky/meta-ai$ tree -L 4
.
├── COPYING.MIT
├── README
├── conf
│   └── layer.conf
├── recipes-example
│   └── example
│       └── example_0.1.bb
├── recipes-model
│   └── model
│       ├── model
│       │   ├── model.service
│       │   ├── model_int8_openvino_model_H
│       │   └── track.py
│       └── model_1.0.bb
```

Fig.33 *Content of our meta-ai layer*

```
ubuntu@ubuntu:~/EmbeddedLinux/YOCTO/poky/meta-grad-distro$ tree -L 4
.
├── COPYING.MIT
├── README
├── conf
│   ├── distro
│   │   ├── grad.conf
│   │   └── include
│   │       └── systemd.inc
│   └── layer.conf
├── recipes-core
│   └── images
│       └── grad-test-image.bb
├── recipes-example
│   └── example
│       └── example_0.1.bb
```

Fig.34 Content of our meta-grad-distro layer

```
ubuntu@ubuntu:~/EmbeddedLinux/YOCTO/poky/meta-apps$ tree -L 5
.
├── COPYING.MIT
├── README
├── conf
│   └── layer.conf
├── recipes-arduino
│   └── arduino
│       └── arduino_1.0.bb
├── recipes-example
│   └── example
│       └── example_0.1.bb
├── recipes-main
│   └── mainApplication
│       ├── mainApplication
│       │   └── main.service
│       └── mainApplication_1.0.bb
├── recipes-osm
│   └── openstreetMap
│       ├── openstreetMap
│       │   ├── app.py
│       │   ├── app.service
│       │   ├── db.service
│       │   ├── initialize_db.py
│       │   ├── request.py
│       │   └── templates
│       │       └── map.html
│       └── openstreetMap_1.0.bb
├── recipes-v2c
│   └── vehicleToCloud
│       └── vehicleToCloud_1.0.bb
├── recipes-v2v
│   └── vehicleToVehicle
│       └── vehicleToVehicle_1.0.bb
```

Fig.35 Content of our meta-apps layer

We used the automated way to create our own layer. This layer contains:

- Multiple recipes which are organized so that each set of recipes with the same description or configuration are in one folder as:

recipes-core: It contains recipes related to the image.

recipes-kernel: It contains recipes related to the kernel.

recipes-support: It contains some recipes which add some configuration to an already made recipes inside another layer.

- conf folder: It contains the layer configurations and our own custom-made distribution.

layer.conf: This file contains the layer configurations like the compatible Yocto releases, the location of the recipes (.bb) files and some other configurations.

distro folder: This folder contains the recipes that is responsible for the custom-made distro which will be discussed later.

4.6.4 Creating our own Linux Distribution

Now, we will create our own Linux Distribution (often shortened to ‘Linux Distro’) which is basically a version of the open-source Linux operating system that is packed with other components, such as installation programs, management tools, and additional software. Our distribution will be based on Yocto’s Poky distribution.

The custom distro will contain some extra information about the distro itself as (Name, Version, Code Name, SDK Vendor, SDK Version, Maintainer and Target Vendor) each of these information are already present in the Yocto’s Poky Distro so by adding these info in our distro, it overwrites the data already present inside Poky distribution.

In addition to the basic distro information, we will be adding some extra configurations to be included in the distro, like systemd, wifi, etc.

The distribution configuration file needs to be created in the `conf/distro` directory of our layer. We will be naming it using our distribution name (e.g. `grad.conf`).

The `DISTRO` variable in our `local.conf`, present inside the build directory, file determines the name of our distribution.

4.6.5 Configuring our Layer

We will configure the files created inside the layer that we created.

- `recipes-core`: It contains the recipe which is responsible for building our own image inside images directory and it includes the packages to be included in our image. The image recipe is called “`grad-test-image.bb`”.
- `recipes-kernel`: It contains recipes related to the kernel like device drivers.
- `recipes-model`: it contains the model script.
- `recipes-arduino`: it contains Arduino code for the main car so that when the Raspberry Pi detects damage, it sends to the main car to stop.
- `recipes-main`: it contains the main app in which during the startup of the raspberry pi it runs the three apps (`v2v`, `v2c`, `osm`).
- `recipes-osm`: it contains `osm` app.
- `recipes-v2v`: it contains `v2v` app.
- `recipes-v2c`: it contains `v2c` app.

4.7 Bitbake

4.7.1 Introduction

Fundamentally, BitBake is a generic task execution engine that allows shell and Python tasks to be run efficiently and in parallel while working within complex inter-task dependency constraints.

One of BitBake's main users, OpenEmbedded, takes this core and builds embedded Linux software stacks using a task-oriented approach.

Conceptually, BitBake is like GNU Make in some regards but has significant differences:

- BitBake executes tasks according to the provided metadata that builds up the tasks. Metadata is stored in recipe (.bb) and related recipe “append” (.bbappend) files, configuration (.conf) and underlying include (.inc) files, and in class (.bbclass) files. The metadata provides BitBake with instructions on what tasks to run and the dependencies between those tasks.
- BitBake includes a fetcher library for obtaining source code from various places such as local files, source control systems, or websites.
- The instructions for each unit to be built (e.g. a piece of software) are known as “recipe” files and contain all the information about the unit (dependencies, source file locations, checksums, description and so on).
- BitBake includes a client/server abstraction and can be used from a command line or used as a service over XML-RPC and has several different user interfaces.

4.7.1.1 Building our image

```
ubuntu@ubuntu:~/EmbeddedLinux/YOCTO/poky$ bitbake grad-test-image
Loading cache: 100% |#####| Time: 0:00:33
Loaded 4687 entries from dependency cache.
WARNING: /home/ubuntu/EmbeddedLinux/YOCTO/poky/meta-apps/recipes-v2v/vehicleToVehicle/vehicleToVehicle_1.0.bb: QA Issue: PN: vehicleToVehicle is upper case, this can result in unexpected
behavior. [uppercase-pn]
WARNING: /home/ubuntu/EmbeddedLinux/YOCTO/poky/meta-apps/recipes-v2c/vehicleToCloud/vehicleToCloud_1.0.bb: QA Issue: PN: vehicleToCloud is upper case, this can result in unexpected behavi
or. [uppercase-pn]
WARNING: /home/ubuntu/EmbeddedLinux/YOCTO/poky/meta-apps/recipes-main/mainApplication/mainApplication_1.0.bb: QA Issue: PN: mainApplication is upper case, this can result in unexpected be
havior. [uppercase-pn]
Parsing recipes: 100% |#####| Time: 0:00:03
Parsing of 3828 .bb files complete (3016 cached, 4 parsed). 4690 targets, 562 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION      = "2.0.0"
BUILD_SYS       = "x86_64-linux"
NATIVELSBSTRING = "ubuntu-22.04"
TARGET_SYS      = "aarch64-oe-linux"
MACHINE         = "raspberrypi5"
DISTRO          = "grad"
DISTRO_VERSION  = "1.0"
TUNE_FEATURES   = "aarch64 armv8-2a crypto cortexa76"
TARGET_FPU      = ""
meta
meta-poky
meta-yocto-bsp   = "kirkstone:3810d71ad8ee9dee94903901c87cd0b642425cd1"
meta-raspberrypi = "kirkstone:9e12ad97b4c95772c6f483b9318f2bec2ab09e53"
meta-grad-distro
meta-ai          = "kirkstone:3810d71ad8ee9dee94903901c87cd0b642425cd1"
meta-oe
meta-python
meta-networking  = "kirkstone:45bdd258a3d1ded925faf8389e01bb948dc7f5b"
meta-features    = "kirkstone:3810d71ad8ee9dee94903901c87cd0b642425cd1"
meta-multimedia  = "kirkstone:45bdd258a3d1ded925faf8389e01bb948dc7f5b"
meta-apps        = "kirkstone:3810d71ad8ee9dee94903901c87cd0b642425cd1"
meta-python2     = "kirkstone:f02882e2aa9279ca7becca8b0cedbffe88b5a253"
meta-freescale   = "kirkstone:ccb66d5754fb3027670e15d33b5d683daffaf2eb"

Initialising tasks: 100% |#####| Time: 0:00:07
Sstate summary: Wanted 232 Local 216 Mirrors 0 Missed 16 Current 2323 (93% match, 99% complete)
Removing 1 stale sstate objects for arch raspberrypi5: 100% |#####| Time: 0:00:00
Removing 5 stale sstate objects for arch cortexa76: 100% |#####| Time: 0:00:00
WARNING: Please look at the following tasks:
```

Fig.36 *Bitbake*

From within the Poky Git repository, we can use the following command to display the list of directories within the source directory that contain image recipe files.

```
ubuntu@ubuntu:~/EmbeddedLinux/YOCTO/poky/Graduation_rpi5/tmp-glibc/deploy/images/raspberrypi5$ ls
grub-efi-bootaa64.efi          rpi-test-image-raspberrypi5-20250605182831.rootfs.wic.bz2  rpi-test-image-raspberrypi5.wic.bmap
linuxaa64.efi.stub            rpi-test-image-raspberrypi5-20250605182831.testdata.json  rpi-test-image-raspberrypi5.wic.bz2
rpi-test-image-raspberrypi5-20250605182831.rootfs.ext3      rpi-test-image-raspberrypi5.ext3                          rpi-test-image.env
rpi-test-image-raspberrypi5-20250605182831.rootfs.manifest  rpi-test-image-raspberrypi5.manifest                      systemd-bootaa64.efi
rpi-test-image-raspberrypi5-20250605182831.rootfs.tar.bz2   rpi-test-image-raspberrypi5.tar.bz2
rpi-test-image-raspberrypi5-20250605182831.rootfs.wic.bmap  rpi-test-image-raspberrypi5.testdata.json
```

Fig.37 *Our image*

4.7.2 Concepts

4.7.2.1 Recipes

BitBake Recipes, which are denoted by the file extension `.bb`, are the most basic metadata files. These recipe files provide BitBake with the following:

- Descriptive information about the package (author, homepage, license, and so on).
- The version of the recipe.
- Existing dependencies (both build and runtime dependencies).
- Where the source code resides and how to fetch it.
- Whether the source code requires any patches, where to find them, and how to apply them.
- How to configure and compile the source code.
- How to assemble the generated artifacts into one or more installable packages.
- Where on the target machine to install the package or packages created.

Within the context of BitBake, or any project utilizing BitBake as its build system, files with the `.bb` extension are referred to as recipes.

4.7.2.2 Configuration File

Configuration files, which are denoted by the `.conf` extension, define various configuration variables that govern the project's build process. These files fall into several areas that define machine configuration, distribution configuration, possible compiler tuning, general common configuration, and user configuration. The main configuration file is the sample `bitbake.conf` file, which is located within the BitBake source tree `conf` directory.

4.7.2.3 Classes

Class files, which are denoted by the `.bbclass` extension, contain information that is useful to share between metadata files. The BitBake source tree currently comes with one class metadata file called `base.bbclass`. You can find this file in the `classes` directory. The `base.bbclass` class file is special since it is always included automatically for all recipes and classes. This class contains definitions for standard basic tasks such as fetching, unpacking, configuring (empty by default), compiling (runs any Makefile present), installing (empty by default) and packaging (empty by default). These tasks are often overridden or extended by other classes added during the project development process.

4.7.2.4 Layers

Layers allow you to isolate different types of customizations from each other. While you might find it tempting to keep everything in one layer when working on a single project, the more modular your metadata, the easier it is to cope with future changes.

To illustrate how you can use layers to keep things modular, consider customizations you might make to support a specific target machine. These types of customizations typically reside in a special layer, rather than a general layer, called a Board Support Package (BSP) layer.

Furthermore, the machine customizations should be isolated from recipes and metadata that support a new GUI environment, for example. This situation gives you a couple of layers: one for the machine configurations and one for the GUI environment. It is important to understand, however, that the BSP layer can still make machine-specific additions to recipes within the GUI environment layer without polluting the GUI layer itself with those machine-specific changes. You can accomplish this through a recipe that is a BitBake append (`.bbappend`) file.

4.7.2.5 Append Files

Append files, which are files that have the `.bbappend` file extension, extend or override information in an existing recipe file.

BitBake expects every append file to have a corresponding recipe file. Furthermore, the append file and corresponding recipe file must use the same root filename. The filenames can differ only in the file type suffix used (e.g. `formfactor_0.0.bb` and `formfactor_0.0.bbappend`).

Information in append files extends or overrides the information in the underlying, similarly named recipe files.

When you name an append file, you can use the “%” wildcard character to allow for matching recipe names. For example, suppose you have an append file named as follows:

- `busybox_1.21.%.bbappend`

That append file would match any `busybox_1.21.x.bb` version of the recipe. So, the append file would match the following recipe names:

- `busybox_1.21.1.bb`
- `busybox_1.21.2.bb`
- `busybox_1.21.3.bb`

If the `busybox` recipe was updated to `busybox_1.3.0.bb`, the append name would not match.

However, if you named the append file `busybox_1.%.bbappend`, then you would have a match.

In the most general case, you could name the append file something as simple as

busybox_%.bbappend to be entirely version independent.

4.7.3 The BitBake Command

The bitbake command is the primary interface to the BitBake tool.

4.7.4 Execution

The primary purpose for running BitBake is to produce some kind of output such as a single installable package, a kernel, a software development kit, or even a full, board-specific bootable

Linux image, complete with bootloader, kernel, and root filesystem. Of course, you can execute the bitbake command with options that cause it to execute single tasks, compile single recipe files, capture or clear data, or simply return information about the execution environment.

4.7.5 Parsing the Base Configuration Metadata

The first thing BitBake does is parse base configuration metadata. Base configuration metadata consists of your project's bblayers.conf file to determine what layers BitBake needs to recognize, all necessary layer.conf files (one from each layer), and bitbake.conf. The data itself is of various types:

- Recipes: Details about particular pieces of software.
- Class Data: An abstraction of common build information (e.g. how to build a Linux kernel).
- Configuration Data: Machine-specific settings, policy decisions, and so forth. Configuration data acts as the glue to bind everything together.

The layer.conf files are used to construct key variables such as BBPATH and BBFILES. BBPATH are used to search for configuration and class files under the conf and classes directories respectively.

BBFILES is used to locate both recipe and recipe append files (.bb and .bbappend).

If there is no bblayers.conf file, it is assumed the user has set the BBPATH and BBFILES directly in the environment.

Next, the bitbake.conf file is located using the BBPATH variable that was just constructed. The bitbake.conf file may also include other configuration files using the include or require directives.

Prior to parsing configuration files, BitBake looks at certain variables, including:

- BB_ENV_PASSTHROUGH
- BB_ENV_PASSTHROUGH_ADDITIONS
- BB_PRESERVE_ENV
- BB_ORIGENV
- BITBAKE_UI

The first four variables in this list relate to how BitBake treats shell environment variables during task execution. By default, BitBake cleans the environment variables and provides tight control over the shell execution environment. However, through the use of these first four variables, you can apply your control regarding the environment variables allowed to be used by BitBake in the shell during execution of tasks.

The base configuration metadata is global and therefore affects all recipes and tasks that are executed.

BitBake first searches the current working directory for an optional conf/bblayers.conf configuration file. This file is expected to contain a BBLAYERS variable that is a space-delimited list of ‘layer’ directories. Recall that if BitBake cannot find a bblayers.conf file, then it is assumed the user has set the BBPATH and BBFILES variables directly in the environment.

For each directory (layer) in this list, a `conf/layer.conf` file is located and parsed with the `LAYERDIR` variable being set to the directory where the layer was found. The idea is these files automatically set up `BBPATH` and other variables correctly for a given build directory.

BitBake then expects to find the `conf/bitbake.conf` file somewhere in the user-specified `BBPATH`.

That configuration file generally has include directives to pull in any other metadata such as files specific to the architecture, the machine, the local environment, and so forth.

Only variable definitions and include directives are allowed in BitBake `.conf` files.

Some variables directly influence BitBake's behavior. These variables might have been set from the environment depending on the environment variables previously mentioned or set in the configuration files.

After parsing configuration files, BitBake uses its rudimentary inheritance mechanism, which is through class files, to inherit some standard classes. BitBake parses a class when the `inherit` directive responsible for getting that class is encountered.

The `base.bbclass` file is always included. Other classes that are specified in the configuration using the `INHERIT` variable are also included. BitBake searches for class files in a `classes` subdirectory under the paths in `BBPATH` in the same way as configuration files.

4.7.6 Locating and Parsing Recipes

During the configuration phase, BitBake will have set `BBFILES`. BitBake now uses it to construct a list of recipes to parse, along with any append files (`.bbappend`) to apply. `BBFILES` is a space-separated list of available files and supports wildcards. An example would be:


```
BBFILES = "/path/to/bbfiles/*.bb /path/to/appends/*.bbappend"
```

BitBake parses each recipe and append file located with BBFILES and stores the values of various variables into the datastore.

For each file, a fresh copy of the base configuration is made, then the recipe is parsed line by line.

Any inherit statements cause BitBake to find and then parse class files (.bbclass) using BBPATH as the search path. Finally, BitBake parses in order any append files found in BBFILES.

4.7.7 Dependencies

Each target BitBake builds consists of multiple tasks such as fetch, unpack, patch, configure, and compile.

At a basic level, it is sufficient to know that BitBake uses the DEPENDS and RDEPENDS variables when calculating dependencies.

CHAPTER 5

COMMUNICATION AND INTEGRATION SYSTEMS



5.1 Introduction

In the Advanced Road Safety Project, real-time communication between smart vehicles, cloud infrastructure, and mapping platforms is crucial for enhancing road safety. This chapter details the communication and integration mechanisms used to ensure efficient data exchange and timely alerts. The system is built around a primary smart vehicle equipped with a Raspberry Pi 5 (RPI 5), a camera, GPS module, ESP32, and an LCD display. When a road hazard such as a pothole is detected by an AI model running on the RPI 5, three critical actions are simultaneously triggered: 1) alert nearby vehicles via V2V communication using ESP-NOW, 2) update the pothole location on OpenStreetMap (OSM), and 3) report the incident to government agencies through V2C communication using MQTT protocol.

5.2 Vehicle-to-Vehicle (V2V) Communication

V2V communication enables smart vehicles to share hazard information directly with each other without relying on cloud infrastructure. This improves reaction time and situational awareness, especially in scenarios involving unexpected obstacles like potholes or uncovered manholes.



Fig.38 *V2V*

5.2.1 *ESP-NOW & ESP32 Integration*

The RPI 5 communicates with an onboard ESP32 via UART to relay detected road hazards. Upon detecting a pothole using the AI model, the RPI 5 sends an alert message to the ESP32 by using Uart, which in turn uses ESP-NOW to transmit this alert wirelessly to nearby vehicles equipped with their own ESP32 modules. Each receiving ESP32 then displays a warning on its connected LCD, alerting the driver of the approaching hazard.

ESP-NOW facilitates:

- Direct, low-latency communication between ESP32 modules.
- No Wi-Fi infrastructure required.
- Encryption for secure transmission.

This setup ensures instantaneous hazard propagation among vehicles, enhancing reaction time and road safety.



Fig.39 *ESP-NOW*

5.2.2 V2V Alert Scenarios

Examples of how V2V communication improves safety:

- **Real-Time Pothole Alert:** Car A detects a pothole. The RPI 5 sends a UART message to ESP32, which alerts Car B via ESP-NOW.
- **Driver Notification:** Both Car A and Car B display the alert on their LCDs, ensuring drivers are immediately informed.
- **Dynamic Hazard Relay:** As more vehicles receive and forward the alert, awareness spreads across a convoy of smart vehicles.

5.3 Vehicle-to-Cloud (V2C) Communication

5.3.1 Cloud Platforms and APIs

Alongside local communication, the RPI 5 publishes the pothole data to a cloud server using the MQTT protocol. This ensures that detected hazards are centrally logged and monitored by government agencies. The RPI 5 uses a lightweight MQTT client to publish messages that include:

- GPS coordinates
- Type of damage (e.g., pothole)
- Time of detection
- Optional camera snapshot

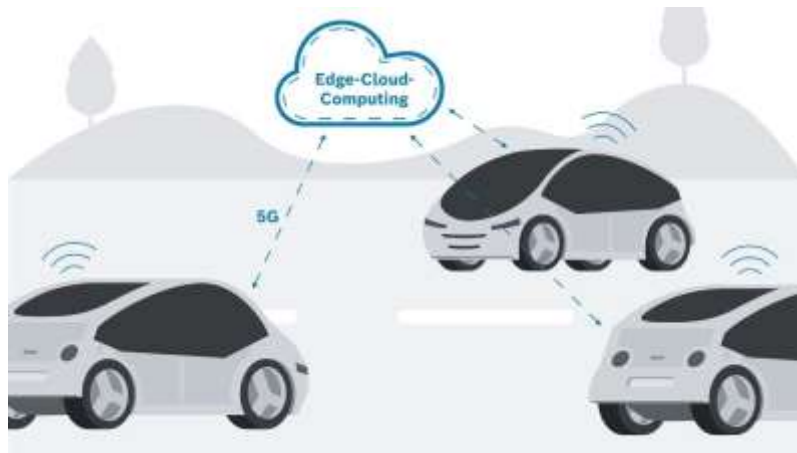


Fig.40 **V2C**

REST APIs or Firebase integration can be used for database storage and dashboard visualization.

5.3.2 ***Integration with Government Agencies***

Government agencies receive real-time alerts through a secured cloud dashboard. This enables:

- Quick dispatch of maintenance teams.
- Statistical insights for infrastructure planning.
- Prioritization of frequently reported hazard zones.

The system replaces slow manual reporting with automated, accurate, and timely updates.



5.3.3 MQTT Protocol

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol that provides resource-constrained network clients with a simple way to distribute telemetry information. The protocol, which uses a publish/subscribe communication pattern, is used for machine-to-machine (M2M) communication and plays an important role in the internet of things (IoT). The MQTT protocol is a good choice for wireless networks that experience varying levels of latency due to occasional bandwidth constraints or unreliable connections. The MQTT protocol surrounds two subjects: a client and a broker. An MQTT broker is a server, while the clients are the connected devices. When a device (or client) wants to send data to a server (or broker) it is called a publish. When the operation is reversed, it is called a subscribe. If the connection from a subscribing client to a broker is broken, then the broker will buffer messages and push them out to the subscriber when it is back online. If the connection from the publishing client to the broker is disconnected without notice, then the broker can close the connection and send subscribers a cached message with instructions from the publisher.

Advantages:

MQTT has a few distinct advantages when compared to competing protocols:

- Efficient data transmission and quick to implement due to its being a light weight protocol.
- Low network usage, due to minimized data packets.
- Efficient distribution of data. Successful implementation of remote sensing and control.
- Fast and efficient message delivery.
- Reduction of network bandwidth
- Usage of small amounts of power, which is good for the connected devices.

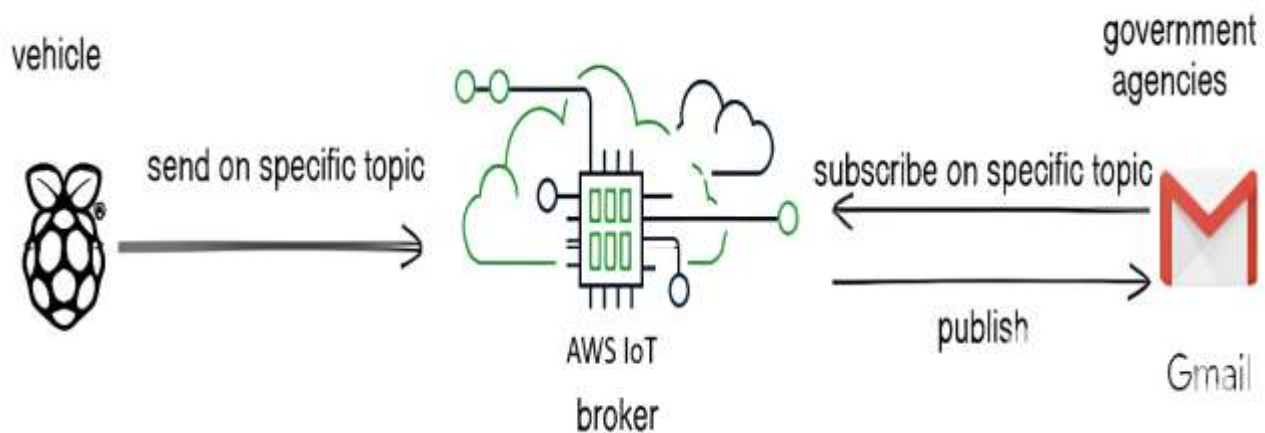


Fig.41 MQTT Overview

5.3.4 AWS iot core broker

AWS IoT Core is a managed cloud service provided by Amazon Web Services that enables connected devices such as sensors, vehicles, or embedded systems to interact securely and reliably with cloud applications and other devices. At the heart of this communication is the MQTT broker, which acts as a central hub for publishing and subscribing to messages between devices.

How AWS IoT Core Works as a Broker?

AWS IoT Core uses the MQTT (Message Queuing Telemetry Transport) protocol, a lightweight messaging protocol designed for low-bandwidth, high-latency networks. Devices (called clients) communicate using the publish/subscribe model:

- Publisher: A device or service that sends (publishes) a message to a specific topic.
- Subscriber: A device or service that listens (subscribes) to a topic to receive messages.
- AWS IoT Core acts as the broker, managing message delivery between publishers and subscribers. Devices can also use HTTP and WebSocket protocols, but MQTT is the most common for IoT due to its efficiency.

Key Features of AWS IoT Broker

- **Scalability:** Automatically scales to handle billions of messages from millions of devices.
- **Security:** Uses X.509 certificates, AWS IAM policies, and TLS encryption to authenticate and authorize devices. Ensures secure device-to-cloud and cloud-to-device communication.
- **Device Shadows:** AWS maintains a device shadow, a virtual representation of a device's last known state, which helps in managing and syncing device state even when offline.
- **Rules Engine:** Allows automatic routing of incoming messages to AWS services (like Lambda, DynamoDB, S3, etc.) for further processing, storage, or triggering actions.
- **Integration:** Easily integrates with analytics, machine learning, databases, and other AWS services to enable real-time insights and control.

In Advanced road safety system AWS IoT Core is used as a broker to:

- Receive real-time alerts about road surface damage (e.g., potholes, cracks) from smart vehicles.
- Generate automated damage reports.

- Forward reports to government agencies for maintenance action.



5.4 OpenStreetMap Integration

5.4.1 Live Road Condition Updates

When a pothole is detected, the RPI 5 obtains GPS coordinates via the onboard GPS module. These coordinates are then used to mark the hazard location on OpenStreetMap (OSM). This map update process uses tools like Leaflet.js or Mapbox to visualize the hazard publicly.

Each marker includes:

- Hazard type (e.g., pothole).
- Severity level (optional, based on detection confidence).
- Timestamp of detection.

This mapping informs non-smart vehicle users, cyclists, and pedestrians about current road conditions.



Fig.42 *OSM*

5.4.2 Visualization for Users

The road hazard data is presented on a user-friendly dashboard linked to the OSM interface. Features include:

- Interactive map with color-coded hazard markers.
- Detailed information on click.
- Filters by date, location, or type of hazard.

This allows users and city planners to view real-time road conditions and prioritize responses effectively.

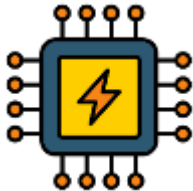
Map photo:

Fig.43 OSM map

This integrated system structure leverages both local (V2V) and global (V2C) communication mechanisms to transform a single detection event into a multi-layered response network. With components like the Raspberry Pi 5, ESP32, GPS, and LCD working in unison, and backed by powerful protocols like ESP-NOW and MQTT, the Advanced Road Safety Project ensures comprehensive monitoring and proactive intervention for urban road safety

CHAPTER 6

HARDWARE AND SOFTWARE DESIGN



6.1 Introduction

In the pursuit of enhancing road safety through smart vehicular technologies, the hardware backbone of the system plays a pivotal role in enabling real-time detection, alert dissemination, and data communication. This chapter delves into the technical specifications, purpose, and configuration of each hardware component and their interactions within the system, showcasing how their integration forms a cohesive and efficient safety framework.

At the core of the system lies a Raspberry Pi 5, integrated with a camera module for visual data capture, a GPS module for geolocation services, and an LCD display for real-time driver alerts. Additionally, ESP32 microcontrollers facilitate both inter-vehicle communication (V2V) and cloud integration (V2C), supported by MQTT protocol. The coordinated operation of these components allows for simultaneous detection of potholes, warning of nearby vehicles, updating of OpenStreetMap (OSM), and alerting government agencies for maintenance action.

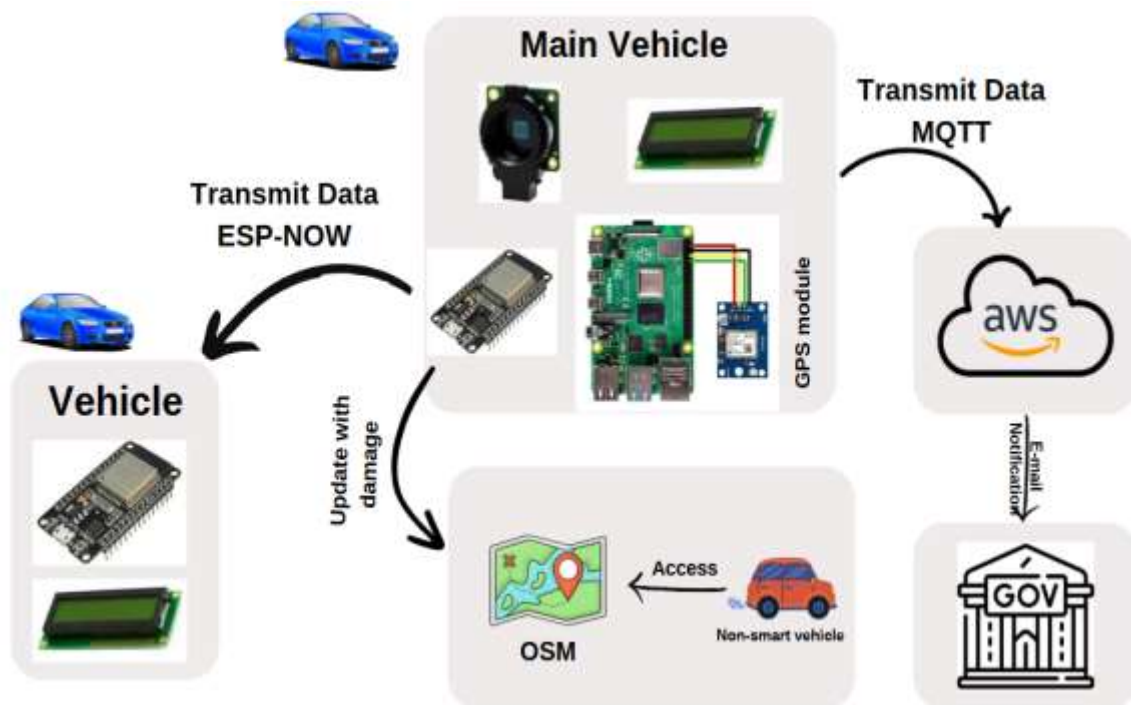
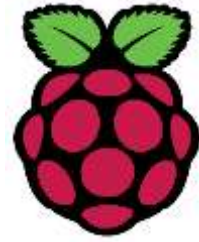


Fig.44 *System Architecture*

6.2 Hardware Overview



6.2.1 Raspberry Pi

6.2.1.1 Introduction

Raspberry Pi 5 is defined as a compact, powerful single-board computer roughly the size of a credit card that supports interoperability with a wide range of input and output devices such as monitors, TVs, keyboards, and sensors—essentially transforming any setup into a fully functional and low-cost computing system.

The core purpose of Raspberry Pi 5 is to promote learning in computing, electronics, and programming, as well as to serve as a robust platform for innovative embedded and AI-driven projects. With its enhanced processing power, improved I/O capabilities, and faster interfaces, it is well-suited for real-time and high-performance applications.

Some of the applications relevant to this project are described below:

Smart Road Damage Detection System

The Raspberry Pi 5 serves as the central processing unit in a road safety system that uses a camera module and machine learning models (e.g., YOLOv11) to detect potholes, cracks, and other road anomalies. The system processes frames in real-time, classifies road defects, and communicates alerts to nearby vehicles and government servers.

Vehicle-to-Vehicle (V2V) & Vehicle-to-Cloud (V2C) Communication

Using serial communication with ESP32 modules and the ESP-NOW protocol, the Raspberry Pi 5 exchanges safety warnings with nearby smart vehicles, enabling

coordinated responses to road hazards. It also transmits critical location-tagged information to central servers or municipal authorities for timely road maintenance.

AI-Powered Monitoring

With its quad-core CPU and support for GPU acceleration, Raspberry Pi 5 is capable of running edge-based AI tasks such as object detection, lane detection, and motion tracking. This allows the system to operate autonomously without depending on cloud infrastructure, ensuring quick decision-making.

Real-Time Alerts and Display

The Pi 5 interfaces with LCD displays to show real-time status messages about detected hazards, vehicle positions, and traffic signals. This provides visual feedback to users or traffic control systems, enhancing transparency and user awareness.

GPS and Map Integration

The Raspberry Pi 5 communicates with GPS modules and OpenStreetMap APIs to accurately tag and visualize the location of detected damage. This integration allows users and authorities to monitor road conditions and maintenance needs geographically.

6.2.1.2 Why Raspberry pi ?

The **Raspberry Pi 5** is an ideal choice for implementing the **Advanced Road Safety System**, owing to its enhanced performance, flexibility, and wide hardware support. Some of the major reasons for its selection in the project are:

- **Cost-Effective and Globally Available**

Raspberry Pi 5 remains a highly **affordable platform**, making it accessible for large-scale prototyping and deployment. Its availability across global markets ensures consistent sourcing of components.

- **Enhanced GPIO and Peripheral Support**

The Raspberry Pi 5 includes a **40-pin GPIO header**, allowing seamless interfacing with **cameras, LCD displays, and communication modules** like **Bluetooth** and **ESP32**. This enables the system to transmit alerts in real time.

- **Support for Multiple Programming Languages**

Raspberry Pi 5 supports languages like **Python, C++, Java, Node.js, and Bash**, offering great flexibility in software development. In this project, **Python** is primarily used for **computer vision** (YOLOv11-based detection) and **UART communication** between the Raspberry Pi and ESP32.

- **High-Performance Processor**

Equipped with a **2.4 GHz quad-core ARM Cortex-A76 CPU**, the Raspberry Pi 5 offers significantly improved performance over its predecessors. This is especially valuable for **real-time image processing, edge detection algorithms, and machine learning inference** required in this road safety application.

- **Suitable for downloading our operating system (Linux) by using sd card**

Raspberry Pi initially has its own operating system previously called Raspbian based on Linux. In the emerging software world, there are few non-Linux based OS options available in the market. the preferred OS for the Pi are Linux distribution (Debian, Puppy Linux, Arch Linux, Fedora Remix and OpenELEC) as they are easily available at no cost, but majorly owing to their capability to function on the Raspberry Pi's ARM processor.

In summary, the **Raspberry Pi 5** plays a central role in the advanced road safety system by enabling high-speed processing, multi-sensor integration, and intelligent communication—all while remaining compact, affordable, and scalable.

6.2.1.3 Raspberry Pi 5 Technical Specifications

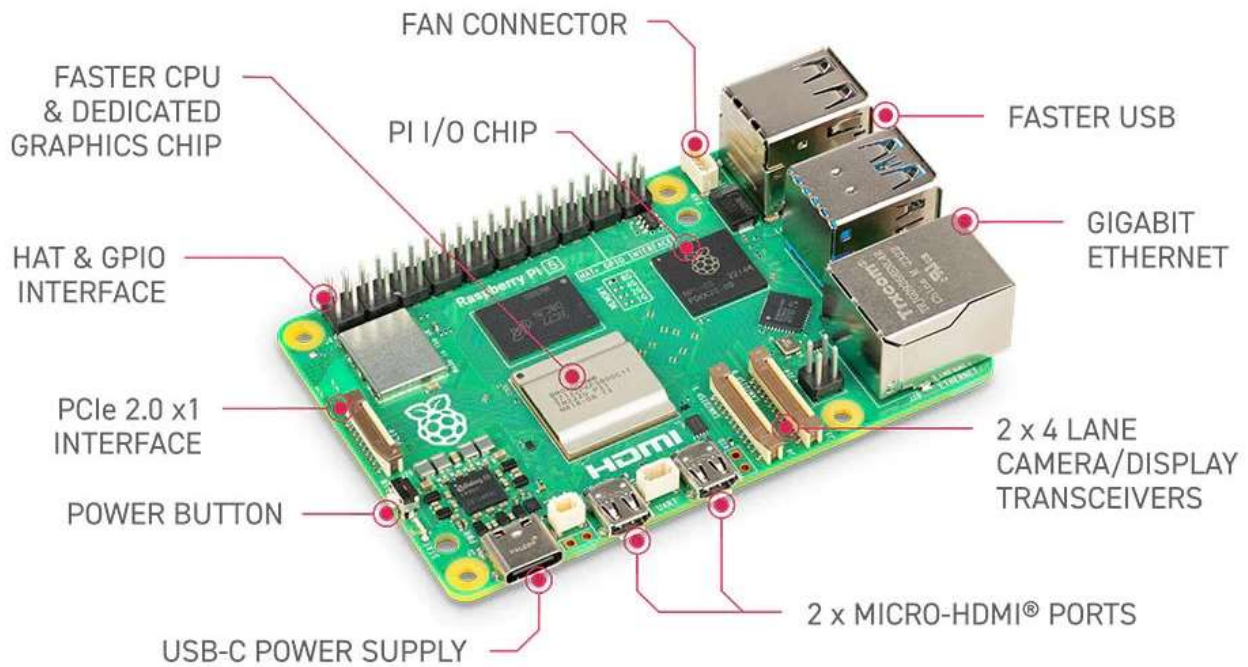


Fig.45 *Raspberry pi 5 specifications*

- 2.4GHz quad-core 64-bit Arm Cortex-A76 CPU
- VideoCore VII GPU
- Dual 4Kp60 HDMI® display output with HDR support
- 4Kp60 HEVC decoder
- LPDDR4X-4267 SDRAM (2GB, 4GB, 8GB, and 16GB)
- Dual-band 802.11ac Wi-Fi®
- Bluetooth 5.0 / Bluetooth Low Energy (BLE)
- microSD card slot, with support for high-speed SDR104 mode
- 2 × USB 3.0 ports and 2 × USB 2.0 ports
- Gigabit Ethernet
- 2 × 4-lane MIPI camera/display transceivers
- 5V/5A DC power via USB-C, with Power Delivery support
- Raspberry Pi standard 40-pin header
- Real-time clock (RTC), powered from external battery

6.2.1.4 GPIO and the 40-pin header

A key feature of the Raspberry Pi 5 is its 40-pin General Purpose Input/Output (GPIO) header located along the edge of the board. This header provides versatile digital input and output capabilities, allowing the Pi to interface with a wide array of hardware modules, sensors, motors, displays, and communication devices.

Each pin can be programmed in software to function as either an input or an output, with certain pins having dedicated roles for communication protocols such as UART, SPI, and I2C.

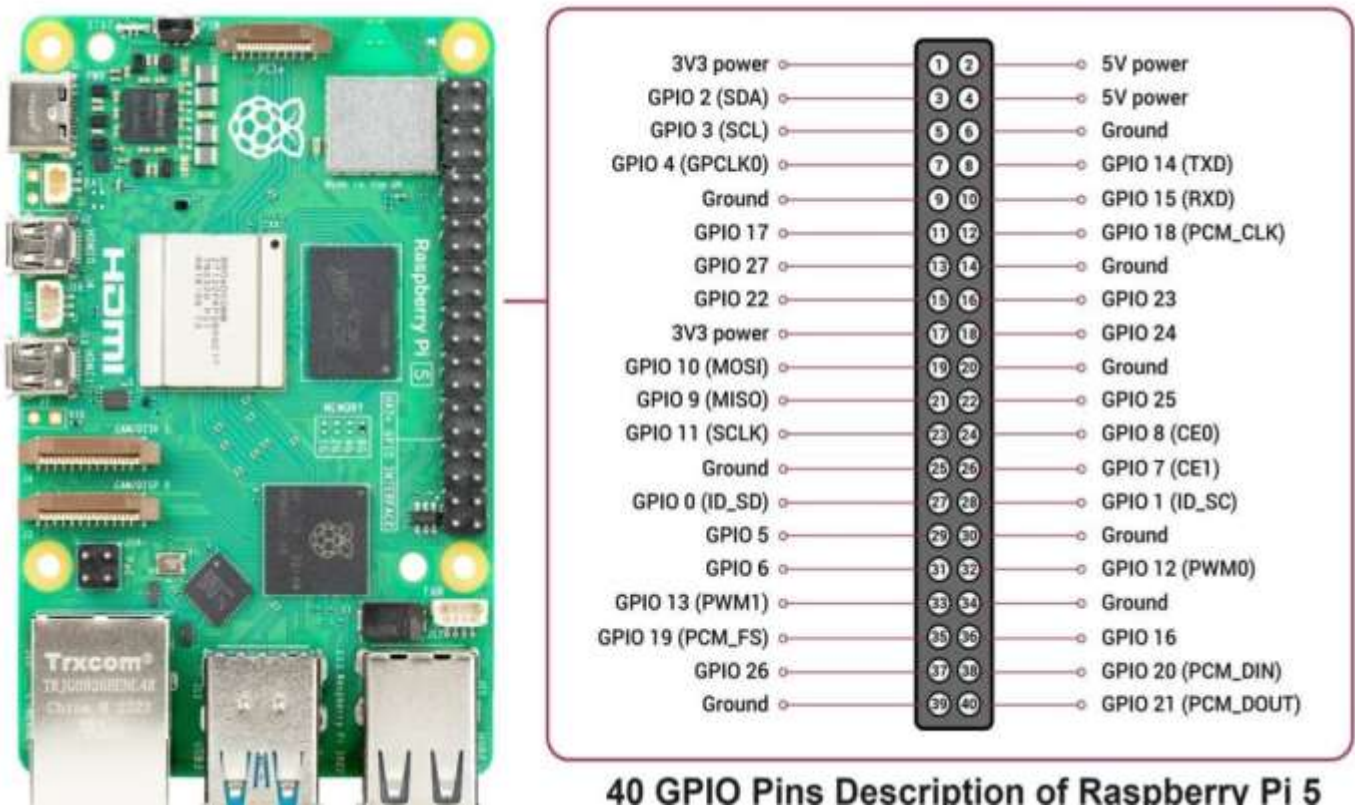


Fig.46 Raspberry pi 5 GPIO

6.2.1.5 Voltages

The Raspberry Pi 5 provides **two 5V power pins** and **two 3.3V power pins**, along with **multiple Ground (GND) pins**. These power and ground pins are fixed and cannot be reconfigured.

All remaining GPIO pins operate at **3.3V logic levels**—meaning that a high output delivers 3.3V and inputs should not exceed 3.3V to avoid damaging the board.

6.2.1.6 Outputs

When a GPIO pin is configured as an **output**, it can be programmatically set to **high (3.3V)** or **low (0V)**. The Raspberry Pi 5 maintains fast GPIO switching performance, allowing real-time control of actuators such as motors, relays, or LEDs in embedded applications.

6.2.1.7 Inputs

GPIO pins configured as **inputs** can detect **high (3.3V)** or **low (0V)** voltage levels. To aid signal stability and noise reduction, the Raspberry Pi 5 supports **internal pull-up and pull-down resistors**, which can be enabled via software. Note: **GPIO2 (SDA)** and **GPIO3 (SCL)** are equipped with permanent pull-up resistors due to their I²C functionality.

6.2.1.8 Other Functions

Beyond basic digital I/O, Raspberry Pi 5 GPIO pins support **alternative functions** for communication protocols and signal control. Some functions are pin-specific, while others are more flexible:

- **PWM (Pulse-Width Modulation)**
 - **Software PWM:** Available on all GPIO pins
 - **Hardware PWM:** Available on **GPIO12, GPIO13, GPIO18, and GPIO19**

- **SPI (Serial Peripheral Interface)**

- **SPI0:**

- MOSI: GPIO10
 - MISO: GPIO9
 - SCLK: GPIO11
 - CE0: GPIO8
 - CE1: GPIO7

- **SPI1:**

- MOSI: GPIO20
 - MISO: GPIO19
 - SCLK: GPIO21
 - CE0: GPIO18
 - CE1: GPIO17
 - CE2: GPIO16

- **I²C (Inter-Integrated Circuit)**

- Data (SDA): GPIO2
 - Clock (SCL): GPIO3
 - EEPROM Data: GPIO0
 - EEPROM Clock: GPIO1

- **UART (Universal Asynchronous Receiver/Transmitter)**

- TX (Transmit): GPIO14
 - RX (Receive): GPIO15

These versatile features make the Raspberry Pi 5 highly suitable for real-time sensing, actuation, and communication tasks in IoT and embedded system applications.

6.2.2 Raspberry Pi Camera

6.2.2.1 Introduction

The **Raspberry Pi Camera Module V2 – 8MP IMX219** is a high-quality, lightweight imaging device designed for seamless integration with Raspberry Pi hardware. In the **Advanced Road Safety System**, this camera plays a critical role in capturing real-time road conditions to detect **potholes**, **cracks**, and **obstacles** using **computer vision algorithms** (YOLOv11).

The camera connects via the **CSI (Camera Serial Interface)** port of the Raspberry Pi 5 and supports high-resolution imaging, making it ideal for real-time road analysis in smart vehicle environments.

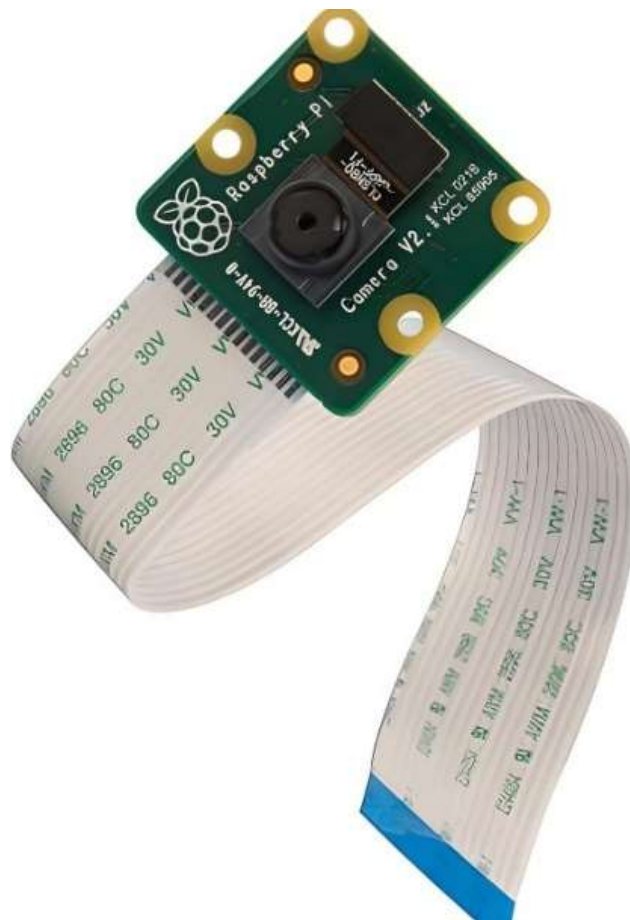


Fig.47 Raspberry pi camera

6.2.3.2 Camera Specification

Feature	Specification
Sensor	Sony IMX219
Resolution (Still)	8 Megapixels (3280 x 2464)
Video Modes	1080p @ 30fps, 720p @ 60fps, 640×480 @ 90fps
Interface	CSI (Camera Serial Interface)
Compatibility	Raspberry Pi 5 (via CSI port)
Weight	3g (lightweight and compact)
Lens Type	Fixed focus

Table.4 Raspberry Pi Camera Specification

6.2.2.3 Schematic of the Raspberry Pi CSI camera

The schematic shows how the CSI ribbon cable connects the **camera module** to the **Raspberry Pi 5** through the CSI port located beside the GPIO header.

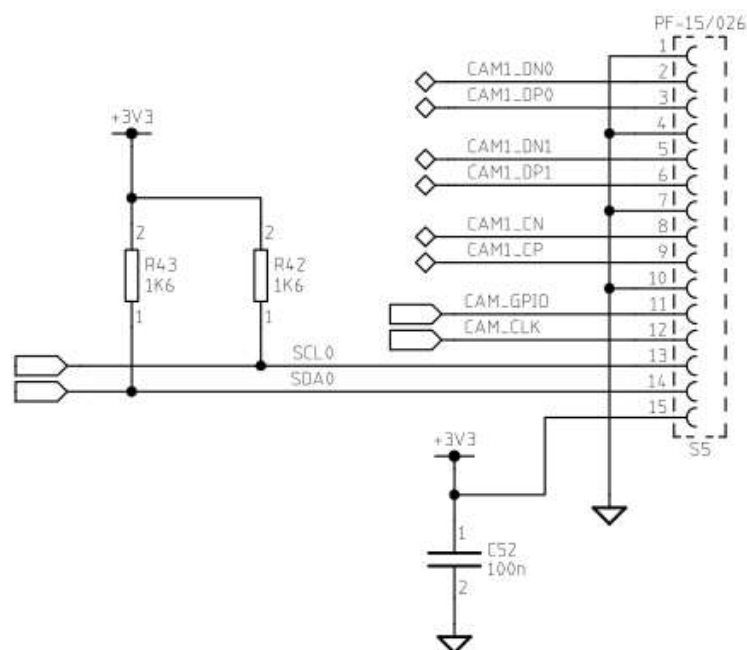


Fig.48 Schematic of Raspberry pi camera

6.2.2.4 Connecting the Camera Module

Follow the steps below to connect the Raspberry Pi Camera Module to the Raspberry Pi 5:

1. **Ensure your Raspberry Pi 5 is powered off.**
2. **Locate the CSI camera port** near the GPIO pins.
3. **Gently lift the plastic clip** of the CSI port.
4. **Insert the ribbon cable** from the camera module, ensuring the silver connectors face the port contacts.
5. **Secure the cable** by pressing the clip back into place.
6. **Power on your Raspberry Pi 5.**
7. **Open Raspberry Pi Configuration** from the main menu.
8. **Enable the camera interface** under the "Interfaces" tab.
9. **Reboot the Raspberry Pi 5** to apply changes.



Fig.49 Raspberry pi 5 with Camera

6.2.2.5 Control the Camera Module via Command Line

After the camera module is connected and enabled in the configuration, you can control it using command-line tools:

Steps:

1. Open the terminal window.
2. To **capture a still image**, run: `raspistill -o Desktop/image.jpg`

This will preview the camera for 5 seconds before saving an image.

3. To **record a video**, run: `raspivid -o Desktop/video.h264`

This command starts recording and saves the video file.

6.2.2.6 Control the Camera Module with Python

The **picamera** library allows advanced control of the camera in Python scripts, perfect for integrating into the road damage detection pipeline.

Steps:

1. Open **Thonny IDE** or any Python 3 editor.
2. Create a new file and save it as `camera.py`.
3. Enter the following code:

```
from picamera import PiCamera

from time import sleep

camera = PiCamera()

camera.start_preview()

sleep(5)
```

```
camera.capture('/home/pi/Desktop/image.jpg')
```

```
camera.stop_preview()
```

4. Run the script. The camera will preview for 5 seconds, then capture and save an image.

6.2.3 ESP32 Modules

6.2.3.1 Introduction

ESP32 is a low-power, dual-core, 2.4 GHz Wi-Fi and Bluetooth combo chip designed using TSMC's 40nm technology. It provides an excellent balance of power efficiency, RF performance, and connectivity versatility, making it ideal for embedded and IoT applications.

The ESP32, developed by Espressif Systems, is a successor to the widely used ESP8266 chip. It incorporates Tensilica's 32-bit Xtensa LX6 microprocessor and is available in both single-core and dual-core variants. This SoC (System on Chip) integrates key RF components such as the power amplifier, low-noise receive amplifier, filters, and antenna switches, which greatly reduces the need for external components in hardware design.

In the context of this project, the ESP32 plays a key role in **vehicle-to-vehicle (V2V)** communication using the **ESP-NOW** protocol and interacts with the **Raspberry Pi 5** via **UART** for real-time warning signal exchange and display through an LCD.

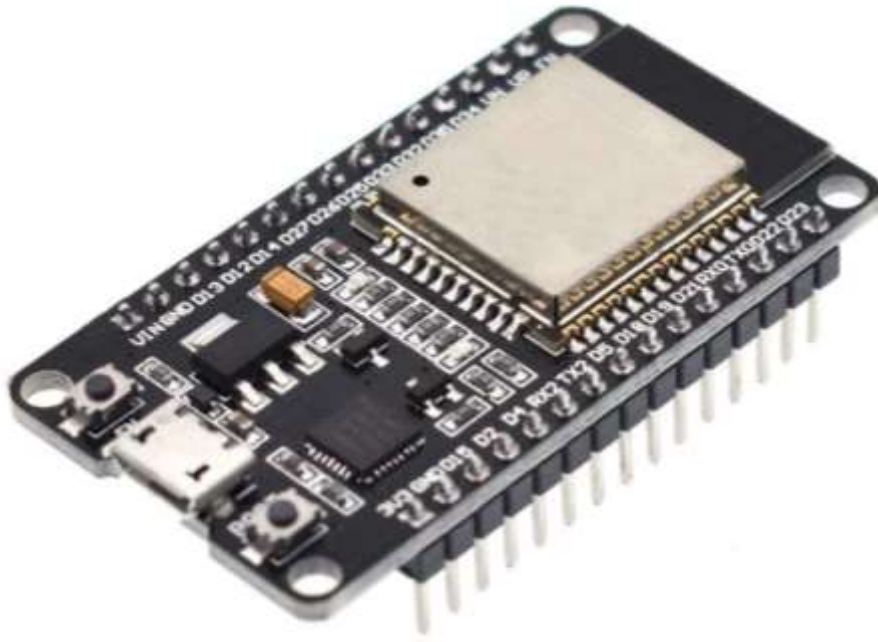


Fig.50 ESP-32 BOARD

6.2.3.2 Communication Protocols

The ESP32 supports multiple communication protocols that make it a robust choice for vehicular communication systems:

1. **UART Communication:**

The ESP32 receives pothole detection alerts from the **Raspberry Pi 5** via **UART serial communication**. This wired protocol ensures low-latency and reliable data transfer within the main smart vehicle system.

2. **ESP-NOW (V2V Communication):**

ESP-NOW is a connectionless Wi-Fi communication protocol developed by Espressif. It enables peer-to-peer communication between ESP32 modules **without requiring a Wi-Fi access point or internet**. In this system, once the main car detects a pothole, the ESP32 transmits this warning to other nearby vehicles via ESP-NOW, enhancing road safety through real-time alerts.

3. MQTT (Vehicle-to-Cloud - V2C):

Though MQTT is handled by the Raspberry Pi 5 in this system, it is worth mentioning that ESP32 also supports MQTT for cloud connectivity. This lightweight protocol can be used for future expansion or offloading cloud reporting from the Raspberry Pi.

4. Wi-Fi & Bluetooth:

ESP32 includes built-in support for Wi-Fi and Bluetooth Low Energy (BLE). These features provide additional flexibility in networking and diagnostics, although they are not the primary modes used in this road safety system.

Thus, the ESP-NOW protocol is selected for smart vehicle communication due to its low-latency, low-power, and infrastructure-independent nature.

6.2.3.3 ESP32 COMPONENTS

The ESP32 development board contains several important components that make it suitable for deployment in smart vehicles:

- **ESP-WROOM-32 Module:**

The core chip integrating the CPU, memory, Wi-Fi, Bluetooth, and I/O peripherals.

- **UART and GPIO Pins:**

The board provides around **30 GPIO pins** for interfacing with external components such as the LCD display. UART pins are used for receiving data from the Raspberry Pi.

- **CP2102 or CH340 USB–UART Bridge IC:**

Enables serial communication between the board and computer for firmware uploads and monitoring.

- **Micro-USB Connector:**

Used for power supply and uploading code to the board.

- **AMS1117 3.3V Regulator:**

Provides stable voltage required for the ESP32 module to function reliably.

- **Enable (EN) and Boot Buttons:**

Used to reset the device or enter bootloader mode for flashing new firmware.

- **Power LED (Red):**

Indicates the board is powered on.

- **User LED (Blue):**

Often used for debugging or simple output tasks through GPIO control.

- **Passive Components:**

Include resistors, capacitors, and inductors to support circuit stability and signal integrity.

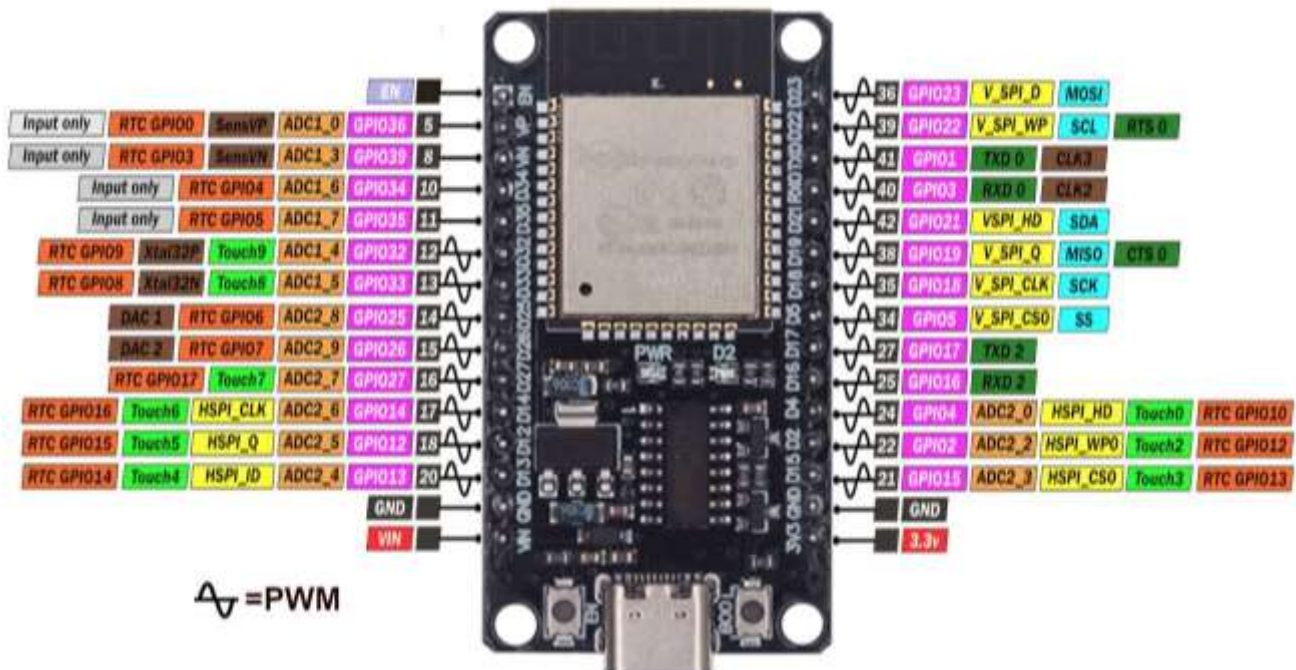


Fig.51

ESP32 GPIO COMPONENTS

6.2.3.4 ESP32 Software

The ESP32 can be programmed using multiple software environments. In this project, the ESP32 is used for UART reception, ESP-NOW broadcasting, and LCD message display.

Supported development platforms include:

- **Arduino IDE:**

Most widely used platform for programming the ESP32 using C/C++. This is used in this project. This is used in this project. This is used in this project.

- **PlatformIO (VS Code):**

An advanced IDE for embedded systems development, offering better tooling and project management.

- **ESP-IDF (Espressif IoT Development Framework):**

Official development environment for advanced ESP32 projects.

- **MicroPython:**

A lightweight Python implementation for embedded systems. Optional for prototyping and simpler scripts.

In our project, the **Arduino IDE** is used due to its simplicity and community support.

6.2.3.5 ESP32 Role In Our Project

In this project, the ESP32 is specifically responsible for **wireless communication and driver alerts**. Its role includes:

- **Receiving pothole alerts** from Raspberry Pi 5 via UART.
- **Broadcasting alerts** to nearby vehicles using ESP-NOW (V2V).
- **Displaying real-time warnings** on LCDs in both the main and surrounding vehicles.
- **Decoupling communication tasks** from the Raspberry Pi to improve system performance.
- **Supporting modular expansion**, such as future cloud updates or direct MQTT communication.

These features make ESP32 a critical component of the system's **Vehicle-to-Vehicle and Vehicle-to-Cloud safety communication layer**, helping prevent accidents and improve road quality reporting.

6.2.4 Arduino Uno

6.2.4.1 Introduction

Arduino Uno is an open-source microcontroller board based on the ATmega328P, widely used in electronics prototyping. It provides an easy-to-use platform for beginners and professionals to build embedded systems that interact with sensors, actuators, and various digital or analog peripherals.

The Arduino Uno, introduced in 2005, is part of the Arduino family of microcontroller development boards designed for ease of use. It allows users to write and upload programs via the **Arduino IDE** using a USB interface.

It features:

- A simple development workflow.
- Cross-platform IDE (Windows, macOS, Linux).
- Support for C/C++ based programming.

The Uno allows real-time data input from sensors and output to motors, LEDs, and displays, making it ideal for interactive projects.



Fig.52 ARDUINO UNO BOARD

6.2.4.2 Features

- **Communication Interfaces:**

- UART: Serial communication with modules like GPS or Bluetooth.
- SPI: For high-speed communication with devices like SD cards.
- I2C: For interfacing with displays and sensors using fewer pins.

- **Memory Specifications:**

- Flash Memory: 32 KB for storing sketches (programs).
- SRAM: 2 KB for runtime data.
- EEPROM: 1 KB for non-volatile data storage.

- **Shield Compatibility:**

Supports a variety of stackable expansion boards ("shields") such as motor drivers, Wi-Fi, Bluetooth, and GPS modules.

- **Power Management:**

- Operates on 5V.
- Input voltage range: 7–12V (via barrel jack or V_{in}).
- Can be powered via USB or external adapter.
- Includes onboard voltage regulators.

6.2.4.3 Arduino Uno Components

1. Microcontroller (ATmega328P):

Acts as the processing unit. 8-bit AVR architecture, 16 MHz, with digital I/O, PWM, and ADC support.

2. USB Port

- Used for power supply and serial communication.
- Upload programs using a USB-A to B cable.

3. ISCP Pins

- Used for in-system programming or burning bootloaders.
- Two sets: one for USB interface IC and one for the main microcontroller.

4. Reset Button

Restarts the program or resets the microcontroller.

5. Analog Input Pins (A0–A5)

- For reading analog sensors (0–5V).
- Can also be used as digital I/O if required.

6. Digital I/O Pins (0–13)

- General-purpose pins for controlling devices.
- Pins 0 and 1 are used for serial communication (RX/TX).
- Pins 3, 5, 6, 9, 10, 11 support PWM.

7. Crystal Oscillator (16 MHz)

Maintains system timing and clock cycles.

8. Voltage Regulator

Converts higher input voltage (7–12V) into a stable 5V for safe operation.

9. LED Indicators

- Power LED (ON)
- Pin 13 LED for basic testing or debugging.

10. Power and GND Pins

- 3.3V, 5V, Vin, and multiple GND pins for external components.

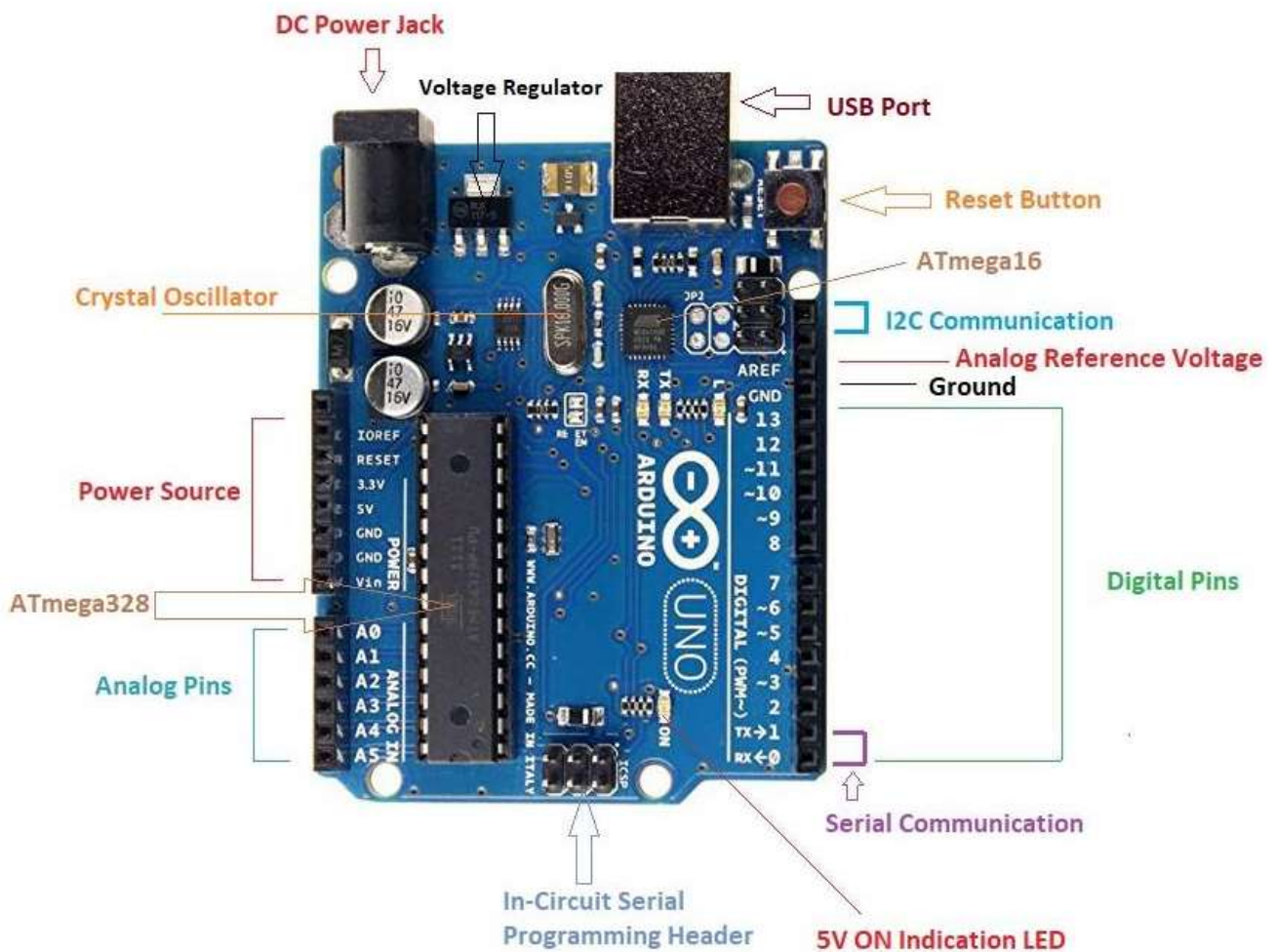


Fig.53 ARDUINO UNO COMPONENTS

6.2.4.4 Arduino Uno Software

The Arduino IDE is a cross-platform software used to write, compile, and upload programs (called *sketches*) to the Arduino Uno.

Key Components of the IDE:

- Text Editor: Write simplified C/C++ code.
- Compiler: Converts code into machine-readable format.
- Message Area: Shows error messages and build status.
- Serial Monitor: View data sent from the Arduino board in real-time.

Programs are uploaded using a USB cable through the bootloader pre-installed on the microcontroller.

6.2.4.5 Applications

Due to its flexibility and wide range of libraries, the Arduino Uno is used in numerous fields:

- **Home Automation Systems**
- **Robotics and Mechatronics Projects**
- **IoT (Internet of Things) Prototypes**
- **Environmental Monitoring and Data Logging**
- **Wearable Devices**
- **STEM Education and Learning Kits**
- **Game Controllers and DIY Entertainment Systems**
- **Rapid Prototyping of Embedded Systems**

We use it in our project to control our main car.

6.2.5 GPS Module

The **Ublox NEO-6M GPS module** is a high-performance GNSS receiver based on the **u-blox 6 chipset**. It is designed for applications requiring precise location information and high sensitivity, making it an ideal choice for road safety and tracking systems.

The module features a **25mm x 25mm active GPS antenna**, a **rechargeable battery** for fast GPS lock acquisition, and a **UART TTL interface** for serial communication. It delivers reliable and accurate geographic positioning, even under weak signal conditions. Its small size and integration of EEPROM for configuration retention make it especially suitable for embedded applications.



Fig.54 GPS NEO-6M

A Global Positioning System (GPS) module is integrated into our system to accurately capture the geographic location of road damage at the moment of detection. When the onboard AI model (YOLOv12) identifies a road anomaly—such as potholes, cracks, or other damages—the system simultaneously retrieves the current GPS coordinates of the vehicle.

These GPS coordinates are used for multiple purposes:

- **Vehicle-to-Vehicle (V2V) Communication:** The captured location data is transmitted to nearby vehicles through ESP-NOW protocol using ESP32 modules. This allows surrounding vehicles to be aware of upcoming road damage and take precautionary actions such as slowing down or rerouting.
- **Vehicle-to-Cloud (V2C) Communication:** The location is also published to AWS IoT Core via MQTT, where it is stored and processed in the cloud. Government agencies and municipal authorities can access this data to monitor the road network in real-time and prioritize maintenance operations.
- **Interactive Map Visualization:** The GPS data is used to place a marker on an online OpenStreetMap (OSM) interface. This map provides a visual representation of the exact location of detected damage, allowing both users and authorities to easily identify and navigate around affected areas.

The use of GPS ensures that every detected road issue is not only recognized but also precisely localized and shared across the system's ecosystem, enabling timely alerts, efficient repairs, and enhanced road safety.

6.2.5.1 Hardware Interface

The GPS module is connected to the Raspberry Pi via UART. This simple two-wire communication ensures reliable and low-latency data transfer.

- **TX (GPS) → RX (Raspberry Pi GPIO15)**
- **RX (GPS) → TX (Raspberry Pi GPIO14)**

- **GND → GND**
- **VCC (3.3V/5V) → Raspberry Pi 3.3V or 5V pin (depending on module version)**

Additionally, the **external antenna** is attached via the **u.FL connector** to improve satellite visibility and performance in moving vehicles.

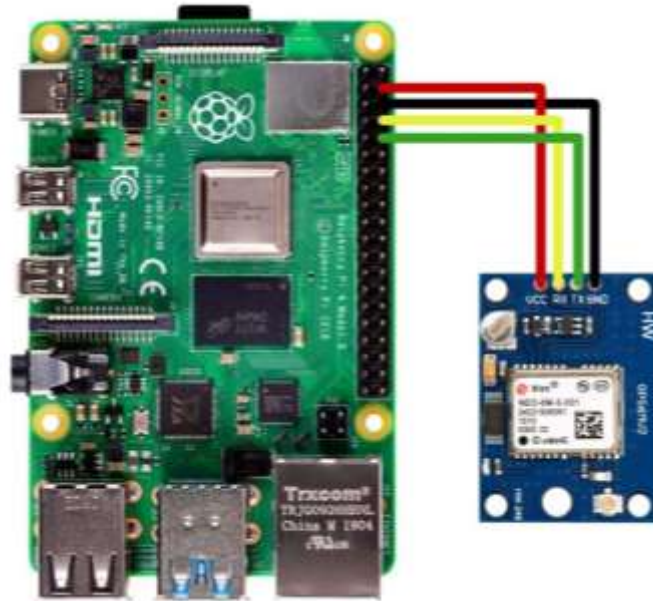


Fig.55 Raspberry Pi Connected to GPS Module

Receiver type:	56 channels, GPS L1(1575.42Mhz) C/A code, SBAS:WAAS/EGNOS/MSAS
Horizontal position accuracy:	2.5mCEP (SBAS:2.0mCEP)
Navigation update rate:	5 Hz maximum (1HZ default)
Capture time:	Cool start: 27s (fastest); Hot start: 1s
Communication protocol:	NMEA(default)/UBX Binary
Serial baud rate:	9600
Operating voltage:	2.7V~5.0V(power supply input via VCC)
Operating current:	35mA
Operating temperature range:	-40 TO 85°C UART TTL socket
TXD/RXD impedance:	510Ohms

Table.5 GPS NEO-7M Parameters

6.2.6 LCD Display

The 16x2 LCD display module is a key component in both the **main smart vehicle** and **companion smart cars**. It is used to **present real-time alerts and system status** such as pothole detection and road damage warnings. The display operates in **4-bit parallel mode**, directly interfaced with the **ESP32 microcontroller** in both systems.

A Liquid Crystal Display (LCD) is a flat-panel output device that leverages the light-modulating properties of liquid crystals. The 16x2 LCD allows 16 characters per line over 2 lines, making it ideal for concise and visible text output.

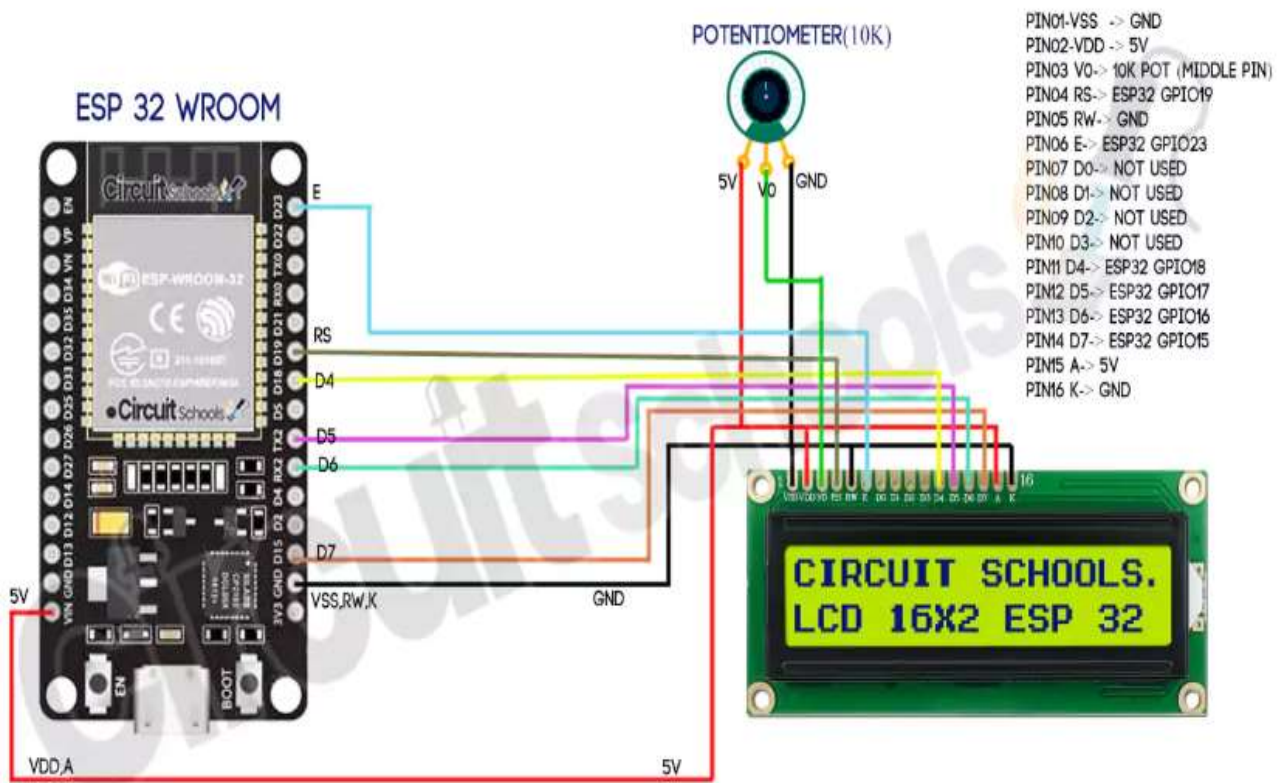


Fig.56 LCD with ESP32

6.2.7 RC Car Setup

Remote Control (RC) cars are miniature vehicles operated remotely through wireless communication methods such as Bluetooth. In this project, the RC car functions as a mobile robotic unit used for road inspection and safety warning, controlled via Bluetooth and powered by onboard electronics including motors and motor drivers.



Fig.57 RC Car

6.2.7.1 Connection Diagram

A detailed connection diagram illustrates how all hardware components—including the Arduino UNO, Bluetooth module (HC-05), L298N motor driver, and BO motors—are interconnected to enable wireless movement control.

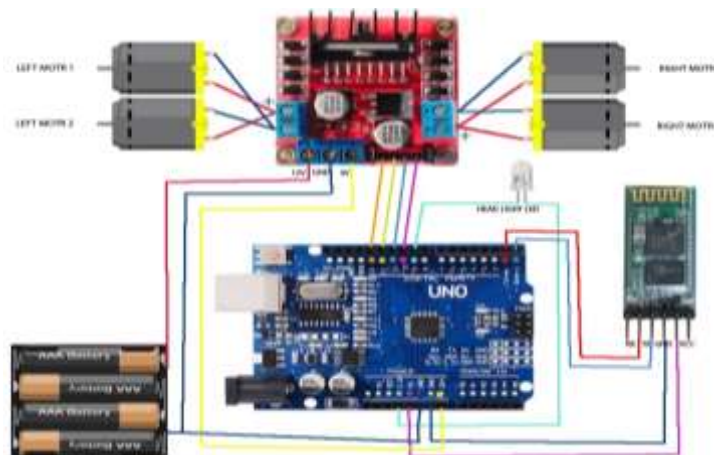


Fig.58 RC Car Connection Diagram

6.2.7.2 Hardware Components

The RC car system is composed of the following core hardware components:

- **Arduino UNO:** Microcontroller to control the system logic.
- **Bluetooth Module HC-05:** For wireless communication.
- **L298N Motor Driver:** For driving the motors.
- **BO Motors:** DC gear motors used for movement.
- **Chassis (3D Printed):**

The main structural frame of the car fabricated with a 3D printer. The design includes mounting slots and supports for the Arduino board, motor driver, battery, and wheels. The use of 3D printing allows for:

- Custom geometry tailored to component layout
 - Lightweight and durable construction
 - Easy modification and prototyping
- **12V Battery:** Power supply.
- **Wheels:** For linear movement.

Jumper Wires and Arduino USB Cable:

- **Jumper Wires:** Used for establishing electrical connections between the Arduino, modules, and motor driver.
- **USB Cable:** Used for programming the Arduino UNO and powering it during development or testing.

6.2.7.2.1 HC-05 Bluetooth Module

The HC-05 is a serial Bluetooth module that allows the RC car to wirelessly receive commands from a smartphone or Raspberry Pi. It can operate in Master or Slave mode and communicates using UART.

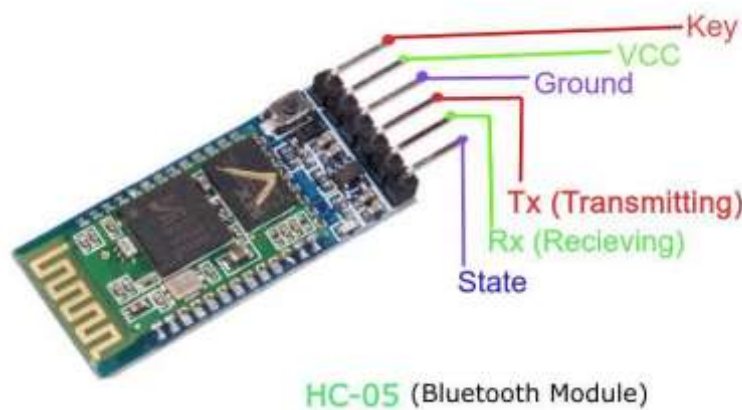


Fig.59 HC-05 Bluetooth Module

Pin Description:

1. Key/EN: Enables AT command mode (high = command mode).
2. VCC: Connects to 5V.
3. GND: Ground.
4. TXD: Transmit pin.
5. RXD: Receive pin.
6. State: Connection status indicator.

Modes:

- Command Mode: For setting configurations via AT commands.
- Data Mode: For actual data transmission between devices.

6.2.7.2.2 BO Motor

BO (Battery Operated) motors are low-cost gear motors widely used in educational and DIY robotics projects. They convert electrical energy into mechanical rotation to drive the car wheels.



Fig.60 Battery Operated motors

Key Points:

- Compact and lightweight
- Low power consumption
- Commonly used with wheels for forward/reverse motion

6.2.7.2.3 L298N Motor Driver

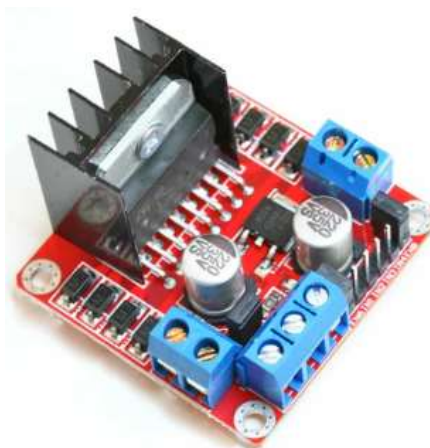


Fig.61 L298N Motor Driver

The L298N Motor Driver is a dual H-Bridge motor driver integrated circuit (IC) that allows you to control the direction and speed of two DC motors or one stepper motor. It is widely used in robotics and embedded systems for controlling motorized movement.

Key Features:

- Can control 2 DC motors independently.
- Supports voltage range from 5V to 35V.
- Can supply up to 2A current per channel.
- Works well with Arduino, Raspberry Pi, and other microcontrollers.
- Includes built-in heat sink to handle higher currents.

How It Works:

- It takes control signals (IN1, IN2, ENA, etc.) from a microcontroller (like Raspberry Pi or Arduino).
- It then sends power to the motor terminals (OUT1, OUT2) based on the control signals.
- Enable pins (ENA, ENB) are often connected to PWM outputs for speed control.
- IN1 and IN2 (or IN3/IN4 for second motor) control the direction of the motor.

in our Project:

When road damage is detected and confirmed via sensors:

- The Raspberry Pi sends a signal to stop the motors using the L298N driver.
- Once the road is clear and the car is moved again, the L298N receives signals to resume motion.

6.2.7.2.4 Ultrasonic Sensor

In addition to visual detection using the YOLOv12 AI model, this project incorporates an ultrasonic sensor to enhance immediate vehicle safety in response to detected road hazards.

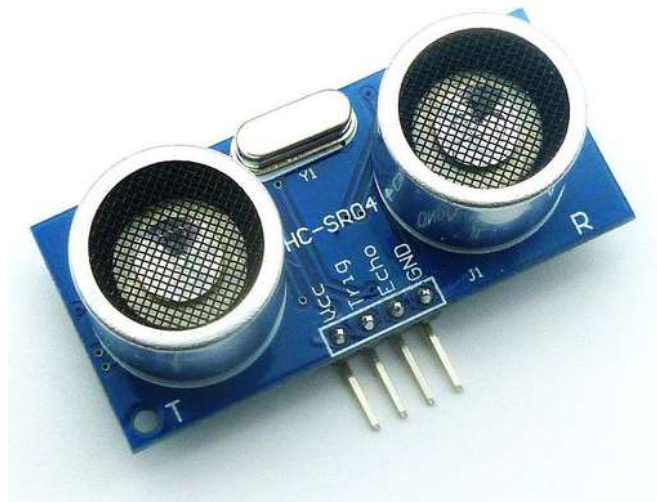


Fig.62 Ultrasonic Sensor

The ultrasonic sensor continuously measures the distance between the main vehicle and any object or obstacle ahead such as road damage, debris, or garbage. If the sensor detects that an object is closer than a predefined safety threshold, the system automatically triggers the vehicle to stop. This safety mechanism ensures that the vehicle does not proceed over a dangerous area, potentially preventing damage or accidents.

The vehicle remains stationary until the detected object is cleared or the user manually overrides the stop condition. This feature provides an additional layer of real-time response, particularly useful when visual recognition alone is not sufficient to prevent impact.

By integrating ultrasonic sensing with AI detection and GPS localization, the system ensures a multi-modal safety response, reinforcing the reliability and intelligence of the proposed road safety framework.

6.3 Software

6.3.1 *ESP-NOW*

ESP-NOW is a low-power, low-latency, peer-to-peer wireless communication protocol developed by Espressif, the creators of the ESP32 microcontroller. It allows multiple ESP32 devices to communicate directly with one another without the need for a central Wi-Fi network or internet connection.

Purpose in the Project:

In our system, ESP-NOW forms the foundation of the V2V (Vehicle-to-Vehicle) communication layer. After the Raspberry Pi detects a road hazard and sends the alert to the ESP32 via UART, the ESP32 uses ESP-NOW to broadcast this information to nearby vehicles, each equipped with its own ESP32 module

This enables instant, infrastructure-free communication of road hazards (e.g., potholes, cracks) between vehicles on the road.

Key Features of ESP-NOW in Our Project:

- **No Wi-Fi Required:** Devices do not need to be connected to a Wi-Fi network or router.
- **Broadcast Capable:** One ESP32 can send messages to multiple other ESP32s simultaneously.
- **Low Latency:** Ideal for real-time communication like road safety alerts.
- **Low Power:** Designed for battery-powered or low-energy devices.
- **Security Supports:** encrypted communication using peer-to-peer keys.

6.3.2 *Flask Web Application for Road Damage Reporting and Visualization*

Flask is a lightweight and flexible web framework for Python, making it ideal for building RESTful APIs and serving dynamic content. In our project, Flask is used as the backend system to receive, process, and store road damage reports, and to dynamically generate an interactive map interface using OpenStreetMap and Leaflet.js. This allows real-time visualization of all reported road damages from vehicles.

Why Flask?

Simplicity: Flask allows fast development with a minimal setup and clean code structure.

Flexibility: Easily integrates with SQLite, frontend frameworks, and third-party APIs.

Lightweight: Only essential components are included by default, keeping the backend efficient.

Scalable: Flask supports extensions and can be scaled if needed for future features (e.g., user login, cloud hosting).

in our project:

Creating the Database:

This file creates the database and the table for storing damage reports.


```
import sqlite3

conn = sqlite3.connect('damages.db')
cursor = conn.cursor()

cursor.execute('''
CREATE TABLE IF NOT EXISTS road_damage (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    latitude REAL,
    longitude REAL,
    type TEXT,
    description TEXT
)
''')

conn.commit()
conn.close()

print("Database created.")
```

Backend Responsibilities:

- Receive damage data via an API
- Store it in the database
- Render the map and pass damage data to the frontend

```
# Endpoint to receive GPS + damage info
@app.route('/api/add', methods=['POST'])
Tensor Go
def api_add():
    data = request.get_json()
    add_road_damage(data['latitude'], data['longitude'], data['type'], data['description'])
    return jsonify({'status': 'success', 'message': 'Damage recorded'})
```

Map View:

- Fetches all road damage entries from the database
- Passes them to map.html using render_template

```
# Serve the map
@app.route('/')
Tensor Go
def map_view():
    damages = get_all_damages()
    return render_template('map.html', damages=damages)
```

Sending Data to the Server:

This script simulates devices reporting road damage.

```
import requests
# Function to read coordinates from file (latitude and longitude separated by space)
Tensor Go
def read_coordinates(file_path):
    coordinates = []
    with open(file_path, 'r') as file:
        for line in file:
            line = line.strip()
            if line: # Skip empty lines
                lat, lon = line.split(' ', 1) # Split on first space only
                coordinates.append((float(lat), float(lon)))
    return coordinates
# Modified function to read damage types from file (read only up to first space)
Tensor Go
def read_damage_types(file_path):
    damage_types = []
    with open(file_path, 'r') as file:
        for line in file:
            line = line.strip()
            if line: # Skip empty lines
                # Split on first space and take only the first part
                damage_type = line.split(' ', 1)[0]
                damage_types.append(damage_type)
    return damage_types
# Paths to your input files
coordinates_file = 'coordinates.txt' # Format: "lat lon" on each line
damage_types_file = 'damage_types.txt' # Format: "type description" on each line
# Read data from files
coordinates = read_coordinates(coordinates_file)
damage_types = read_damage_types(damage_types_file)
# Check if we have matching number of coordinates and damage types
if len(coordinates) != len(damage_types):
    print(f"Warning: Mismatch in data counts - {len(coordinates)} coordinates vs {len(damage_types)} damage types")
# Create damage reports
damage_reports = []
for (lat, lon), damage_type in zip(coordinates, damage_types):
    damage_reports.append((lat, lon, damage_type))
```

```

damage_reports = []
for (lat, lon), damage_type in zip(coordinates, damage_types):
    damage_reports.append({
        "latitude": lat,
        "longitude": lon,
        "type": damage_type,
        "description": f"{damage_type} detected at this location"
    })

# Send data to server
for damage in damage_reports:
    try:
        r = requests.post("http://localhost:5000/api/add", json=damage)
        print(r.json())
    except requests.exceptions.RequestException as e:
        print(f"Failed to send data for {damage}: {e}")

```

Leaflet Map Visualization:

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Road Damage Map</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link
        rel="stylesheet"
        href="https://unpkg.com/leaflet/dist/leaflet.css"
    />
    <style>
        #map { height: 100vh; }
    </style>
</head>
<body>
    <div id="map"></div>

    <script src="https://unpkg.com/leaflet/dist/leaflet.js"></script>
    <script>
        // Center on Cairo with zoom level 13 (shows streets)
        var map = L.map('map').setView([30.0444, 31.2357], 13); // Cairo
    </script>

```

Marker Code:

For each damage:

- A marker is added to the map at the specified latitude and longitude

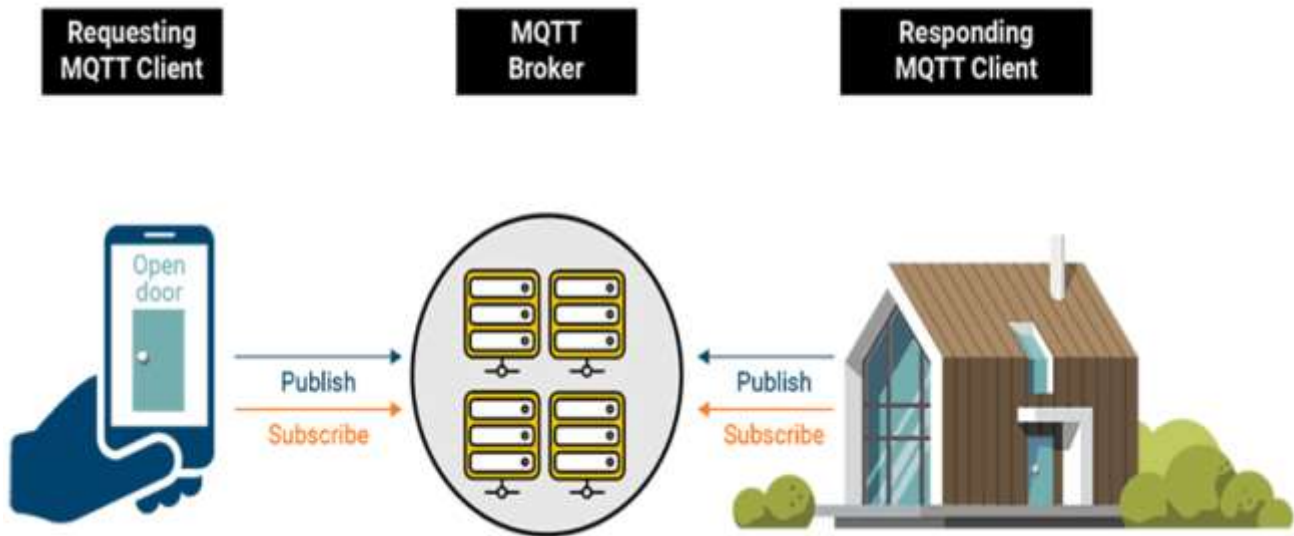
- A popup is bound showing the damage type and description

```
var damages = {{ damages | toJson }};  
damages.forEach(function(d) {  
  var marker = L.marker([d[0], d[1]]).addTo(map);  
  marker.bindPopup(`<b>${d[2]}</b><br>${d[3]}`);  
});
```

6.3.3 MQTT

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol that provides resource-constrained network clients with a simple way to distribute telemetry information. The protocol, which uses a publish/subscribe communication pattern, is used for machine-to-machine (M2M) communication and plays an important role in the internet of things (IoT). The MQTT protocol is a good choice for wireless networks that experience varying levels of latency due to occasional bandwidth constraints or unreliable connections.

The MQTT protocol surrounds two subjects: a client and a broker. An MQTT broker is a server, while the clients are the connected devices. When a device (or client) wants to send data to a server (or broker) it is called a publish. When the operation is reversed, it is called a subscribe. If the connection from a subscribing client to a broker is broken, then the broker will buffer messages and push them out to the subscriber when it is back online. If the connection from the publishing client to the broker is disconnected without notice, then the broker can close the connection and send subscribers a cached message with instructions from the publisher.



Purpose in the project:

- Enables efficient and lightweight communication between devices.
- Publishes road damage data (location, type, time) from the Raspberry Pi to AWS IoT Core (cloud broker).
- Subscribes government agency interfaces to receive real-time damage alerts.

code :

```
#include <iostream>
#include <fstream>
#include <string>
#include <thread>
#include <chrono>
#include <mqtt/client.h>
#include <sstream>
#define ADDRESS "ssl://abr0s9fb0umjr-ats.iot.us-east-2.amazonaws.com:8883"
#define CLIENTID "new-rpi"
#define TOPIC "road/damage"
#define QOS 1
#define CA_PATH "/home/root/main_app/root-ca.pem"
#define CERT_PATH "/home/root/main_app/certificate.pem.crt"
#define KEY_PATH "/home/root/main_app/private.pem.key"
#define AI_FILE "/home/root/main_app/ai.txt"
#define GPS_FILE "/home/root/main_app/gps.txt"
#define BACKUP_FILE "/home/root/main_app/unsent.txt"
// Utility: Check file exists and is not empty
bool isEmpty(const std::string& filename) {
    std::ifstream file(filename);
    return file && file.peek() != std::ifstream::traits_type::eof();
}
// Utility: Store data locally in case of failure
void storeLocally(const std::string& payload) {
    std::ofstream out(BACKUP_FILE, std::ios::app);
    out << "\n"<<payload ;
    std::cout << "Stored locally: " << payload << std::endl;
}
// Utility: Read entire file into a string
std::string readFile(const std::string& filepath) {
    std::ifstream file(filepath);
    return std::string((std::istreambuf_iterator<char>(file)), std::istreambuf_iterator<char>());
}
// Utility: Clear file content
void clearFile(const std::string& filepath) {
    std::ofstream ofs(filepath, std::ios::trunc);
}
```



```

int main() {
    try {
        mqtt::client client(ADDRESS, CLIENTID);
        mqtt::ssl_options ssl_opts;
        ssl_opts.set_trust_store(CA_PATH);
        ssl_opts.set_key_store(CERT_PATH);
        ssl_opts.set_private_key(KEY_PATH);
        mqtt::connect_options conn_opts;
        conn_opts.set_ssl(ssl_opts);
        conn_opts.set_keep_alive_interval(std::chrono::seconds(20));
        conn_opts.set_clean_session(false);
        std::cout << "Connecting to AWS IoT Core..." << std::endl;
        client.connect(conn_opts);
        if (!client.is_connected()) {
            std::cerr << "Client failed to connect!" << std::endl;
            // throw mqtt::exception();
        }
        std::cout << "Connected to AWS IoT Core!" << std::endl;
        // --- Send current gps + ai if both are non-empty ---
        if (!isEmpty(GPS_FILE) && !isEmpty(AI_FILE)) {
            std::string gps_data = readFile(GPS_FILE);
            std::string ai_data = readFile(AI_FILE);
            std::string combined_payload = ai_data + gps_data;

            mqtt::message_ptr msg = mqtt::make_message(TOPIC, combined_payload, QOS, false);
            client.publish(msg);
            std::cout << "Published current data.\n";
        } else {
            std::cout << "One or both files are empty. Skipping current data.\n";
        }
        if (!isEmpty(BACKUP_FILE)) {
            std::string all_unsent_data = readFile(BACKUP_FILE);

```

```

        if (!isEmpty(BACKUP_FILE)) {
            std::string all_unsent_data = readFile(BACKUP_FILE);

            if (!all_unsent_data.empty()) {
                mqtt::message_ptr backup_msg = mqtt::make_message(TOPIC, all_unsent_data, QOS, false);
                client.publish(backup_msg);
                std::cout << "Published all unsent data in one message.\n";
            }
            clearFile(BACKUP_FILE);
        }

        client.disconnect();
        std::cout << "Disconnected from AWS IoT Core.\n";
    } catch (const mqtt::exception& exc) {
        std::cerr << "MQTT Exception: " << exc.what() << std::endl;
        // Store current data locally if valid
        if (!isEmpty(GPS_FILE) && !isEmpty(AI_FILE)) {
            std::string gps_data = readFile(GPS_FILE);
            std::string ai_data = readFile(AI_FILE);
            std::string combined_payload = ai_data + gps_data;
            storeLocally(combined_payload);
        }
    }

    return 0;
}

```

6.3.4 Uart

UART (Universal Asynchronous Receiver/Transmitter) is a fundamental hardware communication protocol used for asynchronous serial communication between two digital devices. In this project, UART provides the physical and logical connection between the Raspberry Pi 5 (main controller and AI processor) and the ESP32 module (V2V communication unit).

Purpose of UART in the Project:

The UART interface is used for one-way communication from the Raspberry Pi to the ESP32. It plays a key role in Vehicle-to-Vehicle (V2V) communication by allowing the Raspberry Pi to transmit road damage alerts to the ESP32, which in turn broadcasts them to nearby vehicles using the ESP-NOW protocol to stop the car.

Detailed Operation Flow:

1. UART Communication Setup

Raspberry Pi 5 and ESP32 are connected via TX (transmit), RX (receive), and GND pins.

The Raspberry Pi uses `/dev/ttyAMA2` as its UART interface.

2. Sending the Data

When a road damage is detected (by camera + YOLO), the Raspberry Pi sends the alert message to ESP32. The message is like (Object 1: Pothole).

Then ESP32 will display the warning message to the driver that contains the object name.

Then ESP32 will send by ESP NOW to the other ESP32 to display the warning message to the driver that contains the object name and.

Receiving on ESP32 by using uart and send by esp now:

The ESP32 receives the UART message using its built-in UART interface (Serial2 for example) and parses the incoming string.

Once parsed, the ESP32 rebroadcasts the alert over ESP-NOW, a fast, connectionless wireless protocol optimized for low-latency, short-distance peer-to-peer communication.

```
1  #include <esp_now.h>
2  #include <WiFi.h>
3  #include <HardwareSerial.h>
4  #include <LiquidCrystal.h>
5
6  // LCD pin setup
7  LiquidCrystal lcd(19, 23, 18, 17, 16, 4);
8
9  // UART pins
10 #define RXD2 25
11 #define TXD2 26
12 HardwareSerial mySerial(2); // Serial2 for UART
13
14 // ESP-NOW receiver MAC address
15 uint8_t broadcastAddress1[] = {0xCC, 0xDB, 0xA7, 0x9D, 0x08, 0x48};
16
17 // Struct for ESP-NOW data
18 typedef struct struct_message {
19     char object[32];
20 } struct_message;
21
22 struct_message outgoingMessage;
23
24 esp_now_peer_info_t peerInfo;
25
26 // ESP-NOW send callback
27 void DataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
28     Serial.print("Send Status:\t");
29     Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Delivery Success" : "Delivery Fail");
30 }
```

```
31     if (status == ESP_NOW_SEND_SUCCESS) {
32         mySerial.println("ESP-NOW: Delivery Success");
33     } else {
34         mySerial.println("ESP-NOW: Delivery Failed");
35     }
36 }
37
38 void setup() {
39     Serial.begin(9600);
40     mySerial.begin(9600, SERIAL_8N1, RXD2, TXD2);
41
42     lcd.begin(16, 2);
43     lcd.print("Waiting for Msg");
44
45     WiFi.mode(WIFI_STA);
46
47     if (esp_now_init() != ESP_OK) {
48         Serial.println("Error initializing ESP-NOW");
49         return;
50     }
51     esp_now_register_send_cb(DataSent);
52
53     memcpy(peerInfo.peer_addr, broadcastAddress1, 6);
54     peerInfo.channel = 0;
55     peerInfo.encrypt = false;
56
57     if (esp_now_add_peer(&peerInfo) != ESP_OK) {
58         Serial.println("Failed to add peer");
59         return;
60     }
61
62     Serial.println("ESP32 Ready");
63     mySerial.println("ESP32 Ready and Listening");
64 }
65
66 void loop() {
67     if (mySerial.available() > 0) {
68         String input = mySerial.readStringUntil('\n');
69         input.trim();
70
71         Serial.println("Received from Pi: " + input);
72
73         // Example input: "Object 1: Pothole, Avg Depth: 0.78 meters"
74         int objIndex = input.indexOf(':');
75
76         if (objIndex != -1) {
77             String object = input.substring(objIndex + 1);
78             object.trim();
```

```
1  #include <esp_now.h>
2  #include <WiFi.h>
3  #include <LiquidCrystal.h>
4
5  // LCD pins (D7, D6, D5, D4, Enable, RS)
6  LiquidCrystal lcd(19, 23, 18, 17, 16, 15);
7
8  // Define UART pins for communication with Arduino
9  #define UART_TX_PIN 25 // TX
10 #define UART_RX_PIN 26 // RX
11
12 // Define the struct used by the sender
13 typedef struct struct_message {
14     char object[32];
15 } struct_message;
16
17 struct_message incomingData;
18
19 // Callback function for receiving ESP-NOW messages
20 void DataRecv(const esp_now_recv_info_t *recv_info, const uint8_t *data, int len)
21 {
22     memcpy(&incomingData, data, sizeof(incomingData));
```

Receiving on another ESP32 by esp now:

```
79
80     // Show on local LCD
81     lcd.clear();
82     lcd.setCursor(0, 0);
83     lcd.print("Warnning:");
84     lcd.setCursor(0, 1);
85     lcd.print(object);
86
87     // Send via ESP-NOW
88     strncpy(outgoingMessage.object, object.c_str(), sizeof(outgoingMessage.object));
89
90     esp_now_send(broadcastAddress1, (uint8_t *)&outgoingMessage, sizeof(outgoingMessage));
91
92     Serial.println("ESP-NOW message sent.");
93     mySerial.println("ESP-NOW message sent.");
94 } else {
95     Serial.println("Invalid format");
96     mySerial.println("Invalid format");
97 }
98 }
99
100 delay(100);
101 }
```

```
24     Serial.println("Data received via ESP-NOW:");
25     Serial.print("Object: ");
26     Serial.println(incomingData.object);
27
28     // Send 's' to Arduino over UART
29     Serial2.write('s');
30
31     // Display on LCD
32     lcd.clear();
33     lcd.setCursor(0, 0);
34     lcd.print("Warnning:");
35     lcd.setCursor(0, 1);
36     lcd.print(incomingData.object);
37 }
38
39 void setup() {
40     Serial.begin(9600);
41
42     // Initialize UART2 for Arduino communication
43     Serial2.begin(9600, SERIAL_8N1, UART_RX_PIN, UART_TX_PIN);
44
45     // Initialize LCD
46     lcd.begin(16, 2);
47     lcd.print("Waiting for Msg");
48
49     // Initialize WiFi and ESP-NOW
50     WiFi.mode(WIFI_STA);
51
52     if (esp_now_init() != ESP_OK) {
53         Serial.println("Error initializing ESP-NOW");
54         lcd.clear();
55         lcd.print("ESP-NOW Init Fail");
56         return;
57     }
58
59     // Register the receive callback
60     esp_now_register_recv_cb(DataRecv);
61 }
62
63 void loop() {
64     // Nothing needed in loop
65 }
```

CHAPTER 7

CONCLUSION AND

FUTURE WORK



6.1 Conclusion

This project has successfully developed and demonstrated an intelligent road damage detection and alerting system that enhances vehicular safety and infrastructure maintenance. By leveraging AI, embedded systems, and modern communication technologies, the system can accurately detect road damage powered by YOLOv12. Once detected, the system utilizes V2V communication to notify nearby vehicles and V2C communication to report incidents to government agencies. The integration of OpenStreetMap (OSM) allows for intuitive visualization of the reported road damage locations, supporting both real-time alerts and strategic planning.

6.2 System Performance Summary

The system achieves efficient and timely performance across all stages:

- **Detection:** YOLOv12 offers high accuracy and fast inference time for identifying longitudinal cracks, transverse cracks, alligator cracks, Potholes, and other corruptions.
- **V2V Communication:** ESP-NOW between two ESP32 devices provides low-latency and reliable peer-to-peer communication between vehicles.
- **V2C Communication:** AWS IoT Core serves as a robust cloud infrastructure for transmitting data to government agencies, ensuring scalability and security.
- **Visualization:** Road damage is accurately marked on an online OpenStreetMap interface, offering a clear and interactive geographic context.

6.3 Benefits to Smart , Non-Smart Vehicles and Government Agencies

- **Smart Vehicles:** Receive real-time alerts from nearby vehicles, allowing them to slow down or reroute autonomously.
- **Non-Smart Vehicles:** Benefit from updated maps and government responses, enhancing safety for all drivers regardless of technological capabilities.
- **Government Agencies:** Gain immediate awareness of road damage, enabling faster response and data-driven maintenance planning.

6.4 Future Enhancements

6.4.1 Integration with Emergency Services: Automatically

Notifying emergency services when severe road damage is detected—especially if correlated with sudden stops or accidents—could reduce response times and enhance public safety.

6.4.2 Expanded Dataset and Multi-Country Deployment:

Collecting Road damage images and data from different environments and countries will enhance model robustness and global applicability. Multilingual support and adherence to regional traffic laws will be necessary for scaling internationally.

6.4.3 Implement Vehicle Action Response:

Auto Slow Down: When a hazard is detected nearby, the vehicle reduces its speed to avoid damage or collision.

Rerouting: If a major hazard is detected ahead (e.g., a blocked road or construction zone), the navigation system recalculates a new route in real time.

6.4.4 Dataset Expansion

Incorporate additional diverse datasets beyond RDD2022, particularly focusing on datasets specific to urban environments and varied weather conditions. This will enhance the model's generalization across different scenarios, improving overall accuracy and reliability.

6.4.5 Optimize Architecture

Experiment with ensemble methods and attention mechanisms to enhance feature extraction and matching capabilities.

6.4.6 Local Dataset Training

Training on local Egyptian data could improve accuracy in region-specific conditions, enhancing safety for local drivers.

REFERENCES



- [1] Cooper, Schall & Levy. (2023, December 5). *How is responsibility for an accident due to road damage determined? from <https://www.cooperschallandlevy.com/2023/12/05/how-is-responsibility-for-an-accident-due-to-road-damage-determined/>
- [2] Krishnan, R. S., Sangeetha, A., Kumari, D. A., Nandhini, N., Karpagarajesh, G., Narayanan, K. L., & Robinson, Y. H. (2022). A Secured Manhole Management System Using IoT and Machine Learning. In *Recent Advances in Internet of Things and Machine Learning: Real-World Applications* (pp. 19-30). Cham: Springer International Publishing.
- [3] Vialytics. (n.d.). *The dangers of potholes: How they impact road safety and infrastructure*. From <https://www.vialytics.com/blog/dangersofpotholes?>
- [4] Youwai, S., Chaiyaphat, A., & Chaipetch, P. (2024). A Fused Deep Learning Expert System for Road Damage Detection and Size Analysis Using YOLO9tr and Depth Estimation. In *IEEE The 3rd International Conference on Intelligent Computing and Next Generation Networks (ICNGN 2024)*.
- [5] Li, Y., Yin, C., Lei, Y., Zhang, J., & Yan, Y. (2024). RDD-YOLO: Road Damage Detection Algorithm Based on Improved You Only Look Once Version 8. *Applied Sciences*, 14(8), 3360.
- [6] Zulkifli, N. A. F. M., Mustaffa, Z., & Sulaiman, M. H. (2025). Road Damage Detection for Autonomous Driving Vehicles using YOLOv8 and Salp Swarm Algorithm. *Applications of Modelling and Simulation*, 9, 1-11.
- [7] Li, X., & Zhang, Y. (2025). A Lightweight Method for Road Damage Detection Based on Improved YOLOv8n. *Engineering Letters*, 33(1).
- [8] Li, X., & Zhang, Y. (2024). Improved Road Damage Detection Algorithm Based on YOLOv8n. *IAENG International Journal of Computer Science*, 51(11).

- [9] Zeng, J., & Zhong, H. (2024). YOLOv8-PD: an improved road damage detection algorithm based on YOLOv8n model. *Scientific reports*, 14(1), 12052.
- [10] Afshar, M. F., Shirmohammadi, Z., Ghahramani, S. A. A. G., Noorparvar, A., & Hemmatyar, A. M. A. (2023). An Efficient Approach to Monocular Depth Estimation for Autonomous Vehicle Perception Systems. *Sustainability*, 15(11), 8897.
- [11] Pham, V., Ngoc, L. D. T., & Bui, D. L. (2024, December). Optimizing YOLO Architectures for Optimal Road Damage Detection and Classification: A Comparative Study from YOLOv7 to YOLOv10. In *2024 IEEE International Conference on Big Data (BigData)* (pp. 8460-8468). IEEE.
- [12] Pan, W., Kang, J., Wang, X., Chen, Z., & Ge, Y. (2024). DAPONet: A Dual Attention and Partially Overparameterized Network for Real-Time Road Damage Detection. *arXiv preprint arXiv:2409.01604*.
- [13] Adi, T. J. W., Suprobo, P., & Waliulu, Y. E. P. R. (2024). iRodd (intelligent-road damage detection) for real-time infrastructure preservation in detection, classification, calculation, and visualization. *Journal of Infrastructure, Policy and Development*, 8(11), 6162.
- [14] Youwai, S., Chaiphaphat, A., & Chaipetch, P. (2024). A Fused Deep Learning Expert System for Road Damage Detection and Size Analysis Using YOLO9tr and Depth Estimation. In *IEEE The 3rd International Conference on Intelligent Computing and Next Generation Networks (ICNGN 2024)*.
- [15] Alkalbani, A., Saqib, M., Alrawahi, A. S., Anwar, A., Adak, C., & Anwar, S. (2025). RDD4D: 4D Attention-Guided Road Damage Detection And Classification. *arXiv preprint arXiv:2501.02822*.
- [16] Khanam, R., & Hussain, M. Yolov11: An overview of the key architectural enhancements. arXiv 2024. *arXiv preprint arXiv:2410.17725*.

- [17] Alif, M. A. R. (2024). YOLOv11 for vehicle detection: Advancements, performance, and applications in intelligent transportation systems. *arXiv preprint arXiv:2410.22898*.
- [18] He, L., Zhou, Y., Liu, L., & Ma, J. (2024). Research and Application of YOLOv11-Based Object Segmentation in Intelligent Recognition at Construction Sites. *Buildings*, 14(12), 3777.
- [19] Rasheed, A. F., & Zarkoosh, M. (2024). YOLOv11 Optimization for Efficient Resource Utilization. *arXiv preprint arXiv:2412.14790*.
- [20] He, L. H., Zhou, Y., Liu, L., & Zhang, Y. Q. Research on the Directional Bounding Box Algorithm of YOLOv11 in Tailings Pond Identification. *Available at SSRN 5055415*.
- [21] Redmon, J. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*.
- [22] Shelke, S. M., Pathak, I. S., Sangai, A. P., Lunge, D. V., Shahale, K. A., & Vyawahare, H. R. (2023). A review paper on computer vision. *International Journal of Advanced Research in Science, Communication and Technology*, 673-677.
- [23] Zeadally, S., Guerrero, J., & Contreras, J. (2020). A tutorial survey on vehicle-to-vehicle communications. *Telecommunication Systems*, 73(3), 469-489.
- [24] Daddanala, R., Mannava, V., Tawlbeh, L. A., & Al-Ramahi, M. (2021). Vehicle to vehicle (V2V) Communication Protocol: components, benefits, challenges, safety and machine learning applications. *arXiv preprint arXiv:2102.07306*.
- [25] Salim, A., Ali, M., & Al-Hemairy, E. H. (2021, June). Communication module for V2X applications using embedded systems. In *Journal of Physics: Conference Series* (Vol. 1933, No. 1, p. 012112). IOP Publishing.
- [26] Hussein, N. A., & Shujaa, M. I. (2020). Secure vehicle to vehicle voice chat based MQTT and coap internet of things protocol. *Indonesian Journal of Electrical Engineering and Computer Science*, 19(1), 526-534.

- [27] Kegenbekov, Z., & Saparova, A. (2022). Using the MQTT protocol to transmit vehicle telemetry data. *Transportation Research Procedia*, 61, 410-417.
- [28] Sharma, S., & Kaul, A. (2021). VANETs cloud: architecture, applications, challenges, and issues. *Archives of Computational Methods in Engineering*, 28, 2081-2102.
- [29] Ulil, A. M. R., Sukaridhoto, S., Tjahjono, A., & Basuki, D. K. (2019, February). The vehicle as a mobile sensor network base IoT and big data for pothole detection caused by flood disaster. In *IOP Conference Series: Earth and Environmental Science* (Vol. 239, No. 1, p. 012034). IOP Publishing.
- [30] Forkan, A. R. M., Kang, Y. B., Marti, F., Banerjee, A., McCarthy, C., Ghaderi, H., ... & Jayaraman, P. P. (2024). Aiot-citysense: Ai and iot-driven city-scale sensing for roadside infrastructure maintenance. *Data Science and Engineering*, 9(1), 26-40.
- [31] He, W., Li, H., Zhi, X., Li, X., Zhang, J., Hou, Q., & Li, Y. (2019). Overview of V2V and V2I wireless communication for cooperative vehicle infrastructure systems, in 2019 IEEE 4th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC).
- [32] Zhou, H., Xu, W., Chen, J., & Wang, W. (2020). Evolutionary V2X technologies toward the Internet of vehicles: Challenges and opportunities. *Proceedings of the IEEE*, 108(2), 308-323.
- [33] Rakouth, H., Alexander, P., Brown, A. J., Kosiak, W., Fukushima, M., Ghosh, L., ... & Shen, J. (2013). V2X communication technology: Field experience and comparative analysis. In *Proceedings of the FISITA 2012 World Automotive Congress: Volume 12: Intelligent Transport System (ITS) & Internet of Vehicles* (pp. 113-129). Springer Berlin Heidelberg.

- [34] Adi, T. J. W., Suprobo, P., & Waliulu, Y. E. P. R. (2024). *Road repair delay costs in improving the road rehabilitation strategy through a comprehensive road user cost model*. ResearchGate.
- [35] Krishnan, R. S., Sangeetha, A., Kumari, D. A., Nandhini, N., Karpagarajesh, G., Narayanan, K. L., & Robinson, Y. H. (2022). A Secured Manhole Management System Using IoT and Machine Learning. In *Recent Advances in Internet of Things and Machine Learning: Real-World Applications* (pp. 19-30). Cham: Springer International Publishing.

