A

PROJECT SCHOOL REPORT

ON

Safe Street - Road Damage Detection System

Submitted By

D. Venkat Madhu Mohan	23P81A6921
A. Rushika	23P81A6905
P. Santhi Sri	23P81A05C0
K. Shivanandan	23P81A6926
C. Sathvik	23P81A6913

Under the guidance

of

Mrs. K. Navatha

Keshav Memorial College of Engineering

Koheda Road, Chintapalliguda(V), Ibrahimpatnam(M), R.R Dist – 501510

May, 2025

Keshav Memorial College of Engineering

Sponsored by Keshav Memorial Educational Society (KMES)

Approved by AICTE, New Delhi & Affiliated to JNTUH, Hyderabad

CERTIFICATE

This is to certify that the project work entitled “**Safe Street- Road Damage Detection System”** *is a bonafide work carried out by* “**D. Venkat Madhu Mohan**”, “**A. Rushika**”, “**P. Santhi Sri**”, “**K. Shivanandan**”, “**C. Sathvik**” of II year II semester **Bachelor of Technology** *in* **CSE/CSE(IOT)** *during the academic year* **2024-2025** *and is a record of bonafide work carried out by them*.

Project Mentor

Mrs. K. Navatha

Assistant Professor

CSE(AIML),KMCE

ABSRACT

Road damage pose serious risks, leading to accidents, vehicle damage, and traffic disruptions. Traditional methods of identifying road damage rely heavily on manual inspections, which are often slow, labor-intensive, and inconsistent. To address this challenge, SafeStreet offers an AI-driven approach that automates the detection, classification, and reporting of road surface issues. By analyzing images of roads, the system identifies damage patterns, assesses their severity, and generates clear, human readable summaries for effective communication. Developed using advanced computer vision, SafeStreet enhances the accuracy, speed, and scalability of road monitoring. This solution empowers municipal bodies to prioritize repairs efficiently, improving road safety, and optimizing infrastructure maintenance with minimal human intervention, ultimately contributing to safer transportation and smarter urban infrastructure management.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER-1

INTRODUCTION

1.1. Problem Statement

Safe Street leverages a Vision Transformer (ViT) to automate damage identification and prioritization from Captured images. First, the ViT classifies the type of damage (e.g., cracks, dents, corrosion) and assesses its severity, drawing on learned features that capture subtle visual details. Next, the system generates a concise text summary that documents the specific damage type, severity level, and recommended repair priority. This integrated approach streamlines inspection workflows, cuts down manual evaluation, and delivers consistent, data-driven insights, particularly useful for industries like insurance, automotive, construction, or manufacturing. By combining powerful vision models with intelligent text generation, the project ensures both accurate visual analysis and clear actionable reporting for damage assessment.

1.2. Objective

The objective of the SafeStreet project is to implement an AI-powered system capable of automatically detecting and classifying road damage from images and generating descriptive summaries to assist in maintenance decisions. This solution aims to improve the efficiency, accuracy, and scalability of road inspection processes.The specific objectives of the project include:

- To develop a computer vision pipeline that can identify road surface anomalies such as potholes and cracks from real world images.
- To classify the severity of detected road damages to prioritize maintenance tasks effectively.
- To generate human readable textual descriptions that summarize the condition of the road, making it easier for authorities to interpret the data.
- To reduce reliance on manual inspections by providing a faster, automated, and reliable road condition monitoring system.

1.3. Scope of the project

The Safe Street system focuses on detecting and classifying road damage using AI models, ensuring accurate identification and severity assessment. The system provides the following functionalities:

- **Image Capture System**: Users can capture images of roads directly through the camera, with the system automatically accessing GPS location to tag the images for accurate mapping and damage assessment.
- **AI-based Classification**: The system classifies road damage into different categories using deep learning models.
- **Severity Assessment**: It assigns a severity score to the damage based on pre-defined thresholds.
- **Report Generation**: Generates automated text reports for municipal authorities.
- **User Feedback Mechanism**: Allows users to provide feedback on the system's accuracy to improve future predictions.

1.4. Bussiness Cases

The SafeStreet system offers scalable benefits across multiple sectors by automating road damage detection, classification, and reporting. Below are key business domains and how SafeStreet addresses their specific needs:

I. Government & Municipalities

For Government and Municipal Corporations, SafeStreet serves as an efficient tool for automating road inspection and prioritizing maintenance operations. It eliminates the need for manual surveys, which are often time-consuming and inconsistent, by providing real-time damage reports from field uploads. This allows governments to optimize budgets, allocate resources based on severity, and act quickly in critical repair zones.

II. Smart Cities & Urban Planning

In the context of Smart Cities and Urban Planning, this system enables continuous monitoring of road conditions, feeding essential data into urban development dashboards for data-driven planning and timely intervention.

III. Insurance Companies

Insurance companies can leverage SafeStreet to validate road conditions in accident-prone areas. The system provides automated, timestamped road assessments that reduce fraudulent or exaggerated claims, thus ensuring faster claim settlements and minimizing the need for manual inspections.

IV. Road Construction & Maintenance Firms

For Road Construction and Maintenance Firms, SafeStreet facilitates pre- and post-repair documentation. By generating visual and data-backed records of road damage and repairs, it supports warranty claims and annual maintenance contract (AMC) tracking. This brings greater transparency and accountability to public and private construction work.

V. Logistics & Transportation Companies

Logistics and Transportation Companies benefit from SafeStreet by incorporating road condition data into route optimization systems. This ensures safer, more efficient delivery paths while minimizing vehicle damage, reducing fuel consumption, and cutting operational .

VI. Crowdsourced Reporting Apps

For Crowdsourced Reporting Platforms, empowers citizens to participate in road safety by allowing them to upload images of damaged roads. These images are automatically analyzed and geo-tagged, offering verified and prioritized data to authorities. This improves civic engagement and enhances the responsiveness of municipal services.

CHAPTER-2

LITERATURE SURVEY

To design an AI-driven system for road damage detection and summarization, it is crucial to examine existing advancements in computer vision, deep learning, and intelligent transportation systems. This chapter presents a comprehensive review of relevant research, highlighting key methodologies, technologies, and real world applications that have shaped the development of automated infrastructure monitoring. Understanding these foundational contributions helps identify current challenges and justify the approach adopted in this project.

2.1. Existing methodologies

Various methods have been proposed for road damage detection over the time, evolving from manual surveys to automated AI-powered systems. Traditional methods rely on human inspection. These methods, while simple, are labor-intensive, slow, and prone to inconsistencies.

With advancements in computer vision, several researchers adopted image processing techniques such as edge detection and thresholding to identify the road damage in grayscale images. However, these techniques often failed under varying lighting and texture conditions.

Later, machine learning models such as Support Vector Machines (SVM), Random Forests, and K-Nearest Neighbors (KNN) were applied using handcrafted features like . These models offered better accuracy but were still limited in generalizing across diverse road types.

Recently, deep learning approaches using Convolutional Neural Networks (CNNs) have become dominant due to their ability to learn features directly from raw image data. Architectures like Faster R-CNN and YOLO are detecting multiple types of damage with improved speed and accuracy. These models are often trained on datasets like RDD2020, RDD2022, or self-collected datasets from different regions.

2.2. Technologies used in existing methodologies

Earlier systems for road damage detection used technologies that ranged from traditional image processing libraries to complex neural network frameworks. Key technologies include:

- **OpenCV**: Extensively used for pre-processing tasks such as grayscale conversion, edge detection, and segmentation.
- **MATLAB**: Preferred for signal-based detection and image analysis in early academic research.
- **Python & Scikit-learn**: Used for implementing classical machine learning algorithms using handcrafted features.
- **TensorFlow & PyTorch**: The most common frameworks for building and training deep learning models such as CNNs and object detectors.
- **Pre-trained Models**: In many studies, transfer learning with pre-trained models like ResNet and InceptionNet was used for damage classification.
- **Mobile Platforms**: Some solutions have started integrating models with mobile apps for crowdsourced data collection and on-device inference.

Method	Advantages	Disadvantages
Support Vector Mechine	-Effective in high dimensional spaces, -Works well with small to medium datasets	-Slow training time with large datasets, -Requires careful kernel selection
KNN(K-Nearest Neighbors)	-Simple and easy to implement, -No training phase, -Effective with low dimensional data	-Computationally expensive for prediction, -Performance degrades with high-dimensional data, -Sensitive to irrelevant features and noise
R-CNN (Region-based Convolutional Neural Network)	-High accuracy for object detection tasks, -Can detect and localize multiple objects in an image	-Slow and resource-intensive, -Less efficient than single-shot detectors like YOLO, -Not suitable for real-time applications
GLCM (Gray Level Co-occurrence Matrix)	-Captures texture information effectively, -Useful for structured surface analysis	-Computationally expensive for large images, -Sensitive to image orientation and noise, -Limited when dealing with complex damage patterns
HOG (Histogram of Oriented Gradients)	-Good for capturing object shape and structure, -Invariant to shadow changes, -Widely used in object detection	-Sensitive to image rotation and scale, -Computationally heavy for large datasets, -Less effective on low contrast images

Table 1. Camparing various researches methods on Road damage detection

2.3. How SafeStreet Overcomes Existing Limitations

Limitations	Solution by SafeStreet
Manual Dependency: Traditional methods rely on human surveyors, leading to inconsistent, delayed, and error-prone assessments.	SafeStreet uses AI-driven automation to detect road damage directly from images captured. This eliminates the need for constant manual observation and speeds up reporting by automatically generating summaries.
Limited Generalization: Classical models like SVM struggle with generalizing across different lighting, surfaces, and angles.	SafeStreet uses deep learning models (YOLOv8l and ViT) trained on diverse datasets. Image preprocessing and augmentation techniques are used to make the model robust to various real-world conditions.
Dataset Constraints: Many deep learning models depend on large, well-labeled datasets, which are often limited in scope.	SafeStreet uses custom, expanding coverage of regional road types and damage categories. This improves classification accuracy and generalization.
Computational Costs: Two-stage detectors like Faster R-CNN require significant computational power, making them unsuitable for lightweight systems.	SafeStreet uses YOLOv8l, a lightweight, single-stage detector that balances speed and accuracy. It can run on standard hardware, making it accessible and deployable even on mid-range devices.
Actionable Insights for Authorities: Many existing detection systems stop at identifying damage, offering no structured output that helps city authorities prioritize or plan maintenance. This leads to inefficient resource allocation and slow response times.	SafeStreet overcomes this by classifying severity (high,medium,low),Generating summarized reports in natural language,tagging GPS or location metadata, enabling area-wise damage mapping.

Table 2. Comparing how safe street overcomes the existing limitations

CHAPTER-3

Proposed Work and Architecture, Software Requirements Specification Technology Stack & Implementation Details

3.1. Proposed Work and Architecture

This project aims to automate road damage detection, classification, and reporting through an AI-driven solution. The proposed system processes input road images to detect damage, classify their type and severity, and finally generate a human-readable summary report. The overarching goal is to assist municipal authorities in prioritizing repair actions based on severity levels, thereby improving response time, maintenance efficiency, and public safety.

The Safe Street system is designed to execute the following core tasks:

i. **Object Detection**:
Identifies the location and boundaries of road damage using a real-time object detection model. This includes detecting potholes, alligator cracks, longitudinal cracks, and lateral cracks.

ii. **Damage Classification and Severity Assessment**:
Classifies the type of damage and evaluates its severity (e.g., minor, moderate, or severe) based on area, width, and number of occurrences. This step helps prioritize maintenance efforts by assigning urgency levels to different damages.

iii. **Summarized Report Generation:**

The system generates a structured summary using a predefined format. This summary dynamically updates based on the detected damage type, severity level, and number of occurrences. It provides a clear and concise textual report. This method ensures consistency, readability, and quick interpretation for field teams and municipal decision-makers, helping them take timely maintenance actions.

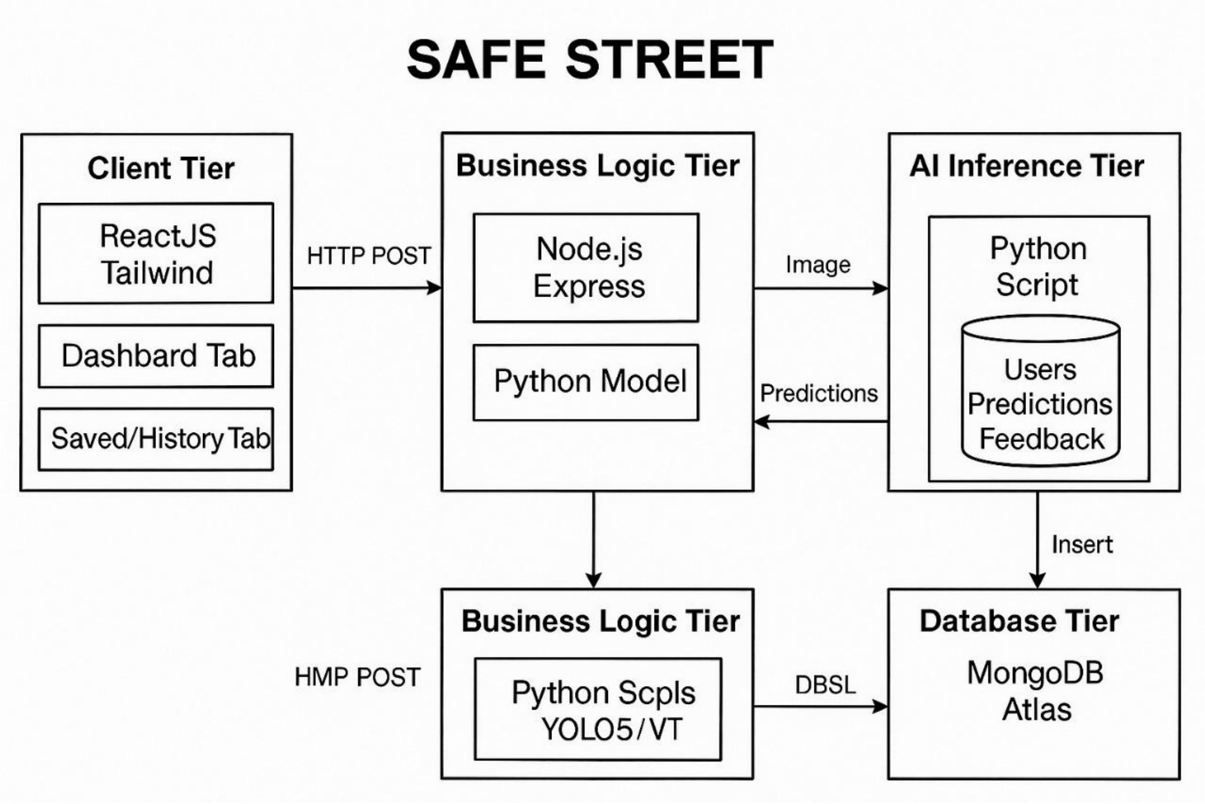


Figure 1. System architecture

The Safe Street system follows a three-tier architecture that ensures smooth data flow between components. The system integrates a frontend built with modern web technologies, a backend application server to manage business logic and API communication, an AI inference tier for deep learning-based image analysis, and a cloud-based database tier for persistent storage. This layered structure enables the system to handle real-time road damage detection and classification tasks efficiently.

Client Tier

The Client Tier is the presentation layer developed using ReactJS and Tailwind CSS. It provides users with an interactive and responsive interface to interact with the system. The dashboard tab displays high-level analytics such as the number of roads analyzed, their statuses and severity trends. Additionally, the Saved/History tab enables users to view previously analyzed images along with timestamps and geolocation. This tier initiates the data flow by sending HTTP POST requests to the backend server, which includes the road image and its location data captured from the user’s device.

Business Logic and AI Inference Tier

This middle layer is divided into two subcomponents, the Business Logic Tier and the AI Inference Tier. The Business Logic Tier is powered by Node.js and Express.js. It handles all incoming requests from the frontend, validates inputs, and forwards the image to the AI model hosted in the AI Inference Tier. Once the prediction is received, it is returned to the frontend and simultaneously logged into the database.

The AI Inference Tier consists of Python scripts executing deep learning models such as YOLOv8l or DeiT. These models are responsible for object detection and classification of road damages. The inference system processes each uploaded image, identifies damage regions or types, estimates severity, and returns structured prediction data to the backend.

Database Tier

The Database Tier uses MongoDB Atlas, a cloud-based NoSQL database, to store structured data such as prediction results, user information, image metadata, geolocation coordinates, timestamps, and user feedback. While features like real-time notifications, map overlays, and historical tracking are implemented at the application level, they are powered by the data retrieved from MongoDB. The database's flexible schema design and indexing capabilities allow for efficient querying of location-specific and user-specific records, enabling the system to support dynamic front-end features and timely backend response

3.2. Software Requirement Specifications

I. Software Requirements

Operating System: Windows 10/11

Programming Language: React.js (Frontend), Node.js with Express(backend), Python 3.10+(AI Engine)

Libraries & Frameworks:

- PyTorch
- torchvision
- transformers
- OpenCV

- PIL (Pillow)
- timm
- Mongoose

II. Hardware Requirements

Recommended:

- **CPU:** Intel i5 or above
- **RAM:** Minimum 8 GB
- **GPU:** NVIDIA GTX 1050 or above (for model training)
- **Storage:** Minimum 10 GB of free space

III. Functional Requirements

Data Collection and Storage

- **YOLOv8l Dataset:**
  - Road damage dataset annotated in YOLO format
  - Organized into train/, valid/, and test/ folders with corresponding labels
- **DeiT Dataset:**
  - Images classified into four damage categories: Pothole, Longitudinal Crack, Lateral Crack, Alligator Crack
  - Dataset stored with class-wise separation and label mapping
- **Database: MongoDB**
  - Stores image paths, bounding boxes, severity, user feedback, and report metadata

IV. Data Preprocessing

For YOLO:

- Resize images to 640×640

- Apply augmentations (rotation, brightness, flip)

For ViT:

- Normalize pixel values
- Resize to model-compatible input shape (224×224)
- Tokenize class labels for classification

Location data is captured via geotagging

Model Development

YOLOv8:

- Used for real-time object detection with bounding box output
- Severity calculated based on bounding box area

DeiT (Data-efficient Image Transformer):

- Performs classification into damage types
- Uses transformer-based attention mechanism for contextual understanding

Loss Functions:

- YOLO: CIoU loss and objectness loss
- DeiT: Cross-Entropy Loss

Visualization

- Bounding boxes drawn on image previews
- Damage types color-coded
- Frontend shows summary cards with class, severity, time, and location

Non Functional Requirements

Performance

- YOLOv8l and DeiT inference optimized using batch size tuning and mixed precision

- Model inference time recorded and displayed on UI
- Average detection and classification within 2-4 seconds

Security

- JWT-based user authentication
- OTP-based registration
- No model files or sensitive API keys are publicly accessible

Scalability

- Modular MERN stack enables future scaling
- MongoDB handles large volume of records and feedback
- AI engine supports easy integration of new models
- Real-time communication scalable using Socket.IO

3.3. Technology Stack

AI Model Pipeline (Backend AI Engine)

Tools/Libraries Used	**Description**
Python	Core language for model development
OpenCV, PIL	Used for preprocessing road images
PyTorch	Model training and inference
YOLOv8l	Detects potholes and cracks in road images
Vision Transformer	Assigns severity level to each damage
Custom Dataset	For training and evaluating detection & classification
Google Colab / Local GPU	Used for running training and testing

Table 3. Tools/ Libraries/Models used for implementing Safe Street

Web Application – MERN Stack

Layer	Tool	Purpose
Frontend	React.js	User interface for capturing images and viewing results
Backend	Node.js+Express.js	Handles API requests, model inference integration, and data flow
Database	MongoDB	Stores user inputs, image metadata, and generated reports
API Communication	Axios/Fetch	Connects React frontend with Express backend

Table 4. Technology stack for web application

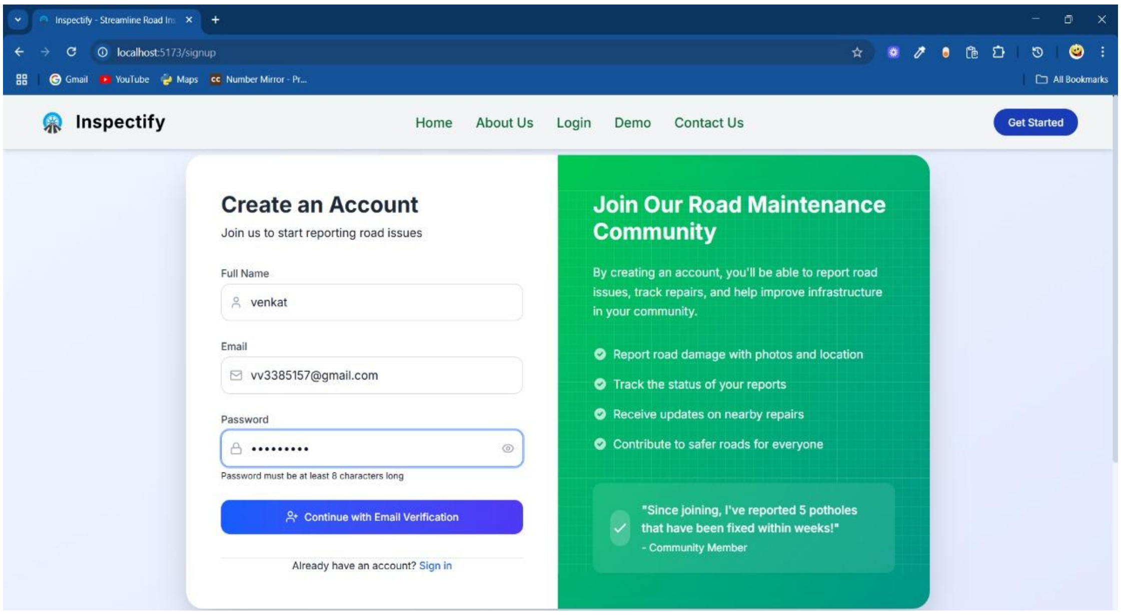
Implementation Details

3.3.1 User Login & Authentication Flow

Safe street implements a secure user authentication system with OTP-based registration and JWT-based login sessions. Safe Street uses a secure email/password authentication system combined with OTP verification during user registration. When a new user signs up, they

provide their name, email, and password. A 6-digit OTP is generated and sent via EmailJS to the provided email address. This OTP is valid for 10 minutes and must be entered to verify the user's identity. On successful verification, the user account is created and they are redirected to the login page.

During login, users submit their credentials, which are validated by the backend via the /api/login endpoint. If authentication is successful, a JWT token is issued and key user details—such as user ID, name, type (user/admin), and admin status are stored in localStorage. Based on the user’s role, they are redirected to the appropriate dashboard. Security is enforced through OTP rate limiting, password hashing, and token-based session handling to ensure safe access control.



Create an Account

Join us to start reporting road issues

Full Name: venkat

Email: vv3385157@gmail.com

Password: •••••••••

Password must be at least 8 characters long

Already have an account? Sign in

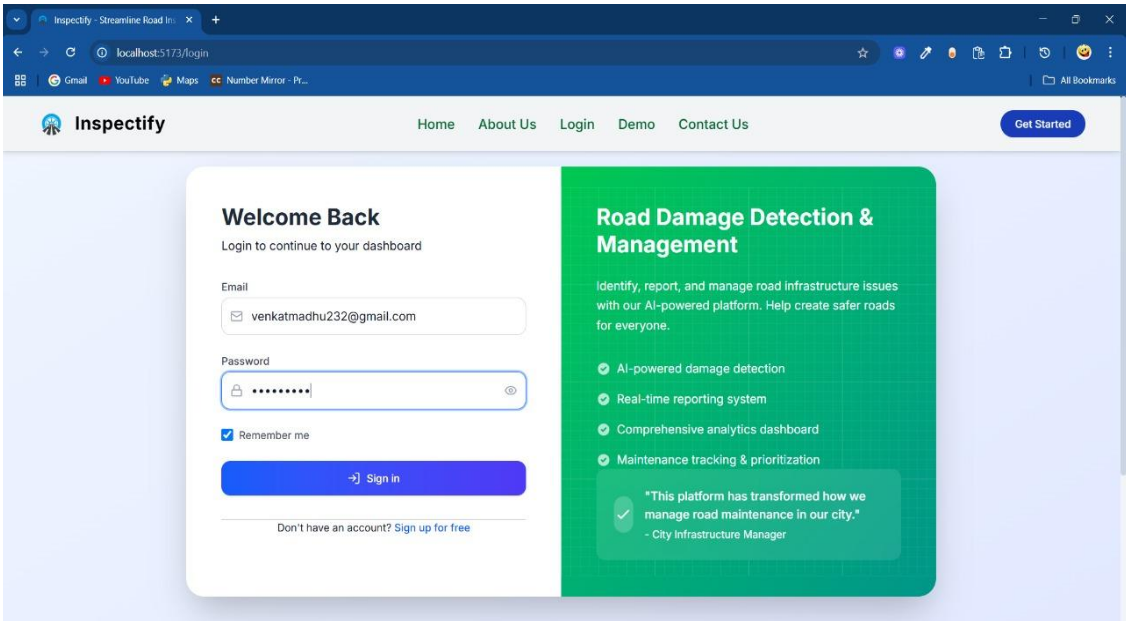
Join Our Road Maintenance Community

By creating an account, you'll be able to report road issues, track repairs, and help improve infrastructure in your community.

- Report road damage with photos and location
- Track the status of your reports
- Receive updates on nearby repairs
- Contribute to safer roads for everyone

"Since joining, I've reported 5 potholes that have been fixed within weeks!"
- Community Member

Figure 2. Sign Up Page



Welcome Back

Login to continue to your dashboard

Email: venkatmadhu232@gmail.com

Password: •••••••••

☑ Remember me

Don't have an account? Sign up for free

Road Damage Detection & Management

Identify, report, and manage road infrastructure issues with our AI-powered platform. Help create safer roads for everyone.

- AI-powered damage detection
- Real-time reporting system
- Comprehensive analytics dashboard
- Maintenance tracking & prioritization

"This platform has transformed how we manage road maintenance in our city."
- City Infrastructure Manager

Figure 3. Sign In Page

3.3.2 Image Capture via Camera with Location Access

Safe Street enables users to capture road images using their device's camera along with precise location tagging. The system first enumerates all available cameras using navigator.mediaDevices.enumerateDevices() and allows the user to select one if multiple are detected. The live video stream is accessed via getUserMedia() and displayed in a video element. An image is then captured from this stream using a canvas element and prepared for upload.

Simultaneously, the application requests for location access via the browser’s geolocation API. On user permission, it captures latitude and longitude coordinates with high accuracy and uses OpenStreetMap's Nominatim API to perform reverse geocoding, converting the coordinates into a readable address. In case geocoding fails, the raw coordinates are still used. Both the image and associated location data are bundled and sent for further analysis, with the UI providing real-time feedback throughout the capture process.

3.3.3 Image Preprocessing and Data Handling

Image Preprocessing: Preprocessing techniques are applied to images to enhance model performance and by standardizing input format. These techniques ensure that the dataset is clean, structured and suitable for training deep learning models.

1.Resizing:

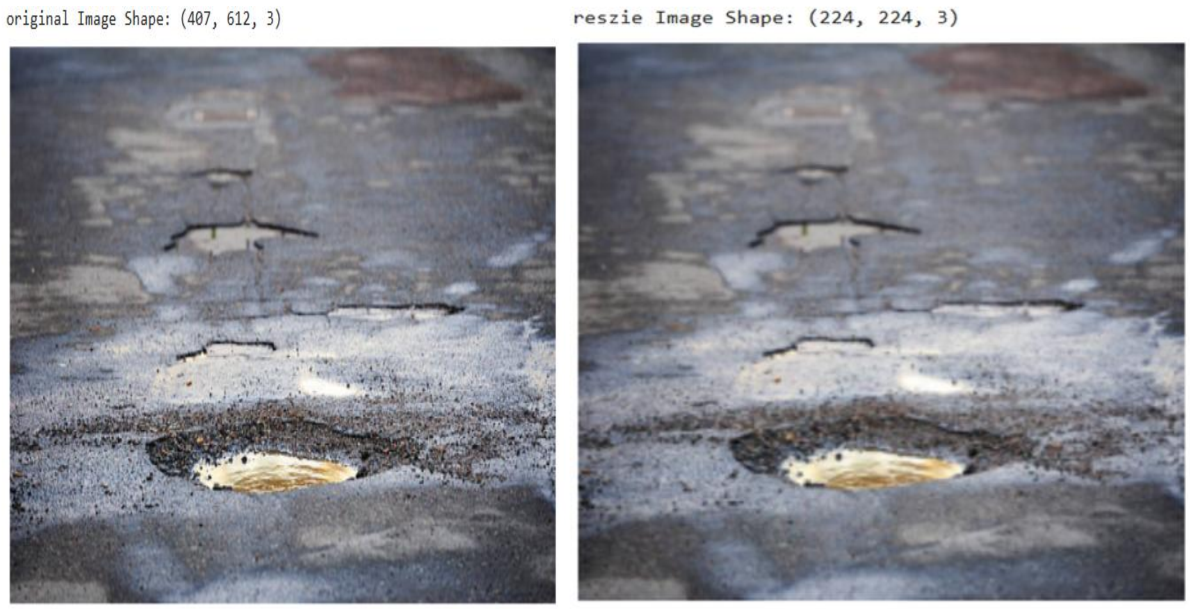
Rescales all images to a fixed dimension (e.g., 224×224 pixels) for uniformity. Ensures that the model receives images of the same size, reducing computational complexity by keeping images manageable in size.

Code for Resizing the image:

```
from PIL import Image

from torchvision import transforms

import matplotlib.pyplot as plt

import numpy as np

# Define image path

image_path = r"C:\Users\RITWIKA\Downloads\resize.jpg"

# Open the image

image = Image.open(image_path)

# Convert image to numpy array
```

```
image_array = np.array(image)

print("original Image Shape:", image_array.shape)  # Print shape

# Display the image

plt.imshow(image)  # Use cmap="gray" for grayscale images

plt.axis("off")  # Hide axis

plt.show()  # Show the image

# Open the second image (original color)

image_path1 = r"C:\Users\RITWIKA\Downloads\resize.jpg"

image1 = Image.open(image_path1)

resize_transform = transforms.Resize((224,224)) # Resize to 224x224

image = resize_transform(image)

# Resize the image

image1 = resize_transform(image1)

# Convert to numpy array

image_array1 = np.array(image1)

print("reszie Image Shape:", image_array1.shape)  # Print shape

# Display the original image

plt.imshow(image1)#prepares image to display

plt.axis("off")#revomes axis

plt.show()#displays image
```

Output:



```
original Image Shape: (407, 612, 3)
```

```
reszie Image Shape: (224, 224, 3)
```

Figure 4. Image before and after resizing

2.Rotation and Brightness, Contrast Adjustment: Rotating the image helps the model recognize objects from different orientations. This simulates real-world conditions where objects might not always appear upright. Typically rotated within a range to prevent extreme distortions. Brightness and contrast adjustment ensures the model does not reply on brightness levels for classification.

Code for Rotating the image:

```
# Import necessary libraries

from PIL import Image

from torchvision import transforms

import matplotlib.pyplot as

# Load Raw Image

image_path = r"C:\Users\RITWIKA\Downloads\Sample dataset\ds0\img\AC_7420.jpeg"

image = Image.open(image_path).convert('RGB')

# Define Transformation: Random Rotation + Brightness and Contrast Adjustment

augmentation_transform = transforms.Compose([
    transforms.RandomRotation(degrees=5),
    transforms.ColorJitter(brightness=0.5, contrast=0.5)
])
augmented_image = augmentation_transform(image)
# Display Original and Augmented Images Side by Side
plt.figure(figsize=(10, 4))
# Display the Original Image
plt.subplot(1, 2, 1)
plt.title("Original Image")
plt.imshow(image)
plt.axis('off')
# Display the Augmented Image
plt.subplot(1, 2, 2)
plt.title("After Rotation + Brightness + Contrast")
plt.imshow(augmented_image)
plt.axis('off')
# Adjust spacing between subplots and display both images
plt.tight_layout()
plt.show()
```

Output:

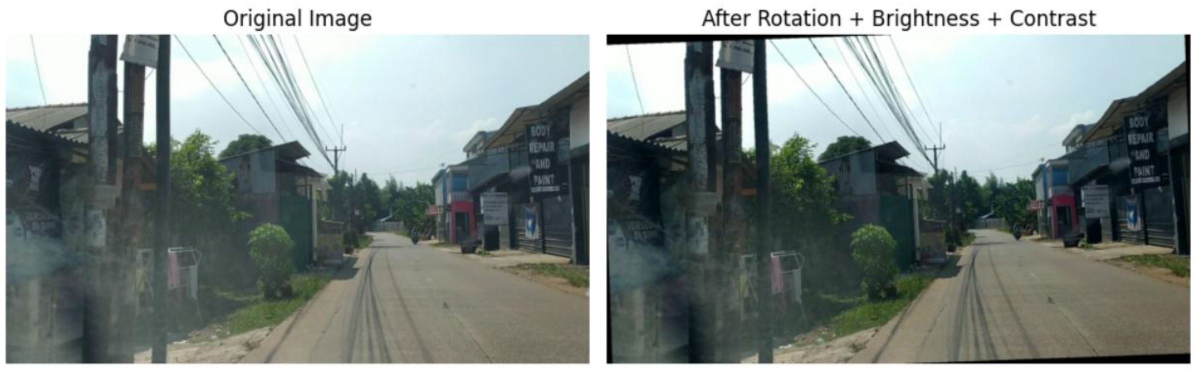


Figure 5. Image before and after rotation, applying brightness and contrast

3. Flipping: Flipping is used to augment the image by creating mirrored versions of images. This prevents the model from learning positional bias and helps in recognizing objects from different orientations**.** It Ensures that the model can recognize objects even if their left-right and upside-down orientation changes**.**

Code for Flipping the image:

```
import torchvision.transforms as transforms

from PIL import Image

import matplotlib.pyplot as plt

# Define the transformations

flip_transform = transforms.Compose([

    transforms.RandomHorizontalFlip(p=0.5),

    transforms.RandomVerticalFlip(p=0.5)])

# Load an example image

image = Image.open(r"C:\Users\RITWIKA\Downloads\flipping.jpg")  # Replace with your image file

# Apply transformations

flipped_image = flip_transform(image)

# Display original and flipped images

fig, ax = plt.subplots(1, 2, figsize=(20, 10))

ax[0].imshow(image)
```

```
ax[0].set_title("Original Image")

ax[0].axis("off")

ax[1].imshow(flipped_image)

ax[1].set_title("Flipped Image (Horizontal & Vertical)")

ax[1].axis("off")

plt.show()
```

Output:

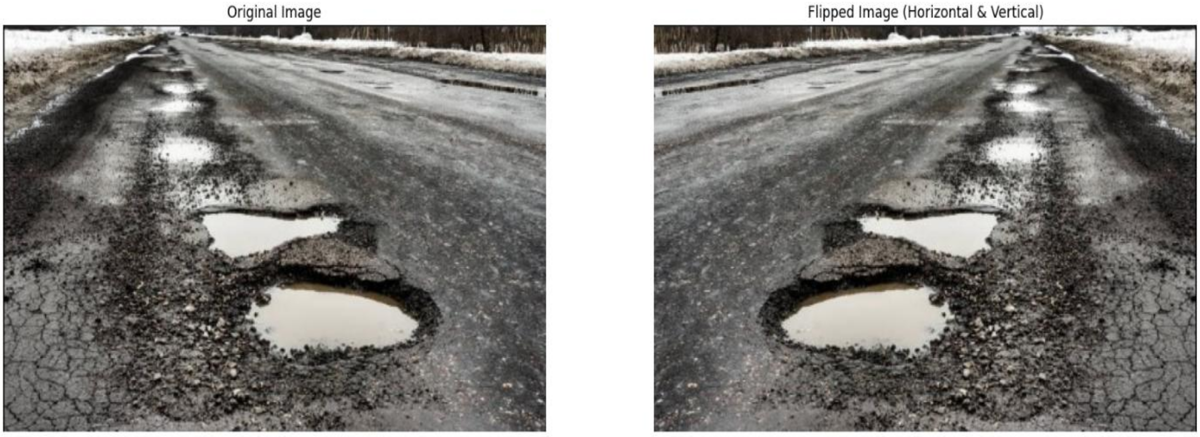


Figure 6. Image before and after flipping

4.Normalization: Normalization ensures that pixel values are standardized, improving model convergence and preventing certain features from dominating the learning process. Z-score normalization is particularly useful for road damage detection as it ensures that pixel distributions remain consistent. It ensures that all images have a mean of 0 and a standard deviation of 1 and normalized image retains important features

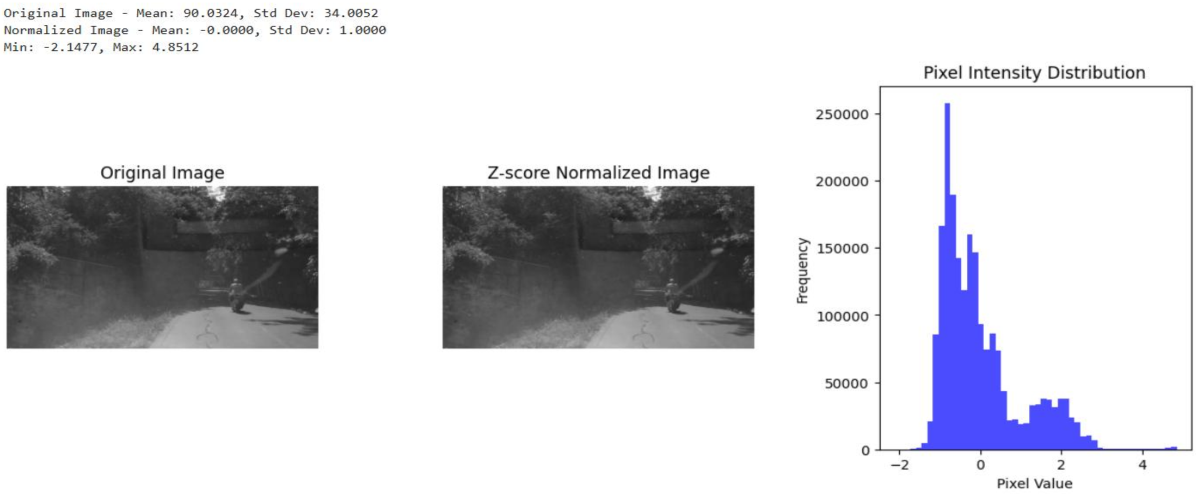
Code for Normalization:

```
import cv2

import numpy as np

import matplotlib.pyplot as plt

# Load the image in grayscale

image_path = r"C:\Users\RITWIKA\Downloads\Normalization.jpeg"
```

```
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
# Check if the image is loaded correctly
if image is None:  # if image is not found this will execute the  below print statement
    print("Error: Unable to load image. Check the file path!")
else:
    # Compute mean and standard deviation
    mean = np.mean(image)
    std = np.std(image)
    # Apply Z-score normalization
    z_score_normalized = (image - mean) / std   # this is the formulae for the calculation for z-score
    #  if the image is said to the normalized  ----> the normalized mean should be zero and standardization  should be  1
    # Print normalization stats
    print(f"Original Image - Mean: {mean:.4f}, Std Dev: {std:.4f}")
    print(f"Normalized Image - Mean: {np.mean(z_score_normalized):.4f}, Std Dev: {np.std(z_score_normalized):.4f}")
    print(f"Min: {np.min(z_score_normalized):.4f}, Max: {np.max(z_score_normalized):.4f}")
    # Rescale to 0-255 for visualization
    normalized_rescaled = cv2.normalize(z_score_normalized, None, 0, 255, cv2.NORM_MINMAX) # this normalized image is again converted into the RGB
    normalized_rescaled = np.uint8(normalized_rescaled) # converts the image back to an interger format
    # Display original and processed images
    fig, axes = plt.subplots(1, 3, figsize=(15, 5))
    axes[0].imshow(image, cmap="gray")
    axes[0].set_title("Original Image")
```

```
axes[0].axis("off")

axes[1].imshow(normalized_rescaled, cmap="gray")

axes[1].set_title("Z-score Normalized Image")

axes[1].axis("off")

# Adjust spacing between subplots

plt.subplots_adjust(wspace=0.4)

# Plot histogram to check normalization

axes[2].hist(z_score_normalized.ravel(), bins=50, color='blue', alpha=0.7)

axes[2].set_title("Pixel Intensity Distribution")

axes[2].set_xlabel("Pixel Value")

axes[2].set_ylabel("Frequency")

plt.show()  #Displays all three subplots (original image, normalized image, histogram)
```

Output:



```
Original Image - Mean: 90.0324, Std Dev: 34.0052
Normalized Image - Mean: -0.0000, Std Dev: 1.0000
Min: -2.1477, Max: 4.8512
```

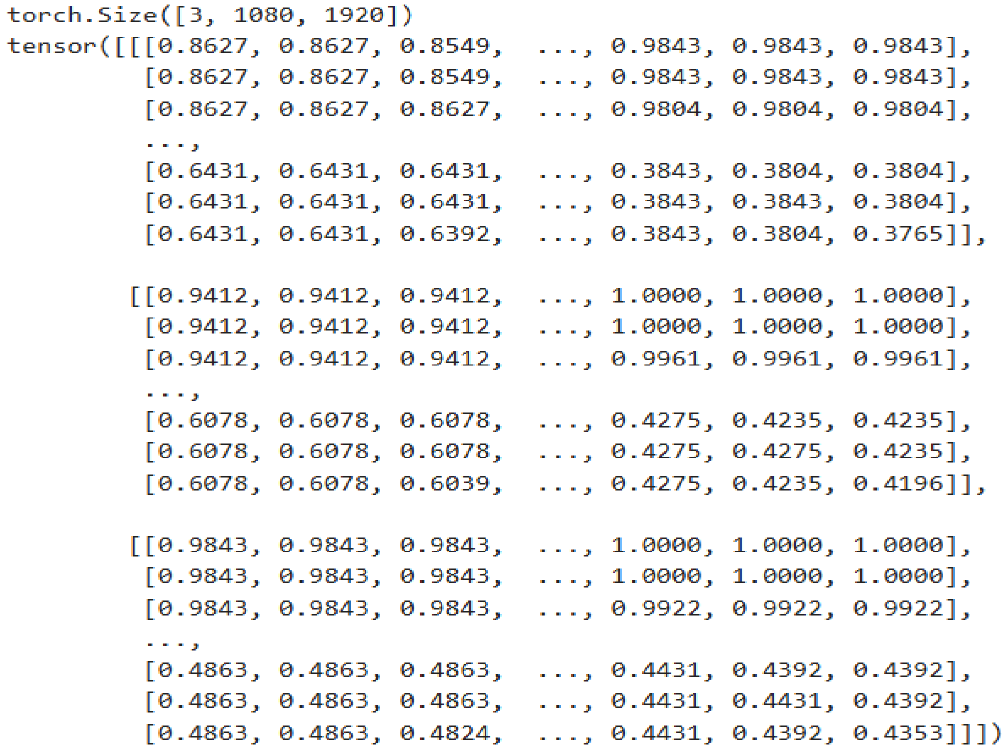
Figure 7. Image before and after Normalizing

5. ToTensor: The ToTensor transformation is applied to convert images into numerical tensor format, which is required for deep learning models in PyTorch. It also scales pixel values from [0, 255] to [0, 1], by reshaping them into(C, H, W) format, making computations more efficient.

Code for converting image to tensors:

```
from torchvision import transforms

from PIL import Image

image = Image.open(r"C:\Users\RITWIKA\Downloads\Sample dataset\ds0\img\AR_1820.jpeg")

transform = transforms.ToTensor() #Define a transform to convert the image to a tensor and scales pixel values from [0, 255] to [0, 1]

tensor_image = transform(image) #applying tranform on image

print(tensor_image.shape)  # Output: (C, H, W) shape of the tensor

print(tensor_image)  # Displays pixel values between [0,1]
```

Output:



```
torch.Size([3, 1080, 1920])
tensor([[[0.8627, 0.8627, 0.8549,  ..., 0.9843, 0.9843, 0.9843],
         [0.8627, 0.8627, 0.8549,  ..., 0.9843, 0.9843, 0.9843],
         [0.8627, 0.8627, 0.8627,  ..., 0.9804, 0.9804, 0.9804],
         ...,
         [0.6431, 0.6431, 0.6431,  ..., 0.3843, 0.3804, 0.3804],
         [0.6431, 0.6431, 0.6431,  ..., 0.3843, 0.3843, 0.3804],
         [0.6431, 0.6431, 0.6392,  ..., 0.3843, 0.3804, 0.3765]],

        [[0.9412, 0.9412, 0.9412,  ..., 1.0000, 1.0000, 1.0000],
         [0.9412, 0.9412, 0.9412,  ..., 1.0000, 1.0000, 1.0000],
         [0.9412, 0.9412, 0.9412,  ..., 0.9961, 0.9961, 0.9961],
         ...,
         [0.6078, 0.6078, 0.6078,  ..., 0.4275, 0.4235, 0.4235],
         [0.6078, 0.6078, 0.6078,  ..., 0.4275, 0.4275, 0.4235],
         [0.6078, 0.6078, 0.6039,  ..., 0.4275, 0.4235, 0.4196]],

        [[0.9843, 0.9843, 0.9843,  ..., 1.0000, 1.0000, 1.0000],
         [0.9843, 0.9843, 0.9843,  ..., 1.0000, 1.0000, 1.0000],
         [0.9843, 0.9843, 0.9843,  ..., 0.9922, 0.9922, 0.9922],
         ...,
         [0.4863, 0.4863, 0.4863,  ..., 0.4431, 0.4392, 0.4392],
         [0.4863, 0.4863, 0.4863,  ..., 0.4431, 0.4431, 0.4392],
         [0.4863, 0.4863, 0.4824,  ..., 0.4431, 0.4392, 0.4353]]])
```

Figure 8. Tensor values

Multiclass Classification Using CNN

Before implementing advanced models like Vision Transformers and YOLO in the Safe Street system, we explored Convolutional Neural Networks (CNNs) for a traditional multiclass classification task. This helped us build a strong foundational understanding of how deep learning models work. This experiment was designed to classify images into one of five categories-streets, buildings, seas, mountains, and forests, based on visual features. Although the dataset was unrelated to road damage detection, this helped us understand how neural networks process and differentiate between multiple classes.

The CNN model was built with three convolutional layers followed by ReLU activations and max pooling, which progressively extracted spatial features from the input image. These feature maps were then flattened and passed through two fully connected layers. The final output was processed using the Softmax activation function.

The dataset was organized into folders by class labels and loaded using ImageFolder from torchvision.datasets. Each image was resized to 128×128 pixels and normalized to enhance model convergence. Training was conducted over 5 epochs using CrossEntropyLoss, a standard loss function for multiclass classification, and the Adam optimizer for efficient parameter updates.

Once trained, the model was saved and used for testing on new, unseen images. A utility function was defined to load and preprocess a test image, run it through the model, and return the predicted class name. The final prediction was visualized using matplotlib for quick validation.

This hands-on experiment reinforced our understanding of:

- Data preprocessing and normalization,
- The role of convolutional layers in feature extraction,
- Loss functions for multiclass classification,
- Model evaluation using predicted class labels.

Multiclass classification code

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import PIL.Image as Image
# Define transformations (resize, convert to tensor, normalize)
transform = transforms.Compose([
    transforms.Resize((128, 128)),  # Resize images to 128x128
    transforms.ToTensor(),          # Convert images to tensors
    transforms.Normalize((0.5,), (0.5,))  # Normalize
])
dataset_path = r"C:\Users\RITWIKA\Downloads\Multiclass classification"
train_dataset = datasets.ImageFolder(root=dataset_path, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=4, shuffle=True)
print("Class labels:", train_dataset.classes)  # ['streets', 'buildings', 'seas', 'mountains', 'forests']
num_classes = len(train_dataset.classes)
#Define Multiclass CNN Model
class CNN(nn.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 16 * 16, 128)
        self.fc2 = nn.Linear(128, num_classes)
        self.softmax = nn.Softmax(dim=1)
    def forward(self, x):
        x = self.pool(nn.ReLU()(self.conv1(x)))
        x = self.pool(nn.ReLU()(self.conv2(x)))
        x = self.pool(nn.ReLU()(self.conv3(x)))
        x = x.view(-1, 64 * 16 * 16)  # Flatten
        x = nn.ReLU()(self.fc1(x))
        x = self.fc2(x)
        x = self.softmax(x)
        return x
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CNN(num_classes=num_classes).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
num_epochs = 5
for epoch in range(num_epochs):
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(train_loader):.4f}")
print("Training complete!")
model_path = "multiclass_cnn.pth"
torch.save(model.state_dict(), model_path)
```

```
print(f"Model saved to {model_path}")
model = CNN(num_classes=num_classes).to(device)
model.load_state_dict(torch.load("multiclass_cnn.pth"))
model.eval()
print("Model loaded successfully!")
def predict_image(image_path):
    image = Image.open(image_path)
    image = transform(image).unsqueeze(0).to(device)
    with torch.no_grad():
        output = model(image)
    predicted_class = torch.argmax(output, dim=1).item()
    class_name = train_dataset.classes[predicted_class]
    return class_name
test_image_path = r"C:\Users\RITWIKA\Downloads\Multiclass classification\test.jpeg"
plt.imshow(Image.open(test_image_path))
plt.axis("off")
plt.title(f"Predicted: {predict_image(test_image_path)}")
plt.show()
```

Output:

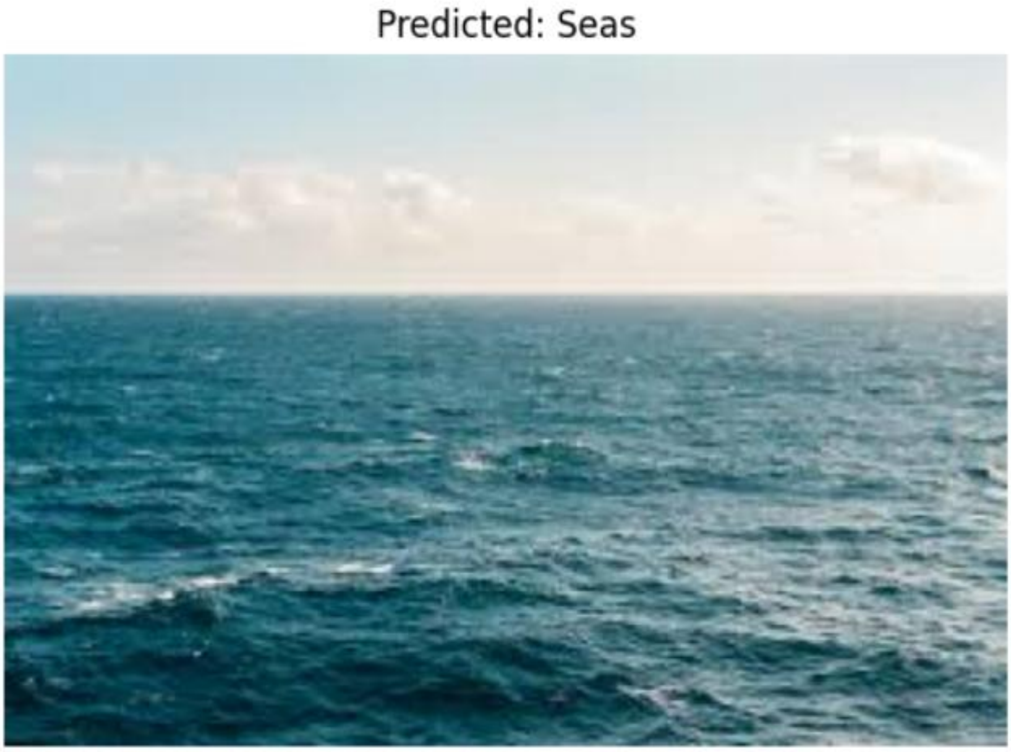


Figure 9. Multiclass classification predicted output

Dataset Description

To effectively train and evaluate the Safe Street system, two separate custom datasets were created, one for YOLOv8l and another for DeiT. Both datasets were manually created and structured to match the respective model requirements.

1. YOLOv8 Object Detection Dataset

A custom dataset was created for road damage detection using YOLOv8l. It follows the YOLO annotation format and contains 456 labeled images.

- **Train:** 320 images
- **Validation:** 92 images
- **Test:** 46 images

Folder Structure:

Organized into train/, valid/, and test/ folders with corresponding annotation .txt files.

Annotation Format:

Each object is annotated as:

<class_id> <x_center> <y_center> <width> <height>

All values are normalized (0-1). Each .txt file matches its image name.

Metadata Files:

- data.yaml – defines class names and dataset paths
- README.dataset.txt – describes sources
- README.roboflow.txt – export and preprocessing details

2. DeiT Classification Dataset

A separate dataset was prepared for image classification using the Vision Transformer (DeiT) model. Unlike object detection, this dataset focused on classifying full images based on the predominant type of road damage.

- **Classes**:
  - Pothole (700 images)
  - Alligator Crack (700 images)
  - Longitudinal Crack (700 images)
  - Lateral Crack (700 images)
  - Mixed Damage (300 images)
- **Total Images**: 3100
- **Labelling**: Images were manually categorized into folders based on visible damage type.

This dataset allowed the DeiT-based model to learn high-level semantic features of different damage categories and classify entire road images accordingly.

3.3.4. Road Damage Detection Module (Object Detection)

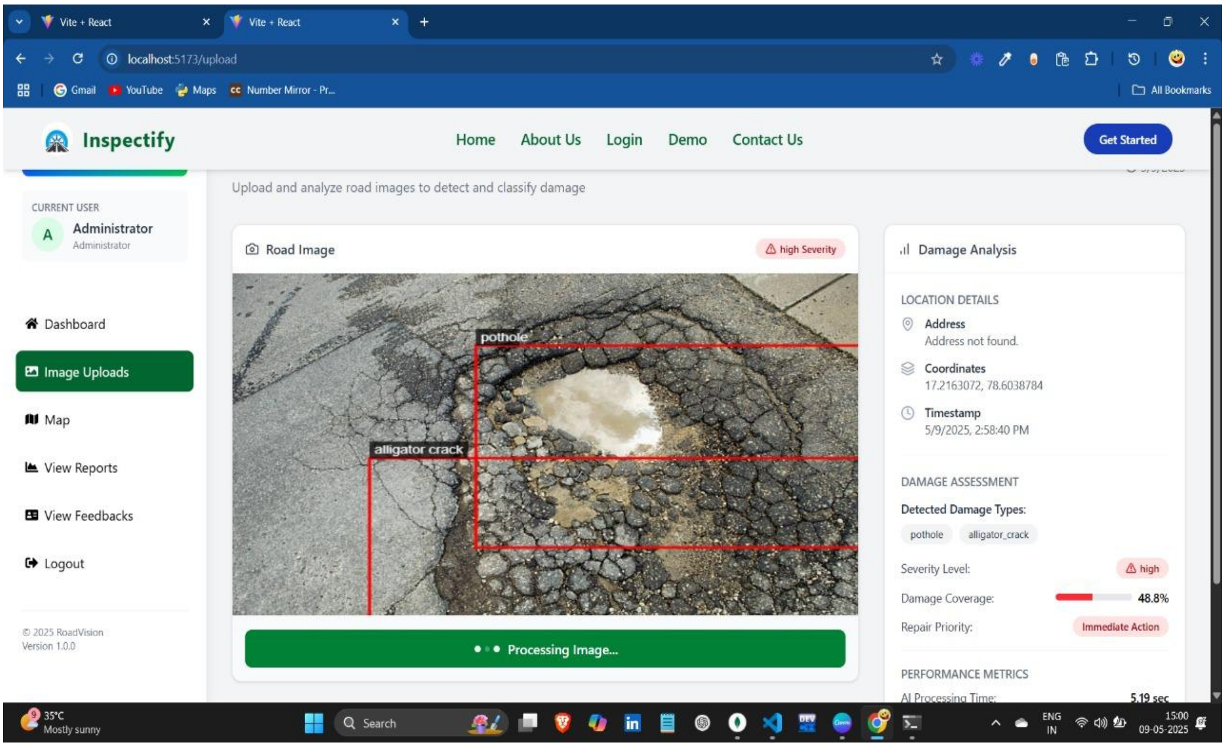
This application integrates an AI-powered module to detect road damages using object detection techniques. This module identifies potholes, cracks, and other forms of damage from images captured.

The process begins with the user capturing a road image through the system's interface. Alongside the image, geolocation data, specifically latitude and longitude which is automatically captured and sent to the backend for further processing.

Once received, the backend initiates AI-based processing using a pre-trained object detection model. This model analyzes the image to detect different types of road damage such as potholes, alligator cracks, longitudinal cracks, and lateral cracks. For each detection, the system generates output including the damage type, confidence score, bounding box coordinates, and an estimation of severity based on the area and number of detections.

After processing, the results are transmitted back to the frontend for visualization. The detected damages are rendered using the HTML5 Canvas API. Each bounding box is color-coded according to the type of damage and is accompanied by a label indicating its class. The implementation dynamically adjusts to varying image sizes to ensure accurate rendering of the detection output.

From a technical perspective, the image and location are packaged using the FormData API and transmitted through a POST request to the /analyze-damage endpoint. The frontend uses JavaScript Canvas to draw bounding boxes and overlay labels on the image. Additionally, real-time feedback is provided to the user using loading indicators and timestamps that reflect both backend inference and client-side processing times.



Upload and analyze road images to detect and classify damage

Damage Analysis

LOCATION DETAILS

Address
Address not found.

Coordinates
17.2163072, 78.6038784

Timestamp
5/9/2025, 2:58:40 PM

DAMAGE ASSESSMENT

Detected Damage Types:
pothole alligator_crack

Severity Level: high

Damage Coverage: 48.8%

Repair Priority: Immediate Action

PERFORMANCE METRICS

AI Processing Time: 5.19 sec

Figure 10. Damage Analysis

Yolov8l code:

```
from google.colab import drive

drive.mount('/content/drive')

# Copy the zip file from your Google Drive to Colab

!cp "/content/drive/MyDrive/Highly damaged dataset.yolov8.zip" "/content/"

# Unzip the dataset

!unzip "/content/Highly damaged dataset.yolov8.zip" -d "/content/road_damage_dataset"

!ls "/content/road_damage_dataset"

!cat /content/road_damage_dataset/data.yaml

from ultralytics import YOLO

model = YOLO("yolov8l.pt")

model.train(
    data='/content/road_damage_dataset/data.yaml',  # Path to your data.yaml
    epochs=80,  # Set the number of epochs (can be adjusted based on your system)
    batch=16,   # Batch size
    imgsz=640,  # Image size (can try 640 or 416)
    workers=4   # Number of workers for data loading
)
results = model.val()
model.predict('/content/road_damage_dataset/test/images')
# Search for 'best.pt' in all subdirectories of /content
!find /content -name "best.pt"
model = YOLO('/content/runs/detect/train/weights/best.pt')
!find /content -name "best.pt"
!ls /content/runs/detect
from google.colab import files
uploaded = files.upload()
from ultralytics import YOLO
from IPython.display import Image
model = YOLO('/content/runs/detect/train/weights/best.pt')
image_path = 'road_damage.png'
results = model.predict(image_path)
from ultralytics import YOLO
from IPython.display import Image, display
import os
```

```
# Run prediction again to make sure something is saved
predict_results = model.predict('/content/road_damage_dataset/test/images', save=True)
# Print where results are saved
print("Saved prediction directory:", predict_results[0].save_dir)
# List files in prediction folder
predicted_dir = predict_results[0].save_dir
predicted_files = os.listdir(predicted_dir)
print("Files in prediction folder:", predicted_files)
# Display the first image with bounding boxes
for file in predicted_files:
    if file.endswith(('.jpg', '.png')):
        output_path = os.path.join(predicted_dir, file)
        display(Image(filename=output_path))
        break
else:
    print("No image found in prediction output.")
import shutil
# Remove old prediction folder if it exists
shutil.rmtree('runs/detect/predict', ignore_errors=True)
from google.colab import files
uploaded = files.upload()
# Predict the new image
predict_result = model.predict(source='pothole.png', save=True)
```

```
# Display the result
from IPython.display import Image, display
import os
predicted_dir = predict_result[0].save_dir
predicted_files = os.listdir(predicted_dir)
for file in predicted_files:
    if file.endswith(('.jpg', '.png')):
        output_path = os.path.join(predicted_dir, file)
        display(Image(filename=output_path))
        break
    else:
    print("No image found in prediction output.")
```

Output:

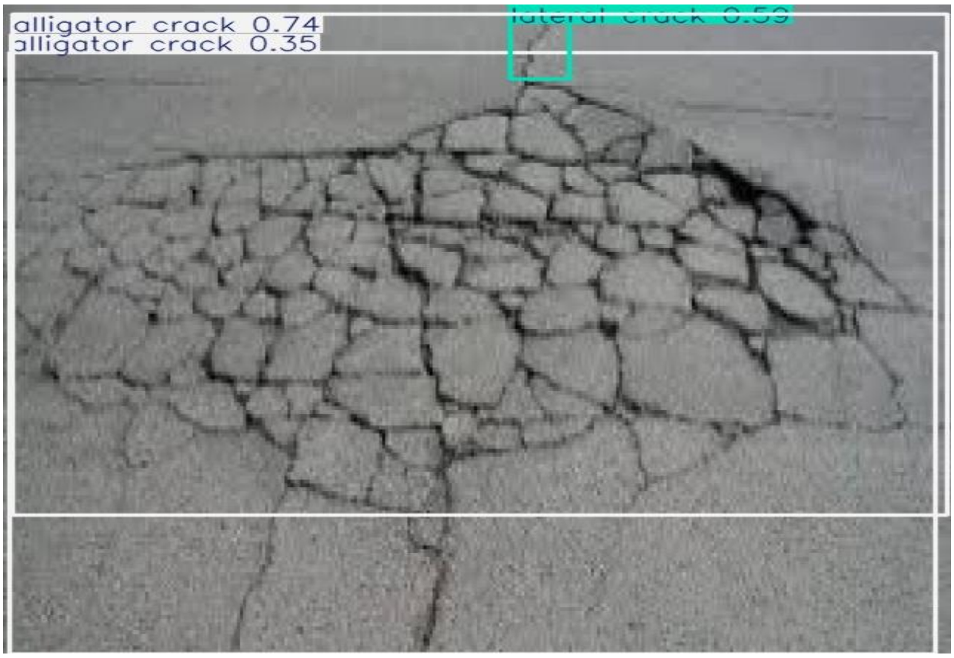


Figure 11. Yolov8l image with bounded box

DeiT Code:

```
from google.colab import drive

drive.mount('/content/drive')

import os

import pandas as pd

# Paths

csv_path = "/content/drive/MyDrive/hmm/old/labels/final_labels.xls"

images_dir = "/content/drive/MyDrive/hmm/old"  # since images are directly here

# Load CSV

df = pd.read_csv(csv_path)

# Check if each image exists

df["exists"] = df["filename"].apply(lambda x: os.path.exists(os.path.join(images_dir, x)))

# Report

total = len(df)

valid = df["exists"].sum()

missing = total - valid

print(f"Total annotations in CSV: {total}")

print(f"Images found: {valid}")

print(f"Missing image files: {missing}")

# List missing images (optional)

if missing > 0:

    print("\nMissing image filenames:")

    print(df[~df["exists"]]["filename"].tolist())
```

```
# Save a cleaned version
df_valid = df[df["exists"]].drop(columns=["exists"])
df_valid.to_csv("/content/drive/MyDrive/hmm/old/labels/final_labels_cleaned.csv",
index=False)
print("\nCleaned CSV saved as: final_labels_cleaned.csv")
from torchvision import transforms
# Standard ViT preprocessing values (from timm)
vit_mean = [0.5, 0.5, 0.5]
vit_std = [0.5, 0.5, 0.5]
# Transforms
train_transforms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),          # Augmentation
    transforms.RandomRotation(10),              # Augmentation
    transforms.ToTensor(),
    transforms.Normalize(mean=vit_mean, std=vit_std)
])
val_transforms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=vit_mean, std=vit_std)])
from torch.utils.data import Dataset
from PIL import Image
import torch
```

```
class RoadDamageDataset(Dataset):
    def __init__(self, csv_path, image_dir, transform=None):
        self.data = pd.read_csv(csv_path)
        self.image_dir = image_dir
        self.transform = transform
        self.label_cols = ["pothole", "longitudinal_crack", "lateral_crack", "alligator_crack"]
    def __len__(self):
        return len(self.data)
    def __getitem__(self, idx):
        row = self.data.iloc[idx]
        img_path = os.path.join(self.image_dir, row["filename"])
        image = Image.open(img_path).convert("RGB")
        if self.transform:
            image = self.transform(image)
        labels = torch.tensor(row[self.label_cols].values.astype("float32"))  # Multi-label
        return image, labels
from torch.utils.data import random_split, DataLoader
# Paths
csv_path = "/content/drive/MyDrive/hmm/old/labels/final_labels_cleaned.csv"
image_dir = "/content/drive/MyDrive/hmm/old"
# Dataset
full_dataset = RoadDamageDataset(csv_path, image_dir, transform=train_transforms)
# Split into train and validation (80/20)
```

```
train_size = int(0.8 * len(full_dataset))
val_size = len(full_dataset) - train_size
train_dataset, val_dataset = random_split(full_dataset, [train_size, val_size])
# Apply val_transforms to validation set
val_dataset.dataset.transform = val_transforms
# DataLoaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=2)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False, num_workers=2)
images, labels = next(iter(train_loader))
print(images.shape)   # torch.Size([32, 3, 224, 224])
print(labels.shape)   # torch.Size([32, 4])
import matplotlib.pyplot as plt
import numpy as np
# Helper: de-normalize image for visualization
def denormalize(tensor):
    mean = torch.tensor([0.5, 0.5, 0.5]).view(3, 1, 1)
    std = torch.tensor([0.5, 0.5, 0.5]).view(3, 1, 1)
    return tensor * std + mean
# Pick a sample index
sample_idx = 10  # Change this to view different samples
# Get raw data from the CSV
row = pd.read_csv(csv_path).iloc[sample_idx]
img_path = os.path.join(image_dir, row["filename"])
```

```
original_img = Image.open(img_path).convert("RGB")

label=row[["pothole","longitudinal_crack","lateral_crack","alligator_crack"]].values.astype("int")

# Apply transform (to simulate augmented image)

transformed_img = train_transforms(original_img)

transformed_img_vis = denormalize(transformed_img).permute(1, 2, 0).numpy()

# Plot

plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)

plt.imshow(original_img)

plt.title(f"Original Image\nLabels: {label}")

plt.axis("off")

plt.subplot(1, 2, 2)

plt.imshow(transformed_img_vis)

plt.title("After Augmentation")

plt.axis("off")

plt.tight_layout()

plt.show()
```

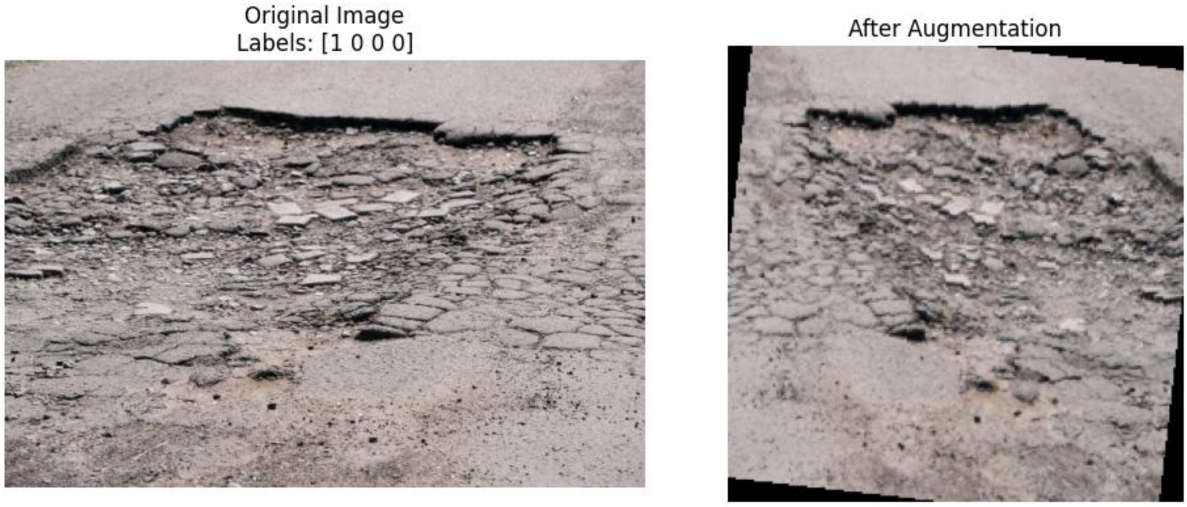


Figure 12. Image before and after preprocessing

```
import timm
import torch.nn as nn
# Load pretrained DeiT-Tiny
model = timm.create_model('deit_tiny_patch16_224', pretrained=True)
# Modify classifier for 4 output labels (multi-label)
model.head = nn.Sequential(
    nn.Linear(model.head.in_features, 4),
    nn.Sigmoid()  # For multi-label (not softmax!))
import torch.nn as nn
criterion = nn.BCELoss()  # Binary Cross Entropy for multi-label
import torch.optim as optim
optimizer = optim.Adam(model.parameters(), lr=1e-4, weight_decay=1e-5)
# (Optional) Learning rate scheduler
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.5)
from tqdm import tqdm
```

```
def train_one_epoch(model, dataloader, optimizer, criterion, device):
    model.train()
    running_loss = 0.0
    for images, labels in tqdm(dataloader):
        images, labels = images.to(device), labels.to(device).float()
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * images.size(0)
    return running_loss / len(dataloader.dataset)
def validate(model, dataloader, criterion, device):
    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device).float()
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item() * images.size(0)
    return val_loss / len(dataloader.dataset)
import torch
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
# Training for one round of epochs without saving
def run_training_epochs(model, train_loader, val_loader, optimizer, criterion, scheduler,
device, num_epochs):
    history = {"train_loss": [], "val_loss": []}
    for epoch in range(num_epochs):
        train_loss = train_one_epoch(model, train_loader, optimizer, criterion, device)
        val_loss = validate(model, val_loader, criterion, device)
        scheduler.step()  # Optional
          print(f"Epoch [{epoch+1}/{num_epochs}] | Train Loss: {train_loss:.4f} | Val Loss:
{val_loss:.4f}")
        history["train_loss"].append(train_loss)
        history["val_loss"].append(val_loss)
    return history
# Save model manually if you believe it's the best one
def save_model(model, path="best_vit_multi_label.pth"):
    torch.save(model.state_dict(), path)
    print(f"Model saved to {path}")
# Load model weights from saved checkpoint
def load_model(model, path="best_vit_multi_label.pth", device='cuda' if
torch.cuda.is_available() else 'cpu'):
    model.load_state_dict(torch.load(path, map_location=device))
    print(f" Loaded model from {path}")
    return model
# Example usage (run training and save manually)
num_epochs = 5
history = run_training_epochs(model, train_loader, val_loader, optimizer, criterion, scheduler, device, num_epochs)
# Optional: If final epoch has lowest val loss, save it
save_model(model)
```

In another file for loading the saved path and predicting

```
from google.colab import drive
drive.mount('/content/drive')
# Step 1: Imports
import torch
import torch.nn as nn
from torchvision import transforms
from PIL import Image
import matplotlib.pyplot as plt
from google.colab import files
import timm
# Step 2: Load model
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# Create model and modify for 4-label multi-label prediction
model = timm.create_model('_tiny_patch16_224', pretrained=False)
model.head = nn.Sequential(
    nn.Linear(model.head.in_features, 4),
    nn.Sigmoid())
# Load your trained weights
model_path = "/content/drive/MyDrive/"best_vit_multi_label.pth",
model.load_state_dict(torch.load(model_path, map_location=device))
model.to(device)
model.eval()
# Step 3: Define transforms (from your val_transforms)
vit_mean = [0.5, 0.5, 0.5]
vit_std = [0.5, 0.5, 0.5]
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=vit_mean, std=vit_std)])
# Step 4: Upload image
uploaded = files.upload()
for filename in uploaded.keys():
    # Load and display the image
    img = Image.open(io.BytesIO(uploaded[filename])).convert("RGB")
    plt.imshow(img)
    plt.axis('off')
    plt.title("Uploaded Image")
    plt.show()
    # Transform and predict
    input_tensor = transform(img).unsqueeze(0).to(device)
    threshold = 0.05# Try 0.2 to 0.4 depending on output behavior
    with torch.no_grad():
        output = model(input_tensor)
        print("Raw model outputs:", output.cpu().numpy())  # For debugging
        predicted = (output > threshold).int().squeeze().cpu().numpy()
    # Step 5: Display predictions
    labels = ["pothole", "longitudinal_crack", "lateral_crack", "alligator_crack"]
    print("\nPredicted Damage Types:")
    for i, label in enumerate(labels):
        if predicted[i]:
            print(f"{label}")
```

Output:

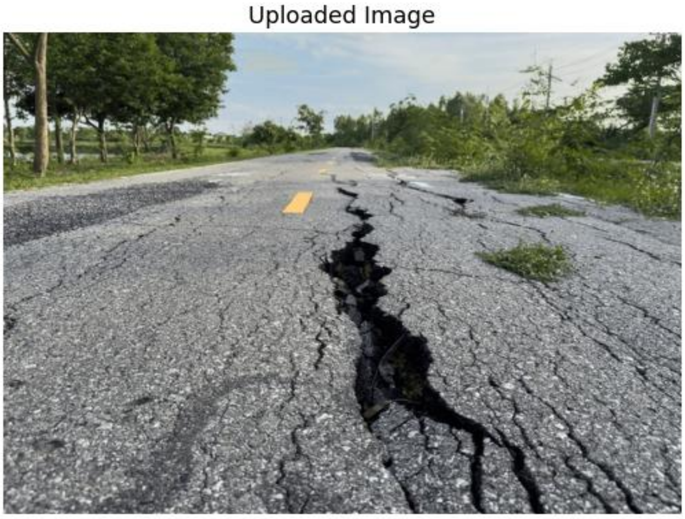


Figure 13. Uploaded image for DeiT prediction

```
Raw model outputs: [[0.00954702   0.6907124   0.12942971   0.03603716]]
Predicted Damage Types:
```

longitudinal_crack

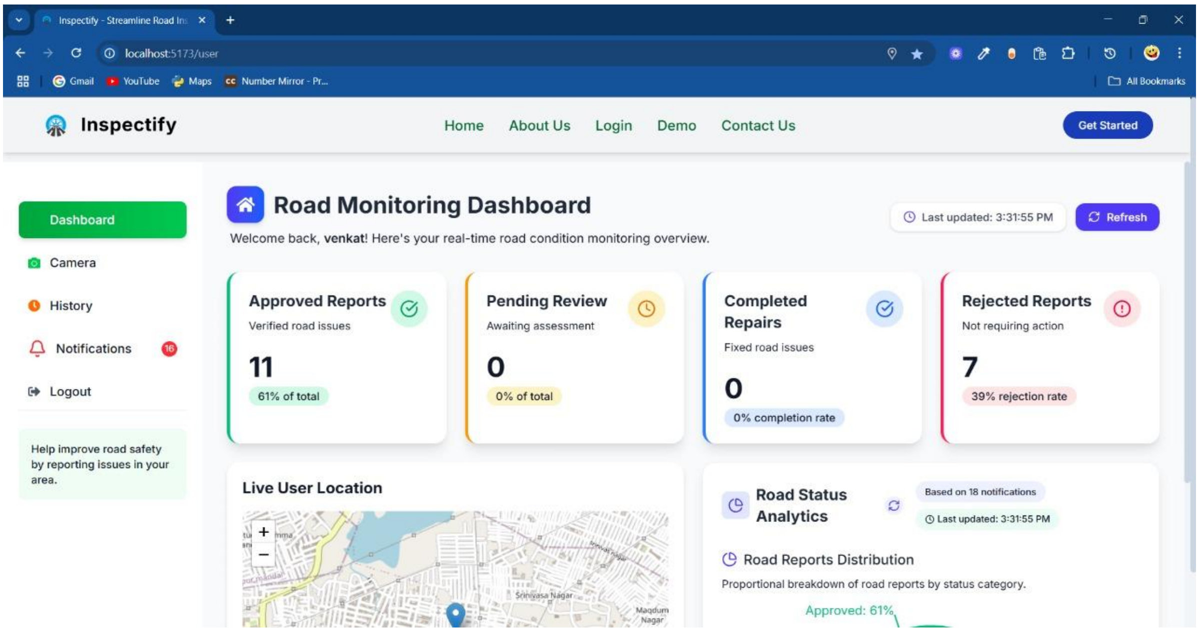
lateral_crack

3.3.5 Result Display on User Dashboard & Real time notifications

The system enhances user experience by delivering timely and detailed results through the user dashboard, along with real-time notifications to keep users informed throughout the damage analysis process. Once an image has been submitted and analyzed, the results are displayed on the user's dashboard using clearly structured cards. Each card includes the detected damage type, the severity level, visually highlighted with color-coded indicators, and the geographical location of the damage. This allows users to easily interpret the severity and type of damage reported. The dashboard also maintains a history of previously submitted images and their results, allowing users to track reports over time using sortable and filterable views, including a timeline format for improved usability.

To support immediate user feedback, the system integrates Socket.IO for real-time communication. As soon as an image is processed, the backend server emits events such as analysis-complete or prediction-complete, which are captured on the frontend. Users receive instant toast notifications for events like successful analysis, processing errors, or status changes. Important updates are also displayed using persistent notification cards and visual badges for unread alerts. These notifications help ensure that users are continuously aware of their report status without needing to manually refresh or check their dashboard.

In addition to notifications, the system includes progress indicators during the image analysis phase. Users can view the current status through animated loading states and a percentage-based progress bar that estimates the time remaining for processing.



Road Monitoring Dashboard

Welcome back, **venkat**! Here's your real-time road condition monitoring overview.

Last updated: 3:31:55 PM

Help improve road safety by reporting issues in your area.

Approved Reports	Pending Review	Completed Repairs	Rejected Reports
Verified road issues	Awaiting assessment	Fixed road issues	Not requiring action
11	0	0	7
61% of total	0% of total	0% completion rate	39% rejection rate

Road Status Analytics

Based on 18 notifications

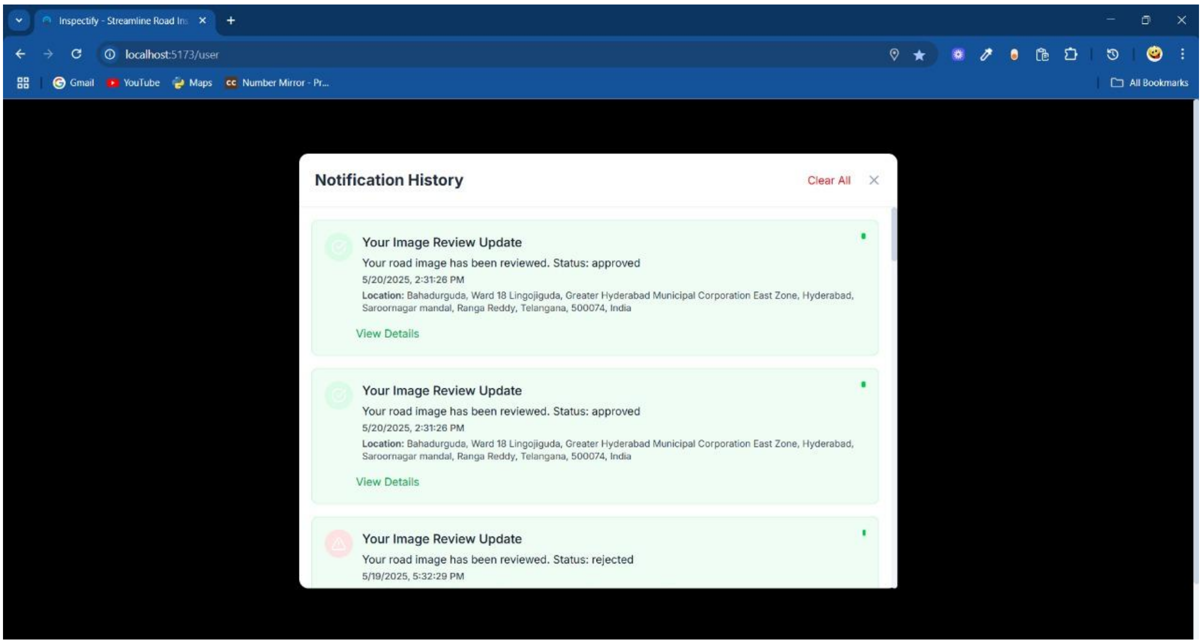
Last updated: 3:31:55 PM

Road Reports Distribution

Proportional breakdown of road reports by status category.

Approved: 61%

Figure 14. User Dashboard



Notification History

Your Image Review Update

Your road image has been reviewed. Status: approved

5/20/2025, 2:31:26 PM

Location: Bahadurguda, Ward 18 Lingojiguda, Greater Hyderabad Municipal Corporation East Zone, Hyderabad, Saroornagar mandal, Ranga Reddy, Telangana, 500074, India

View Details

Your Image Review Update

Your road image has been reviewed. Status: approved

5/20/2025, 2:31:26 PM

Location: Bahadurguda, Ward 18 Lingojiguda, Greater Hyderabad Municipal Corporation East Zone, Hyderabad, Saroornagar mandal, Ranga Reddy, Telangana, 500074, India

View Details

Your Image Review Update

Your road image has been reviewed. Status: rejected

5/19/2025, 5:32:29 PM

Figure 15. Notification tab in user dashboard

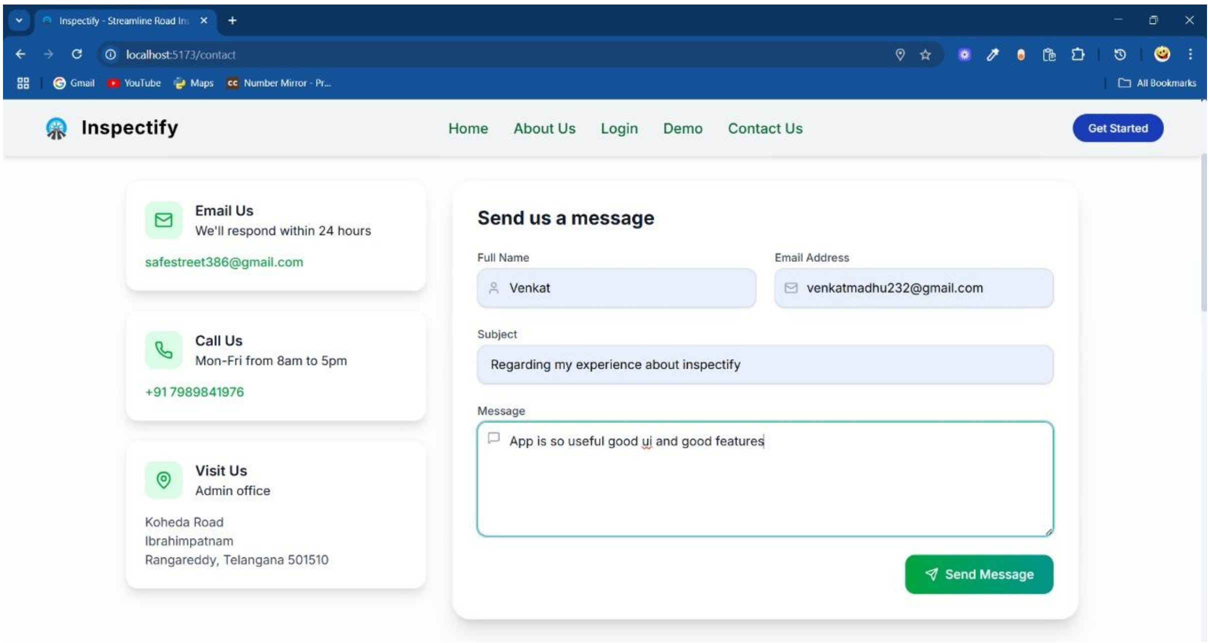
3.3.6 Feedback Mechanism from User

Users are provided with an intuitive feedback form that appears after damage analysis results are displayed. This form allows users to share their experience, categorize any issues they encountered and add optional comments to describe their concerns in detail. For improved communication, users may also provide contact information if they wish to follow-up. Feedback submissions are directly linked to the specific image analysis session, ensuring contextual understanding. The system automatically captures relevant metadata such as the timestamp, browser, and device details to assist in debugging.

Once submitted, feedback is stored in the system's database and associated with the corresponding user ID if logged in. The platform also supports anonymous feedback to encourage open communication. Submissions are classified into predefined categories such as bugs, suggestions, or compliments, with each entry tagged by priority and feature area. This enables efficient filtering and analysis by administrators.

On the admin side, the dashboard provides tools to review all feedback received. Admins can analyze trends, spot recurring issues, and respond to users where needed. Feedback metrics and charts help highlight areas needing improvement or user appreciation.

From a technical perspective, feedback handling is managed via REST APIs. When a user submits a form, the data is sent to a secure API endpoint, which logs the input in the backend. Proper responses and notifications are provided to confirm successful submission or to alert the user in case of failure.



Email Us
We'll respond within 24 hours
safestreet386@gmail.com

Call Us
Mon-Fri from 8am to 5pm
+91 7989841976

Visit Us
Admin office

Koheda Road
Ibrahimpatnam
Rangareddy, Telangana 501510

Send us a message

Full Name: Venkat
Email Address: venkatmadhu232@gmail.com

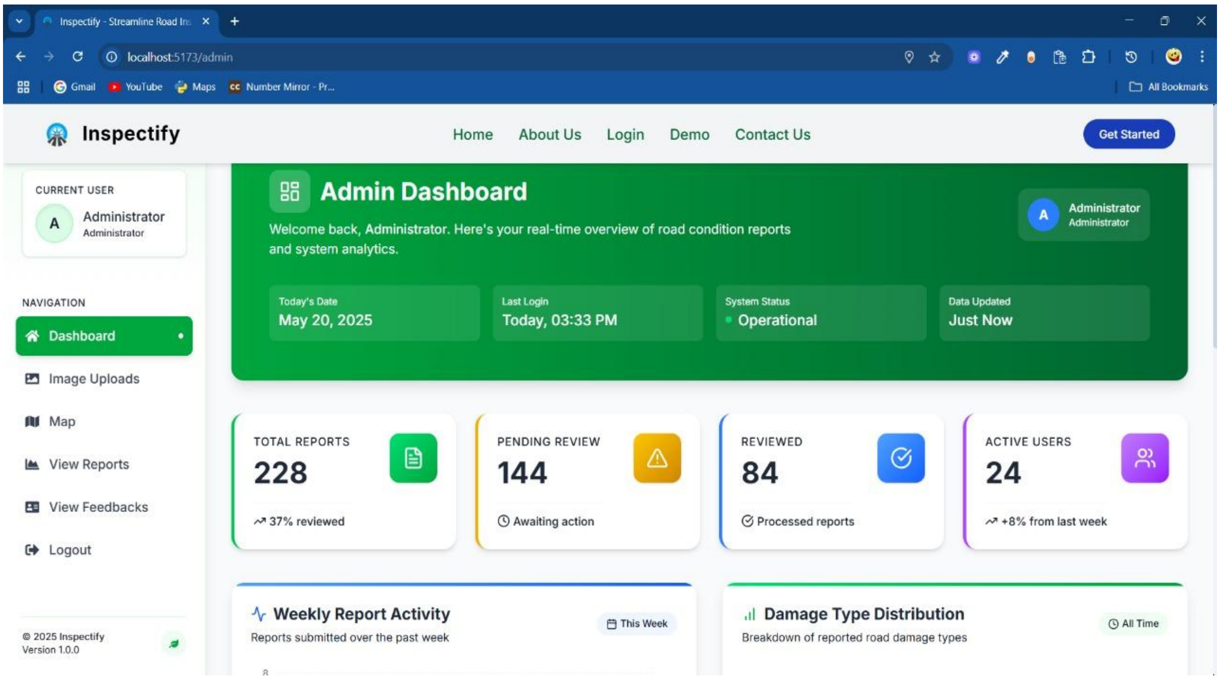
Subject: Regarding my experience about inspectify

Message: App is so useful good ui and good features

Figure 16. User Feedback

3.3.7. Admin Dashboard Functions

The Admin Dashboard serves as a centralized interface for administrators to review submitted road damage reports, and oversee the system's activity. It provides real-time access to AI-generated analysis, including severity, location, and visual damage indicators. Admins can filter reports, track trends, and respond to user feedback effectively. This enables better monitoring, streamlined operations, and informed decision-making for infrastructure planning.



Welcome back, Administrator. Here's your real-time overview of road condition reports and system analytics.

Today's Date: May 20, 2025

Last Login: Today, 03:33 PM

System Status: Operational

Data Updated: Just Now

TOTAL REPORTS 228 — 37% reviewed

PENDING REVIEW 144 — Awaiting action

REVIEWED 84 — Processed reports

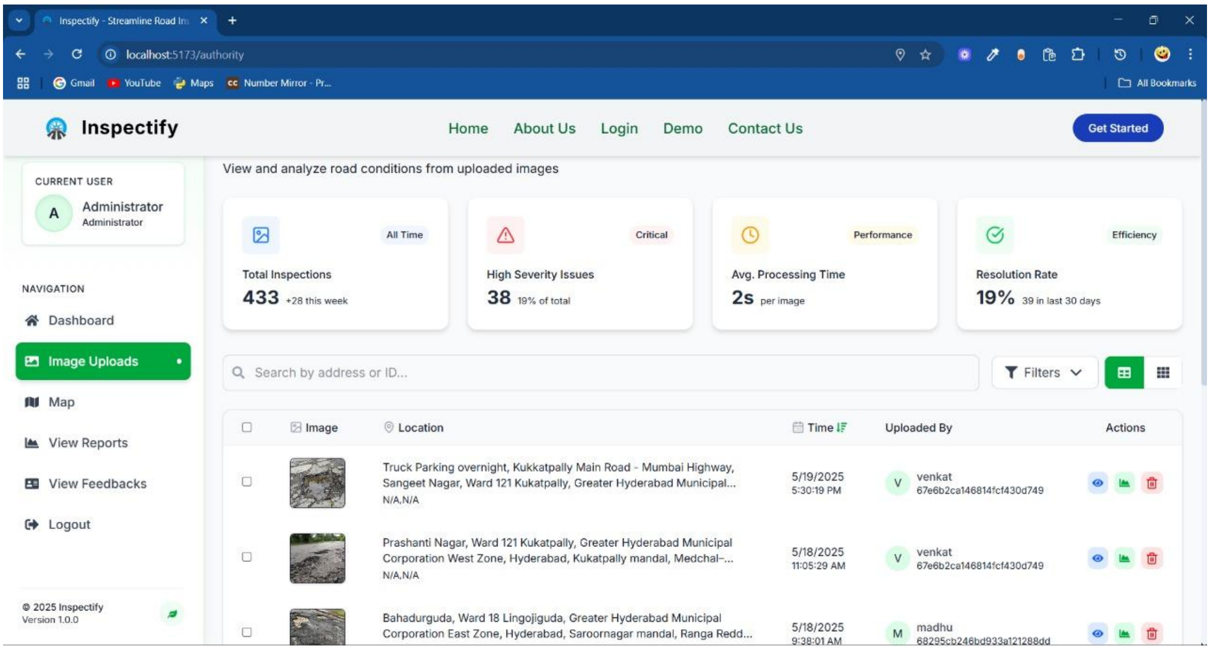
ACTIVE USERS 24 — +8% from last week

Weekly Report Activity — Reports submitted over the past week

Damage Type Distribution — Breakdown of reported road damage types

Figure 17. Admin Dashboard

Report management forms the core of the admin dashboard. All submitted damage reports are displayed in a structured table view, which can be filtered by damage type, severity level, and geographic location using integrated map filters. Each report includes an image, AI-detected bounding boxes, location data, severity score, and classification result. Administrators can click on a report to view full analysis details and update the report status. Furthermore, the system includes a feedback reply mechanism where admins can view user-submitted feedback linked to individual reports and respond to it if required.



View and analyze road conditions from uploaded images

Total Inspections
433 +28 this week

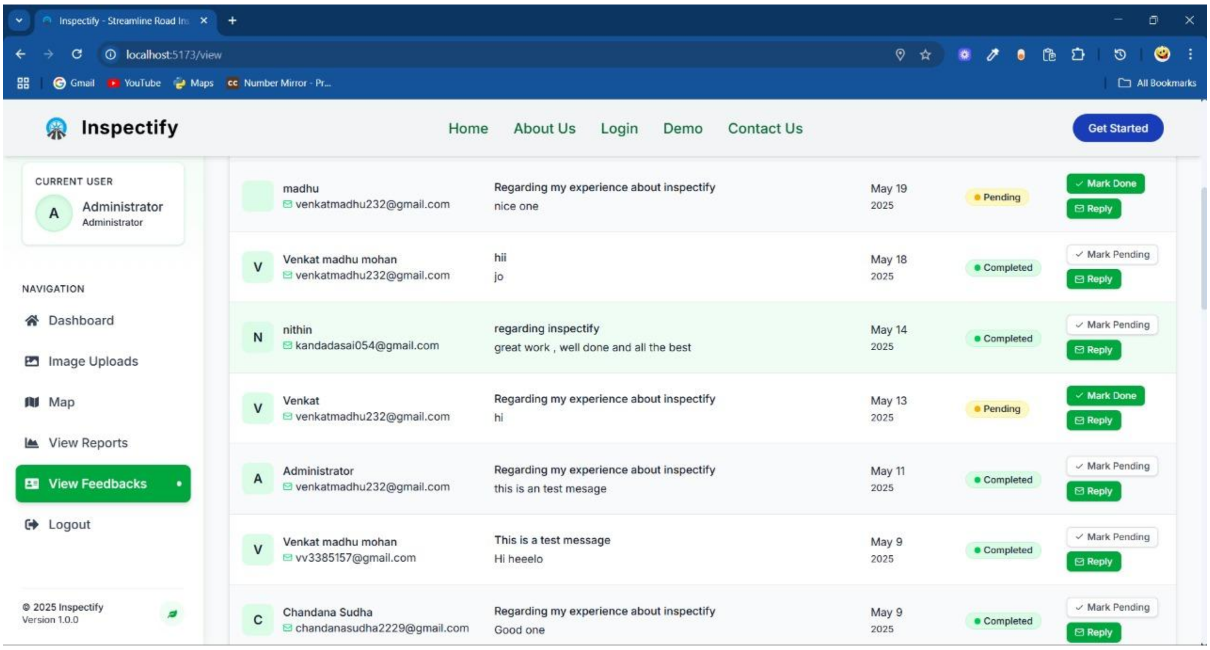
High Severity Issues
38 19% of total

Avg. Processing Time
2s per image

Resolution Rate
19% 39 in last 30 days

□	Image	Location	Time	Uploaded By	Actions
□		Truck Parking overnight, Kukkatpally Main Road - Mumbai Highway, Sangeet Nagar, Ward 121 Kukatpally, Greater Hyderabad Municipal... N/A,N/A	5/19/2025 5:30:19 PM	V venkat 67e6b2ca146814fcf430d749	
□		Prashanti Nagar, Ward 121 Kukatpally, Greater Hyderabad Municipal Corporation West Zone, Hyderabad, Kukatpally mandal, Medchal–... N/A,N/A	5/18/2025 11:05:29 AM	V venkat 67e6b2ca146814fcf430d749	
□		Bahadurguda, Ward 18 Lingojiguda, Greater Hyderabad Municipal Corporation East Zone, Hyderabad, Saroornagar mandal, Ranga Redd...	5/18/2025 9:38:01 AM	M madhu 68295cb246bd933a121288dd	

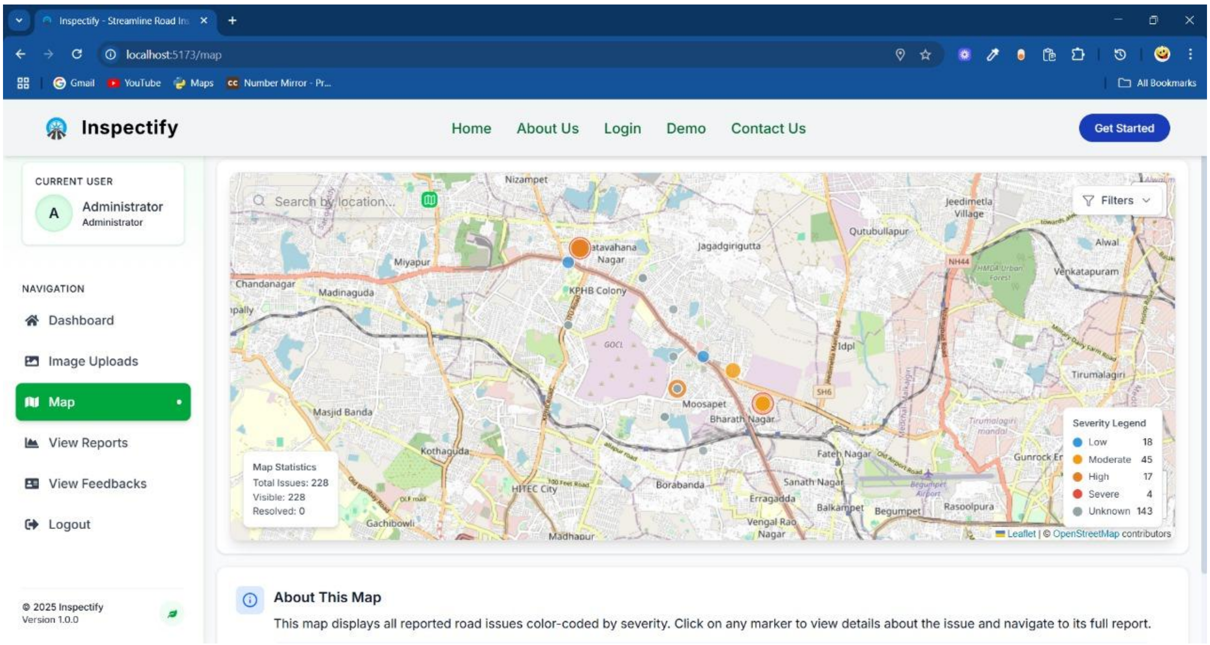
Figure 18. Submitted damage reports



	madhu venkatmadhu232@gmail.com	Regarding my experience about inspectify nice one	May 19 2025	Pending	✓ Mark Done Reply
V	Venkat madhu mohan venkatmadhu232@gmail.com	hii jo	May 18 2025	Completed	✓ Mark Pending Reply
N	nithin kandadasai054@gmail.com	regarding inspectify great work , well done and all the best	May 14 2025	Completed	✓ Mark Pending Reply
V	Venkat venkatmadhu232@gmail.com	Regarding my experience about inspectify hi	May 13 2025	Pending	✓ Mark Done Reply
A	Administrator venkatmadhu232@gmail.com	Regarding my experience about inspectify this is an test mesage	May 11 2025	Completed	✓ Mark Pending Reply
V	Venkat madhu mohan vv3385157@gmail.com	This is a test message Hi heeelo	May 9 2025	Completed	✓ Mark Pending Reply
C	Chandana Sudha chandanasudha2229@gmail.com	Regarding my experience about inspectify Good one	May 9 2025	Completed	✓ Mark Pending Reply

Figure 19. User feedbacks in Admin dashboard

The dashboard also features a map-based visualization of geo-tagged reports, where administrator can see the spread of road damage on an interactive map. The markers reflect the type and severity of each report, helping authorities prioritize repairs in high-risk areas. These geospatial filters allow admin users to analyze location-specific patterns.



Map Statistics
Total Issues: 228
Visible: 228
Resolved: 0

Severity Legend

●	Low	18
●	Moderate	45
●	High	17
●	Severe	4
●	Unknown	143

About This Map

This map displays all reported road issues color-coded by severity. Click on any marker to view details about the issue and navigate to its full report.

Figure 20. Map visualization in Admin dashboard

3.3.8 Data Storage and Database Integration

This application uses MongoDB as the primary database for storing user data, model predictions, location, and feedback submissions. Integration with MongoDB is handled using Mongoose in Node.js, which enables schema definitions, field validation, and easy querying. The data architecture is designed to support both object detection and classification modules in an organized manner.

Databases and Collections

We two main databases: Safestreet and road_damage_detection.

1. Safestreet Database: The Safestreet database contains the following collections:

I. feedback: Stores user feedback submitted through the frontend.

- name: User's name.
- email: Contact email.
- subject: Feedback subject or category.
- message: Detailed user message.
- dateSubmitted: Timestamp.

II. final_img: Holds predictions from the Vision Transformer (ViT) classifier.

- imagePath: Input image path.
- latitude, longitude: Location data.
- address: Physical location.
- analysisResult: Contains:
  - severity: Estimated damage severity.
  - vit_predictions: Encoded damage type (0–3).
  - image_dimensions: Includes height and width.
- timestamp: Time of classification.

III. login: Handles user authentication data.

- email: Registered user email.
- password: Encrypted user password.

IV. roadloc: Stores geolocation metadata of submitted images.

- imagePath: Image file path.
- latitude, longitude: GPS Coordinates.
- address: Location name or street address.
- timestamp: Date and time of submission.

2. road_damage_detection Database: This database is used for storing end-to-end prediction records in a unified format. The collection in this database is

detections: This single collection aggregates multiple aspects of detection.

- filename: Name of the image file.
- timestamp: Time of inference.
- severity: Contains:
  - level: Overall severity score.

    - count_score, area_score, type_score: Internal metrics.
- image_dimensions: Numerical indicator (0–3) representing image class.
- detections: List of detected objects with:
    - bboxes, class, conf, area, rel_area, color
- vit_predictions: Output from ViT classifier.
- location: Detected location string or coordinates.
- annotated_image: Path or reference to the image with drawn bounding boxes.

This modular schema design enables scalable storage of AI predictions, user submissions, and location-aware data. It supports both backend analytics and frontend display features efficiently.

3.3.9 REST API Communication Between Frontend & Backend

The system follows a structured REST API architecture for secure and efficient communication between the React.js frontend and the Node.js + Express.js backend. This architecture enables seamless data exchange for operations like user authentication, damage report submission, and analysis result retrieval.

API Design

The API follows RESTful principles, organizing endpoints around resources such as users, feedback, and road entries. All endpoints are prefixed with a versioned structure (e.g., /api/v1/) to support future scalability. Authentication is handled via JWT (JSON Web Tokens), allowing secure access to protected routes and enabling role-based access control for users and admins.

Request & Response Handling

- Requests support both application/json and multipart/form-data, allowing flexibility for image uploads and structured data.
- Middleware functions validate inputs, handle authentication, and enforce rate limits.
- Responses are returned in a consistent JSON format, with clearly defined HTTP status codes and informative error messages for debugging and UI feedback.

Frontend API Utilities

The frontend includes a reusable helper function apiRequest() in apiHelper.js, abstracting the fetch logic. This utility automatically attaches headers, parses responses, and handles errors consistently across the application.

Example: API request utility function

```
export const apiRequest = async (endpoint, options = {}) => {
  try {
    const headers = options.headers || {
      'Content-Type': 'application/json',
      'Accept': 'application/json'};
    const url = endpoint.startsWith('http') ? endpoint
      : `${BACKEND_URL}${endpoint.startsWith('/') ? endpoint : `/${endpoint}`}`;
    const response = await fetch(url, { ...options, headers });
    let data;
    const contentType = response.headers.get('content-type');
    if (contentType && contentType.includes('application/json')) {
      data = await response.json();
    }
    return { response, data, ok: response.ok };
  } catch (error) {
    return { error, ok: false, data: { error: error.message || 'Network error' } };
  }
};
```

All backend endpoints are centrally listed for easy access and modularity:

Authentication

/api/signup: User registration

/api/login: User authentication

/api/verify-auth: Token verification

/api/generate-otp & /api/verify-otp: OTP-based authentication

Road Damage Analysis

/predict: Initial image upload and processing

/analyze-damage: AI-powered damage detection and classification

/save-canvas: Save analysis results with bounding box visualization

Data Management

/api/road-entries: Retrieve road damage reports

/api/road-data: Get comprehensive road data (with filtering options)

/api/dashboard-stats: Get aggregated statistics for dashboards

Feedback System

/api/feedback: Submit user feedback

/api/feedbacks: Retrieve feedback submissions

/api/feedbacks/:id: Update feedback status

/api/feedbacks/:id/reply: Send replies to feedback

Notifications

/api/user-notification: Send notifications to specific users

Statistics and Reporting

/api/report-stats: Get reporting statistics

/api/road-stats: Get road condition statistics

/api/weekly-reports: Get weekly report data

/api/damage-distribution: Get damage type distribution

/api/severity-breakdown: Get severity breakdown statistics

/api/recent-reports: Get recent report data

Backend Implementation

The Express.js backend organizes logic using controllers, routers for each API group. This separation improves maintainability and reusability. Security features include, JWT validation, and IP-based rate limiting for sensitive operations like OTP generation.

3.3.10 Report Download

This system includes a module for report generation and download. This functionality is essential for users who wish to retain documented evidence of detected road damage or share it with relevant authorities. Reports are generated based on the damage analysis results and can be downloaded in multiple formats. The PDF reports are created using the jsPDF library, DOCX reports use the docx package, and CSV exports enable structured data analysis for administrative purposes.

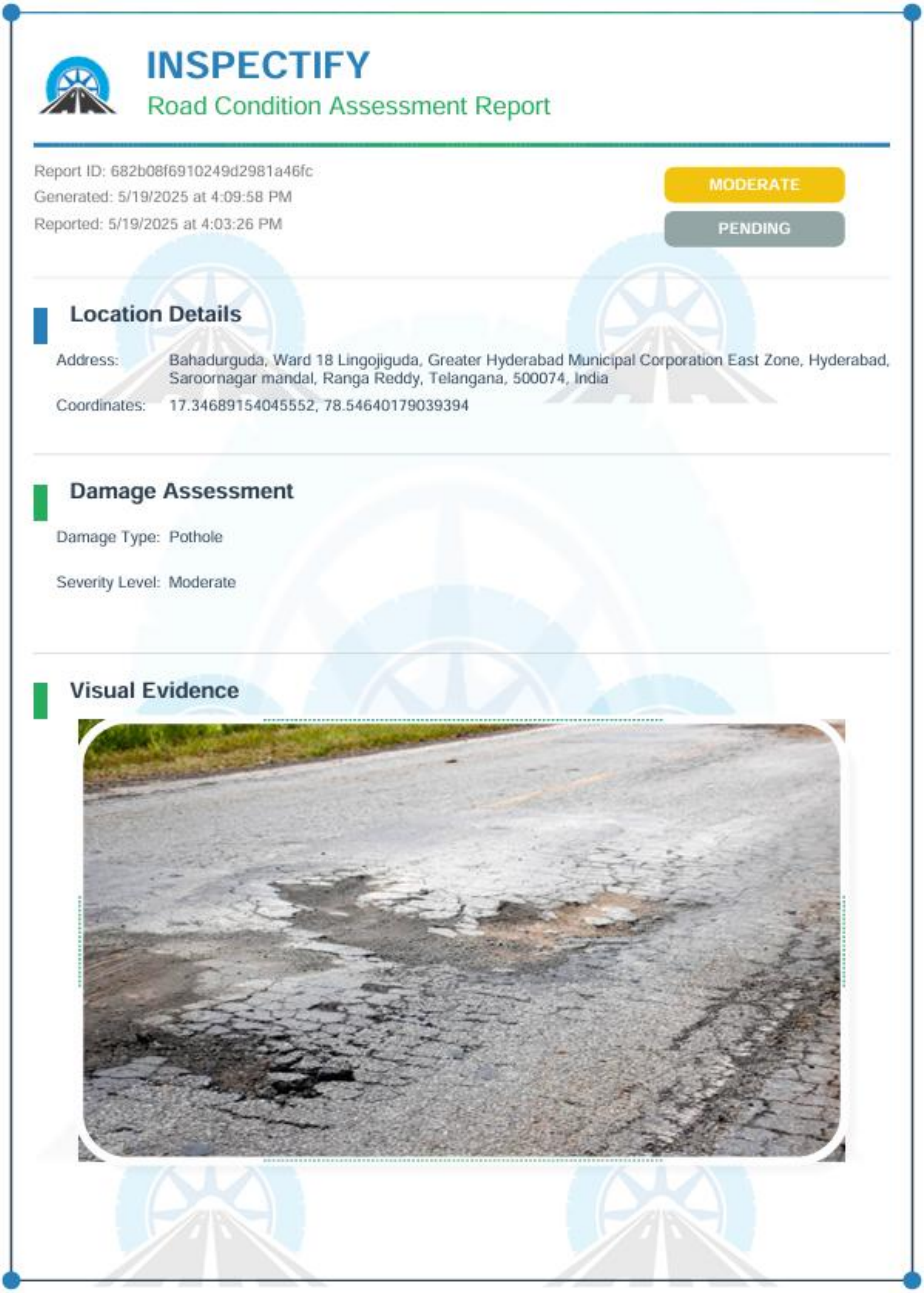
Each report contains essential details such as the type of damage detected, its severity level, the geographic location and the timestamp of the detection. Reports contain images of the damaged roads with bounding boxes to visually highlight the identified damage regions.These boxes are color-coded and labelled according to the classification results to ensure clarity.

From a user experience perspective, report generation is tightly integrated into the user interface. Users can access download options directly from their dashboard or result views, where buttons allow them to select their preferred file format. The system also supports a preview mode where users can review the contents before finalizing the download. Print-friendly formatting is applied to ensure that the downloaded reports are suitable for official use.

On the implementation side, the frontend handles report generation on the client device using standard JavaScript libraries. A sample report generation function uses jsPDF to insert

formatted text, metadata (such as report ID and date), and embedded images into the PDF. Detection results are listed with confidence scores and damage categories, presented in a readable structure.

This report download feature significantly improves the usability by offering transparent, portable, and shareable summaries of analysis results. Whether for internal records, audits, or communication with municipal stakeholders, these downloadable reports add substantial value to the road monitoring workflow.



INSPECTIFY

Road Condition Assessment Report

Report ID: 682b08f6910249d2981a46fc

Generated: 5/19/2025 at 4:09:58 PM

Reported: 5/19/2025 at 4:03:26 PM

MODERATE

PENDING

Location Details

Address: Bahadurguda, Ward 18 Lingojiguda, Greater Hyderabad Municipal Corporation East Zone, Hyderabad, Saroornagar mandal, Ranga Reddy, Telangana, 500074, India

Coordinates: 17.34689154045552, 78.54640179039394

Damage Assessment

Damage Type: Pothole

Severity Level: Moderate

Visual Evidence

Figure 21. Road Condition Assessment Report

3.3.11 Overall Workflow Integration (Frontend + Backend + AI Engine)

This application is integrated with workflow that connects the frontend user interface, backend server logic, and the AI engine responsible for road damage detection and classification. This integration ensures smooth and real-time communication between all components of the system, enabling a responsive and intuitive experience for users.

The frontend communicates with the backend using a RESTful API. All file uploads, such as road images, are handled using FormData, which includes image files along with geolocation metadata like latitude and longitude. Real-time updates, such as analysis completion or prediction status, are managed through WebSockets using Socket.IO, ensuring that users receive immediate feedback as soon as the AI processing is done. The backend, implemented using Express.js, manages the routes, processes the incoming requests, and forwards the image data to the AI engine written in Python.

The backend sends image paths and location data to the Python script, which performs preprocessing, model inference, and post-processing. The processed results, including bounding boxes, class labels, severity levels, and confidence scores, these are parsed into structured JSON format and sent back to the backend, which then forwards the response to the frontend for visualization.

In the data flow, users first authenticate and capture an image through the interface. Once submitted, the image and related data are uploaded to the server. The backend then routes this input to the appropriate AI model for processing. After inference, the results are stored in the MongoDB database and simultaneously sent to the frontend, where they are rendered using canvas overlays, labels, and severity indicators.

The application includes workflows, starting from user login, image capture, AI analysis, result visualization, to feedback submission. Progress indicators, and notifications help guide the user and maintain engagement during longer processing times. Behind the scenes, each analysis request undergoes several processing stages, including image preprocessing, model inference, result formatting, and database storage. To ensure reliability, the system includes error-handling mechanisms. Retry logic is in place for temporary failures, and user-friendly error messages ensure that users are well-informed. The backend also logs performance metrics and errors, enabling administrator to monitor system health and troubleshoot issues efficiently.

3.3.12 Deployment

This application is built with flexibility, allowing for both local development and production-level cloud deployment. The architecture separates the frontend, backend, and AI engine, enabling independent deployment and scaling of each component.

The frontend is developed using React with Vite, which allows for fast development builds Once the project is ready for production, Vite compiles and minifies the application, generating static files that can be served via any web server. During deployment, environment-specific variables (such as backend URLs) are managed using .env files to ensure configuration consistency between development and production environments.

The backend is built using Node.js with Express and serves as the logic and API layer of the application. It manages user authentication, report storage, and coordination with the AI engine. The backend also connects to a MongoDB database, which can be hosted locally during development or deployed to the cloud in production. Images and result files are stored either on the server file system or integrated with a cloud storage solution.

The AI engine, responsible for road damage analysis, runs in a Python environment with all necessary machine learning dependencies installed. The trained models are stored on the server, and the backend invokes them via Python scripts using Node's child process or PythonShell integration. The backend handles API communication and passes the image inputs to the AI model for prediction, returning the results to the frontend.

In a local development setup, all components run independently, the frontend runs on port 3000 with hot module reloading, the backend runs on port 5000, and MongoDB operates on the local system. Communication between the frontend and backend is managed via proxy settings defined in vite.config.js, and API routes are forwarded accordingly.

Sensitive data such as database credentials, API keys, and service URLs are managed securely using environment variables. These are defined in .env files and accessed in the codebase

without hardcoding sensitive information. Cross-Origin Resource Sharing (CORS) settings are configured on the backend to allow safe communication between the frontend and backend services deployed on different origins. This deployment approach makes the system scalable, portable, and secure, capable of operating efficiently across local and cloud infrastructures.

CHAPTER-4

Results & Discussions

4.1 Results

4.1.1 Comparative Analysis of YOLO Models

In the initial stage of project, multiple variants of Yolov8 models were explored for object detection:

- **YOLOv8s (Small)**: Optimized for speed, but showed reduced precision and recall in detecting fine-grained road damages like small cracks or partial potholes. Best suited for mobile and edge devices with limited computation.
- **YOLOv8m (Medium)**: Balanced between performance and resource usage. Showed moderate results in terms of precision and mAP but struggled slightly with mixed damage scenarios and overlapping classes.
- **YOLOv8l (Large)**: This model achieved the **best performance** in terms of localization accuracy and robustness. It was chosen for deployment due to its higher mAP@0.5 of 0.55, and more reliable detection of bounding boxes across different road damage types.

YOLO Model	Precision	Recall	mAP@0.5	Model Size	Advantages	Disadvantages
YOLOv8s	0.43	0.51	0.47	Small	Suitable for edge devices, low latency	Lower detection accuracy, misses small damages
YOLOv8m	0.48	0.55	0.50	Medium	Balanced performance and size	Inconsistent in detecting mixed or overlapping damages
YOLOv8l	0.52	0.58	0.55	Large	High accuracy, reliable bounding boxes, robust damage detection	Requires more computation, not ideal for low-resource deployment

Table 5. Comparative analysis of yolov8l with other models

YOLOv8l Model Performance

As a result, YOLOv8l was finalized due to its superior detection capability, high precision, and consistent performance in generating bounding boxes for diverse road damage types, particularly for potholes and alligator cracks. Among the variants tested, YOLOv8l achieved the best balance between detection and accuracy.

It effectively handled varied illumination, complex backgrounds, and overlapping damage types, which are common challenges in road damage imagery. The model's larger backbone enabled it to capture more semantic and contextual details, leading to more accurate object localization. Its performance was not only consistent across multiple image types but also stable across different validation runs.

Furthermore, YOLOv8l demonstrated efficient inference times on GPU-based systems while retaining high confidence scores, making it a practical choice for near real-time detection tasks. It is capable of processing large batches of high-resolution images efficiently, making it suitable for large-scale municipal datasets. These qualities make YOLOv8l a vital component in the Safe Street for prioritizing high-risk areas and supporting proactive road maintenance planning.

The performance of the YOLOv8l model was evaluated using standard object detection metrics provided by the Ultralytics framework. These metrics measure how well the model detects and localizes objects in images. The key results and interpretations are as follows:

Precision (0.52):

Indicates that just over 51% of predicted bounded boxes are correct. This suggests the model is cautious, it avoids too many false positives but may miss some true objects.

Recall (0.58):

The model was able to correctly detect around 58% of the actual damaged areas. While this is reasonable, there is scope for improvement, especially for hard-to-detect or small cracks.

mAP@0.50 (0.55):

This value shows the model has good detection capability. It can detect over half of the road damages correctly at a basic threshold of 0.50.

4.1.2 Comparative Analysis of ViT Models

In the classification module of the Safe Street project, various Vision Transformer (ViT) architectures were evaluated to determine the optimal model for accurately categorizing road damage into one of four predefined classes: Pothole, Longitudinal Crack, Lateral Crack, and Alligator Crack. The evaluation criteria included accuracy, precision, recall, F1-score, and inference efficiency.

- **ViT Tiny**: Being the lightest variant, it was tested first. However, it struggled with low-quality or noisy images, particularly under lighting conditions. The model often confused damage types with similar textures, such as longitudinal and lateral cracks, leading to reduced classification reliability.
- **ViT Small**: There was a noticeable improvement in classification accuracy and feature extraction. It outperformed the ViT Tiny in terms of both recall and F1-score. However, the performance gain was relatively marginal compared to the increase in computational requirements. This made it less ideal for real-time or low-latency applications, especially when integrated with other heavy processing components like object detection.
- **ViT Base**: This model delivered the highest classification accuracy and generalization across test sets. It captured finer spatial and contextual cues that helped differentiate between subtle damage patterns. Despite its robust performance, the model required significantly more memory and compute power, which posed challenges for integration into a system designed for real-time field use or deployment on limited infrastructure.
- **DeiT Tiny**: It offered an excellent balance between model efficiency and classification performance. DeiT Tiny achieved high macro-averaged F1-scores across all four damage classes and maintained consistent behaviour across varied image conditions. Its distilled training approach and lightweight architecture allowed it to maintaining fast inference speeds, making it highly practical for real-world applications. The model's strengths in both performance and efficiency made it the ideal choice for supporting Safe Street's goal of fast, accurate, and interpretable road damage classification.

Model	Accuracy	Precision	Recall	F1-Score	Speed	Advantages	Disadvantages
ViT-Tiny	0.76	0.74	0.70	0.72	Fast	Lightweight, low inference time	Struggles with fine-grained damage distinctions
ViT-Small	0.79	0.77	0.73	0.75	Moderate	Improved feature extraction and better class separation	Slight increase in resource usage
ViT-Base	0.81	0.80	0.78	0.79	Slow	Excellent classification accuracy, good generalization	High computational and memory demands
DeiT-Tiny	0.82	0.84	0.81	0.82	Fast + Distilled	Best performance for its size, efficient, good with fewer labels	Slight misclassification in edge cases

Table 6. Comparative analysis of DeiT with other models

Deit Model Performance

To assess the performance of the Data efficient image transformer macro-averaged classification metrics were used. The model was trained to classify road damage into four categories: Pothole, Lateral Crack, Longitudinal Crack, and Alligator Crack. The evaluation consists of macro metrics, class-wise insights, and confusion matrix analysis.

Accuracy (0.82): It indicates that the model correctly classified 82% of the total test samples

Precision (0.84): It shows that 84% of the predicted damage types.

Recall (0.81): The model was able to correctly identify 81% of the true damage classes.

F1-Score (0.82): Harmonic mean of the precision and recall. A strong score above 80% indicates reliable and balanced performance

These values suggest that the model performs quite well in a multi-class classification setting, maintaining high prediction confidence across varied damage types.

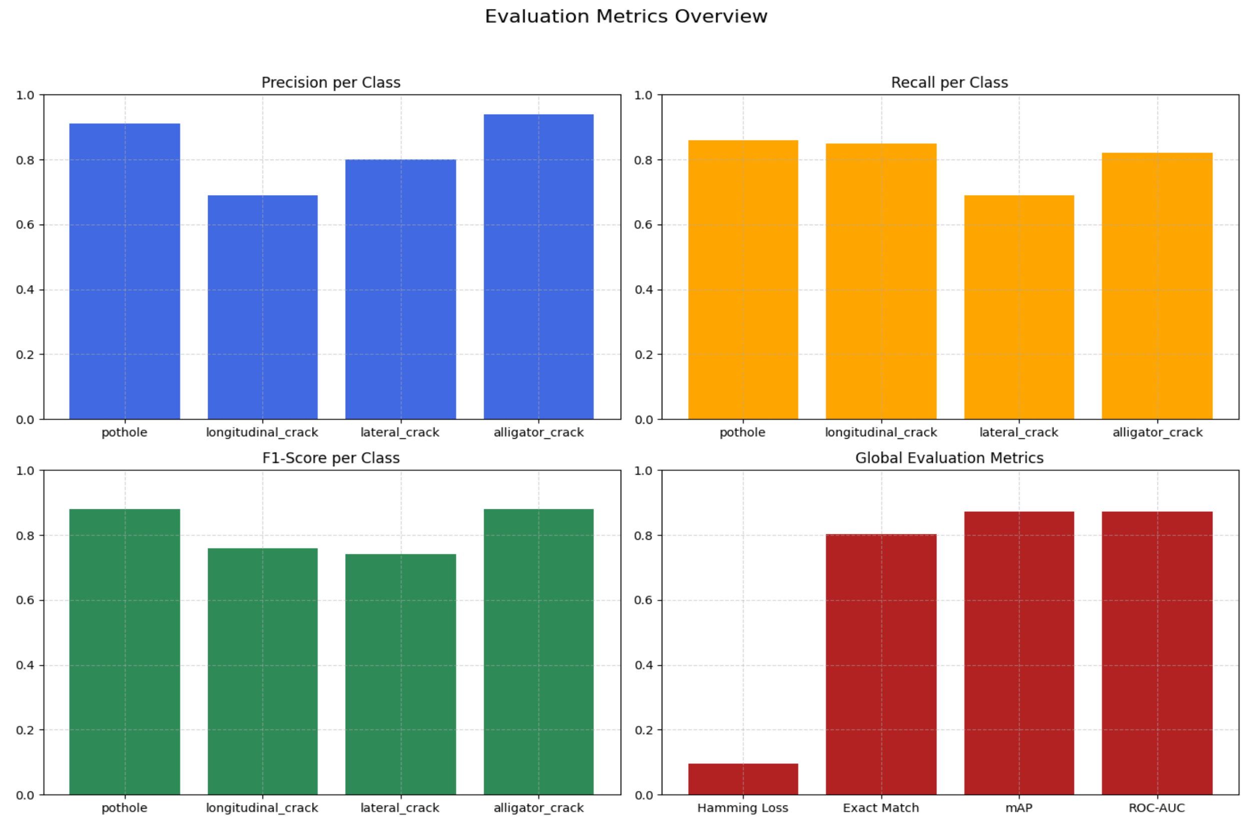


Figure 22. Evaluations metrics overview

DeiT Tiny's ability to generalize across crack types and lighting conditions made it the final choice for integration in SafeStreet. Its class-wise performance revealed that potholes and alligator cracks were classified with 88% F1-score, while longitudinal and lateral cracks showed slight overlap due to visual similarities.

One of the features of DeiT is its efficiency in learning from limited annotated data. Designed for data-efficient training, it performs well without needing large-scale datasets, making it ideal for road damage classification where labeled samples may be hard to obtain. The DeiT-Tiny specifically provides the best balance between speed and accuracy, allowing faster inference without compromising on model performance.

DeiT's transformer-based architecture brings the advantage of self-attention mechanisms, enabling it to capture long-range dependencies and fine-grained visual patterns that traditional CNNs may overlook. In addition, the model's design includes a distilled token, which mimics the output of a larger teacher network and enhances performance while keeping the architecture lightweight. The model's flexibility ensures that it can be further fine-tuned to classify new types of road anomalies in future upgrades, supporting the scalability and adaptability of the overall system.

Class-wise Performance Insights

Class	**Precision**	**Recall**	**F1-Score**	**Explanation**
Pothole	0.89	0.87	0.88	Excellent balance between precision and recall.
Longitudinal Crack	0.67	0.87	0.76	Hight recall but low precision over predicting this class.
Lateral Crack	0.84	0.68	0.75	Good precision but missing many true positives.
Alligator Crack	0.96	0.81	0.88	Highest confidence, least ambiguity in predictions.

Table 7. Class wise evaluation metric values of Deit Model

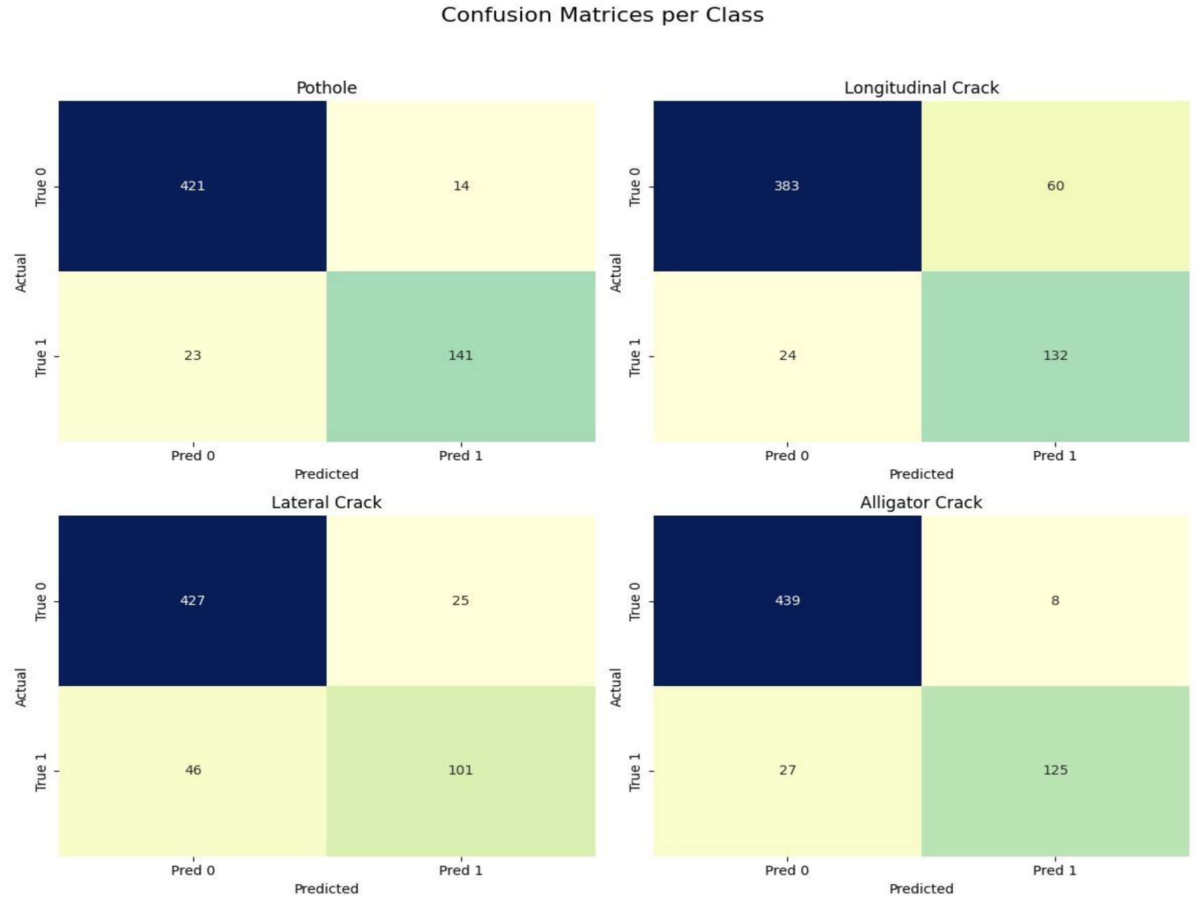


Figure 23. Confusion matrix of DeiT Model

4.2 Discussions

The Safe Street project tackles a critical infrastructure issue by automating the traditionally manual process of road inspection through AI-based object detection and classification. By using a combination of YOLOv8l for detecting damage locations and severity and DeiT-Tiny for classifying those damages, the system achieves a strong balance of spatial accuracy and meaningful categorization.

The object detection module successfully identifies visual damages such as potholes, longitudinal cracks, lateral cracks, and alligator cracks. It draws bounding boxes on the input image using color-coded labels for each damage type. Although the detection accuracy is moderate under stricter evaluation metrics, real-world usage has shown reliable performance, particularly for clearly visible and major damages like potholes and large cracks.

By assessing the severity of each detected damage, it helps to determine how urgent a repair is and are used to assign priority levels for each case. These results are then visualized and summarized in the frontend, allowing municipal workers or administrators to act accordingly. From a system architecture perspective, the frontend interface displays results with real-time notifications and user feedback options, while the backend handles model inference, stores images, and manages user inputs. The use of Socket.IO ensures live updates and smooth user interactions. MongoDB manages all data, including images, predictions, user information, and feedback, ensuring the system is scalable and responsive.

One important observation from the evaluation is that some types of road cracks (like lateral and longitudinal) are occasionally confused due to their visual similarity, especially when the image is taken at an angle. However, damages like potholes and alligator cracks are consistently detected with high confidence and precision.

The admin dashboard presents the results in a well-organized format. It also supports map-based visualization, where detected damage locations are marked using their GPS coordinates. This helps admins monitor damage geographically and observe trends across different areas.

Overall, the results and evaluation metrics demonstrate that Safe Street is an effective and practical solution for real-time road damage monitoring. The system is modular and can be extended in the future with features like real-time camera integration, Feedback-driven retraining etc.

CHAPTER-5

CONCLUSION & FUTURE SCOPE

5.1. Conclusion

The Safe Street project has effectively demonstrated the power of deep learning and computer vision in solving real-world challenges, specifically in the domain of automated road damage detection and reporting. Through the seamless integration of object detection (YOLOv8l) and classification (DeiT-Tiny) models, the system is capable of accurately detecting and categorizing various types of road surface damages, including potholes, longitudinal cracks, lateral cracks, and alligator cracks.

The system's ability to analyze images, assess severity, and present results with visual overlays and structured summaries empowers municipal bodies and infrastructure agencies to make informed, data-driven decisions regarding road maintenance. By eliminating the need for traditional manual inspections, which are often time-consuming and expensive. Safe Street transforms road monitoring into a proactive, scalable, and technology-driven solution.

Additionally, the project highlights a modern, full-stack implementation approach using the MERN stack integrated with AI-powered Python scripts. This architecture not only ensures modularity and scalability but also offers real-time feedback, interactive dashboards, user feedback mechanisms, and geo-tagged mapping features for comprehensive infrastructure management.

In real-world deployments and testing, the models have shown robust performance, particularly in detecting clear and severe damage cases. The admin dashboard, report generation, and user interfaces further improve usability, enabling stakeholders from varied technical backgrounds to engage with the system effectively.

5.2. Future Scope

While the current system provides a strong foundation, several enhancements can be made in future versions:

- **Real-time Camera Integration**: Incorporating live camera feeds from vehicles or surveillance systems to enable real-time damage detection on the move.

- **Severity Scoring Algorithm**: Refining severity assessment by using geometric and contextual cues like damage depth and frequency of occurrence.
- **Larger Dataset & Data Augmentation**: Expanding the dataset with more diverse images from different regions and applying data augmentation for better generalization.
- **Multilingual Support in Summary Generation**: Generating summaries in regional languages for better accessibility by local authorities.
- **Feedback-Driven Retraining**: Allowing users or authorities to correct predictions, which can then be fed back into the model to improve accuracy over time.
- **Voice-Enabled Assistance**: Implementing voice-based interaction for field officers, allowing them to interact with the dashboard hands-free during inspections or while driving.
- **Integration with Government APIs**: Connecting Safe Street with municipal APIs or Smart City portals to automate complaint logging, maintenance tracking, and budget allocation workflows.
- **Crowdsourced Dataset Expansion for Retraining**:
  A feature where user-submitted images via the app are stored and optionally used to retrain the ViT or YOLO models. This will enable the system to continuously improve by learning from real-world, diverse, and evolving road conditions, ensuring long-term adaptability and robustness.

CHAPTER-6

REFERENCES

[1] M. G. Rizal, “Road Damage Detection and Classification Using Deep Learning Techniques,” Bulletin of Electrical Engineering and Informatics (BEEI). Available on https://www.beei.org/index.php/EEI/article/view/2348

[2] S. Y. Sari et al., “Detection and Classification of Road Damage Using Camera with GLCM Features,” Universitas Hasanuddin Research Portal. Available on https://scholar.unhas.ac.id/en/publications/detection-and-classification-of-road-damage-using-camera-with-glc

[3] S. R. Sinha et al., “Road Damage Detection using YOLO and Deep Learning,” SN Applied Sciences, Springer, 2024.Available: https://link.springer.com/article/10.1007/s42452-024-06129-0

[4] “Enhance Road Quality with AI Road Damage Detection System,” TMA Solutions, Available on https://www.tmasolutions.com/case-studies/enhance-road-quality-with-ai-road-damage-detection-system

[5] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, Herve Jegou, et al., “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale,” arXiv preprint arXiv:2012.12877, 2020. Available: https://arxiv.org/abs/2012.12877

[6] Facebook AI Research, “DeiT: Data-efficient Image Transformers,” GitHub Repository, Available: https://github.com/facebookresearch/deit

[7] EmailJS, "EmailJS - Send emails directly from your JavaScript code,". Available: https://www.emailjs.com/

[8] Nodemailer, "Nodemailer- Send e-mails with Node.js,". Available: https://nodemailer.com/

[9] LeafletJS, "Leaflet - an open-source JavaScript library for interactive maps,". Available: https://leafletjs.com/

[10] Framer Motion, "Framer Motion- A production-ready motion library for React,". Available: https://www.npmjs.com/package/framer-motion