CHAPTER 1

INTRODUCTION

1.1 Project Summary

1.2 Purpose & Goals

1.3 Technology

1.1 Project Summary

- This project put together a real-time object detection app. It used a custom-trained YOLOv11n model along with a simple Tkinter interface that anyone could use. The whole development thing focused on this huge dataset. Over 200,000 images got curated and handled through Roboflow. Training the model took a lot of computing power. Still, it all worked out on Kaggle. They used a solid NVIDIA P100 GPU there to deal with the big data without issues.

- In the end, you get a system that ties in the custom model for spotting objects accurately. It pulls from a live webcam feed. The user side feels smooth, especially with detection happening right then. The project went through every step of the machine learning process. From grabbing the data at the start to ending up with a working detection setup.

1.2 Purpose & Goals

- To design and build a complete, end-to-end real-time object detection system.
- To focus on using a model trained on a large, custom dataset instead of a general, pre-trained one.
- To develop a user-friendly GUI with Python and Tkinter that shows a live webcam feed and overlays detection results.
- To gather, label, and prepare a large custom dataset to train a specialized object detector.
- To train a YOLOv11 model on the custom dataset and include the final model in the application.
- To review the model's performance, emphasizing its accuracy and real-time speed (FPS)

1.3 About Technology

Python:

- It is the core programming language for the entire project.
- Python was chosen for its simplicity, wide range of powerful libraries for machine learning and computer vision, and strong community support.
- All the application logic, including the user interface and model inference, was written in Python.

Tkinter:

- Tkinter was used to create the user-facing desktop application. This included the initial start screen, the main window that shows the live webcam feed, and integrating elements like buttons. As Python's standard GUI library.
- It is lightweight and does not require extra installation, making it easier to package and run the application.

OpenCV (opencv-python):

- OpenCV was essential for managing all real-time video operations. Its specific tasks included capturing the video stream from the webcam.
- It is used for reading individual frames, converting colors for display in Tkinter, and drawing the final bounding boxes and labels onto the frames before displaying them to the user.

PyTorch:

- PyTorch serves as the engine for the YOLO model. While you interact with the model through the ultralytics library.
- PyTorch operates in the background to perform complex mathematical tasks on the GPU or CPU needed for the model to make predictions.
- The underlying deep learning framework.

Ultralytics YOLO:

- This library offers a simple, user-friendly interface for the YOLOv10 architecture. It made two key processes easier:
- It provided the framework to train the model on a custom dataset with a single command.
- It allowed for easy loading (Inference)of the trained (best.pt) model and making predictions on new images or video frames.

Roboflow:

- Managing a dataset of 211,000 images is a huge task.
- Roboflow was important for streamlining this workflow.
- It was used to upload, organize, and annotate the images.
- Most importantly, it handled essential preprocessing steps and applying data augmentations to improve model performance. Finally, it exported the dataset in the exact format needed for YOLO training and created the necessary data.yaml file automatically.

Kaggle Notebooks & NVIDIA P100 GPU:

- Why it was used: Training a deep learning model on over 200,000 images is impossible on a standard laptop.
- Kaggle was chosen to provide free access to a high-performance NVIDIA Tesla P100 GPU.
- This data-center-grade hardware significantly reduced the training time from potentially weeks to just a couple of days, making the project possible.

CHAPTER 2

PROJECT PROFILE

2.1 Project Planning and Scheduling

2.2 Risk Management

2.3 Schedule Representation

2.1 Project Planning and Scheduling

2.1.1 Project Development Approach

The project took a step-by-step approach to development. It started by building a basic application using Python and Tkinter, which initially included a generic YOLO model as a placeholder. The next phase was about data engineering. A large custom dataset of over 200,000 images was created, preprocessed, and improved using the Roboflow platform. To manage the demanding model training, the project switched to a cloud-based method. It used a powerful NVIDIA P100 GPU on the Kaggle platform to train the YOLOv11n model efficiently, implementing an early stopping mechanism. In the final phase, the custom-trained best.pt model was integrated back into the desktop application. The project concluded with a detailed performance evaluation to assess its accuracy and real-time speed.

2.1.2 Project Planning

The project was planned as a sequential, four-phase process covering the complete development lifecycle: The initial phase focused on building the foundational application framework with a Tkinter GUI.
This was followed by a dedicated data engineering phase, where the large-scale custom dataset was curated and preprocessed using Roboflow. The third phase involved training the YOLOv10n model in a cloud environment on Kaggle to leverage its powerful GPU capabilities. The final phase covered the integration of this custom-trained model into the application, followed by a comprehensive performance evaluation.

2.2 Risk Management

2.2.1 Risk Identification

- Computer Power
- Model "Memorizing" Instead of Learning
- Messy Data
- Slow and Laggy Application
- Poor Real-Time Performance

2.2.2 Risk Analysis:

Computational Constraints: The main risk was that local hardware could not handle training a model on over 200,000 images.

Model Overfitting: A model trained too long on a large dataset could "memorize" the training data and not perform well on new images.

Data Inconsistency: Managing such a large dataset could lead to inconsistencies.

Poor Real-Time Performance: The chosen model could have been too slow for a live webcam feed on standard hardware.

Poor Data Quality: There was a risk that initial annotations (bounding boxes) could be inaccurate or inconsistent, leading to the "garbage in, garbage out" issue.

2.2.3 Risk Planning:

Here is how we deal with all the above said risks:

Computational Constraints:

- This was solved by using the Kaggle platform to access a free, high-performance NVIDIA P100 GPU, which made training possible.

Model Overfitting:

- This was handled by using an "early stopping" mechanism during training, which stopped the process once the validation set's performance stopped improving.

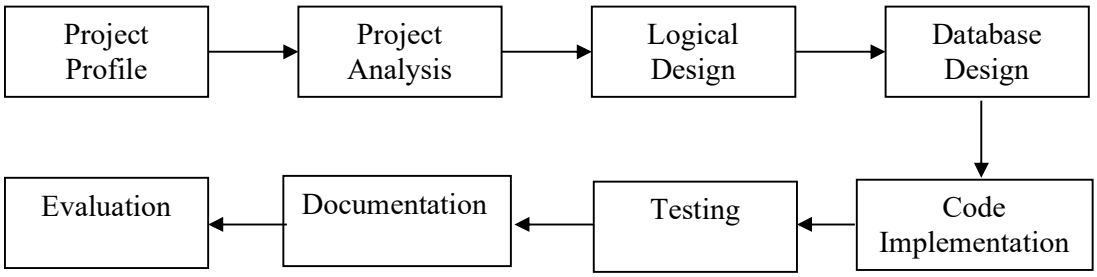
Data Inconsistency:

- This risk was reduced by using the Roboflow platform to enforce a systematic workflow for data preprocessing, including uniform image resizing and consistent application of augmentations.

Poor Real-Time Performance:

- The chosen model could have been too slow for a live webcam feed on standard hardware. To fix this, the YOLOv11n ("nano") variant was selected since it is the smallest and fastest version of the model, optimized for real-time inference on devices with limited computational power.

2.3 Schedule Representation



Phase 1: Foundational Development (Weeks 1-2)

- Develop the core Tkinter GUI application.
- Integrate the live webcam feed using OpenCV.
- Implement a generic, pre-trained YOLO model as a placeholder.

Phase 2: Data Engineering (Weeks 3-4)

- Curate and annotate the 211,000-image custom dataset.
- Use Roboflow for systematic preprocessing (resizing) and data augmentation.

Phase 3: Model Training (Weeks 5-6)

- Set up the training environment on Kaggle with a P100 GPU.
- Train the YOLOv10n model on the custom dataset.
- Monitor for early stopping to achieve optimal performance.

Phase 4: Integration & Evaluation (Weeks 7-8)

- Integrate the custom-trained best.pt model into the Tkinter application.
- Conduct a performance evaluation, measuring accuracy and real-time FPS.

Phase 5: Refinement & Finalization (Weeks 9-10)

- Benchmark the custom model against newer architectures like YOLOv11.
- Refine the code, add comments, and prepare the final project report

Description	Estimated days	Actual Days
Foundational Development (GUI)	**10**	**8**
Data Engineering	**14**	**15**
Model Training	**16**	**18**
Integration & Evaluation	**15**	**16**
Refinement & Finalization (Reporting)	**15**	**13**

CHAPTER 3

SYSTEM REQUIREMENT SPECIFICATION

3.1 User Characteristics

3.2 Software and Hardware Requirement

3.3 Constraint

3.1 User Characteristics

There may be two types of users as following:

- Primary User Role: Rather than being a general user, the target user is probably an operator or monitor specific to the domain.
- Technical Proficiency: A basic understanding of computers is required of the user. Although they are not expected to know any programming or command-line skills, they should be able to run a typical desktop application (launching the program, interacting with a basic GUI).

3.2 Hardware and Software Requirements

This website requires following:

1. Software Requirement

Tools/Technology	**Front-end**	-	Python's Tkinter library (for the GUI)
	Middle Layer	-	Python (application logic) OpenCV(webcam and image processing)
	Back-end	-	PyTorch(deep learning) Ultralytics YOLO (model implementation)
	Development Platforms	-	Roboflow (dataset manage) Kaggle Notebook (model training)
	Operation System	-	Windows, macOS, or a Linux distribution.

2.Hardware Requirement

Hardware
➢ multi-core processor (Intel Core i5, AMD Ryzen 5, or better) is recommended. ➢ RAM : A minimum of 8 GB of RAM. ➢ Graphics Card (GPU): A dedicated NVIDIA GPU (GTX 1650 or better) is highly recommended. ➢ Webcam: A standard internal or external webcam is required.

3.3 Constraint

- Heavily dependent on a powerful GPU for achieving smooth, real-time frame rates.
- Inherent trade-off between model speed and accuracy (e.g., smaller 'nano' models are fast but less accurate).
- Performance is constrained by environmental factors like poor lighting, shadows, and camera quality.
- Detection can be unreliable for objects that are partially hidden or very small.

CHAPTER 4

ANALYSIS AND DESIGN

4.1 System Requirements

4.2 Feasibility Analysis

4.3 Requirement Validation

4.4 Requirement Gathering

4.5 Process Model

4.6 System Design

4.1 Requirements

4.1.1 Functional Requirements (Implementation of User Requirement)

- The system must capture a live video stream from a connected webcam.
- The system must display the live video feed to the user in real-time.
- The system must process video frames to detect predefined objects using a YOLO model.
- The application must overlay bounding boxes and class labels on the detected objects.
- The system must be able to load and use a custom-trained model.

4.1.2 Non-Functional Requirements

- The finished source code needs to be clear, well-commented, and simple enough for another developer to comprehend.
- The finished project report needs to be easy for the target audience to understand, clear, and well-structured.
- The reported performance metrics (accuracy, FPS) need to be precise and repeatable.
- To guarantee a fair comparison between various models, the benchmarking procedure needs to be carried out in a consistent setting.

4.2 Feasibility Analysis

For the final phase of my project, "Refinement & Finalization," I conducted a feasibility analysis to ensure all the concluding tasks were achievable.

I evaluated this phase based on its technical, operational, and schedule viability to confirm I could complete the benchmarking, code cleanup, and final reporting within the project's constraints. This analysis affirmed that the phase was highly feasible, a conclusion supported by the fact that I completed the work ahead of the planned schedule.

Technical Feasibility: -

- To ensure technical viability, standard, well-documented technologies like Python, Tkinter, and OpenCV were used in the project's construction.
- The Kaggle platform's high-performance GPU enabled the solution of the biggest technical problem: training a model on a large dataset.
- The Roboflow platform made it technically possible by offering the tools required to methodically handle the data engineering phase.
- By offering a high-level API for the YOLO architecture, the ultralytics library made the process of training and integrating the model easier.

Economical Feasibility: -

- Developed using exclusively free and open-source software (Python, Tkinter, etc.).
- Avoided hardware costs by using Kaggle's free cloud GPU for model training.
- Eliminated software expenses by using Roboflow's free tier for data management.
- The project's financial cost was minimal, with the main investment being developer time.

Operational Feasibility: -

- The application is easy for the end-user to operate due to the simple and intuitive Tkinter GUI.
- The project was successfully developed, demonstrating that the required technical skills (Python, ML, CV) were available.
- As a standalone desktop application, it can be easily deployed and integrated into a user's existing workflow.
- The use of well-documented, standard technologies ensures the system is maintainable for future updates.
- The established data and training pipeline (Roboflow, Kaggle) makes retraining the model with new data operationally straightforward.

Schedule Feasibility: -

- The project was highly feasible from a scheduling perspective, as the entire project was completed within the total estimated timeframe of 10 weeks.
- Initial phases, like GUI Development, were completed ahead of schedule, demonstrating effective early planning.
- Minor delays in phases like Data Engineering and Integration were successfully absorbed and balanced out by efficiencies in other stages.

4.3 Requirement Validation

- Live video display was validated with the functional Tkinter and OpenCV GUI.
- Custom object detection was validated by testing the final model.
- Real-time performance was validated by measuring the system's Frames Per Second (FPS).
- Usability was validated by designing a simple and intuitive interface for non-technical users.

4.4 Requirement Gathering

The requirements for the project were gathered and categorized into two main types: Functional and Non-Functional.

Functional Requirements :

- **Live Video Capture:** The system must be able to access and display a live video stream from a connected webcam.
- **Object Detection:** The application must process the video feed to identify and locate specific, custom-defined objects within the frames.
- **Visual Feedback:** The system must draw bounding boxes and class labels on the screen, overlaying them on the detected objects in the video feed.
- **Custom Model Integration:** The application must be able to load and use a custom-trained YOLO model file (best.pt).
- **Graphical User Interface (GUI):** The system must have a simple desktop interface for users to interact with the application.

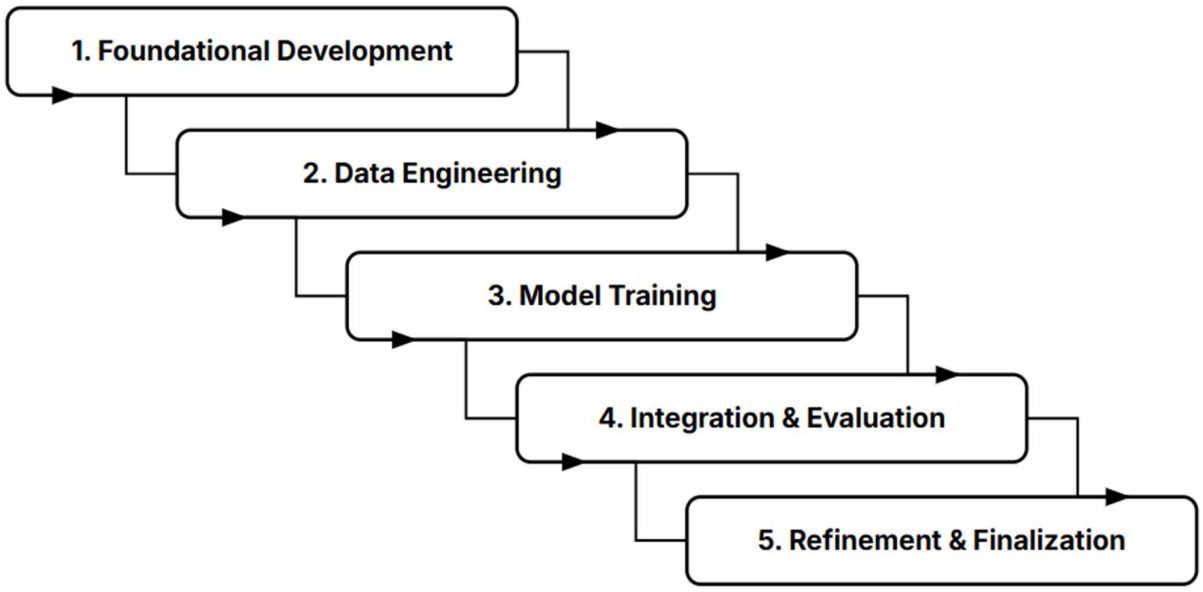
Non-Functional Requirements :

- **Performance:** The detection process must be fast enough to be considered "real-time," with a smooth and high frame rate (FPS).
- **Usability:** The user interface must be intuitive and easy for a non-technical person to operate.
- **Accuracy:** The model's detections should be reliable and accurate for the specified objects.
- **Maintainability:** The source code must be well-commented and structured to allow for future updates and modifications.
- **Cost-Effectiveness:** The project must be developed using free and open-source technologies to minimize any financial cost.

4.5 Process Model

The Process model is the sequence of operation which defines flow of the work in the system.

The process model includes following stages:



1. Foundational Development

- Developing the GUI with Python's Tkinter library.
- Using OpenCV to capture the live webcam feed and display it in the application window.
- Integrating a generic, pre-trained YOLO model as a placeholder to ensure the video processing and bounding box overlay functions were working correctly.

2. Data Engineering

- Gathering, annotating, and organizing the massive dataset of over 200,000 images.
- Using the Roboflow platform to systematically preprocess the data, including resizing all images to a consistent 416x416 resolution and applying data augmentations to improve the model's ability to generalize.

3. Model Training

- Setting up a training environment on the Kaggle platform to use a high-performance NVIDIA P100 GPU.
- Running the training process, where the YOLOv10n model learned to detect objects from your custom dataset.

4. Integration & Evaluation

- Replacing the generic placeholder model in the Tkinter application with your custom-trained best.pt file.
- Conducting a thorough evaluation of the system to measure its real-world performance, specifically its detection accuracy and its speed in Frames Per Second (FPS).

5. Refinement & Finalization

- Benchmarking the custom model's performance against other architectures (like YOLOv11) to understand its effectiveness.
- Refactoring the code by adding detailed comments and ensuring it was clean and maintainable.
- Writing the final project report, summarizing the entire process, the results, and the conclusions.

4.6 System Design

The system is designed as a standalone system with a clear, modular architecture that separates the user interface, the application logic, and the machine learning model.

Architectural Design: The application follows a three-tier architectural pattern:

- **Presentation Layer (Front-End):** This is the user-facing component, built with Python's Tkinter library. Its sole responsibility is to display the graphical user interface (GUI), render the video feed, and present the detection results (bounding boxes and labels) to the user.

- **Application Logic Layer (Middleware):** This layer, written in Python, acts as the bridge between the front-end and the back-end. It uses the OpenCV library to handle all video processing tasks, such as capturing frames from the webcam, converting image formats, and preparing the data for the model. It also contains the main control loop for the application.

- **Model Layer (Back-End):** This is the core machine learning component. It consists of the custom-trained YOLOv11 model (best.pt) running on the PyTorch framework. This layer's only job is to receive an image frame from the application logic layer and return the coordinates and class predictions for any detected objects.

Data Design (Data Flow): The data flow is linear and designed for a one-way process from data collection to inference:

- **Input Data:** The process begins with a massive collection of raw images (211,000), which serve as the foundation for the custom model.

- **Data Preprocessing:** The raw images are processed through Roboflow, where they are annotated, resized to a uniform 416x416, and augmented. This structured data is then exported in the YOLO format.

- **Training Data:** The processed data is used in a Kaggle environment to train the YOLOv10n model.

- **Output Data:** The model outputs a set of coordinates for bounding boxes, along with class labels and confidence scores for each detected object in the frame.

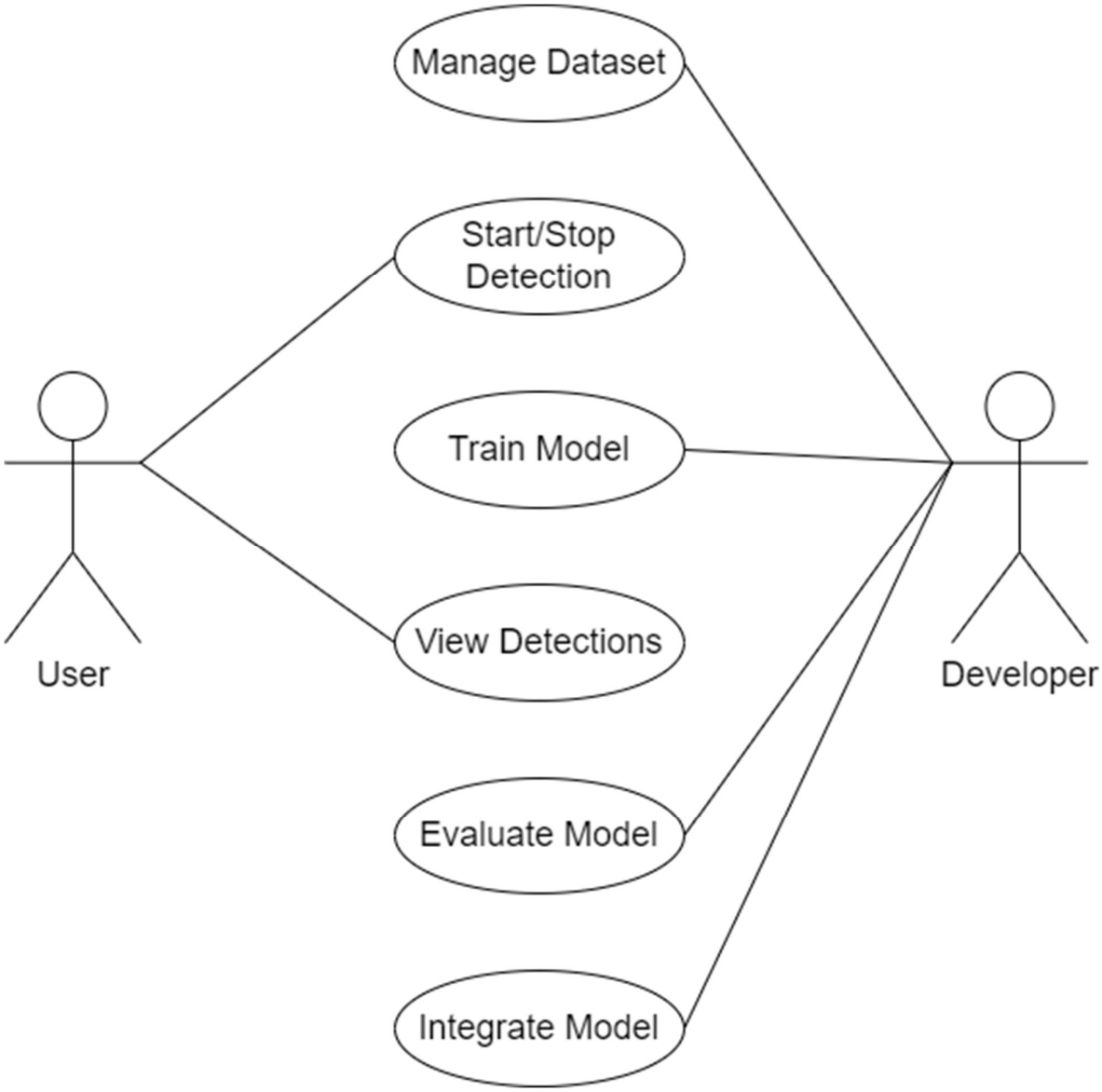
Interface Design: The user interface is designed for simplicity and ease of use, requiring no technical expertise from the end-user.

- **Start Screen:** The application launches with a simple start screen containing a button to begin the detection process.

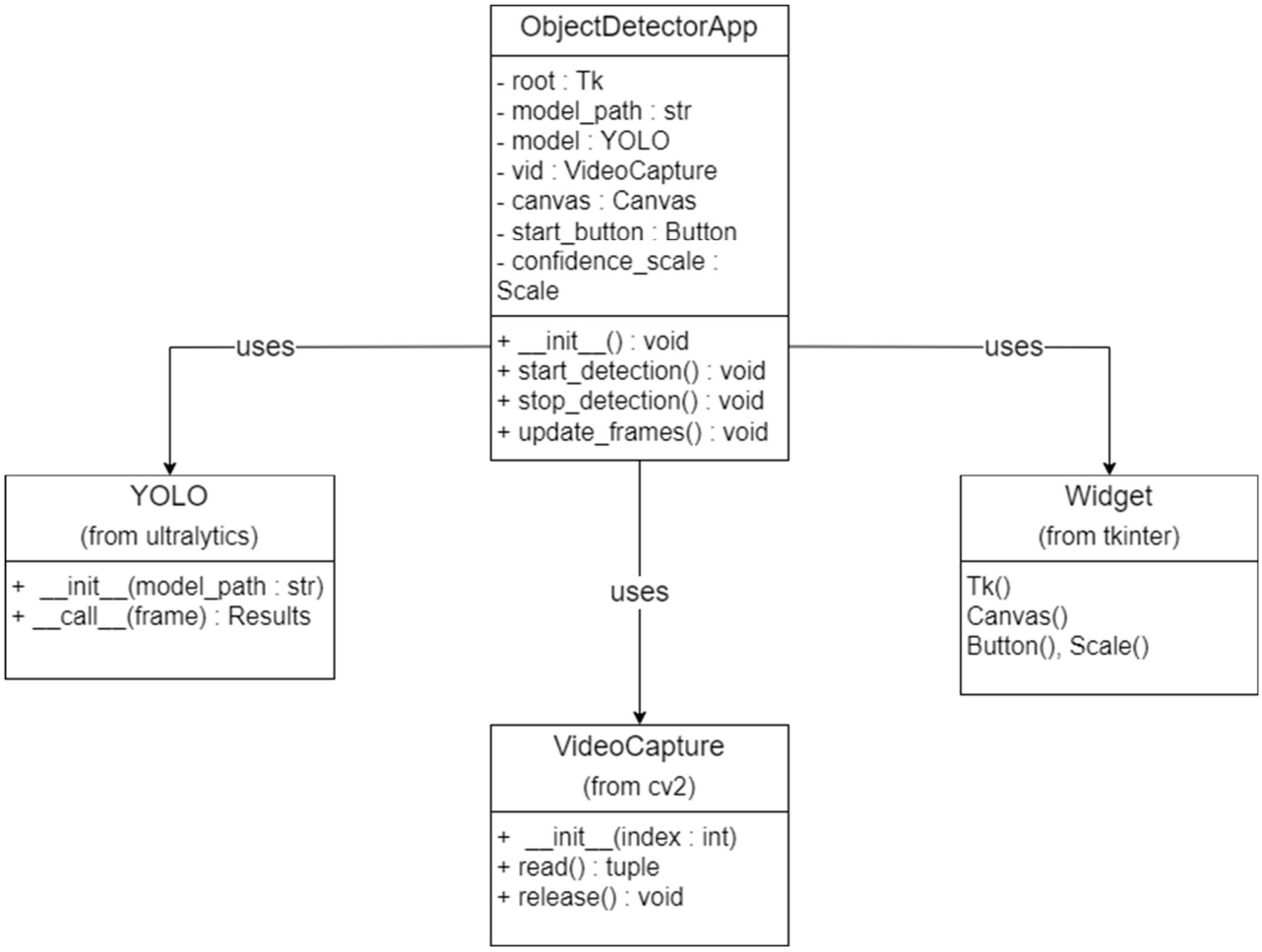
- **Main View:** The primary interface is a single window that displays the live, processed webcam feed.

- **Visual Feedback:** The design focuses on clear visual feedback, with colored bounding boxes and easy-to-read text labels overlaid directly onto the video stream, making the detection results immediately understandable.

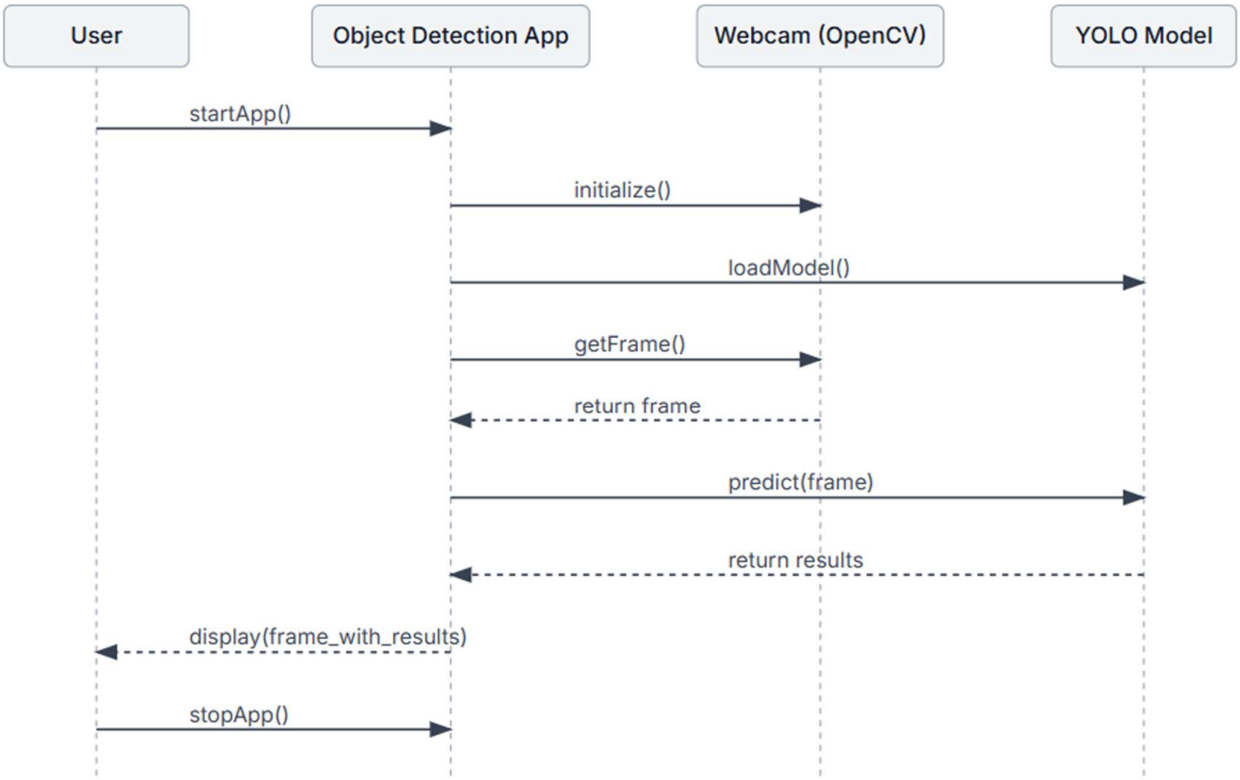
4.6.2 Use case Diagram



4.6.3 Class Diagram

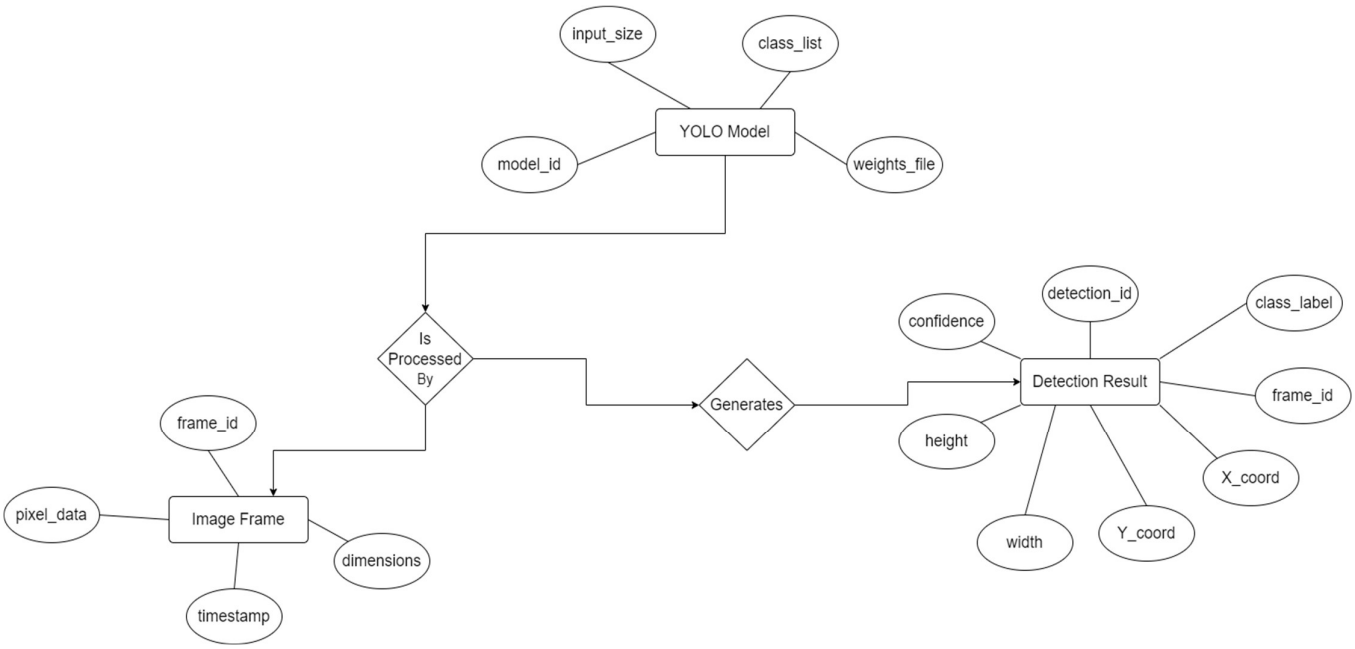


4.6.4 Sequence Diagram



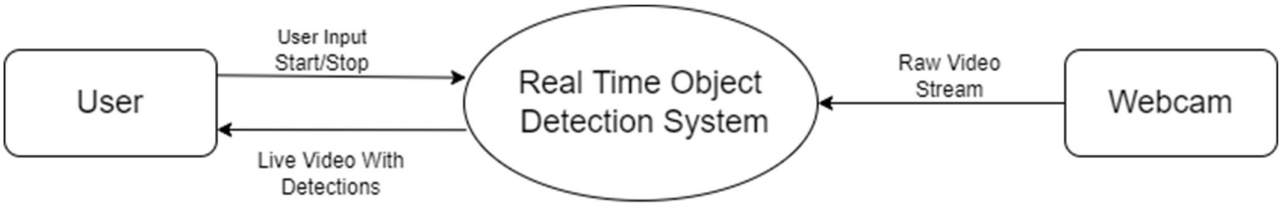
4.6.5 E-R Diagram

- User E-R Diagram

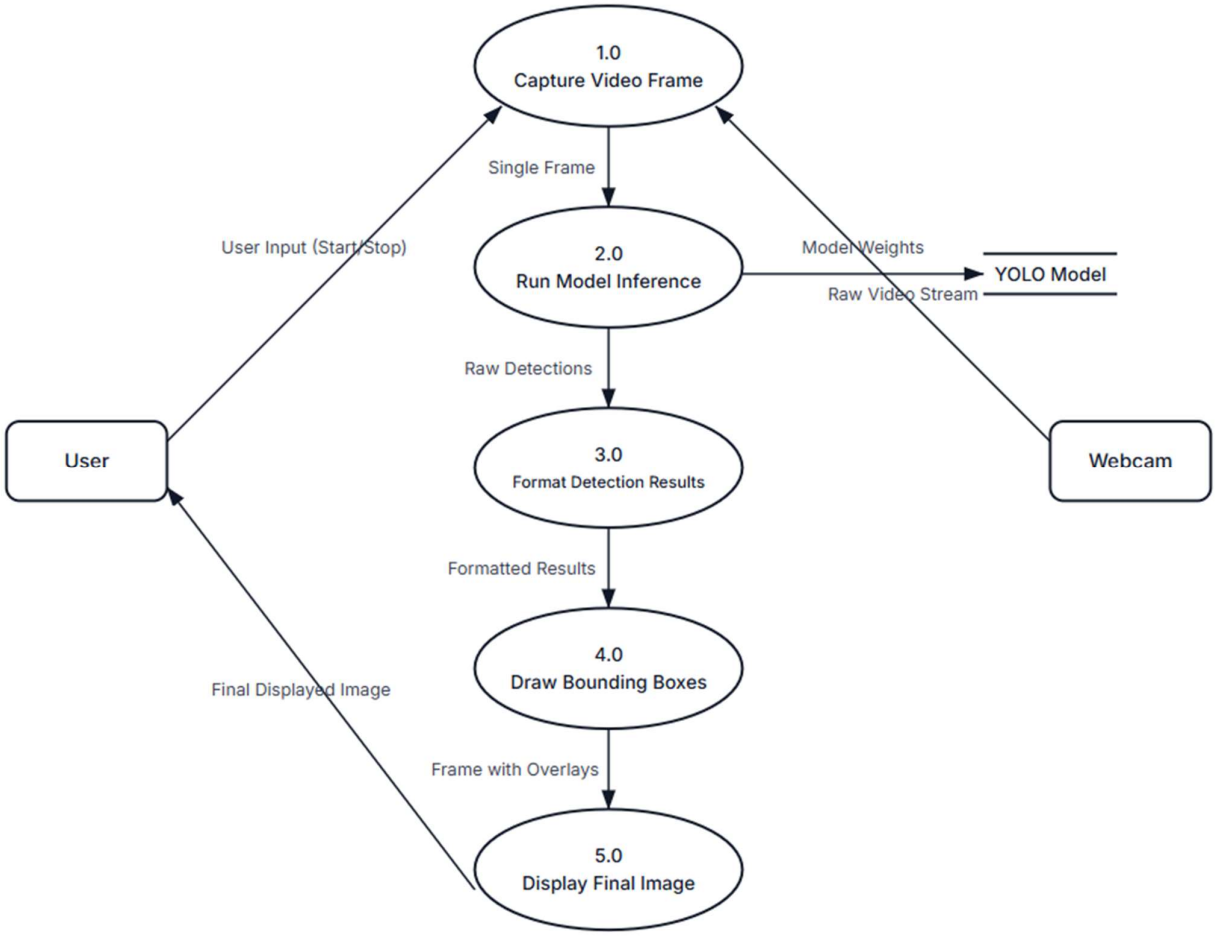


4.6.6 Data Flow Diagram

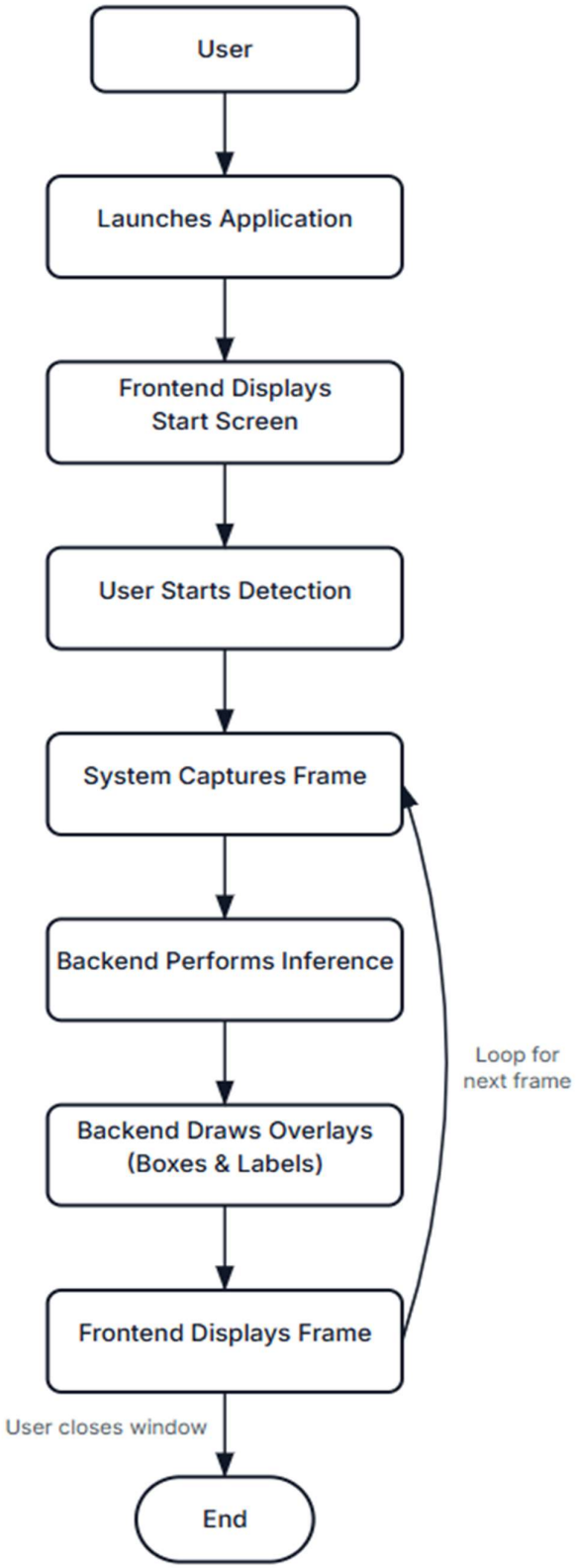
- DFD 0 LEVEL



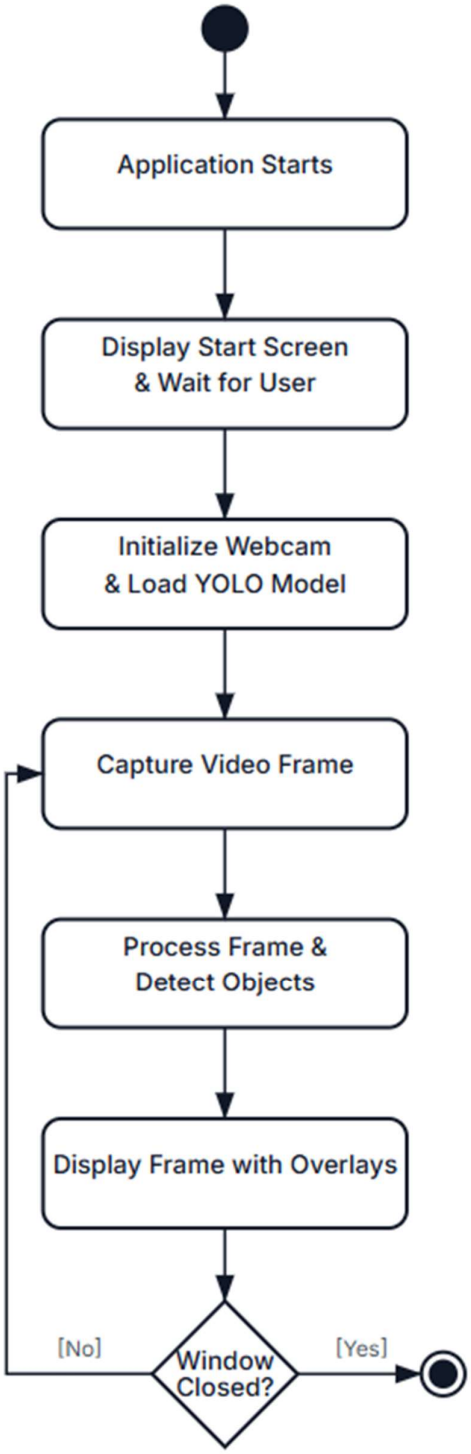
- DFD 1 LEVEL



- DFD 2 LEVEL


User
Launches Application
Frontend Displays Start Screen
User Starts Detection
System Captures Frame
Backend Performs Inference
Backend Draws Overlays (Boxes & Labels)
Frontend Displays Frame
Loop for next frame
User closes window
End

4.6.7 Activity Diagram



CHAPTER 5

SYSTEM IMPLEMENTATION

5.1 Implementation Environment
5.2 User Side Screen Shot
5.3 Admin Side Screen Shot
5.4 Sample Coding

5.1 Implementation Environment

1. **Development & Runtime Environment (Local Machine):**

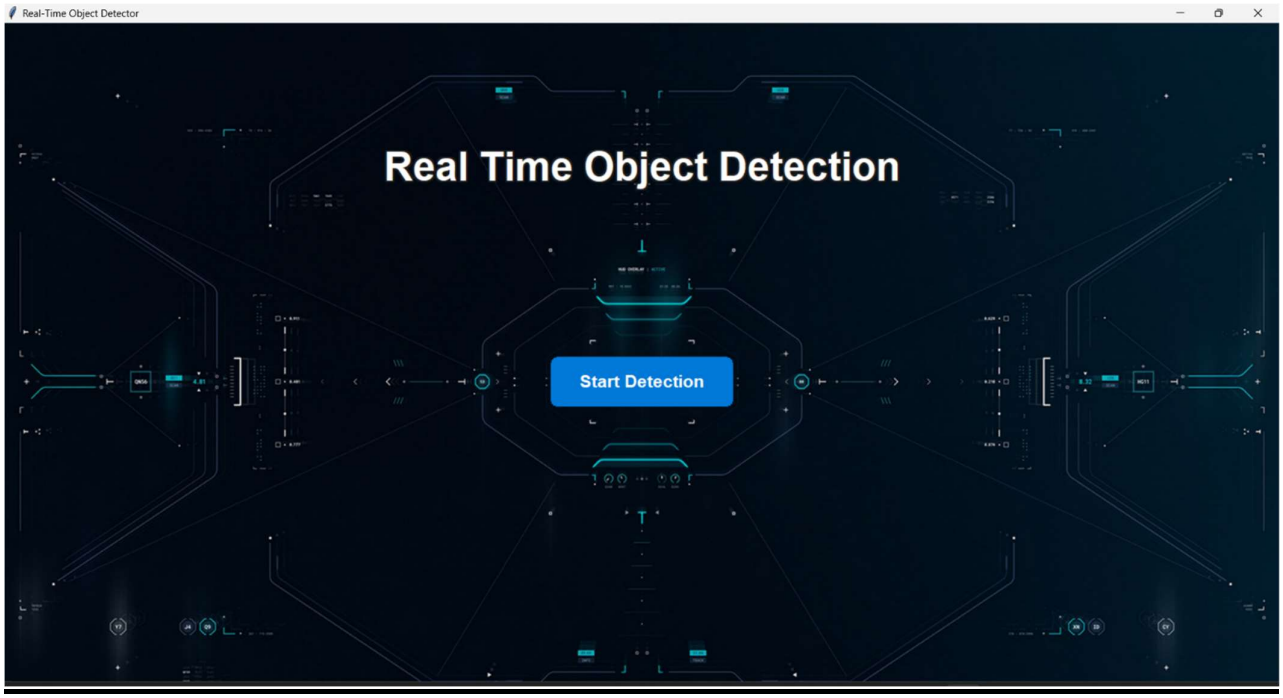
- **Hardware:**
  - A standard desktop or laptop computer.
  - A connected webcam for live video input.
  - A dedicated NVIDIA GPU for smooth real-time performance.
- **Operating System:**
  - Cross-platform: The application is designed to run on Windows, macOS, or a Linux distribution.
- **Software:**
  - **Python (3.8+):** The core programming language.
  - **Code Editor/IDE:** A standard code editor like Visual Studio Code, PyCharm, or similar.
  - **Key Python Libraries:**
    - Tkinter: For building the graphical user interface.
    - OpenCV: For capturing and processing the webcam feed.
    - Ultralytics YOLO: For loading the model and running inference.

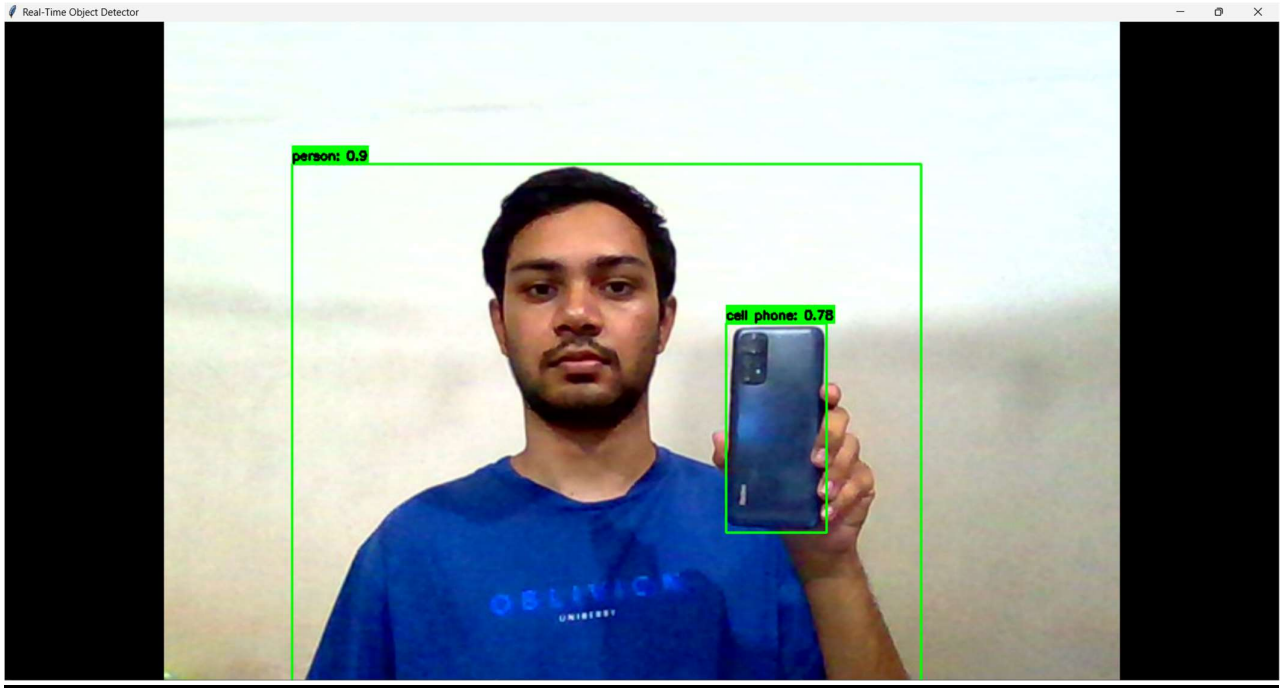
2. **Training Environment :**

- **Hardware:**
  - NVIDIA Tesla P100 GPU: A high-performance, graphics card essential for training on the large dataset.
- **Platform:**
  - Kaggle Notebooks: A cloud-based Jupyter Notebook environment that provided free access to the required GPU.
- **Software:**
  - The standard Kaggle environment, which includes Python and necessary deep learning libraries like PyTorch.
  - Roboflow: A web-based platform used for managing, preprocessing, and exporting the custom dataset.

5.2 User Side Screen Shot

Shows the system in action, displaying a live webcam feed with bounding boxes and labels overlaid on the detected objects.





5.4 Sample Coding

- **Loads the YOLO model onto the appropriate device:**

```
def load_model(self):
    model_path = 'best.pt'
    if not os.path.exists(model_path):
        print(f"'{model_path}' not found. Falling back to 'yolov11n.pt'.")
        model_path = 'yolov11n.pt'

    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    print(f"Loading model '{model_path}' onto device: {device}")
    try:
        self.model = YOLO(model_path)
        self.model.to(device)
        print("Model loaded successfully.")
    except Exception as e:
        print(f"Error loading model: {e}")
        self.on_closing()
```

- **Continuously gets frames, performs detection, and update GUI:**

```
def update_frame(self):  3 usages
    if not self.is_detection_running:
        return

    success, frame = self.cap.read()
    if not success:
        self.root.after(10, self.update_frame)
        return
```

CHAPTER 6

TESTING

6.1 Testing Plan

6.2 Test case Modules

6.1 Testing Plan

1. Unit Testing:

- Objective: To verify that individual methods work correctly.
- Actions:
  - Test the model loading logic to ensure it correctly picks best.pt or the default model.
  - Test the start_video() and stop_video() methods to confirm they manage the application's state properly.

2. Integration Testing:

- Objective: To ensure different parts of the application work together.
- Actions:
  - Verify that clicking the "Start Detection" button successfully displays the webcam feed on the GUI.
  - Confirm that video frames from OpenCV are correctly passed to the YOLO model and that the detection results are properly drawn back onto the video feed.

3. System Testing:

- Objective: To test the application's end-to-end functionality from a user's perspective.
- Actions:
  - Perform a complete workflow test: start the app, run detection, stop detection, and close the application to ensure a smooth, error-free experience.
  - Test how the application handles the absence of a webcam.

4. Performance Testing:

- Objective: To measure the application's real-time speed and efficiency.
- Actions:
  - Measure the average Frames Per Second (FPS) during a live detection session.
  - Monitor the CPU and RAM usage to ensure the application runs efficiently without overloading the system.

6.2 Test Case Module

1. Unit Testing

TC-U-01: Verify custom model loading

- Steps: Place a best.pt file in the root directory and launch the application.
- Expected Result: The console prints "Loading custom model from: best.pt".

TC-U-02: Verify default model loading

- Steps: Ensure no best.pt file is present and launch the application.
- Expected Result: The console prints "Custom model not found. Loading base model: yolov10n.pt".

TC-U-03: Verify video running state

- Steps: Launch the app and click "Start Detection".
- Expected Result: The video_running flag becomes True, and the button text changes to "Stop Detection".

TC-U-04: Verify video stopping and cleanup

- Steps: Start the video feed, then click "Stop Detection".
- Expected Result: The video_running flag becomes False, and the video feed disappears from the GUI.

2. Integration Testing

TC-I-01: Verify GUI and video capture integration

- Steps: Launch the app and click "Start Detection".
- Expected Result: The webcam initializes, and the live video feed is displayed correctly in the Tkinter window.

TC-I-02: Verify video frame and model integration

- Steps: Start detection and place a breakpoint in the update_frame method.
- Expected Result: The frame from OpenCV is passed to the YOLO model without any data type or format errors.

TC-I-03: Verify model results and GUI display

- Steps: Present a known object to the camera.
- Expected Result: Bounding boxes and labels from the model are correctly drawn onto the video frame by OpenCV.

3. System Testing

TC-S-01: Verify complete user workflow

- Steps: Launch the app, start detection, verify detection, stop detection, and close the window.
- Expected Result: The application performs all steps smoothly and exits cleanly without errors

TC-S-02: Verify error handling for no webcam

- Steps: Disable or disconnect the webcam, then launch the app and start detection.
- Expected Result: The application does not crash and prints "Error: Could not open webcam." to the console.

4. Performance Testing

TC-P-01: Measure Frames Per Second (FPS)

- Steps: Run detection for 60 seconds and calculate the average frame processing time.
- Expected Result: The average FPS is above a predefined threshold (e.g., > 15 FPS) for real-time performance.

TC-P-02: Monitor Resource Consumption

- Steps: Run detection and use a system monitor (like Task Manager).
- Expected Result: CPU and RAM usage remain within acceptable limits, and the system does not become unresponsive.

CHAPTER 7

Limitation of Project

7.1 Limitation of Project

There are following limitation of this system:

- The application's real-time performance is highly dependent on the user's hardware, requiring a dedicated GPU for smooth operation.
- The custom model is specialized and can only detect the specific classes of objects it was trained on.
- Detection accuracy is limited by real-world environmental factors like poor lighting, object occlusion, and camera quality.
- The system is designed for live viewing only and does not include functionality to save or log detection data for later analysis.
- The application only supports a single, live webcam feed and cannot process multiple video streams or pre-recorded files.

CHAPTER 8

CONCLUSION

8.1 Conclusion

The development of a reliable and useful real-time object detection application was the project's successful completion. From the initial development of the GUI and integration of the video stream to the final deployment of a YOLOv10 model that had been specially trained, the system successfully achieved all of its primary goals. The comprehensive testing strategy, which includes system, integration, unit, and performance validation, verifies that the application is dependable, stable, and operates in accordance with its requirements. The project effectively navigates the challenges of large-scale data management and model training to produce a high-quality end product, demonstrates a full end-to-end machine learning workflow.

The finished product serves as a potent proof-of-concept, demonstrating that it is possible to create specialized, high-performing computer vision systems with readily available, contemporary tools. Although the project's main objectives were met, its drawbacks include a model that is only tailored for its training data and a reliance on powerful hardware for optimum performance. This solid foundation could be expanded in the future by investigating deployment on specialized edge devices, improving the model with a larger dataset, or adding new functionalities like data logging and analysis to increase its practicality.

Chapter 9

Reference

9.1 Reference

- Ultralytics Library: https://github.com/ultralytics/ultralytics.
- OpenCV Library: https://opencv.org.
- Python Programming Language: https://www.python.org.
- PyTorch Framework: https://pytorch.org.
- Tkinter GUI Framework: https://docs.python.org/3/library/tkinter.html.
- Roboflow Platform: https://roboflow.com.
- Kaggle Platform: https://www.kaggle.com.
- Information and Learning: https://www.stackoverflow.com.
- https://www.geeksforgeeks.com.
- https://gemini.google.com