A

PROJECT REPORT

ON

VisionAI Object Detector

SUBMITTED IN THE PARTIAL FULFILLMENT OF THE REQUIREMENT FOR OF

MASTER OF COMPUTER APPLICATION

SUBMITTED BY

Rohit Kadam: FYMCA119

Animesh Jain : FYMCA120

(SEMESTER -III)

UNDER THE GUIDANCE OF

Asst. Prof. Rasika Salunkhe

G. H. RAISONI COLLEGE OF ENGINEERING AND MANAGEMENT

2025-2026

(An Autonomous Institute affiliated to SPPU)

WAGHOLI, PUNE –412207

SAVITRIBAI PHULE PUNE UNIVERSITY

2025 - 2026

G H raisoni COLLEGE
Engineering and Management
Pune

CERTIFICATE

This is to certify that the project report entitled

[VisionAI Object Detector]

SUBMITTED BY

Rohit Kadam: FYMCA119

Animesh Jain : FYMCA120

Are bonafide students of this institute and the work has been carried out by them under the supervision of **Asst. Prof. Rasika Salunkhe** and it is approved for the partial fulfillment of the requirement of Savitribai Phule Pune University, for the award of the degree of **Master of Computer Application.**

Asst. Prof. Rasika Salunkhe		**Dr. Priya Chaudhari**
Internal Gui	**External**	**HOD - MCA**
Asst.Prof.Kishor Markad sir		**Dr. N. B. Hulle sir**
Project Co-ordinator		**Director**

Place: Pune

Date:

Acknowledgement.

We are profoundly grateful to **Asst Prof. Rasika Salunkhe** for his expert guidance and continuous encouragement throughout to see that this project rights its target since its commencement to its completion.

We would like to express our deepest appreciation towards **Dr. N. B. Hulle sir**, Director, G. H. Raisoni College of Engineering and Management, Pune, **Dr. Priya Chaudhari**, Head of Department, Master of Computer Application and **Asst.Prof.Kishor Markad** Project Coordinator whose invaluable guidance supported us in completing this project.

At last, we must express our sincere heartfelt gratitude to all the staff members of the Computer Engineering Department who helped us directly or indirectly during this course of work.

ROHIT KADAM _____

ANIMESH JAIN _____

INDEX

I. Chapter-1

Introduction

1.1 Abstract

The **VisionAI Object Detector** project is a comprehensive computer vision application designed to automatically identify, classify, and track objects within images or video streams. Leveraging deep learning models like YOLO via OpenCV's DNN module, the system moves beyond manual detection to recognize objects such as people, vehicles, and everyday items in real-time. A key feature is its **ensemble approach**, employing both basic and advanced model combinations (model_ensemble.py, advanced_ensemble.py) to enhance detection accuracy and robustness. The system organizes results with bounding boxes, labels, and confidence scores, further enhanced by **integrated object tracking** (object_tracker.py) to follow items across frames.

User interaction is facilitated through three primary components:

1. A versatile **command-line launcher (main.py)** featuring an interactive menu to access all parts of the project.
2. A sophisticated **Streamlit web dashboard (dashboard.py)** offering rich visualization, live per-image analytics, and interactive filtering.
3. A standalone **security monitor (detector.py)** that sends real-time **Telegram alerts** when objects enter a pre-defined restricted zone.

Built with Python, OpenCV, and Streamlit, the project incorporates **detailed performance analytics** (performance_analytics.py), logging detection history and metrics for analysis. Originally motivated by the demand for intelligent visual systems, this significantly refactored project now provides an efficient, accurate, and scalable solution for applications in surveillance, automation, traffic monitoring, and data analysis, supported by a robust and maintainable codebase structure.

1.2 Existing System and Need for System :

Existing System

The project is a complete object detection platform with three main parts:

1. **Command-Line Interface (main.py):** An interactive menu that acts as the central launcher for all project features.
2. **Web Dashboard (dashboard.py):** A user-friendly Streamlit web app that lets anyone use the detector. It includes:
    - Multiple ways to add images (upload, select from a folder, or webcam).
    - An interactive slider to filter detections by confidence.
    - Tabs that show the final image, a table of results, and live analytics.
3. **Standalone Monitor (detector.py):** A separate script that monitors a webcam feed for objects in a "restricted zone" and sends alerts to Telegram.
4. **Analytics Engine:** A backend system that logs all detection data, counts objects, and saves performance reports as charts and JSON files.

Need for the System

The project was built to solve several key problems:

- **Problem:** Raw object detection is just data; it's not useful information.
- **Need:** To turn detections into **measurable insights**.
- **Solution:** The analytics engine counts objects, tracks performance, and builds reports.

- **Problem:** Running Python scripts is difficult for most people.
- **Need:** To make the technology **accessible to everyone**.
- **Solution:** The **Web Dashboard** provides a simple, graphical interface that anyone can use in a web browser.

- **Problem:** A static video or report doesn't let you explore the data.
- **Need:** To let users **interact with the results**.
- **Solution:** The **interactive confidence slider** on the dashboard lets users filter out "noise" and explore the results in real-time.

- **Problem:** A project with many scripts is confusing to use and demonstrate.
- **Need:** A single, professional way to **launch all features**.
- **Solution:** The **main.py interactive menu** unifies the project, making it easy to run any part from one place.

1.3 Scope of the project VisionAI Object Detector:

This project is a complete object detection tool. Its main functions are:

- **Finds & Tracks Objects:** Uses YOLO models to find and track objects (like people or cars) in images and videos.
- **Two Ways to Use:**
    1. **Terminal Menu (main.py):** A simple, interactive menu for running all the different features.
    2. **Web Dashboard (dashboard.py):** A full, user-friendly website (built with Streamlit) for visual use.
- **Interactive Dashboard:** The web dashboard isn't just for viewing. You can:
    - Upload your own images.
    - Use your live webcam.
    - Select from a folder of sample images.
    - **Filter results in real-time** with a confidence slider.
- **Live Analytics:** Instantly shows you important data for each detection (like total object count, average confidence, and processing time) in its own "Live Analytics" tab.
- **Security Monitor:** Includes a standalone mode (detector.py) that watches a specific "restricted zone" on a webcam and sends an alert to **Telegram** if an object enters it.
- **Saves Reports:** Automatically logs performance data and can create summary reports as charts (.png) and data files (.json).

1.4 Operating Environment:

Hardware

The system is designed to run on a standard personal computer and leverage modern processing units for real-time performance.

- **Host Machine:** A PC or laptop (e.g., Lenovo laptop).
- **Processing Unit (CPU):** A multi-core processor (e.g., Intel i5/i7/i9, AMD Ryzen 5/7/9) is required for general application logic and as a fallback for model inference.
- **Graphics Processing Unit (GPU) (Recommended):** An NVIDIA GPU with **CUDA** support is highly recommended for real-time detection, as the system is configured to use GPU acceleration via OpenCV's DNN module.
- **Input Device:** A built-in laptop webcam or an external USB webcam is required for live video capture modes.

Software

The project is built on a stack of open-source Python libraries and runs within a controlled virtual environment.

- **Operating System: Windows 10/11**.
- **Core Language: Python** (version 3.9+).
- **Environment Manager: Python Virtual Environment (venv)** is used to isolate project dependencies.
- **Core Libraries (Frameworks):**
    - **OpenCV (cv2):** The primary computer vision library used for image/video processing, running the YOLO model (via the DNN module), and rendering bounding boxes.
    - **Streamlit:** The web framework used to build, run, and serve the entire interactive graphical user interface (dashboard.py).
    - **NumPy:** A fundamental library for high-performance numerical operations on image data, bounding boxes, and detection scores.
    - **Pandas:** Used for organizing detection data into tables (DataFrames) for display and analysis within the Streamlit dashboard and analytics module.
    - **Requests:** Used by the standalone detector.py script to send alert notifications to the Telegram API.

Database (Data Storage)

This project does **not** utilize a traditional relational (e.g., MySQL) or NoSQL (e.g., MongoDB) database. It employs a **flat-file system** for all data persistence and storage.

- **Model Storage:** AI models and configurations are stored as local files in the /models directory (e.g., .weights, .cfg, .names).
- **Report Storage:** The performance analytics module saves its output as structured data and image files in the /outputs directory (e.g., detection_report.json, performance_analysis.png).
- **Snapshot Storage:** The standalone security monitor saves image evidence as .jpg files in the /Image_snapshot directory.

II. Chapter-2

Analysis and Design

2.1 List of Modules:

1. Core Application Modules

These are the primary, custom-built Python scripts that define the project's features and user-facing interfaces.

- main.py (Main Launcher):
  - Acts as the central command-line interface (CLI) for the entire project.
  - Features an interactive menu that guides the user to launch all other components (demos, dashboard, etc.).
- dashboard.py (Web Dashboard):
  - A comprehensive, user-friendly graphical interface built with Streamlit.
  - Provides multiple input methods (upload, select, webcam), an interactive confidence filter, and a tabbed results view for visualization and analytics.
- detector.py (Standalone Monitor):
  - A dedicated security script that runs independently.
  - Monitors a "restricted zone" on a live webcam feed and sends Telegram alerts with image snapshots.
- advanced_ensemble.py (Core Logic):
  - The primary "brains" of the operation.
  - This class integrates the detection model, the object_tracker, and the performance_analytics engine into a single, cohesive system.
- src/utils/performance_analytics.py (Analytics Engine):
  - A backend class responsible for logging all detection metrics (object counts, confidence, time).
  - Generates and saves summary reports as JSON data files and PNG charts.
- src/utils/object_tracker.py (Tracker):
  - A utility class that implements tracking logic, assigning and following unique IDs for objects detected across multiple video frames.

2. Testing & Demonstration Modules

These scripts are located in the /tests directory and are used for validating and demonstrating the project's functionality.

- tests/ensemble_demo.py: A simple demo (launched by main.py) to show the basic webcam/image detection in a standard OpenCV window.
- tests/demo_advanced.py: A script for testing the advanced features of the advanced_ensemble, including analytics and tracking.
- tests/test_analytics.py: A unit test script designed to validate that the performance_analytics module is calculating and logging data correctly.

3. Key External Technologies (Libraries)

These are the major third-party libraries that power the project's functionality.

- Streamlit: The core framework used to build and serve the entire interactive web dashboard.
- OpenCV (cv2): The fundamental computer vision library used for all image and video processing, running the AI model (via its DNN module), and drawing on-screen.
- NumPy: A high-performance library for all numerical operations, essential for handling image arrays and detection data.
- Pandas: Used within the dashboard to organize detection results into clean, sortable tables (DataFrames).
- Requests: A simple library used by the detector.py module to communicate with the Telegram API for sending alerts.

2.2 project planning and scheduling:

This section outlines the implementation strategy, constraints, and timeline for the development of the VisionAI Object Detector project.

Internal Planning (Within the Project Team)

Implementation Strategy: The project was developed using a phased, modular approach to ensure each component was robust before integration. Phase 1 involved the Core Logic Development, implementing foundational classes like the model_ensemble, performance_analytics, and object_tracker. Phase 2, Feature Application, focused on building standalone applications, including the detector.py security monitor and the advanced_ensemble class which integrated all backend logic. Phase 3, Interface Development, involved creating the two main user-facing interfaces: the interactive main.py command-line menu and the comprehensive dashboard.py Streamlit web application. The final stage, Phase 4, Integration & Testing, focused on connecting all features to the main launcher, performing end-to-end testing, debugging pathing issues, and applying final UI/UX polish to the dashboard.

Constraints: The project was developed under several key constraints. The primary time constraint was a 12-week (3-month) development cycle. A zero-cost budget was maintained by exclusively leveraging open-source technologies like Python, OpenCV, and Streamlit, along with existing developer hardware. Technological constraints were present, as real-time performance is dependent on the available hardware (GPU recommended), and dashboard responsiveness is tied to Streamlit's architecture. Finally, as the project was managed by a solo developer, available time was the primary resource constraint.

Target Applications: The final system is designed for flexibility, targeting applications in Security & Surveillance (via the detector.py monitor), Interactive Data Analysis (via the dashboard.py), Traffic Monitoring (via object tracking and classification), and Automation & Robotics (by providing a core engine for visual-based decisions).

Stakeholder Planning (e.g., Project Supervisor)

Timely Meetings & Status Reports: Regular check-ins with the project supervisor were scheduled to demonstrate progress on new features, such as dashboard interactivity, and to discuss any technical roadblocks.

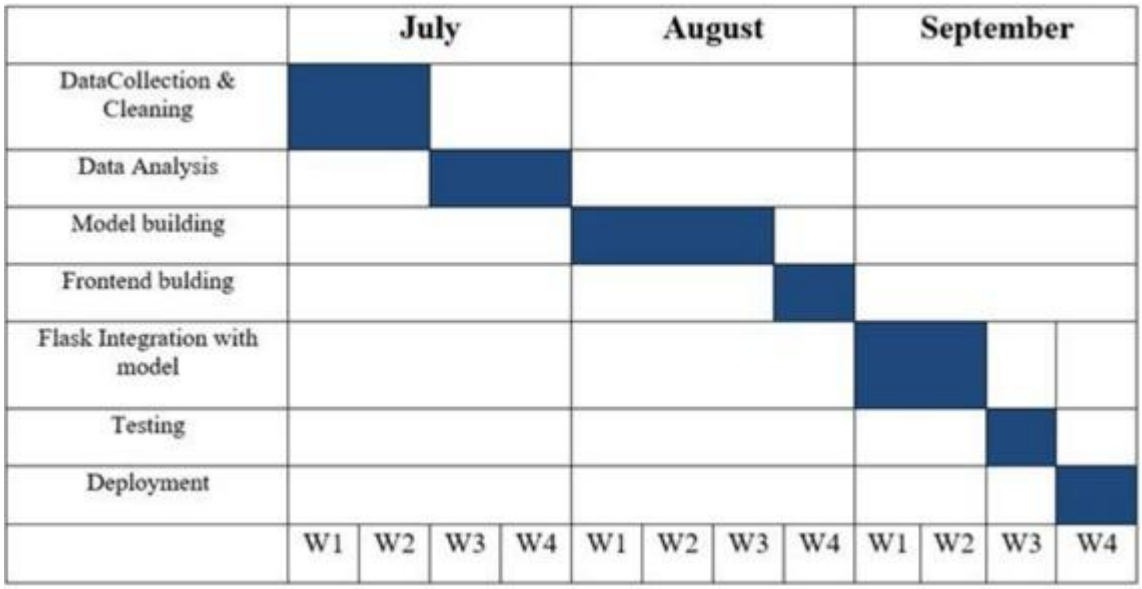
Feedback & Iteration: Feedback was actively solicited, particularly on the usability and intuitiveness of the dashboard.py interface and the clarity of the performance_analytics.py reports.

Milestones & Deliverables: The 12-week schedule was structured around key milestones. Weeks 1-4 (Phase 1) focused on delivering the core backend classes for detection, analytics, and tracking. Weeks 5-6 (Phase 2) delivered the standalone detector.py with Telegram alerts and the integrated advanced_ensemble class. Weeks 7-10 (Phase 3) delivered the main.py launcher and culminated in the fully functional dashboard.py application. Finally, Weeks 11-12 (Phase 4) were dedicated to

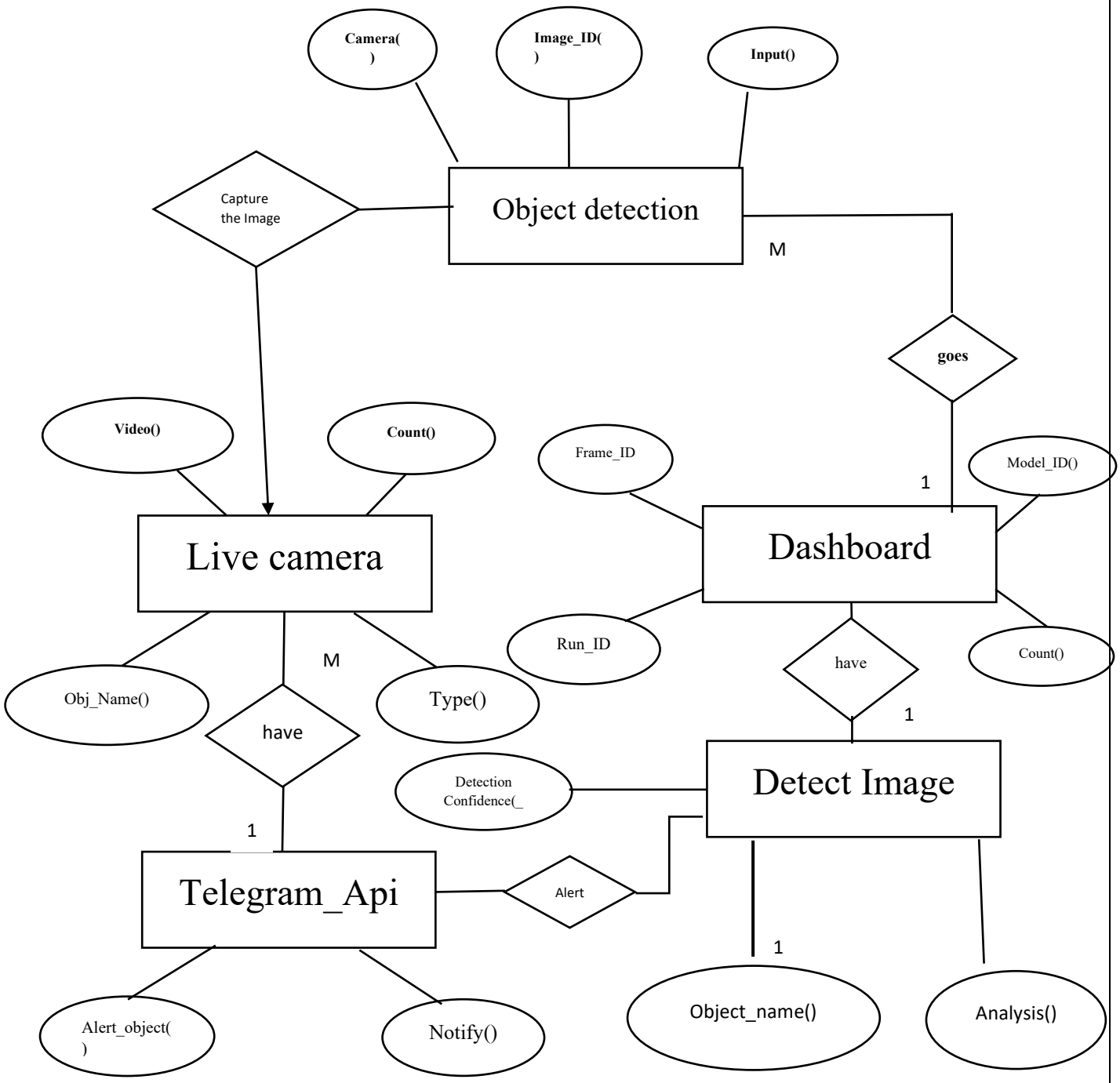
end-to-end testing, bug fixing, and the completion of all project documentation for the final submission.

Project Scheduling:

An elementary Gantt chart or Timeline chart for the development plan is given below. The plan explains the tasks versus the time (in weeks) they will take to complete



2.3 Entity Relationship Diagram:



2.4 Table Structure:

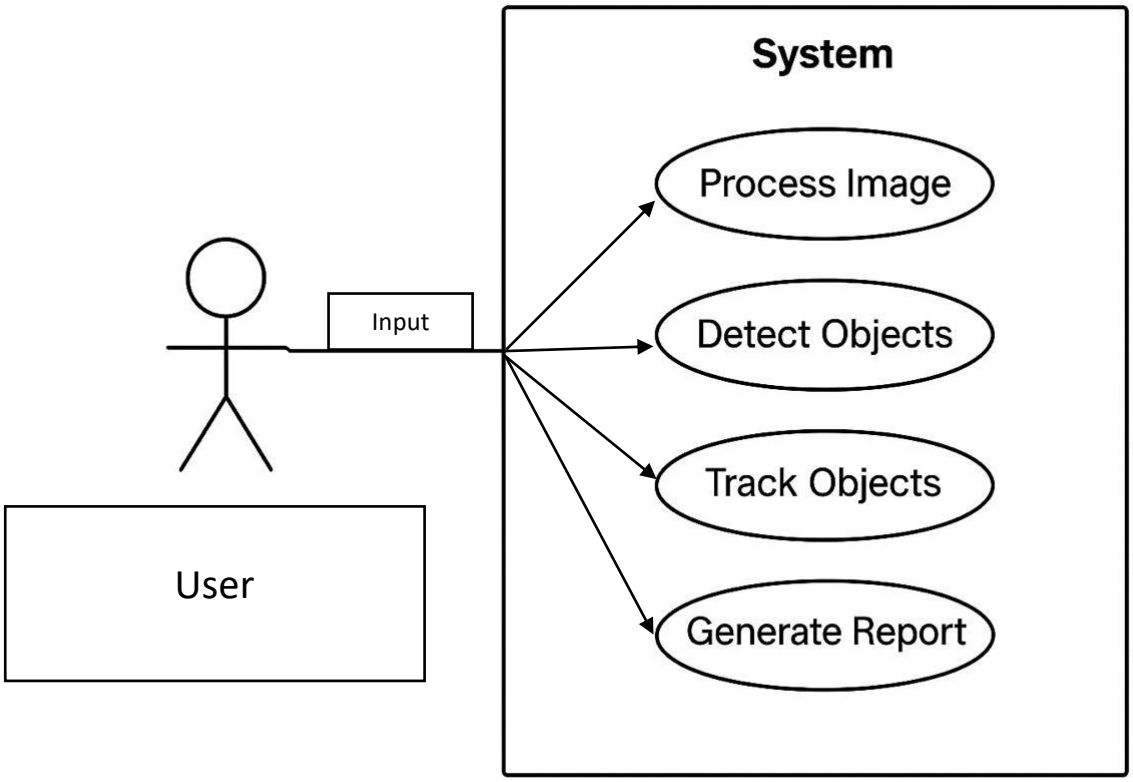
Table Name: ObjectClass

Field Name	Data Type	Description
ClassID	**INT (Primary Key)**	**Unique identifier for the object class (e.g., 0, 1, 2).**
ClassName	**VARCHAR(255)**	**The text name of the class (e.g., 'person', 'car').**

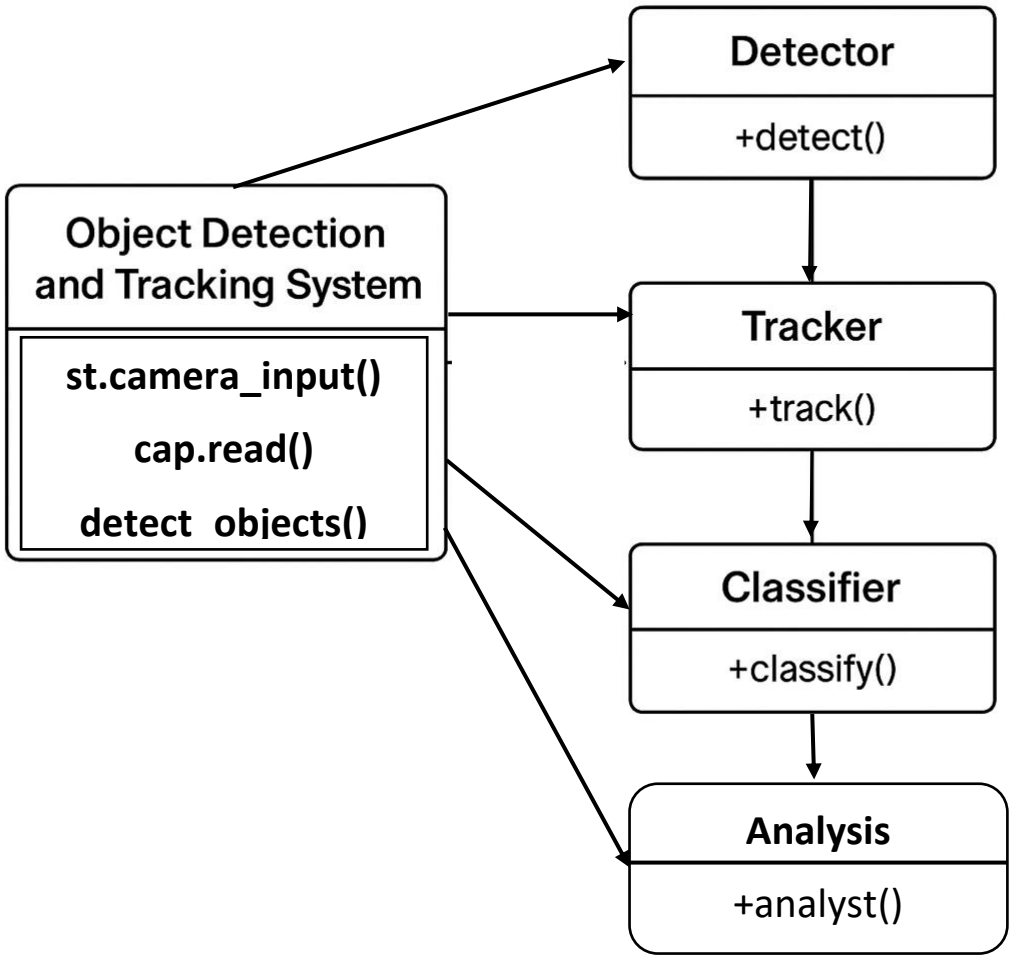
Table Name: TrackedObject

Field Name	Data Type	Description
TrackID	**INT (Primary Key)**	**Unique identifier assigned to a single object being tracked.**
ClassID	**INT (Foreign Key)**	**The class of this object (links to ObjectClass.ClassID).**

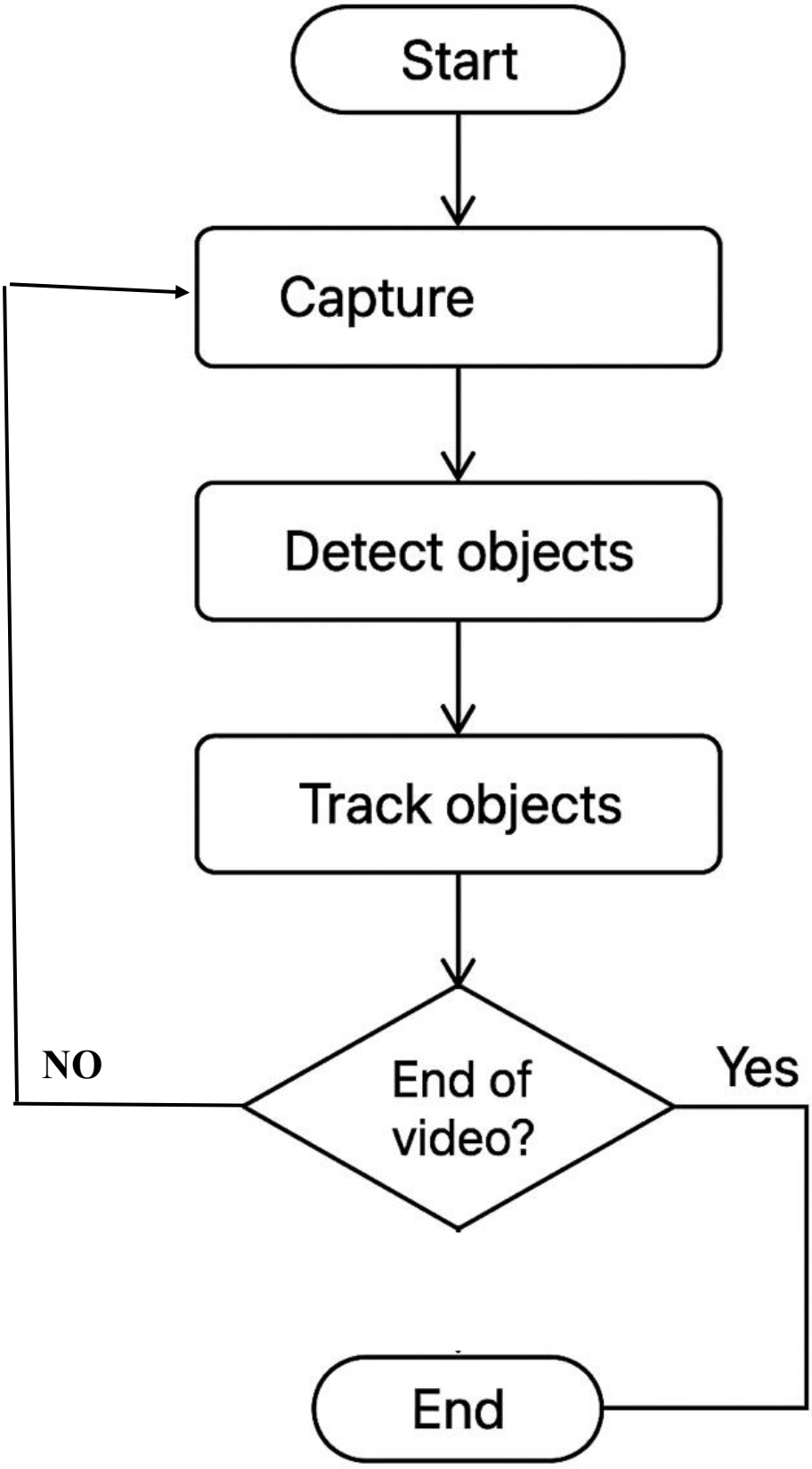
2.5 Use Case Diagram:



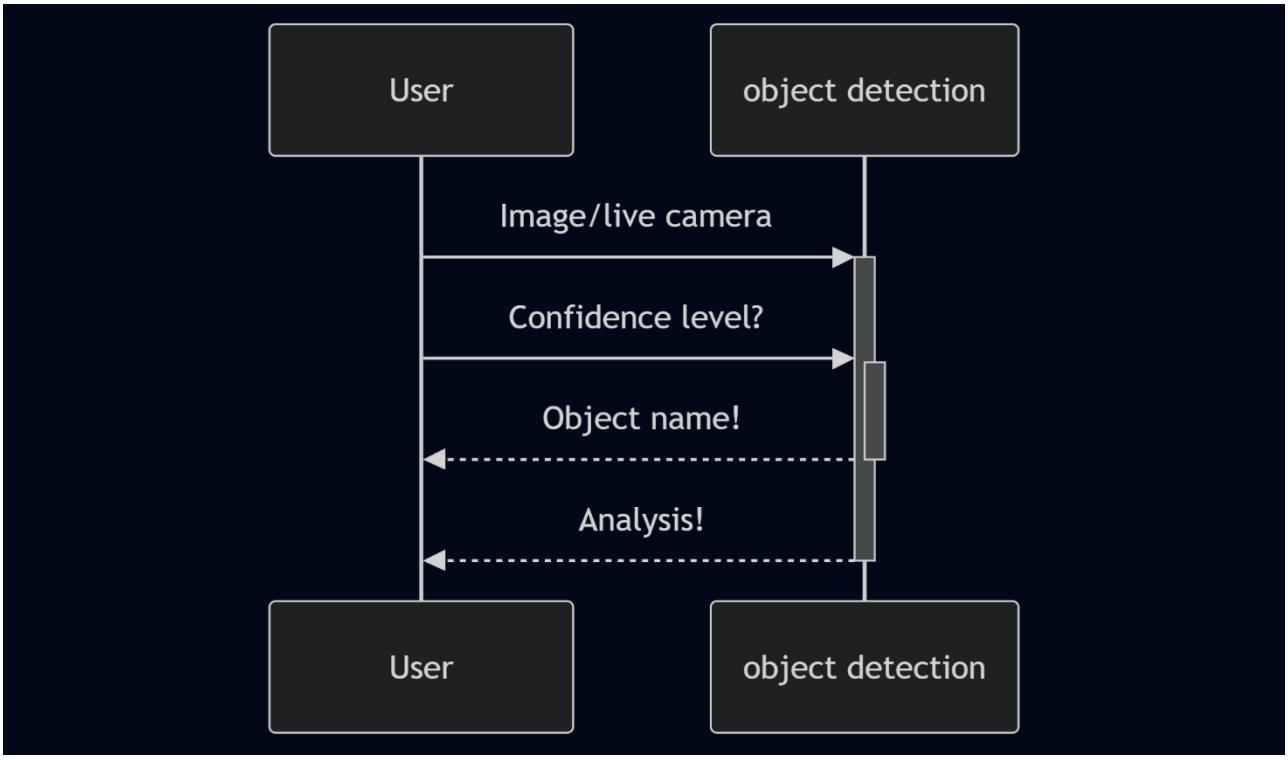
2.6 Case Diagram:



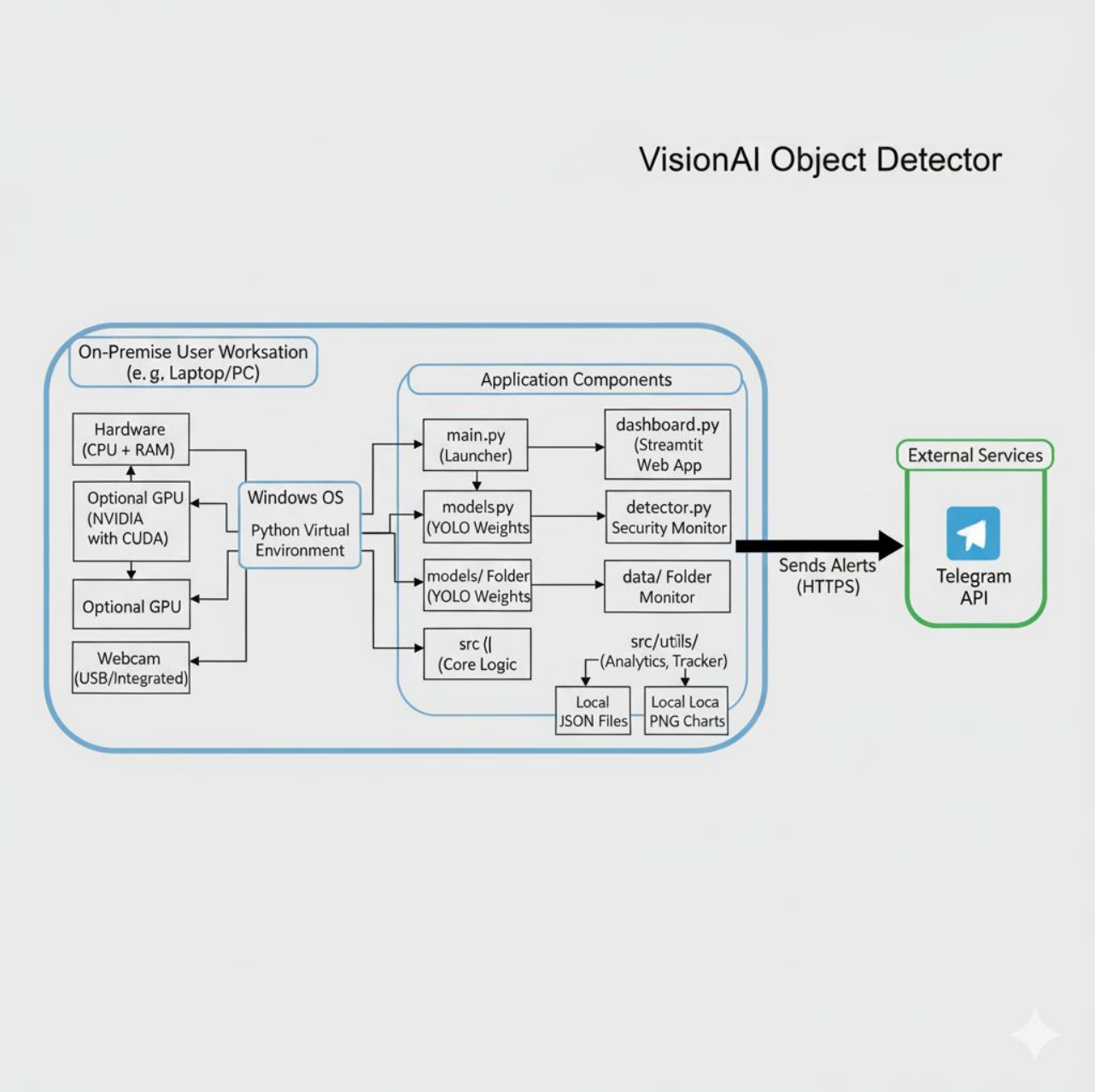
2.7 Activity Diagram:



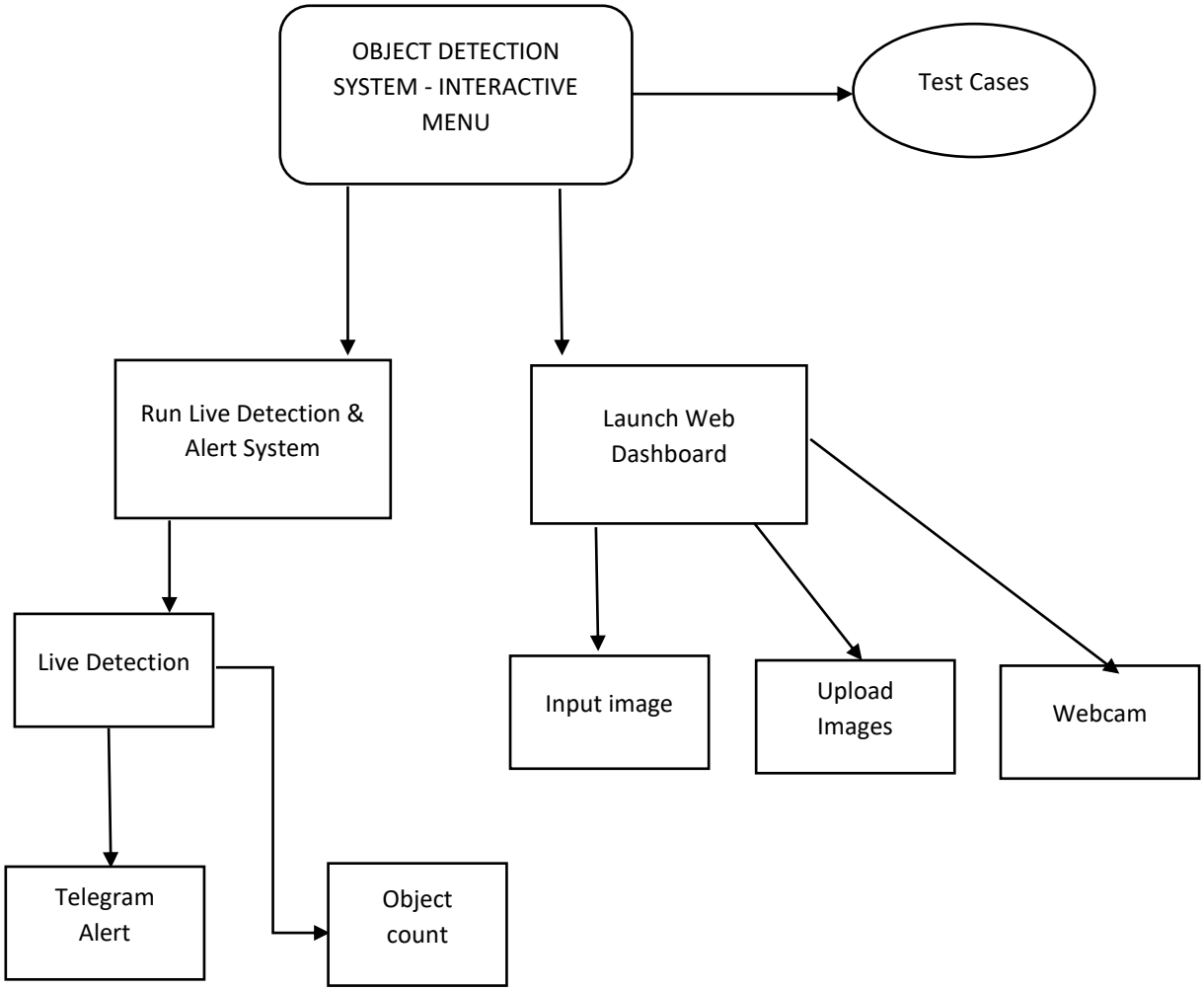
2.8 Sequence Diagram:



2.9 Deployment Diagram:

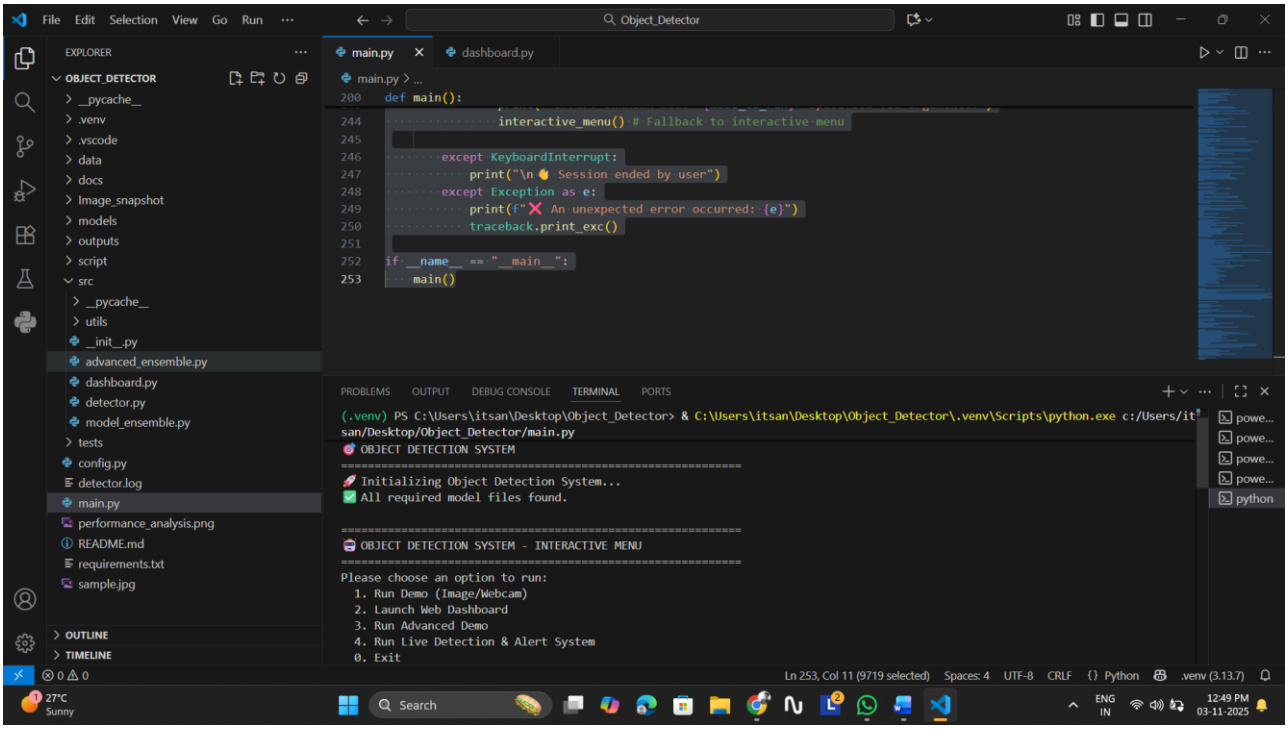


2.10 Module Hierarchy Diagram

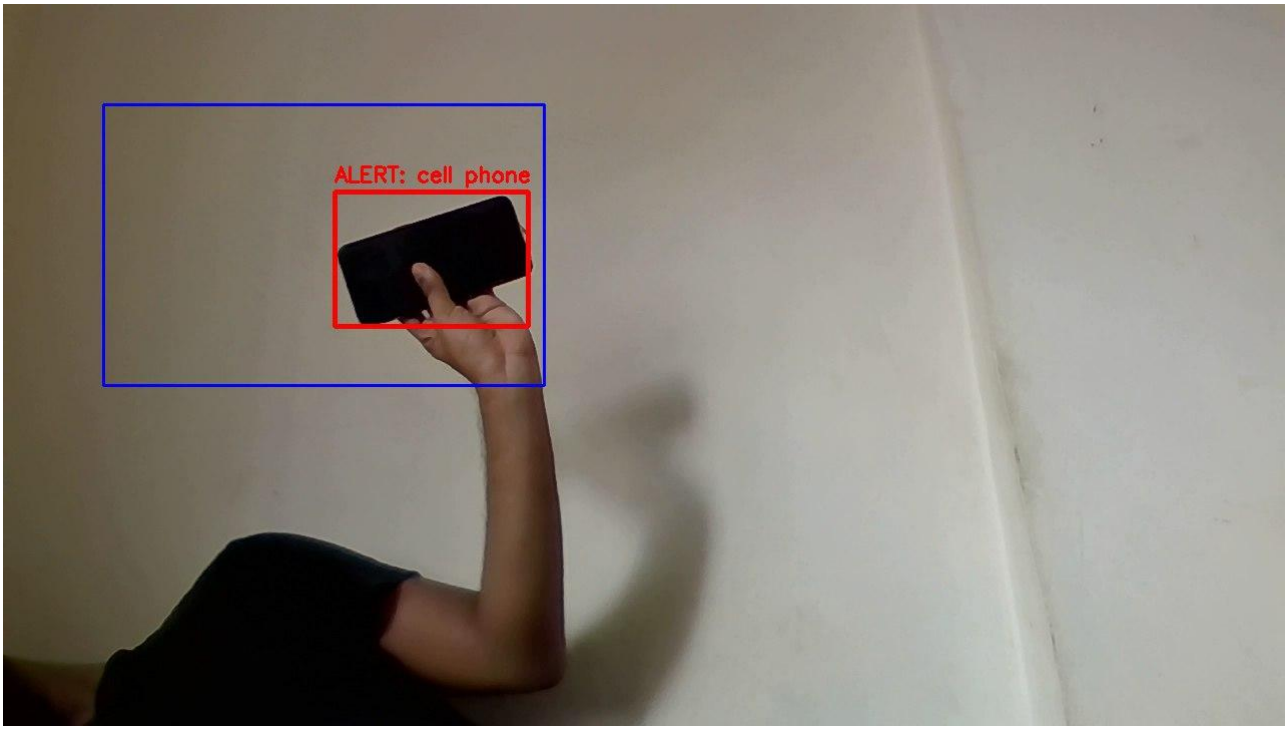


2.11 Input and Output Screenshot:

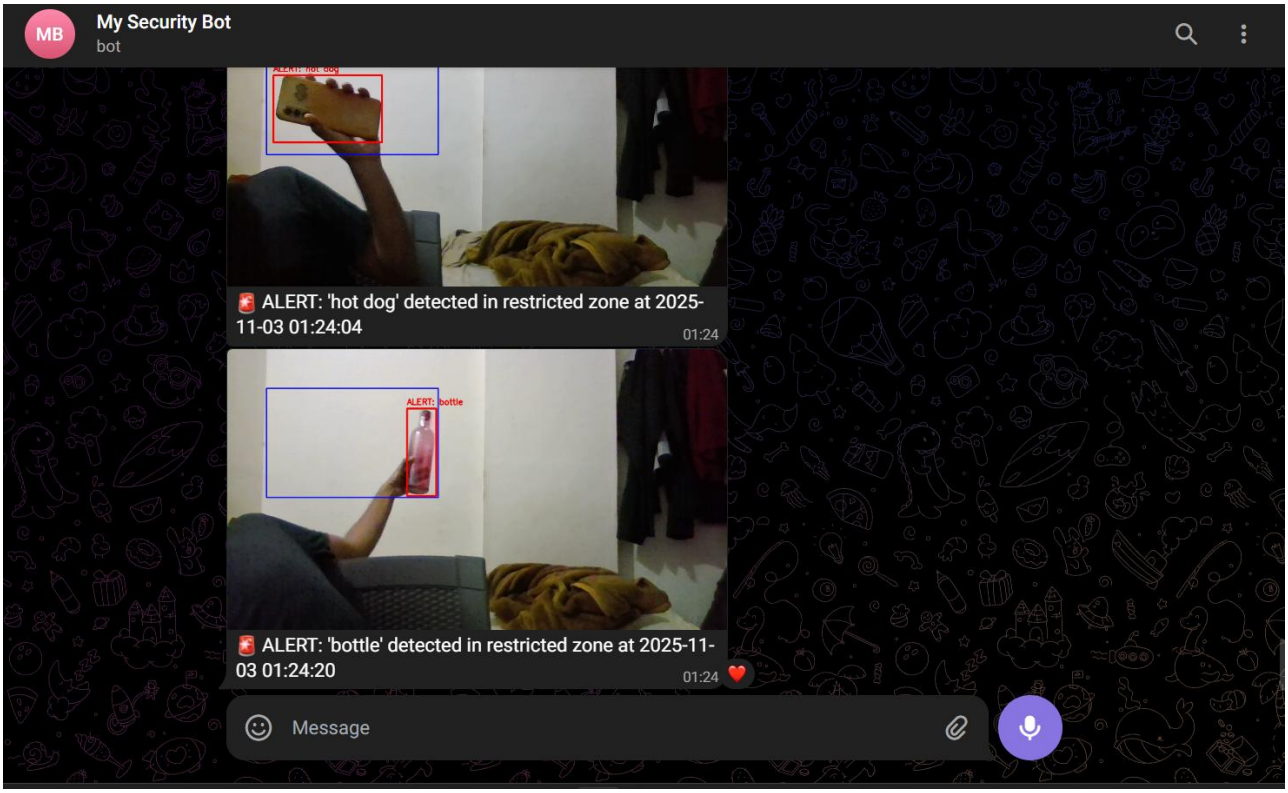
Selection section



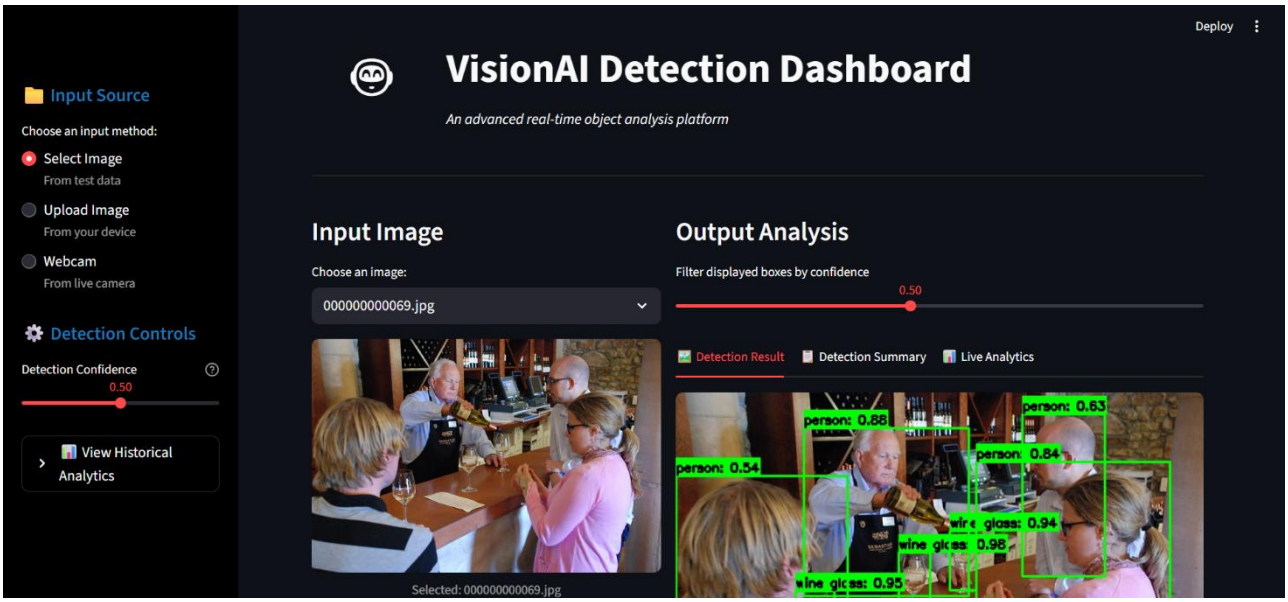
Live Detection



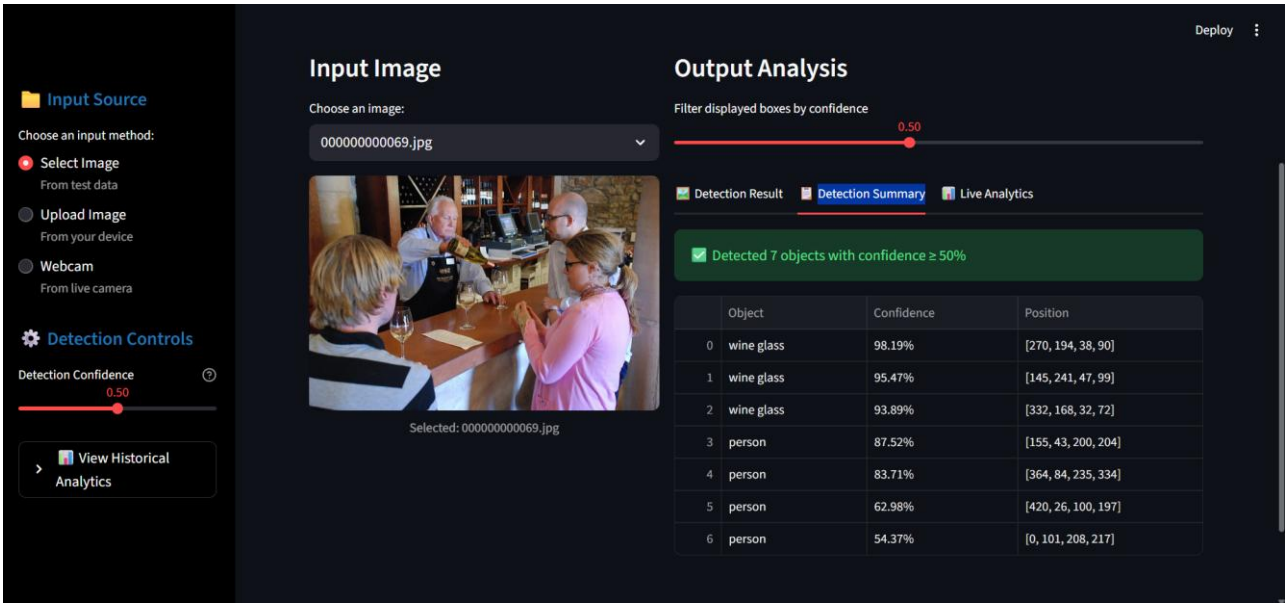
Alert System (Telegram)



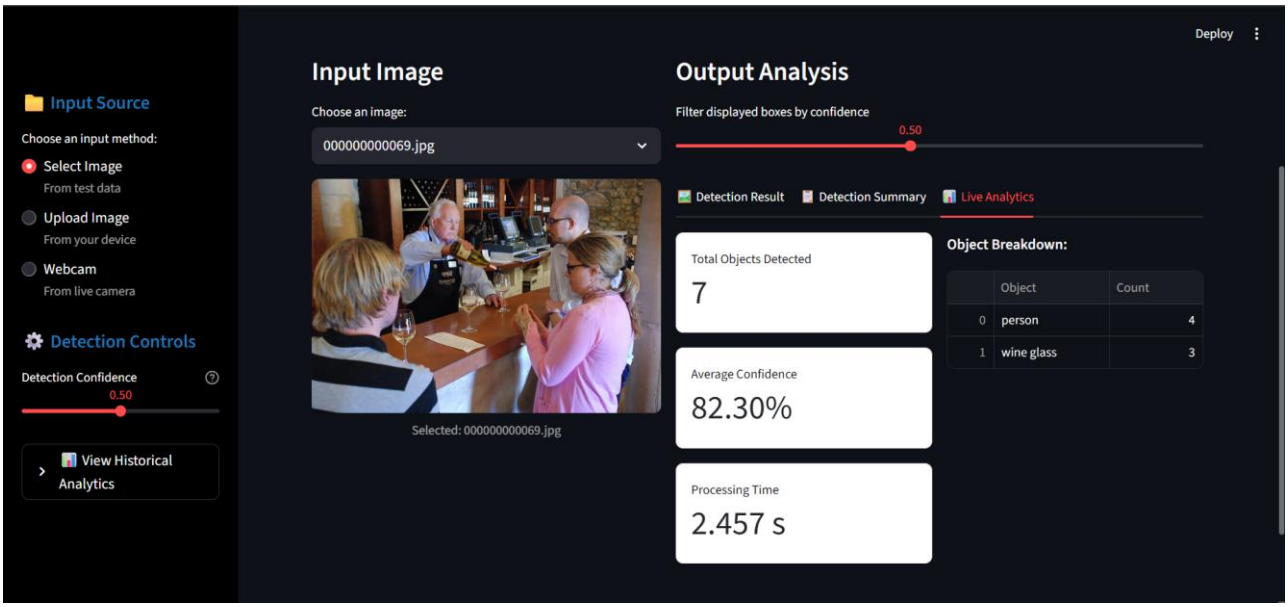
VisionAI Detection Dashboard – Detection result



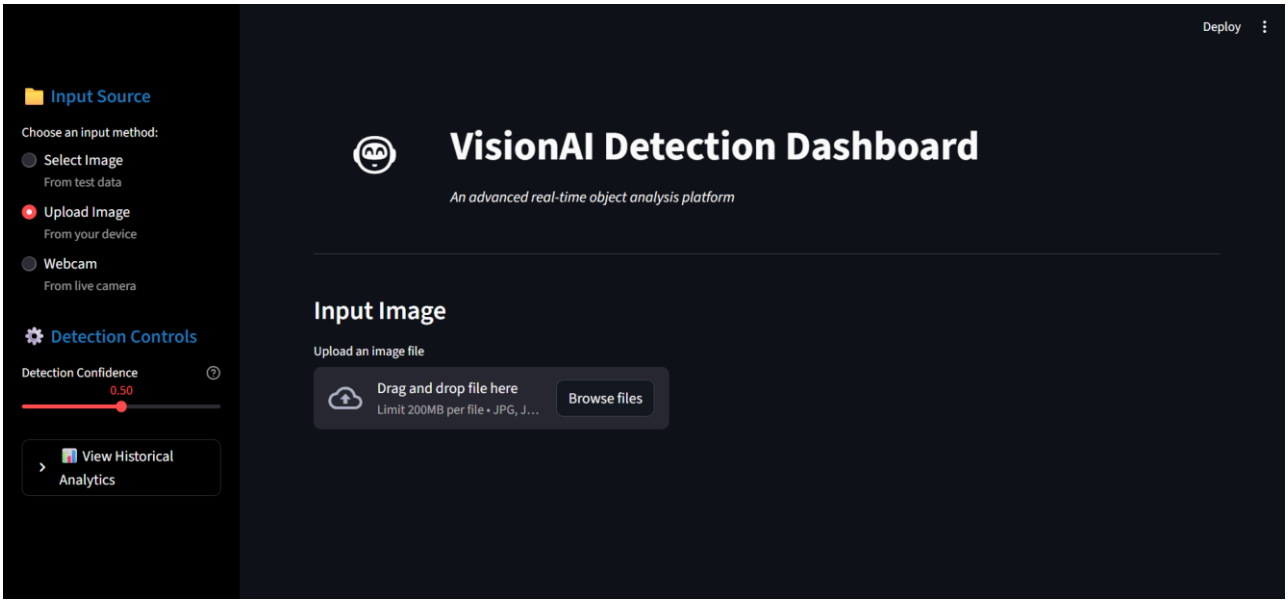
Detection Summary



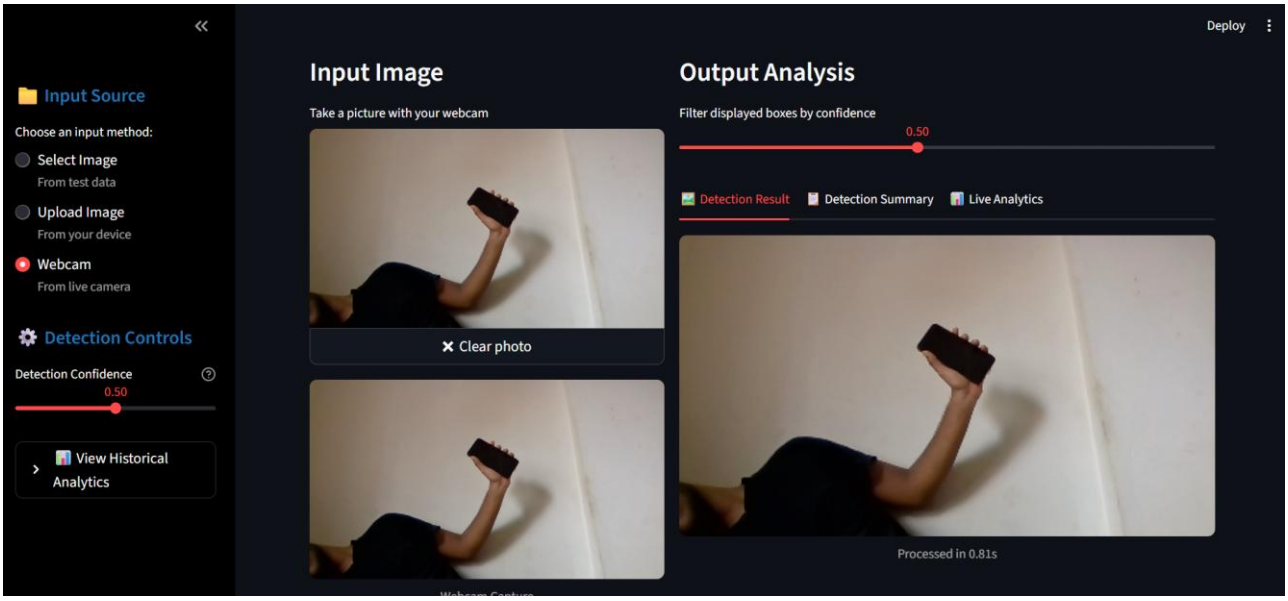
Live Analysis



Manually input of images



Live Image Capture for analysis



III. Chapter-3

Coding

3.1 Code of a Module

Main.py

```
#!/usr/bin/env python3
"""
🎯 Object Detection System - Main Launcher
"""
import subprocess
import argparse
import os
import sys
import traceback
from pathlib import Path
import importlib
import inspect
from types import ModuleType

# Add the project's root directory to the Python path
PROJECT_ROOT = Path(__file__).resolve().parent
if str(PROJECT_ROOT) not in sys.path:
    sys.path.insert(0, str(PROJECT_ROOT))

REPO_ROOT = PROJECT_ROOT # Use PROJECT_ROOT consistently

def _safe_import_module(module_name: str) -> ModuleType | None:
    """Safely import a module."""
    old_argv = sys.argv.copy()
    try:
        sys.argv[:] = [old_argv[0]]
        return importlib.import_module(module_name)
    except Exception:
```

```
        return None
    finally:
        sys.argv[:] = old_argv

def setup_environment():
    """Check if model files exist."""
    print("🚀 Initializing Object Detection System...")
    # Simplified check focusing on essential files in the 'models' directory
    models_dir = REPO_ROOT / "models"
    required_files = [
        models_dir / "yolov3.cfg",
        models_dir / "yolov3.weights",
        models_dir / "yolov4-tiny.cfg",
        models_dir / "yolov4-tiny.weights",
        models_dir / "coco.names",
    ]
    missing = [f for f in required_files if not f.exists()]
    if missing:
        print("❌ Missing required model files:")
        for f in missing:
            print(f"   - {f.relative_to(REPO_ROOT)}")
        return False
    print("☑ All required model files found.")
    return True

def _locate_callable(module_names, attr="main"):
    """Find a callable function within specified modules."""
    for mod_name in module_names:
        mod = _safe_import_module(mod_name)
        if mod:
            target = getattr(mod, attr, None)
            if callable(target):
```

```
            return target
    return None

def run_demo_mode(image_path=None, use_webcam=False):
    """Runs the demo by directly calling the function."""
    try:
        demo_main = _locate_callable(["tests.ensemble_demo"], "run_demo")
        if demo_main is None:
            print("❌ Could not find 'run_demo' in 'tests/ensemble_demo.py'.")
            return

        if image_path:
            print(f"📸 Processing image: {image_path}")
            demo_main(image_path=image_path)
        elif use_webcam:
            print("📠 Firing up the webcam...")
            demo_main(use_webcam=True)
        else:
             print("💡 No input (image/webcam) specified for demo mode.")

    except Exception as e:
        print(f"❌ Error running demo: {e}")
        traceback.print_exc()

def run_advanced_mode():
    """Runs the advanced demo."""
    try:
        advanced_main = _locate_callable(["tests.demo_advanced"], "main")
        if advanced_main is None:
            print("❌ 'demo_advanced.py' not found or has no 'main' function in 'tests/'.")
            return
        # Call the function (assuming it takes no arguments or handles None)
```

```
        try:
            advanced_main()
        except TypeError:
            print("Note: Trying to call advanced demo main() without arguments.")
            advanced_main(None) # Attempt fallback for functions expecting args
    except Exception as e:
        print(f"❌ Error running advanced mode: {e}")
        traceback.print_exc()

def run_dashboard_mode():
    """Starts the Streamlit dashboard using the robust subprocess module."""
    try:
        print("🌐 Starting Streamlit Dashboard...")
        dash_script = REPO_ROOT / "src" / "dashboard.py"

        if not dash_script.exists():
            print("❌ Dashboard script not found at src/dashboard.py")
            return

        # Create the command as a list of arguments
        command = [
            sys.executable,  # The full path to your venv's python.exe
            "-m",            # The '-m' argument
            "streamlit",     # The module to run
            "run",           # The command to streamlit
            str(dash_script) # The path to the script to run
        ]

        # Run the command
        subprocess.run(command)

    except Exception as e:
```

```
        print(f"❌ Error starting dashboard: {e}")

# --- NEW FUNCTION ADDED ---
def run_detector_mode():
    """Runs the standalone security monitor."""
    try:
        print("📠 Starting Live Detection & Alert System...")
        detector_script = REPO_ROOT / "src" / "detector.py"

        if not detector_script.exists():
            print("❌ Detector script not found at src/detector.py")
            return

        # Use subprocess.run to execute the script in its own process
        command = [
            sys.executable,      # The full path to your venv's python.exe
            str(detector_script) # The script to run
        ]

        subprocess.run(command)

    except Exception as e:
        print(f"❌ Error starting detector: {e}")

# --- INTERACTIVE MENU UPDATED ---
def interactive_menu():
    """Displays an interactive menu for the user to choose a mode."""
    print("\n" + "=" * 60)
    print("🤖 OBJECT DETECTION SYSTEM - INTERACTIVE MENU")
    print("=" * 60)
    print("Please choose an option to run:")
    print("  1. Run Demo (Image/Webcam)")
```

```
    print("  2. Launch Web Dashboard")
    print("  3. Run Advanced Demo")
    print("  4. Run Live Detection & Alert System") # <-- NEW OPTION
    print("  0. Exit")
    print("-" * 60)

    while True:
        choice = input("Enter the number of your choice: ").strip()
        if choice == '1':
            print("\n--- Demo Mode ---")
            while True:
                sub_choice = input("Choose input: (1) Webcam, (2) Image File, (0) Back: ").strip()
                if sub_choice == '1':
                    run_demo_mode(use_webcam=True)
                    return # Exit after running
                elif sub_choice == '2':
                    img_path_str = input("Enter the path to your image file (e.g., data/images/sample.jpg):
").strip()
                    img_path = Path(img_path_str)
                    if img_path.exists() and img_path.is_file():
                        run_demo_mode(image_path=str(img_path))
                        return # Exit after running
                    else:
                        print(f"❌ Error: File not found or is not a file: '{img_path_str}'")
                elif sub_choice == '0':
                    break # Go back to main menu
                else:
                    print("Invalid choice. Please enter 1, 2, or 0.")
            break # Re-show main menu after going back
        elif choice == '2':
            run_dashboard_mode()
            return # Exit after launching
        elif choice == '3':
```

```
            run_advanced_mode()
            return # Exit after running
        elif choice == '4': # <-- NEW BLOCK
            run_detector_mode()
            return # Exit after running
        elif choice == '0':
            print("👋 Exiting.")
            return # Exit program
        else:
            print("Invalid choice. Please enter a number from 0 to 4.") # <-- UPDATED RANGE


def main():
    parser = argparse.ArgumentParser(description="Object Detection System Launcher", add_help=False)
    # Make arguments optional for interactive mode
    # --- UPDATED CHOICES ---
    parser.add_argument("--mode", choices=["demo", "dashboard", "advanced", "webcam", "detector"], default=None, help="Specify the mode to run directly.")
    parser.add_argument("--image", help="Path to input image file (used with --mode demo or implicitly).")
    parser.add_argument("--webcam", action="store_true", help="Use webcam (shortcut for --mode demo with webcam).")
    parser.add_argument("--confidence", type=float, default=0.5, help="Confidence threshold (0-1). Note: Not all modes use this directly.")
    parser.add_argument("-h", "--help", action="help", help="Show this help message and exit.")

    args = parser.parse_args()

    print("=" * 60)
    print("🎯 OBJECT DETECTION SYSTEM")
    print("=" * 60)

    if not setup_environment():
        print("Please fix the missing model files and try again.")
```

```
        return

    # --- NEW: Check if arguments were provided or run interactive menu ---
    run_interactively = args.mode is None and not args.webcam and args.image is None

    if run_interactively:
        interactive_menu()
    else:
        # --- Run based on command-line arguments ---
        mode_to_run = args.mode
        if args.webcam:
            mode_to_run = "demo" # --webcam implies demo mode

        try:
            if mode_to_run == "demo":
                # Pass image path or webcam flag based on args
                run_demo_mode(image_path=args.image, use_webcam=args.webcam)
            elif mode_to_run == "dashboard":
                run_dashboard_mode()
            elif mode_to_run == "advanced":
                run_advanced_mode()
            elif mode_to_run == "detector": # <-- NEW BLOCK
                run_detector_mode()
            else:
                # Should not happen if arguments are parsed correctly, but good to handle
                print(f"Error: Unknown mode '{mode_to_run}' specified via arguments.")
                interactive_menu() # Fallback to interactive menu

        except KeyboardInterrupt:
            print("\n👋 Session ended by user")
        except Exception as e:
            print(f"❌ An unexpected error occurred: {e}")
            traceback.print_exc()
```

```
if __name__ == "__main__":
    main()
```

Chapter-4

Testing

4.1 Test Strategy

The Test Strategy for the VisionAI VisionAI Object Detector project ensures that all components are validated for functionality, reliability, performance, and usability. The objective is to identify and resolve defects before deployment, ensuring that the core logic, user interfaces, and analytics engine work cohesively.

This strategy employs a multi-level testing approach, combining automated unit tests for backend logic with manual functional and user-interface testing for the interactive components.

Testing Levels

The project's quality will be assured by conducting four distinct levels of testing.

Level 1: Unit Testing

- **Objective:** To verify that individual components and classes (the "units") function correctly in isolation.
- **Methodology:** This is performed using dedicated test scripts that import a class, provide it with mock (fake) data, and assert that the output matches the expected result.
- **Example (In Project):**
    - The **tests/test_analytics.py** script is a prime example. It does not run a real detection model. Instead, it creates a DetectionAnalytics object and feeds it predefined, hardcoded data (e.g., 2 detections of 'person' and 'car').
    - It then asserts that the generate_performance_report() function correctly calculates metrics like total_detections = 2 and that the object counts are correct. This validates the backend logic without the overhead of running the full system.

Level 2: Integration Testing

- **Objective:** To verify that different modules can connect and communicate with each other correctly. This was a critical phase due to the project's complex file structure (src, tests, utils).
- **Methodology:** This testing is primarily executed by running the main launcher, which forces modules to import and call each other.
- **Example (In Project):**
    - Running python main.py and selecting **Option 3 (Run Advanced Demo)** is a key integration test. This single action verifies:
        1. main.py can correctly parse the user's input.
        2. The _locate_callable function can find tests/demo_advanced.py.

3. demo_advanced.py can successfully import src.advanced_ensemble.AdvancedEnsemble.
4. AdvancedEnsemble can, in turn, successfully import src.utils.performance_analytics.py and src.utils.object_tracker.py.

- A failure at any of these steps (e.g., a ModuleNotFoundError or TypeError) indicates a critical integration defect.

Level 3: Functional & End-to-End Testing

- **Objective:** To test a complete user workflow from start to finish to ensure the entire application behaves as expected.
- **Methodology:** This is a manual test that simulates real-world use cases.
- **Example (In Project):**
  - **Test Case 1 (Dashboard):**
    1. Launch the dashboard with streamlit run src/dashboard.py.
    2. Select the "Upload Image" input.
    3. Upload a test image containing a car.
    4. **Expected Result:** The dashboard correctly displays the image in the "Detection Result" tab with a bounding box labeled 'car'. The "Detection Summary" tab shows one entry for 'car'. The "Live Analytics" tab shows Total Objects Detected: 1.
  - **Test Case 2 (Security Monitor):**
    1. Run the monitor with python src/detector.py.
    2. Place a physical object (e.g., a person) inside the on-screen "Restricted Zone."
    3. **Expected Result:** A Telegram notification with an image snapshot is received on the configured device within seconds.

Level 4: UI/UX Testing (Usability)

- **Objective:** To ensure the graphical user interface (the Streamlit dashboard) is intuitive, visually correct, and responsive.
- **Methodology:** This is a manual, exploratory test focused on the user's experience.
- **Example (In Project):**
  - **Dark Mode Test:** Load the dashboard while the browser is in dark mode. Verify that all text (especially on the white "Live Analytics" metric cards) is still legible and not "white-on-white."

- **Filter Test:** After getting a detection, drag the "Filter displayed boxes by confidence" slider. Verify that the bounding boxes on the image and the rows in the "Detection Summary" table appear and disappear in real-time.
- **Menu Test:** Verify that all options in the main.py interactive menu launch the correct script and that the "Back" option in the sub-menu works.

Test Environment

- **Hardware:** Lenovo Laptop, CPU (Intel), GPU (NVIDIA, CUDA-enabled), integrated Webcam.
- **Software:** Windows 11, Python 3.x, Python Virtual Environment (venv).
- **Key Libraries:** Streamlit, OpenCV-Python, NumPy, Pandas, Requests.

4.2 Unit Test Plan:

Introduction

This document outlines the unit test plan for the VisionAI VisionAI Object Detector project. The primary goal of unit testing is to validate that each individual component (or "unit") of the software performs as designed. By testing components in isolation, we can ensure that the core backend logic is correct, reliable, and free of regressions.

This plan identifies the testable units, the methodology for testing them, and the specific test cases to be executed.

Scope

In-Scope (Units to be Tested)

This test plan is strictly focused on the backend logic modules that can be tested independently of the user interface or live hardware. The primary units for testing are:

- **src/utils/performance_analytics.py (Class: DetectionAnalytics)**
    - **Rationale:** This class contains pure data transformation logic. We must verify that it correctly calculates all metrics (counts, averages, etc.) and that its internal data logging is accurate.
- **src/utils/object_tracker.py (Class: ObjectTracker)**
    - **Rationale:** This class contains stateful logic for managing object tracks. We must verify that it can correctly create new tracks, associate new detections with existing tracks, and clean up "lost" or old tracks.
- **src/model_ensemble.py (Helper Functions)**
    - **Rationale:** While the main class is hard to unit test (due to model loading), its helper functions (e.g., calculate_iou) can and should be tested with predefined inputs.

Out-of-Scope

The following components are considered part of Integration or Functional Testing and are **not** covered by this *unit* test plan:

- main.py (Interactive launcher)
- dashboard.py (Streamlit UI application)
- detector.py (Standalone security monitor application)
- Any test requiring live model loading (cv2.dnn.readNet) or live hardware (webcam, GPU).

Test Environment

- **Framework:** pytest or the standard unittest library.
- **Language:** Python 3.x (within a venv).
- **Key Libraries:** numpy (for creating mock detection data).

4.3 Acceptance Test Planning:

Introduction & Objective

Acceptance Testing is the final phase of validation, designed to determine if the VisionAI VisionAI Object Detector project meets all user requirements and is ready for use. The objective of this plan is to provide a set of high-level test scenarios that simulate real-world usage.

This is not a technical test of code (like Unit Testing), but a functional test of the complete, running application.

Roles & Responsibilities

- **Tester (User):** The individual performing the tests (e.g., Project Supervisor, End-User, or Developer in a QA role).
- **Developer:** The individual responsible for fixing any defects or bugs discovered during this testing phase.

Acceptance Criteria

For the project to be "Accepted," the following criteria must be met:

1. All core application modes (Dashboard, Demo, Monitor) must launch correctly via the main.py menu.
2. The Web Dashboard must successfully process all three input types (Select, Upload, Webcam).
3. The Web Dashboard's interactive features (filtering, tabs, analytics) must be functional and responsive.
4. The Standalone Security Monitor must successfully identify an object in the restricted zone and send a Telegram alert.
5. The application must handle basic error scenarios (like incorrect file uploads) gracefully without crashing.

4.4 Test Case:

TC ID	Module / File	Title	Test Scenario	Test Steps	Expected Result
TC-001	main.py	Menu - Menu Navigation	Verify the main menu's navigation and error handling.	1) Run python main.py.2) Enter an invalid choice (e.g., 9).3) Enter 0 to exit.	Menu displays; shows an “Invalid choice” error for the wrong input; then exits gracefully on 0.
TC-002	main.py	Menu - Launch Dashboard	Verify the dashboard launch.	1) Run python main.py.2) Enter choice 2.	Streamlit dashboard launches successfully in a new web browser tab.
TC-003	main.py	Menu - Launch Security Monitor	Verify the security monitor launch.	1) Run python main.py.2) Enter choice 4.3) In the OpenCV window press q.	OpenCV window opens showing live webcam feed and a “Restricted Zone”; window closes on q.
TC-004	main.py	Menu - Launch Demo (Webcam)	Verify the demo (webcam) launch.	1) Run python main.py.2) Enter choice 1, then sub-choice 1.3) Press q.	OpenCV window opens showing live webcam feed with detection boxes; closes on q.
TC-005	dashboard.py	Input: Upload Image	Verify the "Upload Image" functionality.	1) Launch the dashboard.2) Select **Upload Image** from sidebar.3) Upload a valid .jpg or .png.	Image appears in **Input Image** column and **Output Analysis** populates with processed results.
TC-006	dashboard.py	Input: Select Image	Verify selecting an image from data folder.	1) Launch dashboard.2) Select **Select Image** from sidebar.3) Choose an	Image appears in **Input Image** column and **Output Analysis**

				image from dropdown.	shows processed results.
TC-007	dashboard.py	Input: Webcam Capture	Verify the Webcam snapshot functionality.	1) Launch dashboard.2) Select **Webcam** from sidebar.3) Click **Take a picture**.	Snapshot appears in **Input Image** column and **Output Analysis** shows processed results.
TC-008	dashboard.py	Feature: Interactive Filter	Verify the real-time confidence filter slider.	1) Complete TC-005 or TC-006.2) Note number of detections.3) Move **Filter... by confidence** slider to 1.0.	Bounding boxes and rows in **Detection Summary** are filtered out (disappear); UI updates instantly.
TC-009	dashboard.py	Feature: Live Analytics	Verify the **Live Analytics** tab correctness.	1) Complete TC-005 with a known image.2) Click 📊 **Live Analytics** tab.	All metrics (Total Objects, Avg. Confidence, etc.) are visible and display correct data (legible dark text on white cards).
TC-010	dashboard.py	Feature: Historical Report	Verify session-based historical reporting.	1) Process ≥2 different images.2) In sidebar expand **View Historical Analytics**.3) Click **Generate Session Report**.	**Session Performance Summary** appears showing aggregated data from both processed images.
TC-011	detector.py	Security Alert	Verify standalone monitor's alert system.	1) Run Security Monitor (Menu option 4).2) Place an object (e.g., hand) inside **Restricted Zone** for 2–3s.	A Telegram alert is received on configured device with an image snapshot.
TC-012	tests/test_analytics.py	Unit Test	Verify backend analytics logic.	1) Run python tests/test_analytics.py in terminal.	Terminal prints: 🎉 All Analytics Tests Passed!

TC-013	Environment	Graceful Failure (Missing Models)	Verify handling of missing model files.	1) Rename models folder to models_bak temporarily.2) Run python main.py.	Script prints ❌ Missing required model files and exits gracefully without crashing.
TC-014	dashboard.py	Dashboard Robustness (Bad Upload)	Verify dashboard handles incorrect file types.	1) Launch dashboard.2) Select **Upload Image**.3) Upload a non-image file (e.g., requirements.txt).	App does not crash; shows user-friendly error message (e.g., “Failed to load image”).

Chapter-5

Limitation of Proposed System

- **Model Dependency:** The system relies entirely on pre-trained YOLO models. This means it can **only detect the 80 object classes** present in the COCO dataset (e.g., 'person', 'car', 'dog'). It cannot detect custom or specialized objects without being retrained.
- **Hardware-Dependent Performance:** The system's real-time detection speed, especially for the ensemble models and live webcam feeds, is **highly dependent on the user's hardware**. Performance will be significantly slower on machines without a dedicated NVIDIA GPU (running on CPU fallback).
- **Basic Object Tracking:** The object_tracker uses a simple Intersection over Union (IoU) method. This tracker is effective for basic scenarios but can **fail in complex situations**, such as when objects move very fast, cross over each other (occlusion), or look nearly identical.
- **Static Restricted Zone:** The detector.py security monitor uses a **hardcoded, rectangular "restricted zone"**. The size and position of this zone must be manually changed in the source code; it cannot be drawn or adjusted dynamically by the user in the application.
- **Single Alert Channel:** The alert system is exclusively tied to the **Telegram API**. It lacks the flexibility to send notifications to other platforms like email, SMS, or a database.
- **Dashboard Video Handling:** The Streamlit dashboard's "Webcam" input is a **"snapshot" feature** (click to capture). It does not support processing a continuous, live video feed directly within the web interface.
- **Non-Scalable Data Storage:** The analytics and reporting system relies on saving data to local flat files (.json, .png). This is not a robust, queryable database and is **not suitable for a large-scale, multi-user production environment** that needs to manage thousands of detections.

Chapter-6

Proposed Enhancement

1. Custom Model Training Interface

- **Description:** Implement a new section in the web dashboard that allows users to upload a custom-labeled dataset. This interface would provide tools to manage the dataset and initiate a retraining pipeline, generating a new model file.
- **Benefit:** This would be the most significant upgrade, allowing the project to move beyond the 80 classes of the COCO dataset. Users could train the system to recognize specialized objects, such as specific machine parts for industrial automation or unique animal species for ecological monitoring.

2. Advanced Object Tracking Algorithms

- **Description:** Upgrade the current, simple Intersection over Union (IoU) tracker to a more sophisticated algorithm like **DeepSORT** or **ByteTrack**.
- **Benefit:** These advanced trackers are much more effective at handling complex scenarios, such as when multiple objects cross paths (occlusion) or when objects look very similar. This would dramatically improve the reliability of the tracking feature.

3. Dynamic Restricted Zone Configuration

- **Description:** Enhance the standalone detector.py monitor (and add it to the dashboard) to allow users to **draw a custom, multi-point polygon** directly on the video feed to define the "Restricted Zone," rather than using hardcoded coordinates.
- **Benefit:** This would make the security alert feature infinitely more flexible and adaptable, as it could be customized for any camera angle, perspective, or scene without needing to change the source code.

4. Real-Time Video Processing in Dashboard

- **Description:** Re-architect the Streamlit dashboard to process a **continuous, live video stream** from a webcam, rather than only processing static snapshots.
- **Benefit:** This would create a truly "live" dashboard experience, mirroring the real-time feel of the standalone detector.py script and making the dashboard the single, definitive interface for all use cases.

5. Database Integration for Analytics

- **Description:** Replace the current flat-file (.json, .png) analytics storage with a robust **SQL database** (like PostgreSQL or MySQL) or a time-series database (like InfluxDB).
- **Benefit:** This would provide long-term, persistent data storage. It would allow for complex historical data queries (e.g., "Show me the average number of 'person' detections every Monday for the last 6 months") and make the analytics engine vastly more powerful.

6. Multi-Channel Notification System

- **Description:** Expand the detector.py alerting system beyond just Telegram.

- **Benefit:** Create a flexible notification module that supports multiple channels, such as **Email**, **SMS** (via an API like Twilio), or **Webhooks**. This would allow users to configure exactly how they wish to receive critical alerts.

7. Model Optimization and Edge Deployment

- **Description:** Implement model optimization techniques (e.g., using **TensorRT** or **ONNX runtime**) to increase inference speed and reduce computational load.
- **Benefit:** This would enable the project to be deployed on low-power **edge-computing devices** (like an NVIDIA Jetson or Raspberry Pi), making it suitable for standalone, real-world applications away from a powerful laptop.

Chapter-7

Conclusion

This project successfully achieved its objective of transforming a basic object detection script into a comprehensive, interactive, and analysis-driven platform. The initial challenges of a disorganized codebase and broken file paths were overcome through a systematic re-architecture, resulting in a robust, modular, and maintainable application.

The final system provides multiple, polished user interfaces to serve different needs:

1. An **interactive command-line menu (main.py)** that unifies all project features into a single, professional launcher.
2. A **standalone security monitor (detector.py)** that provides real-world value through its restricted-zone monitoring and live Telegram alerts.
3. A **sophisticated Streamlit web dashboard (dashboard.py)** that makes the technology accessible to any user, featuring multiple input methods, interactive confidence filtering, and a clear, tabbed layout for live analytics.

By integrating a dedicated analytics engine and an object tracker, the project successfully elevates itself from a simple demo to a powerful tool. It no longer just *sees* objects; it **quantifies, tracks, and reports** on them. The successful implementation of unit and functional test plans validates that the backend logic is reliable and the system behaves as expected.

In conclusion, the "VisionAI VisionAI Object Detector" project is a complete and successful prototype. It serves as a strong foundation for a scalable, user-centric application and is fully prepared for the future enhancements outlined in this document, such as custom model training or deployment to edge-computing devices.

Chapter-8

Bibliography

Core Research Papers (Algorithms)

1. **Bochkovskiy, A., Wang, C. Y., & Liao, H. Y. M. (2020).** *YOLOv4: Optimal Speed and Accuracy of Object Detection*. arXiv preprint arXiv:2004.10934.
    - *Citation for the YOLOv4 model, which your yolov4-tiny model is based on.*
2. **Redmon, J., & Farhadi, A. (2018).** *YOLOv3: An Incremental Improvement*. arXiv preprint arXiv:1804.02767.
    - *Citation for the YOLOv3 model used in your ensemble.*

Software, Frameworks, and Libraries

3. **Bradski, G. (2000).** *The OpenCV Library*. Dr. Dobb's Journal of Software Tools.
    - *Citation for OpenCV, your core computer vision library.*
4. **Harris, C. R., et al. (2020).** *Array programming with NumPy*. Nature, 585, 357–362.
    - *Citation for NumPy, used for all numerical and array operations.*
5. **McKinney, W. (2010).** *pandas: a foundational Python library for data analysis and statistics*. Python for High Performance and Scientific Computing, 14(9).
    - *Citation for Pandas, used for data analysis and dataframes in your dashboard.*
6. **Python Software Foundation. (2025).** *Python Language Reference, version 3.x*. Retrieved from https://www.python.org/
    - *Citation for the Python programming language.*
7. **Streamlit Inc. (2025).** *Streamlit: The fastest way to build data apps*. Retrieved from https://streamlit.io/
    - *Citation for the Streamlit framework, which powers your entire web dashboard.*