BinWise – Project Documentation

BinWise is a full-stack recycling management platform designed to encourage proper waste disposal through education, AI automation, and a reward-based system. The platform allows users to scan waste items using Computer Vision, determine recyclability, estimate weight, calculate reward points and monetary value, and redeem rewards through scheduled pickups or in-center drop-offs.

Program: DEPI - Digital Egypt Pioneers
Track: AI & Data Science - Microsoft Machine Learning Engineer + React Frontend Web Development
Team members:

React Frontend:

- Nourseen Saeed Tantawy
- Rudaina Aly Hassan
- Rawda Mamdouh
- Nourhan Ashraf

Machine Learning:

- Rowan Kamal Fayad
- Ghada Elsayed Farag
- Ahmed Maher Abdallah
- Youssef Mohamed Mostafa

Table of Contents

I. System Architecture Overview
II. Frontend Documentation
III. User & Admin Flows
IV. MongoDB ERD
V. Backend Documentation
VI. AI Module Documentation
VII. AI Technical Implementation
VIII. AI Testing & Quality Assurance
IX. Deployment & Monitoring

I. System Architecture Overview

Frontend

- React.js
- Tailwind CSS
- React Leaflet (Map integration)
- React ChartJS 2 (Data visualization)

Backend

- Node.js
- Express.js
- Authentication (JWT)

Database

- MongoDB

AI Module

- Python 3.10
- YOLOv8 (Ultralytics)
- FastAPI & Uvicorn
- OpenCV, NumPy
- Docker containerization
- HuggingFace Spaces deployment

Architecture Type:

→ Full-Stack MERN Architecture with AI Microservice Integration

System Components:

The BinWise platform operates as a distributed system with three main layers:

1. **Frontend Layer:** User-facing React application handling UI/UX
2. **Backend Layer:** Node.js/Express API managing user data, authentication, and business logic
3. **AI Microservice:** FastAPI-based Computer Vision service for waste classification

---

II. Frontend Documentation

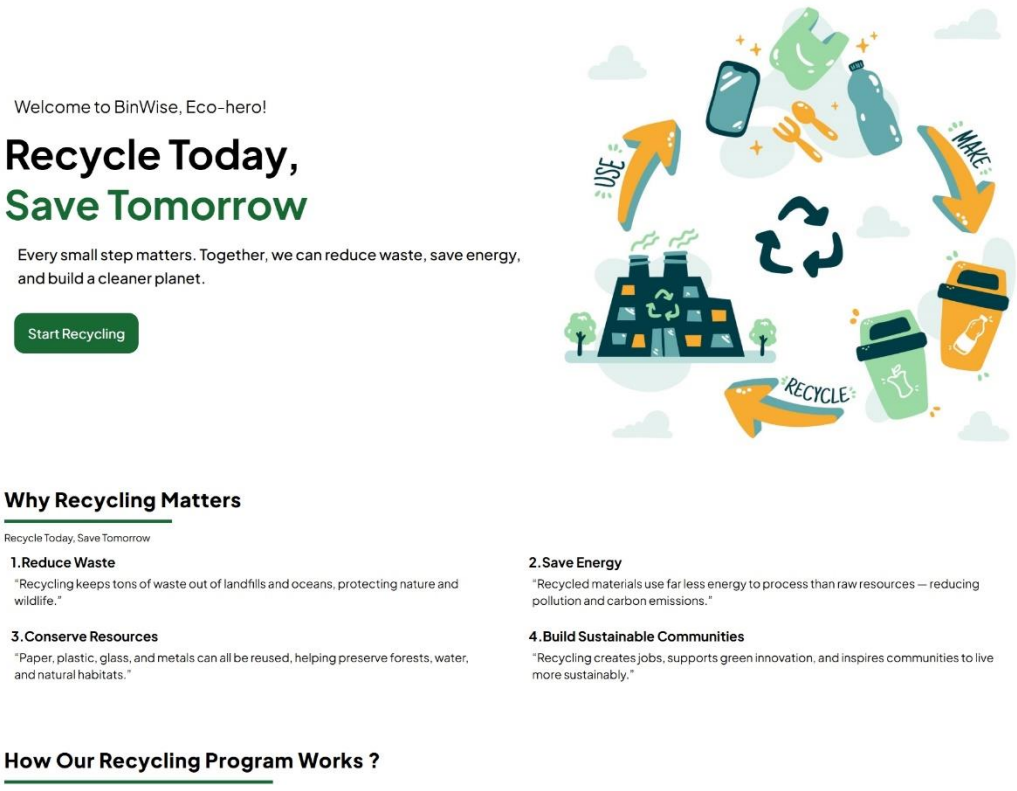
- **Page-Level Documentation**

1.Home Page

Purpose: Introduce the platform, explain its value, and guide users to start recycling

Key Sections:

- **Hero Section:** Primary CTA "Start Recycling" → Redirects to Scan page
- **Why Recycling Matters:** Environmental importance messaging
- **How It Works:** Step-by-step guide with feature buttons
- **Testimonials:** User feedback for credibility
- **Features Showcase:** AI scanning, rewards, pickups, awareness content
- **Footer:** Social links and contact information

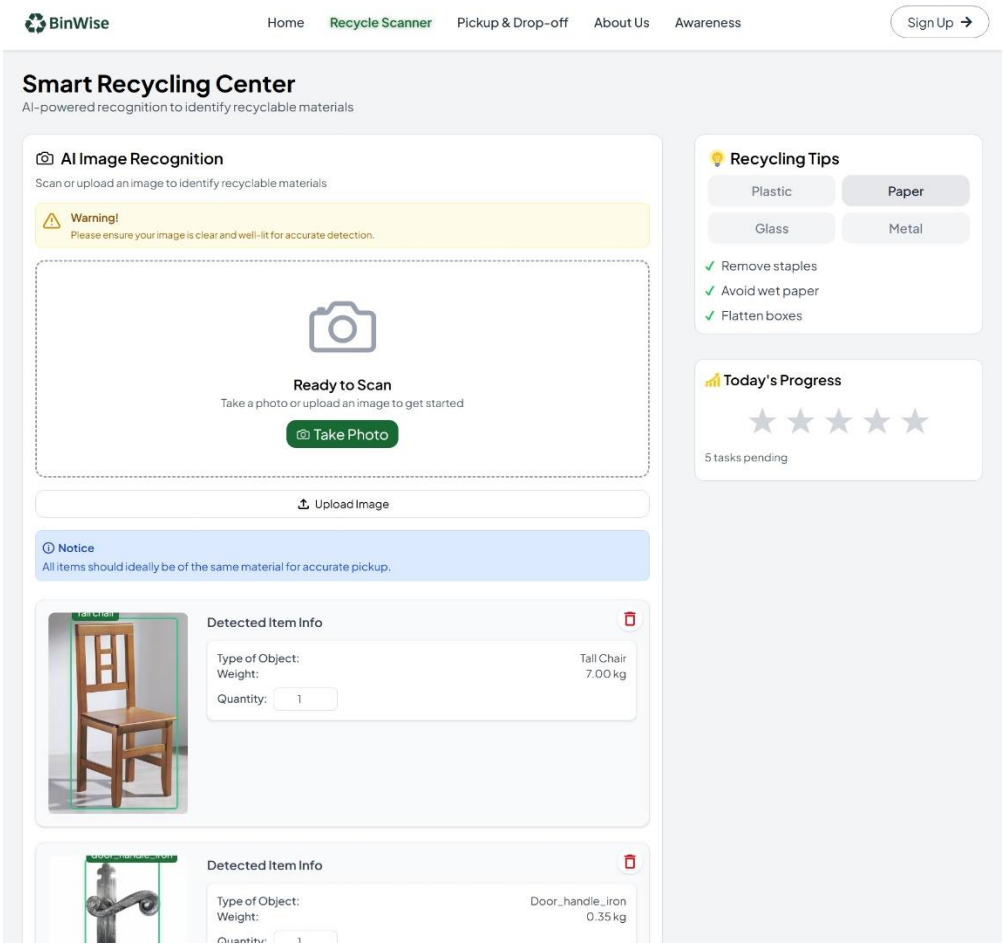


2.Recycle Scanner (AI Image Recognition) Page

Purpose: Allow users to scan or upload waste items for automatic classification

Key Features:

- **AI Scanner Interface:**
  - Take photo via React webcam
  - Upload existing image
  - Computer Vision returns: recyclable status, category, weight, points, monetary value
- **Schedule Pickup Button:** Direct navigation to Pickup Scheduling
- **Recycling Tips:** Practical advice section
- **Today's Progress Tracker:** 5-star visual system showing assigned pickups with encouraging message when goals are meet.

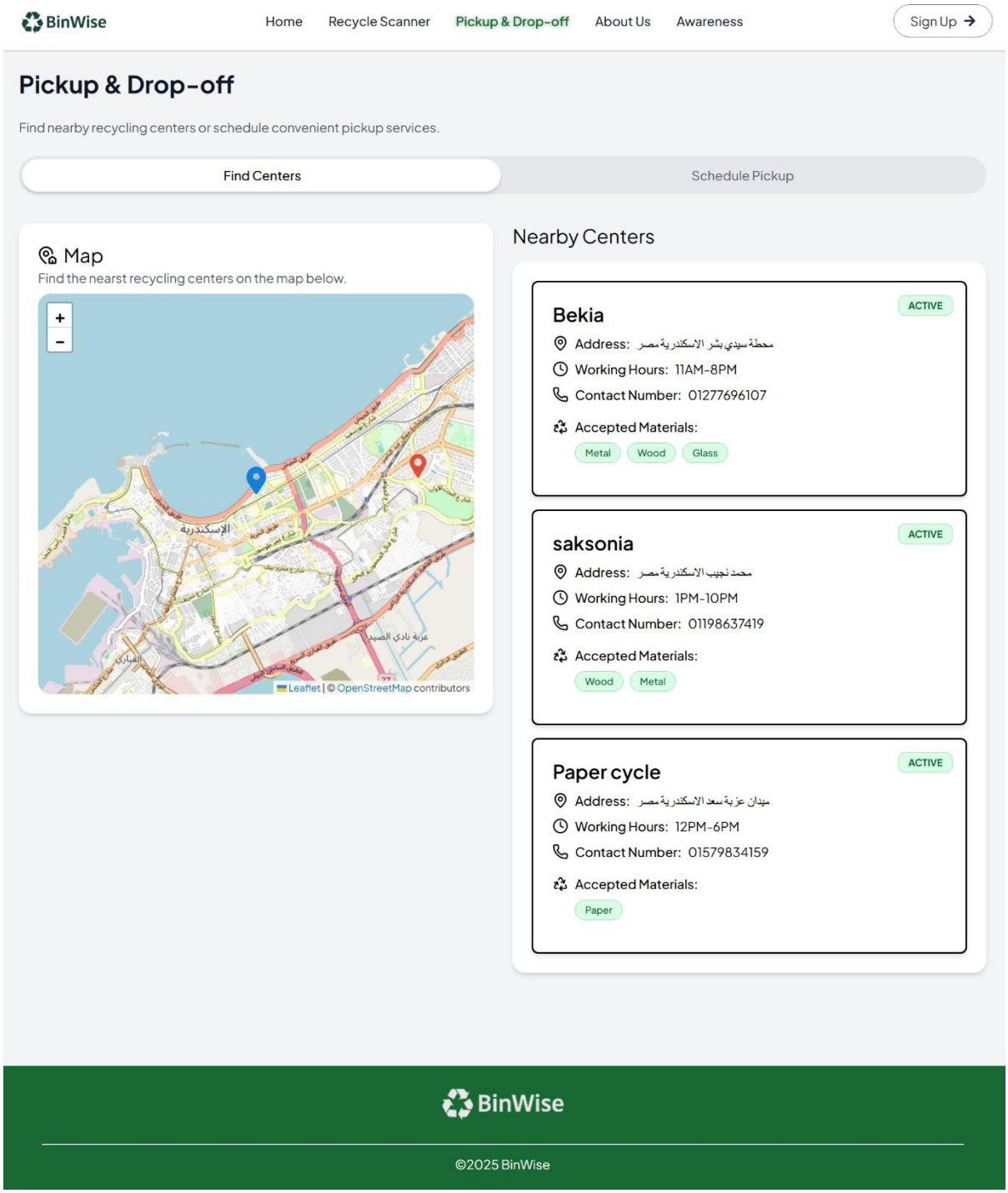


3.Pickup & Drop-Off Page

Purpose: Enable users to redeem recyclables and track pickup history

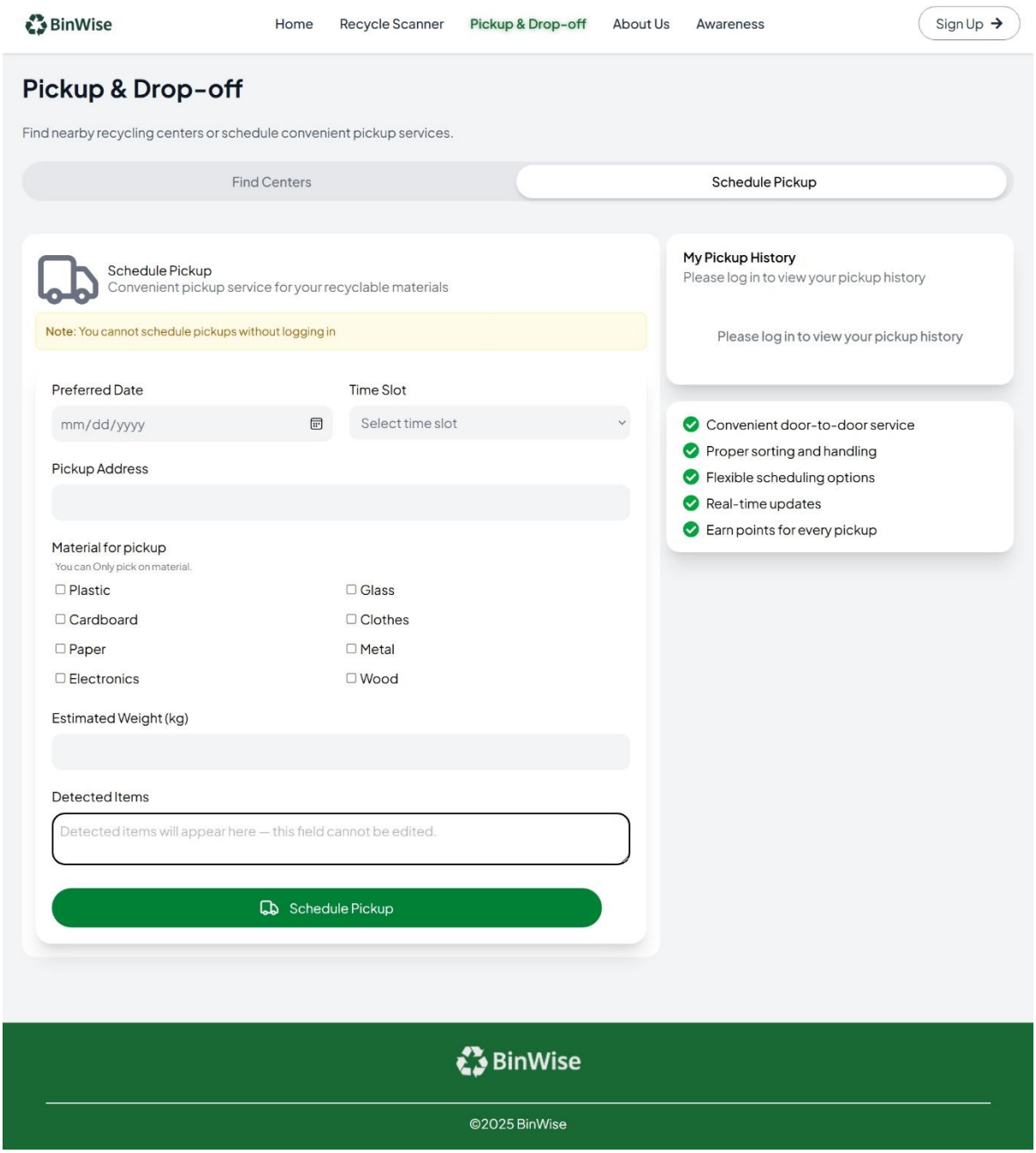
Sections:

1. **Centers Tab**
    - Interactive map (React Leaflet) showing recycling center locations
    - Sidebar with center details (address, contact, operating hours)



2. **Schedule Pickup Tab**
    - Pickup request form:
        - Preferred date & time picker

        - Item category selector
        - Weight input
        - Special instructions (optional)
    - "Schedule Pickup" button to submit request
3. **Pickup History**
    - Past pickups list
    - Current status tracking (Pending → Assigned → Completed)
    - Progress visualization
4. **Pickup Features Section**
    - Platform capabilities highlight: convenience, rewards, automated scheduling

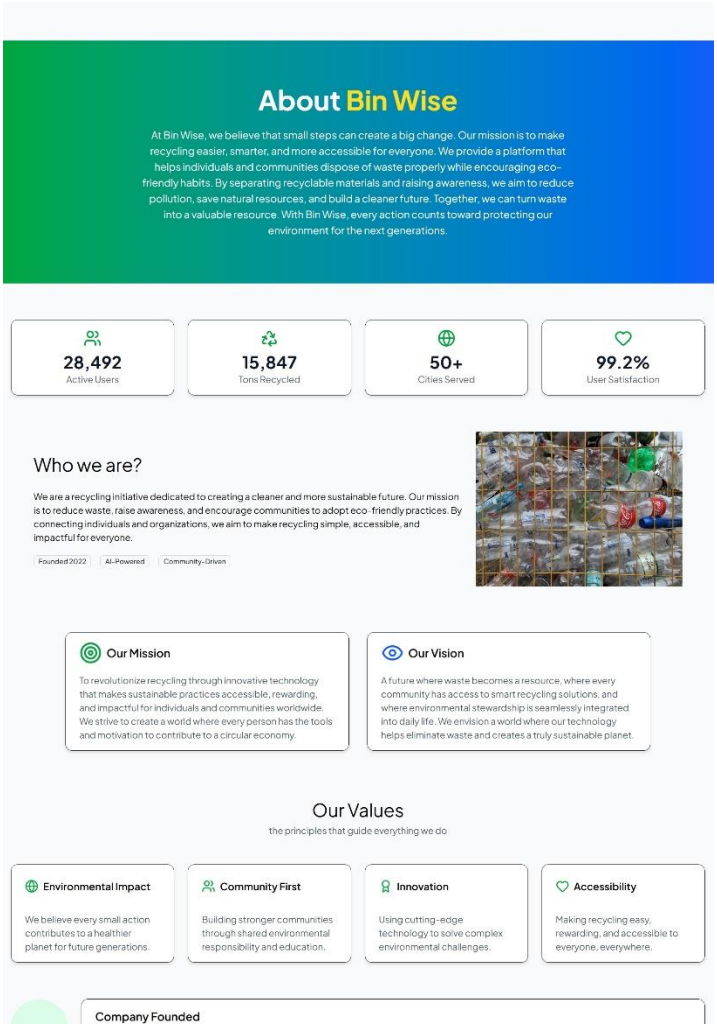


4.About Us Page

Purpose: Build trust and communicate platform vision

Content:

- **Vision & Mission Statements:** Platform goals and purpose
- **Company History:** Timeline of platform evolution
- **Team Profiles:** Key team member introductions
- **Call-to-Action:** Encourages users to start recycling (links to Scan page)



5.Awareness Page

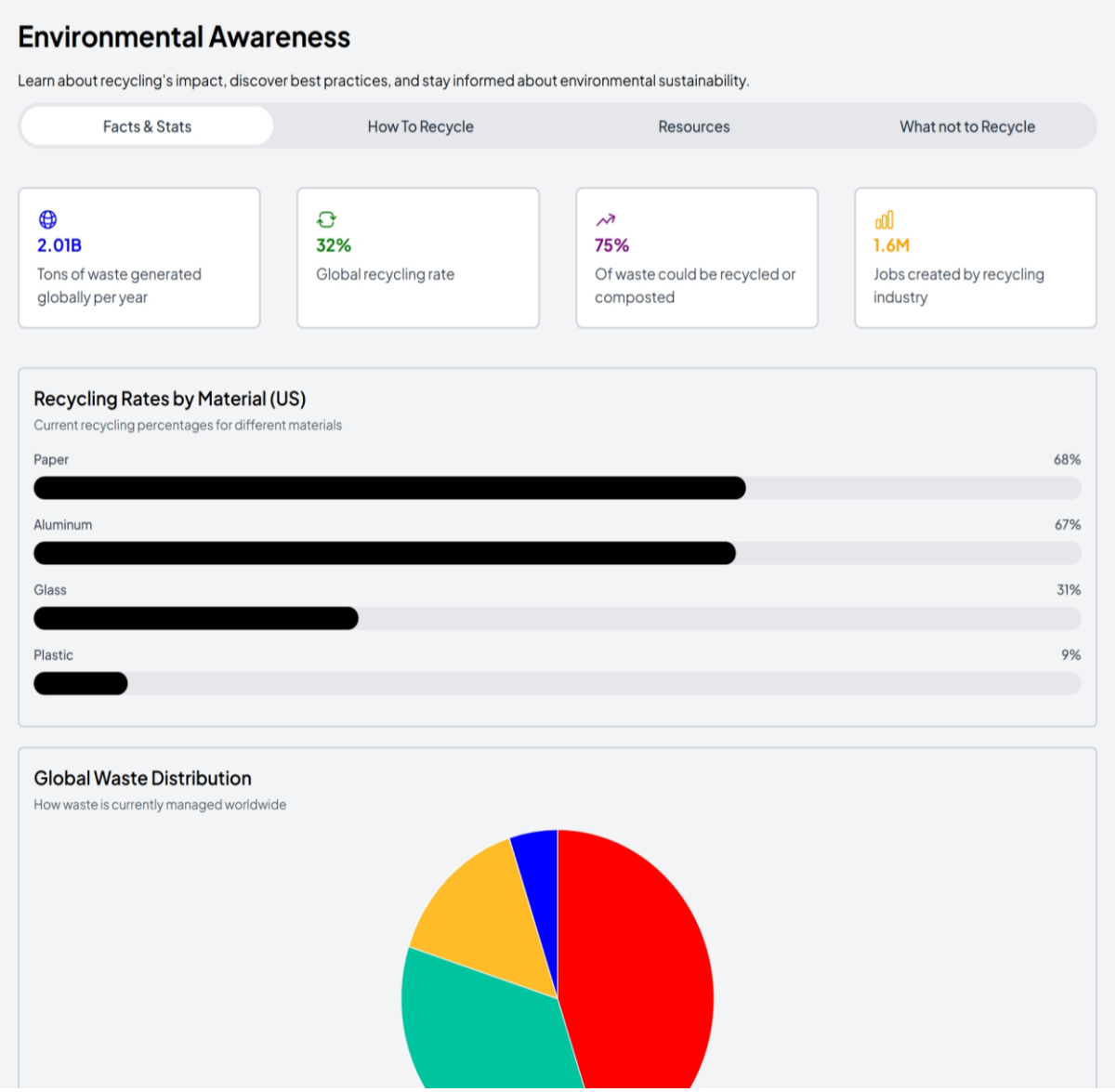
Purpose: Educate users about recycling and sustainability

Sections:

1. **Facts & Stats**

- Recycling rates by material (US data)
- Global waste distribution pie chart (React ChartJS 2)
-
- Waste reduction trends 2020-2024 line chart (React ChartJS 2)

2. **How to Recycle**
   - Material-specific recycling guidelines
   - Proper sorting instructions
3. **Resources**
   - Curated articles and guides
   - External sustainability resources
4. **What Not to Recycle**
   - Non-recyclable items list
   - Proper disposal methods for excluded items



6.Login & Signup Page

Purpose: Secure user authentication and account creation

Features:

Login Form:

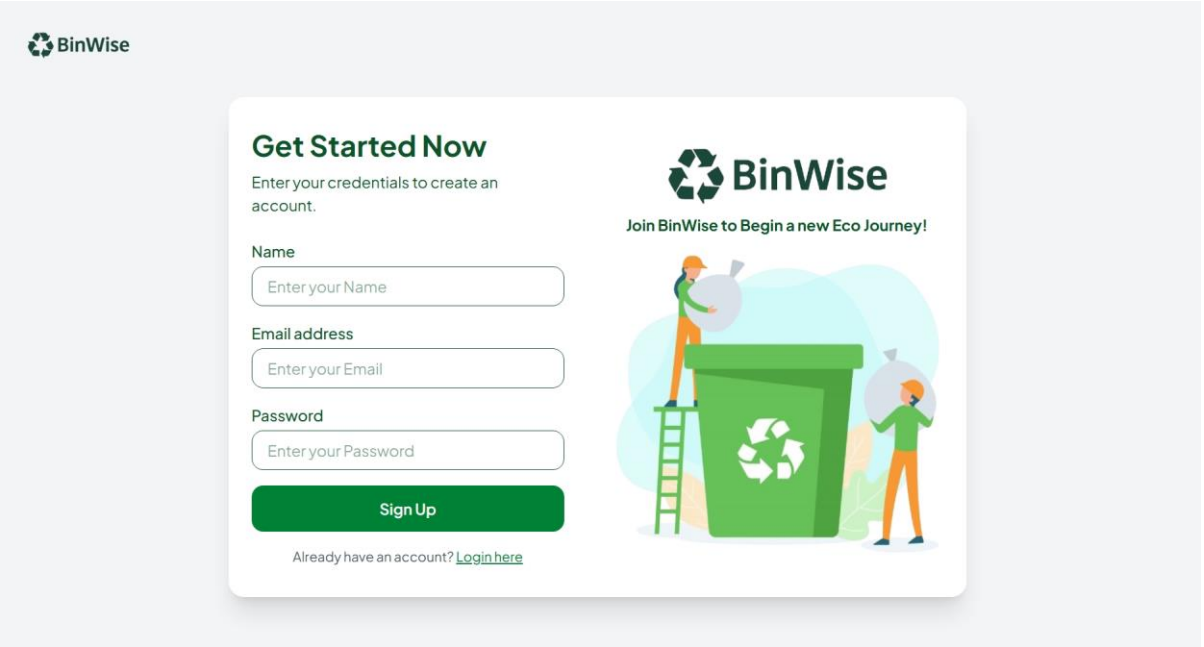
- Email and password fields
- Password reset option
- Input validation and error handling
- Redirects authenticated users to Home/Dashboard

Signup Form:

- Name, email, and password fields
- Validation and confirmation
- Default user role assignment

Security:

- Password hashing
- JWT authentication
- Role-based access controls (User vs Admin)



7.Email Verification Page

Purpose: Ensure account security through email verification

Features:

- OTP input form (code sent to user email)
- Resend OTP option
- Validation with feedback messages
- Successful verification redirects to Login

8.Profile Page

Purpose: Display user progress, activity, and achievements

Sections:

- **Overview Panel:**
  - Current level (e.g., Beginner, Intermediate, Expert)
  - Days recycled counter
  - Total points accumulated
  - Total monetary gains
- **Recent Activities Log:** Scans, pickups, and platform interactions
- **Rewards Summary:** Points and money earned over time
- **Achievements:** Milestone-based badges

9.Admin Dashboard Page

Purpose: Centralized interface for managing pickups and delivery agents

Features:

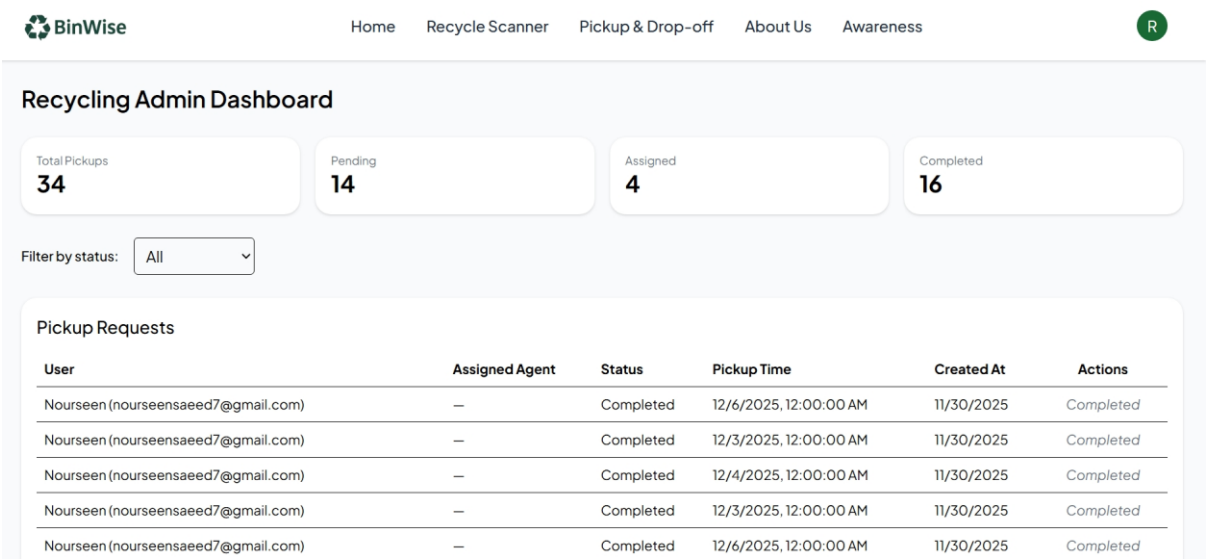
Stats Overview:

- Total pickups count
- Pending pickups
- Assigned pickups
- Completed pickups

Pickup Management Table:

- Filter by status (All, Pending, Assigned, Completed)

- Columns: User info, assigned agent, status, pickup time, creation date
- Action buttons:
  - "Assign" (for Pending pickups)
  - "Complete" (for Assigned pickups)



III. User & Admin Flows

- **User Flow:**

Start → Home Page

```
│
├─> Click "Sign up"
│     │
│     └─> Login / Signup Page
│           │
│           └─> Email Verification (OTP)
│                 │
│                 └─> Authenticated ✓
│
└─> Recycle Scanner (AI)
      │
      ├─> Take Photo / Upload Photo
      │     │
      │     └─> AI Processing
      │           │
      │           ├─> Category Detection
```

│ ├–> Recyclable Status
│ ├–> Weight Estimation
│ └–> Points & Money Calculation
│
├–> "Schedule Pickup" → Pickup Scheduling Page
│ │
│ ├–> Fill Form (Date, Time, Category, Weight, Instructions)
│ ├–> Submit Request
│ └–> View Pickup History / Status Tracking
│
├–> Go to Center (Map + Center Info)
│
├–> Profile Page
│ └–> View: Level, Points, Gains, Badges, Recent Activities
│
└–> Awareness Page
└–> Facts & Stats, How to Recycle, Resources, What Not to Recycle

Additional User Paths:

├–> Profile Page (View Level, Points, Gains, Badges, Recent Activities)

└–> Awareness Page (Facts & Stats, How to Recycle, Resources, What Not to Recycle)

- **Admin Flow**

Start → Admin Login

└–> Admin Dashboard

├–> View System Stats

│ ├–> Total Pickups

│ ├–> Pending Pickups

│ ├–> Assigned Pickups

│ └–> Completed Pickups

│

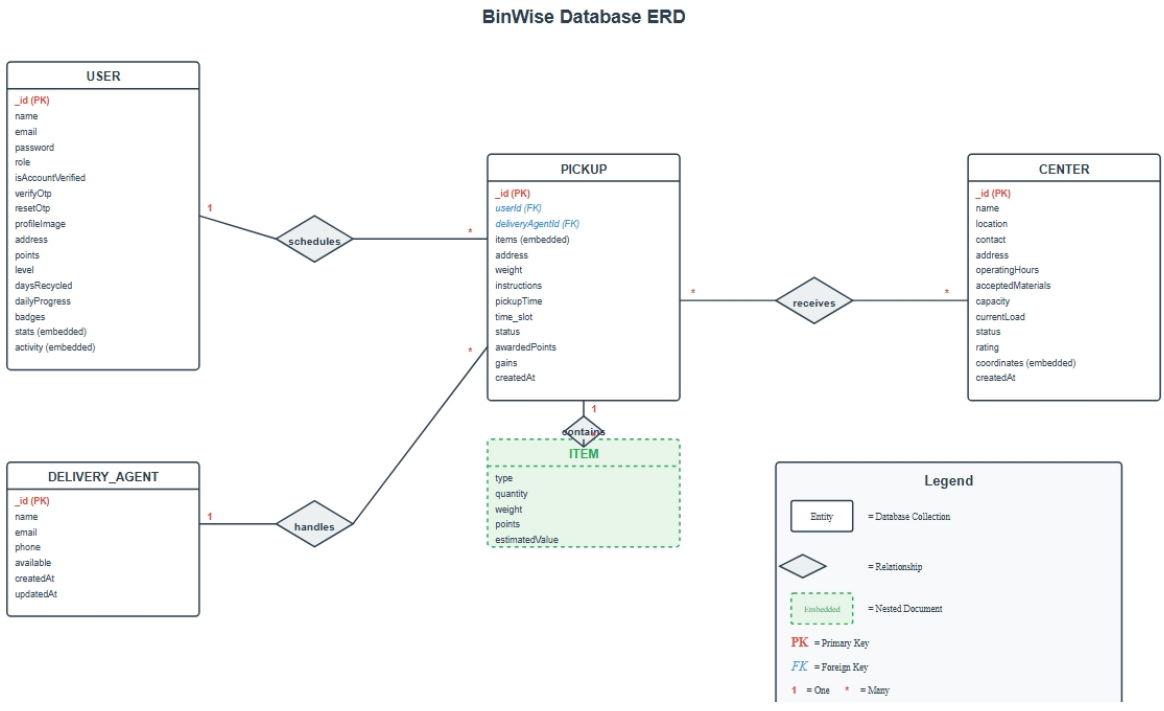
├–> View Pickup Requests Table

│

├–> Filter by Status

```
│      ├–> All
│      ├–> Pending
│      ├–> Assigned
│      └> Completed
│
├–> Assign Pickup to Agent
│      ├–> Select Agent from List
│      └> Schedule Pickup Date & Time
│
└> Mark Pickup as Completed
    └> Update Status in System
```

IV. DataBase ERD



V. Backend Documentation

❖ Architecture Pattern

- **Models:** Define MongoDB schemas and data structure using Mongoose
- **Controllers:** Handle business logic and process requests
- **Routes:** Define API endpoints and map to controllers
- **Middleware:** Process authentication, validation, and error handling

❖ Core Components

Component	Technology	Purpose
Server Framework	Node.js + Express.js	HTTP server and routing engine
Database	MongoDB + Mongoose	NoSQL database and object modeling
Authentication	JWT (jsonwebtoken)	Secure token-based authentication
Email Service	Nodemailer	OTP verification and notifications
AI Integration	Fetch API	Communication with AI microservice
File Upload	Multer	Image upload and processing
Password Security	bcryptjs	Password hashing and verification

❖ Authentication & Authorization

1. JWT Token Structure

Token Payload Contains:

- User ID (MongoDB ObjectId)
- Email address
- User role (user or admin)
- Issued at timestamp (iat)
- Expiration timestamp (exp)

Token Expiration: 7 days
Storage: Client stores token in cookies

2. Authentication Flow

Registration:

1. User submits name, email, password
2. Backend validates input and checks email uniqueness
3. Password is hashed using bcrypt
4. User document created in database
5. 6-digit OTP generated and sent to email
6. OTP expires after 10 minutes

Login:

1. User submits email and password
2. Backend finds user and verifies password
3. Checks if email is verified
4. Generates JWT token with user data
5. Returns token and user profile

Protected Routes:

- Middleware verifies JWT token from request header
- Extracts user ID from token payload
- Fetches user from database
- Attaches user object to request
- Proceeds to route handler if valid

3. **Admin Authorization:**

- Only users with role "admin" can access

❖ Error Handling

Standard Error Response Format

All errors return consistent JSON structure:

- `success`: false
- `message`: Human-readable error description

HTTP Status Codes

Code	Meaning	Usage
200	OK	Successful GET, PUT, DELETE requests
201	Created	Successful POST (resource created)
400	Bad Request	Validation errors, invalid input
401	Unauthorized	Missing or invalid authentication token
403	Forbidden	Insufficient permissions (not admin)
404	Not Found	Resource doesn't exist in database
409	Conflict	Duplicate resource (e.g., email exists)
500	Server Error	Internal server error, database issues

Error Types

Validation Errors:

- Invalid email format
- Password too short
- Required fields missing
- Invalid date format

Authentication Errors:

- Token expired or invalid
- User not found
- Incorrect password
- Email not verified

Authorization Errors:

- Admin access required
- Cannot modify other user's data

Business Logic Errors:

- Cannot cancel completed pickup
- Center at full capacity
- Invalid OTP code
- Agent not available

❖ Password Security Implementation

Hashing Strategy:

- Uses bcryptjs library
- Salt rounds: 10
- Passwords hashed before saving to database
- Plain text passwords never stored
- Password comparison done using bcrypt.compare()

❖ Database Strategy

Connection Features:

- Connection pooling for performance

- Automatic reconnection on failure
- Graceful shutdown handling
- Error logging and monitoring

Database Indexes:

I. **User Collection:**

- Unique index on email (fast lookup, prevent duplicates)
- Index on points (descending, for leaderboards)
- Index on createdAt (for sorting)

II. **Pickup Collection:**

- Index on userId (user's pickup history)
- Index on status (filtering)
- Index on pickupTime (scheduling queries)
- Compound index on userId + status

III. **Center Collection:**

- Geospatial index on coordinates (nearby search)
- Index on status (active centers only)

IV. **DeliveryAgent Collection:**

- Unique index on email
- Index on available (finding free agents)

❖ Request/Response Examples

Example 1: User Registration

Request:

- Method: POST
- Endpoint: `/api/auth/register`
- Content-Type: application/json
- Body: name, email, password

Success Response (201):

- success: true
- message: "User registered. Please verify email."
- data: userId

Error Response (400):

- success: false
- message: "Email already exists"

Example 2: Schedule Pickup

Request:

- Method: POST
- Endpoint: `/api/pickup/schedule`
- Authorization: Bearer token
- Body:
  - items: array (type, quantity, weight)
  - address: pickup address
  - weight: total weight in kg
  - pickupTime: ISO date string
  - time_slot: "morning", "afternoon", or "evening"
  - instructions: optional string

Success Response (201):

- success: true
- message: "Pickup scheduled successfully"
- data:
  - pickupId: MongoDB ObjectId
  - status: "pending"
  - pickupTime: scheduled date/time
  - estimatedPoints: calculated points

VI. AI Documentation

Project Planning & Management

1.1 Project Proposal: Overview, Objectives, and Scope

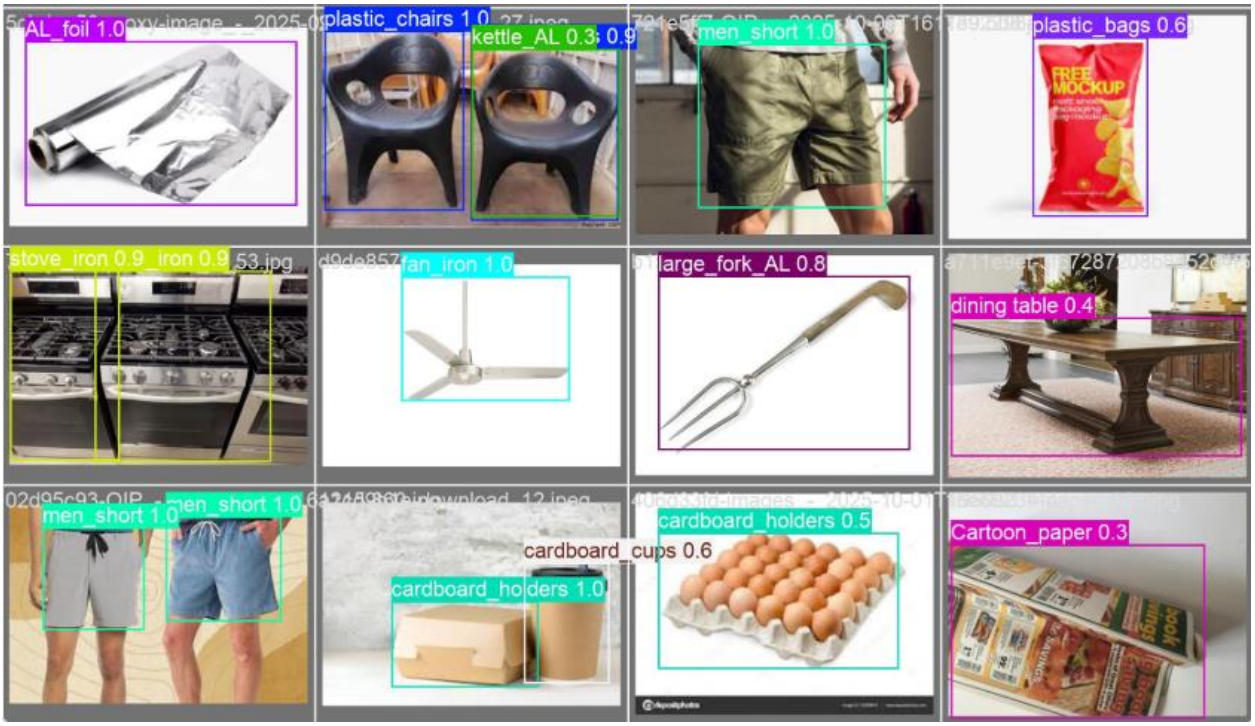
1.1.1 Project Title

BinWise – AI-Powered Waste Classification, Weight Estimation, and Reward Automation System

1.1.2 Project Overview

BinWise is a full-stack recycling management platform designed to encourage proper waste disposal through education, AI automation, and a reward-based system. The AI Module is the core predictive engine responsible for the real-time identification, classification, and quantification of recyclable waste.

The system processes upload user images, and the AI module determines the Material Type (1 of 7 categories), the Specific Object Label (1 of 55 classes), the Weight Estimation (using a lookup table), and the corresponding Reward Points. This information is delivered via a scalable FastAPI service deployed using Docker and HuggingFace Spaces.



1.1.3 Project Objectives

The AI team's primary objectives:

- **Data Preparation:** Develop a high-quality, manually collected and annotated dataset of 6,000+ images across 55 object classes.
- **Model Implementation:** Successfully transition from MobileNetV2 to YOLOv8 to achieve robust multi-object detection.
- **Weight Estimation:** Create a deterministic weight estimation system based on fixed object-based lookup tables.

Category	Avg/Median Price per kg	Calculation Method	Exact Points per kg	Rounded Points per kg (User)	Rounded Points After Approximation
Plastic	25 kg	Mean	166.66	167	25.05
Paper	8 kg	Mean	53.33	53	7.95
Carton	8 kg	Mean	53.33	53	7.95
Glass	3.5 kg	Mean	23.33	23	3.45
Clothes (Textile)	17.5 kg	Mean	116.66	117	17.55
Metals	43 kg	Median	286.66	287	43.05

- **API Design & Deployment:** Deploy the trained model within a documented FastAPI container, ensuring low-latency inference performance.
- **Performance:** Achieving specific Key Performance Indicators (KPIs) (response time, system uptime, user adoption rate) was a key objective, including an accuracy of 70%, a Precision of 0.65, and an API response time of < 2 seconds.

1.1.4 Project Scope

The scope of the AI team's work is strictly limited to the Computer Vision pipeline.

- Dataset creation and annotation, YOLOv8 model training, object category detection (55 classes), material classification (7 categories), weight estimation logic, reward calculation framework (formula design), deployment setup (FastAPI, Docker, HuggingFace), and the API schema design.
- **Handled by Full-Stack Team:** Frontend UI design, backend databases and user management, scheduling pickups, rewards wallet implementation, and admin panels.

1.1.5 Project Planning

Phase	Duration (Weeks)	Key Activities	Status
1: Data Acquisition & Preprocessing	1–3	Manual collection of 6,000 images; Annotation using Label Studio; Data augmentation strategy; Initial dataset split (Train/Validation/Test).	Complete
2: Model Selection & Training Setup	4–6	Finalize transition from MobileNetV2 to YOLOv8; Define the 55 object classes and 7 material categories; Set up training environment (Google Colab/Cloud GPU); Establish baseline hyperparameters (e.g., batch size, learning rate).	Complete
3: Core Model Training & Optimization	7–9	Execute YOLOv8 training for 50 epochs (14 hours); Apply mitigation strategies (oversampling, class weighting); Hyperparameter tuning; Final model selection based on 70% Accuracy.	Complete
4: Weight Logic & API Development	10–11	Integrate deterministic Weight Estimation Lookup Tables; Design the Reward Calculation Framework; Develop the FastAPI inference service (/detect/ endpoint); Implement lifespan model loading logic.	Complete
5: Containerization & Deployment	12–13	Create Docker container image (FastAPI, Python environment, YOLOv8 model); Test container locally; Deploy the service to HuggingFace	Complete

Phase	Duration (Weeks)	Key Activities	Status
		Spaces; Test the live API response time (Target < 2 seconds).	
6: Testing & Quality Assurance	14–15	Execute comprehensive Test Cases (Classification, Multi-object, Weight Consistency, Confidence Threshold); Final validation of KPIs (Precision 0.65, Recall 0.70); Conduct final integration tests with the Backend team's JSON parsing logic.	Complete
7: Documentation & Finalization	16	Finalize academic documentation; Prepare Project Presentation; Submit all code, models, and documentation to the Version Control Repository (GitHub).	Complete

1.2 Risk Assessment & Mitigation Plan

Risk Assessment & Mitigation Plan involves identifying risks and solutions.

- **Imbalanced Dataset**
    - *Cause:* Varying image counts across the 7 material categories and 55 object classes.
    - *Mitigation:* The team applied data augmentation techniques (oversampling of minority classes, and utilized class weighting within the YOLOv8 training configuration) to balance the model's focus.
- **Long Training Duration**
    - *Cause:* Training for 50 epochs took approximately 14 hours.
    - *Mitigation:* Optimized training pipelines and reliance on external cloud GPU providers (Google Colab) were used to reduce training time and leverage online resources efficiently.
- **Hardware Limitations**
    - *Cause:* Limited GPU resources affected the potential batch size and model complexity.
    - *Mitigation:* The team accepted a lower batch size and continuously monitored training stability, preparing to switch to lighter YOLO models (like YOLOv8n) if resource constraints became severe.
- **Multi-Object Failure**

- *Cause:* The previous model, MobileNetV2, was unable to detect multiple objects in a single image.
- *Mitigation:* This risk was directly mitigated by replacing MobileNetV2 with the superior YOLOv8 architecture, which natively supports multi-object detection and returns bounding boxes for all detected items.

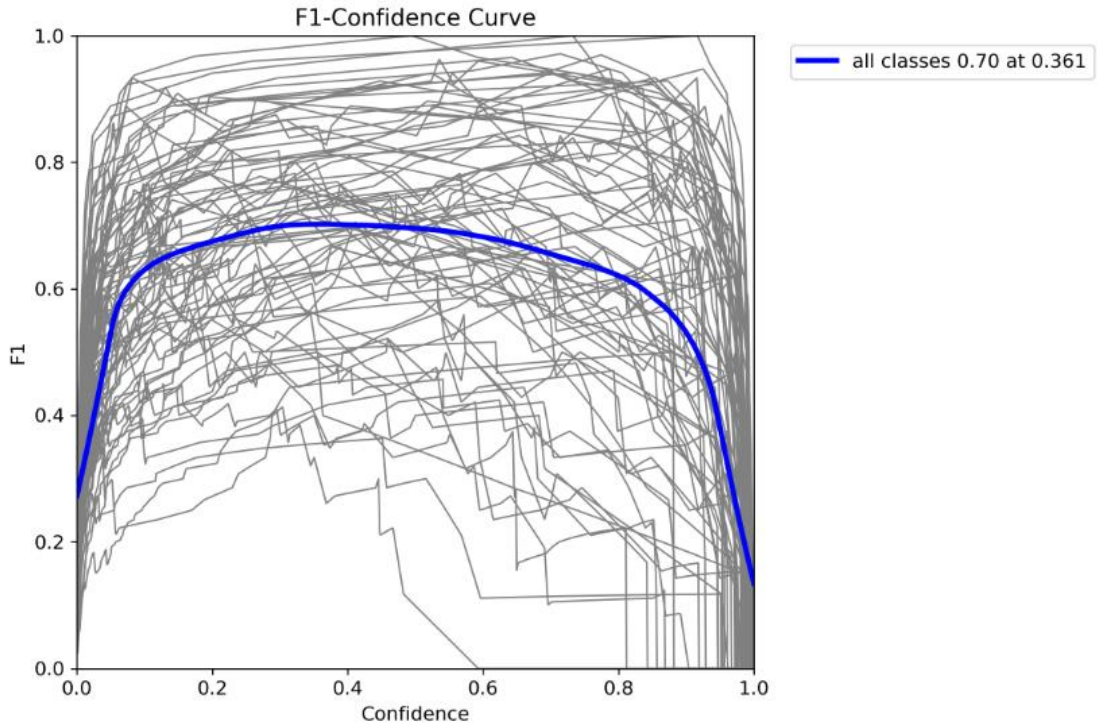
```
YAML_PATH = 'splitted/data.yaml'
MODEL_SIZE = 'yolov8n.pt'
EPOCHS = 50
PROJECT_NAME = 'yolov8_custom'
SAVE_PERIOD = 10
if torch.cuda.is_available():
  DEVICE = 0              # GPU 0
  BATCH_SIZE = 16
  IMG_SIZE = 640
  print("GPU Detected")
else:
  DEVICE = 'cpu'
  BATCH_SIZE = 4
  IMG_SIZE = 416
  print("CPU Mode")
# Verify YAML
if not os.path.exists(YAML_PATH):
  raise FileNotFoundError(f"YAML missing: {os.path.abspath(YAML_PATH)}")
# ESSENTIALS
if __name__ == '__main__':
  print(f"\n Training: {MODEL_SIZE} on {DEVICE}")
  print(f"📂 Dataset: {YAML_PATH} | Epochs: {EPOCHS}")
  print(f"⚙️  Batch: {BATCH_SIZE} | ImgSz: {IMG_SIZE}")
  #  Load Model
  model = YOLO(MODEL_SIZE)
  # TRAIN
  results = model.train(
```

```
    data=YAML_PATH,
    epochs=EPOCHS,
    imgsz=IMG_SIZE,
    batch=BATCH_SIZE,
    device=DEVICE,
    project=PROJECT_NAME,
    name='train',
    save_period=SAVE_PERIOD,
    plots=True,
    save_json=True
)
#  Validate
print("\n📊 Val Results:")
model.val()
# 4. Export
model.export(format='onnx')
print("\n DONE! Check: runs/detect/yolov8_custom/train/")
print(" Best: best.pt | Last: last.pt")
```

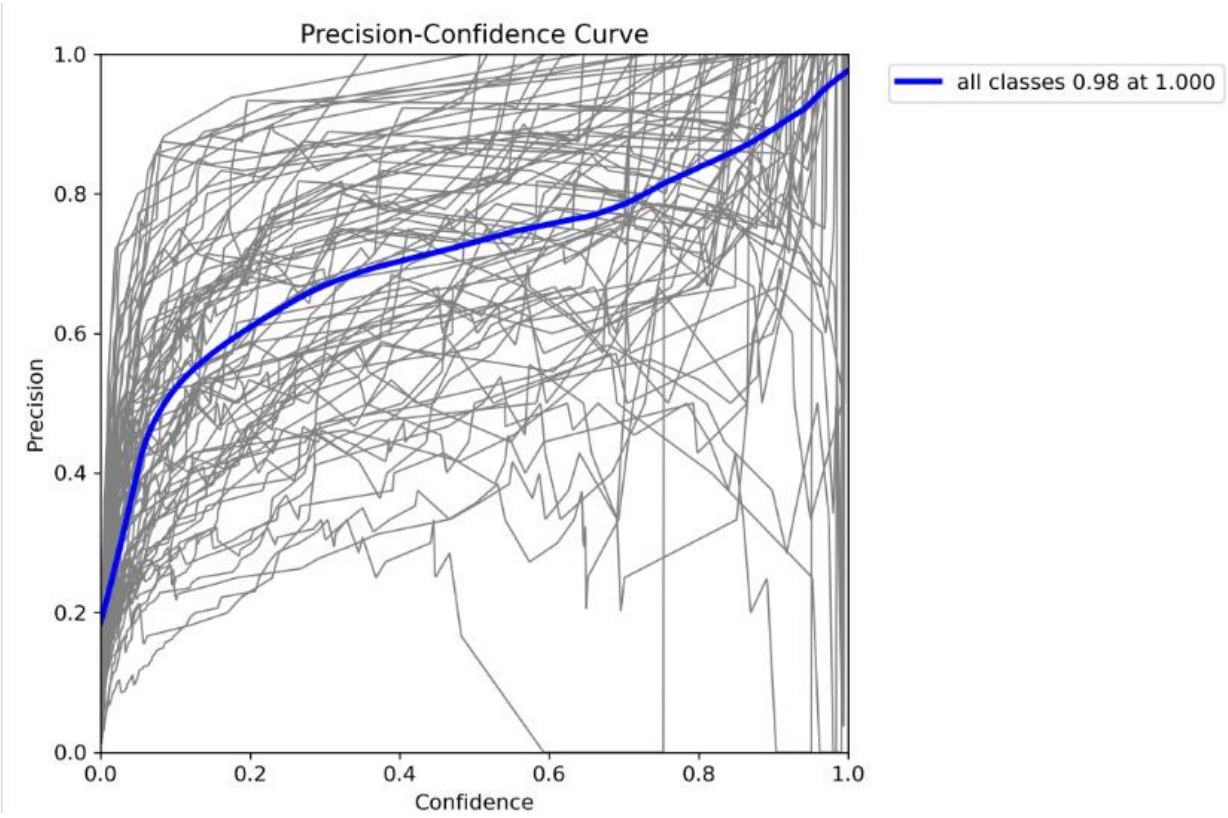
1.3 Key Performance Indicators (KPIs)

KPIs (Key Performance Indicators) are metrics for project success (response time, system uptime, user adoption rate). The following KPIs were set as measurable targets for the AI module's success:

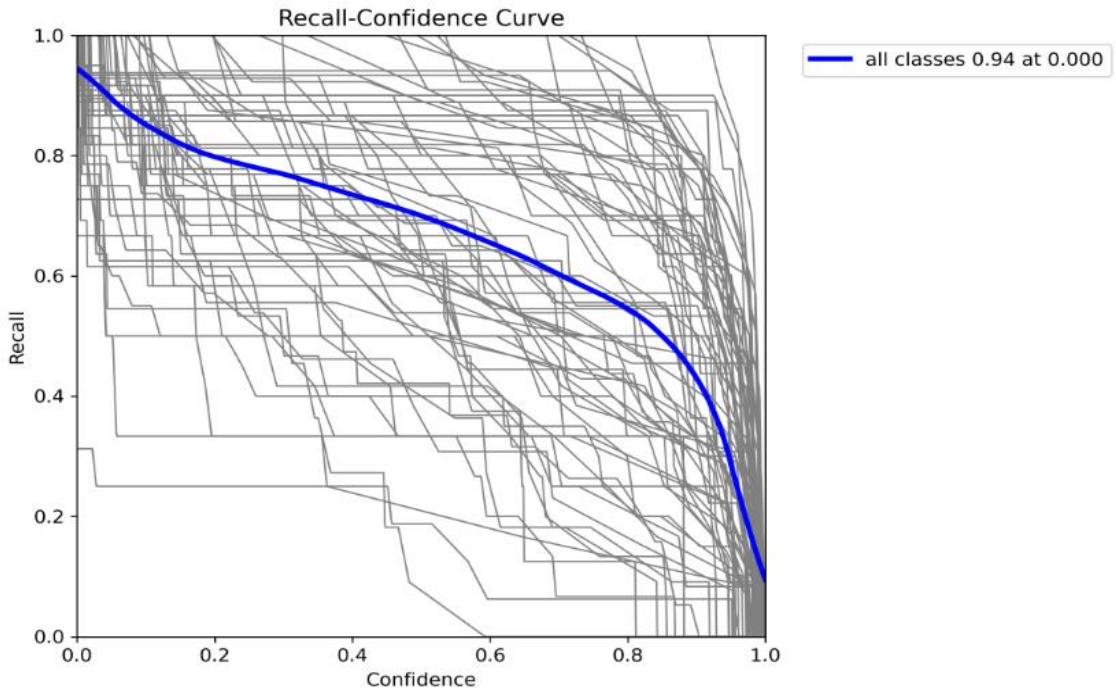
- **Model Accuracy:** The final achieved accuracy was 70%.



- **Precision:** The final achieved precision was 0.69.



- **Recall:** The final achieved recall was 0.74332.



- **API Response Time:** Target < 2 seconds.
- **Model Load Time:** Target < 6 seconds.
- **Detection Confidence:** A required threshold of 0.5 was implemented to filter unreliable predictions.

Requirements Gathering

The Requirements Gathering process involves Functional Requirements, Non-functional Requirements, Stakeholder Analysis, and User Stories & Use Cases.

2.1 Functional Requirements

Functional Requirements are a list of features and functionalities. The AI Module fulfill the following core functions:

- **Object Detection:** Detect the presence and location of objects within an image and output precise bounding boxes.
- **Classification:** Accurately classify detected objects into one of the 55 specific object labels.
- **Material Identification:** Infer the broad material type (Plastic, Metals) for each object into one of the 7 categories.
- **Multi-Object Handling:** Process images containing multiple distinct recyclable items and return a list of all successful detections.

- **Weight Estimation:** Calculate the estimated weight (in grams) based on the detected object label using the internal lookup table.
- **Reward Calculation:** Apply the predetermined formula to calculate reward points based on material type and estimated weight.
- **API Output:** Return all prediction results as a structured JSON response with the label, material, confidence score, bounding box, and estimated weight.

2.2 Non-Functional Requirements

Non-functional Requirements are performance, security, usability, and reliability criteria. The system must adhere to these constraints:

- **Performance:** The API must maintain a stable response time of under 2 seconds for inference to ensure a seamless user experience.
- **Reliability:** The model must maintain a minimum accuracy of 70% and handle concurrency during high user traffic via the deployment setup.
- **Scalability:** The deployment using Docker and HuggingFace Spaces must allow for easy scaling to handle increased API request load without manual intervention.

2.3 Use Case Description

User Stories & Use Cases are scenarios illustrating how users interact with the system. The primary use case involving the AI team's deliverable is the Waste Scanning and Classification scenario:

- **Actor:** User.
- **Scenario:** The user captures or uploads an image of waste items. The image is passed to the FastAPI API. The YOLOv8 model executes inference, and the prediction results (Label, Weight, Points) are returned to the backend for final processing and displayed on the user dashboard.

System Analysis & Design

System Analysis & Design includes Problem Statement & Objectives, Use Case Diagram & Descriptions, Functional & Non-Functional Requirements, Software Architecture, Database Design & Data Modeling, Data Flow & System Behavior, UI/UX Design & Prototyping, and System Deployment & Integration.

3.1 Problem Statement and Objectives

Problem Statement & Objectives define the problem being solved and project goals. The core problem addressed by the AI module is the inefficiency and lack of accuracy in manual waste sorting. By automating the classification process using AI,

BinWise provides the necessary tool to accurately classify recyclables and quantify their value, thus solving the problem of improper disposal due to user uncertainty.

3.2 Software Architecture

Software Architecture is a high-level design outlining system components, interactions, and architecture style. The AI Module is architected as a Microservice communicating with the main BinWise backend via a RESTful API. This decouples the compute-intensive CV task from the transactional database and user management system.

Key Components and Data Flow:

1. **Image Upload:** The Backend sends image bytes to the FastAPI service.
2. **FastAPI Server:** Receives the image and, using the lifespan event, ensures the YOLOv8 model is loaded once at startup.
3. **YOLOv8 Engine:** Performs the object detection and classification inference.
4. **Weight Estimation Logic:** The system retrieves the fixed weight from the lookup table based on the classified object label.
5. **JSON Response:** The API returns the structured detection data (including Bounding Boxes, Confidence, Label, Weight, and Points) to the Backend.

3.3 Data Flow Diagram

DFD (Data Flow Diagram) shows how data moves through the system.

3.4 Data Flow and System Behavior

Sequence Diagrams represent process flow representation of key interactions between components. The sequence for the core inference process:

1. **Backend FastAPI:** Sends the image file to the /detect/ endpoint.
2. **FastAPI YOLOv8 Engine:** Checks if the model is loaded. If yes, calls the run_inference() function with the image bytes.
3. **YOLOv8 Engine Weight Lookup:** Performs detection and classification, then calls the internal function to look up the object's estimated weight.
4. **Weight Lookup YOLOv8 Engine:** Returns the estimated weight.
5. **YOLOv8 Engine FastAPI:** Returns the complete list of detections (bounding boxes, labels, confidence, weight, and points).
6. **FastAPI Backend:** Returns the final structured JSON response.

3.5 Technology Stack

Technology Stack includes Backend, frontend, and database technologies. The AI module is built entirely using Python-based technologies:

- **Core Language:** Python
- **Model Framework:** Ultralytics (YOLOv8)
- **Deployment Server:** FastAPI, Uvicorn
- **Data Science Libraries:** OpenCV, NumPy, Scikit-learn, Matplotlib
- **Development Environment:** Google Colab
- **Containerization:** Docker

```
# Use an official Python runtime as a parent image

FROM python:3.10-slim

# Set the working directory in the container

WORKDIR /code

# --- UPDATED SECTION ---

# Install missing system libraries for OpenCV and GLib

RUN apt-get update && apt-get install -y \

  libgl1 \

  libglib2.0-0 \

  && rm -rf /var/lib/apt/lists/*

# --- END UPDATED SECTION ---

# Copy the requirements file and install dependencies

COPY ./requirements.txt /code/requirements.txt

RUN pip install --no-cache-dir -r requirements.txt

# Copy the rest of your application code into the container

# This copies main.py, weights.py, and your_model.pt

COPY . /code/

# Tell the container what command to run when it starts

# This runs your FastAPI app on port 8000

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

- **Hosting Platform:** HuggingFace Spaces

VII. AI Technical implementation

Implementation includes Source Code, Version Control & Collaboration, and Deployment & Execution.

4.1 Source Code Structure and Standards

Source Code should be Structured & Well-Commented Code, follow Coding Standards & Naming Conventions, and be Modular Code & Reusability. Security & Error Handling is also essential. The code is designed to be modular and reusable. The key logic is separated into two files:

1. **file.py (FastAPI Logic):** Handles API endpoints, request/response lifecycle, and model loading via the lifespan context manager.
2. **model.py (AI Logic):** Contains the core functions: load_model(), which initializes YOLOv8, and run_inference()

```
def load_model(model_path: str) -> YOLO:
    """
    Loads the YOLO model from the specified path.
    """
    try:
        # *** USE 'r' FOR YOUR WINDOWS PATHS ***
        # The path should be passed from the main FastAPI file
        model = YOLO(model_path)
        print(f"Model loaded successfully. Detecting {len(model.names)} classes.")
        return model
    except Exception as e:
        print(f"Error loading model: {e}")
        print("Please check the path to your 'best.pt' file.")
        raise # Re-raise the exception to stop FastAPI startup
def run_inference(model: YOLO, image_bytes: bytes) -> dict:
    """
    Runs inference on the image bytes, calculates weight, and returns results.
    Args:
        model: The loaded YOLO model object.
```

```
    image_bytes: The raw bytes of the image file received from FastAPI
Returns:
    A dictionary containing detections, total weight, and image URL (or bytes).
"""
# 1. Convert image bytes to an OpenCV image format (numpy array)
image_stream = io.BytesIO(image_bytes)
image_np = cv2.imdecode(
    np.frombuffer(image_stream.read(), np.uint8), cv2.IMREAD_COLOR
)
if image_np is None:
    raise ValueError("Could not decode image bytes
# Run prediction directly on the numpy array (OpenCV image)
# The 'stream=True' is often useful in FastAPI to prevent blocking
results = model.predict(image_np, stream=True)
total_weight = 0
detections = []
# NEW: Create an 'Annotator' object to draw on our image
annotator = Annotator(image_np.copy(), line_width=2, example=str(model.names))
# 2. Process results
if results:
    # The stream=True returns a generator, so we iterate to get the result
    result = next(results)
    boxes = result.boxes
    for box in boxes:
        # Get class ID and name
        cls_id = int(box.cls[0])
        class_name = model.names[cls_id]
        # Get bounding box coordinates (normalized for output, absolute for drawing)
        x1_abs, y1_abs, x2_abs, y2_abs = [int(x) for x in box.xyxy[0]]
        # Look up the weight
        weight_of_item = AVERAGE_WEIGHTS_G.get(class_name)
```

```
        # Structure the detection result
        detection_data = {
          "box_xyxy": [x1_abs, y1_abs, x2_abs, y2_abs],
          "material": class_name,
          "weight_g": weight_of_item,
        }
        detections.append(detection_data)
        # Update total weight and draw label
        if weight_of_item:
          total_weight += weight_of_item
          label = f"{class_name}: {weight_of_item}g"
        else:
          label = f"{class_name}: ??g"
        # Draw the box and our custom label on the image
        annotator.box_label((x1_abs, y1_abs, x2_abs, y2_abs), label, color=(0, 200, 0)) # Green box
    # 3. Prepare the annotated image for response (e.g., as base64 or saved file path)
    # For simplicity, we'll skip returning the image bytes, but the logic is ready.
    annotated_img = annotator.result()
    # cv2.imwrite('annotated_result.jpg', annotated_img) # Save for debugging
    # 4. Return the structured results
    return {
      "total_weight_g": total_weight,
      "detections": detections,
    }
```

4.2 Version Control and Deployment

Version Control Repository is hosted on GitHub, GitLab, or Bitbucket.

- **Version Control:** The entire source code and project documentation are managed on a GitHub Repository. A Feature Branching Strategy was used, with all stable, tested code merged into the main branch upon approval.
- **Deployment Strategy:** The module uses a containerized deployment strategy:

1. A Dockerfile is used to build the final production image, ensuring all dependencies are met.
2. The Docker image runs the FastAPI application using Uvicorn.
3. The container is uploaded to and automatically deployed on HuggingFace Spaces, providing the final stable execution link.

VIII. AI testing & Quality Assurance

Testing & Quality Assurance includes Test Cases & Test Plan, Automated Testing, and Bug Reports.

5.1 Test Cases and Test Plan

The AI module's quality assurance focused on verifying its primary functional requirements:

- **Test Case 1: Correct Classification**
  - Upload an image containing a single, clearly visible object (plastic_bottle_small).
  - The API returns exactly one detection, with the correct label (plastic_bottle_small) and the correct material type (Plastic).
- **Test Case 2: Multi-Object Detection**
  - Upload an image containing two distinct objects (men_t_shirt and small_can_AL).
  - The API returns a list of two detections, each with the correct bounding box and individual classification.
- **Test Case 3: Weight Estimation Consistency**
  - Upload an image of an object whose weight is known via the lookup table (glass_cup).
  - The estimated_weight_g in the JSON response must exactly match the value in the fixed lookup table for that specific object.
- **Test Case 4: Confidence Threshold**
  - Upload a blurry image that produces a low-confidence detection.

- The system should either not return the detection or the confidence score must be below the required 0.5 threshold, proving the filter logic is functional.

5.2 Bug Reports and Resolution

The most significant "bug" resolved was the multi-object failure experienced with the initial MobileNetV2 architecture. This was resolved by the system redesign to utilize YOLOv8, which provided native support for returning a list of detections and bounding boxes per image.

IX. Deployment & Monitoring

AI Monitoring & Maintenance

6.1 System Monitoring

Monitoring is crucial for maintaining the AI service's reliability and meeting the Non-Functional Requirements.

- **API Health Check:** The root endpoint (/) provides a simple health check, confirming the FastAPI application is running and the model state (model_loaded) is true.
- **KPI Monitoring:** The system monitors the API response time and the error rate. Deviations outside acceptable thresholds trigger alerts to the maintenance team.
- **Model Performance Drift:** Since the model uses a static deployment, performance drift is primarily monitored through the reported confidence scores. A decline in average confidence for certain object classes would signal a need for model retraining.

6.2 Maintenance Strategy

Maintenance for the AI Module is categorized into two areas:

- **Software Maintenance:** Requires updating dependencies (ultralytics, fastapi) and patching the Docker container environment. This is handled by rebuilding and redeploying the Docker image on HuggingFace Spaces.
- **Model Maintenance:** Retraining is scheduled when significant performance drift is detected, or when new types of waste or new image quality standards are introduced. The established training pipeline (Google Colab/Cloud GPU usage) allows for efficient retraining within the 14-hour baseline training duration.

Final Deliverables

Final Presentation & Reports includes User Manual, Technical Documentation, Project Presentation (PPT/PDF), and Video Demonstration.

7.1 Technical Documentation

Technical Documentation includes System architecture, database schema, and API documentation. Key technical deliverables include:

- **Source Code Repository:** The complete public/private repository link on GitHub.
- **API Documentation:** The complete documentation for the /detect/ endpoint, detailing request format, JSON response schema, and error codes.
- **Deployment Link:** The executable link to the live, deployed service on HuggingFace Spaces. Here

Frontend: Deployment & Monitoring

Deployment

The frontend follows a modern static site deployment approach:

1. **Build Process:** Development code compiled using npm run build with React components optimized and assets minified.
2. **Deployment Platform:** Vercel or Netlify with automatic deployment from GitHub repository.
3. **CDN Configuration:** Global CDN distribution with edge caching and automatic HTTPS/SSL.

Environment Variables:

- Development: Local API endpoints
- Production: Live API URLs (Backend + AI service)

Key Features:

- Zero-downtime deployments
- Instant rollback capability
- Code splitting and lazy loading
- Image optimization

Monitoring

Performance Monitoring:

- **Page Load Time:** Target < 3 seconds
- **API Response Time:** Track frontend-to-backend requests
- **Bundle Size:** Monitor JavaScript bundle size

User Experience Monitoring:

- **Error Tracking:** Monitor JavaScript errors and React component failures
- **Conversion Rates:** Track signup, scan upload, and pickup scheduling success rates
- **Browser Compatibility:** Monitor across Chrome, Firefox, Safari, Edge

Key Metrics:

- Signup completion rate
- Scan success rate
- Pickup scheduling completion
- Daily/monthly active users

Maintenance

Code Maintenance:

- Regular React and dependency updates (monthly)
- Security vulnerability patches
- Performance optimization
- Bug fixes (Critical: 24 hours, Major: 1 week, Minor: next release)

Content Updates:

- FAQ updates based on user questions
- Awareness page statistics refresh
- Recycling tips updates

Performance Optimization:

- Image lazy loading
- Code splitting improvements
- Caching strategy updates