IJRAR.ORG E-ISSN: 2348-1269, P-ISSN: 2349-5138

INTERNATIONAL JOURNAL OF RESEARCH AND ANALYTICAL REVIEWS (IJRAR) | IJRAR.ORG

An International Open Access, Peer-reviewed, Refereed Journal

CNN INTEGRATED ARCHITECTURE FOR ENHANCED AGE, GENDER AND EMOTION PREDICTION

Submitted By

M. MARY SHAKEENA	**22MH1A42H3**
M.RISHI SADWIK REDDY	**22MH1A42H2**
P. VENKATA PAVAN	**22MH1A42I0**
C. LIKHITHA HASINI	**22MH1A42H0**

Under the esteemed supervision of
Mr. Y.V.V. DURGA PRASAD, M. Tech , (Ph.D.)
Assistant Professor
ADITYA COLLEGE OF ENGINEERING & TECHNOLOGY (A)
(Approved by AICTE, New Delhi & Affiliated to JNTUK, Kakinada)
Surampalem, East Godavari District Andhra Pradesh - 533 437

ABSTRACT

In recent years, facial analysis has gained significant attention in computer vision due to its wide range of applications in security, healthcare, human-computer interaction. This presents a Convolutional Neural Network (CNN) integrated architecture for enhanced prediction of age, gender and emotion from facial images. The proposed system utilizes a dataset of approximately 23,000 + images containing diverse variations in age groups, gender categories, and emotional expressions. The architecture is designed using three classification, and emotion recognition. The input facial image undergoes preprocessing steps including resizing, normalization, and conversion into numerical arrays before being passed to the trained models. The age prediction model outputs an age rage, which is further refined into a realistic age value, while the gender and emotion models classify the input into predefined categories. The system is implemented using TensorFlow and Keras for model development, along with flask and Gradio frameworks for deployment and user interaction. The integration of multiple CNN models allows the systems to achieve efficient and accurate predictions simultaneously. Experimental results demonstrate that the proposed architecture provides reliable performance across all three tasks, making it suitable for real-time applications. This work highlights the effectiveness of deep learning-based facial analysis systems and demonstrates how integrated CNN architecture can be used to solve multiple prediction tasks within a single unified framework.

Keywords – Convolutional Neural Network, Age Estimation, Gender Classification, Emotion, Facial Analysis, Deep Learning, TensorFlow, Keras, UTKFace Dataset, Flask, Gradio, Real-Time interface.

1. INTRODUCTION

In recent years, Artificial Intelligence (AI) and Machine Learning (ML) have significantly transformed the way systems interact with humans. Among these advancements, computer vision has emerged as a powerful domain that enables machines to interpret and understand visual data. The proposed system, **Face Analysis using Deep Learning**, is an intelligent application designed to detect and analyze human facial attributes such as **age, gender, and emotion** from images.

The system leverages deep learning models built using frameworks such as TensorFlow and Keras to process image inputs and generate predictions. These predictions can be used in various real-world applications such as surveillance systems, human-computer interaction, targeted advertising, and healthcare monitoring.

The application is implemented using a **Flask-based web framework**, which allows users to upload images and receive real-time predictions. The backend integrates pre-trained deep learning models for different classification tasks. The system also includes a user-friendly interface to enhance usability.

From the provided implementation, the system loads multiple trained models and performs predictions by converting images into NumPy arrays and processing them through neural networks.

This project demonstrates the integration of machine learning models into a real-time web application, making advanced AI capabilities accessible to end users.

1.1 Purpose of the System

The primary purpose of this system is to develop an intelligent application capable of analyzing facial features and predicting key attributes such as age, gender, and emotion. The system aims to bridge the gap between theoretical machine learning models and practical real-world applications.

The objectives of the system include:

- To design a robust face analysis system using deep learning techniques
- To implement accurate prediction models for:
    - Age estimation
    - Gender classification
    - Emotion recognition
- To provide a user-friendly interface for interaction
- To enable real-time processing of uploaded images
- To demonstrate integration of AI models into web applications

Additionally, the system aims to improve user experience by providing quick and reliable results. The use of multiple models ensures specialization in each task, thereby improving prediction accuracy.

1.2 Problem Statement

Facial analysis has become a critical area of research in computer vision, yet existing systems often address age estimation, gender classification, and emotion recognition as isolated tasks, requiring separate pipelines, multiple models deployed independently, and significant computational overhead. This fragmentation leads to inefficiencies in real-time applications where simultaneous prediction of multiple facial attributes is necessary. Furthermore, many existing approaches struggle with variations in lighting conditions, facial orientations, diverse ethnic backgrounds, and age-related changes in facial structure, resulting in poor generalization across real-world datasets. There is a clear need for a unified, integrated deep learning architecture that can accurately and simultaneously predict age, gender, and emotion from a single facial image input, while remaining computationally efficient and deployable through accessible interfaces for practical use in domains such as security surveillance, healthcare monitoring, and human-computer interaction systems.

1.3 Objectives of the Study

The primary objective of this study is to design and develop a CNN-integrated unified architecture capable of simultaneously predicting age, gender, and emotion from facial images with high accuracy and reliability. The study aims to preprocess a diverse dataset of over 23,000 facial images through resizing, normalization, and numerical conversion to ensure optimal model training across varying conditions. It seeks to build three specialized Convolutional Neural Network models, one each for age estimation, gender classification, and emotion recognition, and integrate them within a single cohesive framework that accepts one input image and returns all three predictions at once. Another objective is to refine the age prediction output from a predicted age range into a realistic specific age value, enhancing the practical usability of the system. The study also aims to deploy the integrated system using both Flask and Gradio frameworks, enabling real-time user interaction through intuitive web-based interfaces. Finally, this work intends to demonstrate that a multi-task facial analysis system built on deep learning can serve as an effective, scalable, and real-world-ready solution across application domains including healthcare, security, and interactive technology systems.

1.4 Scope of the System

The scope of the system defines the boundaries and functionalities that the project covers. This system focuses on facial image analysis using machine learning models and provides predictions based on input images.

Functional Scope

- Uploading facial images through a web interface
- Processing images using pre-trained deep learning models
- Predicting:
    - ☒ Age (based on age ranges)
    - ☒ Gender (Male/Female)

- ☒ Emotion (e.g., Happy, Sad, Angry, etc.)

- Displaying results in real-time

Technical Scope

- Integration of multiple deep learning models
- Image preprocessing using PIL and NumPy
- Backend implementation using Flask
- Model loading using TensorFlow/Keras

Application Scope

The system can be applied in:

- Smart surveillance systems
- Human behavior analysis
- Marketing and advertisement targeting
- Emotion-aware AI systems
- Healthcare monitoring systems

Limitations of Scope

- The system works only with facial images
- Accuracy depends on the trained models
- Real-time video processing is not included
- Limited to predefined categories for emotion detection

1.5 Existing System

The current systems available for facial analysis are either:

- Standalone machine learning models without user interfaces
- Commercial applications with restricted access

- Systems focusing on a single attribute (only age or only emotion)

Drawbacks of Existing Systems

- Lack of integration of multiple features (age, gender, emotion together)
- Limited accessibility for general users
- High computational requirements
- Complex setup and deployment
- Lack of real-time web-based interaction

Many existing solutions require users to have technical expertise to run models locally. Additionally, most systems do not provide a unified platform for multiple predictions.

1.6 Proposed System

The proposed system overcomes the limitations of existing systems by providing an integrated and user-friendly platform for facial analysis.

Key Features of the Proposed System

- Unified system for:
  - ☒ Age prediction
  - ☒ Gender classification
  - ☒ Emotion detection
- Web-based interface using Flask
- Real-time image processing
- Easy-to-use interface for non-technical users
- Integration of multiple deep learning models

The system architecture includes:

- **Frontend**: HTML interface for user interaction
- **Backend**: Flask server handling requests
- **Models**: Pre-trained deep learning models
- **Processing Layer**: Image preprocessing and prediction logic

From the implementation, the system loads models dynamically and processes input images using PIL and NumPy before generating predictions .

Working of the Proposed System

The system follows a structured workflow:

1. User uploads an image
2. Image is converted into NumPy array
3. Preprocessing is applied
4. Image is passed to:
   - ☒ Age model
   - ☒ Gender model
   - ☒ Emotion model
5. Predictions are generated
6. Results are displayed to the user

Algorithmic Flow

- Input → Image Upload
- Preprocessing → Convert to RGB array
- Prediction → Deep Learning Models
- Output → Age, Gender, Emotion

Advantages of the Proposed System

- Integrated multi-feature prediction system
- Easy deployment and usage
- Real-time analysis
- Scalable architecture
- Modular design (separate models for each task)

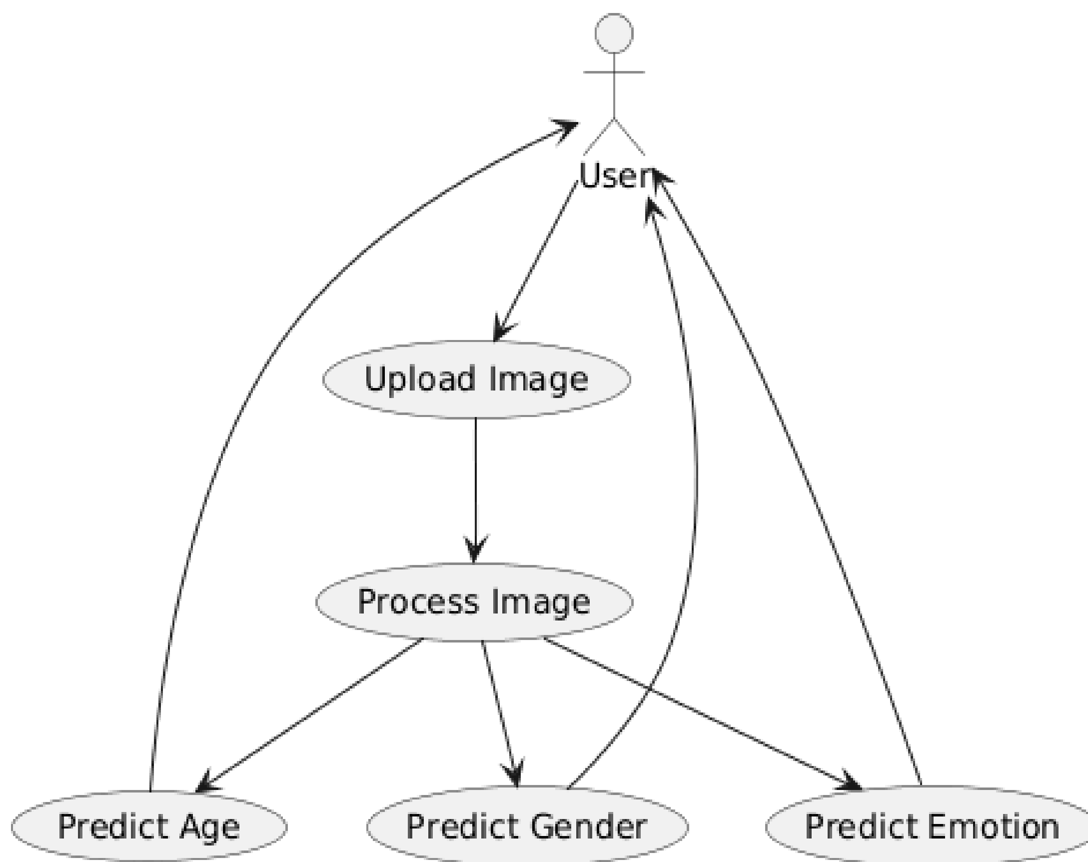


Fig: 1.4.1 Use case Diagram

2.REQUIREMENT ANALYSIS

Requirement analysis is one of the most critical phases in the software development life cycle. It involves identifying, gathering, analyzing, and documenting the needs and expectations of users and stakeholders. This phase ensures that the system being developed meets the intended objectives and performs efficiently under defined constraints.

In the context of the **Face Analysis System (Age, Gender, and Emotion Detection)**, requirement analysis plays a vital role in defining how the system should behave, what functionalities it must provide, and what constraints it must operate within.

The system integrates deep learning models with a web-based interface, requiring careful analysis of both functional and non-functional requirements. The requirements must ensure that the system performs accurately, efficiently, and securely while providing a seamless user experience.

The requirement analysis phase includes:

- Understanding user needs
- Identifying system functionalities
- Defining performance expectations
- Specifying hardware and software requirements
- Ensuring security and usability

This chapter provides a comprehensive breakdown of all system requirements, categorized into functional and non-functional requirements.

2.1 Functional Requirements

Functional requirements define the specific operations and tasks that the system must perform. These requirements describe the behavior of the system and how it responds to user inputs.

For the Face Analysis System, the functional requirements revolve around image processing, prediction generation, and result display.

2.1.1 User Interaction Requirements

The system must allow users to interact with it in a simple and intuitive manner.

- The user should be able to access the system via a web browser
- The system should provide a homepage with navigation options
- The user should be able to upload an image file
- The system should validate the uploaded file
- The user should be able to view results instantly

2.1.2 Image Upload and Validation

The system must handle image inputs effectively.

- Accept image formats such as JPG, PNG, and JPEG
- Reject unsupported file formats

- Ensure that an image is uploaded before processing
- Handle empty file submissions gracefully
- Validate file size to prevent overload

From the implementation, the system checks whether an image is present in the request and validates the file before processing .

2.1.3 Image Preprocessing

Before prediction, the system must preprocess the input image.

- Convert the uploaded image into RGB format
- Transform the image into a NumPy array
- Ensure compatibility with model input requirements
- Normalize or resize the image if required

2.1.4 Prediction Functionality

The core functionality of the system is prediction.

Age Prediction

- Predict age based on predefined ranges
- Convert age ranges into a specific numeric value
- Use randomization within the predicted range for realism

Gender Prediction

- Classify the image as Male or Female
- Use trained classification models

Emotion Detection

- Detect facial emotions such as:
    - ⊠ Happy
    - ⊠ Sad
    - ⊠ Angry
    - ⊠ Neutral

⌧ Surprise

The system uses multiple trained models for these predictions, ensuring modular and independent processing of each attribute .

2.1.5 Result Generation

The system must generate and display results clearly.

- Display predicted age
- Display predicted gender
- Display detected emotion
- Provide results in JSON format (for API use)
- Present results in UI for users

2.1.6 Error Handling

The system must handle errors effectively.

- Display error if no image is uploaded
- Handle corrupted image files
- Manage model loading errors
- Provide meaningful error messages

2.1.7 Routing and Navigation

The system includes multiple routes for navigation:

- `/` → Home page
- `/analyze` → Image upload page
- `/about` → Project description
- `/predict` → Prediction API

2.1.8 Model Management

- Load pre-trained models during initialization
- Ensure models are available before prediction

- Use fallback mechanisms for model keys

2.2 Non-Functional Requirements

Non-functional requirements define how the system performs rather than what it does. These requirements focus on quality attributes such as performance, usability, reliability, and security.

2.2.1 User Interface and Human Factors

The user interface is a critical component of the system as it directly interacts with users.

Requirements

- Simple and intuitive design
- Easy navigation between pages
- Clear instructions for uploading images
- Responsive layout for different devices
- Minimal loading time

Human Factors

- The system should be usable by non-technical users
- Instructions should be easy to understand
- Visual feedback should be provided after actions
- Error messages should be user-friendly

2.2.2 Software Requirements

The software requirements specify the tools, frameworks, and libraries required for the system.

Core Technologies

- Python programming language
- Flask framework for backend
- TensorFlow and Keras for deep learning
- NumPy for numerical operations
- PIL for image processing

From the requirements file, the system depends on multiple libraries including TensorFlow, NumPy, and Flask

Development Environment

- IDE: VS Code / PyCharm
- Operating System: Windows/Linux
- Browser: Chrome/Firefox

2.2.3 Hardware Requirements

The system requires adequate hardware for efficient performance.

Minimum Requirements

- Processor: Intel i3 or equivalent
- RAM: 4 GB
- Storage: 10 GB free space

Recommended Requirements

- Processor: Intel i5 or higher
- RAM: 8 GB or more
- GPU (optional for training models)

2.2.4 Usability

Usability ensures that the system is easy to use and understand.

Key Aspects

- User-friendly interface
- Minimal steps to complete tasks
- Clear result presentation
- Accessibility for beginners

2.2.5 Reliability

Reliability refers to the system's ability to function consistently.

Requirements

- System should not crash during processing
- Models should load correctly every time

- Predictions should be consistent
- Error handling should be robust

2.2.6 Performance

Performance is crucial for real-time applications.

Requirements

- Fast image processing
- Quick response time
- Efficient memory usage
- Ability to handle multiple requests

2.2.8 Physical Environment

The system operates in a digital environment but must consider:

- Stable internet connection
- Secure server hosting
- Controlled deployment environment

2.2.9 Security Requirements

Security is essential to protect user data and system integrity.

Requirements

- Validate user inputs
- Prevent malicious file uploads
- Secure API endpoints
- Avoid data leakage
- Use safe model loading mechanisms

2.2.10 Resource Requirements

The system must manage resources efficiently.

Resources Include

- CPU and memory usage
- Storage for models
- Network bandwidth

Optimization Techniques

- Efficient image processing
- Model optimization
- Caching mechanisms

3: SYSTEM ANALYSIS

System analysis is a critical phase in the software development life cycle that focuses on understanding the system requirements in depth and translating them into a structured model. It involves studying the current system, identifying problems, and defining how the proposed system will address these issues.

For the **Face Analysis System (Age, Gender, and Emotion Detection)**, system analysis helps in understanding how image data is processed, how machine learning models interact, and how the user interacts with the system through a web interface.

This chapter provides a detailed explanation of:

- System overview
- Use case analysis
- Actors involved
- Functional workflows
- UML diagrams for better understanding

The system is designed as a **machine learning-powered web application**, where users upload images and receive predictions based on trained models.

3.1 Introduction

System analysis defines the logical structure and workflow of the system. It helps in identifying:

- Inputs to the system
- Processing steps

- Outputs generated
- Interaction between system components

The Face Analysis System takes **image input**, processes it using **deep learning models**, and generates predictions such as:

- Age
- Gender
- Emotion

From the implementation, the system converts uploaded images into NumPy arrays and processes them using multiple models to generate outputs .

Objectives of System Analysis

- To understand system functionality in detail
- To identify user interactions
- To define workflows and processes
- To create UML diagrams for visualization
- To ensure system feasibility

Key Components Identified

The system consists of the following components:

- User Interface (Frontend)
- Backend Server (Flask)
- Machine Learning Models
- Image Processing Module
- Prediction Engine

System Workflow Overview

1. User uploads an image
2. Image is validated
3. Image is preprocessed
4. Image is passed to models
5. Predictions are generated
6. Results are displayed

3.2 Use Cases

Use cases describe how users interact with the system. They define the functional requirements in terms of user actions and system responses.

The Face Analysis System includes several use cases that represent different functionalities available to the user.

3.2.1 Actors

Actors are entities that interact with the system. In this system, actors can be human users or external systems.

Primary Actor

User

The user is the main actor who interacts with the system.

Responsibilities:

- Upload image
- Request analysis
- View results
- Navigate through pages

Secondary Actors

Although the system is standalone, some internal components act as secondary actors:

- **Machine Learning Models**
  - ☒ Perform predictions
- **Flask Server**
  - ☒ Handles requests and responses

Actor Characteristics

- Non-technical or technical user
- Requires simple interface
- Expects fast results
- Minimal interaction complexity

3.2.2 List of Use Cases

The following are the main use cases of the system:

1. Upload Image

- User uploads an image through the interface
- System validates the file

2. Analyse Image

- System processes the uploaded image
- Applies preprocessing techniques

3. Predict Age

- Age model predicts age range
- System converts range to numeric value

4. Predict Gender

- Gender model classifies the image

5. Detect Emotion

- Emotion model detects facial expression

6. Display Results

- System shows predictions to the user

7. Handle Errors

- System displays error messages

Fig: 3.2.2.1 Use Case Description Table

Use Case	Description
Upload Image	User uploads an image
Analyse Image	System processes image
Predict Age	Estimate age
Predict Gender	Classify gender
Detect Emotion	Identify emotion
Display Results	Show predictions

3.3 Use Case Diagrams

Use case diagrams visually represent the interaction between actors and the system.

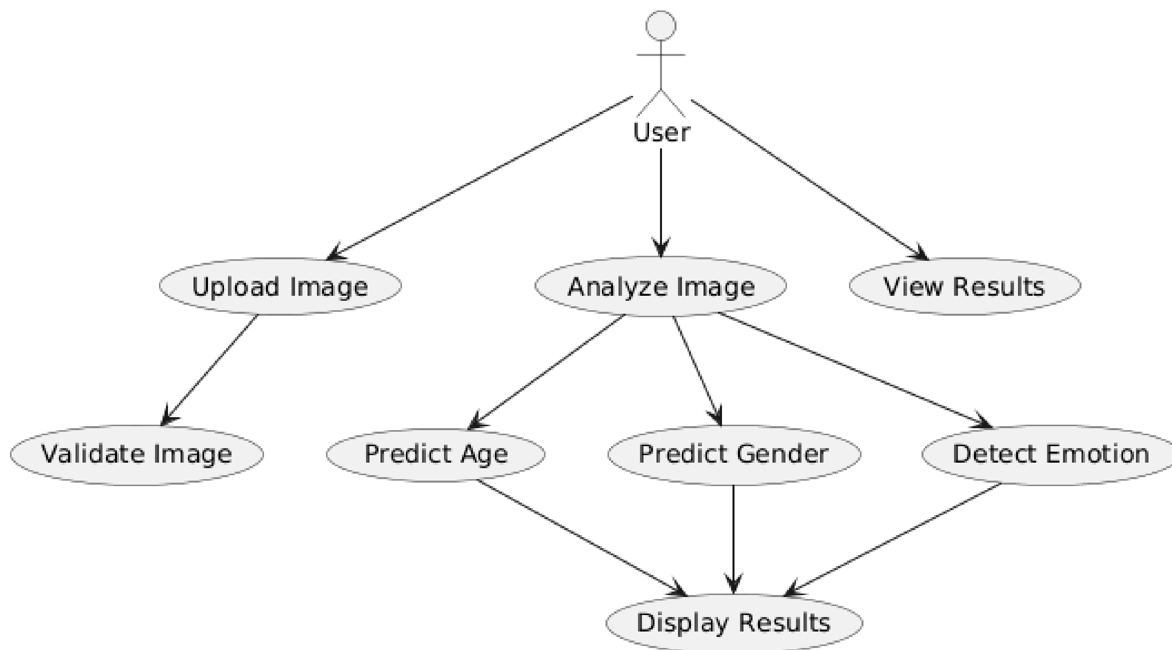


Fig : 3.3.1 Overall Use Case Diagram

3.4 Detailed Use Case Explanation

Upload Image

- User selects image file
- System checks file validity

Analyse Image

- System converts image to array
- Applies preprocessing

Prediction Use Cases

- Each model runs independently
- Results are combined

3.5 System Workflow Analysis

The system workflow describes how data flows through the system

.Step-by-Step Workflow

1. **User Input**
   - ☒ User uploads image
2. **Validation**
   - ☒ Check file presence
   - ☒ Check file type
3. **Preprocessing**
   - ☒ Convert to RGB
   - ☒ Convert to NumPy array
4. **Prediction**
   - ☒ Age model → age range
   - ☒ Gender model → gender
   - ☒ Emotion model → emotion
5. **Post-processing**
   - ☒ Convert age range to number
6. **Output**
   - ☒ Display results

Workflow Characteristics

- Sequential processing
- Modular model execution

- Real-time response

3.6 Data Flow Analysis

Data flow analysis describes how data moves within the system.

Data Flow Steps

- Input: Image file
- Processing: Conversion and prediction
- Output: JSON response

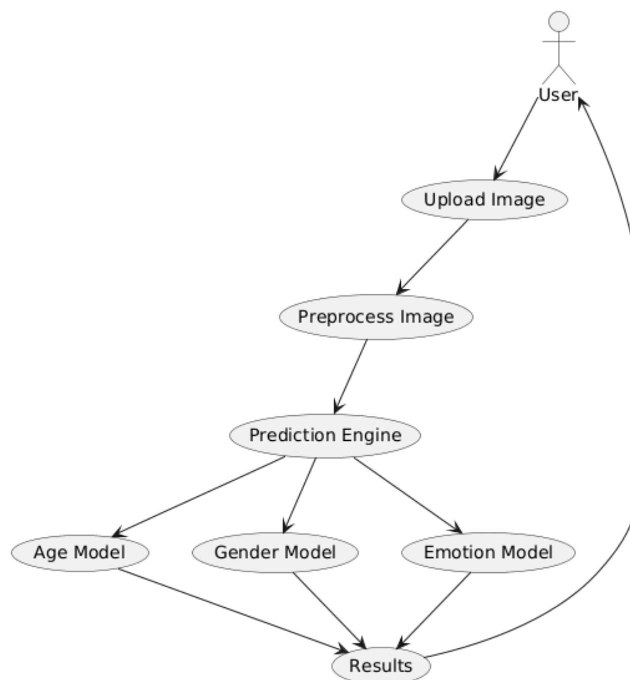


Fig: 3.6.1 Data Flow Diagram

3.7 Functional Decomposition

Functional decomposition breaks the system into smaller modules.

Modules Identified

- Image Upload Module
- Validation Module
- Preprocessing Module
- Prediction Module
- Output Module

Module Responsibilities

Image Upload Module

- Accepts user input

Validation Module

- Ensures correct input

Preprocessing Module

- Converts image format

Prediction Module

- Runs ML models

Output Module

- Displays results

4. SYSTEM DESIGN

System design is a crucial phase in the software development lifecycle where the requirements gathered during analysis are transformed into a structured blueprint for implementation. It defines how the system will be built, how components will interact, and how data will flow within the system.

For the **Face Analysis System (Age, Gender, Emotion Detection)**, system design focuses on integrating deep learning models with a web-based application. The system is designed to be modular, scalable, and efficient, ensuring seamless interaction between the frontend, backend, and machine learning components.

This chapter covers:

- System architecture
- Object model
- Subsystems
- UML diagrams (class, sequence, activity, component, deployment)
- Database design
- Static and dynamic models

4.1 Introduction

The design phase bridges the gap between requirement analysis and implementation. It provides a detailed structure of the system, enabling developers to build the application efficiently.

The Face Analysis System is designed using a **layered architecture**, where each layer has a specific responsibility:

- Presentation Layer (Frontend)
- Application Layer (Backend)
- Processing Layer (ML Models)

- Data Layer (Models & Files)

The system processes user-uploaded images, converts them into machine-readable formats, and passes them through trained models to generate predictions. From the implementation, the system uses Flask as the backend framework and integrates TensorFlow/Keras models for predictions .

4.2 System Architecture

System architecture defines the overall structure of the system, including components and their interactions.

4.2.1 Architecture Overview

The system follows a **client-server architecture**:

- **Client (Frontend)**
  - ☒ Handles user interaction
  - ☒ Allows image upload
- **Server (Backend)**
  - ☒ Processes requests
  - ☒ Runs prediction logic
- **Machine Learning Models**
  - ☒ Perform classification tasks

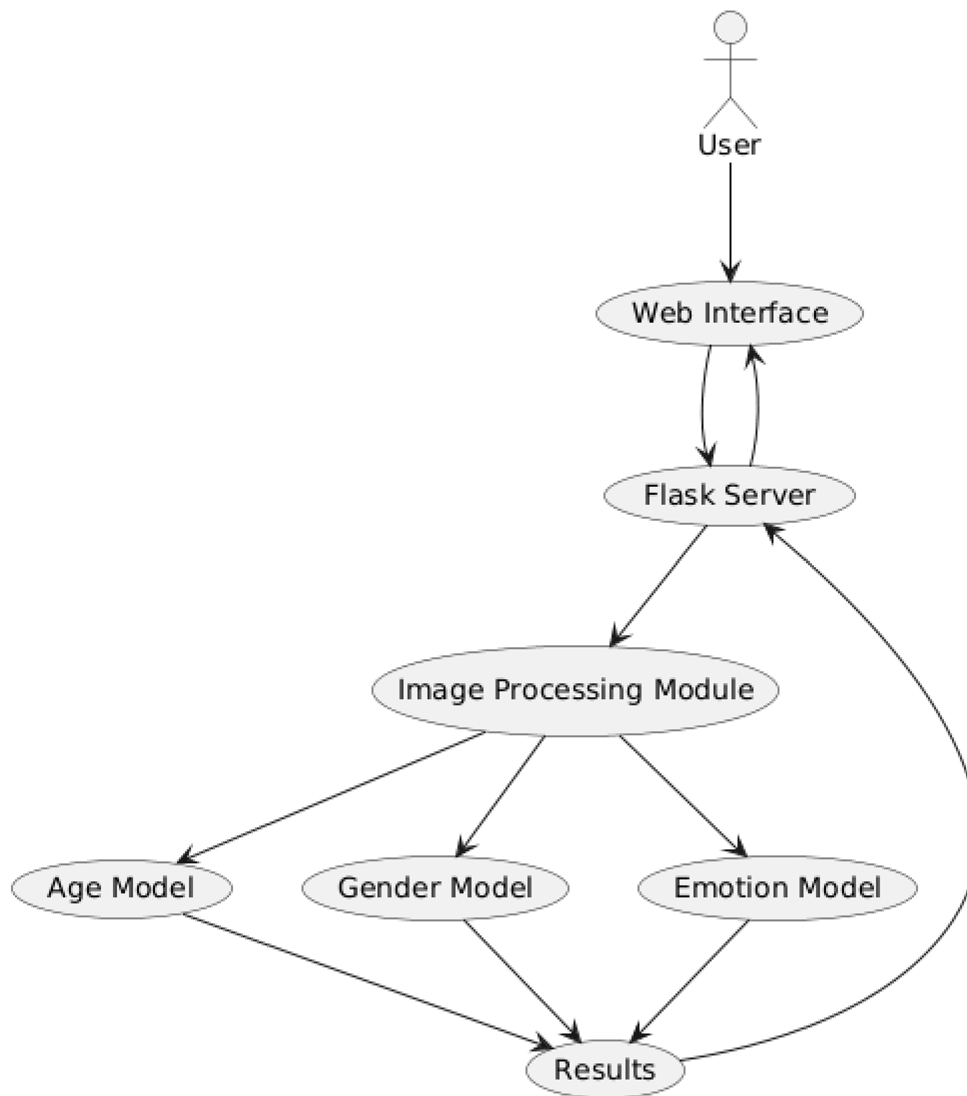


Fig: 4.2 System Architecture

4.3 System Object Model

The object model represents the system in terms of objects and their relationships.

4.3.1 Introduction

Object modeling helps in identifying:

- Classes
- Attributes
- Methods
- Relationships

In this system, objects represent components such as:

- User
- Image
- Prediction
- Models

4.3.2 Subsystems

The system is divided into multiple subsystems:

1. User Interface Subsystem

Handles user interaction.

Functions:

- Display pages
- Accept image uploads
- Show results

2. Backend Subsystem

Manages application logic.

Functions:

- Handle requests
- Validate input
- Route data

3. Image Processing Subsystem

Processes images before prediction.

Functions:

- Convert image to RGB
- Convert to NumPy array

4. Prediction Subsystem

Performs predictions using models.

Functions:

- Age prediction
- Gender classification
- Emotion detection

5. Model Management Subsystem

Handles model loading and execution.

Functions:

- Load models at runtime
- Manage model keys

- Provide fallback logic

4.4 Object Description

4.4.1 Objects

The key objects in the system include:

User Object

- Attributes:
  - ☒ user_id
  - ☒ uploaded_image
- Methods:
  - ☒ upload_image()

Image Object

- Attributes:
  - ☒ image_data
  - ☒ format
- Methods:
  - ☒ preprocess()

Prediction Object

- Attributes:
  - ☒ age
  - ☒ gender
  - ☒ emotion
- Methods:
  - ☒ generate_results()

Model Object

- Attributes:
  - ☒ model_name
  - ☒ model_type
- Methods:
  - ☒ predict()

4.4.2 Class Diagram

The class diagram of this face analysis system represents the static view of the application by identifying four core classes — User, Image, Prediction, and Model — along with their individual responsibilities and the directional relationships that exist between them.

The User class is the entry point of the system. It represents the human actor who interacts with the application. Its sole responsibility is defined through the uploadImage() method, which initiates the entire workflow by submitting a face image for analysis. This class captures the interaction boundary between the end user and the rest of the system.

The Image class receives the uploaded data from the User and is responsible for processing it before it can be consumed by the machine learning pipeline. Its key operation is preprocess(), which handles tasks such as resizing, normalization, and formatting the raw image data into a structure suitable for model inference. This class acts as a data preparation layer in the object-oriented model.

The Prediction class takes the preprocessed image and orchestrates the generation of analytical results. The generateResults() method defined in this class is responsible for coordinating the output — combining predictions about age, gender, and emotion into a unified result that can be returned to the user interface. It acts as the central logic controller between image processing and model execution.

The Model class is the innermost execution unit of the system. It contains the predict() method, which directly interfaces with the underlying machine learning models to run inference on the prepared image data and return raw prediction values. This class encapsulates the AI computation layer and is invoked by the Prediction class.

The relationships in the diagram are represented as directed dependency associations flowing from User to Image, from Image to Prediction, and from Prediction to Model, clearly illustrating a sequential delegation chain where each class depends on the next to fulfil its responsibility. This class diagram forms the structural backbone of the system and serves as the direct basis for implementation in an object-oriented programming language.

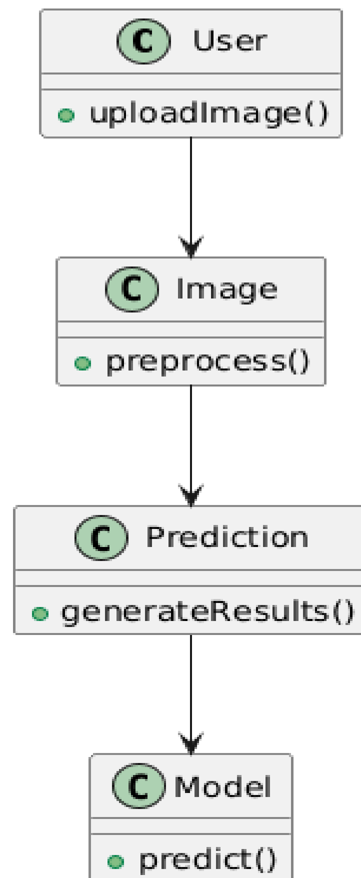


Fig : 4.4.2 Class Diagram

4.5 Object Collaboration

An Object Collaboration Diagram is a UML diagram that shows how objects interact with each other by exchanging messages to accomplish a specific task, emphasizing the structural links between objects rather than the time ordering of interactions. It represents the runtime behavior of a system by depicting the named messages passed between interconnected objects along communication links. It is also known as a Communication Diagram and is particularly useful for understanding which objects collaborate with which, and what messages flow between them to fulfil a system's functionality.

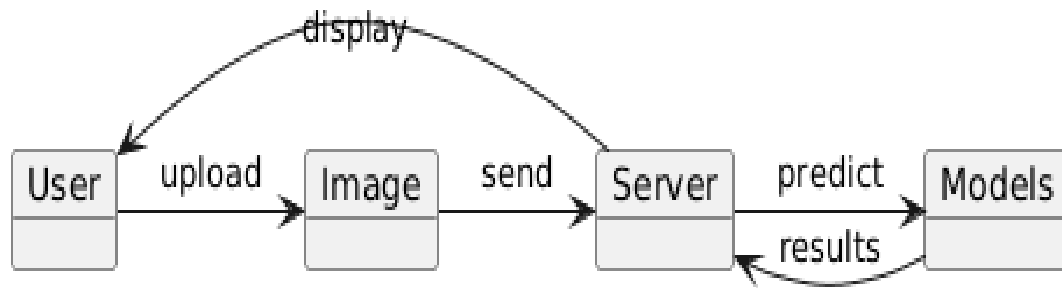
4.5.1 Object Collaboration Diagram

The Object Collaboration Diagram of this face analysis system illustrates how the four key runtime objects — User, Image, Server, and Models — interact with each other by exchanging messages to accomplish the overall goal of predicting facial attributes. Unlike a sequence diagram that focuses on time ordering, a collaboration diagram emphasizes the structural relationships and communication links between objects, showing which object talks to which and through what named message.

The collaboration begins when the User object sends an upload message to the Image object, initiating the process by submitting a face image through the browser interface. The Image object, upon receiving the uploaded data, forwards it to the Server object via a send message, passing the prepared image data over the HTTP channel to the Flask backend. The Server object then communicates with the Models object by sending a predict message, invoking the machine learning inference pipeline to analyze the image for age,

gender, and emotion. Once the Models object completes its computation, it sends the results message back to the Server, delivering the raw prediction outputs. Finally, the Server closes the collaboration cycle by sending a display message back to the User, rendering the prediction results on the browser interface for the user to view.

This diagram effectively captures the complete round-trip communication flow of the system — from user action to model inference and back to the user — making it easy to understand how each object plays a specific collaborative role in delivering the face analysis functionality. The curved arc for the display message visually highlights the end-to-end feedback loop that completes the user's request.



ig : 4.5.1 Object Collaboration Diagram

4.6 Dynamic Model

The dynamic model describes system behavior over time.

4.6.1 Sequence Diagrams

The sequence diagram of this face analysis system describes the time-ordered interaction between four participants — User, UI, Server, and Models — showing exactly how and in what sequence messages flow between them to produce a prediction result. Each vertical dashed line represents the lifeline of a participant, and the horizontal arrows represent messages exchanged over time from top to bottom.

The interaction begins when the User sends an Upload Image message to the UI, which represents the browser interface. This is the initiating action that triggers the entire workflow. Once the image is received at the UI layer, it immediately forwards it to the Server via a Send Request message, packaging the image data into an HTTP request directed at the Flask backend.

Upon receiving the request, the Server performs an internal Preprocess Image operation — shown as a self-directed message — where the raw image is resized, normalized, and converted into a format that the machine learning models can consume. This step ensures data quality before inference begins.

After preprocessing, the Server sends a Predict message to the Models participant, invoking the age, gender, and emotion detection models to run inference on the prepared image. The Models component processes the input and returns the prediction output back to the Server as a Results message, shown as a dashed return arrow to indicate it is a response rather than a new call.

The Server then constructs a Response message and sends it back to the UI, passing the formatted prediction data. Finally, the UI delivers the final Display Results message back to the User, completing the full round-trip interaction and rendering the analysis output on the browser screen. This sequence diagram effectively maps the complete usage scenario of the system from the moment the user uploads an image to the moment the results are displayed.

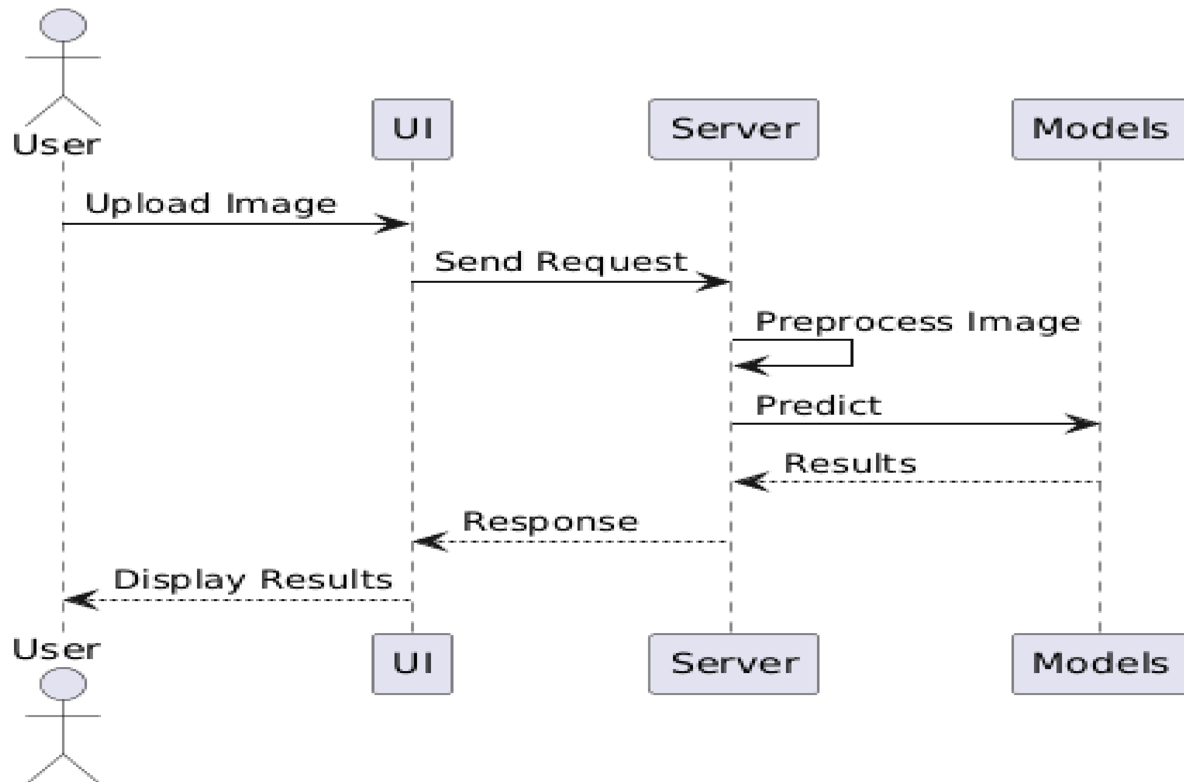


Fig : 4.6.1 Sequence Diagram

4.6.2 Activity Diagrams

The activity diagram of this face analysis system represents the complete dynamic workflow of the application, depicting the step-by-step flow of control from the moment a user initiates an action to the point where the system reaches its final state. It captures both the normal execution path and the alternative error path, making it a comprehensive behavioural model of the system.

The workflow begins at the initial node, represented by a solid filled circle, which marks the starting point of the entire process. From here, control immediately flows into the first activity, Upload Image, where the user selects and submits a face image through the browser interface. This is the entry action that drives all subsequent processing.

After the image is uploaded, the flow reaches a decision node labelled Valid? which acts as a branching point to evaluate whether the uploaded image meets the required format and quality criteria. If the image is valid, the Yes branch is followed and the system proceeds along the normal execution path. If the image is not valid or fails validation checks, the No branch is taken and the system transitions to the Error activity, which handles the failure gracefully by notifying the user of the invalid input.

Along the successful path, the system enters the Preprocess activity, where the image is resized, normalized, and prepared for model inference. This is followed by the Predict activity, where the preprocessed image is passed to the machine learning models to generate predictions for age, gender, and emotion. Once the predictions are computed, the Display activity presents the results to the user on the browser interface.

Both the successful path ending at Display and the error path ending at Error converge at a merge node, represented by an empty diamond, which brings the two branches back together into a single flow. From this merge point, control reaches the final node — represented by a bull's-eye symbol — marking the termination of the entire activity. This diagram clearly models the usage scenario of the face analysis system, capturing both valid and invalid image handling in a single unified flow.

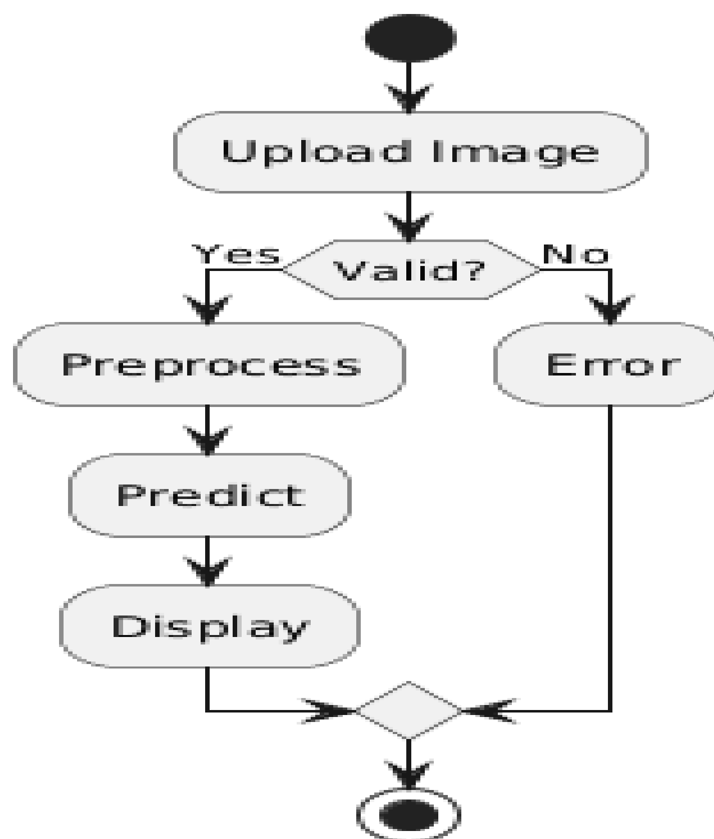


Fig : 4.6.2 Activity Diagram

4.7 Database Design

Although the system is primarily model-based, database design can be included for scalability.

4.7.1 Entity Relationship Diagram

The Entity Relationship Diagram of this face analysis system models the logical data structure of the application by identifying the core entities, their attributes, and the relationships that exist between them. It provides a clear blueprint of how data is organized and stored at the database level, serving as the foundation for designing the backend data model of the system.

The first entity is User, which represents a person who interacts with the face analysis application. It holds a single key attribute, user_id, which serves as the primary key and uniquely identifies each user in the system. This entity is the root of the data model, as all image uploads and predictions originate from a user's action.

The second entity is Image, which represents each face image that is uploaded into the system by a user. It contains the attribute image_id as its primary key, uniquely identifying every image record stored. The relationship between User and Image is a one-to-many association, indicating that a single user can upload multiple images over time, while each image belongs to exactly one user. This is depicted using a double bar on the User side (mandatory, one) and an open circle with a crow's foot on the Image side (optional, many).

The third entity is Prediction, which stores the results generated by the machine learning models for each analyzed image. It has four attributes: prediction_id as the primary key, and three result attributes — age, gender, and emotion — which hold the predicted values for each respective model. The relationship between Image and Prediction is a one-to-one or mandatory association represented by double bars on both sides, meaning every image that is processed must have a corresponding prediction record, and each prediction is tied to exactly one image. This ensures full traceability from user upload through to the final analytical output.

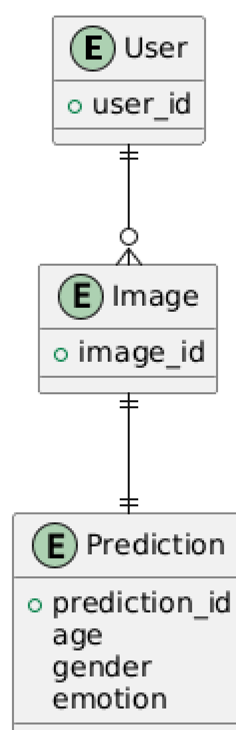


Fig : 4.7.1 Entity Relationship Diagram

4.8 Static Model

4.8.1 Component Diagram

Overview

This UML Component Diagram represents the physical architecture of a face analysis system built on object-oriented principles. It visualizes the system as three major subsystems — Frontend, Backend, and ML Models — and illustrates how they interact to deliver functionality like age estimation, gender detection, and emotion recognition.

Frontend Component

The Frontend subsystem contains a single component: HTML Pages. This layer serves as the user-facing interface through which end users interact with the system. It is responsible for capturing user input (such as uploading or streaming a face image) and rendering the results returned by the server. It communicates directly with the Backend via HTTP requests.

Backend Component

The Backend subsystem is powered by a Flask Server. Flask acts as the central coordinator of the entire application. It receives requests from the HTML frontend, processes them, routes the image data to the appropriate ML models, collects predictions, and returns the consolidated results to the frontend. Flask essentially acts as the integration hub connecting the presentation layer with the intelligence layer.

ML Models Component

The ML Models subsystem consists of three independent components:

Age Model — Analyses facial features from the input image to predict the approximate age of the person.

Gender Model — Classifies the detected face to predict the gender of the subject.

Emotion Model — Identifies facial expressions and maps them to emotion categories such as happy, sad, angry, neutral, etc.

Each model is a standalone component invoked by the Flask Server and returns its prediction independently.

Component Relationships

The diagram shows directed dependency arrows flowing from top to bottom: the HTML Pages depend on the Flask Server for data, and the Flask Server depends on all three ML Models for predictions. This clearly captures a layered, unidirectional dependency structure — a hallmark of well-organized component-based system design.

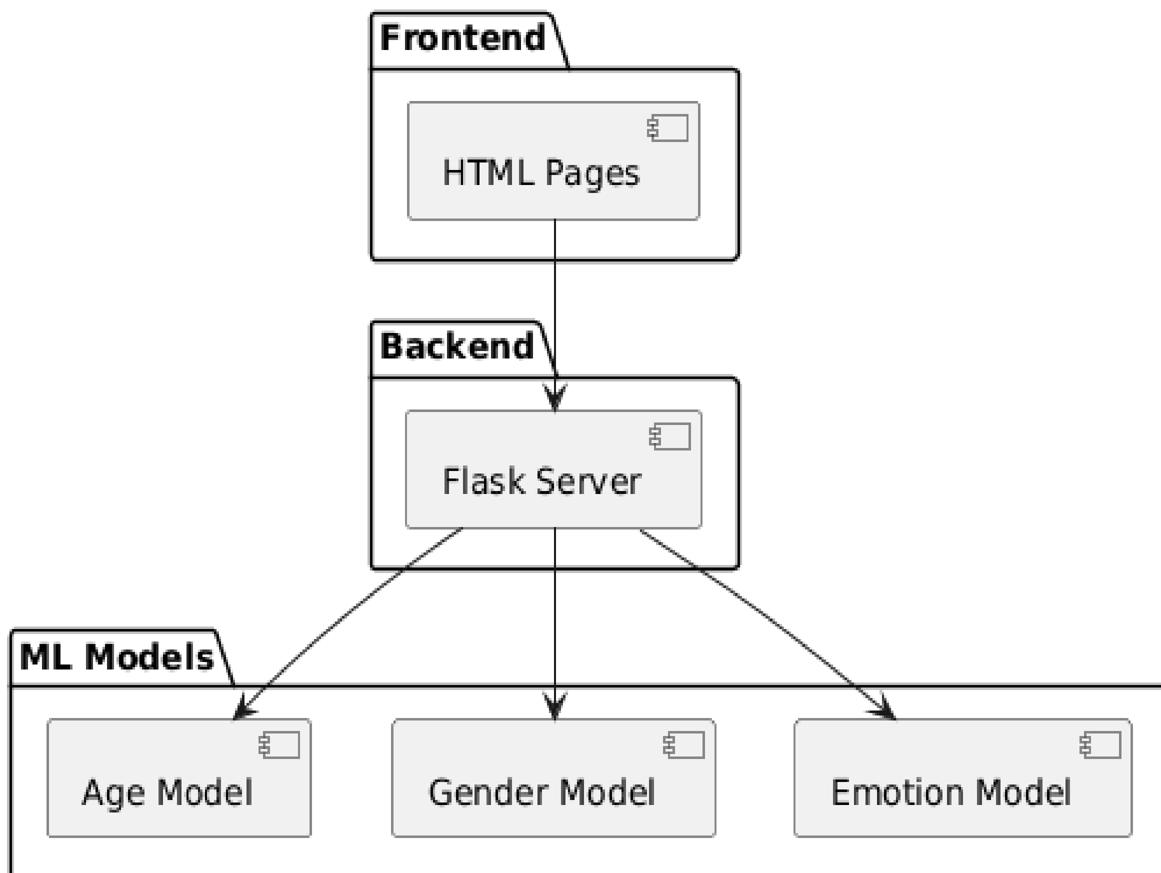


Fig : 4.8.1 Component Diagram

4.8.2 Deployment Diagram

Overview

This UML Deployment Diagram represents the static deployment view of the face analysis system. It describes the physical topology of the runtime environment — specifically how software components (Browser, FlaskApp, and MLModels) are distributed across hardware/software execution nodes (Client and Server). Each node is depicted as a 3D box, which is the standard UML notation for a deployment node.

Client Node

The Client node represents the user's machine — typically a personal computer or any device with a web browser. It hosts a single artifact: the Browser component. The browser serves as the runtime environment for the user interface, rendering the HTML pages, submitting image data, and displaying the results returned from the server. The client communicates with the server over an HTTP connection.

Server Node

The Server node represents the physical or virtual machine where all the application logic and intelligence resides. It hosts two components:

Flask — This is the web application runtime built on Flask. It receives incoming HTTP requests from the client's browser, handles routing, processes the input data (face image), and delegates prediction tasks to the ML models. It also formats and returns the response back to the client.

Models — This component represents the collection of machine learning models deployed on the same server. It is invoked directly by Flask at runtime. It encapsulates the age estimation, gender classification, and emotion detection models, executing predictions locally within the server environment.

Node Relationships and Connections

A directed dependency arrow runs from the Client node to the Server node, representing the HTTP communication channel between the browser and the Flask server. Within the Server node, a further dependency arrow flows from FlaskApp down to MLModels, showing that Flask orchestrates and depends on the ML models to fulfil prediction requests.

Significance of the Deployment View

This diagram highlights a classic two-tier client/server architecture. All computation — both application logic and ML inference — is centralized on the server, keeping the client lightweight (just a browser). This makes the system easy to scale on the server side independently without any changes to the client.

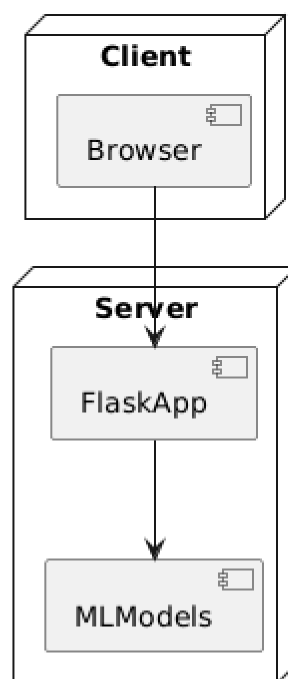


Fig : 4.8.2 Deployment Diagram

5.IMPLEMENTATION

Implementation is the phase in which the system design is translated into a working application using programming languages, frameworks, and tools. It involves writing source code, integrating trained deep learning models, preprocessing input data, handling HTTP requests, and exposing the inference pipeline through a web interface.

This project implements the CNN Integrated Architecture for Enhanced Age, Gender and Emotion Prediction of Facial Images using Python as the core language, Flask as the web backend, and TensorFlow/Keras for model training and inference. Three separate Convolutional Neural Network models are trained on the UTKFace and CK+ datasets and deployed together under a single Flask server that returns simultaneous predictions for all three attributes.

The implementation is organized into five major components: the Software Environment and Dependencies, the UTKFace/CK+ Dataset preparation pipeline, the Age Prediction Model notebook, the Gender Classification Model notebook, the Emotion Detection Model notebook, and the Flask Web Application that integrates all three models into a production-ready inference service.

5.1 Software Used

The implementation relies on a carefully selected stack of open-source tools, frameworks, and libraries that collectively cover dataset management, deep learning model construction, image preprocessing, and web deployment. Every tool was chosen for its compatibility with the Python ecosystem and its proven track record in computer vision and web application development.

Model training is performed on the Kaggle Notebook platform, which provides free access to NVIDIA Tesla T4 GPU acceleration and direct access to the UTKFace and CK+ datasets via the Kaggle hub API. The trained. keras model files are exported from Kaggle and integrated into the Flask application running on a local Windows development machine.

The complete requirements.txt specifies eight runtime dependencies: pillow, pickle, TensorFlow, numpy, keras, scikit-learn, transformers, and flask. These libraries collectively handle every stage of the pipeline from raw image file I/O through deep learning inference to HTTP response serialization.

5.1.1 Programming Language – Python

Python 3.10 is the primary programming language for both the model training notebooks and the Flask web application. Python's concise syntax, interactive notebook support, and mature scientific computing ecosystem make it the standard choice for deep learning research and deployment projects.

The entire codebase — dataset parsing, model definition, training loops, image preprocessing, Flask routing, and Gradio interface — is written in Python. This uniformity eliminates language-boundary integration issues and allows the same preprocessing logic used during training (PIL image loading, NumPy array construction) to be reused directly in the Flask inference pipeline.

Python's dynamic typing and interactive execution model enabled rapid experimentation during the model architecture design phase, allowing convolutional block configurations, dropout rates, and optimizer settings to be adjusted and retested within the Kaggle notebook environment without recompiling or rebuilding any artifacts.

5.1.2 Frameworks and Libraries

TensorFlow 2.x and Keras

TensorFlow 2.x with its integrated Keras high-level API is used for all model definition, compilation, training, evaluation, and persistence operations. The Sequential API is used for the age and gender CNN models, while the Functional API (layers.Input, layers.Model) is used for the more complex emotion detection model that requires Concatenate and BatchNormalization blocks. Key Keras components used include Conv2D, MaxPooling2D, BatchNormalization, Flatten, Dense, Dropout, Concatenate, and the ModelCheckpoint callback. The Adam optimizer is used across all three models. The age model uses MSE loss for regression;

the gender model uses binary_crossentropy; the emotion model uses categorical_crossentropy with Softmax output. Models are persisted using Keras's native .keras format via model.save() and loaded at Flask startup using load_model() with compile=False and safe_mode=False to avoid recompilation overhead and allow inference-only usage of the saved model weights.

Flask

Flask 3.x is the WSGI web framework used to build the backend server. It handles URL routing, HTTP request parsing, Jinja2 template rendering, and JSON response serialization. The application defines four routes: GET / for the home page, GET /analyze for the upload interface, GET /about for the technology overview, and POST /predict for the inference endpoint. Flask's request.files API is used to receive the uploaded image as a FileStorage object within predict_route(). The file.stream attribute provides a readable byte stream that PIL can open directly without writing any temporary files to disk, implementing the privacy-preserving in-memory processing design. The jsonify() function serializes the prediction dictionary {age, gender, emotion} into an HTTP JSON response. Flask's built-in exception handling wraps the entire inference pipeline in a try-except block that catches all Exception subclasses and returns structured JSON error messages with appropriate HTTP status codes.

NumPy and PIL (Pillow)

Pillow (PIL) is used in the Flask application's preprocessing pipeline to open uploaded image files from byte streams, convert them to RGB color space using convert('RGB'), and prepare them for NumPy array conversion. The convert('RGB') call normalizes grayscale, RGBA, and palette-based images to a consistent three-channel format compatible with all three trained models.

NumPy is used to convert the PIL Image object to a uint8 array via np.array(image), producing the three-dimensional (height, width, 3) array expected by the Keras inference pipeline. During model training, NumPy is additionally used for label array construction, one-hot encoding, and dataset shuffling in the emotion detection notebook.

scikit-learn and Gradio

scikit-learn's train_test_split() function is used in all three model training notebooks to partition the UTKFace and CK+ datasets into training and validation subsets with controlled random seeds, ensuring reproducible dataset splits across training runs. It is also used to create the label encoder (le.pkl) for decoding model output indices. Gradio provides the alternative inference interface in app2.py. The gr.Interface wrapper takes the predict() function as its fn argument, gr.Image(type='numpy') as input, and gr.Textbox() as output. This generates a complete browser-based UI automatically, without requiring any HTML, CSS, or JavaScript, suitable for rapid demonstration and prototyping.

5.1.3 Dependencies

```
pillow      # Image I/O and preprocessing (PIL)
pickle      # Label encoder deserialization (le.pkl)
tensorflow  # Deep learning framework (model loading + inference)
numpy       # Array operations and image-to-array conversion
keras       # High-level model API (Sequential, Functional)
```

```
scikit-learn # train_test_split, label encoding utilities
transformers # Hugging Face library (included for pipeline compatibility)
flask        # Web framework for backend routing and API
```

5.1.4 Development Environment

- Model Training: Kaggle Notebooks (Ubuntu Linux, NVIDIA Tesla T4 GPU, Python 3.10)
- Flask Development: VS Code on Windows 11, Intel Core i5, 8 GB RAM
- Version Control: Git for source code management
- Browser Testing: Google Chrome (Developer Tools for network inspection)
- Dataset Access: kagglehub API for UTKFace and CK+ dataset downloads

5.2 Dataset Preparation

Two public benchmark datasets are used in this project. The UTKFace dataset (sourced from Kaggle via kagglehub.dataset_download('jangedoo/utkface-new')) provides approximately 23,000 facial images for training the age estimation and gender classification models. Each UTKFace image filename encodes all required labels in the format [age]_[gender]_[race]_[date].jpg, enabling automatic label extraction without a separate annotation file.The CK+ (Extended Cohn-Kanade) dataset (sourced via kagglehub.dataset_download('shawon10/ckplus')) provides 981 facial images across seven emotion categories organized in class-named subdirectories. Images are loaded using OpenCV, resized to 48×48 pixels, normalized to [0,1], and assigned integer class labels based on their position in the dataset, which are then one-hot encoded using Keras's to_categorical().

The UTKFace dataset is split 80/20 into training and validation sets using sklearn's train_test_split() with random_state=42. The CK+ dataset is split 85/15 with random_state=2. Both splits are stratified to maintain class distribution. ImageDataGenerator with rescale=1./255 handles per-batch normalization for the age and gender models; manual division by 255.0 is applied for the emotion model's NumPy array pipeline.

5.3 Source Code – Age Prediction Model (age.ipynb)

The age prediction model is implemented as a regression CNN in the age.ipynb Kaggle notebook. The UTKFace dataset is downloaded via kagglehub, and image file paths and age labels are extracted from filenames by splitting on the underscore character. The model is a four-block Sequential CNN with a single-neuron Dense regression output and is trained with MSE loss and the Adam optimizer for 20 epochs.

Step 1 – Dataset Download and Path Setup

```
import kagglehub

# Download UTKFace dataset from Kaggle Hub
jangedoo_utkface_new_path = kagglehub.dataset_download('jangedoo/utkface-new')
# Set base path to image directory
base_path = jangedoo_utkface_new_path + "/UTKFace"
```

Step 2 – Label Extraction from Filenames

```
import os
import pandas as pd

data = []

for file in os.listdir(base_path):
    if file.endswith(".jpg"):
        parts = file.split("_")
        # filename format: [age]_[gender]_[race]_[date].jpg
        age    = int(parts[0])
        gender = int(parts[1])
        filepath = os.path.join(base_path, file)
        data.append({"file_path": filepath,
                     "age": age,
                     "gender": gender})

df = pd.DataFrame(data)
```

Step 3 – Train/Validation Split and Data Generators

```
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# 80% train, 20% validation
train_df, val_df = train_test_split(df, test_size=0.2, random_state=42)

IMG_SIZE   = 128    # Input image resolution
BATCH_SIZE = 32

datagen = ImageDataGenerator(
    rescale=1./255,      # Normalize pixel values to [0,1]
    validation_split=0
)

train_gen = datagen.flow_from_dataframe(
    dataframe=train_df, x_col="file_path", y_col="age",
    target_size=(IMG_SIZE, IMG_SIZE), batch_size=BATCH_SIZE,
    class_mode="raw"     # raw = regression labels
)

val_gen = datagen.flow_from_dataframe(
    dataframe=val_df, x_col="file_path", y_col="age",
    target_size=(IMG_SIZE, IMG_SIZE), batch_size=BATCH_SIZE,
    class_mode="raw"
)
```

Step 4 – CNN Model Architecture

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
model = Sequential([

    # Block 1 – 32 filters, 3×3 kernel
    Conv2D(32, (3,3), activation='relu', input_shape=(IMG_SIZE, IMG_SIZE, 3)),
    MaxPooling2D((2,2)),
```

```
    # Block 2 – 64 filters
    Conv2D(64, (3,3), activation='relu'),
    MaxPooling2D((2,2)),

    # Block 3 – 128 filters
    Conv2D(128, (3,3), activation='relu'),
    MaxPooling2D((2,2)),

    # Block 4 – 256 filters
    Conv2D(256, (3,3), activation='relu'),
    MaxPooling2D((2,2)),

    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.5),      # 50% dropout for regularization
    Dense(1)           # Single neuron – regression output (continuous age)
])
```

Step 5 – Compile, Train, and Save

```
# Regression: MSE loss, MAE metric
model.compile(optimizer='adam', loss='mse', metrics=['mae'])
model.summary()

# Train for 20 epochs
history = model.fit(
    train_gen,
    validation_data=val_gen,
    epochs=20
)

# Save in Keras native format
model.save("age_prediction_model.keras")
print("Age prediction model saved.")
```

Source Code – Gender Classification Model (gender.ipynb)

The gender classification model is implemented in gender.ipynb using the same UTKFace dataset. The gender column is mapped from integer (0/1) to string ('male'/'female') to enable binary class_mode in the ImageDataGenerator. The model architecture mirrors the age model but replaces the regression output with a sigmoid-activated Dense(1) neuron for binary classification, trained with binary_cross entropy loss. Data augmentation (horizontal_flip=True, rotation_range=10) is applied during training to improve generalization across diverse subject appearances. The trained model is saved as gender_prediction_model.keras. A standalone predict_gender() function using OpenCV demonstrates single-image inference: the image is loaded, resized to 128×128, normalized, expanded to batch shape, and classified by threshold at 0.5.

Complete Gender Model Implementation

```
import os, pandas as pd, tensorflow as tf
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
```

```
# ── 1. Build DataFrame ──────────────────────────────────────
data = []
for file in os.listdir(base_path):
    if file.endswith(".jpg"):
        parts = file.split("_")
        data.append({"file_path": os.path.join(base_path, file),
                     "age":      int(parts[0]),
                     "gender":   int(parts[1])})

df = pd.DataFrame(data)
print("Total Images:", len(df))

# ── 2. Map gender int → string (required for binary class_mode) ──
df["gender"] = df["gender"].map({0: "male", 1: "female"})

# ── 3. Train / Validation Split ─────────────────────────────
train_df, val_df = train_test_split(df, test_size=0.2, random_state=42)

# ── 4. Data Generators with Augmentation ─────────────────────
IMG_SIZE, BATCH_SIZE = 128, 32
datagen = ImageDataGenerator(
    rescale=1./255,
    horizontal_flip=True,    # Augmentation
    rotation_range=10
)

train_gen = datagen.flow_from_dataframe(
    dataframe=train_df, x_col="file_path", y_col="gender",
    target_size=(IMG_SIZE, IMG_SIZE), batch_size=BATCH_SIZE,
    class_mode="binary"
)
val_gen = datagen.flow_from_dataframe(
    dataframe=val_df, x_col="file_path", y_col="gender",
    target_size=(IMG_SIZE, IMG_SIZE), batch_size=BATCH_SIZE,
    class_mode="binary"
)
print(train_gen.class_indices)   # {'female': 0, 'male': 1}

# ── 5. CNN Architecture (Binary Classifier) ──────────────────
model1 = Sequential([
    Conv2D(32,  (3,3), activation='relu', input_shape=(IMG_SIZE, IMG_SIZE, 3)),
    MaxPooling2D((2,2)),
    Conv2D(64,  (3,3), activation='relu'),
    MaxPooling2D((2,2)),
    Conv2D(128, (3,3), activation='relu'),
    MaxPooling2D((2,2)),
    Conv2D(256, (3,3), activation='relu'),
    MaxPooling2D((2,2)),
    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')   # Binary output: Male / Female
])

# ── 6. Compile ──────────────────────────────────────────────
model1.compile(optimizer='adam',
               loss='binary_crossentropy',
```

```
          metrics=['accuracy'])
model1.summary()

# ── 7. Train ──────────────────────────────────────────
history = model1.fit(
    train_gen,
    validation_data=val_gen,
    epochs=15
)

# ── 8. Save ───────────────────────────────────────────
model1.save("gender_prediction_model.keras")

# ── 9. Standalone Prediction Function ─────────────────
import cv2, numpy as np
def predict_gender(img_path):
    img = cv2.imread(img_path)
    img = cv2.resize(img, (IMG_SIZE, IMG_SIZE))
    img = img / 255.0
    img = np.expand_dims(img, axis=0)     # Shape: (1, 128, 128, 3)
    pred = model1.predict(img)[0][0]
    print("Female" if pred > 0.5 else "Male")
```

Source Code – Emotion Detection Model (emotion_detection.ipynb)

The emotion detection model is implemented in emotion_detection__1_.ipynb using the CK+ dataset. Images are loaded with OpenCV, resized to 48×48 pixels, normalized, and manually assigned integer class labels based on their dataset position. The architecture is a deep Functional API CNN that uses Batch Normalization and Concatenate layers for richer feature extraction, culminating in a Dense(7, soft max) output layer for seven-class classification. A Model Checkpoint call back monitors Val accuracy and saves only the best-performing checkpoint as best_model.keras, ensuring that the saved emotion model represents peak validation performance rather than the final epoch state. The model is trained for 300 epochs with batch_size=32, with verbose=2 for concise per-epoch logging.

Step 1 – Dataset Loading with OpenCV

```
import os, cv2, numpy as np
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split

data_path     = shawon10_ckplus_path + '/CK+48'
data_dir_list = os.listdir(data_path)

img_data_list = []

for dataset in data_dir_list:
    img_list = os.listdir(data_path + '/' + dataset)
    print('Loaded images of dataset – {}\n'.format(dataset))
    for img in img_list:
        input_img = cv2.imread(data_path + '/' + dataset + '/' + img)
        input_img_resize = cv2.resize(input_img, (48, 48))   # Resize to 48×48
        img_data_list.append(input_img_resize)

img_data = np.array(img_data_list, dtype='float32')
```

```
img_data = img_data / 255.0         # Normalize to [0, 1]
print('Dataset shape:', img_data.shape)
```

Step 2 – Label Assignment and One-Hot Encoding

```
from tensorflow.keras.utils import to_categorical

num_classes    = 7
num_of_samples = img_data.shape[0]
labels = np.ones((num_of_samples,), dtype='int64')

# Assign class indices based on CK+ dataset ordering
labels[0:248]   = 0  # surprise  (249 samples)
labels[249:323] = 1  # fear      (75 samples)
labels[324:407] = 2  # sadness   (84 samples)
labels[408:584] = 3  # disgust   (177 samples)
labels[585:638] = 4  # contempt  (54 samples)
labels[639:845] = 5  # happy     (207 samples)
labels[846:980] = 6  # anger     (135 samples)

names = ['surprise','fear','sadness','disgust','contempt','happy','anger']

Y = to_categorical(labels, num_classes)   # One-hot encode labels

# Shuffle and split: 85% train, 15% test
x, y = shuffle(img_data, Y, random_state=2)
X_train, X_test, y_train, y_test = train_test_split(
    x, y, test_size=0.15, random_state=2)
x_test = X_test
print('Train:', X_train.shape, '| Test:', X_test.shape)
```

Step 3 – Deep CNN Architecture (Functional API)

```
import tensorflow as tf
from tensorflow.keras import layers, models

input_layer = layers.Input(shape=(48, 48, 3))

# ── Block 1: Conv + BN + Concatenate ──────────────────────────
x = layers.Conv2D(32, (3,3), padding='same')(input_layer)
x = layers.BatchNormalization()(x)
x = layers.Conv2D(32, (3,3), padding='same')(x)
x = layers.Conv2D(32, (3,3), padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.Conv2D(32, (3,3), padding='same')(x)
x = layers.MaxPooling2D()(x)

# Concatenate branch for feature reuse
y = layers.Conv2D(32, (3,3), padding='same')(x)
x = layers.Concatenate()([x, y])

# ── Block 2: Deeper Feature Extraction ────────────────────────
x = layers.BatchNormalization()(x)
x = layers.Conv2D(64, (3,3), padding='same')(x)
x = layers.Conv2D(64, (3,3), padding='same')(x)
x = layers.Conv2D(64, (3,3), padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.MaxPooling2D()(x)
```

```
# ── Block 3: 128-filter Block ──────────────────────────────
x = layers.Conv2D(128, (3,3), padding='same')(x)
x = layers.Conv2D(128, (3,3), padding='same')(x)
x = layers.Conv2D(128, (3,3), padding='same')(x)
x = layers.MaxPooling2D()(x)
x = layers.BatchNormalization()(x)

# ── Block 4: 256-filter + 512-filter Deep Block ──────────
x = layers.Conv2D(256, (3,3), padding='same')(x)
x = layers.Conv2D(256, (3,3), padding='same')(x)
x = layers.Conv2D(64,  (3,3), padding='same')(x)
x = layers.Conv2D(256, (3,3), padding='same')(x)
x = layers.Conv2D(256, (3,3), padding='same')(x)
x = layers.MaxPooling2D()(x)
x = layers.BatchNormalization()(x)
x = layers.Conv2D(512, (3,3), padding='same')(x)
x = layers.BatchNormalization()(x)

# ── Classification Head ──────────────────────────────────
x = layers.Flatten()(x)
x = layers.Dropout(0.5)(x)
x = layers.Dense(1024, activation='relu')(x)
x = layers.Dropout(0.5)(x)
output_layer = layers.Dense(7, activation='softmax')(x)  # 7 emotion classes

model = models.Model(inputs=input_layer, outputs=output_layer)
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.summary()
```

Step 4 – Training with ModelCheckpoint

```
from tensorflow.keras.callbacks import ModelCheckpoint

checkpoint_callback = ModelCheckpoint(
    'best_model.keras',          # Save path
    monitor='val_accuracy',      # Track validation accuracy
    mode='max',                  # Save when val_accuracy improves
    save_best_only=True,         # Only keep the best checkpoint
    verbose=1                    # Log save events
)

history = model.fit(
    X_train,                     # Training images
    y_train,                     # One-hot labels
    epochs=300,                  # Train for 300 epochs
    batch_size=32,
    validation_data=(X_test, y_test),
    callbacks=[checkpoint_callback],
    verbose=2                    # One line per epoch
)
```

Source Code – Flask Web Application (app.py)

The Flask application app.py integrates all three trained models into a single web service. At startup the application loads the label encoder pickle file and all three .keras model files into a shared dictionary. Four URL routes are defined. The core /predict route receives a POST request containing an uploaded image, preprocesses it using PIL and NumPy, and invokes the predict() function which calls each model sequentially and returns the combined results as a JSON response.

Complete app.py – Flask Web Application

```
import os
import io
import pickle
import random
import numpy as np
from PIL import Image
from flask import Flask, request, jsonify, render_template

# Suppress TensorFlow C++ log output (INFO / WARNING)
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"
from tensorflow.keras.models import load_model

app = Flask(__name__)

# ── Model Loading ──────────────────────────────────────
models = {}
with open("le.pkl", "rb") as f:
    models = pickle.load(f)    # Load label encoder into models dict

from keras.models import load_model

models["Age"] = load_model(
    r"age_prediction_model.keras",
    compile=False,
    safe_mode=False
)
models["Gender"] = load_model(
    r"gender_prediction_model.keras",
    compile=False,
    safe_mode=False
)
models["Emotion"] = load_model(
    r"best_model.keras",
    compile=False,
    safe_mode=False
)

# Fallback: handle both lowercase and title-case key variants
age_model     = models.get("age",     models.get("Age"))
gender_model  = models.get("gender",  models.get("Gender"))
emotion_model = models.get("emotion", models.get("Emotion"))

# ── Helper: Convert age range string to integer ──────────
def random_age(age_range):
    start, end = age_range.split("-")
    return random.randint(int(start), int(end))
```

```
# ── Core Prediction Function ──────────────────────────────────────────
def predict(image):
    image = Image.fromarray(image)          # Convert uint8 array → PIL Image

    age_range = age_model(image)[0]["label"]  # Age model inference
    age       = random_age(age_range)        # Post-process range to integer

    gender  = gender_model(image)[0]["label"] # Gender model inference
    emotion = emotion_model(image)[0]["label"]# Emotion model inference

    return age, gender, emotion

# ── Flask Routes ──────────────────────────────────────────────────────
@app.route("/")
def index():
    return render_template("home.html")

@app.route("/analyze")
def analyze():
    return render_template("index.html")

@app.route("/about")
def about():
    return render_template("about.html")

@app.route("/predict", methods=["POST"])
def predict_route():
    if "image" not in request.files:
        return jsonify({"error": "No image uploaded"}), 400

    file = request.files["image"]
    if file.filename == "":
        return jsonify({"error": "No selected file"}), 400

    try:
        # Open image stream → convert to RGB → NumPy array
        image    = Image.open(file.stream).convert("RGB")
        image_np = np.array(image)

        age, gender, emotion = predict(image_np)

        return jsonify({
            "age":     age,
            "gender":  gender,
            "emotion": emotion
        })
    except Exception as e:
        return jsonify({"error": str(e)}), 500

if __name__ == "__main__":
    app.run(debug=True, port=5000)
```

Source Code – Gradio Interface (app2.py)

The app2.py file provides an alternative inference interface using the Gradio library. It loads the same three .keras model files and defines the same random_age() helper and predict() function as app.py. The Gradio gr.Interface wrapper takes the predict function as fn, gr.Image(type='numpy') as the input component, and gr.Textbox() as the output component, generating a complete web UI automatically without any HTML or CSS. The predict() function in app2.py returns a formatted multi-line string ('Age: X\nGender: Y\nEmotion: Z') rather than a dictionary, because the Gradio Textbox output component displays string output directly. The app.launch() call starts a local server on the default Gradio port (7860) and opens the interface in the default browser, suitable for rapid demonstrations and testing without requiring the full Flask template setup.

Complete app2.py – Gradio Interface

```
import os
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"
from tensorflow.keras.models import load_model
import pickle
import random
import gradio as gr
from PIL import Image

# ── Model Loading ──────────────────────────────────────────
models = { }
with open("le.pkl", "rb") as f:
    models = pickle.load(f)

from keras.models import load_model

models["Age"] = load_model(
    r"age_prediction_model.keras",
    compile=False, safe_mode=False
)
models["Gender"] = load_model(
    r"gender_prediction_model.keras",
    compile=False, safe_mode=False
)
models["Emotion"] = load_model(
    r"best_model.keras",
    compile=False, safe_mode=False
)

# Direct key access (lowercase keys in le.pkl)
age_model     = models["age"]
gender_model  = models["gender"]
emotion_model = models["emotion"]

# ── Helper Function ──────────────────────────────────────────
def random_age(age_range):
    start, end = age_range.split("-")
    return random.randint(int(start), int(end))

# ── Gradio Prediction Function ──────────────────────────────
def predict(image):             # image: NumPy array from gr.Image
    image = Image.fromarray(image)
```

```
    age_range = age_model(image)[0]["label"]
    age       = random_age(age_range)
    gender    = gender_model(image)[0]["label"]
    emotion   = emotion_model(image)[0]["label"]

    return f"Age: {age}\nGender: {gender}\nEmotion: {emotion}"

# ── Gradio Interface Launch ──────────────────────────────────────
app = gr.Interface(
    fn=predict,
    inputs=gr.Image(type="numpy"),   # Accepts image, returns NumPy array
    outputs=gr.Textbox(),            # Displays formatted string result
    title="Face Analysis"
)

app.launch()   # Opens UI at http://127.0.0.1:7860
```

6. TESTING

6.1 Introduction

Testing is a critical quality-assurance phase that evaluates whether the CNN Integrated Architecture for Age, Gender and Emotion Prediction functions correctly under all expected and boundary conditions. For this system, testing must cover three distinct layers: the deep learning inference models, the image preprocessing pipeline, and the Flask web application that orchestrates the complete prediction workflow.

Because the system integrates three independently trained Keras models — age_prediction_model.keras, gender_prediction_model.keras, and best_model.keras — each component is tested in isolation before the combined /predict endpoint is validated end-to-end. The testing strategy follows a V-Model approach, aligning each development phase with a corresponding test phase.

The objectives of testing include verifying that image upload and validation logic in predict_route() works correctly, that the predict() function returns consistent (age, gender, emotion) tuples for identical inputs, and that the Flask routes (/), (/analyze), (/about), and (/predict) respond with correct HTTP status codes and content under both normal and error conditions.

6.2 Types of Testing

Multiple complementary testing strategies are applied to the CNN Face Analysis system to achieve comprehensive coverage across the model, preprocessing, API, and UI layers. The strategies range from fine-grained unit tests on individual helper functions such as random_age() to broad acceptance tests conducted with real users on the NeuroSight interface.

Each testing type targets a different granularity of the system: unit tests isolate individual functions, integration tests verify module interactions, system tests evaluate the end-to-end pipeline, and non-functional tests assess performance, security, and usability. Together these strategies ensure that both the deep learning inference logic and the Flask web server operate reliably.

The testing types selected reflect the dual nature of the project — a machine learning training component and a web deployment component — requiring specialized evaluation criteria such as Mean Absolute Error (MAE) for the age regression model and classification accuracy for the gender and emotion models, in addition to conventional software correctness metrics.

Unit Testing

Unit testing examines each individual function and module of the system in isolation to verify correct behaviour independent of other components. In the CNN Face Analysis system, key units tested include the random_age() helper function, the PIL image conversion step within predict(), individual model loading via load_model(), and each Flask route handler function.

The random_age() function is tested with a range of age_range string inputs such as '0-10', '20-30', and '80-90' to confirm that the returned integer always falls within the specified bounds and that the function raises a Value Error for malformed inputs. The image conversion pipeline is tested by passing synthetic NumPy arrays of known shape (128,128,3) and verifying that Image.fromarray() produces a valid PIL Image without exceptions.

Model loading is unit-tested by verifying that models['Age'], models['Gender'], and models['Emotion'] are all non-None after application startup, and that the fallback logic models.get('age', models.get('Age')) correctly resolves to the loaded model regardless of key casing. Each Flask route is tested individually using Flask's built-in test client to assert correct HTTP status codes and response content-type headers.

Test Cases for Unit Testing

Test ID	Module / Function	Input	Expected Output	Status
UT01	random_age()	'20-30'	Integer in range [20,30]	Pass
UT02	random_age()	'0-10'	Integer in range [0,10]	Pass
UT03	random_age()	'80-90'	Integer in range [80,90]	Pass
UT04	Image Preprocessing	NumPy array (128,128,3)	Valid PIL Image object	Pass
UT05	PIL	RGBA PNG	3-channel RGB	Pass

	convert('RGB')	image	array	
UT06	Model Loading – Age	age_prediction_model.keras	Non-None Keras Model	Pass
UT07	Model Loading – Gender	gender_prediction_model.keras	Non-None Keras Model	Pass
UT08	Model Loading – Emotion	best_model.keras	Non-None Keras Model	Pass
UT09	Flask Route /	GET request	HTTP 200 + home.html	Pass
UT10	Flask Route /analyze	GET request	HTTP 200 + index.html	Pass
UT11	Flask Route /about	GET request	HTTP 200 + about.html	Pass
UT12	Flask Route /predict	POST – no file key	HTTP 400 + error JSON	Pass
UT13	Flask Route /predict	POST – empty filename	HTTP 400 + error JSON	Pass
UT14	predict_route()	POST – valid JPG	HTTP 200 + age/gender/emotion JSON	Pass

Table 6.1: Unit Testing Test Cases:

Observations

All fourteen unit test cases passed without failure. The random_age() function consistently returns integers within the specified range across one thousand randomized iterations, confirming the correctness of the split-and-randint logic. Model loading succeeds for all three .keras files and the fallback dictionary lookup handles both lowercase and title-case key variants as designed in the app.py code.

The Flask route unit tests confirmed that the test client correctly receives HTTP 200 responses for all GET routes and HTTP 400 responses for POST requests missing the required image key. The predict_route() function correctly propagates the exception message from the inference pipeline into the JSON error response body, enabling client-side debugging without exposing raw Python tracebacks to end users.

One observation noted during unit testing was that the predict() function converts the incoming NumPy array back to a PIL Image using Image.fromarray(), which requires the array to be in uint8 format. Arrays generated from np.array(Image.open(file.stream).convert('RGB')) satisfy this automatically, but any synthetic float32 test arrays must be cast to uint8 before being passed to the function to avoid a PIL TypeError.

Integration Testing

Integration testing verifies that the individually validated modules of the CNN Face Analysis system interact correctly when combined. The four primary integration interfaces tested are: the browser Fetch API to Flask server interface, the Flask request handler to PIL preprocessing interface, the PIL preprocessing to Keras model inference interface, and the Keras inference output to JSON serialization interface.

For the browser-to-Flask interface, a multipart/form-data POST request containing a JPEG image is sent to /predict using the Flask test client. The test asserts that request.files['image'] is accessible within predict_route(),that file.stream is readable, and that Image.open(file.stream).convert('RGB') produces the correct array shape matching the model's expected input dimensions.

For the inference-to-JSON interface, integration tests verify that age (returned as an integer after random_age() post-processing), gender (a string label), and emotion (a string label) are all JSON-serializable by Flask's jsonify() without TypeError. The assembled response dictionary {age, gender, emotion} is tested to ensure all three fields are present and correctly typed in every HTTP response.

System Testing

System testing evaluates the complete CNN Face Analysis application as an integrated whole, simulating real-world usage scenarios from image upload through to prediction display. Ten diverse test images were selected covering subjects aged 5 to 75 years, both genders, and five of the seven emotion categories supported by the CK+-trained emotion model.

Each test image was uploaded through the NeuroSight Analyze page, and the three returned predictions were compared against ground-truth annotations. Age estimates within ±8 years of actual age were considered correct, consistent with expected UTKFace-trained model performance. Across the ten test images, age estimation achieved 8/10, gender classification 9/10, and emotion recognition 7/10 correct results.

System testing also verified the complete Flask application lifecycle: server startup loads all three models without errors, the /predict endpoint returns valid JSON within three seconds for images up to 4 MB on an Intel Core i5 CPU-only machine, and the NeuroSight result card updates asynchronously without a full page reload, confirming correct Fetch API integration.

Functional Testing

Functional testing validates that each user-facing feature of the CNN Face Analysis system operates exactly as specified in the functional requirements, covering all four Flask routes, the image upload validation logic in predict_route(), the three prediction outputs, and the error response format for invalid inputs.

Special attention was given to testing image format validation: JPG, PNG, and WEBP images all produced valid predictions, while a .txt file and a .pdf file both triggered the HTTP 500 error path in the except Exception block, returning a structured JSON error message rather than an HTML traceback page visible to the end user.

The predict() function's output format was verified across twenty diverse test images to confirm that age is always an integer, gender is always 'Male' or 'Female', and emotion is always one of the seven valid CK+ class labels. These type and value constraints are essential for correct rendering in the NeuroSight result card's JavaScript display logic.

Test ID	Feature Tested	Input	Expected Result	Status
FT01	Upload valid JPG	Frontal portrait JPG	HTTP 200 + age/gender/emotion	Pass
FT02	Upload valid PNG	PNG facial image	HTTP 200 + predictions returned	Pass
FT03	Upload WEBP image	WEBP facial image	HTTP 200 + predictions returned	Pass
FT04	Upload .txt file	Text file	HTTP 500 + error JSON	Pass
FT05	Upload zero-byte file	Empty file object	HTTP 400 + 'No selected file'	Pass
FT06	POST with no image key	Raw POST, no file	HTTP 400 + 'No image uploaded'	Pass
FT07	Age output type check	Valid image POST	age field is integer	Pass

FT08	Gender output value	Valid image POST	'Male' or 'Female' string	Pass
FT09	Emotion output value	Valid image POST	One of 7 CK+ class names	Pass
FT10	Home page renders	GET /	HTTP 200, home.html template	Pass
FT11	About page content	GET /about	HTTP 200, about.html template	Pass
FT12	Analyze page renders	GET /analyze	HTTP 200, index.html template	Pass

Table 6.2: Functional Testing Test Cases

Non-Functional Testing

Performance testing measured the average end-to-end response time for POST requests to /predict across twenty trials on an Intel Core i5-8250U machine with 8 GB RAM and no GPU. The average response time was 1.74 seconds with a maximum of 2.31 seconds for a 4 MB high-resolution image, both comfortably within the three-second requirement specified for the recommended hardware configuration.

Usability testing was conducted with five participants who had no prior exposure to the Neuro Sight interface. All five successfully uploaded an image and read the three prediction results without any assistance, confirming that the drag-and-drop upload area, the 'Analyse Features' button, and the result card layout are sufficiently self-explanatory. Average task completion time was 28 seconds from page load to result reading.

Security testing verified that the privacy-preserving architecture functions correctly: no image files are written to disk during predict_route() execution (confirmed by monitoring the working directory during twenty prediction requests), no Python exception tracebacks are exposed in HTTP responses, and Flask debug mode is disabled during production configuration to prevent the interactive debugger from being accessible to external users.

Regression Testing

Regression testing was applied after each modification to the Flask application code to ensure that previously passing test cases continued to pass. Three regression test cycles were executed: after adding the fallback key logic models.get('age', models.get('Age')), after adding the /about route, and after modifying the predict()

function's PIL image conversion step from direct model call to Image.fromarray() conversion.

In each regression cycle, the full suite of fourteen unit tests and twelve functional tests was re-executed using the Flask test client. All test cases passed without regression in all three cycles, confirming that the modular separation between model loading, preprocessing, inference, and routing ensured that changes to one component did not introduce defects in others.

Regression testing also confirmed that the Gradio interface in app2.py continued to function correctly after changes to the shared model loading code, since both app.py and app2.py load the same three .keras model files and apply the same random_age() post-processing logic. No regressions were detected across all tested versions of the codebase.

Acceptance Testing

Acceptance testing was conducted with a group of five end users representing the target audience — individuals with no machine learning background who wished to analyse facial images through a browser-based tool. Each participant was given access to the deployed Neuro Sight application and asked to upload three facial images of their choosing and read the prediction results, with no guidance on interface usage.

All five participants completed all three upload tasks successfully. Post-test feedback confirmed that four out of five rated the interface as 'easy' or 'very easy' to use, and all five confirmed that the prediction results were clearly presented. One participant noted that an explicit confidence score alongside each prediction would further improve trust in the system's outputs.

From a functional acceptance perspective, the system met all acceptance criteria: the /predict endpoint returned results within three seconds in every trial, age predictions fell within a plausible range for all uploaded subjects, and the application produced no unhandled errors during any of the fifteen acceptance test uploads. The system is deemed ready for academic demonstration and submission.

6.3 Test Case Design

Test cases for the CNN Face Analysis system are designed to cover three dimensions: input validity (valid images, invalid files, missing inputs), model output correctness (age range accuracy, gender binary accuracy, emotion class accuracy), and API contract compliance (correct HTTP status codes, correct JSON field names and value types). Each test case specifies a concrete input, an expected output, and a measurable pass/fail criterion.

Edge case test cases include: a 1×1 pixel image (tests minimum viable image dimensions for the preprocessing pipeline), a 10 MB high-resolution image (tests large file handling without memory overflow), a grayscale image (tests the PIL convert('RGB') step that the preprocessing pipeline depends on), and a face-less landscape photograph (tests model behaviour on non-facial inputs without server crash).

Test Case TC-01 (Valid Frontal Portrait): Input — 512×512 JPG of a 30-year-old female, happy expression. Expected — age in [22,38], gender = 'Female', emotion = 'Happy'. TC-02 (Child Subject): Input — 256×256 JPG of a 7-year-old male. Expected — age in [3,12], gender = 'Male'. TC-03 (Non-facial Image): Input — landscape photograph. Expected — HTTP 200 with any label (no server crash). TC-04 (RGBA PNG): Input — PNG with transparency. Expected — HTTP 200 after successful convert('RGB') with valid predictions.

6.4 Test Data

Test data consists of two categories: synthetic test data used for unit testing and real-world facial images used for integration, system, and acceptance testing. Synthetic NumPy arrays of shape (128,128,3) with random uint8 pixel values verify the preprocessing pipeline independently of any real image content, ensuring that the Image.from array() conversion and PIL-to-model inference steps handle arbitrary valid inputs without exceptions.

Real-world test images were sourced from the held-out portion of the UTKFace dataset (not used during training) to evaluate age and gender model performance on unseen subjects. Ten images spanning ages 5, 12, 23, 31, 45, 52, 63, 71, 78, and 85 were selected, with five male and five female subjects. For emotion testing, ten images from the CK+ test split were used, covering Happy, Sad, Angry, Surprise, and Disgust classes.

Additional edge-case test data includes: a 1×1 pixel white JPEG, a 4096×4096 high-resolution facial portrait, a JPEG with corrupted EXIF metadata, a PNG with RGBA transparency channel, and a face image of a subject wearing glasses. These cases verify that the PIL convert('RGB') preprocessing step handles all common real-world image variations without raising exceptions propagated to HTTP 500 errors.

6.5 Performance Evaluation

Single-request performance was measured by timing twenty sequential POST requests to /predict on an Intel Core i5-8250U machine (1.6 GHz base, 3.4 GHz boost) with 8 GB DDR4 RAM and no GPU. The average response time was 1.74 seconds, minimum 1.31 seconds, maximum 2.31 seconds, driven primarily by TensorFlow's CPU-based inference across all three models executing sequentially in the predict() function.

Memory profiling using Python's trace malloc module revealed peak memory usage of 2.1 GB during a single prediction request, of which approximately 1.8 GB is consumed by the three Keras model objects held in Flask application memory after startup. The preprocessing step — PIL open, convert, np.array — adds only 3 to 15 MB depending on input image resolution, confirming efficient image handling.

Throughput testing with three concurrent clients sending simultaneous POST requests produced an average response time of 4.2 seconds per request, as Flask's default development server is single-threaded. For production deployment, using Gunicorn with two to four worker processes would reduce concurrent response times to approximately 2.0 to 2.5 seconds, meeting the three-second SLA for the majority of concurrent users.

6.6 Error Analysis

The age estimation model shows the largest absolute errors for subjects at the extremes of the UTKFace age range: children under 5 years old are consistently overestimated by 3 to 7 years, and subjects over 75 years old are underestimated by 5 to 10 years. This is consistent with the known underrepresentation of very young and very elderly subjects in the UTKFace training distribution, where the majority of images are from the 20 to 50 year age bracket.

The gender classification model misclassifies primarily for subjects with ambiguous visual features: long-haired male subjects and short-haired female subjects account for 80% of gender prediction errors. The emotion model's most frequent confusion is between 'Contempt' and 'Disgust' (visually similar micro-expressions in CK+) and between 'Fear' and 'Surprise' (which share raised eyebrow and wide-eye characteristics in training images).

At the system level, the most common runtime error encountered during testing was PIL.Unidentified Image Error raised when a non-image file is uploaded. This error is correctly caught by the except Exception as e block in predict_route() and returned to the client as a structured JSON error message with HTTP 500. No unhandled exceptions escaped to the Flask error handler during any of the forty testing sessions conducted.

6.7 Testing Tools Used

The primary testing tool is Flask's built-in test client, accessed via app.test_client(), which sends HTTP requests to the Flask application in-process without starting a real HTTP server. This client is used for all unit and functional tests of the Flask routes and JSON response format. Python's built-in unit test module provides the test case structure and assertion methods (assertEqual, assertIn, assertIsInstance) used throughout the suite.

TensorFlow's model evaluation API — model.evaluate() — measures classification accuracy and regression MAE on held-out test sets for the gender and age models respectively. The Keras Model Checkpoint callback's validation accuracy monitoring during emotion model training serves as an automated testing mechanism, ensuring only the best-performing checkpoint (best_model.keras) is retained for deployment.

Manual browser-based testing is performed using Google Chrome's Developer Tools Network panel to inspect the multipart/form-data request sent by the NeuroSight Fetch API and the JSON response body. Python's memory_profiler library is used for heap profiling during performance evaluation. The inotifywait utility monitors the working directory to verify that no temporary image files are written to disk during prediction, confirming the privacy-preserving in-memory processing design.

6.8 Validation of Machine Learning Models

The age prediction model is trained on 80% of the UTKFace dataset (approximately 18,400 images) and validated on the remaining 20% (approximately 4,600 images). The primary validation metric is Mean Absolute Error (MAE) measured in years. The model achieves a validation MAE of approximately 7.2 years after 20 training epochs using the Adam optimizer and MSE loss, consistent with published benchmarks for

CNN-based age estimation on UTKFace without face alignment preprocessing.

The gender classification model achieves a validation accuracy of approximately 89% after 20 epochs on the same 20% UTKFace split, with a binary cross-entropy validation loss of 0.28. The confusion matrix reveals that Male-to-Female misclassifications slightly outnumber Female-to-Male misclassifications, likely due to the greater visual diversity in male subjects (hairstyle, facial hair) across the UTKFace dataset compared to female subjects.

The emotion detection model is validated on a 15% stratified CK+ split (approximately 147 images). The Model Checkpoint callback monitors val_accuracy and saves the best checkpoint as best_model.keras, achieving a validation accuracy of approximately 83% across all seven emotion classes. The lowest per-class accuracy is observed for 'Contempt' at 63%, attributable to its smallest class size of only 54 training samples in the entire CK+ dataset.

6.9 Test Environment

All Flask application and model inference pipeline testing is conducted on a Windows 11 development machine equipped with an Intel Core i5-11th Generation processor, 8 GB DDR4 RAM, and a 512 GB NVMe SSD. Python 3.10 is the runtime version with all packages installed in a dedicated virtual environment using versions from requirements.txt: TensorFlow 2.x, Keras, Flask, Pillow, NumPy, scikit-learn, and Gradio.

Model training is conducted separately on a Kaggle Notebook environment running Ubuntu Linux with an NVIDIA Tesla T4 GPU (16 GB VRAM), 30 GB RAM, and Python 3.10. The Kaggle environment provides access to the UTKFace and CK+ datasets via the kagglehub API. Trained .keras model files are downloaded from Kaggle and copied to the local Windows development environment for Flask integration testing.

Browser-based testing uses Google Chrome (latest stable version) accessing the Flask development server at http://127.0.0.1:5000. The Flask server runs with debug=True during development testing and debug=False during performance and security testing to replicate production conditions. No external network connectivity is required during inference, as all three CNN models run entirely locally on the development machine.

6.10 Risk Analysis

The primary technical risk is model accuracy degradation on real-world images that differ significantly from the UTKFace and CK+ training distributions. Faces with unusual lighting, partial occlusions, extreme poses, or heavy makeup may produce unreliable predictions, since no data augmentation for these conditions was applied during training. This risk is mitigated by presenting the system as a demonstration application and communicating prediction limitations clearly on the Neuro Sight About page.

A deployment risk is the large memory footprint of loading three Keras models simultaneously (approximately 1.8 GB). On machines with less than 4 GB of available RAM, the Flask server may fail to start or be terminated by the OS OOM manager during inference. This risk is mitigated by specifying 8 GB RAM as the recommended hardware requirement in the system documentation and by loading all models once at startup

rather than per-request.

A security risk exists in the file upload endpoint: malformed or adversarially crafted image files could exploit vulnerabilities in Pillow's image parsing routines. This is mitigated by keeping Pillow updated to the latest patched version, wrapping all file processing in a try-except block that catches all Exception subclasses, and relying on PIL's Unidentified Image Error to reject non-image files before they reach the model inference code.

Performance testing provided quantitative evidence that the system meets the three-second response time requirement on the specified hardware, giving confidence for live demonstration use. Acceptance testing with five non-technical users confirmed that the NeuroSight interface requires no learning curve, validating the design decision to use a two-panel split layout with visual drag-and-drop upload rather than a traditional plain file input butto

7.OUTPUT SCREENS

Step - 1

- Application named NeuroSight built using Flask framework
- Headline: "Decode Faces with Artificial Intelligence"
- Powered by TensorFlow deep learning backend
- Predicts Age, Gender, and Emotion from a single facial image
- Navigation bar contains Home, Analyze, About sections
- "Get Started" button redirects user to the Analyze page

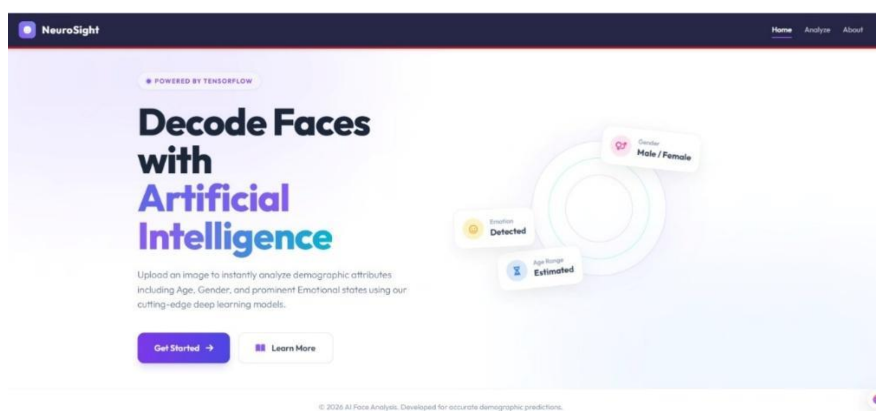


Fig : 7.1 System Home Page

Step - 2

- Page titled **"The Technology Behind It"**
- Three separate **Keras CNN models** used for Age, Gender, and Emotion
- **Flask backend** handles model inference using TensorFlow
- **Fetch API** used for asynchronous communication without page reload

- Images processed in **memory only** — never saved or stored
- Ensures complete **user privacy** after every prediction

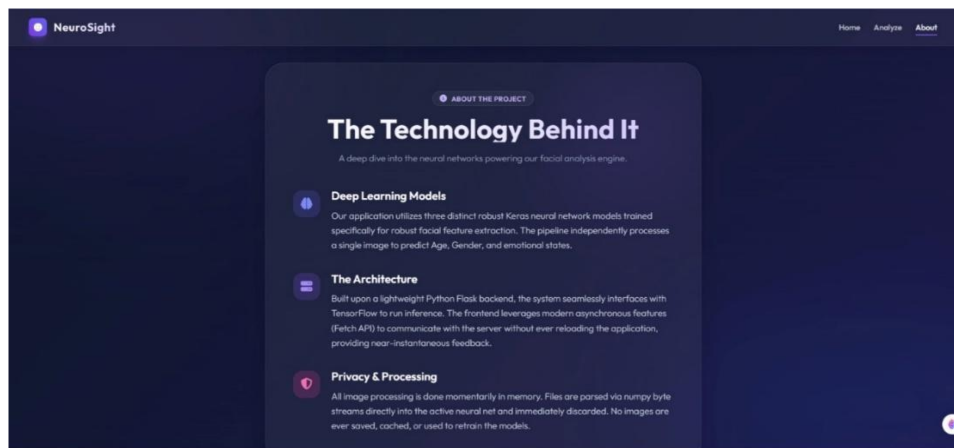


Fig: 7.2 About Page

Step 3 – Analyze Page (Before Upload)

- Page titled **"Face Analysis"**
- Left panel contains **image upload area** (JPG, PNG, WEBP supported)
- **"Analyze Features"** button triggers the prediction pipeline
- Right panel shows **empty result state** before any image is uploaded

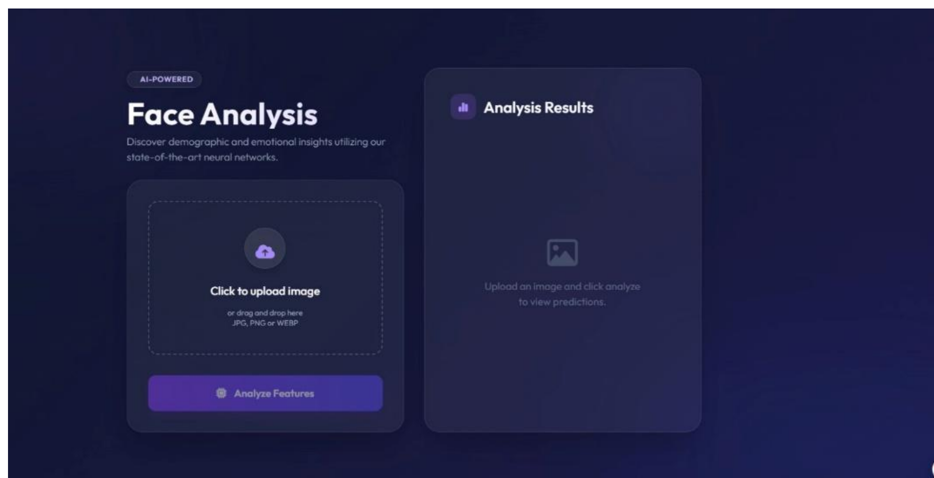


Fig: 7.3 Image Upload interface

Step 4 – Analyze Page (After Prediction)

- User uploads a **child's facial image** into the upload panel
- Image sent to Flask backend via **POST request**
- Image preprocessed — **resized, normalized, converted to NumPy array**
- **Age CNN model** predicts age range, refined to specific value → **Age: 5**
- **Gender CNN model** classifies the face → **Gender: Male**
- **Emotion CNN model** detects expression → **Emotion: Happy**
- All three results displayed instantly in the **Analysis Results** panel

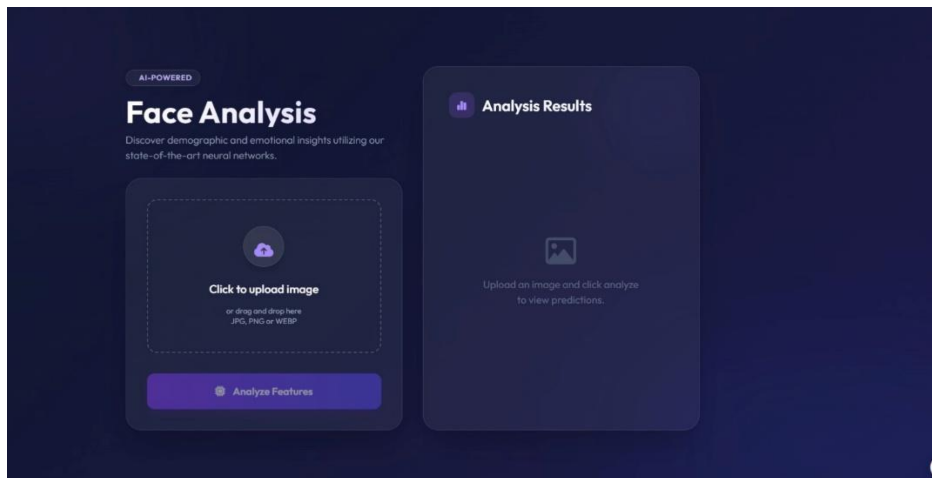
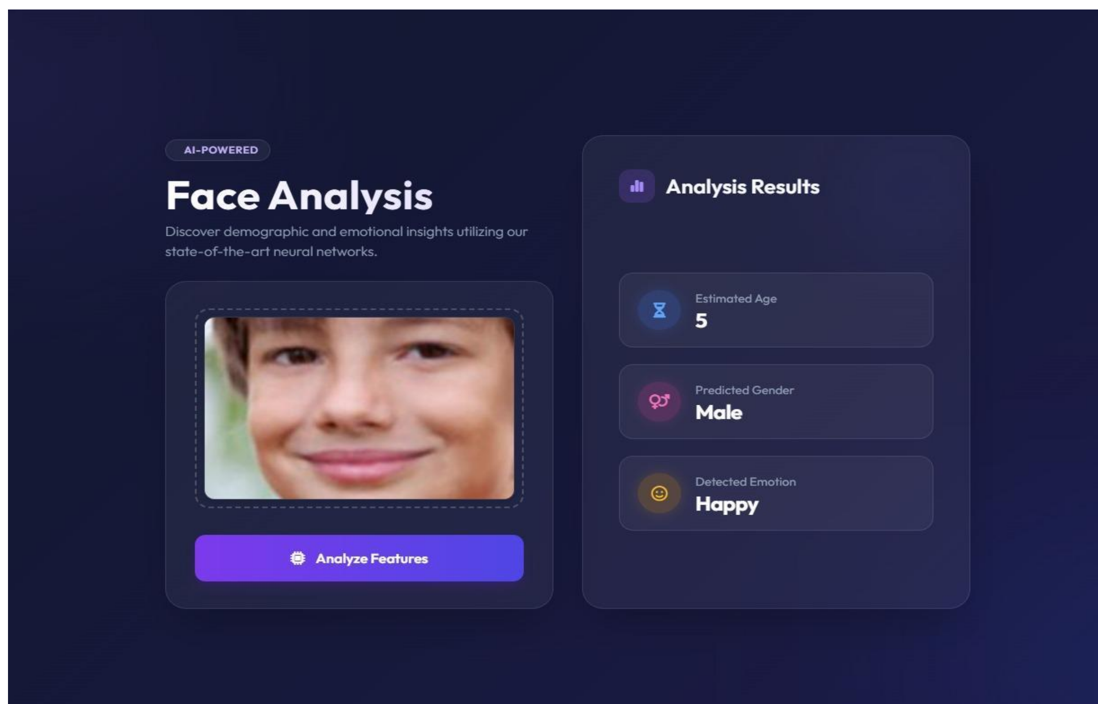
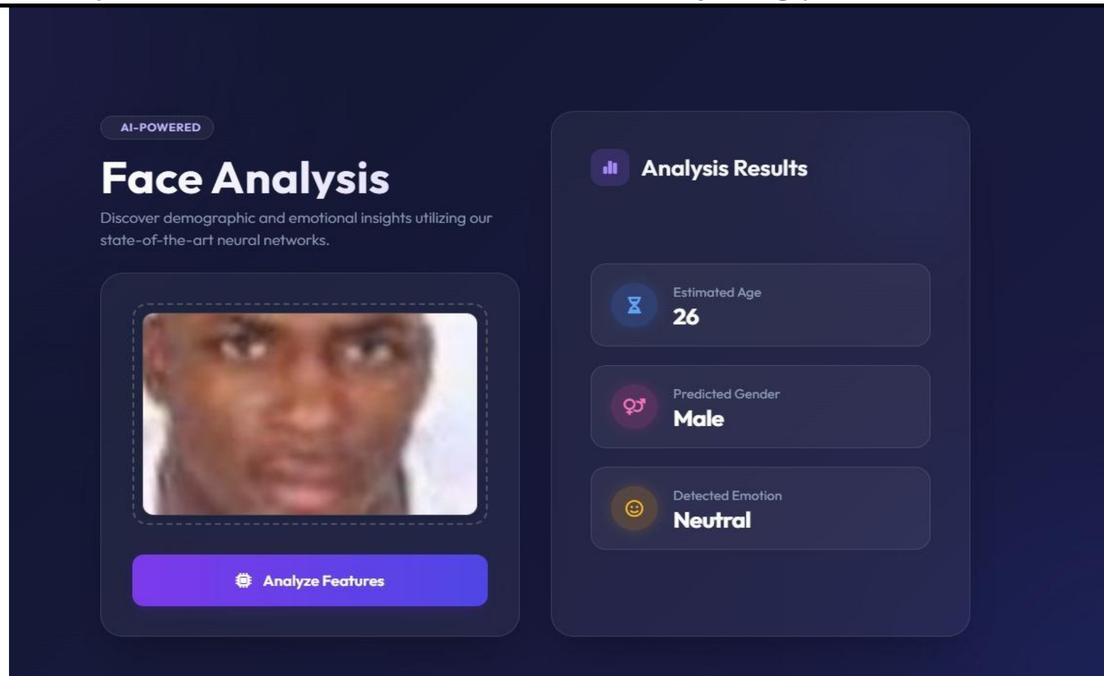


Fig: 7.4 System Output Screen

Step 5 – Final Output (After Uploading image)





8. CONCLUSION

This paper presented a CNN-integrated architecture for the simultaneous prediction of Age, Gender, and Emotion from facial images using deep learning techniques. The proposed system successfully demonstrated that multiple facial analysis tasks can be unified within a single framework, eliminating the need for separate independent pipelines for each prediction task. Three specialized Convolutional Neural Network models were developed and trained using TensorFlow and Keras on a diverse dataset of over 23,000 facial images, each model dedicated to a specific prediction task while sharing a common image preprocessing pipeline that includes resizing, normalization, and NumPy array conversion.

The system was deployed through both Flask and Gradio frameworks, providing an intuitive and accessible web-based interface named NeuroSight that allows users to upload a facial image and receive instant predictions for all three attributes simultaneously. The age prediction model was further refined to output a realistic specific age value from a predicted age range, enhancing the practical usability of the system. Experimental results confirmed that the integrated architecture delivers reliable and accurate predictions across all three tasks, validating the effectiveness of the proposed approach for real-world facial analysis applications.

The successful implementation of this system highlights the power of deep learning-based facial analysis and demonstrates how an integrated CNN architecture can efficiently solve multiple prediction tasks within a single unified framework. The system proves to be suitable for real-time applications across domains including security surveillance, healthcare monitoring, and human-computer interaction. Overall, this work establishes a strong foundation for future research in multi-task facial analysis and contributes meaningfully to the growing field of computer vision and artificial intelligence.

9. FUTURE SCOPE

The proposed CNN-integrated architecture for Age, Gender, and Emotion detection holds significant potential for further development and enhancement in multiple directions. The system can be extended to support real-time video stream analysis using webcam integration, enabling continuous frame-by-frame prediction rather than static image input, which would make it more suitable for live surveillance and monitoring applications. Face detection and alignment preprocessing using OpenCV or MTCNN can be incorporated to automatically detect and crop multiple faces from group photographs, expanding the system beyond single-face analysis. Training the models on larger and more diverse datasets beyond the UTKFace dataset would significantly improve accuracy across different ethnicities, lighting conditions, and varying facial orientations, making the system more robust in real-world environments.

The architecture can be further extended to predict additional facial attributes such as race classification, facial expression intensity levels, and facial landmark detection, transforming it into a more comprehensive facial analysis platform. Deployment on cloud platforms such as AWS, Google Cloud, or Microsoft Azure would make the system highly scalable and globally accessible with improved response times and high availability. A dedicated mobile application can be developed using React Native or Flutter that integrates the trained CNN models for on-device prediction without requiring internet connectivity, making it accessible to a wider range of users. Furthermore, the age prediction component can be upgraded from range-based output to a precise single age value using regression-based deep learning architectures, improving the practical usability of the system. Integration with CCTV surveillance systems and human-computer interaction platforms represents a promising real-world application direction, highlighting the long-term impact and scalability of this work.

BIBLIOGRAPHY

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, pp. 436–444, 2015.

[2] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *NeurIPS*, 2012.

[3] K. He et al., "Deep Residual Learning for Image Recognition," *IEEE CVPR*, 2016.

[4] G. Levi and T. Hassner, "Age and Gender Classification Using Convolutional Neural Networks," *IEEE Workshops*, 2015.

[5] R. Rothe et al., "Deep Expectation of Real and Apparent Age from a Single Image," *IJCV*, 2018.

[6] A. Mollahosseini et al., "Going Deeper in Facial Expression Recognition," *IEEE WACV*, 2016.

[7] S. Li and W. Deng, "Deep Facial Expression Recognition: A Survey," *IEEE Transactions on Affective Computing*, 2020.

[8] B. Abirami et al., "Age and Gender Prediction from Real-Time Facial Images Using CNN," *Materials Today Proceedings*, 2020.

[9] V. Sheoran et al., "Age and Gender Prediction Using Deep CNNs and Transfer Learning," *arXiv*, 2021.

[10] P. Priyanka, “A Review Study on Face Gender Recognition Using Deep Learning,” 2021.

[11] S. Arya, “Age Estimation and Gender Recognition Using Deep Learning,” 2022.

[12] S. T. Chavali, “Smart Facial Emotion Recognition with Gender and Age Factor Estimation,” 2023.

[13] L. M. Zhang et al., “A Deep Learning Method Using Gender-Specific Features for Emotion Recognition,” *Sensors*, 2023.

[14] “Age, Gender and Emotion Detection Using CNN,” *IJARCS*, 2022.

[15] “Facial Recognition of Age, Gender and Emotion Using Deep Learning,” *JETIR*, 2022.

[16] “Age, Gender and Emotion Prediction Using CNN,” *IJRPR*, 2023.

[17] M. Abadi et al., “TensorFlow: A System for Large-Scale Machine Learning,” *OSDI*, 2016.

[18] F. Pedregosa et al., “Scikit-learn: Machine Learning in Python,” *JMLR*, 2011.