Table of Contents

1. Introduction 3

1.1 Purpose 3

1.2 Scope 3

1.3 Objectives 3

2. Planning Requirements 4

2.1 Functional Requirements 4

2.3 Tech Stack 4

3. Design 5

3.1 Architecture Design 5

3.2 Use Case Diagram 6

3.3 Entity-Relationship Diagram (ERD) 7

3.4 Flowchart 8

3.5 DFD Level-0 (Context-Level Diagram) 9

3.6 DFD Level-1 9

4. Implementation 10

4.1 Setting Up the Environment 10

4.2 Installation Steps 10

4.3 Backend Development 11

4.4 Computer Vision Integration 12

4.5 Database Integration 12

4.6 Frontend Development 12

5. Testing 13

5.1 Unit Testing 13

5.2 Integration Testing 14
5.3 User Acceptance Testing (UAT) 15
5.4 Performance Testing 15
6. Maintenance 16
6.1 Monitor Logs for Errors and Resolve Issues 16
6.2 Update Computer Vision Models 17
7. Documentation 18
7.1 User Manual for Non-Technical Users 18
7.2 Developer Documentation 18

1. Introduction

1.1 Purpose

The purpose of this application is to develop an intelligent system that scans physical MCQ answer sheets, detects student responses using computer vision, and automatically evaluates them against a predefined answer key. It aims to streamline grading processes, reduce manual effort, and provide accurate, efficient feedback for educational assessments.

1.2 Scope

The MCQs checker will primarily target:

- Educational Institutions: Automate grading for teachers and exam administrators.
- Students: Provide instant feedback on practice tests or mock exams.
- Scalable Use: Support small-scale (individual) and large-scale (classroom) assessments.

1.3 Objectives

- Enhance grading efficiency by automating answer detection and evaluation.
- Minimize human error in marking MCQs through precise computer vision techniques.
- Provide reliable, real-time results for educators and learners.
- Offer a scalable solution adaptable to various answer sheet formats.

2. Planning Requirements

2.1 Functional Requirements

Feature	Description
Image Input	Accept images of answer sheets via camera or file upload.
Answer Detection	Identify marked answers (e.g., filled bubbles, ticks) using computer vision.
Evaluation	Compare detected answers to a stored answer key and compute scores.
Result Output	Display scores and detailed feedback (e.g., correct/incorrect answers).

2.2 Non-Functional Requirements

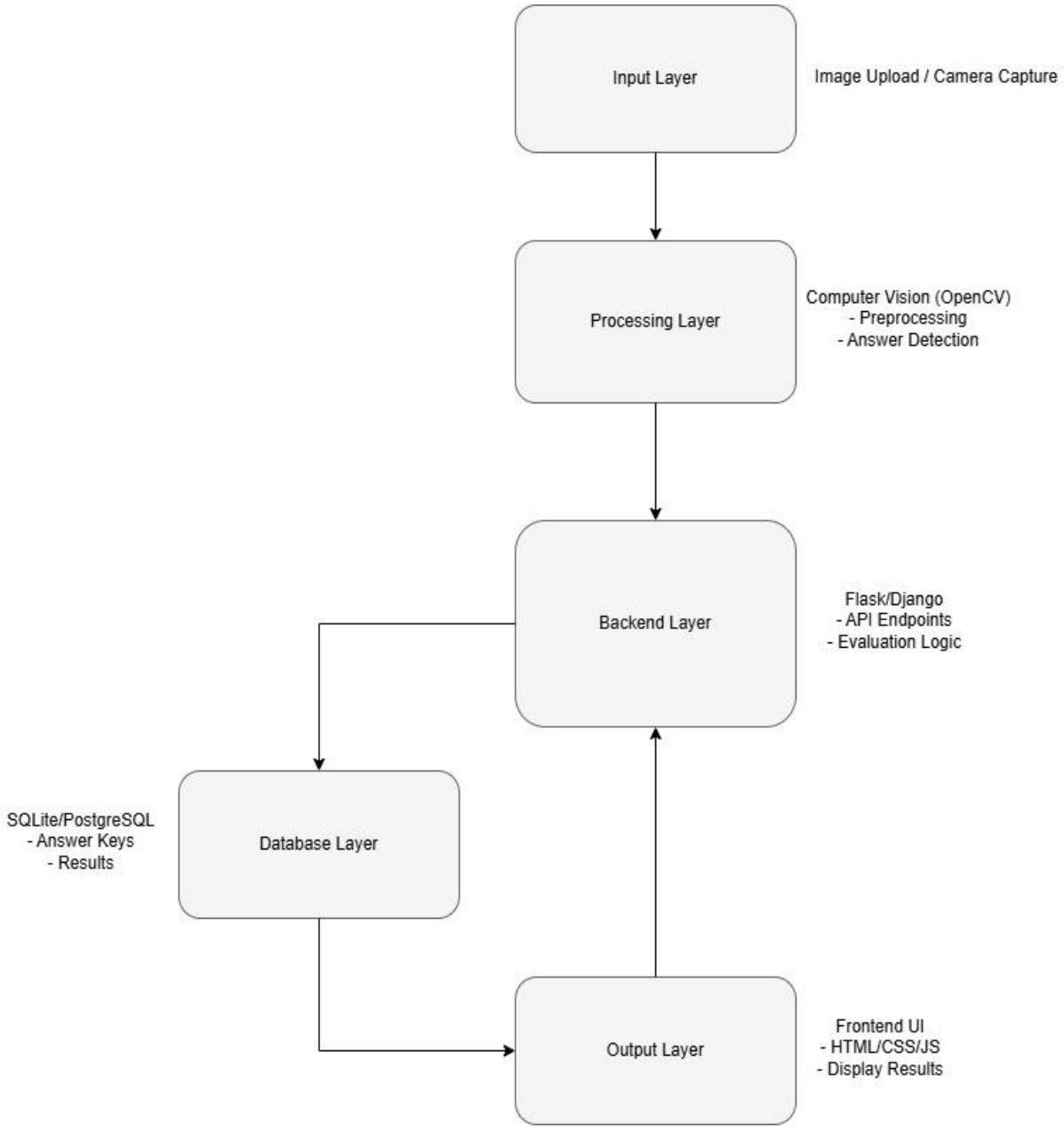
Attribute	Description
Accuracy	Achieve at least 95% accuracy in detecting marked answers.
Processing Time	Process and evaluate a sheet in under 5 seconds.
Scalability	Handle multiple sheets concurrently for classroom use.

2.3 Tech Stack

- Frontend: HTML, CSS, JavaScript (simple interface for image upload and results).
- Backend: Flask or Django (manage image processing and logic).
- Computer Vision: OpenCV, NumPy (image processing and contour detection).
- Database: SQLite or PostgreSQL (store answer keys and results).
- Optional: Tesseract OCR (if handwritten answers are supported).

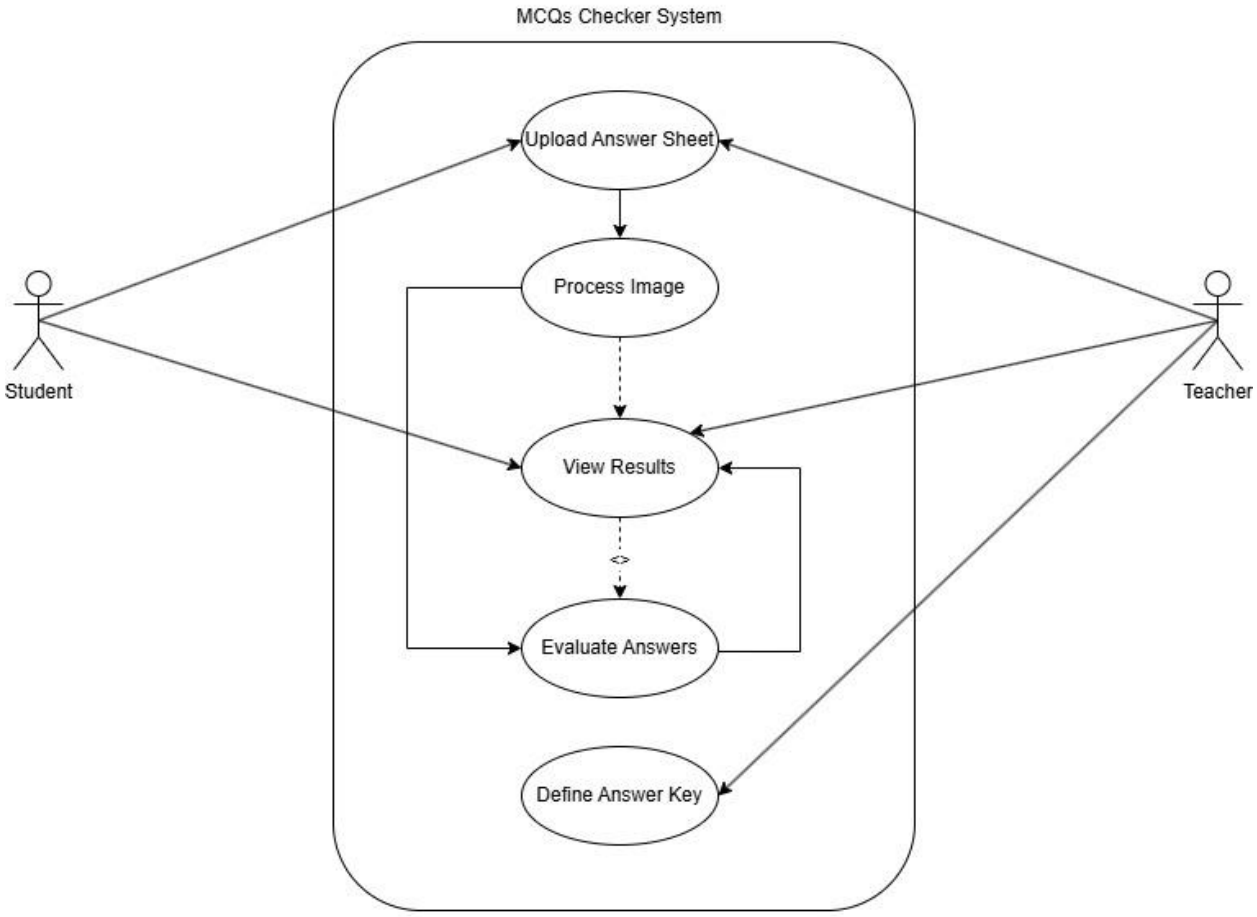
3. Design

3.1 Architecture Design



This diagram provides a layered view of the system's architecture, highlighting the separation of concerns and the flow of data from image input to result output, ensuring modularity and scalability.

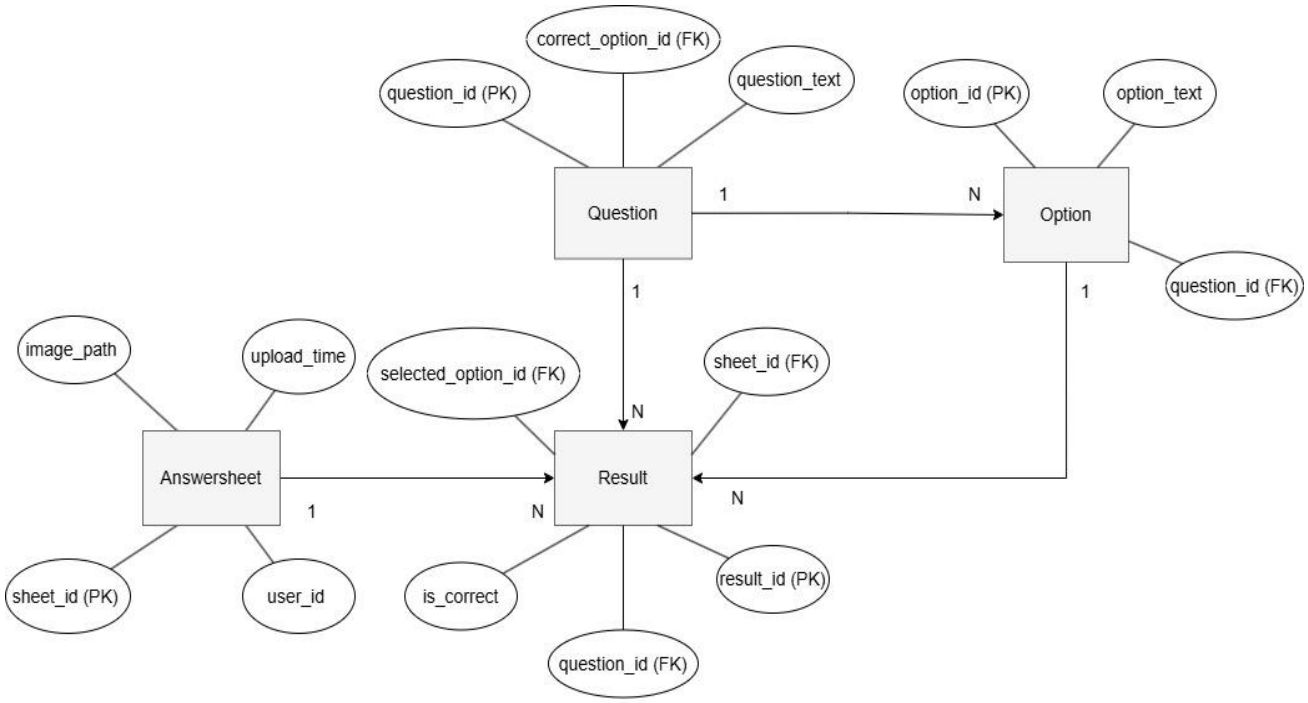
3.2 Use Case Diagram



Use Case Diagram

This diagram captures the system's functional scope from the perspective of its users, showing how students and teachers interact with the system to submit, process, and review MCQ answer sheets.

3.3 Entity-Relationship Diagram (ERD)



ER-Diagram

Entities:

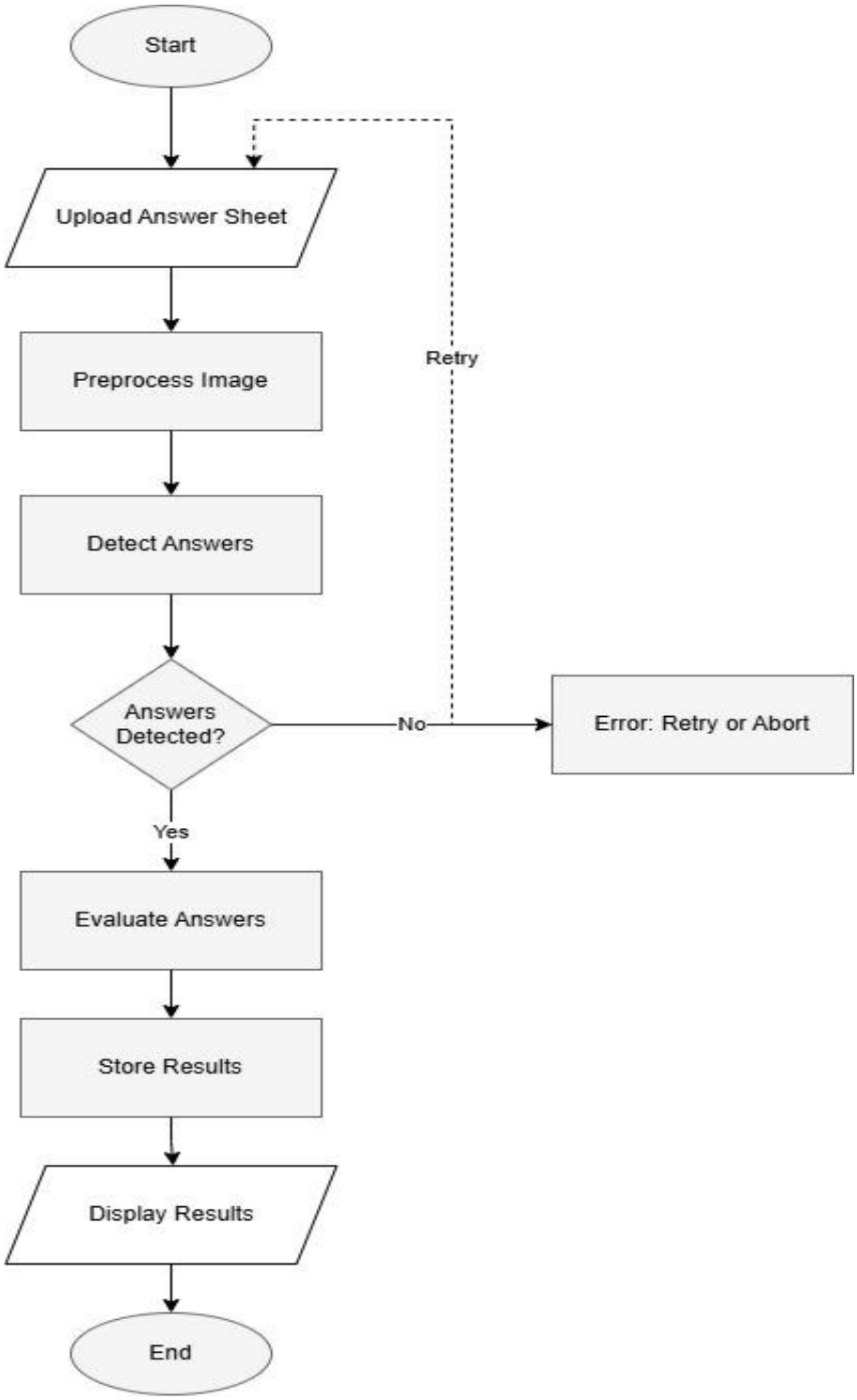
- AnswerSheet (sheet_id, user_id, image_path, upload_time).
- Question (question_id, question_text, correct_option_id).
- Option (option_id, question_id, option_text).
- Result (result_id, sheet_id, question_id, selected_option_id, is_correct).

Relationships:

- AnswerSheet 1:N Result.
- Question 1:N Option.
- Question 1:N Result.
- Option 1:N Result.

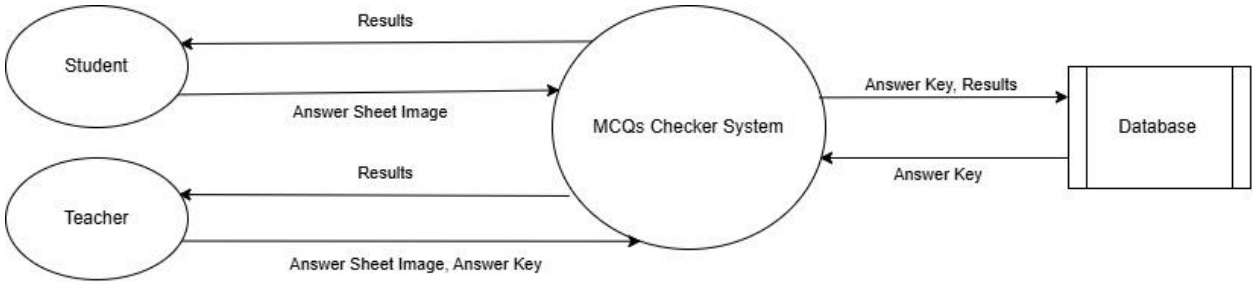
Description: The ERD outlines the relational database schema, detailing how answer sheets link to evaluation results, and how questions connect to their options and correct answers, ensuring efficient data management.

3.4 Flowchart



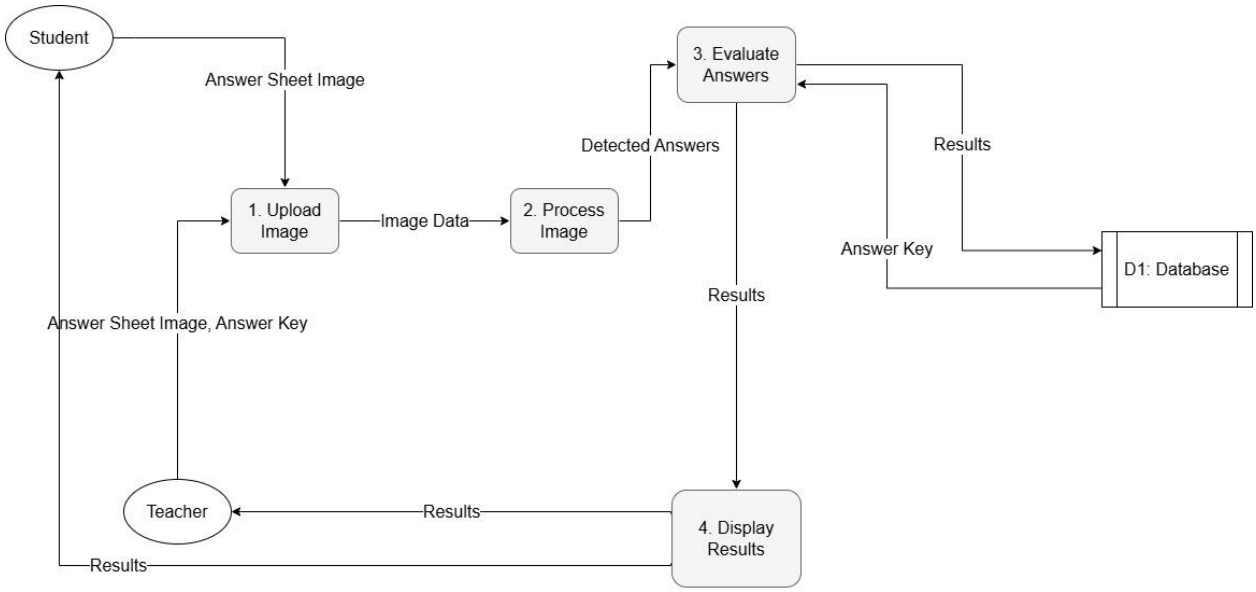
This flowchart provides a clear, sequential view of the system's workflow, from image upload to result display, emphasizing the logic and conditions involved in processing an answer sheet.

3.5 DFD Level-0 (Context-Level Diagram)



This diagram simplifies the system to a single process, focusing on its interactions with users and the database, serving as a starting point for understanding data exchange.

3.6 DFD Level-1



DFD Level-1

This diagram refines the context-level view by breaking down the system into specific processes, showing the detailed flow of data from image input to result output, including storage interactions.

4. Implementation

The implementation phase involves translating the design and requirements of the MCQs Checker Application into a functional system. This section details the setup, development, and integration steps to create an application that scans physical MCQ answer sheets, evaluates them using computer vision, and displays results to users.

4.1 Setting Up the Environment

This stage prepares the development environment with all necessary tools and configurations.

a. Tools and Dependencies

Python 3.9+: Core programming language for backend and computer vision.

Libraries:

- opencv-python: For image processing and computer vision tasks.
- numpy: For array operations on image data.
- flask: Lightweight web framework for backend and API development.
- sqlalchemy: ORM for database interactions (with SQLite/PostgreSQL).
- pillow: Optional for additional image handling.

b. Development Tools:

IDE: VS Code or PyCharm.

Version Control: Git (e.g., GitHub repository).

Virtual Environment: venv for dependency isolation.

4.2 Installation Steps

a. Install Python:

Download and install Python 3.9+ from python.org.

Verify: python --version.

b. Set Up Virtual Environment:

python -m venv venv

venv\Scripts\activate # On Windows

c. Install Dependencies:

pip install opencv-python numpy flask sqlalchemy pillow

d. Project Structure:

MCQs-Checker/

```
├── app.py              # Main Flask application
├── static/             # CSS, JS, and uploaded images
├── templates/          # HTML templates
├── models.py           # Database models
├── vision.py           # Computer vision logic
├── requirements.txt    # List of dependencies
└── config.py           # Configuration settings
```

e. Configuration:

Create config.py:

```
import os

class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY', 'your-secret-key')
    SQLALCHEMY_DATABASE_URI = 'sqlite:///mcqs.db'
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    UPLOAD_FOLDER = 'static/uploads'
```

4.3 Backend Development

The backend handles user requests, processes images, and manages data flow. It uses Flask to create RESTful API endpoints.

Key Components

API Endpoints:

- /upload: POST endpoint to receive answer sheet images.
- /evaluate/<sheet_id>: GET endpoint to process and return results.
- /results/<sheet_id>: GET endpoint to fetch stored results.

4.4 Computer Vision Integration

The computer vision module processes images to detect marked answers using OpenCV.

Key Functionality

- Preprocessing: Convert to grayscale, apply thresholding.
- Detection: Identify filled bubbles via contour detection.
- Mapping: Convert detected marks to question-option pairs.

4.5 Database Integration

The database stores answer sheets, questions, options, and results using SQLAlchemy.

CRUD Operations

- Create: Handled in /upload and /evaluate endpoints.
- Read: Fetch results via /results/<sheet_id> (add this endpoint if needed).
- Update/Delete: Optional, e.g., for correcting answer keys (not implemented here).

4.6 Frontend Development

The frontend provides a simple interface for users to upload images and view results.

5. Testing

Testing ensures the MCQs Checker Application functions correctly, meets user requirements, and performs reliably under various conditions. This section details the testing approach, covering individual component validation (unit testing), system integration (integration testing), user satisfaction (user acceptance testing), and performance under load (performance testing).

5.1 Unit Testing

Unit testing verifies the functionality of individual components in isolation to ensure they perform as expected.

Objectives

- Confirm that each module (e.g., image processing, answer detection, database operations) works correctly.
- Identify and fix defects early in the development cycle.

Test Cases

a. Computer Vision Module (vision.py)

Test: Validate contour detection for filled bubbles.

- Input: Sample image with 5 questions, 4 options each, 3 filled bubbles.
- Expected Output: Dictionary {1: 1, 2: 0, 4: 3} (question_id: option_index).
- Tool: pytest with mock image data.

Test: Handle empty or unclear images.

- Input: Blank image or noisy image.
- Expected Output: Empty dictionary {} or error flag.
- Tool: pytest.

b. Database Operations (models.py)

Test: CRUD operations for AnswerSheet and Result.

- Input: Create an AnswerSheet with image_path="test.jpg", add a Result with question_id=1, selected_option_id=1.
- Expected Output: Records saved and retrievable from SQLite database.
- Tool: pytest with SQLAlchemy session rollback.

c. API Endpoints (app.py)

Test: /upload endpoint with valid file.

- Input: POST request with a sample JPG file.
- Expected Output: JSON response {"sheet_id": 1}, HTTP 200.
- Tool: pytest with Flask test client.

Test: /evaluate/<sheet_id> with valid sheet_id.

- Input: GET request for sheet_id=1 with mocked process_image.
- Expected Output: JSON {"score": 2, "total": 3}, HTTP 200.

5.2 Integration Testing

Integration testing verifies the interaction between components to ensure seamless communication and data flow.

Objectives

- Validate end-to-end functionality from image upload to result display.
- Ensure components (frontend, backend, vision, database) work together without errors.

Test Scenarios

a. Image Upload to Processing Flow

Test: Upload image → Process with computer vision.

- Input: POST /upload with a test image, followed by GET /evaluate/1.
- Expected Output: Detected answers stored in Result table, score returned.
- Tool: pytest with Flask test client and mocked process_image.

b. Backend and Database Integration

Test: Evaluate answers and store results.

- Input: GET /evaluate/1 with predefined answer key.
- Expected Output: Result records created with correct is_correct values.
- Tool: pytest with SQLite in-memory database.

c. Frontend-Backend Communication

Test: Upload form submission and result display.

- Input: Submit image via HTML form.

- Expected Output: Score displayed in #results div (e.g., "Score: 2/3").
- Tool: Selenium or manual browser testing.

5.3 User Acceptance Testing (UAT)

User acceptance testing ensures the application meets end-user expectations and business requirements.

Objectives

- Confirm usability, accuracy, and functionality from a user perspective.
- Gather feedback for improvements.

5.4 Performance Testing

Performance testing evaluates the application’s responsiveness and stability under load.

Objectives

- Ensure processing time meets requirements (<5 seconds per sheet).
- Verify stability with multiple concurrent users.

6. Maintenance

Maintenance ensures the MCQs Checker Application remains operational, accurate, and secure over time. This section details the processes for monitoring system health, updating the computer vision components, and implementing regular enhancements to address evolving user needs and technological advancements.

6.1 Monitor Logs for Errors and Resolve Issues

Continuous monitoring of application logs helps detect and resolve errors promptly, ensuring system reliability.

a. Error Monitoring

Objective: Identify and address runtime issues such as failed image processing, database errors, or API failures.

Logs to Monitor:

- Server Logs: Capture Flask application errors (e.g., 500 Internal Server Errors).
- Vision Logs: Log failures in contour detection or preprocessing (e.g., due to poor image quality).
- Database Logs: Track SQLAlchemy exceptions (e.g., connection timeouts, integrity errors).

b. Resolution Workflow

- Detection: Use log aggregation to spot anomalies (e.g., repeated "contour detection failed" messages).
- Analysis: Investigate root causes (e.g., low-resolution images, server overload).
- Debugging: Reproduce the issue in a development environment (e.g., test with a problematic image).
- Fix Implementation: Apply patches (e.g., adjust vision thresholds, optimize database queries).
- Testing: Validate the fix in a staging environment with sample sheets.
- Deployment: Roll out updates via Git and monitor for recurrence.

6.2 Update Computer Vision Models

Regular updates to the computer vision system ensure it adapts to new answer sheet formats and maintains high accuracy.

Objectives

- Enhance detection accuracy for diverse layouts or lighting conditions.
- Address user-reported issues (e.g., missed bubbles).

Update Process

a. Add New Detection Patterns:
   - Scenario: Support new sheet layouts (e.g., checkboxes instead of bubbles).
   - Action: Modify vision.py to detect additional shapes (e.g., rectangles) using OpenCV's contour properties.

b. Improve Accuracy:
   - Scenario: Improve detection under poor lighting.
   - Action: Adjust preprocessing (e.g., adaptive thresholding) or train a machine learning model (e.g., CNN) with diverse image samples.

c. Version Control:
   - Tool: Git for tracking vision code changes.
   - Process: Tag releases (e.g., v1.1-detection-update) and maintain rollback capability if updates fail.

7. Documentation

Proper documentation ensures that end-users can effectively interact with the MCQs Checker Application and that developers can understand, maintain, and extend its functionality. This section is divided into a user manual for non-technical users and developer documentation for technical stakeholders.

7.1 User Manual for Non-Technical Users

This manual is designed for end-users (e.g., students and teachers) who interact with the application to scan and evaluate MCQ answer sheets.

Introduction

a. Overview: The MCQs Checker Application is a tool that scans physical MCQ answer sheets, automatically detects marked answers using computer vision, and provides instant evaluation results. It simplifies grading for teachers and offers quick feedback for students.
b. Purpose: Streamline the process of checking multiple-choice tests by eliminating manual grading.

Key Features:

a. Upload Answer Sheet: Upload a scanned or photographed image of an MCQ answer sheet.
b. View Results: Receive immediate feedback with a score and correct/incorrect answers.

7.2 Developer Documentation

This documentation targets developers and technical stakeholders, providing insights into the system's architecture, implementation details, and maintenance procedures.

System Architecture

Overview: The MCQs Checker Application follows a layered architecture:

- Input Layer: Handles image uploads via a web interface.
- Processing Layer: Uses OpenCV for computer vision tasks (preprocessing, answer detection).
- Backend Layer: Manages logic and API endpoints with Flask.

- Database Layer: Stores data in SQLite/PostgreSQL using SQLAlchemy.
- Output Layer: Displays results via HTML/CSS/JS frontend.

API Endpoints

List of Endpoints:

a. /:

   Method: GET

   Description: Renders the homepage with the upload form.

   Response: HTML page.

b. /upload:

   Method: POST

   Parameters: file (multipart/form-data, image file).

   Response: JSON {"sheet_id": <int>} (success) or {"error": <string>} (failure).

c. /evaluate/<sheet_id>:

   Method: GET

   Parameters: sheet_id (integer, path parameter).

   Response: JSON {"score": <int>, "total": <int>}.

Computer Vision Configuration

Module: vision.py

Details:

- Uses OpenCV for grayscale conversion, thresholding, and contour detection.
- Assumes a grid layout (e.g., 4 options per question).

Customization:

- Adjust options_per_question and contour size thresholds (100 < area < 1000) in process_image.
- Update preprocessing (e.g., adaptive thresholding) for different lighting conditions.

Database Schema

Entities:

- AnswerSheet: sheet_id (PK), user_id, image_path, upload_time.
- Question: question_id (PK), question_text, correct_option_id (FK).
- Option: option_id (PK), question_id (FK), option_text.
- Result: result_id (PK), sheet_id (FK), question_id (FK), selected_option_id (FK), is_correct.

Diagram: Refer to the ERD figure (relationships: 1:N between AnswerSheet-Result, Question-Option, etc.).

CRUD Guidelines:

- Create: Use db.session.add() and db.session.commit() in API endpoints.
- Read: Query with Model.query.get() or filter_by().
- Update/Delete: Optional, implement as needed (e.g., /update_result/<result_id>).