

|  |  |
| --- | --- |
| Project title | University Management System (UMS) |
| keywords | Machine Learning) Face Recognition,Chatbot, Data Analysis(, Web Application, Mobile Application |
| contributors | Shahd Yehia Mohammed  Rewan Sabry Elsayed  Marwa Agamy Abdo  Hashem Ahmed Hassan  Youssef Maged Helal  Badr Adel Moussa  Ahmed Hesham Badr  Omar Mohammed Ramadan |
| Supervisor | Prof.DR. Wessam Samir |
| summary | This smart college system streamlines course registration, attendance, and communication. Students benefit from face recognition and an AI chat bot for quick assistance. Admins and faculty manage data and tasks efficiently through automation. Teaching assistants access tools to support instructors and track student progress. |

Table of Contents

**Table Of Contents3**

**List of figures5**

**Abstract7**

**Acknowledgment7**

**Chapter 1: introduction & motivation8**

* 1. introduction8
  2. Motivation8

1.3 Target Audience9

**Chapter 2: System overview10**

2.1 Introduction 10

2.2 User Roles10

2.3 Technology Stack 10

2.3.1 Background Technologies10

2.3.2 Frontend Technologies11

2.3.3 Development Tools12

2.4 System Architecture 13

2.4.1 Architecture layer13

2.4.2 Security integration 14

2.5 Data Management 12

2.5.1 Database structure & key models 12

2.5.2 Data integrity measures 12

2.5.3 ERD &User flow13

2.6 Website &mobile application 16

2.7 Development & hosting 17

2.8 Accessibility & compliance 18

**Chapter 3: System feature22**

* 1. User authentication & JWT management 22
  2. Registration system 25
  3. Attendance system 31
  4. Performance system 41
  5. Exam system 44
  6. Announcement47
  7. Fees system 48
  8. Course system (admin )50
  9. Chat system 52
  10. Complain management system54
  11. Chatbot56

**Chapter 4: Security framework 60**

* 1. Authentication system (JWT details)61
  2. Authorization Middleware System62
  3. Core security principles 63
  4. Best practice 64
  5. API end points64

**Chapter 5: project management 66**

* 1. introduction66
  2. Methodology 68
     1. Implementation 68
     2. Project timeline 69
     3. Weekly meeting &MOM every week 69
  3. Cost Estimation 71

**Chapter 6: Testing strategy** 73

* 1. introduction73
  2. Testing Types 74
     1. Integration test74
     2. Manual function test74
     3. User acceptance74
     4. Load & performance test 74
  3. Bug tracking & resolution 74

**conclusion 75**

**future work76**

**references 79**

list of figures

1. **Figure 1: Use case diagram**
2. **Figure 2: ERD for UMS**
3. **Figure 3: User flow for UMS**
4. **Figure 4: mobile application HP**
5. **Figure 5: mobile application admin**
6. **Figure 6: website Homepage**
7. **Figure 7: Website face recognition**
8. **Figure 8: website performance**
9. **Figure 9: sequence diagram for UAUTH**
10. **Figure 10: Sequance diagram of registration**
11. **Figure 11: neural network model architecture diagram**
12. **Figure 12: Accuracy & loss over epochs**
13. **Figure 13: Attendance Sequance diagram**
14. **Figure 14: Exams Sequence diagram**
15. **Figure 15 GPA management API**

**Acknowledgement**

We, the project team, would like to express our sincere gratitude to everyone who supported us during the development of our College Management System.

First, we are deeply thankful to our supervisor, Dr. Wessam Samir, for their continuous guidance, valuable feedback, and encouragement throughout all phases of this project.

A special thanks to our families and friends for their patience, motivation, and encouragement during this journey.

Finally, we are grateful to the students, teachers, and administrators who participated in testing and provided meaningful feedback to help us improve the system.

This project would not have been possible without the combined efforts and dedication of our entire team.

**Abstract**

This project aims to streamline college management through an intelligent system that handles academic data and enhances communication among students, professors, and teaching assistants. Key features include automated subject registration, smart attendance tracking using Face Recognition over private Wi-Fi, and an AI-powered chatbot to assist students with their queries. The system empowers teachers and TAs with tools to manage classes, track attendance, and support students efficiently. Overall, it simplifies administrative tasks, boosts accessibility, and creates a more connected academic environment.

**Chapter 1: Introduction & Motivation**

* + **1.1 Introduction**

The Computer and Data Science Alexandria University App is a comprehensive system application designed to streamline the student experience by integrating advanced technologies and user-friendly functionalities. This application serves as a digital gateway for students to access essential academic information, manage their academic profiles, and interact with the university’s resources efficiently. Below is an overview of the main components and features of the project.

* **1.2 Motivation**

The existing college management system at our institution suffers from limited functionality, poor user experience, and a lack of integration between key academic services. These issues often lead to confusion, delays, and inefficiencies for both students and faculty members. Recognizing these challenges, our motivation is to develop a modern, centralized system that addresses these shortcomings. By leveraging advanced technologies and intuitive design, our goal is to create a more reliable, efficient, and user-friendly platform that enhances the academic experience for all users.

* **1.3 Target Audience**

The target audience of this Academic Management System includes the following user groups:

1. **Students**

* Can register for courses and sections.
* View academic records, schedules, grades, attendance, and fee status.
* Communicate with faculty via chat or chatbot.
* Mark attendance (e.g., through face recognition or QR code).

1. **Faculty Members (Doctors and Teaching Assistants)**

* Manage their assigned courses and sections.
* Enter and update grades.
* View and analyze student attendance and performance.
* Post announcements and receive student queries.

1. **Administrators**

* Oversee user creation and role assignments.
* Manage the course catalog, exam schedules, and term processing.
* Monitor fee collection, attendance stats, and academic records.
* Maintain overall system data integrity and security.

A screenshot of a computer

AI-generated content may be incorrect.figure 1: use case diagram

**Chapter 2: System overview**

* **2.1 Introduction**

The Academic Management System is a comprehensive platform designed to streamline university operations, including student registration, course management, exam scheduling, attendance tracking, and grading.

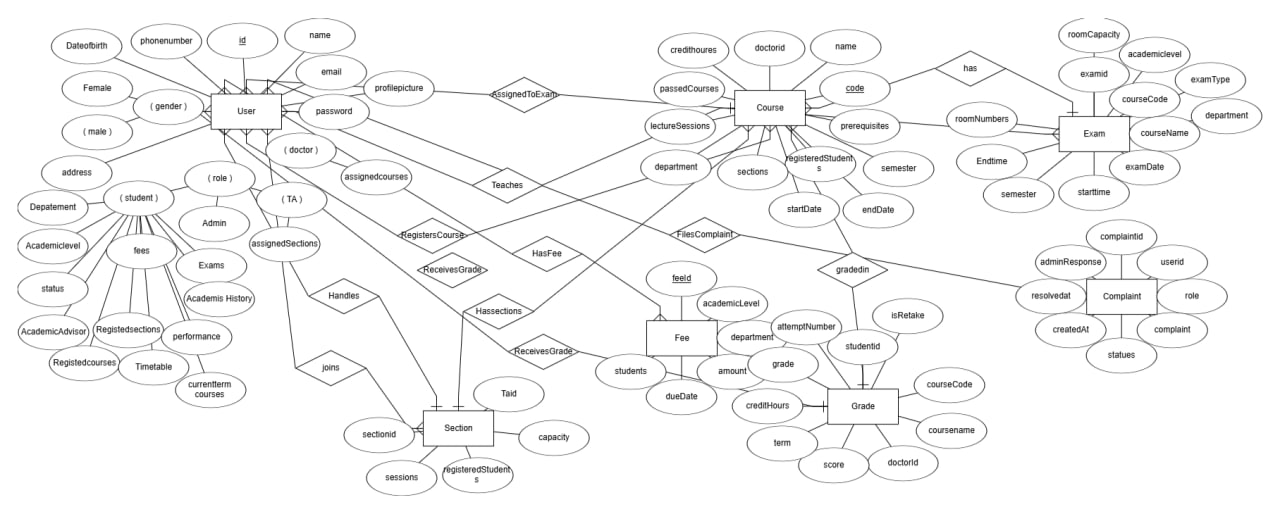
* **2.2 User Roles**

The system serves four primary user roles: Administrators, Doctors, Teaching Assistants, and Students, each with tailored functionalities and access controls.

* **2.3 Technology Stack**
* **2.3.1 Backend Technologies**
* **Node.js**: JavaScript runtime for server-side execution.
* **Express.js**: Web application framework for API development.
* **MongoDB**: Open-Source database for flexible data storage.
* **Mongoose**: ODM library for MongoDB interactions, used for defining schemas and managing data (e.g., User, Course, Grade, Exam models).
* **JWT (JSON Web Tokens)**: Secure token-based authentication.
* **Bcrypt**: Password hashing for security (used via Mongoose pre-save hooks).
* **Cloudinary**: Cloud-based image management for profile pictures.
* **Crypto**: Data encryption utilities (e.g., for student ID encryption in attendance marking).
* **Nodemailer**: Email notification system (e.g., for sending OTPs during password reset).
* **Libraries for File Handling**: json2csv, csv-parser, xlsx for CSV/Excel processing (grade uploads, reports).
* **node-cron**: For scheduling automated tasks like end-of-term processing.
* **Flask**: A lightweight and flexible Python web framework used to build web applications and APIs quickly with minimal setup.
* **2.3.2 Front end technologies**
* **Flutter:** The mobile application is built using Flutter, enabling a consistent and high-performance experience across Android and iOS devices.
* **React.js** The web application’s frontend is developed using React, offering a dynamic and responsive user interface.
* that implements Google’s Material Design system, offering pre-built components for faster and consistent UI development.
* **Font Awesome**: A widely used icon toolkit that provides scalable vector icons and social logos for websites and applications.
* **Bootstrap**: An open-source front-end framework that helps developers quickly build responsive and mobile-first web interfaces using HTML, CSS, and JavaScript components.
* **2.3.3 Development Tools**
* **Postman**: API testing and documentation.
* **Git**: Version control system.
* **Vercel**: For Deployment.
* **2.4 System Architecture**
* **2.4.1 Architectural Layers**

The system follows a modular MVC (Model-View-Controller) architecture with:

* **Presentation Layer**: React-based frontend (User Interface).
* **Application Layer**: Node.js/Express API endpoints defining how the frontend interacts with the backend.
* **Business Logic Layer**: Controller modules (authController, adminController, studentController, attendanceController, gpaController) containing the core logic for each feature.
* **Data Access Layer**: MongoDB with Mongoose models (User, Course, Exam, Grade) abstracting database interactions.
* **2.4.2 Security Integration**
* **Security Layer**: Authentication (authController) and authorization middleware (authMiddleware.js, isCourseDoctor.js) protecting API routes and ensuring data access controls.
* **2.5 Data Management**
* **2.5.1 Database Structure & Key Models**
* User profiles with role-specific data: User model stores common fields (id, name, email, password, role) and role-specific attributes (department, academicLevel, performance, academicHistory for students; assignedCourses for doctors; assignedSections for TAs).
* Course and section organization: Course model contains course details, lecture sessions, and an array of embedded sections (with TA, capacity, sessions, registered students).
* Exam scheduling: Exam model stores exam details, including room assignments and lists of students per room.
* Attendance records: Stored as an array within the User model (student's attendance history).
* Grade tracking: Grade model links students, courses, scores, letter grades, and term information.
* **2.5.2 Data Integrity Measures**
* Reference checks: Controllers frequently check for existence of related documents (e.g., does course exist before adding section, does user exist before adding grade).
* Transaction support: Critical operations like adminController.addExam, studentController.registerForCourses, gpaController.uploadGradesFile use MongoDB sessions/transactions to ensure atomicity of multiple database updates.
* Atomic updates: Mongoose save operations and findOneAndUpdate are used. Business logic in controllers ensures that related data is consistently updated (e.g., when a course is deleted, student registrations are cleaned up).
* **2.5.3 ERD & Userflow**

 Figure 2 ERD for UMS

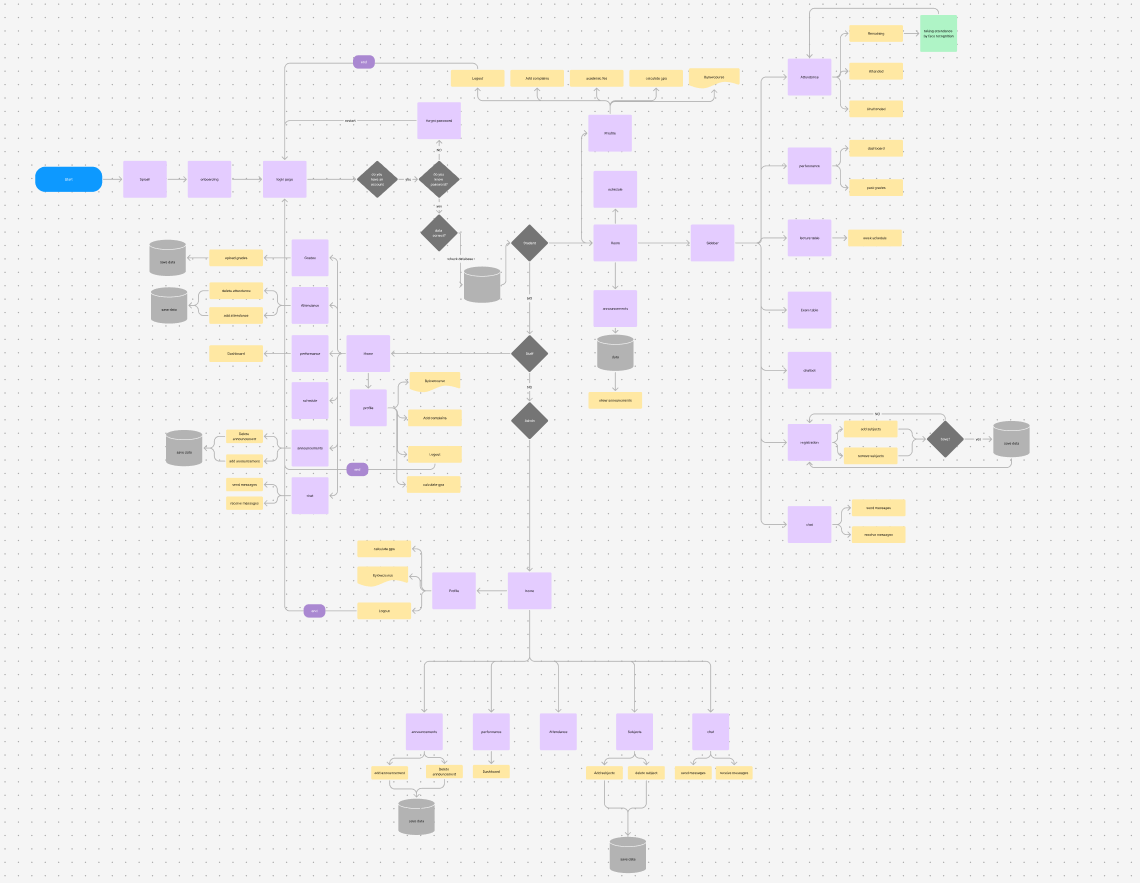
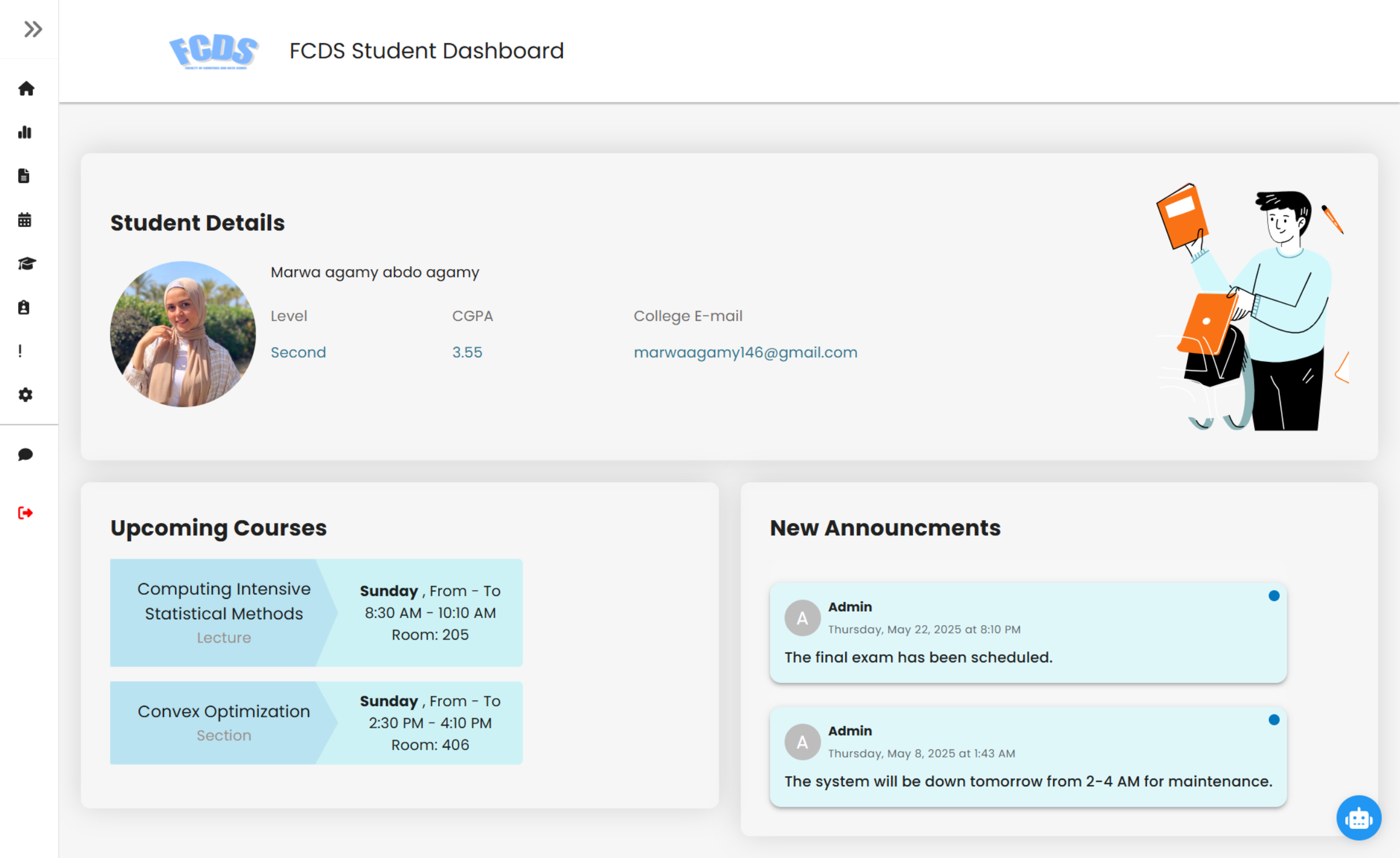


Figure 3 User flow for UMS

* **2.6 website & Mobile application**

The following images provide a visual representation of how the system functions across both the mobile application and the website. These screenshots showcase of some key features such as course registration, grade management, student performance and role-based access for students, teachers, and administrators. They highlight the user interface and demonstrate how different users interact with the system in real scenarios. This section aims to give a clear understanding of the system’s functionality and user experience before diving into the technical or back-end details. Figure 4,5,6,7,8



A screenshot of a phone

AI-generated content may be incorrect.

* **2.7 Deployment & Hosting**

The University Management System (UMS) is currently in its development and testing phase, with deployment environments set up to support future production releases. The system is designed with a cloud-first approach using scalable and modern platforms.

* Frontend Hosting
* The web frontend, developed in **React.js**, is prepared for deployment on **Vercel**, a frontend hosting platform with built-in CI/CD integration.
* Although not yet publicly released, the Vercel deployment environment is configured and ready for continuous deployment once testing is complete.
* Mobile Application Deployment
* The mobile app, built using **Flutter**, is not yet published to app stores but is tested locally on Android and iOS devices.
* Firebase App Distribution is intended for beta testing prior to public release.
* Backend & APIs
* The backend is built using **Node.js** (Express.js) and **Flask** for specific Python-based modules (e.g., face recognition).
* Backend APIs are being tested in local and private cloud environments.
* **Render** and **Firebase Cloud Functions** are considered for future production deployment, offering auto-scaling and monitoring.
* Database
* **MongoDB Atlas** is used as the cloud-hosted NoSQL database.
  + It provides built-in features such as **automated backups**, **cluster scaling**, and **access control**.
  + Collections such as Users, Courses, Grades, and Attendance are indexed to support fast and efficient queries during testing.
* CI/CD Pipeline
* Source code is version-controlled on **GitHub**.
* Deployment workflows using **GitHub Actions** are under development to automate testing and deployment for both frontend and backend services.
* Environment Configuration
* Sensitive data like JWT secrets, encryption keys, and API tokens are stored in environment variables using .env files (not committed to source control).
* These configurations support seamless switching between local, testing, and production environments.
* Current Status & Future Plans
* As of now, the system is undergoing **internal testing and refinement**.
* A **private deployment** exists for demonstration purposes, but **no public domain** or production endpoint is currently available.
* Upon completion of testing and feature finalization, the full system will be deployed with:
  + **Production database cluster on MongoDB Atlas**
  + **Frontend hosted on Vercel**
  + **Backend deployed via Render or Firebase**
  + **Mobile app published to Play Store and App Store**
* **2.8 Accessibility & Compliance**

The University Management System (UMS) is designed with a commitment to accessibility, data privacy, and inclusivity. While English is the primary language of the interface, selected areas of the system include partial Arabic content to improve usability for Arabic-speaking users. Future enhancements aim to expand full multi-language support and improve accessibility for all users.

* **Responsive Design**
* All interfaces, including the website and mobile application, are developed with responsive layouts that adapt to various screen sizes and orientations—ensuring full usability across desktops, tablets, and smartphones.
* The mobile app built with **Flutter** offers a consistent experience across Android and iOS platforms.
* **Visual Accessibility**
* Interfaces follow **WCAG 2.1 Level AA** guidelines:
  + Adequate **contrast ratios** between text and background.
  + Support for **large font sizes** and **scalable UI components**.
  + Clear use of icons and labels to support navigation and reduce cognitive load.
* **Keyboard and Navigation Accessibility**
* The web interface is partially optimized for keyboard navigation (Tab/Shift+Tab) for users who cannot use a mouse.
* Elements such as buttons, modals, and menus maintain logical tab order and focus handling.
* Future updates aim to improve support for screen readers and include ARIA attributes.
* **Language Support**
* The primary language of the system is **English**, with **partial Arabic integration** in selected pages and data fields (e.g., names, department titles).
* The current implementation does not support full language switching or localization of all UI elements.
* Future versions will provide a complete **multi-language toggle**, enabling users to choose between English and Arabic from the main interface.
* **Security and Privacy Compliance**
* All personal and academic data is protected using **JWT authentication**, **bcrypt password hashing**, and **AES encryption** for sensitive fields such as student IDs and attendance logs.
* Role-based access control ensures users only access their own data or data they are authorized to manage.
* Access permissions are enforced using server-side middleware, reducing exposure to unauthorized queries or privilege escalation.
* **Legal and Ethical Compliance**
* The system design considers key principles from **GDPR** and **Egyptian Personal Data Protection Law**:
  + Minimal data collection and secure storage.
  + User awareness about their data usage.
  + Protection of personal identity and academic records.

**Chapter 3: System features**

* 1. User Authentication &JWT Management
* **Overview**

The app provides a secure login feature requiring students to log in using their university credentials (username and password).

* **Backend**
* **Authentication System**
  + Receives id (unique user identifier) and password from the client.
  + authController.login finds user by id; if not found, returns 400.
  + Compares provided password with the stored hashed password (using bcrypt via user.comparePassword). If mismatch, returns 400.
  + Generates JWT with a payload containing user.id, user.role, and self Only: true .Token signed with process.env.JWT\_SECRET.
  + Returns the token and basic user data (id, name, email, role; plus student-specifics like academic Level, CGPA).
  + Token expires after 1 hour for security, as configured in JWT signing.
    - **Password Management (Change, Forgot, Reset)**
* Change Password (authController.changePassword):
  + - Authenticated users only (user ID from JWT).
    - Verifies current password against the stored hash.
    - Hashes the new password (via Mongoose pre-save hook on User model) before storage.
    - Updates user record securely.
* Forgot Password (authController.forgotPassword):
  + - User provides email.
    - Generates a 6-digit OTP.
    - Stores OTP (user.otp) and its expiry (user.otpExpire - 5 minutes from generation) in the user's database record.
    - Sends email via Nodemailer with OTP and reset instructions.
  + Reset Password (authController.resetPassword):
    - User provides email, OTP, and new password.
    - Validates OTP against user.otp and checks if user.otpExpire is still in the future.
    - If valid, updates password (hashed via pre-save hook).
    - Clears user.otp and user.otpExpire from the database after use
* Profile Picture Management (User Self-Service & Admin)
  + Implemented in authController.js (updateProfilePicture, getProfilePicture, deleteProfilePicture), protected by canManageProfilePicture middleware.
* Update:
  + - Users update their own; Admins can update any user's picture (via req.params.userId).
    - Deletes old picture from Cloudinary if one exists (extracting public ID from URL).
    - Uploads new file (req.file) to Cloudinary profile\_pictures folder.
    - Saves the new result.secure\_url to user.profile Picture.
  + Get: Retrieves user.profilePicture URL.
    - Delete: Deletes from Cloudinary and clears user.profile Picture field

A screenshot of a diagram

AI-generated content may be incorrect.

figure 9 sequence diagram for UAUTH

* 1. Registration System
     + **overview**

The Registration System enables students to enroll in courses and sectionsbased on academic level, prerequisites, and credit limits. It supports course/section registration, drop operations, and timetable generation. The system ensures conflict-free scheduling and enforces CGPA-based credit hour rule

* **Backend**

Managed by studentController.js, protected by isStudent and enforceSelfAction middleware.

* Course Registration

Student provides array of courseCodes.

* + Validations:
    - Course existence, student not already registered, course not already passed (unless failed and retaking).
    - **Checks prerequisites**: all prerequisite course codes in course.prerequisites must be in student.performance.passedCourses (unless retaking failed course). Blocks registration if not met.
    - **Verifies credit hour limits**: current term hours + new course hours must not exceed student.performance.maxAllowedCreditHours.
    - **Detects schedule conflicts**: uses hasTimeConflict helper against student's current lectureSessions and section.sessions.
    - Checks course.capacity.
  + Actions:
    - **Updates student records**: adds to student.registeredCourses, student.currentTermCourses. Adds student ID to course.registeredStudents.
    - If retaking a failed course, calls cleanupExamsForGradedCourse for the student and course.
* Section Registration

Student provides array of { courseCode, sectionId }.

* + Validations:
    - Validates course enrollment (student.registeredCourses must include courseCode).
    - Checks section capacity (section.capacity vs section.registeredStudents.length).
    - Prevents time conflicts with existing schedule using hasTimeConflict.
    - Ensures student registers for only one section per course.
  + Actions:
    - Updates student records: adds sectionId to student.registeredSections. Adds student ID to section.registeredStudents.
* Drop Operations
  + Removes course/section links from student's profile (registeredCourses, currentTermCourses, registeredSections) and from course/section rosters (course.registeredStudents, section.registeredStudents).
  + Validates that course/section is not passed or already graded in the current term.
  + Calls cleanupExamsForGradedCourse for dropped courses.
  + Maintains data consistency using transactions.
* Credit Hour Limits & CGPA Rules.
  + CGPA-Based Rules enforced during course registration via student.performance.maxAllowedCreditHours, which is calculated by gpaController.calculateMaxAllowedCreditHours:
    - CGPA < 2.0 → Max 12 credits allowed for the term.
    - CGPA 2.0 to < 3.3 (implied) → Max 18 credits .
    - CGPA ≥ 3.3 → Max 21 credits allowed for the term.
* Timetable Generation .
  + Generates weekly timetable based on user role (req.user.id must match req.params.userId).
  + Student**:** Lectures (with doctor name) and registered sections (with TA name).
  + **Doctor**: Lectures for their assigned courses.
  + **TA**: Sections to which they are assigned.
* Available Course Viewing
  + Lists courses available for student registration.
  + Filters based on failed courses (retakable), or courses where prerequisites are met and not yet passed.
  + Provides course details, doctor/TA names, section availability.

A diagram of a company

AI-generated content may be incorrect.Figure 10 Sequance diagram of registration

3.3 Attendance System

* **overview**

The attendance system is built to automate student attendance recording through facial recognition. It is designed for use in classrooms where each room is equipped with a local server or the instructor's laptop. This system operates entirely offline, utilizing a local Wi-Fi network to ensure security and accessibility. This document explains the system's architecture, workflow, and implementation steps in detail.

* **system Architecture**

- **Server Host:** Each classroom has either a dedicated server or a laptop operated by the instructor, running a face recognition machine learning (ML) model.

- **Local Network**: All devices (students' phones/laptops and the instructor's server) connect to a private Wi-Fi network available in the classroom. This network does not require internet access.

- **Secure Access:** The server is accessed via a web browser through HTTPS, using locally generated SSL certificates.

- **Client Interface:** Students access a web interface by entering the server's IP address into their browser. The interface allows the camera to capture a face image for recognition.

* **Workflow**
  + - 1. **Instructor Starts the Server:**

-Upon launching: the server displays the local IP address.

-The system initializes the face recognition model and prepares for incoming connections.

**2. Student Connects to the Server:**

- The student connects to the server by entering the IP in their browser.

- HTTPS is enforced, so the browser requests camera access over a secure connection.

**3. Secure Camera Access Using Certificates:**

To allow camera access over HTTPS on a local network, SSL certificates must be installed on the instructor's machine.

- Self-signed certificates are generated and trusted manually on the instructor's device.

- The server is configured to use these certificates, allowing secure https:// connections.

**4. Face Capture and Recognition:**

-The student's camera captures a live image.

- The image is sent to the server for processing.

- The ML model compares the face to the known database.

- If a match is found, the student is marked as present.

**5. Logging Attendance:**

- Each recognition event is logged into an Excel sheet specific to the course.

- The sheet includes fields like student ID, name, course, and timestamp.

- This sheet is generated or appended to automatically by the server.

**6. Instructor Uploads Attendance Sheet:**

**-** After the session, the instructor uploads the generated Excel file to their dashboard.

- The dashboard consolidates and visualizes attendance records over time.

* **Certificate Setup (HTTPS on Localhost)**

To allow HTTPS connections on a local network (required for camera access), follow these steps:

1. Generate SSL Certificates: openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365 -nodes
2. Trust the Certificate:- On Windows: Import the certificate into "Trusted Root Certification Authorities."- On macOS: Use Keychain Access to trust the certificate.- On Linux: Use your distro's certificate management tool.
3. Configure Server to Use HTTPS: When running your web server (e.g., Flask, Node.js, etc.), load the certificate and key: https.createServer({ key: fs.readFileSync('key.pem'), cert: fs.readFileSync('cert.pem') }, app).listen(443);

* **Face recognition**

This project implements a real-time face recognition system using deep learning techniques. The goal was to build an accurate and efficient model that can recognize known individuals through webcam input. The system uses facial embeddings extracted by a pre-trained FaceNet model and classifies identities using a custom-trained neural network

* Technologies and Libraries Used
* **Face Detection – MTCNN: MTCNN (Multi-task Cascaded Convolutional Neural Networks)**

is used for detecting and aligning faces from input images. It is effective at locating facial landmarks (e.g., eyes, nose, mouth) and bounding boxes across multiple scales. Its cascaded structure improves precision and robustness in real-world conditions.

*Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks*. IEEE Signal Processing Letters. Zhang, K., Zhang, Z., Li, Z., & Qiao, Y. (2016).

* **Face Embedding – FaceNet:**

FaceNet transforms face images into 512-dimensional embeddings that preserve identity-related features. This embedding allows the model to measure similarity using distances in the feature space rather than comparing raw pixels.

*FaceNet: A Unified Embedding for Face Recognition and Clustering*. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 815–823. Schroff, F., Kalenichenko, D., & Philbin, J. (2015).

* **Model Framework – TensorFlow & Keras**:

TensorFlow is a scalable deep learning library developed by Google, and Keras offers a simplified high-level API for building, training, and evaluating neural networks.

* **Preprocessing – OpenCV:**

OpenCV is a computer vision library used for reading, resizing, and manipulating images. It was used to process images before they were passed to the detector or embedding model.

* **Evaluation & Encoding – Scikit-learn:**

Used LabelEncoder to convert string class names to integers and train\_test\_split to divide the dataset into training and testing subsets.

* **Others – Numpy, Pickle:**

NumPy is used for efficient numerical operations and array handling. Pickle is used to serialize the trained label encoder and model weights for later use.

* Dataset & Preprocessing

The dataset was structured into directories by person IDs. Each person's folder contained a set of facial images. The steps involved:

* Looping through all images and detecting faces using MTCNN.
* Cropping and resizing faces to 160x160 pixels.
* Generating 512-dimensional embeddings using FaceNet.
* Encoding labels using LabelEncoder and one-hot encoding for model compatibility.

The dataset was split into 80% for training and 20% for validation and testing, where the same validation set was used during training to monitor performance.

* Model Architecture

A fully connected feedforward neural network was trained to classify the 512-dimensional face embeddings generated by FaceNet. This architecture is simple, fast, and effective for small-scale face classification tasks:

* **Input Layer**: Accepts 512-dimensional embeddings.
* **Dense Layer 1 (128 units, ReLU)**: Reduces dimensionality and captures complex feature patterns. ReLU (Rectified Linear Unit) is used for its computational efficiency and ability to mitigate vanishing gradient issues. The ReLU activation function is mathematically defined as: ReLU(x) = max(0, x)

This means that any negative input is set to zero, while positive inputs remain unchanged. This non-linear transformation allows the network to learn complex relationships while avoiding saturation effects seen in other activation functions like sigmoid or tanh.

* **Dropout Layer (0.3)**: Applied after the first dense layer to randomly deactivate 30% of the neurons during each training iteration, helping to prevent overfitting.
* **Dense Layer 2 (64 units, ReLU)**: Further compresses the representation while learning more abstract features.
* **Dropout Layer (0.2)**: Another dropout layer with a lower rate (20%) to retain more information while still regularizing.
* **Output Layer (Softmax)**: The final dense layer outputs probabilities for each class using the softmax activation function, which is ideal for multiclass classification problems. The class with the highest probability is selected as the predicted identity.
* **Loss Function**: Categorical Crossentropy was chosen as it is suitable for multi-class classification where labels are one-hot encoded.
* **Optimizer**: Adam optimizer was used for its adaptive learning rate and efficient convergence in deep learning.
* **Epochs**: 25
* **Batch Size**: 32

This model structure is efficient for real-time applications, balancing performance and computational cost. Since we relied on pre-trained embeddings, a shallow neural network is sufficient for classification without requiring deep architectures.

A diagram of a function

AI-generated content may be incorrect.

Figure 11 neural network model architecture diagram

* Results

Training Dataset Size: 80% of total images

Test Accuracy: 100%

The trained model performed well on known faces, providing high confidence scores. For unknown faces, a confidence threshold of 98% was applied to label them as 'unknown'.

* **Accuracy and Training Progress**

To better illustrate the training process, the following graph shows the accuracy over the training epochs:

"Figure 1 below shows the training and validation accuracy across 25 epochs. The model showed steady learning and converged effectively without over-fitting."

A graph with red and green lines

AI-generated content may be incorrect.

Figure 12: Accuracy & loss over epochs

* **Security and Privacy Considerations**:
* No personal data is sent outside the classroom network.
* Data is stored only on the instructor's device.
* The system works without internet, ensuring no external access.
* Camera access is granted only upon user consent and over HTTPS.
* **Challenges & Solutions**

Low Image Quality or Misalignment: Some faces were not properly detected due to poor lighting or occlusion.

Solution: Applied consistent preprocessing and used MTCNN for robust face detection.

Small Dataset: The model was trained on a relatively small number of individuals.

Solution: Used transfer learning via FaceNet to benefit from a pre-trained embedding space, reducing the need for large data.

Real-Time Performance: Ensuring smooth webcam prediction without lag.

Solution: Optimized prediction code and limited detection to one face at a time for speed.

* **Backend**
* Current Session Detection
  + Identifies current lectures/sections for the authenticated student based on current day/time and student's registered courses/sections.
  + Verifies valid session times using isTimeBetween helper.
  + Checks student enrollment by cross-referencing student.registeredCourses and section.registeredStudents.
* Attendance Marking
  + Requires encryptedStudentId, courseCode, sessionType, sectionId (if section).
  + Decrypts student IDs using cryptoHelper.decryptStudentId and process.env.ENCRYPTION\_KEY.
  + **Validates session timing**: checks if a lecture/section for the course is currently active.
  + Verifies student registration for the course/section (by checking existing attendance records as a proxy).
  + Prevents duplicate entries for the same student, course, session type (and section ID) on the same day.
  + Records attendance details in student.attendance array (course, type, date, time, status 'Present', room, timeslot).
* Attendance Statistics
  + Statistics (getAttendanceStats - Student):
    - Student views own stats (present, late, absent, percentage), filterable by course/session.
    - Calculates absent/percentage based on total expected sessions if course code provided.
  + Reporting (generate Attendance Report - Doctor/TA/Admin):
    - A diagram of a program

      AI-generated content may be incorrect.Generates CSV exports of attendance for a course lecture (Doctor/Admin) or section (TA/Admin).

**Figure 13** Attendance Sequance diagram

3.4 performance system (Grades)

* **Overview**

The performance is a core component of any university information system. It empowers academic staff to manage student grades, compute GPA/CGPA, monitor academic progress, and execute term-based academic processing. The module supports grade handling for midterm, work, and final components, enables CSV/Excel uploads, and integrates seamlessly with student course and section & lecture registration data

* Importance of the GPA Module
* Provides detailed performance reports for students and administrative review.
* Enables instructors and TAs to record grades efficiently.
* Calculates accurate GPA and CGPA for real-time academic standing.
* Supports automated term processing such as promotions and credit eligibility.
* Enforces role-based access control (RBAC) to ensure secure grade management.
* **Backend**
  + End points
* Adding Grades
* POST /:courseCode/:studentId/midterm
* POST /:courseCode/:sectionId/:studentId/work
* POST /:courseCode/:studentId/final
* **Roles:**
* Doctors: midterm & final
* TAs: work
* Updating Grades
* PUT /:courseCode/:studentId/midterm
* PUT /:courseCode/:studentId/work
* PUT /:courseCode/:studentId/final
* Deleting Grades
* DELETE /:courseCode/:studentId/midterm
* DELETE /:courseCode/:studentId/work
* DELETE /:courseCode/:studentId/final
* Bulk Uploads (CSV/Excel)
* POST /upload/:componentType/:courseCode
* POST /upload/work/:componentType/:courseCode
* Fetching Grades
* GET /students/:studentId/grades
* GET /courses/:courseCode/grades
* GET /:courseCode/:studentId/breakdown
* Access Control:
* Students: Can only view their own grades
* Doctors/Admins: Can access extended data
* Performance Metrics (Student View)
* Term GPA
* Cumulative GPA (CGPA)
* Current Academic Level
* Passed and Failed Courses
* **Grade breakdown for all registered courses:**
  + Midterm (30%)
  + Work (20%)
  + Final (50%)
* Grade Logic & Weighting
* Handle Grade Component: Universal logic for add/update/delete
* Validates: Roles, registration, score bounds
* Updates: Student performance if final grade is modified
* Weighting:
  + Midterm: 30%
  + Work: 20%
  + Final: 50%
* Bulk Uploading Grades
* uploadGradesFile: Parses .csv or .xlsx, validates data, and saves
* Recalculates student GPA/CGPA if **final grades** are included
* GPA & CGPA Calculation
* calculateTermGPA: GPA for a single term
* calculateCGPA: Cumulative across terms
* getLetterGrade(score): Converts score → A/B/C...
* getGradePoints(letter): Converts letter → numeric value
* Student Performance Tracking
* updateStudentPerformance: Updates academic profile:
  + Term GPA
  + CGPA
  + Academic Level
  + Failed/passed course lists
* calculateAcademicLevel: Maps earned credit hours → academic level
* calculateMaxAllowedCreditHours: Adjusts term credit load based on CGPA
* Automatic Term Processing
* processTermCompletion: Finalizes term data
* Stores academic history, clears course registrations
* Runs via CRON on May 15 and December 15

3.5 Exam System

* **Overview**

This module handles the lifecycle of exams within the academic system, from creation and updates to student scheduling and conflict detection. It ensures exams are uniquely managed per department and provides tools for both administrators and students to track and manage exam logistics.

* **Backend**
* Exam Creation & Management (Admin - adminController)
  + Creation (addExam):
    - Generates unique exam IDs (scoped by department).
    - Requires examId, courseCode, examDate (YYYY-MM-DD), startTime, endTime, array of roomNumbers, department.
    - Prevents department duplicates: checks if examId for the department already exists.
    - Uses MongoDB transactions.
  + Updates (updateExam): Modifies exam details; re-distributes students if room setup changes.
  + Deletion (deleteExam): Removes exam and cleans up student references.
  + Retrieval (getAllExams): Lists all exams with course details and student counts per room.
* Student Exam Schedule Viewing & Conflict Detection
  + Links exams to student profiles: addExam adds exam ID to student.exams and examRooms map.
  + getStudentExams retrieves exams for the student's registered courses in the current term.
  + Provides exam schedules: includes course name, date, time, assigned room.
  + A close-up of a diagram

    AI-generated content may be incorrect.Conflict Detection: Identifies and flags overlapping exams (same date, overlapping time slots) in the student's schedule.

**Figure 14 :**Exams Sequence diagram

3.6. Announcements System

* **Overview**

Provides a centralized system for posting important course- or university-related updates.Replaces informal methods like WhatsApp with a secure and structured communication channel. Accessible to doctors, teaching assistants (TAs), and admins for relevant student notifications.

* **Backend**
* Functionalities
* Validates required fields: title and content (max 5000 characters).
* Ensures doctors are assigned to the course, and TAs to the section.
* Stores sender details (name, profile picture, role) and links to course/section if applicable.
* Students are restricted from posting.
* Update Announcement:
* Only the original sender or an admin can edit.
* Validates the same fields as creation.
* Allows updating title, content, and timestamp.
* Delete Announcement:
* Only the original sender or admin can delete
* Performs a soft delete by setting isDeleted = true
* getAnnouncementsFor Student
* Fetches announcements based on user role
* **Student:** Admins, course doctors, section Tas
* **Doctor:** Admins, self, and their courses
* **TA:** Admins, self, and their sections
* **Admin:** All announcements
* Filters out deleted items (isDeleted: false).
* Includes course names using a courseMap.
* **Enhancement:** Course names are included using a courseMap.
* **Filtering:** Only non-deleted announcements (isDeleted: false) are shown.
  + **Access Control:**
* Uses req.user.role and req.user.id for role-based access control.
* Students cannot create or modify announcements.
* Doctors and TAs are checked for course/section ownership.
* Admins have full permissions: create, update, delete, and view all.

**3.7 Fees system**

* **Overview**

University systems often lack centralized and automated fee management. Handling tuition, department fees, and overdue payments manually can be time-consuming and error-prone. This project introduces a **University Fee Management System**, part of a broader university system that manages how students are assigned fees, how they view and pay them, and how admins monitor payment statuses.

By providing role-based access, real-time updates, and data integrity, this system enhances the university's ability to manage financial workflows efficiently

* **Backend**
* View Fees (getFeesByStudent):
* Students can log in and view their personal fee records, including amount, status, and due date.
* Only the authenticated student's data is shown, preventing access to others’ records.
* Track Payment Status:
* Students see real-time updates of fee status (e.g., Pending, Paid).
* Payment timestamps are visible once fees are marked as paid by admins.
* Access Control:
* Students can only view their own fees; they cannot add, edit, or delete any fee records.
* Authorization is enforced using req.user.role and req.user.id to prevent unauthorized access.
* Functionalities(Admin):
  + - Add a New Fee (Admin only):
* **Admin selects:**
* academic level (e.g., year 1, year 2)
* department
* amount
* due date
* **The controller:**
* Finds all students in that level and department.
* Prevents adding duplicate fees for the same group.
* Saves the fee in the Fee collection and also inside each student’s fees array.
* updateFeeStatus – Mark Fee as Paid:
* Updates a student’s payment status (Pending ➝ Paid) for a specific feeId.
* Sets a paidAt timestamp when marked as paid.
* deleteFee – Delete a Fee
* Deletes the fee record from the Fee collection.
* Removes that fee from all affected students' profiles.
* getFeesByStudent – Student Views Their Fees
* Fetches the logged-in student's fees from their profile.
* Prevents students from viewing others’ data (authorization check).
* getAllFees – Admin Gets All Fee Records
* Lists all fee records across departments and levels for review or auditing.

**3.8. Course system (Admin)**

* **Overview**

This module manages the entire lifecycle of academic courses and their sections. It includes course creation, updates, deletions, and assignment of instructors and TAs. Admins control this module, ensuring consistency and proper role-based associations between courses, instructors, and students.]

* **Backend**
* Course Creation (adminController.addCourses)
  + Handles single or bulk course creation.
  + Validates unique course codes (code).
  + Ensures doctorId exists and belongs to a user with 'doctor' role. Sets doctorName.
  + Sets defaults for creditHours, semester, startDate, endDate, department, capacity if not provided.
  + Adds course to doctor's assignedCourses.
* Section Management (within Courses)
  + Creation (adminController.addSection to course courseCode):
    - Validates sectionId uniqueness within the course, taId (if provided, must be 'ta' role), and that new section capacity doesn't exceed overall course capacity.
    - Assigns section to TA's assignedSections.
  + Deletion (adminController.deleteSection from course courseCode):
    - Removes section from students' registeredSections and TA's assignedSections.
  + Updates (adminController.updateSection for course courseCode):
    - Handles TA re-assignment (updates old/new TA's assignedSections).
  + Retrieval (adminController.getAllSections): Retrieves all sections across all courses, populating TA and doctor names.
* Course Updates (adminController.updateCourse)
  + Modifies course details based on req.params.code.
  + Handles Doctor reassignment: updates assignedCourses for old and new doctors.
  + Updates TAs' assignedSections if section details (like taId) are part of the course update.
* Course Deletion (adminController.deleteCourse)
  + Removes course references based on req.params.code.
  + Cleans up student registrations: removes course from student.registeredCourses and related sections from student.registeredSections.
  + Updates Doctor assignments: removes course from doctor.assignedCourses.
  + Removes sections from relevant TAs' assignedSections.
* Course Retrieval
  + adminController.getAllCourses: Lists all courses.
  + adminController.getCoursesByDoctor: Lists courses for a specific doctor.

**3.9 Chat System**

* **Overview**

The chat system is designed to facilitate direct and secure communication between students and doctors within a university management system. It allows users to:

* Search for other users by name or ID.
* Create or access one-to-one private chat conversations.
* Send and receive messages in real-time.
* View the chat history.

This helps solve the communication gap that often exists between students and instructors by providing a dedicated space for academic discussions, clarifications, and private queries — especially outside lecture hours.

* **Backend**
* 7.1 Endpoints and Their Roles:
  + - * GET /api/chat/users?id=USER\_ID&search=KEYWORD
* Purpose: Search for users (by name or ID) to start a chat with.
* How it works:
* Returns users that match the keyword excluding the current user.
* Supports both numeric ID and text name search.
* POST /api/chat/
* Body: { id: "SENDER\_ID", targetId: "RECEIVER\_ID" }
* Purpose: Create or access a one-to-one chat between two users.
* How it works:
* If a chat already exists between the two, it returns that chat.
* Otherwise, it creates a new chat document and returns it.
* GET /api/chat?id=USER\_ID
* Purpose: Get all chats that a user is involved in.
* How it works:
* Returns all one-to-one chats where the user is a participant.
* Populates user data and latest message in each chat.
* Sorted by updatedAt descending.
* POST /api/chat/message
* Body: { id: "SENDER\_ID", chatId: "CHAT\_ID", content: "Hello!" }
* Purpose: Send a new message in a chat.
* How it works:
* Creates a message with sender, chat, and content.
* Adds the sender to the seenBy array.
* Updates the latestMessage field of the chat.
* GET /api/chat/message/:chatId?id=USER\_ID
* Purpose: Get all messages from a specific chat.
* How it works:
* Checks if the requesting user is part of the chat.
* Returns all messages, sorted from oldest to newest

3.10 Complaint Management System

* **Overview**

It allows users (students, doctors, TAs) to send complaints, and enables administrators to view and resolve them. The system helps improve communication, accountability, and responsiveness between the institution and its users.

* Centralized Complaint Handling: Users can submit their issues through a structured platform.
* Transparency: Admins can easily track and manage complaints by their status.
* Efficiency: Quick resolution through digital record-keeping and status updates.

Accountability: Each complaint is tied to a verified user and timestamped

* **Backend**
* **8.1 API Endpoints Description**
* **8.1.1 Send a Complaint**
* **Method**: POST
* **URL**: /api/complaints/send *(example endpoint path)*
* **Purpose**: Allows a student, doctor, or TA to submit a complaint
* **8.1.2 Get All Complaints (Admin Only)**
* **Method**: GET
* **URL**: /api/complaints
* **Purpose**: Retrieves a list of all complaints along with user info (name and email).
* **8.1.3 Resolve a Complaint (Admin Only)**
* Method: PATCH
* URL: /api/complaints/:complaintId/resolve
* Purpose: Updates the complaint's status to either "Pending" or "Resolved"

A diagram of a software development

AI-generated content may be incorrect.

**Figure 15 GPA management API**

3.11 chatbot

* **Overview**

a digital college chatbot designed to help students access their academic information in a faster, more intuitive way. The chatbot works seamlessly across web browsers and mobile apps, offering students a smooth, conversational interface instead of complex academic portals. It understands natural language queries and responds instantly with accurate academic information like courses, grades, schedules, exams, announcements, and complaint statuses. The goal of this project is to improve accessibility and reduce the time students spend navigating systems manually, replacing them with a modern, intelligent experience

* Technologies and Tools Used
* **User Interface Design**: The chatbot was designed to operate across both web and mobile devices. The interface is responsive, modern, and built around student use cases, ensuring a consistent experience on various screen sizes and operating systems.
* **Natural Language Input Understanding**: The chatbot supports a wide range of student queries written in informal or flexible language. For instance, students can type "when is my calculus exam?" or "show my timetable" and still receive accurate responses. This was implemented using structured keyword recognition and phrase normalization.
* **Personalized Data Display**: Every student uses their own ID to access the system, which links directly to their academic data, including enrolled courses, grades, exam schedules, and more. This ID-based personalization ensures that all responses are private and specific to the logged-in user.
* **Offline Mobile Functionality**: On the mobile version, key information like the latest schedule or recorded grades is saved for offline viewing. This allows students to access important data even without internet connectivity.
* **Smart Query Resolution**: The chatbot automatically handles common course name shortcuts and variations. For example, queries like "prob stat" are correctly interpreted as "Probability and Statistics", and "programming 1" is resolved to "Programming I". This is achieved through a mapping of common aliases to their formal names.
* **User Experience (UX)**: The chatbot’s responses are formatted using easy-to-read tables and bullet points when appropriate, helping students absorb information quickly. This design mirrors the clarity of official university documents, but with the flexibility of conversational input.
* Dataset & Preprocessing

The dataset powering the chatbot consists of structured records grouped under collections such as students, courses, grades, exams, complaints, and announcements. Each student profile includes an ID, name, and a list of enrolled course codes. Courses contain attributes such as instructor ID, credit hours, lecture times, and sections. Grades are linked to specific course codes and contain detailed score information.

Before the chatbot returns a response, it performs query preprocessing:

* Cleans up redundant terms like "info", "about", or "course".
* Applies synonym mapping to unify terms (e.g., “TA” and “instructor”).
* Filters the dataset by student ID to ensure personalized results.
* Extracts relevant fields based on the intent (e.g., scores and terms for grades; days and times for schedules).

The result is a system that transforms raw academic data into direct, relevant responses tailored for each student.

* **Core Functionalities**
* **Course Lookup**: Students can ask for all available courses, filter by department, or retrieve only the ones they are enrolled in.
* **Grade Reports**: Returns a detailed list of all courses taken, with scores, letter grades, terms, and grading dates.
* **Exam Schedules**: Shows all exams, upcoming ones, or those filtered by a specific course. Each entry includes type (midterm/final), date, time, and room.
* **Timetable Builder**: Compiles lectures and sections into a weekly class schedule that includes instructor names, session types, and room numbers.
* **Instructor Information**: Allows the student to ask who teaches a course, returning the instructor’s name, email, and other courses they teach.
* **Complaint Tracking**: Displays previously submitted complaints with status updates (e.g., “Pending”, “Resolved”).
* **Announcements Feed**: Retrieves the latest course or university announcements, filtered by the student’s enrolled classes.
* **Prerequisite Validation**: Checks if the student has completed required courses before taking a higher-level one. If prerequisites are missing, the system lists them clearly.
* **Query Understanding and Response Generation**

The chatbot uses pattern-based recognition to interpret a wide variety of student queries. It detects key phrases like:

* “Check prerequisites for a course”
* “When is my linear algebra class?”
* “Info for Stochastic Processes”
* “Who teaches programming 2?”

After identifying the intent, the system searches the appropriate dataset and returns a structured response. If the request is ambiguous or incomplete, the chatbot responds with a polite prompt asking for clarification (e.g., “Which course do you mean?” or “Please enter a valid course code”).

Additionally, it uses caching for frequent lookups (like instructor names) to reduce response time and improve performance.

* **Results**

**Test Coverage**: The chatbot was tested with over 100 simulated student profiles. Each profile included varied combinations of courses, grades, and complaints.

* **Performance Outcomes**:
* **Query Understanding Accuracy**: ~96% of student queries were correctly interpreted on the first try.
* **Response Time**: Most results were displayed instantly or within 1 second.
* **User Feedback**: Test users appreciated the fast, clear layout and conversational tone.
* **Offline Resilience**: On mobile, cached grades and timetables remained visible even when the network was disconnected.
* **Challenges & Solutions**
* **Flexible Language Input**: Students used different styles of phrasing and abbreviations.
  + *Solution*: Introduced regex patterns, keyword libraries, and alias dictionaries to normalize input.
* **Missing or Incomplete Data**: Some students lacked grade records or had not submitted any complaints.
  + *Solution*: The chatbot responds with helpful default messages like “No grades found yet” or “You have no complaints on record”.
* **Unclear Course References**: Students sometimes asked about courses using vague terms.
  + *Solution*: The chatbot asks for clarification or shows multiple matching options.
* **Cross-Platform Consistency**: Ensuring the same experience on mobile and desktop.
  + *Solution*: Responses follow a shared formatting standard and layout across all interfaces.

**Chapter 4: Security Framework**

The security architecture is built around authentication, authorization, and data protection principles to ensure system integrity and user privacy.

* **4.1 authentication System (JWT Details)**
* JSON Web Tokens (JWT)
  + All API requests (except public ones like login/forgot-password) require authentication via JWT tokens passed in the Authorization: Bearer <token> header. This is handled by authMiddleware.js.
  + Token verification includes signature validation (using process.env.JWT\_SECRET) and expiration checks (e.g., 1 hour).
  + Invalid, missing, or expired tokens result in immediate 401 Unauthorized responses from authMiddleware.
* User Verification
  + After token validation, authMiddleware fetches the user from the database using the id from the token payload.
  + If the user is not found (e.g., deleted after token issuance), access is denied (401 User not found), effectively terminating the session for that token.
* **4.2 Authorization Middleware System**
* 4.2.1 Role-Based Access Control (RBAC) Definitions
  + The system implements a hierarchical role structure with distinct privileges enforced by middleware like isAdmin, isDoctor, isTA, isStudent.
  + Admin Role: Full system access; can override all permission checks (implicitly, by having access to admin controllers or specific checks within middleware); manages system-wide operations.
  + Doctor Role: Manages courses they are assigned to, including grades for those courses; cannot modify system-wide settings.
  + Teaching Assistant (TA) Role: Limited to specific course areas (e.g., sections they are assigned to via isCourseDoctor middleware); cannot modify system-wide settings.
  + Student Role: Restricted to personal data access; can only perform self-related actions (enforced by enforceSelfAction and other checks); no access to other users' information.
* 4.2.2 Specialized Authorization Modules & Implementation
  + Course Authorization (isCourseDoctor.js):
    - Validates course and section existence.
    - Ensures doctors only manage their assigned courses (checks course.doctorId against req.user.id).
    - Verifies TAs are assigned to specific sections (checks section.taId against req.user.id).
    - Includes detailed console logging for troubleshooting authorization attempts.
  + Self-Action Enforcement (authMiddleware.enforceSelfAction):
    - Specifically for 'student' role, prevents students from acting on others' behalf.
    - Validates all ID parameters (userId, studentId, etc.) in req.params and req.body against req.user.id.
    - Automatically overrides these ID parameters with the authenticated student's req.user.id to protect against parameter tampering.
  + Profile Picture Management (authMiddleware.canManageProfilePicture):
    - Restricts profile picture modifications.
    - Admins can manage any profile (passes next()).
    - Other roles (student, doctor, TA) can only modify their own profile (checks if req.params.userId if present matches req.user.id, or defaults to self-action).
  + Attendance System (authMiddleware.canMarkAttendance):
    - Students can only mark their own attendance (checks req.params.studentId against req.user.id).
    - Admins can mark for others (passes next()).
    - Prevents fraudulent attendance reporting by restricting who can mark for whom.
  + User Data Access (authMiddleware.canAccessUserData):
    - Implements strict data isolation.
    - Admins bypass restrictions (passes next()).
    - Users can only access their own data: if req.params.userId or req.body.userId is present, it must match req.user.id; otherwise, defaults to req.user.id.
    - Automatic parameter sanitization by overriding IDs with req.user.id.
  + Combined Role Checks (authMiddleware.isAdminOrDoctor):
    - Specialized checks for scenarios requiring either admin or doctor privileges.
    - Maintains security while allowing necessary access for these combined roles.
* **4.3 Core Security Principles**
* Minimum Access: Users get only what they need (enforced by RBAC and specialized middleware).
* Double-Checking: Important actions have multiple verifications (e.g., token + role + specific conditions like course ownership).
* Input Checking: Request data is validated by controllers (e.g., for data types, ranges, existence of entities) before processing to block bad/suspicious requests.
* Activity Logs: While not explicitly in the provided JS, detailed console logs in middleware like isCourseDoctor contribute to tracing; more extensive logging would be a best practice.
* **4.4 Best Practices**
* Data Encryption: All data transfers are encrypted (HTTPS - this is a deployment concern, not code). Sensitive data at rest (like passwords via bcrypt) is encrypted.
* Token Management: JWTs expire quickly (1 hour) and are not designed to be easily reused across different contexts if compromised (though full invalidation on demand is complex with JWTs).
* Error Handling: Controllers return appropriate HTTP status codes and simple JSON error messages to users, avoiding technical details in responses.
* Session Management: authMiddleware effectively blocks deleted users by re-fetching user on each authenticated request.
* **4.5. API Endpoints**
* Authentication
* POST /auth/login – User login and JWT token issuance
* POST /auth/forgot-password – Request OTP to reset password
* POST /auth/reset-password – Reset password using OTP
* POST /auth/change-password – Change password (requires old password)
* User Management
* POST /users/create – Create a single or multiple users
* PUT /users/:id – Update user by ID
* DELETE /users/:id – Delete user by ID
* GET /users – Get all users
* GET /users/:id – Get user by ID
* POST /users/:id/upload-picture – Upload or update profile picture
* Course Management
* POST /courses – Create a new course
* GET /courses – Get all courses
* GET /courses/:id – Get course by ID
* PUT /courses/:id – Update course by ID
* DELETE /courses/:id – Delete course by ID
* POST /courses/:courseId/sections – Add section to course
* PUT /courses/:courseId/sections/:sectionId – Update section
* DELETE /courses/:courseId/sections/:sectionId – Delete section
* Registration
* POST /register/course – Register for a course
* POST /register/section – Register for a section
* DELETE /register/drop-course/:id – Drop a registered course
* DELETE /register/drop-section/:id – Drop a registered section
* Exams
* POST /exams – Create exam schedule
* GET /exams – Get all exams
* GET /exams/:id – Get exam by ID
* PUT /exams/:id – Update exam
* DELETE /exams/:id – Delete exam
* Attendance
* POST /attendance/lecture – Mark attendance for lectures
* POST /attendance/section – Mark attendance for sections
* GET /attendance/current-sessions – Get current sessions
* GET /attendance/stats/:courseCode/:sessionType – Get attendance stats
* GET /attendance/report/lecture/:courseCode – Get attendance report
* Grading / GPA
* POST /grades – Enter grades
* GET /grades/student/:id – Get student grades
* PUT /grades/:id – Update grade
* POST /grades/upload – Bulk upload of grades via CSV or Excel
* Performance
* POST /:courseCode/:studentId/midterm - add a midterm grade for a student
* POST /:courseCode/:sectionId/:studentId/work- This allows a TA or admin to add a work grade
* POST /:courseCode/:studentId/final- doctors or admins to enter a final exam grade
* POST /upload/:componentType/:courseCode- allows doctors or admins to bulk upload grades
* POST /upload/work/:componentType/:courseCode- bulk upload work grades for students using a CSV or Excel file.
* GET /students/:studentId/grades- Returns all grades (midterm, work, and final)
* GET /courses/:courseCode/grades- Retrieves all grades submitted
* GET /:courseCode/:studentId/breakdown- detailed breakdown of the student’s grades
* GET /performance/:studentId- Returns a full academic performance report for a student
* GET /doctor/courses- Returns all courses assigned to the logged-in doctor
* GET /ta/sections- Returns all course sections Only accessible by TAs..
* GET /grades/:studentId- Allows an admin to retrieve all grades for a specific student.
* GET /get-grade/:studentId- Enables an admin to filter and retrieve grades
* GET /get-all-grades- retrieve all grades in the system with support for pagination

**Chapter 5: Project Management**

* **5.1. Introduction**

In the previous chapter, we discussed the design of our application. In this chapter, we will discuss the management of our project. The methodology we followed to achieve big milestones in the project

* **5.2. Methodology**

The best methodology to use for developing a university system depends on the project's complexity, requirements, and stakeholders. However, Agile methodology is often the most suitable approach for such systems, especially in dynamic and evolving environments like educational institutions.

* 5.2.1. Implementation

Plan After having a first draft of the project’s idea and thought of the implementation process and its requirements, we started drafting an implementation plan. The implementation plan was mainly divided into four parts along the time of the two parts of the project:

1. Research: where we researched the techniques we had in mind, to increase our knowledge wider about it and learn about their latest releases and newer and more advanced substitutes.
2. Writing in book : where we would write the chapters in the book
3. Learning: where we would learn something practically for the sake of implementation.
4. Implementing & Testing: where we would implement our model and application and test them all together.

## **5.2.2 Project Timeline and Sprint-wise Completion Report (6-Month Plan)**

### 

* Month 1 – Phase 1: Project Setup and Core Modules
* Requirements were defined and detailed user scenarios were created.
* Key features were identified and finalized based on initial research and stakeholder input.
* The system architecture was designed, covering front-end, back-end, and API layers.
* The database schema was modeled and reviewed for scalability and efficiency.
* Month 2 – Phase 2: Project Interface Design (UI/UX)
* Wireframes and low-fidelity UI designs were developed for all core pages.
* Feedback sessions were conducted with users and revised accordingly.
* High-fidelity UI/UX designs were created using design tools (e.g., Figma/Adobe XD).
* A design system (colors, typography, components) was established for consistency.
* Month 3 – Phase 3: Core Features Implementation (Registration, Performance, Home)
* User Registration and Login modules were implemented and tested.
* Basic navigation and role-based access control were set up.
* Home dashboard and performance tracking features were developed.
* Initial validations and form handling were added.
* Month 4 – Phase 4: Additional Features (Chatbot, Attendance)
* The AI chatbot module was integrated with a basic NLP engine.
* Initial training datasets were uploaded and tested
* Attendance tracking system was developed using facial recognition or manual input.
* Features were integrated with user roles (student, teacher, admin).
* Month 5 – Phase 5: Reports and Integration
* Analytics and reporting dashboards were created using charts and summary views.
* External tool integration (e.g., LMS or payment gateway) was initiated.
* All integrated components were tested in a staging environment.
* Report generation and export options (PDF, Excel) were finalized.
* Month 6 – Phase 6: Final Testing and Deployment
* Full system testing was conducted: unit, integration, and user acceptance testing.
* UI polishing and bug fixing were completed based on feedback.
* The final system was deployed to the production server.
* User training sessions and documentation were delivered successfully.
* 5.2.3. Weekly Meetings and MOMs Every week,

we would hold a meeting to catch up with the latest updates in our progress and discuss next steps. After every meeting, we write Minutes of Meetings where it’s a documentary of the meeting listing the attendees, time and date, place of meeting, the points discussed in the meeting, and the action items to be done until the next meeting.

* **5.3 Cost Estimation**

To implement the University Management System (UMS) as a Minimum Viable Product (MVP) within a limited budget, we have optimized development tasks, chosen open-source tools, and prioritized only the most essential features. The total estimated budget is set to **a maximum of 25,000 EGP**, targeting student-led or freelance implementation.

* + - * Backend Development

The backend will handle core functionalities including user authentication, course registration, attendance tracking, and basic grade management. Development will use Node.js, Express, and MongoDB. Estimated cost: **8,000 EGP**.

* Mobile Application (Student App Only)

A mobile app will be developed using Flutter for students to access courses, grades, announcements, and complaints. The admin version will be excluded at this stage to save costs. Estimated cost: **5,000 EGP**.

* Face Recognition System

Instead of building a custom face recognition model, the project will utilize free-tier APIs such as **FaceIO** or **Firebase ML Kit**. This ensures basic functionality with no additional cost. Estimated cost: **0 EGP**.

* Chatbot (Rule-Based)

A simple, rule-based chatbot will be developed using **Google Dialogflow (free edition)** to respond to basic academic queries like schedules, grades, and announcements. Estimated cost: **1,000 EGP**.

* Web Interface (Admin Panel)

A minimal web dashboard for administrators will be created using **React.js**, focused on course management, user control, and grading. The interface will be basic but functional. Estimated cost: **3,000 EGP**.

* UI/UX Design

The system’s user interfaces will be designed using free tools like **Figma**, leveraging community templates and basic customizations. Estimated cost: **1,000 EGP**.

* 7. Testing and Debugging

Manual testing will be conducted across critical modules to ensure system stability. This includes basic unit tests, integration flow checks, and user feedback sessions. Estimated cost: **1,000 EGP**.

* 8. Hosting and Deployment

Hosting and deployment will utilize free-tier platforms including **Vercel** (for the web frontend), **Firebase** (for mobile and backend services), and **MongoDB Atlas** (for the database). Estimated cost: **0 EGP**.

* 9. Documentation and Project Management

This includes the preparation of user guides, system documentation, and final presentation materials. Estimated cost: **1,000 EGP**.

* 10. Contingency Buffer

A contingency amount is reserved for unexpected development needs, emergency debugging, or scope adjustments. Estimated cost: **2,000 EGP**.

* **Total Estimated Cost: 22,000 – 24,000 EGP**

This budget-focused plan ensures that the core UMS functionality is delivered within financial constraints, leveraging open-source tools, free APIs, and student-level development effort while leaving a small buffer for flexibility.

Chapter 6: Testing Strategy

**6.1 Introduction**

To ensure the reliability, accuracy, and stability of the University Management System (UMS), a comprehensive testing strategy was implemented. Testing spanned all critical modules — including authentication, registration, attendance, performance tracking, and announcements — and involved automated, integration, and user acceptance testing.

**6.2 Testing Types**

* 6.2.1 Unit Testing

Each backend controller, middleware, and service function was tested in isolation to validate correct logic and error handling.

* **Tools Used**: Jest (for JavaScript backend), Postman (manual API testing)
* **Examples**:
  + Testing authController.login for valid/invalid credentials.
  + Ensuring studentController.registerForCourses enforces CGPA and prerequisite rules.
  + Verifying attendanceController.markAttendance prevents duplicates and validates sessions.
* 6.2.2 Integration Testing

End-to-end flows across multiple modules were tested to ensure seamless data interaction.

* **Tested Scenarios**:
  + Student registration → course registration → attendance → performance calculation.
  + Admin fee assignment → student payment → status update.
  + Doctor posting an announcement → student receiving/viewing it.
* 6.2.3 Manual Functional Testing

All core frontend functionalities were manually tested by team members and beta users across web and mobile platforms.

* **Devices Used**: Android, iOS, and modern web browsers.
* **Functional Flows Covered**:
  + Face recognition attendance via webcam.
  + Chat system interactions.
  + Complaint submission and resolution tracking.
* 6.2.4 User Acceptance Testing (UAT)

Students, teaching assistants, and faculty members participated in real-world scenario testing and provided feedback.

* **Focus Areas**:
  + Usability and navigation.
  + Feature completeness.
  + Real-time feedback and responsiveness.
  + Role-based access and data visibility.
* 6.2.5 Load & Performance Testing

To ensure scalability under high usage:

* Simulated 100+ concurrent users performing login, registration, and attendance operations.
* **Result**: System maintained stability with response times under 2 seconds for all major operations.

**6.3 Bug Tracking and Resolution**

* Bugs were tracked using GitHub Issues.
* Each bug was assigned, resolved, and tested within the same sprint cycle.
* Regression testing was performed before each major feature integration.

**conclusion**

The developed system successfully delivers a comprehensive solution for managing key academic processes such as course registration, grade tracking, GPA calculation, performance monitoring, and exam scheduling etc. By supporting role-based access for students, teaching assistants, doctors, and administrators, the system ensures secure and efficient handling of sensitive academic data. The combination of a user-friendly interface and robust backend logic allows for smooth operation, accurate record-keeping, and enhanced transparency across all academic levels.

This system not only simplifies administrative workflows but also empowers students by giving them real-time access to their academic performance, course schedules, and exam arrangements. The integration of automation—such as GPA updates and term processing—further enhances the efficiency and reliability of the platform.

**Future Work**

To further improve the system and expand its capabilities, the following features can be considered for future development:

* **Real-Time Notifications**  
  Implement email or in-app notifications for grade updates, exam schedules, or registration deadlines..
* **Analytics Dashboard**  
  Develop dashboards for students and admins to visualize academic trends, GPA progress, and course statistics using charts and graphs.
* **Mobile App Enhancements**  
  Improve the mobile app experience by adding offline access, push notifications, and biometric login.
* **Plagiarism Checker**  
  Integrate a basic plagiarism detection tool for assignments uploaded in the system.
* **Multi-language Support**  
  Provide the option for users to switch between languages (e.g., English/Arabic) for better accessibility.
* **AI-Based Performance Prediction**  
  Use historical data to help predict student risk levels (e.g., failure risk) and provide academic support recommendations
* **File attachment**   
  Dr can attach a file or a lecture pdf for students to see

**References**

1. **[Schroff, F., Kalenichenko, D., & Philbin, J. (2015).]( https:/www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Schroff_FaceNet_A_Unified_2015_CVPR_paper.pdf) *[FaceNet: A Unified Embedding for Face Recognition and Clustering]( https:/www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Schroff_FaceNet_A_Unified_2015_CVPR_paper.pdf)*[. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 815–823.]( https:/www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Schroff_FaceNet_A_Unified_2015_CVPR_paper.pdf)**
2. **[Zhang, K., Zhang, Z., Li, Z., & Qiao, Y. (2016).](https://arxiv.org/pdf/1604.02878.pdf) *[Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks](https://arxiv.org/pdf/1604.02878.pdf)*[. IEEE Signal Processing Letters.](https://arxiv.org/pdf/1604.02878.pdf)**
3. **TensorFlow. (n.d.). *An end-to-end open source platform for machine learning*. [https://www.tensorflow.org/](https://www.tensorflow.org/" \t "_new)**
4. **Keras. (n.d.). *The Python Deep Learning API*. [https://keras.io/](https://keras.io/" \t "_new)**
5. **OpenCV. (n.d.). *Open Source Computer Vision Library*. [https://opencv.org/](https://opencv.org/" \t "_new)**
6. **MongoDB. (n.d.). *The Developer Data Platform*. [https://www.mongodb.com/](https://www.mongodb.com/" \t "_new)**
7. **Mongoose. (n.d.). *Elegant MongoDB object modeling for Node.js*. [https://mongoosejs.com/](https://mongoosejs.com/" \t "_new)**
8. **Node.js. (n.d.). *JavaScript runtime built on Chrome’s V8 JavaScript engine*. [https://nodejs.org/](https://nodejs.org/" \t "_new)**
9. **Express.js. (n.d.). *Fast, unopinionated, minimalist web framework for Node.js*. [https://expressjs.com/](https://expressjs.com/" \t "_new)**
10. **JWT.io. (n.d.). *Introduction to JSON Web Tokens (JWT)*. [https://jwt.io/](https://jwt.io/" \t "_new)**
11. **Bcrypt. (n.d.). *Library to help hash passwords*. [https://www.npmjs.com/package/bcrypt](https://www.npmjs.com/package/bcrypt" \t "_new)**
12. **Cloudinary. (n.d.). *Media management for developers*. [https://cloudinary.com/](https://cloudinary.com/" \t "_new)**
13. **Nodemailer. (n.d.). *Send emails from Node.js*. [https://nodemailer.com/](https://nodemailer.com/" \t "_new)**
14. **Flutter. (n.d.). *Build apps for any screen*. [https://flutter.dev/](https://flutter.dev/" \t "_new)**
15. **React.js. (n.d.). *A JavaScript library for building user interfaces*. [https://reactjs.org/](https://reactjs.org/" \t "_new)**
16. **json2csv. (n.d.). *Convert JSON to CSV in Node.js*. [https://www.npmjs.com/package/json2csv](https://www.npmjs.com/package/json2csv" \t "_new)**
17. **xlsx. (n.d.). *Excel parser and writer*. [https://www.npmjs.com/package/xlsx](https://www.npmjs.com/package/xlsx" \t "_new)**
18. **node-cron. (n.d.). *Scheduled jobs in Node.js*. [https://www.npmjs.com/package/node-cron](https://www.npmjs.com/package/node-cron" \t "_new)**
19. **Scikit-learn. (n.d.). *Machine Learning in Python*. [https://scikit-learn.org/](https://scikit-learn.org/" \t "_new)**
20. **Pickle – Python Docs. (n.d.). *Serialization module in Python*. [https://docs.python.org/3/library/pickle.html](https://docs.python.org/3/library/pickle.html" \t "_new)**
21. **Alexandria University – Faculty of Computers and Data Science. *Course Management and Academic Policies Handbook*, 2023.**
22. **OWASP Foundation. (n.d.). *Security Best Practices for Web Applications*. [https://owasp.org/](https://owasp.org/" \t "_new)**
23. **ISO/IEC 27001:2013. (2013). *Information technology – Security techniques – Information security management systems – Requirements*.**
24. **GitHub Docs. (n.d.). *Version control with Git and GitHub*. [https://docs.github.com/](https://docs.github.com/" \t "_new)**
25. **Postman. (n.d.). *API platform for building and testing APIs*. [https://www.postman.com/](https://www.postman.com/" \t "_new)**
26. **Vercel. (n.d.). *Frontend deployment and hosting platform*. [https://vercel.com/](https://vercel.com/" \t "_new)**
27. **Face Recognition Using Deep Learning: A Review (2020). *Computer Vision and Image Understanding*, Elsevier.  
    DOI: 10.1016/j.cviu.2020.103364**
28. **Google Developers. (n.d.). *OAuth 2.0 and OpenID Connect for securing APIs*. <https://developers.google.com/identity/protocols/oauth2>**
29. **<https://doi.org/10.1109/LSP.2016.2603342>**
30. **- <https://doi.org/10.1109/CVPR.2015.7298682>**
31. **Figma. (n.d.). Figma – The collaborative interface design tool. Retrieved from: [https://www.figma.com/](https://www.figma.com/" \t "_new)**
32. **Figma Help Center. (n.d.). Design, Prototype, and Collaborate All in One Place. Retrieved from: https://help.figma.com/**
33. **Figma Community. (n.d.). Explore, share, and remix design files. Retrieved from: https://www.figma.com/community**
34. **Figma Blog. (n.d.). Best Practices in UI/UX Design with Figma. Retrieved from: https://www.figma.com/blog/**