PHENIKAA UNIVERSITY

PHENIKAA SCHOOL OF COMPUTING

ACADEMIC MANAGEMENT SYSTEM

Course: Software Architecture

Course Class: CSE703110-1-2-25(N02)

Group 7:

1. Le Thi Kieu Trang	**ID: 23010502**
2. Quach Huu Nam	**ID: 23012358**
3. Trieu Tien Quynh	**ID: 23010648**

Hanoi, January 8, 2026

TASK ASSIGNMENT TABLE

No.	Student ID	Full Name	Assigned Tasks
1	23010502	Le Thi Kieu Trang	Use Case Modeling Conclusion & Reflection
2	23012358	Quach Huu Nam	Key Quality Attributes (Architectural Goals) Testing, Verification & Deployment
3	23010648	Trieu Tien Quynh	Core Functional Requirements Architectural Design & Implementation

MỤC LỤC

PREFACE

In the context of rapid digital transformation, the application of information technology in higher education management is no longer just an option but an inevitable trend. Traditional academic management systems, or legacy Monolithic systems, often face significant challenges regarding scalability, maintenance, and the integration of new features as the volume of students and data continues to grow.

Recognizing these challenges, our team decided to undertake the project titled "Building an Academic Management System based on Microservices Architecture." The project focuses on addressing the management of student information, courses, credit enrollment, grading, and more, by decomposing the system into independent services that communicate flexibly via standard RESTful APIs.

This report details the system analysis, design, and implementation process. It covers the construction of core services such as Identity, Student, and Grade Services using Python (Flask), as well as the design of a highly interactive Single Page Application (SPA) user interface. Notably, the project delves into specific distributed architecture techniques, such as Asynchronous Communication and shared database management.

We hope that the results of this project will serve as a useful reference model for the application of modern software architecture to real-world management problems.

To complete this project, alongside the efforts of our team members, we have received dedicated attention, encouragement, and guidance from M.S.Vu Quang Dung. Throughout the implementation of this topic, he spared no effort in imparting knowledge, guiding our problem-solving mindset, and providing valuable feedback to help us refine the system, ranging from Microservices organization to source code optimization.

Although the team has made every effort to apply learned knowledge to practice, due to time constraints and limited practical experience, the project inevitably contains shortcomings. We sincerely look forward to receiving your feedback to further improve the topic and gain valuable lessons for our future careers.

We would like to express our sincere gratitude!

1. Executive Summary

The project focuses on the design and implementation of an "Academic Management System" aimed at supporting the credit-based training model. The system addresses the management requirements for university, faculty, major, course, and student information; facilitates students in course registration and tracking academic results. Simultaneously, it allows lecturers to enter and manage grades, ensuring transparency and accuracy in the academic assessment process. Throughout the development process, the system has successfully transitioned from a Monolithic architecture to a Microservices Architecture.

The final system comprises:

+ Independent Microservices: Identity, Student, Course, Grade, Enrollment, Email, Faculty, University, and Major Service (including KKT Service).
+ Modern Frontend: Built using Vanilla JavaScript (Single Page Application), enabling smooth interaction for both Students and Lecturers without page reloads.
+ Centralized Database with Logical Separation: The system currently uses a shared MySQL database schema, where each microservice accesses only the tables relevant to its own business domain. This design ensures clear logical separation of data ownership while simplifying deployment and development. The architecture is designed to be transition-ready toward a full Database-per-Service model in future iterations.
+ Hybrid Communication: Combines REST API (Synchronous) and Threading (Asynchronous).

The achieved result is a system with high availability, capable of handling large traffic volumes during peak course registration periods.

2. Project Requirements & Goals

2.1 Core Functional Requirements

The system fulfills the following key functional requirements:

ID	Function	Detailed Description
FR-01	Catalog Management	Administrator can add, edit, or delete Student and Course information (CRUD).
FR-02	Course Registration	Students can log in and self-register for courses opened in the current semester. The system validates course eligibility and records the enrollment.

FR-03	Grading & GPA Calculation	Lecturers enter component scores (Attendance, Mid-term, Final) with customizable weights. The system automatically calculates the overall score (10-point scale) and converts it to letter grades (A, B, C...) and the 4.0 scale score immediately upon entry.
FR-04	Asynchronous Notification	The system automatically sends notification emails to students immediately after the Lecturer completes grading (background processing).
FR-05	Result Lookup	Students can view detailed transcripts of completed courses and their Cumulative GPA.
FR-06	Security & Authorization	The system requires authentication and enforces access authorization (Administrator/Faculty/Student) via the Identity Service.

2.2 Key Quality Attributes (Architectural Goals)

❖ **Performance & Scalability**

Particularly critical for UC4 (Course Registration) due to anticipated traffic spikes.

NFR-01 (Response Time): The system must respond to data read requests (GET) within < 2 seconds (under stable network conditions). For write transactions (such as Grade Submission, Course Registration), processing time must not exceed 3 seconds.

NFR-02 (Concurrency / Load Capacity): The system must support a minimum of 500 - 1,000 Concurrent Users (CCU) during peak registration periods without crashing or service interruption.).

NFR-03 (API Gateway Latency): The API Gateway must not introduce a latency exceeding 50ms when routing requests from the Client to backend Microservices.

NFR-04 (Asynchronous Processing): Time-consuming tasks (such as sending Email notifications, importing large Excel files) must be handled as Background Tasks (Asynchronous) to prevent blocking the user interface.

❖ **Security**

Crucial as the system stores personal information and academic records.

NFR-05 (Authentication & Authorization): All APIs accessing resources (except for

login/public pages) must be protected via JWT (JSON Web Token) mechanisms. The API Gateway must reject requests without a valid token (returning HTTP 401).
NFR-06 (Role-Based Access Control - RBAC): The system must strictly enforce authorization based on roles:

- Student: Can only view their own grades and register for themselves.
- Lecturer: Can only enter grades for classes they teach.
- Administrator: Has full Administrator privileges.

NFR-07 (Data in Transit Security): All communication between the Client and Server (API Gateway) must be encrypted via HTTPS protocol (TLS 1.2 or higher).
NFR-08 (Password Hashing): User passwords in the Identity Service must not be stored in plain text but must be hashed using strong algorithms (e.g., BCrypt or Argon2).

❖ **Reliability & Availability**

Ensures stable system operation in a distributed environment.
NFR-09 (Availability): The system guarantees an Uptime of 99% during business hours.
NFR-10 (Fault Tolerance): The system achieves fault isolation through microservice separation, ensuring that failures in non-critical services (e.g., Email Service) do not interrupt core functionalities. While a full Circuit Breaker pattern is not implemented, services handle failures gracefully by logging errors and allowing the main business flow to continue uninterrupted.
NFR-11 (Data Consistency): The system ensures strong consistency in the Course Registration workflow by performing atomic operations within a shared database environment. Slot availability and enrollment records are updated within controlled transactions to prevent over-registration scenarios. This approach avoids the complexity of distributed transactions while maintaining ACID guarantees.

❖ **Maintainability & Extensibility**

Serving the Development Team.
NFR-12 (Microservices Architecture): Services (Student, Grade, Course, etc.) must be independently deployable. Updating the code of the Grade Service must not require restarting the Student Service.
NFR-13 (API Standard): APIs must adhere to RESTful standards, correctly utilizing HTTP Methods (GET, POST, PUT, DELETE) and Status Codes (200, 201, 400, 404, 500).
NFR-14 (Code Quality): Python source code must comply with PEP 8 standards, with full comments for complex business logic functions (e.g., GPA calculation, registration processing).

NFR-15 (Logging): The system must have a centralized Error Logging mechanism to facilitate Debugging when incidents occur.

- **Usability**

Serving the End-User Experience (Students/Lecturers).

NFR-16 (Responsive Design): The web interface must display correctly on both Desktop and Mobile/Tablet devices, especially for Grade Viewing and Course Registration functions.

NFR-17 (User-Friendly Error Messages): When a system error (Backend Error) occurs, the interface must display an understandable message to the user (e.g., *"System is busy, please try again later"*) instead of showing technical error codes or Stack Traces.

NFR-18 (Minimal Interaction/Efficiency): Lecturers can enter and save grades for an entire class with a single button (Bulk Save), without being forced to save line by line.

2.3 Use Case Modeling

- Use Case Diagram

Figure 1: Use Case Diagram – User Interaction

This diagram presents the primary interactions between external actors and the system, focusing on architecturally significant use cases rather than detailed operational behaviors.

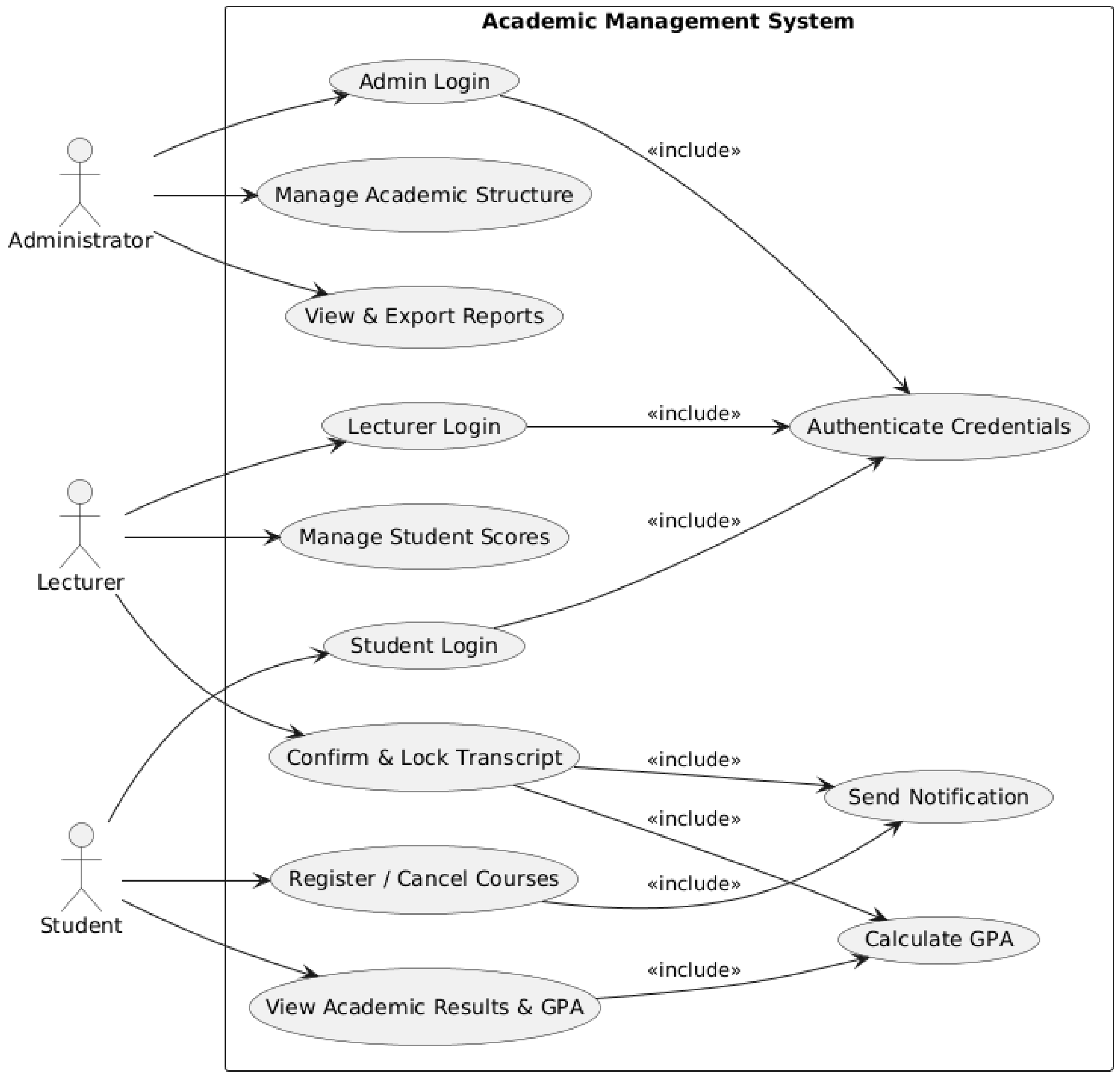
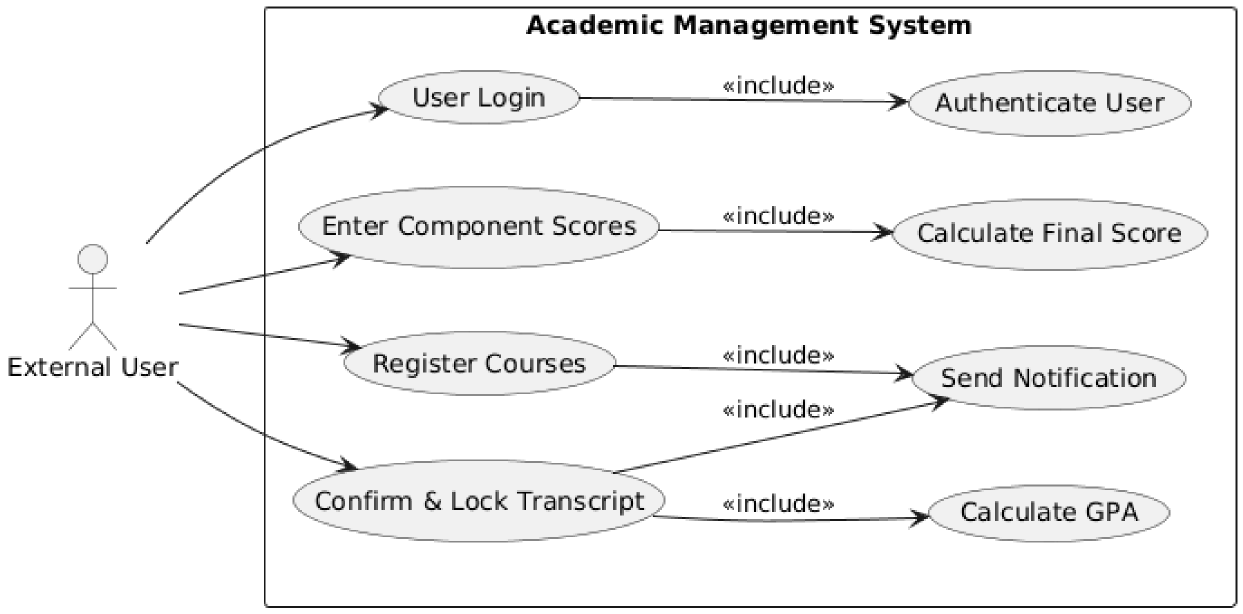


Figure 2: Use Case Diagram – System Functionalities

In the Use Case Diagram – System Functionalities, the system itself is not modeled as an actor. Instead, internal system behaviors such as authentication, GPA calculation, and notification sending are represented as included use cases using the <<include>> relationship, as they are automatically executed as part of user-initiated actions.

The User actor represents a generalized external user, while role-specific interactions are detailed in other diagrams.



❖ Specification and Construction of Administrator Use Case Group

Overview Description

The Administrator is the actor responsible for managing core academic and organizational data within the system. Administrator interactions mainly focus on system access control and high-level data management, including universities, faculties, majors, courses, and student records. In addition, the Administrator is authorized to monitor academic outcomes by viewing and exporting confirmed grade sheets and generating summary reports to support management and assessment activities.

Use Case Specification

+ *UC-A1* Administrator Login

Use Case Name	Administrator Login		
Created By	Development Team	Last Updated By	System Officer
Created Date	Jan 1, 2026	Last Modified Date	Jan 8, 2026
Description	Allows the Administrator to authenticate their identity using a username and password to access the system's administrative functions. This function utilizes the centralized Identity Service.		
Actors	- Administrator (Primary Actor) - Identity Service (Authentication System)		
Pre-conditions	1. The Identity Service is operational. 2. The user has accessed the Login page. 3. The Administrator account exists in the system database.		
Post-conditions	Success: 1. The system returns an Access Token (e.g., fake-jwt-token-for-Administrator). 2. The user is redirected to the Administrator Dashboard. 3. The Administratoristration menu displays full functions according to permissions.		

	Failure: 1. The user remains on the login page. 2. The system displays a corresponding error message.
Main Flow	1. Access Page: - Administrator accesses the admin portal URL; the system displays the Login Form. 2. Enter Credentials: - Administrator enters Username and Password. - Clicks the "Login" button. 3. Validation: - The System (Frontend) performs preliminary validation: Fields must not be empty. 4. Send Request: - Frontend sends an HTTP POST request to the API. - Payload: { "username": "admin", "password": "..." } 5. Authentication (Backend): - Identity Service receives the request. - Checks if the username exists. - Compares the submitted password with the stored hash in the Database. 6. Generate Token: - If credentials are correct, Identity Service generates a success response. - Response: { "message": "Login successful", "token": "...", "role": "admin" } 7. Redirect: - Frontend receives the Token and saves it (LocalStorage/Cookie). - Frontend checks the role: If role == "admin" $\rightarrow$ Redirect to the Dashboard. 8. Use Case Ends.
Alternative Flow	Step 5a. Incorrect credentials: → Identity Service does not find the user or the password matches. → Returns HTTP 401 with JSON: { "error": "Invalid credentials" }. → Frontend displays a red notification: *"Incorrect username or password."* *Step 7a. Non-Administrator privileges:* → User logs in with a valid account but the role is "Student" or "Lecturer". → System detects incorrect role permission. → Notification: *"You do not have permission to access the Administrator page"* and redirects to the appropriate portal.
Exceptions	1. Service Connection Loss: → Identity Service (Port 5004) is down. → Frontend receives a connection error (Connection Refused).

	→ Displays notification: *"System is under maintenance, please try again later."*
Requirements	1. Security: Passwords sent must be encrypted via HTTPS (in Production environment). 2. Performance: Login response time must be < 1 second.

+ *UC-A2* Manage Academic Structure

Use Case Name	Manage Academic Structure		
Created By	Development Team	Created By	Development Team
Created Date	Jan 1, 2026	Created Date	Jan 1, 2026
Description	Allows Administrators to manage the hierarchical organization of the institution. This includes managing Universities (Root), Faculties (Child of University), and Majors (Child of Faculty).		
Actors	- Administrator (Primary Actor) - Uni Service (Manages Universities) - Faculty Service (Manages Faculties) - Major Service (Manages Majors)		
Pre-conditions	1. Administrator has successfully logged in. 2. All structure-related services (uni, faculty, major) are operational.		
Post-conditions	Success: 1. Structure data is created/updated/deleted in the respective Databases. 2. The hierarchical tree is refreshed on the interface. Failure: 1. Data remains unchanged. 2. Error message displayed (e.g., Constraint Violation).		
Main Flow	1. Select Entity Type: - Administrator selects "Academic Structure" menu. - Chooses the entity to manage: University, Faculty, or Major. 2. View List (Read): - System calls the corresponding API based on selection: + University: GET /api/universities + Faculty: GET /api/faculties (requires University ID filter) + Major: GET /api/majors (requires Faculty ID filter) - Displays the data table. 3. Add New Entity (Create): - Admin clicks "Add New". - If University: Enters Code, Name, Address. - If Faculty: Selects Parent University \$\rightarrow\$ Enters Code, Name		

	- If Major: Selects Parent Faculty $\rightarrow$ Enters Code, Name. - Clicks "Save". - System calls POST to the respective Service. 4. Update Entity (Update): - Admin selects a row and modifies information. - Clicks "Update". - System calls PUT API to save changes. 5. Delete Entity (Delete): - Admin clicks "Delete". - System checks for child data (Referential Integrity). - If safe, System calls DELETE API. - System notifies: *"Deleted successfully".* *6. Use case kết thúc.*
Alternative Flow	Step 3a. Duplicate Code: → Administrator enters a code that already exists. → Service returns 409 Conflict. → System notifies: *"Entity Code already exists."* *Step 5a. Data Integrity Violation (Delete Blocked):* → Administrator tries to delete a Faculty that still has Majors assigned. → System blocks the request. → Notification: *"Cannot delete this Faculty because it contains Majors. Please remove child data first."*
Exceptions	1. Service Unavailable: → One of the microservices is down. → System displays: *"Unable to load structure data. Please try again later."*
Requirements	1. Hierarchy Rule: A Major must belong to a Faculty; a Faculty must belong to a University. 2. Uniqueness: Codes must be unique within their scope (e.g., Major Codes must be unique within a Faculty).

+ *UC-A3* View & Export Reports

Use Case Name	View & Export Reports		
Created By	Development Team	Created By	Development Team
Created Date	Jan 1, 2026	Created Date	Jan 1, 2026
Description	Allows Administrators to view academic statistics (e.g., Enrollment status, Grade distribution, Student list per major) and export these reports to external files (Excel) for administrative purposes.		

Actors	- Administrator (Primary Actor) - Grade Service (Data Source) - Enrollment Service (Data Source) - Student Service (Data Source)
Pre-conditions	1. Administrator has successfully logged into the system. 2. Data exists in the system (Students, Grades, Enrollments). 3. Relevant services are operational.
Post-conditions	Success: 1. Statistical data is displayed visually (Table/Chart). 2. The report file is successfully downloaded to the Administrator's device. Failure: 1. Error notification is displayed. 2. File download fails.
Main Flow	1. Access Report Dashboard: - Administrator selects the "Reports & Statistics" menu. - System displays a list of available report types (e.g., "Semester GPA Summary", "Enrollment Statistics", "Student List"). 2. Configure Parameters: - Administrator selects a Report Type. - Sets filters: Semester (e.g., Fall 2025), Faculty (e.g., IT), Major. - Clicks "Generate Report". 3. Data Retrieval: - System calls the relevant API (e.g., GET /api/grades/reports/gpa or GET /api/enrollment/stats). - The Service aggregates data and returns JSON. 4. View Report: - System renders the data into a Table or Chart on the screen. 5. Export Data: - Administrator clicks "Export to Excel" (or PDF). - System calls the Export API (e.g., GET /api/grades/reports/export?format=xlsx). - The Backend generates the file and streams it back. - The Browser initiates the file download. 6. Use Case Ends.
Alternative Flow	Step 3a. No Data Found: → The filter criteria yield no results (e.g., A new semester with no grades yet). → System notifies: *"No data available for the selected criteria."* → Export button is disabled.

	Step 5a. Large Dataset (Async Export): → The report contains thousands of records (taking > 30s to generate) → System notifies: *"Report is being generated. You will be notified when it is ready."* → The task is pushed to a Background Queue.
Exceptions	1. Service Timeout: → The aggregation query takes too long, causing a Gateway Timeout (504). → System suggests: *"Data is too large. Please narrow down the date range or use Export feature.* *2. File Generation Error:* → Error occurs during file creation (e.g., Library failure). → Notification: *"Failed to generate file. Please contact IT support."*
Requirements	1. Formats: Must support .xlsx (Excel) for data manipulation. 2. Accuracy: Statistics must reflect real-time data (or cached data not older than 24 hours, depending on configuration).

❖ Specification and Construction of Lecturer Use Case Group

Overview Description

The Lecturer Module is a core component responsible for assessing and recording student academic results. The primary actor of this module is the Lecturer, who directly teaches and is responsible for the accuracy of grades.

The main objective of this module is to provide a closed-loop process ranging from receiving assigned classes, entering grades (manually or via Excel import), automatically calculating final grades, to locking the grade sheet for official archiving.

Use Case Specification

+ UC-L1 Lecturer Login

Use Case Name	Lecturer Login		
Created By	Development Team	Created By	Development Team
Created Date	Jan 1, 2026	Created Date	Jan 1, 2026
Description	Allows Lecturers to authenticate their account information to access the teaching management and grading module.		
Actors	- Lecturer		
Pre-conditions	1. The Lecturer has been granted an account (Username/Password) in the system. 2. The Identity Service is operational.		
Post-conditions	Success: 1. The system issues an Authentication Token. 2. The Lecturer is redirected to the Lecturer Homepage (Lecturer Dashboard).		

	Failure: 1. The user remains on the login page. 2. An error message is displayed.
Main Flow	1. Access: - Lecturer accesses the general system Login page. 2. Input: - Enters Username (or Official Email) and Password. - Clicks the "Login" button. 3. Authentication Processing: - The system calls API POST /api/login to the Identity Service. - The Service checks login credentials against the Database. 4. Authorization: - After the password is verified, the system checks the account's Role. - Confirms the Role is "Lecturer". - The system returns the Token and navigation information. 5. Redirect: - The interface redirects to the Class List screen (Instead of the Administrator screen). 6. Use Case Ends.
Alternative Flow	Step 3a. Invalid Information: → Wrong Username or Password entered. → System error: *"Incorrect username or password."* Step 4a. Unauthorized Role: → Account credentials are correct but the Role is "Student" or "Administrator" (in cases where specific access is restricted). → System notifies: *"This account does not have permission to access Lecturer functions."*
Exceptions	1. Identity Service timeout: → System displays: *"Cannot connect to authentication server."* 2. Account Locked: → The Lecturer has resigned or the account is locked. → System notifies: *"Your account has been disabled. Please contact the Administrator."*
Requirements	1. Passwords must be encrypted during transmission. 2. Response time < 3 seconds. 3. The login interface should be user-friendly and support a "Forgot Password" feature (Optional).

+ UC-L2 Manage Student Scores

Use Case Name	Manage Student Scores		
Created By	Development Team	Created By	Development Team
Created Date	Jan 1, 2026	Created Date	Jan 1, 2026
Description	Allows Lecturers to view the list of course classes assigned to them by the university for the current semester or previous semesters.		
Actors	- Lecturer (Primary Actor) - Grade Service (System) - Course Service (Verifies assignment)		
Pre-conditions	1. Lecturer has logged in successfully. 2. The Lecturer is officially assigned to the class. 3. The Grade Sheet is NOT yet Locked (Status is OPEN or DRAFT).		
Post-conditions	Success: 1. Grades are saved to the Database. 2. The total_score (Course Grade) is automatically re-calculated based on weights. Failure: 1. Data remains unchanged. 2. Error message displayed (e.g., Invalid score range).		
Main Flow	1. Select Class: - Lecturer navigates to "My Courses" and selects a specific class. - System displays the list of students enrolled in that class. 2. Input Scores: - Lecturer enters values into the columns: Attendance (CC), Mid-term (GK), Final (CK). - *Note:* Lecturer can enter scores for multiple students at once. 3. Validation (Frontend): - As the Lecturer types, the interface checks if the value is numeric and within the range [0, 10]. 4. Save: - Lecturer clicks "Save Grades". - System calls PUT /api/grades/batch-update to Grade Service. 5. Processing: - Grade Service validates the Grade Sheet status (must be Unlocked). - Updates records in the Database. - Triggers a background calculation for the Final Total Score (UC-SYS3). 6. Feedback: - System notifies: *"Grades saved successfully."*		

	- The interface refreshes with the updated data. 7. Use Case Ends.
Alternative Flow	Step 3a. Invalid Input: → Lecturer enters a score like "11" or "-5". → System highlights the cell in red and disables the "Save" button. → Message: *"Score must be between 0 and 10."* Step 5a. Concurrent Edit Conflict: → Two lecturers (e.g., Main and Assistant) try to save scores for the same student at the same time. → System detects version conflict. → Notification: *"Data has been modified by another user. Please refresh and try again."*
Exceptions	1. Locked Grade Sheet: → Lecturer attempts to save, but the Grade Sheet was locked (by Admin or previously by Lecturer). → Grade Service returns 403 Forbidden. → System notifies: *"This grade sheet is locked and cannot be edited."*
Requirements	1. Range: Scores must be strictly between 0.0 and 10.0. 2. Log: Every score change must be logged (Who changed, Old Value, New Value) for audit purposes. 3. Batch Processing: The API should support batch updates (saving the whole class at once) to reduce network latency.

+ UC-L3 Confirm & Lock Transcript

Use Case Name	Confirm & Lock Transcript		
Created By	Development Team	Created By	Development Team
Created Date	Jan 1, 2026	Created Date	Jan 1, 2026
Description	Allows Lecturers to finalize the grading process for a specific class. Once locked, the grades become official, the Lecturer can no longer edit them, and the system automatically triggers GPA calculation for the students.		
Actors	- Lecturer (Primary Actor) - Grade Service (System) - Notification Service (System)		
Pre-conditions	1. Lecturer has logged in and selected the class. 2. Component scores have been entered. 3. The Grade Sheet status is currently OPEN or DRAFT.		
Post-	Success:		

conditions	1. Grade Sheet status changes to LOCKED. 2. Editing is disabled for the Lecturer. 3. System triggers UC-SYS4 (Calculate GPA). 4. Students receive notifications (via UC-SYS5). Failure: 1. Status remains OPEN. 2. Error message displayed.
Main Flow	1. Review Grades: - Lecturer reviews the calculated Total Scores and ensures all data is correct. 2. Initiate Lock: - Lecturer clicks the "Finalize & Lock" button. 3. System Confirmation: - System displays a warning modal: *"Are you sure you want to lock this grade sheet? This action cannot be undone. Grades will be published to students immediately."* 4. Confirmation: - Lecturer selects "Confirm". - System calls API POST /api/grades/lock/{class_id}. 5. Validation & Processing: - Grade Service checks if all required grades are filled (optional, depending on policy). - Updates status to LOCKED in the Database. - Triggers an asynchronous event: GRADE_SHEET_LOCKED. 6. Feedback: - System notifies: *"Grade sheet locked successfully."* - The input fields become read-only. 7. Use Case Ends.
Alternative Flow	Step 5a. Missing Grades (Validation Error): → System detects that some students have missing component scores (Null) without a specific exemption. → System denies the Lock action. → Notification: *"Cannot lock grade sheet. Please enter grades for all students or mark them as Exempt."* Step 5b. Already Locked: → The grade sheet was already locked by an Admin. → System refreshes the page to Read-only mode.
Exceptions	1. Trigger Failure: → The Lock is successful, but the trigger for GPA Calculation (UC-SYS4) fails due

	to Message Queue error. → Interface still shows "Locked" to the Lecturer.
Requirements	1. Irreversibility: Once locked, the Lecturer cannot unlock it themselves. Unlocking requires an Administrator (UC-A-Unlock). 2. Audit Log: Must record exactly *when* and *who* locked the transcript to resolve disputes

❖ Specification and Construction of Student Use Case Group

Overview Description

The Student Module is the primary information portal for learners, serving as the interface where students interact with the university to perform credit-based training processes.

Unlike the Lecturer module (which focuses on data entry), the Student module's primary activities are Information Lookup (Viewing Grades, Viewing Schedules) and executing Registration Transactions (Registering/Canceling courses). The goal of this module is to provide transparent, accurate information and ensure fairness in credit registration.

Use Case Specification

+ UC-S1 Student Login

Use Case Name	Student Login		
Created By	Development Team	Created By	Development Team
Created Date	Jan 1, 2026	Created Date	Jan 1, 2026
Description	Allows Students to authenticate their account credentials (Student ID/Password) to access learner-specific functions such as: Viewing Grades, Course Registration, and Viewing Class Schedules.		
Actors	- Student - Identity Service		
Pre-conditions	1. The Student has an existing account in the system (usually granted upon enrollment). 2. The Identity Service is operational.		
Post-conditions	Success: 1. Student receives an Authentication Token. 2. System redirects to the Student Dashboard. Failure: 1. User remains on the login page. 2. Error message is displayed.		
Main Flow	1. Access: - Student accesses the Academic Management Website.		

	2. Input Information: - Enters Username (Usually Student ID, e.g., SV001) and Password. - Clicks the "Login" button. 3. Authentication: - System calls API POST /api/login to Identity Service. - Service verifies credentials against the Database. 4. Role Check: - System confirms the account is valid. - System verifies the account Role is "Student". - Returns Token and User info. 5. Redirect: - Interface redirects the student to the "Personal Info & Academic Results" screen (or Main Dashboard). 6. Use Case Ends.
Alternative Flow	Step 3a. Invalid Credentials: → Wrong Student ID or Password entered. → System notifies: *"Incorrect username or password."* Step 4a. Account Locked/Reserved: → Student is currently under disciplinary action or academic suspension. → System notifies: *"Account is temporarily locked. Please contact the Student Affairs Office."*
Exceptions	1. Connection Error: → Cannot connect to the server. → Notification: *"System error, please try again later."*
Requirements	1. Login interface must be simple and clear. 2. Must have a "Forgot Password" feature (reset password via student email). 3. Information security (Password hashing).

+ UC-S2 Register / Cancel Courses

Use Case Name	Register / Cancel Courses		
Created By	Development Team	Created By	Development Team
Created Date	Jan 1, 2026	Created Date	Jan 1, 2026
Description	Allows Students to view the list of open classes for the current semester, register for new courses, or cancel existing registrations within the allowed timeframe.		
Actors	- Student (Primary Actor) - Enrollment Service (Main Handler) - Course Service (Check Slots)		

	- Notification Service (Confirmations)
Pre-conditions	1. Student has logged in successfully. 2. The Course Registration Period is currently active (Open). 3. Student has paid tuition fees (if required by policy).
Post-conditions	Success: 1. Register: New record added to enrollments, Class current_slots increases by 1. 2. Cancel: Record removed from enrollments, Class current_slots decreases by 1. 3. Student receives a confirmation email. Failure: 1. Registration rejected (Class full, Schedule conflict). 2. Error message displayed.
Main Flow	1. View Open Classes: - Student selects "Course Registration". - System displays list of available classes for the semester (showing Subject Name, Lecturer, Schedule, Current/Max Slots). 2. Select Class: - Student selects a class to register - Clicks "Register". 3. Validation (Enrollment Service): - Check Slots: Is current_slots < max_slots? - Check Schedule: Does the class time overlap with already registered courses? - Check Prerequisite: Has the student passed the required prerequisite subjects? - Check Credit Limit: Does total credits exceed the max allowed (e.g., 24 credits)? 4. Process Registration: - If all checks pass, System creates a "Pending" transaction. - Atomic Update: System increments the slot count in Course Service. - Saves registration record in DB. 5. Feedback: - System notifies: *"Registration Successful: [Subject Name]"*. - The class appears in the "Registered Courses" table. - Notification Service sends a confirmation email. 6. Use Case Ends.
Alternative Flow	Step 3a. Class Full: → Stud System detects current_slots >= max_slots. → Notification: *"Class is full. Please choose another class."* Step 3b. Schedule Conflict: → The selected class overlaps with an existing one.

	→ Notification: *"Schedule conflict with [Existing Subject]."* Step 1a. Cancel Registration: → (Refer to UC-S5 details if separated, or here as a sub-flow) → Student clicks "Cancel" on a registered course. → System validates deadline → Removes record → Decrements slot count.
Exceptions	1. Race Condition (Concurrency): → Student A and B click "Register" at the exact same moment for the last slot. → Database applies Row Locking or Optimistic Locking. → One student succeeds, the other receives: *"Registration failed. The class just became full."* 2. System Overload: → Too many requests (> 10,000 req/s). → System puts the request in a Queue or returns 503 Service Unavailable.
Requirements	1. Data Integrity: Slot counting must be strictly accurate (ACID properties) to prevent over-subscription. 2. Performance: The "Check Slot" query must be highly optimized. 3. User Experience: The interface should update slot numbers in near real-time (via WebSocket if possible).

+ UC-S3 View Academic Results & GPA

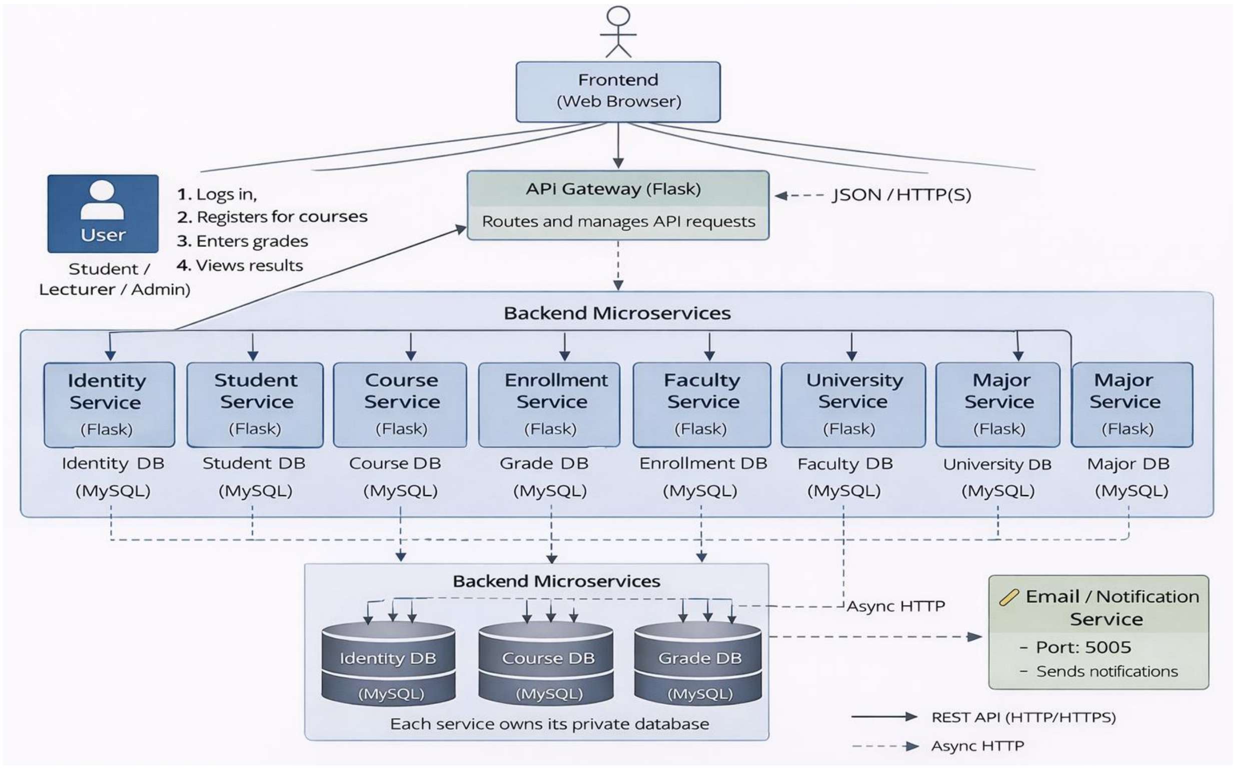
Use Case Name	View Academic Results & GPA		
Created By	Development Team	Created By	Development Team
Created Date	Jan 1, 2026	Created Date	Jan 1, 2026
Description	Allows Students to view their complete academic history, including detailed grades for each course (Component scores, Final scores) and system-calculated performance metrics (Semester GPA, Cumulative CPA, Total Credits).		
Actors	- Student (Primary Actor) - Grade Service (System)		
Pre-conditions	1. Student has successfully logged in. 2. Academic data (Grades) exists in the system. 3. Grade Service is operational.		
Post-conditions	Success: 1. Academic transcript is displayed correctly. 2. GPA/CPA indicators match the latest calculation. Failure: 1. "No data available" message is displayed. 2. Error notification (if service fails).		

Main Flow	1. Access Academic Profile: - Student selects "Academic Results" from the dashboard. 2. Load Data: - System calls API GET /api/student/academic-profile to Grade Service. - Grade Service aggregates data: + *Summary:* Total Credits, GPA (Semester), CPA (Cumulative), Academic Status (Normal/Warning). + *Details:* List of courses grouped by Semester. 3. Display Overview (Dashboard): - System displays Key Performance Indicators (KPIs): + CPA: e.g., 3.24/4.0 + Total Credits: e.g., 85/150 + Status: "Good" 4. Display Detailed Transcript: - System displays a table of subjects. - Columns: Subject Name, Credits, Component Scores (Att/Mid/Final), Letter Grade (A, B, C...). 5. Filter (Optional): - Student filters by "Semester 1 - 2025". - System updates the view to show only that semester's grades and GPA. 6. Use Case Ends.
Alternative Flow	Step 2a. Financial Hold (Tuition Debt): → System checks with Tuition Service (if integrated) and finds unpaid fees. → System blocks the view of final grades. → Notification: *"Please complete tuition payment to view final results."* (Only Component scores might be visible). Step 2b. No Data (Freshman): → Student has not completed any courses yet. → System displays: *"No academic records found."* with GPA = 0.0.
Exceptions	1. Calculation Error: → Data corruption causes a division by zero in GPA calculation. → System handles the exception gracefully, displaying "N/A" instead of crashing. 2. Display Error: → Subject names are missing due to sync issue with Course Service. → System displays "Unknown Subject [ID]" temporarily.
Requirements	1. Privacy: Students can strictly ONLY view their own grades. 2. Visualization: It is recommended to include a Line Chart showing GPA trends over

	semesters for better user experience. 3. Accuracy: The displayed GPA must be the result of the latest UC-SYS4 calculation.

3. Architectural Design & Implementation

3.1 Overall System Architecture



3.2 Architectural Pattern: Microservices Architecture

The Academic Management System is designed following the Microservices Architecture pattern. Instead of building a single monolithic application, the system is decomposed into a set of small, domain-oriented services, each responsible for a specific business capability such as authentication, student management, course management, enrollment processing, grading, and notification.

Each microservice:

- Runs as an independent Flask application on a dedicated port.
- Encapsulates its own business logic and exposes a well-defined RESTful API.
- Can be developed, tested, and deployed independently at the code level.

This architectural approach is well suited for the Academic Management domain due to the clear separation of responsibilities among functional areas (e.g., enrollment, grading, identity), which reduces coupling and improves long-term maintainability.

The key benefits of applying Microservices Architecture in this system include:

Scalability (Design-Level):

Services with high load potential, such as Enrollment Service or Grade Service, are designed to be independently scalable in future deployments when supported by containerization and orchestration technologies.

Maintainability:

Clear service boundaries allow teams to modify or extend one functional area without impacting others, simplifying maintenance and future enhancements.

Fault Isolation (Partial):

Failures in non-critical services (e.g., Email Service) do not block core academic functionalities. However, shared infrastructure components such as the database and identity service remain critical dependencies.

In this system, supporting domain services such as Faculty Service, University Service, Major Service, and KKT Service are responsible for managing organizational structure and curriculum-related data, further demonstrating the domain-driven decomposition applied in the overall architecture.

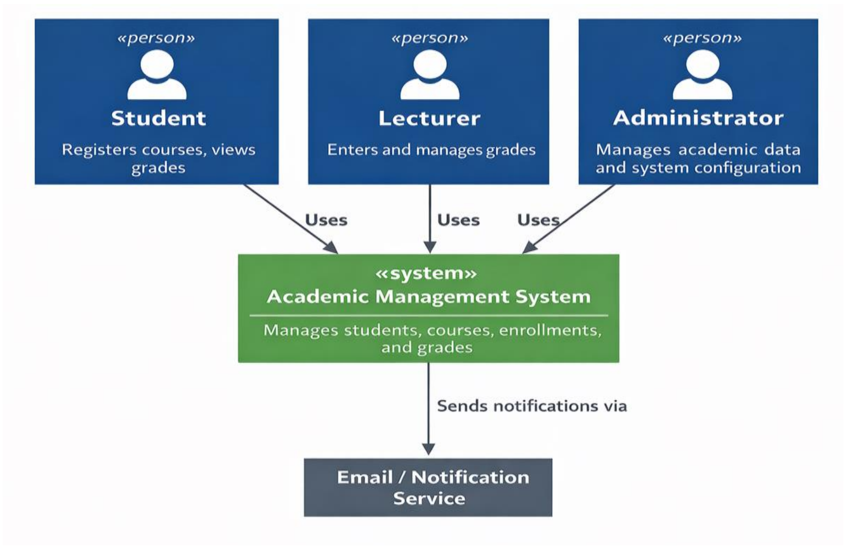

«person»
Student
Registers courses, views grades
«person»
Lecturer
Enters and manages grades
«person»
Administrator
Manages academic data and system configuration
Uses
Uses
Uses
«system»
Academic Management System
Manages students, courses, enrollments, and grades
Sends notifications via
Email / Notification Service

Figure 3.2. C4 – System Context Diagram for Academic Management System

3.3 Technical Stack and Data Model

❖ Technical Stack

The Academic Management System is implemented using the following technologies:

Frontend:

+ Vanilla JavaScript (ES6+) for client-side logic and dynamic DOM manipulation, providing a lightweight Single Page Application (SPA) experience.

+ HTML5 and CSS3 for structure and presentation.

+ Fetch API for asynchronous RESTful communication with backend services.

Backend:

+ Python Flask is used to implement all microservices.

+ Each service exposes RESTful APIs using JSON over HTTP and runs independently on a dedicated port.

Database:

+ MySQL is used as the relational database management system.

+ The system currently employs a centralized database schema (sa), which is physically shared across services.

Communication:

+ Synchronous HTTP communication is used between the Frontend and backend Microservices.

+ Certain non-critical operations, such as email notifications, are handled asynchronously using background threads. In this case, the Grade Service initiates a non-blocking HTTP request to the Email Service to avoid delaying user-facing operations.

❖ Data Model

All microservices interact with a centralized MySQL database schema (sa). Although the database is physically shared, data ownership is logically separated by service boundaries. Each microservice accesses and manipulates only the tables related to its own business domain.

The core domain entities include:

User *(Identity Service)*: (id, username, password, role)

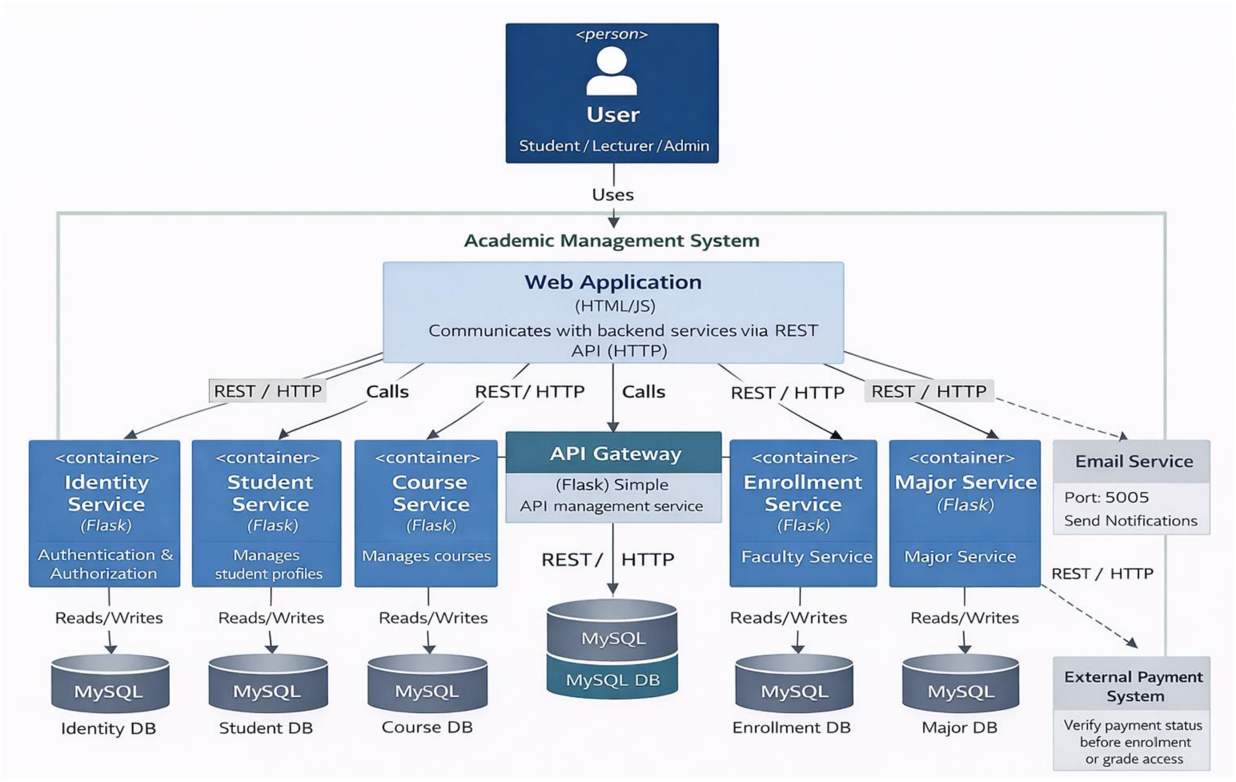
Student *(Student Service)*: (student_id, name, date_of_birth, class_name, email)

Course *(Course Service)*: (course_id, course_name, credits)

Enrollment *(Enrollment Service)*: (enrollment_id, student_id, course_id, semester, status)

Grade *(Grade Service)*: (grade_id, enrollment_id, attendance_score, midterm_score, final_score, total_score, letter_grade)

This design enforces clear logical data ownership while simplifying development and deployment. The architecture is intentionally designed to be evolution-ready, allowing a future transition toward a full Database-per-Service model when system scale and operational requirements increase.



<person>
User
Student / Lecturer / Admin
Uses
Academic Management System
Web Application
(HTML/JS)
Communicates with backend services viia REST API (HTTP)
REST / HTTP
Calls
REST / HTTP
Calls
REST / HTTP
REST / HTTP
<container> Identity Service (Flask)
Authentication & Authorization
<container> Student Service (Flask)
Manages student profiles
<container> Course Service (Flask)
Manages courses
API Gateway
(Flask) Simple API management service
<container> Enrollment Service (Flask)
Faculty Service
<container> Major Service (Flask)
Major Service
Email Service
Port: 5005
Send Notifications
REST / HTTP
REST / HTTP
Reads/Writes
MySQL
Identity DB
Student DB
Course DB
MySQL DB
Enrollment DB
Major DB
External Payment System
Verify payment status before enrolment or grade access

Figure 3.2 presents the C4 Level 2 – Container Diagram, showing the Web Application, the set of backend microservices, and the shared MySQL database.

3.4 Implementation: Server-Side Logic (Inter-service Communication)

On the server side, each microservice exposes a set of RESTful endpoints implemented using Flask.
Each service follows a consistent internal structure to improve maintainability and readability:

- Controller (app.py):
  Handles HTTP requests, configures CORS policies, and defines REST API routes.
- Repository (repository.py):
  Encapsulates data access logic and executes raw SQL queries against the MySQL database.
- Model (models.py):
  Defines domain data structures and provides object-to-dictionary mappings for JSON serialization.

Communication Model

Most interactions between the client application and backend services are implemented as synchronous HTTP requests, following standard RESTful communication patterns.
In addition to client-service communication, the system includes a limited service-to-service interaction for handling non-critical tasks asynchronously:

- When a lecturer updates or confirms grades, the Grade Service initiates a background task.
- A background worker thread is spawned using Python's threading module.
- This worker sends a non-blocking HTTP POST request to the Email Service to trigger notification emails to students.

This fire-and-forget asynchronous processing mechanism ensures that time-consuming operations such as email sending do not block the main grading workflow. While this approach improves responsiveness, it does not rely on a message broker and therefore provides best-effort delivery. More robust asynchronous communication mechanisms can be introduced in future system enhancements.

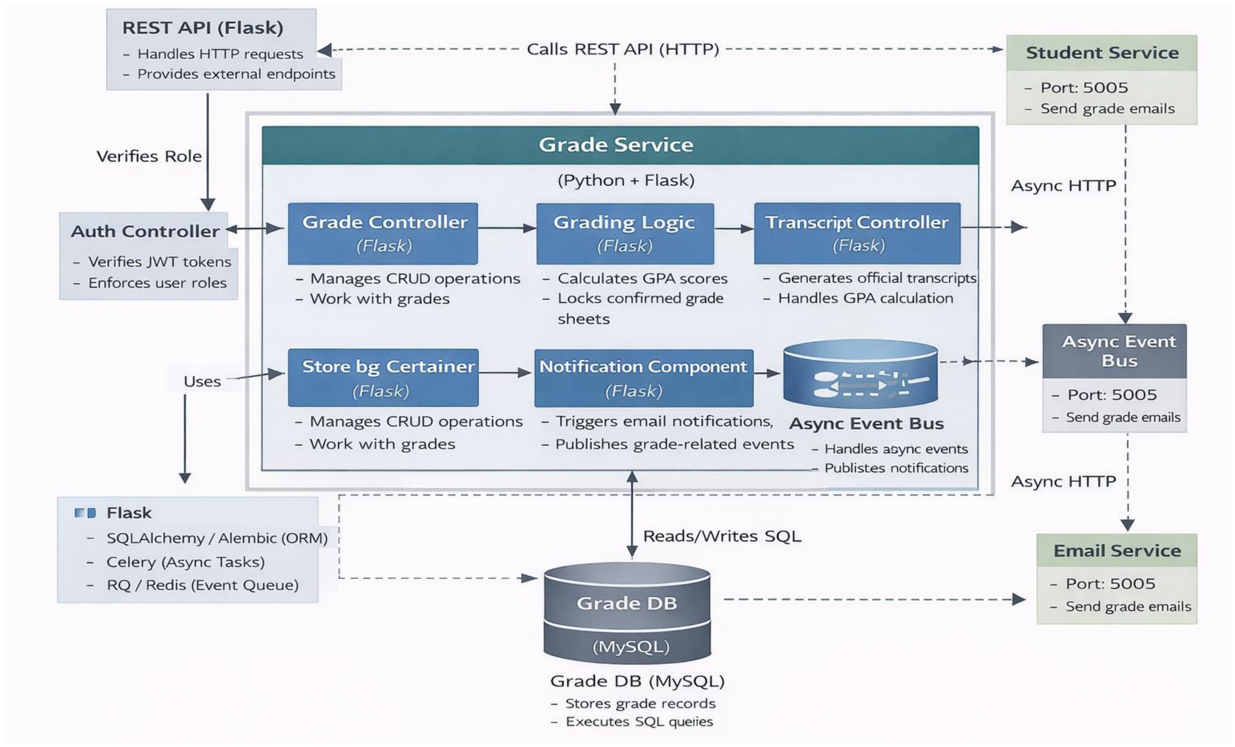


Figure 3.3 shows the C4 Level 3 – Component Diagram of the Grade Service, illustrating the Controller, Repository, Model, and Async Worker components and their interactions.

Email Notification Mechanism

The system includes an Email Service responsible for sending notification emails to students. When a lecturer confirms and locks a grade sheet, the Grade Service triggers an asynchronous HTTP request to the Email Service using a background thread. The Email Service then sends an email to the corresponding student to inform them that the grades have been officially published. This asynchronous mechanism ensures that email delivery does not block the main grading workflow.

3.5 Implementation: Client-Side Logic (Vanilla JS)

The client side is implemented as a Single Page Application (SPA) using pure JavaScript (Vanilla JS). The frontend communicates directly with each microservice through REST APIs.

Main Responsibilities of the Frontend

- User authentication: managing login tokens and redirection.

- Dynamic UI rendering: Using DOM manipulation to switch views (Dashboard, Student List, Grade Input) without reloading the page.

- Sending HTTP requests to appropriate services:

+ Identity Service (Port 5004) for authentication.

+ Student Service (Port 5001) for profile management.

+ Course Service (Port 5002) and Enrollment Service (Port 5006) for registration logic.

+ Grade Service (Port 5003) for viewing and updating grades.

Communication Flow

- The user interacts with the HTML interface (e.g., clicks "Save Grade").

- JavaScript captures the event and sends an HTTP request (via fetch) to the corresponding microservice endpoint.
- The microservice processes the request and accesses the database.
- The response is returned in JSON format, and JavaScript updates the DOM to reflect changes immediately.

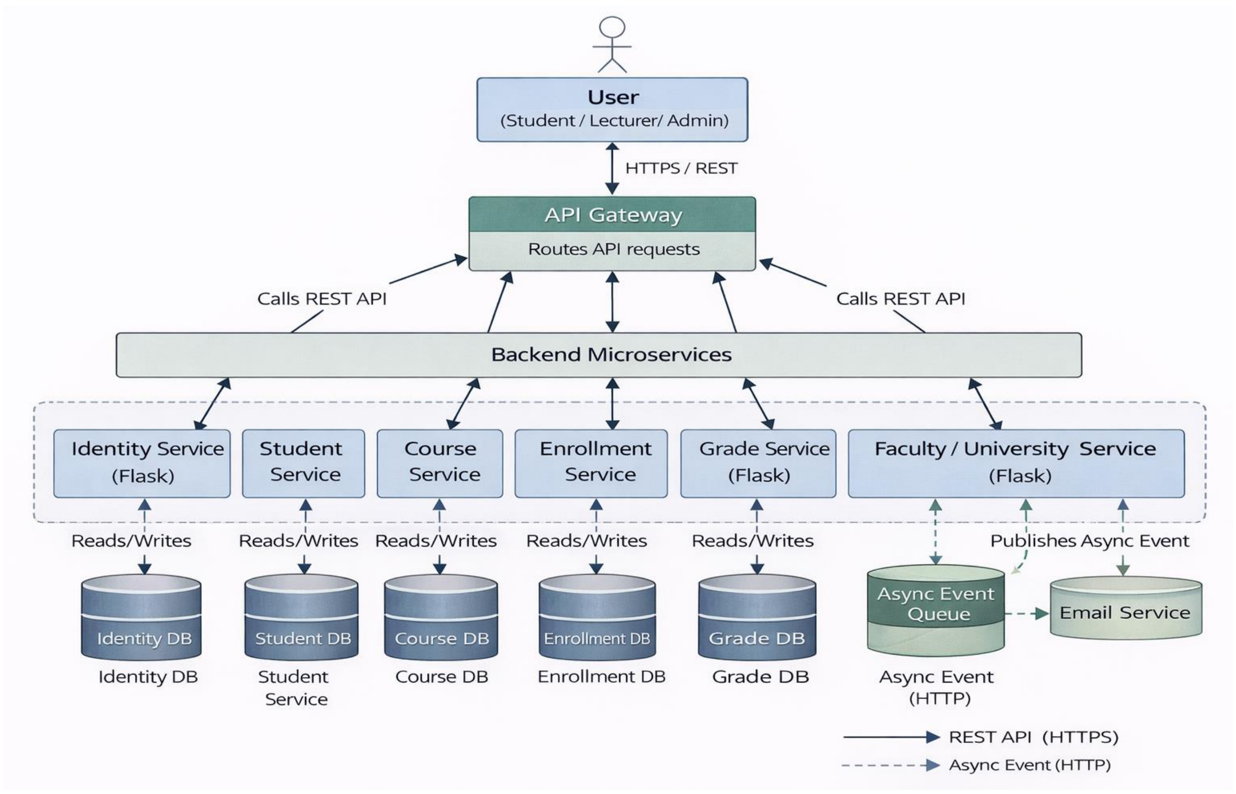


Figure 3.4 shows the High-Level Communication Diagram of the system, highlighting interactions between the Frontend, backend microservices, databases, and the Email Service.

Summary: This architectural design combines Microservices Architecture with a lightweight JavaScript frontend and a centralized MySQL database. The design ensures modularity, scalability, and clear separation of concerns between presentation, business logic, and data layers.

4. Testing & Verification (Kiểm thử & Xác minh)

4.1 Testing Strategy (Unit/Integration Test)

Due to the specific nature of the distributed Microservices architecture, the team applied a multi-layer testing strategy to ensure each service functions correctly before system-wide integration.

4.1.1 Unit Testing (Kiểm thử Đơn vị)

This focuses on testing the smallest components within each Microservice, ensuring internal data processing logic is accurate.

Model Testing: Tests Entity classes (such as Student, Course, Grade) in models.py to ensure data is correctly initialized and converted to JSON (to_dict).

Repository Testing: Tests data access methods in repository.py. This ensures SQL statements (INSERT, SELECT, UPDATE) function correctly with the MySQL database and handle Exceptions (e.g., connection failures) properly.

4.1.2 Integration Testing

Tests the interaction between components within a Service and between Services via API.

API Testing: Uses tools like Postman and cURL to send HTTP Requests (GET, POST, PUT, DELETE) to each Endpoint.

- *Objective:* Verify HTTP Status Codes (200, 201, 400, 404, 500) and the returned JSON structure.
- *Example:* Sending a POST /api/grades request missing required data to check if the server correctly returns a 400 Bad Request error.

Inter-service Communication: Tests the asynchronous communication flow between the Grade Service and Email Service.

- *Scenario:* When a Lecturer finishes entering grades, verify if the Email Service receives the signal and sends the email successfully (by observing the logs of both services).

4.2 Deployment Configuration (Ports & Env)

The system is designed for easy deployment on a local environment (Localhost) with clear port configurations and environment variables.

4.2.1 Environment Variables

To ensure security and facilitate configuration changes without modifying the code, the system uses a .env file to manage Database connection information. All repository.py files utilize the python-dotenv library to read these variables.

File cấu hình (.env):

```
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=your_password
DB_NAME=sa
```

4.2.2 Port Mapping

Each Microservice is configured to run on a distinct port to avoid conflicts. Below is the port configuration table extracted from the app.py files:

Service Name	Port	Main Function	Configuration File
Identity Service	5004	Authentication & Authorization	identity_service/app.py
Student Service	5001	Student Profile Management	student_service/app.py
Course Service	5002	Course Module Management	course_service/app.py
Grade Service	5003	Grade & GPA Management	grade_service/app.py
Email Service	5005	Email Notifications	email_service/app.py
Enrollment Service	5006	Credit Registration	enrollment_service/app.py
Faculty Service	5007	Faculty Management	faculty_service/app.py
University Service	5008	University Management	uni_service/app.py

Major Service	5009	Major Management	major_service/app.py
KKT Service	5010	Knowledge Block Management	kkt_service/app.py

4.3 End-to-End Test Scenarios

This section describes an End-to-End test scenario simulating a real-world business process from the Frontend interface (Vanilla JS) down to the Database.

Kịch bản: Quy trình Đăng ký học và Nhập điểm

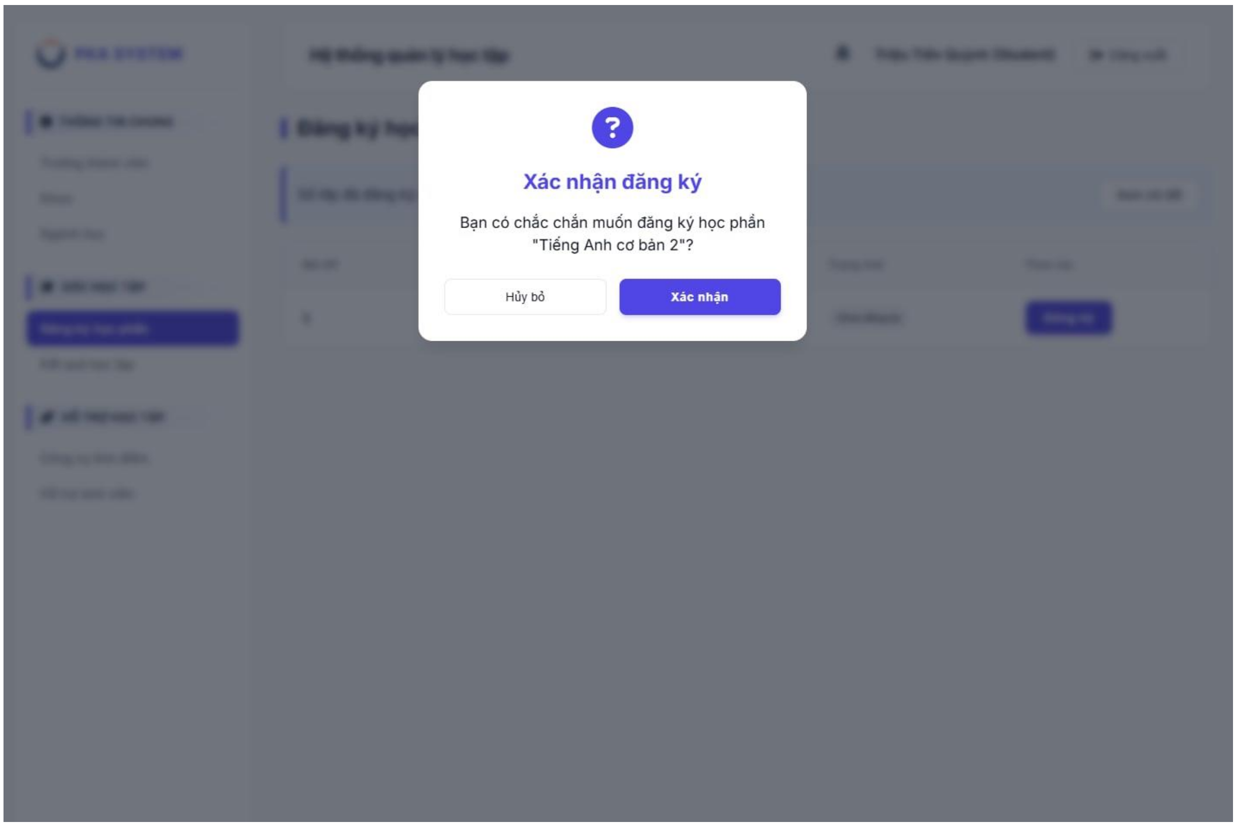
Scenario: Course Registration and Grade Entry Process Objective: Verify data flow across multiple Services: Identity → Course → Enrollment → Grade → Email.

Step	Actor	System Action	Expected Result
1	Student	Logs into the system with a student account.	Receives an Authentication Token; redirected to the Student Dashboard.
2	Student	Accesses "Course Registration" and selects the course "Software Architecture".	System calls Course Service to retrieve info, calls Enrollment Service to save registration. Returns notification "Registration Successful".
3	Lecturer	Logs in and accesses "Enter Grades" for the "Software Architecture" class.	The student list (registered in Step 2) is fully displayed on the grade sheet.
4	Lecturer	Enters process scores, exam scores, and clicks "Save & Lock".	Grade Service saves scores to MySQL, calculates GPA, and triggers the email sending flow (Async).
5	System	(Automated) Grade Service calls Email Service in the background.	Email Service console log displays: *"Email sent to [Student Email]: Your grade has been updated."*
6	Student	Checks mailbox and returns to "View Grades" page.	Receives email notification and sees updated scores on the web interface.

Actual Results:

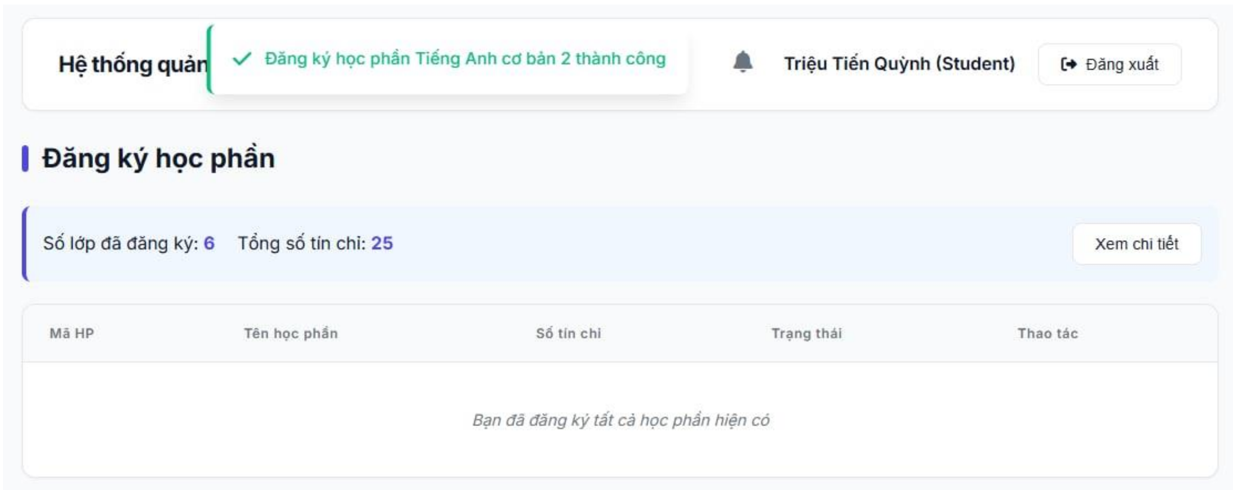
The test scenario was successfully executed in the Localhost environment. Login, course registration,

and grade entry/locking functions operated according to design.
Regarding the email notification function, the system successfully triggered the asynchronous email flow from the Grade Service to the Email Service, and success was recorded via Email Service logs. Actual email delivery to student inboxes was not implemented within the scope of this project; therefore, results are confirmed at the system-level rather than the end-user level.


Xác nhận đăng ký

Bạn có chắc chắn muốn đăng ký học phần "Tiếng Anh cơ bản 2"?

Figure 4.3.1 Course Registration Interface (Student View)


Hệ thống quản

✓ Đăng ký học phần Tiếng Anh cơ bản 2 thành công

Triệu Tiến Quỳnh (Student)

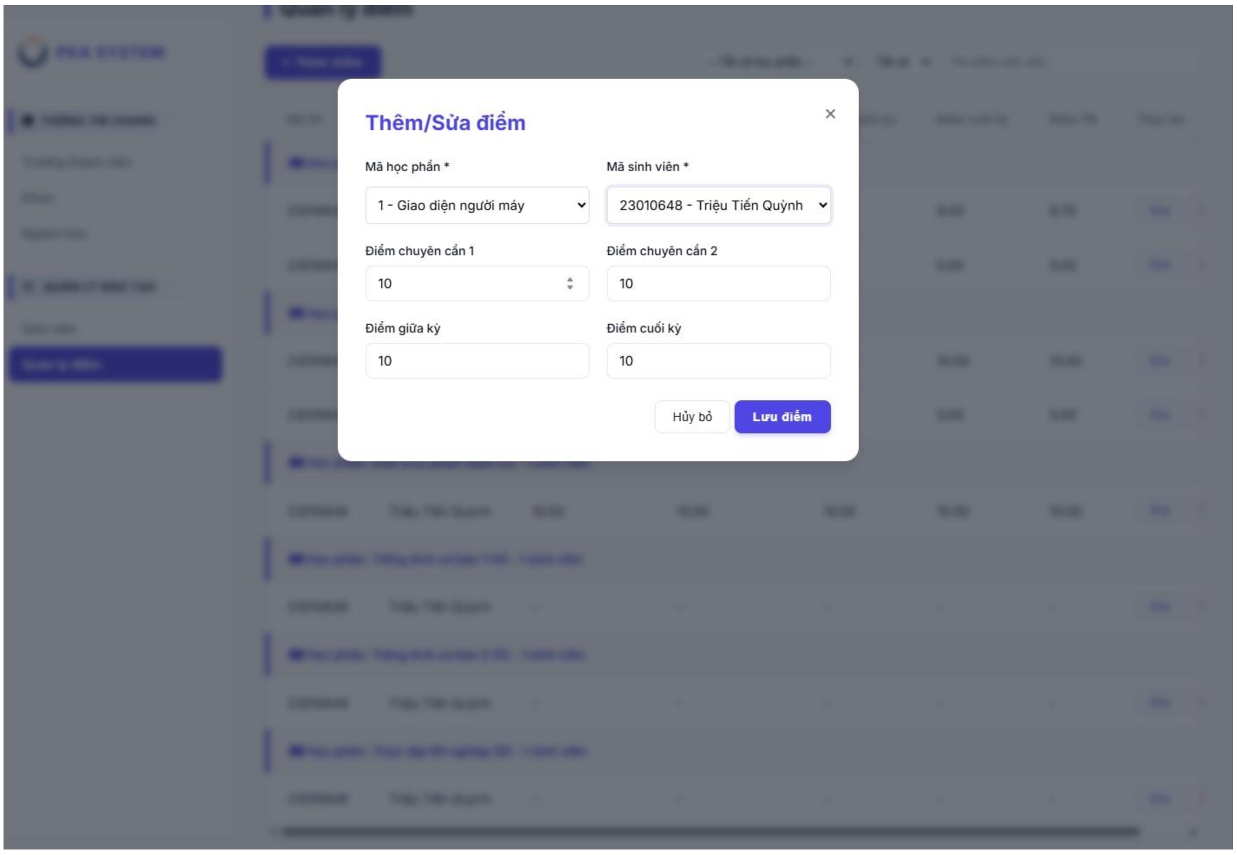
Đăng ký học phần

Số lớp đã đăng ký: 6 Tổng số tín chỉ: 25

Mã HP	Tên học phần	Số tín chỉ	Trạng thái	Thao tác

Bạn đã đăng ký tất cả học phần hiện có

Figure 4.3.2 "Registration Successful" notification on web interface


Thêm/Sửa điểm

Mã học phần *: 1 - Giao diện người máy
Mã sinh viên *: 23010648 - Triệu Tiến Quỳnh
Điểm chuyên cần 1: 10
Điểm chuyên cần 2: 10
Điểm giữa kỳ: 10
Điểm cuối kỳ: 10

Figure 4.3.1 Lecturer's Grade Entry Interface (Lecturer View)

5. Conclusion & Reflection

5.1 Lessons Learned

Through the implementation of the *Academic Management System based on Microservices Architecture*, the project team has gained valuable practical experience that goes beyond theoretical knowledge covered in class.

Firstly, the project helped the team clearly understand the differences between Monolithic and Microservices architectures. While Microservices offer significant advantages in scalability, independent deployment, and fault isolation, they also introduce complexity in service coordination, data consistency, and inter-service communication. Designing clear service boundaries (Identity, Student, Course, Grade, Enrollment, etc.) proved to be a critical architectural decision that directly impacted system maintainability and extensibility.

Secondly, the team gained hands-on experience with distributed system challenges, particularly in ensuring data consistency and fault tolerance. Scenarios such as course registration under high concurrency required careful handling to avoid issues like over-enrollment. This reinforced the importance of architectural patterns such as Database-per-Service, asynchronous processing, and circuit breaker mechanisms to prevent cascading failures when a service becomes unavailable.

Thirdly, the project highlighted the importance of non-functional requirements (NFRs) in architectural design. Performance, security, and reliability were not treated as afterthoughts but were explicitly translated into concrete design decisions, such as JWT-based authentication, role-based access control (RBAC), background email processing, and response-time constraints. This approach helped the team appreciate how quality attributes drive architecture selection and implementation strategies.

Finally, teamwork and collaboration were also key lessons learned. Dividing responsibilities among team members while maintaining consistent API standards and coding conventions required effective communication and documentation. The experience emphasized the value of clear interface contracts, proper logging, and structured testing to support teamwork in real-world software development.

5.2 Future Improvements

Although the system has achieved its primary objectives, several areas can be improved and extended in future iterations.

From an architectural perspective, the system could benefit from introducing a message broker (such as RabbitMQ or Kafka) to replace basic threading for asynchronous tasks. This would improve reliability, scalability, and observability for background processes like email notifications and large data imports.

In terms of scalability and deployment, containerization using Docker and orchestration with Kubernetes would allow services to scale independently based on workload, especially during peak course registration periods. Additionally, implementing centralized monitoring and logging (e.g., Prometheus, Grafana, ELK stack) would enhance system observability and simplify maintenance in production environments.

Regarding security, further enhancements could include token refresh mechanisms, rate limiting at the API Gateway, and more advanced auditing features to track sensitive operations such as grade unlocking or Administrator data changes.

From a functional standpoint, the system can be extended to support additional features such as online payment integration for tuition fees, advanced academic analytics dashboards, and integration with existing university systems (e.g., Learning Management Systems – LMS).

In conclusion, this project serves as a solid foundation and a practical case study for applying Microservices Architecture to a real-world academic management problem. The lessons learned and proposed improvements provide a clear roadmap for future development and deeper exploration of modern distributed software architectures.