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
3	23010648	Trieu Tien Quynh	Core Functional Requirements Architectural Design & Implementation

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. Overview

This report presents the development of an Academic Management System based on Microservices Architecture. The report first describes the core functional requirements of the system, including student management, course registration, grading, GPA calculation, and access control. It then presents the overall architectural design, selected architectural patterns, and the technical stack used for system implementation. Finally, the report explains the implementation details and communication mechanisms between frontend and backend microservices.

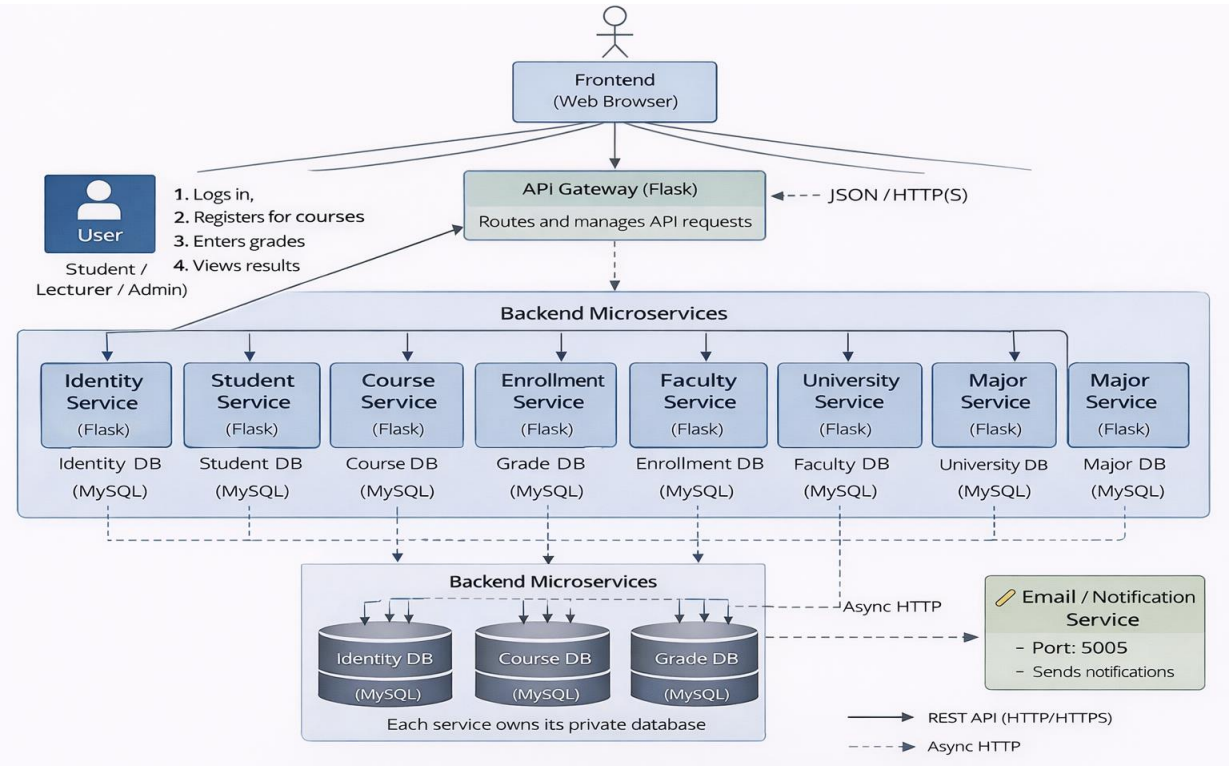
2. Core Functional Requirements

The system fulfills the following key functional requirements:

ID	Function	Detailed Description
FR-01	Catalog Management	Administratoristratoristrators 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 (Administratoristrator/Faculty/Student) via the Identity Service.

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.

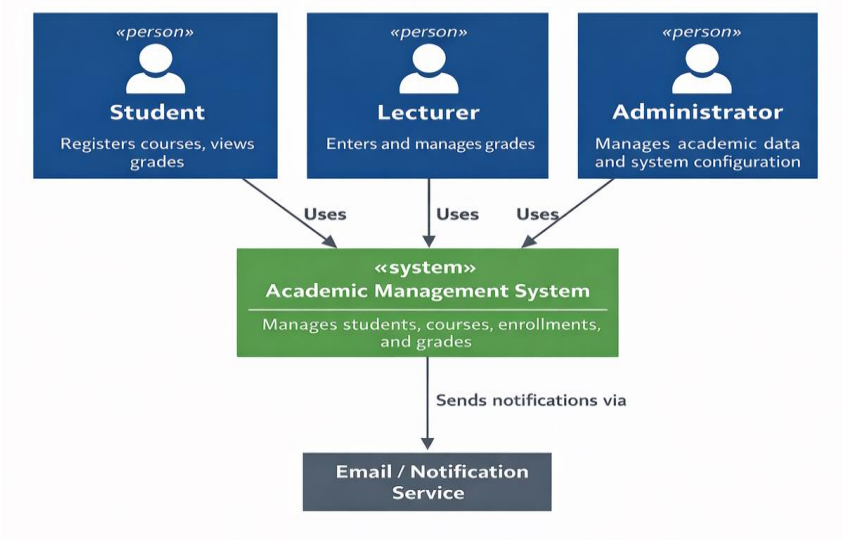


Figure 3.2. C4 – System Context Diagram for Academic Management System

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.

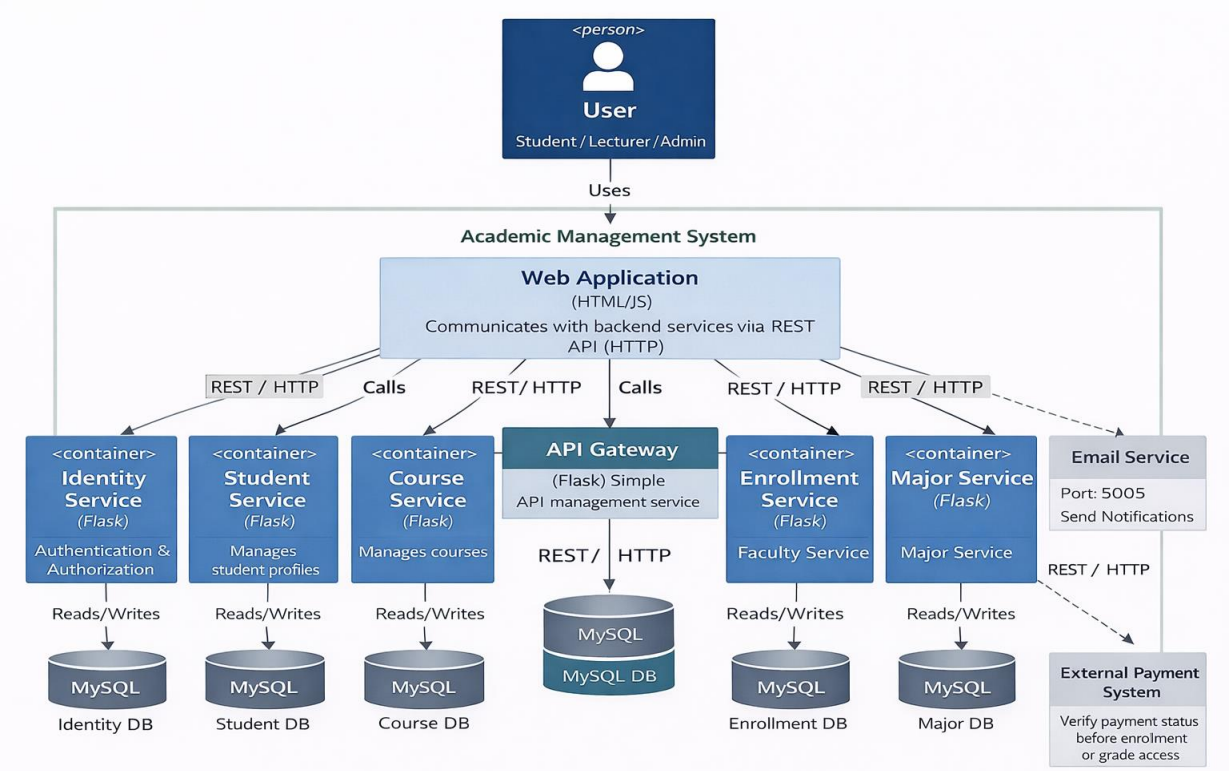


Figure 3.2 presents the C4 Level 2 – Container Diagram, showing the Web Application, the set of backend microservices, and the shared MySQL database.

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.

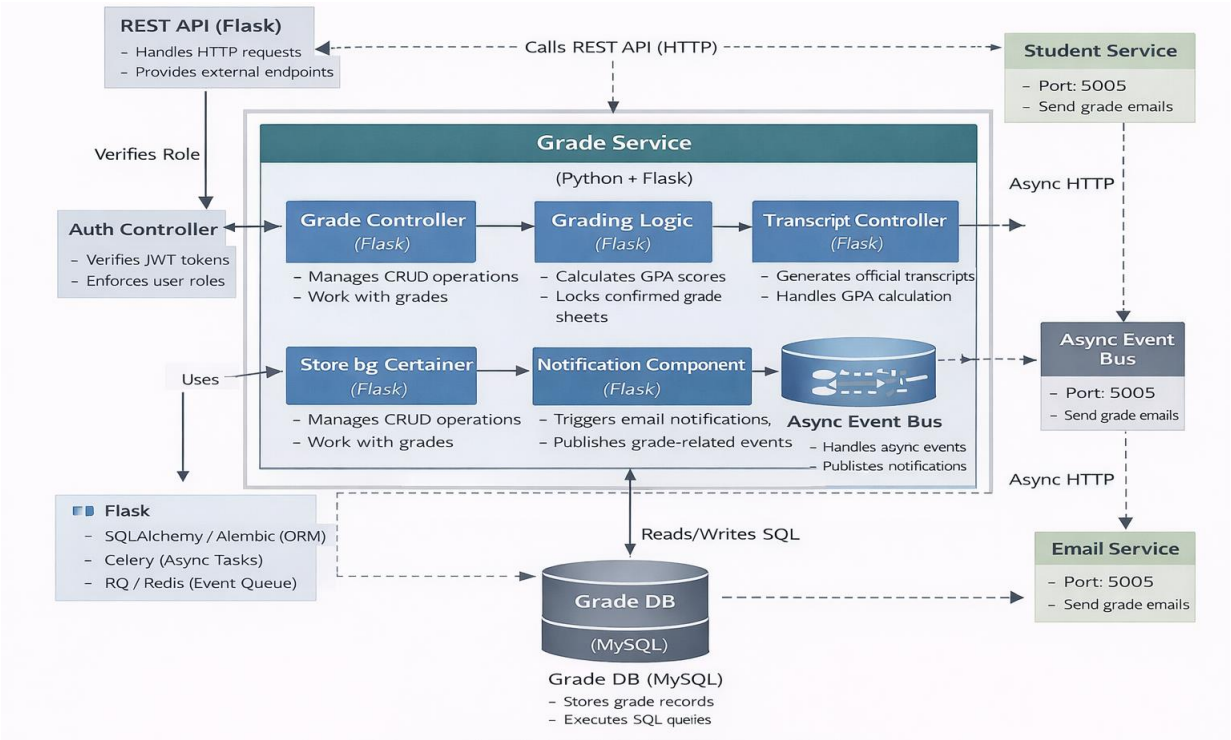


Figure 3.3 shows the C4 Level 3 – Component Diagram of the Grade Service, illustrating the Controller, Repository, Model, and Async Worker components and their interactions.

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.

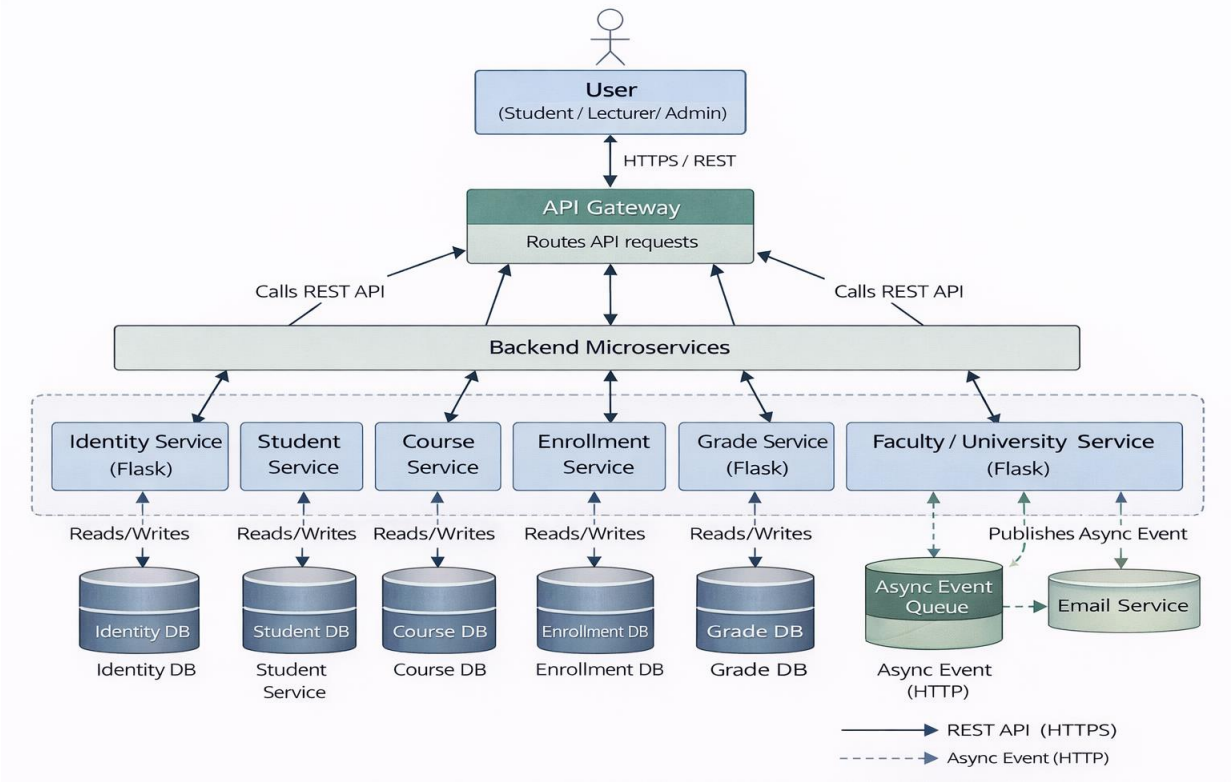


Figure 3.4 shows the High-Level Communication Diagram of the system, highlighting interactions between the Frontend, backend microservices, databases, and the Email Service.

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.