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, February 6, 2026

TASK ASSIGNMENT TABLE

No.	Student ID	Full Name	Assigned Tasks
3	23010648	Trieu Tien Quynh	Part 2 – Requirements Analysis Architecturally Significant Requirements (ASRs) Use Case Modeling: Lecturer Use Cases Part 3 – Architecture Design Component Diagrams C4 Code-Level Design Backend Microservices Structure Communication Model Part 4 – Deployment & Testing Deployment Configuration (Ports & Environment Variables) End-to-End Testing Scenarios

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

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

2.3 Architecturally Significant Requirements (ASR)

- ASR-01 – Performance & Scalability

The Academic Management System must support a large number of concurrent users, especially during peak periods such as course registration phases. During these periods, the system experiences a significant increase in access frequency and processing requests from students.

High-traffic and computation-intensive functionalities, particularly course registration and academic result lookup, must not become performance bottlenecks that degrade the overall system responsiveness.

To address this requirement, the system is designed using a Microservices Architecture. High-load functionalities are implemented as independent services, such as the Enrollment Service and the Grade Service, instead of being tightly coupled within a monolithic application.

This architectural decision allows workload distribution across services, reduces resource contention, and provides a solid foundation for horizontal scalability (scale-out) during future system growth.

- ASR-02 – Security

The system must ensure information security and strict access control among different user roles, including Administrator, Lecturer, and Student. Unauthorized users must not be able to access or manipulate sensitive academic data.

All requests sent from the Frontend to the Backend must be authenticated before any business logic is executed. Authentication ensures that the identity of the user is verified.

In addition to authentication, authorization must be enforced based on user roles. Different roles are granted different permissions, such as grade management for Lecturers and system configuration for Administrators.

To satisfy this requirement, the system adopts a centralized authentication and authorization mechanism through the Identity Service. Access Tokens containing role information are issued after login and validated by backend microservices for every protected API request.

- ASR-03 – Reliability & Availability

The system must ensure high reliability and continuous operation, even when individual components encounter failures. A failure in one service should not cause the entire system to become unavailable.

Non-time-critical functionalities, such as sending notification emails for grade updates or course registration confirmations, must not block or slow down user-facing operations.

Therefore, these functionalities are handled asynchronously through the Email Service and Notification Service. The main business services only trigger events and continue processing without waiting for notification tasks to complete.

This approach improves fault tolerance, limits failure propagation, and ensures that core system operations remain responsive under various failure scenarios.

- ASR-04 – Maintainability & Extensibility

The system must be easy to maintain and adaptable to future changes in business requirements. Adding new features or modifying existing ones should not require extensive changes across the entire system.

To meet this requirement, the system is decomposed into independent microservices based on business domains, such as Student, Course, Enrollment, Grade, Faculty, Major, and University.

Each service encapsulates its own business logic and data access responsibilities. Communication between services is performed through well-defined RESTful APIs.

This architectural decomposition reduces coupling between components, simplifies maintenance, and enables the system to evolve incrementally as new requirements emerge.

- ASR-05 – Usability

The system's user interface must be intuitive, user-friendly, and easy to modify or enhance over time. Changes in the user interface should not impact backend business logic.

To achieve this goal, the system adopts a fully decoupled Frontend–Backend architecture. The Backend focuses solely on business logic and data processing and exposes functionality through RESTful APIs.

All presentation logic and user interactions are implemented on the Frontend as a Single Page Application using Vanilla JavaScript.

This separation allows the user interface to be updated, redesigned, or replaced independently, thereby improving usability and long-term flexibility of the system.

2.4 Use Case Modeling: Use case Lecturer

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-conditions	Success: 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.		

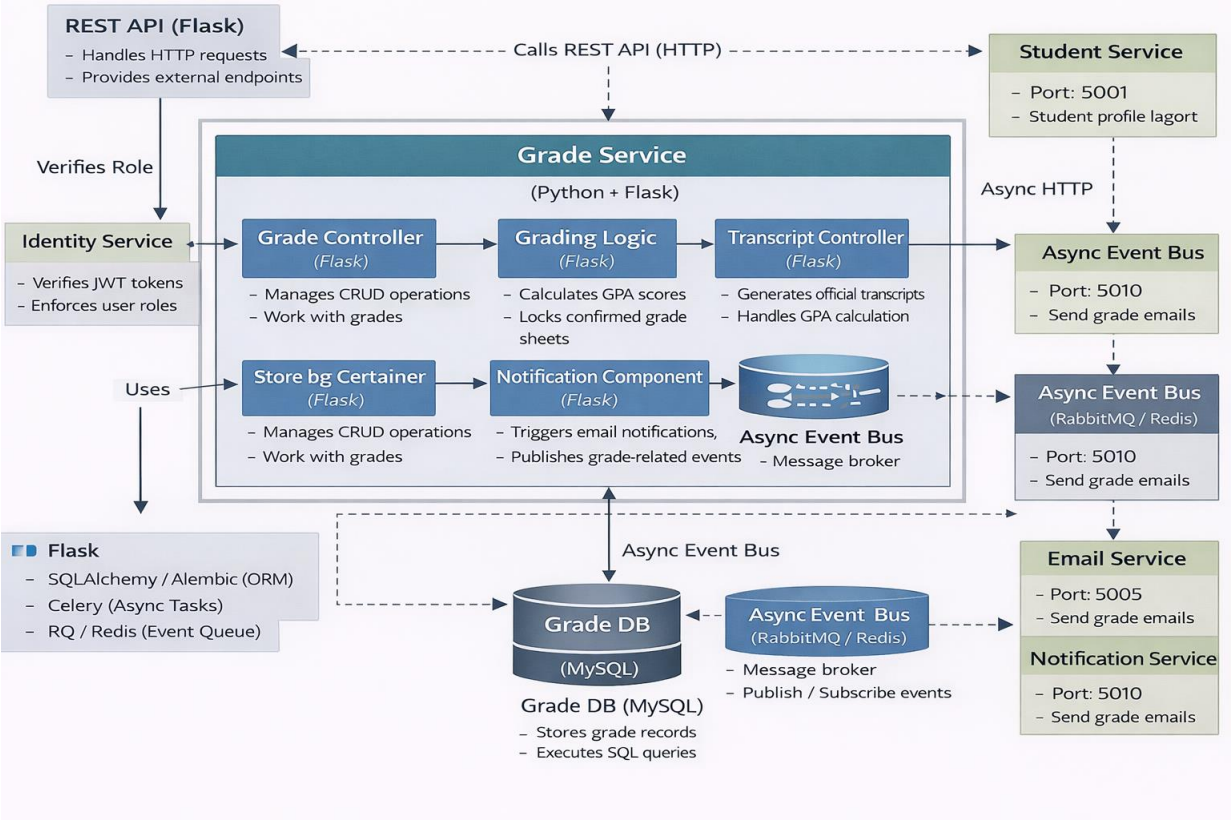
Main Flow	1. Review Grades: - Lecturer reviews the calculated Total Scores and ensures all data is correct. 2. Initiate Lock: - Lecturer clicks the "Finalize & Lock" button. 3. System Confirmation: - System displays a warning modal: *"Are you sure you want to lock this grade sheet? This action cannot 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

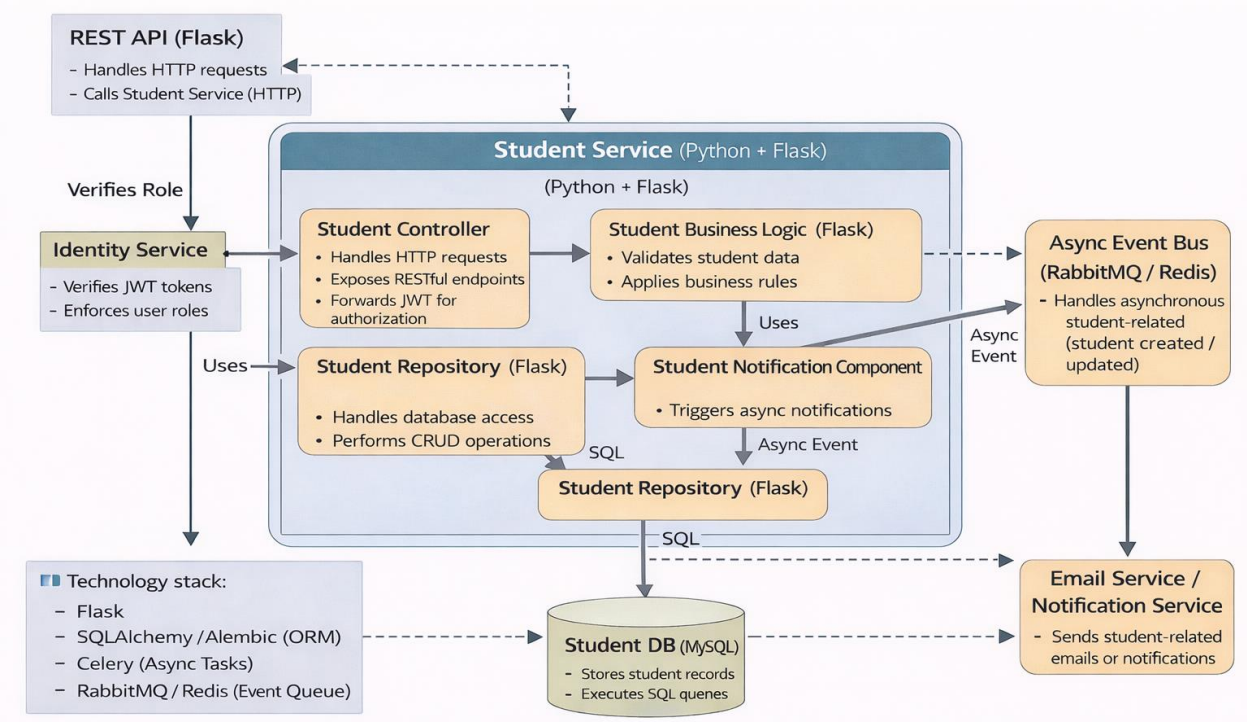
3. Architectural Design & Implementation

3.3 Architectural Views – C4 Model Representation

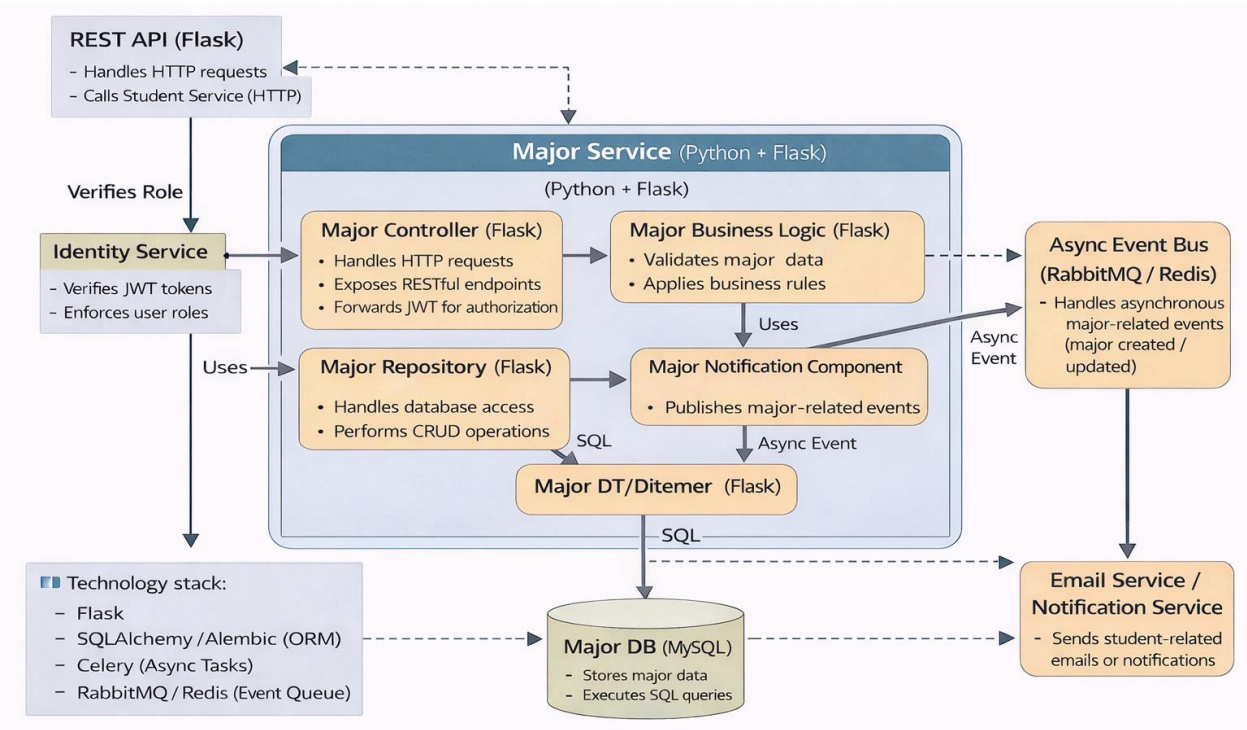
3.3.2 C4 – Component Diagram



The figure shows the C4 – Component Diagram of the Grade Service

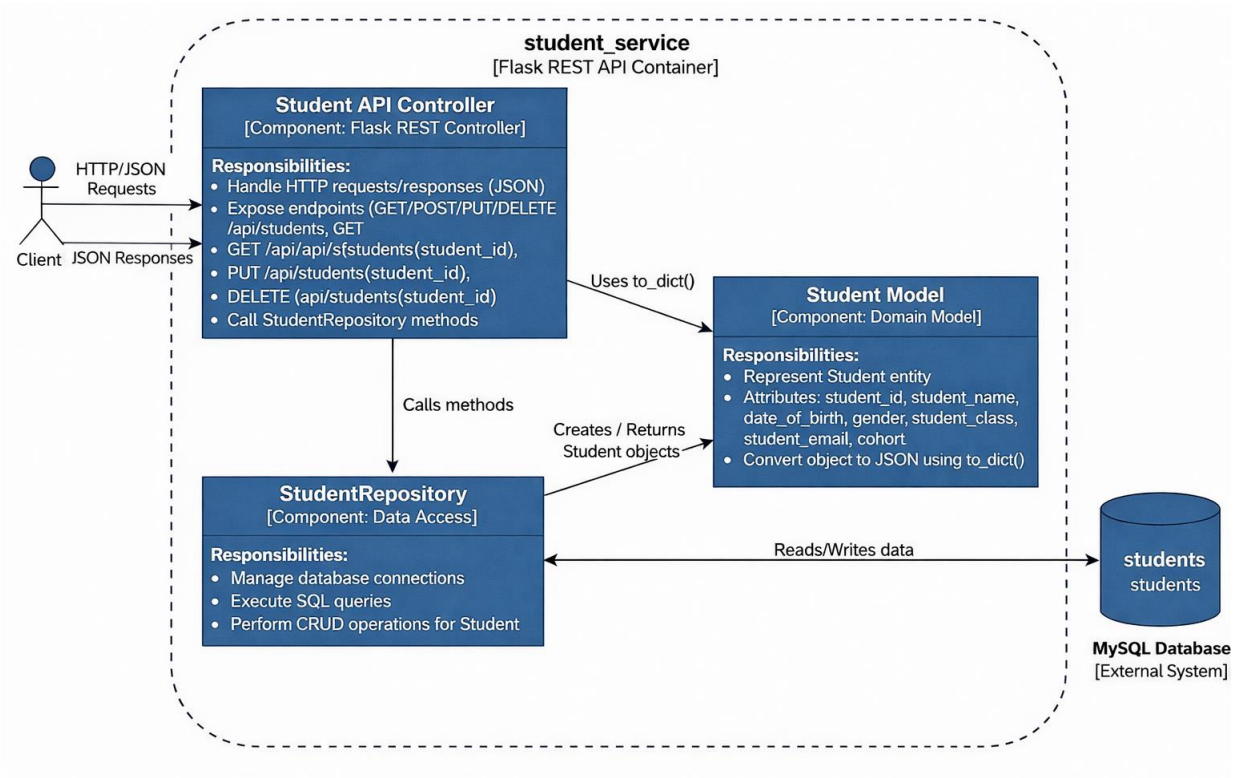


The figure shows the C4 – Component Diagram of the Student Service



The figure shows the C4 – Component Diagram of the Major Service

3.3.3 C4 – Code Level



The figure illustrates the C4 Code Level for the Student Service

3.5 Implementation Design

3.5.1 Backend Microservice Structure

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.

3.5.2 Communication Model

The system implements a Hybrid Communication Strategy combining synchronous and asynchronous patterns to ensure both performance and reliability:

Synchronous Communication (REST API): Most interactions between the Client and Microservices (e.g., Login, Course Registration) use standard HTTP/REST requests.

Asynchronous Communication (Hybrid Approach):

+ Threading (Simple Async): Non-critical notifications use Python's threading module for "fire-and-forget" tasks (e.g., triggering email calls via requests.post to Port 5005).

+ Event-Driven Architecture (RabbitMQ): For critical data consistency, specifically in the Grade Service, the system integrates RabbitMQ as a Message Broker. When a grade is updated, a GRADE_UPDATED event is published to the grade_events queue. This allows other services to subscribe and react to grade changes independently, decoupling the grading logic from downstream effects.

4. Testing & Verification

4.2 Deployment Configuration (Ports & Env)

To support local development and testing, the system is designed with clear environment variable configuration and distinct port assignments for each Microservice.

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.

Configuration file (.env):

DB_HOST=localhost

DB_USER=root

DB_PASSWORD=your_password

DB_NAME=sa

This approach improves security and simplifies configuration changes across environments.

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
Notification Service	5010	RabbitMQ consumer for real-time student alerts.	notification_service/app.py

4.2.3 Deployment Experiment

❖ Class Diagram of Student Service

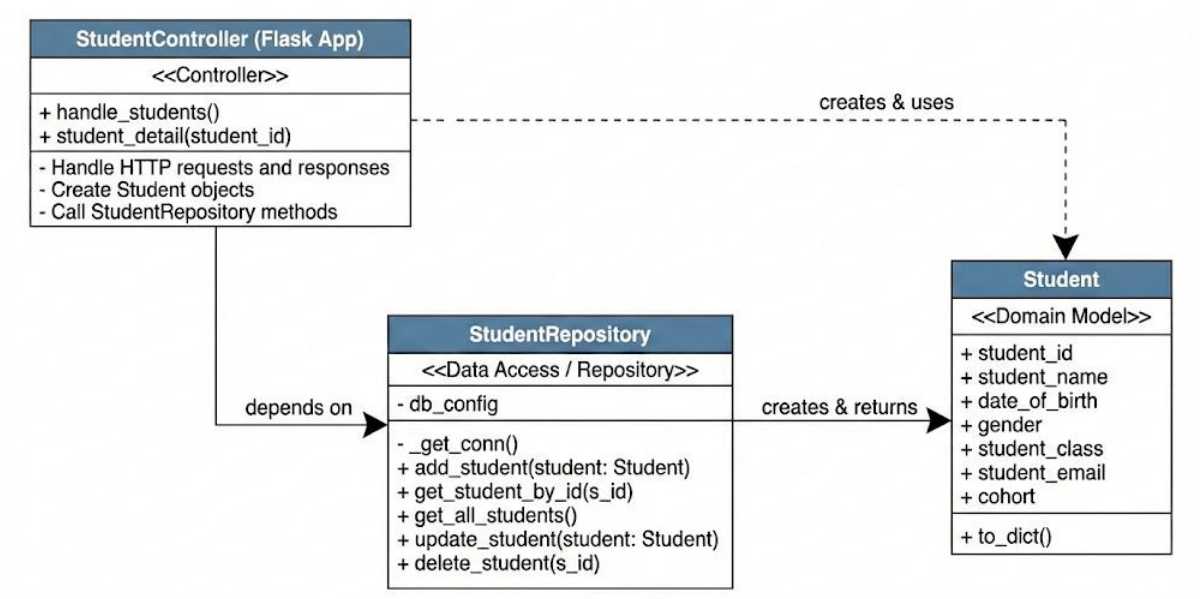
The class diagram illustrates the implementation structure of the Student Service during deployment.

+ StudentController: Receives HTTP requests, handles request/response processing, and invokes methods provided by the StudentRepository.

+ StudentRepository: Performs CRUD operations on the MySQL database and returns Student objects.

+ Student (Domain Model): Represents the student entity and provides the to_dict() method to convert data into JSON format.

The experiment demonstrates that the processing flow Controller → Repository → Database → Domain Model operates correctly and is consistent with the designed architecture.



The figure Class diagrams for Student Services

❖ Class Diagram of Faculty Services

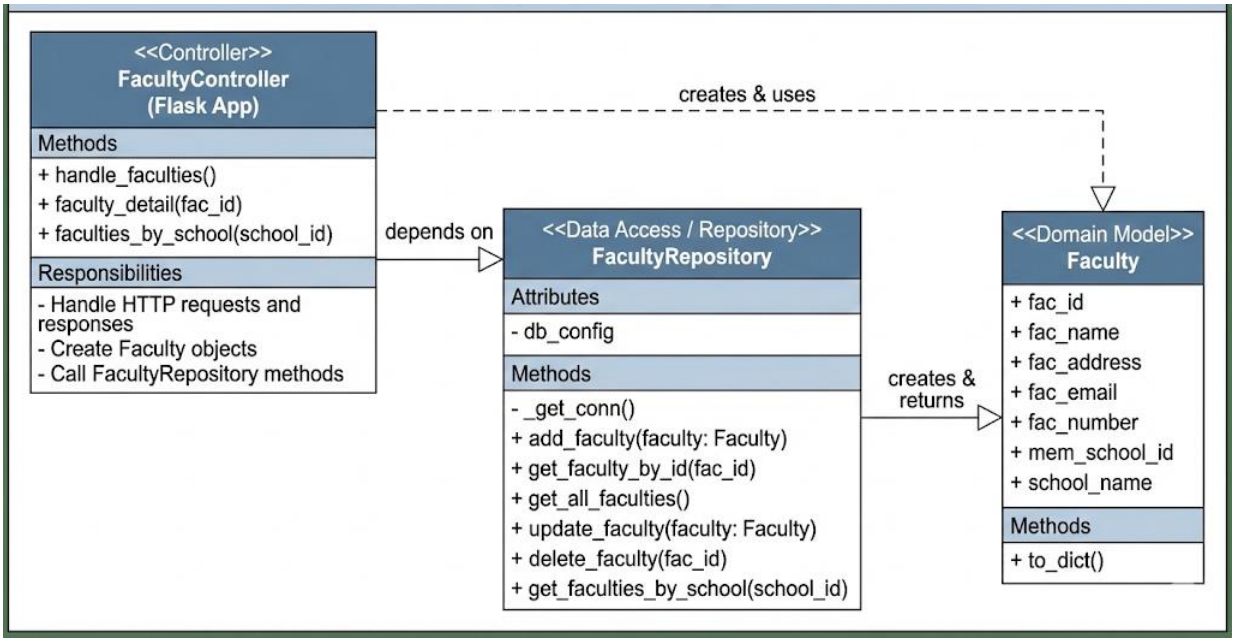
The class diagram presents the implementation of the Faculty Service in the deployed system.

+ FacultyController: Handles HTTP requests related to faculty management and coordinates business logic processing.

+ FacultyRepository: Accesses the database and performs create, retrieve, update, and delete operations for faculty data.

+ Faculty (Domain Model): Represents faculty information and supports data conversion to JSON format.

The experimental results confirm that the Faculty Service is implemented according to the designed model and ensures clear separation of responsibilities among components.



The figure Class diagrams for Faculty Services

4.3 End-to-End Test Scenarios

End-to-End Testing simulates a real-world business workflow, validating data flow across multiple services from the Frontend to the Database.

Scenario: Course Enrollment and Grading Workflow

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.