PHENIKAA UNIVERSITY

PHENIKAA SCHOOL OF COMPUTING

COURSE: SOFTWARE ARCHITECTURE

TOPIC:

DEVLOPMENT OF TRAINING PROGRAM MANAGEMENT SOFTWARE

FINAL REPORT

Code Course	:	CSE703110-1-1-25
Class	:	N01
Instructor	:	THS. Vũ Quang Dũng
Group	:	12
Members	:	1. Đồng Đại Đạt : KTPM.EL2 - 23010877
		2. Nguyễn Cao Hải : KTPM.EL2 - 23013124
		3. Ngô Nhật Quang : KTPM.EL1 - 23010134
		4. Nguyễn Nam Khánh : KTPM.EL2 - 23010771

Hanoi, November 2025

Student Name	Position	Task
Đồng Đại Đạt	Leader	(1) Requirement Analysis (2) Architectural Design
Nguyễn Nam Khánh	Member	(1) Requirement Analysis
Nguyễn Cao Hải	Member	(2) Architectural Design
Ngô Nhật Quang	Member	(3) API & Testing

Table of Contents

1. Executive Summary

This project details the design and implementation of the University Management Information System (UniMIS), a centralized platform developed to streamline curriculum management and academic structuring. The system addresses the critical challenges of fragmented data and inconsistent updates across university departments.

Architecturally, the project prioritizes long-term maintainability and scalability over short-term ease of implementation. To satisfy requirements for both modifiability and performance, the system employs a Modular Monolithic architecture paired with a Layered Design approach. By integrating microservice design principles - such as clear business boundaries and service contracts - the architecture ensures modularity while avoiding the overhead of premature distributed deployment.

UniMIS is built as a single deployable unit and containerized via Docker to guarantee environment consistency and simplify orchestration. The system's robustness was validated through rigorous functional testing, while performance benchmarks confirmed its reliability: the system successfully handled 200 concurrent users under stress testing, maintaining response times under two seconds. These results demonstrate that the chosen architecture effectively balances operational simplicity with high performance, providing a sustainable foundation for future architectural evolution.

2. Project Requirements & Goals

2.1. Core Functional Requirements

a. Actors

Actor	Role / Responsibilities	Interaction With System
Admin	Manage the entire system, authorize, and manage users	Login → Account Management → Assign Roles
	Lecturer management, organizational structure	Login → Account Management→ Update lecturer
Training Management Department	Create/edit/delete training programs by course, training program structure, knowledge block, tuition fees,..	Training Program Management → Assign subjects to semesters → Export PDF/Excel
Student / Viewer	Look up training programs, tuition fees, and knowledge blocks,…	CRUD subject → Update lecturer → Assign subject to knowledge block

b. Functional Requirements (FRs)

ID	Functional Requirement	Priority
FR-01	The system must allow Academic to create, update, and delete training programs for each intake year.	High
FR-02	The system must allow to manage course information, including credits, prerequisites, lecturers, and knowledge blocks	High
FR-03	The system must allow the assignment of courses to semesters within a training program.	High
FR-04	The system must allow the management of departments, faculties, and lecturers.	Medium
FR-05	The system must generate downloadable PDF files of tuition fee structures.	Medium
FR-06	The system must allow Admin to manage user accounts and assign roles.	High
FR-07	The system must allow Students and Viewers to view training programs and tuition information by academic year.	Medium
FR-08	The system allows copying training programs to facilitate adding new and editing training programs.	Medium

c. Non-Functional Requirements (NFRs)

ID	Attribute	Description	Impact
NFR-01	**Performance (Latency)**	The system must load a complete training program (including courses, knowledge blocks, and tuition details) within **2.0 seconds** under normal load.	High
NFR-02	**Security (Integrity)**	All sensitive academic data (program structures, tuition fee configurations, user roles) must be securely stored . Access to modification functions must enforce strict role-based control.	Critical
NFR-03	**Reliability (Availability)**	The system must maintain at least 99.0% uptime during academic administrative hours	Critical
NFR-04	**Usability**	The user interface must be intuitive and optimized for desktop, allowing users to easily navigate through training programs and course structures.	Medium

d. Use Case List by Functional Group

UC1: Organizational Structure Management

UC-1.1 Manage Faculty	
Description	Admin performs Add/Edit/Delete (or Deactivate)/Assign/Search/Display operations for Faculty information.
Trigger	Admin opens Organizational Structure -> Faculty.
Preconditions	Admin is authenticated and authorized; system and database are available.
Postconditions	Faculty data is updated according to the selected action; the list is refreshed; audit information is recorded.
Normal Flows	1) Admin accesses Faculty management. 2) System displays list and available actions. 3) Admin performs a management action. 4) System validates and applies changes.
Alternative Flows	A1) Admin uses filters/keyword search to narrow results. A2) Admin cancels an in-progress form.
Exceptions	E1) Duplicate code/name. E2) Deletion conflicts with dependent data -> suggest deactivation. E3) System/DB error.
Priority	High
Frequency of Use	Medium
Business Rules	Faculty code is unique; required fields must be provided; status lifecycle is enforced.
Assumptions	Faculty includes at least code, name, description, status, created/updated metadata.

UC-1.2 Manage Major	
Description	Admin performs Add/Edit/Delete (or Deactivate)/Assign/Search/Display operations for Major information and manages the Major–Faculty linkage.
Trigger	Admin opens Organizational Structure->Major.
Preconditions	Admin is authenticated and authorized; at least one Faculty exists.
Postconditions	Major data and Major–Faculty linkage are updated; list is refreshed; audit is recorded.
Normal Flows	1) Admin opens Major management. 2) System shows list/actions. 3) Admin performs a management action. 4) When required, Admin selects a valid Faculty. 5) System validates and saves.
Alternative Flows	A1) Admin reassigns a Major to another Faculty. A2) Filter by Faculty/status.
Exceptions	E1) Duplicate code/name. E2) Selected Faculty invalid/inactive. E3) Deletion blocked by curriculum references.
Priority	High

Frequency of Use	Medium
Business Rules	Each Major must belong to a valid Faculty; codes are unique; only active units can be assigned.
Assumptions	Major includes code, name, faculty_id, status, metadata.

UC-1.3 Query Faculty/Major	
Description	Student/Viewer searches and views the list/details of Faculties and Majors.
Trigger	Student/Viewer opens Faculty/Major Directory.
Preconditions	View access is allowed; system is available.
Postconditions	User sees accurate lists/details of Faculties and Majors.
Normal Flows	1) User accesses the directory. 2) System displays data. 3) User searches/filters. 4) System returns the matching results.
Alternative Flows	A1) View details of a selected Faculty/Major.
Exceptions	E1) System unavailable.
Priority	Medium
Frequency of Use	High
Business Rules	Student/Viewer sees only active/approved records.
Assumptions	Search supports basic keyword and filter conditions.

UC2: Personnel Management

UC-2.1 Query Faculty/Major	
Description	Admin performs Add/Edit/Delete (or Deactivate)/Assign/Search/Display operations for lecturer profiles.
Trigger	Admin opens Personnel->Lecturers.
Preconditions	Admin is authenticated and authorized; organizational units exist if assignment is required.
Postconditions	Lecturer data/assignments are updated; list refreshed; audit recorded.
Normal Flows	1) Admin opens Lecturer management. 2) System shows list/actions. 3) Admin performs a management action. 4) System validates and applies changes.
Alternative Flows	A1) Assign lecturers to Faculty/Major/Department if supported.
Exceptions	E1) Duplicate lecturer code/email. E2) Hard delete blocked by teaching/curriculum links.
Priority	Medium
Frequency of Use	Medium
Business Rules	Lecturer identifiers are unique; assignment only to valid active units.

Assumptions	Lecturer includes code, name, degree/title, contact, unit linkage, status.

UC-2.2 Query Faculty/Major	
Description	Student/Viewer searches and views the list/details of lecturers.
Trigger	Student/Viewer opens Lecturer Directory.
Preconditions	View access is allowed.
Postconditions	Lecturer lists/details are shown correctly.
Normal Flows	1) User opens directory. 2) System displays list. 3) User searches/filters.
Alternative Flows	A1) Filter lecturers by Faculty/Major.
Exceptions	E1) System unavailable.
Priority	Medium
Frequency of Use	Medium
Business Rules	Only active lecturers are visible to Student/Viewer.
Assumptions	Basic search by name/code is supported.

UC3: Curriculum Development

UC-3.1 Curriculum Development	
Description	Admin performs Add/Edit/Delete/Assign/Search/Display operations for course/module records.
Trigger	Admin opens Curriculum -> Courses/Modules.
Preconditions	Admin is authenticated and authorized.
Postconditions	Course/module data is updated; list refreshed; audit recorded.
Normal Flows	1) Open management screen. 2) Perform a management action. 3) System validates credits/prerequisites. 4) Save changes.
Alternative Flows	A1) Search/filter by code, name, status.
Exceptions	E1) Duplicate course code. E2) Invalid credit/prerequisite.
Priority	High
Frequency of Use	High
Business Rules	Course code unique; credit value must be valid; prerequisite must exist when specified.
Assumptions	Course includes code, name, credits, optional prerequisites, status.

UC-3.2 Manage Enrollment Cohorts	
Description	Admin performs Add/Edit/Delete/Assign/Search/Display operations for enrollment cohorts/years.
Trigger	Admin opens Curriculum -> Enrollment Cohorts/Years.
Preconditions	Admin is authenticated and authorized.
Postconditions	Cohort/year data is updated and ready for CTĐT versioning.
Normal Flows	1) Open cohort management. 2) Create/update/deactivate cohort. 3) Save.
Alternative Flows	A1) Create a new cohort based on a previous template (if supported).
Exceptions	E1) Duplicate cohort identifier/year.
Priority	Medium
Frequency of Use	Medium
Business Rules	Cohort identifier must be unique; cohort should map to valid majors/programs.
Assumptions	Cohort includes year, code, status, metadata.

UC-3.3 Manage Knowledge Blocks	
Description	Admin performs Add/Edit/Delete/Assign/Search/Display operations for knowledge blocks within the curriculum.
Trigger	Admin opens Curriculum -> Knowledge Blocks.
Preconditions	Admin is authenticated and authorized.
Postconditions	Knowledge block data is updated; list refreshed; audit recorded.
Normal Flows	1) Open blocks screen. 2) Perform a management action. 3) Save.
Alternative Flows	A1) Configure ordering or optional credit constraints.
Exceptions	E1) Duplicate block code/name.
Priority	High
Frequency of Use	Medium
Business Rules	Block code unique; only active blocks can be assigned into CTĐT.
Assumptions	Block includes code, name, description, optional rules, status.

UC-3.4 Manage CTĐT Structure	
Description	Admin performs Add/Edit/Delete/Assign/Search/Display operations for the CTĐT structure, organized by semester/course groups.
Trigger	Admin opens Curriculum -> CTĐT Structure.
Preconditions	Admin is authenticated; courses/blocks exist.

Postconditions	CTĐT structure by semester/groups is updated and stored.
Normal Flows	1) Select major/cohort/version. 2) Open structure editor. 3) Arrange semesters/groups. 4) Assign courses/blocks. 5) Save.
Alternative Flows	A1) Reorder semesters/groups.
Exceptions	E1) Constraint violations based on framework rules.
Priority	High
Frequency of Use	High during curriculum revisions
Business Rules	Structure must comply with CTĐT framework constraints; only active entities can be assigned.
Assumptions	The system supports semester-based organization.

UC-3.5 Manage CTĐT Framework	
Description	Admin performs Add/Edit/Delete/Assign/Search/Display operations for the CTĐT framework, based on program standards.
Trigger	Admin opens Curriculum ->CTĐT Framework.
Preconditions	Admin is authenticated and authorized.
Postconditions	Framework standards/credit rules are updated and versioned.
Normal Flows	1) Open framework management. 2) Configure standards and constraints. 3) Save.
Alternative Flows	A1) Compare framework versions (if supported).
Exceptions	E1) Conflicts with existing approved structures.
Priority	High
Frequency of Use	Medium
Business Rules	Framework changes should not invalidate historical cohorts; versioning is recommended.
Assumptions	Framework includes total credits, rules, applicability range.

UC-3.6 Copy CTĐT	
Description	Admin copies an existing CTĐT to create a new version for a different enrollment cohort or year.
Trigger	Admin selects Copy CTĐT on a source program.
Preconditions	Source CTĐT exists; Admin has copy permission.
Postconditions	A new CTĐT version is created for the target cohort/year.
Normal Flows	1) Select source CTĐT. 2) Choose target cohort/year/major. 3) Confirm scope.

	4) System duplicates data. 5) Admin reviews.
Alternative Flows	A1) Copy structure without tuition settings if policy allows.
Exceptions	E1) Target already has a CTĐT version -> system prompts resolution.
Priority	Medium
Frequency of Use	Medium
Business Rules	Copy must preserve data integrity; edits on the new version must not affect the source.
Assumptions	CTĐT supports versioning by cohort/year.

UC-3.7 Display/Query CTĐT	
Description	Admin/Student/Viewer views the CTĐT based on major, cohort, or version.
Trigger	Admin/Student/Viewer opens View CTĐT.
Preconditions	CTĐT data exists; view access is allowed.
Postconditions	The selected CTĐT is displayed correctly.
Normal Flows	1) User selects major/cohort/version. 2) System loads structure, blocks, courses. 3) Display details.
Alternative Flows	A1) Export CTĐT (if enabled).
Exceptions	E1) Missing or invalid selected version.
Priority	Medium
Frequency of Use	High
Business Rules	Student/Viewer sees only approved/active versions.
Assumptions	Filters by major/cohort/version are available.

UC4: Tuition Fee Calculation

UC-4.1 Set Up Tuition Fee Unit	
Description	Admin configures the unit price and calculation basis for tuition fees according to regulations.
Trigger	Admin opens Tuition -> Fee Unit Setup.
Preconditions	Admin is authenticated and authorized.
Postconditions	Tuition unit price and calculation basis are stored; audit recorded.
Normal Flows	1) Open settings. 2) Input unit price/rules. 3) Validate. 4) Save.
Alternative Flows	A1) Configure by cohort/major if supported.
Exceptions	E1) Invalid input ranges or rule conflicts.
Priority	Medium

Frequency of Use	Medium
Business Rules	Unit settings must follow institutional regulations; effective dating/versioning is recommended.
Assumptions	Tuition configuration is linked to cohorts/CTĐT versions.

UC-4.2 Display Tuition Fee Information	
Description	Admin/Student/Viewer views tuition fee information specific to a CTĐT, cohort, or academic year.
Trigger	Admin/Student/Viewer opens View Tuition for a selected CTĐT/cohort/year.
Preconditions	Tuition units/rules are configured; view access is allowed.
Postconditions	Tuition information is displayed accurately.
Normal Flows	1) Select major/cohort/year/CTĐT. 2) System retrieves/calculates tuition. 3) Display breakdown.
Alternative Flows	A1) Export tuition information (if enabled).
Exceptions	E1) Missing tuition configuration for selection.
Priority	Medium
Frequency of Use	Medium
Business Rules	Displayed tuition must align with the active, approved configuration.
Assumptions	Calculation uses credits and configured unit prices.

2.2. Key Quality Attributes

Architecturally Significant Requirements (ASRs)

ID	Quality Attribute	Description	Impact
ASR-01	**Modifiability**	Updating or adding a new training program for a new intake must not affect the Course or Lecturer components.	Requires Layered Architecture, separating Program Logic from Course Repository; ensures maintainability.
ASR-02	**Security**	Only Admin may change training program structures, tuition data, or knowledge blocks.	Requires centralized Authorization Service in the Business Logic Layer; enforces role-based access.
ASR-03	**Scalability**	System Support application extension, easy to add new functions	Encourages stateless backend, Docker deployment, and potential horizontal scaling.

2.3. Propose model of ASR

Propose Model of Architecturally Significant Requirements (ASR)

Architecturally Significant Requirements (ASRs) are requirements that have a direct and substantial impact on the architectural design of the system. For the Curriculum Management System, the proposed ASRs are derived from the key quality attributes identified in Section 2.2 and serve as the primary drivers for selecting and structuring the layered monolithic architecture.

ASR-01: Maintainability and Modifiability

- Type: Quality Attribute
- Description: The system must support frequent changes to training programs, course structures, and curriculum regulations with minimal impact on existing functionality.
- Architectural Impact: This requirement drives the adoption of a layered monolithic architecture, where responsibilities are clearly separated into presentation, service, domain, and data access layers. Such separation allows changes to be localized within specific layers, improving maintainability and reducing coupling between components.

ASR-02: Curriculum Versioning Support

- Type: Functional / Quality Attribute
- Description: The system must support multiple versions of a curriculum, ensuring that changes to a curriculum do not affect students who are already following an earlier version.
- Architectural Impact: This requirement influences the domain model design, introducing explicit entities for curriculum versions and enforcing business rules that prevent modification of published or active versions. Centralized domain logic within the monolithic architecture simplifies version control and ensures consistent behavior across the system.

ASR-03: Data Consistency and Integrity

- Type: Quality Attribute
- Description: All operations related to curriculum creation, modification, and approval must maintain strong data consistency and integrity.

- Architectural Impact: This requirement motivates the use of a centralized database and transaction management within a single application context. The monolithic architecture enables straightforward handling of transactional boundaries, reducing the risk of data inconsistency compared to distributed approaches.

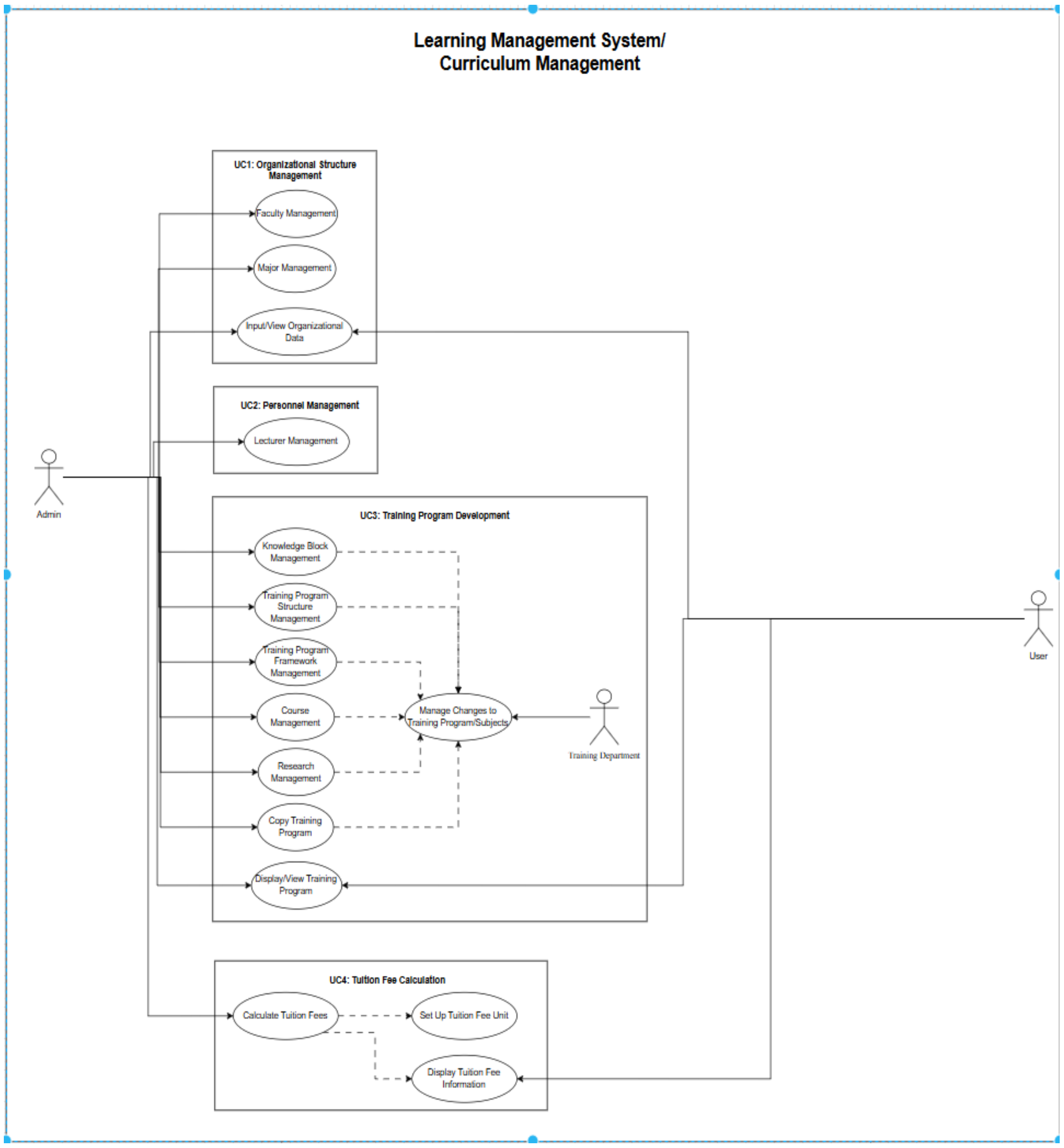
ASR-04: Role-Based Access Control

- Type: Quality Attribute (Security)
- Description: The system must restrict access to curriculum management functions based on user roles, such as administrators, training staff, lecturers, and students.
- Architectural Impact: This requirement affects the design of the service and presentation layers, where authorization checks are enforced consistently before executing sensitive operations. Centralized security logic within the monolith ensures uniform policy enforcement and simplifies security management.

ASR-05: Acceptable Performance for Academic Environment

- Type: Quality Attribute (Performance)
- Description: The system must provide responsive interaction for common operations, such as viewing curriculum details and performing administrative updates, under a moderate number of concurrent users.
- Architectural Impact: This requirement confirms that a monolithic architecture is sufficient for the expected workload, avoiding unnecessary complexity while still meeting performance expectations for an academic context.

2.4. Use Case Modeling


Learning Management System/
Curriculum Management
UC1: Organizational Structure Management
Faculty Management
Major Management
Input/View Organizational Data
UC2: Personnel Management
Lecturer Management
Admin
UC3: Training Program Development
Knowledge Block Management
Training Program Structure Management
Training Program Framework Management
Course Management
Research Management
Copy Training Program
Display/View Training Program
Manage Changes to Training Program/Subjects
Training Department
User
UC4: Tuition Fee Calculation
Calculate Tuition Fees
Set Up Tuition Fee Unit
Display Tuition Fee Information

The use case diagram illustrates the functional scope of the Curriculum Management System by presenting the interactions between different user roles and the core system functionalities. The system boundary represents the Learning Management System with a focus on curriculum and training program management, while the actors placed outside the boundary indicate the primary stakeholders interacting with the system.

The diagram identifies multiple user roles, including administrators, training department staff, and general users. Each actor interacts with a specific set of use cases that reflect their responsibilities within the system. Administrative users are primarily responsible for organizational structure management, user and lecturer management, and configuration-related tasks. These use cases support the foundational setup of the system and ensure that academic structures and personnel data are maintained consistently.

The central part of the diagram focuses on training program development and curriculum management, which represent the core business capabilities of the system. Use cases such as knowledge block management, training program structure definition, course management, and training program publishing are grouped together to emphasize their close relationship within the same business domain. The diagram also highlights that changes to training programs are managed through controlled processes, ensuring that curriculum updates follow defined rules and approval flows.

In addition, the diagram includes use cases related to tuition fee calculation and information display. These use cases are logically separated to indicate that they depend on curriculum and program data but do not directly modify it. This separation reflects a read-oriented interaction pattern, particularly relevant for users who only need to view academic or financial information.

Overall, the use case diagram is intentionally designed at a high level to capture only architecturally significant use cases. Supporting or trivial use cases, such as authentication or profile management, are omitted to maintain focus on functionality that influences architectural decisions. This modeling approach helps clarify system scope, actor responsibilities, and the relationship between core business processes without overwhelming the architecture analysis with excessive detail.

3. Architectural Design & Implementation

3.1. Architectural Pattern

The Curriculum Management System adopts a Modular Monolithic Architecture as its primary architectural pattern. This architectural style combines the simplicity of a monolithic deployment with the structural benefits of modularization, making it well-suited for the project scope and academic context.

In a modular monolith, the system is deployed as a single application but internally organized into well-defined, cohesive modules. Each module encapsulates a specific business capability and exposes its functionality through clearly defined interfaces. Direct access to internal implementation details of other modules is restricted, which helps reduce coupling and improve maintainability.

Rationale for Choosing a Modular Monolithic Architecture

The decision to adopt a modular monolithic architecture is driven by the following factors:

Clear Separation of Business Domains: The system naturally decomposes into distinct business domains such as curriculum management, course management, user management, and authorization. Modularization allows each domain to be implemented and evolved independently within the same application.

Maintainability and Modifiability: By enforcing module boundaries, changes to one functional area (e.g., curriculum versioning rules) have minimal impact on other modules. This directly supports the ASRs related to maintainability and modifiability.

Strong Data Consistency: As the system operates within a single deployment unit and database, transactional consistency across modules can be easily maintained. This is particularly important for academic data where correctness and integrity are critical.

Reduced Operational Complexity: Compared to microservices, a modular monolith avoids the overhead of distributed communication, service orchestration, and complex deployment pipelines, making it suitable for a small development team and limited project timeline.

Future Evolution Readiness: The modular structure provides a solid foundation for potential future evolution. If required, individual modules can later be extracted into independent services with minimal refactoring.

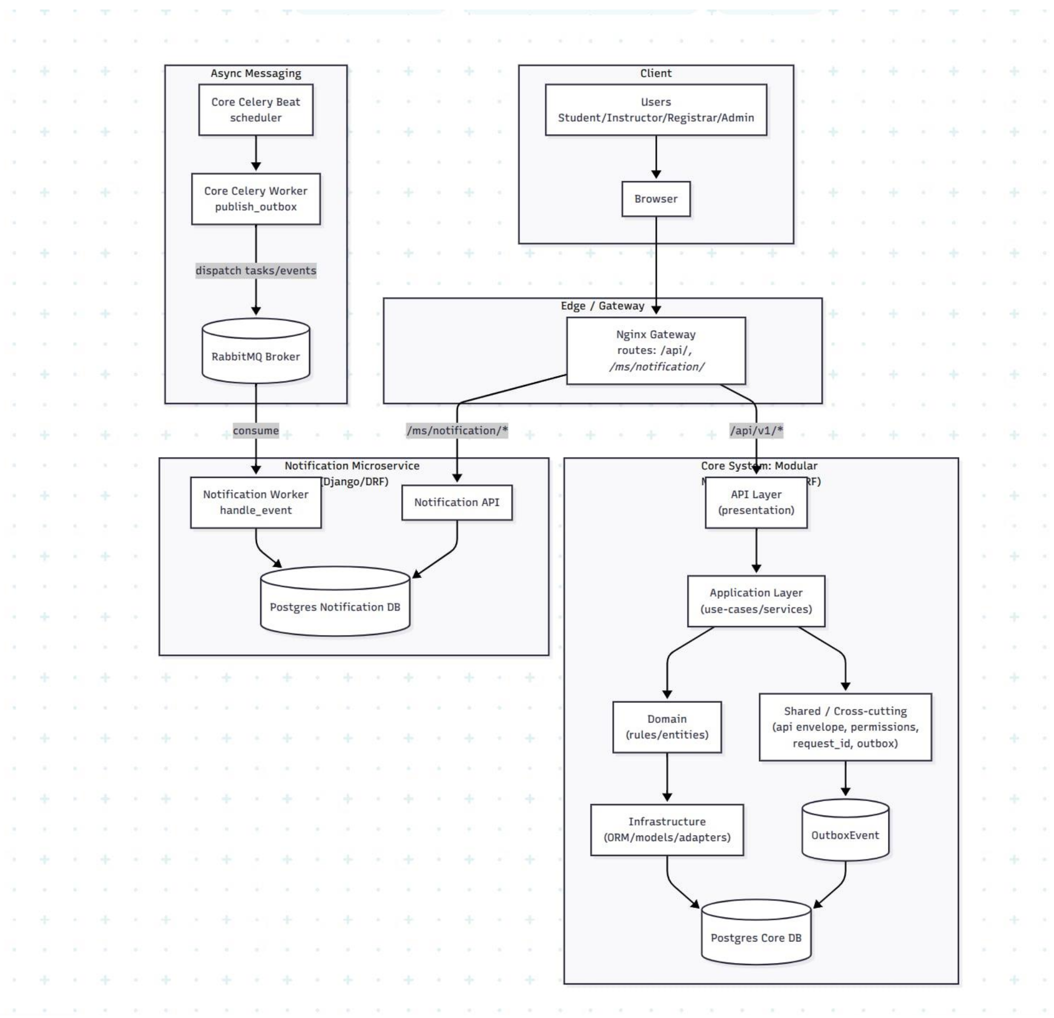
Modular Structure Overview

The system is organized into the following core modules:

- Curriculum Management Module: Responsible for managing training programs, curriculum structures, and curriculum versioning. This module encapsulates all business rules related to curriculum creation, updates, and publication.
- Course Management Module: Manages course information, prerequisites, credits, and relationships between courses and curricula.

- User and Role Management Module: Handles user accounts, roles, and permissions, ensuring role-based access control across the system.
- Reporting and Query Module: Provides read-only access and summary views of curriculum data for different stakeholders, such as students and lecturers.

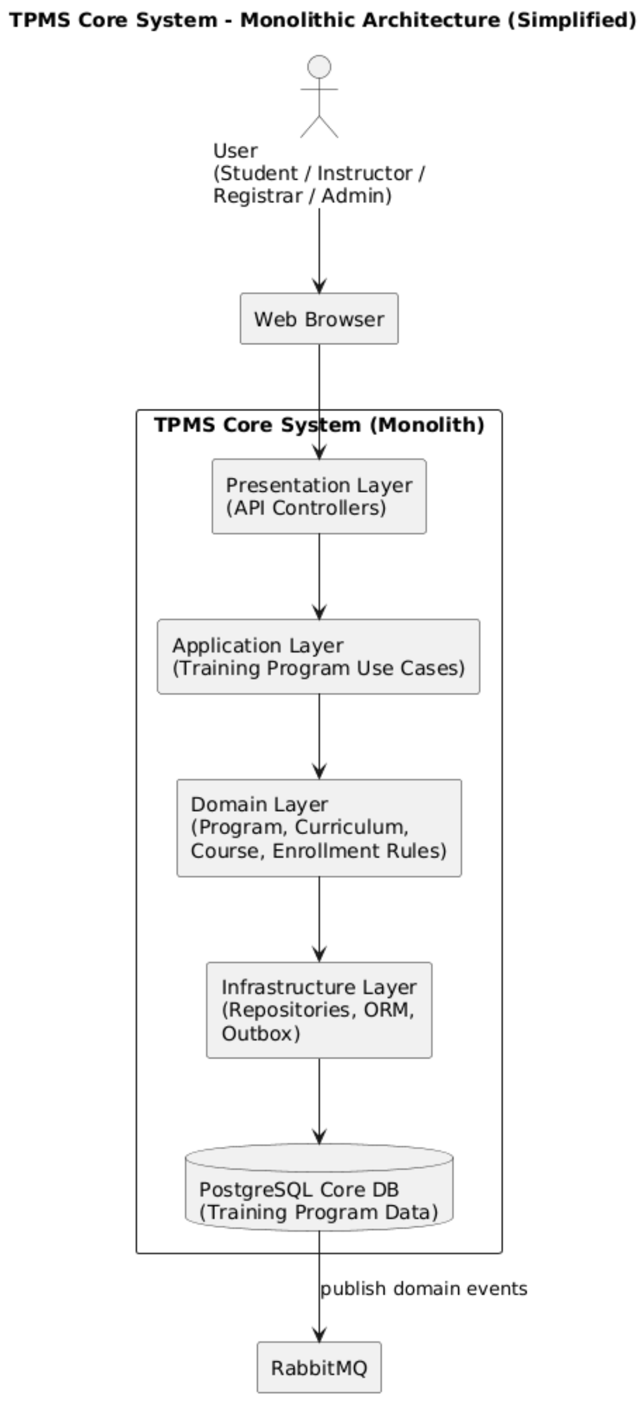
Each module communicates with others through well-defined interfaces at the application/service layer, while direct access to internal domain models or data repositories of another module is avoided.

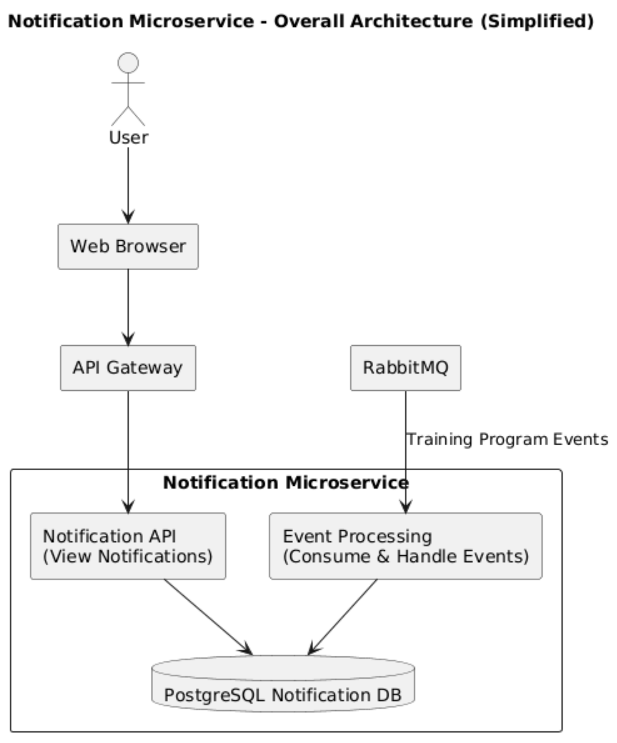


Async Messaging
Core Celery Beat scheduler
Core Celery Worker publish_outbox
dispatch tasks/events
RabbitMQ Broker
Client
Users Student/Instructor/Registrar/Admin
Browser
Edge / Gateway
Nginx Gateway routes: /api/, /ms/notification/
consume
/ms/notification/*
/api/v1/*
Notification Microservice (Django/DRF)
Notification Worker handle_event
Notification API
Postgres Notification DB
Core System: Modular
API Layer (presentation)
Application Layer (use-cases/services)
Domain (rules/entities)
Shared / Cross-cutting (api envelope, permissions, request_id, outbox)
Infrastructure (ORM/models/adapters)
OutboxEvent
Postgres Core DB

The overall system architecture is illustrated in the first diagram, which presents a high-level view of how clients, gateways, core services, and supporting components interact within the system. This diagram emphasizes the general interaction flow and the main architectural building blocks without focusing on internal implementation details.

Building upon this overview, the second diagram provides a more detailed architectural perspective by separating the system into two major parts: the core system implemented as a modular monolith and the notification functionality implemented as a dedicated microservice. While the core system encapsulates the primary business logic related to training program management within a layered modular monolithic structure, the notification microservice is designed to independently consume domain events and handle asynchronous notification processing. This architectural decomposition preserves a simple and maintainable core while selectively applying microservice principles to components that benefit from asynchronous processing and independent scalability.

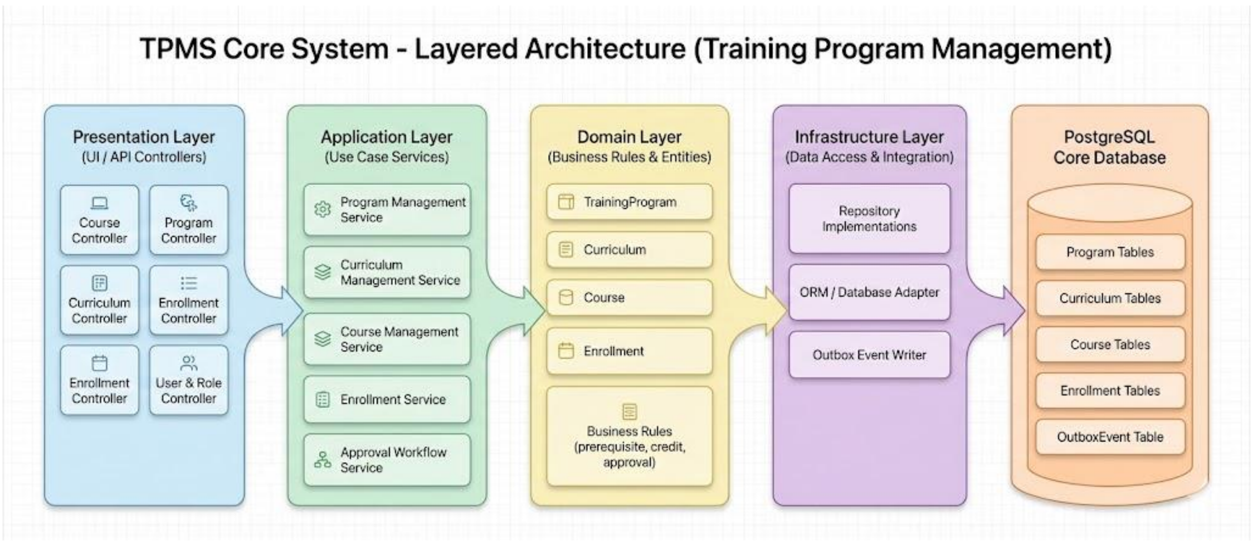

TPMS Core System - Monolithic Architecture (Simplified)
User (Student / Instructor / Registrar / Admin)
Web Browser
TPMS Core System (Monolith)
Presentation Layer (API Controllers)
Application Layer (Training Program Use Cases)
Domain Layer (Program, Curriculum, Course, Enrollment Rules)
Infrastructure Layer (Repositories, ORM, Outbox)
PostgreSQL Core DB (Training Program Data)
publish domain events
RabbitMQ


Notification Microservice - Overall Architecture (Simplified)
User
Web Browser
API Gateway
RabbitMQ
Training Program Events
Notification Microservice
Notification API (View Notifications)
Event Processing (Consume & Handle Events)
PostgreSQL Notification DB

Architectural Implications

The modular monolithic architecture enables the system to achieve a balance between architectural robustness and implementation simplicity. It supports the identified Architecturally Significant Requirements, particularly maintainability, data consistency, and security, while remaining aligned with the educational objectives of the Software Architecture course.

Layered Architecture Structure



3.2. System Context Diagram (C4 Level)

Level 1 : System Context Diagram

At Level 1-System Context Diagram, the focus is on defining the overall scope of the system and identifying the external actors and systems that interact with it, without going into internal technical details.

The Training Program Management System (TPMS) is the core system responsible for managing training programs and academic activities, including courses, enrollments, grades, and certificates.

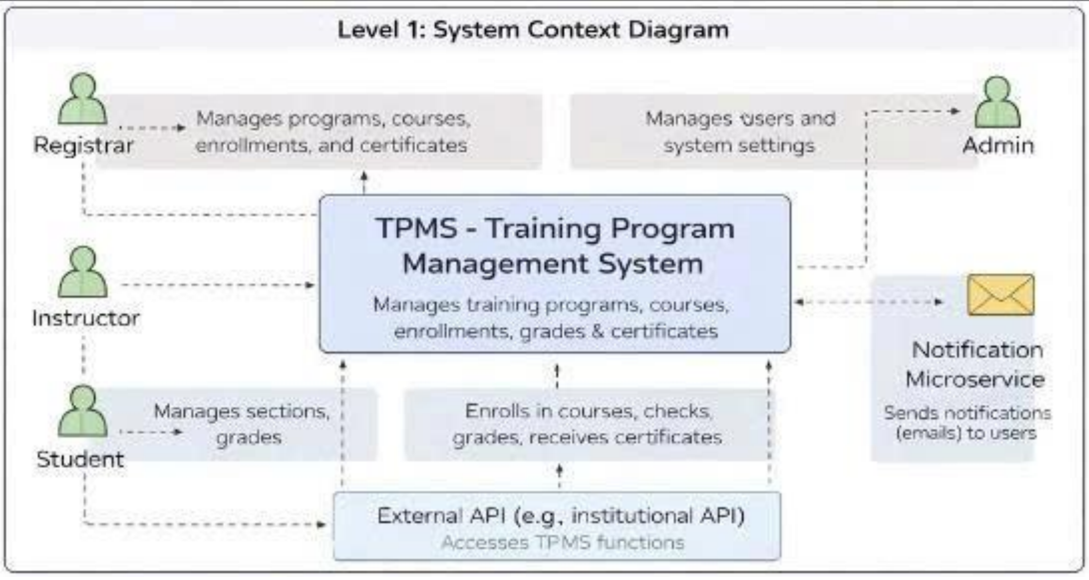
TPMS interacts with the following primary user roles:

- Students: use the system to enroll in courses, view schedules, check grades, andr eceive certificates.
- Instructors: manage course sections, update grades, and monitor student participation.

- Registrars: manage training programs, courses, enrollments, and certification processes.
- Administrators: manage users, roles, permissions, and system configurations.

In addition TPMS also communicates with external systems:

- Notification Microservice: Sends notifications (such as emails) to users when important events occur, including successful enrollments, grade updates, or certificate issuance.
- External APIs : Allow external systems to access or integrate with TPMS functionalities.


Level 1: System Context Diagram
Registrar
Manages programs, courses, enrollments, and certificates
Manages users and system settings
Admin
TPMS - Training Program Management System
Manages training programs, courses, enrollments, grades & certificates
Instructor
Notification Microservice
Sends notifications (emails) to users
Student
Manages sections, grades
Enrolls in courses, checks, grades, receives certificates
External API (e.g. institutional API)
Accesses TPMS functions

Level 2 : Container Diagram

The diagram illustrates the container level architecture of the Training Program Management System and shows how users interact with the system and how internal containers communicate with each other.

Users such as students and staff access the system through a web browser using the HTTPS protocol. The web browser loads the user interface of the system and forwards user actions to the React Frontend.

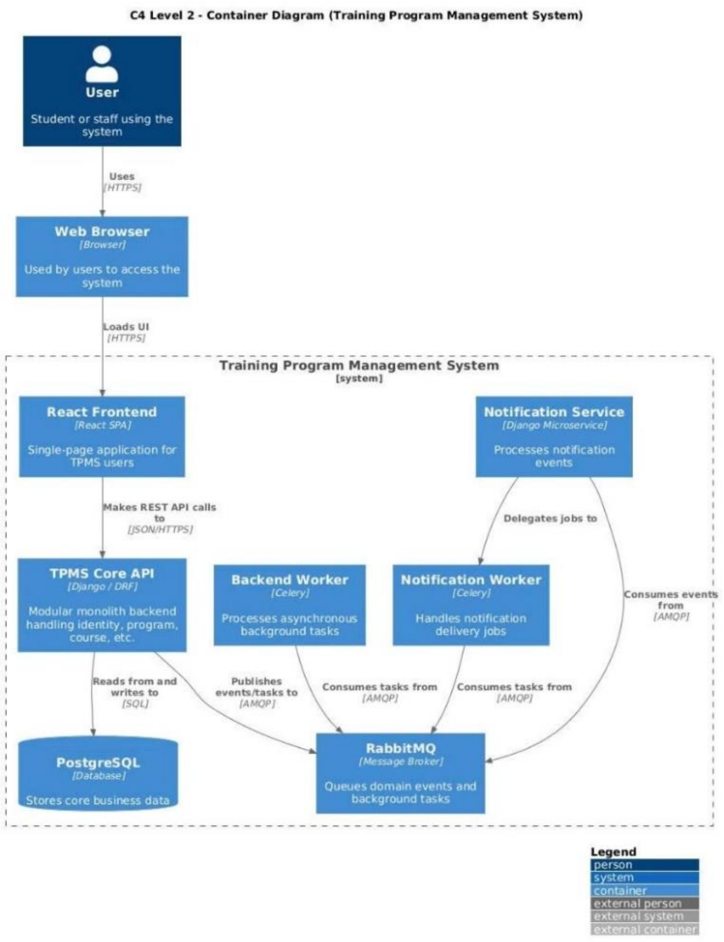
The React Frontend is responsible for presenting the user interface and handling user interactions. It sends REST API requests in JSON format to the TPMS Core API and receives responses to display to users.

The TPMS Core API acts as the central backend component of the system. It processes business logic related to identity management training programs courses and enrollments. The Core API reads from and writes to the PostgreSQL database where core business data is stored.

For asynchronous processing the Core API publishes tasks and domain events to RabbitMQ. The Backend Worker consumes tasks from RabbitMQ to process background jobs such as data preparation report generation and other long running operations.

The Notification Service consumes events from RabbitMQ to handle notification related logic. Notification delivery tasks are then processed by the Notification Worker which is responsible for sending notifications such as emails to users.

RabbitMQ serves as the message broker that enables asynchronous communication between the Core API Backend Worker and Notification Worker ensuring reliable task processing and system stability.

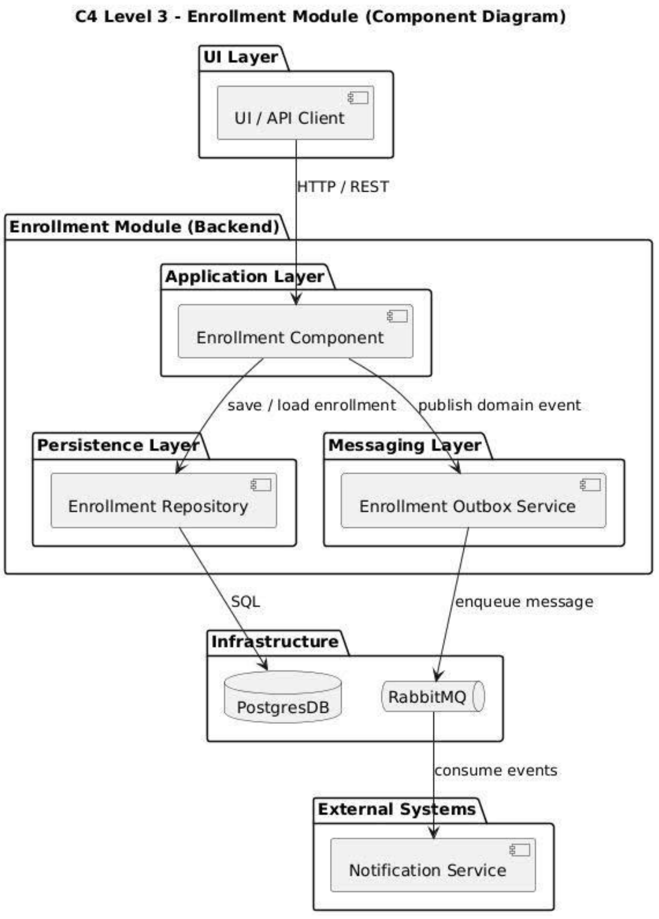

C4 Level 2 - Container Diagram (Training Program Management System)
User
Student or staff using the system
Uses [HTTPS]
Web Browser [Browser]
Used by users to access the system
Loads UI [HTTPS]
Training Program Management System [system]
React Frontend [React SPA]
Single-page application for TPMS users
Makes REST API calls to [JSON/HTTPS]
Notification Service [Django Microservice]
Processes notification events
Delegates jobs to
Consumes events from [AMQP]
TPMS Core API [Django / DRF]
Modular monolith backend handling identity, program, course, etc.
Backend Worker [Celery]
Processes asynchronous background tasks
Notification Worker [Celery]
Handles notification delivery jobs
Reads from and writes to [SQL]
Publishes events/tasks to [AMQP]
Consumes tasks from [AMQP]
Consumes tasks from [AMQP]
PostgreSQL [Database]
Stores core business data
RabbitMQ [Message Broker]
Queues domain events and background tasks
Legend
person
system
container
external person
external system
external container

Level 3 Component Diagram:

At Level 3 Component Diagram the diagram focuses on explaining how components inside the Enrollment Module of the Training Program Management System collaborate to implement course enrollment functionality. User requests originate from the UI layer where the UI or API client sends HTTP REST requests to the backend Enrollment Module.

Within the Enrollment Module the Application Layer contains the Enrollment Component which is responsible for executing enrollment business logic such as validating rules and processing enrollment requests. The Enrollment Component interacts with the Persistence Layer through the Enrollment Repository to save and load enrollment data from the PostgreSQL database. In parallel it communicates with the Messaging Layer through the Enrollment Outbox Service to publish domain events related to enrollment activities.

These domain events are enqueued to RabbitMQ which provides asynchronous messaging infrastructure. External systems such as the Notification Service consume these events to handle notification related processing. This component level design ensures clear separation of concerns improves maintainability and supports reliable event driven communication within the system



Level 4 Code / Class Level:

C4 Level 4 (Code / Class Level) provides the most detailed view of the system architecture by describing the internal code structure of the core system. At this level, the system is decomposed into concrete classes, services, and repositories that directly correspond to the source code implementation. The purpose of this level is to help developers understand how the system is organized internally and how different code-level components collaborate to realize business functionality.

The architecture follows a layered structure consisting of the presentation layer, application layer, domain layer, and infrastructure layer. Each layer has a clear responsibility and communicates with adjacent layers through well-defined interfaces. This separation ensures that business logic remains independent from technical concerns such as databases or messaging systems.

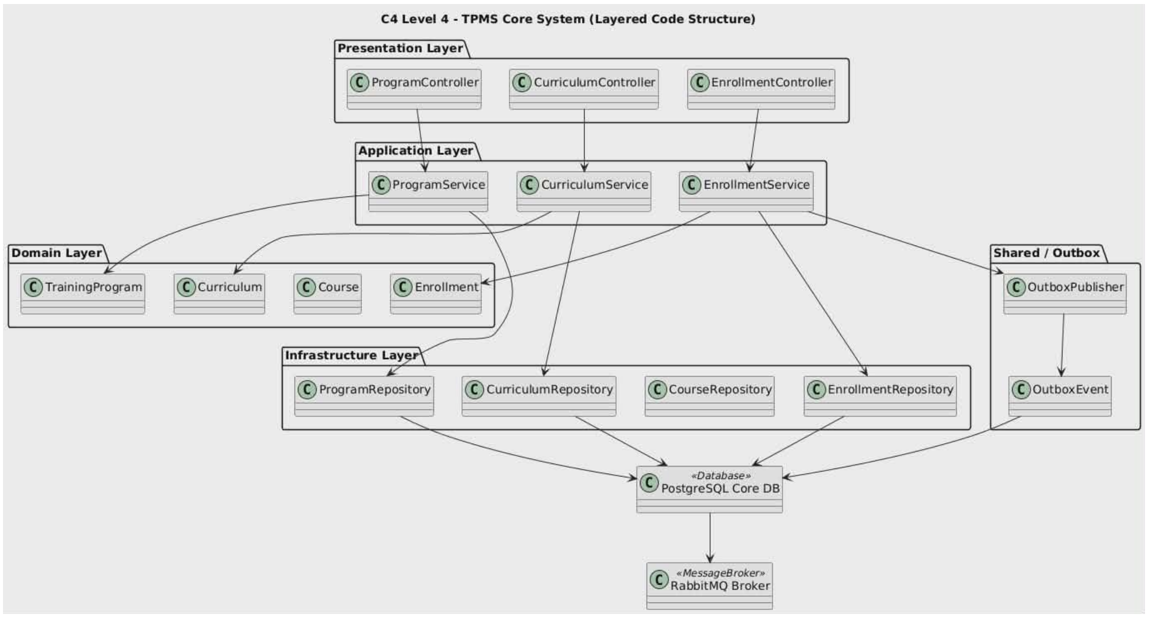
The presentation layer contains controllers that handle HTTP requests and responses. These controllers act as entry points to the system and delegate all business processing to the application layer, ensuring that no business logic is embedded in the API layer.

The application layer coordinates system use cases and workflows. Application services orchestrate interactions between domain objects, enforce use case boundaries, and manage transactional consistency. This layer serves as the bridge between external requests and core business logic.

The domain layer encapsulates the core business concepts of the system. It includes entities, value objects, and domain services that represent training programs, curricula, courses, enrollments, and related rules. All critical business rules and invariants are enforced within this layer to maintain consistency and correctness.

The infrastructure layer provides technical implementations for persistence, messaging, and integration with external systems. Repository implementations manage data access to the PostgreSQL database, while messaging components support asynchronous communication through RabbitMQ. Cross-cutting mechanisms such as the Outbox pattern are used to ensure reliable event publishing without coupling business logic to infrastructure concerns.

Overall, C4 Level 4 offers a code-oriented architectural view that bridges the gap between high-level design and actual implementation. It enables developers to quickly understand module boundaries, class responsibilities, and interaction patterns, thereby improving maintainability, testability, and long-term evolution of the Training Program Management System.



3.3. Technical Stack and Data Model

This section describes the technical stack and the data model used in the Curriculum Management System. The selected technologies and data design are aligned with the modular monolithic architecture, supporting maintainability, data consistency, and ease of development.

Technical Stack

The system is implemented as a modular monolithic application using the following technologies:

- Backend Framework: A server-side web framework is used to implement the core application logic following the modular monolith approach. Each functional module is structured as an independent package within the same codebase, enforcing clear module boundaries while sharing a common runtime.
- Presentation Layer: A web-based user interface is used to interact with the system. The presentation layer communicates with the application layer through well-defined

controllers or API endpoints, ensuring separation between user interaction and business logic.

- Database Management System: A relational database management system (RDBMS) is used to store persistent data. The choice of a relational database supports strong transactional consistency and structured data modeling, which are essential for managing academic information such as curricula and courses.
- Persistence Technology: An object-relational mapping (ORM) or data access abstraction is used to manage interactions between the application and the database. This approach simplifies data access logic and improves maintainability.
- Security and Authorization: Role-based access control is implemented at the application layer to ensure that only authorized users can perform sensitive operations, such as modifying or publishing curricula.

Data Model Overview

The data model is designed to reflect the core academic concepts of the system while supporting curriculum versioning and data integrity. The main entities include:

- Program: Represents a training program offered by the institution. A program may have multiple curriculum versions over time.
- CurriculumVersion: Represents a specific version of a training program. Each version defines a fixed set of courses and rules that apply to a group of students. Once published, a curriculum version is treated as immutable to ensure academic consistency.
- Course: Represents an individual course, including attributes such as course code, name, number of credits, and prerequisites.
- CurriculumCourse: Defines the relationship between a curriculum version and its associated courses, including additional attributes such as semester placement or required/elective status.
- User: Represents a system user, including administrators, training staff, lecturers, and students.
- Role: Defines user roles and permissions used to enforce role-based access control.

Modular Data Ownership

In line with the modular monolithic architecture, each module owns and manages its corresponding data and domain logic. For example, the Curriculum Management Module is responsible for curriculum and curriculum version entities, while the Course Management Module manages course-related data. Although all modules share the same physical database, logical ownership boundaries are maintained at the application level to reduce coupling and improve clarity.

Data Consistency and Integrity

Strong data consistency is ensured through centralized transaction management within the monolithic application. Business rules, such as preventing modifications to published curriculum versions or disallowing deletion of courses currently in use, are enforced at the service and domain layers. This approach simplifies consistency management and aligns with the system's Architecturally Significant Requirements.

3.4. Implementation Overview

This section provides an overview of how the Curriculum Management System is implemented based on the modular monolithic architecture. The implementation focuses on enforcing clear module boundaries, centralized business logic, and consistent data handling within a single deployable application.

Overall Application Structure

The system is implemented as a single monolithic application, internally organized into multiple independent modules. Each module represents a specific business capability and contains its own presentation interfaces (controllers), application services, domain models, and data access components.

Although all modules are deployed together, strict structural conventions are applied to prevent tight coupling and to preserve modularity. Interactions between modules occur only through exposed service interfaces rather than direct access to internal data or implementation details.

Module-Level Implementation

Each module follows a consistent internal structure:

- Controller / Interface Layer: Handles incoming requests from the user interface or API layer. This layer is responsible for input validation, request mapping, and invoking appropriate application services.
- Application / Service Layer: Implements use case logic and coordinates interactions between domain objects and repositories. This layer acts as the primary integration point between modules, ensuring that cross-module communication follows defined contracts.
- Domain Layer: Encapsulates core business rules and entities such as curriculum, curriculum version, and course. Domain logic enforces invariants, such as immutability of published curriculum versions and prerequisite constraints.
- Data Access Layer: Manages persistence operations using repositories or data access abstractions. This layer isolates database-specific logic and ensures consistency in data retrieval and storage.

Inter-Module Communication

Inter-module communication is implemented through explicit service interfaces defined at the application layer. For example, the Curriculum Management Module may query course information through the Course Management Module's service interface rather than accessing its data directly. This approach reduces coupling and supports future evolution of the system, including the potential extraction of modules into standalone services if required.

Transaction Management

Transaction boundaries are managed centrally within the monolithic application. Operations that involve multiple domain entities or modules are executed within a single transactional context, ensuring atomicity and consistency. This design simplifies error handling and aligns with the strong data consistency requirements of the system.

Security Enforcement

Security and authorization checks are implemented consistently at the service and controller layers. Role-based access control ensures that only authorized users can perform sensitive operations, such as modifying curricula or publishing new curriculum versions. Centralized security handling reduces duplication and improves maintainability.

Deployment Considerations

The application is deployed as a single unit, simplifying deployment, configuration, and runtime management. This deployment model is well-suited for the academic environment and project scope, while the internal modular structure ensures that the system remains maintainable and extensible.

4. Testing & Verification

4.1. Environment & Configuration

All testing activities were conducted on a local Ubuntu (x86_64) environment using a full-stack deployment based on Docker Compose. This approach ensured that all system components, including the core backend, the notification microservice, and the messaging infrastructure, were started and configured consistently. The system was launched using the infra/docker/docker-compose.dev.yml configuration file, while runtime parameters such as service ports, database credentials, and message broker settings were centrally managed through the infra/docker/.env file. This setup provided a reproducible and controlled environment for functional, integration, and stress testing.

Verification was carried out using a combination of complementary tools. Docker and Docker Compose were used to guarantee environment reproducibility across test runs. Postman Desktop was employed for manual and chained API workflow testing, with dedicated collections for the TPMS Core API and the TPMS Notification API. A Postman environment was configured to manage shared variables such as base URLs, authentication tokens, and entity identifiers. For performance and stress evaluation, Grafana k6 version 1.5.0 was used to execute load testing scenarios, including read-heavy and mixed realistic workloads. In addition, the web-based user interface was used to validate end-to-end user flows at the browser level, covering key scenarios such as navigating from the dashboard to grade entry, grade viewing, and notification delivery.

All tests were executed against a single base host, http://localhost, using path-based routing to distinguish system components. Requests targeting the core backend were routed through the /api/ namespace, including versioned endpoints such as /api/v1/, while requests to the notification microservice were routed through the /ms/notification/ path. The frontend single-page application was accessed via the root path. Authentication was verified using the /api/v1/auth/login/ endpoint, with access control enforced through bearer tokens included in the Authorization header. Role-based testing was performed to validate access policies for different user roles, including administrator, registrar, instructor, and student.

Test data readiness was ensured prior to execution by applying database migrations using manage.py migrate and loading predefined demo data through a custom seeding process. The seeded dataset included a sample academic term and representative courses, allowing realistic testing of core workflows. During this process, a custom seed command was evaluated, and it was observed that correct command naming and registration are required for Django's command discovery mechanism to function reliably, which is essential for maintaining a stable test setup.

Integration testing also covered the asynchronous notification pipeline. Domain events generated by the core system were first stored in an outbox and subsequently published through Celery Beat and worker processes to RabbitMQ. The notification microservice consumed these events and persisted the resulting messages, which were then validated through a dedicated API endpoint. This end-to-end verification confirmed the correctness of event propagation and asynchronous processing across system boundaries.

4.2. Functional Testing

Functional testing was conducted to verify that the Curriculum Management System correctly satisfies its specified functional requirements. The primary focus of this testing phase was to ensure that core business functions behave as expected under normal operating conditions and that business rules are consistently enforced across different user roles.

The testing covered key functional areas, including curriculum management operations such as creating, updating, and publishing curriculum versions, as well as course management activities such as adding courses and associating them with curricula. Role-based access control was also verified to ensure that administrative users, training staff, lecturers, and students were granted appropriate permissions according to their roles. In addition, read-only access for students was validated to confirm that academic information can be viewed without allowing unauthorized modifications.

Functional tests also verified correct system behavior when multiple users interacted with the system concurrently, focusing on logical correctness rather than performance metrics. These tests ensured that concurrent actions did not violate business rules or lead to inconsistent system states.

Overall, all functional requirements were successfully validated. The system consistently produced correct results across tested scenarios, demonstrating that the application logic and access control mechanisms function as intended.

4.3. Non-Functional Testing

Non-functional testing was conducted to evaluate the system's quality attributes beyond functional correctness. These tests focus on performance, scalability within project scope, security, and reliability, ensuring that the system meets its architecturally significant requirements.

Performance Testing

Performance testing was carried out to measure system responsiveness under concurrent user access. A stress test was performed by simulating 200 concurrent users interacting with the system, primarily executing common operations such as viewing curriculum information and submitting administrative requests.

Stress and Performance Testing

In addition to functional validation, a stress test was conducted to evaluate system behavior under concurrent usage. The test simulated 200 concurrent users performing common operations such as viewing curriculum data and submitting administrative requests.

The results indicate that the modular monolithic architecture is capable of handling the expected workload in an academic environment while maintaining responsive performance.

- Test Type: Stress / Load Testing
- Concurrent Users: 200
- Average Response Time: Less than 2 seconds
- Peak Load Behavior: Stable
- System Errors: None observed

The results demonstrate that the system maintains acceptable response times under expected academic usage conditions. This confirms that the modular monolithic architecture is sufficient to meet performance requirements without introducing unnecessary architectural complexity.

4.4. API Testing

API testing was conducted to confirm that the TPMS backend exposes correct, consistent, and secure REST endpoints supporting the main user flows, particularly authentication and core academic operations for Instructor and Student roles. The verification emphasized correct HTTP behavior (methods/status codes), response structure (IDs/status/timestamps), authorization enforcement (token-based access), correct request handling (query parameters and JSON bodies), and basic response-time sanity checks observed directly in Postman.

Test Environment and Tooling:

All API validation was performed using Postman Desktop, combining manual execution with lightweight scripted assertions. Requests were executed via Postman collections and chained using environment variables to make tests reproducible. Base URLs and dynamic IDs were parameterized (e.g., {{coreV1}}, {{sectionId}}). Tokens returned by login endpoints were captured and stored automatically into the Postman environment through post-response scripts (pm.environment.set(...)), enabling consistent authorization across subsequent requests.

Test Data and Preconditions:

The API workflow assumes the presence of at least one valid Instructor account and Student account, and that the database contains at least one Section accessible by the instructor. A valid section_id is required to query roster data and submit grades, and the selected section must have at least one enrolled student to validate the grading workflow.

Test Design Strategy:

The API tests followed a “happy-path” sequence to validate both endpoint correctness and cross-endpoint integrity. The chain begins with Instructor and Student login to obtain tokens, then proceeds with Instructor operations: listing available sections, selecting a section ID, retrieving the roster for that section, and submitting grades in bulk (draft). This approach ensures that IDs and entities returned from one endpoint are usable as valid inputs for downstream endpoints, confirming consistency across modules.

Test Cases and Results (Observed Evidence):

Endpoint / Method	Purpose	Observed Result
POST /auth/login/ (Instructor)	Authenticate instructor and store tokens for chained requests	200 OK; access/refresh tokens returned; Postman environment variables set via script; ~133 ms
POST /auth/login/ (Student)	Authenticate student and store token for student-authorized calls	200 OK; tokens returned; Postman environment variable set; ~112 ms
GET /section/sections/	Retrieve instructor-accessible sections	200 OK; section list returned with expected fields (IDs/codes/capacity/status/timestamps); ~18 ms
GET /enrollment/	Retrieve roster/enrollments for a specific section	200 OK; query param enabled; roster/enrollments returned with student identifiers; ~15 ms
POST /assessment/grades /bulk_enter/	Bulk submit grades as draft for a section	200 OK; grade_record_id returned; status: DRAFT; ~26 ms

Notes on Security and Reporting Hygiene:

Because login responses contain access/refresh tokens, screenshots used in the report should be masked or blurred to prevent credential leakage. Evidence should focus on endpoint identity, HTTP status, key response fields, environment-variable chaining ({{coreV1}}, {{sectionId}}), and the Postman scripts that persist tokens.

4.5. Validation Summary

The Testing & Verification phase aimed to demonstrate that the TPMS system is functionally correct and meets baseline operational quality under expected usage. Validation was structured into three layers: API Testing (correctness of individual endpoints and authentication enforcement), Functional Testing (end-to-end workflows such as instructor and student flows), and Non-Functional Testing (performance and stability under concurrent load using latency/error thresholds to identify capacity limits).

Evidence was collected primarily from two sources: Postman (manual/semiautomated API verification with screenshots showing request setup, authorization, and HTTP 200 results) and k6 (load/stress testing with execution summaries showing threshold evaluation, latency percentiles, and failure rates). Together, the screenshots confirm that the system works correctly under normal conditions and remains stable under moderate concurrency, while also showing clear saturation behavior under extreme load.

From the functional/API evidence, TPMS was proven to support the critical instructor/student workflow steps. Both Instructor and Student login succeeded (HTTP 200) and returned valid access/refresh tokens; Postman scripts stored these tokens to enable chained requests . Instructor data retrieval was validated through successful section listing and roster retrieval by section_id, confirming that authorization works beyond login and that academic structures (sections/enrollments) are queryable and filterable. Grade entry was validated via bulk submission: the API returned grade_record_id and status = DRAFT, confirming persistence and draft-state support for later publish workflows .

Non-functional validation used k6 to assess performance and stability through metrics such as p95 latency, http_req_failed, and check success rate. The read-heavy scenario showed strong baseline scalability: at moderate concurrency the system maintained 0% request failures and p95 latency within thresholds . The mixed realistic scenario (reads + writes) demonstrated two regimes: within the expected operating range, results remained stable with low failures and acceptable p95 latency . Under stress, p95 latency breached thresholds before error rate increased, and at higher saturation the system exhibited instability such as EOF/connection failures, increased http_req_failed, and failing checks . This indicates that the system becomes slow first (tail-latency collapse) and then transitions to hard failures when resources are exhausted.

Overall, the validation provides high confidence that: (1) core authentication and instructor workflows (sections, roster, grade entry) function correctly; (2) the system meets performance expectations under normal-to-moderate concurrency; and (3) the system has a measurable capacity limit where responsiveness degrades and reliability drops under extreme load. To strengthen completeness, the report should note gaps such as limited negative testing (401/403 for invalid tokens, invalid payload validation errors, invalid section_id) and clarify that absolute performance numbers may differ in production, although the observed saturation pattern is still meaningful.

5. Conclusion & Reflection

This project presented the design and implementation of a Curriculum Management System based on a modular monolithic architecture. The primary objective was not only to build a functional system but also to apply software architecture principles in a structured and justifiable manner.

Starting from business requirements, the project identified key quality attributes and architecturally significant requirements (ASRs) that guided architectural decisions. The selected modular monolithic architecture successfully balances simplicity, maintainability, and data consistency, making it suitable for the scope of the project and the academic environment. Through systematic testing and validation, the system demonstrated that it meets both functional and non-functional requirements, particularly in terms of performance, security, and reliability.

Overall, the project achieved its learning objectives by translating architectural concepts into a practical system design and implementation.

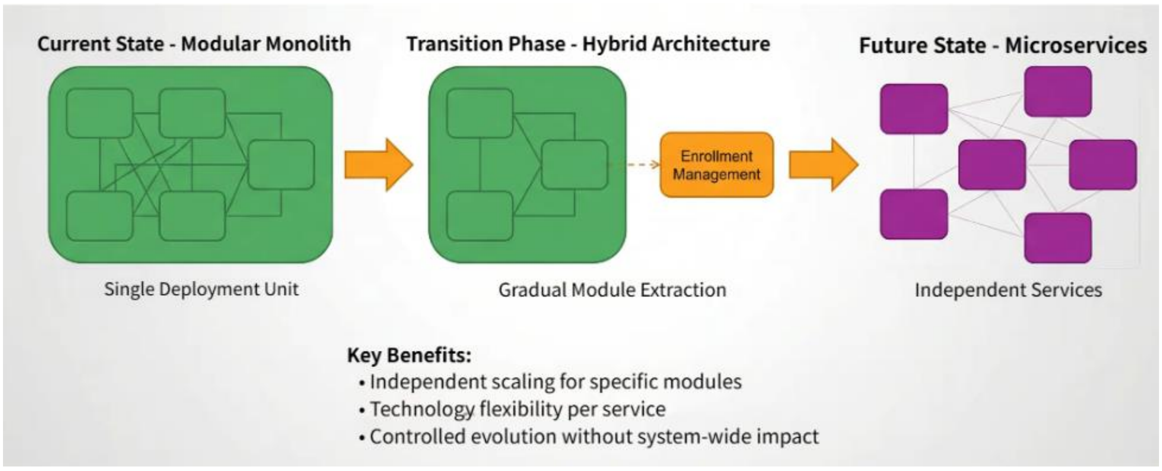
5.1. Lessons Learned

One of the most important lessons learned from this project is that software architecture should be driven by requirements rather than technology trends. By carefully analyzing functional requirements and quality attributes, the team was able to justify the adoption of a modular monolithic architecture instead of more complex distributed approaches. This reinforced the understanding that an architecture is considered effective only when it aligns with the actual needs and constraints of the system.

The project also highlighted the value of modularity within a monolithic system. Although the application is deployed as a single unit, enforcing clear module boundaries significantly improved maintainability and reduced coupling between different parts of the system. This experience demonstrated that monolithic architectures, when designed properly, can still achieve high levels of structural clarity and extensibility.

Another key lesson was the importance of early architectural modeling. Techniques such as use case modeling, C4 diagrams, and ASR analysis helped clarify system scope and design decisions before implementation began. These models served as a common reference point throughout the development process, reducing misunderstandings and guiding consistent design choices. Additionally, systematic testing played a crucial role in validating architectural assumptions, particularly regarding performance and reliability under concurrent user access.

5.2. Future Improvements


Current State - Modular Monolith
Transition Phase - Hybrid Architecture
Future State - Microservices
Enrollment Management
Single Deployment Unit
Gradual Module Extraction
Independent Services
Key Benefits:
• Independent scaling for specific modules
• Technology flexibility per service
• Controlled evolution without system-wide impact

Although the current system satisfies the project requirements, several improvements can be considered for future development. From a scalability perspective, performance can be further enhanced through database optimization, caching mechanisms, or resource tuning as user demand increases. If the system grows significantly in size or complexity, the existing modular structure provides a solid foundation for gradually extracting selected modules into independent services.

Security can also be strengthened by introducing advanced mechanisms such as audit logging, finer-grained authorization policies, and enhanced authentication methods. These improvements would further protect sensitive academic data and support stricter governance requirements. In addition, the user experience could be improved by refining the user interface, enhancing search and navigation features, and providing better visualization of curriculum structures for different stakeholders.

Finally, future work may include the adoption of automated testing and monitoring tools to improve long-term reliability and maintainability. Continuous testing and system monitoring would allow early detection of issues and support more efficient system evolution.