Alfaisal University - College of Engineering and Advanced Computing
Software Engineering Department
SE 495: Software Engineering Capstone Project I
Task 3 – Design Document
Fall 2025 - Due Date: Nov 13, 2025 at 11:59 pm (midnight)

Instructions and guidelines:

1. This is a team-based task. You need to work with your team members to complete this task.
2. Use the given template, in MS-Word format, to complete the requirements of this task.
3. Upload your completed document file, in PDF format, on Moodle.
4. One submission is needed by each team.

Course Learning Outcomes (CLO)	
CLO1.1	Demonstrate the ability to identify, formulate, and solve engineering problems, particularly in the areas of software requirements, architecture, and design.
CLO2.2	Design a system, component, or process to meet desired needs within realistic constraints such as economic, environmental, social, political, ethical, health and safety, manufacturability, and sustainability

Task 3 - Design Document

Use the following template to prepare your project design document.

Note: Not all sections in this template are mandatory. Supervisors may advise students to omit sections that are not applicable to their specific project.

Task:	Design Document
Project Title:	Intelligent Academic Advisory System (IAAS)
Team Members' Names:	1) Feras Alkahtani 2) Abdulaziz Albaz 3) Mohammed Alhajri
Supervisor Name:	Professor Anis Koubaa
Date:	11/20/2025

Table of Contents

1. System Design 3

1.1 System Decomposition 3

1.2 Concurrency Identification 3

1.3 Deployment View 3

1.3.1 System Performance 4

1.3.2 Connectivity 4

1.3.3 Network Architecture 4

1.4 Data Management 5

1.5 Software Control Implementation 5

1.5.1 External Control Flow Between Subsystems 5

1.5.2 Concurrent Control 5

1.5.3 User Interface 5

1.6 Boundary Conditions 6

1.6.1 Initialization 6

1.6.2 Termination 6

1.6.3 Failure 6

2. Object Oriented Design 6

2.1 Class Diagrams 6

2.2 Object Diagrams 6

2.3 Packages 7

2.3.1 Package Diagrams 7

2.3.2 Software Package Documentation 7

2.3.3 Design Constraints 7

2.3.4 Design Rationale and Alternatives 7

1. System Design

1.1 System Decomposition

The Intelligent Academic Advisory System (IAAS) is decomposed into four major subsystems. The decomposition follows a layered client–server architecture, providing clear separation between UI, API, business logic, and data management.

Subsystem 1: Chat Advisor Subsystem

Responsibilities:

- Handle student natural-language questions
- Generate course recommendations
- Validate prerequisites
- Provide explanations
- Log interactions
- Switch to rule-based fallback when LLM unavailable

Components:

- *Planned class: LLMServiceAdapter* (OpenAI client)
- *Intended component: RuleBasedAdvisor*
- *AdvisorEngine*
- *ResponseBuilder*
- *ChatLogger*

Subsystem 2: Degree Progress Visualization Subsystem

Responsibilities:

- Calculate completion status
- Build prerequisite paths
- Determine eligibility for next courses
- Produce JSON structures for front-end visualization

Primary Outputs:

- Degree progress tree
- Remaining requirements summary
- Eligible future courses

Subsystem 3: Dataset Management & Validation Subsystem

Used by administrators only.

Provides:

- Schema validation
- Referential integrity enforcement
- Dataset versioning
- Bulk import functionality
- Validation reports

Subsystem 4: Backend Core Services Subsystem

Contains all cross-cutting backend logic:

- REST APIs
- Authentication
- Logging & quota enforcement
- Repository-based DB access
- Prerequisite graph construction
- System configuration loader

Architectural Style:

- **Client–Server:** Browser interacts with backend through REST APIs.
- **Layered:** UI → API → Services → Repositories → SQLite.
- **Modular:** Independent subsystems minimize coupling.

1.2 Concurrency Identification

IAAS must support concurrent usage by many students and administrators.

The Sources of Concurrency

- Multiple student chat queries
- Multiple degree progress requests
- Admin dataset upload
- Background dataset validation
- Background quota monitor
- Continuous logging

Concurrency Management

- FastAPI will be implemented because it uses asynchronous workers to handle requests in parallel
- SQLite may be used to enforce consistency using transactional locking
- Append-only logging avoids write conflicts
- Dataset version activation is atomic
- Rule-based fallback prevents system blocking during LLM delays

---

Shared Resources

- SQLite database
- Active dataset version
- Log table
- Token quota counters

Race Condition Mitigation

- Transactions
- Atomic version switching
- Serialized validation pipeline
- Lock-safe database writes

1.3 Deployment View

IAAS' planned architecture is over five logical nodes:

Node 1 — Student Browser

Runs UI for Chat Advisor and Degree Progress.

Node 2 — Admin Browser

Runs UI for dataset upload, logs, and administrative operations.

Node 3 — Backend FastAPI Server

Executes:

- Business logic services
- LLM queries
- Rule-based advising
- Dataset validation
- Degree progress computation
- Logging
- Authentication

Node 4 — SQLite Database

Local storage for all persistent system data.

Node 5 — OpenAI Cloud API

Provides LLM model used by Chat Advisor.

Communication Links

- Browsers ↔ Backend: HTTPS (JSON)
- Backend ↔ SQLite: Local DB connection
- Backend ↔ OpenAI: HTTPS authenticated

1.3.1 System Performance

Performance Requirements (as targets)

- Chat Advisor response time ≤ **2 seconds**
- Degree progress computation ≤ **2 seconds**
- Dataset upload + validation ≤ **60 seconds**

Performance Strategies

- In-memory prerequisite graph
- Optimized SQL queries
- Lightweight SQLite database
- Local caching of static course
- Deterministic rule-based fallback
- Minimal API payloads

1.3.1.1 General System Performance

The IAAS system is designed to provide fast, stable, and responsive academic advising services to students and administrators. The overall performance requirements focus on maintaining low response times, supporting multiple concurrent sessions, and ensuring smooth data processing across all core subsystems.

Expected Response Time and Throughput

- **Chat Advisor:** Must respond to typical student queries within **2 seconds** under normal load.
- **Degree Progress Evaluation:** Must compute the student's progress and eligibility tree within **2 seconds**, even with large prerequisite chains.
- **Dataset Validation:** Full validation and activation should complete within **60 seconds**, supporting datasets up to several thousand records.
- **Throughput:** The backend must handle **multiple concurrent API requests** without noticeable performance degradation, allowing many students to access the advisor in the same period.

Techniques Used to Improve Performance

To meet these measurable performance targets, IAAS incorporates several architectural optimizations:

- **In-Memory Prerequisite Graph:**
  The prerequisite relationships are preloaded at system startup, dramatically improving advising speed.
- **Async Processing via FastAPI:**
  FastAPI's asynchronous request handling enables concurrent student interactions without blocking or waiting on slow I/O events.
- **SQLite for Fast Read-Heavy Workloads:**
  The system performs many more reads than writes. SQLite excels at high-speed read performance, ensuring rapid database access.
- **Minimal Payload JSON APIs:**
  Only essential data is exchanged with the browser, reducing network latency and improving UI responsiveness.
- **Rule-Based Fallback Engine:**
  When LLM response times exceed limit, the system falls back to a deterministic advisor, ensuring consistent response times regardless of cloud latency.

- **Efficient Logging and Batching:**
  Logging operations are lightweight and append-only, minimizing write contention.

Identified Bottlenecks and Mitigation Strategies

The following potential bottlenecks were identified during system analysis:

- **LLM Latency (Cloud Response Time):**
  Cloud-based LLM responses can exceed 3–4 seconds during high load.
  Mitigation: Automatic switch to rule-based advisor, guaranteeing stable advising performance.
- **SQLite Single-Writer Limitation:**
  SQLite allows only one writer at a time, potentially bottlenecking logging and dataset uploads.
  Mitigation: Append-only logs, bulk-write batching, and strict limiting of write-heavy operations to admin-only workflows.
- **Dataset Validation Complexity:**
  Large datasets increase validation time.
  Mitigation: Validation is executed as a background process, preventing UI blocking and allowing the system to remain responsive during validation.

Overall, the system design emphasizes responsiveness, predictable performance under concurrency, and resilience against external latency fluctuations.

1.3.1.2 Input/Output Performance

The IAAS system is designed to handle input and output operations efficiently to ensure responsive user interactions, fast dataset processing, and reliable communication with external services such as the OpenAI API. Input/output performance focuses on managing user-generated requests, data retrieval from the database, dataset uploads, and outbound communication with the LLM service.

Type and Volume of Expected Inputs/Outputs

User Inputs

- Student text queries to the Chat Advisor
- Requests for academic progress visualization
- Admin uploads of synthetic datasets (CSV format, typically **1–3 MB**)

System Outputs

- JSON responses for Chat Advisor results
- JSON-structured degree progress trees
- Validation reports for dataset uploads
- Logged advisor interactions for audit and analytics

External Service Inputs/Outputs

- Outbound requests to the OpenAI API for LLM-based advising
- Inbound LLM-generated responses (typically < 10 KB)

The expected input/output volume is moderate, primarily consisting of lightweight JSON objects and small CSV files.

Mechanisms for Efficient Data Transfer

To maximize input/output performance, IAAS uses several design techniques:

1. **Lightweight JSON-Based Communication**

All REST API endpoints use compact JSON structures that minimize serialization and parsing overhead, improving transfer speed between the UI and backend.

2. **Batch Processing for Dataset Uploads**

Admin dataset uploads are handled through:

- **Batch validation** (schema + referential integrity)
- **Bulk import operations** to reduce transaction overhead
- Preprocessing steps to ensure efficient writes

This significantly reduces overhead when handling large CSV datasets.

3. **Buffered File Upload Handling**

Dataset files are streamed into memory buffers rather than loading entire files into long-lived memory, allowing efficient handling of multi-MB CSVs.

4. **Efficient Read-Heavy Access Pattern**

The database workload is primarily read-heavy:

- Course information
- Prerequisite data
- Student progress data

SQLite is optimized for rapid read performance, allowing the system to serve multiple users quickly.

5. **Asynchronous LLM Communication**

Requests to the OpenAI API use asynchronous I/O:

- Prevents blocking of worker threads
- Allows parallel LLM request handling
- Ensures high responsiveness even during network delays

Handling of Large Files, Real-Time Updates, and Network Delays
Large File Handling

- Dataset uploads are processed incrementally, avoiding full in-memory file loading
- Bulk imports reduce the number of write operations to SQLite
- Validation is executed before import to avoid partial updates

IAAS can comfortably handle datasets containing **thousands of rows**.

Real-Time Updates

The system manages live interactions with students:

- Chat Advisor responses are returned immediately once LLM result or fallback result is ready
- Degree progress views are computed on-demand using preloaded prerequisite graphs

Because of efficient in-memory structures, real-time responsiveness is maintained.

Network Delay Mitigation

Network delays primarily impact the LLM service. The design counteracts this through:

- **Timeout thresholds**: slow LLM calls trigger fallback mode
- **Non-blocking async architecture**: workers stay free to serve other users
- **Minimal payload size**: reduces transfer time over slow networks

This ensures that even with unstable internet conditions or high LLM latency, the system continues to function reliably.

1.3.1.3 Processor Allocation

The IAAS system distributes processing tasks across several computing units to achieve optimal performance, scalability, and reliability. Each processor or logical compute node executes specific software components that match its capabilities and operational role.

Processors / Computing Units Involved

1. **Client-Side Processor (Student or Admin Browser)**
    - Runs on user devices (laptops, desktops, or mobile browsers)
    - Executes lightweight UI rendering via HTML/CSS/JavaScript
    - Performs no heavy computation
2. **Backend Application Server Processor (FastAPI Server)**
    - Main compute node handling API requests
    - Executes business logic and computational tasks
    - Manages interactions with the database and external LLM API
3. **Database Processor (SQLite Engine)**
    - Embedded within the backend server machine
    - Processes SQL queries
    - Ensures ACID-compliant transactions
4. **External Cloud Processor (OpenAI LLM Service)**
    - Hosted externally by OpenAI
    - Executes natural language processing and text generation
    - Handles the heaviest computational workload in the advising pipeline

---

Software Components Running on Each Processor

1. Client (Browser CPU)

Runs:

- User Interface (UI)
- Request formatting and form submissions

- JavaScript-based dynamic rendering
- Degree progress visualization via lightweight charting libraries

Does *not* perform:

- Advising calculations
- Dataset validation
- Database queries

This keeps the client fast and compatible with low-end devices.

---

2. Backend Server Processor

Executes the core of the system:

API Layer

- Chat request handling
- Progress request handling
- Admin upload endpoints
- Authentication

Service Layer

- AdvisorEngine
- LLMServiceAdapter
- RuleBasedAdvisor
- DegreeProgressService
- DatasetValidator
- BulkImporter
- QuotaMonitor

Data Access Layer

- Repository classes for students, courses, prerequisites, logs, and dataset versions

This processor performs *all* logical computation except LLM-based advising.

---

3. SQLite Database Processor

Handles:

- Query execution
- Constraint checking
- Transactional writes
- Read-optimized operations for progress checks

SQLite runs as part of the backend process but uses a separate engine within the host machine.

---

4. External Cloud Processor (OpenAI)

Handles the most CPU-intensive task:

- Natural language inference
- Context interpretation
- Response generation for the Chat Advisor

This offloads the majority of computational cost from your system to a scalable cloud service.

Justification for Allocation

Performance

- Heavy computation (LLM inference) is offloaded to OpenAI, reducing backend CPU load.
- Backend handles business logic efficiently using asynchronous non-blocking execution.

- Database operations remain fast due to SQLite's low overhead and optimized read performance.

Scalability

- LLM processing scales automatically with OpenAI's cloud infrastructure.
- Backend server handles large numbers of concurrent students due to FastAPI's async model.
- Since clients perform minimal work, many users can access the system without local hardware constraints.

Fault Tolerance

- Rule-based fallback advisor ensures advising continues even if the OpenAI processor is unavailable.
- SQLite transactions prevent partial writes during failures.
- UI remains responsive even when backend temporarily slows, due to asynchronous request handling.

1.3.1.4 Memory Allocation

The IAAS system is designed to use memory efficiently across the client, backend server, database engine, and external cloud components. Memory allocation focuses on minimizing resource usage, ensuring stable performance during concurrent interactions, and preventing memory-related bottlenecks during dataset upload or LLM communication.

Estimated Memory Usage of Key Components (Estimates as backend memory has not been measured)

1. Client Browser (Student/Admin UI)

- Memory usage: **5–20 MB**
- Contains UI rendering, JavaScript logic, and lightweight visualizations.
- No heavy data structures stored locally.

2. Backend FastAPI Server

Estimated memory usage breakdown:

Component	Estimated Memory
FastAPI application	~30–50 MB
AdvisorEngine + Rule-Based Advisor	~5–10 MB
In-memory prerequisite graph	**~2–5 MB** (depending on dataset size)
Cached course metadata	~1–2 MB
Active request buffers	~2–10 MB depending on load

Total server memory footprint: **~50–90 MB**, easily supported by basic VPS or on-prem machines.

3. SQLite Database Engine

- Uses **1–5 MB** of memory for internal caching and query execution.
- Database file stored on disk is typically **<5 MB** for synthetic datasets.

4. OpenAI LLM Service

- External; memory usage is not local.
- Offloading inference drastically reduces required memory on your backend.

Memory Management and Optimization Strategies

1. In-Memory Prerequisite Graph (Efficient Lookup)

- Loaded once at system initialization.
- Enables constant-time (O1) or linear-time (O(V+E)) eligibility checks.
- Prevents repeated database queries during advising.

2. Lazy Loading of Non-Critical Components

- Validation and import modules load additional data only when triggered by admin actions.
- Reduces baseline memory footprint during normal student usage.

3. Minimal Data Retention

- Chat logs stored in the database only; not kept in long-term memory.
- JSON responses are generated and immediately released after sending.

4. Python Garbage Collection

- Active by default.
- Automatically frees temporary objects used during LLM communication, validation, or import steps.

5. Buffered File Streaming for Dataset Upload

- Admin-uploaded CSV files are not fully loaded into memory.
- They are streamed in chunks to avoid memory spikes for large datasets.

System Behavior Under Memory Constraints

1. Graceful Degradation

If memory availability decreases:

- Python garbage collector increases cleanup frequency.
- Cached objects (course metadata, progress trees) may be cleared and reloaded on demand.
- Low-memory warnings can trigger reduced in-memory caching.

2. Dataset Validation Safety

- Validation performs incremental line-by-line processing if memory is tight.
- Prevents "out of memory" crashes during large dataset uploads.

3. Stable Student-Facing Performance

- Core advising tasks use minimal memory.
- Even under heavy load, memory-constrained environments still support:
  - Rule-based advisor
  - Progress computation
  - Logging

4. LLM Offloading Protects Local Memory

- Since inference happens in the cloud, peak memory consumption does not scale with query complexity.
- Ensures the backend remains stable even during rapid or repeated Chat Advisor usage.

1.3.2 Connectivity

The IAAS system uses a web-based client–server communication model with an external cloud LLM service. Connectivity focuses on how the student and admin interfaces interact with the backend server, how the backend interacts with the database, and how the system connects securely to the OpenAI API.

Type of Communication

- **Client–Server Communication**
  - Student and Admin browsers act as clients.
  - The FastAPI backend acts as the server.
  - All user interactions (chat queries, progress requests, dataset uploads) are sent as HTTP requests from the client to the server.

- **Local In-Process Communication**
  - The backend communicates with the SQLite database through a local file-based driver.
  - Service-layer components (AdvisorEngine, DegreeProgressService, DatasetValidator, etc.) communicate via direct function/method calls.
- **Cloud Service Communication**
  - The backend communicates with the external OpenAI API over the public internet.
  - This is a server-to-server pattern, with the backend as the client of the LLM service.

There is **no peer-to-peer** communication and **no IoT network** involved.

Protocols and Data Formats

Protocols

- **HTTP/HTTPS**
  - Used between browsers and the backend API.
  - All production communication is intended to be over **HTTPS** to protect credentials and payloads.
- **HTTPS (TLS-secured HTTP)**
  - Used between the backend and the OpenAI API.
  - Ensures confidentiality and integrity of prompts and responses.
- **SQLite Local File Access**
  - The backend uses the SQLite engine via local file access (no network protocol).

Data Formats

- **JSON (JavaScript Object Notation)**
  - Primary data format for all REST API requests and responses:
    - Chat messages
    - Degree progress data
    - Validation reports
  - Lightweight and easily parsed on both browser and server.
- **CSV (Comma-Separated Values)**
  - Used for admin dataset uploads (course catalog, prerequisites, student data).
  - Parsed on the backend before being imported into the database.
- **Plain Text / Structured Logs**
  - Log entries are stored as structured text or JSON-like entries in the database (chatbot_logs table).

There is **no XML** and **no binary serialization** in the current design.

Middleware, APIs, and Supporting Components

- **FastAPI Framework**
  - Acts as the main API framework and routing layer.
  - Parses incoming HTTP requests and serializes responses as JSON.
  - Provides validation of request payloads and automatic documentation.
- **OpenAI Client / HTTP Library**
  - A small infrastructure component wraps HTTP calls to the OpenAI API.
  - Handles authentication headers (e.g., API key), timeouts, and error handling.
- **ORM / Database Access Layer (Repositories)**

    - Repository classes encapsulate SQL queries and data mappings.
    - They act as an internal "data access API" used by the services.
- **No Message Queue or Event Bus (for now)**
    - All communication is synchronous request–response.
    - Background tasks (quota monitoring, dataset validation) are executed using in-process workers or scheduled tasks, not via external message queues.

Textual UML Deployment Diagram (Connectivity View)
(Your template asks for a UML Deployment Diagram. Since you chose no images, this is a UML-style textual description you can also convert into a diagram later if needed.)
Nodes and Artifacts:

1. **Node: Student/Administrator Device (Browser)**
    - *Artifact:* IAAS_Web_UI (HTML/CSS/JS)
    - Responsibilities:
        - Render Chat Advisor and Degree Progress pages
        - Send HTTP/HTTPS requests to backend
        - Receive JSON responses and update UI
2. **Node: Application Server**
    - *Artifact:* IAAS_Backend (FastAPI Application)
    - Deployed Components:
        - API Layer (controllers for /api/chat, /api/progress, /admin/upload, /admin/logs)
        - AdvisorEngine
        - RuleBasedAdvisor
        - LLMServiceAdapter
        - DegreeProgressService
        - DatasetValidator
        - BulkImporter
        - QuotaMonitor
        - AuthService
        - Repository Layer
3. **Node: Database (Local to Backend)**
    - *Artifact:* iaas.db (SQLite database file)
    - Contains:
        - students, courses, prerequisites, enrollments
        - chatbot_logs, dataset_versions, admin_users
4. **Node: OpenAI Cloud Service**
    - *Artifact:* OpenAI_LLM_API
    - Provides:
        - GPT-4o-mini model for natural language advising

1.3.3 Network Architecture

The IAAS network architecture is a **cloud-assisted client–server model** in which student and admin clients connect to a central backend server, which in turn communicates with a local database and an external LLM service (OpenAI). The design emphasizes low latency for interactive advising, secure communication, and simplicity suitable for a university project environment.

Network Nodes

1. **Client Nodes (Students & Admins)**
    - Devices: Laptops, desktops, or mobile devices.
    - Software: Web browser (Chrome, Edge, etc.).
    - Role:
        - Students: access Chat Advisor and Degree Progress.
        - Admins: upload datasets, review logs, monitor usage.
2. **Application Server Node (Backend Server)**
    - Runs the FastAPI backend application.
    - Exposes REST APIs for all system functionality.
    - Handles business logic, validation, and DB access.
3. **Database Node (Local Database on Server)**
    - SQLite database file hosted on the same physical/virtual machine as the backend.
    - Stores student records, course catalog, prerequisites, enrollments, logs, and dataset versions.
4. **Cloud LLM Node (OpenAI Service)**
    - Hosted by OpenAI in the cloud.
    - Processes natural language prompts and returns model-generated responses.
5. **Network Infrastructure Nodes (Logical)**
    - University LAN / campus network
    - Internet routers and gateways between the backend server and the OpenAI cloud

---

Connections (Wired/Wireless, LAN/Cloud)

- **Client → Application Server**
    - Typically over **wired or wireless LAN** (on campus) or the public internet (off campus).
    - Uses **HTTPS** for secure transport of credentials and advising data.
- **Application Server → Database**
    - Local connection on the same host (no external network hop).
    - File-based SQLite access; effectively “wired” internal communication.
- **Application Server → OpenAI Cloud**
    - Over the public internet via **HTTPS**.
    - This is a **cloud-based** external connection with variable latency.
- **Client → Network Infrastructure**
    - Clients connect via Wi-Fi or Ethernet to the university LAN or home network, then out to the server’s hosting location.

Key Network Properties

- **Bandwidth Requirements**

    - Low to moderate:
        - Requests and responses are small JSON payloads.
        - Dataset uploads (CSV) are modest in size (1–3 MB typical).
    - No streaming video or large media files.
- **Latency**
    - **Client ↔ Backend**:
        - On-campus: typically 10–50 ms.
        - Off-campus: typically 50–200 ms.
    - **Backend ↔ OpenAI Cloud**:
        - Typically 700–1500 ms for LLM responses depending on load and region.
    - Latency impact is mitigated by rule-based fallback and asynchronous I/O.
- **Redundancy**
    - Within the project scope, IAAS assumes a **single backend node** without load balancing.
    - Logical redundancy is provided at the advising level (LLM + rule-based advisor), not at the network infrastructure level.

1.4 Data Management

1.4.1 Overview of Data Storage

IAAS uses a **local SQLite database** as its primary storage mechanism. All persistent data is stored in a single database file (e.g., iaas.db) hosted on the same machine as the backend application server.

The database stores:

- **Student data** (basic info, program, identifiers)
- **Course catalog** (course codes, titles, credits)
- **Prerequisite relationships** (course–prerequisite edges)
- **Enrollments and grades** (student-course history)
- **Chat logs** (queries, responses, timestamps, mode: LLM or rule-based)
- **Dataset versions** (metadata about uploaded synthetic datasets)
- **Admin accounts** (usernames and password hashes)

No data is stored permanently on the client side; browsers act only as frontends, with all persistent storage handled by the backend.

1.4.2 Data Structure Design

The database follows a **normalized relational schema** that reflects the academic advising domain.

Core Tables (Logical Design)

- **students**
    - student_id (PK), name, program, cohort, status
- **courses**
    - course_id (PK), title, credit_hours, level, category
- **prerequisites**
    - course_id (FK → courses)
    - prereq_id (FK → courses)

    - Represents a directed edge in the prerequisite graph.
- **enrollments**
    - student_id (FK → students)
    - course_id (FK → courses)
    - grade, term
- **chatbot_logs**
    - log_id (PK), student_id (FK), timestamp
    - query_text, response_text, mode (LLM / rule-based), success_flag
- **dataset_versions**
    - version_id (PK), label, uploaded_at, active_flag, checksum
- **admin_users**
    - admin_id (PK), username, password_hash

This design avoids redundant data (3NF) and supports efficient joins needed for advising and degree progress.

1.4.3 Data Access and Manipulation

All data access flows through a **repository layer** in the backend. The system does **not** allow raw SQL from UI or high-level services.

Access Patterns

- **Read Operations**
    - Fetch a student's enrollments and grades.
    - Retrieve course details and prerequisites.
    - Build degree progress views and eligibility lists.
    - Read logs for audit and analysis.
- **Write Operations**
    - Insert new logs for every Chat Advisor interaction.
    - Import datasets (students, courses, prerequisites, enrollments) through admin operations.
    - Activate a new dataset version (single record updated to active_flag = 1).

Manipulation Logic

- Dataset uploads are:
    - Parsed from CSV
    - Validated
    - Bulk-inserted into tables using transactions
- Advising and progress logic:
    - Perform **read-only** operations using the repository layer.
    - Never directly modify academic data during student queries.

This separation of concerns makes the system easier to test and maintain.

1.4.4 Data Integrity and Security

IAAS ensures that data remains consistent, correct, and accessible only to authorized users.

Data Integrity Mechanisms

- **Foreign key constraints**
    - Every enrollment must point to a valid student and course.
    - Every prerequisite must reference existing courses.
- **Transactional Dataset Imports**
    - Dataset imports run inside a single transaction.
    - On validation or integrity failure, the entire import is rolled back.
- **Single Active Dataset Version**

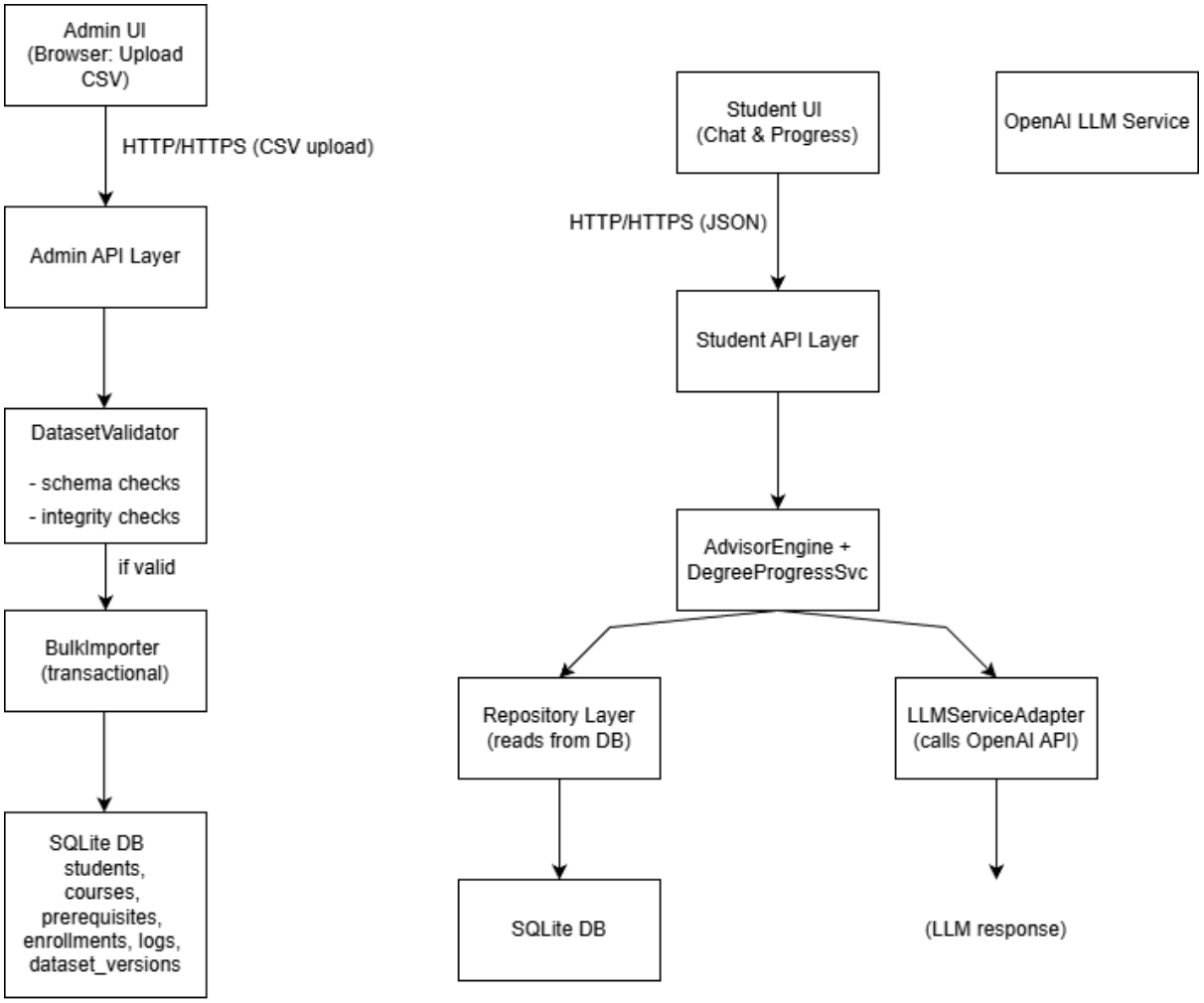
    - Only one dataset_versions.active_flag = 1 at a time.
    - Ensures the system always operates on a consistent dataset snapshot.
- **Validation Before Activation**
    - Schema and referential checks are performed before a new dataset version is activated.

Security Mechanisms

- **Admin Authentication**
    - Admin credentials are stored as salted, hashed passwords.
    - Only authenticated admins can upload or replace datasets.
- **Access Control**
    - Student data is not modified directly by students.
    - Dataset upload and version management is restricted to admins.
- **Communication Security**
    - All client–server interactions are intended to use HTTPS.
    - Sensitive data (e.g., credentials, logs) is not exposed to unauthorized users.

1.4.5 Data Flow and Storage Architecture (Diagram)

Below is a **data flow and storage architecture diagram** showing how data moves through IAAS:



1.5 Software Control Implementation

The IAAS system manages control flow using a layered architecture that clearly separates the user interface, application logic, and data management responsibilities. Control moves predictably from the UI to backend services through REST API calls, ensuring that each subsystem operates independently while maintaining coordinated system-wide behavior.
The design emphasizes **structured control**, **high responsiveness**, and **predictable handling of concurrent operations**. All major events in the system—such as student queries, progress evaluations, dataset uploads, and logging—are governed by well-defined control paths that prevent ambiguity and promote maintainability.

Key Characteristics of IAAS Control Design

- **Layered Control Flow:**
  Control always flows from UI → API Layer → Service Layer → Repository Layer → Database.
  No reverse or circular control is allowed, ensuring predictable system execution.
- **Service-Oriented Coordination:**
  Each major function (advising, validation, progress evaluation, data uploads) is encapsulated within a specific service.
  This ensures that control is modular and components do not interfere with each other.
- **Asynchronous Handling of Tasks:**
  FastAPI's async architecture allows multiple requests to be processed concurrently without blocking.
  LLM queries, dataset validation, and quota monitoring operate asynchronously to maintain smooth performance.
- **Rule-Based Fallback Control:**
  If the LLM service becomes unreachable or slow, control automatically shifts to the rule-based advisor to guarantee uninterrupted advising services.
- **Centralized Logging and Monitoring:**
  Every Chat Advisor interaction is logged through a controlled path, ensuring auditability and consistent behavior.
- **UI-Driven Execution:**
  All operations in IAAS originate from user actions:
  - Student Chat Advisor request
  - Student Degree Progress request
  - Admin Dataset Upload
  - Admin View Logs or Dataset Version
    The UI never accesses data directly and only communicates through controlled APIs.

Purpose of Control Implementation in IAAS

The purpose of this section is to outline how IAAS manages:

1. **External control flow** between major subsystems
2. **Internal coordination** between backend components
3. **Concurrent operations** such as student queries and admin uploads
4. **User Interface control**, including how the UI triggers backend processes
5. **Fallback logic** when external services (LLM API) become unavailable

This control design ensures the system remains reliable, predictable, and robust, even when handling multiple student requests or external API delays.

1.5.1 External Control Flow Between Subsystems

The IAAS system follows a structured, layered control flow architecture in which all control originates from the user interface and moves downward through defined subsystems: the API layer, service layer, repository layer, and database. This ensures a predictable sequence of operations, reduces coupling, and maintains the integrity of system functions such as advising, progress evaluation, dataset upload, and logging.

Control Flow Across Major Subsystems

1. User Interface Layer → API Layer

Control always begins with a user action:

- A student submits a Chat Advisor question
- A student requests degree progress
- An admin uploads a dataset
- An admin views logs or dataset status

The UI does **not** perform computation.
Instead, it sends an **HTTP/HTTPS request** to the appropriate backend endpoint.

2. API Layer → Service Layer

Upon receiving a request, the API layer:

- Validates the request data
- Determines which service must execute the logic
- Delegates control to the correct service component
  Examples:
  - /api/chat → AdvisorEngine
  - /api/progress/{id} → DegreeProgressService
  - /admin/upload → DatasetValidator + BulkImporter

3. Service Layer → Repository Layer

Service components implement the main business logic.
To complete tasks, they often need database data.
The Service Layer passes control to the Repository Layer to:

- Fetch student records
- Retrieve prerequisites
- Load course information
- Insert new chat logs
- Update dataset version information

4. Repository Layer → Database

Repositories translate business requests into SQL queries.
Control flows into the database engine, which:

- Executes SELECT, INSERT, UPDATE statements
- Ensures transactional correctness
- Returns structured results back up the chain

5. **Service Layer → External Systems (Optional)**

In the Chat Advisor subsystem, control may pass to:

- **LLMServiceAdapter**, which calls the OpenAI API

If the LLM request fails or times out, control seamlessly shifts to:

- **RuleBasedAdvisor** (local deterministic reasoning)

Component That Initiates Control
Students and Admins initiate all control
The system does **not** trigger major processes automatically except:

- Background quota monitoring
- Scheduled validation cleanup

All meaningful control events are user-driven.

Key Interactions Ensuring Correct Order of Operations

- **Dataset upload order:**
  UI → API → Validator → BulkImporter → DB
  (Validation must succeed before importing.)
- **Advising order:**
  UI → API → AdvisorEngine → (LLM or fallback) → Repository → Log → UI
  (Ensures advice, then logging, then response.)
- **Degree progress order:**
  UI → API → DegreeProgressService → Repository → Graph Calculation → UI
  (Prevents incomplete results.)
- **Admin operations:**
  Always require authentication before entering service logic.

These enforced sequences ensure system correctness and protect data integrity.

1.5.2 Concurrent Control

IAAS supports multiple operations running simultaneously, primarily due to multiple students accessing the system at the same time, administrator actions occurring in parallel, and background tasks running independently. This section describes how concurrency is handled, how synchronization is maintained, and how the system avoids issues such as race conditions or conflicting updates.

Parallel and Asynchronous Components in IAAS
1. Multiple Student Sessions (Parallel Requests)

- Many students may query the Chat Advisor at the same time.
- Degree progress requests may also occur concurrently.
- Each incoming HTTP request is handled by **an independent FastAPI worker**.

2. Admin Operations in Parallel with Student Requests

- Admin dataset uploads
- Dataset validation
- Log viewing

These operations may occur while students are actively using the system.

3. Background Tasks

- **QuotaMonitor** runs periodically to track API usage.
- Dataset validation may occur in the background before import.
- LLM requests run asynchronously and do not block other user interactions.

4. External LLM Requests (Asynchronous)

- AdvisorEngine sends requests to OpenAI using asynchronous I/O.
- Responses may arrive at different times without blocking the main thread.

Concurrency Management and Synchronization Techniques

1. Asynchronous Request Handling (FastAPI)

FastAPI uses asynchronous event loops to:

- Prevent blocking on network calls
- Allow hundreds of users to interact without slowdowns
- Separate long-running tasks from immediate responses

This ensures the system remains responsive even during heavy load.

---

2. Database Synchronization (SQLite Locks + Transactions)

SQLite enforces:

- **Single-writer, multi-reader model**
- Atomic write transactions
- Full locking during writes to prevent corruption

IAAS leverages this by:

- Using short, efficient transactions
- Performing imports in bulk transactions to minimize lock duration
- Using append-only logs to avoid write conflicts

---

3. Dataset Version Switching (Atomic Operation)

Switching the active dataset:

- Occurs within **a single SQL transaction**
- Ensures no partial state
- Prevents two datasets from being active at once
- Avoids race conditions during updates

This protects the system even when admins perform updates while students are querying data.

---

4. Thread-Safe Fallback Logic for LLM Failures

If the OpenAI API becomes slow or unreachable:

- A timeout triggers the **RuleBasedAdvisor**
- This fallback is deterministic and thread-safe
- No shared state is modified during fallback triggering

This prevents concurrency issues when multiple LLM calls fail simultaneously.

Preventing Race Conditions, Deadlocks, and Conflicts

Techniques Used:

1. **Short-Lived Transactions**
   - Write operations occur quickly and do not hold locks long.
2. **Single-Writer Model (SQLite)**
   - Eliminates deadlocks because SQLite does not allow complex locking cycles.
3. **Immutable Dataset Versions**
   - Once imported, dataset versions are immutable.
   - No concurrent updates to the same dataset occur.
4. **Atomic Activation of New Dataset**
   - Prevents race conditions when toggling active datasets.
5. **Stateless API Design**
   - Each HTTP request is self-contained.
   - No shared mutable state across threads.
   - Eliminates thread-level race conditions.
6. **Async Network Calls**
   - LLM calls do not block event loops, preventing cascading delays.

Simple Concurrency Pseudocode Example:

```
async def handle_chat_request(query):

    # Run LLM call without blocking

    llm_task = asyncio.create_task(call_llm(query))

    # Wait with timeout

    try:

        response = await asyncio.wait_for(llm_task, timeout=3)
    except TimeoutError:
        response = rule_based_fallback(query)

    # Log result in atomic DB transaction

    with sqlite_transaction():
        save_log(query, response)

    return response
```

1.5.3 User Interface

The IAAS User Interface (UI) is a web-based front end accessed through a standard browser. It provides an intuitive interaction layer for students and administrators while ensuring strict separation between presentation, application logic, and data management. All UI actions communicate with backend controllers through well-defined REST API calls, following a simplified MVC-like architecture.

Connection Between UI and System Logic

The UI does not perform any advising, progress computation, or data validation.
Instead, it interacts with the backend through:

- **HTTPS requests** to the FastAPI controllers
- **JSON payloads** sent to and received from the backend
- **Asynchronous UI updates** based on server responses

This ensures:

- Predictable control flow
- Thin client design
- Strict enforcement of business logic on the server

Every user action triggers a backend operation, such as:

- Student submitting a Chat Advisor query → /api/chat
- Student checking their degree progress → /api/progress/{id}
- Admin uploading dataset → /admin/upload
- Admin viewing logs → /admin/logs

Main UI Components and Their Communication Paths

1. Login Screen

- Entry point for students and admins
- Validates credentials via /auth/login
- Redirects to:
  - **Student Dashboard** (role = student)
  - **Admin Dashboard** (role = admin)

2. Student Dashboard

Consists of two primary modules:

a. Chat Advisor UI

- Text input box for questions
- Chat history panel
- Sends requests to /api/chat
- Displays responses from LLM or rule-based advisor

b. Degree Progress UI

- Shows completed courses, requirements, and visual graphs
- Requests data from /api/progress/{student_id}
- Renders a JSON-based structure into UI components

---

3. Admin Dashboard

Contains admin-only UI modules:

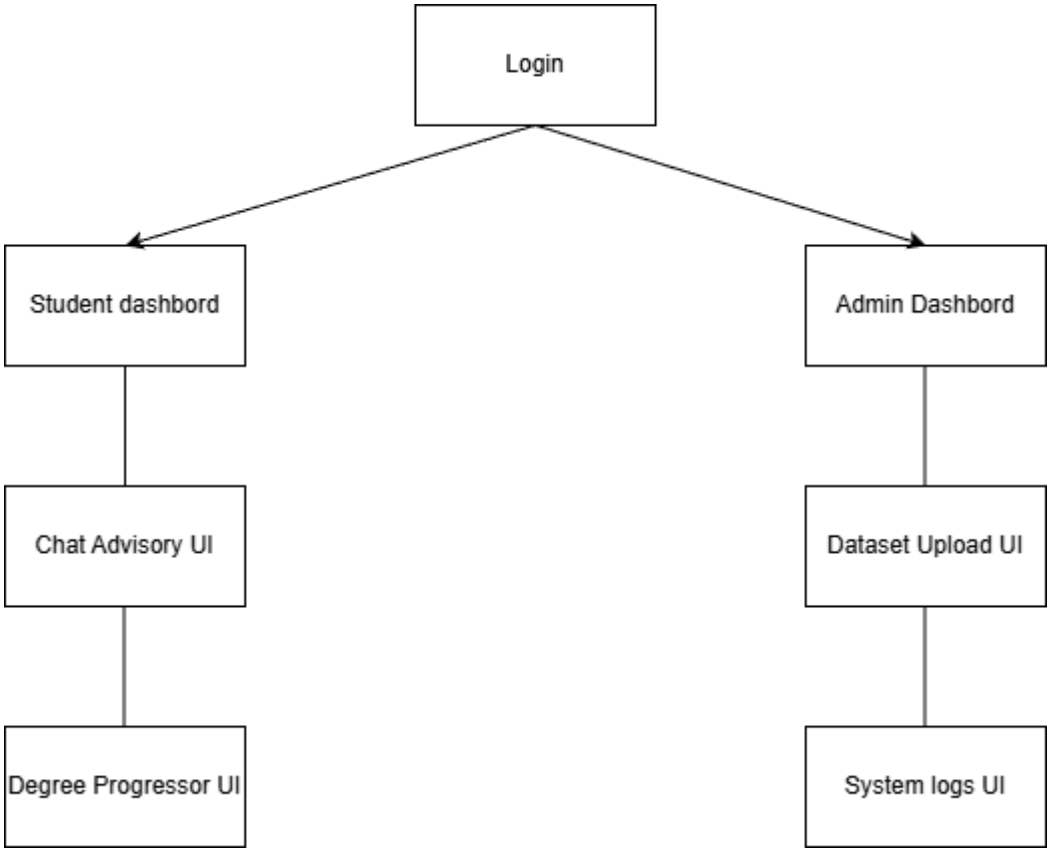
a. Dataset Upload UI

- File picker for CSV datasets
- Sends the file to /admin/upload
- Displays validation results and logs
- Shows active dataset version

b. System Logs / Quota UI

- Retrieves logs from /admin/logs
- Displays number of LLM tokens used
- Shows fallback occurrences and system health indicators

UI Flow Diagram
Below is showing transitions between screens:



1.6 Boundary Conditions

The IAAS behaves during transitional states—system startup, system shutdown, and unexpected failures. These boundary conditions are essential to maintaining reliability, preventing data corruption, and ensuring a smooth user experience even in exceptional situations.
IAAS incorporates controlled initialization sequences, safe shutdown procedures, and robust recovery mechanisms to handle various runtime conditions without disrupting advising functions or compromising data integrity

1.6.1 Initialization

During startup, IAAS performs a structured series of steps to prepare the system for operation. The goal is to ensure that all components (services, database connections, prerequisite graph, logging, and configuration) are ready before user requests are accepted.

Initialization Steps

1. **Load Environment Configuration**
    - Loads API keys, file paths, database location, and server configurations.
    - Performs sanity checks to ensure required variables exist.
2. **Initialize FastAPI Application**
    - Sets up routing, middleware, validators, and exception handlers.
3. **Load Active Dataset Version**
    - Reads the dataset_versions table.
    - Ensures there is exactly **one active dataset**.
    - Loads that dataset into memory.
4. **Build In-Memory Prerequisite Graph**
    - Converts prerequisite tables into a graph structure.
    - Ensures fast eligibility checking for advising and degree progress.
5. **Initialize Services**
    - AdvisorEngine
    - DegreeProgressService
    - DatasetValidator
    - BulkImporter
    - Authentication service
6. **Start Background Processes**
    - QuotaMonitor to track API usage.
    - Optional cleanup or scheduled tasks.
7. **Begin Accepting Requests**
    - Only after all steps succeed.

Startup Error Behavior

- If no dataset is active → system enters **safe mode**, allowing only administrative logins.
- If the database file is missing → system stops and logs a critical error.
- If prerequisite graph fails to build → advising is disabled until the issue is fixed.

1.6.2 Termination

The system follows a **graceful shutdown sequence** to ensure that no data is lost and no inconsistent state remains.

Shutdown Steps

1. **Reject New Incoming Requests**
    - Server stops accepting new HTTP connections.
2. **Complete Ongoing Requests**
    - Active advising or progress computations finish.
    - Outstanding LLM calls are allowed to complete or time out.
3. **Flush Logs and Save State**
    - All queued logs are written to SQLite.
    - Quota usage counters are stored safely.
4. **Close Database Connections**
    - Closes SQLite file handles.

    - Ensures all transactions are completed.
5. **Shutdown Background Tasks**
    - QuotaMonitor and any validation tasks are terminated.
6. **Stop the Server**
    - The system shuts down cleanly.

Abnormal Shutdown Handling

If the server terminates abruptly (e.g., power failure):

- SQLite's journaling mode prevents database corruption.
- Incomplete dataset imports are rolled back automatically.
- On next startup, the system validates integrity before accepting requests.

1.6.3 Failure

IAAS is designed to handle failures gracefully and maintain user experience even when external systems (such as the OpenAI LLM) become unavailable.

Failure Types and Behaviors

---

1. LLM Failure (Timeout, Network Error, Rate Limit)

Behavior:

- AdvisorEngine detects timeout.
- Automatically switches to **RuleBasedAdvisor**.
- Logs failure event for analysis.
- User still receives a valid advising response.

This maintains system reliability even under unstable network conditions.

---

2. Invalid or Corrupted Dataset Upload

Behavior:

- DatasetValidator detects schema mismatches or broken references.
- Import is refused.
- System generates a detailed validation report.
- Active dataset remains unchanged.

This prevents corrupted data from affecting advising services.

---

3. Database Failure

Examples:

- Missing database file
- Permission issues
- Corrupted index

Behavior:

- System enters **safe mode**.
- Only admin routes remain available.
- Advising and progress computations are disabled until fixed.

---

4. API-Level Failures

Examples:

- Invalid request format
- Missing fields
- Unauthorized access

Behavior:

- API returns structured JSON error responses.
- No partial operations occur.
- Errors are logged for diagnosis.

---

5. Background Task Failures
If quota monitoring or dataset validation fails:
- System logs the exception.
- Advising for students remains unaffected.
- Background task may restart or pause depending on severity.

6. Object Oriented Design

The Object-Oriented Design of the Intelligent Academic Advisory System (IAAS) translates the high-level architectural structure (UI, API, Services, Data, and Infrastructure layers) into concrete classes, their responsibilities, and their relationships. The design follows standard OO principles such as **encapsulation**, **separation of concerns**, and **single responsibility**, ensuring that each class has a clear and focused purpose.

The IAAS system uses object-oriented design to organize its functionality into well-structured classes grouped across several logical layers.
Domain Model Classes
These define the core academic entities the system manages:
- **Student** – student information
- **Course** – course catalog data
- **Prerequisite** – course prerequisite relationships
- **Enrollment** – student course history

These classes directly represent the database structure.

Service & Business Logic Classes
These implement the system's core behavior:
- **AdvisorEngine**, **RuleBasedAdvisor**, **LLMServiceAdapter** – handle academic advising
- **DegreeProgressService** – computes student progress
- **DatasetValidator**, **BulkImporter** – manage dataset uploads
- **QuotaMonitor** – tracks LLM usage

Each service encapsulates one major subsystem function.

Data Access (Repository) Classes
These classes provide clean database access:
- StudentRepository
- CourseRepository
- PrerequisiteRepository
- EnrollmentRepository
- ChatLogRepository
- DatasetVersionRepository

They hide SQL details and expose simple methods for retrieving and saving data.

Infrastructure & Support Classes
These handle cross-cutting technical functionality:

- **AuthService** – admin authentication
- **ConfigManager** – configuration loading
- **Logger / AuditLogger** – system logging

UI & API Controller Classes
Controllers serve as the system's entry points:

- **ChatController** – /api/chat
- **ProgressController** – /api/progress/{id}
- **AdminDatasetController** – /admin/upload
- **AdminLogController** – /admin/logs

They receive user requests, call the appropriate service, and return JSON responses.

Design Objectives
The OO design ensures that IAAS:

- Follows a clean layered architecture (UI → API → Services → Data)
- Supports modularity, reuse, and testability
- Can be extended easily (new advising logic, new UI features)
- Isolated external dependencies behind specific adapter or repository classes

2.1 Class Diagrams

The high-level Class Diagram for IAAS captures the main classes in the system, their key attributes and operations, and the relationships between them. The design follows object-oriented principles such as **encapsulation**, **single responsibility**, and **clear separation between domain, services, and data access**.

Major Classes and Their Responsibilities
Domain Classes

- **Student**
  - *Attributes:* studentId, name, program, cohort, status
  - *Operations:* getCompletedCourses(), getGPA() (optional/derived)
- **Course**
  - *Attributes:* courseId, title, creditHours, level, category
  - *Operations:* isElective(), isCore()
- **Prerequisite**
  - *Attributes:* courseId, prereqId
  - *Operations:* none (acts as a relationship entity)
- **Enrollment**
  - *Attributes:* studentId, courseId, grade, term
  - *Operations:* isPassed()

These classes encapsulate academic data and are tightly aligned with the database tables.

Service / Business Logic Classes

- **AdvisorEngine**
  - *Attributes:* references to RuleBasedAdvisor, LLMServiceAdapter, PrerequisiteGraph
  - *Operations:* getRecommendations(studentId, query), selectEngine()
- **RuleBasedAdvisor**

    - *Attributes:* prerequisiteGraph, repositories
    - *Operations:* generatePlan(student, targetCourse), getEligibleCourses(student)
- **LLMServiceAdapter**
    - *Attributes:* apiKey, modelName
    - *Operations:* generateResponse(prompt), buildPromptFromContext(student, query)
- **DegreeProgressService**
    - *Attributes:* repositories, prerequisiteGraph
    - *Operations:* computeProgress(studentId), getRemainingRequirements(studentId)
- **DatasetValidator**
    - *Operations:* validateSchema(csvFile), validateIntegrity(dataset)
- **BulkImporter**
    - *Operations:* importDataset(validatedData), rollbackOnFailure()
- **QuotaMonitor**
    - *Attributes:* currentUsage, limit
    - *Operations:* trackUsage(tokensUsed), isQuotaExceeded()

Service classes encapsulate logic and do not expose internal state directly, supporting encapsulation and reuse.

Data Access (Repository) Classes

- **StudentRepository**
    - *Operations:* findById(studentId), getAllEnrollments(studentId)
- **CourseRepository**
    - *Operations:* findById(courseId), getAllCourses()
- **PrerequisiteRepository**
    - *Operations:* getPrereqsForCourse(courseId)
- **EnrollmentRepository**
    - *Operations:* getEnrollmentsForStudent(studentId)
- **ChatLogRepository**
    - *Operations:* saveLog(entry), getLogsByStudent(studentId)
- **DatasetVersionRepository**
    - *Operations:* getActiveVersion(), setActiveVersion(versionId)

Repositories provide a clean API to the database and hide SQL details, supporting encapsulation and separation of concerns.

Infrastructure / Support Classes

- **AuthService**
    - *Operations:* authenticate(username, password), hashPassword(), verifyPassword()
- **ConfigManager**
    - *Operations:* getValue(key), loadConfig()
- **Logger / AuditLogger**
    - *Operations:* logEvent(message), logError(exception), logChatInteraction(entry)

Key Relationships (Textual View of the Class Diagram)

- **Student ↔ Enrollment ↔ Course**
    - Student has many Enrollment records.

    - Each Enrollment is associated with exactly one Course.
- **Course ↔ Prerequisite**
    - Course may have zero or more Prerequisite entries.
    - Each Prerequisite links a courseId to a prereqId (both pointing to Course).
- **AdvisorEngine**
    - *uses* RuleBasedAdvisor
    - *uses* LLMServiceAdapter
    - *uses* StudentRepository, CourseRepository, PrerequisiteRepository, EnrollmentRepository
- **DegreeProgressService**
    - *uses* StudentRepository, EnrollmentRepository, CourseRepository, PrerequisiteRepository
- **DatasetValidator** and **BulkImporter**
    - *use* CourseRepository, StudentRepository, EnrollmentRepository, DatasetVersionRepository
- **Controllers (e.g., ChatController, ProgressController)**
    - *depend on* AdvisorEngine and DegreeProgressService
    - They are not domain objects but boundary classes in the API layer.

Use of OO Principles

- **Encapsulation**
    - Each class exposes high-level methods (e.g., computeProgress, generatePlan) and hides internal data structures and SQL logic.
- **Single Responsibility**
    - Each service class handles one core concern (advising, progress, validation, importing).
- **Polymorphism (at the advising engine level)**
    - AdvisorEngine can use either LLMServiceAdapter or RuleBasedAdvisor to generate advice, depending on availability and configuration.
- **Clear Separation of Layers**
    - Domain (Student, Course, Enrollment…)
    - Services (AdvisorEngine, DegreeProgressService…)
    - Repositories (StudentRepository, CourseRepository…)
    - Controllers (ChatController, ProgressController…)

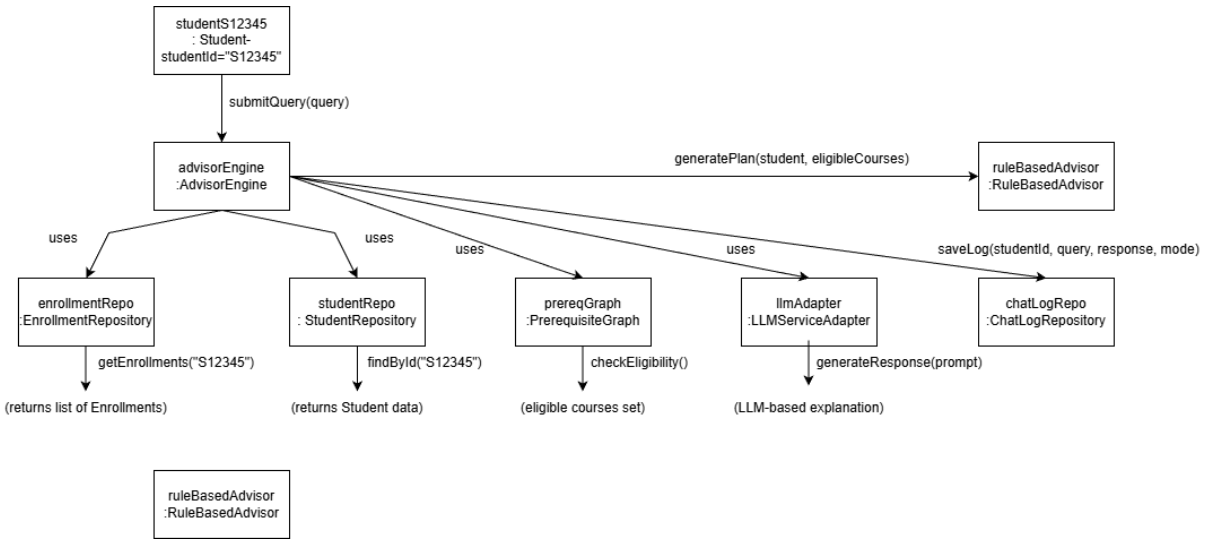
If you want, I can also write a **PlantUML or Mermaid class diagram** snippet so you can generate an *actual visual class diagram image* from this.

2.2 Object Diagrams

The diagram shows the objects involved when a student submits a question to the Chat Advisor and receives a recommendation.

Scenario:

Student "S12345" asks: "What courses can I take next semester?"



2.3 Packages

The IAAS system is organized into a set of logical packages that reflect its layered architecture. Grouping classes into well-defined packages improves maintainability, reduces coupling between components, and supports clear separation of concerns. Each package corresponds to a distinct responsibility in the system, ensuring that related classes are kept together while preventing unnecessary dependencies across layers.

IAAS uses the following primary packages:

1. **ui** – Contains all user interface components and client-side screens.
2. **api** – Contains the FastAPI controllers that serve as the system's entry points.
3. **services** – Contains the business logic and core subsystem controllers.
4. **data** – Contains data access classes and repository implementations.
5. **infra** – Contains external service integrations and infrastructure components.
6. **security** – Contains authentication and authorization logic.
7. **core** – Contains shared utilities and non-domain-specific helpers.

Each package encapsulates a distinct part of the system's functionality.

1. ui Package

Contains client-facing UI components (HTML/JS pages or frontend modules).

- *Purpose:* Display information to students and admins; capture user actions.
- *Examples:*
  - Chat Advisor page
  - Degree Progress visualization
  - Dataset upload screen
  - Admin logs dashboard

The UI does not contain business logic; it communicates only through API endpoints.

---

2. api Package

Contains all controller classes.

- *Purpose:* Receive HTTP requests, validate inputs, call service classes, return JSON responses.
- *Classes:*

    - ChatController
    - ProgressController
    - AdminDatasetController
    - AdminLogController

These controllers form the boundary between the UI and backend logic.

---

3. services Package

Implements the system's key behaviors.

- *Purpose:* Execute advising logic, compute degree progress, validate datasets, and manage imports.
- *Classes:*
    - AdvisorEngine
    - RuleBasedAdvisor
    - LLMServiceAdapter
    - DegreeProgressService
    - DatasetValidator
    - BulkImporter
    - QuotaMonitor

This package forms the “middle layer” of IAAS and contains most of the business rules.

---

4. data Package

Handles all database interactions.

- *Purpose:* Provide clean methods for retrieving and storing data, isolated from SQL details.
- *Classes:*
    - StudentRepository
    - CourseRepository
    - PrerequisiteRepository
    - EnrollmentRepository
    - ChatLogRepository
    - DatasetVersionRepository

This package ensures consistent access to data and supports easier testing and maintenance.

---

5. infra Package

Contains infrastructure-level services.

- *Purpose:* Integrate with external APIs and provide system-level support.
- *Classes:*
    - OpenAIAdapter / LLMServiceAdapter
    - ConfigManager
    - PrerequisiteGraph (structure built at startup)

---

6. security Package

Handles authentication and access control.

- *Purpose:* Manage admin login and ensure that only authorized users can upload datasets or view logs.
- *Classes:*
    - AuthService
    - HashingUtilities (optional)

---

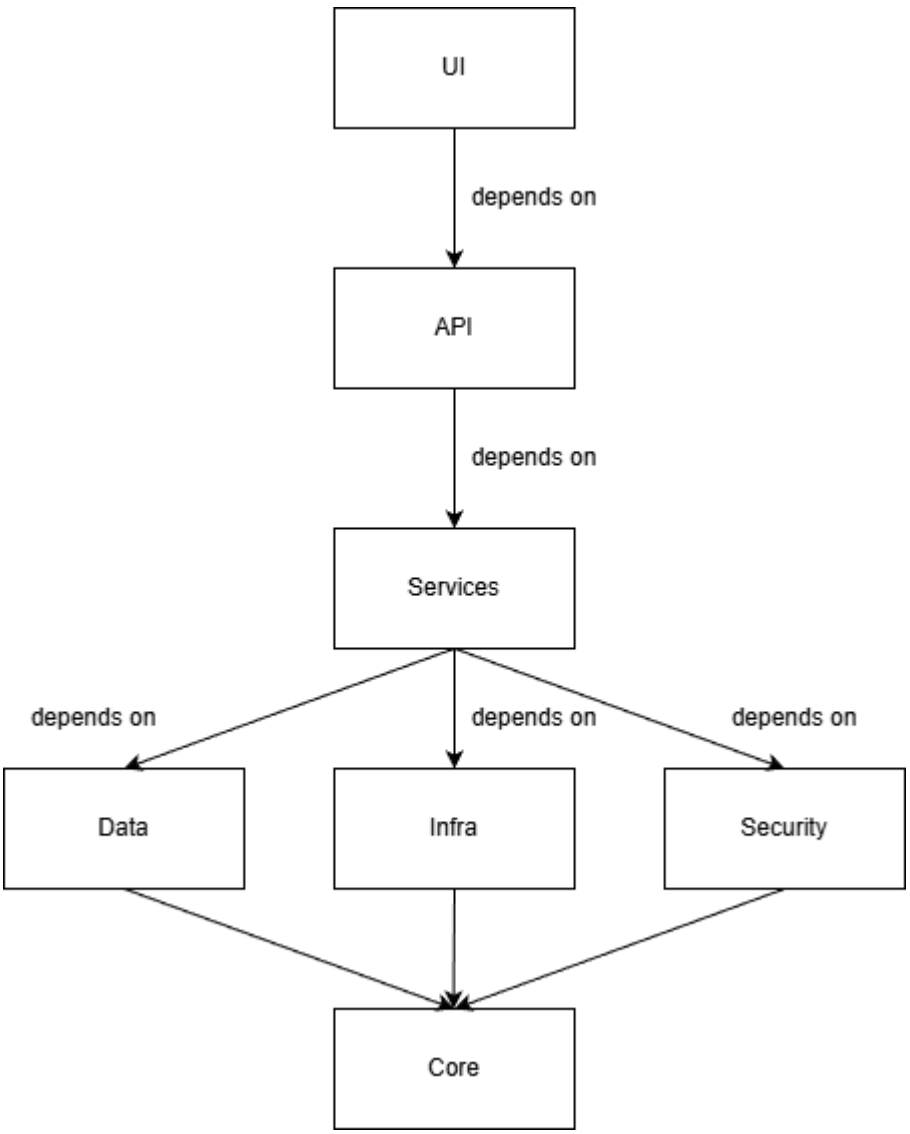
7. core Package
Contains shared utilities and helpers.

- *Purpose:* Provide reusable code used by multiple packages.
- *Examples:*
    - Common exception classes
    - Logging or formatting utilities
    - Shared constants / configuration keys

2.3.1 Package Diagrams

The IAAS system is organized into a set of logical packages that reflect its layered architecture. The UML Package Diagram shows the main packages and their dependencies, emphasizing how higher-level layers (UI, API) depend on lower-level layers (Services, Data, Infrastructure, Security, Core).

UML Package Diagram:



2.3.2 Software Package Documentation

---

1. ui Package
Purpose:
Provides the web-based user interface for students and administrators. It handles screen layout, form inputs, and visualization of responses and progress data. It exists to separate presentation concerns from backend logic.
Key Classes / Components (conceptual):

- LoginPage
- StudentDashboardPage
- ChatAdvisorView
- DegreeProgressView
- AdminDashboardPage
- DatasetUploadView
- LogsView

Responsibilities:

- Render pages for Chat Advisor, Degree Progress, Dataset Upload, and Logs.
- Capture user actions (queries, clicks, uploads) and send them to the backend.
- Display JSON responses as formatted UI elements (tables, charts, messages).

Dependencies:

- Depends on the **api** package via HTTP/HTTPS calls to REST endpoints.
- Does *not* directly depend on services, data, or core.

---

2. api Package
Purpose:
Provides the system's official entry points via REST APIs. It receives HTTP requests from the UI, validates requests, calls service classes, and returns JSON-based responses.
Key Classes:

- ChatController
- ProgressController
- AdminDatasetController
- AdminLogController
- AuthController (optional)

Responsibilities:

- Map URLs (e.g., /api/chat, /api/progress/{id}, /admin/upload) to controller methods.
- Validate incoming request data and parameters.
- Invoke the appropriate service methods in the services package.
- Translate service results into HTTP responses (JSON).

Dependencies:

- Depends on the **services** package for business logic.
- May depend on **security** for authentication and authorization.
- Indirectly relies on data, infra, and core through services.

---

3. services Package
Purpose:
Implements the core business logic of IAAS. It coordinates domain objects, repositories, and external services to perform advising, degree progress computation, dataset validation, and imports.

Key Classes:

- AdvisorEngine
- RuleBasedAdvisor
- LLMServiceAdapter (may also live in infra, depending on implementation)
- DegreeProgressService
- DatasetValidator
- BulkImporter
- QuotaMonitor

Responsibilities:

- Generate course recommendations and advising responses.
- Compute degree progress and remaining requirements.
- Validate and import new datasets.
- Track and enforce LLM usage limits.
- Encapsulate all "business rules" of the IAAS system.

Dependencies:

- Depends on **data** for database access (via repositories).
- Depends on **infra** for interacting with external services (OpenAI, config, prerequisite graph).
- Depends on **security** where admin operations require authentication.
- Uses shared utilities from **core**.

---

4. data Package

Purpose:

Encapsulates all data persistence logic. It acts as the bridge between services and the SQLite database, isolating SQL and schema details from business logic.

Key Classes:

- StudentRepository
- CourseRepository
- PrerequisiteRepository
- EnrollmentRepository
- ChatLogRepository
- DatasetVersionRepository

Responsibilities:

- Provide clean methods for querying and updating database tables.
- Implement CRUD operations for all domain entities.
- Handle transactional operations for bulk imports.
- Ensure data consistency with appropriate queries.

Dependencies:

- Depends on **core** for shared utilities (e.g., error handling, helpers).
- Does *not* depend on services or ui (to keep layering clean).

---

5. infra Package

Purpose:

Provides infrastructure-related capabilities such as integration with external services and core technical components that support the rest of the system.

Key Classes:

- OpenAIAdapter / LLMServiceAdapter (if placed here)
- ConfigManager
- PrerequisiteGraph (graph structure built from the DB)

Responsibilities:

- Handle communication with the OpenAI LLM service (HTTP client, retries, timeouts).
- Load configuration parameters (API keys, DB paths, environment settings).
- Build and maintain in-memory structures like the prerequisite graph.

Dependencies:

- Depends on **core** for shared structures, utilities, and possibly logging helpers.
- May interact with **data** to build the prerequisite graph at startup.

---

6. security Package

Purpose:

Manages authentication, authorization, and security-related functionality. Ensures only authorized users (admins) can perform sensitive operations.

Key Classes:

- AuthService
- PasswordHasher / HashingUtilities

Responsibilities:

- Authenticate admin users based on username/password.
- Hash and verify passwords securely.
- Provide helper methods for checking access rights.

Dependencies:

- Depends on **data** for accessing admin user records.
- Depends on **core** for shared error types, logging, or utility functions.

---

7. core Package

Purpose:

Contains shared utilities, constants, and helper functionality that are used across multiple packages. It should not depend on any higher-level package.

Key Classes / Components:

- AppException or custom exception types
- Common value objects or result wrappers
- Utility classes for formatting, date handling, or mapping

Responsibilities:

- Provide reusable code that reduces duplication.
- Offer a central place for common constructs shared by services, data, infra, and security.

Dependencies:

- **Does not depend on other packages** (base layer).
- Is used by services, data, infra, and security.

2.3.3 Design Constraints

---

1. Platform & Environment Constraints

Constraint:

The system must run in a typical university environment with standard lab machines and student laptops, accessed through a web browser. The backend is expected to run on a single server or lightweight VM.

Source:

- Implied by SRS (web-based system for students and admins)
- Project environment (no dedicated cluster or microservice infrastructure)

Impact on Design:

- Encouraged a **monolithic backend** with layered packages instead of distributed microservices.
- Simplified package structure (ui, api, services, data, infra, security, core) to be compatible with one deployment unit.
- Avoided platform-specific features that would complicate portability.

---

2. Technology Stack Constraints

Constraint:

Use technologies aligned with course expectations and available tools:

- Python-based backend (FastAPI)
- SQLite as the database
- OpenAI API as the LLM provider.

Source:

- Supervisor guidelines and project scope
- Limited time and infrastructure for managing complex DB or heavy frameworks

Impact on Design:

- Packages such as data and infra had to encapsulate details of **SQLite** and **OpenAI** usage.
- Introduced specific classes like LLMServiceAdapter, PrerequisiteGraph, and repository classes to isolate these technologies from the rest of the system.
- Package boundaries were drawn so future migration (e.g., SQLite → PostgreSQL) only affects data, not services.

---

3. Performance & Scalability Constraints

Constraint:

IAAS must handle multiple students using Chat Advisor and Degree Progress in parallel with acceptable response times (around 2 seconds) while running on modest hardware.

Source:

- SRS non-functional requirements (responsiveness for advising)
- Realistic performance expectations for students interacting with the system

Impact on Design:

- Led to the creation of specialized services such as DegreeProgressService and AdvisorEngine to **centralize** computational logic and optimize it.
- Motivated the infra package's PrerequisiteGraph component to maintain prerequisites in memory, reducing repeated DB lookups.
- Enforced a clear separation between services and data so that repository calls are controlled and minimized.
- Encouraged stateless design of controllers and services so they can scale horizontally in the future.

---

4. Security & Privacy Constraints

Constraint:

Student data and admin operations must be protected from unauthorized access. Admin dataset uploads must be restricted, and credentials must be stored securely.

Source:

- SRS security requirements
- General academic and ethical expectations about student data privacy

Impact on Design:

- Introduced a dedicated security package containing AuthService and password hashing utilities.
- Forced all admin-related functionality into the api and services layers, with **no direct UI or DB manipulation** without going through security checks.
- Influenced package dependencies: api depends on security, but ui does not talk to security directly.

---

5. Reliability & Availability Constraints
Constraint:
The system should continue to provide advising even if the LLM service is slow or unavailable, and it must avoid corrupting data under failures.
Source:
- SRS requirement for robust advising
- Practical risk of relying on an external cloud API (OpenAI)

Impact on Design:
- Created RuleBasedAdvisor as an alternative advising mechanism in the services package.
- AdvisorEngine was designed to **encapsulate fallback logic** and decide whether to use LLM or rule-based advising.
- The data package uses transactional repository operations to avoid partial updates during dataset uploads.

---

6. Integration Dependencies
Constraint:
IAAS must integrate with an external LLM (OpenAI) but keep the rest of the system independent of the provider choice.
Source:
- SRS functionality for natural language advising via an LLM

Impact on Design:
- The LLMServiceAdapter (in services or infra) encapsulates OpenAI details in one place.
- Other packages (e.g., services, api) only depend on the **adapter interface**, not directly on the OpenAI SDK.
- This motivated the infra package as a separation layer for infrastructure/integration concerns.

---

7. Deployment & Networking Constraints
Constraint:
The project is intended to be deployed as a **single backend node**, not a distributed microservice architecture, with external LLM over the internet.
Source:
- Project size and time constraints
- No requirement for container orchestration or multi-node setups

Impact on Design:
- Kept packages inside a single process while still reflecting conceptual separation.
- Simplified communication patterns to synchronous calls between packages (no message queues).
- Led to the monolithic, layered package structure rather than a network of individually deployable services.

---

8. Resource Limits & Project Constraints
Constraint:
Limited development time (capstone timeline), limited team size, and limited budget for LLM usage.
Source:

- Course/project timeline
- OpenAI token costs
- Three-member student team

Impact on Design:

- Pushed for **simple, understandable architecture** (layered monolith + clear packages) rather than complex patterns.
- Encouraged clear separation of concerns to allow parallel work (e.g., one member on ui/api, one on services, one on data/infra).
- Led to the introduction of QuotaMonitor to track and restrict centralized LLM usage.

2.3.4 Design Rationale and Alternatives

Chosen Design: Layered Monolithic Architecture with Clear Packages
IAAS adopts a layered architecture with these core layers:

- **UI layer** (ui)
- **API / Controller layer** (api)
- **Service / Business Logic layer** (services)
- **Data Access layer** (data)
- **Infrastructure and Security layers** (infra, security, core)

This structure was chosen because:

1. It **directly matches** the SRS view of the system (students and admins using a web UI, one backend handling logic, one DB).
2. It supports **clean separation of concerns**, making the system easier to reason about, develop, and maintain.
3. It allows the team to work in parallel on different layers without stepping on each other's toes.
4. It is realistic for a course project—fully implementable within the time and resource constraints.

Key Design Decisions
1. Services + Repositories Instead of "Fat Controllers"
Reason:

- Keeping controllers thin and moving logic into service classes (AdvisorEngine, DegreeProgressService, DatasetValidator) improves testability and reuse.
- Repositories (StudentRepository, CourseRepository, etc.) isolate DB access and allow future changes to the database without touching services.

Alternative Rejected:

- Storing logic in controllers (Fat Controllers).
- Rejected because it would mix HTTP routing with advising logic, making the system harder to maintain and test.

---

2. Adapter for LLM Instead of Direct API Calls Everywhere
Reason:

- LLMServiceAdapter centralizes all OpenAI calls and hides external API details from the rest of the system.
- Makes it easier to mock the LLM in tests.
- Required by integration and technology constraints.

Alternative Rejected:

- Calling the OpenAI API directly from multiple services or controllers.
- Rejected because it increases coupling and makes changes (e.g., model switch) risky and scattered.

---

3. Dedicated Rule-Based Advisor as Fallback

Reason:

- Ensures continued advising even when external LLM is unavailable or too slow.
- Aligns with reliability constraints from SRS.
- Keeps the advising domain logic directly under our control.

Alternative Rejected:

- Pure LLM-only solution without rule-based backup.
- Rejected because it would create a single point of failure on the LLM service and violate robustness expectations.

---

4. Single Monolithic Backend Instead of Microservices

Reason:

- Course constraints (team size, time, infra) and SRS do not require microservices.
- Monolith with clean packages is simpler to understand, deploy, and debug.
- Sufficient for expected load (university use, not global SaaS).

Alternative Rejected:

- Splitting into separate services: advising-service, progress-service, dataset-service, etc.
- Rejected because of deployment complexity, increased ops overhead, and little added value at this scale.

---

5. Package Layering (UI → API → Services → Data/Infra/Core)

Reason:

- Prevents dangerous circular dependencies.
- Enforces a clear "direction of control" (top to bottom).
- Makes impact of change easier to predict (e.g., DB schema change mostly affects data).

Alternative Rejected:

- Allowing direct access from UI components to the database or services without controllers.
- Rejected for security, maintainability, and clarity reasons.

How the Design Meets Constraints

- **Scalability:** Stateless services and repository patterns allow horizontal scaling if needed (multiple backend instances).
- **Cost:** SQLite + a monolithic backend + careful LLM usage keeps infrastructure and token costs low.
- **Security:** Clear separation of security package and restrictions on admin endpoints protect sensitive operations.
- **Maintainability:** Each package addresses one responsibility, making it easier to update specific parts (e.g., change advising rules) without touching everything.

- **Extensibility:** New advising strategies, visualization features, or data fields can be added by extending services and domain classes inside existing packages without restructuring the whole system.