System Architecture & Design Decisions

This section details the architectural blueprint of the AI-Assisted Programming Platform, focusing on the centralized management of data flow, modular service breakdown, API design, and deployment strategies required to maintain academic integrity and system scalability.

1. High-Level Architecture

The system follows a **Service-Oriented Architecture (SOA)** where the **Node.js Backend** acts as the central orchestrator (gateway). This design ensures that the Client (Frontend) never communicates directly with sensitive components like the Database or the AI Service, thereby enforcing security and academic integrity rules centrally.

Although the system follows service-oriented principles through external services (Judge0, AI), the backend is implemented as a modular monolithic application acting as a central gateway. This allows consistent policy enforcement (RBAC, exam constraints) and simplifies deployment while keeping integrations loosely coupled.

1.1. Architectural Flow:

1. **Frontend (React):** Serves as the user interface for students and instructors.
2. **Backend (Node.js/Express):** Manages authentication, business logic, and request routing.
3. **External Services:**
    a. **Judge0 API:** Provides an isolated sandbox for secure code execution.
    b. **AI Service (OpenWebUI):** Manages LLM interactions for hint generation and question variations.
4. **Data Layer (PostgreSQL):** Handles persistent storage for user data (SQL) and flexible AI logs (JSONB).

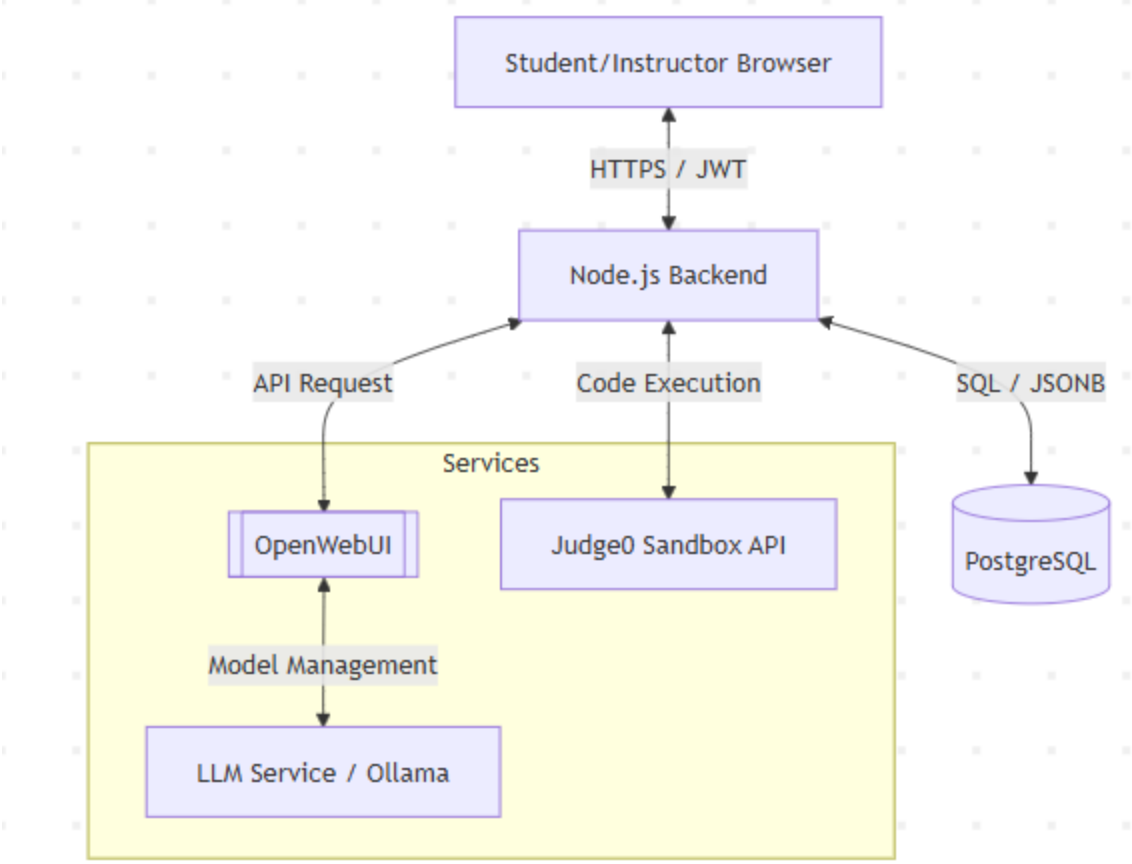

Figure 1: High-Level Architecture illustrating the central role of the Node.js backend in coordinating Frontend, Database, AI, and Sandbox services.

1.2. Operational Workflow

While the static architecture of the system is established on a centralized structure as illustrated in Figure 1, the dynamic process, ranging from a student's interaction with a problem to the instructor's retrieval of analytical data, requires operational synchronization between components. Figure 2 summarizes the end-to-end data flow and user workflows of the system.

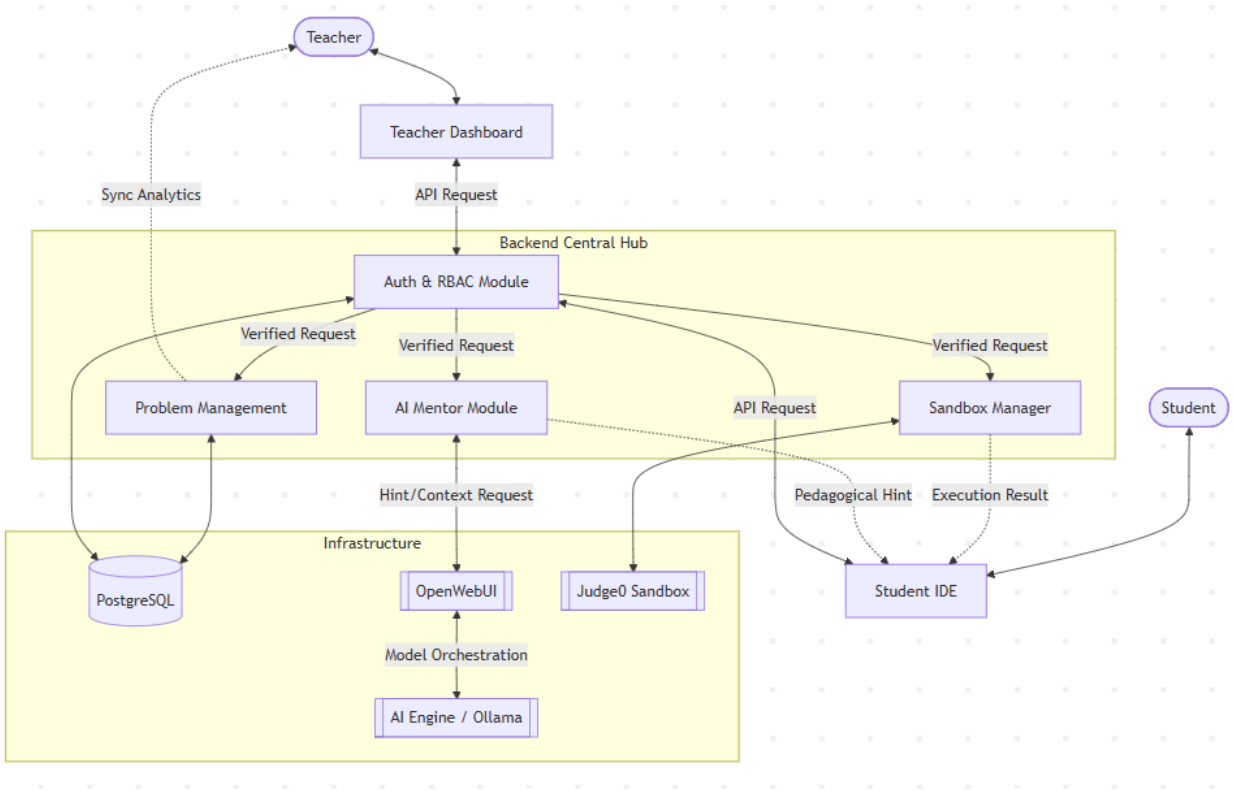

Figure 2: End-to-End Operational Flow.

This diagram illustrates how instructor and student workflows are coordinated through the Backend Central Hub; specifically showing the concurrent operation of the AI Mentor module alongside code execution within the Sandbox environment.

2. System Modules

The backend logic is partitioned into five distinct modules to address specific functional requirements defined in the project scope.

2.1. Identity & Access Management (IAM) Module

Responsible for secure user entry and role management.

- Authentication: Implements JWT (JSON Web Tokens) for stateless authentication, handling secure login and session validation.
- Access Control (RBAC): Enforces permissions to distinguish between "Student" (limited access) and "Teacher" (administrative access) roles.
- Exam Security: Manages Testing Mode Variables, which act as flags to toggle AI features on or off during live exams to prevent academic dishonesty.

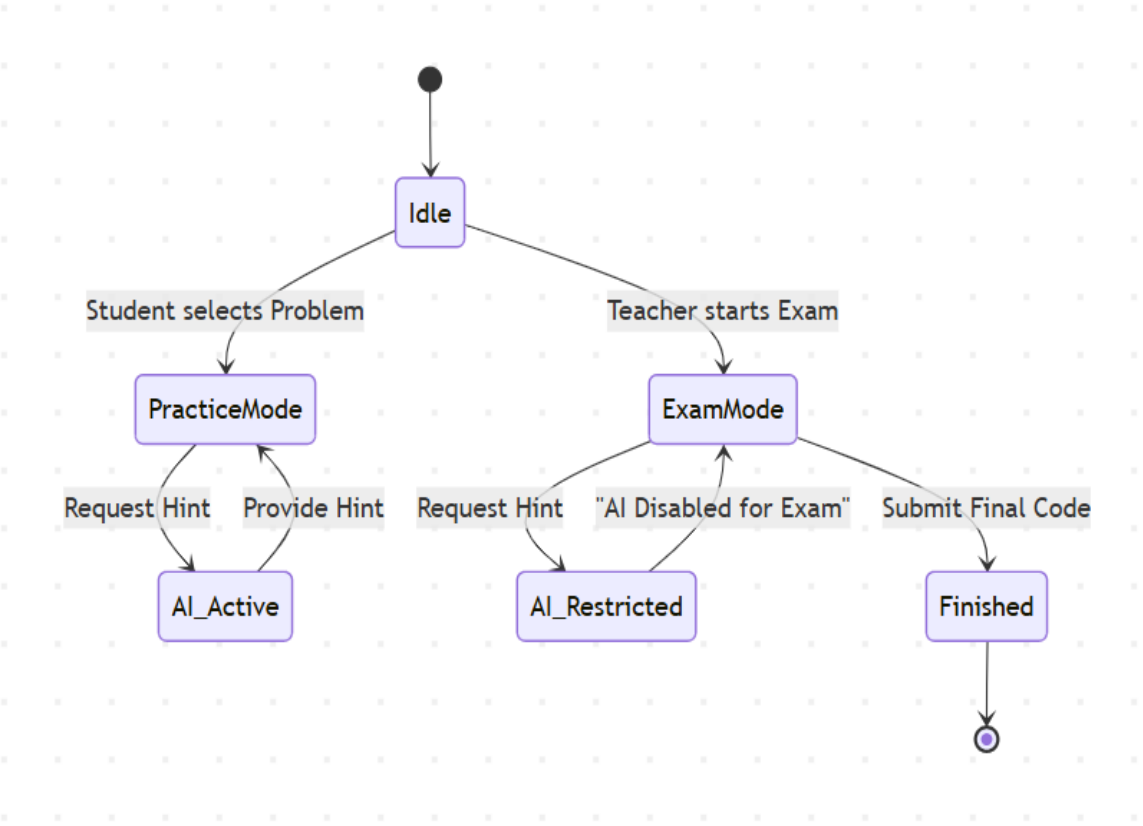

Figure 3: State Transition of Practice and Exam Modes.

Status diagram showing the system's adoption status based on student and teacher actions, and the mode-based activity status of the AI mentoring service.

2.2. AI Mentor & NLP Engine Module

The core pedagogical engine that safeguards against solution leakage.

- **Hint Validation:** Implements a "Tutor/Student" simulation loop. It generates a hint via a "Tutor AI" and validates it using a "Student AI" to ensuring the hint does not reveal the direct answer.
- **Prompt Registry:** Manages a **Prompt Configuration Registry**, storing standardized templates to keep the AI in a strict 'mentor' role.
- **Logging:** Records full **Interaction History** for every student-AI chat session to allow instructors to audit the guidance provided.

2.3. Code Execution & Sandbox Module

Manages the insecure nature of user-submitted code.

- **Isolation:** Relays code snippets and hidden test cases to the **Judge0 API**, ensuring execution happens in a secure, sandboxed environment.
- **Result Parsing:** Processes raw execution outputs (stdout, stderr), memory usage, and execution time to provide feedback to the student.

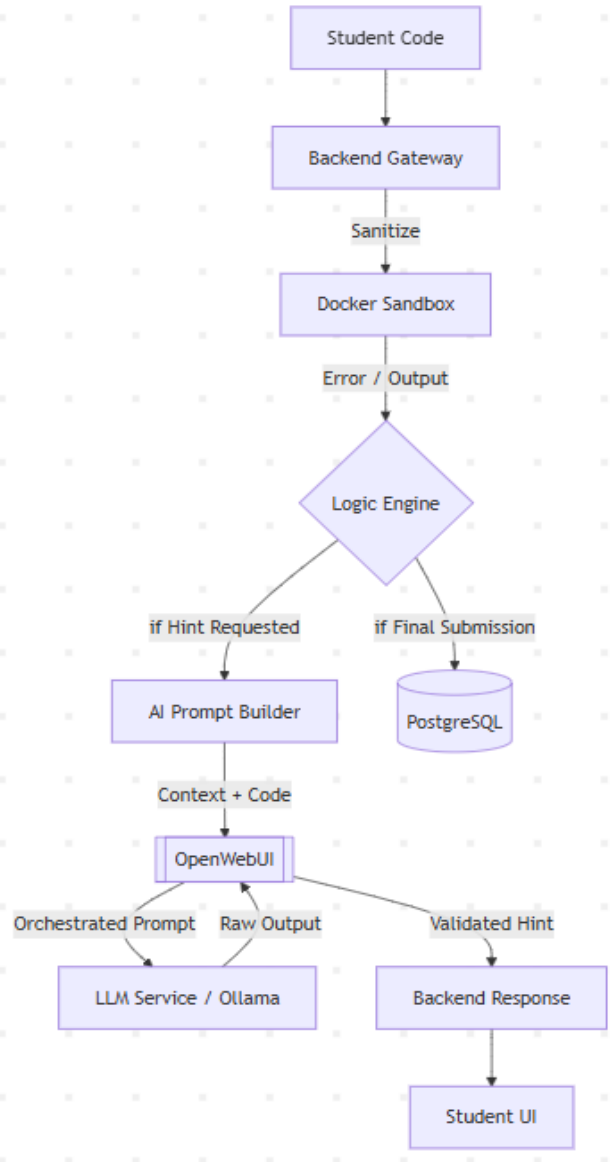

Figure 4: Data Flow Diagram illustrating the lifecycle of a student submission.

The process begins with code sanitization at the Backend Gateway, followed by execution in an isolated Docker Sandbox. The Logic Engine then determines whether to route the resulting error/output to the AI Prompt Builder for pedagogical hint generation or to persistent storage in PostgreSQL for final evaluation.

2.4. Teacher Assistant Module

Automates content creation for instructors.

- Structural Analysis: Uses an Abstract Syntax Tree (AST) parser to analyze the logical structure (loops, conditions) of the instructor's reference code.
- Variation Generation: Sends this structural data to the AI service to generate semantically similar problem variations or conceptual questions.
- Rubric Generation: Suggests grading rubrics based on the complexity of the code structure.

2.5. Analytics & Reporting Module

Tracks student progress and system health.

- Version Control: Maintains Versioned Code Snapshots of student submissions, enabling a timeline view of how a solution evolved.
- Performance Metrics: Aggregates data on common error types and submission success rates to help instructors identify struggling students.
- Submission Tracking: Logs final submissions with precise timestamps to verify compliance with deadlines.

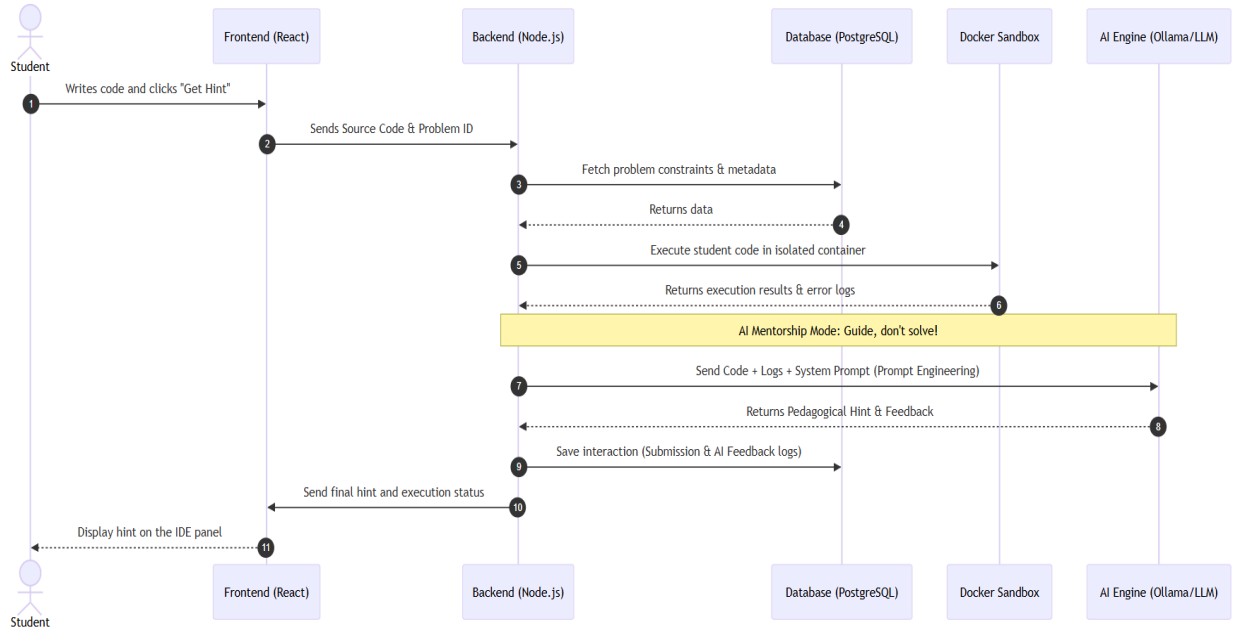

Figure 5: Sequence Diagram illustrating the request/response flow for AI hint generation.

3. API Design

The backend exposes a **RESTful API** to manage resources. The design prioritizes clear separation between student and instructor endpoints.

Core API Endpoints

Method	Endpoint	Description
POST	/api/v1/auth/login	Authenticates credentials and returns a JWT.
GET	/api/v1/problems	Retrieves a list of assigned problems and metadata for the user.
POST	/api/v1/execute	Submits code and test cases to the Judge0 sandbox for execution.
POST	/api/v1/ai/hint	Requests a pedagogical hint; triggers internal validation logic.
POST	/api/v1/teacher/generate	Generates problem variations using AST analysis.

GET	/api/v1/student/history	Fetches versioned code snapshots and interaction logs.
PATCH	/api/v1/admin/exam-mode	Toggles **Testing Mode Variables** to enable/disable AI help.

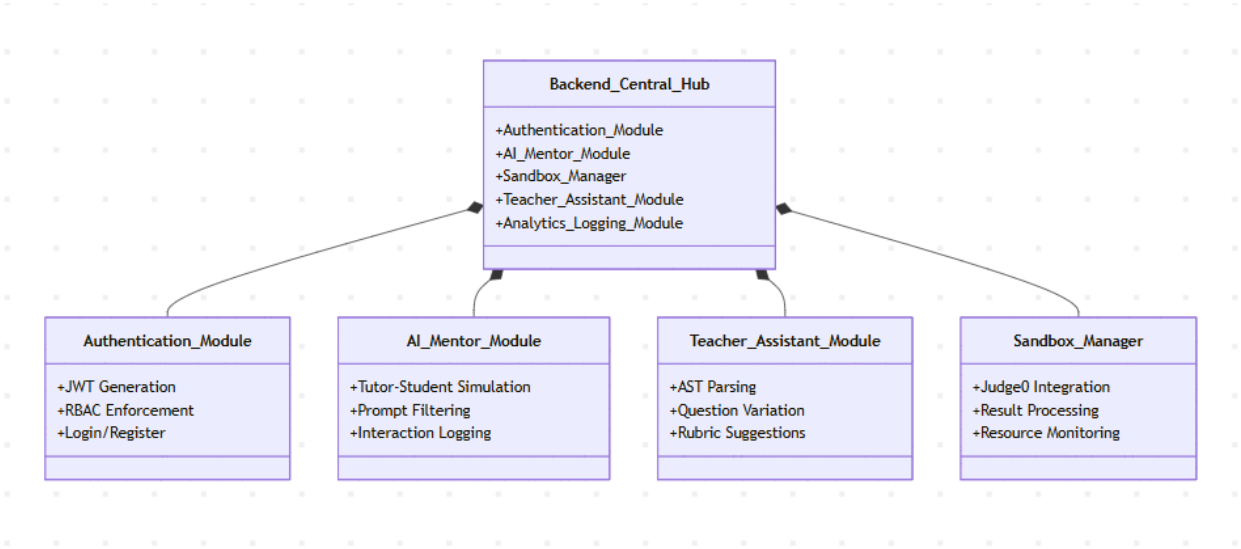

Figure 6: Backend Module Decomposition and Component Architecture.

4. Deployment & Infrastructure

The deployment strategy focuses on **system integrity**, **data privacy**, and **scalability** using containerization and hybrid storage solutions.

4.1. Infrastructure

- **Containerization (Docker):** The Backend, Database, and Frontend are deployed as containerized services. This ensures the system remains available and responsive under varying server loads by isolating dependencies.
- **Judge0 Instance:** The execution sandbox is deployed as a separate service (or accessed via API) to prevent infinite loops or malicious code from crashing the main backend server.

4.2. Database Design

A hybrid database approach is used to balance integrity with flexibility:

- **PostgreSQL (Relational):** Used for **Identity & Access Management** and **Problem Assets**. It ensures **ACID compliance**, which is critical for maintaining accurate academic records and grades.
- **JSONB (Flexible):** Implemented within PostgreSQL to store **AI Prompts**, **Interaction Logs**, and **Dynamic Rubrics**. This allows the system to adapt to the unstructured nature of LLM outputs without altering the database schema.

The database layer is designed to support secure user management, controlled assignment workflows, and auditability of student submissions and AI-assisted interactions. Role-based access control (RBAC) ensures a clear separation between student and instructor privileges.

Figure 7 presents the Entity-Relationship (ER) diagram of the core data model. Users, roles, problems, and submissions are modeled to support versioned code snapshots and detailed submission tracking. AI-assisted interactions are persistently logged to enable instructor review and to preserve academic integrity.

This data model directly supports the system architecture by enabling secure access control, transparent evaluation, and controlled AI mediation.

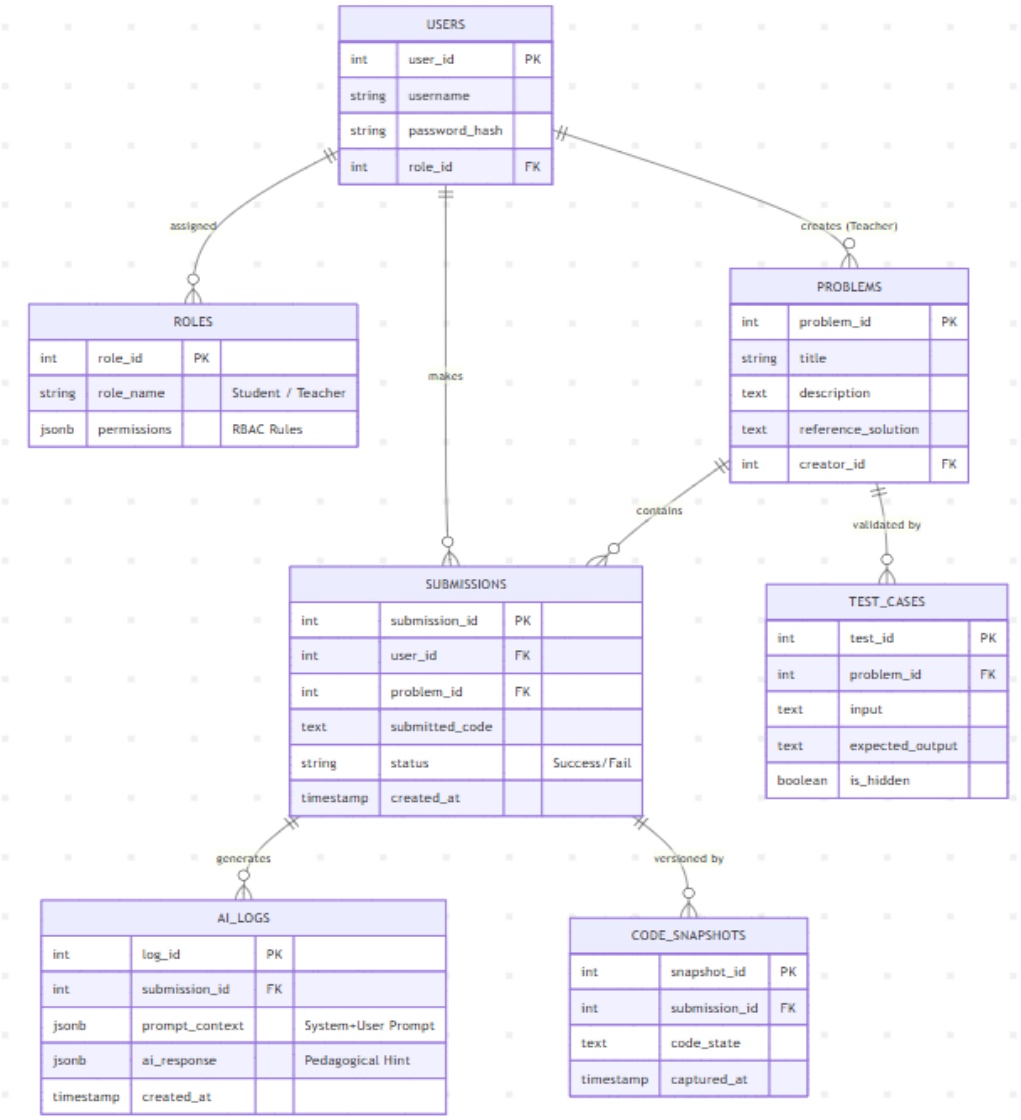

Figure 7: Entity-Relationship Diagram of the Core Data Model

4.3. AI Model Deployment

- **Interface Layer: OpenWebUI** is used as a unified interface to manage model interactions.
- **Hybrid Support:** The architecture supports both **Cloud-based LLMs** and **On-Premise (Local) LLMs** (e.g., via Ollama). This allows institutions to deploy the system locally to ensure that sensitive student data never leaves the campus network.

5. Non-Functional Requirements

Beyond functional requirements, the AI-Assisted Programming Platform is designed to satisfy key non-functional requirements that ensure reliability, scalability, security, and maintainability in academic environments, particularly during high-concurrency scenarios such as laboratory sessions and live examinations.

Scalability: The system supports multiple concurrent users, especially during exams, through containerized deployment that allows horizontal scaling of backend and sandbox services without architectural changes.

Performance: Performance-critical operations, including AI inference and code execution, are isolated from core backend logic. Non-blocking I/O and asynchronous request handling minimize latency, while reusable prompt templates reduce processing overhead.

Availability: Core functionalities such as problem access, code submission, and result storage remain operational even if auxiliary services like the AI inference engine become temporarily unavailable, ensuring uninterrupted exam execution.

Security and Academic Integrity: All sensitive operations are mediated by the backend central hub, enforcing role-based access control, exam mode constraints, and comprehensive audit logging to prevent misuse and maintain fairness.

Maintainability: A modular backend architecture enables independent evolution of system components, simplifying debugging, feature extension, and long-term maintenance.

Observability: Structured logs and interaction histories, particularly for AI-assisted sessions, support instructor auditing, troubleshooting, and post-exam analysis.

5.1 Failure Scenarios & Mitigation Strategies

The system is designed to handle partial failures gracefully, ensuring controlled degradation and recovery without compromising academic integrity.

AI Service Unavailability: If the AI inference service becomes unreachable, AI-assisted features are disabled while the rest of the platform remains functional, and users are informed accordingly.

Hint Validation Failure: AI-generated hints that fail internal validation checks are blocked or replaced with generic conceptual guidance to prevent solution leakage.

Sandbox Execution Timeout or Resource Exhaustion: User code exceeding execution time or memory limits is safely terminated, and a controlled error response is returned without exposing internal details.

Database Connectivity Issues: Transient database failures are handled through retry mechanisms or fallback read-only behavior, preserving data consistency and submission integrity.

Authentication or Authorization Errors: Invalid or expired authentication tokens result in immediate request rejection and enforced re-authentication.

Abuse Prevention During Exams: Rate limiting and request throttling are applied to AI-related endpoints during exam mode to prevent excessive hint requests and resource misuse.

6. Design Decisions & Rationale

- Why Node.js? Chosen for its non-blocking I/O capabilities, which are essential for handling concurrent connections to the AI Service, Sandbox, and Database without latency.
- Why Separate Sandbox (Judge0)? Decoupling code execution from the main backend is a critical security decision to prevent user-submitted code from accessing file systems or environment variables on the main server.
- Why AST Parsing? Regular Expressions are insufficient for understanding code logic. AST parsing allows the system to "understand" the structure (loops, variable scope) of a problem, enabling the AI to generate logically consistent variations rather than just text replacements.