International Journal on Science and Technology (IJSAT)
E-ISSN: 2229-7677 ● Website: www.ijsat.org ● Email: editor@ijsat.org

Labsense: An Ai Integrated Lab Examination Evaluation System

Kandregula Abhiram[1], Mandadi Sainadh Reddy[2], Dr. P. Poornima[3], Dr. K. Rajitha[4], Dr V. Subba Ramaiah[5]

[1, 2] Student, Department of Computer Science and Engineering, Mahatma Gandhi Institute of Technology, Gandipet, India.
[3,4,5] Assistant Professor, Department of Computer Science and Engineering, Mahatma Gandhi Institute of Technology, Gandipet, India.

Abstract
LabSense is an AI-integrated online coding lab examination system addressing critical challenges in Computer Science education where students resort to memorizing or copying code since traditional evaluation systems focus solely on final output, failing to assess logic, effort, and error tolerance. The system overcomes limitations of existing methods that are easily fooled by code variations, miss semantic similarity checks, or require extensive training. LabSense employs a Large Language Model (LLM) to evaluate code logic and semantics, implementing a multi-component scoring system that assesses code quality, logic similarity using semantic analysis, and test case performance, awarding marks fairly even with different coding styles while providing comprehensive AI-generated feedback. The platform features intelligent question allocation with visualized lab layout mapping by assigning unique questions to neighbouring systems, real-time tab monitoring, full screen enforcement, and comprehensive anti-cheat mechanisms, supporting multiple programming languages. It also implements a multi-tenant architecture for scalable institutional deployment. In short, LabSense aims to promote genuine learning and logical understanding in coding lab examinations.

Keywords: AI, evaluation, LLM, assessment, programming, visualized, feedback, multi-tenant

1. Introduction

Programming forms the core of Computer Science education and is assessed through lab practicals meant to test logic and problem-solving skills. However, many students resort to memorizing or copying code to pass exams, as even minor execution errors can lead to failure. This undermines genuine learning and logical understanding, creating a need for a smart, AI-integrated evaluation system that evaluates logic, effort, and error tolerance beyond just final output. LabSense is an AI-powered online coding lab examination system designed to address these challenges. The platform enables faculty to conduct coding exams with automated evaluation, real-time anti-cheat monitoring, and comprehensive AI-generated feedback. Students receive detailed insights into their code quality, logic, and test case performance in a secure, monitored environment. By leveraging Large Language Models (LLMs) and automated test case execution [1], LabSense provides consistent, fair, and educational feedback that goes beyond mere output

verification, evaluating the underlying logic and problem-solving approach while significantly reducing faculty workload.

1.1 Problem Definition

LabSense is an AI-powered platform designed to overcome the critical limitations of traditional coding lab examinations. Conventional systems often encourage students to memorize or copy code rather than truly understand it, while manual grading remains time-consuming, subjective, and difficult to scale across large classes. Feedback is typically limited to pass/fail scores, offering little guidance on logic, code quality, or areas for improvement. Academic integrity is also a growing concern, especially in online settings where students can easily access external resources or collaborate without detection. LabSense addresses all of these issues by automating evaluation, providing rich and meaningful feedback that assesses logic and effort beyond just output correctness, enforcing integrity through unique question sets and real-time tab monitoring, and supporting multiple programming languages within a scalable, cost-effective infrastructure suitable for institutions of any size.

1.2 Existing Applications

Existing coding evaluation systems each fall short in key areas. Token-based tools like MOSS and JPlag are easily bypassed by simple code modifications, while AST-based methods miss semantic similarities when the same logic is expressed differently. Graph-based approaches are too complex and slow for real-time exam use, and machine learning models require extensive training data and struggle with unseen coding styles. Popular platforms like HackerRank and LeetCode are built for competitive programming rather than education, lacking meaningful feedback, anti-cheat mechanisms, and institutional support. LMS plugins offer only basic code execution without intelligent evaluation. LabSense fills these gaps by combining AI-powered semantic evaluation, comprehensive anti-cheat features, multi-tenant architecture, and cost-effective multi-language support into a single platform built specifically for educational institutions.

1.3 Proposed Application

LabSense is a comprehensive web-based platform that modernizes coding lab examinations through AI-powered evaluation, intelligent proctoring, and scalable multi-institutional support. Its multi-component scoring model assesses code quality, logical similarity, and test case performance, ensuring students are judged on understanding and effort rather than just output, while AI-generated feedback provides meaningful guidance beyond simple scores. Academic integrity is maintained through unique question allocation for neighboring systems, full-screen monitoring, clipboard blocking, tab detection, and a three-strike warning system. Supporting multiple programming languages via Judge0 Cloud API with consistent LLM-based evaluation, and built on a multi-tenant architecture with role-based access control, LabSense delivers a fair, scalable, and integrity-driven examination experience for institutions of any size.

2. Literature Survey

"A GPT-Based Code Review System with Accurate Feedback for Programming Education" by Dong-Kyu Lee and Inwhee Joe, proposes a GPT-4o-based system for K-12 programming education with two modules: CRM using Review Necessity Chain to filter trivial submissions and CCM combining test validation with LLM assessment [1]. It avoids direct solutions, provides supportive feedback, detects 42.86% more errors than online judges, aligns 80% with instructor feedback, reduces redundant reviews, and improves learning efficiency while lowering instructor workload.

"Detecting Code Vulnerabilities using LLMs" by Larry Huynh, Yinghao Zhang, Djimon Jayasundera, Woojin Jeon, Hyoungshick Kim, Tingting Bi, and Jin B. Hong, presents an LLM-based framework with prompt engineering for detecting C/C++ vulnerabilities [2]. Using strategies like role-based, zero-shot chain-of-thought, and structured prompting on DiverseVul, results show GPT-3.5 achieved up to 100% F1 improvement, while GPT-4o, Gemini 2.0 Flash, and Llama 3.1 performed better overall. Despite gains, models struggled with vulnerability classification (max F1 = 0.16). GPT-4o achieved a 45.77% success rate in automated patching, highlighting promise but also current limitations in LLM-based software security.

"LLM-Based Code Comment Summarization: Efficacy Evaluation and Challenges" by Peeradon Sukkasem, Chitsutha Soomlek, and Chanon Dechsupa, evaluates BART, Flan-T5, and T5 for summarizing code comments, particularly for self-admitted technical debt (SATD) [3]. While BART generated readable but often irrelevant summaries, T5 showed better ability to capture SATD. The study highlights that current LLMs struggle with coherence, consistency, and relevance, emphasizing the need for improved models to support effective SATD detection and software maintenance.

"Machine Learning Approaches to Code Similarity Measurement: A Systematic Review" by Zixian Zhang and Takfarinas Saber, presents a systematic literature review of 84 studies on machine learning techniques for code similarity tasks like clone detection, plagiarism detection, and recommendation [4]. The review identifies 51 ML algorithms, with Abstract Syntax Tree (AST) representations and BigCloneBench as the most widely used approaches and dataset. The paper consolidates methodologies, metrics, and outcomes while highlighting challenges and future research directions, underscoring the growing role of ML in improving code similarity analysis and software engineering.

"Plagiarism Detection and Similarity Checking Program using Machine Learning and String Matching Algorithm" by Munugapati Bhavana, Dr. Koppula Srinivas Rao, Dr. Gagan Kumar Koduru, S. Parvathi, K. Vijay Krupa Vatsal, and Mogiligidda Sravani, proposes a system that applies machine learning with Knuth-Morris-Pratt and Levenshtein Distance algorithms to detect plagiarism in academic submissions [5]. The system efficiently analyzes text to identify similarities, helping educators maintain academic integrity, reduce plagiarism, and encourage originality. It streamlines evaluation while fostering a culture of honesty and enhancing the overall learning experience.

"AST-Based and Token-Based Neural Networks for Source Code Classification: A Comparative Performance Analysis" ***by Parvathy R and MG Thushara***, compares AST-based and token-based neural network approaches for classifying source code [6]. The AST-based method decomposes large Abstract Syntax Trees into clusters of statement trees, vectorizes them with lexical and syntactical features, and processes them using a bidirectional RNN to capture semantic depth. The token-based method tokenizes source code, vectorizes tokens, and classifies them with a bidirectional RNN, focusing mainly on lexical features. Results show AST-based models capture richer syntactic and semantic information, while token-based models emphasize lexical details, highlighting the trade-offs in accuracy and representation for code classification.

"Can Large Language Model Detect Plagiarism in Source Code?" by William Brach, Kristián Košt'ál, and Michal Ries, evaluates LLMs like GPT-4o, GPT-3.5, LLaMA 3, and CodeLlama for plagiarism detection, showing they outperform traditional tools in handling obfuscation and semantic similarities, with GPT-4o achieving the best results, though higher false positives highlight the need for human oversight [7]. The study emphasizes the role of LLMs in improving accuracy and fairness in plagiarism detection. It contributes to advancing academic integrity by demonstrating the potential of AI-driven plagiarism detection systems.

"Code Review Automation: Strengths and Weaknesses of the State of the Art" by Rosalia Tufano, Ozren Dabić, Antonio Mastropaolo, Matteo Ciniselli, and Gabriele Bavota, evaluates deep learning–based code review automation, focusing on code-to-comment and code & comment-to-code tasks [8]. By manually inspecting 2,291 predictions, the study finds success in simple edits but poor performance in complex logic due to dataset issues. A comparison with ChatGPT shows specialized models perform better in structured tasks, yet both struggle with real-world complexity.

3. Design and Methodology

The development of LabSense follows a systematic, user-centered design approach built on three core principles: educational effectiveness, ensuring the platform promotes genuine learning by addressing real challenges in coding lab examinations; practical feasibility, focusing on features that can be implemented efficiently; and extensibility, establishing a scalable foundation that can be enhanced over time. This philosophy combines iterative development practices with modular architecture to deliver a robust and adaptable educational assessment platform.

3.1 Technologies Used

A robust selection of technologies was adopted to ensure performance, scalability, and ease of maintenance:

Hardware Requirements

- Minimum Requirements (Development): Quad-core CPU (2.5 GHz or higher), 8 GB RAM (16 GB recommended for LLM local hosting), 10 GB free SSD storage, stable broadband connection (10 Mbps minimum), and 1920×1080 display resolution or higher.
- Server Requirements (VM Deployment): 2–4 vCPUs, 4–8 GB RAM, 20 GB SSD storage, public IP address with ports 80/443 open, running Ubuntu 22.04 LTS or a similar Linux distribution.

- Local LLM Hosting: 8+ core CPU, 16+ GB RAM (32 GB for larger models), minimum 50 GB storage for model files, and an optional NVIDIA GPU with 8+ GB VRAM for significantly improved performance.

Software Requirements

- Backend: Python 3.9 or higher (3.11+ recommended) with pip and venv, Uvicorn 0.30.6 as the web server, and key dependencies including FastAPI 0.115.0, Pydantic 2.9.2, python-jose, passlib[bcrypt] 1.7.4, aiohttp 3.10.0, and asttokens 2.4.1.
- Frontend: Node.js 18.0 or higher (20.0+ recommended) with npm and Vite 5.4.8 as the build tool, supporting ES6+ browsers. Key dependencies include React 18.3.1, TypeScript 5.6.2, Material-UI (MUI) 6.1.7, Monaco Editor 4.6.0, and React Router 6.26.2.
- Local LLM Hosting: Ollama 0.1.0 or higher running the llama3.1:8b model or similar.
- Code Execution: Judge0 Cloud API for multi-language code execution and evaluation.

3.2 Development Process

LabSense was developed through an iterative, modular approach, allowing for continuous refinement and adaptability:

Requirement Analysis: Engaged stakeholders (students, faculty, institutions) to define key modules: AI-powered code evaluation, anti-cheat monitoring, lab layout & proctoring, multi-language support, and multi-tenant institutional management.

System Design: The architecture emphasized a three-tier modular structure — Presentation Layer, Application Layer, and Data Layer — enabling each service to function independently while maintaining centralized data management through shared JWT authentication and JSON-based repositories with multi-tenant isolation.

Module Implementation:

- **AI Evaluation Engine:** Implemented first to establish the core LLM-based code assessment logic using a weighted scoring formula (20% effort + 40% logic similarity + 40% test case performance).
- **Exam Management:** Integrated exam creation, question assignment, server-side timer management, per-question auto-save, and exam versioning.
- **Anti-Cheat System:** Enabled fullscreen enforcement, tab visibility monitoring, clipboard blocking, a three-strike warning policy, and adjacency-based unique question allocation to neighboring systems.
- **Multi-Language Code Execution:** Added Judge0 Cloud API integration supporting Python, JavaScript, Java, C, C++, and Go.
- **Multi-Tenant Architecture:** Implemented college-level data isolation with role-based access control for Students, Faculty, Admins, and Super Admins.

Database Integration: Cloud JSON-based repositories and SQLite-compatible storage were used to store and manage student information, exam data, submission results, session logs, and institutional records, with the design allowing future migration to PostgreSQL or MongoDB.

Testing: Conducted manual and workflow-based testing for each module covering authentication, exam creation, student joining, code submission, evaluation pipeline, and result display. User acceptance testing validated end-to-end workflows and usability across all roles.

Deployment and Feedback: The system was deployed with the backend hosted locally on a laptop exposed via Cloudflare Tunnel, and the frontend hosted on Cloudflare Pages. User feedback from testing guided further optimization of the evaluation engine and interface.

3.3 System Design

LabSense is built on a scalable and maintainable modular architecture incorporating object-oriented programming principles, a centralized data management system, and role-based access controls.

3.3.1 System Architecture

As shown in Figure 1, the three-tier architecture includes:

- **Presentation Layer:** A React and TypeScript-based frontend with role-specific dashboards tailored for Students, Faculty, Admins, and Super Admins, featuring Monaco Editor for code editing with syntax highlighting and real-time feedback.
- **Application Layer (API Server):** A FastAPI and Python-based backend containing specialized modules for Authentication, Exam Management, User Management, Code Evaluation Engine, and Multi-Tenancy Control. It supports multiple LLM providers (OpenAI, Anthropic, Ollama) through a unified interface.
- **Evaluation Engine:** Processes student code submissions through a sequential pipeline — code validation, test case execution via Judge0 Cloud API, LLM-based semantic analysis, weighted score calculation (20% effort + 40% logic + 40% test cases), and AI feedback generation.
- **Data Layer:** JSON-based repositories handling persistent storage for Users, Exams, Submissions, College records, and Sessions, with multi-tenant data isolation enforced at the repository level and designed for future migration to PostgreSQL or MongoDB.
- **External Services:** Judge0 Cloud API for multi-language code execution and LLM providers for AI-powered semantic evaluation and feedback generation.

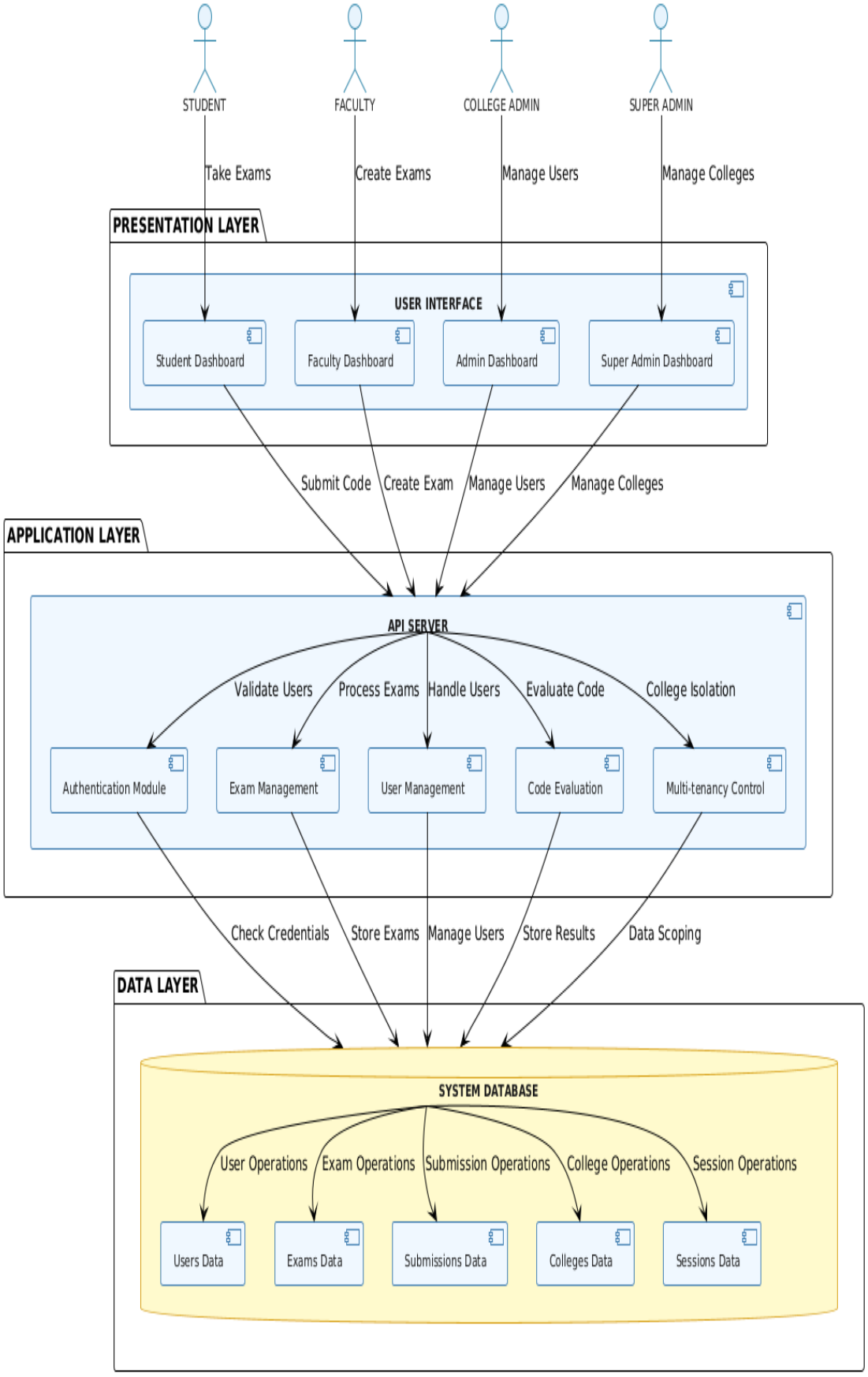

Fig 1: System Architecture Diagram

3.3.2 Database Schema

The system utilizes a JSON-based repository pattern for data storage, designed for future migration to relational or document-based databases such as PostgreSQL or MongoDB. Its schema includes the following collections as shown in the Class Diagram in Figure 2:

- **Users** — Stores user profiles including name, credentials, role (Student, Faculty, Admin, Super Admin), and college/department association, identifiable by a unique user ID.
- **Colleges** — Contains institutional details including college name, department structure, and year/section organization, enabling multi-tenant data isolation across institutions.
- **Departments** — Defines academic departments within each college, with details like department name, associated years, and sections, referencing the parent college.
- **Exams** — Lists all created examinations with details like title, subject, duration, allowed languages, status, and question assignments, referencing the faculty who created them.
- **ExamQuestions** — Records individual questions within each exam including problem statement, ideal solution, test cases, and adjacency-based question allocation mapping, referencing the parent exam.
- **Sessions** — Stores timestamped exam session records for student exam attempts including start time, assigned questions, auto-saved code, violation logs, and submission history, referencing the student and exam.
- **Submissions** — Logs all code submission activities including submitted code, test case results, effort score, logic similarity score, correctness score, final weighted score, and LLM-generated feedback, referencing the involved student and question.
- **Bus_Logs** — Records bus boarding and offboarding events, referencing the student and the specific route.

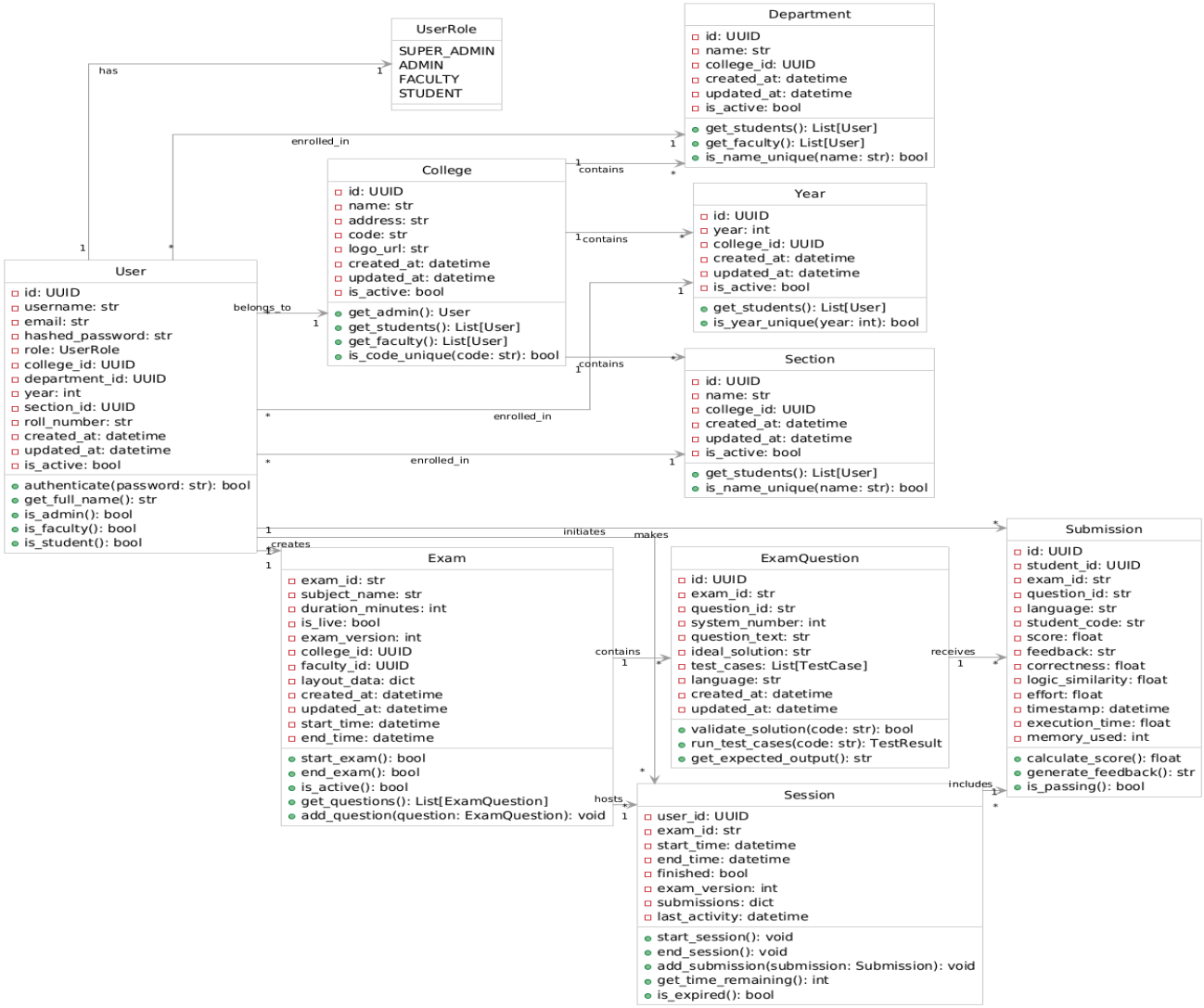

Fig 2: Class Diagram of LabSense

3.3.3 Control Flow

The control flow follows a streamlined pattern for all modules as shown in Figure 3 (Sequence Diagram):

1. **Student Authentication and Session Initialization:** The student logs in via the secure JWT-based authentication interface, and the system validates credentials, verifies role, and creates an active exam session with assigned questions.
2. **Identity Verification and Anti-Cheat Activation**: Upon joining an exam, the system validates the exam start code, enforces fullscreen mode, and activates browser-based monitoring for tab visibility, clipboard access, and fullscreen state changes with a three-strike warning policy.
3. **Action Execution:** The student writes and submits code through the Monaco Editor, triggering sequential processing — code validation, multi-language test case execution via Judge0 API, AST-based structural analysis, and LLM-based semantic similarity and effort assessment.
4. **Database Update and Result Notification:** Submission results, scores, session logs, and LLM feedback are stored in the repository, the session state is updated, and the complete evaluation breakdown is immediately returned and displayed to the student.

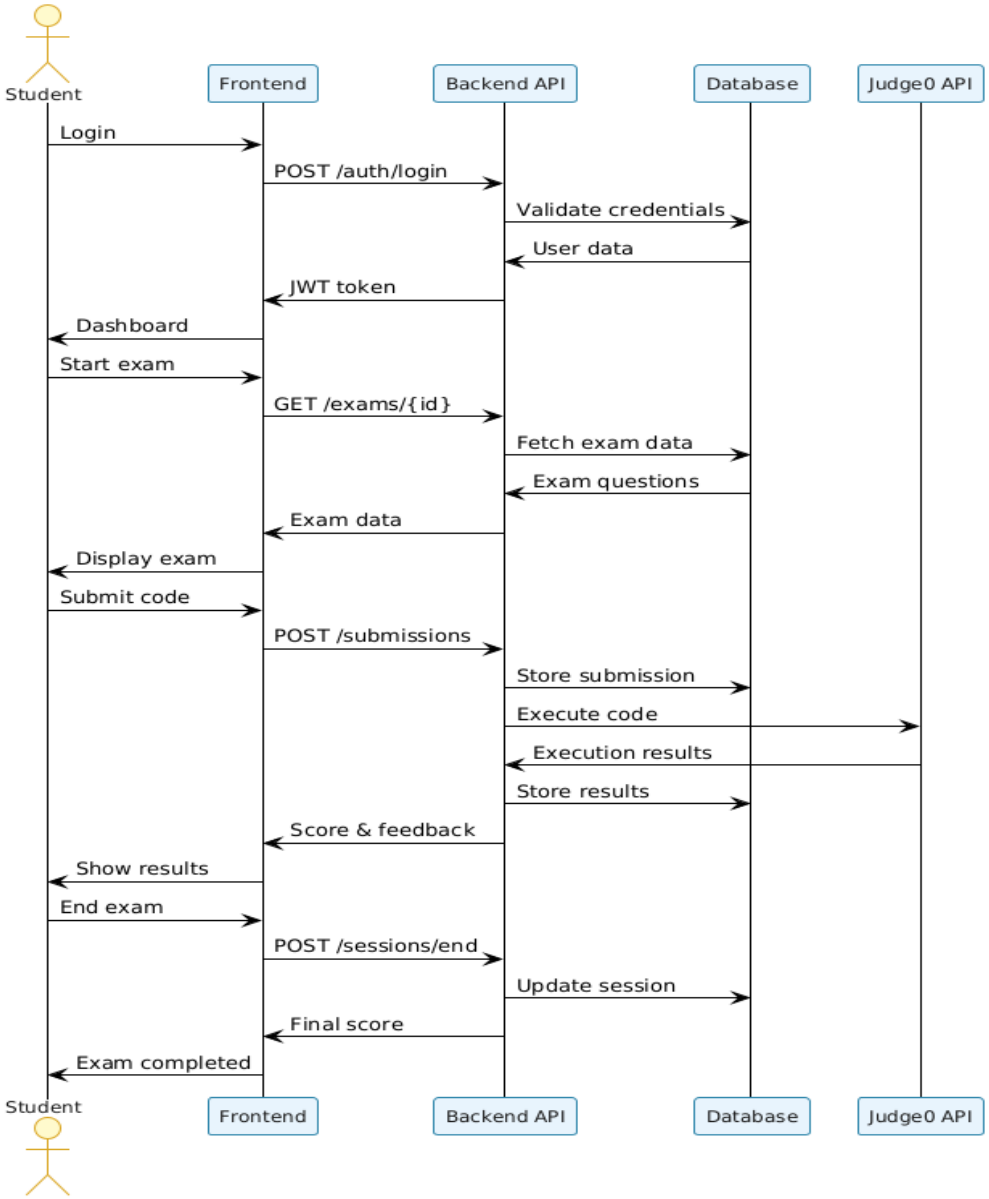

Fig 3: Sequence Diagram of LabSense

3.3.4 Activity Flow

The activity flow details the step-by-step execution within the evaluation pipeline as shown in Figure 4 (Activity Diagram):

1. **Code Submission Trigger:** Student submits code through the exam interface, initiating the evaluation engine.
2. **Test Case Execution:** Public test cases are executed via Judge0 Cloud API or local Python fallback, recording outputs, execution times, and pass/fail status for each case.
3. **Execution Outcome Decision:** A decision point determines whether execution was successful. If unsuccessful, the system logs the execution error and prepares an error response for the student.
4. **Scoring and Semantic Analysis:** If successful, the system calculates the correctness score from test case results, analyzes logic similarity using an AST and LLM hybrid approach, and assesses effort level through LLM evaluation.

5. **Weighted Score Calculation and Feedback**: The final weighted score (20% effort + 40% logic + 40% test cases) is computed, the full marks rule is applied if all test cases pass, and structured AI feedback is generated covering overall assessment, critical observations, and improvement suggestions.
6. **Result Storage and Return:** Both successful and unsuccessful paths converge to store submission results, update the session, and return the complete evaluation response to the student.

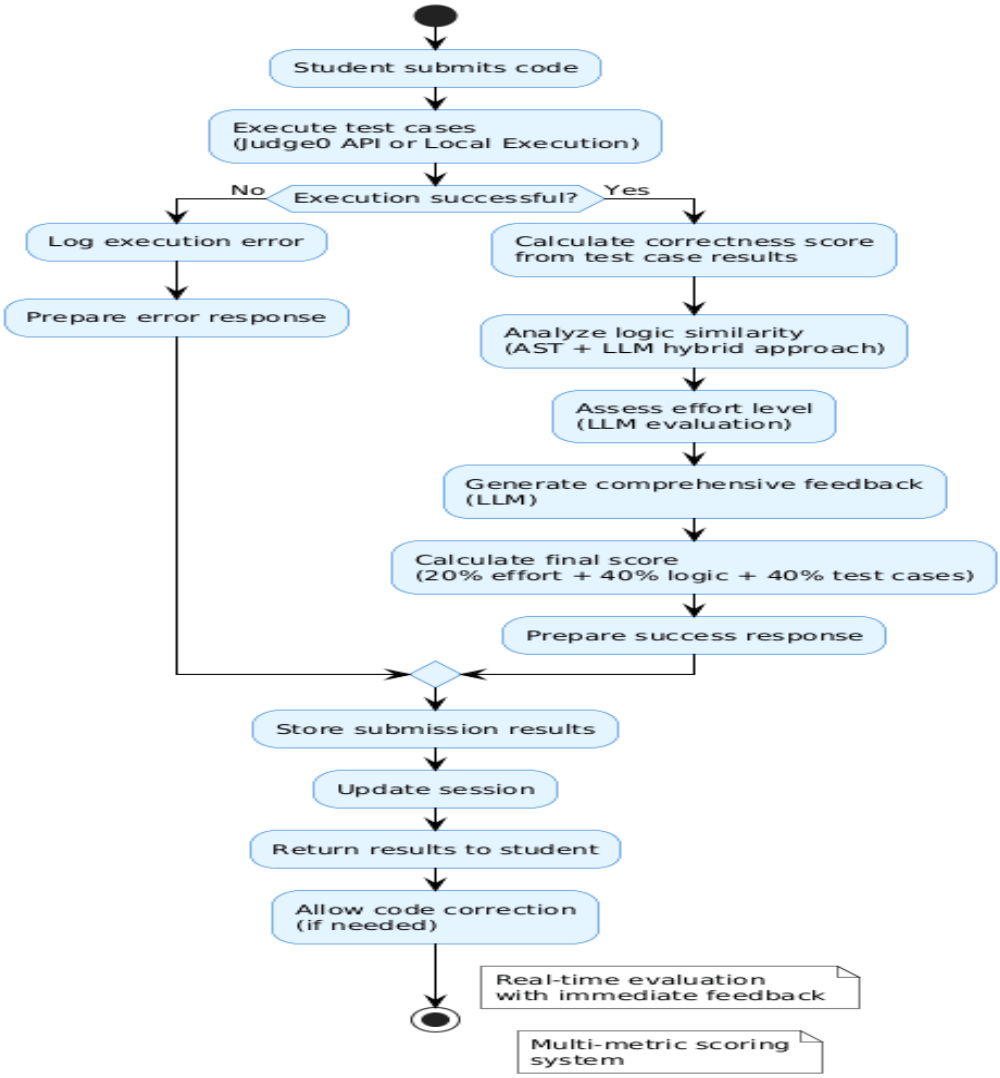

Fig 4: LabSense's Code Evaluation Activity Diagram

3.3.5 Role-Based Access Control

Access is governed by user roles assigned during registration. Each user account is linked to specific permissions as shown in Figure 5 (Use Case Diagram):

- **Student** — Can log into the platform, view available and active exams, join exams using a valid start code, write and submit code through the Monaco Editor, view test case results, access AI-generated feedback, and review their evaluation breakdown including effort, logic similarity, and test case scores.

- **Faculty** — Can create and manage exams, define questions and ideal solutions, assign test cases, configure lab layout and adjacency-based question distribution to neighboring systems, monitor student submissions, and access detailed reports and analytics on student performance.
- **Admin (College Level)** — Can manage institutional user accounts including students and faculty, oversee department and section structures, monitor exam activity within their college, and ensure college-level data isolation is maintained across the platform.
- **Super Admin (Cross-Institutional)** — Has centralized control over all registered institutions on the platform, managing college records, overseeing all admin accounts, monitoring cross-institutional activity, and ensuring scalable multi-tenant data separation across the entire system.

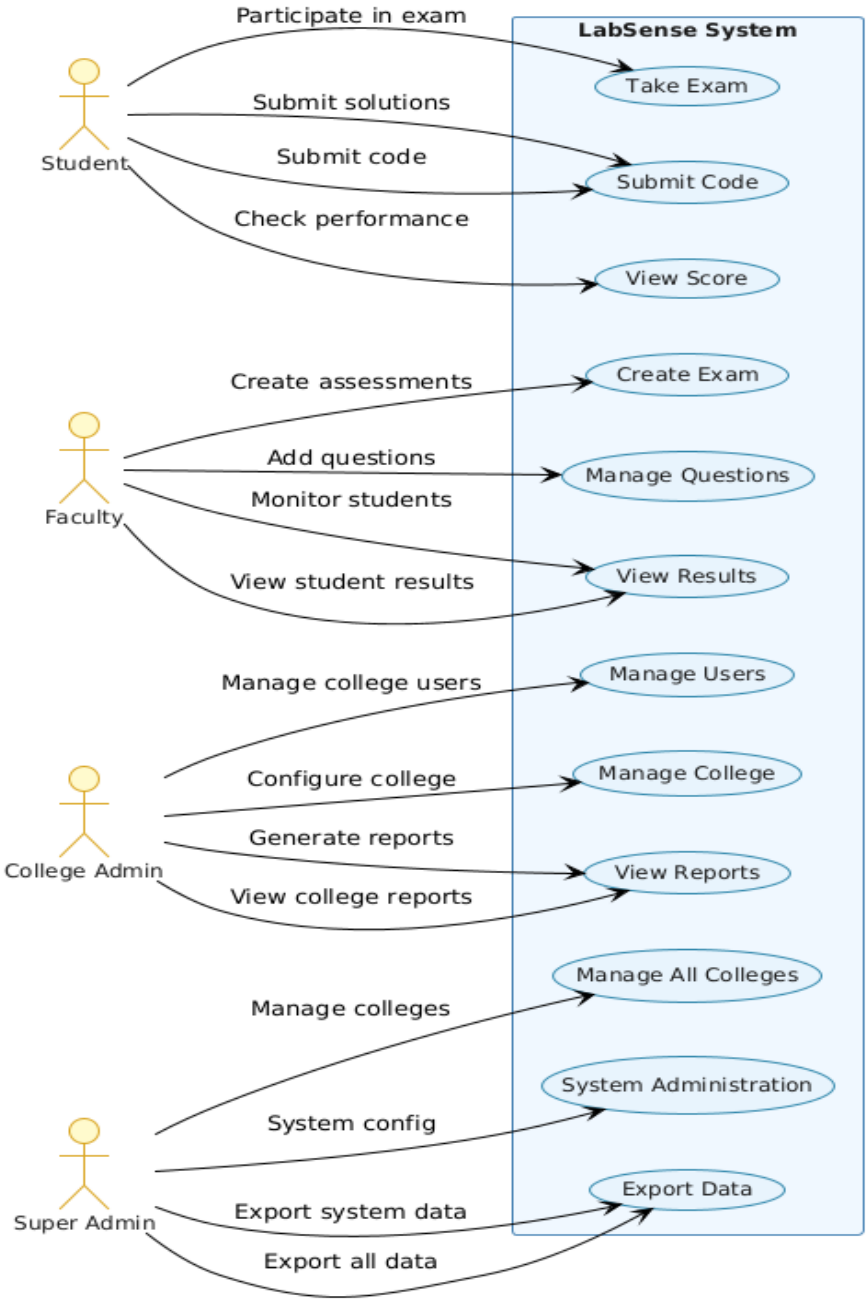

Fig 5: Use Case diagram

4. Implementation

LabSense is architected as a modular web-based application, using React and TypeScript for the frontend and FastAPI with Python for the backend. The codebase is organized into dedicated modules (evaluation engine, exam management, authentication, multi-tenancy), each encapsulated within structured components under the frontend and backend directories, while common logic (JWT authentication, LLM provider selection, code execution, repository operations) resides in shared utility modules for maintainability and reuse.

4.1 Database Integration

LabSense uses a JSON-based repository pattern for persistent data storage, with the design explicitly supporting future migration to relational or document-based databases such as PostgreSQL or MongoDB. The repository layer handles all secure data interactions through abstracted CRUD operations, ensuring business logic remains independent of the underlying storage mechanism.

Collections such as users, exams, submissions, sessions, colleges, and departments are used, with structured documents and role-scoped data access. All data operations are abstracted at the repository level for security and consistency. For instance, a code submission atomically updates the submissions collection, stores the weighted evaluation score, session state, and LLM-generated feedback simultaneously.

The multi-tenant architecture enforces college-level data isolation at the repository layer, ensuring users can only access records within their authorized institutional scope. Role-based access control governs all data interactions, with Student, Faculty, Admin, and Super Admin permissions applied consistently across every data operation.

4.2 LLM and Code Execution Integration

The Judge0 Cloud API is interfaced via HTTP using Python's aiohttp and requests libraries. When a student submits code, the submission is validated and mapped to the corresponding exam question and session record. For logic similarity assessment, an AST-based structural analysis is combined with LLM-based semantic comparison. Code representations are generated and processed during evaluation. During submission, the system captures the student's code, computes its structural and semantic similarity against the ideal solution, and compares it using the LLM evaluator, providing a dual-layer assessment of both code structure and problem-solving approach.

4.3 AI-Powered Evaluation and Recommendation Systems

Pattern-based evaluation and feedback logic is implemented in Python utility functions within the evaluation engine:

Code Evaluation: The system analyzes the student's submitted code against the ideal solution, executes all public test cases, calculates weighted scores, and suggests improvement areas. This logic is triggered upon every code submission during an active exam session. Given below is the core weighted scoring logic:

Python

```
if test_case_score >= 1.0 and public_results and all(public_results):
    final_score = 100.0
else:
    final_score = (effort_score * 0.2 + logic_similarity * 0.4
                + test_case_score * 0.4) * 100.0
```

LLM Feedback Generation: The system queries the student's submitted code and generates structured feedback covering overall assessment, critical analysis, improvement suggestions, and scope for further development. This is implemented as part of the evaluation pipeline that executes on every submission and returns results immediately. Below is the simplified logic for LLM feedback generation:
python

```
llm_evaluator = get_llm_evaluator()

effort_score, effort_breakdown, effort_reasoning = await llm_evaluator.evaluate_effort(
    student_code, ideal_code, language, question_text, test_results_summary
)

ast_sim = get_ast_similarity(language, student_code, ideal_code)

if ast_sim is not None and ast_sim > 0.0:
    llm_sim, logic_breakdown, logic_notes = await llm_evaluator.evaluate_logic_similarity(
        student_code, ideal_code, language, question_text
    )
```

4.4 User Interface

The UI is built using React, TypeScript, and Material-UI, offering a clean, multi-window, role-aware, event-driven interface:

- **Login Screen:** JWT-based secure login for all user roles. Authentication errors and invalid credential attempts are handled gracefully with appropriate feedback messages.
- **Dashboard:** Summarizes active exams, submission history, evaluation scores, and LLM feedback. Visual indicators highlight key alerts such as anti-cheat warnings, exam timer countdowns, and submission status.
- **Module Windows:** Each module has dedicated forms, tables, and panels for interaction. For instance, the student results page shows the full evaluation breakdown including effort score, logic similarity, test case performance, and structured AI feedback sections.
- **Role-Based Interface:** The UI dynamically adapts to the logged-in role, showing relevant features. Faculty get access to exam creation, question management, lab layout configuration, and student analytics. Admins get access to user registration, department management, and institutional oversight.

- **Accessibility:** Monaco Editor provides syntax highlighting and code completion. Fonts, icons, and colours are optimized for readability and usability. Feedback is provided via real-time test case outputs, score chips, and structured feedback panels.

4.5 Security

Security is enforced at multiple levels:

- **JWT Authentication** for user identification and session management across all roles and API endpoints.
- **Role-Based Access Control** for all platform operations, ensuring Students, Faculty, Admins, and Super Admins can only access their authorized features and data.
- **Anti-Cheat Enforcement** during exams using fullscreen monitoring, tab visibility detection, clipboard blocking, and a three-strike warning policy. Adjacent workstations are assigned different questions to prevent direct copying.

All data transmission to the backend is secured via HTTPS, and college-level data isolation is enforced at the repository layer to prevent unauthorized cross-institutional access.

4.6 Error Handling and Reliability

The application includes robust error handling for external service failures (e.g., Judge0 API unavailability with local Python fallback), LLM provider unavailability (e.g., automatic fallback between OpenAI, Anthropic, Gemini, and Ollama providers), and invalid user actions (e.g., duplicate submissions, expired exam sessions, insufficient permissions). User-friendly error messages are displayed on the interface, and session logs are maintained for faculty review and troubleshooting. The modular design allows for independent development, debugging, and extension of each component without affecting the rest of the platform.

5. Testing and Results

The LabSense system underwent thorough interface-level testing across all modules to evaluate functionality, usability, and reliability. Each module was assessed for correct behaviour in real-world scenarios using simulated student and faculty data. Below are the major interfaces and their testing outcomes:

5.1 Login and Authentication Interface

The login page was tested to verify secure role-based access for all user types — Students, Faculty, Admin, and Super Admin. The JWT-based authentication mechanism correctly restricted access to role-appropriate routes. Users attempting to access unauthorized routes were denied, confirming that the authentication and authorization layers functioned as designed.

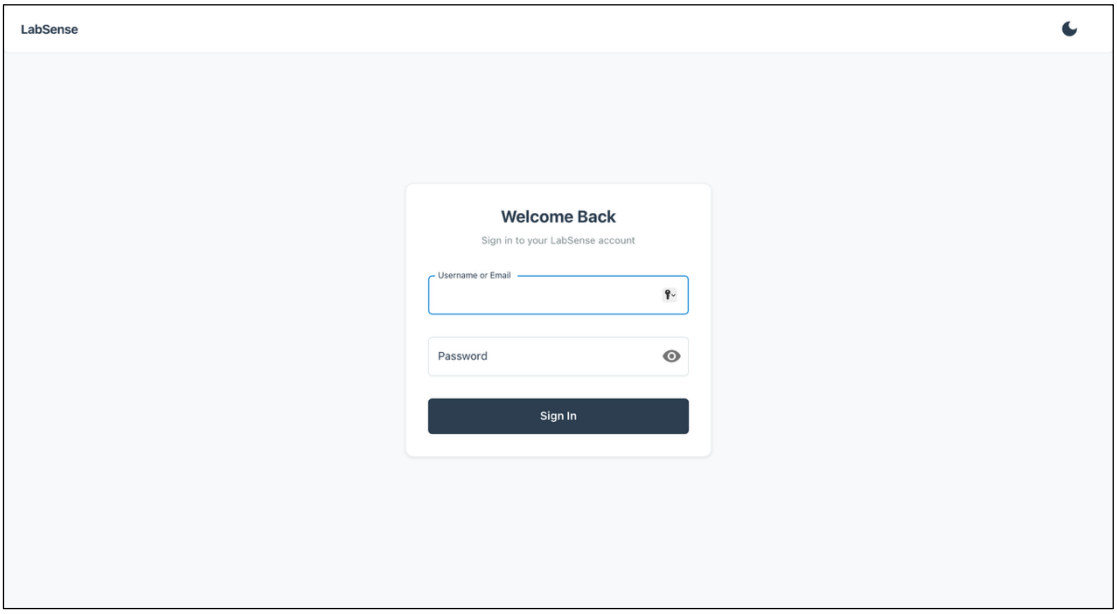

Figure 5.1: Login page showing the authentication interface

5.2 Super Admin Dashboard

The Super Admin Dashboard was validated for cross-institutional management capabilities. The interface correctly displayed and managed data across multiple colleges, confirming that multi-tenant isolation and hierarchical control were functioning. Administrative operations such as managing institutional records were carried out without data leakage between colleges.

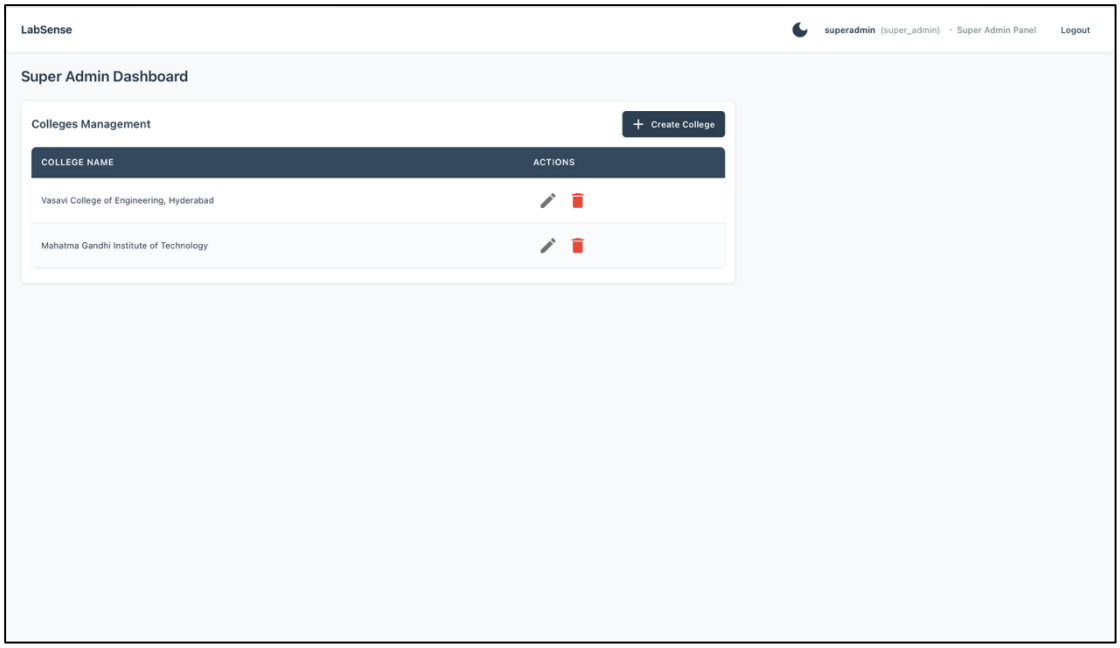

Figure 5.2: Super Admin Dashboard for Cross Institutional Management

5.3 Admin Dashboard

The Admin Dashboard was tested for institutional-level user management. The interface correctly displayed user details including roles, departments, and sections. Account creation, update, and role

assignment operations were verified to work within college-scoped boundaries, ensuring data isolation between institutions.

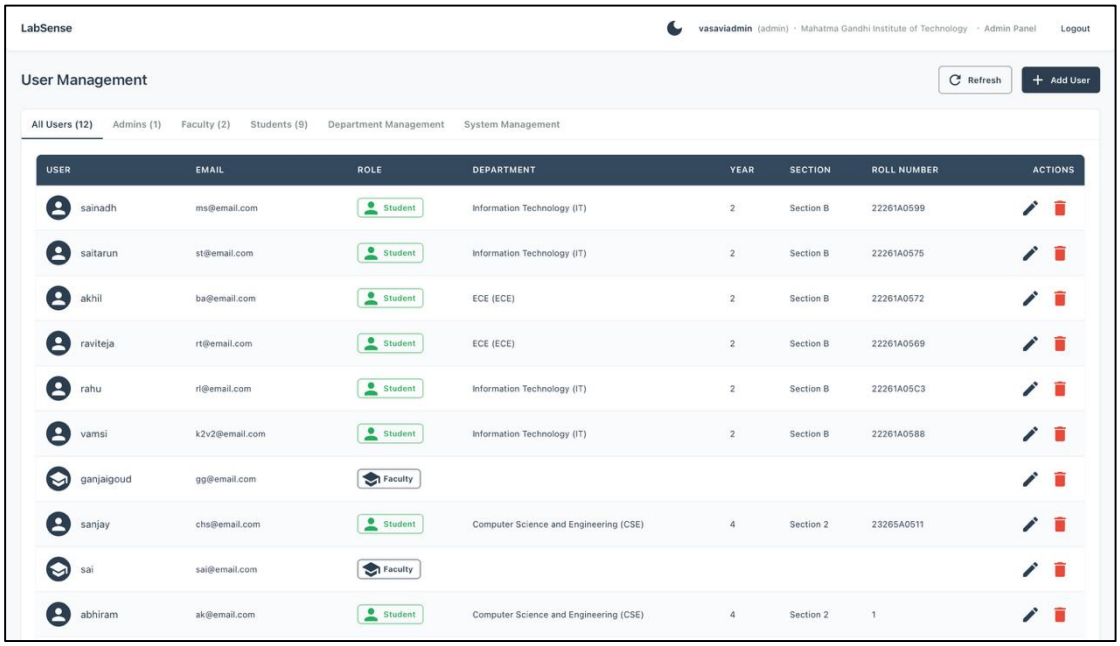

Figure 5.3: Admin Dashboard for Institutional Management

5.4 Exam Creation Interface (Faculty Dashboard)

The exam creation tab on the Faculty Dashboard was tested by defining exams with parameters such as subject, duration, and student access settings. Multiple questions were added, test cases were assigned, and the resulting exam data was confirmed to be retrievable through the student-facing APIs, validating the end-to-end exam setup workflow.

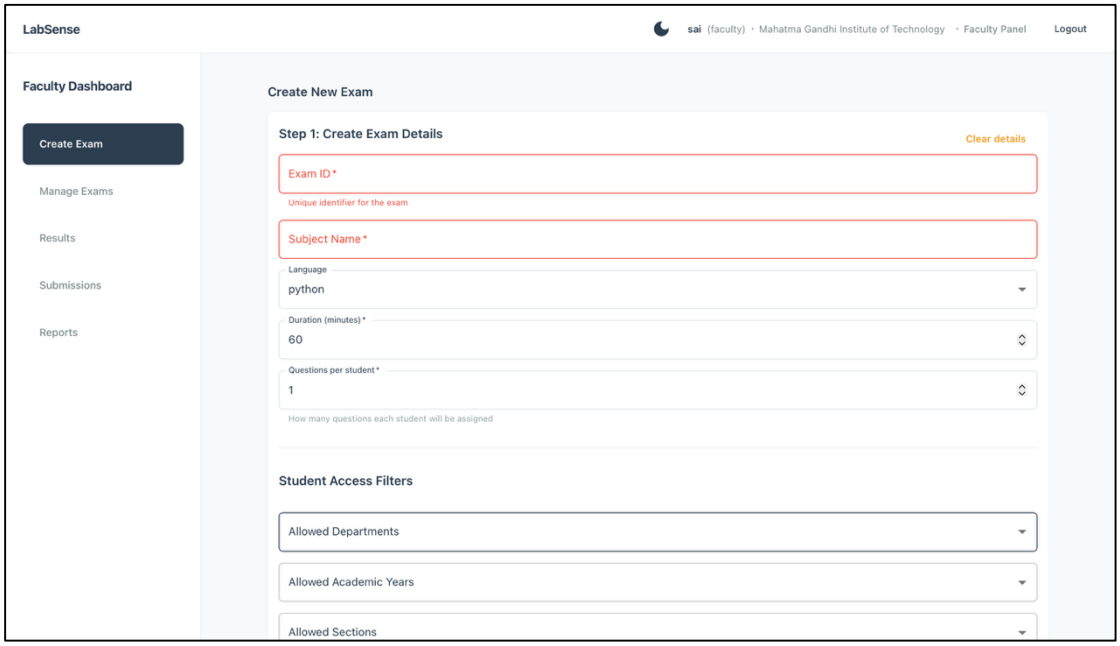

Figure 5.4: Exam Creation tab on Faculty Dashboard

5.5 Lab Layout and Question Assignment Screen

The lab layout and question assignment interface was tested to verify that unique questions were correctly assigned to neighbouring systems in the configured lab setup. The adjacency-based distribution algorithm functioned correctly, ensuring that students seated next to each other received different questions, thereby reducing the risk of direct copying.

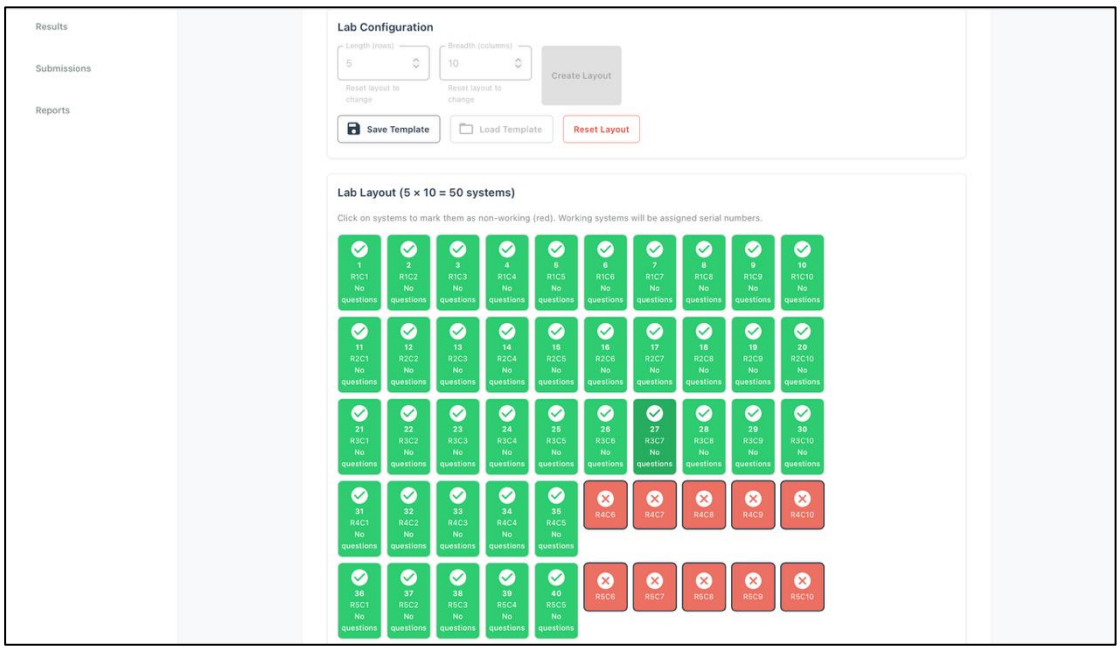

Figure 5.5: Lab layout and question assignment screen on Faculty Dashboard

5.6 Student Dashboard – Exam List

The Student Dashboard was tested to confirm that available and active exams were displayed correctly based on the student's college, department, and section. Students could successfully view and join active exams using the start code, and the session creation logic was validated to correctly initialize timed exam sessions.

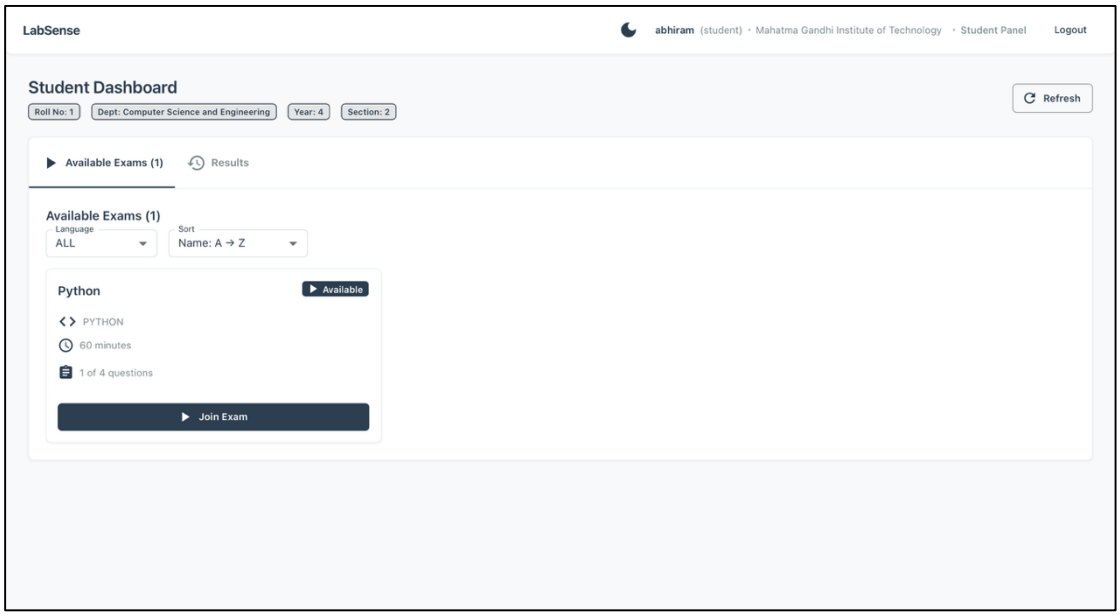

Figure 5.6: Student Dashboard showing the available or active exam list

5.7 Student Exam Interface – Monaco Editor with Anti-Cheat

The student exam interface was tested with full code editing functionality using the Monaco Editor, which provided syntax highlighting and code completion. Anti-cheat mechanisms including fullscreen enforcement, tab visibility monitoring, and clipboard restrictions were verified to trigger correctly. The three-strike warning system activated as expected when violations such as tab switching or fullscreen exit were detected.

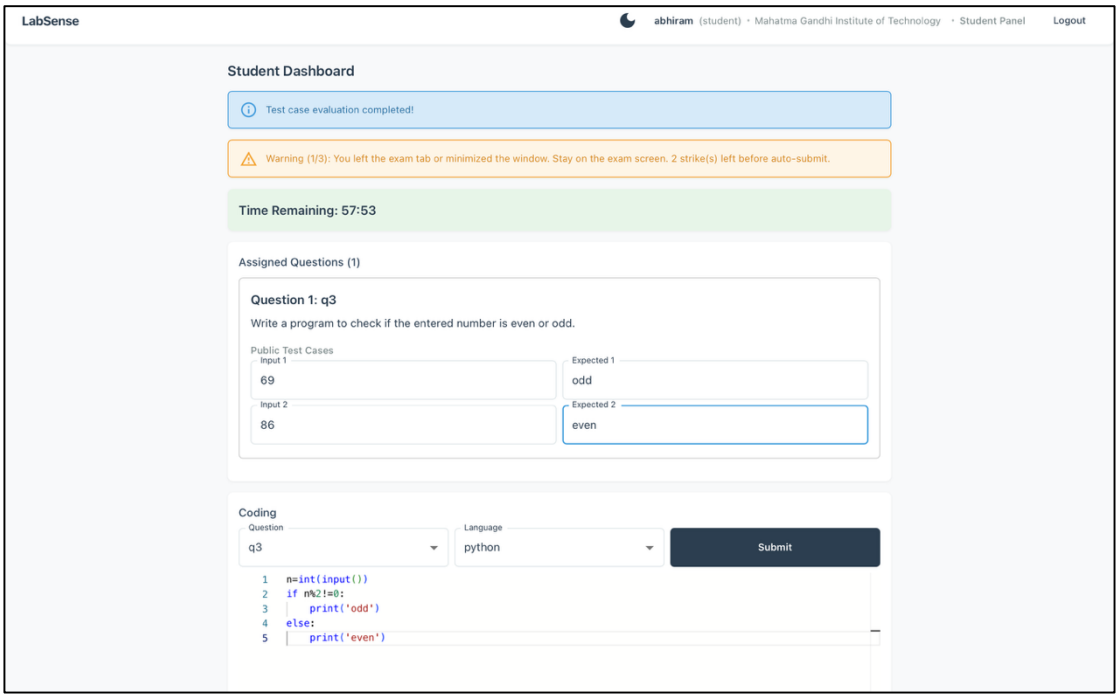

Figure 5.7: Student exam interface with Monaco Editor, question display and Anti-cheat warning

5.8 Code Submission and Test Case Output

The code submission workflow was tested by submitting sample solutions across multiple programming languages. The evaluation engine executed public test cases via the Judge0 Cloud API, returned detailed outputs including pass/fail status, actual vs. expected output, and execution time. Correct submissions received full marks, while partial solutions received proportionate scores, confirming the weighted evaluation formula (20% effort + 40% logic + 40% test cases) functioned as intended.

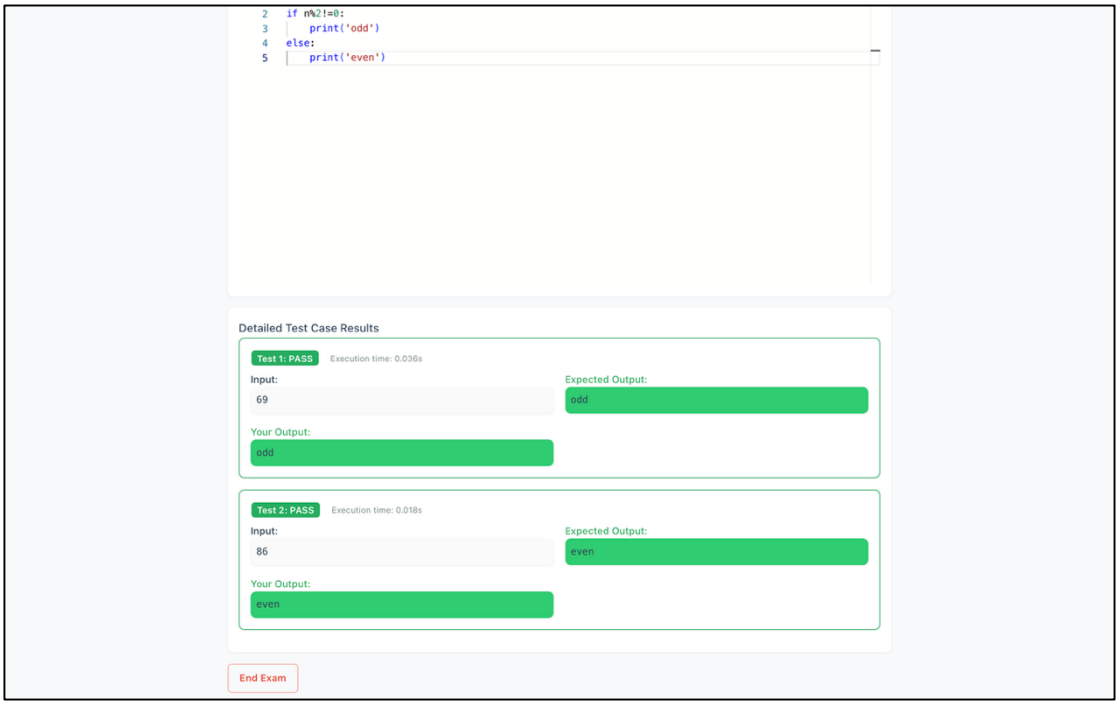

Figure 5.8: Question submission result test-case output

5.9 Student Results Page – Score and LLM Feedback

The results page was tested to verify that the final score, evaluation breakdown, and AI-generated feedback were all rendered correctly. The LLM feedback sections — overall assessment, critical analysis, improvement suggestions, and scope for improvement — were displayed in a structured and readable format. Students could clearly understand how effort, logic similarity, and test case performance contributed to their final score.

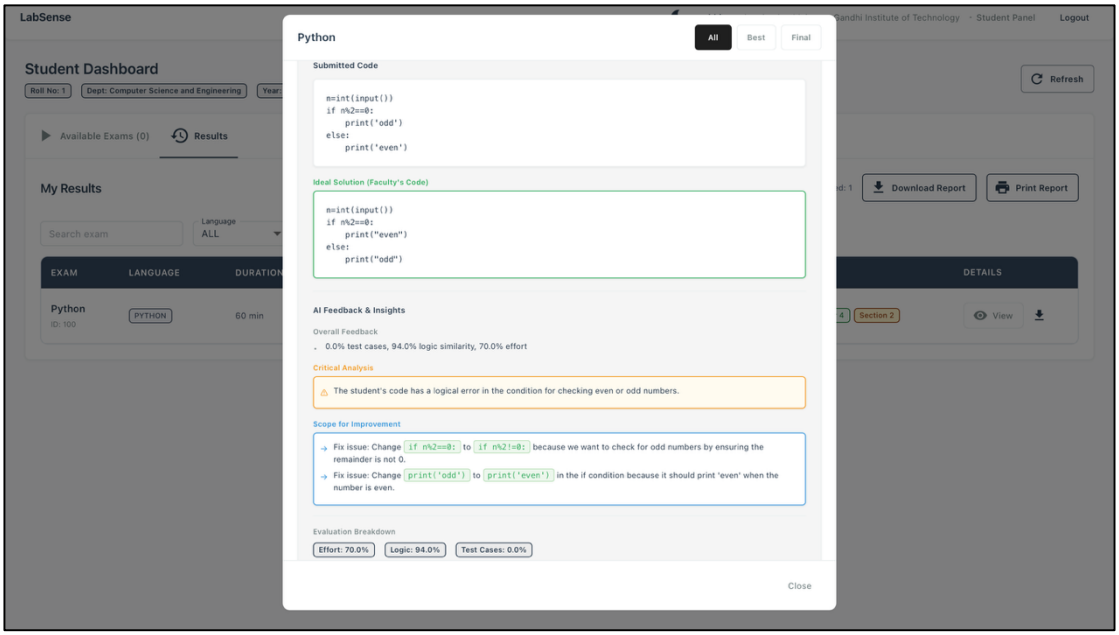

Figure 5.9: Student results page with final score and the LLM feedback sections

5.10 Faculty Dashboard – Reports and Analytics

The faculty-facing reports and analytics interface was tested to confirm that student submission data, performance metrics, and evaluation summaries were displayed accurately. The dashboard presented visual summaries of exam results, enabling faculty to assess class performance and identify students requiring additional support.

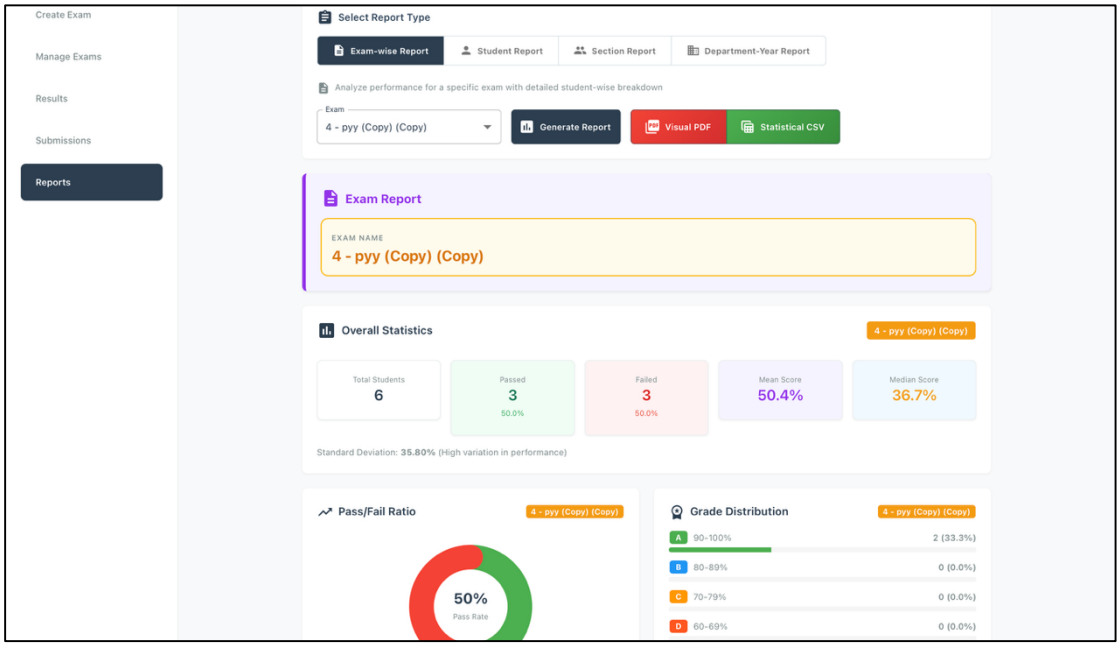

Figure 5.10: Faculty Dashboard showing student reports and analytics

6. Conclusion and Future Scope

6.1 Conclusion

LabSense successfully demonstrates a unified, modular approach to coding lab examination manage-ment by integrating essential services — automated code evaluation, AI-powered semantic analysis, real-time anti-cheat monitoring, and comprehensive feedback generation — into a single, user-friendly platform. The use of Python, FastAPI, and React with TypeScript ensures a robust, scalable, and re-sponsive system. By leveraging Large Language Models (LLMs) and the Judge0 Cloud API for multi-language code execution, LabSense enhances evaluation accuracy and operational efficiency, while AI-generated feedback and detailed performance analytics improve the learning experience for students and faculty alike.

The modular architecture and cloud-compatible backend enable seamless evaluation pipeline execu-tion, centralized role-based control, and easy extensibility across multiple institutions. The multi-tenant design ensures college-level data isolation, supporting hierarchical management of departments, sections, and users without compromise to data integrity. Overall, LabSense achieves its goal of mov-ing beyond traditional output-only grading by assessing effort, logic similarity, and test case perfor-mance in combination, creating a more fair, educational, and student-centered examination environ-ment.

6.2 Future Scope

Future enhancements for LabSense include migration from the current JSON-based storage to a relational or document-based database such as PostgreSQL or MongoDB, which would significantly improve scalability, concurrency handling, and reporting flexibility for larger institutional deployments. Security can be strengthened through more advanced server-side audit logging, richer anomaly detection mechanisms, and stronger session integrity checks to make the examination environment more resilient against deliberate misuse.

The platform's utility can be extended through a formal automated test suite covering backend APIs, repository behavior, evaluation logic, and frontend interaction flows, which would improve system reliability and reduce regression risk during future development. Caching strategies for repeated LLM evaluations can be introduced to reduce response latency and lower dependency costs during high-concurrency exam sessions.

Communication and feedback capabilities can be enhanced with more advanced feedback visualization tools, question generation features, and difficulty calibration mechanisms that adapt to student performance trends over time. Additionally, a customizable modular structure can allow institutions to tailor LabSense to their specific requirements, enabling selective activation of components such as anti-cheat enforcement, LLM provider selection, and multi-language support based on organizational needs and available resources. With these enhancements, LabSense can evolve from a functional examination platform into a comprehensive academic assessment ecosystem that genuinely supports learning outcomes across institutions.

References

1. D. K. Lee and I. Joe, "A GPT-Based Code Review System with Accurate Feedback for Programming Education," in IEEE Access, vol. 13, pp. 105724-105737, 2025. DOI: https://doi.org/10.1109/ACCESS.2025.3581139
2. L. Huynh et al., "Detecting Code Vulnerabilities using LLMs," 2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Naples, Italy, 2025, pp. 401-414. DOI: https://doi.org/10.1109/DSN64029.2025.00047
3. P. Sukkasem, C. Soomlek and C. Dechsupa, "LLM-Based Code Comment Summarization: Efficacy Evaluation and Challenges," 2025 17th International Conference on Knowledge and Smart Technology (KST), Bangkok, Thailand, 2025, pp. 335-340. DOI: https://doi.org/10.1109/KST65016.2025.11003343
4. Z. Zhang and T. Saber, "Machine Learning Approaches to Code Similarity Measurement: A Systematic Review," in IEEE Access, vol. 13, pp. 51729-51764, 2025. DOI: https://doi.org/10.1109/ACCESS.2025.3553392
5. M. Bhavana et al., "Plagiarism Detection and Similarity Checking Program using Machine Learning and String Matching Algorithm," 2024 9th International Conference on Communication and Electronics Systems (ICCES), Coimbatore, India, 2024, pp. 2032-2036. DOI: https://doi.org/10.1109/ICCES63552.2024.10859471
6. R. Parvathy and M. Thushara, "AST-Based and Token-Based Neural Networks for Source Code Classification: A Comparative Performance Analysis," 2024 15th International Conference on

Computing Communication and Networking Technologies (ICCCNT), Kamand, India, 2024, pp. 1-7. DOI: https://doi.org/10.1109/ICCCNT61001.2024.10725521

7. W. Brach, K. Košťál and M. Ries, "Can Large Language Model Detect Plagiarism in Source Code?," 2024 2nd International Conference on Foundation and Large Language Models (FLLM), Dubai, United Arab Emirates, 2024, pp. 370-377. DOI: https://doi.org/10.1109/FLLM63129.2024.10852497
8. R. Tufano, O. Dabić, A. Mastropaolo, M. Ciniselli and G. Bavota, "Code Review Automation: Strengths and Weaknesses of the State of the Art," in IEEE Transactions on Software Engineering, vol. 50, no. 2, pp. 338-353, Feb. 2024. DOI: https://doi.org/10.1109/TSE.2023.3348172