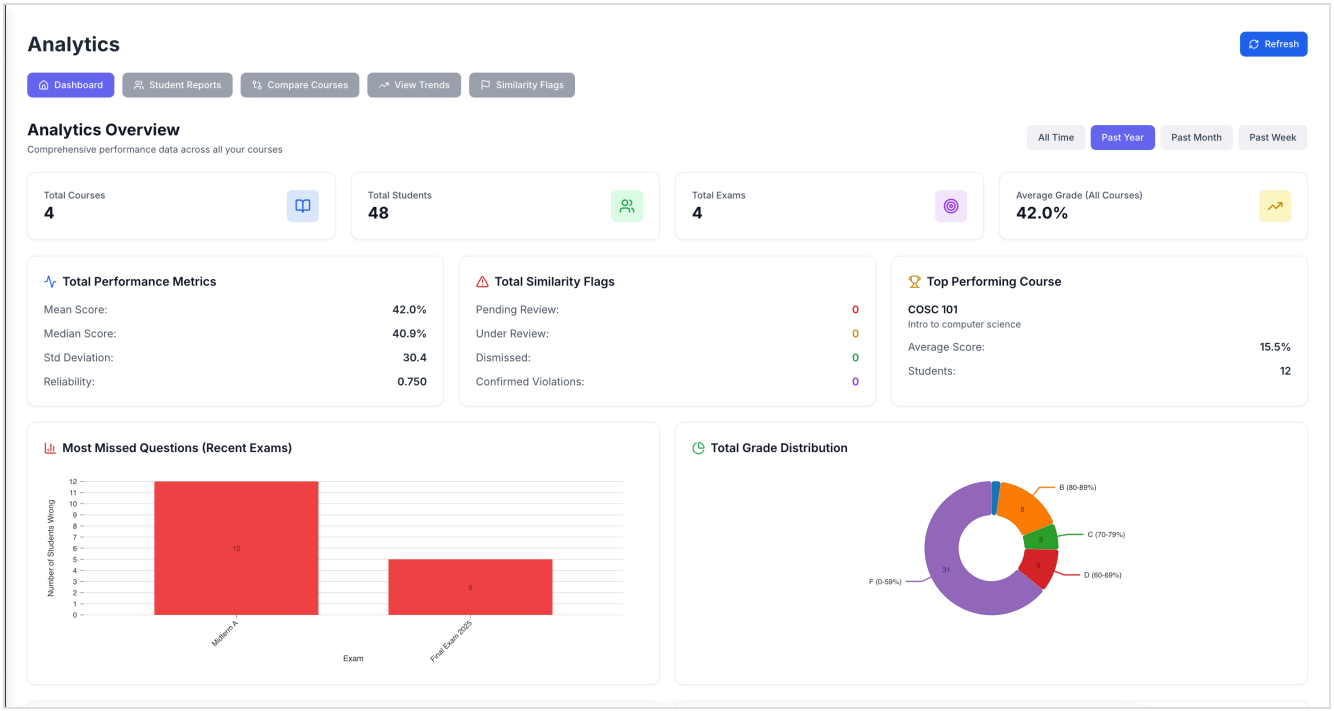
ExamVault: Full-Stack Exam Management System

This is a capstone project I built: a complete exam management system for educational institutions. It handles exam creation, variant generation, result analysis, and anti-cheating detection. Here's what it does and how it works.

Overview

ExamVault is a web application that lets instructors create exams, generate multiple variants to prevent cheating, upload results from OMR machines, and analyze student performance. The system supports courses, question banks, exam creation wizards, analytics dashboards, and similarity detection for potential collusion.

There were 14 teams in the capstone class total, and 7 teams worked on this specific project. Our implementation was selected as the client's first choice among those 7 teams.



The analytics dashboard shows performance metrics across all courses, similarity flags for potential cheating, and visualizations of student performance.

Technical Stack

Backend

- Django 4.2 with Django REST Framework
- PostgreSQL database
- JWT authentication with role-based permissions
- Python libraries: NumPy, SciPy for statistics, python-docx and reportlab for PDF/DOCX generation

Frontend

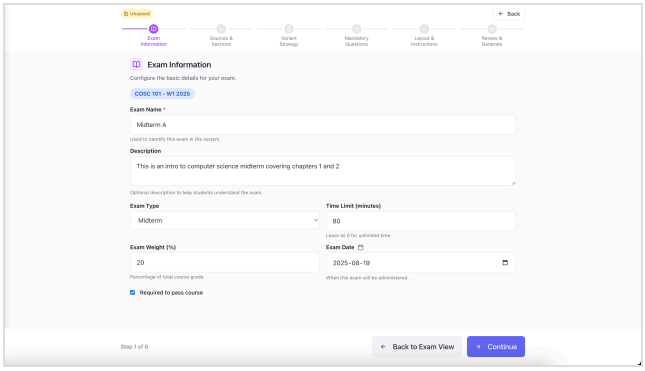
- React 18 with TypeScript
- Tailwind CSS for styling
- Nivo charts and D3.js for data visualization
- Vite for build tooling

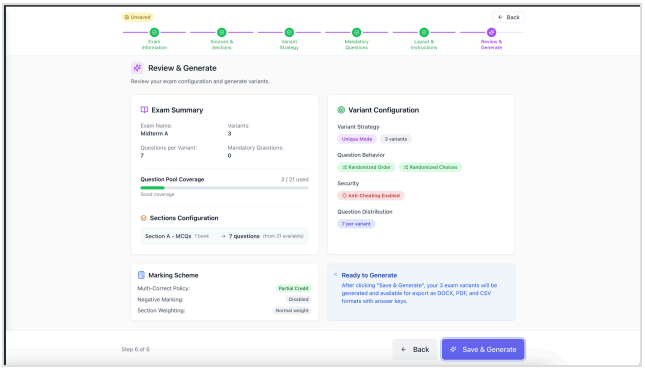
Infrastructure

- Docker and Docker Compose for deployment
- Nginx as reverse proxy
- Separated frontend and backend services

Key Features

Exam Creation Wizard





The exam creation process uses a step-by-step wizard. Instructors configure exam settings, select question banks, set difficulty distributions, and generate multiple variants. The system ensures all variants are fair and balanced in terms of difficulty while minimizing overlap.

Variant Generation Algorithm

The most complex part is generating multiple exam variants that are fair but different enough to prevent cheating. The algorithm:

- Maintains difficulty balance across variants using proportional allocation
- Supports two modes: reuse mode (same questions shuffled) and unique mode (different questions per variant)
- Shuffles answer choices in a round-robin pattern so the same correct answer isn't always in position A
- Validates that section-based question allocation is fair
- Calculates integrity scores using Hamming distances between variants

The algorithm handles edge cases like mandatory questions that must appear in all variants, maintaining section proportions, and ensuring enough unique questions are available in unique mode.

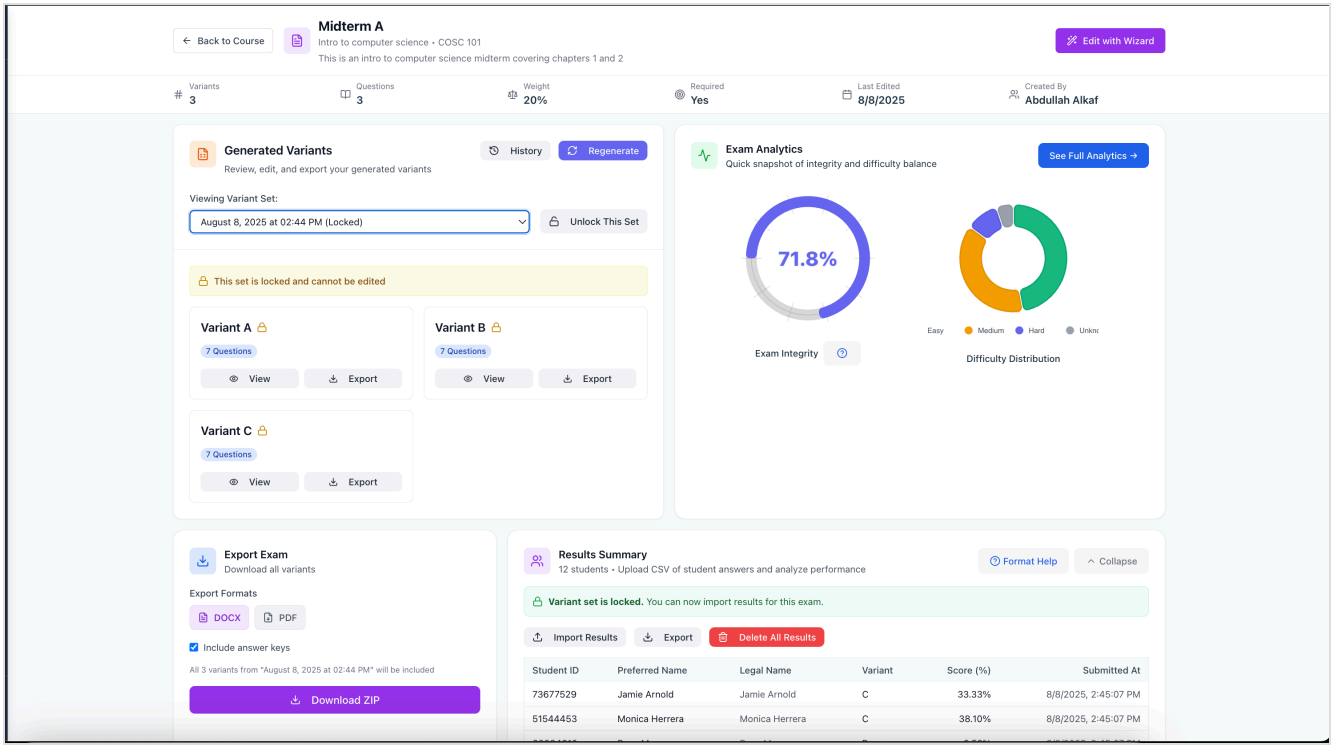
Similarity Detection

After results are uploaded from OMR machines, the system analyzes answer patterns to detect potential collusion. It:

- Compares student answer patterns pair by pair
- Flags suspicious matches where students have the same wrong answers

- Calculates similarity scores and determines severity levels
- Uses thresholds to reduce false positives (requires high overall similarity plus high ratio of matching wrong answers)

The similarity detection helps instructors identify potential cheating without overwhelming them with false alarms from students who just got questions right.

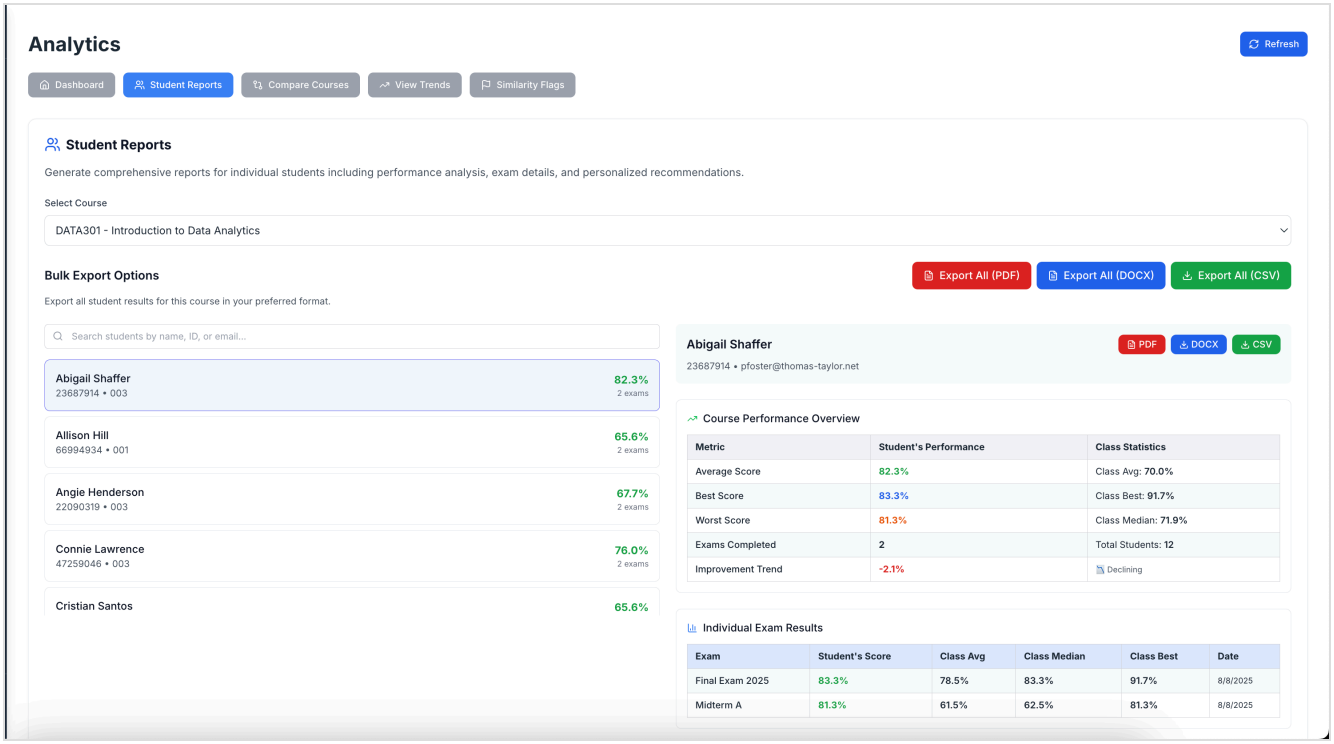


The exam dashboard shows created exams with export functionality. Instructors can export variants as PDF or DOCX files for printing.

Analytics and Reporting

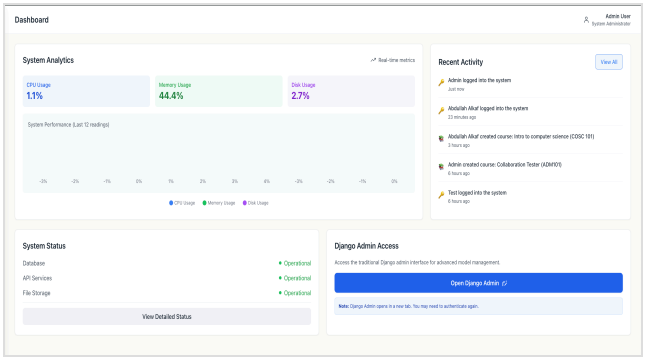
The system provides comprehensive analytics:

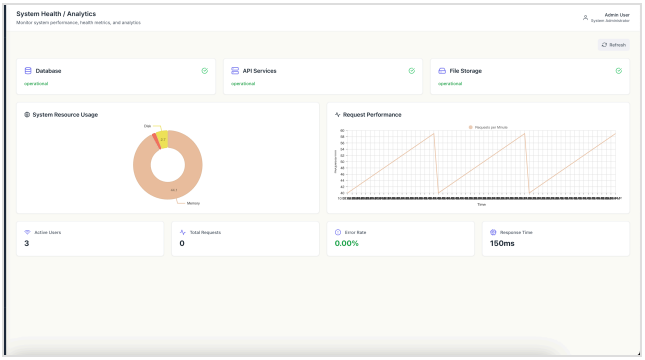
- Performance metrics: mean, median, standard deviation, skewness
- Grade distribution charts
- Question-level performance tracking
- Course comparison and trend analysis
- Export capabilities for CSV, PDF, and DOCX reports



Individual student reports show performance trends and detailed breakdowns.

Admin Interface





The admin interface handles user management, system health monitoring, course assignments, and overall system configuration. Role-based access control ensures instructors only see their courses and students.

Technical Challenges and Solutions

Variant Fairness

Ensuring all variants have equivalent difficulty was tricky. I implemented a difficulty distribution algorithm that groups questions by difficulty level and allocates them proportionally across variants. The system validates that each variant meets the target difficulty distribution before finalizing.

Performance at Scale

Generating variants for exams with 100+ questions could be slow. I optimized by using bulk database operations instead of individual saves, using sets for O(1) lookups when tracking used questions, and implementing efficient data structures for grouping questions by section and difficulty.

Similarity Detection Accuracy

Early versions flagged too many false positives when students simply got questions correct. I refined the algorithm to focus on suspicious patterns: high overall similarity combined with a high proportion of matching wrong answers. This significantly reduced false alarms while still catching actual collusion.

Data Integrity

Preventing duplicate questions and maintaining referential integrity required careful database design and application-level validation. I used Django's model constraints and custom validation logic to ensure data consistency.

Architecture

The system is split into React frontend, Django REST API backend, and PostgreSQL database. Frontend and backend communicate over REST APIs with JWT auth. I used TypeScript for the frontend which helped catch bugs early, and Django's ORM on the backend with query optimizations to avoid the N+1 problem.

Technical Implementation

For the variant generation, I used hash maps to group questions by difficulty and section for fast lookups. The similarity detection does pairwise comparisons between students which

gets expensive at scale, so I optimized it by early termination and efficient data structures. Statistical stuff uses NumPy which makes it fast even with thousands of results.

I wrote tests for the critical parts and got decent coverage on the backend. The code uses standard practices like type hints in Python and TypeScript, and the structure makes sense if you need to add features later.

What I liked about this project was solving the algorithmic challenges. Making sure variants are fair while still being different enough to prevent cheating required thinking through the problem carefully. The similarity detection also needed tuning to avoid false positives while catching actual collusion.

This was a full-stack project using React, TypeScript, Django, and PostgreSQL, with Docker for deployment. I learned a lot about building systems that need to handle real data and scale. The system actually works and could be deployed, which I'm proud of.