Semester Long Internship Report

Autumn, 2025

On

UI/UX Design and Implementation of Dynamic Frontend Interface for Yaksh Online Assessment Platform

Submitted by

Sourasish Bhattacharjee
B.Tech,
National Institute of Technology, Silchar

Under the guidance of

Dr.Kushal Shah, Mr.Prathamesh Salunke
and
Prof. Prabhu Ramachandran

FOSSEE, Indian Institute of Technology Bombay

April 04, 2026

Declaration

I hereby declare that this internship report, entitled " **UI/UX Design and Implementation of Dynamic Frontend Interface for Yaksh Online Assessment Platform**", is an original and authentic work submitted in partial fulfilment of the requirements for **the FOSSEE Semester-Long Internship (Part-Time**), conducted by the **Free/Libre and Open Source Software for Education (FOSSEE) team at the Indian Institute of Technology Bombay**.

All content herein is expressed in my own words, except where explicit references have been made to the ideas or works of others, which have been duly acknowledged and cited. No part of this submission has been plagiarised, misrepresented, fabricated, or falsified, nor has it been submitted to any other institution for the award of any degree or academic credit.

Name	Sourasish Bhattacharjee
Internship	FOSSEE Semester-Long Internship (Part-Time)
Guide	Dr. Kushal Shah & Mr. Prathamesh Salunke
Time period	October 15, 2025 – April 04, 2026

Abstract

This report documents the work carried out during a part-time semester-long internship with the **FOSSEE team at IIT Bombay**, focused on the modernization of the **Yaksh online assessment platform**. The Yaksh platform, built on the Django framework, previously relied on server-side rendering and Django's templating engine, which imposed significant limitations on user experience, interactivity, and scalability.

The internship encompassed two major phases. In the first phase, initial wireframes for key screens were sketched using Figma, covering all primary user journeys for both students and instructors. This involved low-fidelity wireframing, high-fidelity prototyping, and the creation of a comprehensive design system with standardized typography, colour tokens, and reusable component libraries. The designs were iteratively reviewed and approved by project supervisors before any code was written.

In the second phase, a production-grade **React.js** application using a modern technology stack comprising **React 19**, **Vite**, **Zustand**, **Tailwind CSS**, and **React Router DOM v7**. Concurrently, a **RESTful API** layer was architected within the existing **Django/Yaksh backend** using **Django REST Framework (DRF)** to serve as the communication bridge between the new frontend and the core evaluation engine.

Key technical contributions include: a decoupled **Zustand** based state management architecture, a data adapter pattern resolving legacy **MCQ schema** mismatches, **JWT**-secured **Axios** interceptors, private route wrappers for role-based access control, a live Quiz Monitor Panel for instructors, and a student portal with timer-synchronized test-taking and post-exam code diff visualization.

The internship ran from October 2025 to April 2026.

Acknowledgements

I would like to express my sincere gratitude to the **FOSSEE team** at **IIT Bombay** for granting me the opportunity to participate in the **Semester-Long Internship Programme**. This internship has been a transformative experience that broadened my technical knowledge and gave me meaningful exposure to the intersection of open-source software development, educational technology, and modern full-stack engineering.

I am deeply thankful to my mentors, **Mr. Prathamesh Salunke, Dr. Kushal Shah and Prof. Prabhu Ramachandran**, for their invaluable guidance, thoughtful feedback, and continuous encouragement throughout the internship. Their direction was instrumental in shaping the architecture and design philosophy of this project.

I would also like to acknowledge **Ms. Usha Viswanathan, and Ms. Vineeta Ghavri**, and other team members at **FOSSEE**, for their support during the programme. They were always approachable and helpful whenever questions arose.

I am grateful for access to the Yaksh online assessment platform maintained by the **FOSSEE team**, which served as the foundation for all design and development work carried out during this internship.

Finally, I would like to thank my home institution, the **National Institute of Technology Silchar**, for their support throughout this period.

Table of Contents

Declaration 2
Abstract 3
Acknowledgements 4
Table of Contents 5
1. Introduction 7
  1.1 National Mission on Education through ICT (NMEICT) 7
  1.2 The FOSSEE Project 7
  1.3 The Yaksh Platform 7
  1.4 Motivation for the Internship 7
  1.5 Internship Overview and Objectives 7
  1.6 Report Organization 8
2. Screening Task and Project Onboarding 9
  2.1 Screening Task 9
  2.2 Initial Repository Setup and Environment Configuration 9
3: UI/UX Design and Prototyping Phase 11
  3.1 User Research and Journey Mapping 11
    3.1.1 The Instructor Journey 11
    3.1.2 The Student Journey 11
  3.2 Initilal Wireframing 11
  3.3 Prototyping and Supervisor Approval 12
  3.4 Basic Design Tokens and Component Reference 12
    3.4.1 Typography and Color Palette 12
    3.4.2 Reusable Component Library 12
  3.5 Stakeholder Review and Design Approval 13
4: System Architecture and Technology Stack 14
  4.1 Limitations of the Existing System 14
  4.2 Proposed Headless Architecture 14
  4.3 Technology Stack 15
5: Backend API Development (api module) 16
  5.1 Serializer Design (serializers.py) 16
  5.2 ViewSets and API Endpoints (views.py & urls.py) 17
  5.3 Sample API Interaction 17
6: Frontend SPA Development (frontend) 20
  6.1 State Management Architecture (src/store/) 20
  6.2 The Instructor Portal 20
    6.2.1 Course and Media Management 21
    6.2.2 Dynamic Quiz Authoring 24
    6.2.3 Analytics, Grading, and Live Monitoring 26

6.3 The Student Portal 30
6.3.1 Active Assessment Interface (Quiz.jsx & Submission.jsx) 33
6.3.2 Post-Examination Insights 33
6.3.3 Course Discussion Forum 34
6.4 Theming, Layout, and UI/UX Pipeline 35
7. Technical Challenges and Solutions 36
7.1 Complex Test Case Option Bundling — Data Adapter Pattern 36
7.1.1 The Problem 36
7.1.2 The Solution — Read / Write / Evaluate Adapter 36
7.2 User Mode / God Mode for Instructor Quiz Testing 40
7.2.1 The Problem 40
7.2.2 The Solution — Isolated Sandbox Architecture 40
8. Development Workflow and Timeline 42
8.1 Phase-wise Development Summary 42
9. Conclusions and Future Scope 44
9.1 Summary of Contributions and Impact 44
9.2 Skills Developed 44
9.2.1 Technical Skills 44
9.2.2 Professional Skills 44
9.3 Future Scope 44
References 46

1. Introduction

1.1 National Mission on Education through ICT (NMEICT)

The **National Mission on Education through ICT (NMEICT)** is a flagship scheme under the **Department of Higher Education, Ministry of Education, Government of India**. Its primary objective is to leverage the transformative potential of Information and Communication Technologies to enhance the quality, reach, and equity of higher education across India. The mission operates on the three cardinal principles of access, equity, and quality, seeking to empower learners and educators in both urban and rural contexts.

1.2 The FOSSEE Project

The **FOSSEE (Free/Libre and Open Source Software for Education)** project is a key initiative under **NMEICT**, headquartered at **IIT Bombay**. Its mission is to promote the adoption of **Free/Libre and Open Source Software** tools across academic and research institutions in India, reducing dependence on expensive proprietary software. **FOSSEE** supports a wide range of activities including **Textbook Companions, Lab Migrations, and domain-specific software development**. It offers semester-long, summer, and winter internship programmes to students across disciplines.

1.3 The Yaksh Platform

Yaksh is an **open-source online test and grading platform** developed and maintained by the **FOSSEE team** at **IIT Bombay**. It is designed to conduct online assessments, evaluate programming assignments across multiple languages including **Python, Java, C++, and Bash**, and manage course grading efficiently. **Yaksh** is widely deployed in Indian educational institutions for conducting programming courses and evaluations at scale.

Historically**, Yaksh** has been built as a **monolithic, server-side rendered application** using the **Django web framework**. While robust and functionally rich, the reliance on Django's templating engine presented growing limitations: every user interaction required a full page reload, complex UI states were difficult to manage with vanilla JavaScript, and the overall experience felt dated compared to modern web applications.

1.4 Motivation for the Internship

As the web development landscape has increasingly shifted towards highly interactive client-side applications, the need arose to modernize the **Yaksh** platform's **user interface**. The primary motivation was to **decouple the monolithic architecture** by separating the user interface from the core evaluation engine. This separation would allow **Yaksh** to achieve **faster load times, fluid transitions, and a more responsive, application-like experience** for both students and instructors, without disrupting the well-tested backend evaluation logic.

1.5 Internship Overview and Objectives

This internship was **a part-time, semester-long engagement under the Python (UI/UX) team at FOSSEE, IIT Bombay, running from October 13, 2025 to April 4, 2026**. The internship was structured in two sequential phases, each with distinct deliverables:

- **Phase 1 — UI/UX Design**: **Conceptualize, wireframe, and prototype** initial wireframes for key screens of the **Yaksh** platform using **Figma**, covering all student and instructor workflows. Obtain stakeholder approval before proceeding to development.
- **Phase 2 — Frontend Development**: Architect and implement a production-grade **React.js Single Page Application (SPA)** built iteratively during the development phase, using the initial **Figma** designs into a functional, performant web application.
- **API Development**: **Architect and implement a RESTful API layer** within the existing **Django** backend to expose **Yaksh's core data models** to the **new React frontend.**

1.6 Report Organization

This report is organized as follows. Chapter 2 describes the screening task and project onboarding. Chapter 3 covers the initial wireframing and prototyping phase. Chapter 4 details the system architecture and technology stack. Chapter 5 describes the Backend API development. Chapter 6 covers the Frontend SPA development in detail. Chapter 7 discusses key technical challenges and solutions. Chapter 8 describes the development workflow. Chapter 9 concludes with a summary of impact and future scope.

2. Screening Task and Project Onboarding

2.1 Screening Task

Prior to being accepted for the **semester-long internship**, I was selected based on my demonstrated performance on a screening task for a separate **FOSSEE** project involving frontend UI improvements (https://github.com/Sourasish01/workshop_booking_clone).

The **FOSSEE Python (UI/UX) team** reviewed the quality of the UI work submitted for that prior project repository and, on the strength of that evaluation, extended an offer for the **Yaksh semester-long internship**. The selection process assessed my proficiency with **design tools, frontend web technologies, and my ability to understand and improve existing software user interfaces**.

The screening task itself involved studying an **existing FOSSEE project frontend, identifying usability shortcomings, and submitting a redesigned interface concept demonstrating frontend design thinking.** Upon successful evaluation of this prior work, I was offered the semester-long internship on the Yaksh platform.

2.2 Initial Repository Setup and Environment Configuration

Following acceptance, the initial weeks of the internship were spent on onboarding and environment setup. Since the **Yaksh platform** officially supports only **Linux** and **macOS**, a **Windows Subsystem for Linux (WSL)** environment was configured on the development machine running **Windows**. **Ubuntu** was installed via **WSL** to provide a **compatible Linux environment** for running all project dependencies natively.

2.2.1 System Requirements

The project has the following core dependencies:

- **Python 3.6, 3.7, or 3.8**
- **Django 3.0.3**
- **Celery 4.4.2**
- **Redis Server** (required as the message broker for Celery background tasks)

2.2.2 WSL and Miniconda Setup

Since **Yaksh** officially supports only **Linux** and **macOS**, **Windows Subsystem for Linux (WSL)** was set up with **Ubuntu** to provide a **native Linux environment** on the Windows development machine. Where Python 3.6 or above was not available in the system environment, Miniconda was used to manage Python versions and virtual environments cleanly. Miniconda was downloaded from the official Conda documentation and installed following the regular installation instructions, after which the terminal was restarted to apply the changes.

2.2.3 Pre-requisites: Redis and Celery

Redis is required as the **message broker** for **Celery**, which **Yaksh** uses to **run background tasks** for **re-evaluating student code submissions asynchronously.** The following steps were followed within the WSL Ubuntu environment to configure these services:

- Install Redis server on Debian/Ubuntu: **sudo apt install redis-server**
- Start the Redis server: **systemctl start redis**
- Verify Redis is running: **systemctl status redis**
- Start the Celery worker**: celery -A online_test worker -B**

2.2.4 Installing the Yaksh Project

The project repository was cloned and dependencies were installed as follows. The repository (https://github.com/FOSSEE/online_test) was cloned locally, the directory was entered, and then **two sets of dependencies were installed**: the core Django and application dependencies via **pip3 install -r requirements/requirements-common.txt**, and the code server dependencies via **sudo pip3 install -r requirements/requirements-codeserver.txt**.

2.2.5 Running the Application (Quick Start)

Yaksh uses an **invoke-based task runner** to start its services. The code execution server, which evaluates user-submitted code in a sandboxed environment, is started first using **invoke start (with Docker)** or **invoke start --unsafe (locally, without Docker**). In a separate terminal, the Django application server is started on port 8000 using invoke serve. The application is then accessible at http://localhost:8000/exam. Default credentials provided for local development are as follows:

- Student — Username: student | Password: student
- Teacher — Username: teacher | Password: teacher
- Admin — Username: admin | Password: admin

With the local environment fully configured and the existing Yaksh codebase studied, including the Django project structure, database models, and code evaluator modules such as bash_code_evaluator.py and cpp_code_evaluator.py, the project proceeded into the UI/UX design phase.

The key starting commit that marks the beginning of the development-phase contributions to the repository by me is **b87cbc620a2fcee1e0307bd7653da6ac5e55ef19** (https://github.com/FOSSEE/online_test/pull/870/changes/b87cbc620a2fcee1e0307bd7653da6ac5e55ef19). (All development work through to the final build in April 2026 is traceable through the project's pull Git history.

3. UI/UX Design and Prototyping Phase

Before writing a single line of application code, the initial phase of the internship was **strictly dedicated to Research, Design, and Stakeholder Approval**. Because the project involved moving from a legacy server-rendered template system to a modern **SPA**, the entire user journey had to be **reimagined from first principles**.

3.1 User Research and Journey Mapping

The first step was mapping out the distinct user journeys for the **two primary roles on the Yaksh platform: Students and Instructors (Teachers**). A thorough review of the existing Django template-based interface was conducted to identify all existing workflows, pain points, and missing features.

3.1.1 The Instructor Journey

The **instructor workflow** was found to be the most **data-intensive**. Instructors require **dashboards to manage courses, build quizzes, configure grading, and monitor student activity in real or near-real time**. The key pain points identified in the legacy interface included the absence of drag-and-drop question arrangement, cumbersome multi-step workflows for creating programming questions, and no live view of students currently attempting a quiz.

3.1.2 The Student Journey

The redesign of the student **workflow prioritized a distraction-free, highly focused assessment environment**. In the legacy monolithic application, the UI suffered from several UX pain points that degraded the learning experience. This included **the lack of a persistent quiz navigation sidebar, forcing linear progression, and a deficient post-examination feedback mechanism that failed to clearly contrast a student's submitted code against the expected reference solution.** Furthermore, the **legacy system relied on frequent full-page browser reloads during quiz navigation; this not only disrupted student focus but frequently compromised the synchronization of the examination timer**.

Proceeding to a React Single Page Application (SPA) natively resolved these limitations. We integrated a persistent **QuizSidebar** component for non-linear navigation, utilized React Router for seamless UI updates without page reloads, and executed answer validations asynchronously. Finally, the introduction of the **ViewAnswerPaper** page allowed students to view a side-by-side diff of their executed code logic versus the expected test-case assertions, drastically improving post-exam educational value.

3.2 Initial Wireframing

With the user journeys mapped, rough wireframes were sketched in Figma to work out the basic layout direction — where sidebars and navigation should sit, and how the main screens should be structured. These were treated as a starting guide, not a finished spec, since layout decisions continued to evolve throughout the actual frontend development.

Figma Links :

1. https://www.figma.com/design/RIlkLTRdwAZu06enr2rOq2/YAKSH-WIREFRAMES-TEACHER?node-id=0-1&t=9Gp5noxGHkGfMxNR-1

3.3 Prototyping and Supervisor Approval

After iterating on the wireframes, a more refined prototype covering the key screens was put together and presented to project supervisors for approval. The prototype served as a conversation tool to confirm the general direction before coding began — not as a pixel-perfect specification. The actual visual design was refined iteratively during implementation. Key screens covered included:

- **Authentication Flow**: Login, Signup, and Forgot Password screens with input validation state indicators.
- **Teacher Side Interface**: Course management overview, course design canvas, question bank browser, grading system configuration panel, and the Live Quiz Monitor dashboard.
- **Student Side Interface**: Course module listing, active lesson/quiz/exercise viewer, and the quiz enrollment page.
- **Active Quiz Interface**: A distraction-minimized layout featuring a code editor panel, question navigation sidebar, countdown timer, and submission confirmation modal.
- **Post-Exam Views**: The View Answer Paper screen, showing a side-by-side diff of submitted student code versus the correct solution with assertion results.

3.4 Basic Design Tokens and Component Reference

During the prototyping phase, a basic set of design tokens — colour palette and typography — was documented to maintain consistency across screens. This served as an initial reference; **actual styling decisions were refined progressively during frontend development**.

3.4.1 Typography and Color Palette

A standardized typographic scale was defined, covering heading levels H1 through H4 and body copy sizes, all mapped to a consistent type ramp. A **primary and secondary color palette was established, with each color token defined with both light and dark mode variants to support the ThemeController feature planned for the application**. Contrast ratios were verified against WCAG AA accessibility standards.

3.4.2 Reusable Component Library

A comprehensive library of reusable UI components was designed, covering all interactive elements that would be needed in the application. Each component was designed with all required states (default, hover, active, disabled, and error). Key components in the library

included primary and secondary buttons, form input fields, dropdown selectors, modal overlays, toast notification variants, data table rows, and the course module card.

A basic colour and typography reference was established during the prototyping phase. In practice, the Tailwind CSS configuration was set up independently during development, with styling decisions made progressively as components were built.

3.5 Stakeholder Review and Design Approval

Upon completing the prototype, the key user flows were walked through with project supervisors and FOSSEE mentors — a teacher creating a course and building a quiz, and a student enrolling and attempting an assessment. Feedback was incorporated into the layout direction before development began.

Based on feedback, a few layout adjustments were made before receiving approval to proceed into development. **As implementation progressed, the UI continued to evolve based on practical constraints and ongoing mentor feedback — the final product differed considerably from the initial prototype.**

4. System Architecture and Technology Stack

4.1 Limitations of the Existing System

The legacy **Django template system** required a full **HTTP round-trip** and **server-side HTML re-render** for **every button click**, form submission, or navigation event. This created **substantial payload overhead and introduced perceptible latency for every user interaction**. Furthermore, **managing complex client-side UI states** such as live quiz monitoring, dynamic drag-and-drop question ordering, and real-time autosaving **using vanilla JavaScript within Django templates was inherently difficult to maintain and scale**. There was no clear separation between the UI presentation layer and the business logic, making targeted **improvements costly and risky**.

4.2 Proposed Headless Architecture

The **proposed solution** was a **Headless (Decoupled) Architecture**. Under this model, the **Django backend** is responsible purely for **business logic**: compiling and evaluating submitted code through its existing evaluator modules, managing database state via SQLite, and running asynchronous code evaluation tasks through Celery task queues. A **new Django app, api, was introduced to expose this logic as a clean RESTful API**. **The React frontend**, built and served by Vite, handles all UI rendering, client-side routing, and user state management entirely **in the browser**, **communicating** with the **backend** exclusively through **JSON API requests via Axios**.

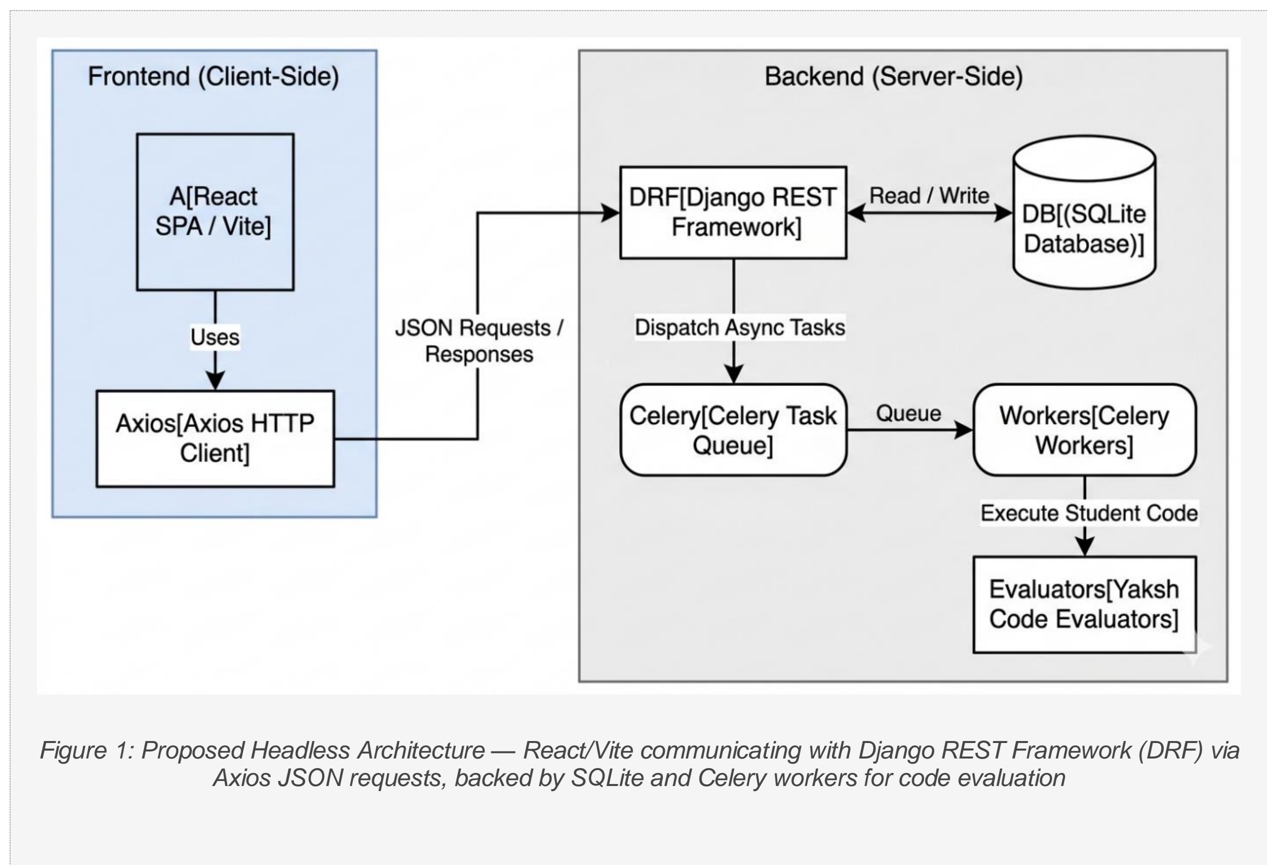

Figure 1: Proposed Headless Architecture — React/Vite communicating with Django REST Framework (DRF) via Axios JSON requests, backed by SQLite and Celery workers for code evaluation

4.3 Technology Stack

Category	Technology	Rationale
Frontend Core	React 19 + Vite	Ultra-fast HMR, optimized production builds
State Management	Zustand	Lightweight, boilerplate-free global state; chosen over Redux for modular store design
Styling	Tailwind CSS + PostCSS	Utility-first, responsive styling with a clean configuration-driven approach
Routing	React Router DOM v7	Client-side routing with protected route HOC wrappers
HTTP Client	Axios	Request/response interceptors for JWT token attachment and error handling
Code Quality	ESLint + Prettier	Enforced consistent code style and static analysis across the codebase
Backend API	Django REST Framework	Serializers and ViewSets exposing existing Yaksh models as RESTful JSON endpoints
Dev Environment	WSL 2 + Ubuntu (Windows)	Yaksh only supports Linux/macOS; WSL with Ubuntu was used on the Windows development machine to provide a compatible Linux environment
Task Queue	Celery 4.4.2 + Redis	Asynchronous code re-evaluation tasks; Redis serves as the Celery message broker
Python Runtime	Python 3.6–3.8 + Miniconda	Required by Django 3.0.3 and Yaksh's evaluation engine; Miniconda used for version management

5. Backend API Development (api module)

To make the **React SPA** functional, the existing **Django database models** needed to be **securely exposed as a RESTful API**. A dedicated Django application, named **api**, was used to house all **API-related logic**, ensuring a **clean separation** from the core **Yaksh evaluation engine.**

5.1 Serializer Design (serializers.py)

Complex serializers were developed to **translate Django ORM models** — including **Courses, Quizzes, Questions, and Grades** — into **JSON representations consumable by the React frontend**. A significant responsibility was ensuring that deeply nested object relationships were serialized efficiently. For example, a Quiz object containing multiple Questions, which in turn each contain multiple Test Cases, needed to be serialized in a single, coherent API response.

To mitigate the **N+1** query problem that can arise with deeply **nested serialization**, **`select_related`** and **`prefetch_related`** were applied strategically to the underlying QuerySets. This ensured that fetching a quiz with all its questions and test cases required a bounded number of database queries regardless of the number of nested objects, keeping API response times consistently low.

The following code demonstrates this concretely for the grading interface, where a single **`AnswerPaper`** response must contain the submitting **`User`**, the **`QuestionPaper`**, and all individual **`Answer`** objects nested within it.

Step 1 — Defining the nested serializer structure (api/serializers.py)

```
# Location: api/serializers.py
class AnswerPaperGradingSerializer(serializers.ModelSerializer):
    """AnswerPaper serializer for grading interface"""
    # Fetching this naively would cause hundreds of DB queries (N+1 
problem)
    user = SimpleUserSerializer(read_only=True)
    answers = AnswerDetailSerializer(many=True, read_only=True)
    question_paper = QuestionPaperSerializer(read_only=True)

    class Meta:
        model = AnswerPaper
        fields = ['id', 'user', 'question_paper', 'answers', 
'marks_obtained',
                 'percent', 'status', 'attempt_number', 'comments',
                 'start_time', 'end_time']
```

Step 2 — Optimizing the QuerySet to eliminate the N+1 problem (views.py)

Declaring nested serializer fields alone does not prevent excess queries. Without explicitly telling **Django's ORM** how to batch-load related objects, it **executes a separate SQL query** for every single nested record — leading to hundreds of database round-trips for a class of 50

students. The fix was applied in views.py inside the **download_quiz_csv** view (and similarly across grading views): select_related collapses **ForeignKey** lookups (**`User`**, **`QuestionPaper`**) into a single **`SQL JOIN`**, while prefetch_related handles the reverse one-to-many Answers relationship in a **single batched query**. Together they reduce what would be hundreds of queries to just two SQL operations, regardless of the number of students or answers..

```
# Location: api/views.py  (inside upload_marks view)
@api_view(['POST'])
@permission_classes([IsAuthenticated])
def download_quiz_csv(request, course_id, quiz_id):
    # ... previous logic omitted ...

    # Collapses FK lookups + batches reverse relation in 2 SQL ops
total
    answerpapers = AnswerPaper.objects.select_related(
        "user", "question_paper"
    ).prefetch_related('answers').filter(
        course_id=course_id,
        question_paper_id=question_paper.id,
        attempt_number=attempt_number
    ).order_by("user__first_name")
```

5.2 ViewSets and API Endpoints (views.py & urls.py)

A comprehensive suite of **RESTful API** endpoints was established across the following functional modules:

- **Authentication Endpoints**: Handling secure login, **token** provisioning (**JWT**), and **session** management.
- **Resource Management (CRUD)**: Endpoints allowing instructors to Create, Read, Update, and Delete courses, course modules, quiz configurations, and grading systems.
- **Question Bank**: Endpoints for creating and retrieving **MCQ, MCC (Multiple Correct Choice)**, and **programming question** types with their **associated test cases**.
- **Code Execution and Evaluation**: Endpoints that bridge the API layer to the existing **Yaksh code evaluators (e.g., bash_code_evaluator.py, cpp_code_evaluator.py),** so that a student clicking 'Run Code' in the React interface triggers secure code execution on the backend.
- **Analytics and Grading**: Endpoints **exposing aggregated student performance data**, **submission histories**, and **grade override capabilities**.

5.3 Sample API Interaction

The following illustrates a sample response for the quiz question retrieval endpoint from teacher side:

```
# GET /api/teacher/questions/4/
HTTP 200 OK
Allow: GET, PUT, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 4,
    "test_cases": [
        {
            "id": 5,
            "question": 4,
            "type": "standardtestcase",
            "testcase_ptr": 5,
            "test_case": "assert is_palindrome(\"hello\") == False",
            "weight": 1.0,
            "test_case_args": "",
            "hidden": false
        },
        {
            "id": 6,
            "question": 4,
            "type": "standardtestcase",
            "testcase_ptr": 6,
            "test_case": "assert is_palindrome(\"nitin\") == True",
            "weight": 1.0,
            "test_case_args": "",
            "hidden": false
        },
        {
            "id": 7,
            "question": 4,
            "type": "standardtestcase",
            "testcase_ptr": 7,
            "test_case": "assert is_palindrome(\"madaM\") == False",
            "weight": 1.0,
            "test_case_args": "",
            "hidden": false
        }
    ],
    "files": [],
    "summary": "Check Palindrome",
    "description": "Write a function <code>is_palindrome(arg)</code>
which will take one string argument.\n<br>\nReturn True if the
argument is palindrome & False otherwise.\n <br>\nThe function should
be case sensitive.\n<br><br>\nFor
Example:<br><code>is_palindrome(\"Hello\")</code> should return
<code>False</code><br>",
    "points": 2.0,
```

```
    "language": "python",
    "topic": null,
    "type": "code",
    "active": true,
    "snippet": "def is_palindrome(s):",
    "partial_grading": false,
    "grade_assignment_upload": false,
    "min_time": 0,
    "solution": "",
    "user": 2
}
```

6. Frontend SPA Development (frontend)

This chapter covers the implementation work carried out over the four-month coding period (December 2025 to April 2026), detailing the development of the React Single Page Application.

6.1 State Management Architecture (src/store/)

Rather than using a monolithic global state store — which tends to create performance bottlenecks and tightly coupled components — a highly **decoupled**, **modular state management architecture** was engineered using **Zustand**. Each distinct domain of the application was given its own independent store, ensuring that UI re-renders were scoped precisely to the components that actually consumed the changed state.

- **authStore.js** & **userStore.js**: Managed persistent **user sessions** using **localStorage**, **role identification** (student vs. teacher), and the **state required for UI access control** throughout the application.
- **questionsStore.js & quiz_QuestionStore.js**: Handled the complex lifecycle of **caching the question bank**, **reducing redundant API calls** during dynamic quiz assembly by the instructor.
- **quizMonitorStore.js**: Managed the **real-time or near-real-time state** of students actively attempting a quiz, feeding live metrics to the **teacher's Quiz Monitor Panel**.
- **quizRegradeStore.js**: Managed the state for **instructor-initiated grade overrides**, decoupling this administrative action from the primary quiz flow stores.
- **forumStore.js**: Managed **discussion thread state for the course discussion forum** feature.

6.2 The Instructor Portal

The instructor portal represented the most complex set of UI requirements, demanding the translation of data-dense into robust, interactive React components.

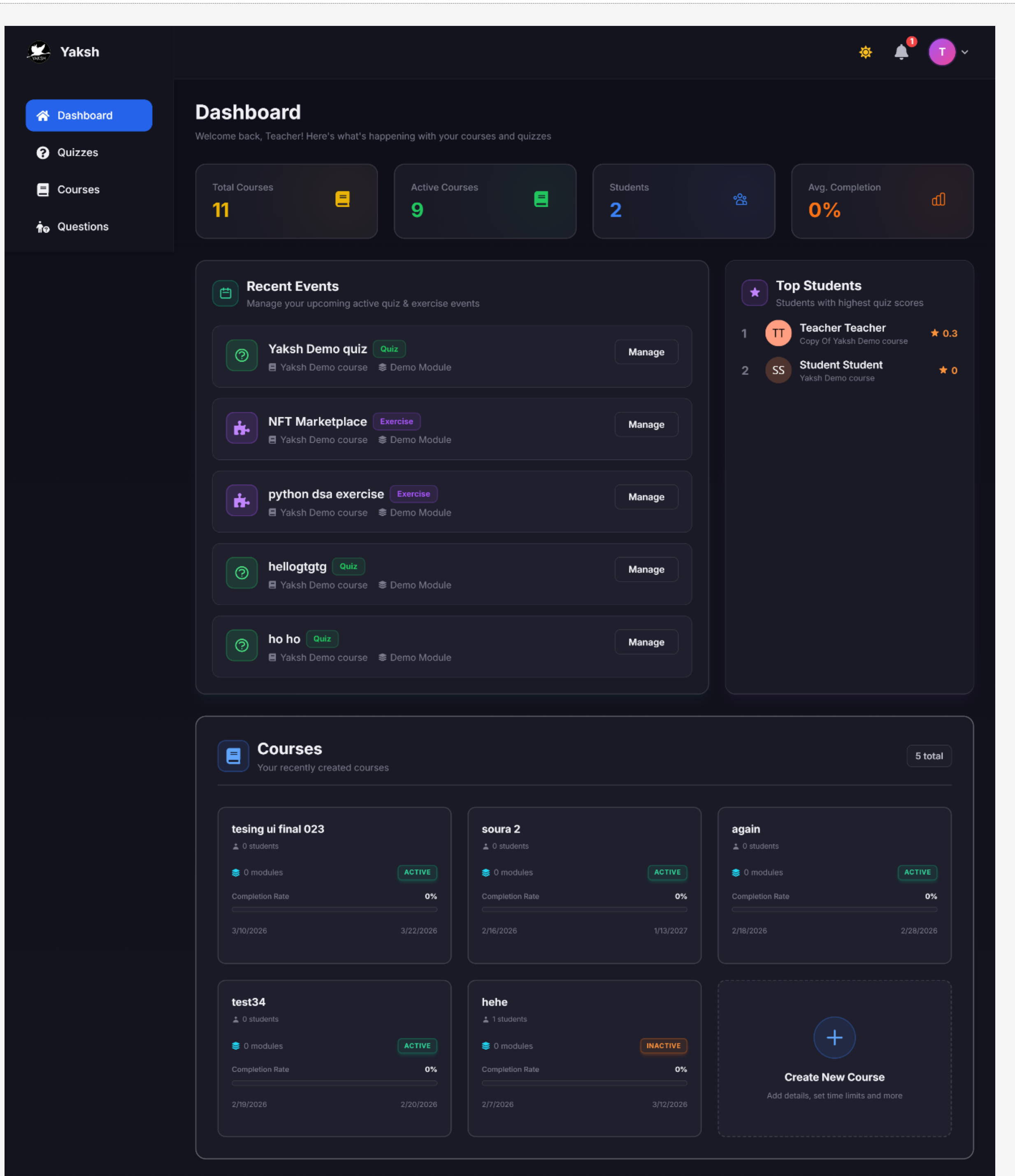

Figure 2: Instructor Portal — Teacher Dashboard overview

6.2.1 Course and Media Management

The **AddCourseModal**, **CourseDesign**, and **CourseMDManager** components were developed to allow instructors **to build and manage course curriculums**. These components supported structured module organization and media attachment. The

CourseDesign component in particular implements a **canvas-like interface for arranging course modules** in a logical sequence.

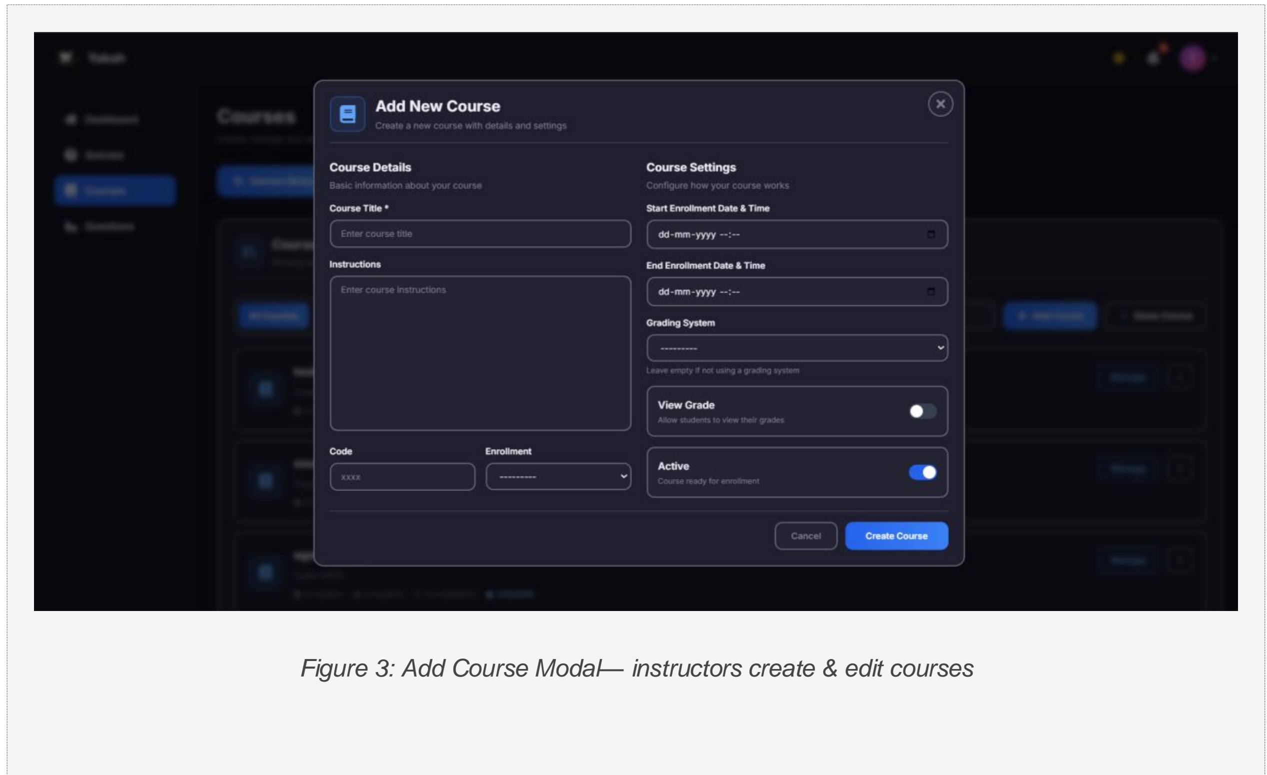

Figure 3: Add Course Modal— instructors create & edit courses

Figure 4: Course Modules— instructors manage courses

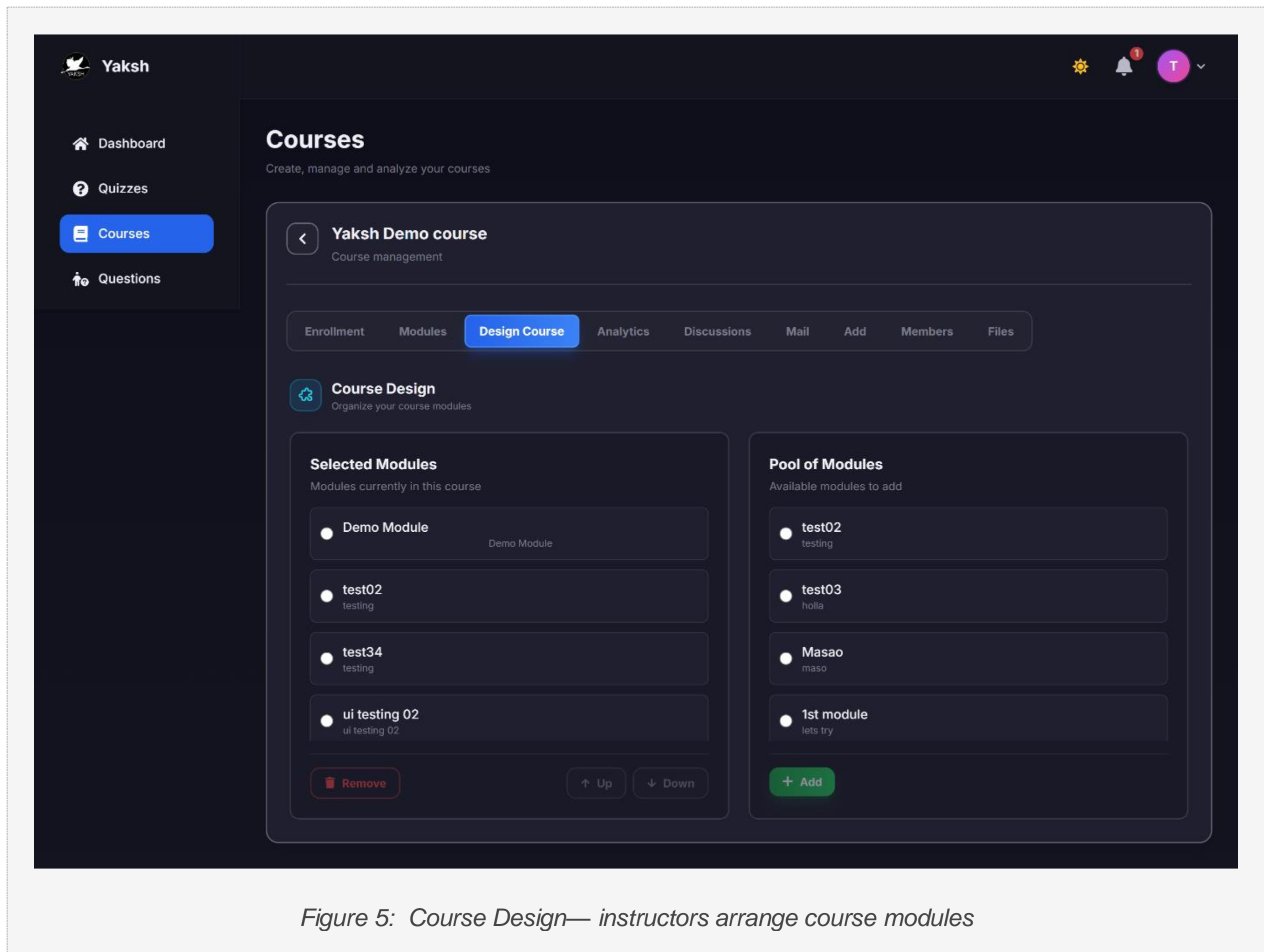

Figure 5: Course Design— instructors arrange course modules

6.2.2 Dynamic Quiz Authoring

The **QuizQuestionManager** and **AddQuestionModal** components were engineered to support the creation of different question types available on **Yaksh: Multiple Choice Questions (MCQs), Multiple Correct Choice questions (MCCs), Programming questions with associated test cases, Arrange in order questions, etc.** A significant UX enhancement was the implementation of **drag-and-drop question reordering within the quiz builder (committed January 14)**, allowing instructors to **arrange questions intuitively** without relying on manual index fields.

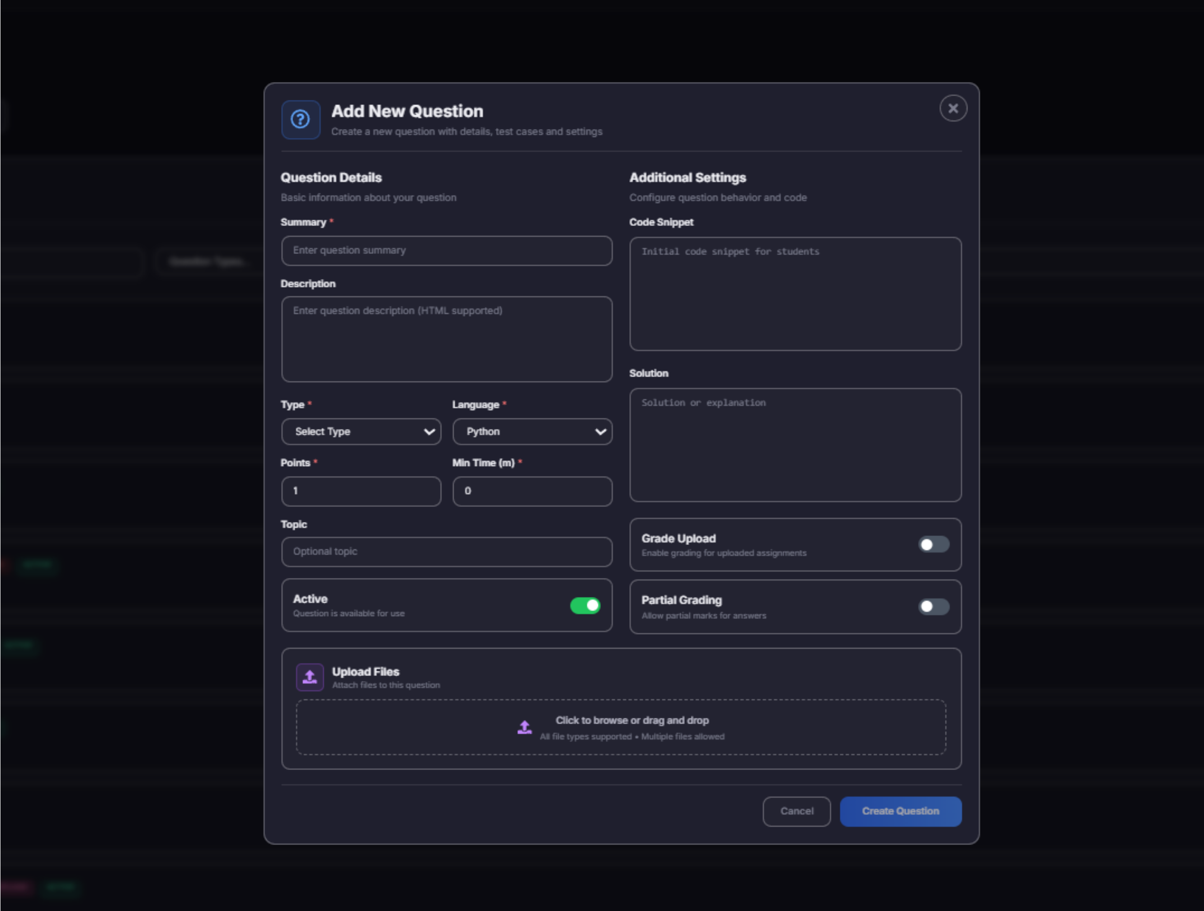

Figure 6: Add Question Modal— instructors create & edit questions

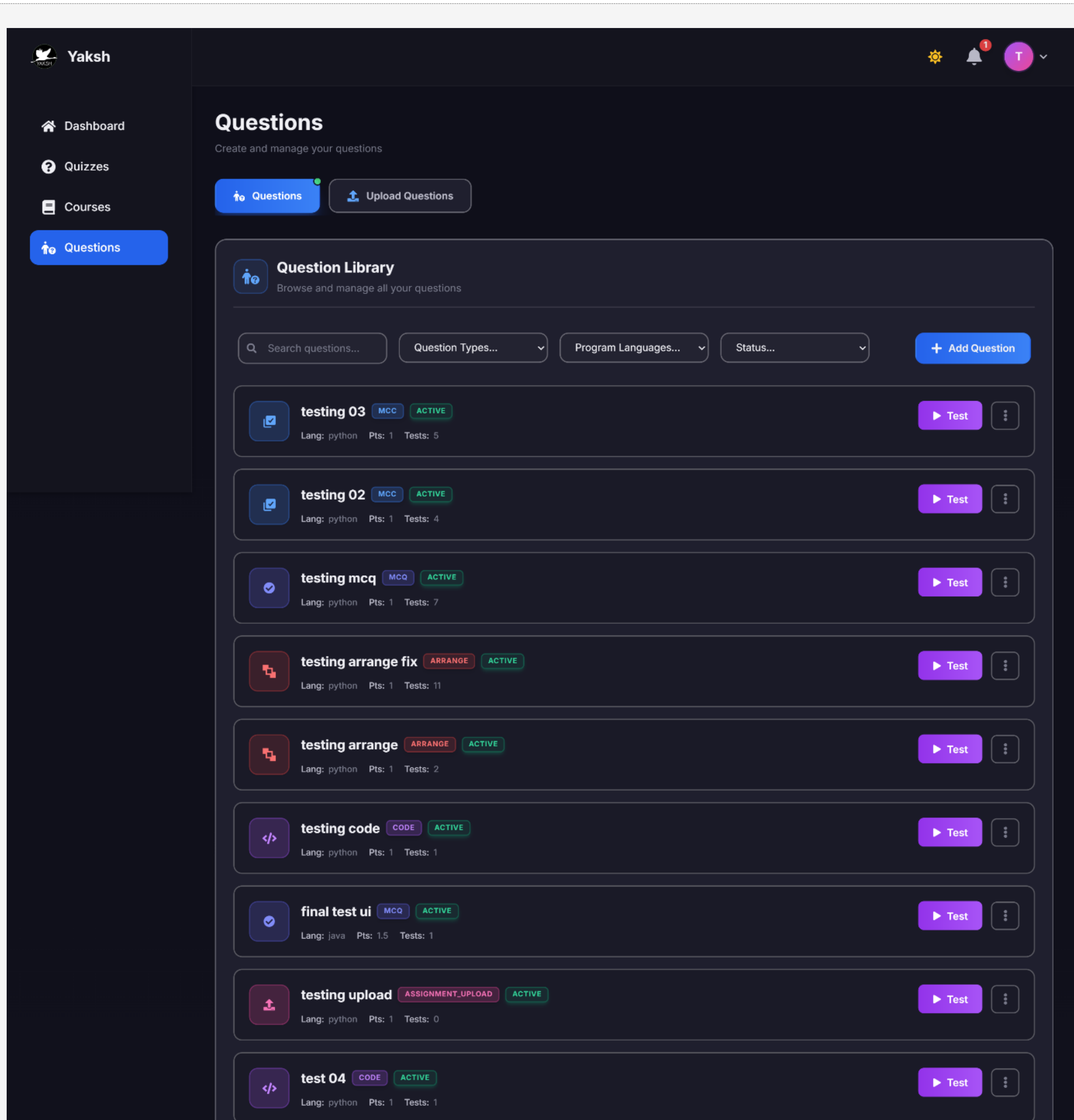

Figure 7: Question Library— instructors browse, manage & test questions

6.2.3 Analytics, Grading, and Live Monitoring

Three dedicated dashboard components were built for instructors**: QuizGradingPanel** for reviewing and overriding student scores, **CourseAnalytics.jsx** for visualizing aggregate performance trends across the student cohort, and **QuizMonitorPanel** for near-real-time visibility into which students are currently attempting a quiz and their progress status. The monitor panel consumes the **quizMonitorStore** and periodically polls the backend monitoring endpoint to refresh student status data.

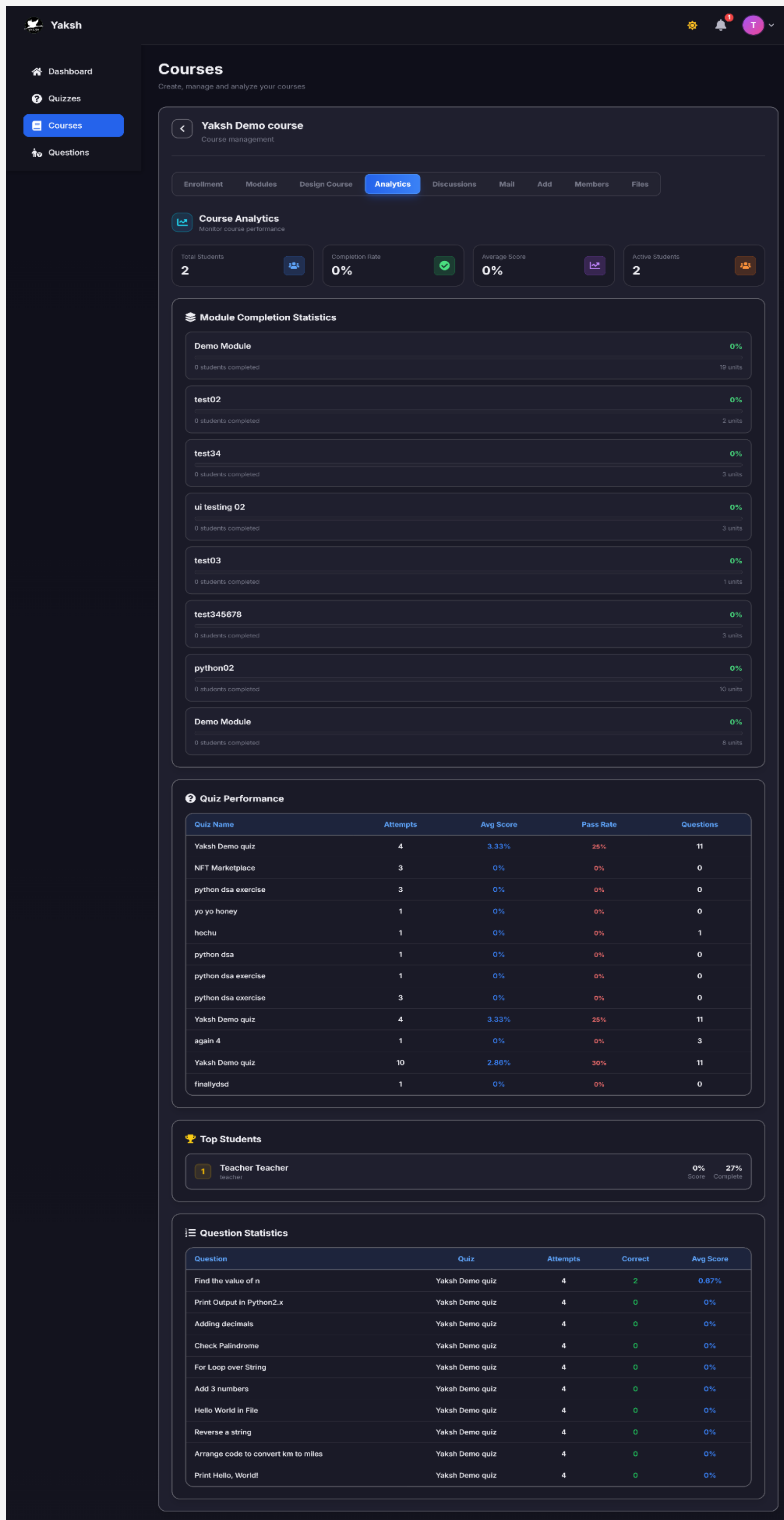

Figure 8: Course Analytics— instructors visualizes aggregate course trends

Yaksh

Dashboard

Quizzes

Courses

Questions

Quizzes

Manage and monitor all your quizzes and exercises

NFT Marketplace EXERCISE

Yaksh Demo course · Demo Module · 12/30/2025

Monitor quiz progress, download statistics, and view live data.

Stats :

Total Papers	Completed	In Progress	Questions
2	1	1	4

Download CSV

Statistics (# 1)

Upload Marks CSV

Upload a CSV file to update marks for this quiz

Choose CSV file

Upload

Attempt 1 | Attempt 2 | Attempt 3 | Attempt 4 | Attempt 5 | Attempt 6 | Attempt 7

Attempted Users

SR NO.	NAME	ROLL NO	MARKS	QUESTIONS ATTEMPTED	TIME LEFT	STATUS	TIME EXTENSION	SPECIAL ATTEMPT
1	Student Student	1234	0	0 out of 4	1509:53:31	COMPLETED	Time +	Allow
2	Teacher Teacher	12345	0	0 out of 4	1556:39:25	INPROGRESS	Time +	Allow

Figure 9: Quiz Monitor Panel — instructors sees near-real-time status of quizzes & exercises

Yaksh

Dashboard
Quizzes
Courses
Questions

Quizzes

Manage and monitor all your quizzes and exercises

NFT Marketplace EXERCISE

Yaksh Demo course · Demo Module · 12/30/2025

Select a user and their attempt to view and grade submissions

Teacher Teacher
teacher . teacher@mail.com

Select User: Teacher Teacher (teacher)

Attempt 1 | Attempt 2 | Attempt 3 | Attempt 4 | **Attempt 5** | Attempt 6 | Attempt 7

Grading Details — Save — Regrade All

Note: Only answered questions can be marked. Unanswered questions cannot be externally graded.

Paper #283 — 0 / 5 (0%)
Status: **inprogress**

1 For Loop over String

Write a python script that accepts a **string** as input
The script must print each character of the string using a for loop. For example;
Input:
box
Output
b
o
x

Answer: Not answered
Correct:
- Input: string, Output: s t r i n g
- Input: s t o p s i g n, Output: s t o p s i g n

Marks: 0 / 1
Status: Skipped

2 Add 3 numbers

Write a program to add 3 numbers.
Function Name is to be called **add**

Note: *You do not have to print anything, neither you have to make the function call.*
Just define the function to perform the required operation, return the output & click on check answer.
Also, note that the function name should exactly be as mentioned above.

Answer: Not answered
Correct:
- #include #include extern int add(int, int, int); template void check(T expect,T result) { if (expect == result) { printf("\nCorrect:\n Expected %d got %d \n",expect,result); } else { printf("\nIncorrect:\n Expected %d got %d \n",expect,result); exit (1); } } int main(void) { int result; result = add(0,0,0); printf("Input submitted to the function: 0, 0, 0"); check(0, result); result = add(2,3,3); printf("Input submitted to the function: 2, 3, 3"); check(8,result); printf("All Correct\n"); }

Marks: 0 / 2
Status: Skipped

3 Square of two numbers

Write a Java code which will take an **integer as input** and **print the square** of the integer -

Answer: Not answered
Correct:
- Input: 2, Output: 4
- Input: 4, Output: 16
- Input: 0, Output: 0
- Input: -3, Output: 9

Marks: 0 / 1
Status: Skipped

4 one more

second

Answer: Not answered
Correct:
- Input: 23 56 98, Output: we are the
- Input: 09876, Output: 98 75

Marks: 0 / 1
Status: Skipped

Figure 10: Quiz Grading Panel — instructors reviews & overrides scores

6.3 The Student Portal

The student experience was designed to be **distraction-free**, keeping **similarity with the Teacher side** to keep the theme and structure of the app same. Particular **emphasis** was placed on **maintaining focus** and **preventing loss of work** during the critical quiz-taking session.

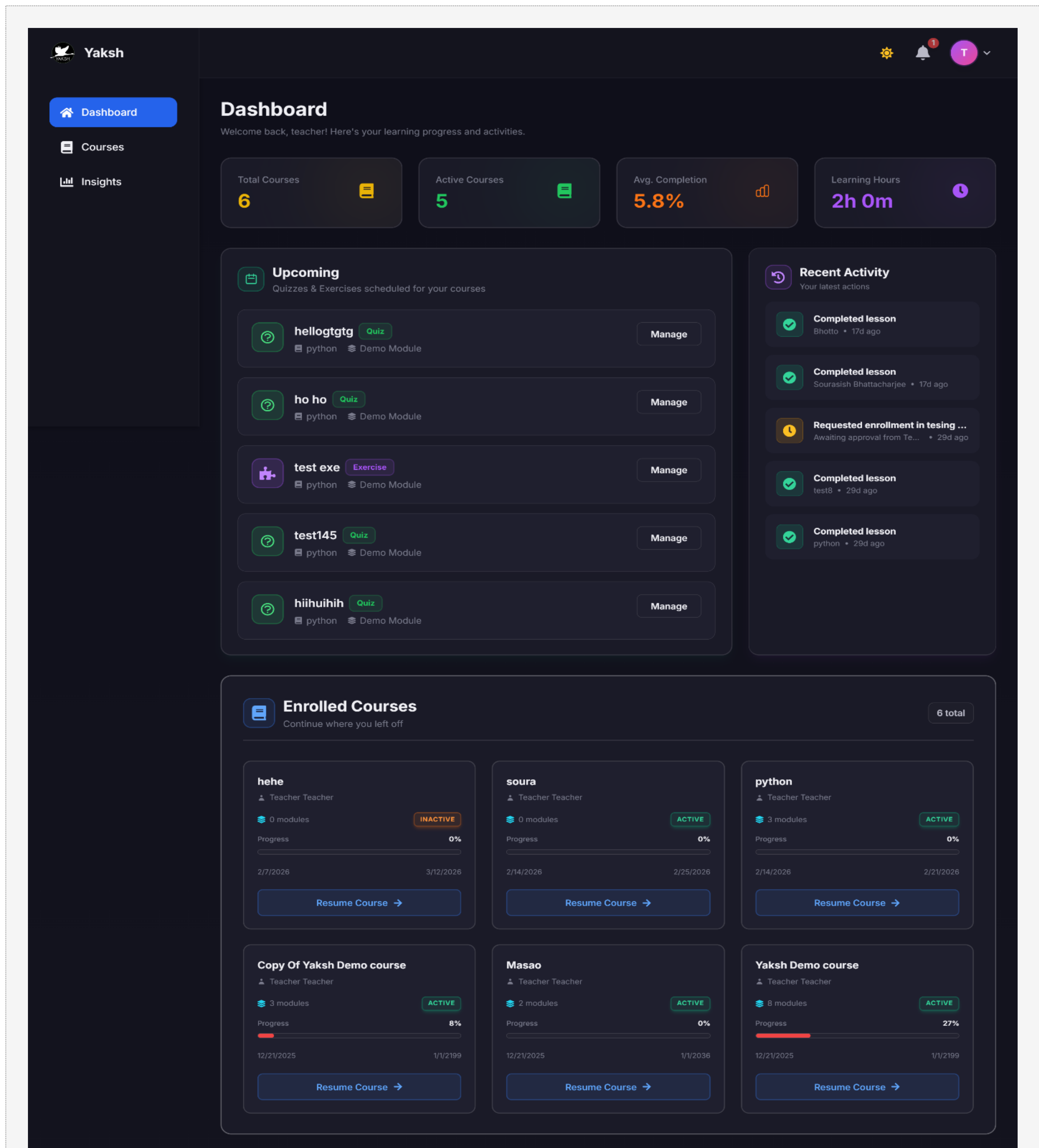

Figure 11: Student Dashboard — course listing with enrollment and progress indicators

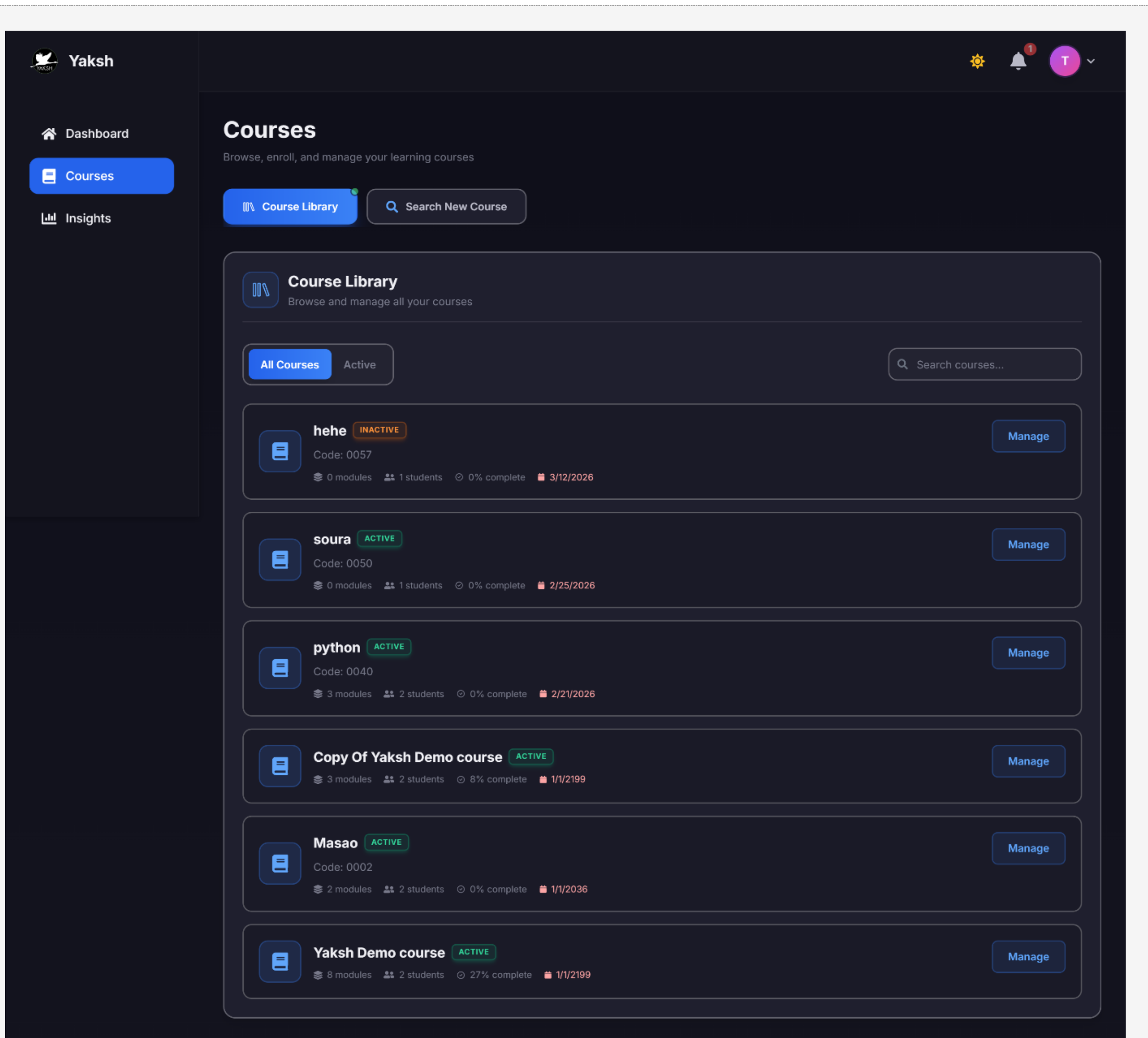

Figure 12: Enrolled Courses Library — students can browse and manage enrolled courses

Yaksh

Dashboard
Courses
Insights

Courses

Browse, enroll, and manage your learning courses

Yaksh Demo course

Course management

Modules | Discussions

Course Modules

Track your learning progress

Total: 8

Module	Description	Progress	Units
Demo Module	<center>Demo Module</center>	37%	19
test02	testing	0%	2
test34	testing	0%	3
ui testing 02	ui testing 02	0%	3
test03 (COMPLETED)	holla amigos	100%	1
test345678	treeff02	0%	3
python02	hachu chacuvygbuh	30%	10
Demo Module	<center>Demo Module</center>	12%	8

STATUS	UNIT DETAILS	TYPE	ACTIONS
COMPLETED	Yaksh Demo quiz ORDER #1	Quiz	Answer Paper Retry
NOT STARTED	Demo Lesson ORDER #1	Lesson	Start
QUITTED	NFT Marketplace ORDER #3	Exercise	Retry
QUITTED	python dsa exercise ORDER #4	Exercise	Retry
NOT STARTED	test04 ORDER #5	Lesson	Start
NOT STARTED	test9 ORDER #6	Lesson	Start
QUITTED	finallydsd ORDER #7	Quiz	Answer Paper Retry
QUITTED	nft 4567 ORDER #8	Exercise	Retry

Figure 13: Course Modules —students manage course modules progress

6.3.1 Active Assessment Interface (Quiz.jsx & Submission.jsx)

The core test-taking engine was built in **Quiz.jsx** and **Submission.jsx**. It handles strict **timer synchronization** (with the timer state managed in Zustand to persist across internal navigation), autosaving of draft answers at configurable intervals to prevent loss of work, and the rendering of different question types in their respective input modes. The **submission flow includes a confirmation modal that summarizes unanswered questions** before final submission.

6.3.2 Post-Examination Insights

Insights.jsx and **ViewAnswerPaper.jsx (committed March 25)** provide students with post-examination feedback. The ViewAnswerPaper component presents a side-by-side diff of the student's submitted code versus the correct reference solution. This was a highly requested feature that the legacy template-based system did not support, and its implementation significantly improves the educational value of the platform.

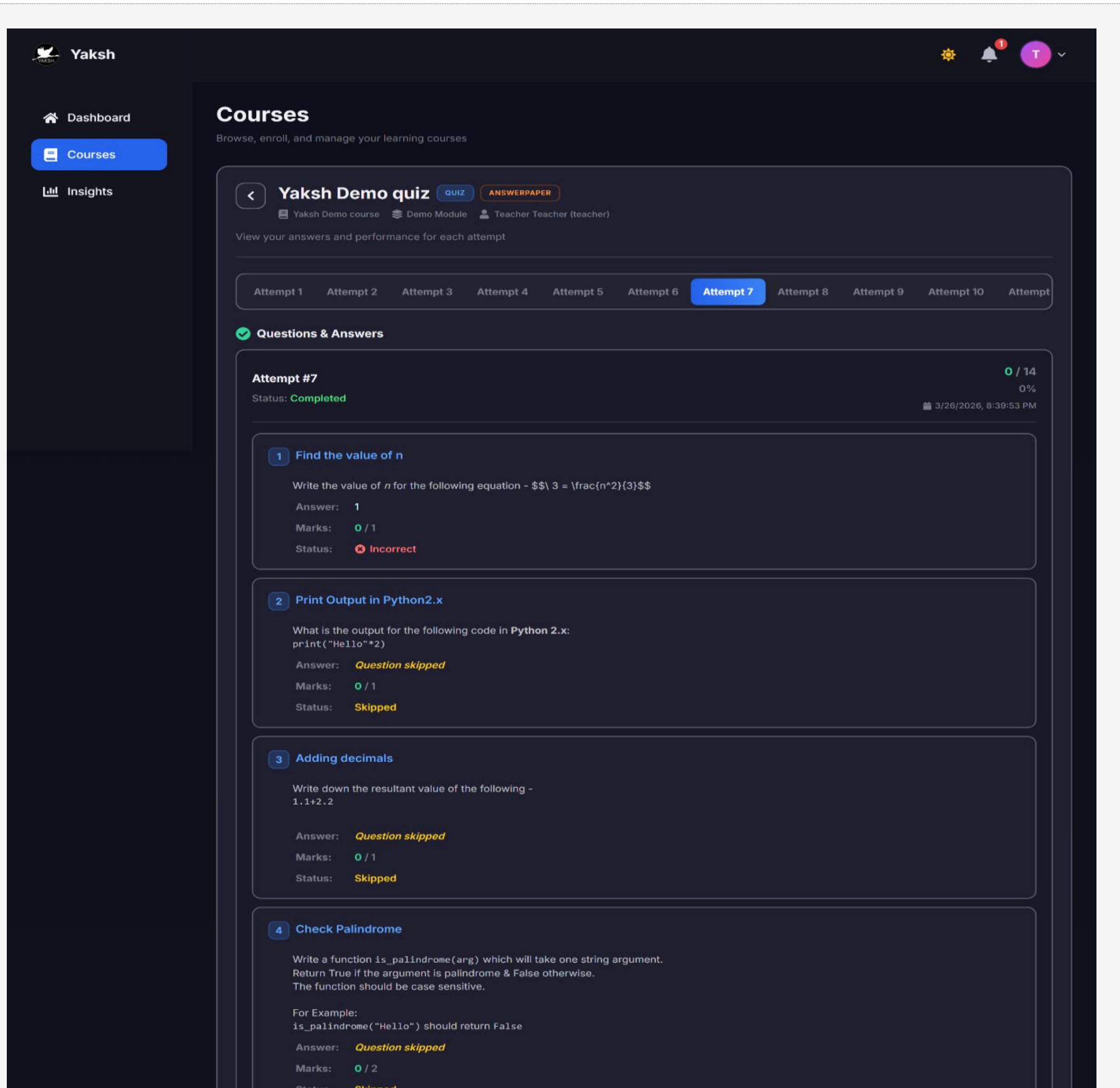

Figure 14: View Answerpaper page —students gets post quiz/exercise feedback

6.3.3 Course Discussion Forum

CourseDiscussion.jsx was developed to provide **course & level forum interactions**, allowing **students(as well as teachers)** to get involved through posts and discuss course and lesson topics through comments within the context of a specific course/lesson posts . The forum state is managed by **forumStore.js** and backed by corresponding discussion API endpoints.

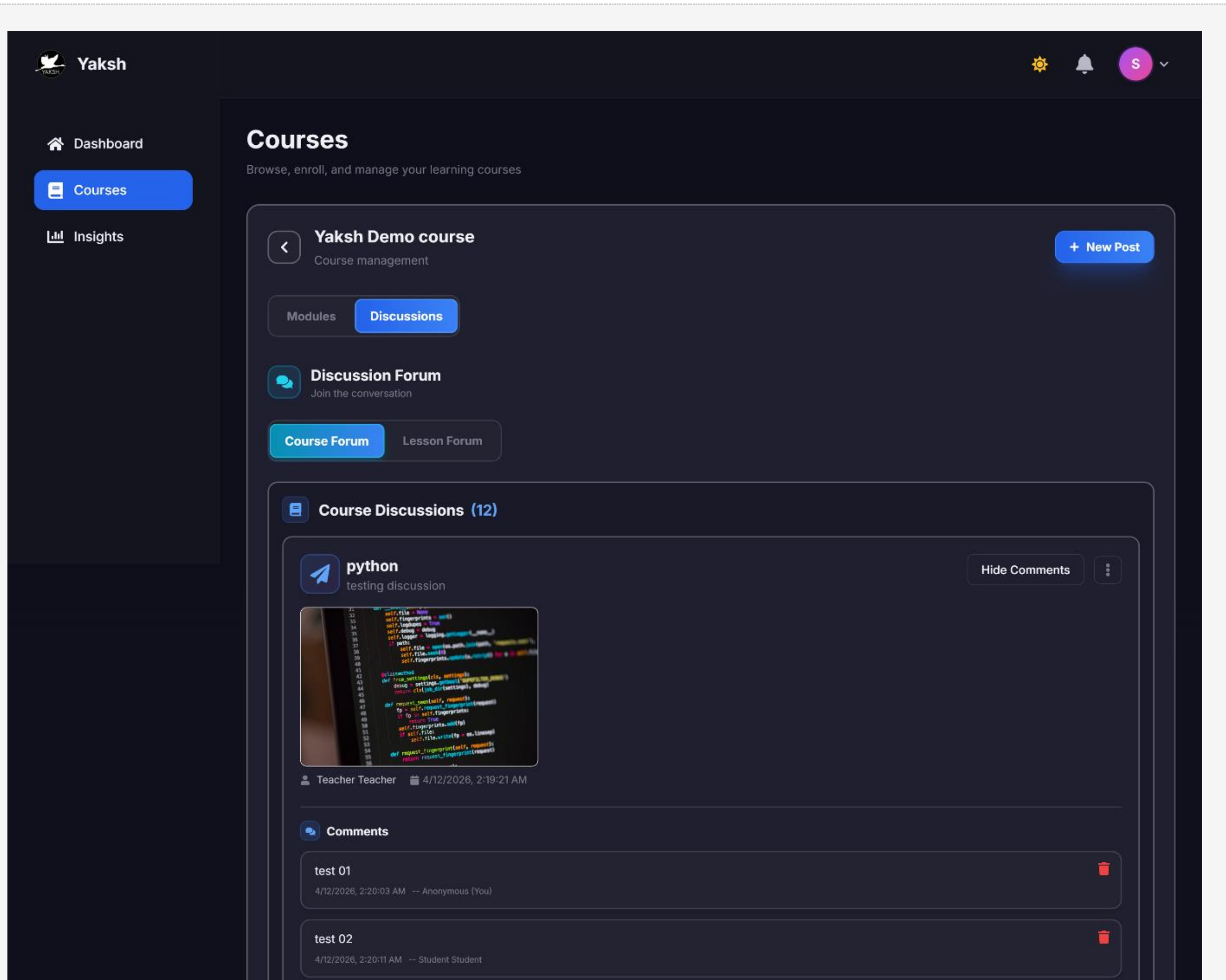

Figure 15:View Answerpaper page —students gets post quiz/exercise feedback

6.4 Theming, Layout, and UI/UX Pipeline

- **ThemeController.jsx** was implemented to manage **application-wide light/dark mode** switching, with the user's preference persisted in authStore.
- **Centralized layout components — Sidebar.jsx and Header.jsx** — were built to maintain **DRY (Don't Repeat Yourself)** principles, ensuring consistent navigation chrome across all authenticated views.

7. Technical Challenges and Solutions

7.1 Complex Test Case Option Bundling — Data Adapter Pattern

7.1.1 The Problem

A significant architectural mismatch existed between the **legacy Yaksh backend schema** and the **React frontend** for three question types: **Multiple Choice (MCQ), Multiple Correct Choices (MCC), and Arrange**. The legacy schema persisted **each answer option as an individual database row (one row per McqTestCase or ArrangeTestCase),** while the React components expected a **Question object to contain a single, clean nested array of options for efficient rendering**.

MCC questions added further complexity by requiring an **array of correct-answer** indices **rather than a single value, and Arrange questions** demanded strict sequence preservation **— both incompatible with the flat, row-per-option database layout.**

7.1.2 The Solution — Read / Write / Evaluate Adapter

Rather than migrating the **legacy schema** — which would have cascaded into **broken migrations and grading logic across Yaksh** — a three-phase data adapter pattern was implemented **entirely at the API layer,** keeping the database untouched.

Phase 1 — On Read (GET): Bundling rows into a React-friendly array (api/serializers.py)

The **get_test_cases** method inside **QuestionSerializer** intercepts the flat row list returned by **get_test_cases_as_dict()** and **bundles** it into a **structured object** before the response reaches React. For **MCQ/MCC**, **individual option rows** are collapsed into a **single options** array; for **MCC** the correct indices are **tracked as a list**; for **Arrange questions**, the **separate row entries** are stitched into a single **ordered array**.

```
def get_test_cases(self, obj):
    try:
        tc_list = obj.get_test_cases_as_dict()

   # Bundle multiple ArrangeTestCase options into a single array for
React
        if obj.type == 'arrange' and tc_list:
            arrange_options = [tc.get('options') for tc in tc_list if
tc.get('type') == 'arrangetestcase']
            if arrange_options:
```

```
    # Keep the first testcase structure and replace options with the
array
                first_tc = tc_list[0].copy()
                first_tc['options'] = arrange_options
                return [first_tc]

    # Bundle MCQ/MCC into a single test case for React
        if obj.type in ['mcq', 'mcc'] and tc_list:
            mcq_tcs = [tc for tc in tc_list if tc.get('type') ==
'mcqtestcase']
            if mcq_tcs:
                first_tc = mcq_tcs[0].copy()
                options_array = []
                correct_data = [] if obj.type == 'mcc' else 0

                for idx, tc in enumerate(mcq_tcs):
      # FOSSEE stores options as JSON '["Option text"]'
                    try:
      # Safely extract option string
                        opt_val = tc.get('options', '[]')
                        if isinstance(opt_val, str) and
opt_val.startswith('['):
                            opt_val = json.loads(opt_val)
                        opt_text = opt_val[0] if isinstance(opt_val,
list) and opt_val else str(opt_val)
                    except Exception:
                        opt_text = str(tc.get('options', ''))

                    options_array.append(opt_text)

                    if tc.get('correct') or tc.get('correct') ==
'True':
                        if obj.type == 'mcc':
                            correct_data.append(idx)
                        else:
                            correct_data = idx

                first_tc['options'] = options_array
                first_tc['correct'] = correct_data
                return [first_tc]

        return tc_list
    except Exception:
        return []
```

Phase 2 — On Write (POST/PUT): Unpacking arrays back into individual DB rows (api/views.py)

When the teacher form submits a bundled options array, **teacher_create_question** and **teacher_update_question** in views.py **unpack** it. For **MCQ/MCC**, a **separate McqTestCase row is created** per option, with the **correct flag set based on the submitted index list**. For **Arrange**, each item gets its **own ArrangeTestCase row inserted in sequence**.

```
if tc_type == 'mcc' or tc_type == 'mcqtestcase':
    # For MCQ and MCC, we need multiple McqTestCase entries (One for
each option)
    options = tc_data.get('options', [])
    if isinstance(options, str):
        try:
            options = json.loads(options)
        except Exception:
            options = [options]

    correct_indices = tc_data.get('correct')
    # Standardize correct answer(s) into a list for iteration
    if not isinstance(correct_indices, list):
        correct_indices = [correct_indices] if correct_indices is not
None else []

    # Filter empty options dynamically as they arrive from frontend
    cleaned_options = [opt for opt in options if str(opt).strip()]

    model_class = get_model_class('mcqtestcase')
    for idx, option in enumerate(cleaned_options):
        model_class.objects.create(
            question=question,
            options=str(option).strip(),
            correct=(idx in correct_indices),
            type='mcqtestcase'
        )

elif tc_type == 'arrangetestcase':
    options = tc_data.get('options', [])
    if isinstance(options, str):
        try:
            options = json.loads(options)
        except Exception:
            options = [options]

    if isinstance(options, list):
        # Create a distinct DB row for each option sequentially
        for opt in options:
```

```
            if str(opt).strip():  # Ignore empty lines
                model_class.objects.create(
                    question=question,
                    options=str(opt).strip(),
                    type=tc_type
                )
    else:
        model_class.objects.create(
            question=question,
            options=str(options),
            type=tc_type
        )
```

Phase 3 — On Evaluation: Mapping frontend indices back to Django model IDs (api/views.py)

When a student **submits** their **Arrange answer** (a comma-separated sequence of 1-based position integers), the view maps these **back to actual ArrangeTestCase primary keys** that the Yaksh grading engine expects, using Python-side sorting to match the exact load order the student saw in the UI.

```
elif current_question.type == 'arrange':
    answer_str = request.data.get('answer', '')
    user_indices = []
    if isinstance(answer_str, str):
        user_indices = [int(ids) for ids in answer_str.split(',') if 
ids.strip()]
    elif isinstance(answer_str, list):
        user_indices = [int(ids) for ids in answer_str]

    # Map 1-based frontend indices to actual ArrangeTestCase IDs
    if user_indices:
        # Avoid Django order_by collisions, use python sorting to 
mimic the UI load order
        actual_test_cases = sorted(current_question.get_test_cases(), 
key=lambda x: x.id)

        user_answer = []
        for idx in user_indices:
            list_idx = idx - 1  # Convert 1-based to 0-based
            if 0 <= list_idx < len(actual_test_cases):
                user_answer.append(actual_test_cases[list_idx].id)
    else:
        user_answer = []
```

This **three-phase adapter — bundle on read, unpack on write, remap on evaluate —** ensured the React frontend could work with clean array-driven data structures while the legacy

Yaksh database and its assessment engine remained completely intact, without a single schema migration.

7.2 User Mode / God Mode for Instructor Quiz Testing

7.2.1 The Problem

Instructors need to verify that their own quiz questions work correctly — exactly as a student would experience them — before publishing. Testing questions directly against a live course is not viable: **instructors are blocked by prerequisite checks, inactive module states, and enrollment validation**. Worse, if an **instructor did successfully attempt their own quiz, the resulting AnswerPaper record would be permanently tied to the live course, silently corrupting analytics, average scores, and grading matrices.**

7.2.2 The Solution — Isolated Sandbox Architecture

Rather than patching live endpoints to skip validation rules, **a transient sandbox architecture** was built using **a test_mode encapsulation engine**. The solution has four interlocking parts:

- **Dynamic Mock Entity Generation.** When an instructor clicks to test a question or quiz, the frontend calls **teacher_test_question**. The backend invokes a **FOSSE**E utility called **test_mode(user),** which provisions a transient, database-isolated environment — **a Trial Course, Trial Module, Trial Learning Unit, and a Trial Question Pape**r wrapping only the question(s) under test. All generated entities are flagged with **is_trial=True**.

Bypass logic — api/views.py

```
# From api/views.py
is_trial_mode = course.is_trial and is_moderator(user)

if not is_trial_mode:
    # Standard prerequisite checks, enrollment validation,
    # and course active queries are executed here.
    validate_enrollment(user, course)
    check_module_prerequisites(user, module)
```

- **Bypassing Prerequisites (the "God Mode" flag).** Because the instructor is operating **inside the Trial Course, is_trial_mode evaluates to True** and all student validation gates — enrollment locks, active module checks, prerequisite sequences — are cleanly bypassed, granting immediate testing access without touching the live course logic.

- **Frontend Execution Parity.** Because the trial course mirrors the exact structure of a live course, the frontend does not need a specialized testing engine. **TestQuestion.jsx** feeds the **trial IDs** into the same **startQuiz and submitAnswer**

Zustand actions (**quiz_QuestionStore.js**) that **real students use** — guaranteeing 1:1 runtime parity between the instructor's test run and the actual student experience.

- **Analytics Isolation.** Because the attempt takes place inside the **trial course (is_trial=True)**, standard analytics queries across the backend filter it out automatically. Queries of the **form user.students.filter(is_trial=False)** ensure that **grading summaries, average scores, and participation dashboards** remain completely unpolluted by instructor QA runs.

The result is a **self-contained testing environment** that gives **instructors full student-parity execution**, with zero risk of contaminating live course data or analytics.

8. Development Workflow and Timeline

The internship followed a continuous deployment philosophy, with iterative commits made directly to feature branches that were subsequently reviewed and merged. The Git commit history serves as a precise audit trail of the project's lifecycle.

8.1 Phase-wise Development Summary

Phase	Period	Key Activities
Design & Onboarding	Oct – Dec 2025	Platform study, initial wireframing, supervisor approval on layout direction, repository setup and environment configuration.
Core Instructor Functionality	Jan 2026	Course design API, question bank logic, drag-and-drop quiz builder, live quiz monitor, bulk user enrollment bug resolution.
Student Implementation	Feb – Mar 2026	Student course pipeline, lesson-to-quiz engine connection, discussion forums, regrading API, ViewAnswerPaper (committed Mar 25).
QA & Final Refinements	Late Mar – Apr 2026	God Mode / User Mode logic, 11 successive UI Fixes branches (padding, mobile views, API crash prevention), final production build in April 2026.

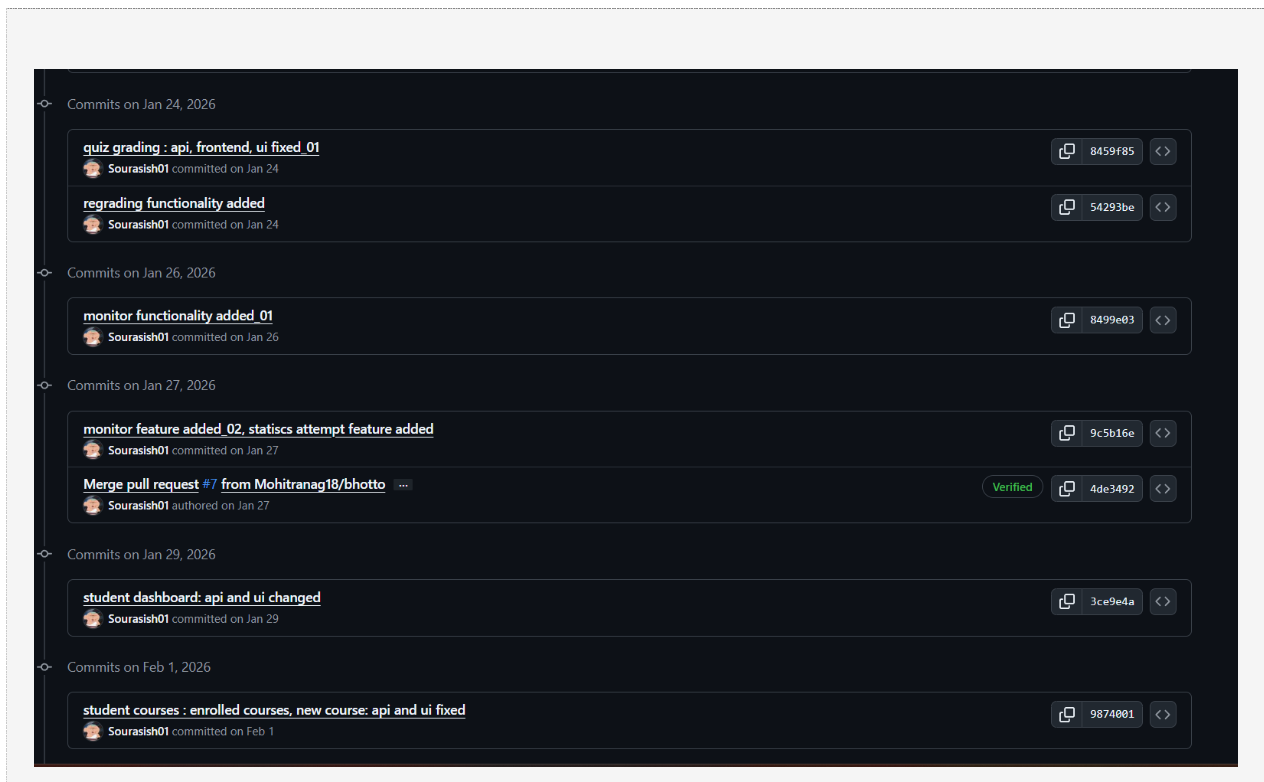

Figure 16:Git commit history

- Pull Request : https://github.com/FOSSEE/online_test/pull/870
- Commit History: https://github.com/Mohitranag18/online_test/commits?author=Sourasish01
- Common Repo: https://github.com/Mohitranag18/online_test

9. Conclusions and Future Scope

9.1 Summary of Contributions and Impact

Over the course of this semester-long internship, the **Yaksh online assessment platform** was transformed **from a server-side rendered**, **monolithic Django application into** a **modern, decoupled web system**. An **initial wireframing** phase **helped establish the layout direction** before development began, **with further improvements** along the development phase. The implementation of a **React** powered by **Vite**, **Zustand**, and **Tailwind CSS** yielded a **highly performant, responsive application.**

From a concrete functional standpoint, instructors can now **build full course curriculam**, **author multi-type question banks**, **configure grading schemes**, and **monitor live quiz activity** — all **without a single full-page reload**. Students experience a **seamless, timer-safe, autosaving assessment environmen**t with access to meaningful post-exam feedback. The **establishment of the RESTful api layer** has future-proofed the platform's architecture, cleanly **separating the visual presentation layer from the core data and evaluation logic**.

9.2 Skills Developed

9.2.1 Technical Skills

- **UI/UX wireframing and prototyping using Figma** to establish initial layout direction.
- Proficiency in modern **React ecosystem**: **React 19, Vite, Zustand, React Router DOM v7**.
- **Tailwind CSS** for utility-first, responsive design with design-token mapping.
- **RESTful API design** and implementation using **Django REST Framework**.
- **JWT authentication**, **Axios interceptors**, and **secure SPA routing patterns**.
- **State management architecture design** for complex multi-role applications.
- **Git workflow management** in a team environment with iterative feature branching.

9.2.2 Professional Skills

- Stakeholder communication: presenting and iterating on design prototypes based on supervisor feedback.
- Technical documentation and structured report writing.
- Agile development philosophy: iterative development, continuous deployment, and progressive feature delivery.

9.3 Future Scope

Several promising directions remain for future development of the Yaksh SPA:

- **WebSockets Integration**: Upgrading the QuizMonitorPanel from HTTP API polling to real-time WebSockets via Django Channels would **reduce latency and provide instructors with a genuinely live view of student activity.**
- **Progressive Web App (PWA)**: Configuring the Vite pipeline to enable PWA capabilities — including **service worker registration and offline caching of course content** — would make **Yaksh accessible in low-connectivity** environments prevalent in rural India.
- **Monaco Editor Integration**: Replacing the current text-area based code input with the Monaco Editor (the engine behind VS Code) would give students a **full-featured coding environment with syntax highlighting, linting, and auto-complete during programming assessments.**
- **Accessibility Improvements**: While basic contrast ratios were addressed in the design system, a comprehensive WCAG AA audit and remediation pass would make the platform more inclusive.

References

1. FOSSEE Team, IIT Bombay, "Yaksh: Open-Source Online Test Platform," https://yaksh.fossee.in, 2025.
2. Meta Platforms Inc., "React Documentation," https://react.dev, 2025.
3. Evan You et al., "Vite — Next Generation Frontend Tooling," https://vitejs.dev, 2025.
4. Daishi Kato, "Zustand: Bear Necessities for State Management," https://github.com/pmndrs/zustand, 2025.
5. Adam Wathan et al., "Tailwind CSS Documentation," https://tailwindcss.com/docs, 2025.
6. Django Software Foundation, "Django REST Framework," https://www.django-rest-framework.org, 2025.
7. FOSSEE Project, IIT Bombay, https://fossee.in, 2025.
8. National Mission on Education through ICT (NMEICT), Ministry of Education, Government of India, https://nmeict.ac.in, 2025.