KING'S College LONDON

Faculty of Natural, Mathematical, and Engineering Sciences
Department of Informatics

Software Engineering Group Project

Major Group Project

Infinite Loop Innovators
Tala Alqarzaee
Iliyan Adov
Manusha Ravindrathas
Thomas Holden
Arham Zahid
Yuanhao Li
Deyu Li

March 27, 2025

Contents

Chapter 1

Introduction

This project presents a comprehensive platform designed to enhance university student engagement by streamlining society discovery, membership, and management. Universities implementing this system benefit from increased student participation in extracurricular activities, improved community building, and efficient administrative oversight of student organizations. For students, the platform eliminates common barriers to society involvement through personalized recommendations and simplified interaction with university clubs, ultimately enriching their academic experience and fostering a sense of belonging within the campus community.

The system is a responsive web application built using Django for the back-end and React for the front-end. Our technology stack incorporates a sophisticated AI recommendation system utilizing neural embeddings and domain-specific semantic relationships to match students with societies aligned to their interests. The application features real-time updates through WebSockets integration, comprehensive user authentication with email verification, and a multi-tiered permission structure to accommodate the needs of various stakeholders including students, society leaders, and university administrators.

Chapter 2

Objectives and Stakeholders

2.1 Project Objectives

The primary objective of this project is to create a cohesive digital ecosystem that strengthens university community engagement through improved society management and participation. This objective can be broken down into the following components:

- **Enhancing student participation** in extracurricular activities by simplifying discovery and reducing barriers to society involvement
- **Enabling efficient management** of university societies through specialized tools for leadership roles
- **Creating meaningful connections** among students with similar interests who might otherwise not interact
- **Providing administrative oversight** for university staff to ensure societies operate within institutional guidelines

The realization of these objectives delivers value to the university community by enriching student experiences beyond academic pursuits, fostering skill development through organized activities, and strengthening institutional community bonds.

2.2 Stakeholders

2.2.1 Stakeholder Analysis

Our stakeholder analysis identified several groups affected by or influencing the implementation of this platform:

Students As the primary users and beneficiaries, students gain improved access to extracurricular activities that complement their academic experience. The platform allows them to discover societies aligned with their interests through AI-driven recommendations, easily join groups, track events, and connect with peers. Students' university experience is directly enhanced through increased participation opportunities.

Society Leaders Presidents, vice presidents, and event managers benefit from specialized tools that simplify administrative tasks. They can manage membership requests, organize events, publish news, and communicate with members efficiently. This reduces their administrative burden and allows them to focus on creating valuable experiences for their members.

University Administrators The administrative staff gains enhanced oversight of all student activities through a comprehensive dashboard. They can ensure compliance with university policies, approve new societies and events, and maintain quality standards. The platform streamlines their workflow by centralizing student organization management.

University as an Institution The university benefits from increased student engagement and satisfaction, which can positively impact retention rates and institutional reputation. A vibrant student community with active societies contributes to a positive campus environment and enhances the university's appeal to prospective students.

Academic Staff Faculty members indirectly benefit when students develop complementary skills through society participation. These extracurricular activities can reinforce classroom learning and provide practical application opportunities for academic concepts.

Potential Students Prospective students considering enrollment decisions are influenced by the richness of campus life and available activities. A well-organized society ecosystem, visible through the platform, serves as a recruitment asset.

External Partners Organizations that collaborate with university societies (such as sponsors, community groups, or industry partners) benefit from more organized interaction through the platform's structured communication channels.

Alumni Former students maintain connections to their university societies through the platform, potentially increasing alumni engagement and support.

The implementation of this platform creates different value propositions for each stakeholder group. For students, the impact is immediate and direct—they gain personalized access to activities that enhance their university experience. Society leaders transform their management capabilities through specialized tools that reduce administrative friction. University administrators receive unprecedented visibility into student activities, allowing for better resource allocation and policy enforcement.

2.2.2 Key Stakeholders

Based on our analysis, we identified the following key stakeholders whose interests and requirements significantly shape the project outcomes:

- **Students** are the primary users whose engagement determines the platform's success. Their requirements for intuitive navigation, personalized recommendations, and social connectivity directly influence design decisions. The platform's value is primarily measured through student adoption and participation rates.
- **Society Leaders** are crucial for content generation and community building within the platform. Their ability to effectively manage membership, events, and commu-

nications directly impacts student experience. Their satisfaction with management tools is essential for sustainable platform growth.

- **University Administrators** hold significant influence over platform implementation and institutional adoption. Their oversight requirements regarding policy compliance, appropriate content, and resource allocation shape the platform's governance features. Administrative approval mechanisms are integrated throughout the system based on their operational needs.

These key stakeholders formed the core focus during requirements gathering and feature prioritization. Regular consultation with representatives from each group ensured that the platform addresses their specific needs while balancing overall project objectives.

Chapter 3

Specifications

This chapter outlines the functional and non-functional specifications of our student society management platform. All features described below have been successfully implemented and deployed in the final system.

3.1 Functional Specifications

3.1.1 User Authentication and Profile Management

- **University-Restricted Registration**: System validates that email addresses end with the university domain (e.g., @kcl.ac.uk) to ensure only legitimate university students can register.
- **Two-Step Verification**: Registration process includes email verification through OTP (One-Time Password) to confirm email ownership.
- **Profile Customization**: Users can set and modify their username, password, and academic major.
- **Social Connections**: Students can search for, follow, and view profiles of other students, facilitating community building.

3.1.2 Personalized Student Dashboard

- **Activity Overview**: Dashboard displays membership counts, upcoming events, and unread notifications.
- **Society Management**: Students can view societies they've joined with options to leave if desired.
- **Event Tracking**: Comprehensive display of all events with RSVP functionality and attendance management.
- **Notification System**: Centralizes all society-related communications with read-/unread status tracking.
- **News Feed**: Displays news published by societies the student has joined.

- **Calendar Integration**: Visual calendar interface showing all events for better planning.
- **Society Application**: Interface for students to propose new societies, requiring administrative approval.
- **Search Functionality**: Allows searching for events and societies across the platform.

3.1.3 Society Management Tools

- **Society Detail Management**: The society managers can edit the name, description, category, social media links, images, icons, and tags of the society (subject to admin approval).
- **Event Creation and Management**: Interface for creating, editing, and managing society events with approval workflows.
- **Membership Administration**: Tools for reviewing pending membership requests and managing existing members.
- **News Publication System**: Functionality to create, draft, publish, and manage society news posts.
- **Member Recognition**: System for assigning awards and roles to society members.
- **Preview Functionality**: Live preview of the society page and events before submission for approval.
- **Administrative Communication**: Direct reporting channel to university administrators for issue resolution.

3.1.4 Administrative Oversight Dashboard

- **Platform Analytics**: Real-time metrics showing active users, events, and pending requests.
- **Student Management**: Comprehensive student database with filtering, search, and administrative actions.
- **Society Approval Workflow**: Tools for reviewing and approving new societies and society modification requests.
- **Event Approval System**: Interface for reviewing event requests to ensure compliance with university policies.
- **Report Management**: System for handling and responding to reports from users.
- **Administrative Team Management**: Tools for managing admin permissions and responsibilities.
- **Activity Logging**: Comprehensive tracking of administrative actions with undo functionality.

3.1.5 AI-Powered Recommendation System

- **Multi-faceted Similarity Scoring**: Combines neural embeddings (35%), TF-IDF cosine similarity (25%), keyword overlap (15%), basic word overlap (5%), and domain-specific semantic relationships (20%).
- **Diversity Algorithm**: Implements Maximal Marginal Relevance to balance relevance with variety in recommendations.
- **Temporal Awareness**: Prioritizes societies with recent activity in recommendations.
- **Adaptive Feedback System**: Collects and processes both explicit ratings and implicit user behavior signals.
- **Personalization**: Recommendations improve over time based on user interactions and preferences.

3.2 Non-functional Specifications

3.2.1 Usability

- **Desktop-Optimized Design**: Platform is designed primarily for desktop use, with limited mobile responsiveness.
- **Accessibility Features**: Interface follows accessibility best practices for inclusive user experience.
- **Appearance Customization**: Users can switch between light and dark modes to enhance readability.
- **Consistent Navigation**: Intuitive navigation patterns maintained across all system interfaces.

3.2.2 Security and Privacy

- **Role-Based Access Control**: Multi-tiered permission structure ensuring users only access appropriate functionality.
- **Data Protection**: Personal information is securely stored with multiple protection measures: user profiles are not searchable by non-logged-in users, email addresses are hidden from other users, authentication tokens are encrypted, and sensitive data is appropriately anonymized where necessary.
- **Approval Workflows**: Multi-step approval processes for society creation, event scheduling, and content publication.
- **Activity Monitoring**: Administrative logging of system actions.

3.2.3 Performance and Reliability

- **Real-time Updates**: WebSocket integration for immediate notification delivery and content updates.

- **Concurrency Management**: System handles multiple simultaneous users without degradation.
- **Data Integrity**: Robust data validation and error handling throughout the system.

3.2.4 Scalability

- **User Capacity**: Architecture designed to accommodate the entire university student population.
- **Society Growth**: No artificial limits on the number of societies or events the system can manage.
- **Data Retention**: Historical events and society information maintained with appropriate archiving.

3.2.5 Maintainability

- **Component Modularity**: System designed with clear separation of concerns for easier updates.
- **Technology Stack Standardization**: Consistent use of Django and React frameworks across the application.
- **Code Organization**: Logical structure facilitating future extensions and modifications.

3.3 Out of Scope

While the implemented system provides comprehensive society management capabilities, the following features were considered but determined to be outside the project scope:

- **Financial Management**: Tools for society budget tracking and financial reporting.
- **External Event Ticketing**: Integration with third-party ticketing platforms for paid events.
- **Mobile Applications**: Native mobile applications for iOS and Android platforms.
- **Integration with University Learning Management Systems**: Direct connection with academic course platforms.

These features represent potential future enhancements that would build upon the current system capabilities while maintaining alignment with the core project objectives.

Chapter 4

Project Management

This chapter presents, reflects on, and evaluates our team's approach to project management throughout the development of the university society management platform. We discuss our methodology, coordination mechanisms, challenges encountered, and critical reflections on the effectiveness of our approaches.

4.1 Project Management Approach

4.1.1 Flexible Agile-Inspired Methodology

Our team adopted a flexible Agile-inspired approach to project management, drawing on iterative principles while adapting them to fit our team's specific needs and academic constraints. This decision was based on several factors:

- **Uncertain Requirements**: At project initiation, we recognized that while we had a clear vision of the platform's purpose, many specific requirements would evolve as we better understood stakeholder needs.
- **Team Composition**: Our team consisted of members with varied experience levels in software development. Our emphasis on frequent communication and knowledge sharing provided a structure for less experienced members to learn from those with more expertise.
- **Project Complexity**: The multi-tiered nature of our platform (serving different user roles with distinct needs) suggested that a sequential approach would be less effective than an iterative one that allowed us to refine features based on feedback.

4.1.2 Practical Implementation

While our approach was influenced by Agile principles, our actual implementation was adapted to better fit our team's working style and academic schedules:

- **Regular Collaboration Sessions**: We held twice-weekly team meetings (2-3 hours each) for collaborative brainstorming, decision-making, and progress reviews.
- **Documentation**: We maintained comprehensive meeting minutes to document all decisions and action items, ensuring team alignment.

- **Continuous Communication**: Daily communication in a group chat addressed immediate problems and facilitated necessary decisions between formal meetings.
- **Phased Development Approach**: Our project naturally evolved through three distinct phases: initial development with rapid feature addition (first 5 weeks), feature refinement and completion (middle period), and final refactoring and debugging (final two weeks).

4.1.3 Critical Reflection on Our Approach

Our flexible approach had both strengths and limitations:

- **Adaptability Advantage**: The absence of rigid processes allowed us to shift focus quickly when needed, particularly when transitioning from feature development to refinement and debugging.
- **Communication Challenges**: While our group chat provided continuous communication channels, important information sometimes got lost in the conversation flow. Our twice-weekly meetings helped realign everyone but occasionally left gaps in coordination between sessions.
- **Documentation Value**: Our meeting minutes proved invaluable for maintaining project continuity, especially when team members missed meetings due to personal issues.

4.2 Task Management and Workflow

4.2.1 Tool Selection and Task Organization

To manage our project tasks and workflow, we used a combination of lightweight tools:

- **Discord**: For quick unscheduled meetings
- **Teams**: For scheduled online meetings
- **WhatsApp Group**: For daily communication and quick decision-making
- **GitHub Pull Requests**: For code review and quality control

Our task organization evolved organically rather than following formal methods. We maintained a prioritized feature list that guided our development focus, with tasks generally self-assigned based on individual interests and expertise.

4.2.2 Critical Reflection on Task Management

Our approach to task management revealed several insights:

- **Initial Overestimation**: During the first few weeks, we consistently overestimated what could be accomplished, committing to more features than we could complete. This led to some feature slippage and occasional team frustration.
- **Task Granularity Issues**: Early task definitions were often too broad (e.g., "Implement student dashboard"), making progress difficult to track. We gradually

improved at breaking down complex features into more manageable components, which improved both estimation accuracy and work distribution.

- **Quality Control Evolution**: Initially, we lacked a clear process for determining when a feature was truly complete. As the project progressed, we developed more rigorous quality standards, including:
  - Code review by at least one team member
  - Manual testing on desktop browsers
  - Documentation updates

  This improved overall code quality but was implemented reactively rather than from the outset.

4.3 Team Organization and Collaboration

4.3.1 Team Structure and Work Distribution

We adopted a flexible team structure based primarily on technical strengths:

- **Natural Leadership**: Rather than assigning formal project management roles, we allowed leadership to emerge naturally based on expertise in different areas
- **Technical Specializations**: Team members gravitated toward components matching their strengths (frontend, backend, AI, etc.)
- **Collaborative Decision-Making**: Major decisions were made collectively during our twice-weekly meetings, with consensus-building as our primary approach

This flexible structure allowed us to adapt to the evolving needs of the project while leveraging each team member's unique skills.

4.3.2 Critical Reflection on Team Organization

Our team organization faced several challenges throughout the project:

- **Knowledge Silos**: By mid-project, we recognized dangerous knowledge silos forming, with only one team member understanding certain critical components. We attempted to address this through more detailed documentation and explanations during meetings, though this remained a challenge throughout the project.
- **Workload Distribution**: Our self-selection approach to tasks occasionally led to uneven workload distribution. Some team members took on significantly more complex components than others, creating bottlenecks when those components needed integration.
- **Remote Collaboration Challenges**: The team's varied schedules meant that much work was done asynchronously. We initially struggled with communication that would enable effective asynchronous work. After identifying this issue, we improved our practices by:
  - Writing more descriptive commit messages

  - Sharing a summary of the implementation in the group chat
  - Sharing explanatory screenshots in the group chat when implementing new features

4.4 Risk Management

4.4.1 Informal Risk Identification and Mitigation

Rather than conducting formal risk management exercises, our approach to risks was more organic and reactive. We identified several key risks through group discussions:

- **Technical Complexity of Multi-Role System**: We recognized that implementing different user roles (student, society president, admin) with distinct interfaces would be challenging. We addressed this by discussing the permission structure early and establishing clear boundaries between role capabilities.
- **Feature Prioritization Challenges**: With numerous possible features to implement, we faced difficult decisions about which to prioritize. We mitigated this by focusing first on core functionality that would provide the most value to users before adding more specialized features.
- **Academic Time Constraints**: University coursework deadlines created variable availability throughout the project. We acknowledged this reality and maintained flexibility in our scheduling.

4.4.2 Critical Reflection on Risk Management

Our informal approach to risk management had mixed results:

- **Feature Prioritization Challenges**: Our initial approach to feature prioritization was sometimes based more on technical interest than user value. As the project progressed, we improved our prioritization by focusing on core user journeys first. However, this initial lack of structured prioritization meant we spent time on some features that were ultimately less important than others we had to rush later.
- **Technical Debt Accumulation**: In our rush to implement features, we accumulated technical debt, particularly in the early phase. This necessity to refactor code later was anticipated but not formally planned for. Our final two weeks became heavily focused on refactoring and debugging as a result.
- **Successful Role System Implementation**: Our early attention to defining user role permissions proved beneficial. By discussing the permission structure early and clearly defining role boundaries, we avoided major redesigns later in the project. This allowed us to build different dashboard interfaces for each role with consistent and appropriate access controls.

4.5 Project Progress Monitoring

4.5.1 Progress Tracking Approach

Our project progress monitoring relied primarily on:

- **Feature Completion Tracking**: Regular reviews of which features were complete, in progress, or not yet started
- **GitHub Activity**: Monitoring commit activity and pull requests to gauge development progress
- **Biweekly Demonstrations**: Demonstrating working features during our meetings to verify progress

This lightweight approach to progress tracking aligned with our flexible project management methodology while providing sufficient visibility into project status.

4.5.2 Critical Reflection on Progress Monitoring

Our monitoring approach revealed several insights:

- **Feature vs. Quality Tension**: Our focus on feature completion sometimes came at the expense of code quality. In later stages, we adjusted by placing greater emphasis on testing and refinement rather than new feature development.
- **Development Rhythm**: We observed that productivity typically increased just before our biweekly meetings, as team members worked to complete features for demonstration. While this created a somewhat uneven development pace, it established natural checkpoints that helped maintain momentum.

4.6 Key Lessons and Future Recommendations

Based on our project management experience, we identified several key lessons that would inform our approach to future projects:

- **Balanced Structure**: While our flexible approach enabled adaptability, a slightly more structured process—particularly for code reviews and quality assurance—would have reduced technical debt without sacrificing creativity.
- **Knowledge Sharing**: More formalized knowledge transfer should be established earlier. Techniques such as pair programming and documentation requirements should be core practices rather than remedial measures.
- **Technical Debt Management**: Explicitly allocating time for refactoring and technical debt reduction should be planned from the outset rather than becoming a necessity at the end of the project.

Overall, our flexible approach provided the adaptability needed to accommodate our team's academic constraints while still delivering a functional platform. Our willingness to communicate frequently and adjust our processes as needed helped us overcome the challenges inherent in a complex project with a diverse team.

Chapter 5

Design and Implementation

This chapter discusses the key design and implementation decisions made during the development of our university society management platform. We explore the architectural choices, component organization, and significant implementation strategies that shaped our system.

5.1 Architecture

5.1.1 Overall System Architecture

We implemented a modern client-server architecture using a clear separation between frontend and backend components. This separation follows the principles of the Model-View-Controller (MVC) pattern, which promotes modularity and maintainability by isolating different aspects of the application. Our system architecture consists of:

- **Frontend Layer**: A React-based single-page application that handles user interface rendering and client-side logic
- **Backend Layer**: A Django-based REST API that provides data services, business logic, and database interactions
- **Database Layer**: Persistent storage for all application data

This architecture was chosen for several reasons:

- **Scalability**: The separation of concerns allows independent scaling of frontend and backend components
- **Development Efficiency**: Different team members could work on frontend and backend components simultaneously
- **Maintainability**: Changes to one layer have minimal impact on other layers when interfaces remain stable
- **Modern Web Standards**: Alignment with industry best practices for web application development

5.1.2 Backend Architecture

Our backend follows Django's recommended project structure with significant customizations to better support our application's requirements:

- **API-First Design**: All functionality is exposed through a comprehensive REST API
- **Model-Driven Development**: Database schema and relationships defined through Django's ORM models
- **Domain-Driven Organization**: Functionality organized into logical modules based on domain responsibilities
- **File Separation by Function Type**: Rather than large monolithic files, we separated models, views, serializers, and tests into dedicated directories with domain-specific files

This organization is evident in our project structure:

- `models_files/`: Contains domain-specific model definitions (`user_models.py`, `society_models.py`, `event_models.py`, etc.)
- `views_files/`: Contains endpoint implementations categorized by functionality (`admin_views.py`, `president_views.py`, `event_views.py`, etc.)
- `serializers_files/`: Contains domain-specific serializers for API data transformation
- `tests/`: Contains hierarchically organized test files mirroring the application structure

The backend architecture employs several key design patterns:

- **Service Layer Pattern**: Complex business logic encapsulated in service classes (e.g., `recommendation_service.py`)
- **Repository Pattern**: Data access logic separated from business logic
- **Signals Architecture**: Django signals used for event-driven functionality (as seen in `signals.py`)
- **Resource-Oriented Design**: API endpoints organized around domain resources following RESTful principles

5.1.3 Frontend Architecture

The frontend architecture employs a component-based design using React and follows these key principles:

- **Component Hierarchy**: UI organized into reusable components with clear parent-child relationships
- **State Management**: Application state managed through React hooks and context
- **Route-Based Code Organization**: Features organized around application routes

- **Responsive Design Approach**: UI components designed to function across different viewport sizes

5.2 Key Design Decisions

5.2.1 Multi-Tiered Dashboard Design

One of the most significant design decisions was implementing a multi-tiered dashboard system to serve different user roles. This approach required careful consideration of:

- **Role-Based Access Control**: Each dashboard provides access only to appropriate functionality
- **Component Reusability**: Shared components across dashboards where functionality overlaps
- **Consistent User Experience**: Maintaining design consistency while accommodating different functionality

We considered two alternative approaches:

1. **Single Dashboard with Conditional Elements**: This would have simplified routing but created more complex components with numerous conditional rendering branches.
2. **Completely Separate Dashboards**: This would have eliminated shared code but led to significant duplication.

Our chosen approach balances these considerations by implementing role-specific dashboards while extracting common functionality into shared components. This decision improved maintainability by reducing duplicated code while preserving clear separation between role-specific functionality.

5.2.2 Data Model Design

Our data model design focused on capturing the complex relationships between users, societies, events, and administrative functions. Key design decisions included:

- **Role Flexibility**: Rather than creating separate user types, we implemented a role-based system where a single user can have multiple relationships with societies (member, president, event manager)
- **Content Approval Workflow**: Integrated approval states for societies, events, and news to support administrative oversight
- **Temporal Modeling**: Proper datetime handling for events and activities to support calendar functionality

This design supports the complex relationships inherent in university societies while maintaining data integrity and facilitating administrative oversight. The approach is evident in our `models.py` file, which defines clear relationships between entities.

5.2.3 Real-Time Notifications

To enhance user experience, we implemented real-time notifications using WebSockets through Django Channels. This design decision required:

- **Asynchronous Communication**: Integration of asynchronous processing with Django's synchronous environment
- **Channel Layer Design**: Implementation of message routing and consumer patterns as seen in the `consumer` directory
- **Event-Driven Architecture**: Using Django signals to trigger notifications based on system events

We considered a polling-based approach as an alternative, but rejected it due to increased server load and potential delays in notification delivery. The WebSocket implementation provides immediate updates to users while being more resource-efficient despite its increased implementation complexity.

5.3 Implementation Strategies

5.3.1 Authentication and Authorization

The implementation of authentication and authorization was critical for our multi-role system. Our approach included:

- **Token-Based Authentication**: Secure authentication using JWT tokens
- **University Email Verification**: OTP-based verification to ensure only university students can register
- **Permission Granularity**: Fine-grained permissions for different actions rather than broad role-based access

This implementation supports our security requirements while providing a streamlined user experience. The token-based approach facilitates authentication across our separated frontend and backend architecture.

5.3.2 Society and Event Management

The society and event management functionality required careful implementation to support various user roles:

- **Administrative Approval Flow**: Implementation of multi-step approval processes for societies, events, and content
- **Role-Specific Interfaces**: Tailored interfaces for society presidents, event managers, and regular members
- **Moderation Controls**: Tools for administrators to monitor and moderate content

The implementation uses a state machine pattern for managing content status (draft, pending, approved, rejected), providing a consistent approach to approval workflows across different content types.

5.3.3 API Design and Frontend Integration

Our API design focused on creating intuitive, resource-oriented endpoints that aligned with frontend requirements:

- **RESTful Resource Modeling**: API endpoints organized around resources with appropriate HTTP methods
- **Comprehensive Serialization**: Detailed serializers to transform complex Django models into appropriate JSON representations
- **Nested Resource Access**: Endpoints structured to minimize client-side data assembly

This design is evident in our `urls.py` and `serializers.py` files, which show a consistent pattern of resource mapping and data transformation. The approach simplified frontend development by providing properly structured data that matched component requirements.

5.4 Technical Challenges and Solutions

5.4.1 Managing Complex User Permissions

Implementing the multi-tiered permission system presented significant challenges:

- **Challenge**: Ensuring users could only access appropriate functionality while minimizing permission checks
- **Solution**: Implementation of a permission caching system and hierarchical permission structure

Our solution, visible in the views and permissions implementations, provides efficient permission checking without compromising security. We used Django's built-in permission system as a foundation but extended it with custom permission classes to handle our specific requirements.

5.4.2 Code Organization for Maintainability

As the codebase grew, maintaining organization became increasingly important:

- **Challenge**: Keeping related functionality together while preventing files from becoming unmanageably large
- **Solution**: Implementation of a modular file structure with domain-specific directories

Our backend structure shows this approach with the separation of views, serializers, and models into dedicated files and directories (e.g., `views_files`, `serializers_files`, `models_files`). This organization improved maintainability by making code locations predictable and keeping related functionality together.

5.4.3 Testing and Quality Assurance

Ensuring system quality required a comprehensive testing approach:

- **Challenge**: Testing complex interactions between components and user roles
- **Solution**: Implementation of multi-layered testing strategy including unit tests and manual testing

The `tests` directory structure shows our approach to organizing tests by functionality rather than by file structure, improving test discoverability and maintenance.

5.4.4 Code Structure and Function Length

While we generally aimed for concise, single-responsibility functions throughout our codebase, certain view methods were intentionally implemented as longer functions due to the complex nature of the operations they handle. Several patterns emerged where longer functions were justified by specific technical considerations:

- **Complex Workflow Management**: Functions handling complete business processes needed to maintain the integrity of multi-step workflows.
- **Transaction Coherence**: Operations requiring database consistency across multiple models benefited from being contained within a single method.
- **Permission and Role Management**: Functions dealing with intricate permission checking and role assignments required comprehensive logic in one place.
- **Real-time Notification Integration**: Methods that both perform database operations and trigger WebSocket notifications needed to handle both concerns to ensure synchronization.

Consider the following examples from our codebase:

Society Approval Workflow (`api.view_files.request_views.AdminSocietyRequestView.put`): This method handles society approval/rejection by an administrator. Its length is justified because it must perform authorization checks, validate and update society records, create activity log entries for audit purposes, clean up expired logs, send real-time WebSocket notifications, and construct appropriate responses. Breaking this function into smaller pieces would require passing state between functions and potentially complicate error handling.

Society Role Management (`api.view_files.president_views.SocietyRoleManagementView.patch`): This method handles the complex task of assigning or removing leadership roles within societies. Its length is necessary because it handles multiple roles (vice president and event manager) with parallel logic, requires complex verification to maintain data integrity, must update both society and student models atomically, and implements different logic paths for role removal and assignment that share context.

Joining Society Requests (`api.view_files.request_views.RequestJoinSocietyView.post`): This method processes student requests to join societies. The function length is appropriate because it must verify student existence, check society existence, validate that the student isn't already a member, check for existing pending requests, and create new requests when appropriate—all within a single transaction to maintain data consistency.

Recommendation Feedback Collection (`api.view_files.recommendation_views.RecommendationFeedbackView.post`): This method handles multiple types of recommendation feedback (ratings, relevance scores, and join indicators). Its length is justified by

the need to validate different combinations of feedback types, process each type appropriately, and maintain consistent processing of metadata across feedback types.

News Publication Approval (`api.view_files.news_views.AdminNewsApprovalView.put`): This method handles the approval or rejection of news publication requests. Its complexity stems from the need to update multiple related models (publication requests and news posts) in a single transaction, create notifications based on approval decisions, construct dynamic notification content, and maintain a complete audit trail.

News Publication Request Creation (`api.view_files.news_views.NewsPublicationRequestView.post`): This method handles creating requests to publish society news. Its length is necessary to verify user permissions (including complex checks for society management rights), validate news post existence, check for existing requests, and create new publication requests with appropriate metadata.

While we could have decomposed these methods into smaller functions, doing so would have introduced additional complexity through state management between functions. The current approach provides better readability by keeping related operations together, ensures transaction integrity by handling all database operations in a single context, and facilitates debugging by making the complete workflow visible in one place.

For future development, we would consider implementing a more systematic approach to managing complex operations, potentially through workflow patterns or service classes, while maintaining the benefits of our current approach to function organization.

5.5 Evolution of Implementation

Throughout the project, our implementation approach evolved in response to challenges and changing requirements:

- **Initial Phase (First 5 Weeks)**: Focus on core entity models, basic CRUD operations, and role-specific dashboards, establishing the foundation of the system with rapid feature development
- **Middle Phase**: Refinement of existing features and implementation of complex business logic and approval workflows
- **Final Phase**: Refinement of user experience, performance optimization, and bug fixing

This phased approach allowed us to deliver a functional system early while progressively adding sophistication. The evolution is visible in our code structure, which shows a progression from basic functionality to more complex implementations.

5.6 Reflections on Design and Implementation

In retrospect, several key decisions significantly influenced our project's outcome:

- **Positive Impact**: The separation of front-end and backend allowed parallel development and clear boundaries between concerns

- **Challenge**: The implementation of real-time features added complexity but significantly enhanced user experience
- **Learning**: An earlier focus on directory organization would have reduced late-stage adjustments.

These reflections inform our understanding of effective design and implementation strategies for future projects, particularly the importance of early architectural decisions on the overall project trajectory.

Chapter 6

Testing

This chapter discusses our approach to testing the university society management platform, the tools and processes we employed, and a critical evaluation of our testing strategy's strengths and weaknesses.

6.1 Approach and tools

Our testing strategy employed a multi-layered approach combining automated and manual testing techniques to ensure comprehensive coverage across different aspects of the application. We implemented the following testing methodologies:

6.1.1 Automated Testing

Unit Testing

Unit tests formed the foundation of our testing pyramid, allowing us to verify individual components in isolation. We used Django's built-in testing framework for backend unit tests and Vitest for frontend components.

- **Backend Unit Tests**: We developed unit tests for models, serializers, and utility functions to verify their behavior in isolation. These tests are located in the `backend/api/tests` directories, organized by domain (users, societies, events, etc.).
- **Frontend Unit Tests**: React component tests using Vitest and React Testing Library to verify component rendering and behavior. These tests are located in each directory in `frontend/src/pages`.

API Integration Testing

We implemented comprehensive API integration tests using Django REST Framework's testing tools to verify the correct functioning of our RESTful endpoints. These tests ensure that:

- Endpoints correctly handle authentication (valid tokens, invalid tokens, missing tokens)
- Data validation works as expected

- Proper HTTP status codes and response formats are returned
- Error cases are handled appropriately

These tests are located in the `backend/api/tests/views` directory and are organized by API resource.

WebSocket Testing

Given the real-time features of our application, testing WebSocket connections was crucial. We used Django Channels' testing utilities to verify:

- Correct routing of WebSocket connections
- Successful establishment of connections
- Proper message handling
- Graceful connection closure

Our WebSocket tests, like `WebSocketRoutingTests`, ensure that the real-time notification system functions correctly across different features (dashboard, society, event, comments).

Mock Testing

We used the `unittest.mock` library to simulate external dependencies and create controlled testing environments. This approach was particularly valuable for testing:

- Error handling when components fail
- Edge cases that would be difficult to reproduce naturally
- Functionality dependent on external services

An example of mock testing can be seen in the `test_get_current_user_with_serializer_error` method, which tests the application's response when serialization fails.

6.1.2 Manual Testing

While we aimed to automate as much testing as possible, we recognized that certain aspects of the application required manual verification:

- **User Interface Flow**: Complex user journeys across multiple screens and interactions that were difficult to capture in automated tests
- **Visual Design Consistency**: Ensuring UI elements appeared as designed
- **Cross-Browser Compatibility**: Verifying functionality in different browsers and environments
- **Usability Testing**: Subjective evaluation of ease of use and intuitiveness of interfaces

Manual testing was conducted according to predefined test scripts that outlined specific steps and expected outcomes. These scripts were executed before each major deployment and whenever significant UI changes were made.

6.2 Quality assurance processes

To ensure consistent testing quality, we implemented several processes throughout our development lifecycle:

6.2.1 Test-Driven Development (TDD)

For critical components, particularly in the backend, we employed TDD practices where tests were written before implementing the functionality. This approach:

- Ensured specifications were clear before development began
- Provided immediate feedback on implementation correctness
- Created a safety net for refactoring

Evidence of our TDD approach can be seen in the comprehensive test suite for core models and API endpoints, where test coverage is particularly strong.

6.2.2 Continuous Integration (CI)

We implemented a CI pipeline using GitHub Actions that automatically ran our test suite on every pull request and push to the main branch. This process:

- Prevented the introduction of failing code into the main codebase
- Provided immediate feedback to developers about test failures
- Maintained a historical record of test results

The CI configuration can be found in the `.github/workflows` directory, demonstrating our commitment to automated quality assurance.

6.2.3 Code Review Process

Our team employed a strict code review process where test coverage was a key criterion for approval:

- New features required accompanying tests
- Pull requests with failing tests were resolved as soon as possible
- Reviewers were responsible for verifying test quality and coverage

6.2.4 Test Coverage Monitoring

We used coverage.py for backend code and Vitest coverage reports for frontend code to monitor the extent of our test coverage. Coverage reports were generated as part of our CI pipeline and reviewed regularly to identify areas needing additional testing.

6.3 Evaluation of testing

6.3.1 Strengths

Our testing approach demonstrated several notable strengths:

- **Comprehensive API Testing**: Our API integration tests provided strong confidence in the correctness of backend functionality, with all key endpoints covered by tests verifying authentication, validation, and error handling.
- **Real-Time Feature Testing**: The WebSocket testing ensured our real-time notification system functioned correctly, a critical aspect of the user experience.
- **Structured Test Organization**: Our domain-driven test organization mirrored the application structure, making it easy to locate and maintain tests as the codebase evolved.
- **Strong Authentication Testing**: Authentication mechanisms were thoroughly tested across endpoints, ensuring security was not compromised.

6.3.2 Weaknesses and Limitations

Despite our efforts, several weaknesses remained in our testing approach:

- **Incomplete Frontend Test Coverage**: While critical components had unit tests, overall frontend test coverage remained lower than backend coverage. Frontend testing focused primarily on component rendering rather than user interactions.
- **Limited End-to-End Testing**: We lacked comprehensive end-to-end tests that would verify complete user journeys across the frontend and backend. This gap was partially addressed through manual testing, but automated E2E tests would have provided more confidence in system integration.
- **Load and Performance Testing**: We did not implement systematic load testing to verify system performance under high concurrent usage, a potential concern for a university-wide platform.

6.3.3 Coverage Results

Our test coverage results demonstrate the effectiveness of our testing strategy while highlighting areas for improvement:

- **Backend Componenets**: 89% coverage
- **Frontend Components**: 80% branch coverage

6.3.4 Balance of Manual and Automated Testing

Our testing strategy balanced automated and manual approaches based on their respective strengths and limitations:

- **Automated Testing Strengths**: Consistent, repeatable verification of functionality; ability to test at scale; early detection of regressions; documentation of expected behavior.

- **Automated Testing Limitations**: Difficulty capturing subjective quality aspects; complexity in testing rich UI interactions; maintenance overhead as the application evolves.
- **Manual Testing Strengths**: Ability to evaluate subjective aspects like usability; flexibility to explore unexpected paths; validation of visual design; discovery of issues that automated tests might miss.
- **Manual Testing Limitations**: Time-consuming; inconsistent execution; limited coverage; dependent on tester attention and thoroughness.

We relied on manual testing primarily for aspects difficult to automate effectively—subjective quality assessment, complex UI flows, and visual consistency. Ideally, we would have automated more of our UI testing, particularly for core user journeys, but time constraints limited our ability to develop these more complex automated tests.

6.3.5 Future Improvements

Based on our experience, several improvements could enhance our testing approach in future projects:

- Implementation of Cypress or similar tools for end-to-end testing of critical user journeys
- Adoption of visual regression testing to automatically detect UI inconsistencies
- Implementation of systematic load and performance testing for critical endpoints

These improvements would address the primary gaps in our current testing approach while building on the solid foundation established in this project.