Dissertation/Report submitted to the

UNIVERSITY OF MOHAMED BOUDIAF - M'SILA, ALGERIA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

in partial fulfillment of the requirements for the degree of

Master's in Computer Science

Specialization: Computer Science

Option: ISSE

Presented by

Barkat Abdelmoumin

Entitled

Intelligent Library Management System.

Under the supervision of

Khadidja DERDOUR

Jury Members

Dr. SAOUDI Lalia	University of M'sila	President
Dr. Khadidja DERDOUR	University of M'sila	Supervisor
Dr. LOUCIF Hemza	University of M'sila	Examiner

June, 2026

الملخص

يعرض هذا التقرير تطوير نظام ذكي لإدارة المكتبات يهدف إلى تحديث خدمات المكتبات التقليدية من خلال دمج تقنيات الذكاء الاصطناعي وحلول الويب الحديثة. يبرز المشروع أهمية الأنظمة الذكية في تحسين تنظيم الموارد وإمكانية الوصول إلى المعلومات وتعزيز تفاعل المستخدم داخل البيئات الأكاديمية للمكتبات. كما يعالج المشروع عدة مشاكل موجودة في أنظمة المكتبات الحالية، مثل الاعتماد على العمليات اليدوية، ومحدودية الخدمات الذكية، وضعف أنظمة البحث التقليدية.

تجمع المنصة المقترحة بين الوظائف التقليدية لإدارة المكتبات وميزات مدعومة بالذكاء الاصطناعي مثل البحث الدلالي عن الكتب، والتوصيات الشخصية، مساعد محادثة، واقتراح التصنيفات بشكل آلي. تم تطوير النظام باستخدام تقنيات حديثة مثل Next.js وTypeScript وPostgreSQL وOllama، مع الاعتماد على نموذج bge-m3 لتوليد المتجهات الدلالية وتحسين البحث والتوصيات.

يهدف هذا النظام إلى توفير بيئة مكتبية أكثر كفاءة ومرونة وتركيزاً على المستخدم، مع إبراز الإمكانيات العملية لدمج تقنيات الذكاء الاصطناعي داخل أنظمة المعلومات الأكاديمية الحديثة.

الكلمات المفتاحية: نظام ذكي لإدارة المكتبات، البحث الدلالي، نظام التوصيات، الذكاء الاصطناعي، تطبيق ويب.

Abstract

This report presents the development of an intelligent library management system aimed at modernizing traditional library services through the integration of artificial intelligence technologies and modern web solutions. The project highlights the importance of intelligent systems in improving resource organization, information accessibility, and user interaction within academic library environments. It also addresses several limitations found in current library systems, including dependence on manual operations, limited intelligent services, and inefficient search mechanisms.

The proposed platform combines traditional library management functionalities with AI-powered features such as semantic book search, personalized recommendations, chatbot, and automatic category suggestion. The system was developed using modern technologies including Next.js, TypeScript, PostgreSQL, and Ollama, while the `bge-m3` model was used for semantic embedding generation and recommendation functionalities.

This intelligent library management system aims to provide a more adaptive, efficient, and user-oriented library environment while demonstrating the practical integration of artificial intelligence technologies into modern academic information systems.

Keywords: Intelligent library management system, semantic search, recommendation sys-

tem, artificial intelligence, web application.

Contents

List of Figures

General Introduction

Technological advancements and the rapid growth of digital technologies have significantly transformed the way information is created, stored, and accessed. In academic environments, libraries remain one of the most important sources of knowledge and learning support. However, many traditional library systems still rely on manual processes and conventional management methods that limit efficiency, accessibility, and the quality of services provided to users. These limitations have encouraged the adoption of intelligent technologies capable of improving library operations and enhancing user interaction.

The integration of artificial intelligence, semantic search, recommendation systems, and digital management tools has opened new possibilities for modern library environments. Intelligent library systems can automate repetitive operations, improve access to information resources, provide personalized recommendations, and support better decision-making through data analysis. Such systems aim to create a more adaptive, efficient, and user-oriented library experience for both students and administrators.

To address these challenges, this project proposes the development of an intelligent library management system that combines traditional library management functionalities with AI-powered services. The system provides features such as semantic book search, personalized recommendations, borrowing and reservation management, digital resource access, notification services, and administrative monitoring tools. In addition, the platform integrates modern web technologies and machine learning techniques to improve the organization and accessibility of library resources while simplifying library operations.

This report is divided into three main chapters to present the different aspects of the project and provide a comprehensive understanding of the proposed system:

Chapter 1: Preliminary Study

This chapter introduces the concept of intelligent library systems, their evolution, technologies, and importance. It also reviews existing library management solutions, discusses the limitations of current systems, and presents the proposed solution together with the main implementation challenges.

Chapter 2: Analysis and Design

This chapter focuses on the analysis and design of the proposed intelligent library management system. It presents the functional and non-functional requirements, describes the

system architecture and module decomposition, and explains the design of the semantic search and recommendation components. UML diagrams are also used to model the system structure and workflows.

Chapter 3: Implementation

This chapter presents the practical implementation of the system, including the development environment, technologies, frameworks, and AI tools used. It also explains the implementation of the embedding generation pipeline, semantic search API, recommendation system, and category suggestion API, chatbot, in addition to presenting the administrator and student interfaces of the platform.

Chapter 1

Chapter 1: Preliminary Study

1.1 Introduction

This chapter presents a preliminary study of intelligent library management systems in order to understand the topic and its general workflow. It introduces the concept of intelligent library systems and explains their importance in improving the organization of library resources, the speed of access to information, and the overall quality of services provided to users. It also reviews existing solutions related to the project context, especially selected current systems, and shows how such systems are used in real library environments.

In addition, the chapter examines the major problems found in current library systems, such as weak integration, limited usability, and dependence on manual work in some operations. Based on these observations, the chapter presents the proposed solution and highlights how an intelligent library management system can help overcome these limitations, thereby providing a clear conceptual understanding of the domain.

1.2 Concept of Library Systems

A library system is an organized framework used to manage library resources, services, and daily operations within a library environment. It provides structured methods for handling activities such as cataloguing, classification, circulation management, user registration, resource tracking, and information retrieval. Library systems are designed to facilitate the organization and accessibility of information resources while supporting both librarians and users in accessing and managing library collections efficiently.

Modern library systems may operate in manual, computerized, digital, or intelligent forms depending on the technologies and management methods adopted by the institution. Their primary objective is to improve the efficiency, reliability, and quality of library services while ensuring effective management of information resources [1], [2].

1.3 Evolution of Library Systems

Library systems have continuously developed to improve the organization, storage, and accessibility of information within libraries. Over time, technological progress and changing user needs have influenced the way library services are managed and delivered, leading to significant transformations in library environments and operations.

1.3.1 Traditional Libraries

Traditional libraries represented the earliest form of organized information management. Their operations were based mainly on paper catalogues, handwritten records, and physical organization of books and documents. Librarians managed activities such as classification, borrowing, returning, and shelf arrangement manually. Despite the simplicity of this model, traditional libraries played an important role in preserving knowledge and providing structured access to information within educational and cultural institutions.

1.3.2 Computerized Library Systems

The development of computer technologies introduced a new stage in library management through computerized systems. Libraries began using databases and software applications to organize catalogues, manage circulation processes, and store bibliographic information electronically. Early implementations were desktop-based systems operating on individual computers, while later systems adopted centralized databases accessible by multiple users and departments. This transformation improved the speed, organization, and accessibility of library operations [1], [2].

1.3.3 Digital Libraries

Digital libraries extended library services beyond physical spaces by providing electronic resources and online access to information. This stage introduced digital collections such as e-books, electronic journals, and multimedia resources accessible through internet-based platforms. Digital libraries supported remote learning, online research, and continuous access to information regardless of geographical location, making library services more flexible and widely available [3].

1.3.4 Smart and Intelligent Libraries

An intelligent library system is a library environment that integrates artificial intelligence technologies, including machine learning, expert systems, natural language processing, and

robotic automation, to automate and optimize core library functions such as cataloguing, circulation, reference services, and resource discovery. Unlike traditional library management systems that rely on static, rule-based processes, an intelligent system adapts dynamically to user behavior and institutional needs [4].

An intelligent library system can also be understood as a system that applies artificial intelligence technologies to deliver knowledge-based services to both library users and staff. These systems support decision-making, improve access to information, and enhance user interaction by simulating aspects of human intelligence within a library environment [1].

1.4 Technologies Used in Smart and Intelligent Libraries

Smart and intelligent libraries rely on modern technologies that improve automation, information access, resource management, and user interaction. These technologies help libraries provide faster, more adaptive, and more efficient services within digital environments.

1.4.1 Artificial Intelligence

Artificial intelligence refers to the ability of computer systems to simulate human intelligence in performing tasks such as learning, reasoning, and decision-making. In intelligent libraries, AI is used to automate operations, improve information retrieval, and support personalized services for users [4], [5].

1.4.2 RFID Technology

Radio Frequency Identification (RFID) is a technology used for automatic identification and tracking of library resources through radio signals. RFID systems simplify borrowing and returning operations, improve inventory management, and enhance the speed and accuracy of resource tracking within libraries [6].

1.4.3 Cloud Computing

Cloud computing enables libraries to store and manage digital resources through online platforms and remote servers. This technology supports remote access to library services, improves scalability, reduces infrastructure costs, and facilitates the sharing of information across different systems and institutions [3].

1.4.4 Data Analytics and Recommendation Systems

Data analytics and recommendation systems help intelligent libraries analyze user behavior, borrowing patterns, and resource usage. These technologies allow the system to generate

personalized recommendations, improve decision-making processes, and provide services that better match user interests and needs [5], [7], [8].

1.5 Importance of Intelligent Library Systems

The importance of intelligent library systems can be understood through the practical advantages they provide in modern library environments. These systems improve operational efficiency, information accessibility, and the quality of services offered to users. Their importance includes:

- **Improved operational efficiency:** Intelligent systems automate repetitive tasks such as cataloguing, circulation, and resource tracking, helping libraries perform operations more quickly and accurately while reducing manual workload [2].
- **Better access to information:** Advanced search tools and intelligent organization methods allow users to locate resources more easily and improve the accessibility of library services [5].
- **Personalized user services:** Intelligent technologies can analyze user interests and borrowing behavior to provide personalized recommendations and adaptive services that improve user satisfaction [9].
- **Support for decision-making:** Data analysis tools help librarians monitor resource usage, understand user needs, and make more effective decisions regarding collection management and service development [7].

1.6 Current Systems

Studying existing library systems helps identify the main functionalities used in library automation and provides useful references for the development of the proposed solution. In this project, the comparison focuses on two systems relevant to the Algerian and Francophone library environment: SINJAB/SYNGEB and PMB [10], [11], [12], [13].

1.6.1 SINJAB/SYNGEB

SINJAB, also known as SYNGEB, is an integrated library management system developed by CERIST in Algeria in 1990 [10], [11]. It was designed for academic and research libraries and became widely used in Algerian universities. The system supports Arabic and French and follows the UNIMARC cataloguing standard.

SINJAB provides modules for cataloguing, acquisitions, serials management, and circulation control, allowing libraries to manage bibliographic records and borrowing operations within a centralized environment, as shown in Figure (1.1) presenting the SYNGEB interface showing borrower and book-loan management information.

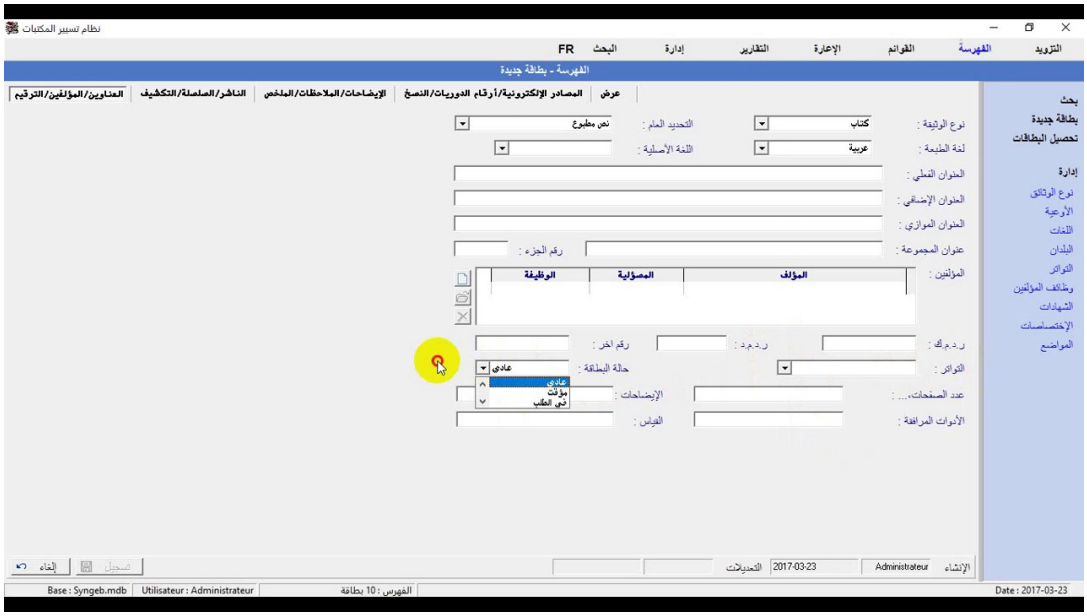

Figure 1.1: SYNGEB interface.

1.6.2 PMB (PhpMyBibli)

PMB is a web-based open-source integrated library system developed in France in 2002 by François Lemarchand [12], [13]. Built using PHP and MySQL, PMB offers a flexible and customizable platform that became widely adopted in many French-speaking libraries, as shown in Figure (1.2) presenting the PMB interface with circulation details and borrowed-book information.

The system supports UNIMARC and standards such as Z39.50, while providing modules for cataloguing, circulation, acquisitions, and OPAC access through a web interface.

1.6.3 Online Catalogue Search Interface

The Faculty of Mathematics and Informatics Library at the University of M'sila also provides a web-based catalogue search interface that allows students and users to search for bibliographic resources online. The system enables searches using criteria such as title, author, and subject through a simple web interface accessible from a browser.

The interface shown in Figure (1.3) shows the online catalogue used for accessing available library resources and improving the accessibility of bibliographic information within the university library environment.

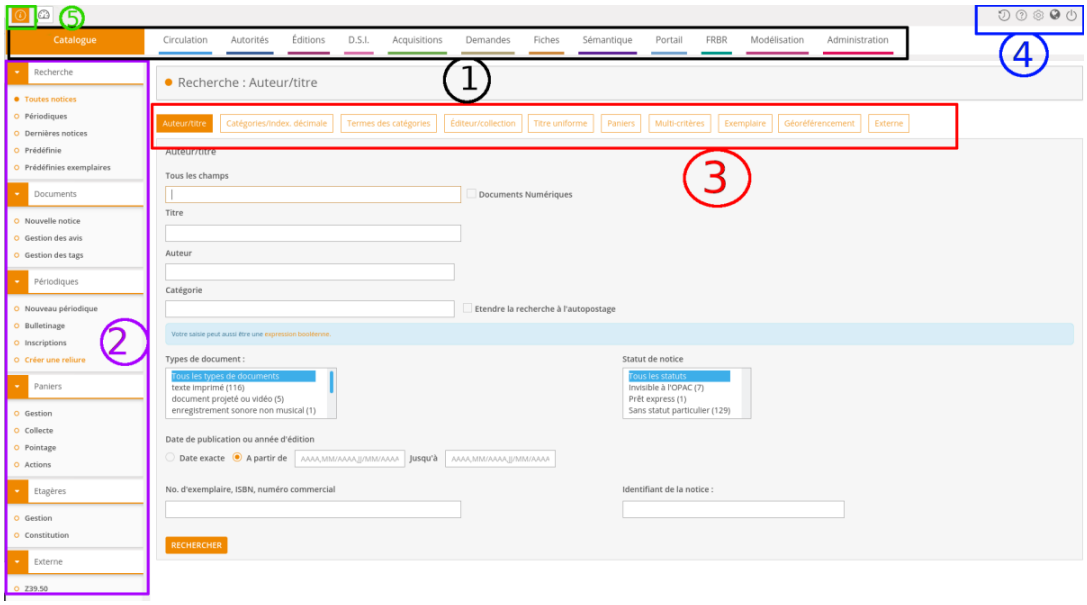

Figure 1.2: PMB interface.

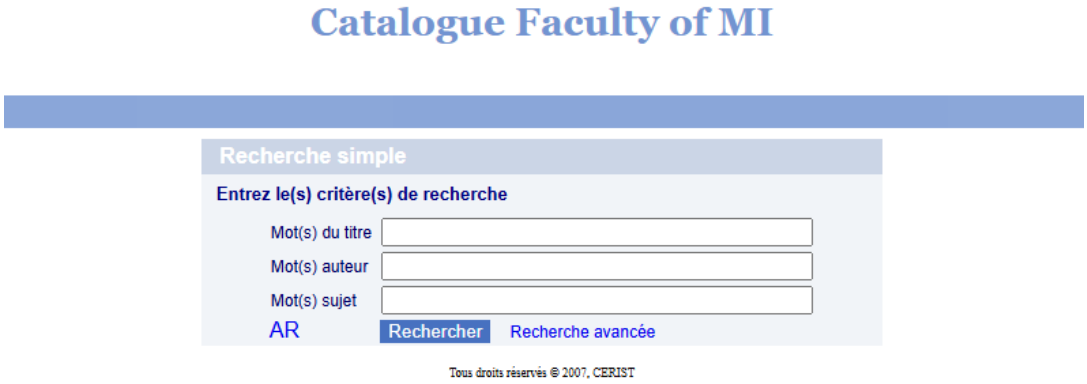

Figure 1.3: Web-based library catalogue.

Both SINJAB and PMB provide important functionalities for library automation, including cataloguing, circulation management, and bibliographic organization. They also con-

tribute to improving access to library resources within academic environments. However, these systems mainly focus on traditional management operations and provide limited intelligent functionalities such as personalized recommendations, adaptive interaction, and AI-assisted services. Similarly, basic online catalogue interfaces improve access to bibliographic information but remain focused primarily on search functionality. These observations highlight the need for more intelligent and integrated library systems capable of providing modern and adaptive services for both users and administrators.

1.7 Problems in Current Library Systems

Despite the development of library automation, many current systems still face limitations that reduce their efficiency and affect the quality of services provided to users. The main problems include:

- **Dependence on manual operations:** Many library activities still require manual work, including data entry, classification, and resource management, which increases workload and the possibility of human error.
- **Limited intelligent services:** Traditional systems often lack intelligent features such as smart search, recommendation systems, and automated assistance, limiting the quality of user interaction and information retrieval.
- **Weak user interaction and adaptability:** Current systems generally provide static services without adapting to user behavior or supporting modern interaction methods such as chatbots and mobile access.
- **Insufficient data analysis:** Many library systems do not effectively analyze user activity and resource usage, reducing the ability of librarians to make data-driven decisions and improve services efficiently.

1.8 Challenges of Intelligent Library Systems

Although intelligent library systems provide many advantages, their implementation and management also introduce several important challenges. These challenges include:

- **High implementation costs:** Intelligent technologies such as AI, RFID, and cloud services may require significant investment in infrastructure, software, and maintenance.
- **Data security and privacy issues:** Intelligent systems process large amounts of user data, creating concerns related to information security, privacy protection, and secure access management.

- **Technical complexity and training:** The use of advanced technologies requires technical expertise and continuous staff training to ensure effective system management.
- **Integration with existing systems:** Many libraries still rely on older infrastructures, making the integration of modern intelligent technologies more difficult and sometimes causing compatibility issues.

1.9 Proposed Solution

The proposed solution of this project is the development of an intelligent library management system designed to modernize traditional library operations and improve the overall accessibility, organization, and management of library resources. The system aims to provide a centralized and user-friendly environment that facilitates the management of books, users, borrowing transactions, reservations, digital resources, and administrative activities within academic libraries.

Unlike conventional library systems that mainly focus on basic cataloguing and circulation operations, the proposed platform integrates artificial intelligence technologies to provide more adaptive and intelligent services. The system incorporates semantic search capabilities that allow users to search for books using natural language queries instead of relying only on exact keyword matching. This approach improves information retrieval by identifying semantically related resources even when different terms or expressions are used.

The proposed solution also includes a personalized recommendation system that analyses user interests, borrowing history, and interactions with library resources in order to suggest relevant books and materials. In addition, the platform provides AI-assisted category suggestion mechanisms to help librarians organize books more efficiently and maintain consistent catalogue classification.

From an administrative perspective, the system supports efficient management of library activities through dedicated interfaces for monitoring borrowing operations, reservations, overdue books, user accounts, and system activity logs. It also supports digital resources by allowing users to access electronic books and documents directly through the platform.

The system is designed using a modern web-based architecture that ensures scalability, maintainability, and accessibility across different devices. By combining traditional library management functionalities with intelligent technologies such as semantic embeddings, recommendation mechanisms, and automated assistance, the proposed solution aims to provide a smarter, more efficient, and more user-oriented library environment capable of meeting the evolving needs of modern academic institutions.

1.10 Conclusion

This chapter presented a general study of intelligent library management systems by introducing the main concepts, technologies, and existing solutions used in modern library environments. It also highlighted several limitations found in current systems and discussed the importance of integrating intelligent features to improve library services and user interaction.

Based on this study, the proposed intelligent library management system was introduced as a solution that aims to provide more efficient, adaptive, and user-oriented library services. The next chapter will focus on the analysis and design phase of the proposed system by presenting its requirements, architecture, intelligent components, and UML models.

Chapter 2

Chapter 2: Analysis and Design

2.1 Introduction

Building on the ideas and challenges discussed in the previous chapter, this chapter focuses on the analysis and design of the proposed intelligent library management system. It identifies the main functional and quality requirements, introduces the general architecture of the platform, and explains the design of its intelligent features, including the NLP and recommendation modules. In addition, UML diagrams are used to model the system's behaviour and structure.

2.2 Requirements Specification

This part presents the essential requirements of the proposed intelligent library management system. It describes the expected system features alongside the performance and usability constraints that guide its development, grouping them into functional and non-functional requirements for better organisation and clarity.

2.2.1 Functional Requirements

Functional requirements describe the essential features and operations that the intelligent library management system must provide to achieve the objectives of the project. These requirements define the services, behaviours, and tasks that the system should perform to support users and manage library resources efficiently. The system includes the following functionalities:

- **User account management:** The system allows administrators, librarians, and members to create accounts, log in securely, and manage their profiles.
- **Document management:** Librarians can add, update, delete, and organize books and digital resources using bibliographic information.

- **Borrowing and return management:** The system manages borrowing transactions, due dates, returns, reservations, and overdue fines.
- **Search and discovery:** Users can search for documents using natural language queries, where the system analyses the meaning of the query to retrieve the most relevant results.
- **Recommendation system:** The system provides personalised book recommendations by analysing user interests, borrowing history, and similarities between documents.
- **Reporting and analytics:** Administrators can generate reports about borrowing activity, overdue documents, and library usage statistics.
- **Notification management:** The system sends reminders and alerts for due dates, reservations, and important library announcements.
- **Digital resource access:** Members can access electronic materials such as PDFs, ebooks, and scanned documents directly through the platform.

2.2.2 Non-Functional Requirements

Non-functional requirements define the quality attributes and operational constraints of the system. They describe how efficiently, securely, and reliably the system should perform its functions within the intended environment.

- **Performance:** The system should provide fast response times for operations such as search, borrowing, and resource management, while supporting multiple concurrent users without significant delays.
- **Security:** User information and system data must be protected through authentication, authorization, and secure communication mechanisms to prevent unauthorized access.
- **Usability:** The system should offer a simple, intuitive, and responsive interface that allows users to perform operations easily across different devices and screen sizes.
- **Maintainability:** The system should be designed using modular and well-organized components to simplify maintenance, updates, and future extensions.
- **Scalability:** The architecture should support the growth of users, resources, and transactions without affecting overall system performance.
- **Reliability and Availability:** The system should operate consistently with minimal downtime while ensuring data integrity and reliable execution of critical operations.

- **Portability:** The system should be deployable across different server environments and operating platforms without requiring major modifications.

2.3 System Architecture

2.3.1 Overview

The proposed intelligent library management system is designed around a layered client-server architecture. This architectural choice promotes separation of concerns, improves maintainability, and allows each component of the system to evolve independently. The architecture consists of three primary layers: the presentation layer, the application layer, and the data layer. An additional intelligent subsystem layer is integrated within the application layer to handle natural language processing and recommendation tasks.

The overall design follows a modular approach in which each functional area of the system, such as user management, circulation, search, and recommendations, is encapsulated in a dedicated module. These modules communicate through well-defined interfaces, which simplifies testing, debugging, and future extension.

2.3.2 Architectural Layers

The intelligent library management system follows a three-layer architecture composed of the presentation layer, application layer, and data layer. This architecture improves modularity, maintainability, and scalability by separating the user interface, business logic, and data management responsibilities.

Presentation Layer

The presentation layer provides the graphical user interface through which users interact with the system. It is responsible for displaying information, handling user input, and ensuring responsive access across different devices. The interface supports separate views for students and administrators, with role-based access control used to restrict access to authorised features and operations.

Application Layer

The application layer contains the core business logic of the system and manages communication between the presentation layer and the data layer. It handles operations related to user management, catalogue management, borrowing processes, notifications, and report generation.

This layer also includes the intelligent subsystem responsible for semantic search and personalised recommendations. The subsystem analyses user queries, borrowing history, and reading interests to retrieve relevant books and generate recommendation results.

Data Layer

The data layer manages the storage and retrieval of system data, including users, books, borrowing records, reservations, notifications, and other library information. It ensures data consistency, integrity, and secure access through the database management system.

The data layer also supports the storage of digital resources such as book cover images and electronic documents while maintaining secure authentication and user identity management.

2.3.3 Module Decomposition

The intelligent library management system is divided into several functional modules Figure (2.1)., where each module performs a specific task within the platform.

- **Authentication Module:** Handles user registration, login, and role-based access control.
- **User Management Module:** Manages user accounts, profiles, permissions, and account status.
- **Catalogue Module:** Allows librarians to add, edit, organize, and manage library resources.
- **Circulation Module:** Manages borrowing, returns, reservations, and overdue fines.
- **Search Module:** Processes natural language queries by converting them into vector representations and comparing them with stored book vectors using cosine similarity to retrieve semantically relevant results.
- **Recommendation Module:** Generates personalized recommendations by analysing user borrowing history and interests through profile vectors and semantic similarity calculations.
- **Notification Module:** Sends reminders, alerts, and announcements to users.
- **Reporting Module:** Generates reports and statistics related to library activities.
- **Administration Module:** Provides tools for administrators to monitor and manage the entire system.

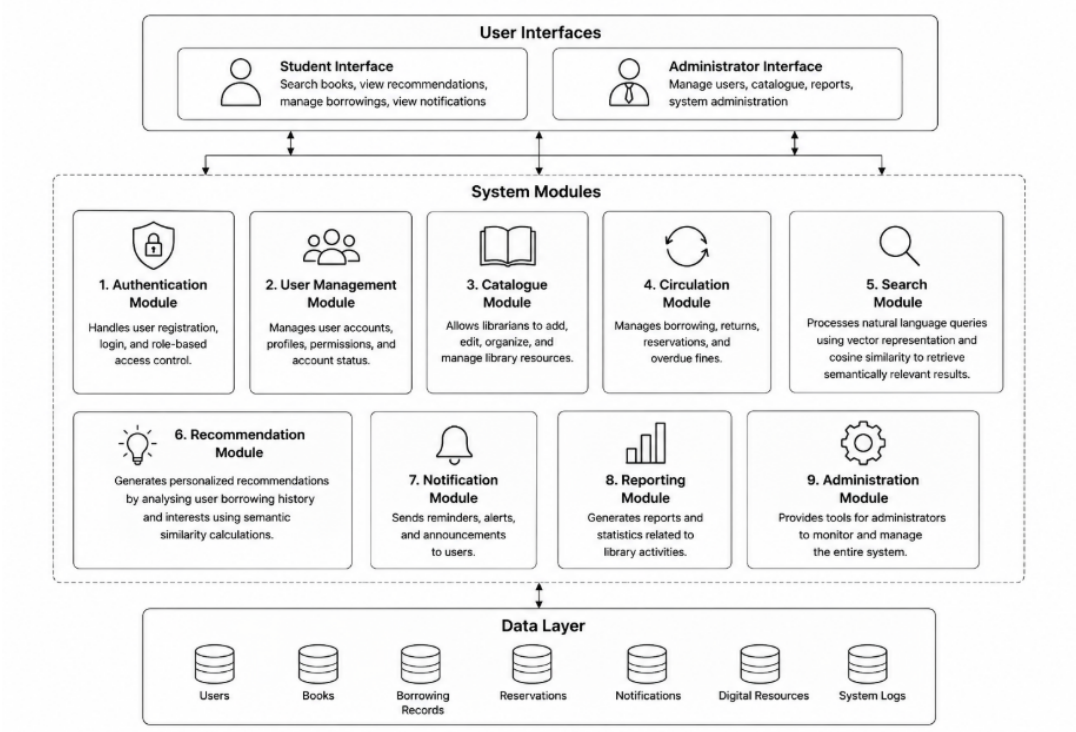

Figure 2.1: Module decomposition diagram.

2.4 NLP and Recommendation System Design

One of the defining characteristics of the proposed system is its integration of a vector-based search engine and a recommendation engine that share the same underlying workflow. Both components operate on the same core principle: converting inputs into numerical vectors, comparing those vectors against the pre-computed book vectors stored in the database, and ranking the results by similarity score before applying a threshold filter to return only the most relevant output. This unified design simplifies the architecture and ensures that the search and recommendation features remain consistent and maintainable.

2.4.1 Semantic Search Concept

Traditional library search systems rely on exact keyword matching, meaning that a user searching for ``computer networks'' will only retrieve documents whose metadata contains those exact terms. Relevant documents that use different wording or related expressions may therefore be excluded from the results. This limitation reduces the effectiveness of conventional information retrieval systems.

The proposed system replaces keyword matching with a vector-based semantic search approach. During the cataloguing phase, each document's title, author names, subject classification, and description are combined and transformed into a numerical embedding using

a word embedding model. These embeddings are pre-computed and stored alongside document records in the database to enable efficient retrieval at query time.

When a user submits a query, the same embedding process is applied to the query text. The resulting representation is compared with stored document embeddings using cosine similarity, which measures semantic closeness within the shared vector space. Documents are then ranked according to similarity scores, and a configurable threshold filters out low-relevance matches before the final results are returned.

This approach enables the system to recognise synonyms, conceptually related terms, and natural language expressions that traditional keyword systems cannot capture. For example, a query such as "borrowing books" may successfully retrieve documents related to "library circulation management" because the embedding model identifies their semantic relationship.

2.4.2 Search and Recommendation Workflow

The search and recommendation components of the system share a common processing workflow. The only difference between the two is the nature of the user input: for search, the input is the query text typed by the user; for recommendations, the input is derived from the user's borrowing history and stated interests. Both inputs are converted into a vector representation that is then processed through identical subsequent stages.

The complete workflow is illustrated in Figure (2.2).

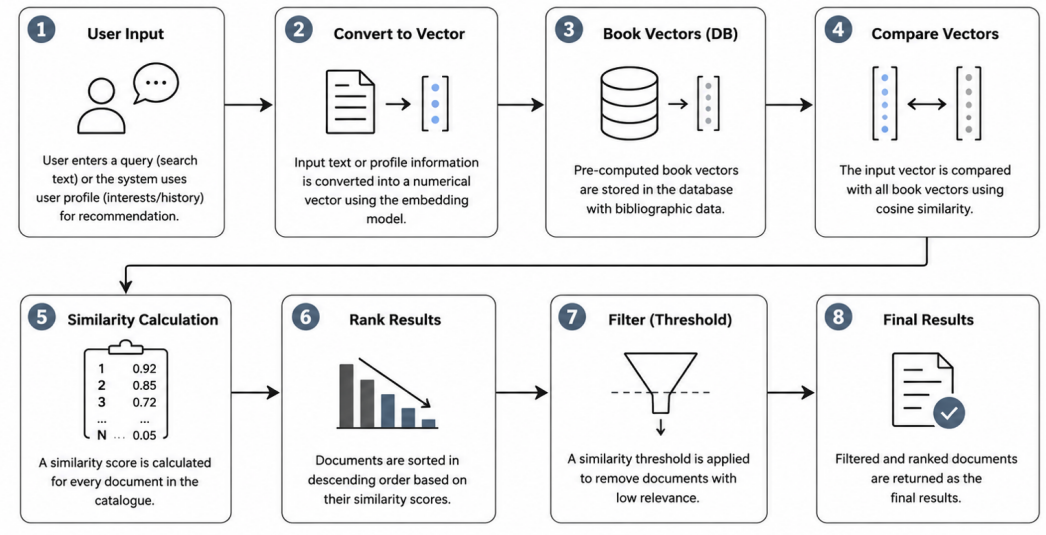

Figure 2.2: Search and recommendation workflow.

The workflow proceeds through the following eight stages:

Stage 1: User Input

The entry point of the workflow differs depending on the module being used.

For the **search** function, the user input is a query text string entered by the member in the search interface. This can be a natural language sentence, a title fragment, an author name, or any subject-related phrase.

For the **recommendation** function, the user input is not a text string but rather a profile vector derived from the member's interaction history. This profile is constructed by aggregating the vectors of all documents the member has previously borrowed or rated, weighted by recency and rating score. The more documents a member has interacted with, the more accurate and specific their profile vector becomes.

This dual-input design means that a newly registered member with no borrowing history can still receive recommendations based on subject preferences declared after registration, which are used to construct an initial profile vector.

Stage 2: Convert to Vector

Once the user input is captured, it is converted into a numerical vector using the same word embedding model that was used to pre-compute the book vectors. For search queries, the raw text is first lightly preprocessed lowercased, stripped of punctuation, and tokenised before being passed through the embedding model to produce a single fixed-length dense vector. For recommendation inputs, the profile vector is computed as a weighted average of the previously stored book vectors corresponding to the member's interaction history, so no additional text processing is required.

The result of this stage is a single query vector or profile vector that occupies the same semantic space as the pre-computed book vectors, making direct comparison possible.

Stage 3: Book Vectors in the Database

All book vectors are pre-computed offline when a document is added to the catalogue and stored as serialised numerical arrays in the database alongside the standard bibliographic metadata. This offline pre-computation strategy means that the database is always ready to respond to similarity queries without needing to perform any real-time text processing on the document side. The only computation performed at query time is on the user input side (Stage 2), which is a single vectorisation operation and therefore fast regardless of the size of the catalogue.

Stage 4: Compare Vectors

Once the query vector or profile vector is available, it is compared against every book vector stored in the database. The comparison is performed using the cosine similarity metric, which measures the cosine of the angle between two vectors in the high-dimensional embedding space. A cosine similarity value of 1.0 indicates that the two vectors are perfectly aligned

(maximum relevance), while a value of 0.0 indicates complete orthogonality (no semantic relationship). Negative values are not possible in this context because the embedding model produces non-negative vector components.

The comparison stage produces a raw similarity score for every document in the catalogue relative to the current query or user profile.

Stage 5: Similarity Calculation

The similarity scores computed in the previous stage are collected and organised into a scored list pairing each document identifier with its corresponding cosine similarity value. This list represents the complete relevance ranking of the entire catalogue with respect to the current input. At this stage, no filtering or truncation has been applied; every document in the database has a score, even those with very low similarity values.

Mathematically, if the query or user profile vector is represented as $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ and a book vector is represented as $\mathbf{b} = (b_1, b_2, \ldots, b_n)$, the cosine similarity is calculated as follows [14]:

$$\text{cosineSimilarity}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \, \|\mathbf{b}\|} = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2} \times \sqrt{\sum_{i=1}^{n} b_i^2}}$$

In this formula, $\sum_{i=1}^{n} a_i b_i$ represents the dot product of the two vectors, while $\sqrt{\sum_{i=1}^{n} a_i^2}$ and $\sqrt{\sum_{i=1}^{n} b_i^2}$ represent their magnitudes. A higher cosine similarity value means that the book is more semantically related to the search query or user profile.

Stage 6: Rank Results

The scored document list is sorted in descending order of similarity score. Documents with the highest cosine similarity to the query or user profile are placed at the top of the ranked list. This ordering ensures that the most semantically relevant documents are presented first when the final results are displayed to the user.

Stage 7: Filter by Threshold

A configurable similarity threshold is applied to the ranked list to remove documents whose relevance scores fall below the minimum acceptable level. This filtering step is important because without it, every document in the catalogue would be returned for every query, including documents with near-zero similarity that are clearly irrelevant. The threshold value can be adjusted by the system administrator based on the desired balance between recall (returning more results) and precision (returning only highly relevant results). Documents that pass the threshold filter are retained; those that do not are discarded.

Stage 8: Final Results

The filtered and ranked list of documents that survive the threshold is the final output of the workflow. For the search function, this list is returned to the presentation layer and displayed to the user as an ordered list of matching documents with their titles, authors, and availability status. For the recommendation function, this list is presented to the member as a personalised reading list of suggested titles that they have not yet borrowed.

2.4.3 Offline Vectorisation of the Document Catalogue

An important design decision in the proposed system is that book vectors are computed offline rather than at query time. When a librarian adds a new document to the catalogue, the system automatically triggers a background vectorisation job that processes the document's metadata and stores the resulting vector in the database. This means that by the time a user submits a search query, all document vectors are already available and the system only needs to perform a single vectorisation operation on the query itself before running the comparison. This design choice ensures that search response times remain fast regardless of the size of the document catalogue, since the computational cost of vectorising thousands of documents is distributed across cataloguing events rather than concentrated at query time.

2.5 UML Diagrams

Unified Modelling Language (UML) is a standardised visual language used in software engineering to describe the structure and behaviour of a system. In this section, three types of UML diagrams are used to model the proposed intelligent library management system from complementary perspectives: the use case diagram models actor interactions, the sequence diagrams illustrate the dynamic flow of key operations, and the class diagram represents the static structural design of the system, as summarised in Figure 2.3 [15].

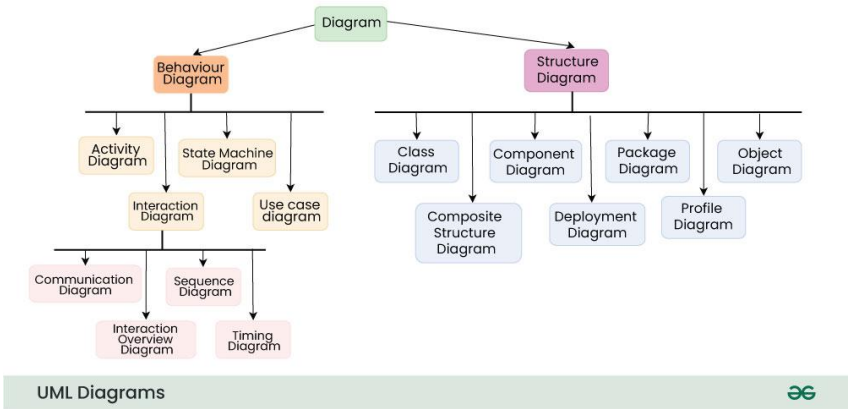

Figure 2.3: Overview of UML diagram types.

2.5.1 Use Case Diagram

The use case diagram represents the interactions between users and the intelligent library management system Figure (2.4). The system involves two primary actors: the **Student** and the **Administrator**. Each actor interacts with the system according to their responsibilities and privileges.

Student is the main user of the system and can perform the following actions:

- Log out from the system.
- Update profile information.
- Add reading interests.
- Search for books.
- View book availability.
- Borrow books.
- Reserve books.
- View borrowing history.
- Receive book recommendations.
- View popular and trending books.
- Receive notifications.

Administrator is responsible for managing and maintaining the library system. The administrator can perform the following actions:

- Manage books.
- Manage student accounts.
- Monitor borrowing and return operations.
- Handle reservations.
- Supervise system activities.

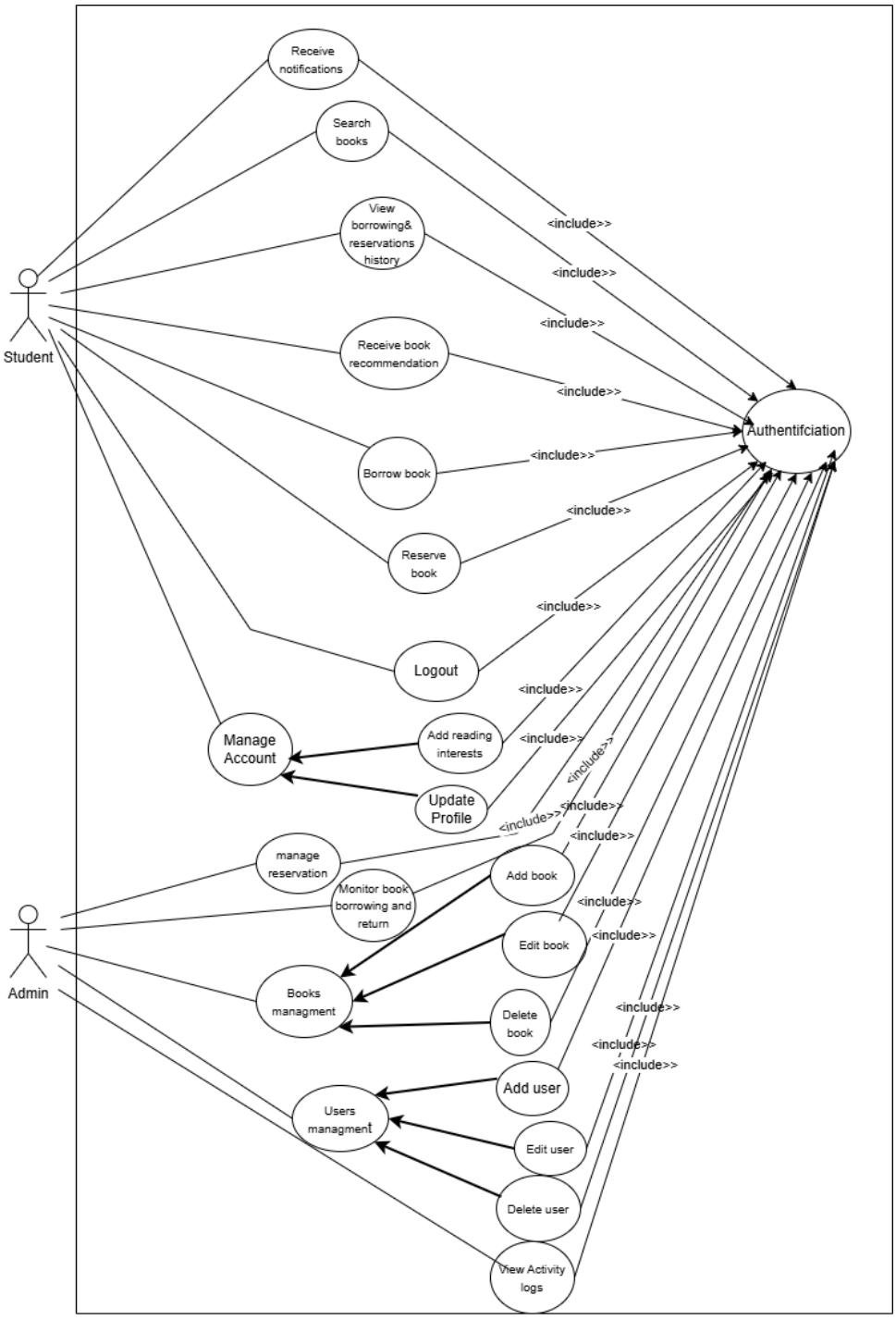

Figure 2.4: Use case diagram.

2.5.2 Sequence Diagrams

Sequence diagrams model the dynamic interactions between system components over time for specific scenarios. They are particularly useful for illustrating the order in which messages are exchanged between actors, the user interface, the application layer, and the database. The following sequence diagrams are presented for the four most critical workflows in the

proposed system.

Sequence Diagram 1: Borrow a book

When the student requests to borrow a book, the librarian scans the barcode or enters the book ISBN into the library system, as shown in Figure (2.5). The system then checks the book status by sending a request to the database. If the book is available, the database returns the current status to the system, which updates the book status to ``Borrowed'' and records the new borrowing transaction. The system then displays a successful borrowing message to the librarian, who hands the book to the student. However, if the book is already reserved or unavailable, the system displays a borrowing denial message, and the librarian informs the student that the borrowing process cannot be completed.

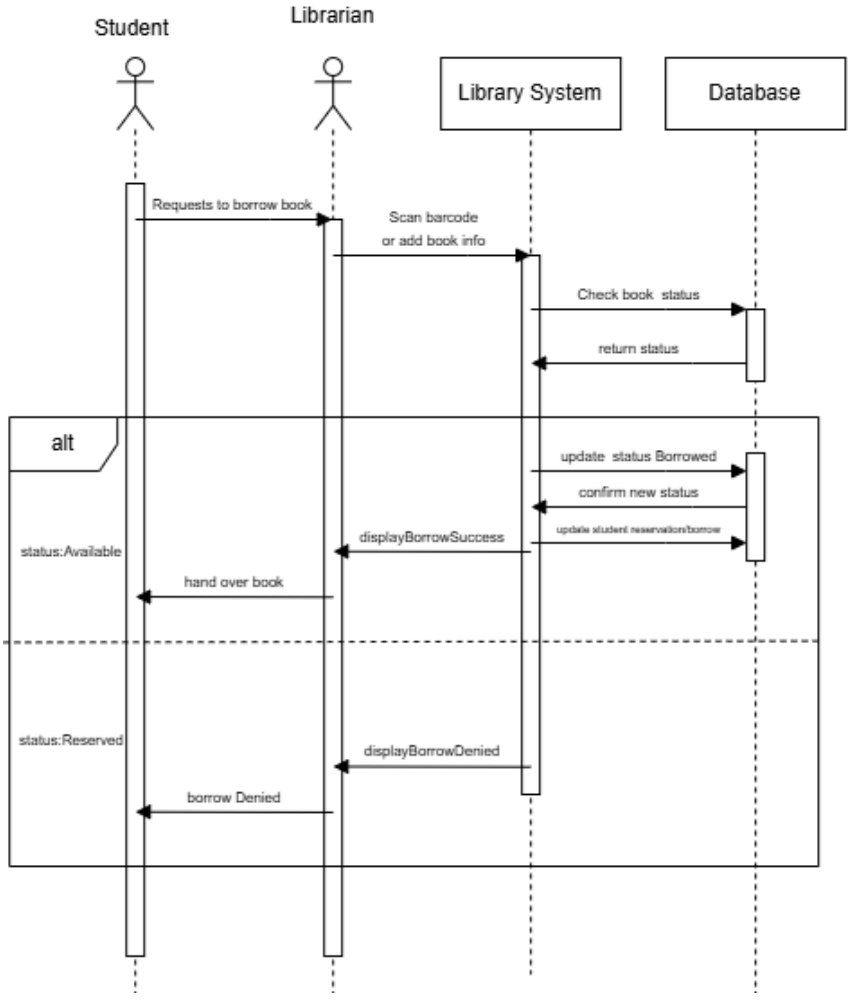

Figure 2.5: Book borrowing sequence diagram.

Sequence Diagram 2: Return a book

When the student returns a book, the librarian searches for the borrowed book using the barcode through the library system, as shown in Figure (2.6). The system then retrieves the

borrowing record from the database, including the borrowing date, due date, and student information. After receiving the record, the system calculates the delay period and computes the corresponding fine amount if the return is overdue.

If the number of delayed days is greater than zero, the system updates the fine record in the database by storing the student identifier and the calculated fine amount. However, if there is no delay, the fine amount remains equal to zero. After processing the return operation, the system updates the book status to ``Available'' in the database and displays the return result, including the fine amount if applicable. Finally, the librarian confirms the return operation to the student.

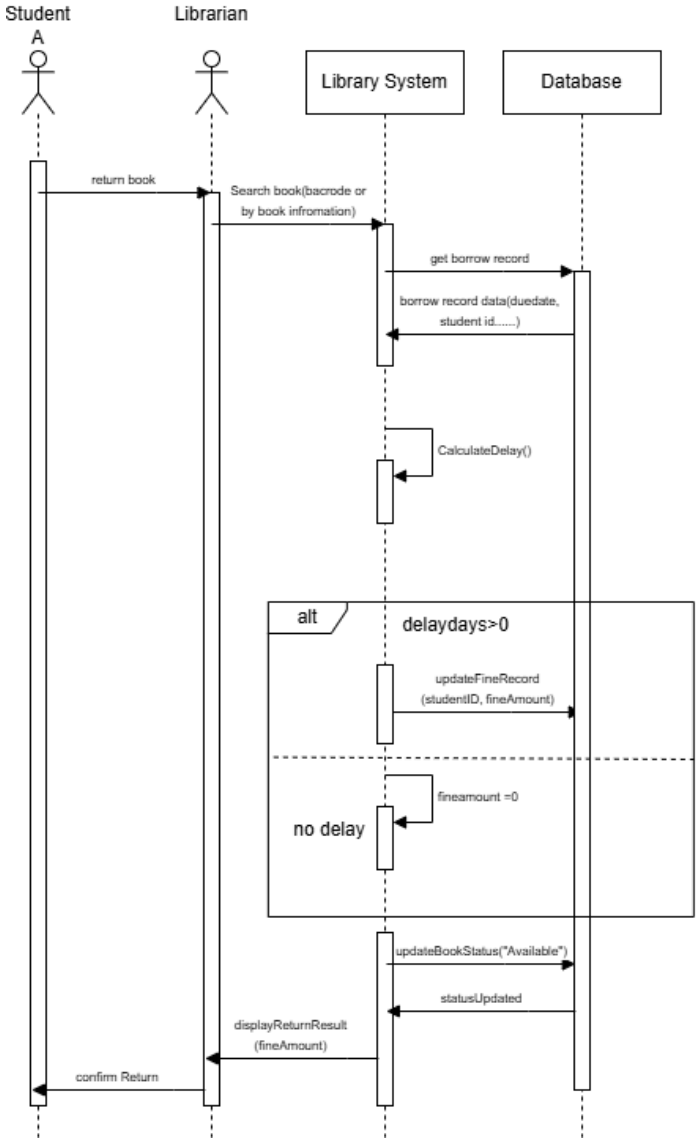

Figure 2.6: Book return sequence diagram.

Sequence Diagram 3: Create a book

When the administrator adds a new book, the book information is first entered into the library system, as illustrated in Figure (2.7). After receiving the book data, the system automatically generates an embedding vector representing the semantic content of the book. The system then sends both the bibliographic information and the generated vector embedding to the database for storage.

Before saving the new record, the database verifies whether the ISBN already exists. If the ISBN is already registered, the database returns an error message indicating that the ISBN is not unique, and the system informs the administrator that the book creation process has failed. However, if the ISBN is unique, the database successfully stores the new book information and vector embedding, after which the system notifies the administrator that the book has been created successfully.

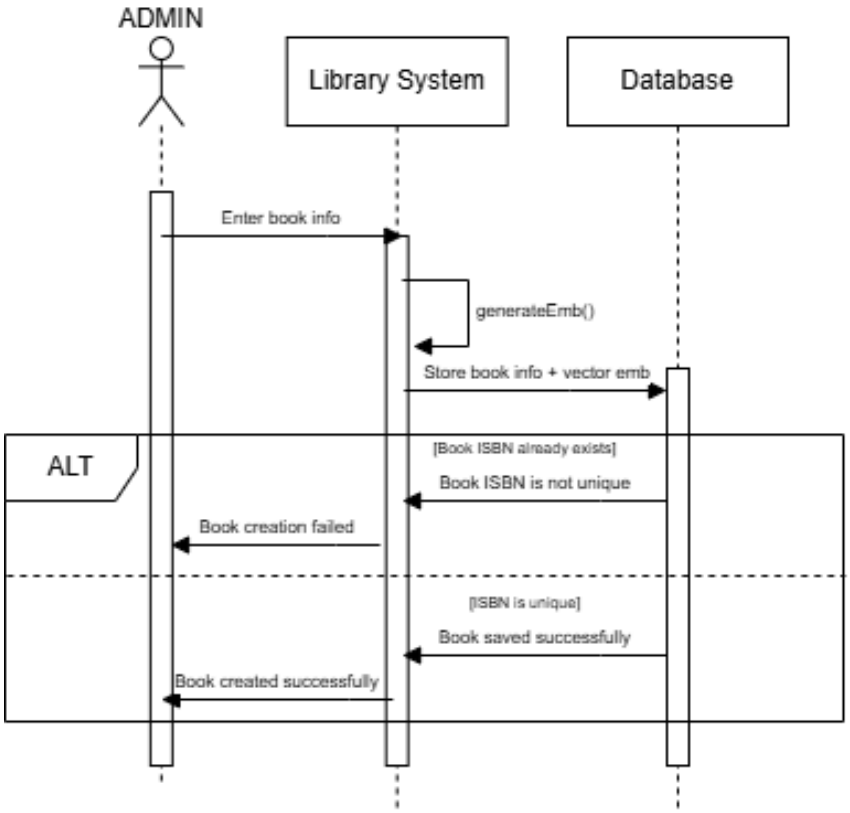

Figure 2.7: Book creation sequence diagram.

2.5.3 Class Diagram

The class diagram represents the static structural design of the intelligent library management system, as shown in Figure (2.8). It illustrates the main entities of the system, their attributes, and the relationships established between them. The diagram serves as a conceptual blueprint for both the application layer and the database architecture. It also reflects the object-oriented

design adopted during system development.

The principal classes identified in the system are described below.

User

The `User` class represents the main entity for all users interacting with the intelligent library management system. It contains general user information such as the identifier (`id`), authentication identifier (`clerkId`), email address (`email`), first name (`firstName`), last name (`lastName`), user role (`role`), reading interests (`readingInterests`), and profile image URL (`avatarUrl`), in addition to creation and update timestamps. The `User` class acts as a parent entity for both `Student` and `Librarian` classes.

Student

The `Student` class represents library members who interact with the system to access library services. A student can log into the platform, manage favourite books and collections, borrow and reserve books, and receive personalised recommendations generated by the intelligent recommendation module.

Librarian

The `Librarian` class represents the administrative user responsible for supervising and managing library operations. Librarians can manage books, authors, categories, users, borrowing operations, reservations, and activity logs within the system.

Book

The `Book` class represents a bibliographic resource available in the library. It includes attributes such as the identifier (`id`), title (`title`), description (`description`), ISBN number (`isbn`), publication year (`publishedYear`), cover image URL (`coverImageUrl`), and total number of copies (`totalCopies`). The class also supports digital resources through the attributes `isDigital` and `digitalFileUrl`. Additionally, each book contains an embedding vector (`embedding`) used for semantic search and recommendation functionalities.

Author

The `Author` class represents the author of one or more books. It contains an identifier (`id`), author name (`name`), biography (`bio`), and timestamps for creation and updates. One author may be associated with multiple books.

Category

The `Category` class is used to classify books according to their subjects or domains. Each category contains an identifier (`id`), a category name (`name`), and timestamps for creation and updates. A category can contain multiple books, while a book may belong to multiple categories.

BookCopy

The `BookCopy` class represents a physical copy of a book available in the library. It contains attributes such as the identifier (`id`), barcode (`barcode`), status (`status`), condition (`condition`), storage location (`location`), and timestamps. Each copy belongs to a specific book and allows the system to manage physical inventory and borrowing status.

Borrows

The `Borrows` class models borrowing transactions within the library system. It stores information such as the borrowing status (`status`), borrowing date (`borrowDate`), due date (`dueDate`), return date (`returnDate`), fine amount (`fineAmount`), and additional notes. Borrow records are associated with students and books in order to manage circulation operations.

Reservation

The `Reservation` class represents the reservation mechanism used when books are unavailable. It contains the reservation status (`status`), queue position (`position`), expiration date (`expiresAt`), and timestamps. Reservations allow students to reserve books and manage waiting lists efficiently.

Notification

The `Notification` class represents system messages sent to users. A notification contains a title (`title`), message body (`message`), notification type (`type`), read status (`read`), redirect link (`link`), and creation timestamp. Notifications are used for events such as borrowing reminders, reservation updates, and administrative announcements.

ActivityLog

The `ActivityLog` class records important actions performed within the platform. It stores the action name (`action`), additional details (`details`), entity type (`entityType`), entity identifier (`entityId`), IP address (`ipAddress`), and creation timestamp. Activity logs are mainly used for monitoring, auditing, and security purposes.

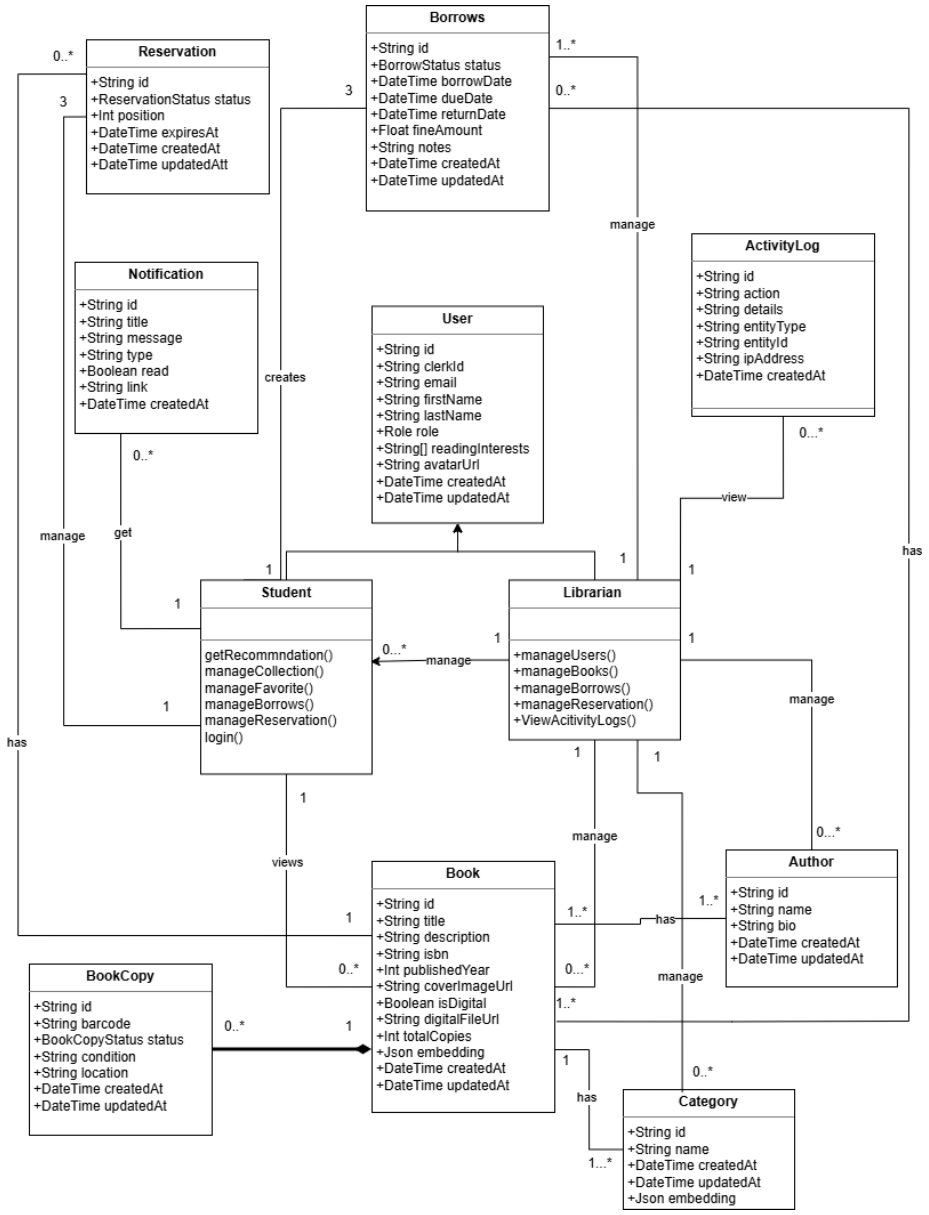

Figure 2.8: Class diagram.

2.6 Conclusion

This chapter presented the analysis and design of the proposed intelligent library management system by defining its functional and non-functional requirements, describing the general system architecture, and modelling the main system components and workflows using UML diagrams. It also introduced the design of the semantic search and recommendation modules.The following chapter will focus on the implementation phase by presenting the technologies used, development environment, and the final interfaces.

Chapter 3

Chapter 3: Implementation

3.1 Introduction

This chapter presents the practical implementation of the proposed intelligent library management system. It describes the development environment used during the project, including hardware and software tools, frameworks, and programming languages. It then explains how the intelligent components, the embedding generation pipeline and the semantic search and recommendation APIs — were implemented using Ollama and the `bge-m3` model. Finally, the chapter presents the developed interfaces of the system, covering both the administrator and student sides of the platform.

3.2 Working Environment

3.2.1 Hardware Materials

The development of the intelligent library management system was carried out using the following hardware equipment:

- **Laptop:** 8 GB RAM, Intel(R) Core(TM) i7-7500U, Nvidia 940MX, 15-inch screen.

3.2.2 Software Materials

Development Environment

Visual Studio Code was used as the primary code editor throughout the development phase. It provides a rich set of extensions, integrated terminal, debugging tools, and version control support, making it well suited for full-stack web development.

Collaboration and Version Control

Git and **GitHub** were used for version control and team collaboration. GitHub facilitated code sharing, branch management, and project tracking across all development stages.

Programming Language

TypeScript was used as the sole programming language for both the frontend and backend of the application. TypeScript provides static type checking over JavaScript, reducing runtime errors and improving code maintainability. The entire codebase---including server-side logic, API routes, and UI components---is written in TypeScript, ensuring type consistency across the full stack.

Frontend Framework

Next.js (version 16) was used to build the user-facing side of the platform. Next.js is a React-based framework (React 19) that supports server-side rendering, file-based routing via the App Router pattern, and React Server Components for efficient data fetching. The UI was built using **shadcn/ui** components (based on Radix UI primitives) styled with **Tailwind CSS v4**, with **Framer Motion** for animations and **React Hook Form** paired with **Zod** for form handling and validation. Dark mode support was implemented using **next-themes**.

Backend Framework

Rather than a separate backend server, the application uses **Next.js Server Actions** as the primary backend pattern. Server Actions (`"use server"` directives) handle all server-side operations including user management, book circulation logic, collection management, and database communication. Two REST API routes were implemented using Next.js Route Handlers: `/api/search-books` for the semantic book search endpoint, and `/api/suggest-categories` for AI-powered category suggestion based on a book's title and description. This architecture eliminates the need for a standalone backend framework such as Express, while still providing full server-side functionality within the Next.js application.

Database

PostgreSQL was used as the relational database management system, hosted on **Neon** (a serverless PostgreSQL platform). It stores all structured data including users, books, borrowing records, reservations, notifications, and activity logs. **Prisma** (version 7) was used as the ORM for type-safe database access and schema management, connecting via the Neon serverless adapter. The database schema defines 15 models covering the full domain: User,

Author, Category, Book, BookCopy, Borrow, Reservation, Notification, ActivityLog, Collection, and their junction tables.

File Storage

Supabase Storage was used for storing book cover images and digital book PDF files. Files are organised into `covers/` and `pdfs/` directories within a Supabase storage bucket, with public URLs generated for frontend display.

Authentication

Clerk was used as the authentication and user management service. It provides sign-in and sign-up flows, session management, and role-based access control (admin and student roles). Clerk middleware enforces route protection at the edge, with user roles stored in Clerk's public metadata.

AI Integration

Ollama was used as the local AI inference server, running the `bge-m3` embedding model for generating vector representations of book metadata (title, description, categories). These embeddings enable two AI-powered features: **semantic book search**, which combines cosine similarity scores with keyword matching to find contextually relevant books, and **personalised recommendations**, which suggest books based on a user's borrowing history and stated reading interests. Similarity computation is performed in application code using a custom cosine similarity function. The chatbot component was implemented using the `deepseek-v4-flash` large language model through OpenCode to generate conversational responses for student queries.

Embedding Model — bge-m3

BGE-M3 is a multilingual text embedding model developed by the BAAI research group. It is designed for semantic retrieval and similarity tasks, supporting multilingual and cross-lingual text representations. The model was selected for this project because of its strong retrieval performance, efficient semantic search capabilities, and its ability to process multilingual bibliographic content, including English Arabic and French text. In addition, the model can be executed locally through Ollama, making it suitable for privacy-preserving and offline deployment environments [16].

3.3 Implementation of the Intelligent Components

3.3.1 Embedding Generation Pipeline

When a librarian adds a new book to the catalogue, the system automatically generates a vector embedding for that book. This process is triggered within a Next.js Server Action after the book's bibliographic information is saved to the database via Prisma. The embedding is computed by constructing a structured prompt that includes the book's title, description, and category names, which is then sent to the Ollama API running locally. The `mxbai-embed-large` model processes this text and returns a 1024-dimensional vector, which is stored in the `embedding` field (of type `Json`) on the `Book` model in the database. The following code shows the embedding generation function implemented as a Server Action:

```
async function generateEmbedding(
  title: string,
  description: string | null,
  categoryIds: string[]
) {
  try {
    const cats = await prisma.category.findMany({
      where: { id: { in: categoryIds } },
    })
    const categoryNames = cats.map((c) => c.name)
    const categoryText = categoryNames.length > 0
      ? `Category: ${categoryNames.join(", ")}`
      : ""
    const res = await fetch(
      "http://localhost:11434/api/embeddings",
      {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({
          model: "bge-m3",
          prompt: `This is a university-level academic book.
Title: ${title}
Description: ${description || ""}
${categoryText}
It belongs to the academic domain of ${
  categoryNames.join(", ") || "general academia"
}.`,
        }),
      }
    )
    const data = await res.json()
    return data.embedding as number[] | null
```

```
  } catch {
    return null
  }
}
```

Listing 3.1: Embedding generation function using Ollama and bge-m3

After a book is created, this function is called automatically and the resulting vector is stored in the database as follows:

```
const book = await prisma.book.create({
  data: {
    title,
    description,
    isbn: isbn || null,
    publishedYear,
    isDigital,
    authors: {
      create: authorIds.map((authorId) => ({ authorId })),
    },
    categories: {
      create: categoryIds.map((categoryId) => ({ categoryId })),
    },
  },
})
// Generate and store embedding
const embedding = await generateEmbedding(
  title, description, categoryIds
)
if (embedding) {
  await prisma.book.update({
    where: { id: book.id },
    data: { embedding },
  })
}
```

Listing 3.2: Storing the book embedding via Prisma after book creation

3.3.2 Semantic Search API

The search API receives a natural language query from the user, converts it into an embedding vector using the same `bge-m3` model, and then performs a hybrid search that combines semantic similarity with keyword matching. Unlike a pure vector database approach, the system loads book embeddings from PostgreSQL via Prisma and computes cosine similarity in the application layer. The following code shows the implementation of the semantic search endpoint:

```
export async function GET(req: Request) {
  const { searchParams } = new URL(req.url)
  const query = searchParams.get('q') || ''
  if (!query) {
    return NextResponse.json([])
  }
  // Convert query to embedding vector
  const embeddingRes = await fetch(
    "http://localhost:11434/api/embeddings",
    {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({
        model: "bge-m3",
        prompt: query,
      }),
    }
  )
  const embeddingData = await embeddingRes.json()
  const queryEmbedding = embeddingData.embedding
  // Fetch all books and compute scores
  const books = await prisma.book.findMany()
  const q = query.toLowerCase()
  const scored = books
    .filter((book) => Array.isArray(book.embedding))
    .map((book) => {
      const bookEmbedding = book.embedding as number[]
      const semanticScore = cosineSimilarity(
        queryEmbedding, bookEmbedding
      )
      let keywordScore = 0
      if (book.title.toLowerCase().includes(q)) {
        keywordScore += 0.3
      }
      if (book.description?.toLowerCase().includes(q)) {
        keywordScore += 0.2
      }
      const finalScore = semanticScore + keywordScore
      return { ...book, score: finalScore,
               semanticScore, keywordScore }
    })
  const sorted = scored.sort((a, b) => b.score - a.score)
  const THRESHOLD = 0.55
  const filtered = sorted.filter((b) => b.score >= THRESHOLD)
  const results = filtered.length > 0
```

```
    ? filtered
    : sorted.slice(0, 5)
  return NextResponse.json(
    results.map((b) => ({
      id: b.id,
      title: b.title,
      coverImageUrl: b.coverImageUrl,
      score: b.score,
    }))
  )
}
```

Listing 3.3: Hybrid semantic search API combining vector similarity with keyword matching

The search uses a hybrid scoring approach. The **semantic score** is computed using cosine similarity between the query embedding and each book's embedding, implemented as a utility function:

```
export function cosineSimilarity(a: number[], b: number[]) {
  const dot = a.reduce((sum, val, i) => sum + val * b[i], 0)
  const magA = Math.sqrt(
    a.reduce((sum, val) => sum + val * val, 0)
  )
  const magB = Math.sqrt(
    b.reduce((sum, val) => sum + val * val, 0)
  )
  return dot / (magA * magB)
}
```

Listing 3.4: Cosine similarity function used for vector comparison

The **keyword score** adds a bonus of 0.3 for a title match and 0.2 for a description match, providing a relevance boost for books that contain the exact query terms. The final score is the sum of both components. A threshold of 0.55 filters out low-relevance results; if no results pass the threshold, the top 5 are returned as a fallback to avoid empty result sets.

3.3.3 Recommendation System

The recommendation system generates personalised book suggestions for a student using a three-level fallback strategy based on their borrowing history, reading interests, and profile completeness.

Level 1: Borrowing and Favourite History. The system aggregates the embedding vectors of the three most recently borrowed books and the three most recently favourited books. These embeddings are averaged into a single *taste profile vector* that represents the user's

reading preferences. This profile vector is then compared against all book vectors using cosine similarity, excluding books the user has already borrowed or favourited. The top 15 most similar books are returned as recommendations.

```
const [recentBorrows, recentFavorites] = await Promise.all([
  prisma.borrow.findMany({
    where: { userId: user.id },
    orderBy: { borrowDate: "desc" },
    take: 3,
    select: { bookId: true },
  }),
  prisma.favoriteBook.findMany({
    where: { userId: user.id },
    orderBy: { createdAt: "desc" },
    take: 3,
    select: { bookId: true },
  }),
])
const sourceBookIds = [...new Set([
  ...recentBorrows.map((b) => b.bookId),
  ...recentFavorites.map((f) => f.bookId),
])]
const sourceBooks = await prisma.book.findMany({
  where: { id: { in: sourceBookIds } },
})
const validEmbeddings = sourceBooks
  .filter((b) => Array.isArray(b.embedding)
    && (b.embedding as number[]).length > 0)
  .map((b) => b.embedding as number[])
// Average embeddings into a user taste vector
const dim = validEmbeddings[0].length
const avgEmbedding = new Array(dim).fill(0) as number[]
for (const emb of validEmbeddings) {
  for (let i = 0; i < dim; i++) {
    avgEmbedding[i] += emb[i] / validEmbeddings.length
  }
}
// Score and rank candidate books
const candidates = await prisma.book.findMany({
  where: excludeIds.length > 0
    ? { id: { notIn: excludeIds } } : {},
  include: {
    authors: { include: { author: true } },
    categories: { include: { category: true } },
    copies: true,
  },
```

```
})
const scored = candidates
  .filter((b) => Array.isArray(b.embedding)
    && (b.embedding as number[]).length > 0)
  .map((b) => ({
    ...b,
    score: cosineSimilarity(avgEmbedding, b.embedding),
  }))
  .sort((a, b) => b.score - a.score)
  .slice(0, 15)
```

Listing 3.5: Level 1 recommendation: averaging embeddings from borrow and favourite history

Level 2: Reading Interests. If the user has no borrowing or favourite history, but has specified reading interests in their profile, the system generates an embedding from those interests using Ollama. This embedding then serves as the taste profile vector and the same cosine similarity comparison is performed against all book embeddings.

```
if (user.readingInterests.length > 0) {
  const interestsText = user.readingInterests.join(", ")
  const embeddingRes = await fetch(
    "http://localhost:11434/api/embeddings",
    {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({
        model: "bge-m3",
        prompt: `Academic reading interests: ${interestsText}`,
      }),
    }
  )
  const embeddingData = await embeddingRes.json()
  if (embeddingData.embedding) {
    const candidates = await prisma.book.findMany({
      include: {
        authors: { include: { author: true } },
        categories: { include: { category: true } },
        copies: true,
      },
    })
    const scored = candidates
      .filter((b) => Array.isArray(b.embedding)
        && (b.embedding as number[]).length > 0)
      .map((b) => ({
```

```
      ...b,
      score: cosineSimilarity(
        embeddingData.embedding, b.embedding
      ),
    }))
    .sort((a, b) => b.score - a.score)
    .slice(0, 15)
  return { books: scored,
           state: "recommendations" as const }
 }
}
```

Listing 3.6: Level 2 recommendation: embedding from user reading interests

3.3.4 Category Suggestion API

When a librarian adds a new book, selecting the appropriate category can be time-consuming, especially when the catalogue contains many categories. To assist with this process, the system provides an AI-powered category suggestion API that recommends the most relevant categories based on the book's title and description. The API receives the book's title and description, generates an embedding vector using the `bge-m3` model via Ollama, and then compares this vector against the pre-computed embedding vectors stored for each category in the database. The comparison uses cosine similarity to measure semantic relatedness, with a small keyword bonus added when the category name appears in the book's text. The top five categories ranked by combined score are returned as suggestions.

```
export async function POST(req: Request) {
 try {
  const { title, description } = await req.json()
  if (!title && !description) {
   return Response.json([])
  }
  // Create embedding for the book input
  const embeddingRes = await fetch(
   "http://localhost:11434/api/embeddings",
   {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({
     model: "bge-m3",
     prompt: `
Title: ${title}
Description: ${description}
`,
    }),
```

```
      }
    )
    const embeddingData = await embeddingRes.json()
    const queryEmbedding = embeddingData.embedding
    // Get categories with pre-computed embeddings
    const categories = await prisma.category.findMany()
    const validCategories = categories.filter(
      (c) => Array.isArray(c.embedding)
    )
    if (validCategories.length === 0) {
      return Response.json([])
    }
    const q = `${title} ${description}`.toLowerCase()
    // Score categories by semantic similarity
    const scored = validCategories.map((cat) => {
      const semanticScore = cosineSimilarity(
        queryEmbedding,
        cat.embedding as number[]
      )
      let keywordScore = 0
      if (cat.name.toLowerCase().includes(q)) {
        keywordScore += 0.2
      }
      const finalScore = semanticScore + keywordScore
      return {
        id: cat.id,
        name: cat.name,
        score: finalScore,
      }
    })
    const sorted = scored.sort((a, b) => b.score - a.score)
    const top = sorted.slice(0, 5)
    return Response.json(top)
  } catch (err) {
    console.error(err)
    return Response.json([])
  }
}
```

Listing 3.7: Category suggestion API using embedding similarity to recommend categories for a book

This API follows the same hybrid scoring pattern used in the semantic book search: a semantic score computed via cosine similarity between the query embedding and each category's embedding, plus a keyword bonus for exact text matches. The approach requires that each category has a pre-computed embedding vector stored in the database, which is generated

when categories are created or updated. By suggesting categories automatically, the system reduces the manual effort required from librarians and improves the consistency of catalogue classification.

3.3.5 Library Assistant Chatbot

The Library Assistant is an AI-powered chatbot embedded across the platform as a floating widget, designed to answer user questions about library services, book availability, borrowing rules, and library regulations. It uses a Retrieval-Augmented Generation (RAG) architecture that combines vector-based book retrieval with structured library regulations injected into the system prompt of a large language model.

RAG Architecture

When a user submits a question, the chatbot executes a three-step pipeline:

1. **Query Embedding:** The user's message is converted into a semantic embedding vector using Ollama and the `bge-m3` model, identical to the embedding generation process used for books and categories.

2. **Context Retrieval:** The query embedding is compared against all stored book embeddings using cosine similarity combined with keyword matching. The top five most relevant books are retrieved and formatted with their title, author, category, availability status, description, and publication year. A similarity threshold of 0.5 filters out unrelated books.

3. **Response Generation:** A system prompt is constructed containing the full text of the library's internal regulations (all 18 articles covering borrowing rules, user obligations, violations, and penalties) and the retrieved book results. This prompt is then sent to the `deepseek-v4-flash` large language model via the OpenCode API, which streams the response back to the user in real time.

The RAG pipeline is implemented in a dedicated module that handles embedding generation, book retrieval, and system prompt construction:

```
export async function fetchQueryEmbedding(query: string): Promise<number[]> {
  const res = await fetch("http://localhost:11434/api/embeddings", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({
      model: "bge-m3",
      prompt: query,
    }),
```

```
  })
  const data = await res.json()
  return data.embedding
}
export async function retrieveRelevantBooks(
  query: string,
  queryEmbedding: number[],
  limit = 5
): Promise<BookResult[]> {
  const books = await prisma.book.findMany({
    include: {
      authors: { include: { author: true } },
      categories: { include: { category: true } },
      copies: true,
    },
  })
  const q = query.toLowerCase()
  const scored = books
    .filter((book) => Array.isArray(book.embedding))
    .map((book) => {
      const bookEmbedding = book.embedding as number[]
      const semanticScore = cosineSimilarity(
        queryEmbedding, bookEmbedding
      )
      let keywordScore = 0
      if (book.title.toLowerCase().includes(q))
        keywordScore += 0.3
      if (book.description?.toLowerCase().includes(q))
        keywordScore += 0.2
      if (book.categories.some((bc) =>
        bc.category.name.toLowerCase().includes(q)))
        keywordScore += 0.2
      return { book,
        score: semanticScore + keywordScore }
    })
    .sort((a, b) => b.score - a.score)
  const THRESHOLD = 0.5
  const filtered = scored.filter(
    (s) => s.score >= THRESHOLD
  )
  return filtered.slice(0, limit).map(({ book }) => ({
    title: book.title,
    author: book.authors
      .map((ba) => ba.author.name).join(", "),
    category: book.categories
      .map((bc) => bc.category.name).join(", "),
```

```
    status: book.copies?.some(
      (c) => c.status === "AVAILABLE"
    ) ? "Available" : "All copies borrowed",
    description: book.description,
    publishedYear: book.publishedYear,
  }))
}
export function buildSystemPrompt(
  relevantBooks: BookResult[]
): string {
  const bookContext = relevantBooks.length > 0
    ? relevantBooks.map((b, i) =>
        `${i + 1}. "${b.title}" by ${b.author}`
      ).join("\n")
    : "No relevant books were found."
  return `You are the official virtual assistant
for the Faculty Library of Mathematics and
Computer Science.
ROLE AND BEHAVIOR:
- Respond in the user's language
- Cite specific Article numbers when
  discussing rules
- Only recommend books from the RELEVANT
  BOOKS section below
LIBRARY REGULATIONS:
${LIBRARY_REGULATIONS}
RELEVANT BOOKS FROM CATALOG:
${bookContext}`
}
```

Listing 3.8: RAG pipeline: query embedding, book retrieval, and system prompt construction

The `LIBRARY_REGULATIONS` constant embedded in the system prompt contains the complete text of the library's internal regulations in English, comprising a preliminary chapter and four main chapters (Library Departments, Library Operations, User Obligations, Violations and Penalties) spanning 18 articles. This allows the chatbot to answer questions such as *``How many books can a graduate student borrow?''* by citing the specific article that defines borrowing quotas.

Chat API Endpoint

The chat endpoint `POST /api/chat` orchestrates the full RAG pipeline and streams the LLM response using Server-Sent Events over newline-delimited JSON:

```
import { createOpenAICompatible }
  from "@ai-sdk/openai-compatible"
```

```
import { streamText } from "ai"
const opencode = createOpenAICompatible({
  baseURL: "https://opencode.ai/zen/go/v1",
  apiKey: process.env.OPENCODE_API_KEY,
})
export async function POST(req: Request) {
  const { message, history } = await req.json()
  if (!message.trim()) {
    return NextResponse.json(
      { error: "Message is required" },
      { status: 400 }
    )
  }
  // Step 1: Generate query embedding
  const queryEmbedding =
    await fetchQueryEmbedding(message)
  // Step 2: Retrieve relevant books
  const relevantBooks =
    await retrieveRelevantBooks(
      message, queryEmbedding)
  // Step 3: Build system prompt
  const systemPrompt =
    buildSystemPrompt(relevantBooks)
  // Construct conversation history
  const chatMessages = [
    ...history.map((m) => ({
      role: m.role,
      content: m.content,
    })),
    { role: "user", content: message },
  ]
  // Stream LLM response
  const result = streamText({
    model: opencode("deepseek-v4-flash"),
    system: systemPrompt,
    messages: chatMessages,
  })
  // Encode as NDJSON
  const encoder = new TextEncoder()
  const stream = new ReadableStream({
    async start(controller) {
      for await (const chunk
                  of result.textStream) {
        controller.enqueue(
          encoder.encode(
            JSON.stringify(
```

```
            { content: chunk }) + "\n"))
      }
      controller.enqueue(
        encoder.encode(
          JSON.stringify({ done: true }) + "\n"))
      controller.close()
    },
  })
  return new Response(stream, {
    headers: {
      "Content-Type": "application/x-ndjson",
    },
  })
}
```

Listing 3.9: Chat API endpoint coordinating embedding, retrieval, and LLM streaming

The `@ai-sdk/openai-compatible` adapter wraps the OpenCode API as an OpenAI-compatible provider, enabling the use of the Vercel AI SDK's `streamText` function for structured streaming. Responses are encoded as newline-delimited JSON (`application/x-ndjson`), where each line contains either a content chunk or a done signal. This format allows the frontend client to progressively decode and display the assistant's reply without waiting for the entire response to complete.

Chat Widget Interface

The chat widget is implemented as a client-side React component displayed as a floating button across the application interface. It maintains conversation history to provide contextual continuity between user interactions and communicates with the backend chat API for response generation. The interface supports role-based visibility, message formatting, loading states, and error handling, as illustrated in Figure 3.1.

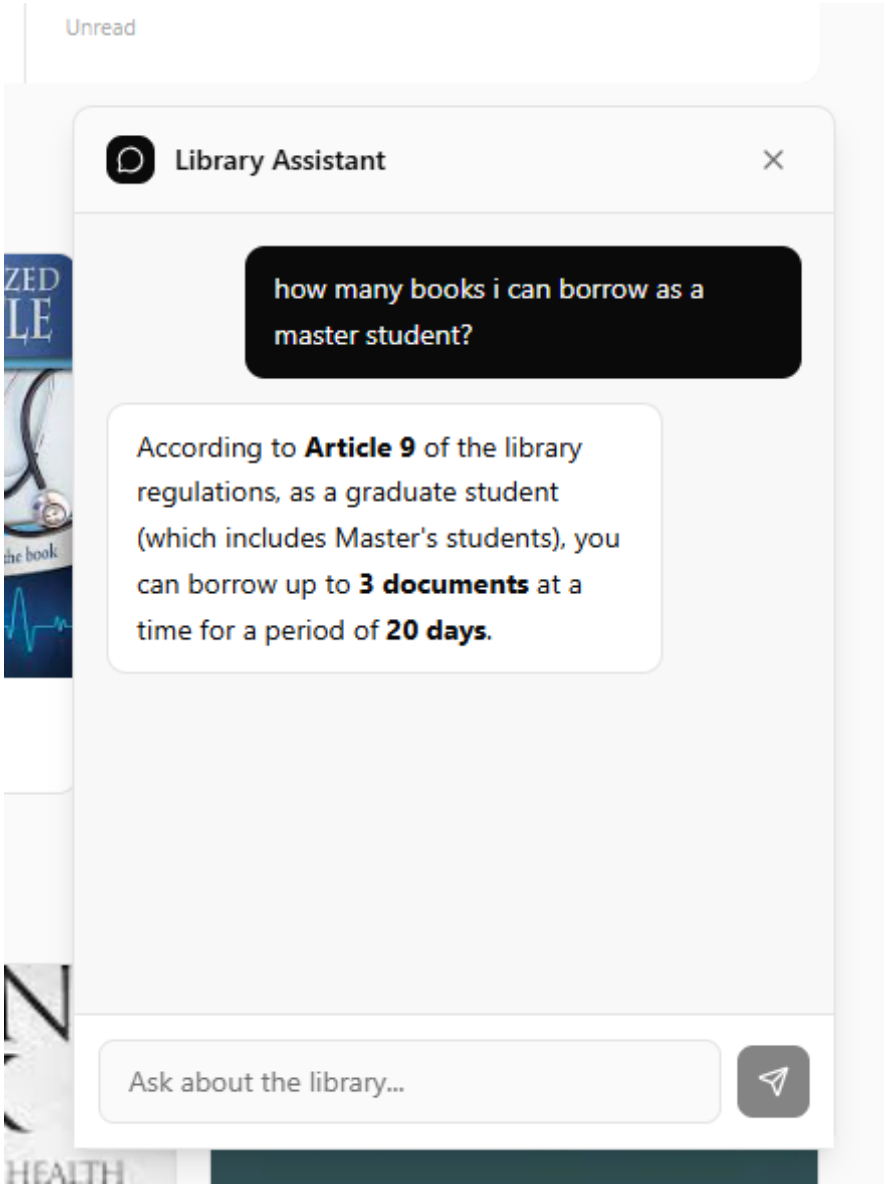

Figure 3.1: The Library Assistant chatbot widget.

3.3.6 Administrator Interfaces

Administrator Dashboard

The administrator dashboard provides a high-level overview of the library system through six summary statistics: total books in the catalogue, registered users, active borrowings, overdue loans, active reservations, and total fines collected. Below the statistics, the dashboard displays a popular books section showing the five most borrowed titles, and a recent activity feed listing the last ten system events with user information and relative timestamps, as shown in Figure 3.2.

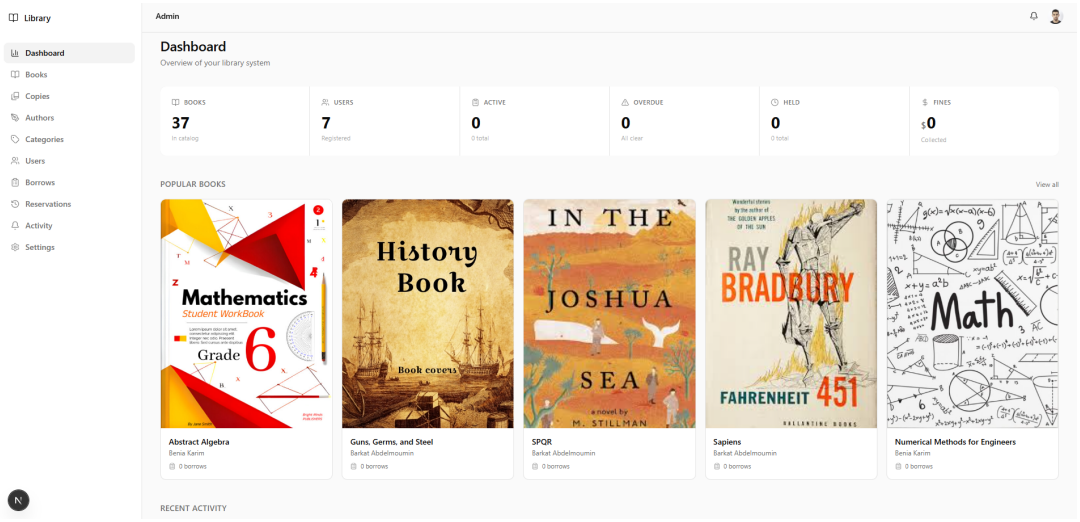

Figure 3.2: Administrator dashboard.

Book Management Page

The book management page lists all books in the catalogue with a search bar for filtering by title, author, or ISBN, and a category dropdown filter. Each book row displays the cover thumbnail, title, ISBN, author names, category badges, type (physical or digital), and copy count. Administrators can add, edit, or delete books. Adding or editing a book opens a dialog form where the librarian fills in the bibliographic information, selects authors and categories, uploads a cover image, and optionally attaches a digital PDF file. Upon submission, the embedding generation pipeline is triggered automatically in the background, as shown in Figure 3.3.

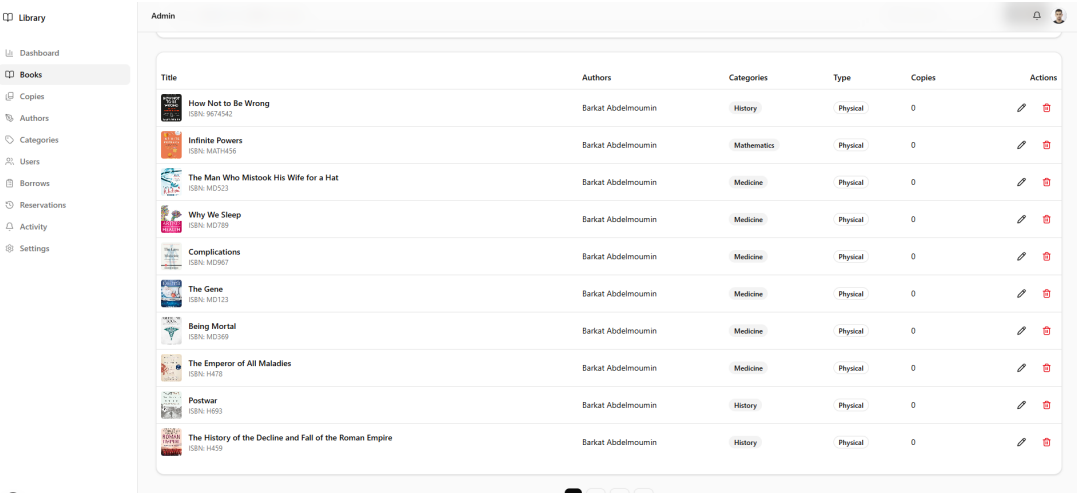

Figure 3.3: Book management page.

Add Book Dialog

The add book dialog collects all required bibliographic information including title, description, ISBN, publication year, book type (physical or digital), cover image upload, and optional PDF file for digital books. Authors are selected using a multi-select field with removable chips. Categories are also selected via a multi-select field, but the system assists the librarian by automatically suggesting relevant categories based on the entered title and description using the category suggestion API (described in Section 3.7). Upon submission, the book is saved to the database and its embedding vector is generated and stored automatically without any manual intervention, as shown in Figure 3.4.

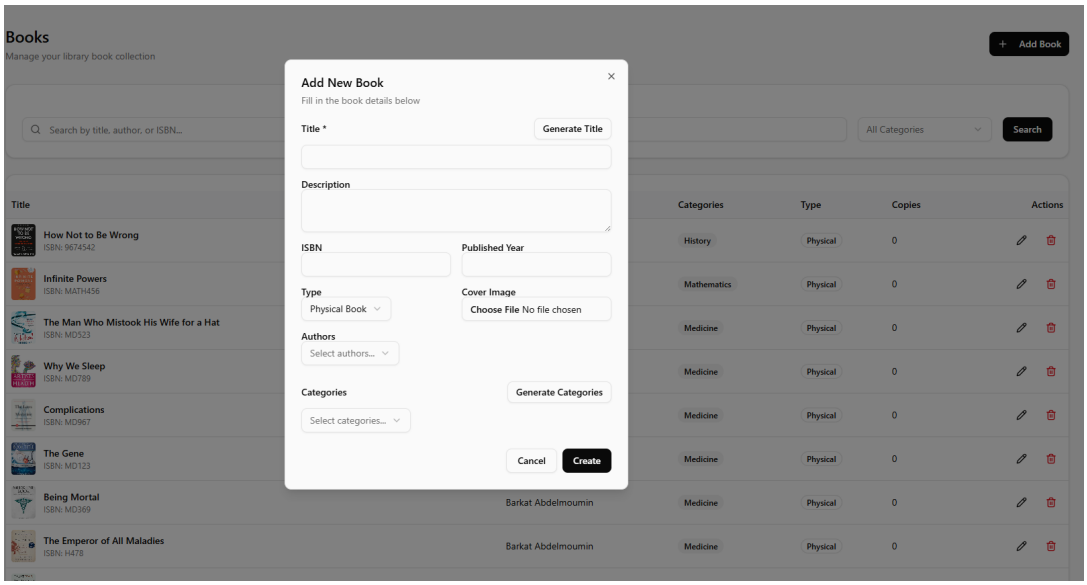

Figure 3.4: Add book dialog.

Book Copies Management Page

The copies management page displays the physical inventory of the library with a summary strip showing the count of available, borrowed, under maintenance, and reserved copies. Individual copies are presented as cards with the book cover image, status badge, barcode, location, and condition. Administrators can filter copies by barcode or status, and add, edit, or delete copies through a dialog form that records the associated book, barcode, condition, and location, as shown in Figures 3.5 and 3.6.

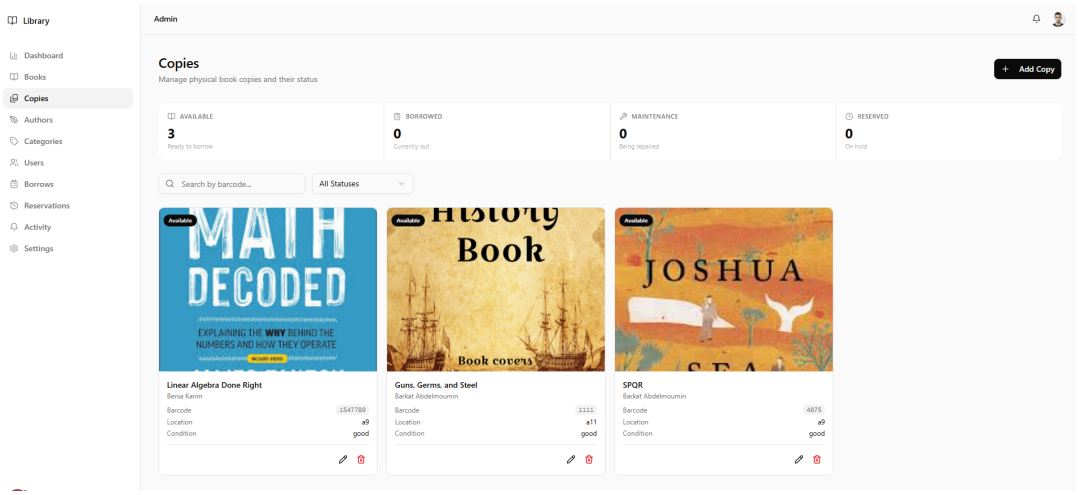

Figure 3.5: Book copies management page.

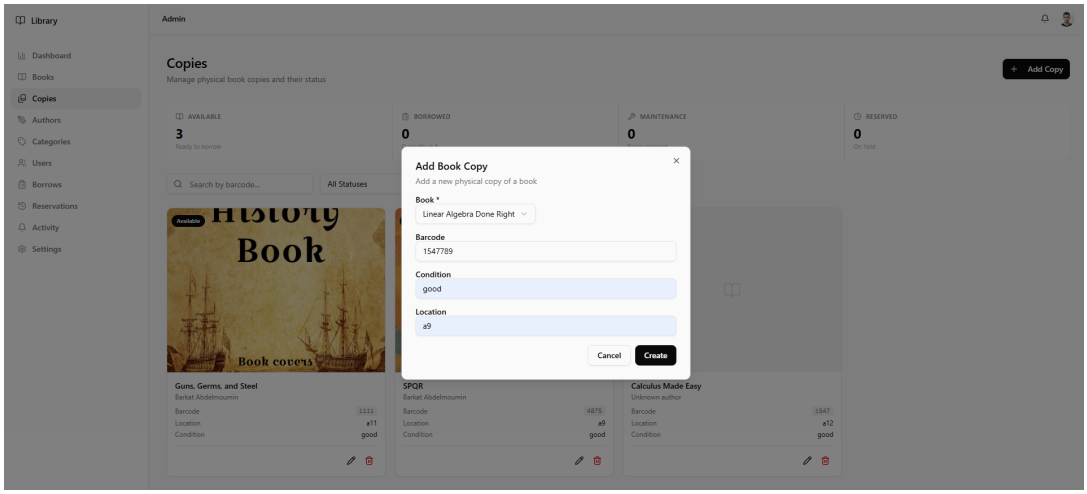

Figure 3.6: Book copy creation dialog.

Author Management Page

The author management page lists all authors with a debounced search input for filtering by name. Each row shows the author name, a truncated biography, and the number of associated books. Administrators can add, edit, or delete authors through a dialog form that collects the name and an optional biography, as shown in Figure 3.7.

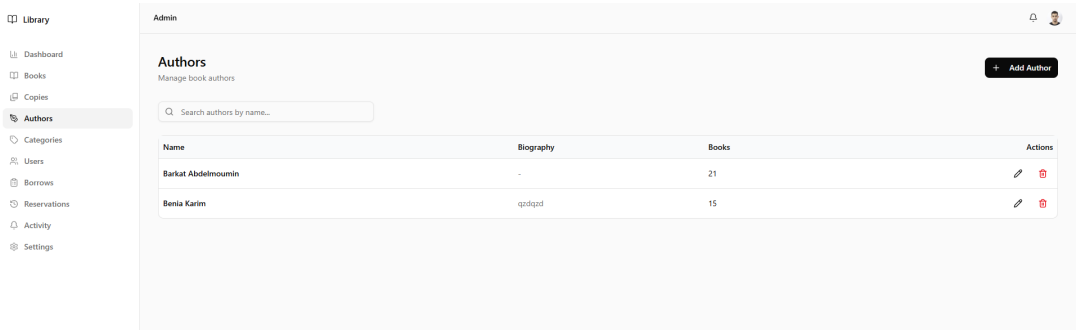

Figure 3.7: Author management page.

Category Management Page

The category management page lists all book categories with a search input for filtering by name. Each row shows the category name and the number of books assigned to it. Administrators can add, edit, or delete categories through a simple dialog form with a single name field, as shown in Figure 3.8.

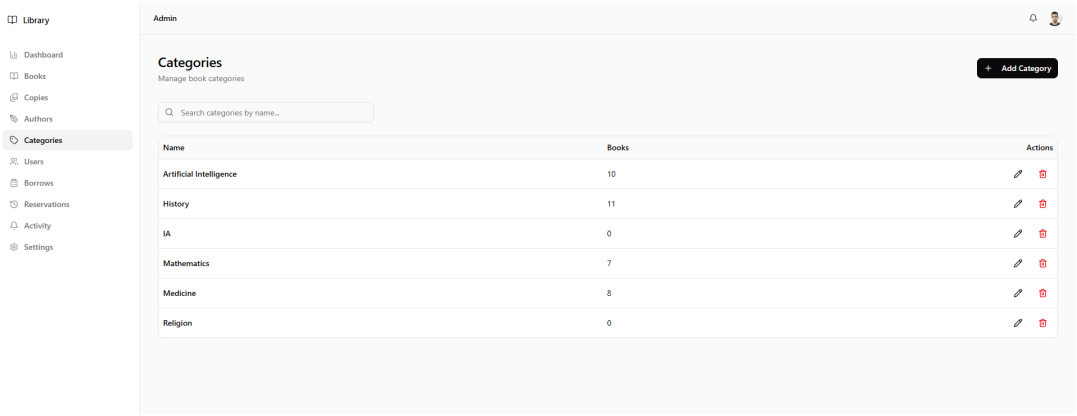

Figure 3.8: Category management page.

User Management Page

The user management page lists all registered users with a search bar for filtering by name or email, and a role filter dropdown. Each row displays the user's avatar, name, email, role badge, and join date. Administrators can create new user accounts, change user roles between student and administrator, and delete accounts. Clicking on a student opens a detail page showing the student's profile information, a manual borrow form, and their borrowing history, as shown in Figure 3.9.

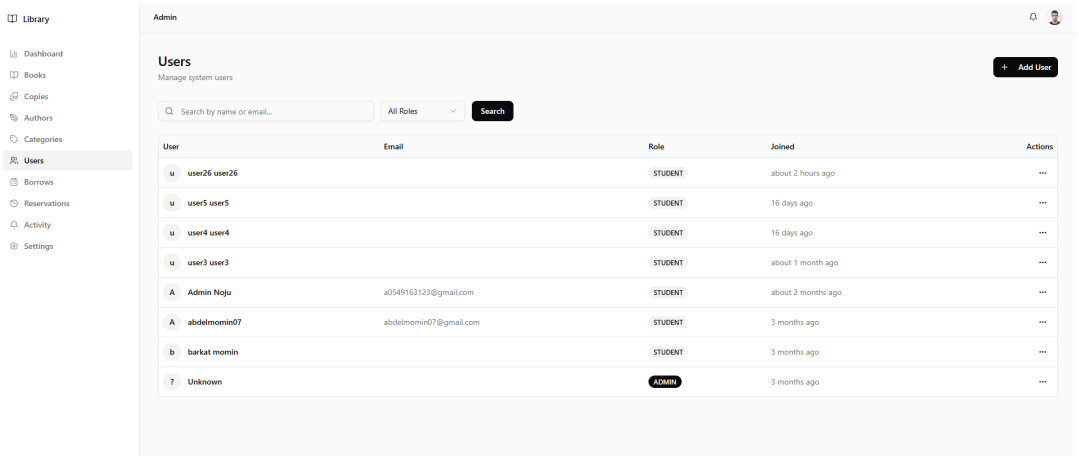

Figure 3.9: User management page.

Borrowing Management Page

The borrowing management page provides a summary strip showing the count of active borrows, overdue loans, returned borrows, and total fines accrued. A status filter dropdown allows filtering by active, returned, or overdue borrows. The table displays each borrowing record with the book title, student name, copy barcode, borrow date, due date (highlighted in red if overdue), status, and fine amount. Administrators can process returns by clicking the return button on active borrows, which calculates and displays any applicable fine, as shown in Figure 3.10.

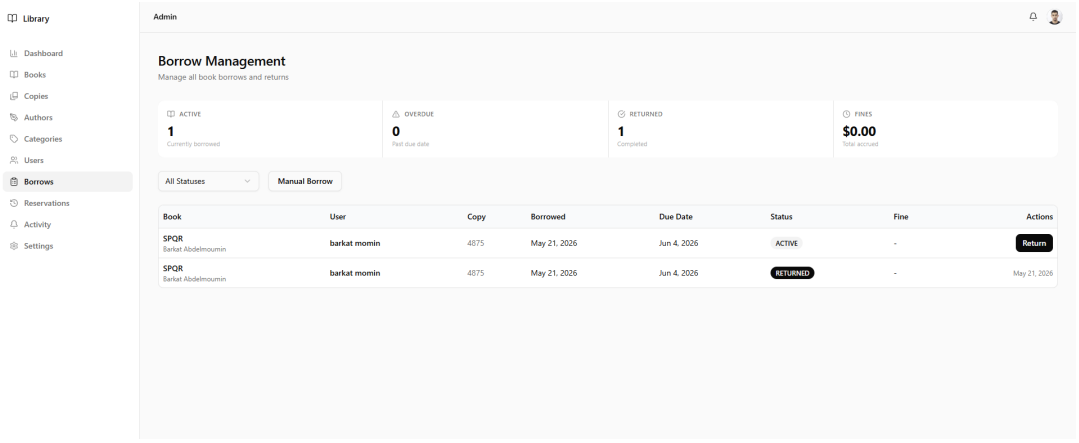

Figure 3.10: Borrowing management page.

Manual Borrowing Form

The manual borrowing form is used when a librarian records a borrowing operation from the administration panel. The librarian enters the book ISBN and the student name, and the system locates the corresponding book and user, creates the borrowing record, and updates

the copy status. A separate version of this form is also available on each student's detail page, where the student is already identified and only the ISBN is required, as shown in Figure 3.11.

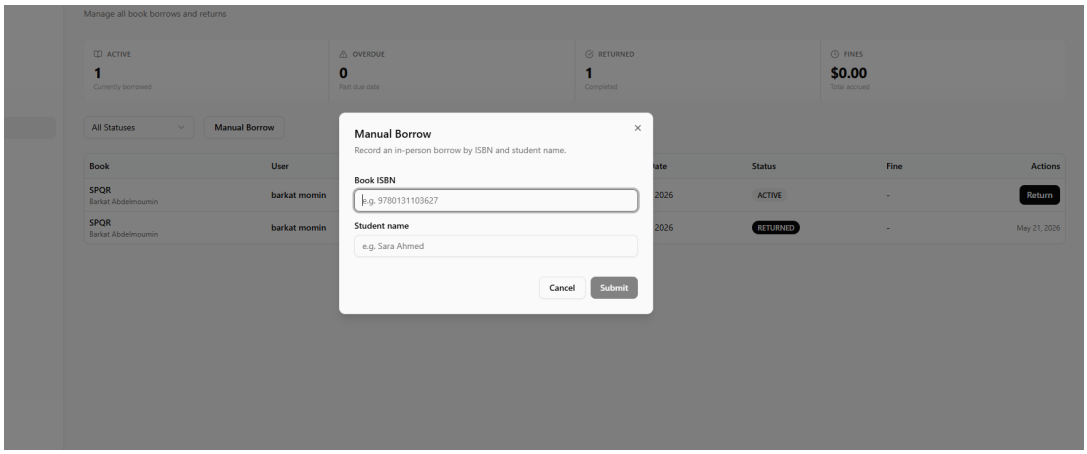

Figure 3.11: Manual borrowing form.

Reservations Management Page

The reservations management page displays a summary strip with counts of active, completed, cancelled, and expired reservations. A status filter allows narrowing the view. The table lists each reservation with the book title, student name, queue position, reservation date, expiry date, and available copy count. Administrators can cancel active reservations, which triggers a notification to the student, as shown in Figure 3.12.

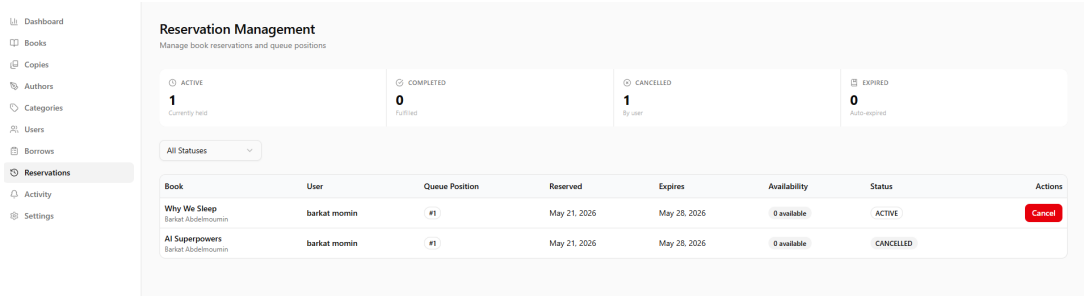

Figure 3.12: Reservations management page.

Activity Logs Page

The activity logs page records important actions performed in the system, such as book management operations, borrowing events, and administrative updates. Each log entry displays a colour-coded action badge (green for creation, blue for updates, red for deletion, purple for borrows, teal for returns), the user who performed the action, a details description, the affected entity, and a relative timestamp. A search input allows filtering logs by action type, user, or details, as shown in Figure 3.13.

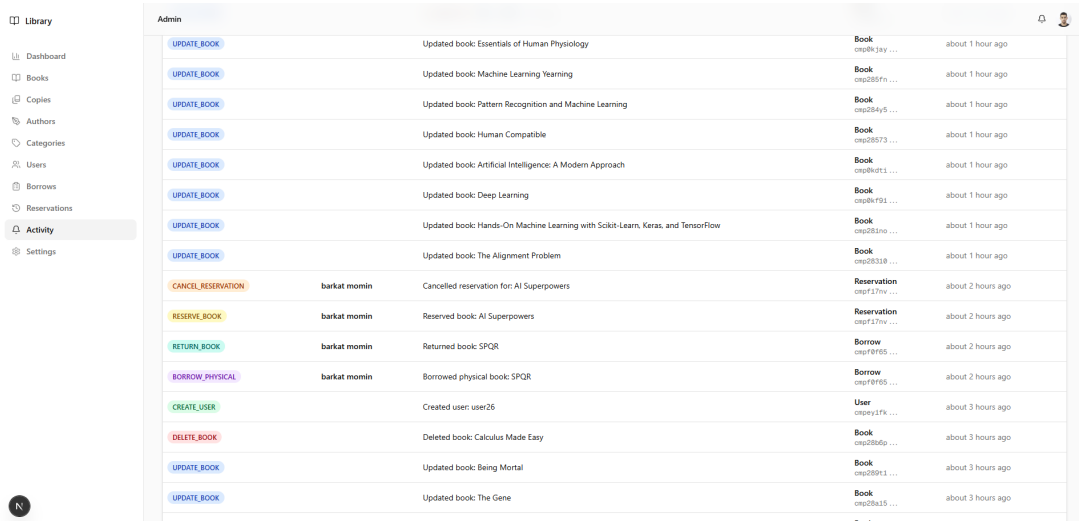

Figure 3.13: Activity logs page.

3.3.7 Student Interfaces

Student Dashboard

The student dashboard greets the user by name and provides a search bar for quick book lookup. It displays four summary statistics: active borrows, active reservations, favourite books, and unread notifications. A **Recommended for You** section shows AI-powered book suggestions based on the student's borrowing and favourite history; if no reading interests are set, it prompts the student to enter them. Below, a popular books section highlights the most borrowed titles, and a current borrows section lists upcoming due dates, as shown in Figure 3.14.

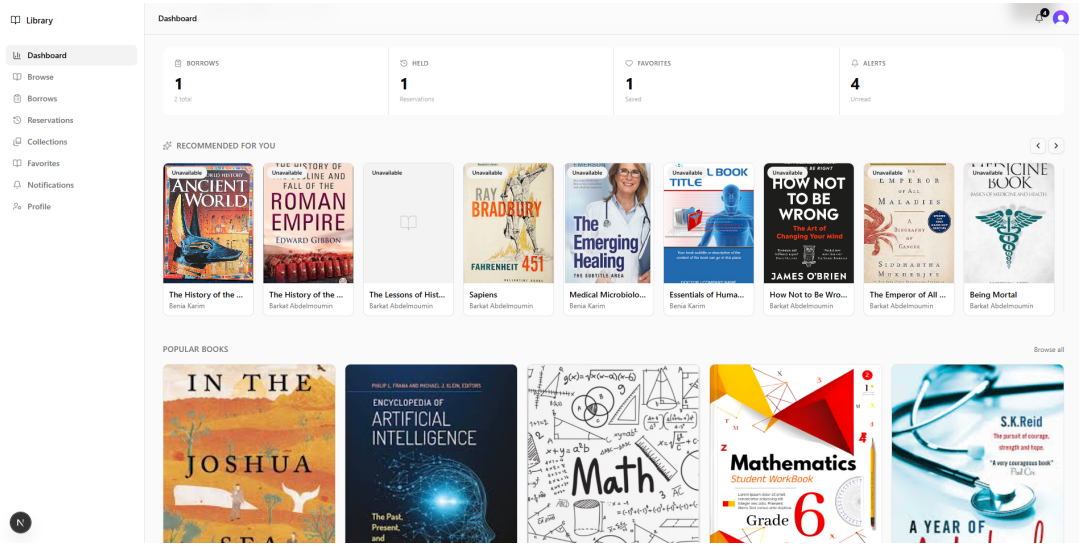

Figure 3.14: Student dashboard.

Browse Books Page

The browse page allows students to explore the catalogue using text search, category filtering, and a book type filter (physical or digital) Figure 3.15. Books are displayed in a grid with cover images, availability badges, titles, and authors. When a search query is submitted, the page switches to semantic search mode, which sends the query to the search API and returns results ranked by a combination of semantic similarity and keyword matching.

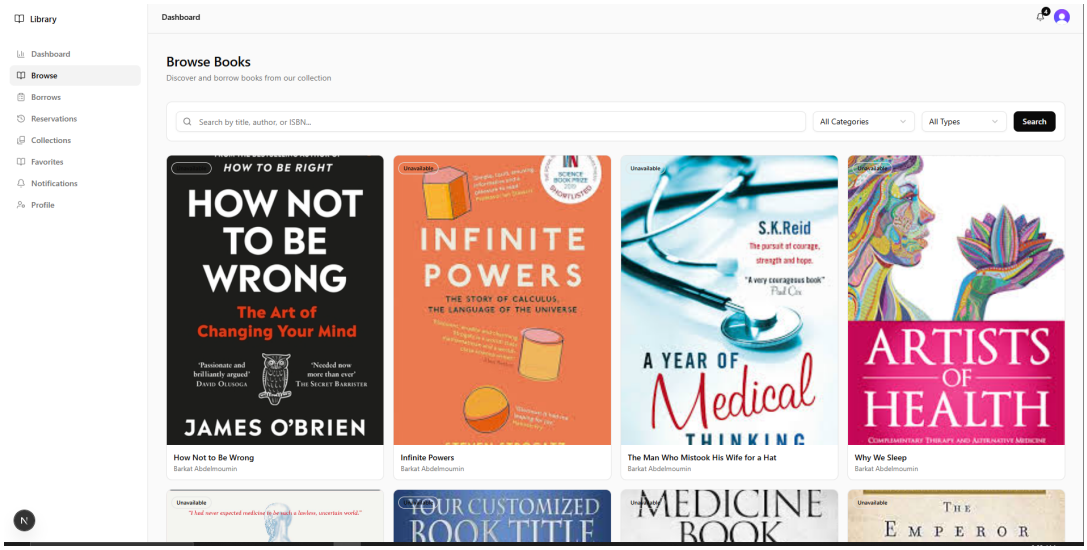

Figure 3.15: Student browse page.

Semantic Search Results

When a student enters a natural language query, the browse page switches to search mode and displays results ranked by the hybrid scoring system. Each result shows the book cover, availability status, and relevance score. The search combines vector similarity with keyword matching to return contextually relevant books even when the query does not exactly match any title or description, as shown in Figure 3.16.

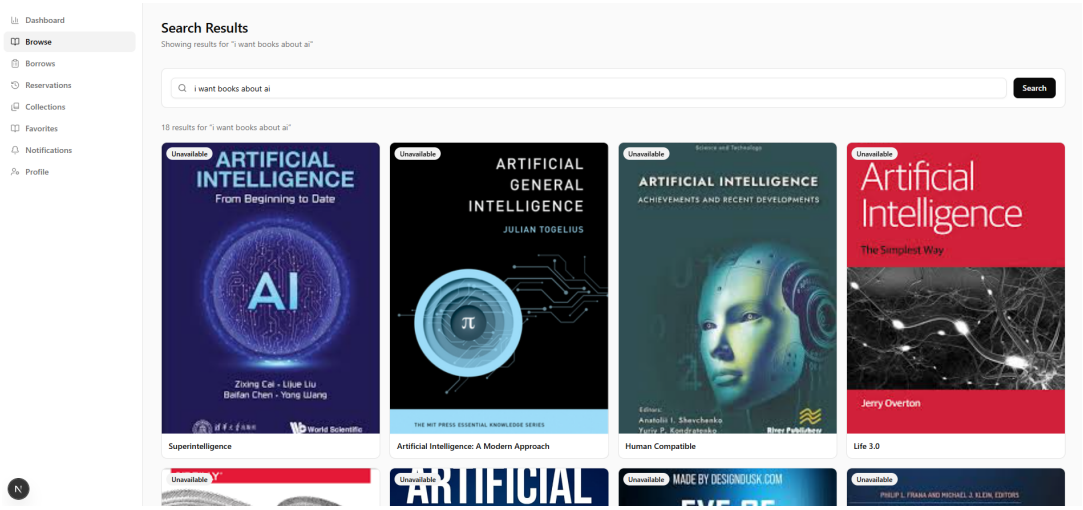

Figure 3.16: Semantic search results.

Borrowing Pages

The borrowing page is organised into two tabs: Active and Returned. The active tab displays the student's current borrows with borrow dates and due dates (highlighted in red if overdue). The returned tab shows the borrowing history including any fines that were applied. Summary counts for active and overdue borrows are displayed at the top, as shown in Figures 3.17 and 3.18.

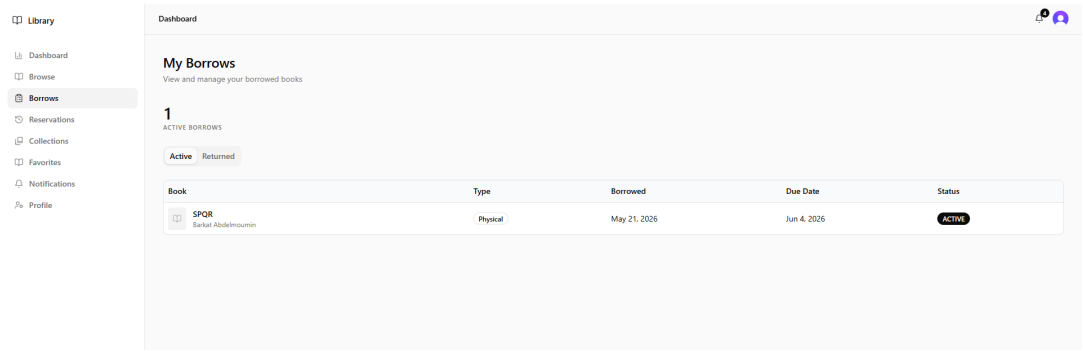

Figure 3.17: Active borrows tab.

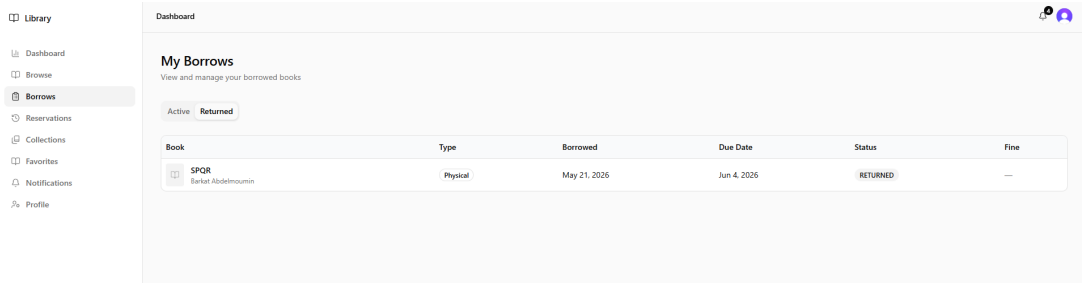

Figure 3.18: Returned borrows tab.

Reservations Page

The reservations page lists the student's book reservations with the queue position, reservation date, and current status (active, completed, or cancelled). Students can cancel active reservations through a confirmation dialog, which updates the reservation and frees the queue position, as shown in Figure 3.19.

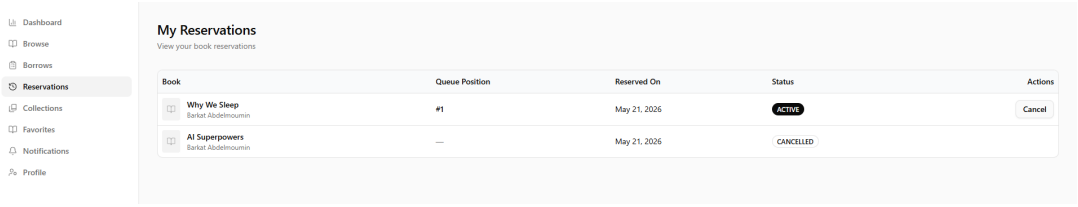

Figure 3.19: Student reservations page.

Collections Page

The collections page allows students to create personal reading collections by grouping selected books together. Each collection is displayed as a card with the collection name, description, and a grid of book thumbnails. Students can create new collections, delete existing ones, and remove individual books from a collection without deleting the book itself, as shown in Figure 3.20.

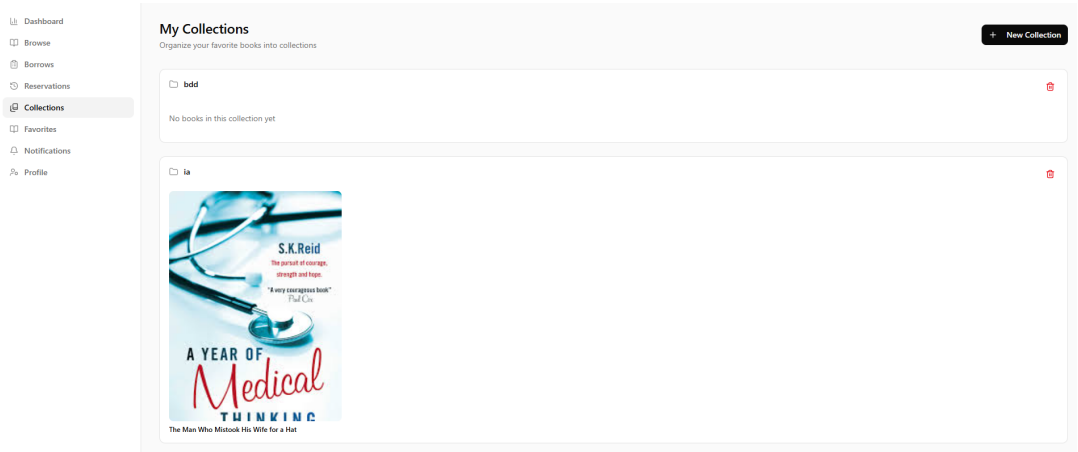

Figure 3.20: Student collections page.

Favourites Page

The favourites page displays the books a student has marked as favourites, presented as a grid of cover images with availability badges. Students can remove a book from their favourites by clicking the heart icon on each card. Favourite books also contribute to the recommendation system's taste profile, influencing the AI-powered suggestions on the dashboard, as shown in Figure 3.21.

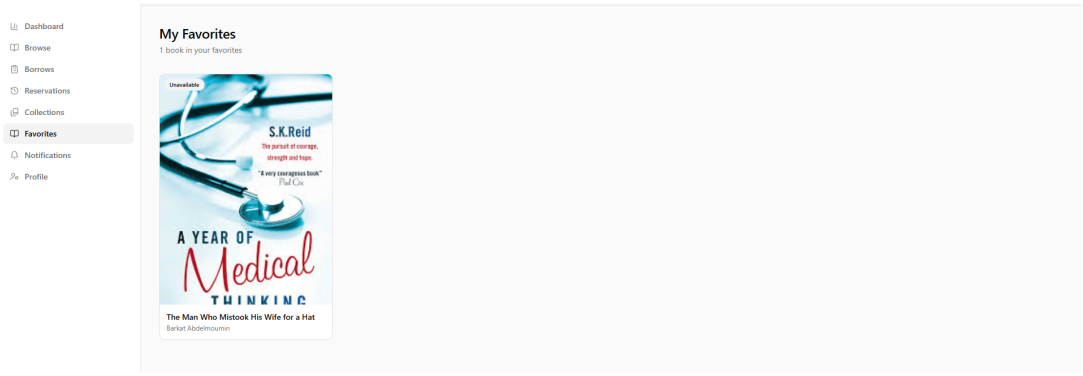

Figure 3.21: Student favourites page.

Notifications Page

The notifications page displays system messages addressed to the student, including borrowing reminders, reservation updates, and administrative announcements. Unread notifications are visually highlighted with a blue border and a distinct icon. Students can mark individual notifications as read or mark all as read at once, as shown in Figure 3.22.

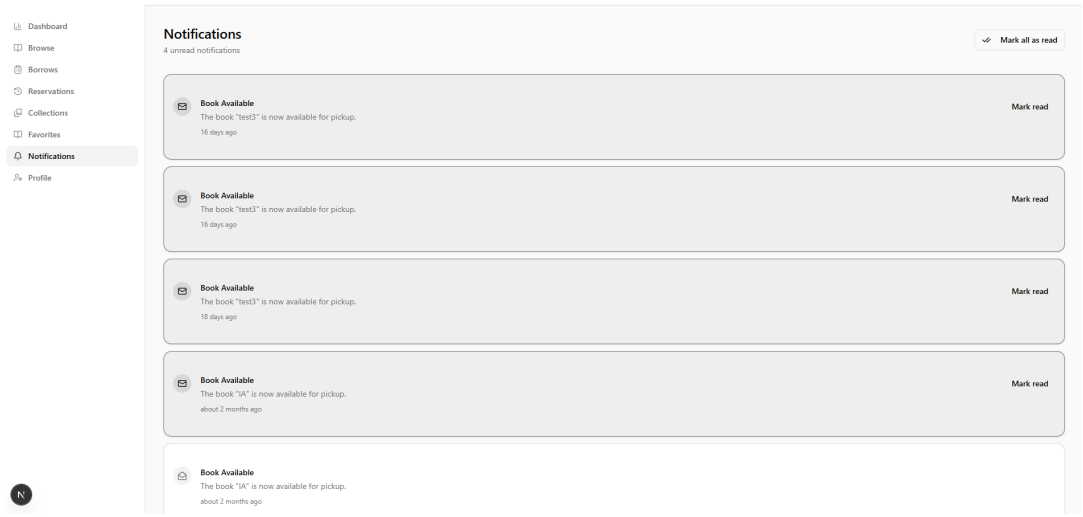

Figure 3.22: Student notifications page.

Profile Page

The profile page displays the student's avatar, name, email, role, and join date alongside a summary grid of their activity statistics (active borrows, reservations, favourites, collections, and unread notifications). The edit form allows students to update their first name, last name, and reading interests. Reading interests are entered as a comma-separated list and are used by the recommendation engine to generate personalised book suggestions when borrowing history is insufficient, as shown in Figure 3.23.

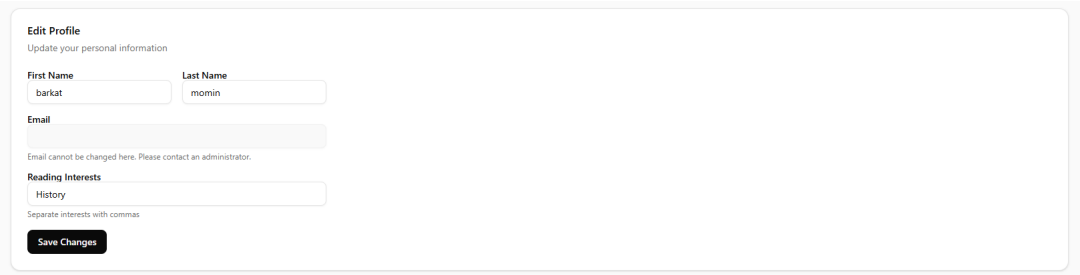

Figure 3.23: Student profile page.

3.4 Conclusion

This chapter presented the full implementation of the intelligent library management system, covering the development environment, AI-powered components, chatbot integration, and the final interfaces of the platform. The embedding generation pipeline, semantic search API, personalised recommendation mechanism, category suggestion API, and conversational chatbot were implemented using Next.js, TypeScript, Prisma, PostgreSQL, and Ollama with the `bge-m3` model. Book embeddings are stored in the database, while similarity computation is performed in the application layer. The developed interfaces satisfy the main functional requirements for both students and administrators, providing a complete and functional intelligent library platform.

General Conclusion

In conclusion, this project resulted in the development of an intelligent library management system designed to modernize and improve traditional library services through the integration of artificial intelligence technologies and modern web development tools. The system combines standard library management functionalities with intelligent features such as semantic search, personalized recommendations, an AI-powered chatbot, and AI-assisted category suggestion, providing a more efficient and user-friendly environment for both students and administrators.

The objectives of this project were successfully achieved through the implementation of a complete platform that supports book management, borrowing and reservation operations, digital resource access, notifications, and activity monitoring. In addition, the integration of semantic search and recommendation technologies using embedding vectors and similarity calculations significantly improved the accessibility and organization of library resources compared to traditional keyword-based systems.

Throughout the realization of this project, many important technical and practical skills were acquired, including full-stack web development, database design, API development, artificial intelligence integration, and software architecture design. Working on a real-world application also strengthened problem-solving abilities and provided valuable experience in designing scalable and maintainable systems capable of addressing practical challenges within modern library environments.

The technologies used in this project, including Next.js, TypeScript, PostgreSQL, Prisma, Clerk, Supabase, and Ollama with the `bge-m3` model, contributed to building a multilingual, modern, responsive, and intelligent platform capable of supporting both physical and digital library services efficiently.

Despite the successful implementation of the main functionalities, several improvements and extensions can still be considered in future work. Future developments may include advanced analytics dashboards, mobile applications, RFID integration for physical inventory management, and more advanced recommendation algorithms based on deep learning techniques.

Overall, this project demonstrates how artificial intelligence and modern web technologies can be effectively integrated to create smarter, more adaptive library management sys-

tems capable of meeting the evolving needs of modern academic environments.

References

[1] J. Charles W. Bailey, ``Intelligent library systems: Artificial intelligence technology and library automation systems,'' in *Advances in Library Automation and Networking*, J. A. Hewitt, Ed., vol. 4, Greenwich, CT: JAI Press, 1991, pp. 1–23.

[2] A. Adepoju and A. Esan, ``Revolutionizing library systems with advanced automation: A blueprint for efficiency in academic resource management,'' *International Journal of Scholarly Research in Multidisciplinary Studies*, vol. 5, no. 2, pp. 19–40, 2024.

[3] C. V. Anunobi and I. B. Okoye, ``Cloud computing implementation in libraries: A synergy for library services optimization,'' *International Journal of Library and Information Science*, vol. 10, no. 2, pp. 14–24, 2018.

[4] A. Asemi, A. Ko, and M. Nowkarizi, ``Intelligent libraries: A review on expert systems, artificial intelligence, and robot,'' *Library Hi Tech*, vol. 39, no. 2, pp. 412–434, 2021. DOI: 10.1108/LHT-02-2020-0038

[5] A. Deschenes et al., ``From card catalogs to semantic search: Building a human-centered discovery platform powered by ai technologies,'' *Information Technology and Libraries*, vol. 45, no. 1, 2026. DOI: 10.5860/ital.v45i1.17511

[6] N. Singh and P. Mahajan, ``Application of rfid technology in libraries,'' *DESIDOC Journal of Library and Information Technology*, vol. 34, no. 5, 2014.

[7] Anonymous, ``A data-driven approach for supporting new academic program collection development,'' *International Journal of Librarianship*, vol. 10, no. 1, 2025. [Online]. Available: https://journal.calaijol.org

[8] C. Musto et al., ``Semantics-aware recommender systems for digital libraries,'' *Proceedings of the ACM Conference*, 2017.

[9] N. Faruqui et al., ``Empowering library users: Creative strategies for engagement and innovation,'' *arXiv preprint*, 2024. [Online]. Available: https://arxiv.org/abs/2411.02993

[10] Y. Bakelli, ``The cerist library: Toward a model of a modern library in algeria,'' *Library Hi Tech News*, vol. 19, no. 2, 2002. DOI: 10.1108/lhtn.2002.23919baf.003

[11] Anonymous, *Challenges of syngeb software under the influence of the internet on algerian university libraries and competition from free and open-source documentary software*, Al-Manhal academic platform, English translation of the cited Arabic-language article title; used here to avoid non-Latin script in the bibliography.

[12] I. Bounhas and Y. Slimani, *Usability degree for arabized open source software phpmybibli integrated library system as a case study*, ResearchGate, 2009.

[13] Wikipedia contributors, *Pmb (software)*, Wikipedia, The Free Encyclopedia, 2025.

[14] scikit-learn developers, *Sklearn.metrics.pairwise.cosine_similarity*, scikit-learn Documentation, Open-source documentation for the cosine similarity metric, 2024.

[15] S. Kaufmann. ``Types of uml diagrams." Accessed: 2026-06-01. [Online]. Available: https://www.glockengiesserstrasse.info/wissenswertes/types-of-uml-diagrams.html

[16] Ollama. ``Bge-m3." Accessed: 2026-06-01. [Online]. Available: https://ollama.com/library/bge-m3

Appendix B: List of Acronyms

- **AI**: Artificial Intelligence
- **ML**: Machine Learning
- **NLP**: Natural Language Processing
- **OPAC**: Online Public Access Catalogue
- **SYNGEB**: Système Normalisé de Gestion des Bibliothèques