THOMPSON RIVERS UNIVERSITY

SENG 3210– Applied Software Engineering

Inked

Isaac Latta T00239008

Matia Landry T00707575

March 24, 2025

Table of Contents

1 Introduction........ 5
2 Design Problem........ 6
2.1 Problem Definition........ 6
2.2 Design Requirements........ 6
2.2.1 Functions........ 6
2.2.2 Non-functional requirements and constraints........ 6
3 Solution........ 7
3.1 Solution 1........ 7
3.2 Solution 2........ 7
3.3 Final Solution........ 7
3.3.1 Features and the Software Architecture........ 7
3.3.2 The system interfaces........ 7
3.3.3 The user interface design........ 7
3.3.4 The requirements traceability matrix........ 8
3.3.5 Environmental, Societal, Safety, and Economic Considerations........ 8
3.3.6 Limitations........ 8
4 Teamwork........ 8
4.1 Meeting 1........ 8
4.2 Meeting 2........ 8
4.3 Meeting 3........ 9
4.4 Meeting 4........ 9
5 Conclusion and Future Work........ 9
6 References........ 10
Appendix........ 11

List of Figures

Figure 1 ………………………………………………………………………………………… pg 10
Figure 2 ………………………………………………………………………………………… pg 11
Figure 3 ………………………………………………………………………………………… pg 11
Figure 4 ………………………………………………………………………………………… pg 12
Figure 5 ………………………………………………………………………………………… pg 14
Figure 6 ………………………………………………………………………………………… pg 15

List of Tables

Table 1 ………………………………………………………………………………………… pg 9

Table 2 ………………………………………………………………………………………… pg
20

1 Introduction

Inked is a book recommender system that addresses a growing need for quick, reliable, and user-friendly book recommendations among students and instructors. Traditional methods—such as emailing lists or relying on word of mouth—are often inefficient, lack personalization, and do not scale well as participation increases. This report details a design for a Recommender System to aid individuals in discovering, rating and managing their book recommendations. Ultimately, the system fosters a community-driven reading culture, where students, instructors, and other stakeholders can collectively benefit from shared knowledge and ongoing book discovery.

The proposed system will integrate an external API to retrieve book titles and cover images while relying on a separate database to store and manage user-related data. The client-side will use an intuitive Android interface, that will allow users to view, submit and search books and book recommendations. Administrators will be able to perform tasks such as adding, editing, or removing book entries. Creating a layered and modular system that ensures secure storage of user credentials and allows for seamless feature addition, as well as easy scalability for future developments.

Following the introduction, the report will discuss the project requirements, specific solution details and the final chosen architecture of the system. The report will then examine the environmental, societal, safety, and economic considerations, as well as present an overview of the team’s work distribution, meeting summaries, and proposed future enhancements.

2 Design Problem

Traditional methods of compiling recommended reading lists—such as email threads or scattered word-of-mouth suggestions—often lead to disorganized information, duplication of effort, and limited participation. In an environment where both students and instructors benefit from timely, relevant book recommendations, an online platform is needed to streamline the process of discovering, rating, and sharing book suggestions. Additionally, administrators require straightforward tools for managing the book database. The Book Recommender System addresses these needs by providing a centralized solution that merges real-time book information with secure user data handling.

2.1 Problem Definition

Design and implement a mobile Book Recommender System that allows users to search for books, submit and view recommendations, and update ranking information in real time. The system must include administrative capabilities for adding, editing, and removing books. By combining external APIs for up-to-date book details and a secure database for user data, the project will deliver an accessible, efficient, and scalable approach to book discovery.

2.2 Design Requirements

This section will outline the functional and non-functional requirements, as well as the objectives.

2.2.1 Functional Requirements

The Inked application is a mobile-based book recommendation system designed to facilitate seamless book discovery and rating. The functional requirements define the specific capabilities the system must provide to ensure its core functionality.

1. **Book Search & Filtering**
   - Users can search books by title, author, or genre.
   - Filtering options include rating, popularity, and latest additions.
2. **Book Recommendation & Rating System**
   - Users can rate books they have read.
   - Users can recommend books to other users on their friends list.
   - Users can preview books.
3. **Real-Time Dashboard**
   - Users can view their reading history and save books.
   - Displays a top 10 most recommended books list based on aggregate user ratings.
   - Updates automatically as new ratings are submitted.
4. **Book Management (Admin Only)**
   - Administrators can add, edit, and delete books from the system.
   - Each book entry includes a title, author, genre, description, cover image, and rating.
5. **User Profile & Activity Tracking**
6. **Notifications & Alerts**
   - Users receive alerts for new book additions and trending books.
7. **Groups and Friend Lists**
   - Users can search and add other users as friends.

- Users can create reading groups
- Reading groups maintain internal book recommendations.

8. **User Authentication & Role Management**
    - Users must be able to sign up and log in using an email and password.
    - Role-based access control:
        - Regular Users: Search for books, rate books, and receive recommendations.
        - Administrators: Add, edit, and remove books from the reading groups.

2.2.2 Objectives

The Inked application is designed to provide an interactive, community-driven book recommendation experience. It aims to achieve the following objectives:

1. **Enable Dynamic Book Discovery**
    - Integrate Google Books API or Open Books API for real-time book search.
    - Allow users to search for books by title, author, or genre, with filtering options
2. **Facilitate Social Book Recommendations**
    - Users can add friends within the app.
    - Books can be recommended directly to friends and shared within reading groups.
3. **Support Collaborative Reading Groups**
    - Users can create and join reading groups.
    - Each group will have its own book collection, managed by group admins.
    - Group members can rate, discuss, and recommend books to the group.
4. **Enhance User Experience with Personalization & Tracking**
    - Users can save books to their personal reading lists.
    - A reading history feature will allow users to track past reads and recommendations.
5. **Implement Role-Based Book Management**
    - Regular users can recommend books, but only reading group admins can add or remove books from their groups.
    - Users with admin roles in reading groups will have management rights over group book lists.
6. **Create a Seamless & Engaging User Experience**
    - The real-time dashboard will showcase saved books and search menu.
    - Users will receive notifications for book updates, friend recommendations, and group activity.
7. **Ensure Secure & Scalable User Authentication**
    - Users will be required to sign up and log in securely via email authentication.
    - The platform must support role-based access control (Regular Users vs. Reading Group Admins).
8. **Prioritize Performance, Scalability, and Security**
    - The system should efficiently handle multiple users and concurrent interactions without performance degradation.
    - All user data and recommendations should be securely encrypted.

2.2.3 Non-functional requirements and constraints

1. **Performance**
    - The application must meet standard mobile app performance expectations
    - Startup launch should take less than 2 seconds
    - The user interface should respond immediately (usually within 100–200 ms) to taps, swipes, and other interactions to feel fluid. Navigation between screens should be smooth, with minimal loading or blank screens.
    - performance-heavy tasks should be optimized or performed in the background.
    - Whenever possible, data transfers should be optimized and compressed to improve loading times and reduce data usage.
2. **Security & Data Integrity**
    - Implement user authentication and privacy mechanisms.
    - Ensure data protection against unauthorized modifications.
3. **Modifiability**
    - The system should be designed to allow easy integration of new UI components with minimal development effort.
    - Modules have compact simple tasks that they will be in charge of executing
4. **Compatibility**
    - The application must be compatible with Android devices running API level 19 (KitKat) or higher

3 Solution

3.1 Solution 1: Single app with local storage

In this approach, the recommender system is entirely self-contained within a mobile application. All data, including books, user ratings, and recommendation logic would be stored locally on the device. This means that users can read, rate, and review books offline without requiring network connectivity.

Limitations:

- Limits scalability as the number of books and user data grows, local storage can quickly become insufficient and cumbersome to manage.
- Provides no real time service. Without an external server or API, it is difficult to synchronize book entries, ratings, or updates across multiple devices.
- Users on different devices would maintain separate data silos, leading to data duplication and inconsistency.
- Deploying updates or new features would require users to manually install updated app versions, complicating version control and requiring users to execute large data downloads.

3.2 Solution 2: External Books API

The improved solution integrates an external books API (e.g., Google Books or Open Library) to pull data such as book titles, authors, and cover images in real-time. However, user data, ratings, and recommendations could still be stored locally or in a very minimal backend.

Limitations:

- If user ratings or recommendations are still stored locally, data is not shared across different devices and requires the user to manually synchronize data.
- Reliance on Third-Party Service creates a dependency on the external API's availability and performance.
- Without a robust server-side component, user authentication and real-time updates remain challenging.

3.3 Final Solution: Hybrid Model

The hybrid solution combines an external books API for real-time book data with a secure backend database for managing user accounts, ratings, and recommendations. Users can search for and fetch book details from the API, while their ratings, profiles, and friend lists are stored in a centralized database. This approach provides real-time synchronization, scalable data management, and up-to-date book information.

Table 1. Solution Comparison

	Local storage	External API	Hybrid (Final solution)

Maintainability	Difficult to maintain	easier to maintain	Easily maintainable and modular
Scalability	Limited potential to scale	more potential to scale	high potential to scale due to flexible modular design
Usability	Incredibly slow, clunky	would offload a lot of space from the local storage	Does not require manual updating.
Real-Time functionality	would require long updates to download to local storage	Would still require uploading and downloading updates	Real time updating with backend.

This solution was chosen because it combines the benefits of real-time book information from an external API with the reliability and security of having a central backend and database. By fetching dynamic content like book details and cover images from the API, the app automatically stays up-to-date without extra effort. User data, including profiles, ratings, and recommendations, is stored centrally, making sure everything stays in sync across multiple devices and avoiding duplicate data. This setup enables updates and new features to be rolled out easily from the server side, so users don't have to constantly reinstall or manually update the app. This solution creates a fluid user experience while fulfilling the objectives and securing the user's data.

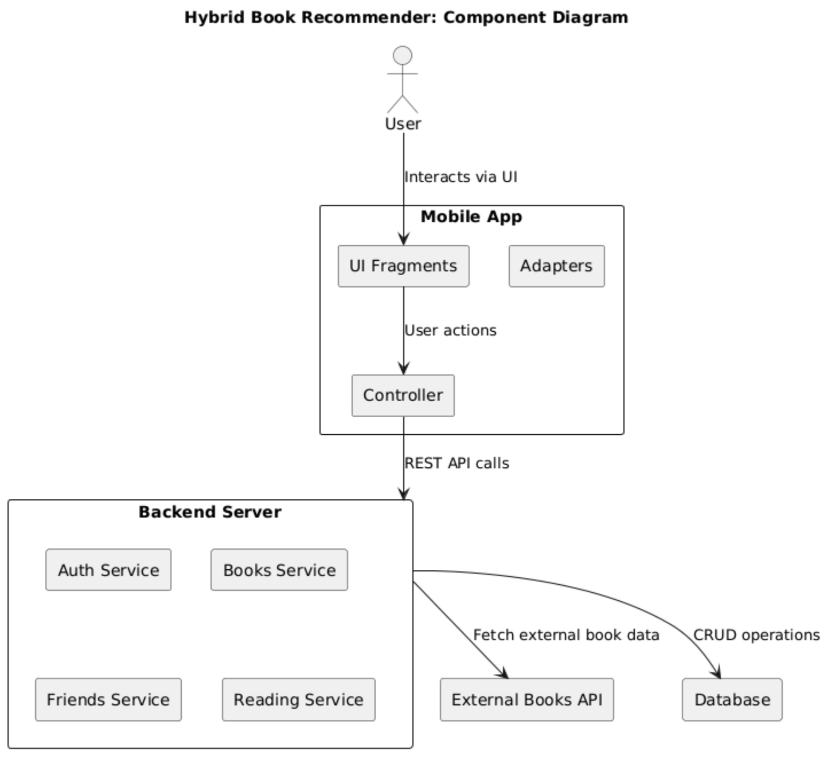

Figure 1. Final solution Component diagram

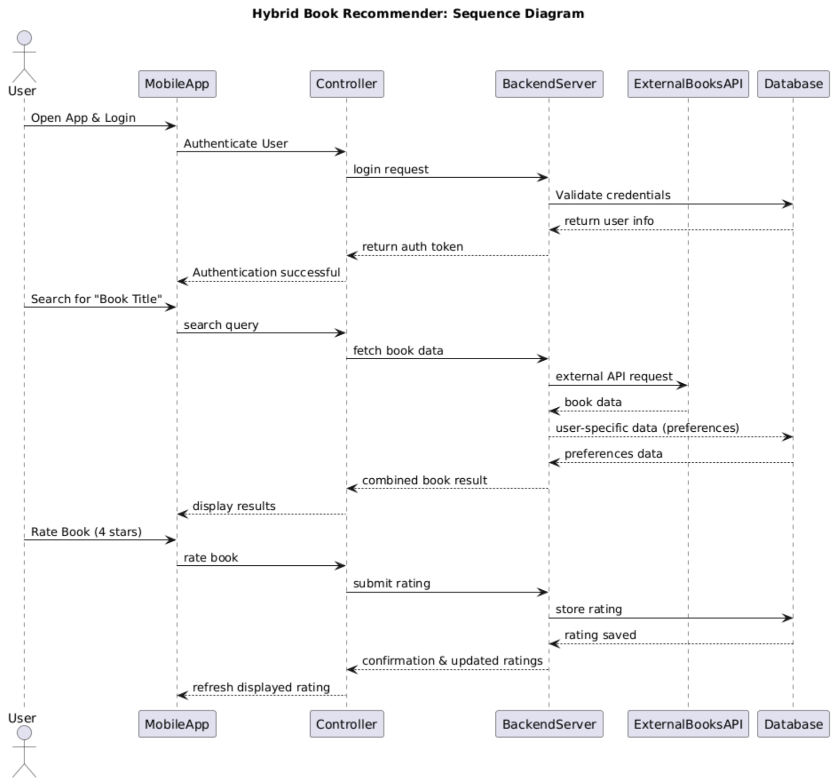

Figure 2. Final Solution Sequence Diagram

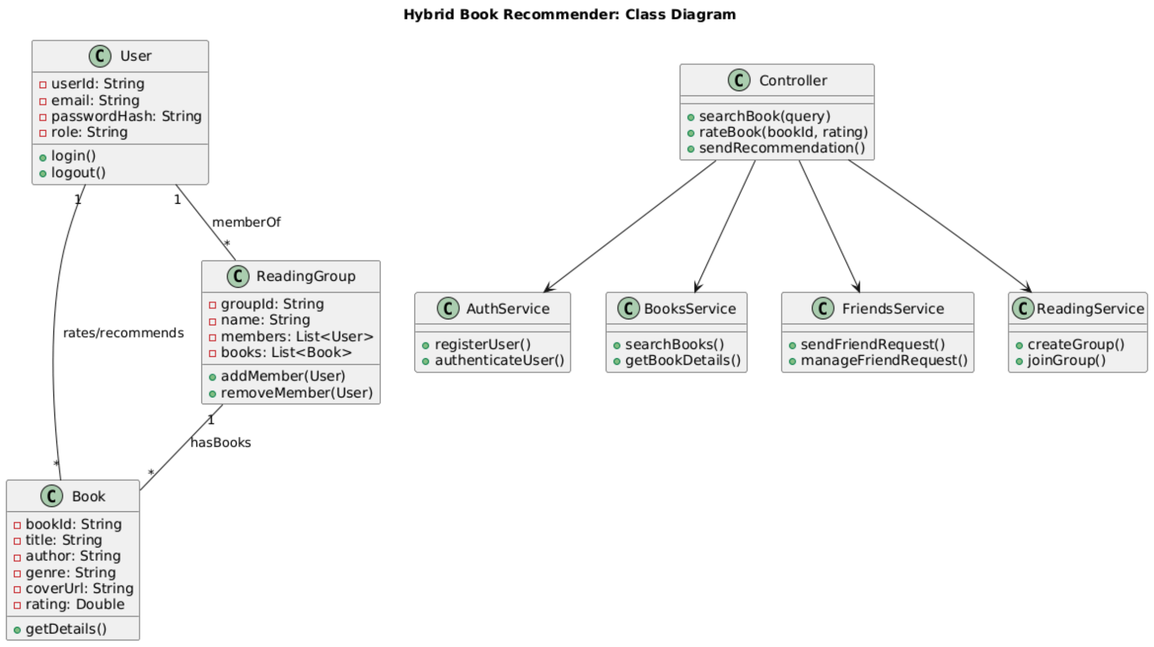

Figure 3. Final Solution Class Diagram

3.3.1 Features and the software architecture

Features:

- Real time book search: Users can search and filter books by title, author or genre using real time data from external APIs
- Book Recoomendations and Ratings: Users can rate books, and the system dynamically updates the ratings and recommendations
- Friends Lists and Reading groups: Enables social interactions through friend lists and group-based book sharing.
- Administrative control: Administrators can manage book databases, user roles, and content moderation.
- User Authentication: Secure sign-up, login, and session management via backend authentication services.
- Real Time Dashboard: Displays real-time top-rated books, recommendations, and user-specific data updates.

3.3.2 Component Interactions and Software Architecture

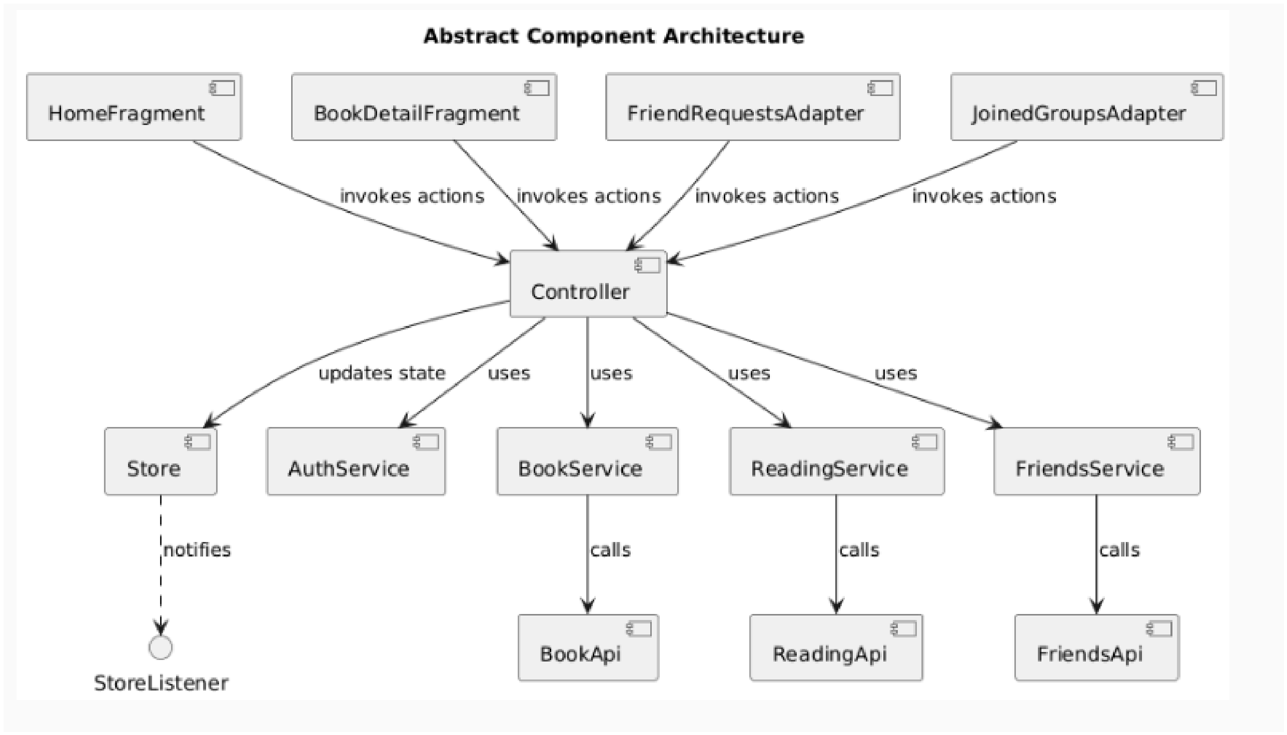

Figure 4: This diagram shows the high-level architecture where the Controller coordinates actions between the UI layer and various service components. The Store holds global state and notifies UI components via the StoreListener interface, while services interact with external APIs.

The application follows a modular design where key components communicate through clearly defined interfaces, ensuring separation of concerns and ease of maintenance. The architecture leverages both the singleton and observer patterns, as detailed below:

Singleton Pattern for Centralized Management

- Controller as a Singleton:
  The Controller (see *Controller.java*) is implemented as a singleton. This ensures a single point of coordination between the UI and backend services. It centralizes all user actions—such as searching for books, managing friends, and handling reading group interactions—by interfacing with various service classes (e.g., *BookService*, *ReadingService*, and *FriendsService*).
- Store as a Singleton:
  The Store (see *Store.java*) is also implemented as a singleton. It acts as the central repository for the application state, holding data such as user details, lists of books, friends, friend requests, and reading groups. By being a singleton, the Store guarantees that all parts of the application reference a consistent state.

Observer Pattern via the Store Listener Interface

- StoreListener Interface:
  UI components subscribe to state updates by implementing the StoreListener interface (see *StoreListener.java*). This design enables a decoupled communication channel where the Store notifies its listeners whenever the state is updated.
- Automatic UI Updates:
  After the Controller completes a service call (for instance, fetching a list of books or processing friend requests), it updates the Store with the new data. The Store then calls notifyListeners(), which iterates over all registered StoreListeners. This mechanism ensures that UI adapters and fragments (such as those in *HomeFragment.java* and *FriendRequestsAdapter.java*) refresh their displays to reflect the latest state.

Flow of Data and Interactions

- User Initiates an Action:
  UI components (e.g., buttons in fragments like *BookDetailFragment.java*) call methods on the Controller. These methods act as entry points to trigger specific actions such as searching for a book or sending a friend request.
- Controller Delegates to Services:
  The Controller, acting as a mediator, forwards these requests to the appropriate service classes. For example, when a book search is initiated, the Controller calls the searchBook() method of *BookService* (see *BookService.java*). Similarly, group operations are managed through *ReadingService* (see *ReadingService.java*).
- Services Return Data:
  After processing, the services return data asynchronously. The Controller then updates the Store with this new data.

- UI Refresh via Store Updates:
  As soon as the Store is updated, it calls notifyListeners(), which triggers the onStoreUpdated() method in all subscribed UI components. This update mechanism allows components like the *HomeFragment* to seamlessly refresh their data and provide immediate feedback to the user.

Benefits of the Architecture

4 Loose Coupling:
By isolating the UI from direct service calls, the Controller abstracts the complexity of API interactions. This promotes loose coupling and makes the system easier to test and maintain.

5 Consistency:
The singleton implementation of both the Controller and the Store ensures that there is a consistent application state throughout the app lifecycle.

6 Responsiveness:
The observer pattern (via StoreListener) enables real-time UI updates whenever the underlying data changes, resulting in a responsive user experience.

7 Scalability:
Modular design allows individual components to be scaled or replaced with minimal impact on the overall system, paving the way for future enhancements or feature additions.

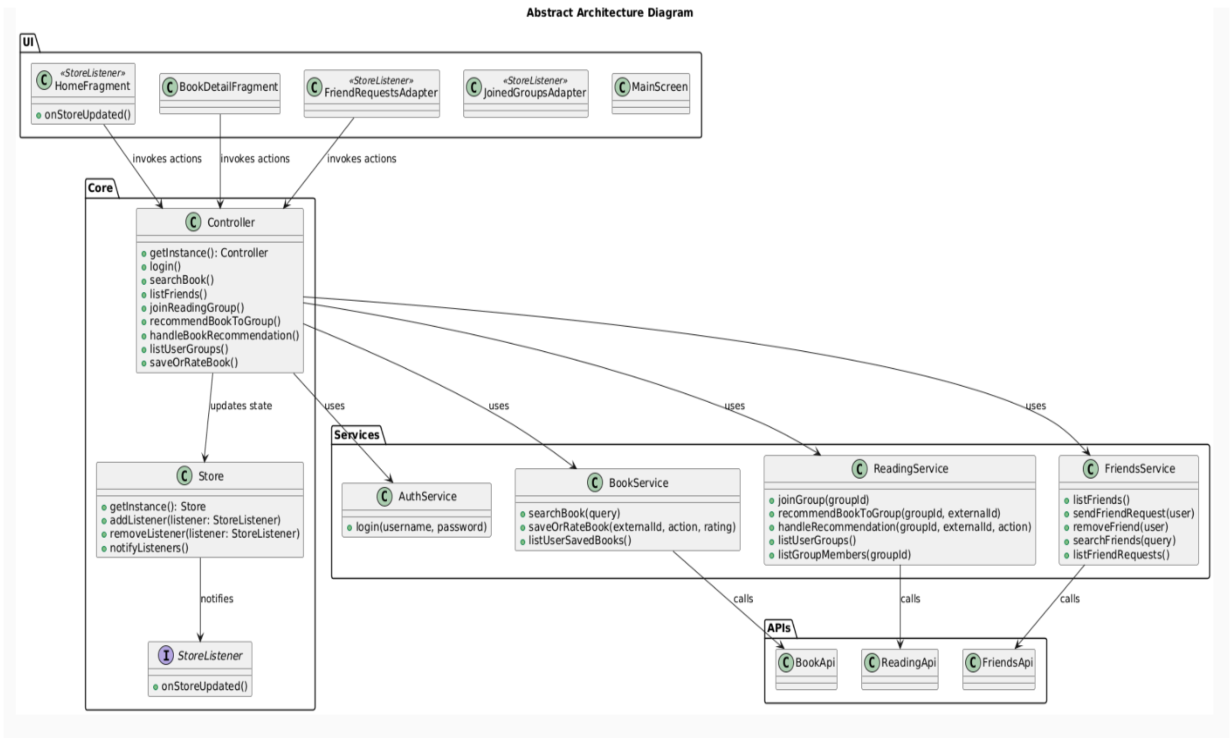

Figure 5: Abstract Architecture Diagram. This diagram illustrates the high-level structure of the application, highlighting the core components—Controller and Store—and their interactions with the UI and service layers. The Controller orchestrates actions by delegating to various services, which in turn communicate with external APIs. The Store manages global application state and notifies UI components through the StoreListener interface, ensuring dynamic updates.

7.1.1 The system interfaces

Backend Documentation

The backend of the Inked application is implemented as a modular Flask application using Blueprints to separate concerns. The system interfaces expose various RESTful endpoints to support user authentication, book management, friend relationships, and reading group operations. Below is a summary of the key components and endpoints:

Authentication (auth.py)

- Endpoint: /login (POST)
- Functionality:
    - Accepts a username and password.
    - Verifies the credentials against the PostgreSQL database.
    - Generates a JSON Web Token (JWT) on successful login.
- Security:
    - Utilizes JWT for session management and secure user authentication.
    - Token expiration is set using a configurable expiration delta.

Book Management (books.py)

- Endpoints:
    - /search_book (GET):
        - Uses an external Books API (e.g., Google Books) to search for books.
        - Parses and returns key book details such as title, authors, publisher, and cover image.
    - /book (POST/GET):
        - POST: Allows users to save a book to their list or rate a book.
        - GET: Lists saved books for the authenticated user by fetching additional book data from the external API.
- Integration:
    - Communicates with the external Books API using the requests library.
    - Handles both saving state changes (e.g., saving a book, updating ratings) and retrieving book details.

Configuration and Database (config.py and db.py)

- Configuration (config.py):
    - Manages application settings including database credentials, JWT secret, and external API keys/URLs.
- Database Access (db.py):
    - Implements helper functions (run_query, connect_to_db) to handle PostgreSQL database interactions.
    - Ensures that queries are executed safely with proper resource cleanup and error logging.

Friend Management (friends.py)

- Endpoints:
    - /friend/remove, /friend/search, /friend/list, /friend/requests, /friend/request,

/friend/request/send

- Functionality:
  - Supports operations to search for users, send friend requests, manage friend lists, and remove friendships.
- Authentication:
  - Each endpoint validates the user's JWT token to ensure secure access.

Reading Group Operations (reading.py)

- Endpoints:
  - /reading/group/join: Enables users to join reading groups.
  - /reading/group/recommend: Allows users to recommend books to their reading groups.
  - /reading/group/handle_recommendation: Permits group admins to approve or deny book recommendations.
  - Additional endpoints support creating groups, listing groups, promoting group members, and retrieving group members or recommendations.
- Role-based Access:
  - Certain operations (e.g., promoting a member or handling recommendations) require the user to have an admin role within the group.

General Features and Error Handling

- Error Handling:
  - Each endpoint incorporates error handling and logs errors using Flask's logging capabilities.
  - Responses include clear success flags and error messages to assist with client-side error handling.
- Security Practices:
  - The application uses JWT for authentication across endpoints.
  - SQL queries are executed via a centralized function that handles parameterized queries to prevent SQL injection.

Signal and Temporal Events (network/api , network/service directories)
This system handles two types of events

- Signal Events: Triggered by external components
  - (API.java, AuthApi.java, AuthService.java): When a user attempts to log in via the login endpoint, these classes work together to verify the provided credentials and return a JWT token for session management.
  - (BookApi.java, BookService.java) These classes handle all book-related actions, such as searching for books, managing book entries (e.g., saving or rating books), and processing related user requests.
  - (FriendsApi.java, FriendsService.java) These classes are triggered when a user sends a friend request, removes a friend, lists their friends, or searches for potential friends.

- ○ (ReadingApi.java, ReadingService.java) These classes manage triggers related to reading groups, including joining groups, recommending books, handling recommendation approvals or rejections, and listing groups or group members.
- Temporal Events: Time triggered events internally managed by the backend (This system mainly focuses on signal events, however there are some implicit aspects)
    - ○ (config.py) JWT token expiration: includes a time based security setting that automatically requires a re-authentication after a set duration.

The backend serves as the bridge between the client-side mobile application and the persistent data store. It manages real-time interactions and ensures data consistency across features like book recommendations, user profiles, friendships, and reading group memberships. The API endpoints are designed to be robust, secure, and modular to facilitate future enhancements and scalability.

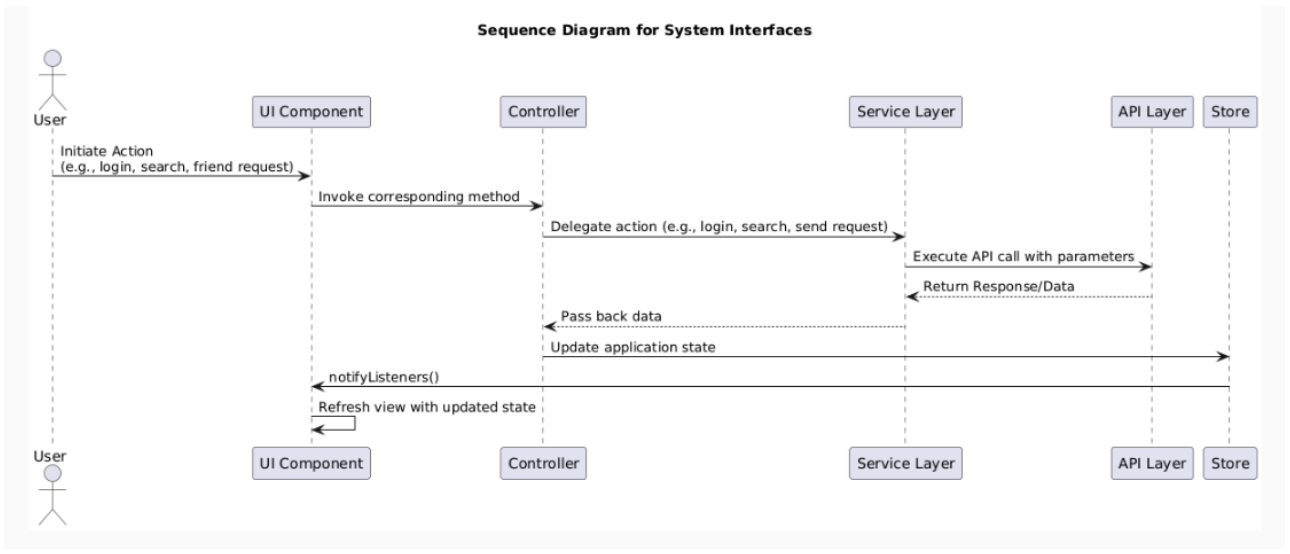
Sequence Diagram for System Interfaces
User
UI Component
Controller
Service Layer
API Layer
Store
Initiate Action (e.g., login, search, friend request)
Invoke corresponding method
Delegate action (e.g., login, search, send request)
Execute API call with parameters
Return Response/Data
Pass back data
Update application state
notifyListeners()
Refresh view with updated state

Figure 6: This sequence diagram illustrates the dynamic flow when a user initiates an action. The UI calls the Controller, which delegates to the Service Layer. The Service Layer performs both database operations and API calls as needed, then returns the result to the Controller. Finally, the Controller updates the Store, which notifies the UI to refresh its view.

7.1.2 The user interface design

User Interface Components and Their Roles

HomeFragment.java

- **Functionality:** Displays the real-time dashboard of top recommended books and recent activities.
- **Response:** The dashboard is refreshed with updated recommendations and

activity feeds.

- **Related Files:** HomeFragment.java, BookSearchAdapter.java, ReadingService.java.

Search Functionality (MainScreen.java / BookSearchAdapter.java)

- **Functionality:** Allows users to enter search queries and view matching books.
- **Business Events:**
  - When a user enters a query, an event is triggered that calls the BookService to query the backend using BookApi endpoints.
- **Response:** A BookResponse is returned, and the UI (RecyclerView) is updated to display the search results.
- **Related Files:** MainScreen.java, BookSearchAdapter.java, BookApi.java, BookService.java.

BookDetailFragment.java

- **Functionality:** Provides detailed information about a selected book and options to rate or recommend it.
- **Business Events:**
  - When a user selects a book, the fragment triggers a call (through BookService) to fetch detailed data.
  - When a user rates or recommends a book, the event is sent to update the rating or recommendation status.
- **Response:** The fragment updates the book details on-screen, reflecting the new rating or recommendation.
- **Related Files:** BookDetailFragment.java, BookService.java, BookApi.java.

RateFragment.java

- **Functionality:** Enables users to submit ratings for books.
- **Business Events:**
  - When a rating is submitted, the fragment sends this event to the backend through BookService.
- **Response:** A confirmation (e.g., BasicResponse) is received, and the displayed rating is updated accordingly.
- **Related Files:** RateFragment.java, BookService.java, BookApi.java.

Profile and Friend Management (MainActivity.java, FriendsAdapter.java, FriendRequestsAdapter.java)

- **Functionality:** Manages user profiles, friend lists, and friend requests.
- **Business Events:**
  - When a user sends, accepts, or removes a friend request, the corresponding service (FriendsService) is invoked.
- **Response:** The UI updates to reflect changes in the friend list or request status.
- **Related Files:** MainActivity.java, FriendsAdapter.java,

FriendRequestsAdapter.java, FriendsApi.java, FriendsService.java.

GroupsFragment.java

- **Functionality:** Supports reading group operations, such as joining groups, recommending books to groups, and managing group membership.
- **Business Events:**
    - Actions like joining a group or recommending a book trigger a call to ReadingService.
- **Response:** The system returns confirmations (e.g., BasicResponse, CreateGroupResponse) and updated group data, which is then displayed in the UI.
- **Related Files:** GroupsFragment.java, GroupBooksAdapter.java, GroupMembersAdapter.java, ReadingApi.java, ReadingService.java.

7.1.3 The requirements traceability matrix

Table 2. Traceability Matrix

Requirement	Design components	Components	Testing
User Authentication	Authentication module, Login UI view (Named MainActivity)	AuthService.java, AuthApi.java, LoginRequest.java, LoginResponse.java, MainActivity.java, activity_main.xml	Verify login with valid/invalid credentials; validate JWT token generation and expiration handling
Book Search and Filtering	Book Management UI (MainScreen)	BookApi.java, BookService.java, MainScreen.java, BookSearchAdapter.java	Enter various search queries and filtering options; confirm that the UI displays the correct, filtered book results.
Book Recommendation and Rating	Book Detail & Rate Fragments	BookDetailFragment.java, BookApi.java, BookService.java	Select a book, submit a rating, and recommend a book; verify that the updated rating/recommendation is reflected in the UI and persisted.
Real-time Dashboard	Home Fragment	HomeFragment.java, BookService.java, ReadingService.java	Simulate multiple rating submissions and verify that the dashboard updates in real time with the latest recommendations.
Performance Scalability, security	Global Application Architecture; Security & Database Access	config.py, db.py, AuthService.java, BookService.java, FriendsService.java, ReadingService.java	Run performance tests (e.g., app startup < 2 sec, UI response < 200 ms); simulate concurrent users; verify encryption and proper SQL

			query execution.
Friend management	Groups, friends tab	FriendsApi.java, FriendsService.java, FriendsAdapter.java, FriendRequestsAdapter.java, friendsTabFragment.java, fragment_requests_tab.xml, fragment_friends_tab.xml	Send and handle friend requests, search for friends, and check that the friend list is updated in real-time.
Reading Group functionalities	Groups Fragment; Reading Group UI Components	ReadingApi.java, ReadingService.java, GroupsFragment.java, GroupBooksAdapter.java, GroupMembersAdapter.java, fragment_groups_tab.xml	Create and join reading groups; recommend a book to a group and promote a member; verify that group data is correctly updated and displayed.

7.1.4 Environmental, Societal, Safety, and Economic Considerations

7.1.4.1 Environmental considerations

Our application is deployed using a low-footprint AWS EC2 micro instance, minimizing energy consumption while meeting performance needs. By hosting on a cloud provider rather than on-premises hardware, we leverage Amazon's energy-efficient infrastructure and scale down during low-demand periods. Additionally, network requests are routed through an API Gateway, which filters traffic and reduces unnecessary data processing, contributing to lower overall energy use. Whenever possible, we design our client-side code to cache book images and resources, limiting repeated network requests and further decreasing resource consumption.

7.1.4.2 Societal considerations

The Inked app fosters a collaborative reading culture by making book recommendations more accessible to a wide audience, including students and instructors. The interface aims for inclusivity by providing clear typography and contrasting color schemes (e.g., orange and white) to improve readability for users with mild visual impairments. Furthermore, by enabling reading groups and friend lists, the app promotes social interaction and peer-based learning, helping communities share knowledge and stay engaged in reading activities.

7.1.4.3 Safety considerations

Security and user safety are paramount in the Inked application. Additionally, we utilize AWS security groups so that only the API Gateway can communicate with the EC2 instance, and only the EC2 instance can access the RDS database. JWT-based authentication prevents unauthorized access to sensitive operations like friend management or group administration. We also implement robust validation and parameterized queries to mitigate risks of SQL injection and other common vulnerabilities.

7.1.4.4 Economic considerations

We operate within AWS's free or low-cost tiers (e.g., EC2 micro instance, RDS micro instance) to reduce hosting expenses, making the solution more financially accessible for small organizations or student communities. Our design prioritizes modularity, enabling teams to add or remove features without incurring large-scale refactoring costs. By only scaling resources as needed, we avoid overprovisioning and keep operational costs to a minimum. This approach balances functionality and affordability for long-term sustainability.

7.1.5 Limitations

Despite meeting the core requirements, the Inked application currently faces several limitations that may affect its scalability and user experience:

1. Dependency on External APIs:
   The application relies heavily on third-party services (e.g., Google Books API). Any downtime or performance issues with these external services can disrupt key features such as searching for books or retrieving book details.
2. Limited Book Previews and Images:
   While book metadata is fetched from the external API, cover images are not always available or consistent. Additionally, the current UI does not offer a rich preview experience, limiting the user's ability to assess a book before adding it to their list.
3. Primitive User Interface:
   The front-end interface is minimal and lacks advanced design elements. This may reduce the overall user experience, especially for users accustomed to highly polished mobile applications.
4. Basic Data Schema:
   The database schema stores only fundamental information (e.g., user credentials, book IDs, simple metadata), limiting the potential for advanced features such as detailed analytics, user reading histories, or personalized recommendation algorithms.
5. Flask Backend Not Production-Ready:
   The backend code, written in Flask, has not been optimized for high traffic or concurrency. Without asynchronous handling or load balancing, the system may experience slowdowns or bottlenecks under heavy use.
6. Limited Testing and Error Handling:
   While basic tests are in place, more comprehensive testing and error handling are needed to ensure robust performance in various real-world scenarios. Unhandled exceptions or partial validations may lead to application crashes or inconsistent

data.

7. No Offline Support:
   The application depends on a constant internet connection for retrieving book data and synchronizing user activities. This limits usage in areas with poor connectivity and may inconvenience users seeking offline reading recommendations.

4 Teamwork

4.1 Meeting 1

Time: February 28, 2025, 7:30pm

Agenda: Brainstorming session; determine project

roles, and initial app design

Team Member	Previous Task	Completion State	Next Task
Isaac Latta	N/A	N/A	- Implement book api(e.g. Open Books/Google Books) - Setup Database
Matia Landry	N/A	N/A	- Design and Implement login page

Outcome: Isaac will begin setting up the database. We have narrowed the database choices down to two options.

1. Firebase
   a. Simpler and directly usable within android studio
   b. Simple JSON storage

   c. Can hande email authentication directly
2. Amazon RDS with custom EC2 backend
   a. A Postgresql instance can be hosted using using AWS services (free micro instance)
   b. A separate AWS instance will manage JWT authentication and middle man the database with the app (free micro instance)
   c. The backend api will be written in Python using either Flask or FastAPI
   d. Will provide both hands on experience with AWS services and industry standard backend frameworks

First Isaac will set up the AWS instances and write the backend, if too complicated or deemed infeasible a more viable option such as Firebase will be chosen.
Matia will implement the front end, and non database dependent in-app features.

Both the MVVM and MVC design pattern seem like obvious choices for the app.

4.2 Meeting 2

Time: March 3, 2025, 12:00 pm

Agenda: Regroup on progress made, assign new tasks

Team Member	Previous Task	Completion State	Next Task
Isaac Latta	- Implement book api(e.g. Open Library/Google Books) - Setup Database	75% - Missing Book api implementation	- Implement book api(e.g. Open Library/Google Books) - Begin implementing basic database functions
Matia Landry	- Design and Implement login page	100%	- Design and Implement fragment switching

Isaac made the decision to use Amazon's AWS services. We have set up an aws api endpoint which forwards to our EC2 instance that runs the flask backend. The flask backend then runs the queries through an AWS RDS Postgres instance hosting the database. The core functionality of the database schema has been implemented(e.g. usernames, passwords, ids). Matia has implemented the login screen and successfully interfaced it with the controller class and will begin implementing the different views of the application

Next, Isaac will choose and implement the book api and Matia will implement the main

4.3 Meeting 3

Time: March 10, 2025, 12:00 pm

Agenda: Regroup on progress made, assign new tasks

Team Member	Previous Task	Completion State	Next Task
Isaac Latta	- Implement book api(e.g. Open Library/Google Books) - Begin implementing basic database functions	100%	- Refactor instrument tests to use the login functionality and JWT
Matia Landry	- Design and Implement fragment switching	100%	- Finish home page UI

Isaac has successfully implemented the database, the book api, and the in-app classes and methods for calling the backend api. The server scripts for testing the backend endpoints are also implemented and passed. The java instrument tests are mostly implemented, they need to be refactored however to provide valid authentication via the /login endpoint.

Matia has successfully implemented the menu bar for switching between the various fragments.

Isaac will then update the android instrument tests to use the login functionality, fully encapsulating the end-to-end usage.

Matia will finish the main screen, and the various sub screens.

4.4 Meeting 4

Time: March 18, 2025, 5:00 pm

Agenda: Regroup on progress made, assign final tasks

Team Member	Previous Task	Completion State	Next Task
Isaac Latta	- Refactor instrument tests to use the login functionality and JWT	100%	- Begin presentation, poster, and finalize report
Matia Landry	- Finish home page UI	100%	- Begin presentation, poster, and

				finalize report

Isaac has successfully refactored the instrument to test the login functionalities and encapsulate the end-to-end usage.

Matia has successfully implemented the fragment switching, users can now toggle through the search book, social, and saved books fragments. The social fragment is composed of 3 subfragments, including views for searching and viewing friends, view reading groups, and viewing friend requests.

5 Conclusion and Future Work

Inked successfully delivers a user-friendly mobile app designed to simplify how users discover, rate, and share book recommendations. By integrating real-time book data from external APIs with a secure backend database, the application provides a seamless, engaging experience. Core features such as book searching, personalized recommendations, friend and group management, and real-time dashboard updates have been effectively implemented, ensuring that the app meets all outlined objectives while satisfying performance, security, and scalability constraints. To conclude, Inked provides a cohesive, community-focused platform designed to enhance users' book discovery experiences.

Moving forward, the user interface could include a dark mode, a profile, and a posts section to enhance interactivity and polish the project further. Incorporating machine learning or AI to tailor recommendations based on user preference, history and analytics from other users would greatly improve the user experience and personalization. Strengthening the backend to handle more user traffic and bolster security and reliability. These improvements have the potential to make Inked an even more engaging and robust platform for students, teachers, and book enthusiasts alike.

6 References

Appendix