Deliverable #2: Library Management System

Github Repository Link: https://github.com/ridaabhatti/3354-team5

Delegation of Tasks

- Ridwan Galib - Implementation code, Test Plan
- Ridaa Bhatti - Cost, Effort, and Pricing Estimation
- Inaaya Rana - Project Scheduling, Conclusion
- Ryan Edward - Project Duration and Staffing
- Oluwadamilare Sunmola - Use Case Diagram, System Architecture and Project Implementation
- Zenidy Le - Class Diagram, compiled and edited presentation video
- Sameer Vashisth - Comparison of Similar Designs, System Architecture

Deliverable #1:
https://docs.google.com/document/d/1KNnXbCqpxsmEfz9YYZfPnzrkoJIFKsRepa2G2MCR14w/edit?usp=sharing

Project Scheduling

1. Requirements and planning - 3 weeks
    a. Collect and analyze requirements
    b. Stakeholder interviews, user surveys
    c. Define software requirements
2. System design - 3 weeks
    a. Use case and sequence diagrams for main operations
    b. Class diagram and data model
    c. Architectural design
3. Implementation (core features) - 4 weeks
    a. Backend: authentication, book catalog, search
    b. Frontend: login, search interface, book list
    c. Database setup and integration
    d. Unit testing
4. Implementation (advanced features) - 4 weeks
    a. Fines and payment integration
    b. Notification system
    c. Admin functions
    d. Integration with core modules
    e. Performance testing
5. System integration and testing - 2 weeks

a. Test interactions between modules
b. Validate requirements
c. Bug fixes and optimization
d. Final review

Start date: Jan 5, 2026
End date: April 27, 2026

*We will only count weekdays. Weekends will be optional overtime, if needed to reach deadlines.
*Working hours per day: 8

Cost, Effort, and Pricing Estimation

Method: Function Point (FP) Estimation Technique. FP is a method that measures the functional size of a system based on what it must do, its inputs, outputs, user interactions, data files and external interfaces. This method is most reliable for estimation, specifically for systems like ours with interactive features.

Implementation: We began by identifying the functional components within our system including user authentication, book searching, borrowing and return operations, fine payments, reporting and profile management. Each function was categorized into External Inputs, External Outputs, External Inquiries, Internal Logical Files, and External Interface Files. Each category is assigned a complexity weight which allows for calculation of Gross Function Points (GFP). Next, the non functional complexity factors including performance, security, data handling, and transaction flow were evaluated. These factors were used to produce a Processing Complexity Adjustment (PCA). The PCA is then applied to the GPF resulting in a final FP value. This value is converted using the industry average FP and results into development effort. The effort is multiplied by the standard development rate. Lastly, the project overheard costs including project management, testing and buffers are added. The final cost reflects an estimate of the total project cost.

Calculations

1. **Function Category Counts**

Type	Count	Weight (Standard)	Total
External Inputs	14	4	56
External Outputs	10	5	50
External Inquiries	6	4	24
Internal Logical Files	6	10	60
External Interface Files	3	5	15

Gross Function Points (GFP) = **205 FP**

2. **Processing Complexity (PC)**
Complexity Scoring: This system includes authentication, fine calculation, payment processing, session management, recommendation logic, reporting, and multi-step borrowing and returning workflows. This results in moderate to high complexity.
PC Score: 40

3. **PCA**
PCA = 0.65 + (0.01 × PC)
PCA = 0.65 + 0.40 = **1.05**

4. **Final FP**
FP = GFP × PCA
FP = 205 × 1.05 = **215 FP**

5. **FP to Effort and Cost**
Industry Averages:
1 FP ≈ 8 person-hours
Hourly labor rate ≈ $35/hour

Effort = 215 FP × 8 = **1720 person-hours**

Labor Cost = 1720 × $35 = **$60,200**

Standard Overhead:

15% project management
10% testing
10% contingency

Adjusted development cost = **$81,270**

Estimated Cost of Hardware Products

Hardware	**Purpose**	**Cost**
Cloud Server	Host backend, web app and database	$360
Backup Storage	Daily backups	$90
Developer/Testing Machines	Local testing/development	$1400

Estimated Cost of Software Products

Software	**Cost**
Payment Gateway	$120
Notification/Email API	$60
Reporting/Analytics	$100
Database	$2000
IDE	Free

Estimated Cost of Personnel

Item	**Cost**
Developer Labor	$81270
Staff Training	$450
Admin Training	$150

Maintenance Tech Onboarding	$300

Total Project Cost Estimate

Category	**Cost**
Hardware	$81270
Software	$2280
Personnel	$620

Final Estimated Total Cost: ≈ $84,000

Project Duration
1/5/2026 - 4/27/2026
81 days (Weekdays only, 8 hour/day)

Staffing
FTE(People)
1720 person-hours
Raw hours/person: 81 days * 8 hours = 648 hours
Assuming 80% of work is effective
Effective hours/person: 648 hours * .8 effort = 518 hours
People needed: 1720 hours / 518 hours = 3.3 average people, 4 people (round up)

FTE weeks
1 FTE week = 5 days * 8 hours * 0.8 effort = 32 hours
FTE weeks: 1,720 hours / 32 hours = 53.75 weeks

Roles and Responsibilities
- Project Lead
- Backend Developer
- Frontend Developer
- QA Tester
- Business Analyst
- Systems Architect

Phase	Weeks	People Needed(FTE)	Roles	Hours(FTE * Weeks * 32)
Requirements and Planning	3	3	Project Lead, Business Analyst, Backend, Frontend	288
System Design	3	2.5	Architect, Frontend, Backend	240
Implementation	4	4	Backend, Frontend, QA, project lead	512
Advanced Implementation	4	4	Backend, Frontend, QA, Project Lead	512
Integration and System Tests	2	2.5	QA, Frontend, Backend	160
Total	16	n/a	n/a	1712

.5 means half time
1720 - 1712 = 8 hours left = one day for integration / buffer

Test Plan
The main purpose of this test plan is to ensure the correctness and reliability of the Loan class, specifically the compute_fine() method, which calculates overdue fines based on the borrow date, return date, daily fine rate, grace period and maximum allowable fine. Fine calculation affects user balances, borrowing permissions, and payment workflows. It is very important that this unit behaves consistently under all normal and edge case conditions. The primary objectives for this method include: returning a zero dollar fine for on time returns, correctly calculating fines for overdue returns, capping the fine at the system's max limit, and handling edge case limits such as invalid return date or extremely large overdue periods.
We used python's built in unittest module.

Code For Unit Testing and Automated Test Tool Output

```
from datetime import datetime, timedelta
class Loan:
    DAILY_FINE = 0.50
    GRACE_DAYS = 2
    MAX_FINE = 20.00

    def __init__(self, loan_id, user_id, book, borrow_date=None):
        self.loan_id = loan_id
        self.user_id = user_id
        self.book = book
        self.borrow_date = borrow_date or datetime.now()
        self.return_date = None

    def mark_returned(self):
        self.return_date = datetime.now()
        self.book.mark_as_returned()

    def compute_fine(self):
        due_date = self.borrow_date + timedelta(days=14 + self.GRACE_DAYS)
        if not self.return_date or self.return_date <= due_date:
            return 0.0
        overdue_days = (self.return_date - due_date).days
        return min(overdue_days * self.DAILY_FINE, self.MAX_FINE)
```

Automated Unit Tests Using Unittest:

```
import unittest
from datetime import timedelta

class TestLoan(unittest.TestCase):

    def test_fine_no_overdue(self):
        loan = Loan(1, 10, Book(1, "1984", "Orwell", "123", "Fiction"))
        loan.return_date = loan.borrow_date + timedelta(days=10)
        self.assertEqual(loan.compute_fine(), 0.0)

    def test_fine_overdue(self):
        loan = Loan(1, 10, Book(1, "1984", "Orwell", "123", "Fiction"))
        loan.return_date = loan.borrow_date + timedelta(days=30)
        self.assertGreater(loan.compute_fine(), 0.0)
```

```
def test_fine_max_cap(self):
    loan = Loan(1, 10, Book(1, "1984", "Orwell", "123", "Fiction"))
    loan.return_date = loan.borrow_date + timedelta(days=300)
    self.assertEqual(loan.compute_fine(), Loan.MAX_FINE)
```

Defined Test Cases and Expected Results

Test Case	Description	Input Conditions	Expected Results
No Overdue	Return early or on time	Return_date = borrow_date + 10 days	Fine = 0.0
Moderate Overdue	Return significantly late	Return_date = borrow_date + 30 days	Fine = positive value
Extreme Overdue	Return months late	Return_date = borrow_date + 300 days	Fine = MAX_FINE(20.00)
Invalid Return Date	Return date before borrow date	Return_date < borrow_date	System returns 0 flags invalid
Fine Cap Enforcement	Extremely large overdue	Very large overdue days	Fine remains 20.00
Missing Book Object	Loan created without book	Book = none	Should error if mark_returned() called
Double Return	Book marked returned twice	Call mark_returned() twice	Should not corrupt availability state

Comparison with Similar Designs

There are Project domains that are similar to ours, like Koha, which is an open-source library system that provides modules for cataloging, patron management, fines, and reporting. Their architecture also relies on client-server, where it handles business logic and database operations. The difference between our project and Koha is that our solution emphasizes a simplified and modern REST-based service layer, allowing easier integration with third-party notification and payment systems.

System Architecture
The system works by connecting the user or librarian client to a central application server that manages all operations such as borrowing, returning, searching, and updating library resources. When a user interacts with the app, the request first goes to the main server, which checks credentials through the authentication service and then processes the action. If the request involves data updates or lookups, the server communicates with the library database to retrieve or modify records like user accounts, books, and activity logs. For tasks that do not need immediate responses, the server uses an asynchronous request queue to keep the system fast and responsive. At the same time, the notification service sends reminders, alerts, and system updates to users and librarians as needed. The admin dashboard also uses the same server to manage inventory, check reports, and monitor system performance. All components work together so clients can access library functions smoothly, securely, and in real time.

Conclusion
Throughout the semester, our group has made many adjustments to our project. These changes can be reflected through the task distribution and the development timeline. There were times that we underestimated how much time or effort it would take to do certain tasks, and this led to us needing to shift responsibilities in order to meet deadlines. Another change we made was switching the architectural design from Layered to Client-Server. This enabled us to centralize management and security, as well as make our project scalable. Every change we made was necessary to ensure that we met our own project standards, as well as the requirements outlined in the project rubric. Overall, our group adapted to the needs of the project and our fellow team members in order to guarantee a successful project.

References

[1] “Main Navigation Menu Homenewsabouthistoryproject Organizationhorowhenua Library Trust Koha Committee Rulespolicytrademark Usage Policycomments Policykoha Community Code of Conductrelease Scheduleworldwidecalendardemodocumentationdownload Kohaget Involvedchatenhancing Kohafor Developersfor LibrariansFor Library Userskohaconkohacon25kohacon24kohacon23kohacon22kohacon21kohacon20kohacon19kohacon18kohacon17kohacon16kohacon15kohacon14schedulevenuesupportcommunity Supportkoha Mailing ListsWeb Forumspaid Supporthow to Get Listedreport a Problem.” *Official Website of Koha Library Software*, 30 Oct. 2025, koha-community.org/.