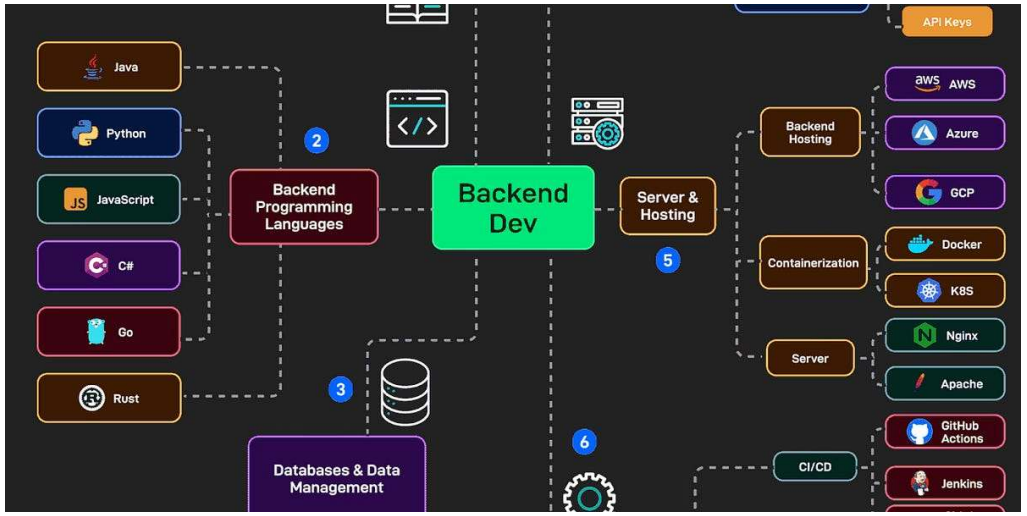
CMP5387
Backend Web Services and Database Engineering

D2 Planning and Design Document

Group Members		
Student Name	**Student ID**	**Student Email**
Rajveer Singh Saini	24159567	rajveer.saini@mail.bcu.ac.uk
Vikramjeet Singh	24157603	vikramjeet.-3@mail.bcu.ac.uk



Contents

Table of Figures

Table of Tables

Table of Code Snippets

Software Requirements

Summary of the target problem

Me and my group members are attempting a : Approved Project Idea Pitch (MODSWAP:- A course/Module Exchange System)

Project Idea type	***Y/N (Select One)***
Default Project Idea provided via Moodle	*N*
Approved Project Idea Pitch	*Y*

Table 1: Project type identification table

Problem Statement (Elevator Pitch): University students want fluidity in their education; modules get in the way. University students are limited to maximize their potential to diversify and learn cross-disciplines due to silos, modules, paper-heavy requirements, and an inability to see modules available in other departments and located elsewhere on campus. A Student in Computer Science may want to take a module on Fashion or Music, but they will never do so due to the red tape involved, meaning learning opportunities exist where educated personal growth could have happened but did not.

The Proposed Intervention: The Module Exchange System (ModSwap) **ModSwap** is a fluid, integrative, digitized university module platform that enables university modules to be accessible to all with transparent, democratized access. Frequently, students learn through email strings what others attempted to take but were denied—and others who were granted the ability to take it due to good standing—but to what end? Thus, equity exists in all classrooms. ModSwap reduces those email threads and provides a fluid, digitized, centralized marketplace where students have modules to give (drop) and modules to gain (pick up). Through student interest and necessity, student participation is funneled for the give and take with ultimate submission to Academic Tutors for approval.

Identification of target stakeholders and users

User/Stakeholder	Description	Problem(s)	Potential Benefit(s)
Students	Enrolled university students seeking to customize their curriculum.	**P1:** Inability to easily discover modules outside their major. **P2:** Cumbersome manual process to request swaps.	**B1:** Empowerment to tailor education to personal interests. **B2:** Instant visibility of available swaps.
Academic Tutors	Faculty members responsible for approving curriculum changes.	**P3:** Inefficient management of requests via email. **P4:** Lack of data on student demand for specific modules.	**B3:** Streamlined dashboard for approvals. **B4:** Centralized audit trail of all student changes.
University Administration	Staff managing student records and compliance.	**P5:** Difficulty ensuring credit compliance during manual swaps.	**B5:** Automated validation ensures degree rules are respected.
Course Coordinators	Staff responsible for monitoring academic progress and ensuring module balance within programs.	**P6:** Hard to Track and Validate exchanged modules to maintain academic credit requirements.	**B6:** Automated validation tools help ensure module exchange comply with degree requirements and maintain balance in student's study loads.

Table 2: Stakeholder identification table

Stakeholder Assessment and Justification: **Students** drive the system at the most significant level. If the "marketplace" is not user-generated by students wanting to exchange modules, there is no need for the system. It does nothing if students do not help one another but instead not using one another's resources. Thus, the more user-friendly UI is. **Academic Tutors** are at the highest powerful level, as 'gatekeepers'. If the system fails to address **P3** concerns (inefficiency), it will not be adopted. If it requires more effort than necessary, it will not be adopted. Thus, successful implementation hinges upon **F4**.

Figure 1: A Power Interest Matrix for Stakeholders

Quadrant Breakdown:

1. **Keep Satisfied (High Power, Low Interest)**

- University Administration
    - Higher power: They can deny the system from being established
    - Low interest: They need to be compliant, but they're not using it, running it
    - If you can keep compliant, you've succeeded with them which is why **F5** is tangential to success from their perspective.

2. **Manage Closely (High Power, High Interest)**

- Academic Tutors
    - Higher power: They can personally approval/deny the exchanges at the required levels.
    - Higher interest: They're deeply involved from a time-saving perspective **P3, P4**
    - Thus successful implementation comes from a user-friendly dashboard and trained support at launch.

3. **Keep Informed (Low Power, High Interest)****

- Students
    - Lower power independently: They need approval processes to get modules.

- High interest: They're the end-users of the system designed for their benefit (B1, B2).
- Thus they need to be kept in the loop consistently with a user-friendly interface so they feel comfortable with **F8** feedback and **F2** suggestion.

4. Monitor (Low Power, Low Interest)

- Course Coordinators
  - They have low power—suggestive and validation role versus an approval role.
  - They have low interest—they oversee but aren't day-to-day transactional.
  - Thus you can give them months reports and **F6** which is a validation report in support of their role.

Key Findings:

Tutors are key because they only have power + day-to-day use against them (F4 is "Must Have"). Students are the ones producing the need for exchange—and without tutor buy in—they must have great UX as a complementary (but not collaborative) force. Administration needs to feel secure in compliance—manual validation **(F5, F6)** facilitates that. Course Coordinators receive **F8**, a non-invasive approach to check in on their efforts.

Identification of feature set

Discussion and Justification

Relative to the feature set of our system, we used the **MoSCoW method** (Must Have/Should Have/Could Have/Won't Have) to ensure that the value proposition can be met within the development timeframe.

Minimum Viable Product (Must Haves) The "Must Have" attributes (**F1–F4**) represent the critical path to solving the primary curricular rigidity issue (**P1**) and administrative inefficiency issues (**P3**) for academic tutors and students.

- **F3 (Module Exchange Request System)** is the crux of the system; without logic linking a student wanting to give a module with one who wants a module, it cannot be a marketplace.
- **F2 (Module Browsing)** is critical to solving **P1**; without students perceiving the potential availability outside their program, they can't even get to an exchange.
- **F4 (Tutor Dashboard)** is critical for stakeholder buy-in; as acknowledged in **P3**, the top pain point comes down to manual emails back and forth; if it's easier to approve in-system than learn an entire new process, tutors will deny it.

(Should/Could Have) Attributes like **F7 (Notifications)** and **F5 (Admin Panel)** are considered "Should Have" attributes; although they're highly critical with the flow "students should know what's going on," it isn't a dire cause; they can look back manually. Therefore these will only be developed after secure exchange logic is established and robustly learned.

ID	Feature	Description	Problem it helps solve	Users/Stakeholders involved	Importance
F1	Student Login & Profile Management / Student Auth & Profile	Allows students to securely log in via university email, manage their profiles, view enrolled modules, and track exchange history.	P1, P2	Students	Must Have
F2	Module Browsing & Search / Module Marketplace	Enables students to browse and search for available modules across departments based on interests (e.g., sports, music, fashion) with filters (Dept, Credits).	P1	Students	Must Have
F3	Module Exchange Request System / Swap Request Logic	Students can request a module exchange by selecting desired modules and submitting for tutor approval. Core matching engine links "Giving" vs "Wanting" modules.	P1, P2	Students, Tutors	Must Have
F4	Tutor Approval Dashboard	Provides tutors with a dashboard to review, approve, or reject exchange requests, with notes or feedback.	P3, P4	Tutors	Must Have
F5	Administrative Management Panel / Admin Moderation	Allows admin staff to manage module data, update records, maintain compliance with academic policies, and oversee system health with force-swap capability.	P5	Administration, Admin	Should Have
F6	Real-Time Chat	SocketIO-based messaging between students negotiating a swap.	P2	Students	Should Have

F7	Course Coordinator Validation System	Automatically checks if module exchanges meet credit and program balance requirements.	P6	Course Coordinators	Should Have
F8	Notification & Status Updates / Notification System	Sends real-time notifications (email and in-app) to students and tutors about exchange status (approved, pending, rejected).	P2, P3	Students, Tutors, All	Should Have
F9	Analytics & Reports / Analytics Reporting	Generates reports for administrators and coordinators to track exchange trends, module popularity, and exportable data on most requested/dropped modules.	P5, P6, P4	Administration, Coordinators, Admin	Could Have
F10	Feedback & Support System	Allows users to report issues or provide feedback about the module exchange process.	P3	Students, Tutors	Could Have
F11	Mobile Interface Support	Optimizes the system for use on mobile devices for convenience.	P2	Students	Won't Have (Initial Release)

Table 3: Feature identification table

Planning and Design

Work Division and timeline

The project continues in a **Feature-Driven Development** style for both group contributors to be exposed at all levels (Database, Backend, Frontend) according to module learning objectives. Group Contributors:

- **Rajveer Singh Saini:** Core Logic (Swaps) and Authentication Lead.
  Contributions: User and SwapRequest schemas in models.py, auth blueprint (Login/Register, etc.), and the matching algorithm in the swaps blueprint.
- **Vikramjeet Singh:** Real-Time Features (Chat) and Admin Dashboard Lead.
  Contributions: Flask-SocketIO implementation on the chat blueprint, admin blueprint (Approval for Tutors at Final Step; Tutor side Module approval) and Module database seed data.
- **Shared Contributions**: Frontend (Jinja2 templates), styling via Tailwind CSS, and general Project Documentation.

Project Schedule (Gantt Chart Reference)

1. **Weeks 1-3:** (Foundation). Apply application factory structure with blueprints, models.py definitions (User, Module, SwapRequest) and Auth implementation.
2. **Weeks 4-6:** (Core Features). Module Browsing (F2) and Swap Creation Logic (F3). This will be the most intensive as it overlaps with M2M relationships.
3. **Weeks 7-9:** (Intermediate Features). SocketIO implementation for Chat (F5) and Tutor Dashboard creation (F4).
4. **Weeks 10-12:** (Finalization). UI Improvement, Unit Testing and Video Presentation.

Planning and Design: Client side

Discussion:-

Client-Side Architecture : The architecture we decided upon was **Server-Side Rendering**, with **Jinja2** templates rendered on the backend and served essentially **by Flask**. That way everything would be in a consolidated environment with necessary security on the backend (Flask-Login) and rendered more efficiently than the need for separated Single Page Applications. Therefore, to create a more modern/responsive user experience without page reloads, we **used HTMX** for content insertion (live search) and for a **lightweight** client-side experience, we used **Alpine.js**. For styling, we use **Tailwind CSS** for its utility-first framework which is attractive for rapid development and consistent presentation.

User Experience (UX) Target We intend to communicate "Cognitive Ease" in order to combat the risky manual process of swaps (Problem P2).

Marketplace Framing: Modules function like products in a store with an initial catalog so students search by department or credits without feeling pressure.

Split-screen Concept: The Swap Creation wizard spatially separates "Giving" (Top/Left) and "Wanting" (Bottom/Right), in an organized fashion to prevent user errors since transactions are like actual sales.

Key Screens for Your Consideration

Student Dashboard: An "Easy to Use" screen like a "Mission Control" whereby all exposed status cards ("Active Swaps", "Pending Approvals") and a "Wishlist Preview" automatically populates for students to have a sense of where they stand at all times.

Swap Creation Wizard: A progressive disclosure option where students are asked first to give (from their enrolled offerings) and then a live search bar (enabled by HTMX) will allow for consideration of what they want.

Admin Dashboard: An attractive design is key (Problem P3) for functionality, therefore, admins benefit from a more densely populated card-based approach with "Approve/Reject" actions essentially as buttons on the cards themselves, as well as filtering options (status/priority) to quickly rid backlogs for tutors.

WIREFRAMES:- (Made Progress till the MUST HAVE Features)

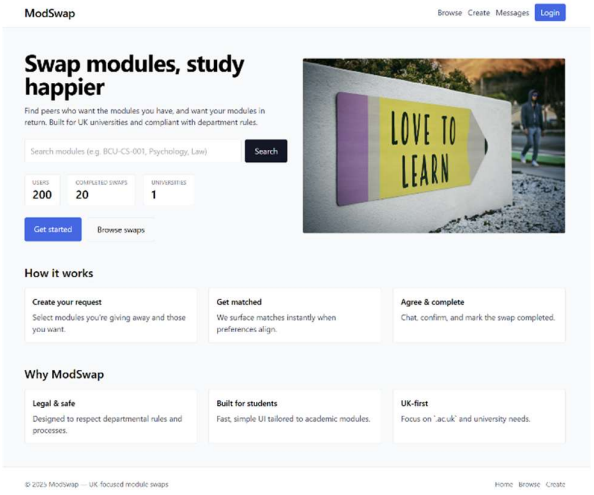


Figure 2: Home Page Interface

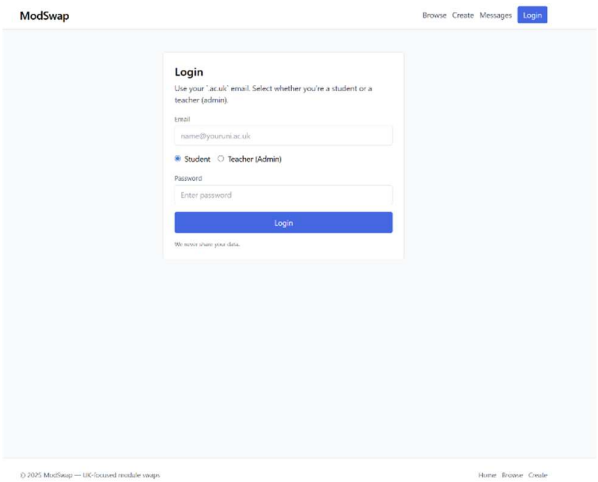


Figure 3: Login Interface

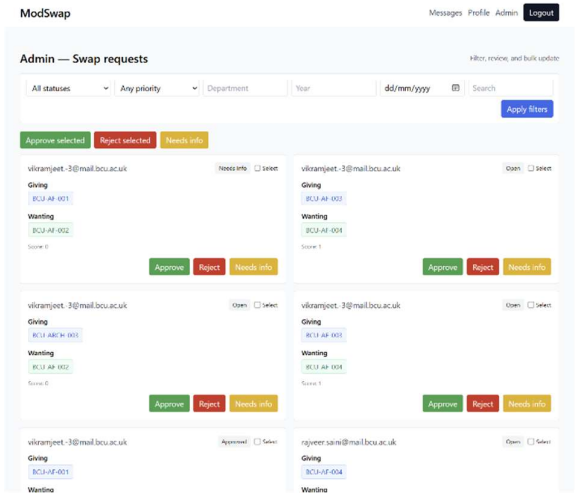


Figure 4: Admin Login Interface

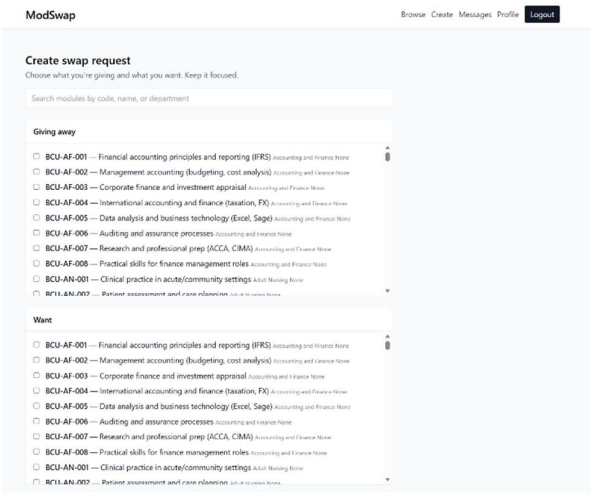


Figure 5: Swap Request in Student Login Interface

Planning and Design: Database

1. Conceptual Design

Commentary on Conceptual Design The conceptual design portrays the entities and connections that must be present at a broader level to accommodate the ModSwap functional set. The system requirements established through research suggest seven entities that operate as primary actors/components.

- **User** - The two user groups are students (Rajveer, Vikram) and tutors. Thus, this profile contains more specific data (degree, interests, campus, preferred_timeslots).
- **Module** - The item/swap in question. This will be limited to the specific university and year.

- **Swap Request** - As this is what drives the transaction in a user-driven space, it will be the method by which Giving vs. Wanting is done (Wanting is more complicated in and of itself, easier to derive from this entity).
- **Notification** - The notifications will exist in the system with timestamps on user-designated persons.
- **Document** - This is the title of a document (with metadata) uploaded to verify a user's identity in the system (i.e., transcript).
- **Message** - An internal message between two users stemming from a single swap request.
- **Rating** - Upon the completion of a Swap, users can rate each other based on their experiences.

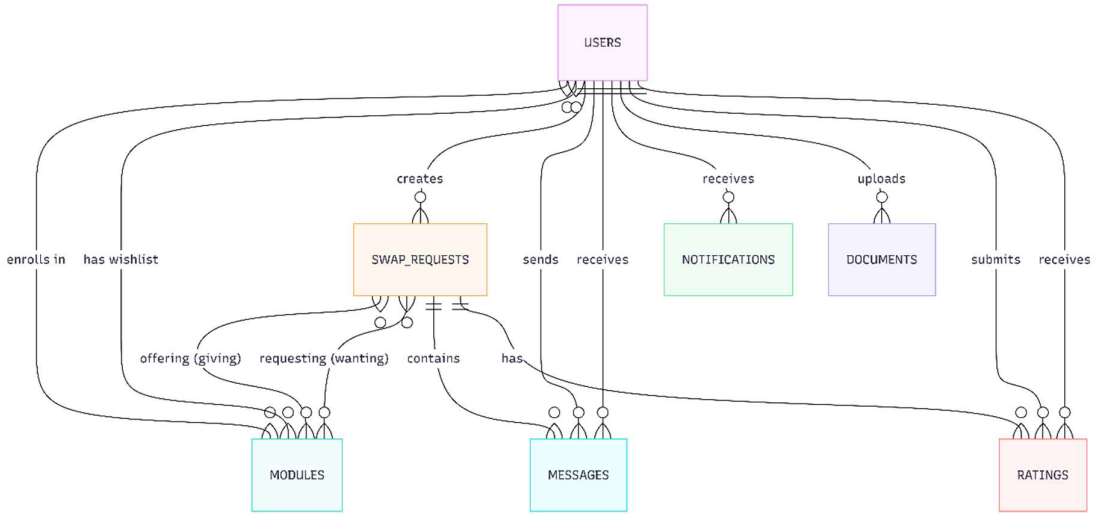


Figure 6: Conceptual Design : ERD Diagram

Key Relationships:

- **Complex Many-to-Many (Enrollment & Wishlist):** Users have a direct M2M relationship with Modules they are enrolled in (enrolls_via) and Modules they desire (wishlist_via).
- **The Swap "Give/Want" Dynamic:** A unique design choice is splitting the Swap Request's relationship to Modules into two distinct paths: giving and wanting. This allows a single request to handle complex trade scenarios (e.g., giving 2 modules to get 1).

---

2. Logical Design

Logical Design : Next, the logical design is supposed to create a normalized relational schema (**3NF**) from the conceptual model.

M2M Relationship Treatment: There are four Association Tables provided as a result of the M2M relationships found in the ERD which allow for effective queries and integrity.

1. USER_MODULES joining USERS and MODULES--Current enrollment.
2. USER_WISHLIST joining USERS and MODULES--Future Wish.
3. SWAP_GIVE_MODULES joining SWAP_REQUESTS and MODULES--(The give).
4. SWAP_WANT_MODULES joining SWAP_REQUESTS and MODULES--(The want).

Meaningful Fields & Constraints:

• Users are defined by verified_ac_email (Boolean) and role (to identify Student Rajveer or Vikram or Admin/Tutors).

• Swap requests are defined by priority and status (to satisfy Tutor Dashboard programming).

• Foreign Keys are established for referential integrity. MESSAGES join SWAP_REQUESTS (swap_id) and USERS (2x, sender_id & receiver_id) as unique identifiers for the ongoing thread of a conversation.

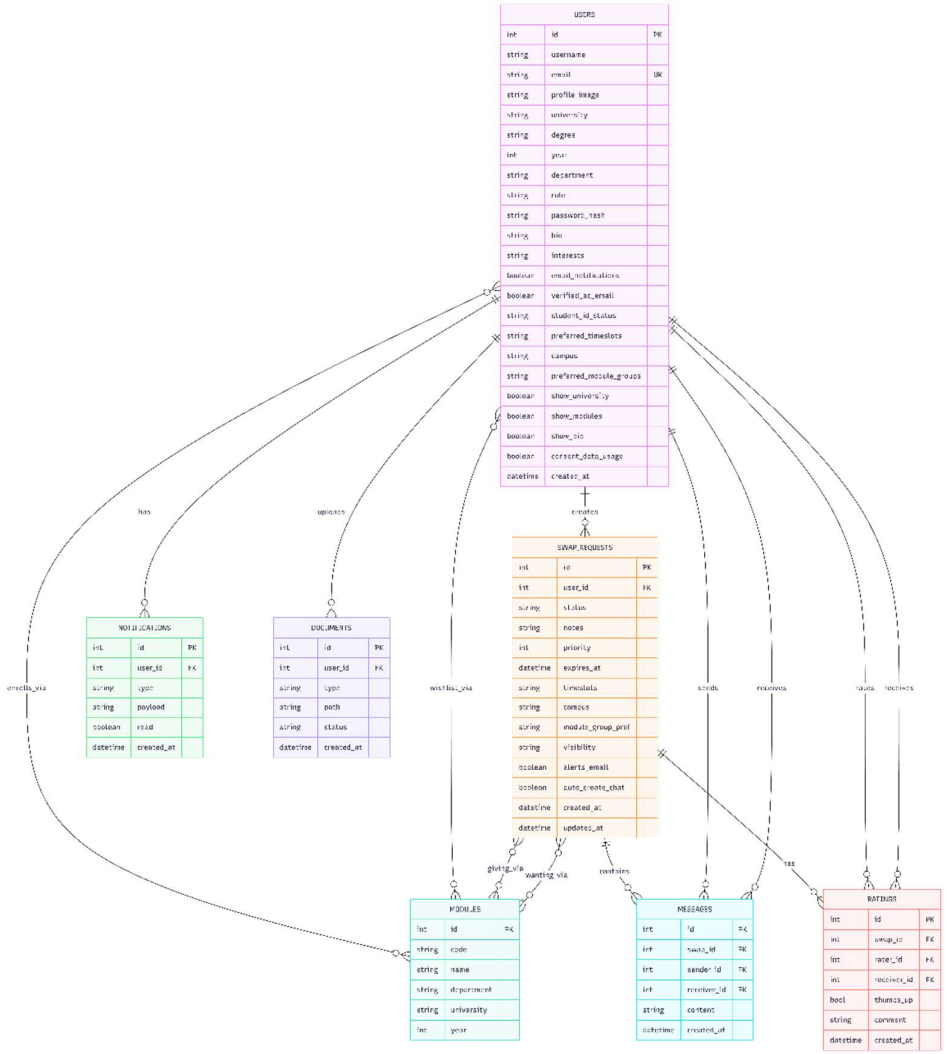


Figure 7: Logical Design : ERD Diagram

3. Physical Design

Physical Design Discussion The physical design defines the implementation details for the database. We will use **SQLite** for the development prototype (compatible with the Flask-SQLAlchemy stack).

- **Data Types:**
    - INTEGER: Used for all Primary Keys (id) and Foreign Keys (e.g., user_id, swap_id).
    - STRING / VARCHAR: Used for short text like username, email, and code.
    - TEXT: Used for longer content like bio, notes, payload (notifications), and content (messages).
    - BOOLEAN: Used for flags like verified_ac_email, read (notifications), and thumbs_up (ratings).
    - DATETIME: Used for auditing (created_at , updated_at, expires_at).
- **Indexing:** Indices will be created on email (User) and code (Module) to speed up search and login performance.

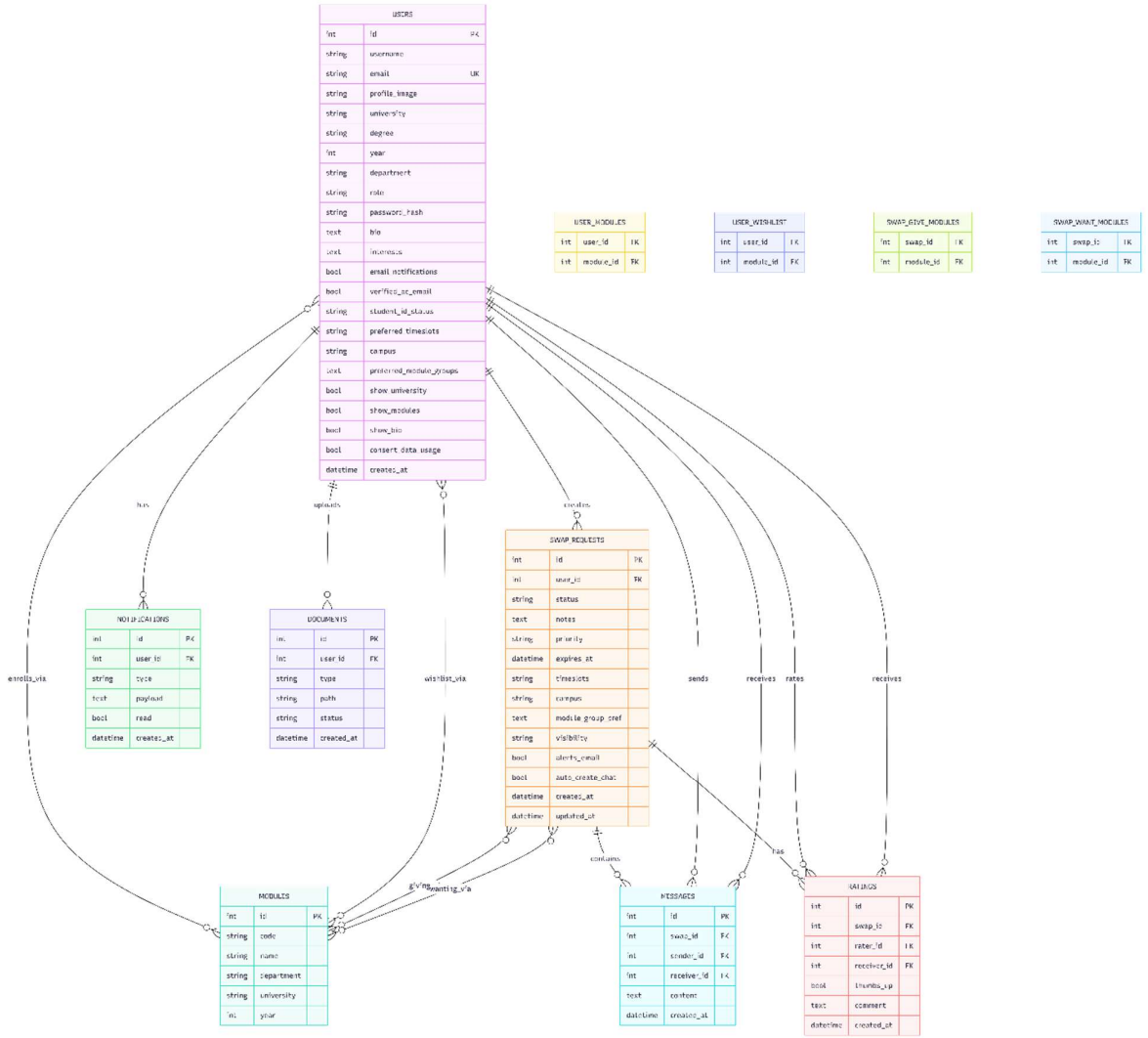


Figure 8: Physical Design : ERD Diagram Final

4. Physical Prototyping

This section provides the SQL scripts required to instantiate the schema shown in the ERD diagram and populate it with the specific group members.

A. DDL Script (Creating the Tables) SQL

```
-- 1. USERS Table

CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username VARCHAR(100) NOT NULL,
    email VARCHAR(120) UNIQUE NOT NULL,
    role VARCHAR(20) DEFAULT 'Student',
    department VARCHAR(100),
    degree VARCHAR(100),
    university VARCHAR(100),
    bio TEXT,
    verified_ac_email BOOLEAN DEFAULT 0,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- 2. MODULES Table
CREATE TABLE modules (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    code VARCHAR(20) UNIQUE NOT NULL,
    name VARCHAR(100) NOT NULL,
    department VARCHAR(50),
    university VARCHAR(100),
    year INTEGER
```

```
);

-- 3. SWAP_REQUESTS Table
CREATE TABLE swap_requests (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    status VARCHAR(20) DEFAULT 'Pending',
    priority VARCHAR(20) DEFAULT 'Normal',
    notes TEXT,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY(user_id) REFERENCES users(id)
);

-- 4. ASSOCIATION TABLES (Resolving M2M)
-- User Enrollment
CREATE TABLE user_modules (
    user_id INTEGER,
    module_id INTEGER,
    PRIMARY KEY (user_id, module_id),
    FOREIGN KEY(user_id) REFERENCES users(id),
    FOREIGN KEY(module_id) REFERENCES modules(id)
);

-- Swap Giving
CREATE TABLE swap_give_modules (
    swap_id INTEGER,
    module_id INTEGER,
    PRIMARY KEY (swap_id, module_id),
    FOREIGN KEY(swap_id) REFERENCES swap_requests(id),
    FOREIGN KEY(module_id) REFERENCES modules(id)
);
```

```
-- Swap Wanting
CREATE TABLE swap_want_modules (
    swap_id INTEGER,
    module_id INTEGER,
    PRIMARY KEY (swap_id, module_id),
    FOREIGN KEY(swap_id) REFERENCES swap_requests(id),
    FOREIGN KEY(module_id) REFERENCES modules(id)
);

-- 5. MESSAGES Table
CREATE TABLE messages (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    swap_id INTEGER NOT NULL,
    sender_id INTEGER NOT NULL,
    receiver_id INTEGER NOT NULL,
    content TEXT NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY(swap_id) REFERENCES swap_requests(id),
    FOREIGN KEY(sender_id) REFERENCES users(id),
    FOREIGN KEY(receiver_id) REFERENCES users(id)
);
```

Code Snippets 1: DDL Script for Creating Tables

B. DML Script (Populating Example Data)

This script inserts Rajveer and Vikram as the sole users, along with their modules and a swap scenario.

```
-- 1. Insert Users (Rajveer & Vikram)

INSERT INTO users (username, email, role, department, university, verified_ac_email) VALUES
('Rajveer Singh Saini', 'rajveer.saini@mail.bcu.ac.uk', 'Student', 'Computing', 'BCU', 1),
('Vikramjeet Singh', 'vikramjeet.-3@mail.bcu.ac.uk', 'Student', 'Computing', 'BCU', 1);

-- 2. Insert Modules
INSERT INTO modules (code, name, department, university, year) VALUES
('CMP5387', 'Backend Web Services', 'Computing', 'BCU', 2),
('MUS202', 'Music Theory II', 'Arts', 'BCU', 2),
('FASH101', 'Intro to Textiles', 'Fashion', 'BCU', 1);

-- 3. Insert Enrollments (Current State)
-- Rajveer (ID 1) has Backend (ID 1)
INSERT INTO user_modules (user_id, module_id) VALUES (1, 1);
-- Vikram (ID 2) has Music (ID 2)
INSERT INTO user_modules (user_id, module_id) VALUES (2, 2);

-- 4. Create Swap Requests
-- Swap 501: Rajveer offers Backend (1) for Music (2)
INSERT INTO swap_requests (id, user_id, status, notes) VALUES (501, 1, 'Pending', 'Looking for Arts
credits.');
INSERT INTO swap_give_modules (swap_id, module_id) VALUES (501, 1);
INSERT INTO swap_want_modules (swap_id, module_id) VALUES (501, 2);

-- Swap 502: Vikram offers Music (2) for Backend (1)
INSERT INTO swap_requests (id, user_id, status, notes) VALUES (502, 2, 'Pending', 'Need technical
credits.');
```

```sql
INSERT INTO swap_give_modules (swap_id, module_id) VALUES (502, 2);
INSERT INTO swap_want_modules (swap_id, module_id) VALUES (502, 1);

-- 5. Insert Message (Negotiation)
INSERT INTO messages (swap_id, sender_id, receiver_id, content) VALUES
(501, 2, 1, 'Hi Rajveer, I see you want Music. I can trade my spot!');
```

Code Snippets 2: DML Script

C. Example SQL Queries

Query 1: The "Perfect Match" Finder *This query implements the core business logic of the app. It finds swaps where User A has what User B wants, and User B has what User A wants.*

```sql
SELECT
    s1.id AS My_Swap_ID,
    u1.username AS Me,
    m_want.name AS I_Want,
    s2.id AS Matched_Swap_ID,
    u2.username AS Matched_User,
    m_give.name AS They_Give
FROM swap_requests s1
-- Join to find what I WANT
JOIN swap_want_modules wm ON s1.id = wm.swap_id
JOIN modules m_want ON wm.module_id = m_want.id
JOIN users u1 ON s1.user_id = u1.id
-- Find a different swap (s2)
JOIN swap_requests s2 ON s1.id != s2.id
JOIN users u2 ON s2.user_id = u2.id
-- Join to find what they GIVE
JOIN swap_give_modules gm2 ON s2.id = gm2.swap_id
```

```
JOIN modules m_give ON gm2.module_id = m_give.id
-- THE MATCH CONDITION: What I want (wm) = What they give (gm2)
WHERE
    u1.username = 'Rajveer Singh Saini'
    AND wm.module_id = gm2.module_id;
```

Code Snippets 3: SQL QUERY

Planning and Design: Web Services

Discussion: The /swaps/suggest endpoint is unique in that it should be hit asynchronously through HTMX or Alpine.js as the user is actually completing the form. It returns an HTML partial (a piece of HTML) as opposed to full JSON, so the browser can essentially just inject the result it receives back onto the page without much client-side rendering required. The /admin endpoints are secured through a custom @teacher_required decorator for security

Endpoint	**HTTP Method**	**Description**	**Example Input (Request)**	**Example Output (Response)**	**Status Code**	**Happy/ Sad**
/auth/login	POST	Authenticates a user and sets the session role.	email: "s.student@uni.ac.uk" password: "securePass123"	Redirect to /profile	302	Happy

/swaps/create	POST	Submits a new swap request with preferences.	give_ids: [101, 102] want_ids: [205] priority: "High"	Flash message: "Swap Created" Redirect to /swaps	302	Happy
/swaps/create	POST	Submission with missing data.	give_ids: [] want_ids: [205]	HTML with validation errors: "You must select a module to give."	400	Sad
/swaps/suggest	POST	**AJAX:** Computes match scores against other users based on selected modules.	give_ids: [101] want_ids: [205]	**HTML Partial:** Match Found: 95%	200	Happy
/admin/swaps/<id>/status	POST	Updates the status of a swap request (Teacher only).	status: "Approved" notes: "Credit requirement met."	JSON: {"success": true, "new_status": "Approved"}	200	Happy
/profile/export	GET	Exports all student data and swap history.	N.A.	**JSON Object:** { "user": "Raj", "swaps": [...] }	200	Happy

/chat/send	WS	SocketIO Event.	{"msg": "Hi, interested?"}	Broadcast to room	N.A.	Happy

Table 4: Resource URI inventory summary table

References

1. Grinberg, M. (2018). *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media.
2. Newman, S. (2015). *Building Microservices*. O'Reilly Media. (Referenced for Modular Monolith concepts).
3. Freeman, E. (2004). *Head First Design Patterns*. O'Reilly Media.
4. Pallets Projects. (2024). *Flask Documentation*. [Online]. Available at: https://flask.palletsprojects.com/
5. SQLAlchemy. (2024). *SQLAlchemy ORM Documentation*. [Online]. Available at: https://www.sqlalchemy.org/
6. OpenAI (2024) API Reference. Available at: https://platform.openai.com/docs/ (Accessed: 18 December 2024).
7. Mozilla Developer Network (MDN), 2025. *HTTP Response Status Codes*. [Online] Available at: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status [Accessed 18 December 2025].
8. Pallets Projects, 2024. *Jinja Documentation (Version 3.1.x)*. [Online] Available at: https://jinja.palletsprojects.com/ [Accessed 18 December 2025].

Appendices

Appendix A:- User Data Sample

UserID	Name	Email	Role	Department
1	Rajveer Singh Saini	rajveer.saini@mail.bcu.ac.uk	Student	Computer Science
2	Vikramjeet Singh	vikramjeet.-3@mail.bcu.ac.uk	Student	Computer Science

Table 5: User Data Sample

Appendix B : Module Data Sample

ModuleID	Code	Name	Credits	Department
101	CMP5387	Backend Web Services	20	Comp Sci
102	CMP5333	Data Structures	20	Comp Sci
201	FASH101	Intro to Textiles	20	Fashion
301	MUS202	Music Theory II	20	Music

Table 6: Module Data Sample

Appendix C: Initial Prototyping and Source Code snippets

Code Snippet 1: Application Factory and Blueprints

This demonstrates the Modular Monolith architecture implemented in modswap/app/__init__.py. This allows us to separate our work (Rajveer on Swaps, Vikram on Chat) without merge conflicts.

```
def create_app(config_class=Config):
    app = Flask(__name__)
    app.config.from_object(config_class)

    # Initialize Extensions
    db.init_app(app)
    bcrypt.init_app(app)
    login_manager.init_app(app)
    socketio.init_app(app) # For Real-time Chat

    # Register Blueprints (Modular Structure)
    from modswap.app.auth.routes import auth
    from modswap.app.swaps.routes import swaps
    from modswap.app.chat.routes import chat
    from modswap.app.admin.routes import admin

    app.register_blueprint(auth, url_prefix='/auth')
    app.register_blueprint(swaps, url_prefix='/swaps')
    app.register_blueprint(chat, url_prefix='/chat')
    app.register_blueprint(admin, url_prefix='/admin')

    return app
```

Code Snippets 4: Application Factory and Blueprints

Code Snippet 2 : SwapRequest Model

This code from models.py defines the complex Many-to-Many relationship needed for the matching logic.

```
class SwapRequest(db.Model):
    __tablename__ = 'swap_requests'
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('users.id'), nullable=False)
    status = db.Column(db.String(20), default='Pending')
    priority = db.Column(db.String(10), default='Normal')

    # Relationships
    giving = db.relationship('Module', secondary='swap_give_modules',
                             backref='swaps_giving')
    wanting = db.relationship('Module', secondary='swap_want_modules',
                              backref='swaps_wanting')
```

Code Snippets 5: SwapRequest Model

Appendix D : Swap Request Data Sample

Transactional data showing a swap scenario between the two group members.

SwapID	UserID	Status	CreatedAt	Giving (ModuleIDs)	Wanting (ModuleIDs)
501	1 (Rajveer)	Pending	2025-12-18	[101] (Backend)	[301] (Music)
502	2 (Vikram)	Pending	2025-12-18	[301] (Music)	[101] (Backend)

Table 7: Swap Request Data Sample Table

Note: This data represents a perfect match: Rajveer gives what Vikram wants, and Vikram gives what Rajveer wants.

Appendix E : Chat/Feedback Data Sample

Data utilized by the Real-Time SocketIO module showing negotiation.

MessageID	SenderID	RecipientID	SwapReferenceID	Content	Timestamp
901	1 (Rajveer)	2 (Vikram)	501	"Hi Vikram, I see you have the Music module I want."	10:00 AM
902	2 (Vikram)	1 (Rajveer)	501	"Yes, I'm trying to switch to Backend Web Services. Let's swap!"	10:05 AM

Table 8: Chat/FeedBack Data Sample

Appendix F : Admin/Maintenance Data Sample

Logs visible in the system (Assuming Rajveer has admin privileges for testing).

LogID	AdminID	Action	TargetID	Timestamp	Notes
801	1 (Rajveer)	APPROVE	502	2025-12-18	"Approved Swap 502 (Vikram) - Credits match."
802	1 (Rajveer)	REVIEW	501	2025-12-18	"Reviewing request for interdisciplinary match."

Table 9: Admin/Maintenance Data Sample Table

Appendix G : Enrollment (User_Modules) Data Sample

The "Truth" table showing current student enrollments before the swap.

EnrollmentID	UserID	ModuleID	Semester
1001	1 (Rajveer)	101 (Backend)	1
1002	1 (Rajveer)	102 (Data Struct)	1
1003	2 (Vikram)	301 (Music)	1
1004	2 (Vikram)	201 (Textiles)	1

Table 10: Enrollment (User_Modules) Data Sample

Appendix Z : Additional Supporting Material

Sample SQL Query for Matching Logic

This query is used to identify the match shown in Appendix D.

```
SELECT s1.id AS MySwap, s2.id AS MatchSwap, s2.user_id AS MatchedUser
FROM swap_requests s1
JOIN swap_requests s2 ON s1.id != s2.id
JOIN swap_give_modules gm1 ON s1.id = gm1.swap_request_id
JOIN swap_want_modules wm2 ON s2.id = wm2.swap_request_id
WHERE s1.user_id = 1 -- Rajveer
AND gm1.module_id = wm2.module_id; -- Rajveer gives what Vikram wants
```

Code Snippets 6: Sample SQL Query for Matching Logic

Example JSON Response (Swap Suggestion API)

```
{
  "matches_found": 1,
  "suggestions": [
    {
      "swap_id": 502,
      "user_name": "Vikramjeet Singh",
      "giving_module": "Music Theory II (301)",
      "match_score": 100
    }
  ]
}
```

Code Snippets 7: SwapRequest Model