StudyFlow

Smart Academic Planner

Technical Design and Specification

Kumar Shivam
22BCE1804

November 9, 2025

Abstract

StudyFlow is a comprehensive, client-first, data-informed academic planning solution designed to address the fragmentation and inefficiency inherent in modern student task management. In an educational environment characterized by multiple concurrent courses, diverse assignment types, and competing deadlines, students face significant cognitive overhead in organizing, prioritizing, and completing their academic responsibilities. Traditional tools such as generic to-do lists, spreadsheets, or disparate course management systems fail to provide an integrated, intelligent approach to academic workflow optimization.

Purpose and Scope

This technical design document serves as the authoritative specification for StudyFlow, detailing its architecture, design patterns, data models, and implementation strategy. The document is intended for multiple stakeholder groups:

- **Students and End Users**: To understand the capabilities, value proposition, and user experience of the platform.
- **Educators and Academic Administrators**: To evaluate the pedagogical benefits and institutional applicability of the system.
- **Software Developers and Engineers**: To comprehend the technical architecture, design decisions, and implementation roadmap for development, extension, or integration purposes.
- **Product Managers and Stakeholders**: To assess the strategic positioning, scalability considerations, and future evolution of the platform.

Core Value Proposition

StudyFlow transcends the limitations of traditional task management tools by providing:

- **Unified Data Model**: A cohesive system that consolidates courses, assignments, tasks, and deadlines into a single, coherent information architecture.
- **Intelligent Prioritization**: Data-driven algorithms that surface high-priority items, upcoming deadlines, and actionable insights to optimize student productivity.
- **Proactive Reminder System**: Automated notification mechanisms that alert users to impending deadlines, overdue items, and schedule conflicts before they become critical.
- **Analytics and Insights**: Visual dashboards and statistical analysis that provide students with quantitative feedback on their productivity patterns, completion rates, course performance, and time management effectiveness.
- **Scalable Architecture**: A modular, service-oriented design that begins as a client-side Single Page Application (SPA) but is architected for seamless evolution into a multi-tenant, cloud-based platform with cross-device synchronization.

Technical Overview

The current implementation leverages modern web technologies (HTML5, CSS3, ES6+ JavaScript) with a modular service architecture that separates concerns between authentication, data management, analytics, and presentation layers. Data persistence is achieved through the Web Storage API (`localStorage`), providing immediate value without backend dependencies while maintaining architectural patterns that facilitate future migration to a full-stack solution.

The forward-looking design incorporates industry best practices including RESTful/GraphQL API design, OAuth2.1/OIDC authentication, NoSQL database modeling (MongoDB), message queue integration for asynchronous operations, and microservice patterns for scalability. Security considerations include Argon2id password hashing, JWT token management, HTTPS enforcement, and GDPR-compliant privacy controls.

Document Structure

This specification encompasses:

- Detailed problem statement and user stories defining the system requirements
- Comprehensive system architecture and design including component diagrams, data models, and technology stack
- Security, performance, and scalability considerations for production deployment
- Quality assurance strategy with unit, integration, and end-to-end testing frameworks
- Visual documentation including Data Flow Diagrams (DFD), Entity-Relationship Diagrams (ERD), UML class and sequence diagrams, and architectural topology illustrations
- Software Requirements Specification (SRS) with functional and non-functional requirements
- API design and endpoint specifications for future backend implementation

The document balances immediate implementation details with strategic architectural considerations, ensuring that StudyFlow can deliver value today while maintaining a clear path toward enterprise-scale capabilities tomorrow.

Contents

Executive Summary

- **Audience**: Students who need a unified planner; educators evaluating tooling; developers extending a modular SPA.
- **Core Value**: Consolidates deadlines across courses, prioritizes work, and surfaces insights to improve outcomes.
- **Now vs. Next**: Today, state and analytics are computed client-side using `localStorage`. Next, we introduce secure auth, a REST/GraphQL API, and push-based reminders for cross-device sync.

1 Problem Statement

In the contemporary academic landscape, students are inundated with a high volume of tasks, assignments, and deadlines across multiple courses. This complexity often leads to increased cognitive load, inefficient time management, and a reactive rather than proactive approach to learning. The absence of an integrated, intelligent system for tracking academic progress results in missed deadlines, last-minute stress, and a lack of insight into personal productivity patterns.

StudyFlow is engineered to address these challenges by providing a centralized, data-driven platform for academic task management. It aims to be more than a simple to-do list; it is a smart planner that leverages data analytics to offer actionable insights, proactive reminders, and a holistic view of a student's academic life. The core problem is the fragmentation of academic information and the lack of intelligent tools to synthesize this information into a personalized, optimized workflow. StudyFlow bridges this gap by offering a unified ecosystem for managing courses, assignments, and tasks, thereby empowering students to take control of their learning journey.

2 User Stories

The system is designed around the primary actor, the "Student," with a secondary "Guest" role for trial purposes.

- **As a Student, I want to create a secure account and log in,** so that my academic data is private and persistent across sessions.
- **As a Student, I want to add, edit, and delete my courses for the semester,** so that I can organize my academic responsibilities by subject.
- **As a Student, I want to add assignments with titles, descriptions, due dates, and priority levels,** so that I can track all my graded work in one place.
- **As a Student, I want to create general tasks with due dates and priorities,** so that I can manage non-assignment related activities like "Review lecture notes" or "Prepare for exam."
- **As a Student, I want to see a consolidated dashboard,** so that I can get an immediate overview of my active courses, pending tasks, and high-priority items.
- **As a Student, I want to view upcoming deadlines on my dashboard,** so that I can prioritize my work effectively.
- **As a Student, I want to receive automated reminders for tasks and assignments that are due soon or overdue,** so that I can stay on top of my deadlines.

- **As a Student, I want to view an analytics dashboard,** so that I can understand my productivity patterns, such as my task completion rate and performance in different courses.
- **As a Guest User, I want to quickly access the app with sample data without creating an account,** so that I can evaluate its features before committing to registration.

3 System Architecture and System Design

While the current implementation is a client-side application, the architecture is designed with a forward-looking, scalable, server-based model in mind. The design follows a modular, service-oriented pattern.

3.1 Architectural Model: Client-Server with Micro-frontend Influence

The system is architected as a **Single Page Application (SPA)** that communicates with a conceptual backend via a RESTful API. The frontend itself is modular, with distinct components for Authentication, Data Management, and UI Rendering, mimicking a micro-frontend approach for maintainability.

- **Client (Browser)**: The client is responsible for all UI rendering and state management. It is built with HTML5, CSS3, and modern JavaScript (ES6+). The client-side logic is partitioned into:
  - **Auth Service (`auth.js`)**: Manages user sessions, registration, and login. In the current model, it interfaces with `localStorage` as a mock database.
  - **Data Service (`app.js`)**: The `DataManager` class acts as an Object-Relational Mapping (ORM) layer, abstracting the data persistence logic (currently `localStorage`). It provides a clean API for CRUD operations on courses, assignments, and tasks.
  - **Analytics Service (`analytics.js`)**: A client-side data processing engine that computes and visualizes metrics from the data service.
  - **View Components (`app.js`, `index.html`)**: A set of functions responsible for rendering the dynamic HTML for each section (Dashboard, Courses, etc.).
- **Conceptual Backend Server**: A stateless server that would expose a REST API for the client.
  - **Authentication Endpoint**: Handles user registration and JWT (JSON Web Token) generation.
  - **API Gateway**: A single entry point for all client requests, routing them to the appropriate microservices.
  - **Data Service**: A microservice responsible for all business logic related to courses, tasks, and assignments.
  - **Notification Service**: A service to manage and push real-time reminders (e.g., via WebSockets or Push API).
- **Database**: A NoSQL database like MongoDB would be ideal for storing user data due to its flexible schema, which aligns well with the structure of courses, tasks, and assignments.

3.2 Data Model

The core data entities are Users, Courses, Assignments, and Tasks.

- **User**: `{id, username, email, passwordHash, createdAt}`
- **Course**: `{id, userId, name, code, instructor, color, createdAt}`
- **Assignment**: `{id, userId, courseId, title, description, dueDate, priority, completed, createdAt}`
- **Task**: `{id, userId, courseId (optional), title, description, dueDate, priority, completed, completedAt, createdAt}`

3.3 Technology Stack

Layer	Technology
UI/SPA	HTML5, CSS3, Vanilla ES6+ (modular services); optional Tailwind/Bootstrap for scale
State & Storage (current)	Web Storage API (`localStorage`) per-user keyspace
Visualization	SVG-based custom charts (no heavy deps)
Auth (future)	OAuth2.1 / OIDC with PKCE, JWT access tokens, refresh token rotation
API (future)	REST/JSON or GraphQL over HTTPS; rate-limited behind API Gateway
Database (future)	MongoDB (flexible schema, TTL indexes for reminders)
Async (future)	Message queue (e.g., RabbitMQ) for reminder fan-out and email/push workers

3.4 Key Quality Attributes

- **Reliability**: Idempotent APIs, deterministic analytics, and safe writes with server-side validation.
- **Performance**: Incremental rendering; O(1) access patterns for hot lists; lazy-loading archives.
- **Security**: Strong hashing (Argon2id/bcrypt), token hardening (DPoP/MTLS optional), strict content security policy.
- **Accessibility**: WCAG 2.1 AA contrast, keyboard-first flows, ARIA roles on dynamic widgets.

3.5 Component Diagram

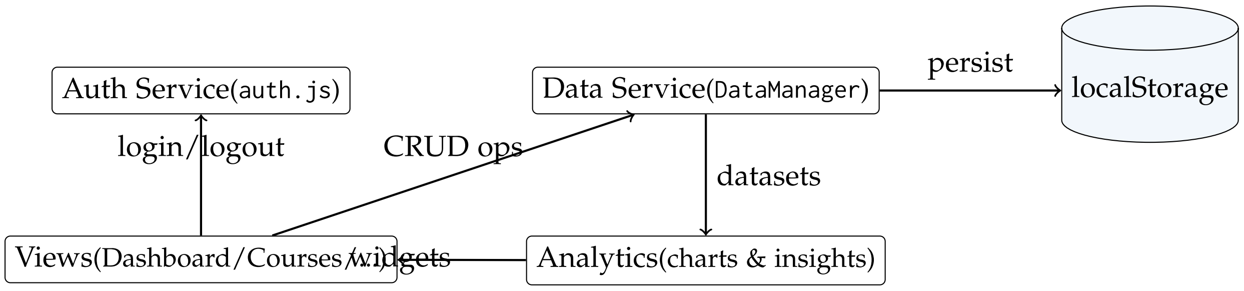
Auth Service(auth.js)
Data Service(DataManager)
persist
localStorage
login/logout
CRUD ops
datasets
Views(Dashboard/Courses/...)
widgets
Analytics(charts & insights)

Figure 1: Frontend component interactions (current client-only model)

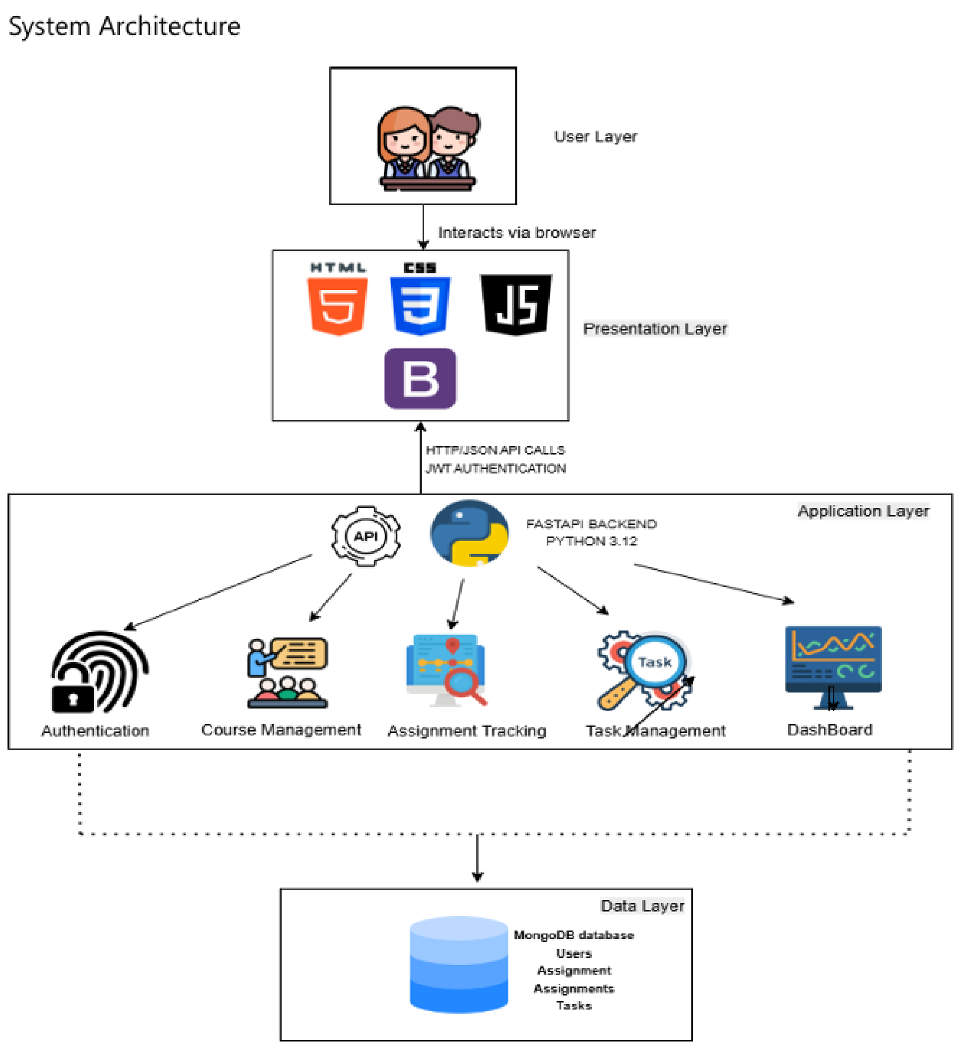

Figure 2: High-level architecture overview: client SPA today, with future API, services, and data stores.

$$\begin{tikzcd}
\textbf{User} \arrow[r, leftrightarrow, "\text{click \textit{Save}\ \ validate fields}", "\text{re-render list}"'] & \textbf{View} \arrow[r, leftrightarrow, "\text{addTask(task)}", "\text{task id}"'] & \textbf{DataManager} \arrow[r, leftrightarrow, "\text{write key}", "\text{ack}"'] & \textbf{Storage}
\end{tikzcd}$$

Figure 3: Happy-path sequence for adding a task

3.6 Representative Sequence (Add Task)

3.7 Security and Privacy

- **Authentication**: Transition to OIDC with short-lived access tokens and rotating refresh tokens; device binding where feasible.
- **Password storage**: Use Argon2id with memory-hard parameters; enforce strong password policy and have breach checking (k-Anonymity).
- **Transport**: Enforce TLS 1.3; HSTS; same-site cookies (`Strict`); CSRF tokens on state-changing requests.
- **Privacy**: Data minimization, user export/delete (GDPR Art. 15/17), regional data residency options.

3.8 Performance and Scalability

- **Client**: Batch DOM updates, memoize computed aggregates, virtualize long lists.
- **Server (future)**: Use CQRS for analytics vs. writes; cache hot queries (e.g., next 7 days) with TTL.
- **Reminders**: Pre-compute next trigger per item; wheel-timer buckets to scale to millions of schedules.

3.9 Deployment Overview (Future)

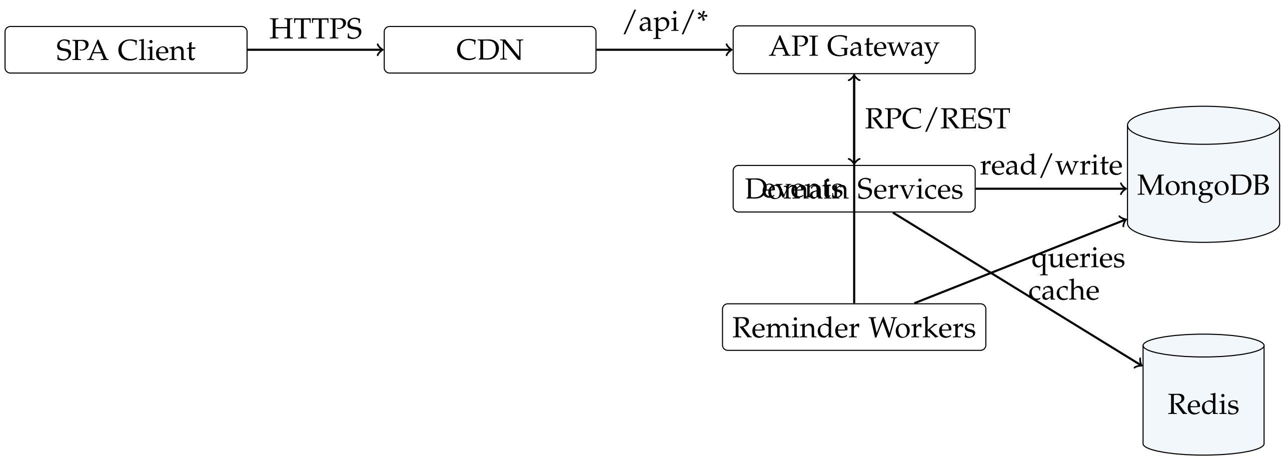

Figure 4: Future deployment topology for a multi-service backend with caching and workers.

4 Design of Tests

A comprehensive testing strategy would involve unit, integration, and end-to-end (E2E) tests.

4.1 Unit Testing

Unit tests would focus on isolating and verifying the smallest parts of the application.

- **`auth.js`**:
  - Test email validation with valid and invalid inputs.
  - Test password length and match constraints.
  - Mock `localStorage` to test user registration (checking for duplicates) and login (password verification).
- **`app.js` (`DataManager`)**:
  - Mock `localStorage` to test all CRUD operations: `addCourse`, `updateCourse`, `deleteCourse`, etc.
  - Verify that IDs and timestamps are correctly generated on creation.
  - Test `toggleTaskCompletion` to ensure the `completed` and `completedAt` fields are updated correctly.
- **`analytics.js`**:

  - Create mock data sets to test the calculation logic for `renderCompletionRate`, `renderPriorityChart`, `renderWeeklyProgress`, and `renderCoursePerformance`.
  - Test edge cases, such as when there are no tasks or courses.

4.2 Integration Testing

Integration tests would verify that different modules work together as expected.

- **Auth and Data**: Test that after a user logs in, the `DataManager` is initialized with the correct `userId` and loads the correct data.
- **Data and UI**: Test that after adding a new course via the UI, the course list view is correctly re-rendered with the new data.
- **Reminders and Data**: Test that the `checkReminders` function correctly identifies upcoming tasks from the `DataManager` and triggers a notification.

4.3 End-to-End (E2E) Testing

E2E tests would simulate real user workflows from start to finish using a framework like Cypress or Playwright.

- **Full User Journey**:
  1. A user registers for a new account.
  2. The user logs in.
  3. The user creates a new course.
  4. The user adds an assignment to that course.
  5. The user creates a standalone task.
  6. The user marks the task as complete.
  7. The user navigates to the analytics page and verifies the completion rate has updated.
  8. The user logs out.

4.4 Test Matrix (Representative)

Area	**Key Cases**	**Priority**
Auth	Register, login, bad password, existing user, email format	High
Courses	Add, edit, delete, list empty state, persistence	High
Assignments	CRUD, validation, due date parsing, course linkage	High
Tasks	CRUD, priority filter, toggle complete, sorting	High
Analytics	Zero-data, partial-data, completion %, weekly bars	Medium
Reminders	$\leq 24\,\text{h}$, overdue, sort order, badge counts	High
Import/Export	Round-trip fidelity, schema mismatch, error handling	Medium

4.5 Edge Cases

- Tasks without due dates should not appear in urgency lists; ensure stable sorting.
- Mixed timezones across entries; normalize to ISO 8601 UTC for computation.
- Deleting a course with linked tasks/assignments prompts confirmation and preserves orphaned tasks or offers reassignment.

5 Appendix

5.1 SRS (Software Requirements Specification)

5.1.1 Functional Requirements

- **FR1: User Authentication**: The system shall allow users to register, log in, and log out.
- **FR2: Course Management**: The system shall allow users to perform CRUD (Create, Read, Update, Delete) operations on courses.
- **FR3: Assignment Management**: The system shall allow users to perform CRUD operations on assignments, which must be linked to a course.
- **FR4: Task Management**: The system shall allow users to perform CRUD operations on tasks. Tasks can optionally be linked to a course.
- **FR5: Dashboard**: The system shall display a dashboard summarizing key metrics (e.g., total courses, pending tasks).
- **FR6: Analytics**: The system shall provide visualizations for task completion rates, priority distribution, and weekly progress.
- **FR7: Reminders**: The system shall automatically check for and notify the user of impending deadlines.

5.1.2 Non-Functional Requirements

- **NFR1: Performance**: The UI must load in under 3 seconds. UI updates in response to user actions must feel instantaneous ($< 200\,\text{ms}$).
- **NFR2: Usability**: The application must be intuitive and usable without a manual. It must be responsive and function on mobile, tablet, and desktop browsers.
- **NFR3: Security**: All user passwords must be hashed before storage. (Note: `btoa` is a placeholder for a proper hashing algorithm like Argon2 or bcrypt).
- **NFR4: Data Persistence**: All user-generated data must be saved and persist between sessions.

5.1.3 API (Future) – Representative Endpoints

Method	Endpoint	Notes
POST	/auth/register	Creates user, returns verification challenge
POST	/auth/login	Issues JWT (short-lived) + refresh token
GET	/courses	List user courses
POST	/courses	Create course
GET	/tasks?dueBefore=...	Query tasks by time window/priority
PATCH	/tasks/{id}	Partial update; ETag required

5.2 DFD (Data Flow Diagram)

5.2.1 Level 0 DFD (Context Diagram)

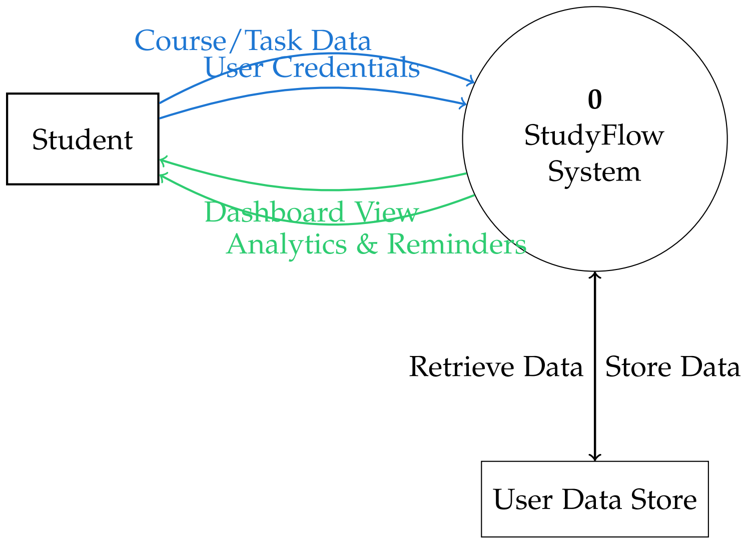

Figure 5: Level 0 DFD (Context Diagram) - High-level system interactions

5.2.2 Level 1 DFD (Detailed Process Decomposition)

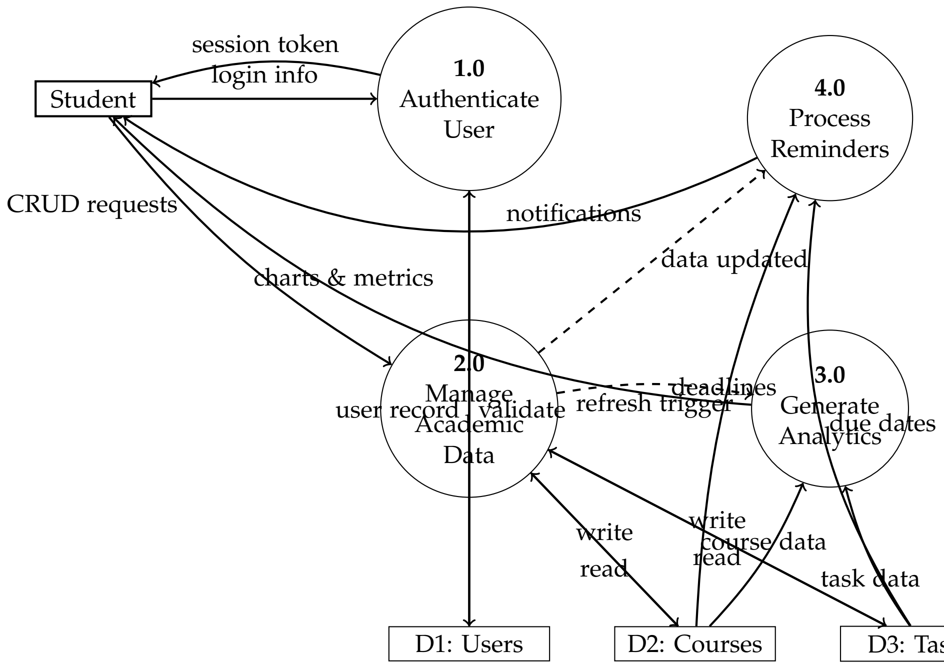

Figure 6: Level 1 DFD - Detailed process decomposition showing all major subsystems

5.3 ERD (Entity-Relationship Diagram)

5.3.1 Conceptual ERD

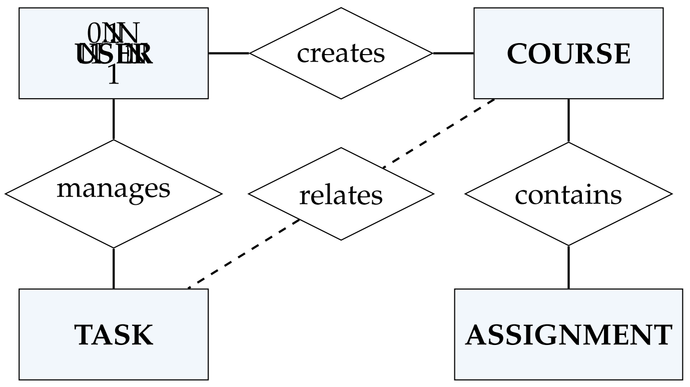

Figure 7: Conceptual ERD showing entity relationships and cardinalities

5.3.2 Logical ERD with Attributes

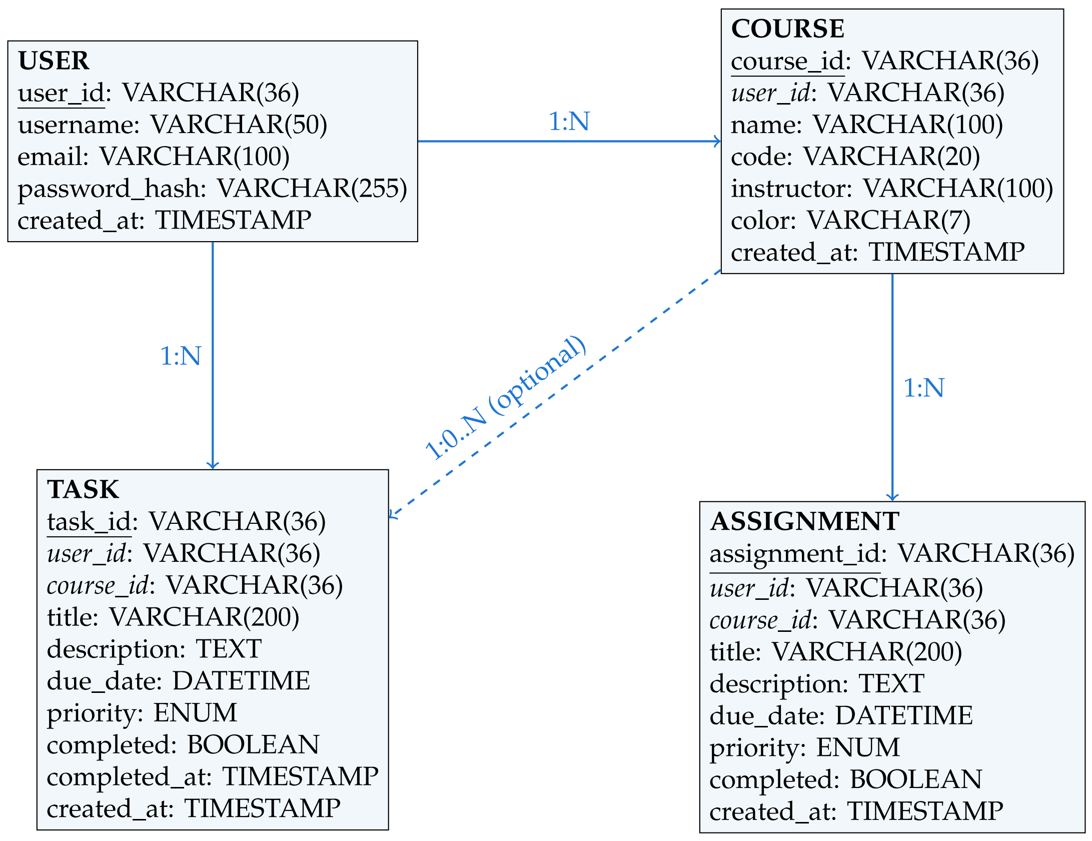

Figure 8: Logical ERD with complete attributes and data types

5.3.3 Database Schema Notes

- **Primary Keys**: All entities use UUID (VARCHAR(36)) for globally unique identifiers
- **Foreign Keys**:
  - `COURSE.user_id` references `USER.user_id`

- `ASSIGNMENT.user_id` and `ASSIGNMENT.course_id` reference respective parent tables
- `TASK.user_id` references `USER.user_id`
- `TASK.course_id` is optional (nullable) for standalone tasks

- **Indexes**: Recommended indexes on `user_id`, `due_date`, and `completed` fields for query performance
- **Priority Enum**: Values = {LOW, MEDIUM, HIGH, URGENT}
- **Cascade Rules**: ON DELETE CASCADE for user deletion; ON DELETE SET NULL for course deletion in tasks

5.4 UML Diagrams

5.4.1 Use Case Diagram

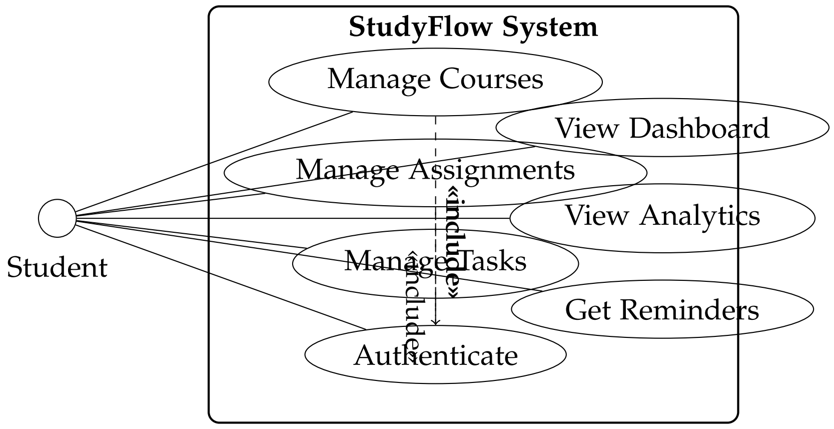

Figure 9: Use Case Diagram showing primary student interactions

5.4.2 Class Diagram (Comprehensive)

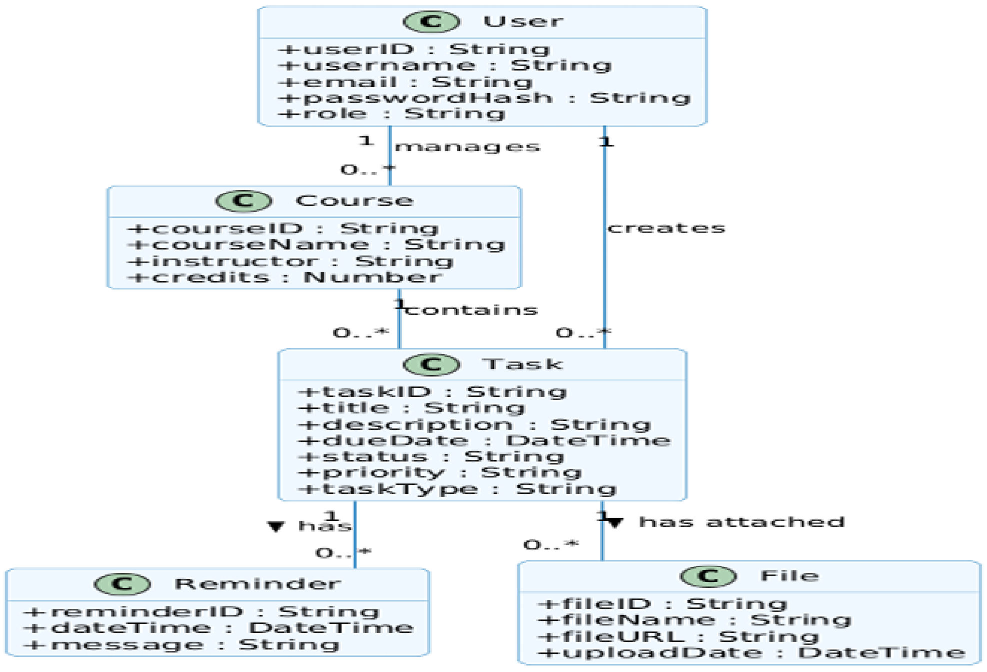

Figure 10: UML Class Diagram showing complete system class structure (imported from assets)

5.4.3 Class Diagram (Simplified Core Classes)

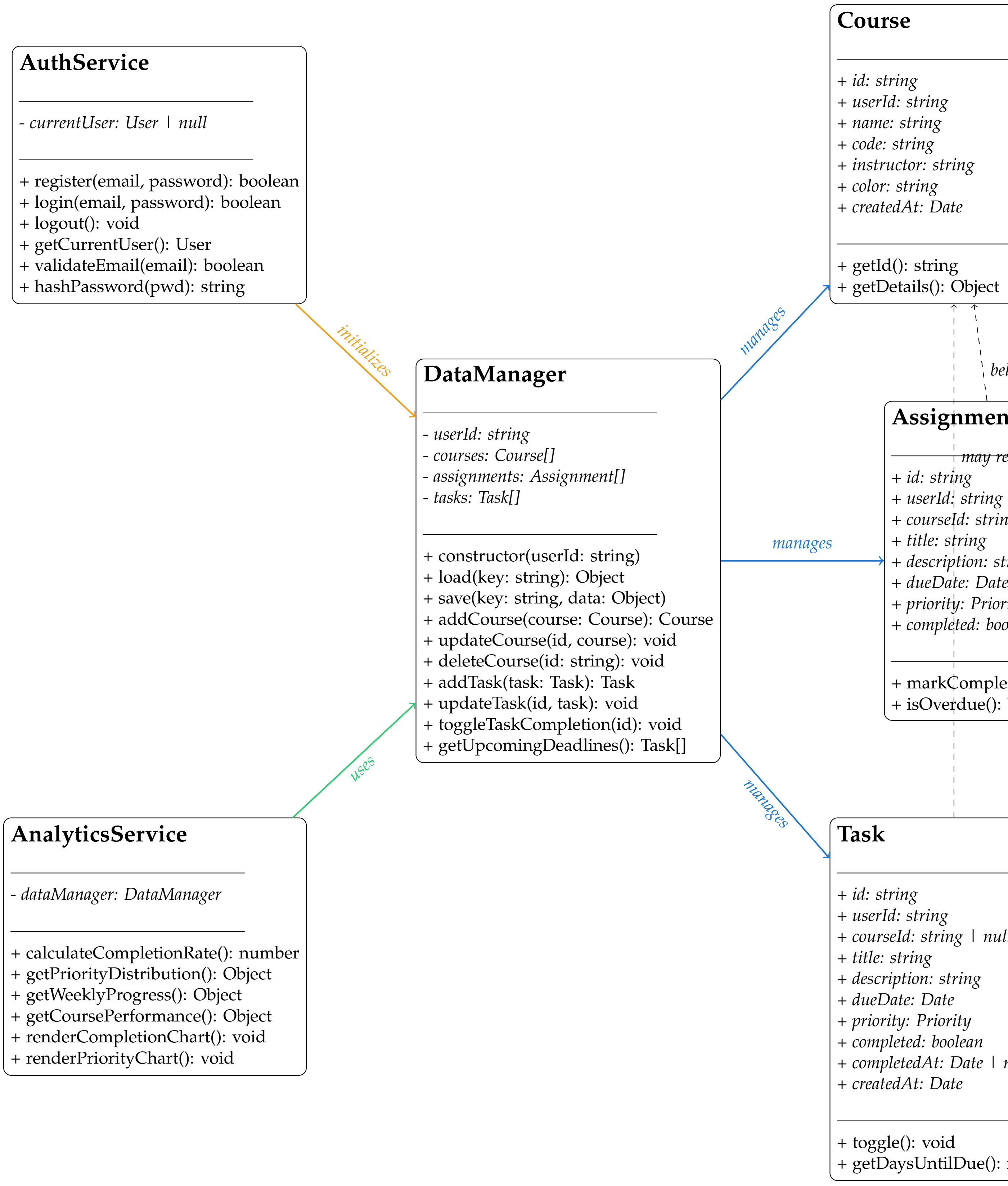

Figure 11: Simplified UML Class Diagram - Core application classes with relationships and methods

5.4.4 Activity Diagram (User Task Flow)

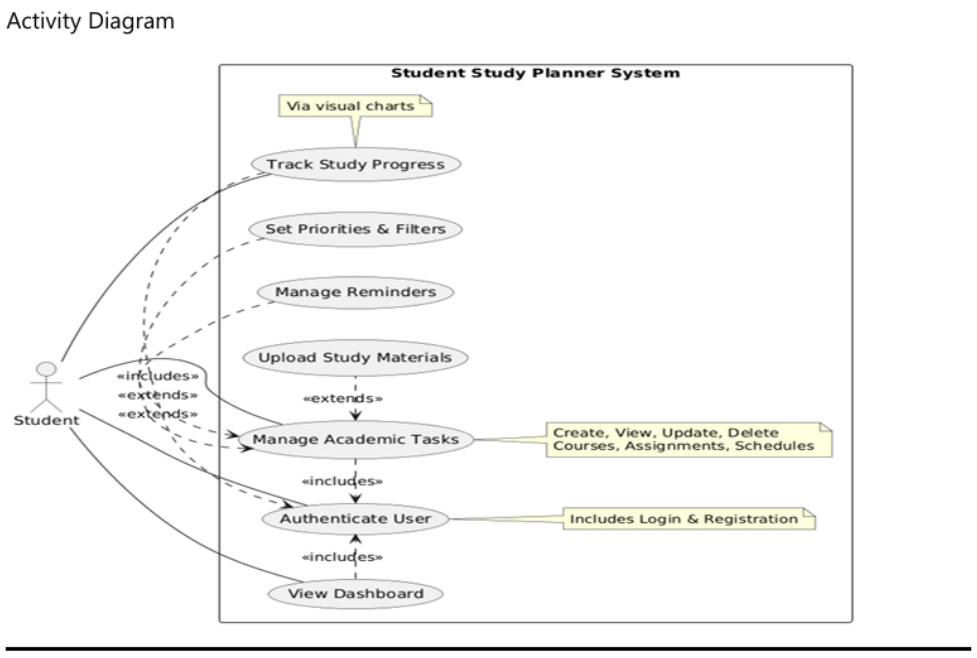

Figure 12: UML Activity Diagram illustrating typical user task management workflow (imported from assets)

5.4.5 Reminder Notification Sequence

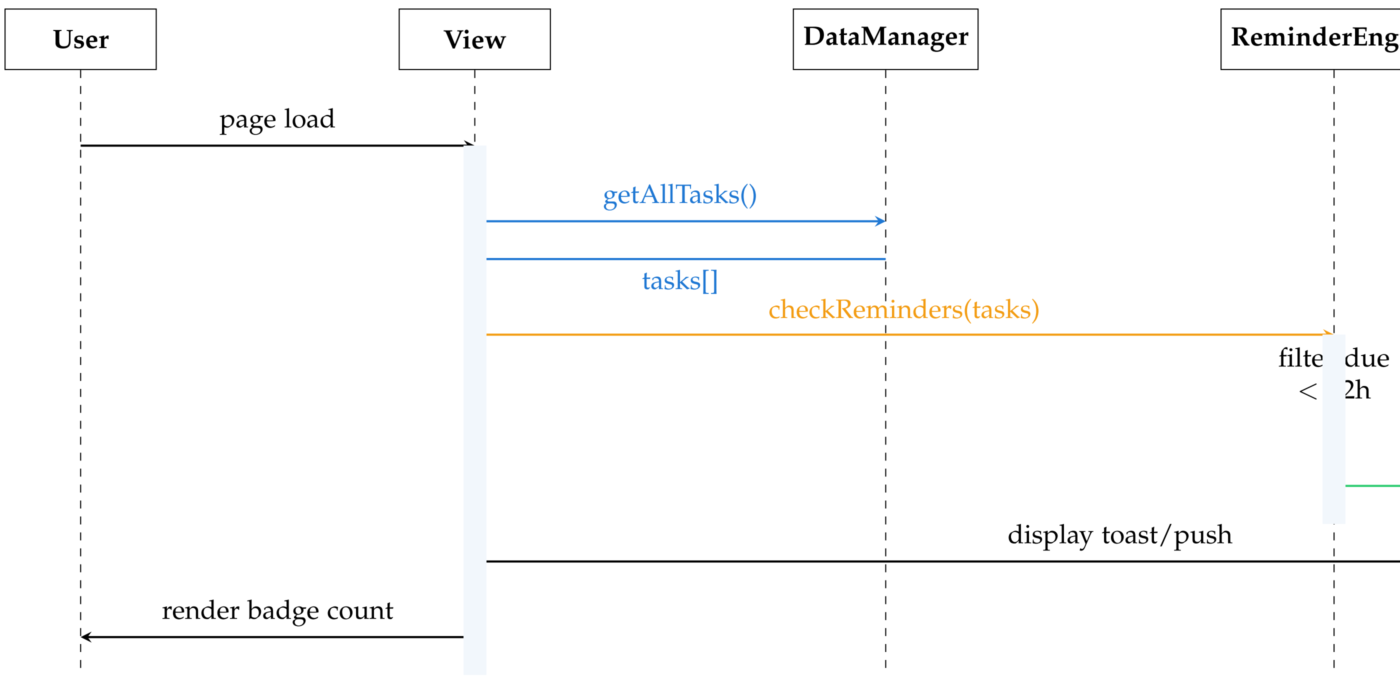

Figure 13: Sequence diagram for reminder notification system showing deadline detection and user alerts

5.5 Code Listing / GitHub Link

The project is a client-side application. The core logic for data management is encapsulated in the `DataManager` class in `app.js`.

```
// Data Storage Manager
class DataManager {
    constructor(userId) {
        this.userId = userId;
        this.courses = this.load('courses') || [];
        this.assignments = this.load('assignments') || [];
        this.tasks = this.load('tasks') || [];
    }

    load(key) {
        const data = localStorage.getItem(`${this.userId}_${key}`);
        return data ? JSON.parse(data) : null;
    }

    save(key, data) {
        localStorage.setItem(`${this.userId}_${key}`, JSON.stringify(data));
    }

    // Courses
    getCourses() {
        return this.courses;
    }

    addCourse(course) {
        course.id = Date.now().toString();
        course.createdAt = new Date().toISOString();
        this.courses.push(course);
        this.save('courses', this.courses);
        return course;
    }

    // ... other CRUD methods for courses, assignments, and tasks
}
```

Listing 1: DataManager Class from app.js

A public GitHub repository can be provided for full source code access: `https://github.com/KumarShivam1908/Planner.git`