МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри інтелектуальних інформаційних систем

_____ Євген СІДЕНКО

«____»_____ 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА

ІНФОРМАЦІЙНА СИСТЕМА ПЛАНУВАННЯ НАВЧАЛЬНОГО НАВАНТАЖЕННЯ ЗДОБУВАЧІВ З ВИКОРИСТАННЯМ ІНТЕЛЕКТУАЛЬНИХ ТЕХНОЛОГІЙ

Спеціальність 122 Комп'ютерні науки

Освітня програма «Комп'ютерні науки»

Здобувачка _______ Юлія ГОРЯЧЕВА

«___»_____ 2026 р.

Керівник канд. техн. наук, ст. викладач ________ Віктор ГОЖИЙ

«___»__________ 2026 р.

м. Миколаїв – 2026 рік

Чорноморський національний університет імені Петра Могили
(повне найменування закладу вищої освіти)

Факультет	Комп'ютерних наук
Кафедра	Інтелектуальних інформаційних систем
Рівень вищої освіти	Перший (бакалаврський)
Освітній ступень	Бакалавр
Спеціальність	122 Комп'ютерні науки
Освітня програма	Комп'ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри інтелектуальних інформаційних систем

_____ Євген СІДЕНКО

«____» _____ 2026 р.

ЗАВДАННЯ
на кваліфікаційну роботу здобувачки

Горячевої Юлія Вікторівни
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: «Інформаційна система планування навчального навантаження здобувачів з використанням інтелектуальних технологій».

Керівник роботи: Гожий Віктор Олександрович, старший викладач кафедри інтелектуальних інформаційних систем, канд. техн. наук.

Затверджена наказом ЧНУ ім. Петра Могили від «25» грудня 2025 р. № 353.

2. Строк представлення кваліфікаційної роботи «___» _____ 2026 р.

3. Очікуваний результат роботи та початкові дані, якщо такі потрібні: розробка

інформаційної системи планування навчального навантаження здобувачів з використанням інтелектуальних технологій, яка забезпечує облік навчальних дисциплін, завдань, термінів виконання, пріоритетів і прогнозованого навантаження, а також формування персоналізованих рекомендацій щодо розподілу навчальної діяльності. Початкові дані містять відомості про освітню програму, перелік навчальних дисциплін, індивідуальний навчальний план здобувача, дані про види навчальної діяльності, строки виконання завдань, орієнтовні витрати часу на їх виконання, а також результати аналізу існуючих інформаційних систем планування та методів інтелектуального аналізу даних, що можуть бути використані для оптимізації навчального навантаження.

4. Перелік питань, що підлягають розробці: дослідження предметної області та аналіз існуючих інформаційних систем планування навчального навантаження здобувачів; визначення функціональних і нефункціональних вимог до інформаційної системи; проєктування архітектури системи та структури бази даних для зберігання відомостей про дисципліни, навчальні завдання, дедлайни, пріоритети, складність і прогнозоване навантаження; вибір і обґрунтування інтелектуальних технологій для аналізу навчального навантаження та формування рекомендацій; побудова UML-діаграм для моделювання роботи системи; розробка користувацького інтерфейсу вебзастосунку; програмна реалізація основних модулів системи, зокрема обліку дисциплін і завдань, аналізу завантаженості, планування навчальної діяльності, візуалізації результатів і формування персоналізованих рекомендацій; проведення тестування та оцінка ефективності розробленої інформаційної системи.

5. Перелік графічних матеріалів: презентація.

Керівник роботи _____ Віктор ГОЖИЙ
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувачка _____ Юлія ГОРЯЧЕВА
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата видачі завдання «23» грудня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

кваліфікаційної роботи

Тема: «Інформаційна система планування навчального навантаження здобувачів з використанням інтелектуальних технологій».

№	Найменування роботи	Початок	Закінчення	Примітки
1	Отримання завдання на виконання КР	21.12.2025	24.12.2025	Виконано
2	Аналіз предметної області та постановка задачі	25.12.2025	30.01.2026	Виконано
3	Огляд науково-технічної літератури та аналіз існуючих інформаційних систем і сервісів планування навчальної діяльності	31.01.2026	01.03.2026	Виконано
4	Формування функціональних і нефункціональних вимог до системи, обґрунтування вибору технологічного стеку та інтелектуальних технологій	02.03.2026	01.04.2026	Виконано
5	Програмна реалізація інформаційної системи планування навчального навантаження.	02.04.2026	24.05.2026	Виконано
6	Перший попередній захист КР на засіданні комісії кафедри	25.05.2026	25.05.2026	Виконано
7	Корегування роботи за результатами попереднього захисту	26.05.2026	04.06.2026	Виконано
8	Другий попередній захист КР на засіданні комісії кафедри	05.06.2026	05.06.2026	Виконано
9	Доробка та остаточне оформлення КР	06.06.2026	14.06.2026	Виконано
10	Подання КР, її електронної копії та інших документів (відгуку, рецензії) до захисту	15.06.2026	19.06.2026	Виконано

Керівник роботи _____ Віктор ГОЖИЙ
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувачка _____ Юлія ГОРЯЧЕВА
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата складання календарного плану
«29» січня 2026 р.

АНОТАЦІЯ

до кваліфікаційної роботи
здобувачки 401 групи ЧНУ ім. Петра Могили

Горячевої Юлії Вікторівни

На тему: **«ІНФОРМАЦІЙНА СИСТЕМА ПЛАНУВАННЯ НАВЧАЛЬНОГО НАВАНТАЖЕННЯ ЗДОБУВАЧІВ З ВИКОРИСТАННЯМ ІНТЕЛЕКТУАЛЬНИХ ТЕХНОЛОГІЙ»**

Актуальність роботи зумовлена потребою раціонального розподілу навчального часу здобувачів в умовах цифровізації освіти, тоді як більшість наявних сервісів планування та навчальних платформ не забезпечують комплексного аналізу навчального навантаження та персоналізованих рекомендацій.

Об'єктом роботи є процес планування та аналізу навчального навантаження здобувача вищої освіти в межах веборієнтованої інформаційної системи.

Предметом роботи є програмна реалізація алгоритму планування навчального навантаження здобувача та основних модулів інформаційної системи: бази даних PostgreSQL, REST API на FastAPI, модуля обліку дисциплін і завдань, аналізу та візуалізації навантаження і формування текстових рекомендацій за допомогою Gemini API.

Метою роботи є розробка інформаційної системи, яка дає змогу здобувачам планувати й контролювати навчальне навантаження, автоматизувати облік дисциплін і завдань, аналізувати динаміку навантаження та отримувати персоналізовані рекомендації щодо розподілу навчального часу.

У роботі проаналізовано існуючі системи планування, спроєктовано клієнт-серверну архітектуру та структуру бази даних, реалізовано серверну частину на FastAPI з PostgreSQL і клієнтську на React, Redux і Tailwind CSS. Розроблено алгоритм оцінювання навчального навантаження, модулі візуалізації, виконано

Тестування основних сценаріїв і сформульовано напрями подальшого розвитку системи.

Робота складається з трьох розділів. У першому розділі виконано аналіз предметної області планування навчального навантаження здобувачів, розглянуто існуючі сервіси планування та сформовано вимоги до інформаційної системи. У другому розділі спроєктовано архітектуру вебзастосунку, структуру бази даних PostgreSQL, REST API на FastAPI та описано алгоритм оцінювання навчального навантаження. У третьому розділі наведено результати програмної реалізації системи на основі FastAPI, PostgreSQL, React, Redux і Tailwind CSS, описано інтеграцію Gemini API, візуалізацію результатів та результати тестування. Загальний обсяг роботи – 96 сторінок. Кваліфікаційна робота містить 1 додаток, 17 рисунків, 6 таблиць і 39 джерел посилання.

Ключові слова: навчальне навантаження, планування, інформаційна система, FastAPI, PostgreSQL, React, Gemini API.

ABSTRACT

to the qualification work by the student of the group 401 Petro Mohyla Black Sea National University

Horiacheva Yuliia

«INFORMATION SYSTEM FOR PLANNING STUDENTS' ACADEMIC WORKLOAD USING INTELLIGENT TECHNOLOGIES»

The relevance of this work is driven by the need for a rational distribution of students' study time in the context of education digitalization, while most existing planning services and learning platforms do not provide comprehensive analysis of academic workload and personalized recommendations.

The object of the work is the process of planning and analyzing the academic workload of a higher education student within a web-oriented information system.

The subject of the work is the software implementation of the student workload planning algorithm and the main modules of the information system: the PostgreSQL database, the REST API based on FastAPI, the module for managing courses and tasks, the module for workload analysis and visualization, and the module for generating text recommendations using the Gemini API.

The aim of the work is to develop an information system that enables students to plan and monitor their academic workload, automate the tracking of courses and tasks, analyze workload dynamics, and receive personalized recommendations on the allocation of study time.

The work analyzes existing planning systems, designs the client–server architecture and database structure, implements the backend using FastAPI with PostgreSQL and the frontend using React, Redux, and Tailwind CSS. An algorithm for assessing

academic workload and visualization modules have been developed, core usage scenarios have been tested, and directions for further system development have been outlined.

The qualification thesis consists of three chapters. The first chapter analyzes the subject area of student workload planning, reviews existing planning services, and defines the requirements for the information system. The second chapter presents the design of the web application architecture, the PostgreSQL database structure, the REST API implemented with FastAPI, and the workload assessment algorithm. The third chapter describes the implementation of the system using FastAPI, PostgreSQL, React, Redux, and Tailwind CSS. It also covers the integration of the Gemini API, data visualization, and the results of system testing. The total volume of the thesis is 96 pages. The qualification work includes 1 appendix, 17 figures, 6 tables, and 39 references.

Keywords: academic workload, planning, information system, FastAPI, PostgreSQL, React, Gemini API.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

API – Application Programming Interface

HTTP – HyperText Transfer Protocol

JSON – JavaScript Object Notation

ORM – Object-Relational Mapping

REST – Representational State Transfer

UI – User Interface

UX – User Experience

LMS – Learning Management System

CSS – Cascading Style Sheets

ВСТУП

У сучасних умовах цифровізації освітнього процесу організація навчальної діяльності здобувачів усе більше залежить від використання інформаційних технологій [1]. Для ефективного планування навчального часу, контролю дедлайнів, розподілу завдань і зменшення ризику перевантаження необхідні зручні вебсистеми [2], здатні автоматизувати облік навчальних задач, забезпечити централізоване зберігання даних і надати інструменти аналізу навантаження.

Одним із ключових компонентів таких систем є серверна частина [3], яка відповідає за бізнес-логіку застосунку, обробку даних, роботу з базою даних та надання програмного інтерфейсу взаємодії між клієнтською частиною і системою. У межах розроблюваної інформаційної системи така взаємодія реалізується через REST API [4], а збереження даних про дисципліни, завдання, дедлайни та результати аналізу здійснюється у PostgreSQL [5].

Актуальність теми роботи зумовлена зростаючою потребою здобувачів у спеціалізованих інформаційних системах [6], які дозволяють не лише фіксувати навчальні завдання, а й аналізувати рівень навантаження, визначати ризик перевантаження та формувати рекомендації щодо раціонального розподілу часу. Особливо важливим є використання інтелектуальних технологій [7] як допоміжного інструменту для пояснення результатів аналізу та формування зрозумілих текстових порад.

Метою роботи є проєктування та практична реалізація інформаційної системи планування навчального навантаження здобувачів, яка забезпечує облік дисциплін і навчальних завдань, обробку даних, розрахунок рівня навантаження та надання програмного інтерфейсу для взаємодії клієнтської частини із серверною логікою системи.

Для досягнення поставленої мети було визначено такі завдання:
– проаналізувати предметну область планування навчального навантаження здобувачів;
– визначити функціональні вимоги до інформаційної системи;
– спроєктувати архітектуру клієнт-серверного вебзастосунку та структуру бази даних [8];
– розробити REST API для роботи з основними сутностями системи [9];
– розробити власний алгоритм оцінювання та розподілу навчального навантаження [10];
– інтегрувати Gemini API як допоміжний модуль формування рекомендацій [11];
– описати реалізацію клієнтської частини системи з використанням React, Redux і Tailwind CSS [12];
– виконати тестування реалізованих програмних модулів [13];
– підготувати звітну документацію за результатами виконання кваліфікаційної роботи.

Практичне значення роботи полягає у створенні інформаційної системи, яка може бути використана як основа для персонального планувальника навчального навантаження здобувача. Така система поєднує засоби обліку навчальних завдань, серверну обробку даних, власний алгоритм планування, візуалізацію результатів і формування рекомендацій з використанням інтелектуальних технологій [14].

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ПЛАНУВАННЯ НАВЧАЛЬНОГО НАВАНТАЖЕННЯ ЗДОБУВАЧІВ ТА ВИМОГ ДО ІНФОРМАЦІЙНОЇ СИСТЕМИ

1.1 Особливості планування навчального навантаження здобувачів

Предметна область системи розглядається з позиції кінцевого користувача – здобувача вищої освіти, який щодня стикається з необхідністю розподіляти обмежений час між кількома навчальними дисциплінами, завданнями різної складності та строками виконання, що постійно змінюються [15]. На відміну від виробничих або проєктних систем планування, навчальний контекст має специфічні характеристики: завдання надходять нерівномірно, пріоритети визначаються не лише важливістю, а й наближенням контролю, а суб'єктивна оцінка власних можливостей у студентів часто не збігається з реальним обсягом необхідної роботи [16].

До ключових об'єктів предметної області належать: навчальна дисципліна як контейнер для групування завдань; навчальне завдання як атомарна одиниця планування; дедлайн як часове обмеження виконання; пріоритет як відносна важливість задачі; оцінювана складність і орієнтовна тривалість виконання як кількісні характеристики трудомісткості; статус виконання як показник поточного прогресу; а також персональний календар як просторово-часова структура, в межах якої розміщуються всі задачі [17]. Окремим параметром виступає доступний час користувача – кількість годин, які здобувач може реально виділити в конкретний день з урахуванням розкладу занять, особистих зобов'язань та фізичної спроможності до продуктивної роботи [18]. Саме цей параметр є одним із найменш врахованих у традиційних системах, хоча він суттєво впливає на коректність будь-якого розкладу [19].

Центральна проблема ручного планування полягає не у відсутності інформації про окремі задачі, а у відсутності механізму агрегації цієї інформації в цілісну картину

навантаження [20]. Здобувач, як правило, бачить список завдань і може впорядкувати їх за дедлайном або важливістю, однак не має інструменту для оцінки того, яке сумарне навантаження формується на конкретний день або тиждень. Окреме завдання з дедлайном через п'ять днів може здаватися малозначущим, але в контексті одночасного виконання лабораторної роботи, підготовки до тесту й опрацювання матеріалів до семінару воно стає частиною перевантаженого тижня. Такий ефект накопиченого навантаження є одним із поширених джерел академічного стресу та зниження якості виконання завдань [21].

Для усунення цієї проблеми система має виконувати не лише функцію сховища задач, а й їхню аналітичну обробку: обчислювати щоденне та тижневе навантаження у годинах, порівнювати його з доступним часом користувача й сигналізувати про можливе перевантаження ще до того, як воно настане [22]. Такий підхід переводить систему з пасивного трекера в активний інструмент підтримки прийняття рішень.

Окремою характеристикою предметної області є висока динамічність навчального процесу [23], яка унеможливлює одноразове статичне планування. Викладач може змінити дату здачі роботи або додати нове завдання; здобувач може виконати частину задачі раніше запланованого терміну або, навпаки, зіткнутися з труднощами, які вимагають більше часу, ніж очікувалось; зовнішні обставини – хвороба, позапланові заходи, зміна розкладу – можуть суттєво скоротити доступний час у певний день. У всіх цих випадках система повинна не просто зафіксувати зміну, а перерахувати весь план із урахуванням нових умов.

Це означає, що алгоритм планування має бути ітеративним і підтримувати повторний запуск при кожній зміні вхідних параметрів [24]. Статичний розклад, складений один раз на початку тижня, втрачає актуальність вже після першої непередбаченої події, тоді як динамічний перерахунок забезпечує постійну відповідність плану реальному стану справ.

З точки зору проєктування користувацького досвіду, ефективна система

планування повинна балансувати між функціональністю та простотою взаємодії [25]. Якщо для внесення кожного завдання здобувач змушений заповнювати десятки полів, налаштовувати складні параметри або вручну розподіляти часові блоки, ймовірність регулярного використання системи різко знижується. Поведінкові дослідження у сфері людино-комп'ютерної взаємодії підтверджують, що когнітивне навантаження від самого інструменту планування не повинно перевищувати когнітивне навантаження від задачі, яку він покликаний спростити [26].

Тому доцільною є стратегія мінімального введення: користувач надає лише базові параметри задачі – назву, дисципліну, дедлайн, орієнтовну складність і пріоритет – а система самостійно розраховує трудомісткість, пропонує розподіл роботи по днях і відображає результат у зрозумілому інтерфейсі [27]. Додаткові параметри, такі як доступний час на день, можуть бути задані один раз у профілі користувача й автоматично враховуватись при кожному перерахунку. Такий підхід знижує поріг входу до системи й підвищує ймовірність її стабільного використання протягом навчального семестру.

1.2 Аналіз існуючих сервісів планування та навчальних платформ

Для обґрунтування доцільності розроблення власної інформаційної системи було проведено порівняльний аналіз наявних сервісів планування та навчальних платформ [28], які здобувачі освіти найчастіше застосовують для організації навчальної діяльності. Розглянуті рішення охоплюють чотири основні категорії: універсальні менеджери задач, календарні сервіси, спеціалізовані навчальні платформи та гнучкі робочі простори [29]. Кожна з цих категорій вирішує певне коло завдань, однак жодна з них не забезпечує комплексного підходу до персонального планування навчального навантаження з урахуванням реального часового ресурсу студента.

Універсальні менеджери задач, зокрема Trello [30] та подібні сервіси на основі

канбан-методології, орієнтовані на візуальне структурування роботи: користувач створює картки, групує їх за статусами, встановлює дедлайни та пріоритети. Для загальної продуктивності такий підхід є ефективним – він дозволяє швидко оцінити, які завдання вже виконані, які в процесі, а які ще не розпочаті. Однак при використанні в навчальному контексті виявляються суттєві обмеження. По-перше, канбан-дошка не передбачає прив'язки задач до конкретних дисциплін як структурних одиниць освітнього процесу, що ускладнює відстеження рівномірності завантаження по предметах. По-друге, такі системи не виконують жодного розрахунку трудомісткості: навіть якщо користувач вказав орієнтовну тривалість виконання завдання, сервіс не агрегує ці дані й не попереджає про перевантаження [31]. По-третє, відсутній механізм зіставлення обсягу запланованої роботи з реально доступним часом на конкретний день, що є ключовою умовою коректного навчального планування.

Календарні сервіси, насамперед Google Calendar [32], вирішують іншу задачу – відображення подій у часовому вимірі . Їхньою безперечною перевагою є наочність: користувач бачить свій розклад у розрізі днів, тижнів або місяців, може додавати нагадування, синхронізувати події між пристроями та ділитись календарем з іншими. Для відображення розкладу занять, консультацій і контрольних заходів такий інструмент є зручним і інтуїтивно зрозумілим.

Разом із тим календарний підхід має принципове обмеження: він фіксує часові межі подій, але не аналізує змістовне навантаження, яке за ними стоїть. Якщо здобувач позначив у календарі «підготовка до іспиту» як одногодинний блок, система прийме цей параметр без будь-якої перевірки його реалістичності. Відповідно, два однаково виглядаючі дні в календарі можуть насправді відрізнятися за реальним інтелектуальним навантаженням у рази – і жодного сигналу про це система не надасть [33]. Крім того, Google Calendar не підтримує поняття «складності завдання» чи «залишкового часу до дедлайну» як аналітичних параметрів, що унеможливлює

автоматичне формування рекомендацій щодо розподілу роботи.

Приклад інтерфейсу сервісу Trello наведено на рис. 1.1, а інтерфейсу Google Calendar – на рис. 1.2.

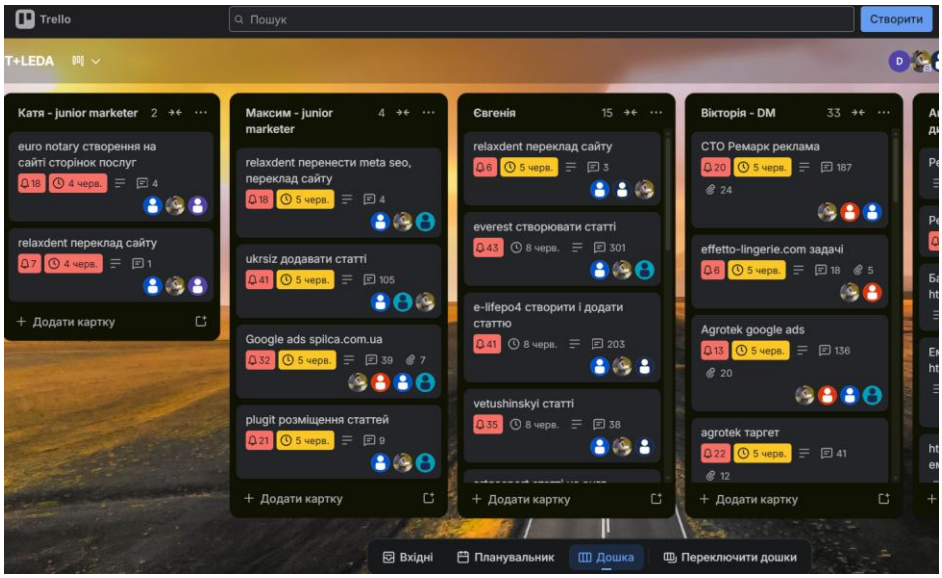

Рисунок 1.1 – Інтерфейс сервісу Trello

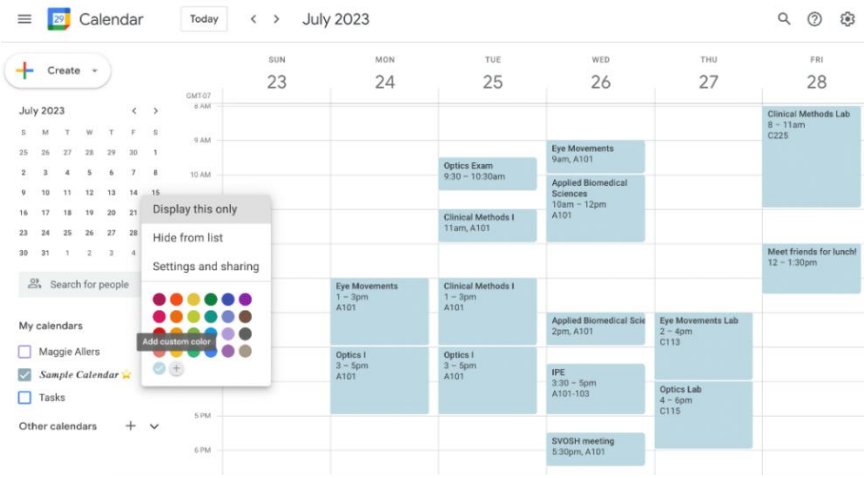

Рисунок 1.2 – Інтерфейс сервісу Google Calendar

Навчальні платформи на зразок Moodle [34] займають окрему нішу серед інструментів, що використовуються в освітньому процесі . На відміну від менеджерів задач або календарних сервісів, такі системи проєктуються насамперед для підтримки

взаємодії між викладачем і здобувачем у межах конкретної дисципліни: вони забезпечують доступ до навчальних матеріалів, лекційних слайдів, тестових завдань, індивідуальних і групових робіт, журналу оцінок та комунікаційних інструментів – форумів, повідомлень, оголошень.

З точки зору закладу освіти, Moodle виконує важливу функцію централізованого середовища для організації навчального контенту та моніторингу академічної успішності. Однак саме в цьому й полягає принципова відмінність від того, що потрібне здобувачеві як індивідуальному плановику власного часу. Платформа відображає, що потрібно зробити і до якого терміну, але не дає відповіді на питання коли саме і в якому обсязі варто розподілити роботу між конкретними днями тижня.

Здобувач, який користується Moodle як єдиним інструментом організації навчання, фактично отримує структурований перелік активностей без будь-якого аналізу їхньої сукупної трудомісткості. Два завдання з однаковим дедлайном можуть вимагати принципово різних часових витрат, однак у стандартному інтерфейсі платформи вони виглядатимуть ідентично – просто як два рядки у списку. Жодного механізму для оцінки реального навантаження, попередження про критичне перевищення можливостей або формування персонального графіка роботи в системі не передбачено, оскільки це не входить до її проєктних цілей [35].

Таким чином, Moodle є ефективним інструментом управління навчальним контентом з боку викладача, проте не вирішує задачу персонального планування часу з боку здобувача. Ця функціональна прогалина є характерною для більшості LMS-платформ (Learning Management Systems) і підтверджує необхідність окремого інструменту, орієнтованого саме на індивідуальне управління навчальним навантаженням [36].

Приклад інтерфейсу навчальної платформи Moodle наведено на рис. 1.3.

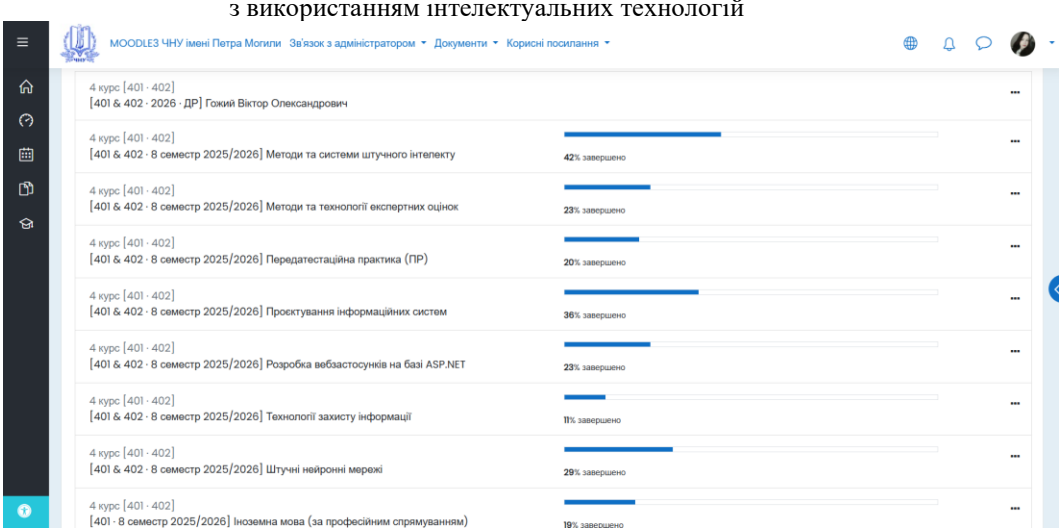

Рисунок 1.3 – Інтерфейс навчальної платформи Moodle

Окрему групу становлять гнучкі робочі простори, найбільш показовим представником яких є Notion [37]. На відміну від вузькоспеціалізованих інструментів, такі сервіси позиціонують себе як універсальне середовище для роботи з інформацією будь-якого типу: в межах одного простору користувач може вести нотатки, будувати бази даних, створювати канбан-дошки, таблиці, календарі, вікі-сторінки та поєднувати їх між собою у довільній структурі. Саме ця багатофункціональність зробила Notion популярним серед студентів, які шукають єдине середовище для організації всіх аспектів навчання.

Практична цінність таких інструментів справді є відчутною на початковому етапі: здобувач може самостійно спроєктувати власну систему планування – створити таблицю дисциплін, додати поля для дедлайнів, пріоритетів і статусів, налаштувати фільтри та подання. За бажання можна реалізувати досить складну логіку з використанням вбудованих формул або автоматизацій через сторонні сервіси на кшталт Zapier чи Make. Для технічно підготовленого користувача це відкриває широкі можливості кастомізації.

Однак тут і виявляється ключове протиріччя подібних платформ: чим гнучкіший інструмент, тим більше часу й зусиль потрібно витратити на його налаштування, підтримку та вдосконалення. Здобувач, замість того щоб планувати навчання, починає «планувати інструмент для планування» – підбирати властивості, виправляти формули, перебудовувати структуру бази даних. Цей ефект, який у літературі з продуктивності іноді називають надмірною інструменталізацією , знижує практичну цінність системи саме для тих користувачів, яким найбільше потрібна допомога з організацією часу.

Крім того, Notion не має вбудованого механізму аналізу навчального навантаження. Система не розраховує, скільки годин роботи заплановано на конкретний день, не порівнює цей показник із доступним часом користувача і не генерує жодних рекомендацій щодо перерозподілу задач. Усе це або реалізується вручну через складні формули, або залишається поза увагою системи повністю. Таким чином, Notion є потужним інструментом для тих, хто готовий інвестувати час у його налаштування, проте не вирішує задачу автоматизованого планування навчального навантаження «з коробки». Приклад інтерфейсу робочого простору Notion показано на рис. 1.4.

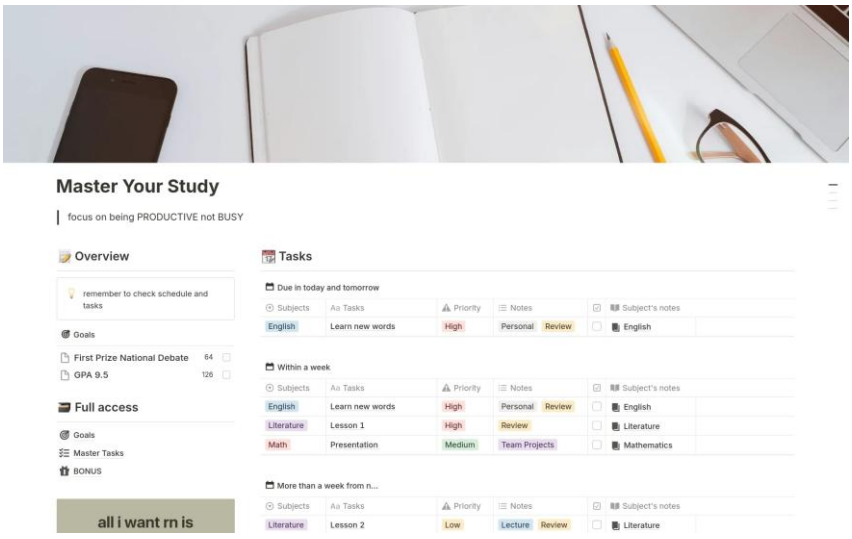

Рисунок 1.4 – Інтерфейс робочого простору Notion

Порівняння функціональних можливостей розглянутих рішень із розроблюваною системою наведено в табл. 1.1. На основі цієї таблиці можна детальніше проаналізувати, які саме аспекти підтримуються окремими сервісами, а які залишаються поза їхнім фокусом.

Таблиця 1.1 – Порівняння існуючих рішень із розроблюваною системою

Критерій	**Trello**	**Google Calendar**	**Moodle**	**Notion**	**Розроблювана система**
Облік дисциплін	частково	ні	так	налаштовується	так
Дедлайни задач	так	так	так	так	так
Оцінка трудомісткості	частково	ні	ні	налаштовується	так
Аналіз перевантаження	ні	ні	частково	через формули	так
Автоматичний розподіл роботи	ні	ні	ні	частково	так
Текстові рекомендації	ні	ні	ні	ні	через Gemini API
Візуалізація навантаження	частково	так	частково	налаштовується	так

Як видно з табл. 1.1, усі розглянуті рішення певною мірою підтримують роботу з дедлайнами: користувач може зафіксувати кінцеву дату виконання задачі або події. Водночас лише частина сервісів дозволяє явно працювати з навчальними дисциплінами. У Trello така можливість реалізується опосередковано – через створення окремих дощок чи списків, що лише частково відображає структуру освітнього процесу. Moodle, навпаки, чітко оперує поняттями курсу та дисципліни, але його механізми орієнтовані насамперед на подання матеріалів і контроль знань, а не на гнучке індивідуальне планування. У Notion можливість обліку дисциплін повністю залежить від того, як користувач сам побудує структуру бази даних та

зв’язків між таблицями.

Окремої уваги потребує критерій оцінки трудомісткості. Лише в окремих випадках користувач може явно зазначити тривалість виконання задачі або її вагу, наприклад, через додаткові поля чи мітки у Trello або самостійно створені стовпці в Notion. У календарних сервісах, як-от Google Calendar, тривалість події може бути вказана, але це не завжди відповідає реальній трудомісткості навчального завдання і не використовується системою для подальшого аналізу. В результаті здобувачеві складно оцінити сукупний обсяг роботи за певний проміжок часу та виявити перевантажені періоди.

Ще більшою є різниця у підтримці аналізу перевантаження та автоматичного розподілу роботи. Більшість сервісів не мають вбудованих механізмів, які б оцінювали сумарне навантаження за тиждень чи семестр та пропонували варіанти перерозподілу задач. В окремих випадках, наприклад у Notion, подібний аналіз можна реалізувати за допомогою формул і фільтрів, але це вимагає від користувача значної підготовки та ручного налаштування. Навчальні платформи частково дозволяють оцінити інтенсивність навчальної роботи через календар завдань і контрольних заходів, проте не виконують автоматизованого розкладання задач у часі з урахуванням індивідуального графіка здобувача.

Критичною відмінністю розроблюваної системи є підтримка текстових рекомендацій та інтелектуального супроводу планування. Існуючі сервіси, описані вище, переважно надають користувачеві інструменти для ручного внесення даних, але не пояснюють, як краще розподілити роботу, які задачі варто перенести, а які – виконати в першу чергу. Впровадження механізмів генерації пояснювальних рекомендацій на основі аналізу навантаження, зокрема із використанням засобів штучного інтелекту (Gemini API), дозволяє запропонувати здобувачу якісно інший рівень підтримки при плануванні навчання.

Візуалізація навантаження в існуючих рішеннях також має обмежений характер.

Календарні сервіси добре показують розподіл подій у часі, але не акцентують на їхній складності та сумарній кількості задач. Менеджери задач відображають списки і стани виконання, проте не завжди надають агреговане уявлення про навантаження в розрізі дисциплін чи тижнів. У гнучких просторах на кшталт Notion відповідні графіки та діаграми потрібно створювати власноруч. Розроблювана система передбачає цілеспрямоване відображення навчального навантаження у вигляді наочних діаграм, що дозволяє здобувачу швидко оцінити ситуацію та прийняти обґрунтовані рішення.

Отже, проведений аналіз показує, що готові рішення можуть успішно закривати окремі потреби здобувача – фіксацію дедлайнів, збереження навчальних матеріалів, ведення списків задач або створення гнучких шаблонів. Разом із тим жоден із розглянутих інструментів не об’єднує в одному застосунку облік дисциплін, планування навчальних задач, оцінку трудомісткості, аналіз перевантаження, автоматизований розподіл часу та формування пояснювальних рекомендацій. Саме ця функціональна прогалина стала підставою для вибору теми практики та формування вимог до власної інформаційної системи.

Розроблювана система не має на меті конкурувати з великими освітніми платформами за всіма функціями і не дублює їхній контентний або контрольно-оцінювальний компоненти. Її призначення є вужчим і прикладним: надати здобувачеві персональний інструмент для планування навчальної діяльності, який автоматично враховує структуру дисциплін, оцінює сумарне навантаження, виявляє потенційне перевантаження та пропонує варіанти перерозподілу задач у часі. Такий інструмент може використовуватись паралельно з наявними університетськими сервісами й навчальними платформами, доповнюючи їх персоналізованою аналітикою та рекомендаційною підтримкою.

1.3 Функціональні та нефункціональні вимоги до системи

Формування вимог до інформаційної системи є одним із ключових етапів

проєктування, оскільки саме на цьому кроці абстрактна ідея перетворюється на конкретний технічний опис того, що система має робити і якими характеристиками повинна володіти [38]. Вимоги традиційно поділяються на дві категорії: функціональні, які описують поведінку системи з точки зору користувача, та нефункціональні, які визначають якісні та технічні обмеження реалізації. Обидві категорії однаково важливі: функціональні вимоги визначають цінність системи для користувача, а нефункціональні – реалістичність і стабільність її повсякденного використання.

Вимоги до системи формувались на основі результатів аналізу предметної області та виявлених обмежень існуючих рішень. Відправною точкою слугували реальні сценарії поведінки здобувача: що саме він робить при плануванні навчання, на якому кроці виникають труднощі, якої інформації йому бракує і яке рішення системи було б для нього практично корисним. Такий підхід дозволяє уникнути ситуації, коли система реалізує технічно цікаві функції, які, однак, не вирішують жодної реальної проблеми користувача.

Основними користувацькими сценаріями є: створення навчальної дисципліни як контейнера для групування завдань, додавання навчального завдання з усіма необхідними атрибутами, перегляд сформованого календарного плану, аналіз рівня навантаження в розрізі днів і дисциплін; а також отримання текстових рекомендацій щодо організації роботи. Кожен із цих сценаріїв безпосередньо пов'язаний із проблемою, виявленою в попередньому розділі: відсутністю інструменту, який не лише зберігає задачі, а й аналізує їхній сукупний вплив на тижневе навантаження. Основні функціональні вимоги до системи наведено в таблиці 1.2.

Таблиця 1.2 – Основні функціональні вимоги до системи

Позначення	**Вимога**	**Опис**
FR-1	Керування дисциплінами	створення, редагування та перегляд навчальних дисциплін
FR-2	Керування завданнями	облік назви, дедлайну, складності, пріоритету, статусу і тривалості
FR-3	Аналіз навантаження	розрахунок сумарного навантаження за днями та дисциплінами
FR-4	Побудова плану	розподіл задач по днях з урахуванням дедлайнів і доступного часу
FR-5	Рекомендації	формування текстових порад на основі результатів алгоритму
FR-6	REST API	обмін даними між клієнтською і серверною частинами

Вимога FR-3 є центральною з аналітичної точки зору, оскільки саме вона реалізує функцію, якої бракує у всіх розглянутих аналогах. Розрахунок сумарного навантаження передбачає агрегацію трудомісткості всіх активних задач у розрізі конкретних днів і порівняння отриманих значень із доступним часом користувача. Результатом є не просто список завдань, а кількісна характеристика кожного дня, яка дає змогу завчасно виявити перевантажені або, навпаки, недостатньо завантажені відрізки тижня.

Вимога FR-4 доповнює аналіз практичним виходом – системою розподілу задач по днях. Алгоритм побудови плану враховує дедлайни як жорсткі обмеження, пріоритет як критерій черговості, а доступний час як ресурсне обмеження. Це дозволяє уникнути ситуації, коли всі задачі формально присутні в системі, але їхній розподіл у часі залишається повністю на розсуд користувача.

Вимога FR-5 забезпечує інтерпретацію результатів аналізу у форматі,

доступному для сприйняття без спеціальної підготовки. Текстові рекомендації, сформовані на основі агрегованих даних про навантаження, дозволяють користувачеві швидко зрозуміти поточну ситуацію та прийняти обґрунтоване рішення щодо перерозподілу роботи.

Нефункціональні вимоги [39] визначають якісні характеристики системи, які безпосередньо впливають на готовність користувача використовувати її щодня. Для навчального планувальника особливо критичними є три аспекти: швидкість відгуку інтерфейсу, простота введення даних і зрозумілість представлення результатів. Якщо будь-який із цих аспектів реалізований незадовільно, користувач швидко відмовляється від інструменту – навіть якщо його функціональна логіка є коректною.

З точки зору клієнтської частини, вибір React як основи інтерфейсу обумовлений необхідністю реактивного оновлення представлення при зміні стану – наприклад, при додаванні нового завдання або зміні статусу виконання. Управління станом застосунку через Redux забезпечує передбачуваність і централізованість даних, що особливо важливо при одночасній роботі з кількома дисциплінами та великою кількістю задач. Tailwind CSS дозволяє реалізувати адаптивний інтерфейс без надмірного обсягу стилів, що позитивно впливає на швидкість завантаження і підтримуваність коду.

На рівні серверної частини ключовими нефункціональними вимогами є структурованість і передбачуваність REST API, коректна валідація вхідних даних та стабільна робота з PostgreSQL як основним сховищем . Кожен ендпоінт має повертати зрозумілі структури даних з відповідними HTTP-статусами, а валідація на рівні FastAPI – запобігати потраплянню некоректних значень до бізнес-логіки та бази даних. Для тестування клієнтських компонентів і ключових сценаріїв взаємодії передбачено використання Jest, що дозволяє виявити регресії при розширенні функціональності.

Окремо варто зазначити вимоги, пов'язані з відповідальним використанням

зовнішніх інтелектуальних сервісів. При формуванні запиту до Gemini API система передає виключно агреговані знеособлені дані про навантаження – без імен, персональних ідентифікаторів або деталей, що дозволяють ідентифікувати конкретного користувача. Такий підхід не лише відповідає базовим принципам захисту персональних даних, а й демонструє усвідомлене проєктне рішення: мінімізація переданої інформації знижує ризики, пов'язані із зовнішніми залежностями, і водночас скорочує обсяг токенів у запиті, що позитивно впливає на швидкість отримання відповіді.

1.4 Роль REST API та інтелектуальних технологій у розробці

Вибір архітектурного підходу для будь-якої інформаційної системи є не технічною формальністю, а проєктним рішенням, яке визначає, наскільки легко система розвиватиметься, тестуватиметься і підтримуватиметься в майбутньому. Для розроблюваного застосунку була обрана класична клієнт-серверна архітектура з чітким розподілом відповідальності між рівнями: React відповідає за формування інтерфейсу та взаємодію з користувачем, FastAPI – за обробку HTTP-запитів, виконання бізнес-логіки та алгоритмів планування, PostgreSQL – за надійне структуроване збереження даних. Такий поділ дозволяє змінювати або вдосконалювати кожен рівень незалежно, не зачіпаючи решту компонентів системи.

REST API виступає зв'язувальним шаром між клієнтською і серверною частинами, забезпечуючи стандартизований і передбачуваний спосіб обміну даними. Кожна функціональна можливість системи відображається у вигляді окремого HTTP-ендпоінту з чітко визначеними методом, маршрутом і структурою відповіді. Список дисциплін отримується через GET-запит, нове завдання створюється через POST, оновлення статусу виконання або пріоритету – через PATCH, видалення – через DELETE. Така відповідність між діями користувача та HTTP-методами робить API інтуїтивно зрозумілим як для розробника, так і для інструментів автоматичного

тестування.

Практична цінність REST-підходу виявляється також у можливості незалежного тестування серверної логіки без запущеного інтерфейсу. FastAPI автоматично генерує інтерактивну документацію у форматі Swagger UI, що дозволяє перевіряти коректність роботи кожного ендпоінту безпосередньо в браузері. Це суттєво прискорює цикл розробки: помилка у бізнес-логіці може бути виявлена і виправлена ще до того, як відповідний інтерфейсний компонент буде реалізований. Окрім цього, REST-архітектура є природно розширюваною – якщо в майбутньому з'явиться потреба додати мобільний клієнт або інтеграцію із зовнішнім календарним сервісом, серверна частина залишиться незмінною, а нова функціональність реалізується через додаткові ендпоінти.

Питання про роль штучного інтелекту в системі є принциповим із точки зору проєктування. Існує два полярних підходи: повністю делегувати логіку планування мовній моделі або використовувати її виключно як допоміжний інструмент для інтерпретації результатів. Перший підхід виглядає привабливо з точки зору простоти реалізації, однак має суттєві недоліки – відповідь мовної моделі є недетермінованою, її важко перевірити формально, а при зміні вхідних даних результат може відрізнятися навіть за однакових умов. Покладати на такий механізм критичну логіку розрахунку навантаження означає жертвувати контрольованістю системи заради зовнішнього ефекту.

У розроблюваній системі прийнято альтернативне рішення: основний розрахунок навантаження виконується власним детермінованим алгоритмом, який оперує числовими параметрами задач – тривалістю, пріоритетом, дедлайном і доступним часом користувача . Результатом є структуровані дані: кількість годин навантаження на кожен день, список задач із ризиком невиконання, розподіл роботи по дисциплінах. Ці дані є точними, відтворюваними і не залежать від зовнішніх сервісів. Gemini API залучається на наступному етапі – для перетворення числових

результатів на зрозумілий природномовний коментар . Система передає до моделі виключно агреговані знеособлені показники, на основі яких модель формує текстову рекомендацію – конкретну, контекстуальну і зрозумілу без додаткових пояснень. Такий поділ відповідальності між детермінованою логікою та генеративною моделлю забезпечує прозорість системи і дозволяє користувачеві в будь-який момент зрозуміти, на основі яких саме даних була сформована та чи інша порада.

Висновки до розділу 1

У першому розділі виконано комплексний аналіз предметної області планування навчального навантаження здобувачів вищої освіти, досліджено наявні програмні рішення та сформовано систему вимог до розроблюваної інформаційної системи.

Аналіз предметної області дозволив визначити ключові об'єкти системи – навчальну дисципліну, навчальне завдання, дедлайн, пріоритет, складність, орієнтовну тривалість, статус виконання та персональний календар – і встановити між ними змістовні зв'язки. Окремо обґрунтовано роль параметра доступного часу користувача як критичної змінної, що визначає коректність будь-якого плану розподілу навчальної роботи. Встановлено, що саме цей параметр найменш враховується в існуючих рішеннях, хоча його вплив на якість планування є визначальним.

Виявлено три системні проблеми ручного планування. Перша – відсутність механізму агрегації навантаження: здобувач бачить окремі задачі, але не має інструменту для оцінки їхнього сукупного впливу на конкретний день чи тиждень. Друга – статичність більшості наявних підходів, які не підтримують автоматичного перерахунку плану при зміні вхідних умов, що є неприйнятним для динамічного навчального процесу. Третя – висока когнітивна вартість налаштування гнучких інструментів, яка фактично унеможливлює їхнє регулярне використання здобувачами

без спеціальної технічної підготовки.

Порівняльний аналіз чотирьох категорій існуючих рішень – універсальних менеджерів задач, календарних сервісів, навчальних платформ та гнучких робочих просторів – підтвердив, що кожен із розглянутих інструментів закриває лише частину потреб здобувача. Trello забезпечує зручне ведення списків задач, але не аналізує трудомісткість і не враховує структуру дисциплін. Google Calendar наочно відображає події в часі, однак не оцінює змістовне навантаження, що стоїть за кожною подією. Moodle є ефективним середовищем для подання навчального контенту з боку закладу освіти, але не орієнтований на персональне планування часу здобувача. Notion надає широкі можливості кастомізації, проте вимагає значного обсягу ручного налаштування і не містить вбудованих механізмів аналізу навантаження. Жоден із розглянутих інструментів не поєднує в єдиному застосунку облік дисциплін, оцінку трудомісткості, аналіз перевантаження, автоматизований розподіл задач у часі та формування пояснювальних рекомендацій, що підтверджує наявність функціональної прогалини та обґрунтовує доцільність розроблення власної системи.

На основі виявлених потреб сформовано систему функціональних і нефункціональних вимог. До функціональних належать: керування дисциплінами та завданнями, розрахунок сумарного навантаження, автоматизований розподіл задач по днях, формування текстових рекомендацій і забезпечення взаємодії через REST API. Нефункціональні вимоги охоплюють швидкість відгуку інтерфейсу, простоту введення даних, структурованість API, коректну валідацію та відповідальне використання зовнішніх інтелектуальних сервісів.

Обґрунтовано вибір клієнт-серверної архітектури на основі React, FastAPI та PostgreSQL як такої, що забезпечує чіткий розподіл відповідальності між рівнями системи, природну розширюваність і можливість незалежного тестування компонентів. Визначено роль Gemini API як допоміжного засобу природномовної інтерпретації числових результатів алгоритму планування – підхід, що поєднує

детермінованість власної бізнес-логіки з комунікативними можливостями генеративних моделей і забезпечує прозорість системи для кінцевого користувача.

Результати першого розділу є основою для проєктування архітектури системи, розроблення структури бази даних та реалізації алгоритму планування навчального навантаження, що розглядаються в наступних розділах.

2 АРХІТЕКТУРА ТА ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1 Загальна архітектура та технологічний стек

Проєктування архітектури інформаційної системи розпочинається з вибору базової структурної моделі, яка визначає, як саме будуть розподілені обов'язки між компонентами застосунку і яким чином вони взаємодіятимуть між собою. Для розроблюваної системи було обрано багаторівневу клієнт-серверну архітектуру з явним розмежуванням рівня представлення, рівня бізнес-логіки та рівня збереження даних. Такий поділ є усталеною практикою у веброзробці і забезпечує кілька практично важливих властивостей: незалежність розробки та тестування окремих рівнів, можливість масштабування кожного компонента без впливу на решту системи, а також спрощення супроводу коду в умовах поступового розширення функціональності.

Клієнтська частина системи реалізована засобами React – бібліотеки для побудови компонентних інтерфейсів, яка забезпечує реактивне оновлення представлення при зміні стану застосунку. Це означає, що будь-яка зміна даних – додавання нового завдання, оновлення статусу виконання або отримання результатів аналізу навантаження – відображається в інтерфейсі миттєво, без повного перезавантаження сторінки. Компонентний підхід React дозволяє будувати інтерфейс із незалежних повторно використовуваних блоків, що полегшує підтримку коду та внесення змін до окремих частин інтерфейсу без ризику порушити роботу інших

елементів.

Для управління глобальним станом застосунку використовується Redux. У контексті системи планування навантаження стан є досить складним: він охоплює список дисциплін, перелік завдань з усіма атрибутами, результати аналізу навантаження, налаштування доступного часу та поточний стан інтерфейсу. Без централізованого сховища передача цих даних між компонентами перетворилась би на заплутану мережу пропсів і колбеків. Redux вирішує цю проблему, надаючи єдине передбачуване джерело правди для всього застосунку, що суттєво спрощує відлагодження і тестування клієнтської логіки. Tailwind CSS забезпечує оформлення інтерфейсу через утилітарні класи, що дозволяє реалізувати адаптивний дизайн без написання окремих файлів стилів і прискорює процес верстки компонентів.

Серверна частина побудована на основі FastAPI – сучасного Python-фреймворку для розробки вебсервісів, який поєднує високу продуктивність, автоматичну валідацію вхідних даних через Pydantic-схеми та вбудовану генерацію інтерактивної документації у форматі Swagger UI. Остання особливість є практично цінною в умовах активної розробки: кожен реалізований ендпоінт одразу стає доступним для ручного тестування без додаткових інструментів. FastAPI також підтримує асинхронну обробку запитів, що є важливим при інтеграції із зовнішніми сервісами – зокрема, при очікуванні відповіді від Gemini API серверний процес не блокується і може обробляти інші запити.

PostgreSQL обрано як реляційну базу даних з кількох міркувань. По-перше, предметна область системи має чітко виражену реляційну структуру: дисципліни містять завдання, завдання мають атрибути, користувачі мають налаштування – усі ці зв'язки природно описуються через зовнішні ключі та нормалізовані таблиці. По-друге, PostgreSQL забезпечує надійність транзакцій і цілісність даних, що критично важливо при операціях оновлення статусів і перерахунку навантаження. По-третє, підтримка складних агрегаційних запитів дозволяє виконувати частину аналітичних

розрахунків безпосередньо на рівні бази даних, знижуючи обсяг даних, що передаються між рівнями системи.

Призначення технологій, використаних у системі, наведено в таблиці 2.1.в таблиці 2.1.

Таблиця 2.1 – Призначення технологій у системі

Технологія	**Роль у системі**
React	побудова компонентного інтерфейсу користувача
Redux	керування глобальним станом застосунку
Tailwind CSS	оформлення сторінок і адаптивна верстка
FastAPI	реалізація серверної логіки та REST API
PostgreSQL	збереження структурованих навчальних даних
REST API	обмін даними між клієнтом і сервером
Власний алгоритм	розрахунок навантаження і побудова плану
Gemini API	генерація текстових рекомендацій
Jest	тестування клієнтської частини

Окремої уваги потребує місце Gemini API в загальній архітектурі системи. Принципово важливо, що цей сервіс не є центральним компонентом і не замінює власну логіку планування – він виконує допоміжну функцію на завершальному етапі обробки запиту. Серверна частина спочатку самостійно розраховує всі необхідні показники: сумарне навантаження по днях, список задач із ризиком невиконання, рекомендований розподіл роботи. Лише після цього агреговані результати передаються до Gemini API з запитом на формування природномовного коментаря.

Така послідовність забезпечує детермінованість і відтворюваність основних розрахунків, тоді як генеративна модель відповідає виключно за комунікативну складову – перетворення числових показників на зрозумілу пораду.

Взаємодія між клієнтською і серверною частинами організована через REST API, де кожна функціональна можливість системи відповідає конкретному HTTP-ендпоінту. Вибір REST як протоколу взаємодії обумовлений його широкою підтримкою, простотою відлагодження та відповідністю задачам системи: операції над ресурсами – дисциплінами, завданнями, результатами аналізу – природно описуються через стандартні HTTP-методи GET, POST, PATCH і DELETE. Кожен ендпоінт повертає структуровану JSON-відповідь із відповідним статус-кодом, що спрощує обробку результатів на клієнті та забезпечує зрозумілу діагностику помилок.

Структурна схема архітектури інформаційної системи планування навчального навантаження наведена на рис. 2.1, діаграма компонентної архітектури – на рис. 2.2. Схеми демонструють основні функціональні модулі системи, зв'язки між ними та місце кожної технології у процесі обробки, збереження й візуалізації навчальних даних.

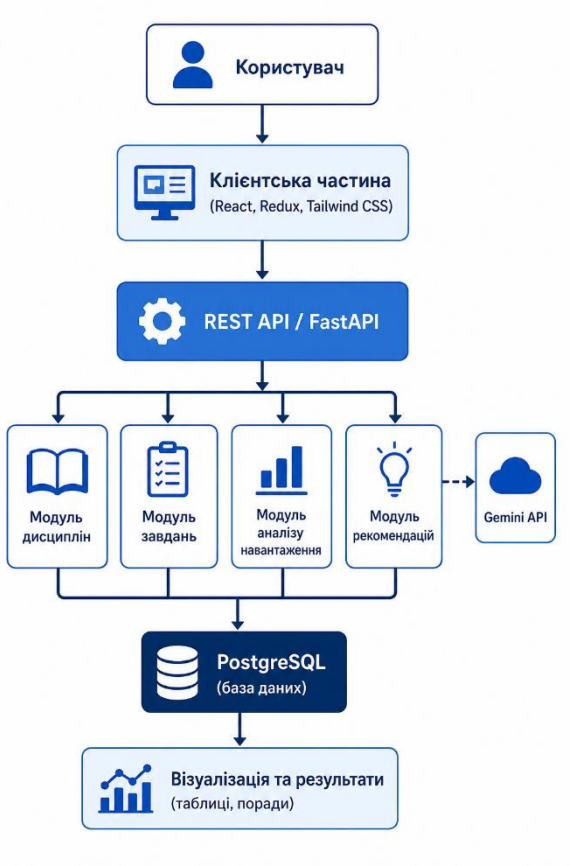

Рисунок 2.1 – Структурна схема архітектури інформаційної системи планування навчального навантаження

Структурна схема архітектури інформаційної системи планування навчального навантаження відображає багаторівневу організацію вебзастосунку та взаємодію його основних компонентів. На верхньому рівні розташований користувач, який здійснює роботу із системою через клієнтську частину – вебінтерфейс, реалізований засобами React, Redux і Tailwind CSS. Клієнтська частина забезпечує введення вихідних даних, перегляд списків дисциплін і завдань, роботу з формами, а також відображення результатів аналізу навантаження та сформованих рекомендацій.

Комунікація між інтерфейсом користувача та серверною частиною здійснюється через REST-інтерфейс, реалізований у середовищі FastAPI. Серверна частина приймає запити від клієнта, виконує валідацію вхідних даних, інкапсулює бізнес-логіку, координує роботу функціональних модулів та формує структуровані

відповіді для подальшого відображення у вебзастосунку. Основу серверної логіки становить набір модулів, кожен з яких відповідає за свій аспект предметної області.

Модуль дисциплін забезпечує збереження та обробку інформації про навчальні дисципліни, їхні характеристики, тривалість та взаємозв’язки з іншими елементами навчального процесу. Модуль завдань відповідає за управління окремими навчальними активностями, дедлайнами, обсягом робіт та прив’язкою завдань до відповідних дисциплін. На основі цих даних працює модуль аналізу навантаження, який виконує розрахунок обсягу навчальної роботи за визначені періоди, виявляє перевантажені дні та тижні, обчислює агреговані показники та готує їх для подальшої оптимізації.

Отримані результати аналізу передаються до модуля рекомендацій, який формує варіанти коригування навчального плану з урахуванням заданих обмежень і цілей користувача. Для підвищення інформативності та зрозумілості повідомлень модуль рекомендацій взаємодіє із зовнішнім інтелектуальним сервісом Gemini API. На основі узагальнених показників, розрахованих серверною частиною, цей сервіс генерує текстові пояснення та поради, що подаються користувачу у зручній для сприйняття формі.

Усі функціональні модулі працюють з єдиною реляційною базою даних PostgreSQL, яка використовується для надійного збереження структурованих навчальних даних, включаючи відомості про дисципліни, завдання, користувацькі налаштування та результати аналізу навантаження. На заключному етапі дані, отримані із серверної частини та бази даних, використовуються для формування візуалізацій і підсумкових результатів. Така структурна організація забезпечує логічне розмежування відповідальностей між шарами системи, спрощує супровід програмного коду та дає можливість масштабування окремих компонентів без порушення роботи всієї інформаційної системи.

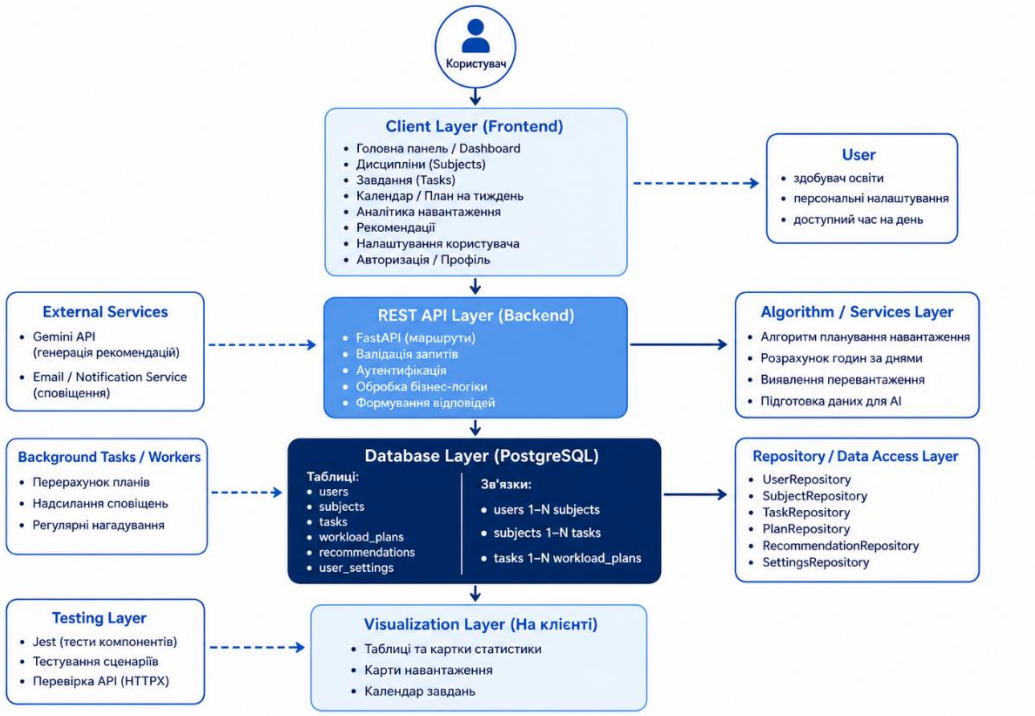

Рисунок 2.2 – Діаграма компонентної архітектури інформаційної системи планування навчального навантаження здобувачів

Діаграма компонентної архітектури, наведена на рис. 2.2, відображає повну структуру інформаційної системи в розрізі логічних рівнів і демонструє, яким чином окремі компоненти взаємодіють між собою в процесі обробки користувацьких запитів. На відміну від структурної схеми, яка описує загальний принцип організації системи, компонентна діаграма розкриває внутрішній склад кожного рівня та характер зв'язків між ними – як синхронних (через REST API), так і асинхронних (через фонові задачі та зовнішні сервіси).

Верхній рівень діаграми представляє кінцевого користувача – здобувача освіти, який взаємодіє із системою через клієнтську частину. Поруч із блоком клієнтської частини виділено окремий компонент профілю користувача, який зберігає персональні налаштування та параметр доступного часу на день. Цей компонент є відносно автономним: він не змінюється при кожному запиті, однак його значення враховується алгоритмом планування під час кожного перерахунку навантаження.

Така архітектурна ізоляція налаштувань дозволяє змінювати логіку їхньої обробки незалежно від основного потоку даних.

Клієнтська частина охоплює вісім функціональних розділів: головну панель, сторінки дисциплін і завдань, календарний перегляд, модуль аналітики навантаження, сторінку рекомендацій, налаштування та авторизацію. Кожен із цих розділів реалізується як окремий React-компонент або набір компонентів, що об'єднуються через маршрутизатор. Стан застосунку, спільний для кількох розділів – наприклад, поточний список дисциплін або результати аналізу – зберігається централізовано у Redux-сховищі, що забезпечує узгодженість даних між різними частинами інтерфейсу без необхідності повторних запитів до сервера.

Рівень REST API є центральним вузлом усієї архітектури: він приймає запити від клієнтської частини, здійснює їхню валідацію, координує виконання бізнес-логіки та формує структуровані відповіді. FastAPI автоматично перевіряє відповідність вхідних даних оголошеним Pydantic-схемам ще до передачі запиту в бізнес-логіку, що суттєво знижує ймовірність потрапляння некоректних значень у базу даних. Аутентифікація реалізована на рівні цього ж шару – через перевірку токена при кожному захищеному запиті, що гарантує розмежування даних між різними користувачами системи.

Праворуч від рівня REST API розташований сервісний шар алгоритмів, який відповідає за виконання всіх аналітичних розрахунків. Його виокремлення в самостійний компонент є усвідомленим архітектурним рішенням: алгоритм планування є найскладнішою та найбільш змінюваною частиною системи, тому його ізоляція дозволяє вдосконалювати логіку розрахунків без впливу на маршрутизацію запитів або структуру бази даних. Сервісний шар отримує агреговані дані від рівня доступу до даних, виконує розрахунок годинного навантаження за днями, виявляє перевантажені відрізки тижня та формує структуровані показники, готові для передачі до модуля рекомендацій або безпосередньо до клієнтської частини.

Ліворуч від рівня REST API розташований блок зовнішніх сервісів, до якого належать Gemini API та сервіс сповіщень. Взаємодія з Gemini API є однонаправленою: система надсилає агрегований запит із числовими показниками навантаження і отримує у відповідь текстовий коментар. Жодні персональні дані користувача при цьому не передаються – лише знеособлені агреговані значення, що відповідає принципу мінімальної достатності при роботі із зовнішніми інтелектуальними сервісами. Сервіс сповіщень використовується для надсилання нагадувань про наближення дедлайнів і може бути реалізований через електронну пошту або push-повідомлення залежно від конфігурації розгортання.

Центральне місце на рівні збереження даних займає PostgreSQL, яка містить шість основних таблиць: users, subjects, tasks, workload_plans, recommendations та user_settings. Між таблицями встановлені реляційні зв'язки типу «один до багатьох»: один користувач може мати кілька дисциплін, кожна дисципліна – кілька завдань, кожне завдання – кілька записів у плані навантаження. Така нормалізована структура забезпечує цілісність даних і дозволяє ефективно виконувати агрегаційні запити при розрахунку навантаження.

Доступ до бази даних здійснюється виключно через рівень репозиторіїв, який містить шість класів: UserRepository, SubjectRepository, TaskRepository, PlanRepository, RecommendationRepository та SettingsRepository. Кожен репозиторій інкапсулює SQL-запити, специфічні для відповідної сутності, що унеможливлює розпорошення логіки роботи з базою даних по всій кодовій базі. Такий підхід відповідає патерну Repository і суттєво спрощує тестування бізнес-логіки – оскільки репозиторії можна замінити тестовими заглушками без зміни сервісного шару.

Ліворуч від рівня бази даних розташований блок фонових задач, який відповідає за операції, що не потребують негайного результату для користувача: перерахунок планів при зміні вхідних параметрів, надсилання сповіщень і виконання регулярних нагадувань. Винесення цих операцій у фонові процеси дозволяє не блокувати обробку

основних запитів і забезпечує стабільну швидкість відгуку інтерфейсу навіть при виконанні обчислювально інтенсивних перерахунків.

Нижній рівень діаграми представляє шар візуалізації, реалізований на стороні клієнта. Він охоплює таблиці та картки статистики, карти навантаження і календарний перегляд завдань. Окремо позначений блок тестування включає Jest для перевірки клієнтських компонентів, тестування сценаріїв взаємодії та перевірку API через HTTPX – інструмент, що дозволяє надсилати тестові HTTP-запити безпосередньо до FastAPI без запущеного клієнтського інтерфейсу.

Описана компонентна архітектура в сукупності забезпечує чіткий розподіл відповідальностей між рівнями, мінімальну міжкомпонентну залежність і достатню гнучкість для поступового розширення функціональності системи без необхідності переписувати вже реалізовані модулі.

2.2 Проєктування бази даних PostgreSQL

База даних є фундаментальним компонентом інформаційної системи, оскільки саме вона забезпечує надійне збереження всіх структурованих даних, що циркулюють між рівнями застосунку. Від якості проєктування схеми залежить не лише коректність роботи алгоритму планування, а й продуктивність запитів, зручність супроводу коду та можливість масштабування системи в майбутньому. Тому проєктування схеми розпочиналось не з визначення таблиць, а з аналізу предметної області – виявлення сутностей, їхніх атрибутів і характеру зв'язків між ними.

Для зберігання даних обрано реляційну модель, яка найкраще відповідає структурі предметної області. Навчальний процес має чітку ієрархію: користувач → дисципліни → завдання → фрагменти плану.Ця ієрархія природно відображається через зовнішні ключі та зв'язки типу «один до багатьох». Використання реляційної моделі забезпечує декілька важливих властивостей: референсну цілісність – неможливість створити завдання без прив'язки до існуючої дисципліни; каскадність

операцій – автоматичне видалення пов'язаних записів при видаленні батьківської сутності; а також можливість ефективного виконання агрегаційних запитів через механізми JOIN і GROUP BY, що є критичним для розрахунку сумарного навантаження.

У межах системи виділено шість основних таблиць, кожна з яких відповідає окремій сутності предметної області. Основні сутності бази даних наведено в таблиці 2.2.

Таблиця 2.2 – Основні сутності бази даних

Сутність	**Ключові поля**	**Призначення**
users	id, name, email	ідентифікація користувача системи
subjects	id, user_id, title, color	збереження навчальних дисциплін
tasks	id, subject_id, title, deadline, priority, difficulty, estimated_hours, status	облік навчальних завдань
workload_plans	id, task_id, plan_date, planned_hours, overload_flag	збереження розрахованого плану
recommendations	id, user_id, created_at, text	збереження текстових рекомендацій
user_settings	user_id, daily_hours, weekend_mode	персональні налаштування планування

Таблиця users є кореневою сутністю всієї схеми – до неї через зовнішні ключі прив'язані дисципліни, рекомендації та налаштування. Вона зберігає мінімально необхідний набір атрибутів: ідентифікатор, ім'я та електронну адресу, яка використовується для аутентифікації. Така мінімалістична структура є свідомим рішенням: зайві персональні дані не зберігаються, що знижує ризики, пов'язані із

захистом приватності.

Таблиця subjects описує навчальні дисципліни користувача. Поле color дозволяє призначити кожній дисципліні індивідуальний колір, що використовується для візуального розмежування завдань різних предметів у календарному поданні. Зовнішній ключ user_id забезпечує ізоляцію даних між різними користувачами: запит на отримання дисциплін завжди фільтрується за ідентифікатором поточного користувача, що унеможливлює доступ до чужих даних.

Таблиця tasks є центральною з точки зору бізнес-логіки, оскільки містить усі атрибути, необхідні для роботи алгоритму планування. Поля priority та difficulty зберігаються як числові значення за шкалою від одного до трьох, що дозволяє алгоритму виконувати арифметичні операції без додаткового перетворення типів. Поле estimated_hours містить орієнтовну тривалість виконання завдання у годинах і є основою для розрахунку навантаження. Поле status приймає одне з чотирьох значень: pending, in_progress, completed або cancelled – що дозволяє фільтрувати активні завдання при побудові плану й не включати вже виконані задачі в розрахунок. Поле deadline зберігається у форматі дати без часового компонента, оскільки планування в системі здійснюється з точністю до дня.

Таблиця workload_plans зберігає результати роботи алгоритму у вигляді конкретних записів: для кожної задачі і кожної дати зберігається кількість запланованих годин та булевий прапорець overload_flag, який встановлюється у значення true, якщо сумарне навантаження на цей день перевищує доступний час користувача. Така структура дозволяє, з одного боку, швидко відтворити повний план для будь-якого часового діапазону, а з іншого – агрегувати навантаження за днями або дисциплінами без повторного запуску алгоритму. При кожному перерахунку плану відповідні записи в цій таблиці оновлюються, забезпечуючи актуальність даних.

Таблиця recommendations зберігає текстові поради, згенеровані Gemini API на основі результатів аналізу навантаження. Поле created_at дозволяє відстежувати

хронологію рекомендацій і надавати користувачеві не лише поточну пораду, а й порівнювати її з попередніми. Зберігання рекомендацій у базі даних, а не лише в пам'яті сервера, дає змогу уникнути повторних дорогих запитів до зовнішнього API при перезавантаженні сторінки.

Таблиця user_settings має відношення «один до одного» з таблицею users: для кожного користувача існує рівно один запис із персональними параметрами планування. Поле daily_hours визначає стандартну кількість годин, доступних для навчання у звичайний день, а поле weekend_mode є булевим прапорцем, який вказує, чи враховуються вихідні дні при побудові плану. Ці параметри задаються користувачем один раз у налаштуваннях і автоматично застосовуються при кожному перерахунку, що реалізує принцип мінімального введення даних, обґрунтований у першому розділі.

Під час проєктування схеми особлива увага приділялась уникненню дублювання інформації. Назва дисципліни зберігається виключно в таблиці subjects і ніде більше не повторюється – всі пов'язані таблиці містять лише числовий ідентифікатор subject_id. Це означає, що при зміні назви дисципліни достатньо оновити один запис, і зміна автоматично відобразиться у всіх пов'язаних контекстах. Для забезпечення каскадного видалення визначено відповідну поведінку зовнішніх ключів: при видаленні дисципліни автоматично видаляються всі пов'язані завдання, а при видаленні завдання – відповідні записи в таблиці workload_plans. Це унеможливлює появу «висячих» записів і підтримує цілісність схеми без ручного очищення даних.

Для забезпечення прийнятної швидкості виконання запитів на таблицях визначено індекси за полями, що найчастіше використовуються в умовах фільтрації. Індекс на полі tasks.subject_id прискорює отримання всіх завдань конкретної дисципліни, індекс на tasks.deadline – фільтрацію за дедлайном при побудові плану, а індекс на workload_plans.plan_date – агрегацію навантаження за конкретну дату або діапазон дат. Без цих індексів кожен запит на розрахунок тижневого навантаження

потребував би повного перебору таблиці, що критично знижувало б продуктивність при збільшенні кількості записів.

PostgreSQL є оптимальним вибором для цього проєкту з кількох причин. По-перше, вона забезпечує повну підтримку транзакцій і властивостей ACID, що критично важливо при операціях одночасного оновлення плану та статусів завдань. По-друге, PostgreSQL підтримує тип даних JSONB, який за необхідності може бути використаний для зберігання гнучких метаданих без зміни схеми. По-третє, її інтеграція з Python-екосистемою через бібліотеки SQLAlchemy та asyncpg є добре задокументованою і широко перевіреною в промислових застосунках, що знижує ризики при реалізації та подальшому супроводі системи.

2.3 Реалізація клієнтської частини на React, Redux і Tailwind CSS

Клієнтська частина інформаційної системи є основним середовищем взаємодії здобувача освіти із застосунком і визначає те, наскільки зручним і зрозумілим буде процес планування навчального навантаження. Структуру клієнтської частини спроєктовано відповідно до функціональних вимог системи: вона містить головну панель, сторінку дисциплін, сторінку завдань, календарний перегляд, аналітичну панель і блок рекомендацій. Кожна з цих частин реалізована у вигляді окремого React-компонента або набору компонентів, що об'єднуються через клієнтський маршрутизатор. Компонентний підхід React дозволяє розробляти, тестувати та вдосконалювати кожну частину інтерфейсу незалежно, не ризикуючи порушити роботу інших розділів застосунку.

Redux застосовується для централізованого управління станом, який є спільним для кількох компонентів або сторінок. До такого стану належать дані авторизованого користувача, список дисциплін, список завдань, поточний стан фільтрів і сортування, результати аналізу навантаження та індикатори завантаження даних. Централізоване сховище усуває потребу передавати дані між компонентами через ланцюги пропсів,

що суттєво спрощує архітектуру інтерфейсу та зменшує обсяг дублювання коду. Після створення або редагування завдання диспетчеризується відповідна Redux-дія, яка оновлює стан у сховищі, і всі компоненти, підписані на цей стан, автоматично отримують актуальні дані без повторного HTTP-запиту до сервера. Це особливо важливо для синхронізації між календарним переглядом і аналітичною панеллю: обидва компоненти підписані на одну й ту саму частину сховища та відображають узгоджені дані в реальному часі.

Взаємодія клієнтської частини із серверним API реалізована через асинхронні Redux-thunk-дії, кожна з яких відповідає за конкретний тип HTTP-запиту. Така організація забезпечує централізований контроль над усіма мережевими операціями застосунку: стан завантаження, отримані дані та помилки зберігаються в Redux-сховищі й доступні будь-якому компоненту без дублювання логіки запитів. Типи клієнтських запитів до серверного API, їхні цільові ендпоінти та призначення наведено в таблиці 2.3.

Таблиця 2.3 – Типи клієнтських запитів до серверного API

Redux-дія	**HTTP-метод**	**Ендпоінт**	**Призначення**
fetchSubjects	GET	/api/subjects	отримання списку дисциплін при завантаженні сторінки
createSubject	POST	/api/subjects	створення нової дисципліни через форму
fetchTasks	GET	/api/tasks	отримання повного списку активних завдань
createTask	POST	/api/tasks	збереження нового завдання після заповнення форми
updateTask	PATCH	/api/tasks/{id}	оновлення статусу, пріоритету або тривалості завдання
analyzeWorkload	POST	/api/workload/analyze	запуск розрахунку навантаження після змін у завданнях

Кінець таблиці 2.3

fetchWeekPlan	GET	/api/workload/week	отримання готового плану для відображення в календарі
generateRecommendation	POST	/api/recommendations/generate	запит текстової рекомендації після аналізу

Кожна Redux-дія містить три стани виконання: pending, fulfilled та rejected. У стані pending відповідний компонент відображає індикатор завантаження, у стані fulfilled – отримані дані, у стані rejected – повідомлення про помилку з деталями, поверненими сервером. Такий підхід гарантує, що користувач завжди отримує зрозумілий зворотний зв'язок щодо стану виконання операції, незалежно від її результату.

Tailwind CSS використовується для оформлення компонентів і побудови адаптивної верстки. Утилітарний підхід цього фреймворку дозволяє застосовувати стилі безпосередньо в JSX-розмітці без створення окремих CSS-файлів для кожного компонента, що прискорює розробку та спрощує синхронізацію між структурою і стилями. Для навчального планувальника особливо важливо, щоб інтерфейс був лаконічним та інформативним одночасно: головна панель відображає картки з кількістю активних завдань, сумарними годинами на поточний тиждень, найбільш завантаженим днем і коротким текстом поточної рекомендації. Така панель дозволяє користувачеві оцінити загальний стан навчального плану без необхідності переходити до інших розділів застосунку.

Адаптивність інтерфейсу забезпечується через систему брейкпоінтів Tailwind: на мобільних пристроях сторінки переходять у однаколоночне подання, картки дисциплін і завдань стають повноширинними, а навігація згортається у компактне меню. Це дозволяє користувачеві переглядати поточний план і статуси завдань із мобільного пристрою без втрати функціональності. Форми введення даних спроєктовано з урахуванням принципу мінімального когнітивного навантаження:

поля згруповані за логічними блоками, обов'язкові поля позначені явно, а клієнтська валідація спрацьовує при спробі відправити форму з некоректними значеннями, відображаючи конкретні підказки поруч із відповідними полями, а не загальне повідомлення про помилку.

Сторінка календарного перегляду формує тижневу сітку, де кожна клітинка відображає сумарне навантаження на відповідний день у годинах і кольорово маркує перевантажені дні на основі прапорця overload_flag, отриманого від сервера. Клацання на конкретний день розкриває перелік завдань, запланованих на цю дату, із зазначенням кількості годин по кожному. Аналітична панель відображає агреговані показники поточного плану: загальну кількість годин на тиждень, розподіл навантаження за дисциплінами у вигляді відсоткових часток, кількість перевантажених днів і перелік завдань із найближчими критичними дедлайнами. Ці дані отримуються із Redux-сховища, куди їх записує дія analyzeWorkload після успішного завершення серверного розрахунку.

2.4 Реалізація серверної частини на FastAPI та REST API

Серверна частина інформаційної системи реалізована засобами FastAPI і забезпечує повний цикл обробки клієнтських запитів: від приймання та валідації вхідних даних до координації роботи функціональних модулів і повернення структурованих відповідей. Організацію серверного коду побудовано за трирівневою схемою: рівень маршрутів відповідає за маршрутизацію HTTP-запитів і визначення ендпоінтів, сервісний рівень інкапсулює бізнес-логіку та координує взаємодію між модулями, а рівень репозиторіїв забезпечує доступ до таблиць PostgreSQL через параметризовані SQL-запити. Такий поділ унеможливлює змішування відповідальностей і спрощує тестування кожного рівня незалежно.

Аутентифікація реалізована на рівні маршрутизатора через перевірку токена при кожному захищеному запиті. Це гарантує, що будь-який запит до захищених

ресурсів без дійсного токена отримує відповідь зі статусом 401 ще до передачі в сервісний шар. Для зв'язку між рівнями застосунку використовуються Pydantic-схеми, які окремо визначають формат вхідних даних запиту і формат вихідних даних відповіді. Це дозволяє контролювати, які саме поля повертаються клієнту, і уникати випадкового витоку внутрішніх атрибутів моделей бази даних. Схему взаємодії модулів серверної частини наведено на рисунку 2.3.

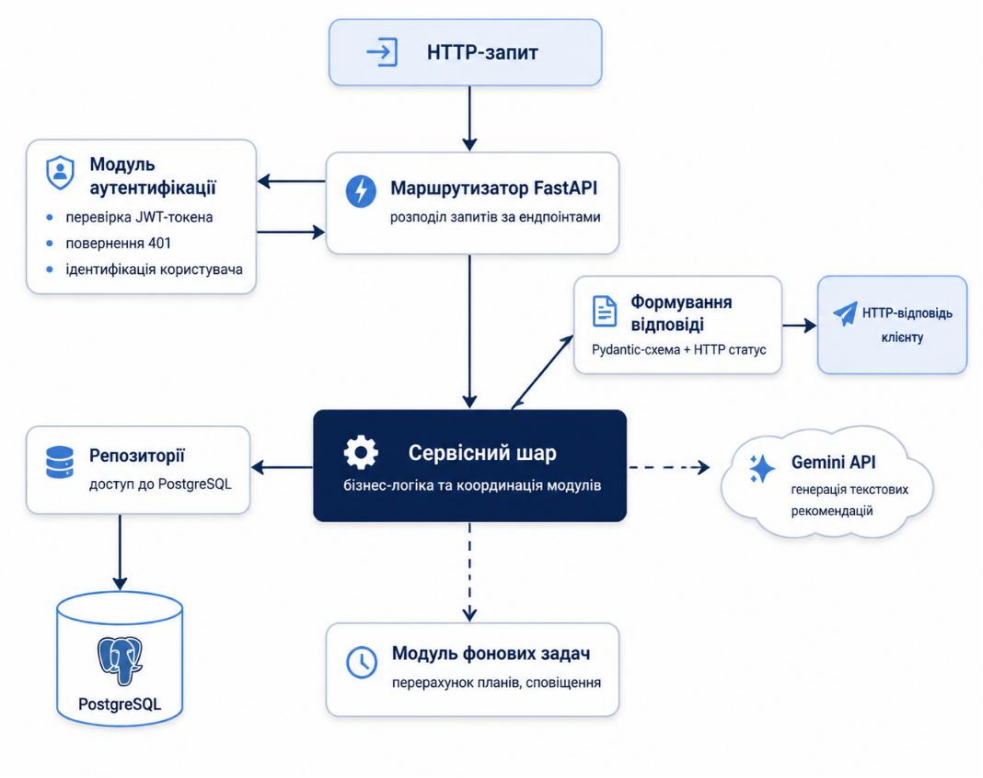

Рисунок 2.3 – Схема взаємодії модулів серверної частини FastAPI

Як видно зі схеми, центральним елементом є сервісний шар, який координує роботу всіх інших модулів: репозиторіїв для доступу до бази даних, модуля аутентифікації, зовнішнього сервісу Gemini API та модуля фонових задач. Взаємодія з Gemini API реалізована в окремому сервісному модулі, який формує запит із знеособлених агрегованих показників навантаження і повертає текстову рекомендацію. Виокремлення цього модуля в самостійний компонент дозволяє за

необхідності замінити зовнішній сервіс або додати резервний варіант без зміни логіки основних маршрутів.

Фонові задачі – перерахунок планів і надсилання сповіщень – виконуються асинхронно і не блокують обробку основних запитів, що забезпечує стабільний час відгуку інтерфейсу незалежно від навантаження на сервер. Усі результати розрахунків зберігаються в таблиці workload_plans, тому повторні запити до ендпоінту /api/workload/week повертають готові дані без повторного запуску алгоритму. Такий підхід суттєво скорочує час відгуку при навігації між сторінками застосунку та забезпечує узгодженість даних між усіма розділами інтерфейсу.

2.5 Власний алгоритм планування навантаження та Gemini API

Ключовою частиною інформаційної системи є власний алгоритм планування навчального навантаження, який перетворює список завдань, дедлайнів, пріоритетів і доступного часу на реалістичний план виконання. Його логіка побудована таким чином, щоб не обіцяти неможливого: якщо сумарна тривалість завдань перевищує доступний час, система не приховує проблему, а явно позначає ризик перевантаження та пропонує користувачеві переглянути план.

Вхідними даними алгоритму є список активних завдань із параметрами дедлайну, орієнтовної кількості годин, складності, пріоритету та статусу, а також налаштування доступного часу користувача на кожен день. На першому етапі алгоритм відсіює виконані завдання, після чого сортує активні записи за наближенням дедлайну та ваговим коефіцієнтом пріоритету. Ваговий показник визначається з урахуванням кількості днів до дедлайну, числового пріоритету та рівня складності: чим менше днів залишилося і чим вищий пріоритет та складність, тим раніше завдання потрапляє у план.

Формуючи план, алгоритм не намагається розмістити всю тривалість завдання в межах одного дня, а розбиває його на кілька частин, якщо орієнтовний час

перевищує доступний денний ліміт. Після первинного розподілу система підраховує сумарні години за кожним днем і позначає дні, де значення перевищує встановлений ліміт, як перевантажені. Далі виконується спроба балансування: завдання з некритичними дедлайнами переносяться на менш завантажені попередні дні; якщо перенесення неможливе, користувач отримує попередження про ризик невиконання плану в межах доступного часу.

Результатом роботи алгоритму є календарний план і набір аналітичних показників: загальна кількість годин на тиждень, кількість перевантажених днів, найближчі критичні дедлайни, дисципліни з найбільшим навантаженням і перелік завдань, які бажано розпочати раніше. Ці дані зберігаються в таблиці планів навантаження, передаються клієнтській частині та використовуються для відображення календаря й аналітичної панелі.

Gemini API використовується після отримання результатів алгоритму як сервіс генерації текстових пояснень. Сервер формує короткий запит, у якому передається не повний список особистих даних, а лише узагальнені показники: рівень навантаження, кількість перевантажених днів, ризикові дні, сумарні години та рекомендовані перенесення. На основі цих даних сервіс генерує рекомендацію у вигляді поради, наприклад: розпочати складну лабораторну роботу за два дні до дедлайну або перенести менш термінове завдання на день із нижчим навантаженням.

Якщо звернення до Gemini API завершується помилкою або тайм-аутом, система не припиняє роботу, а переходить до внутрішнього набору правил. У такому випадку рекомендації формуються на основі заздалегідь визначених умов: перевищення тижневого ліміту годин, наявності перевантажених днів, близьких дедлайнів і великої кількості задач із високим пріоритетом. Це забезпечує відмовостійкість системи та дозволяє користувачеві отримувати корисні поради навіть за відсутності відповіді від зовнішнього сервісу.

Висновки до розділу 2

У другому розділі було виконано повний цикл архітектурного та проєктного опрацювання інформаційної системи планування навчального навантаження – від вибору загальної структурної моделі до деталізації окремих програмних компонентів. Обґрунтовано застосування багаторівневої клієнт-серверної архітектури з чітким розмежуванням рівня представлення, бізнес-логіки та збереження даних. Такий підхід забезпечує технологічну гнучкість, дає змогу незалежно розвивати клієнтську і серверну частини, а також спрощує подальший супровід і масштабування системи.

На рівні технологічного стека показано, що поєднання React, Redux і Tailwind CSS на клієнті з FastAPI та PostgreSQL на сервері відповідає вимогам до інтерактивності, продуктивності та надійності зберігання навчальних даних. React забезпечує компонентну побудову інтерфейсу та миттєве оновлення представлення, Redux – керування складним глобальним станом без «заплутаних» ланцюжків передавання даних, а Tailwind CSS – швидку розробку адаптивної верстки без громіздких файлів стилів. На серверному боці FastAPI поєднує асинхронну обробку запитів, сувору валідацію вхідних даних та автоматичну документацію REST-інтерфейсу, тоді як PostgreSQL виступає надійною основою для транзакційного зберігання реляційно пов’язаних сутностей навчального процесу.

У межах проєктування схеми бази даних було виділено ключові сутності – користувачі, дисципліни, завдання, план навантаження, рекомендації та налаштування – і визначено між ними зв’язки типу «один до багатьох» та «один до одного». Нормалізована структура, використання зовнішніх ключів, каскадних операцій і цілеспрямовано обраних індексів забезпечують цілісність даних та прийнятну продуктивність навіть за зростання обсягу інформації. Окрему увагу приділено мінімізації дублювання даних і збереженню лише тих атрибутів користувача, які необхідні для роботи системи, що позитивно впливає на безпеку та

усуває зайві ризики, пов’язані з обробкою персональної інформації.

Реалізація клієнтської частини показує, як теоретична архітектура перетворюється на набір практичних інтерфейсів: головна панель, сторінки дисциплін і завдань, календарний перегляд, аналітична панель та блок рекомендацій. Завдяки централізованому стану в Redux усі ці компоненти працюють узгоджено: зміна даних в одному розділі автоматично відображається в інших без додаткових запитів до сервера. Окремо підкреслено важливість адаптивності інтерфейсу та зручності введення даних, що є критичним для реального використання системи здобувачами освіти на різних пристроях.

Серверна частина спроєктована за трирівневою схемою «маршрути – сервіси – репозиторії», що дозволяє ізолювати бізнес-логіку від деталей зберігання даних і структури REST-інтерфейсу. Виокремлення сервісного шару алгоритмів в окремий компонент створює передумови для подальшого розвитку й удосконалення розрахунків без радикальних змін у решті системи. У взаємодії з зовнішнім сервісом Gemini API чітко проведено межу відповідальності: власний алгоритм виконує основні детерміновані обчислення, тоді як генеративна модель використовується для формування зрозумілих текстових рекомендацій. Наявність внутрішніх правил на випадок недоступності зовнішнього сервісу підвищує відмовостійкість і робить систему менш залежною від сторонньої інфраструктури.

Остаточно, у підпункті, присвяченому власному алгоритму планування, було формалізовано підхід до перетворення сирого списку завдань на календарний план із позначенням перевантажених днів, балансуванням навантаження та формуванням набору аналітичних показників. Це поєднання обґрунтованої архітектури, продуманої схеми даних, чітко реалізованої клієнтської та серверної частин і спеціалізованого алгоритму створює цілісну основу для реалізації інформаційної системи, здатної не лише відображати поточний стан навчального навантаження, а й активно допомагати користувачеві в його плануванні та оптимізації.

3 РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

3.1 Реалізація та демонстрація роботи серверної частини

Серверна частина інформаційної системи, архітектура якої була спроєктована у підпункті 2.4, реалізована у вигляді повноцінного набору HTTP-ендпоінтів на основі фреймворку FastAPI. Кожен ендпоінт відповідає за обробку конкретного типу запитів від клієнтської частини і виконує чітко визначений перелік дій: перевірку вхідних даних, виклик відповідного сервісу, отримання результату та формування структурованої відповіді. Завдяки такій організації серверна частина не лише виконує бізнес-логіку, а й забезпечує стабільний і передбачуваний контракт взаємодії між клієнтом і сервером, де формат кожного запиту й відповіді чітко визначений на рівні коду.

Для перевірки коректності реалізованих ендпоінтів використовувався вбудований інтерфейс Swagger/OpenAPI, який FastAPI генерує автоматично на основі описів маршрутів і Pydantic-схем. Цей інтерфейс надає повний перелік усіх доступних ендпоінтів із зазначенням методу HTTP, шляху, очікуваного формату тіла запиту та можливих кодів відповіді. Принциповою перевагою такого підходу є те, що розробник може надсилати реальні запити до сервера безпосередньо з браузера і відразу спостерігати за відповідями у форматі JSON, без необхідності встановлення та налаштування стороннього інструментарію. Це суттєво прискорює процес налагодження та дозволяє одразу виявляти невідповідності між очікуваним і фактичним результатом роботи ендпоінта.

Одним із ключових ендпоінтів системи є POST /api/tasks/, що відповідає за створення нового навчального завдання. Цей ендпоінт безпосередньо реалізує функціональну вимогу FR-2, визначену у підпункті 1.3, – забезпечення повного циклу управління завданнями, включаючи їх створення з повним набором параметрів. При

надходженні запиту FastAPI передає тіло запиту в Pydantic-схему TaskCreate, яка перевіряє наявність усіх обов'язкових полів, відповідність типів даних (наприклад, дата дедлайну повинна бути у форматі ISO 8601), а також допустимість значень перелічуваних полів (priority може набувати значень «low», «medium» або «high», difficulty – «easy», «medium» або «hard»). Якщо хоча б одне поле не відповідає схемі, FastAPI автоматично повертає відповідь із кодом 422 Unprocessable Entity і детальним описом усіх порушень, не передаючи некоректні дані у сервісний шар. Такий механізм захищає бізнес-логіку від обробки неповних або некоректних даних і знімає необхідність реалізовувати ручну валідацію у кожному сервісі окремо.

На рисунку 3.1 зображено процес виконання POST-запиту до ендпоінта /api/tasks/ через інтерфейс Swagger UI. У тілі запиту передаються усі обов'язкові поля: ідентифікатор дисципліни (subject_id), назва завдання (title), дата дедлайну (deadline), пріоритет (priority), складність (difficulty) та орієнтовна тривалість виконання у годинах (estimated_hours). Значення кожного поля наведено в реальному форматі, що відповідає Pydantic-схемі: дедлайн записаний у форматі «РРРР-ММ-ДД», пріоритет – як рядок зі встановленого переліку, а кількість годин – як ціле число

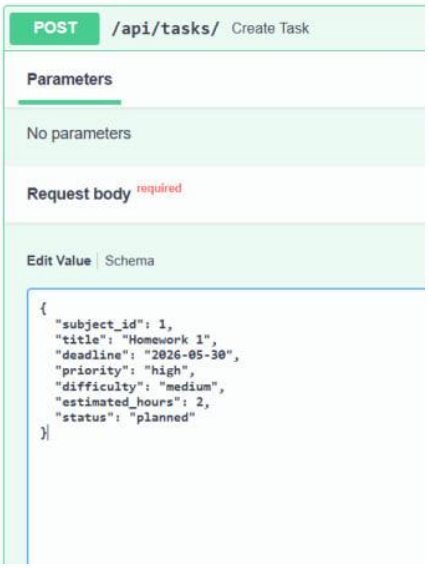

Рисунок 3.1 – Приклад POST-запиту на створення нового завдання

Після того, як схема підтвердила коректність вхідних даних, запит передається до сервісу TaskService. Сервіс виконує логіку, специфічну для операції створення завдання: перевіряє, що вказана дисципліна належить поточному авторизованому користувачеві (щоб запобігти створенню завдань у чужих предметах), і викликає метод репозиторію TaskRepository для збереження нового запису в таблиці tasks бази даних PostgreSQL. Статус нового завдання встановлюється автоматично значенням «planned» – клієнтська частина не передає це поле у запиті, оскільки логіка ініціалізації статусу є відповідальністю серверного сервісу, а не клієнта.

Після успішного збереження запису сервіс формує об'єкт-відповідь відповідно до вихідної Pydantic-схеми TaskResponse, яка відрізняється від вхідної схеми TaskCreate: вона включає автоматично присвоєний ідентифікатор запису (id), поле статусу (status) та мітку часу створення (created_at), але не включає деякі внутрішні атрибути моделі бази даних, не призначені для передачі клієнту. Такий підхід реалізує принцип інформаційного приховування на рівні API: клієнт отримує лише ту підмножину полів, яка необхідна для роботи інтерфейсу, без службових даних, що стосуються виключно рівня збереження.

На рисунку 3.2 зображено відповідь сервера після успішного виконання запиту на створення завдання. Сервер повертає код 200 та JSON-об'єкт, що містить повні дані щойно створеного запису: автоматично присвоєний ідентифікатор, усі передані поля зі збереженими значеннями та автоматично встановлений статус «planned». Наявність ідентифікатора у відповіді є принципово важливою для клієнтської частини: Redux-сховище зберігає цей запис разом із серверним ідентифікатором, що дозволяє у наступних запитах коректно звертатись до конкретного завдання для його редагування, зміни статусу або видалення. на рисунку 3.2.

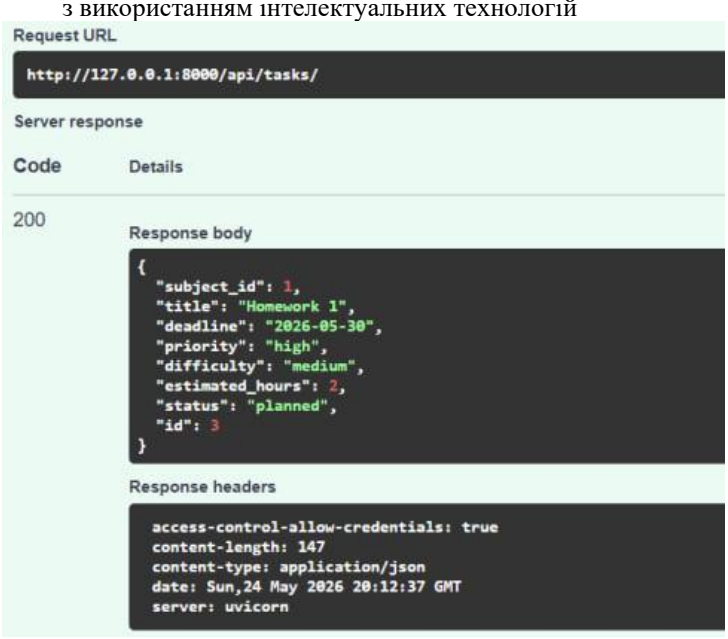

Рисунок 3.2 – Приклад відповіді сервера на створення нового завдання

Аналіз наведених рисунків підтверджує, що реалізована серверна частина повністю відповідає архітектурній моделі, описаній у підпункті 2.4, і коректно виконує свою роль у загальній схемі взаємодії компонентів системи. Весь ланцюжок обробки запиту – від приймання HTTP-запиту через маршрутизатор, валідації вхідних даних схемою, виконання бізнес-логіки у сервісному шарі, збереження запису через репозиторій і аж до формування структурованої JSON-відповіді – реалізований відповідно до спроєктованої архітектури без відхилень. Це забезпечує передбачувану поведінку системи і дозволяє клієнтській частині опиратись на гарантований формат відповідей при реалізації логіки оновлення стану Redux-сховища.

Аналогічним чином були реалізовані та перевірені решта ендпоінтів системи. Ендпоінт PATCH /api/tasks/{id} забезпечує часткове оновлення параметрів завдання, зокрема зміну статусу на «in_progress» або «completed». Ендпоінт POST /api/workload/analyze запускає алгоритм розподілу завдань по днях і повертає сформований календарний план у вигляді впорядкованого за датами масиву об'єктів. Ендпоінт GET /api/workload/week повертає план на поточний тиждень у форматі,

оптимізованому для відображення в інтерфейсі календаря. Ендпоінт POST /api/recommendations/generate ініціює формування текстової рекомендації – або через звернення до Gemini API, або через внутрішній набір правил у разі недоступності зовнішнього сервісу.

3.2 Реалізація інтерфейсу користувача інформаційної системи

Клієнтська частина інформаційної системи реалізована у вигляді односторінкового вебзастосунку на основі бібліотеки React. Усі переходи між розділами застосунку відбуваються без повного перезавантаження сторінки – зміна відображуваного вмісту здійснюється через маршрутизатор на стороні браузера, що забезпечує швидку та плавну навігацію між розділами. Такий підхід є стандартним для сучасних вебзастосунків, орієнтованих на активну взаємодію користувача: замість того, щоб щоразу завантажувати нову сторінку з сервера, браузер лише замінює вміст поточного вікна, зберігаючи усі раніше завантажені дані у пам'яті.

Глобальний стан застосунку – перелік дисциплін, список завдань, поточний план навантаження та отримані рекомендації – зберігається в централізованому Redux-сховищі. Це рішення є ключовим з точки зору узгодженості відображення даних у різних частинах інтерфейсу. Якщо користувач додає нове завдання на сторінці задач, цей запис миттєво стає доступним у календарному поданні та враховується при наступному запуску аналізу навантаження, без необхідності повторно завантажувати дані або вручну синхронізувати стан між сторінками. Аналогічно, позначення завдання як виконаного одразу знімає його з розрахунку – і це відображається у всіх пов'язаних компонентах одночасно. Такий принцип «єдиного джерела правди» є одним із ключових переваг архітектури, описаної у підпункті 2.3, і у реалізації він себе повністю підтвердив.

Робота з системою починається зі створення навчальних дисциплін – структурних одиниць, до яких прив'язуються всі завдання. Рисунок 3.3 демонструє

форму додавання нової дисципліни. Форма є навмисно лаконічною і містить лише два поля: текстове поле для введення назви предмету та інтерактивну палітру для вибору кольорової мітки. Таке рішення обґрунтоване тим, що надмірна кількість параметрів на першому кроці відлякує користувача і знижує ймовірність того, що він взагалі заповнить систему даними. Натомість мінімальна форма дозволяє швидко сформувати базову структуру навчального плану – а деталі додаються пізніше, при створенні конкретних завдань.

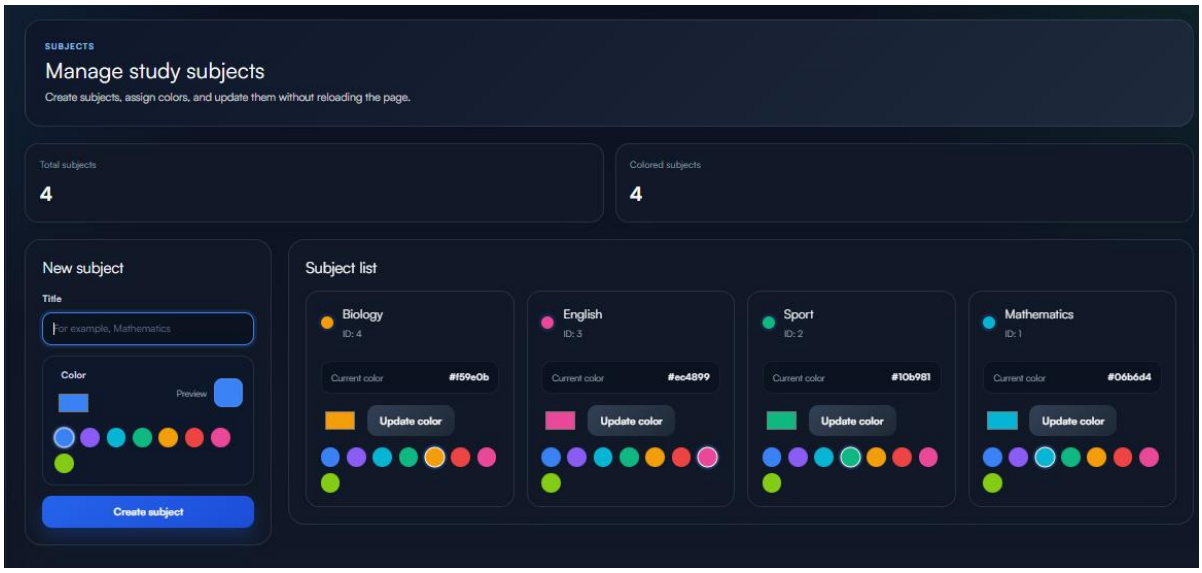

Рисунок 3.3 – Форма додавання нової дисципліни

Колір дисципліни є не лише естетичним елементом – він виконує практичну функцію візуального кодування. У списку завдань, де одночасно можуть відображатися десятки записів із різних предметів, кольорова мітка дозволяє миттєво розпізнати приналежність кожного завдання без необхідності читати назву дисципліни повністю. Аналогічно у календарному поданні різні кольори допомагають зорово оцінити, яка частина тижневого навантаження припадає на кожен предмет – і чи не домінує якась одна дисципліна в конкретному часовому відрізку. Таким чином, кольорове маркування є частиною інформаційної архітектури інтерфейсу, а не просто декором.

Форма реалізована з клієнтською валідацією: кнопка збереження залишається неактивною доти, поки не заповнені обидва поля – назва та колір. Після натискання кнопки клієнтська частина виконує Redux-thunk дію createSubject, яка надсилає POST-запит до серверного ендпоінта /api/subjects/ і після отримання успішної відповіді оновлює Redux-сховище новим записом. Форма автоматично очищується, а щойно додана дисципліна з'являється у списку і стає доступною для прив'язки нових завдань. Такий цикл взаємодії – заповнення форми, збереження, миттєве відображення результату – є єдиним для всіх форм у системі і формує у користувача стабільне очікування щодо поведінки інтерфейсу.

Після формування переліку дисциплін користувач переходить до створення завдань. На рисунку 3.4 зображена форма додавання нового завдання – найбільш розгорнута форма у системі, оскільки саме параметри завдань є основою для роботи алгоритму планування навантаження. Форма містить такі поля: назва завдання, вибір дисципліни зі спадного списку (де кожен пункт відображається разом із кольоровою міткою відповідного предмету), дата дедлайну через вбудований компонент вибору дати, пріоритет виконання (низький, середній або високий), складність (легка, середня або важка) та орієнтована кількість годин, необхідних для виконання. Кожне поле безпосередньо впливає на логіку розподілу: дедлайн визначає крайній термін планування, пріоритет і складність формують ваговий коефіцієнт для сортування завдань, а орієнтована тривалість є основою для розрахунку денного навантаження.

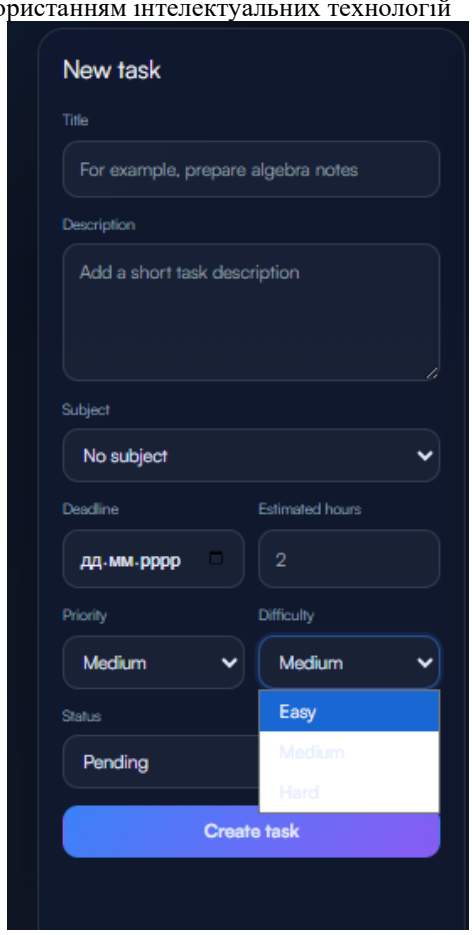

Рисунок 3.4 – Форма створення нового навчального завдання

Важливо зазначити, що вибір значень для полів «пріоритет» та «складність» реалізований не як довільне числове введення, а як список із фіксованих варіантів. Це рішення є усвідомленим: числові шкали (наприклад, від 1 до 10) створюють у користувача відчуття невизначеності – незрозуміло, чи є «7» достатньо важким завданням або лише середнім. Натомість текстові варіанти («легка», «середня», «важка») відповідають природньому способу оцінювання, яким користується студент у повсякденному житті. На рівні алгоритму ці текстові значення відображаються у числові коефіцієнти, однак від користувача ця деталь реалізації прихована.

Клієнтська валідація форми завдання є більш деталізованою порівняно з формою дисципліни, оскільки кількість обов'язкових полів суттєво більша. Якщо користувач натискає кнопку збереження без вибору дисципліни, без вказання

дедлайну або без зазначення орієнтованої тривалості – поряд із відповідними полями з'являються підказки з описом конкретної помилки. Наприклад, при відсутньому дедлайні відображається повідомлення «Вкажіть дату дедлайну», а не загальна фраза «Заповніть обов'язкові поля». Такий рівень деталізації помилок суттєво скорочує час, який користувач витрачає на виправлення форми, оскільки він бачить не лише факт помилки, а й точно знає, що саме потрібно виправити. При цьому запит до серверного ендпоінта не надсилається взагалі до виправлення всіх помилок – перевірка відбувається повністю на боці клієнта.

Після збереження завдання воно відображається у загальному списку. На рисунку 3.5 показана сторінка переліку активних завдань. Для кожного запису одночасно відображаються: назва завдання, кольорова мітка дисципліни з її назвою, дата дедлайну, рівень пріоритету, складність, орієнтована тривалість та поточний статус. Такий рівень деталізації у рядку списку є усвідомленим рішенням – студент, переглядаючи список перед початком тижня, повинен мати можливість оцінити стан усіх завдань без необхідності клікати на кожен запис окремо і відкривати його детальний перегляд. Це відповідає принципу ефективної роботи зі списком задач: огляд усього обсягу має займати секунди, а не хвилини.

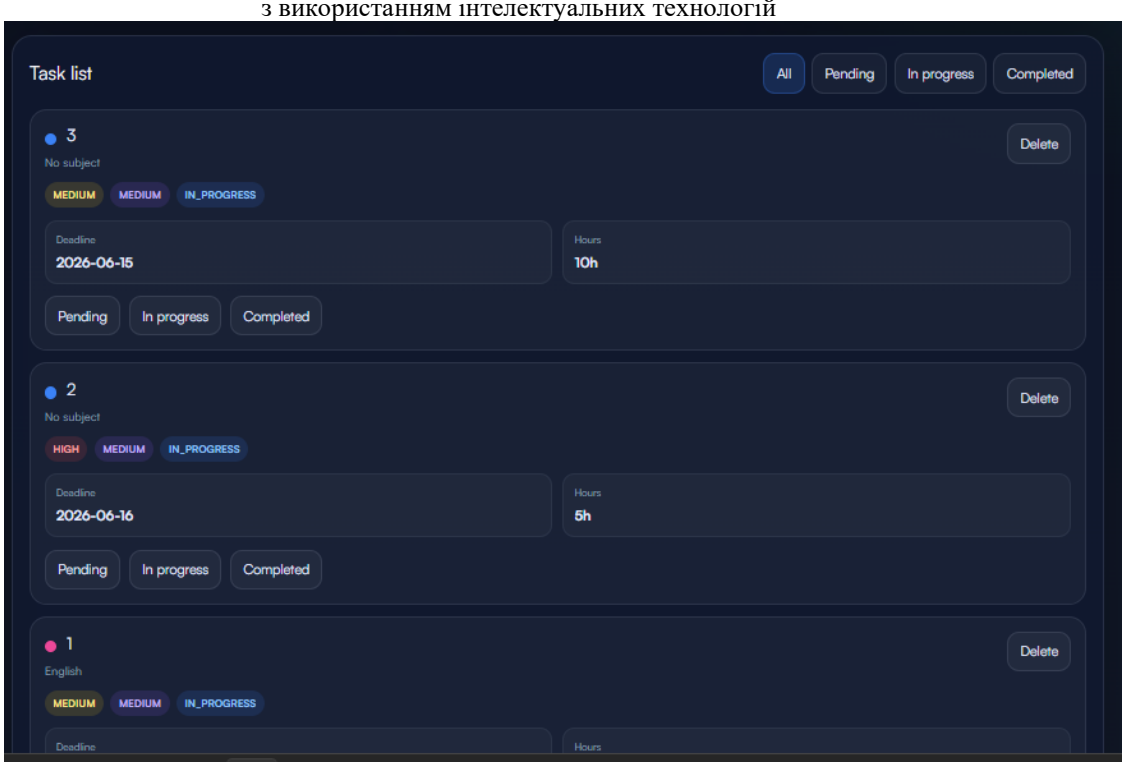

Рисунок 3.5 – Сторінка списку активних завдань із деталізацією параметрів

Список завдань підтримує зміну статусу безпосередньо з перегляду без переходу до форми редагування. Після натискання кнопки зміни статусу клієнтська частина виконує Redux-thunk дію updateTask, яка надсилає PATCH-запит до серверного ендпоінта /api/tasks/{id} з оновленим значенням поля status. Після отримання успішної відповіді Redux-сховище оновлюється, і завдання переходить у новий стан – наприклад, зі статусу «pending» до «in_progress» або «completed». Принципово важливо, що завдання зі статусом «completed» автоматично виключаються з наступних розрахунків навантаження: при запуску алгоритму аналізу сервер вибирає з бази даних лише записи зі статусами «pending» та «in_progress». Завдяки цьому план навантаження завжди відображає реальну картину – вже виконана робота не враховується у розподілі майбутніх годин.

На рисунку 3.6 зображена сторінка налаштувань користувача – розділ, що визначає персональні параметри роботи алгоритму планування. Форма налаштувань

містить чотири групи параметрів. Перша – денний ліміт навчання: тут задається цільова кількість годин (значення, до якого алгоритм прагне при формуванні плану) та максимально допустимий ліміт (перевищення якого позначається алгоритмом як перевантаження і відображається у календарі окремим кольором). Друга – режим планування вихідних: користувач може обрати, чи включати суботу та неділю до розрахунку, чи планувати завдання виключно на будні дні. Третя – часовий пояс, що впливає на коректне відображення дат дедлайнів. Четверта – параметри сповіщень про наближення критичних дедлайнів.

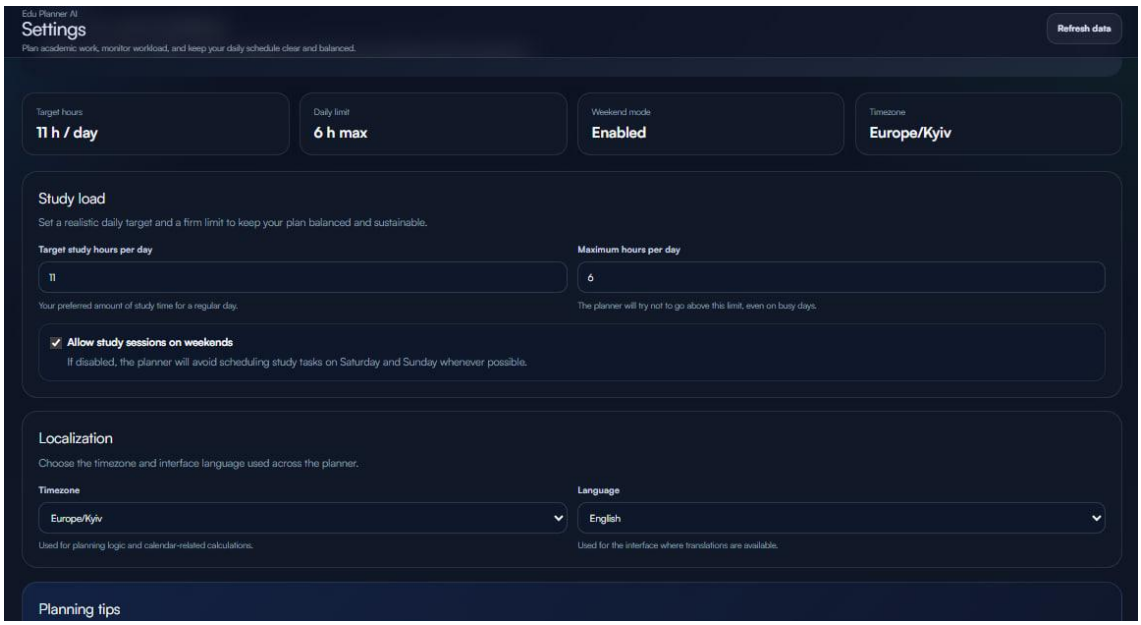

Рисунок 3.6 – Сторінка налаштувань користувача

Кожне поле налаштувань супроводжується коротким поясненням, що описує вплив зміни параметра на результати аналізу. Наприклад, поряд із полем максимального денного ліміту зазначено: «Дні, де сумарне навантаження перевищує це значення, будуть позначені як перевантажені у календарі». Такі пояснення зменшують когнітивне навантаження на користувача – йому не потрібно самостійно здогадуватись, як зміна числа вплине на відображення системи. Після збереження налаштувань інтерфейс пропонує виконати повторний перерахунок плану

навантаження, щоб нові параметри одразу набули чинності у відображенні календаря та блоку рекомендацій – без необхідності чекати наступного ручного запуску аналізу.

Центральним елементом інтерфейсу є сторінка дашборду – головний екран, на який користувач потрапляє після авторизації. На рисунку 3.7 зображена сторінка дашборду у стані, коли план навантаження вже сформований.

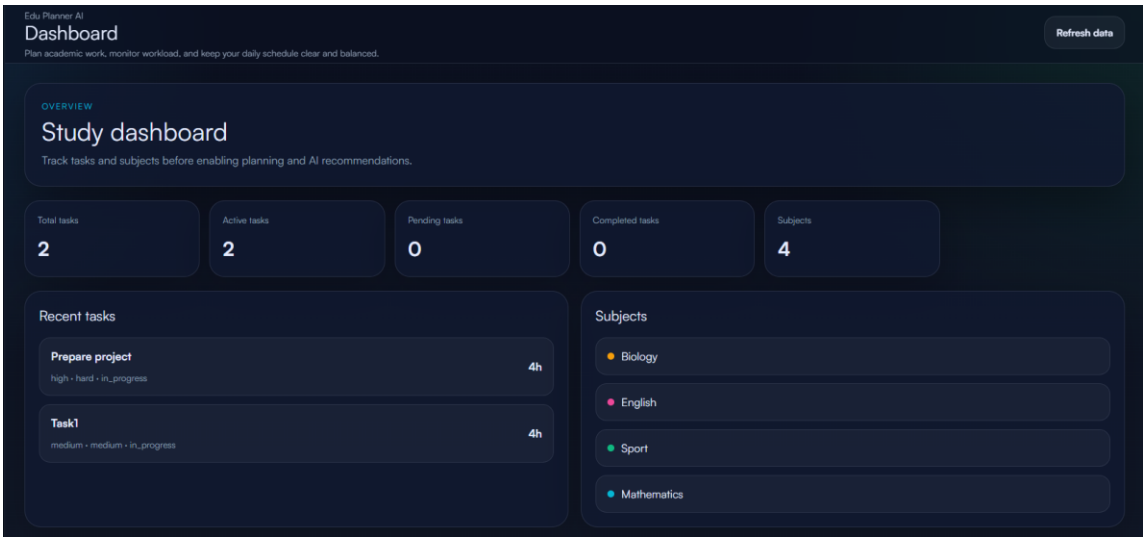

Рисунок 3.7 – Головна аналітична панель (дашборд) інформаційної системи

Центральний блок дашборду складається з панелі метрик, де кожен показник відображається у вигляді окремої картки з назвою та числовим значенням. Серед них: загальна кількість завдань, кількість активних завдань, кількість задач у статусі очікування, кількість виконаних завдань, а також кількість дисциплін. Такий набір показників дозволяє швидко оцінити баланс навантаження: зрозуміти, скільки завдань уже виконано, скільки ще залишилося в роботі, чи не накопичилися «завислі» задачі, які не просуваються до завершення. Нижня частина дашборду поділена на кілька функціональних блоків, що деталізують агреговані показники верхньої панелі. У секції нещодавніх завдань відображаються останні задачі, з якими працював користувач: для кожного запису показано назву, короткий опис, пріоритет та поточний статус виконання. Такий блок виконує роль робочої «стрічки подій» –

студент може швидко повернутися до актуальних задач, не здійснюючи додатковий пошук у повному переліку завдань.

Праворуч розташований блок дисциплін, де в компактному вигляді перелічені всі предмети, заведені в систему. Кожна дисципліна відображається окремим рядком із кольоровим маркером, що відповідає обраному раніше кольору цієї дисципліни, – це полегшує зорове розрізнення предметів у всіх розділах інтерфейсу. Наявність блоку дисциплін на дашборді підкреслює, що планування виконується не для абстрактного набору задач, а в контексті конкретних навчальних напрямів.

Завдяки такій структурі дашборд поєднує зведену аналітику та швидкий доступ до ключових сутностей системи – завдань і дисциплін. Користувач спочатку отримує загальну картину у вигляді агрегованих показників та календаря, а далі може перейти до роботи з окремими записами без зміни контексту. Це дозволяє використовувати дашборд як основний робочий екран, до якого логічно повертатися протягом усього навчального дня, тоді як інші сторінки (список завдань, детальний календар, сторінка рекомендацій) відкриваються вже для поглибленого аналізу або редагування даних

Отже, клієнтська частина інформаційної системи реалізована у вигляді вебзастосунку на основі React, який забезпечує повний цикл роботи користувача з навчальними даними – від створення дисциплін і завдань до перегляду зведених результатів аналізу на дашборді. Описані форми введення, список задач, аналітична панель та сторінка налаштувань демонструють логічно організовану структуру інтерфейсу, що підтримує природну послідовність дій здобувача й надає можливість персоналізувати параметри планування відповідно до індивідуальних потреб користувача.

3.3 Реалізація календарного планування та модуля рекомендацій

Ключовою функціональною складовою розробленої інформаційної системи є модуль календарного планування, який перетворює набір окремих завдань на

узгоджений у часі навчальний розклад. Звичайний список задач дає лише відповідь на запитання «що потрібно зробити», тоді як календарне представлення дозволяє побачити, коли саме варто виконувати кожну задачу, як вони розподіляються в межах найближчих днів і всього місяця та чи не виникають періоди з надмірним навантаженням. Саме тому календар у системі не є другорядним візуальним елементом, а виступає одним з основних інструментів планування, через який користувач сприймає результат роботи алгоритму.

Для формування календарного плану використовуються всі параметри завдань, що зберігаються у базі даних: назва, дисципліна, дедлайн, пріоритет, складність, орієнтовна тривалість виконання, статус, а також персональні налаштування користувача. До таких налаштувань належать бажана кількість годин навчання на день, максимально допустиме денне навантаження, а також можливість або заборона роботи у вихідні. На підставі цих даних серверна частина системи формує набір планових подій, де кожна подія містить дату, заплановану кількість годин і прив’язку до конкретного завдання. У результаті користувач отримує не розрізнений перелік задач, а впорядковану структуру, в якій кожен навчальний блок займає своє місце у часі.

Перший рівень відображення календарного планування реалізовано у вигляді карток найближчих днів. Це подання орієнтоване на короткий часовий горизонт і використовується для щоденної роботи з планом. На сторінці планера у блоці Next 7 days (рисунок 3.8) відображається перелік навчальних блоків, які система запланувала на найближчий тиждень. Для кожного блоку подається номер або назва завдання, дата, кількість годин, а також набір службових позначок, що уточнюють контекст цього інтервалу. До таких позначок належать дата дедлайну, маркер «On deadline day», якщо робота припадає безпосередньо на день здачі, а також індикатор «Manual», якщо цей блок було додано або змінено користувачем вручну. Завдяки такому формату планер показує не лише загальний обсяг роботи, а конкретну послідовність дій на

найближчі дні.

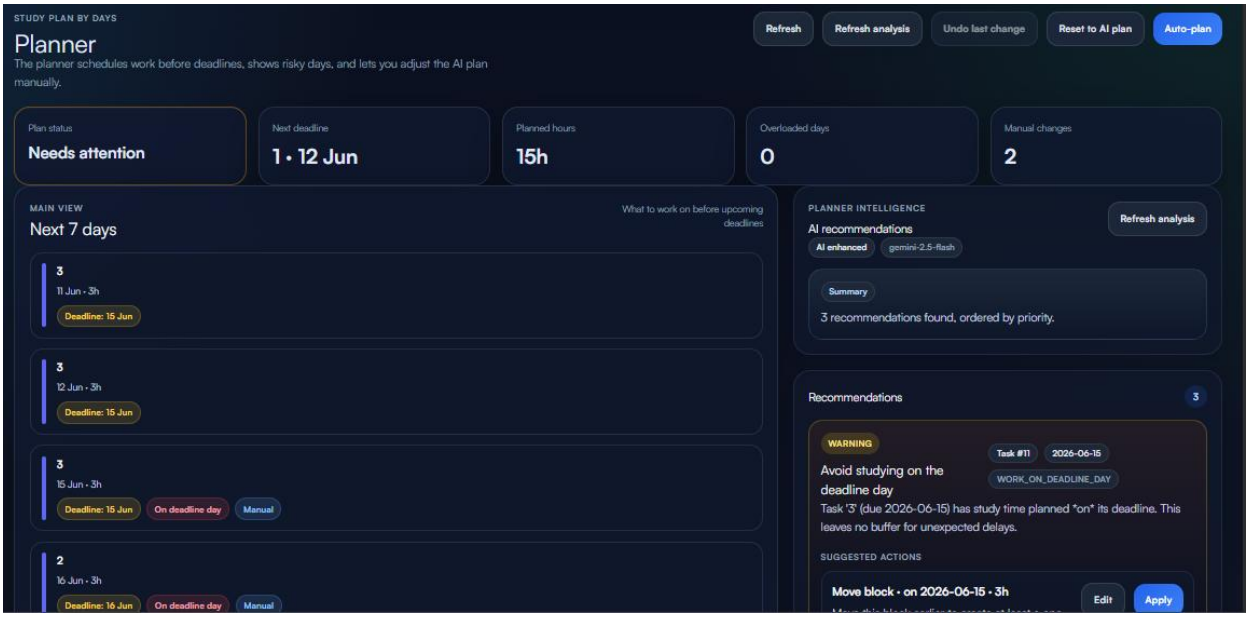

Рисунок 3.8 – Блок планера «Next 7 days» із навчальними картками на найближчі дні

Подання у вигляді карток є особливо зручним для оперативного планування. Користувачеві не потрібно аналізувати весь місяць, якщо його поточна мета – зрозуміти, що саме слід робити сьогодні, завтра або до кінця тижня. Саме тому кожна картка містить лише ту інформацію, яка необхідна для швидкого прийняття рішення: скільки часу потрібно приділити навчанню, до якого дедлайну відноситься цей блок і чи є цей інтервал автоматично згенерованим або відредагованим вручну. Якщо, наприклад, блок позначений як «On deadline day», це одразу сигналізує, що часу до завершення завдання майже не залишилося і такий розподіл є потенційно ризиковим.

Окрему роль у цьому фрагменті планера відіграють ручні зміни. Якщо користувач переносить навчальний блок на іншу дату, змінює його тривалість або додає власний інтервал поза межами автоматичного плану, система фіксує це як ручне втручання і відображає через спеціальну позначку. Завдяки цьому студент не втрачає розуміння, які частини розкладу були сформовані алгоритмом, а які – відкориговані

відповідно до власних потреб. Такий підхід є важливим для прозорості планування: користувач бачить не лише кінцевий результат, а й структуру його походження.

Друге подання календарного модуля реалізовано у вигляді місячного огляду. На відміну від блоку найближчих семи днів, який призначений для короткострокової роботи, місячний календар показує загальну картину розподілу завдань, дедлайнів і навчальних інтервалів у ширшому часовому проміжку. На відповідній сторінці (рисунок 3.9) відображається календарна сітка за поточний місяць із назвами днів тижня та датами. Усередині окремих клітинок показуються навчальні блоки у вигляді кольорових карток із позначенням кількості годин, типу блоку та службових маркерів. Додатково дедлайни позначаються окремими бейджами «DUE», що дозволяє відразу побачити критичні точки у навчальному графіку.

Рисунок 3.9 – Місячний огляд календарного плану з дедлайнами та навчальними блоками

Місячне подання дає користувачеві зовсім інший рівень сприйняття плану. Якщо блок Next 7 days відповідає на запитання «що робити найближчим часом», то календар за місяць дозволяє оцінити, наскільки рівномірно розподілене навантаження в цілому. Користувач може відразу побачити, у які тижні зосереджено найбільше дедлайнів, де кілька задач накладаються одна на одну, а де, навпаки, залишаються відносно вільні дні. Такий огляд є важливим не тільки для контролю за майбутніми строками, а й для усвідомленого коригування плану – наприклад, щоб перенести частину навантаження з перевантаженого тижня на менш насичений.

Важливо, що місячний календар відображає не лише наявність завдань, а й логіку їх розміщення у часі. Кольорові блоки показують, які саме навчальні інтервали система вже розподілила по днях, а бейджі дедлайнів візуально відокремлюють крайні терміни від звичайної запланованої роботи. Якщо в одному дні поєднуються навчальний блок і дедлайн, це може свідчити про відсутність резервного часу, тобто про ризиковий сценарій, за якого все виконання фактично відкладається до останнього моменту. Навпаки, якщо дедлайн позначено в одному дні, а основні навчальні блоки розташовані раніше, це свідчить про більш збалансований і безпечний підхід до розподілу навантаження.

Поєднання короткострокового планера і місячного календаря дозволяє реалізувати дворівневе планування. На першому рівні користувач працює з найближчими днями і виконує конкретні дії, які система рекомендує просто зараз. На другому рівні він бачить ширшу картину та може оцінити, як ці рішення впливають на весь місяць. Така логіка є практично зручною: студенту не потрібно щоразу аналізувати увесь розклад повністю, але в разі потреби він може вийти на стратегічний рівень і переглянути загальне співвідношення завдань, дедлайнів і вільного часу.

Третім компонентом цієї підсистеми є модуль рекомендацій, який виконує аналітичну функцію. Якщо блок найближчих семи днів показує короткостроковий план, а місячний календар – загальну структуру навантаження, то рекомендації

пояснюють, наскільки вдалим є сформований розклад і що саме бажано змінити. Для цього система збирає дані про всі активні завдання, побудований календарний план, користувацькі налаштування та ручні зміни, після чого передає їх до окремого модуля взаємодії з Gemini API.

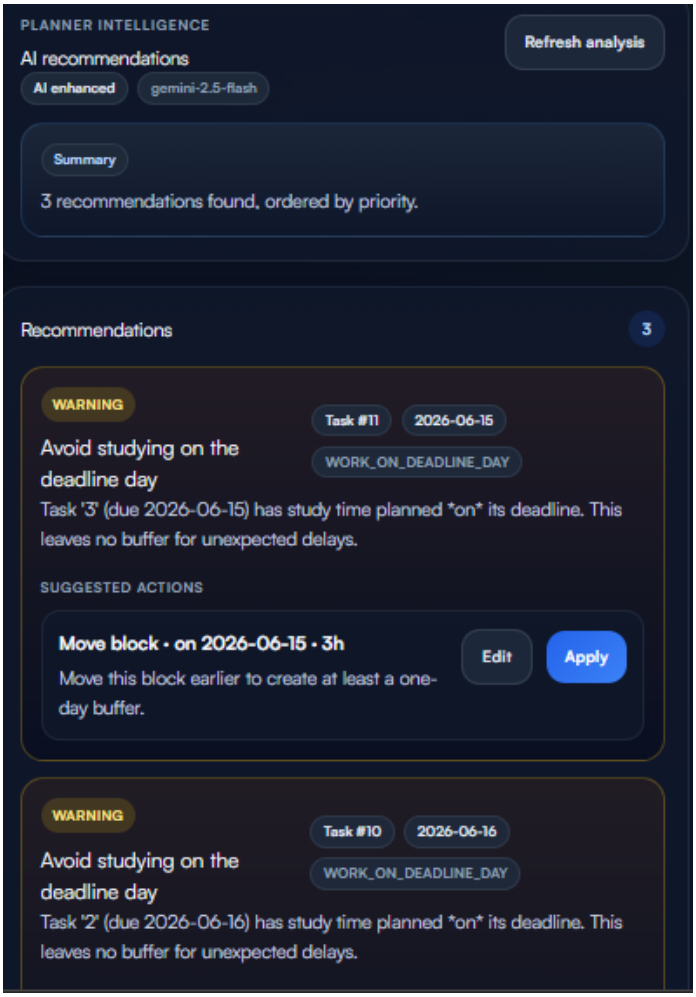

Рисунок 3.10 – Блок рекомендацій щодо коригування навчального плану

У відповідь формується текстовий аналіз, який містить короткий підсумок стану плану, перелік виявлених ризиків і набір пропозицій щодо його корекції. Результат роботи цього модуля відображається у вигляді окремого блока рекомендацій на сторінці планера. У верхній частині блоку зазвичай показується коротке зведення: скільки рекомендацій знайдено, у якому порядку вони впорядковані та який загальний

стан поточного плану. Нижче розташовуються окремі картки рекомендацій. Кожна з них містить тип повідомлення, наприклад попередження про ризик, посилання на відповідне завдання, дату дедлайну та текстове пояснення проблемної ситуації. Додатково в межах картки подається перелік запропонованих дій – наприклад, перенести блок на іншу дату, змінити структуру розподілу годин або уникати навчання безпосередньо в день дедлайну.

Особливу увагу модуль рекомендацій приділяє виявленню перевантажених або потенційно ризикових днів. Аналіз не обмежується лише формальним підрахунком сумарної кількості годин. Система також перевіряє, чи не зосереджено надто багато важливої роботи в останній день перед дедлайном, чи не повторюються поспіль дні з високою щільністю навантаження, чи не формується ситуація, коли завдання високого пріоритету лишається без достатнього часу на підготовку. Саме тому рекомендації можуть з’являтися навіть у випадках, коли денний ліміт формально не перевищено, але сама структура розкладу свідчить про його ненадійність або відсутність резерву часу.

Важливо підкреслити, що рекомендації не змінюють план автоматично без участі користувача. Система лише пропонує варіанти оптимізації, але остаточне рішення завжди залишається за студентом. Користувач може погодитися з порадою повністю, наприклад перенести навчальний блок з дня дедлайну на попередню дату, може частково відкоригувати план на власний розсуд або взагалі відхилити запропоновану дію. Такий механізм зберігає баланс між автоматизованим аналізом і людським контролем: система допомагає побачити слабкі місця графіка, але не забирає в користувача право вирішувати, який режим навчання є для нього прийнятним.

Отже, реалізація календарного планування та модуля рекомендацій розширює функціональні можливості інформаційної системи порівняно зі звичайними засобами обліку завдань. Поєднання календарного представлення, автоматичного розрахунку

навантаження та текстових рекомендацій дозволяє перейти від простого збереження навчальних даних до їх інтерпретації та практичного використання для оптимізації індивідуального навчального плану.

3.4 Тестування функціонування інформаційної системи

Перевірка правильності роботи розробленої інформаційної системи передбачає тестування як серверної, так і клієнтської частин із фокусом на основних користувацьких сценаріях. У межах цієї роботи тестування розглядалося не як формальна перевірка окремих функцій, а як спосіб переконатися, що вся система в цілому підтримує повний робочий цикл: від створення завдання й збереження його у базі даних до формування календарного плану та отримання рекомендацій щодо корекції навантаження. Метою тестування було підтвердити, що система коректно обробляє вхідні дані, адекватно реагує на типові дії користувача, передбачувано поводиться при помилковому введенні та не «ламає» логіку планування за нетипових, але реалістичних сценаріїв.

У межах клієнтської частини основну увагу зосереджено на перевірці роботи форм, списків і механізмів оновлення стану. Оскільки інтерфейс побудований навколо кількох ключових форм – створення дисципліни, створення завдання, редагування параметрів планування – для кожної з них було визначено набір позитивних і негативних сценаріїв. Для форми створення завдання перевірялася, зокрема, реакція системи на відсутність обов’язкових полів (назви, дедлайну, орієнтовної тривалості), введення некоректної дати та значень, що виходять за допустимі межі (наприклад, занадто велика кількість годин на день). Тестувалося, чи блокується відправка форми в таких випадках, чи відображаються зрозумілі повідомлення про помилки поруч із відповідними полями, і чи знімаються ці повідомлення після виправлення введених значень.

Окремий блок тестів був присвячений спискам завдань і дисциплін. Для списку задач перевірялася коректність відображення щойно створених записів, оновлення інтерфейсу після змін статусу (наприклад, перехід задачі зі стану «в очікуванні» до «виконано») та видалення завдання з подальшим оновленням глобального стану. Аналогічним чином для переліку дисциплін контролювалося, чи правильно перераховується кількість пов’язаних завдань після їх додавання або видалення, та чи оновлюється інтерфейс без необхідності ручного перезавантаження сторінки. Уважно перевірявся зв’язок між вмістом Redux-сховища та тим, що бачить користувач: будь-яка зміна в даних (додавання, редагування, зміна статусу, видалення) повинна миттєво відображатися у відповідних компонентах інтерфейсу.

Важливим аспектом клієнтського тестування стала перевірка реакції інтерфейсу на відповіді серверної частини. Після запуску аналізу навантаження перевірялося, чи оновлюються відповідні віджети зведених показників, чи з’являється новий календарний план, та чи відображаються рекомендації лише після успішного завершення запиту. Додатково моделювалися ситуації з помилкою на сервері (наприклад, тимчасова недоступність сервісу рекомендацій), щоб перевірити, чи отримує користувач зрозуміле повідомлення про неможливість отримати рекомендації в даний момент, а інтерфейс не «зависає» у проміжному стані.

На рівні серверної частини тестування було зосереджене на перевірці маршруту роботи з дисциплінами, завданнями, календарним планом і модулем рекомендацій. Для кожного REST-ендпоінта створювалися тестові запити з валідними даними, щоб переконатися, що система повертає очікувані відповіді у форматі JSON і коректно змінює стан бази даних. Наприклад, для ендпоінта створення завдання перевірялося, що у відповідь повертається об’єкт із реальним ідентифікатором, прив’язкою до дисципліни та всіма полями, які потім використовуються клієнтською частиною. Для ендпоінтів оновлення й видалення задач тестувалося, що у базі даних змінюються саме ті записи, до яких звертався клієнт, а у відповідь повертається статус успішного

виконання запиту.

Окрема серія тестів була присвячена обробці помилкових або неповних запитів. Для ендпоінта створення завдання імітувалися запити без обов’язкових полів, із некоректними датами або з недопустимими значеннями пріоритету та складності. Метою було переконатися, що сервер не «мовчки» ігнорує такі запити, а повертає зрозумілі повідомлення про помилки з зазначенням проблемних полів. Аналогічний підхід застосовувався до модулів, що працюють з дисциплінами та налаштуваннями користувача: у відповідь на некоректні запити система має не змінювати дані, а повідомляти клієнтові про причину відмови.

Окрему увагу приділено перевірці алгоритму аналізу навчального навантаження. Для цього створювалися кілька тестових наборів завдань із різною кількістю годин, набором дедлайнів і пріоритетів. Для кожного набору очікувався логічний результат: наприклад, що задачі з близькими дедлайнами і високим пріоритетом розподіляються в першу чергу, що система не перевищує встановлений денний ліміт у звичайних випадках і позначає день як перевантажений лише тоді, коли уникнути перевищення об’єктивно неможливо. Перевірялося, чи формується календарний план без «дір» у часі (коли частина годин завдання чомусь залишається нерозподіленою), та чи коректно розраховуються узагальнені показники на кшталт кількості перевантажених днів, сумарного обсягу запланованих годин тощо. Приклади типових тестових сценаріїв наведено в таблиці 3.1.

Таблиця 3.1 – Приклади тестових сценаріїв

Сценарій	**Очікуваний результат**
Створення задачі з коректними даними	задача з'являється у списку, стан Redux оновлюється
Створення задачі без дедлайну	форма показує повідомлення про помилку

Кінець таблиці 3.1

Запуск аналізу навантаження	система повертає план за днями та показники перевантаження
Отримання рекомендації	відображається текстова порада на основі результатів аналізу
Зміна статусу задачі на виконано	задача не враховується у наступному перерахунку

Окрім наведених базових сценаріїв, додатково перевірялися більш «крайні» випадки: поведінка системи за відсутності жодної задачі (календар та дашборд мають коректно показувати порожні стани, а модуль рекомендацій – чесно повідомляти про відсутність даних для аналізу), ситуації з дуже великою кількістю задач у межах одного тижня, а також сценарії, коли користувач радикально змінює налаштування ліміту годин і знову запускає аналіз. Це дозволило переконатися, що алгоритм не «ламається» при нетипових даних, а інтерфейс завжди має зрозумілу поведінку й повідомлення.

Узагальнюючи результати проведеного тестування, можна зробити висновок, що основні користувацькі сценарії реалізовано відповідно до поставлених вимог. Система коректно обробляє як успішні випадки (створення дисциплін і завдань, запуск аналізу, побудова календаря, отримання рекомендацій), так і ситуації з помилковими даними, коли необхідно заблокувати неправильне введення й повернути користувачеві зрозуміле пояснення. Інтерфейс належним чином реагує на відповіді серверної частини, а зміни у даних одразу відображаються у відповідних компонентах без потреби додаткового перезавантаження сторінки. Для навчального проєкту такого рівня деталізації тестів є достатньо, щоб підтвердити працездатність ключових модулів інформаційної системи та своєчасно виявити можливі недоліки логіки роботи.

Перспективним напрямом розвитку є як розширення функціональності, так і поглиблення підходів до тестування. Надалі систему можна доповнити інтеграцією з зовнішніми календарями (наприклад, синхронізацією дедлайнів із сервісами Google Calendar), імпортом завдань із навчальних платформ та розширеною аналітикою, що показуватиме, як фактичне виконання задач співвідноситься із запланованим навантаженням. З боку тестування логічним кроком є застосування автоматизованих тестів інтерфейсу (end-to-end-тестування основних сценаріїв) та модульних тестів для окремих частин серверної логіки. Водночас у межах даної кваліфікаційної роботи основний акцент зроблено на ручному відпрацюванні базового циклу: введення задач, збереження даних, розрахунок навантаження, візуалізація результатів і отримання рекомендацій. Це дозволило продемонструвати повний цикл функціонування інформаційної системи та переконатися, що всі ключові ланки цього циклу працюють узгоджено.

Висновки до розділу 3

У третьому розділі було реалізовано і проаналізовано повний цикл роботи інформаційної системи – від побудови інтерфейсу до перевірки коректності функціонування основних модулів. Детальний опис клієнтської частини показав, що інтерфейс не обмежується простим відображенням списків, а підтримує цілеспрямовану роботу користувача з навчальними даними. Форми створення дисциплін і завдань, сторінка дашборду, календарне подання, планер на найближчі дні та блок рекомендацій побудовані так, щоб не лише приймати дані, а й допомагати студенту швидко зрозуміти поточний стан свого навчального навантаження й ухвалювати рішення щодо його корекції. Узгодженість між компонентами забезпечується через централізоване сховище стану, завдяки чому будь-яка зміна в одній частині інтерфейсу миттєво відображається в усіх пов’язаних розділах.

Другим важливим результатом цього розділу є реалізація календарного

планування разом із модулем рекомендацій. Було продемонстровано, що система не обмежується збереженням завдань і виведенням їх списком, а на основі заданих параметрів (дедлайни, тривалість, пріоритет, користувацькі ліміти) формує календарний план у двох взаємодоповнювальних поданнях: короткостроковому (картки на найближчі дні) та довгостроковому (місячний огляд). Поверх цього плану працює модуль рекомендацій, що аналізує структуру навантаження, виявляє перевантажені або ризикові дні й формує текстові поради щодо зміни плану. Таким чином, система переходить від пасивного відображення інформації до активної підтримки прийняття рішень, залишаючи остаточний контроль за користувачем.

Окремий акцент у розділі зроблено на тестуванні функціонування системи. Перевірено не лише «щасливі» сценарії (успішне створення завдань, запуск аналізу, отримання рекомендацій), а й випадки з помилковим введенням, некоректними запитами до серверної частини та нетиповими наборами даних. Це дозволило підтвердити, що система коректно обробляє помилки, не втрачає цілісності даних та адекватно реагує інтерфейсом на результати серверних обчислень. Наведені тестові сценарії охоплюють ключові точки взаємодії – форми, списки, аналіз навантаження, модуль рекомендацій – і демонструють, що реалізовані алгоритми працюють передбачувано і відповідають закладеним функціональним вимогам.

У сукупності результати, отримані в третьому розділі, показують, що розроблена інформаційна система не є набором ізольованих сторінок чи скриптів, а становить цілісний інструмент управління навчальним навантаженням. Вона забезпечує повний робочий цикл: введення і редагування даних, автоматичний розрахунок плану, візуалізацію результатів у зручних для сприйняття формах, аналітичні рекомендації та перевірку коректності роботи основних функцій. Закладена архітектура та вже реалізовані сценарії дозволяють у подальшому розширювати систему – інтегрувати зовнішні календарі, додавати нові види аналітики й автоматизовані тести – без кардинальних змін у її базовій логіці.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було успішно вирішено актуальну науково-практичну задачу – розроблено інформаційну систему планування навчального навантаження здобувачів з використанням інтелектуальних технологій. Обрана тематика зумовлена потребою раціонального розподілу навчального часу в умовах зростання обсягу завдань, дедлайнів та використання дистанційних форм навчання, тоді як більшість наявних сервісів планування надають лише базові можливості фіксації подій у календарі та не забезпечують глибокого аналізу навантаження й персоналізованих рекомендацій для здобувачів.

На етапі системного аналізу було досліджено предметну область планування навчального навантаження, розглянуто можливості сучасних сервісів управління завданнями та навчальних платформ і виявлено їхні обмеження щодо комплексної підтримки студента в організації навчальної діяльності. Встановлено, що переважна частина таких рішень зосереджується на реєстрації подій і нагадуваннях, не пропонуючи інструментів оцінювання сумарного навантаження, виявлення перевантажених періодів та формування рекомендацій щодо перерозподілу навчального часу. Це обґрунтувало доцільність розроблення спеціалізованої інформаційної системи, яка поєднує механізми календарного планування, аналітики та інтелектуальної підтримки прийняття рішень.

У теоретичній частині роботи було визначено об’єкт, предмет і мету дослідження, сформульовано функціональні та нефункціональні вимоги до системи. Об’єктом дослідження виступає процес планування й аналізу навчального навантаження здобувача вищої освіти в межах веборієнтованої інформаційної системи, а предметом – програмна реалізація алгоритму планування навантаження та основних модулів системи, зокрема бази даних PostgreSQL, REST API на FastAPI, модулів обліку дисциплін і завдань, аналізу та візуалізації навантаження і формування

рекомендацій за допомогою Gemini API. Метою роботи визначено створення інформаційної системи, яка дозволяє здобувачам планувати та контролювати навчальне навантаження, автоматизувати облік навчальних дисциплін і завдань, аналізувати динаміку навантаження та отримувати персоналізовані рекомендації щодо розподілу навчального часу.

На етапі проєктування було спроєктовано клієнт-серверну архітектуру вебзастосунку, розроблено UML-діаграми та структуру бази даних. У реляційній базі даних PostgreSQL виокремлено сутності для зберігання інформації про навчальні дисципліни, завдання, дедлайни, пріоритети, складність, орієнтовну тривалість виконання, налаштування користувача та результати аналізу навантаження. Нормалізація таблиць забезпечила цілісність і несуперечливість даних, а використання чітко визначених зв’язків між сутностями – можливість ефективного виконання запитів, необхідних для розрахунку навчального графіка та формування рекомендацій.

Серверну частину системи реалізовано з використанням FastAPI, що дало змогу організувати REST-інтерфейс для роботи з даними та алгоритмами аналізу. Реалізовано маршрути для обліку дисциплін, управління завданнями, виконання аналізу навчального навантаження, формування календарного плану, роботи з користувацькими налаштуваннями та взаємодії з модулем рекомендацій. Вбудовані механізми валідації вхідних даних забезпечують перевірку коректності полів, а автоматично згенерована документація Swagger спрощує налагодження й тестування API. У серверну логіку інтегровано модуль, який на основі списку активних завдань, параметрів користувача та дозволених обмежень формує план розподілу навантаження за днями, визначає перевантажені періоди й узагальнені показники.

Клієнтську частину інформаційної системи реалізовано на основі бібліотеки React з використанням Redux і Tailwind CSS. Побудовано набір компонентів, що забезпечують повний цикл роботи користувача із системою: форми створення

навчальних дисциплін і завдань, список задач, головну аналітичну панель (дашборд), сторінку календарного планування та модуль налаштувань. Інтерфейс організовано таким чином, щоб відображати природну послідовність дій здобувача: спочатку створення переліку дисциплін, далі введення параметрів завдань, потім перегляд зведених показників і детального плану на дашборді та в календарі. Реалізовано можливість персоналізації параметрів планування – цільових і максимальних годин навчання на день, режиму використання вихідних, вибору часового поясу та мови інтерфейсу, що дозволяє адаптувати систему до індивідуального режиму навчання користувача.

Особливу увагу в роботі приділено реалізації календарного подання й модуля інтелектуальних рекомендацій. Запропонований алгоритм розподілу навантаження за днями враховує дедлайни, пріоритети, складність завдань та обмеження щодо щоденного навантаження, формуючи календарний план, у якому виділяються перевантажені та відносно вільні дні. На основі сформованого плану та поточних налаштувань користувача система готує структурований запит до Gemini API, у відповідь на який отримує текстові рекомендації щодо оптимізації навчального графіка. Інтерфейс блоку рекомендацій надає користувачеві стислий підсумок, перелік порад і попереджень, що дозволяє не лише бачити розподіл завдань у часі, а й отримувати пояснення та пропозиції щодо його вдосконалення.

У роботі також реалізовано засоби візуалізації результатів. У межах аналітичної панелі відображаються ключові показники навчального навантаження, списки останніх задач, перелік дисциплін і календарний розклад. Така організація інтерфейсу забезпечує поєднання зведеної та деталізованої інформації, дозволяє швидко оцінити загальний стан навчального навантаження, виявити потенційно проблемні періоди та перейти до аналізу окремих завдань.

Якість функціонування системи перевірено за допомогою тестових сценаріїв, які охоплюють як клієнтську, так і серверну частини. Перевірено створення завдань із

коректними та некоректними параметрами, обробку помилок валідації, роботу форм і списків, запуск аналізу навантаження, формування календарного плану, отримання рекомендацій і зміну статусу задач. Результати тестування показали, що система коректно реагує на основні користувацькі дії, правильно опрацьовує помилки введення та узгоджено оновлює інтерфейс залежно від відповідей серверної частини. Для навчального проєкту це свідчить про достатній рівень стабільності та завершеності реалізованого програмного забезпечення.

Підсумовуючи виконану роботу, можна стверджувати, що поставлена мета – розробка інформаційної системи планування навчального навантаження здобувачів з використанням інтелектуальних технологій – досягнута повною мірою. Створене програмне рішення поєднує сучасні технології веброзробки, модульну серверну архітектуру, зручний користувацький інтерфейс, алгоритми календарного планування та інструменти формування текстових рекомендацій, забезпечуючи повний цикл опрацювання навчальних даних – від їх введення до отримання аналітичних висновків. Розроблена система може бути використана як допоміжний інструмент для здобувачів, які прагнуть раціонально організувати власний навчальний час, уникати перевантаження та спиратися на обґрунтовані рекомендації під час планування навчальної діяльності.

Подальший розвиток проєкту може полягати в інтеграції системи з зовнішніми навчальними платформами та календарями для автоматичного імпорту дедлайнів, розширенні аналітичних можливостей за рахунок урахування результатів успішності та рівня засвоєння матеріалу, удосконаленні алгоритмів формування рекомендацій, а також реалізації мобільної версії застосунку. Це дозволить ще більше підвищити практичну цінність інформаційної системи та розширити сферу її застосування в освітньому процесі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Голуб Н. О., Осадча К. П. Цифровізація освіти: сучасні тенденції та виклики // Інформаційні технології і засоби навчання. 2021. Т. 84, № 4. [Електронний ресурс]. URL: https://journal.iitta.gov.ua (date of request: 24.05.2026).

2. Ярема О. О. Порівняльний аналіз онлайн-сервісів для організації навчальної діяльності студентів // Інформаційні технології і засоби навчання. 2022. № 2. С. 50–63. [Електронний ресурс]. URL: https://journal.iitta.gov.ua (date of request: 24.05.2026).

3. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation. University of California, Irvine. 2000. [Електронний ресурс]. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (date of request: 24.05.2026).

4. Richardson L., Ruby S. RESTful Web Services. O'Reilly Media. 2007.

5. PostgreSQL 16 Documentation. PostgreSQL Global Development Group. [Електронний ресурс]. URL: https://www.postgresql.org/docs (date of request: 25.05.2026).

6. Жалдак М. І., Рамендик Д. В. Інформаційні системи підтримки прийняття рішень в освітньому процесі // Інформаційні технології і засоби навчання. 2019. Т. 72, № 4. С. 22–35. [Електронний ресурс]. URL: https://journal.iitta.gov.ua (date of request: 26.05.2026).

7. Іванченко О. В. Інтеграція штучного інтелекту в навчальні платформи вищої школи // Педагогічні науки: теорія і практика. 2025. № 3. С. 45–56.

8. Coronel C., Morris S. Database Systems: Design, Implementation, and Management. 13th ed. Cengage Learning. 2018.

9. Masuda Y. Web API Design Patterns. Manning Publications. 2022.

10. Pinedo M. Scheduling: Theory, Algorithms, and Systems. 5th ed. Springer. 2016.

11. Google. Gemini API Overview. [Електронний ресурс]. URL: https://ai.google.dev (date of request: 26.05.2026).

12. React Documentation. [Електронний ресурс]. URL: https://react.dev (date of request: 28.05.2026).

13. Redux Documentation. [Електронний ресурс]. URL: https://redux.js.org (date of request: 24.05.2026).

14. Tailwind CSS Documentation. [Електронний ресурс]. URL: https://tailwindcss.com (date of request: 28.05.2026).

15. Jest Documentation. [Електронний ресурс]. URL: https://jestjs.io (date of request: 29.05.2026).

16. Кондратенко Є. В. Інтеграція штучного інтелекту в систему професійної підготовки здобувачів вищої освіти педагогічних спеціальностей // Педагогічна інноватика: сучасність та перспективи. 2025. № 10. С. 1–10. [Електронний ресурс]. URL: https://journals.uzhnu.uz.ua/index.php/ped/article/view/1445 (date of request: 30.05.2026).

17. Misra R., McKean M. College Students' Academic Stress and Its Relation to Their Anxiety, Time Management, and Leisure Satisfaction // American Journal of Health Studies. 2000. Vol. 16(1). P. 41–51.

18. Kember D., Leung D. Y. Academic Workload, Stress, and Study Efficiency in University Students // Educational Psychology. 2006. Vol. 26(3). P. 329–343.

19. Кузьменко Г. Н. Математичне моделювання навчального навантаження студентів // Вісник університету. 2018. № X. С. 45–52.

20. Журавльов В. С. Методи оцінювання навчального навантаження студентів у вищій школі // Педагогіка вищої школи. 2020. № 2. С. 30–38.

21. Cepeda N. J. et al. Distributed Practice in Verbal Recall Tasks: A Review and Quantitative Synthesis // Psychological Bulletin. 2006. Vol. 132(3). P. 354–380.

22. Maes P. Agents That Reduce Work and Information Overload // Communications of the ACM. 1994. Vol. 37(7). P. 31–40.

23. Paas F., van Merriënboer J. Cognitive Load Theory: Methods to Manage Working Memory Load in the Learning Process // Cognitive Load Theory. Springer. 2011. P. 99–110.

24. Жалдак М. І., Морзе Н. В. Інформаційні технології в освіті: системи підтримки прийняття рішень.

25. Doucek P., Maryska M. Dynamic Scheduling in Educational Processes // Proceedings of the International Conference on E-Learning. 2015. P. 101–108.

26. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms. 4th ed. MIT Press. 2022.

27. Norman D. The Design of Everyday Things. Revised and Expanded Edition. Basic Books. 2013.

28. Sharp H., Rogers Y., Preece J. Interaction Design: Beyond Human-Computer Interaction. 5th ed. Wiley. 2019.

29. Sweller J. Cognitive Load During Problem Solving: Effects on Learning // Cognitive Science. 1988. Vol. 12(2). P. 257–289.

30. Clough G. et al. Informal Learning with PDAs and Smartphones // Journal of Computer Assisted Learning. 2008. Vol. 24(5). P. 359–371.

31. Trello Product Guide. [Електронний ресурс]. URL: https://trello.com/guide (date of request: 24.05.2026).

32. Google Calendar Help Center. [Електронний ресурс]. URL: https://support.google.com/calendar (date of request: 30.05.2026).

33. Google Calendar API Overview. [Електронний ресурс]. URL: https://developers.google.com/calendar (date of request: 31.05.2026).

34. Moodle Docs: Course Management. [Електронний ресурс]. URL: https://docs.moodle.org (date of request: 31.05.2026).

35. Dougiamas M., Taylor P. Moodle: Using Learning Communities to Create an Open Source Course Management System // EdMedia + Innovate Learning. 2003. P. 171–178.

36. Dalsgaard C. Social Software: E-learning Beyond Learning Management Systems // European Journal of Open, Distance and E-Learning. 2006. Vol. 9(2).

37. Notion Help & Support. [Електронний ресурс]. URL: https://www.notion.so/help (date of request: 31.05.2026).

38. Sommerville I. Software Engineering. 10th ed. Pearson. 2015.

39. Google. Gemini API Terms of Service. [Електронний ресурс]. URL: https://ai.google.dev/gemini-api/terms (date of request: 31.05.2026).

ДОДАТОК А

Лістинг фрагментів коду

database.py

```
import os
from dotenv import load_dotenv
from sqlalchemy import create_engine
from sqlalchemy.orm import declarative_base, sessionmaker
load_dotenv()
DATABASE_URL = os.getenv("DATABASE_URL")
engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

planner.py

```
from datetime import date, timedelta
from collections import defaultdict

PRIORITY_MAP = {
    "low": 1,
    "medium": 2,
    "high": 3,
}
DIFFICULTY_MAP = {
    "easy": 1,
    "medium": 2,
    "hard": 3,
}

def parse_deadline(deadline):
    if not deadline:
        return None
    if isinstance(deadline, str):
        try:
            return date.fromisoformat(deadline)
        except ValueError:
            return None
    return deadline


def is_weekend(day: date) -> bool:
```

```
    return day.weekday() >= 5


def sort_tasks(tasks):
    active_tasks = [
        task
        for task in tasks
        if (getattr(task, "status", None) or "pending") != "completed"
    ]

    def sort_key(task):
        deadline = parse_deadline(getattr(task, "deadline", None))
        if deadline is None:
            deadline = date.max

        return (
            deadline,
            -PRIORITY_MAP.get((getattr(task, "priority", None) or "medium"), 2),
            -DIFFICULTY_MAP.get((getattr(task, "difficulty", None) or "medium"), 2),
            -(getattr(task, "estimated_hours", None) or 0),
        )

    active_tasks.sort(key=sort_key)
    return active_tasks


def normalize_existing_load(existing_events):
    loads = {}

    for item in existing_events or []:
        work_date = getattr(item, "plan_date", None) or getattr(item, "work_date",
None)
        planned_hours = getattr(item, "planned_hours", 0) or 0

        if not work_date:
            continue

        loads[work_date] = loads.get(work_date, 0) + planned_hours

    return loads


def get_existing_task_hours(existing_events):
    task_hours = defaultdict(float)

    for item in existing_events or []:
        task_id = getattr(item, "task_id", None)
        planned_hours = getattr(item, "planned_hours", 0) or 0

        if task_id is None:
            continue

        task_hours[task_id] += planned_hours

    return task_hours
```

```
def get_day_load_map(events):
    loads = {}

    for item in events or []:
        work_date = getattr(item, "plan_date", None) or getattr(item, "work_date",
None)
        planned_hours = getattr(item, "planned_hours", 0) or 0

        if not work_date:
            continue

        loads[work_date] = loads.get(work_date, 0) + planned_hours

    return loads


def build_schedule(
    tasks,
    study_hours_per_day=3,
    max_hours_per_day=6,
    weekend_study=False,
    existing_events=None,
):
    tasks = sort_tasks(tasks)
    schedule = []
    unplanned = []
    today = date.today()

    target_loads = normalize_existing_load(existing_events)
    hard_loads = normalize_existing_load(existing_events)
    existing_task_hours = get_existing_task_hours(existing_events)

    for task in tasks:
        estimated_hours = getattr(task, "estimated_hours", None) or 0
        already_planned_for_task = existing_task_hours.get(task.id, 0)
        hours_left = max(estimated_hours - already_planned_for_task, 0)

        if hours_left <= 0:
            continue

        current_day = today
        deadline = parse_deadline(getattr(task, "deadline", None))

        if deadline is None:
            deadline = today + timedelta(days=14)

        while hours_left > 0 and current_day <= deadline:
            if not weekend_study and is_weekend(current_day):
                current_day += timedelta(days=1)
                continue

            current_target_load = target_loads.get(current_day, 0)
            current_hard_load = hard_loads.get(current_day, 0)

            free_target_hours = max(study_hours_per_day - current_target_load, 0)
            free_hard_hours = max(max_hours_per_day - current_hard_load, 0)
```

```
            if free_hard_hours <= 0:
                current_day += timedelta(days=1)
                continue

            if free_target_hours <= 0:
                current_day += timedelta(days=1)
                continue

            assigned = min(hours_left, free_target_hours, free_hard_hours)

            if assigned <= 0:
                current_day += timedelta(days=1)
                continue

            schedule.append(
                {
                    "task_id": task.id,
                    "title": task.title,
                    "subject_id": getattr(task, "subject_id", None),
                    "date": current_day,
                    "hours": assigned,
                    "overload_flag": False,
                    "deadline": deadline.isoformat() if deadline else None,
                }
            )

            target_loads[current_day] = current_target_load + assigned
            hard_loads[current_day] = current_hard_load + assigned
            existing_task_hours[task.id] = existing_task_hours.get(task.id, 0) +
assigned
            hours_left -= assigned

            current_day += timedelta(days=1)

        if hours_left > 0:
            unplanned.append(
                {
                    "task_id": task.id,
                    "title": task.title,
                    "hours_left": hours_left,
                    "deadline": deadline.isoformat() if deadline else None,
                }
            )
    return {
        "schedule": schedule,
        "unplanned": unplanned,
    }

def validate_event_move(
    event_id,
    new_date,
    new_hours,
    all_events,
    settings,
):
    if not settings.weekend_study and new_date.weekday() >= 5:
        return {
```

```
            "ok": False,
            "message": "Weekend study is disabled in settings.",
        }

    total_hours = 0

    for event in all_events:
        if event.id == event_id:
            continue
        event_date = getattr(event, "plan_date", None) or getattr(event, "work_date",
None)
        event_hours = getattr(event, "planned_hours", 0) or 0

        if event_date == new_date:
            total_hours += event_hours

    total_hours += new_hours

    if total_hours > settings.max_hours_per_day:
        free_hours = max(settings.max_hours_per_day - (total_hours - new_hours), 0)
        return {
            "ok": False,
            "message": (
                f"Cannot place this block on {new_date}. "
                f"Only {free_hours}h available on that day, but this block needs
{new_hours}h."
            ),
        }
    return {
        "ok": True,
        "message": "Move is valid.",
    }

def mark_overload_flags(events, settings):
    day_loads = get_day_load_map(events)
    overload_map = {}

    for event in events:
        work_date = getattr(event, "plan_date", None) or getattr(event, "work_date",
None)
        total = day_loads.get(work_date, 0)
        overload_map[event.id] = total > settings.max_hours_per_day

    return overload_map
```

planner_rules.py

```
from __future__ import annotations

from collections import defaultdict
from datetime import datetime
from typing import Any

from app.schemas.ai import (
    PlannerAnalysisResponse,
    PlannerAnalysisMeta,
```

```
    PlannerRecommendation,
    RecommendationAction,
)

def _parse_date(value: str | None) -> datetime | None:
    if not value:
        return None
    try:
        return datetime.fromisoformat(str(value))
    except ValueError:
        try:
            return datetime.strptime(str(value), "%Y-%m-%d")
        except ValueError:
            return None


def _date_key(value: str | None) -> str | None:
    dt = _parse_date(value)
    if not dt:
        return None
    return dt.strftime("%Y-%m-%d")


def _days_between(start: str | None, end: str | None) -> int | None:
    a = _parse_date(start)
    b = _parse_date(end)
    if not a or not b:
        return None
    return (b.date() - a.date()).days


def _today_key() -> str:
    return datetime.now().strftime("%Y-%m-%d")


def build_rule_based_analysis(payload: dict[str, Any]) -> PlannerAnalysisResponse:
    tasks = payload.get("tasks", []) or []
    blocks = payload.get("blocks", []) or []
    settings = payload.get("settings", {}) or {}

    study_hours_per_day = float(settings.get("study_hours_per_day") or 4)

    recommendations: list[PlannerRecommendation] = []
    warnings: list[str] = []
    changes: list[RecommendationAction] = []

    task_map = {int(task["id"]): task for task in tasks if task.get("id") is not
None}

    blocks_by_day: dict[str, list[dict[str, Any]]] = defaultdict(list)
    blocks_by_task: dict[int, list[dict[str, Any]]] = defaultdict(list)

    for block in blocks:
        key = _date_key(block.get("work_date"))
        if key:
            blocks_by_day[key].append(block)
        task_id = block.get("task_id")
```

```
        if task_id is not None:
            blocks_by_task[int(task_id)].append(block)

    today = _today_key()
    for day, day_blocks in sorted(blocks_by_day.items()):
        total_hours = sum(float(b.get("planned_hours") or 0) for b in day_blocks)
        overloaded_blocks = [b for b in day_blocks if b.get("overload_flag")]
        if total_hours > study_hours_per_day or overloaded_blocks:
            worst_block = max(day_blocks, key=lambda b: float(b.get("planned_hours")
or 0))
            extra = round(total_hours - study_hours_per_day, 1)
            extra = extra if extra > 0 else 1.0

            recommendations.append(
                PlannerRecommendation(
                    code="OVERLOAD_DAY",
                    level="warning" if total_hours <= study_hours_per_day + 2 else
"critical",
                    title="Too much workload in one day",
                    message=f"{day} has {total_hours} planned hours, above the
preferred {study_hours_per_day}h limit.",
                    reason="Concentrating too much work in one day increases the
chance of burnout or unfinished blocks.",
                    task_id=worst_block.get("task_id"),
                    block_id=worst_block.get("id"),
                    date=day,
                    actions=[
                        RecommendationAction(
                            type="move_block",
                            block_id=worst_block.get("id"),
                            task_id=worst_block.get("task_id"),
                            date=day,
                            to_date=None,
                            hours=min(extra, float(worst_block.get("planned_hours")
or 1)),
                            reason="Move part of the heaviest block to another day to
reduce overload.",
                        )
                    ],
                )
            )
    for task in tasks:
        task_id = int(task["id"])
        deadline = _date_key(task.get("deadline"))
        task_blocks = blocks_by_task.get(task_id, [])

        if not deadline or not task_blocks:
            continue

        deadline_day_blocks = [b for b in task_blocks if
_date_key(b.get("work_date")) == deadline]
        if deadline_day_blocks:
            block = deadline_day_blocks[0]
            recommendations.append(
                PlannerRecommendation(
                    code="WORK_ON_DEADLINE_DAY",
                    level="warning",
```

```
                    title="Study is scheduled on the deadline day",
                    message=f"Task '{task.get('title')}' still has planned work on
its deadline date {deadline}.",
                    reason="Leaving study work for the deadline day leaves no buffer
for unexpected delays.",
                    task_id=task_id,
                    block_id=block.get("id"),
                    date=deadline,
                    actions=[
                        RecommendationAction(
                            type="move_block",
                            block_id=block.get("id"),
                            task_id=task_id,
                            date=deadline,
                            to_date=None,
                            hours=float(block.get("planned_hours") or 1),
                            reason="Move this block earlier to create at least a one-
day buffer.",
                        )
                    ],
                )
            )
    for task in tasks:
        task_id = int(task["id"])
        if str(task.get("status")).lower() == "completed":
            continue

        deadline = _date_key(task.get("deadline"))
        if not deadline:
            continue

        days_left = _days_between(today, deadline)
        if days_left is None:
            continue

        estimated_hours = float(task.get("estimated_hours") or 0)
        planned_hours = sum(float(b.get("planned_hours") or 0) for b in
blocks_by_task.get(task_id, []))
        remaining_hours = round(max(estimated_hours - planned_hours, 0), 1)

        if days_left <= 2 and remaining_hours > 0:
            recommendations.append(
                PlannerRecommendation(
                    code="DEADLINE_RISK",
                    level="critical",
                    title="High deadline risk",
                    message=f"Task '{task.get('title')}' is due in {days_left} day(s)
and still has about {remaining_hours}h unscheduled.",
                    reason="The remaining work volume is too high for the time left
before the deadline.",
                    task_id=task_id,
                    date=deadline,
                    actions=[
                        RecommendationAction(
                            type="create_block",
                            task_id=task_id,
                            to_date=today,
```

```
                                hours=min(remaining_hours, study_hours_per_day),
                                reason="Create an extra block immediately to reduce
deadline risk.",
                            )
                        ],
                    )
                )
    for task in tasks:
        task_id = int(task["id"])
        if str(task.get("status")).lower() == "completed":
            continue
        if blocks_by_task.get(task_id):
            continue

        recommendations.append(
            PlannerRecommendation(
                code="TASK_NOT_SCHEDULED",
                level="warning",
                title="Task has no study blocks",
                message=f"Task '{task.get('title')}' is active but has no scheduled
study blocks.",
                reason="An active task without any blocks is likely to be forgotten
or delayed.",
                task_id=task_id,
                date=_date_key(task.get("deadline")),
                actions=[
                    RecommendationAction(
                        type="create_block",
                        task_id=task_id,
                        to_date=today,
                        hours=min(float(task.get("estimated_hours") or 2),
study_hours_per_day),
                        reason="Create the first block so the task enters the study
plan.",
                    )
                ],
            )
        )

    # 5. Manual locks reduce flexibility
    locked_blocks = [b for b in blocks if b.get("manual_locked")]
    if locked_blocks:
        recommendations.append(
            PlannerRecommendation(
                code="TOO_MANY_LOCKED_BLOCKS",
                level="info",
                title="Too many manually locked blocks",
                message=f"{len(locked_blocks)} block(s) are manually locked, which
limits planner flexibility.",
                reason="Locked blocks reduce the system's ability to rebalance
overload and deadline pressure.",
                actions=[],
            )
        )

    # 6. Uneven subject distribution
    subject_hours: dict[str, float] = defaultdict(float)
```

```
    for block in blocks:
        subject = block.get("subject_title") or f"Subject #{block.get('subject_id')}"
if block.get("subject_id") else "Unknown"
        subject_hours[subject] += float(block.get("planned_hours") or 0)

    if len(subject_hours) >= 2:
        max_subject = max(subject_hours.items(), key=lambda x: x[1])
        min_subject = min(subject_hours.items(), key=lambda x: x[1])
        if max_subject[1] >= max(min_subject[1] * 2.5, 4):
            recommendations.append(
                PlannerRecommendation(
                    code="SUBJECT_IMBALANCE",
                    level="info",
                    title="Study time is uneven across subjects",
                    message=f"{max_subject[0]} has {max_subject[1]}h planned while
{min_subject[0]} has only {min_subject[1]}h.",
                    reason="A strong imbalance can leave weaker or ignored subjects
underprepared.",
                    actions=[],
                )
            )

    recommendations.sort(
        key=lambda r: {"critical": 0, "warning": 1, "info": 2}.get(r.level, 99)
    )

    if not recommendations:
        summary = "The current planner looks balanced. No major rule-based risks were
detected."
    else:
        summary = f"{len(recommendations)} recommendation(s) found. The most
important issues are shown first."

    return PlannerAnalysisResponse(
        summary=summary,
        recommendations=recommendations,
        warnings=warnings,
        changes=changes,
        meta=PlannerAnalysisMeta(source="rule-based", model=None),
    )
```