Міністерство освіти і науки України
Рівненський державний гуманітарний університет
Кафедра інформаційних технологій та моделювання

Кваліфікаційна робота

за освітнім ступенем «бакалавр»

на тему:

«Застосунок для персонального планування навчальних завдань»

Виконав:
здобувач IV курсу
групи КН-41
спеціальності 122 «Комп’ютерні науки»
Семенюк Юрій Романович

Науковий керівник:
старший викладач Тетяна Анатоліївна Кирик

М. Рівне, 2026 рік

ЗМІСТ

ВСТУП 4

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 7

1.1 Системи управління завданнями в освітньому процесі 7

1.1.1 Практична цінність персонального планувальника для студентів 8

1.1.2 Основні вимоги до сучасних task-трекерів 9

1.2 Порівняння з комерційними рішеннями 11

1.2.1 Аналіз існуючих продуктів (Notion, Trello, Google Calendar) 11

1.2.2 Переваги власної системи з інтеграцією ШІ 13

РОЗДІЛ 2 ТЕХНОЛОГІЧНИЙ СТЕК ДЛЯ РОЗРОБКИ ВЕБЗАСТОСУНК . 15

2.1 Огляд екосистеми React для розробки інтерфейсів 15

2.2 Використання Firebase як хмарної інфраструктури 19

2.2.1 Переваги Firestore для роботи з даними в реальному часі 19

2.2.2 Недоліки NoSQL баз даних та способи їх вирішення 20

2.3 Штучний інтелект як інструмент персоналізації навчання 22

2.3.1 Огляд можливостей API Google Generative AI (Gemini) 22

РОЗДІЛ 3 ПРОЄКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ ТА РОЗРОБКА КОРИСТУВАЦЬКОГО ІНТЕРФЕЙСУ 25

3.1 Архітектура вебзастосунку та структура бази даних 25

3.2 Розробка вебінтерфейсу та компонентної бази 26

3.2.1 Прототипування (Wireframing) та UX/UI рішення 29

3.2.2 Реалізація головної панелі та системи маршрутизації 31

3.2.3 Візуалізація місячного навантаження та історії завдань 32

РОЗДІЛ 4 ІНТЕГРАЦІЯ ШТУЧНОГО ІНТЕЛЕКТУ ТА РЕАЛІЗАЦІЯ БІЗНЕС-ЛОГІКИ ДОДАТКУ 34

4.1 Механізми взаємодії з API генеративного ШІ 34

4.1.1 Промпт-інжиніринг для отримання структурованих порад 37

4.1.2 Обробка та збереження згенерованих кроків у Firestore 39

4.2 Алгоритми пріоритезації та управління станами завдань 41

4.2.1 Архітектура управління життєвим циклом завдань 41

4.2.2 Багатокритеріальний алгоритм пріоритезації на головному екрані 42
4.2.3 Ізоляція станів та формування історії активності 44
4.4.2 Тестування взаємодії з API та хмарною інфраструктурою 46
ВИСНОВКИ 49
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 51
Додатки 53
Додаток А 53
Додаток Б 57

ВСТУП

Сучасний освітній процес вимагає від здобувачів вищої освіти високого рівня самоорганізації та ефективного планування часу. В умовах цифровізації та використання змішаних форматів навчання студенти щоденно стикаються з великою кількістю завдань, дедлайнів та різнопланових проєктів. Відсутність зручних та спеціалізованих інструментів для тайм-менеджменту часто призводить до нерівномірного розподілу навантаження та зниження академічної успішності [1, 2].

Актуальність дослідження зумовлена тим, що існуючі універсальні системи управління завданнями є надмірно перевантаженими функціоналом або, навпаки, не мають інструментів для візуалізації специфічного академічного навантаження. Студентам потрібен простий, швидкий та сфокусований інструмент для планування завдань, розбиття їх на тижні та відстеження власної успішності. Розробка власної системи дозволяє врахувати ці вимоги, а також інтегрувати додаткові допоміжні сервіси, зокрема інструменти автоматичної декомпозиції складних завдань, що робить процес навчання більш структурованим та зрозумілим [2].

Мета дослідження полягає у розробці персональної електронної системи планування навчальної діяльності для оптимізації управління завданнями та візуалізації академічного навантаження студентів.

Завдання дослідження:

1. Здійснити аналіз предметної області та існуючих рішень для тайм-менеджменту в освітньому процесі.
2. Обґрунтувати вибір сучасного технологічного стеку для розробки вебзастосунку.
3. Спроєктувати архітектуру системи та розробити інтуїтивно зрозумілий користувацький інтерфейс.
4. Реалізувати основний функціонал системи: додавання завдань, місячне планування та відстеження статистики.

5. Дослідити та інтегрувати додатковий сервіс для генерації покрокових підказок до завдань.
6. Провести тестування розробленого програмного продукту.

Об’єкт дослідження – процес управління та планування самостійної навчальної роботи здобувачів вищої освіти.

Предмет дослідження – методи, алгоритми та програмні засоби розробки вебзастосунку для управління навчальними завданнями.

Методи дослідження. У кваліфікаційній роботі застосовуються такі методи дослідження для досягнення поставленої мети та завдань:

1. Аналіз і порівняння: для вивчення предметної області студентського тайм-менеджменту та обґрунтування вибору технологій (екосистема React, Firebase, генеративний ШІ).
2. Моделювання: для розробки загальної архітектури вебзастосунку та побудови логічної структури нереляційної бази даних Firestore.
3. Метод швидкого прототипування: для створення чорнових макетів (wireframes) та розробки клієнтської частини застосунку з урахуванням сучасних принципів UI/UX дизайну.
4. Емпіричне тестування: для перевірки коректності роботи алгоритмів сортування завдань, синхронізації даних у реальному часі та оцінки якості згенерованих ШІ підказок.

Практичне значення дослідження полягає у створенні повноцінного вебзастосунку «StudyPlan», який дозволяє студентам зручно керувати своїм часом, візуально оцінювати рівень щотижневого навантаження та вести історію успішності. Система може бути використана студентами для підвищення власної продуктивності під час навчання.

Апробація та впровадження результатів дослідження

Основні теоретичні та практичні результати дослідження доповідалися на XIX Всеукраїнській науково-практичній конференції здобувачів вищої освіти і молодих учених «Наука, освіта, суспільство очима молодих» (м. Рівне, 14 травня 2026 р.), звітній науковій конференції викладачів, співробітників і здобувачів

вищої освіти Рівненського державного гуманітарного університету (м. Рівне, __ травня 2026 року).

Структура роботи. Основна частина кваліфікаційної роботи складається з чотирьох розділів.

Перший розділ розкриває проблематику тайм-менеджменту в сучасному освітньому процесі, містить аналіз предметної області та порівняння з існуючими комерційними рішеннями для планування завдань.

У другому розділі обґрунтовується вибір технологічного стеку для розробки вебзастосунку, зокрема використання бібліотеки React, хмарної інфраструктури Firebase для роботи з даними та можливостей API Google Generative AI (Gemini).

Третій розділ присвячено проєктуванню архітектури системи, створенню логічної структури бази даних Firestore та розробці інтерфейсу користувача (UI/UX). Детально розглядається програмна реалізація клієнтської частини: головної панелі керування (Dashboard), системи візуалізації місячного плану та історії завдань.

Четвертий розділ описує реалізацію бізнес-логіки застосунку, налаштування синхронізації даних у реальному часі та практичну інтеграцію сервісу генеративного штучного інтелекту для автоматизованої декомпозиції складних навчальних завдань на покрокові підказки.

РОЗДІЛ 1.
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Системи управління завданнями в освітньому процесі

Сучасний формат здобуття вищої освіти, особливо в умовах поєднання очного, змішаного та дистанційного навчання, вимагає від здобувачів освіти високого рівня автономності та самоорганізації [3]. Збільшення частки самостійної роботи, впровадження індивідуальних освітніх траєкторій та проєктно-орієнтованого підходу призводить до того, що студенти стикаються з серйозною проблемою управління власним часом. Без використання спеціалізованих інструментів контролювати множинні дедлайни, етапи виконання лабораторних робіт, підготовку до семінарів та паралельну участь у позанавчальних активностях стає вкрай важко. Це закономірно призводить до накопичення боргів і негативно впливає на загальну академічну успішність.

Для розв'язання проблеми організації часу традиційно використовуються системи управління завданнями (Task Management Systems). В сучасному освітньому середовищі найчастіше застосовуються два основні класи таких програмних продуктів, проте жоден з них не задовольняє потреби студента повною мірою.

Перший клас – це інституційні платформи (LMS – Learning Management Systems, тобто системи управління навчанням), такі як Moodle, Canvas, Blackboard або Google Classroom. Вони чудово справляються з адмініструванням навчального процесу з боку університету: забезпечують публікацію лекційних матеріалів, збір виконаних робіт, проведення тестувань та ведення електронних журналів оцінок. Проте зазначені системи є неадаптивними до умов персонального планування. Студент не може додати до LMS власні, не пов'язані з офіційним курсом завдання (наприклад, «прочитати додаткову статтю» або «підготуватися до олімпіади»), не може змінювати пріоритетність чи групувати завдання за власним бажанням.

Другий клас – це багатоцільові корпоративні трекери завдань та проєктів, такі як Trello, Asana, Jira чи Notion [5, 6]. Хоча вони пропонують широкий варіативний

функціонал, їхня архітектура є занадто перевантаженою для щоденного студентського використання [6]. Вони орієнтовані переважно на командну роботу, управління складними бізнес-процесами та делегування повноважень. Для студента необхідність постійно налаштовувати дошки, створювати складні інформаційні структури тасту заповнювати десятки необов'язкових полів стає додатковою перепоною.

Отже, на ринку програмного забезпечення існує потреба в легких, спеціалізованих інструментах, які були б сфокусовані виключно на специфіці академічного розкладу, дозволяли б оперативно фіксувати завдання у кілька кліків та не відволікали користувача надлишковим корпоративним функціоналом.

1.1.1 Практична цінність персонального планувальника для студентів

Впровадження спеціалізованого цифрового планувальника у щоденну навчальну рутину має ключове значення для зниження когнітивного навантаження здобувачів освіти. Згідно з концепціями когнітивної психології (зокрема, ефектом Зейгарнік), людська пам'ять має властивість постійно актуалізувати інформацію про перервані або незавершені дії. Це створює ефект незавершеного гештальту, що призводить до виснаження ресурсів уваги та формування стійкого фонового стресу в користувача [4].

Людська пам'ять не пристосована для одночасного утримання десятків різнопланових статусів, дат та технічних вимог до лабораторних робіт.

Використання застосунку створює так званий «зовнішній мозок» – надійне сховище, куди повністю делегується функція утримання організаційної інформації. Коли студент фіксує всі свої обов'язки в системі, рівень його тривожності знижується, що вивільняє ментальні ресурси безпосередньо для глибокої інтелектуальної праці та засвоєння нових знань.

Окрім розвантаження робочої пам'яті, персональний планувальник виконує надзвичайно важливу аналітичну та підтримувальну функцію. Наочна візуалізація завдань дозволяє об'єктивно оцінити майбутні обсяги роботи та запобігти ефекту прокрастинації. Найчастіше відкладання справ на потім виникає через страх перед

неструктурованими, глобальними проєктами (наприклад, написанням курсової роботи чи підготовкою до комплексної сесії). Розуміння того, з чого складається завдання, значно полегшує старт роботи.

Також регулярна фіксація виконаних завдань і ведення історії успішності формує потужне позитивне психологічне підкріплення. Переведення статусу завдання в категорію «Виконано» викликає відчуття задоволення від досягнутого результату. Можливість переглядати власну персональну статистику не лише стимулює користувача до подальшої продуктивності, але й дозволяє рефлексувати над власними помилками (наприклад, аналізувати причини пропущених дедлайнів) та значно точніше розраховувати власні сили при плануванні наступних навчальних періодів.

1.1.2 Основні вимоги до сучасних task-трекерів

Для того щоб розроблена система електронного планування була не просто формальним додатком, а ефективним інструментом, який дійсно буде використовуватися студентами на щоденній основі, вона повинна відповідати низці ключових технічних та функціональних вимог. Звичайний текстовий перелік справ (To-Do list) більше не задовольняє сучасні потреби здобувачів вищої освіти [5]. На основі глибокого аналізу специфіки освітнього процесу та проблем тайм-менеджменту було сформовано такі ключові вимоги до архітектури та функціоналу розроблюваного вебзастосунку:

1. Інтуїтивний та швидкодіючий інтерфейс (UI/UX). Процес взаємодії з програмою має бути максимально безшовним. Додавання нового академічного запису повинно відбуватися за мінімальну кількість кліків, щоб користувач не витрачав час на адміністрування. Інтерфейс створення завдання не повинен містити зайвих полів; обов'язковими для заповнення є лише критичні параметри: назва дисципліни чи роботи, дедлайн, пріоритетність та орієнтовний час, необхідний на виконання.
2. Візуалізація та аналіз часового навантаження. Система має перевершувати звичайні трекери, не просто зберігаючи завдання у вигляді лінійного списку, а й автоматично групуючи їх за тижнями поточного місяця.

Це дозволить студентові наочно бачити сумарну кількість годин, необхідну для навчання у конкретний період. Такий підхід дає змогу завчасно виявляти тижні з критичним перевантаженням (наприклад, тижні модульного контролю) та заздалегідь перерозподіляти свої зусилля.

3. Автоматизовані механізми пріоритезації. Застосунок повинен самостійно ранжувати завдання без потреби ручного втручання користувача. Алгоритм має автоматично сортувати картки завдань за рівнем їхньої академічної важливості (High, Medium, Low) та наближенням кінцевого терміну здачі (дедлайну). Завдяки цьому студент, відкриваючи головну панель застосунку, завжди бачитиме найважливіші та найбільш термінові завдання на самому початку списку.

4. Синхронізація даних у реальному часі. Сучасний студент вирізняється високим рівнем мобільності і використовує для навчання різноманітні пристрої: стаціонарні комп'ютери, ноутбуки в університеті та смартфони в позааудиторний час. Відповідно, застосунок має використовувати централізовану віддалену базу даних, що забезпечує миттєву синхронізацію статусів завдань та надійне збереження інформації незалежно від того, з якого комп'ютера чи смартфона студент входить у систему.

5. Інтелектуальна підтримка користувача (ШІ). З огляду на стрімкий розвиток генеративних технологій, сучасний застосунок повинен використовувати алгоритми штучного інтелекту для активної допомоги користувачу. Зокрема, корисною є наявність інтегрованого інструменту, який здатний автоматично аналізувати та розбивати складні академічні завдання на прості, зрозумілі покрокові інструкції (метод декомпозиції). Це допомагає студенту подолати ефект «чистого аркуша» та суттєво полегшує старт виконання об'ємних робіт.

1.2 Порівняння з комерційними рішеннями

На сучасному ринку програмного забезпечення представлено велику кількість рішень для організації робочого процесу та управління завданнями. Проте більшість із них розроблялася з орієнтацією на корпоративний сектор, управління бізнес-проєктами або розробку програмного забезпечення (Agile-методології). Для обґрунтування доцільності розробки власної системи електронного планування необхідно провести аналіз найбільш популярних комерційних продуктів, які студенти найчастіше намагаються адаптувати під свої академічні потреби.

Метою цього порівняння є виявлення сильних сторін існуючих систем, які варто імплементувати у власну розробку, та їхніх критичних недоліків, які роблять ці продукти незручними для щоденного використання у закладах вищої освіти.

1.2.1 Аналіз існуючих продуктів (Notion, Trello, Google Calendar)

Для детального аналізу було обрано три принципово різні за своєю архітектурою та філософією продукти: Notion (система баз даних та робочих просторів), Trello (візуальна Kanban-дошка) та Google Calendar (класичний інструмент календарного планування).

Notion – це багатофункціональний інструмент, який дозволяє створювати складні реляційні бази даних, текстові документи та Kanban-дошки в єдиному робочому просторі. Переваги: Користувач має повну свободу в налаштуванні інтерфейсу; можливість пов'язувати конспекти лекцій із конкретними завданнями; наявність великої кількості шаблонів. Недоліки для студента: Головна перевага Notion одночасно є його найбільшим недоліком у контексті швидкого тайм-менеджменту. Система має досить високий поріг входження. Щоб налаштувати автоматичне сортування завдань за пріоритетом або підрахунок навчальних годин на тиждень, користувачу необхідно самостійно створювати складні бази даних та прописувати математичні формули. Така надмірна гнучкість призводить до того, що студент витрачає більше часу на адміністрування самої системи, ніж на виконання навчальних завдань.

Trello – це класичний представник систем управління проєктами, побудований на методології Kanban. Інтерфейс складається з дощок, списків (колонок) та карток завдань, які користувач перетягує між статусами. Перевагами системи є: інтуїтивно зрозумілий візуальний інтерфейс; простота зміни статусу завдання за допомогою функції drag-and-drop; зручна система кольорових міток для різних навчальних дисциплін. Недоліки для студента: Trello погано пристосований для довгострокового хронологічного планування. У базовій безкоштовній версії відсутня зручна візуалізація тижневого навантаження. Якщо завдань стає занадто багато, вертикальні списки перетворюються на нескінченні стрічки, у яких важко орієнтуватися. Крім того, система не вміє самостійно ранжувати картки в межах однієї колонки залежно від наближення дедлайну, що вимагає постійного ручного сортування.

Google Calendar – один із найпопулярніших інструментів для тайм-блокінгу та відстеження подій, прив'язаних до конкретного часу. Переваги: Жорстка прив'язка подій до хронологічної сітки; зручна інтеграція з іншими сервісами Google; ефективна система нагадувань на мобільних пристроях. Недоліки для студента: Календар ідеально підходить для фіксації розкладу пар (подій, що мають чіткий час початку та завершення), але він менш зручний для ведення списку асинхронних завдань. Якщо студент не встиг виконати завдання у відведений часовий блок, запис просто залишається у минулому, і користувач може про нього забути, оскільки статус «Не виконано» не переноситься автоматично на наступний день у зручному вигляді. Також відсутня можливість розбивати складні події на дрібні підзадачі безпосередньо у вікні перегляду розкладу.

Проведений аналіз показує, що існуючі комерційні рішення є або занадто складними для швидкого старту, або не мають інструментів для автоматичного підрахунку тижневого академічного навантаження, або не пристосовані для управління асинхронними завданнями без чіткого часу виконання. Окрім того, у безкоштовних версіях цих продуктів відсутні вбудовані алгоритми штучного інтелекту для автоматичної декомпозиції складних навчальних проєктів.

1.2.2 Переваги власної системи з інтеграцією ШІ

Зважаючи на виявлені недоліки комерційних продуктів, розробка спеціалізованої системи управління завданнями є найбільш доцільним рішенням для здобувачів вищої освіти. Створення власного програмного забезпечення дозволяє повністю врахувати специфіку академічного середовища та впровадити сучасні технології для автоматизації процесів планування.

Однією із ключових переваг розроблюваної системи є глибока інтеграція алгоритмів генеративного штучного інтелекту. Як було зазначено раніше, однією з основних причин студентської прокрастинації є складність початкового етапу роботи над об'ємними проєктами, такими як написання курсових робіт чи підготовка до комплексних іспитів. Впровадження ШІ безпосередньо у функціонал task-трекера дозволяє реалізувати механізм автоматичної декомпозиції. Користувачу достатньо ввести назву глобального завдання, а система за допомогою інтелектуальних запитів генерує логічний покроковий план його виконання з детальними підказками. Це суттєво знижує когнітивний бар'єр, оскільки абстрактна та складно структурована задача перетворюється на набір чітких, здійсненних кроків [6].

Другою вагомою перевагою є ергономіка та цільова спрямованість інтерфейсу. На відміну від універсальних корпоративних платформ, розроблений застосунок позбавлений надлишкового функціоналу. Інтерфейс спроєктовано виключно під потреби студента: додавання, редагування та зміна статусів завдань відбувається максимально швидко. Це гарантує, що користувач не буде витрачати більше часу на підтримку самої системи планування, ніж на реальне навчання.

Третя ключова перевага полягає в алгоритмізації рутинних розрахунків. Власна система дозволяє імплементувати специфічні метрики, яких немає у стандартних рішеннях. Зокрема, йдеться про автоматичний підрахунок академічного навантаження у годинах з візуальним розбиттям по поточних тижнях місяця. Разом із вбудованими алгоритмами автоматичного сортування завдань за пріоритетністю (від найвищого до найнижчого) та наближенням дедлайну, це

створює екосистему, яка самостійно направляє фокус уваги студента на найбільш критичні ділянки роботи.

Таким чином, розробка персоналізованого вебзастосунку з інтеграцією штучного інтелекту формує ексклюзивний продукт. Він поєднує в собі простоту використання, прозору візуалізацію часових витрат та інтелектуальну підтримку користувача, що робить його значно ефективнішим інструментом для студентського тайм-менеджменту порівняно з існуючими комерційними аналогами.

РОЗДІЛ 2.
ТЕХНОЛОГІЧНИЙ СТЕК ДЛЯ РОЗРОБКИ ВЕБЗАСТОСУНКУ

2.1 Огляд екосистеми React для розробки інтерфейсів

Для розробки клієнтської частини вебзастосунку було обрано бібліотеку React [7], яка на сьогоднішній день є одним із найпоширеніших інструментів для побудови динамічних інтерфейсів з високим рівнем інтерактивності. Вибір цієї технології зумовлений необхідністю реалізації концепції SPA (Single Page Application), де взаємодія з користувачем відбувається без повного перезавантаження сторінок, що забезпечує швидкість роботи, порівнянну з нативними десктопними програмами.

Архітектурні принципи та компонентний підхід**.** React базується на декларативному підході до програмування. Замість маніпулювання кожним окремим елементом інтерфейсу вручну, розробник описує кінцевий стан компонента, а бібліотека бере на себе завдання щодо ефективного оновлення відображення. Весь інтерфейс застосунку спроєктовано як ієрархічне дерево функціональних компонентів. Такий підхід дозволяє розбити складну структуру планувальника на прості блоки: навігаційну панель, календарну сітку, списки завдань та інтерактивні форми. Використання JSX дозволяє безпосередньо в коді JavaScript описувати структуру інтерфейсу, що значно підвищує читабельність коду та полегшує його підтримку при масштабуванні проєкту [8, 9].

Система керування станом та використання Hooks. Ключовою особливістю сучасної розробки на React є використання хуків (Hooks) – спеціальних функцій, які дозволяють керувати станом та життєвим циклом компонентів без написання складних класів. У межах даного проєкту реалізовано наступні механізми:

- useState: використовується для локального керування даними всередині кожного компонента. Наприклад, стан відкритого модального вікна, поточний текст пошукового запиту або динамічні списки завдань, що редагуються користувачем (рис. 2.1);

```jsx
import { useState } from 'react';

const StateExample = () => {
  const [data, setData] = useState(null);

  return <div>{data}</div>;
};
```

Рисунок. 2.1. Приклад використання useState.

- useEffect: критично важливий хук для обробки побічних ефектів. Саме він забезпечує взаємодію з зовнішніми сервісами, такими як Firebase та Gemini API. Завдяки useEffect реалізовано автоматичне отримання даних при першому завантаженні застосунку та налаштування «слухачів» (listeners), які в реальному часі відстежують зміни в базі даних Firestore (рис. 2.2);

```jsx
import { useEffect } from 'react';

const EffectExample = () => {
  useEffect(() => {
    // Виконання побічного ефекту
  }, []);

  return <div>Компонент завантажено</div>;
};
```

Рисунок. 2.2. Приклад використання useEffect.

- useNavigate та useLocation: використовуються спільно з бібліотекою React Router для програмного керування переходами між сторінками або розділами застосунку[17]. Наприклад, ці хуки забезпечують миттєву навігацію між головною робочою панеллю завдань та іншими компонентами системи) без перезавантаження вікна браузера (рис. 2.3).

```jsx
import { useNavigate, useLocation } from 'react-router-dom';

const RouteExample = () => {
  const navigate = useNavigate();
  const location = useLocation();

  return (
    <button onClick={() => navigate('/dashboard')}>
      Поточний шлях: {location.pathname}
    </button>
  );
};
```

Рисунок. 2.3. Приклад використання useNavigate та useLocation.

Навігація та маршрутизація. Для створення структурованого багатосторінкового досвіду в межах однієї сторінки використано бібліотеку **react-router-dom**. Це дозволяє реалізувати чітку систему маршрутів: сторінка авторизації, основна робоча панель та архів. Використання компонентів <Routes> та <Route> забезпечує миттєве перемикання між розділами застосунку. Це особливо важливо для студентів, які часто використовують програму «на ходу» і потребують швидкого доступу до різних функцій без очікування завантаження серверних сторінок.

Обґрунтування вибору мови програмування JavaScript. Під час проєктування системи було прийнято свідоме рішення на користь використання JavaScript замість TypeScript. Незважаючи на популярність строгої типізації, вибір JavaScript для даного проєкту обґрунтовується наступними факторами:

1. Нижчий поріг входження та швидкість розробки. JavaScript дозволяє швидше переходити від стадії ідеї до реалізації функціоналу. Відсутність необхідності описувати складні інтерфейси та типи для кожного об'єкта (особливо при роботі з гнучкими структурами документів у NoSQL базі даних Firestore та відповідями від ШІ) дозволила суттєво пришвидшити ітераційну розробку прототипу.

2. Гнучкість при роботі з динамічними даними. Оскільки відповіді від Gemini API мають імовірнісний характер і їх структура може змінюватися залежно від запиту, динамічна природа JavaScript є більш адаптивною для обробки таких даних без необхідності постійного коригування описів типів.

3. Екосистема та прототипування. Для індивідуального проєкту, де головним пріоритетом є валідація ідеї інтелектуального планувальника, JavaScript забезпечує максимальну гнучкість. Це дозволяє зосередитися на бізнес-логіці та алгоритмах штучного інтелекту, не витрачаючи значну частину часу на налагодження сумісності типів сторонніх бібліотек.

Інтеграція з ШІ через Google Generative AI. Окремої уваги заслуговує використання бібліотеки **@google/generative-ai**. В екосистемі React взаємодія з цією бібліотекою винесена в окремий сервісний шар. Це дозволяє компонентам просто «підписуватися» на результат генерації, поки асинхронна логіка виконує складні запити до хмарних потужностей Google. Таке поєднання клієнтської бібліотеки з потужним API забезпечує можливість інтелектуальної декомпозиції завдань безпосередньо в інтерфейсі, зберігаючи при цьому високу плавність роботи застосунку.

Завдяки Virtual DOM, React виконує лише мінімально необхідні маніпуляції з реальним деревом об'єктів у браузері. Це гарантує, що навіть при активному використанні ШІ-генерації та постійній синхронізації з Firebase, інтерфейс залишається стабільним, а відгук на дії користувача - миттєвим.

2.2 Використання Firebase як хмарної інфраструктури

Сучасна розробка вебзастосунків демонструє стійку тенденцію до переходу від монолітних серверних архітектур до використання хмарних сервісів. Одним із найпопулярніших підходів є використання моделі BaaS (Backend-as-a-Service - «бекенд як послуга»). У цьому контексті платформа Firebase від Google виступає потужним інструментом, що надає готовий набір серверних рішень для розробників.

Використання Firebase дозволяє повністю уникнути етапу розробки та підтримки власного класичного бекенду (наприклад, з використанням Node.js, Python чи Java) та налаштування серверної інфраструктури. Платформа бере на себе відповідальність за маршрутизацію, безпеку, масштабування бази даних та хостинг.

Для інтеграції з клієнтськими застосунками Firebase надає офіційні SDK (Software Development Kit). У випадку використання екосистеми React, інтеграція Firebase SDK дозволяє фронтенд-застосунку безпосередньо і безпечно звертатися до бази даних, що значно прискорює цикл розробки [10]. Такий архітектурний підхід є оптимальним для сучасних SPA, оскільки дозволяє розробникам зосередити основну увагу на створенні якісного та інтерактивного користувацького інтерфейсу і логіки на стороні клієнта.

2.2.1 Переваги Firestore для роботи з даними в реальному часі

Основною базою даних, що пропонується платформою Firebase для сучасних вебзастосунків, є Cloud Firestore. Це гнучка, масштабована документо-орієнтована NoSQL база даних. Її архітектура відрізняється від традиційних реляційних таблиць: дані зберігаються у вигляді ієрархії «колекції – документи». Кожен документ є об'єктом, схожим на JSON, який містить набір пар «ключ-значення» і може включати складні вкладені об'єкти або масиви [11].

Ключовою технологічною перевагою Firestore є вбудована підтримка синхронізації даних у реальному часі. У класичній архітектурі REST API для отримання актуальних даних клієнтський застосунок змушений періодично

надсилати HTTP-запити на сервер для перевірки змін. Цей підхід створює зайве навантаження на мережу та сервер.

Натомість Firestore використовує протокол WebSockets та систему активних підписок (listeners). Коли застосунок підписується на певний документ або колекцію, сервер автоматично «надсилає» будь-які оновлення до клієнта в момент їх виникнення у базі. Для систем планування та управління завданнями це означає, що як тільки зовнішній сервіс записує нові дані в базу, інтерфейс користувача миттєво оновлюється без необхідності ручного перезавантаження сторінки.

2.2.2 Недоліки NoSQL баз даних та способи їх вирішення

Незважаючи на високу продуктивність, горизонтальне масштабування та гнучкість схеми даних, документо-орієнтовані NoSQL бази даних, такі як Firestore, мають суттєві архітектурні обмеження порівняно з реляційними SQL-базами (наприклад, PostgreSQL або MySQL). У розробці застосунків ці обмеження вимагають застосування специфічних інженерних підходів.

Відсутність повноцінних реляційних зв'язків (JOIN операцій): У NoSQL базах неможливо ефективно та швидко об'єднати дані з кількох різних колекцій одним запитом на стороні сервера. Шлях обходу: Головним методом вирішення цієї проблеми є денормалізація даних та використання вкладених структур. Замість зберігання пов'язаних сутностей у різних таблицях, пов'язані дані записуються безпосередньо всередину батьківського документа. Наприклад, кроки виконання конкретного завдання зберігаються як масив усередині документа самого завдання (рис. 2.4). Це дозволяє отримати всю необхідну інформацію за одне звернення до бази, жертвуючи при цьому певним дублюванням даних заради швидкості читання.

```
{
  "task_id": "unique_id_123",
  "title": "Підготувати звіт",
  "priority": "high",
  "status": "active",
  "ai_steps": [  // Ось це і є Embedding (вкладення)
    {"id": 1, "text": "Зібрати дані за квартал"},
    {"id": 2, "text": "Побудувати графіки в Excel"},
    {"id": 3, "text": "Оформити висновки"}
  ]
}
```

Рисунок. 2.4. Приклад денормалізованої структури документа у NoSQL.

Обмеження складності запитів та фільтрації: Firestore має жорсткі правила щодо індексації та виконання складних багатокритеріальних запитів. Наприклад, база не підтримує одночасне використання операторів нерівності (>, <, !=) для різних полів в одному запиті, а також має обмежені можливості текстового пошуку.

Для подолання цих обмежень архітектура сучасних фронтенд-застосунків передбачає перенесення частини обчислювального навантаження на сторону клієнта. Замість виконання складного сортування на сервері, застосунок отримує ширший масив даних за базовими критеріями, після чого використовує вбудовані можливості JavaScript (методи filter, sort) для точного ранжування даних безпосередньо на стороні клієнта у браузері. Це забезпечує гнучкість інтерфейсу та обходить серверні ліміти платформи.

2.3 Штучний інтелект як інструмент персоналізації навчання

Стрімкий розвиток систем штучного інтелекту та великих мовних моделей відкрив нові можливості для індивідуалізації освітнього процесу. Традиційні системи управління завданнями вимагають від користувача самостійного аналізу обсягу робіт, оцінки необхідного часу та ручного розбиття складних проєктів на менші кроки. Такий підхід часто призводить до підвищення когнітивного навантаження та прокрастинації, особливо коли студент стикається з об'ємними чи незрозумілими завданнями [12, 13].

Впровадження технологій ШІ трансформує застосунки для тайм-менеджменту з пасивних інструментів запису у проактивні системи допомоги. Використання генеративного штучного інтелекту дозволяє реалізувати такі механізми персоналізації:

1. Автоматична декомпозиція: Здатність ШІ аналізувати короткий опис абстрактного завдання (наприклад, «написати курсову роботу») і генерувати деталізований, логічно структурований покроковий план дій.
2. Адаптивні поради**:** Надання рекомендацій щодо пріоритезації та оптимального розподілу часу на основі семантичного аналізу назви завдання.
3. Зниження бар'єру входу: Користувачу достатньо сформулювати свою мету природною мовою, а ШІ бере на себе рутинну роботу з формалізації та структурування даних.

Таким чином, інтеграція штучного інтелекту виступає не просто як додаткова функція, а як ключовий інструмент зниження психологічного бар'єра перед початком виконання складних навчальних завдань.

2.3.1 Огляд можливостей API Google Generative AI (Gemini)

Для реалізації функцій штучного інтелекту в сучасних вебзастосунках розробники використовують спеціалізовані програмні інтерфейси (API), які надають доступ до потужних хмарних нейромереж. Одним із передових рішень на ринку є сімейство моделей Gemini від компанії Google, доступ до яких забезпечується через Google Generative AI API.

Gemini – це мультимодальна велика мовна модель нового покоління, здатна розуміти, узагальнювати та генерувати високоякісний контент на основі вхідних даних (промптів). Вибір цього API для освітніх та планувальних застосунків обґрунтовується низкою архітектурних та функціональних переваг:

- Висока швидкість обробки природної мови : Моделі Gemini оптимізовані для швидкої генерації тексту, що важливо для інтерактивних вебзастосунків, де користувач очікує миттєвої реакції на свої дії.
- Структурований вивід: API дозволяє налаштувати модель таким чином, щоб вона повертала відповідь не у вигляді суцільного тексту, а у строго визначеному машиночитаному форматі, наприклад, JSON. Це дозволяє легко інтегрувати згенеровані кроки або поради безпосередньо в базу даних застосунку без додаткового парсингу.
- Контекстне розуміння: Модель здатна враховувати специфіку академічного середовища. При правильному налаштуванні промпту, Gemini може генерувати поради, адаптовані саме для студентів, враховуючи принципи тайм-менеджменту.
- Простота інтеграції: Завдяки офіційному SDK для JavaScript/Node.js, інтеграція API у клієнтський застосунок на базі React відбувається безпечно та не вимагає розгортання окремих серверних потужностей для обробки ШІ-моделей (рис. 2.5).

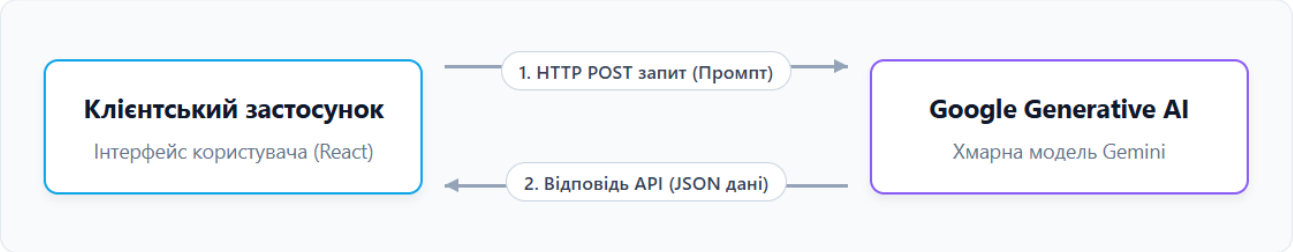

Рисунок. 2.5. Схема взаємодії клієнтського застосунку з API генеративного штучного інтелекту

Технологічна архітектура Gemini API базується на архітектурі Transformer, що дозволяє моделі обробляти великі масиви контексту одночасно. Це особливо важливо для навчальних завдань, де контекст може включати не лише назву задачі, а й додаткові методичні вказівки або попередні нотатки студента. Модель здатна виявляти семантичні зв'язки між різними предметами, пропонуючи міждисциплінарні підходи до вирішення завдань.

Окремої уваги заслуговують інструменти безпеки та фільтрації контенту, інтегровані в Google Generative AI API. Для освітніх застосунків це критично важливо, оскільки система має вбудовані фільтри, що блокують генерацію шкідливого, неетичного або небезпечного контенту.

Наприклад, система встановлює суворі обмеження на генерацію контенту за кількома ключовими напрямками, зокрема: запобігання використанню мови ворожнечі, блокування будь-яких проявів кібербулінгу чи психологічного переслідування, фільтрація матеріалів відвертого характеру, а також жорстка заборона на створення описів небезпечних або незаконних дій. Такий багаторівневий підхід до автоматичної модерації даних відіграє вирішальну роль у мінімізації репутаційних та етичних ризиків, гарантуючи абсолютно безпечне, контрольоване та комфортне академічне середовище для кожного користувача.

Таким чином, використання можливостей Gemini API перетворює вебзастосунок на інтелектуальну екосистему, де ШІ виступає посередником між ідеєю користувача та її практичною реалізацією. Перехід від простої фіксації завдань до їхнього автоматичного структурування та експертного супроводу дозволяє делегувати штучному інтелекту найбільш виснажливі процеси декомпозиції цілей. Впровадження такої інтелектуальної підсистеми забезпечує якісно новий рівень продуктивності, допомагаючи здобувачам освіти долати складні академічні виклики завдяки ефективній взаємодії визначеної користувачем мети та обчислювальної потужності генеративного штучного інтелекту.

РОЗДІЛ 3.
ПРОЄКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ ТА РОЗРОБКА КОРИСТУВАЦЬКОГО ІНТЕРФЕЙСУ

3.1 Архітектура вебзастосунку та структура бази даних

Проєктування архітектури застосунку базується на інженерному принципі розділення відповідальності [14, 15]. Оскільки теоретичне обґрунтування вибору технологічного стека було наведено у попередньому розділі, фокус даного етапу зосереджено на розробці практичної моделі взаємодії компонентів системи. Архітектура програми побудована за трирівневою клієнт-серверною моделлю, де кожен прошарок виконує чітко ізольовану функцію, гарантуючи високу модульність та легкість подальшого масштабування кодової бази.

Першим і найвищим рівнем виступає рівень представлення, реалізований за допомогою компонентів React [16]. Цей шар відповідає виключно за рендеринг графічного інтерфейсу, відстеження дій користувача та маршрутизацію між екранами планувальника. Він є повністю відокремленим від прямої роботи з базою даних, що відповідає сучасним стандартам розробки безпечних вебзастосунків.

Другим рівнем є прошарок бізнес-логіки та інтеграції зовнішніх сервісів. Саме тут зосереджені основні алгоритми системи: функції обробки вхідних даних, механізми формування запитів до нейромережі Gemini для декомпозиції завдань, а також парсинг отриманих результатів [12, 13]. Цей рівень діє як маршрутизатор, який отримує команди від інтерфейсу, звертається до інтелектуальних API та передає підготовлену інформацію далі.

Третім рівнем виступає шар доступу до даних, який забезпечує асинхронну взаємодію з хмарною інфраструктурою Firebase. На цьому етапі реалізовано методи створення, читання, оновлення та видалення записів у базі даних Firestore [11]. Використання такої чіткої ієрархії потоків інформації дозволяє уникнути хаотичного змішування коду інтерфейсу із запитами до сервера, що робить архітектуру застосунку максимально передбачуваною та стійкою до критичних помилок.

3.2 Розробка вебінтерфейсу та компонентної бази

Розробка користувацького інтерфейсу застосунку базується на компонентному підході, який є фундаментальною парадигмою екосистеми React. Цей підхід дозволяє декомпозувати складний візуальний інтерфейс на незалежні, ізольовані та придатні для багаторазового використання блоки. Таке архітектурне рішення значно спрощує тестування окремих елементів системи та забезпечує високу підтримуваність кодової бази при масштабуванні проєкту [16, 17].

Для візуального оформлення та стилізації вебінтерфейсу було інтегровано сучасний CSS-фреймворк Tailwind CSS [18]. Використання utility-first методології, що передбачає застосування низькорівневих атомарних класів безпосередньо в JSX-розмітці, дозволило уникнути створення громіздких зовнішніх таблиць стилів та конфліктів CSS-селекторів. Завдяки вбудованим механізмам Tailwind CSS у застосунку реалізовано адаптивний дизайн. Це гарантує коректне масштабування та реорганізацію елементів інтерфейсу як на широкоформатних десктопних моніторах, так і на екранах мобільних пристроїв.

Клієнтська архітектура застосунку спроєктована з урахуванням принципів модульності та атомарного дизайну. Усі React-компоненти логічно розділені на три ієрархічні рівні залежно від їхньої складності та зони відповідальності (рис. 3.1):

1. Базові UI-елементи: До цієї категорії належать кнопки, поля введення тексту, чекбокси, індикатори завантаження. Ці компоненти є «чистими» - вони не містять складної бізнес-логіки, не взаємодіють безпосередньо з базою даних, а їхній зовнішній вигляд повністю залежить від отриманих вхідних параметрів (props).
2. Складені компоненти: Включають картки навчальних завдань, форми створення нових записів та модальні вікна. Вони об'єднують базові елементи у функціональні блоки, здатні обробляти локальний стан) та реагувати на складні події користувача.
3. Компоненти-контейнери: Відіграють роль сторінок або головних структурних блоків. Вони відповідають за встановлення з'єднання з Firebase, отримання масивів даних із хмарної бази та їхню передачу

дочірнім компонентам для подальшого рендерингу на екрані. Програмна реалізація структури компонентів та логіка їхньої взаємодії наведена у додатку А.

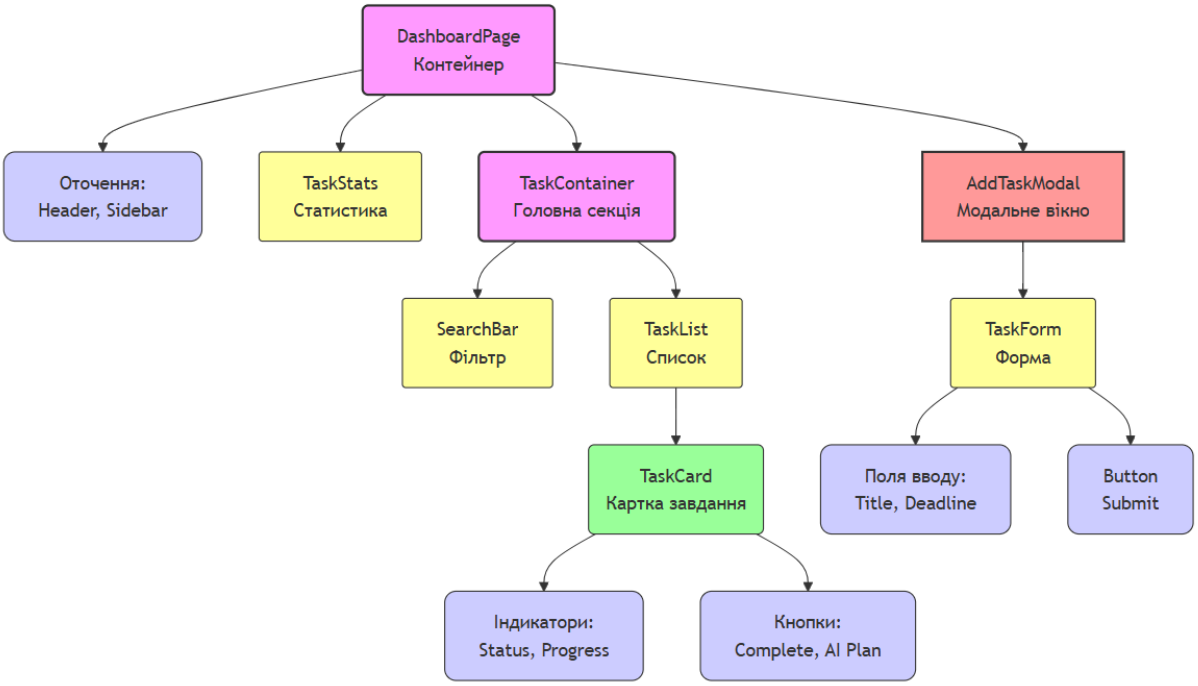

Рисунок 3.1. Ієрархічна структура React-компонентів застосунку

Важливим аспектом розробки компонентної бази є управління станом. У проєкті використовуються сучасні функціональні компоненти у поєднанні з React Hooks. Зокрема, хук useState забезпечує збереження та оновлення локальних даних інтерфейсу, виконуючи роль механізму відстеження введеного тексту у формах, тоді як useEffect відповідає за обробку побічних ефектів – підключення слухачів подій та виконання асинхронних запитів до Firestore під час монтування сторінок.

Окрему увагу під час проєктування інтерфейсу було приділено системі модальних вікон та механізмам зворотного зв'язку. Оскільки взаємодія з ШІ та детальне планування завдань вимагають підвищеної концентрації уваги, ці процеси винесено у спливаючі діалогові вікна (рис. 3.2). Такий підхід дозволяє студенту зосередитися на конкретній задачі без необхідності переходу на нову сторінку, зберігаючи візуальний контекст попереднього робочого простору.

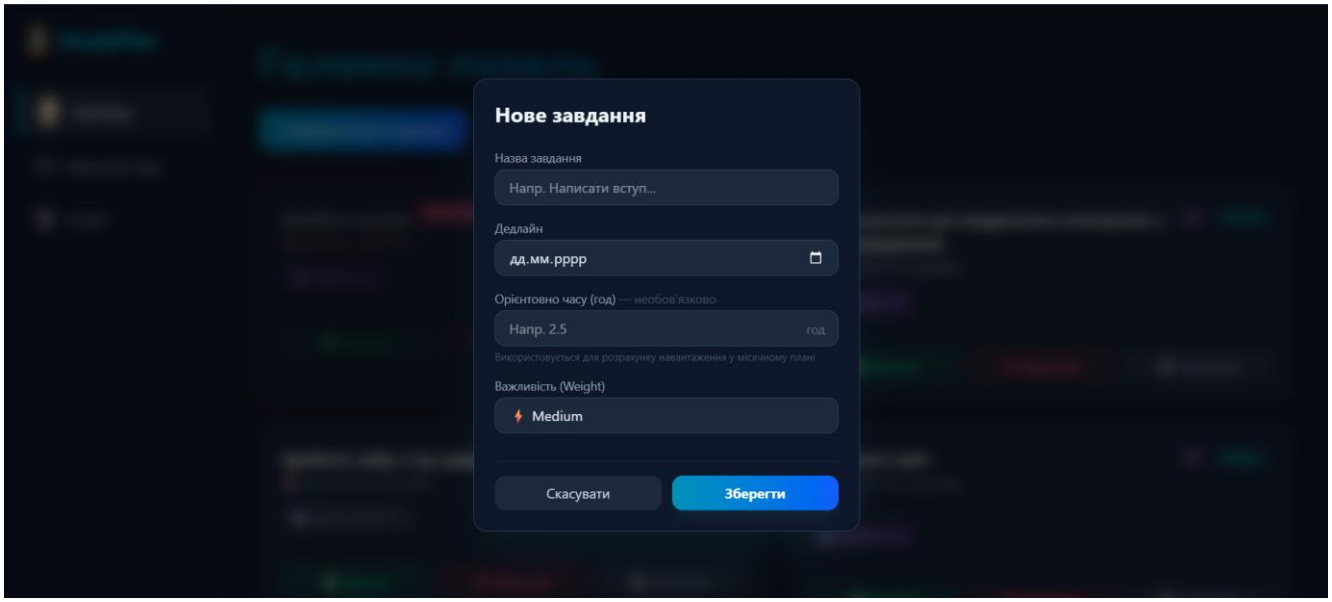

Рисунок 3.2. Реалізація модального вікна для фокусного виконання задач

Невід'ємною складовою якісного користувацького досвіду є коректна обробка асинхронних операцій. Враховуючи специфіку взаємодії з хмарною базою даних Firestore та інтеграцію з API штучного інтелекту, у вебінтерфейсі передбачено механізми управління станом під час очікування відповіді від сервера. Це гарантує цілісність даних, синхронізацію інтерфейсу з актуальним станом бази даних та запобігає дублюванню запитів при створенні нових навчальних завдань чи генерації планів за допомогою ШІ. Такий підхід забезпечує стабільну роботу застосунку та робить його зрцчним для кінцевого користувача.

3.2.1 Прототипування (Wireframing) та UX/UI рішення

Процес розробки користувацького інтерфейсу розпочався зі створення структурних прототипів (wireframes). На цьому етапі (рис. 3.3) було спроєктовано базову сітку сторінки: лівосторонню навігаційну панель, верхній блок керування та грід-систему для відображення карток завдань. Це дозволило затвердити логіку розташування елементів, інформаційну архітектуру та забезпечити інтуїтивно зрозумілу навігацію ще до етапу написання коду та вибору фінального кольорового оформлення.

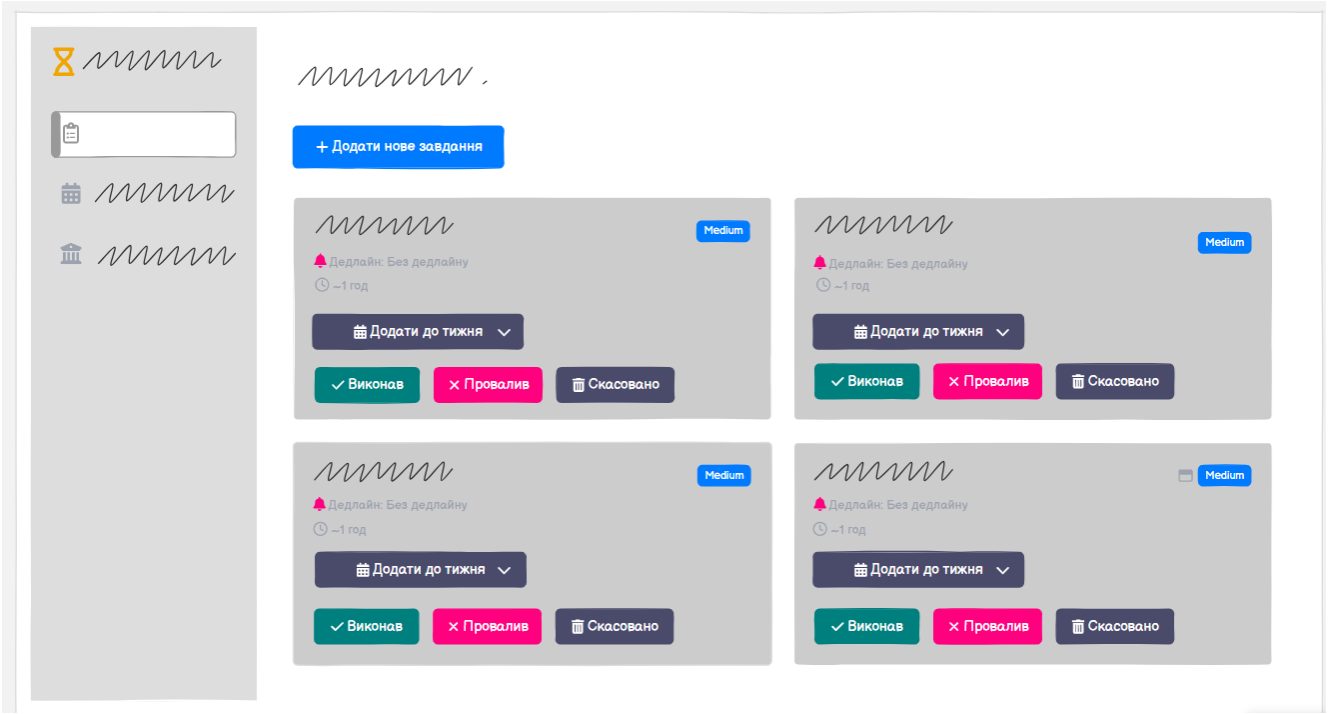

Рисунок 3.3. Структурний прототип головної панелі застосунку

Щодо візуальних UI-рішень, фінальний дизайн системи реалізовано у темній темі), що є сучасним стандартом для застосунків, які передбачають тривалу концентрацію уваги. Завдяки можливостям фреймворку Tailwind CSS було сформовано узгоджену дизайн-систему. Основний фон застосунку виконано у глибоких темно-синіх та графітових відтінках, що значно знижує навантаження на зір студента під час тривалого планування або навчання.

Для створення чіткої візуальної ієрархії та ефективного керування увагою користувача використано яскраві акцентні кольори. Насичений синій відтінок

застосовується для головних цільових дій, зокрема для виділення центральної кнопки додавання нового завдання на панелі керування. Водночас фіолетовий колір логічно відокремлює інноваційні функції штучного інтелекту, слугуючи візуальним маркером для інструментів генерації деталізованого ШІ-плану виконання.

Крім того, в інтерфейсі впроваджено комплексну семантичну кольорову індикацію статусів, яка на інтуїтивному рівні інформує студента про стан поточних справ. У цій системі зелений колір позначає успішно виконані навчальні цілі, стриманий сірий використовується для скасованих ініціатив, а насичений червоний виступає чітким сигналом для прострочених дедлайнів або провалених задач.

Геометрія інтерфейсу базується на використанні відокремлених карток із м'яко заокругленими кутами та тонкими підсвіченими контурами. Це архітектурне рішення чітко відмежовує інтерактивні блоки від загального темного фону, створюючи відчуття просторової глибини. Наприклад, застосування яскравої червоної рамки навколо картки простроченого завдання працює як додатковий візуальний тригер. Він миттєво фокусує погляд на критично важливій проблемі та спонукає користувача до негайних дій. Додатковим елементом покращення навігації є використання лаконічної іконографіки та компактних тегів пріоритетності, що дозволяє швидко оцінити складність кожної цілі без необхідності вчитуватися в текст.

Приклади візуальної реалізації ключових екранів розробленого користувацького інтерфейсу наведені у додатку Б.

3.2.2 Реалізація головної панелі та системи маршрутизації

Для забезпечення швидкої та безшовної навігації між функціональними розділами застосунку використано архітектурний підхід односторінкового застосунку. Динамічна маршрутизація клієнтської частини реалізована за допомогою спеціалізованої бібліотеки React. Зокрема, у даному проєкті застосовано популярне рішення React Router [17]. Цей інструмент дозволяє ефективно перехоплювати зміни адреси в пошуковому рядку та миттєво підміняти контент робочої області без необхідності повного перезавантаження вебсторінки браузером. Основна навігаційна структура зосереджена у фіксованій бічній панелі, яка в термінології проєктування інтерфейсів називається сайдбаром. Вона включає три ключові маршрути: головну панель керування, розділ довгострокового планування та архів історії завдань. Такий механізм навігації суттєво економить мережевий трафік та забезпечує стабільно високу швидкість відгуку інтерфейсу на дії користувача.

Головна інформаційна панель керування, що функціонує як інтерактивний дашборд, виступає центральним вузлом екосистеми застосунку. З архітектурної точки зору, цей елемент виконує роль розумного компонента-контейнера. Його першочергова функція полягає в агрегації даних та координації взаємодії між дочірніми компонентами. До таких елементів відображення належать списки завдань, кнопки керування станами та спливаючі діалогові вікна. Під час монтування цієї сторінки автоматично ініціюється асинхронний запит до хмарної бази даних Firestore для отримання актуального масиву навчальних завдань студента з сервера.

Отримані з бази даних масиви інформації надійно зберігаються у локальному стані головного компонента за допомогою хуків. Після цього вони каскадно передаються вниз по ієрархічному дереву компонентів через базовий для React механізм передачі властивостей. Візуальне представлення інтерфейсу головної панелі керування наведено у додатку Б.1.

3.2.3 Візуалізація місячного навантаження та історії завдань

Для комплексного управління навчальним процесом, окрім щоденного планування на головній панелі, критично важливою є можливість довгострокового аналізу та ретроспективного огляду виконаної роботи. З цією метою у структурі вебінтерфейсу розроблено два додаткові функціональні розділи: модуль візуалізації місячного навантаження та архів історії завдань. Перехід до цих сторінок здійснюється через інтерактивне бічне меню навігації, що забезпечує логічне розділення оперативного та стратегічного планування без перевантаження контенту робочої області надлишковою інформацією.

Розділ візуалізації місячного навантаження спроєктований для ефективного управління довгостроковими цілями та відстеження глобальних дедлайнів. Відмінною рисою розробленого інтерфейсу є відхід від традиційного відображення у вигляді класичного календаря на користь методології планування за тижневими спринтами. Технічна реалізація цього компонента передбачає вибірку активних завдань із хмарної бази Firestore та їхнє алгоритмічне групування за чотирма основними тижнями місяця.

Особлива увага приділена системі автоматичного підрахунку академічного навантаження. Для кожного завдання студент може вказати орієнтовний час на його виконання. Програмна логіка застосунку агрегує ці дані в межах кожного тижня та формує візуальні індикатори завантаженості, зокрема такі статуси, як «Легко» або «Перевантаження», що супроводжуються відповідним кольоровим кодуванням та шкалою прогресу. Крім того, інтерфейс гнучко обробляє неповні дані: для карток, яким ще не призначено конкретний період виконання, передбачено окремий структурований блок «Без тижня». Приклад візуальної організації цього розділу наведено у додатку Б.2. Такий підхід дозволяє студенту наочно оцінити щільність свого графіка, заздалегідь виявити періоди пікового навантаження та превентивно перерозподілити задачі для уникнення перевтоми.

Модуль історії завдань виконує роль цифрового архіву та інструменту глибокого самоаналізу для користувача. У цьому розділі, візуальна реалізація якого представлена у додатку Б.3, акумулюються всі завершені, скасовані або прострочені навчальні цілі. На рівні програмного коду компонент застосовує фільтрацію за статусом, відсікаючи активні завдання та формуючи хронологічний перелік виключно неактивних записів. Окрім простого відображення списку, модуль включає блок статистичної аналітики: на основі співвідношення виконаних та провалених дедлайнів система генерує кругову діаграму загальної успішності студента.

Винесення цієї інформації в окремий маршрут вирішує відразу два архітектурні та користувацькі завдання. По-перше, це повністю звільняє оперативний простір головної панелі керування, приклад якої наведено у додатку Б.1, від неактуальних на даний момент даних. По-друге, створення видимого архіву виконаної роботи та графічної статистики формує потужний позитивний психологічний ефект підкріплення, наочно демонструючи реальний обсяг досягнутих результатів за весь період використання застосунку.

РОЗДІЛ 4.
ІНТЕГРАЦІЯ ШТУЧНОГО ІНТЕЛЕКТУ ТА РЕАЛІЗАЦІЯ БІЗНЕС-ЛОГІКИ ДОДАТКУ

4.1 Механізми взаємодії з API генеративного ШІ

Інтеграція функцій штучного інтелекту в сучасні системи управління навчальними завданнями є закономірним кроком в еволюції програмного забезпечення для тайм-менеджменту. Перехід від пасивних інструментів типу CRUD (створення, читання, оновлення, видалення), які лише зберігають введені користувачем дані, до проактивних інтелектуальних систем вимагає побудови стабільного та швидкого каналу зв’язку між клієнтською частиною застосунку та хмарною нейромережевою моделлю. Успішна реалізація такого зв'язку дозволяє делегувати системі рутинні аналітичні процеси, зокрема декомпозицію складних академічних завдань.

У процесі проєктування даного застосунку було проведено порівняльний аналіз кількох провідних генеративних моделей на ринку. Для кінцевої імплементації було обрано модель Google Gemini 1.5 Flash. Цей вибір обґрунтовується низкою архітектурних та економічних переваг:

1. Оптимізація швидкості: Версія Flash спеціально оптимізована виробником для виконання завдань із мінімальною затримкою відповіді. У контексті користувацького інтерфейсу швидкість генерації тексту має критичне значення для комфорту користувача.
2. Висока точність: Модель демонструє високу семантичну відповідність при виконанні інструкцій за короткими текстовими запитами (промптами).
3. Доступність: Наявність безкоштовного рівня доступу з достатніми квотами робить цю технологію оптимальною для розробки та розгортання студентських проєктів.

На програмному рівні взаємодія між React-застосунком та серверами Google реалізована за допомогою офіційної JavaScript-бібліотеки @google/generative-ai. Цей пакет виконує роль абстракційного шару, що інкапсулює складну логіку формування мережевих пакетів та серіалізації даних (рис. 4.1).

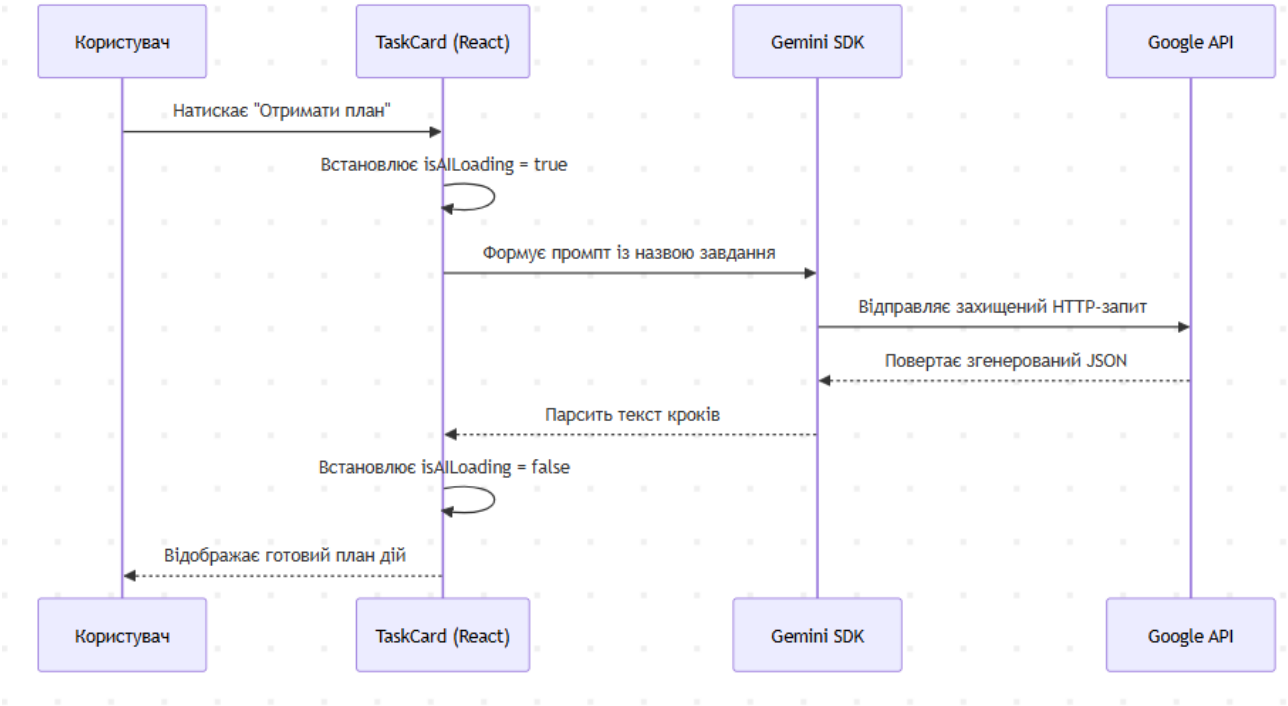

Рисунок 4.1 Діаграма послідовності взаємодії клієнтського застосунку з Google Gemini API

Окрему увагу під час проєктування механізмів взаємодії з API було приділено питанням інформаційної безпеки. Будь-який запит до хмарних сервісів Google вимагає автентифікації за допомогою унікального ключа доступу API. Зберігання таких конфіденційних токенів безпосередньо у клієнтському коді шляхом прямого вписування є грубим порушенням стандартів безпеки, оскільки це створює ризик компрометації ключа при публікації сирцевого коду у відкритих репозиторіях, зокрема на сервісі GitHub.

Для вирішення цієї проблеми архітектурою проєкту передбачено винесення ключа у змінні оточення. Використання файлу .env дозволяє ізолювати секретні дані від основного коду. Під час ініціалізації моделі система звертається до об'єкта import.meta.env, що забезпечує гнучкість налаштувань при деплої на різні сервери (рис. 4.2).

```
import { GoogleGenerativeAI } from '@google/generative-ai';

const genAI = new GoogleGenerativeAI(import.meta.env.VITE_GEMINI_API_KEY);
```

Рисунок 4.2 Безпечна ініціалізація SDK за допомогою змінних оточення

Такий підхід гарантує, що конфіденційні дані залишаються захищеними на етапі розробки та можуть бути безпечно сконфігуровані на стороні хостинг-провайдера під час фінального розгортання (деплою) програмного продукту.

Окрім питань безпеки, важливою складовою надійної роботи з API є забезпечення відмовостійкості системи та управління станом інтерфейсу. Оскільки звернення до нейромережі є асинхронною операцією, що потребує певного часу, програмною логікою передбачено блокування елементів керування (через стан завантаження) на період очікування відповіді. Це запобігає випадковому дублюванню запитів з боку користувача та неефективному витрачанню квот API.

Крім того, процес мережевої взаємодії інкапсульовано у блоки обробки винятків (try...catch). У разі виникнення непередбачуваних ситуацій, таких як відсутність інтернет-з'єднання у користувача, перевищення лімітів запитів або тимчасова недоступність серверів Google, система коректно перехоплює помилку. Замість критичного збою застосунку, користувач отримує відповідне інформаційне повідомлення, що дозволяє зберегти стабільність роботи всього клієнтського середовища.

4.1.1 Промпт-інжиніринг для отримання структурованих порад

Ефективність використання великих мовних моделей у програмних продуктах безпосередньо залежить від якості та структури вхідних запитів. Цей процес оптимізації взаємодії між системою та нейромережею отримав назву промпт-інжинірингу [19]. Головною проблемою при вільному неструктурованому зверненні до штучного інтелекту є висока ймовірність отримання надлишкової, розмитої або алгоритмічно нерелевантної інформації. Для застосунку персонального планування завдань критично важливо, щоб користувач отримував лаконічні та чіткі інструкції, які можна швидко переглянути у компактній картці інтерфейсу.

З огляду на це, у програмній логіці компонента генерації порад було розроблено жорстко структурований шаблон запиту. Він формується динамічно за допомогою механізму шаблонних рядків мови JavaScript, куди програмно підставляється назва конкретного завдання, введена студентом. Оптимізований запит складається з трьох обов'язкових логічних частин:

1. Встановлення контексту та ролі: система налаштовує модель на роботу в якості віртуального помічника студента, що дозволяє уникнути абстрактних філософських міркувань нейромережі та сфокусувати її на практичних порадах.
2. Вимоги до форматування: для забезпечення коректного відображення результату у вебінтерфейсі без написання складних алгоритмів синтаксичного аналізу тексту, у запиті міститься пряма вказівка генерувати відповідь виключно у вигляді маркованого списку.
3. Кількісні обмеження: для збереження зручності користувацького інтерфейсу та економії місця на екрані встановлено ліміт на генерацію від трьох до чотирьох конкретних кроків. Це запобігає ситуаціям, коли модель генерує надмірно довгий текст, який виходив би за межі візуального контейнера картки завдання..

Програмна реалізація формування такого запиту виглядає як конкатенація базової інструкції та змінної із назвою цілі користувача (рис. 4.3).

```
const handleAiBreakdown = async (e) => {
    e.stopPropagation();
    setIsAILoading(true);
    setIsExpanded(true);
    setIsSaved(false);
    try {
      const model = genAI.getGenerativeModel({ model: "gemini-flash-latest" });
      const prompt = `Ти помічник студента. Завдання: "${task.title}". Напиши 3-4 дуже короткі і
      конкретні кроки для його виконання. Формат: тільки список маркерів.`;
      const result = await model.generateContent(prompt);
      setAiSteps(result.response.text());
    } catch (error) {
      setAiSteps("Помилка. Спробуй ще раз.");
    } finally {
      setIsAILoading(false);
    }
  };
```

Рисунок 4.3 Програмна реалізація генерації та обробки підказок від ШІ

Завдяки такій інженерній архітектурі запиту, незалежно від специфіки введеної інформації, чи то підготовка до екзамену з вищої математики, чи організація побутового процесу, нейромережа повертає високоякісний стандартизований результат. Отриманий текст не потребує складної постобробки на стороні клієнта і відразу передається у функцію оновлення стану для подальшого відображення та збереження у базі даних.

Окрім безпосереднього формування промпту, програмна реалізація методу генерації (рис. 4.3) базується на принципах асинхронного програмування та включає комплексне управління станами користувацького інтерфейсу. Перед ініціалізацією мережевого запиту система переводить інтерфейс у режим очікування за допомогою функціонального стану setIsAILoading. Це не лише активує візуальну індикацію процесу у вигляді анімації завантаження, а й програмно блокує можливість дублювання запитів, що запобігає надмірному навантаженню на канали зв'язку та нераціональному використанню лімітів API.

Паралельно з цим метод забезпечує динамічну зміну візуального представлення картки завдання, примусово розгортаючи її для відображення майбутньої відповіді та скидаючи індикатори попереднього збереження. Такий підхід гарантує цілісність відображення даних у реальному часі.

4.1.2 Обробка та збереження згенерованих кроків у Firestore

Після успішного отримання текстової відповіді від моделі Google Gemini дані зберігаються у локальному стані компонента для миттєвого відображення користувачу. Однак для забезпечення довгострокового зберігання та доступу до цих порад у майбутніх сесіях необхідно реалізувати механізм синхронізації з хмарною базою даних NoSQL Firestore [8]. Процес збереження ініціюється користувачем лише після ознайомлення зі згенерованим планом, що дозволяє уникнути надлишкового навантаження на систему та засмічення бази даних нерелевантними або випадковими запитами.

Архітектурно цей процес розділений на два взаємопов’язані рівні. На рівні сервісів, що виконує роль прошарку доступу до даних, у файлі taskService.js реалізовано спеціалізовану асинхронну функцію updateTaskAdvice. Ця функція використовує метод updateDoc з офіційної бібліотеки Firebase Firestore, який дозволяє виконувати точкове оновлення виключно конкретного поля документа [12]. Це гарантує цілісність інформації, оскільки при оновленні порад від штучного інтелекту інші атрибути завдання, зокрема його назва, пріоритет або статус, залишаються незмінними.

Водночас на рівні компонентів, який відповідає за логіку користувацького інтерфейсу, у файлі TaskCard.jsx реалізовано функцію-обробник для взаємодії з кнопкою збереження. Вона забезпечує передачу ідентифікатора завдання та тексту порад до рівня сервісів, а також керує візуальним підтвердженням успішності операції (рис. 4.4).

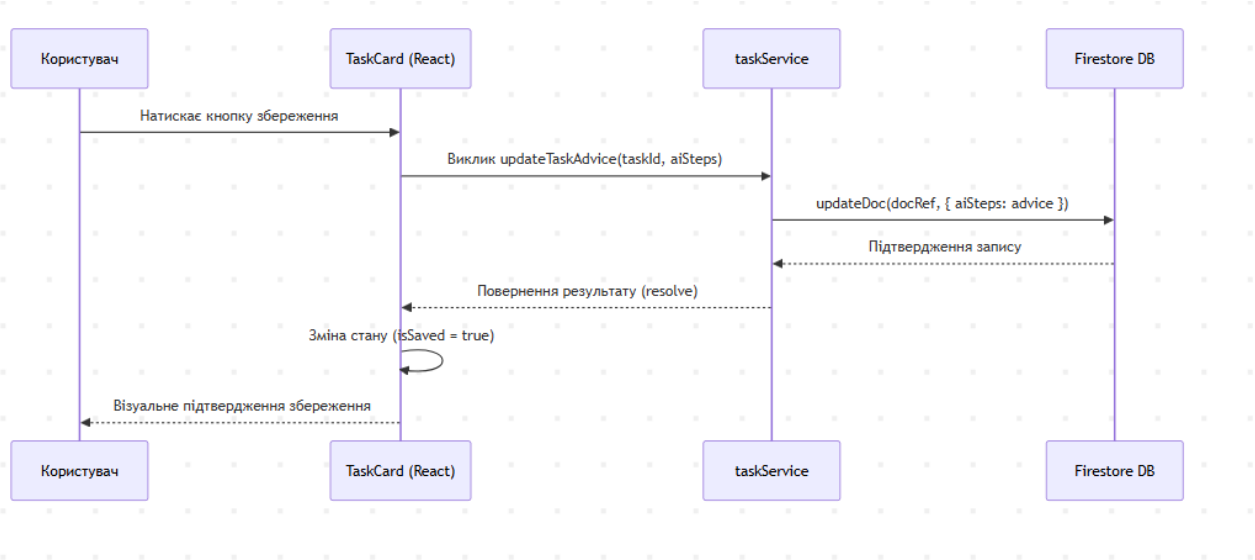

Рисунок 4.4 Схема взаємодії між клієнтом та Firestore при збереженні даних

Технічна особливість реалізації полягає у використанні унікального ідентифікатора документа firebaseId для точного позиціонування запису в колекції. Оскільки Firestore є документо-орієнтованою базою даних, оновлення відбувається за прямим посиланням на об'єкт. Програмну реалізацію функції оновлення, яка забезпечує запис згенерованих кроків у поле aiSteps (рис. 4.5).

```
export const updateTaskAdvice = async (taskId, advice) => {
  try {
    const taskRef = doc(db, 'tasks', taskId);
    await updateDoc(taskRef, { aiSteps: advice });
  } catch (error) {
    console.error("Помилка при збереженні поради:", error);
    throw error;
  }
};
```

Рисунок 4.5 Програмний код функції оновлення документа в базі даних

Важливим аспектом проектування інтерфейсу є миттєвий зворотний зв'язок. Після успішного завершення мережевої операції стан компонента isSaved змінюється на позитивний, що призводить до зміни візуального стилю елемента керування. Така дволанкова система обробки даних гарантує високу швидкість роботи інтерфейсу за рахунок локальних станів React.

4.2 Алгоритми пріоритезації та управління станами завдань

Розробка ефективної системи тайм-менеджменту вимагає не лише забезпечення можливості створення записів, але й побудови надійної архітектури управління їхнім життєвим циклом. У контексті студентського навантаження, завдання рідко мають лише бінарний стан (створено або виконано). Вони можуть бути прострочені, скасовані через зміну навчальних планів або переведені в архів. Крім того, критично важливою вимогою є автоматична пріоритезація завдань, щоб фокусувати увагу користувача на найважливіших та найтерміновіших дедлайнах. З огляду на це, логіка застосунку була розділена на модуль управління станами та модуль багатокритеріального сортування.

4.2.1 Архітектура управління життєвим циклом завдань

Кожен об'єкт завдання у системі розглядається як сутність із визначеним набором допустимих станів, що утворюють концептуальну кінцеву модель автоматів [7]. Під час створення нового завдання у компоненті TaskModal йому за замовчуванням присвоюється статус active (активне). Цей базовий статус є маркером того, що завдання потребує виконання і має відображатися на головній панелі інструментів.

Управління зміною станів реалізовано на рівні користувацького інтерфейсу через взаємодію з кнопками дій у розгорнутій картці завдання. Користувачу надається три альтернативні шляхи завершення роботи над завданням, кожному з яких відповідає унікальний ідентифікатор стану:

- completed – успішне виконання завдання;
- missed – невиконання в межах встановленого терміну (провал);
- cancelled – скасування завдання як неактуального.

Технічна реалізація цього процесу ініціюється викликом локальної функції-обробника, яка приймає цільовий статус та передає його до сервісного шару. Для забезпечення цілісності даних застосовується асинхронна функція updateTaskStatus з модуля taskService.js (рис. 4.6). Вона здійснює точкове оновлення поля status відповідного документа у базі даних Firestore за допомогою методу updateDoc.

Важливим аспектом є те, що зміна статусу не призводить до видалення документа з колекції, що дозволяє зберігати повну історію активності студента для подальшого аналізу.

Завдяки впровадженню технології підписки на зміни в реальному часі, після оновлення статусу на сервері, Firestore миттєво відправляє оновлений масив даних на клієнтську частину. React-компонент перехоплює ці зміни і автоматично ініціює процес повторного рендерингу. У результаті завдання, яке втратило статус active, динамічно зникає з головного екрана, забезпечуючи ефект миттєвого відгуку системи без необхідності ручного оновлення сторінки браузера користувачем.

```
export const updateTaskStatus = async (taskId, newStatus) => {
  try {
    const taskRef = doc(db, "tasks", taskId);
    await updateDoc(taskRef, { status: newStatus });
  } catch (error) {
    console.error("Помилка оновлення статусу:", error);
  }
};
```

Рисунок 4.6 Програмна реалізація методу оновлення статусу завдання у хмарній базі даних

4.2.2 Багатокритеріальний алгоритм пріоритезації на головному екрані

Наявність великої кількості активних завдань може викликати когнітивне перевантаження у користувача, що знижує ефективність планування [3, 4]. Для вирішення цієї проблеми у компоненті головної панелі було розроблено алгоритм автоматичної пріоритезації. На відміну від простих застосунків, які сортують елементи лише за датою створення, розроблений алгоритм враховує відразу кілька зважених критеріїв, імітуючи людську логіку прийняття рішень.

Процес впорядкування даних відбувається на стороні клієнта за допомогою вбудованого методу масивів JavaScript sort() [6, 11]. Базовий масив, отриманий з бази даних, спочатку проходить етап фільтрації: метод filter(task => task.status ===

'active') ізолює лише ті завдання, які потребують уваги. Далі до відфільтрованого масиву застосовується двоступенева логіка порівняння.

Першим та найважливішим критерієм є рівень важливості завдання (Weight). Для програмного опрацювання текстових значень пріоритетів застосовано патерн об'єкта-словника, де кожному рівню присвоєно числову вагу: High дорівнює 3, Medium - 2, а Low - 1. Алгоритм обчислює різницю між ваговими коефіцієнтами двох порівнюваних об'єктів. Якщо один об'єкт має вищий пріоритет, функція сортування негайно розміщує його вище у списку (рис. 4.7).

```
tasks
  .filter(task => task.status === 'active')
  .sort((a, b) => {
    // за пріоритетом
    const weights = { 'High': 3, 'Medium': 2, 'Low': 1 };
    const weightDiff = (weights[b.weight] || 0) - (weights[a.weight] || 0);

    if (weightDiff !== 0) return weightDiff;

    //Якщо пріоритет однаковий
    if (!a.deadline || a.deadline === 'Без дедлайну') return 1;
    if (!b.deadline || b.deadline === 'Без дедлайну') return -1;
    return new Date(a.deadline) - new Date(b.deadline);
  })
```

Рисунок 4.7 Реалізація двоступеневого алгоритму сортування масиву завдань

Другий ступінь пріоритезації активується у випадку конфлікту, коли два завдання мають однаковий рівень важливості. У такій ситуації алгоритм переходить до порівняння хронологічних метрик – дедлайнів. Програмна логіка конвертує строкові представлення дат у стандартні об'єкти Date мови JavaScript, що дозволяє виконувати математичні операції порівняння. Завдання з ближчими термінами здачі отримують вищий пріоритет.

Особливістю даної реалізації є коректна обробка крайових випадків. Зокрема, якщо користувач створив завдання без вказаної дати виконання (статус «Без дедлайну»), алгоритм містить умовні оператори, які примусово зміщують такі елементи у кінець списку в межах своєї групи пріоритетності. Це гарантує, що термінові завдання завжди залишатимуться у верхній частині екрана.

4.2.3 Ізоляція станів та формування історії активності

Наслідком роботи описаних алгоритмів є чіткий розподіл інформаційних потоків у застосунку. Завдання, які змінили свій статус з active на будь-який інший, автоматично виключаються з процесу рендерингу на головному екрані. Однак вони стають доступними у спеціалізованому компоненті історії (History.jsx).

Такий архітектурний підхід, відомий як логічне видалення, має значні переваги над фізичним знищенням записів у базі даних. Він дозволяє реалізувати функціонал збору статистичних даних щодо продуктивності студента. Відфільтровані неактивні завдання групуються за їхніми фінальними статусами, що дає змогу візуалізувати відсоток успішних, провалених та скасованих планів, створюючи базу для самоаналізу та покращення навичок тайм-менеджменту в майбутньому.

4.3 Схема життєвого циклу завдання в системі

Розробка надійної архітектури програмного забезпечення вимагає чіткої формалізації бізнес-процесів [5] та розуміння того, як дані трансформуються впродовж усього часу використання застосунку. У системі управління навчальним часом центральним інформаційним об'єктом є завдання студента. Життєвий цикл цього об'єкта концептуально побудований за принципом моделі скінченних автоматів, де сутність послідовно проходить через визначені стани від моменту створення до остаточного архівування.

Процес розпочинається з етапу ініціалізації. Користувач взаємодіє з модальним вікном створення завдання, де заповнює необхідні атрибути, такі як назва, дата завершення та рівень пріоритетності. Після локальної валідації введених даних система ініціює асинхронний запит до хмарної бази даних Firestore. У момент успішного створення документа системна логіка за замовчуванням присвоює новому запису первинний статус активного. Саме цей статус виступає тригером для алгоритмів відображення, дозволяючи завданню з'явитися на головній панелі інструментів користувача.

Перебуваючи в активній фазі, інформаційний об'єкт стає частиною складніших процесів. Він піддається багатокритеріальному сортуванню, що визначає його позицію на екрані залежно від важливості та терміновості. На цьому ж етапі користувач отримує можливість взаємодіяти із зовнішнім програмним інтерфейсом генеративного штучного інтелекту для декомпозиції академічної цілі на дрібніші структурні кроки. Отримана від нейромережі інформація також інтегрується в об'єкт завдання і зберігається у хмарному середовищі, розширюючи корисне навантаження документа.

Завершальна стадія життєвого циклу передбачає незворотний перехід завдання до одного з трьох фінальних станів. Залежно від успішності тайм-менеджменту студента, активний процес може бути логічно завершений успішним виконанням, скасований через втрату актуальності або переведений у категорію провалених у разі порушення встановлених термінів здачі. Зміна стану ініціює механізм логічного видалення з активної робочої області.

Незважаючи на зникнення з головного екрана, об'єкт не знищується фізично. Він назавжди зберігається в колекції бази даних, переходячи в стадію архівування. На цьому етапі завдання слугує виключно як джерело історичних даних для компонента статистики, дозволяючи користувачеві ретроспективно аналізувати свою академічну успішність та вдосконалювати навички планування (рис. 4.8).

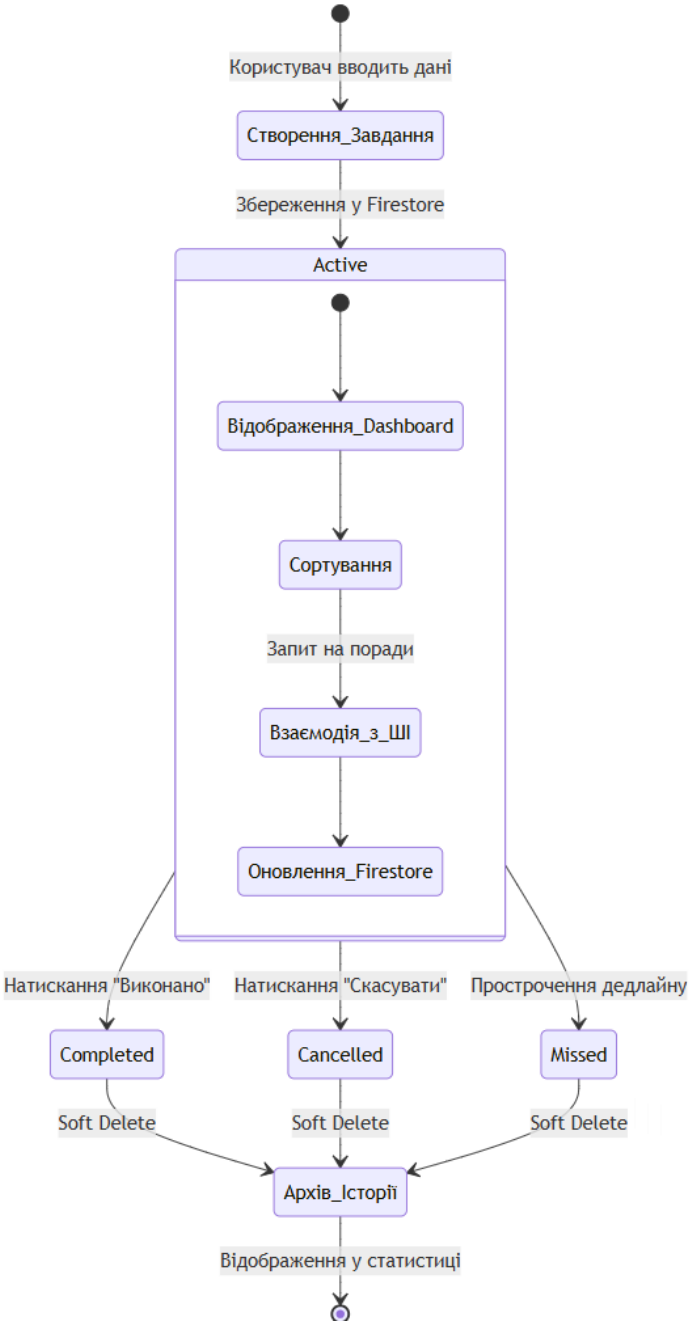

Рисунок 4.8 Схема життєвого циклу завдання в системі

4.4.2 Тестування взаємодії з API та хмарною інфраструктурою

Специфіка етапу налагодження розробленого застосунку зумовлена використанням безсерверної архітектури та інтеграцією з готовими хмарними рішеннями. Це дозволило зосередити увагу на перевірці логіки взаємодії клієнтської частини з віддаленими сервісами.

Для первинного тестування взаємодії з API Google Gemini було використано інструментарій Postman (рис. 4.9). Перед тим як імплементувати виклики до нейромережі безпосередньо в код на JavaScript, запити формувалися у середовищі Postman. Це дозволило ізольовано протестувати HTTP-запити, підібрати правильну конфігурацію заголовків авторизації та відкалібрувати структуру системного промпту. Тестування підтвердило, що модель стабільно повертає дані у потрібному JSON-форматі (статус-код 200 OK), після чого логіку було перенесено в сервісний шар застосунку [20].

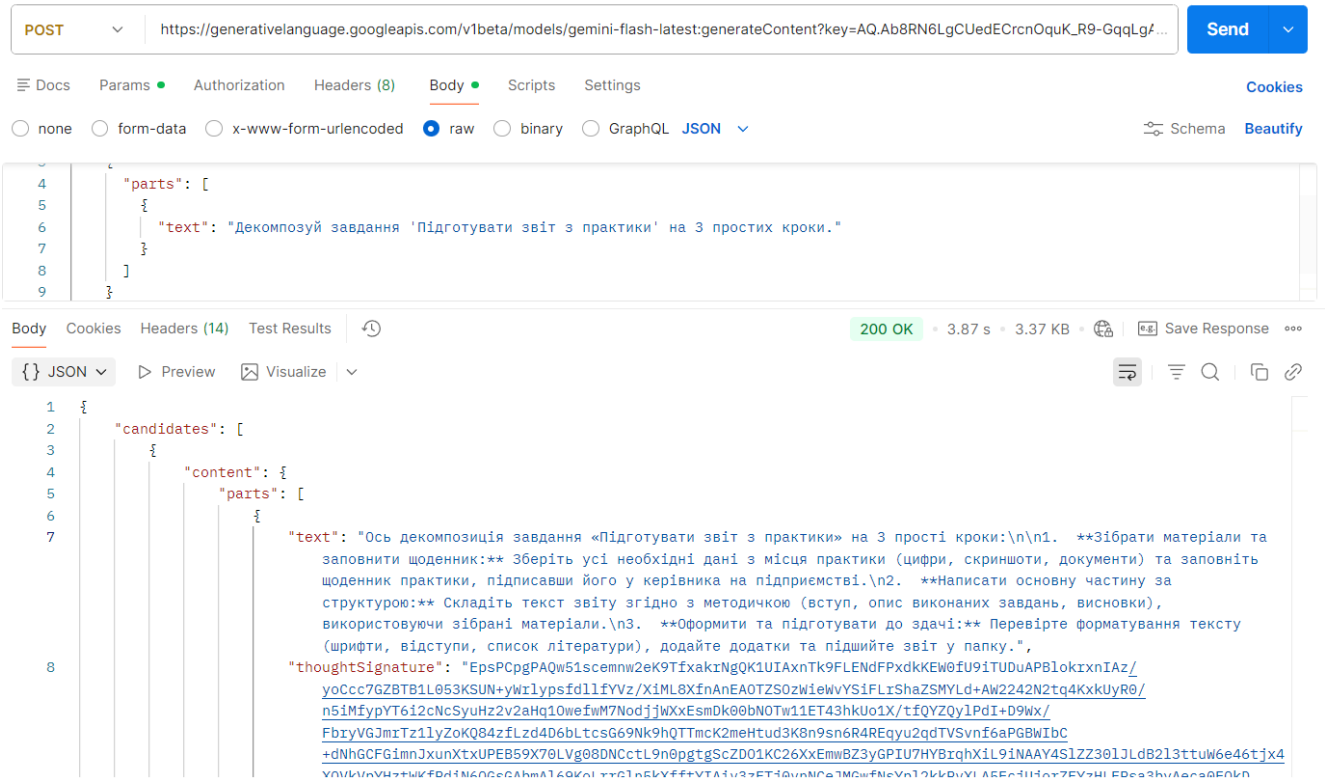

Рисунок 4.9 Тестування запиту до API Google Gemini у середовищі Postman

Тестування взаємодії з базою даних Firestore та управління станами здійснювалося методами ручного тестування за допомогою вбудованих інструментів браузера Chrome DevTools та офіційного розширення React Developer Tools (рис. 4.10). Перевірялися ключові CRUD-операції та передача пропсів (props) між компонентами. Особлива увага приділялася тестуванню підписок у реальному часі. Було успішно підтверджено, що при зміні статусу завдання безпосередньо в

консолі Firebase, клієнтський інтерфейс автоматично та без затримок оновлює відповідні компоненти, відображаючи актуальні дані.

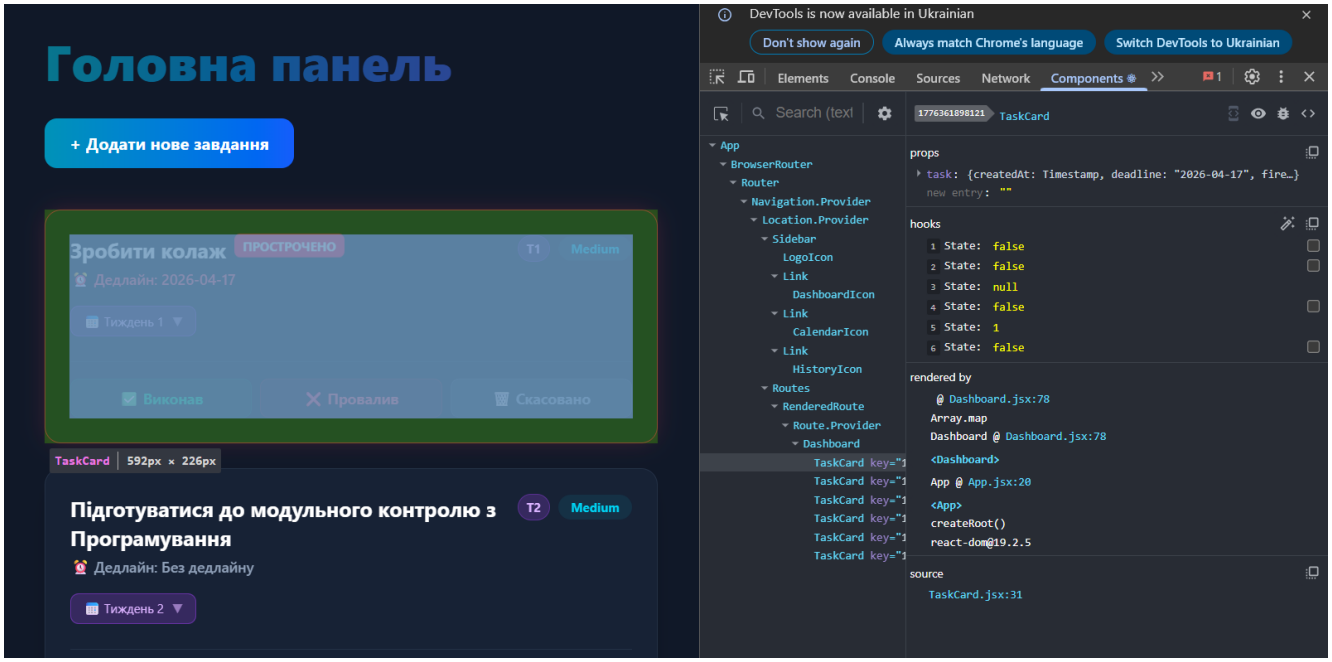

Рисунок 4.10 Перевірка стану компонентів та пропсів за допомогою React Developer Tools

Окремим етапом стало тестування обробки виняткових ситуацій. Було зімітовано умови відсутності інтернет-з'єднання та вичерпання лімітів на кількість запитів до API Gemini. Система успішно перехопила ці помилки за допомогою блоків try...catch та вивела користувачу відповідні інформаційні сповіщення, запобігши критичному збою програми.

ВИСНОВКИ

Розробка спеціалізованого програмного рішення для менеджменту навчальної діяльності сприяє суттєвій оптимізації часових ресурсів студентів та підвищенню їхньої академічної продуктивності в умовах високої інтенсивності освітнього процесу. Під час виконання кваліфікаційної роботи розроблено застосунок для персонального планування навчальних завдань. Створений програмний продукт дозволяє структурувати освітнє навантаження, здійснювати пріоритезацію задач та отримувати персоналізовані рекомендації щодо їх виконання за допомогою інструментів штучного інтелекту.

У процесі дослідження було проведено детальний аналіз типових проблем студентського тайм-менеджменту та визначено ключові функціональні вимоги до системи. Дослідження існуючих комерційних аналогів показало доцільність створення власного, сфокусованого на академічних потребах рішення, яке позбавлене надлишкового корпоративного функціоналу та максимально адаптоване до ритму життя студента.

Практична частина роботи реалізована з використанням сучасного технологічного стеку, що забезпечує гнучкість, надійність та високу швидкість відгуку інтерфейсу. Клієнтську архітектуру побудовано на базі бібліотеки React у поєднанні з сучасними методологіями управління станами. Для реалізації безпечної авторизації користувачів та збереження історії їхньої активності використано хмарну інфраструктуру Firebase, зокрема гнучку документо-орієнтовану базу даних Firestore. Ключовою особливістю розробленого програмного продукту стала успішна інтеграція генеративної моделі Google Gemini, що дозволило впровадити унікальний механізм автоматичної декомпозиції складних цілей на зрозумілі практичні кроки.

Практичне значення отриманих результатів полягає у створенні вебзастосунку, який може безпосередньо використовуватися студентами для організації навчальної діяльності, оперативного контролю дедлайнів, оцінювання власного навантаження та отримання персоналізованих рекомендацій щодо

виконання складних завдань. Розроблена система не лише дозволяє зручно структурувати навчальні плани, але й допомагає знизити рівень прокрастинації завдяки інтелектуальній підтримці користувача.

У результаті виконання роботи було значно поглиблено практичні навички проєктування архітектури інтерактивних вебзастосунків, розробки інтуїтивно зрозумілих користувацьких інтерфейсів та взаємодії з хмарними NoSQL базами даних у режимі реального часу. Важливим інженерним досвідом стало застосування методів промпт-інжинірингу для налаштування передбачуваної взаємодії зі штучним інтелектом, а також реалізація складних клієнтських алгоритмів багатокритеріального сортування та управління життєвим циклом інформаційних об'єктів.

У перспективі функціонал розробленого застосунку може бути суттєво розширений. Доцільним є впровадження системи автоматичних сповіщень про наближення дедлайнів та адаптація клієнтської частини у вигляді повноцінного мобільного застосунку. Важливим вектором технічного розвитку стане двостороння інтеграція з популярними календарями та системою управління навчанням Moodle за допомогою її відкритого програмного інтерфейсу. Це дозволить реалізувати автоматичний імпорт актуальних академічних завдань і розкладу безпосередньо у профіль студента без необхідності ручного введення даних. Разом з цим для покращення якості персоналізованих порад, обробки великих обсягів навчальних матеріалів та зняття лімітів на кількість запитів планується перехід на комерційний рівень доступу до програмного інтерфейсу генеративної моделі Google Gemini. Крім того, додавання можливості спільного виконання групових проєктів перетворить розроблений інструмент на комплексний освітній простір, що сприятиме ще вищому рівню організації навчальної діяльності.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Кові С. Р. 7 звичок надзвичайно ефективних людей. Харків : Клуб сімейного дозвілля, 2020. 384 с.
2. Трейсі Б. Зроби це зараз. 21 чудовий спосіб зробити більше за менший час. Харків : Клуб сімейного дозвілля, 2019. 112 с.
3. Meskhi B., Ponomareva S., Ugnich E. E-learning in higher inclusive education: needs, opportunities and limitations. *International Journal of Educational Management*. 2019. Vol. 33. P. 424–437.
4. Baddeley A., Eysenck M. W., Anderson M. C. Memory. 3rd ed. London : Psychology Press, 2020. 604 p.
5. Норман Д. Дизайн звичних речей. Київ : ArtHuss, 2020. 320 с.
6. Круг С. Не змушуйте мене думати. Про вебюзабіліті. Київ : Фабула, 2021. 256 с.
7. React documentation. *Meta Open Source*. URL: https://react.dev/ (дата звернення: 03.05.2026).
8. Фленаган Д. JavaScript: вичерпне керівництво. 7-ме вид. Київ : Діалектика, 2021. 720 с.
9. MDN Web Docs: JavaScript. *Mozilla Foundation*. URL: https://developer.mozilla.org/uk/docs/Web/JavaScript (дата звернення: 03.05.2026).
10. Sadalage P. J., Fowler M. NoSQL distilled: a brief guide to the emerging world of polyglot persistence. Boston : Addison-Wesley Professional, 2012. 192 p.
11. Firebase Firestore documentation. *Google*. URL: https://firebase.google.com/docs/firestore (дата звернення: 03.05.2026).
12. Google Gemini API documentation. *Google for Developers*. URL: https://ai.google.dev/docs (дата звернення: 03.05.2026).
13. White J. et al. A prompt pattern catalog to enhance prompt engineering with ChatGPT. *arXiv preprint*. 2023. URL: https://arxiv.org/abs/2302.11382 (дата звернення: 03.05.2026).

14. Мартін Р. Чистий код. Створення і рефакторинг за допомогою Agile. Київ : Фабула, 2019. 416 с.
15. Соммервілл І. Інженерія програмного забезпечення. 10-те вид. Київ : Діалектика, 2019. 856 с.
16. Banks A., Porcello E. Learning React: modern patterns for developing React apps. 2nd ed. Sebastopol : O'Reilly Media, 2020. 350 p.
17. React Router Documentation. *Remix Software*. URL: https://reactrouter.com/en/main (дата звернення: 03.05.2026).
18. Tailwind CSS Documentation. *Tailwind Labs*. URL: https://tailwindcss.com/docs (дата звернення: 03.05.2026).
19. Osmani A. Learning JavaScript Design Patterns. 2nd ed. Sebastopol : O'Reilly Media, 2023. 280 p.
20. Postman Learning Center. *Postman*. URL: https://learning.postman.com/docs/ (дата звернення: 03.05.2026).

Додатки

Додаток А

Реалізація головного контейнера Dashboard.jsx

```
import { useState, useEffect } from 'react';
import { TaskCard } from '../components/TaskCard';
import { TaskModal } from '../components/TaskModal';
import { addTaskToDB, subscribeToTasks } from '../services/taskService';

export default function Dashboard() {
  const [tasks, setTasks] = useState([]);
  const [isModalOpen, setIsModalOpen] = useState(false);

  useEffect(() => {
    const unsubscribe = subscribeToTasks((tasksFromDB) => {
      setTasks(tasksFromDB);
    });
    return () => unsubscribe();
  }, []);

  const handleAddTask = async (newTask) => {
    try {
      await addTaskToDB(newTask);
      setIsModalOpen(false);
    } catch (error) {
      console.error("Помилка при збереженні:", error);
      alert("Сталася помилка при збереженні в базу.");
    }
  };
```

```
  return (
    <div className="p-10 w-full relative">

      <h1 className="text-5xl font-extrabold mb-8 text-transparent bg-clip-text bg-gradient-to-r from-cyan-400 via-blue-500 to-indigo-500 animate-pulse">
        Головна панель
      </h1>

      <div className="flex gap-6 mb-10">
        <button
          onClick={() => setIsModalOpen(true)}
          className="
            relative px-6 py-3 font-bold text-white rounded-xl
            bg-gradient-to-r from-cyan-600 to-blue-600
            transition-all duration-300 ease-in-out
            hover:scale-105 hover:shadow-[0_0_20px_rgba(8,145,178,0.5)] hover:from-cyan-500 hover:to-blue-500
            active:scale-95
          "
        >
          + Додати нове завдання
        </button>
      </div>

      <div className="grid grid-cols-1 xl:grid-cols-2 gap-6">
        {tasks.length === 0 ? (
          <p className="text-slate-500">У вас поки немає завдань. Додайте перше!</p>
        ) : (
```

```
          tasks
            .filter(task => task.status === 'active')
            .sort((a, b) => {
              const weights = { 'High': 3, 'Medium': 2, 'Low': 1 };
              const weightDiff = (weights[b.weight] || 0) - (weights[a.weight] || 0);

              if (weightDiff !== 0) return weightDiff;

              if (!a.deadline || a.deadline === 'Без дедлайну') return 1;
              if (!b.deadline || b.deadline === 'Без дедлайну') return -1;
              return new Date(a.deadline) - new Date(b.deadline);
            })
            .map(task => (
              <TaskCard key={task.id} task={task} />
            ))
        )}
      </div>

      {isModalOpen && (
        <TaskModal
          onClose={() => setIsModalOpen(false)}
          onAdd={handleAddTask}
        />
      )}

    </div>
  );
}
```

Фрагмент коду А.1 Реалізація головного контейнера Dashboard.jsx

```jsx
// Складений компонент для відображення масиву завдань (TaskList.jsx)
export const TaskList = ({ tasks }) => {
  return (
    <div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-4">
      {tasks.map(task => (
        <TaskCard key={task.id} task={task} />
      ))}
    </div>
  );
};

// Компонент картки (TaskCard.jsx) з використанням базових UI-елементів
const TaskCard = ({ task }) => {
  return (
    <div className="p-4 bg-white border border-gray-200 rounded-xl shadow-sm">
      <div className="flex justify-between items-start mb-2">
        <h3 className="font-semibold text-lg text-gray-800">{task.title}</h3>
        {/* Базовий UI: StatusBadge */}
        <span className="px-2 py-1 text-xs rounded-md bg-yellow-100 text-yellow-800">
          {task.status}
        </span>
      </div>

      <p className="text-gray-600 text-sm mb-4 line-clamp-2">
        {task.description}
      </p>

      <div className="flex gap-2 mt-auto">
        {/* Базовий UI: ActionButtons */}
        <button className="text-sm bg-green-50 hover:bg-green-100 text-green-700 px-3 py-1.5 rounded transition">
          Виконано
        </button>
        <button className="text-sm bg-purple-50 hover:bg-purple-100 text-purple-700 px-3 py-1.5 rounded font-medium transition">
          ШІ План ✨
        </button>
      </div>
    </div>
  );
};
```

Рисунок А.2 Фрагменти реалізації складених компонентів (TaskList.jsx, TaskCard.jsx)

Додаток Б
Готовий дизайн застосунку

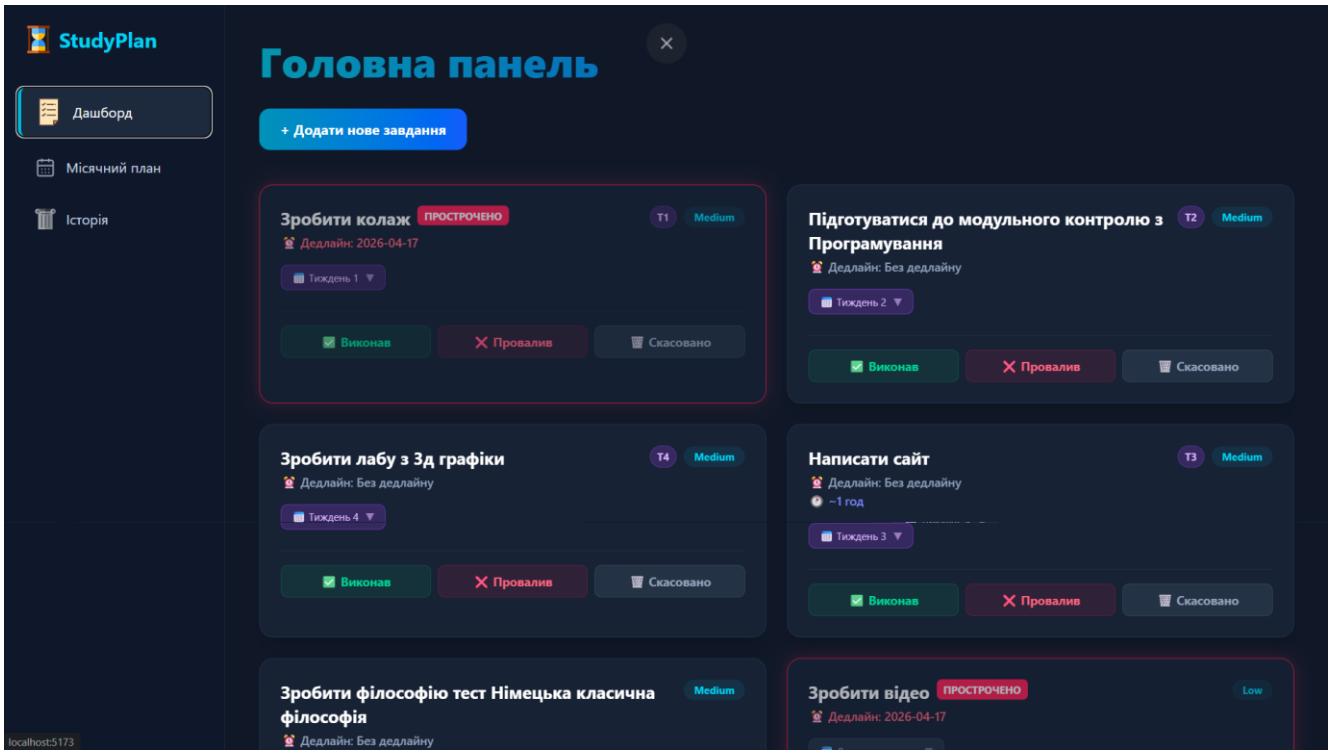

Рисунок Б.1 Головна панель керування (Dashboard) із переліком поточних навчальних завдань

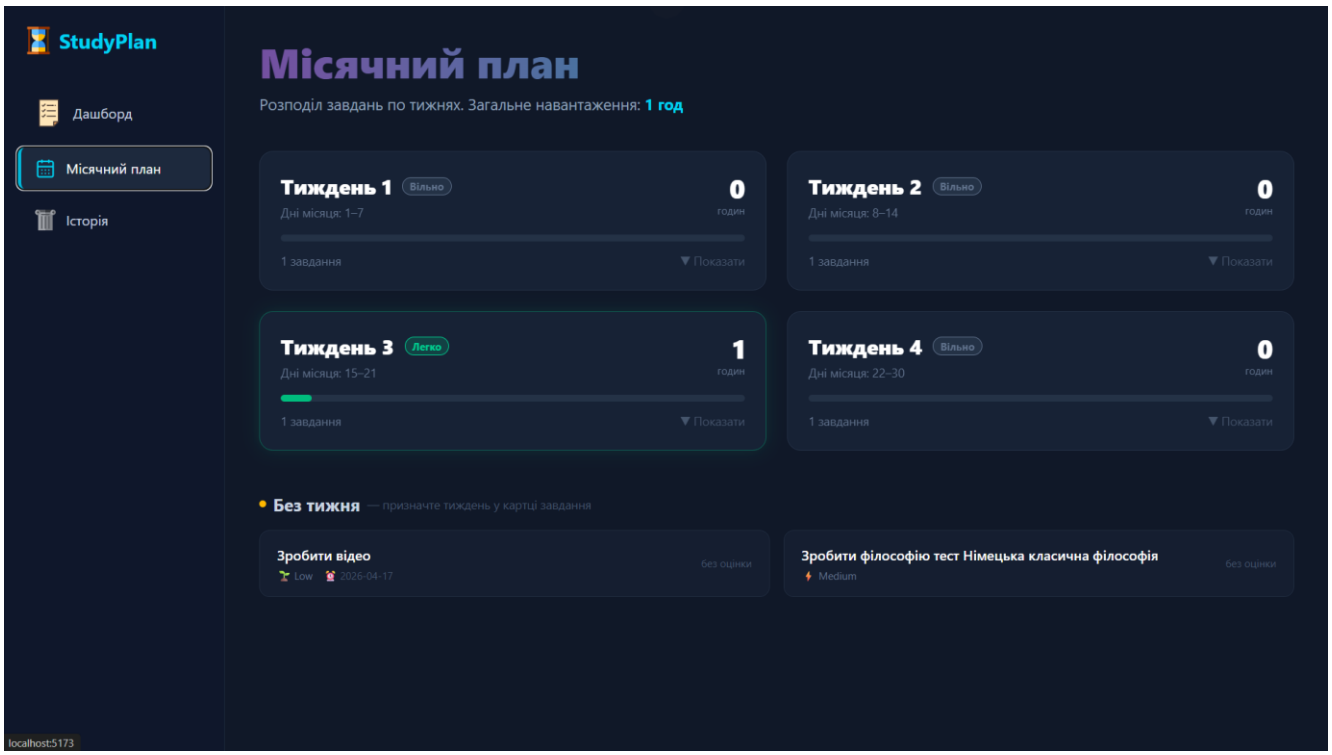

Рисунок Б.2 Візуалізація розділу «Місячний план» для довгострокового планування

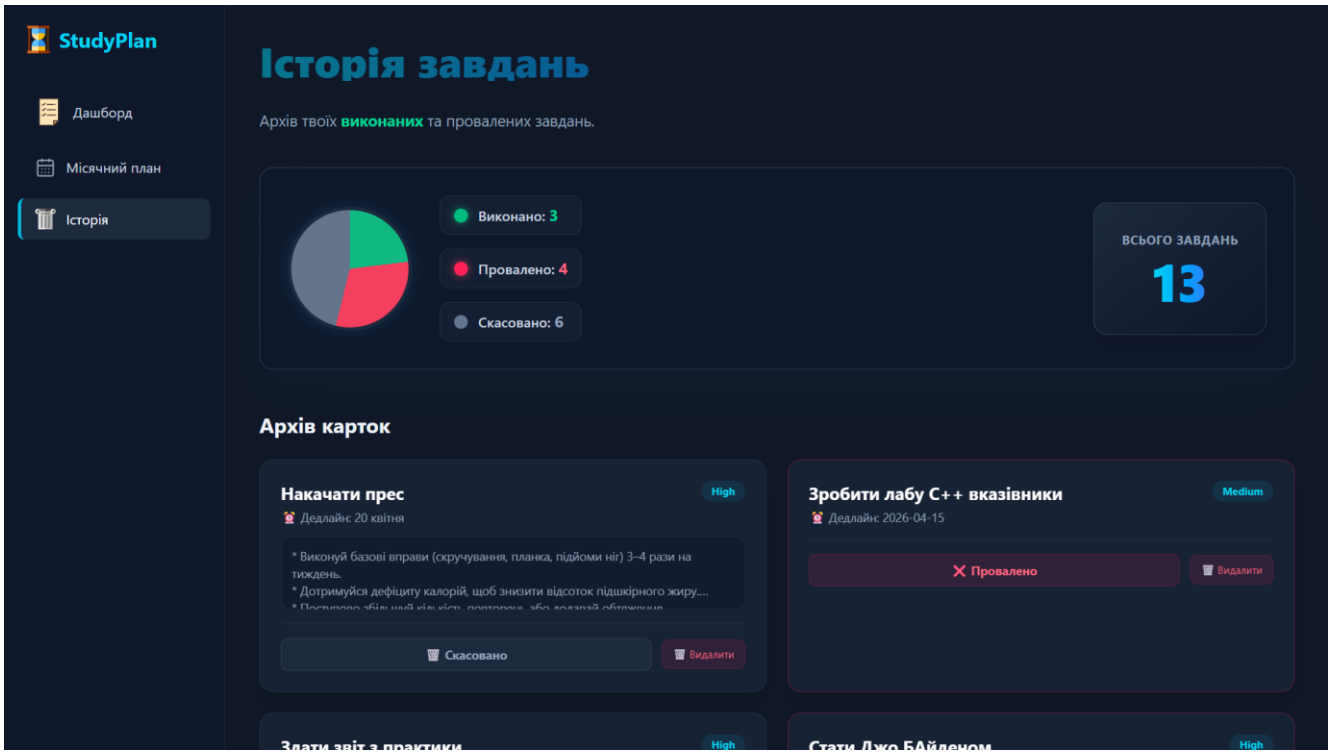

Рисунок Б.3 Відображення розділу «Історія» з архівом виконаних та скасованих цілей