МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Острозька академія»
Навчально науковий інститут ІТ та бізнесу
Кафедра інформаційних технологій та аналітики даних

КВАЛІФІКАЦІЙНА РОБОТА
на здобуття освітнього ступеня магістра

на тему: «***Розробка платформи для менеджменту проєктів***»

Виконав: студент 2 курсу, групи МУП-2
другого (магістерського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
ОПП «Управління проєктами»
Матвійчук Максим Михайлович

Керівник: старший викладач
Клебан Юрій Вікторович

Рецензент: *кандидат технічних наук, доцент, доцент кафедри прикладної математики Донецького національного університету імені Василя Стуса*

Загоруйко Любов Василівна

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ

Завідувач кафедри інформаційних технологій та аналітики даних
_____ (проф., д.е.н. Кривицька О.Р.)

Протокол № 5 від «04» грудня 2025 р.

Острог, 2025

АНОТАЦІЯ
кваліфікаційної роботи
на здобуття освітнього ступеня бакалавра

Тема: *Розробка платформи для менеджменту проєктів*

Автор: *Матвійчук Максим Михайлович*

Науковий керівник: *Клебан Ю. В.., старший викладач*

Захищена «……»…… 2025 року.

Пояснювальна записка до кваліфікаційної роботи: *79 с., 7 рис., 1 табл., 32 джерела.*

Ключові слова: застосунок, клієнт-серверна архітектура, Next.js, Nest.js, інтеграція LLM

Короткий зміст праці:

Ця кваліфікаційна робота присвячена розробці платформи для управління проєктами з елементами інтеграції AI у певний функціонал, задля покращення досвіду користування та зменшення затрат часу на виконання рутинних задач. Описана актуальність теми та проблематика. У першому розділі роботи йдеться про постановку проблеми та архітектурний підхід до розробки. Другий розділ розглядає способи інтеграції AI та потенційні сфери застосунку. У третьому розділі обґрунтовується технічний стек та його використання. Четвертий розділ містить у собі безпосередньо деталі розробки платформи та функціонал. Результат роботи це застосунок та підтвердження позитивного впливу інтеграції нових технологій.

_____

ANNOTATION
of qualification paper
for bachelor's degree

Theme: *Development of a platform for project management*

Author: *Maksym Matviichuk*

Scientific supervisor: *Senior Lecturer Yurii Kleban*

Defended «……»…… of 2025.

Explanatory note to the qualification work: *79 p., 7 pic., 1 table, 32 sources.*

Keywords: *application, client-server architecture, Next.js, Nest.js, LLM integration*

Summary of the paper:

This thesis is devoted to the development of a project management platform with elements of AI integration into specific functionality in order to improve the user experience and reduce the time spent on routine tasks. The relevance of the topic and the issues involved are described. The first chapter of the thesis discusses the problem statement and the architectural approach to development. The second chapter examines ways to integrate AI and potential areas of application. The third chapter justifies the technical stack and its use. The fourth chapter contains the details of the platform development and functionality. The result of the work is an application and confirmation of the positive impact of integrating new technologies.

_____

ЗМІСТ

ВСТУП

Сучасне управління проєктами переживає період стрімкої трансформації, зумовленої зростанням складності бізнес-процесів та появою нових технологічних можливостей. За даними дослідницької компанії Gartner, світовий ринок програмного забезпечення для управління проєктами досяг у 2024 році понад 7 мільярдів доларів, демонструючи щорічне зростання на рівні 10-12%. Водночас організації стикаються з безпрецедентними викликами: збільшенням обсягів даних, необхідністю координації розподілених команд, підвищенням вимог до швидкості прийняття рішень та точності прогнозування результатів.

Революція у сфері штучного інтелекту, яка набрала обертів після появи великих мовних моделей (LLM) та доступних API машинного навчання, відкрила нові горизонти для оптимізації робочих процесів. Технології на основі AI вже трансформували такі галузі як медицина, фінанси, маркетинг та обслуговування клієнтів. Управління проєктами стає наступною сферою, де інтеграція штучного інтелекту може принести значущі практичні результати.

Аналіз поточного стану показує, що менеджери проєктів витрачають від 30% до 40% свого робочого часу на рутинні адміністративні завдання: оновлення статусів, складання звітів, моніторинг прогресу та комунікацію між учасниками команди. Крім того, традиційні системи управління проєктами не надають інструментів для інтелектуального аналізу історичних даних, прогнозування ризиків та автоматичної оптимізації розподілу ресурсів. Це призводить до затримок у виконанні проєктів, перевитрат бюджету та зниження ефективності командної роботи.

Штучний інтелект пропонує розв'язання цих проблем через кілька ключових напрямків. По-перше, технології обробки природної мови (NLP) дозволяють автоматизувати створення та аналіз проєктної документації, виділяти ключові завдання з текстових описів, генерувати звіти та резюме обговорень. По-друге, алгоритми машинного навчання здатні аналізувати історичні дані проєктів для прогнозування термінів виконання завдань,

ідентифікації потенційних ризиків та рекомендацій щодо оптимального розподілу ресурсів. По-третє, інтелектуальні асистенти можуть надавати персоналізовані підказки учасникам команди, автоматизувати рутинні операції та покращувати координацію між відділами.

Мета дипломної роботи полягає у розробці платформи для управління проєктами з інтегрованими AI-можливостями, що забезпечують автоматизацію рутинних операцій, інтелектуальний аналіз даних та підтримку прийняття рішень.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

1. Проаналізувати сучасні технології та фреймворки для розробки вебплатформ управління проєктами, обґрунтувати вибір технічного стека на основі критеріїв продуктивності, масштабованості та підтримки AI-інтеграції.
2. Дослідити доступні рішення у сфері штучного інтелекту (API великих мовних моделей, бібліотеки машинного навчання, сервіси предиктивної аналітики) та оцінити можливості їх інтеграції в контексті управління проєктами.
3. Спроєктувати архітектуру платформи, що забезпечує ефективну взаємодію між основним функціоналом системи та AI-компонентами, з урахуванням вимог до безпеки, продуктивності та користувацького досвіду.
4. Реалізувати специфічні функції, які використовують можливості штучного інтелекту для вдосконалення процесів планування, моніторингу, аналітики та комунікації в рамках управління проєктами.
5. Провести тестування розробленої платформи, оцінити ефективність AI-функціоналу та сформулювати рекомендації щодо подальшого розвитку системи.

РОЗДІЛ 1
ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1. Постановка проблеми

Управління проєктами є однією з ключових управлінських функцій сучасних організацій, оскільки значна частина змін, інновацій та розвитку бізнесу реалізується саме через проєкти. Проєкт у класичному розумінні - це тимчасове підприємство, спрямоване на створення унікального продукту, послуги або результату за обмежених ресурсів, у визначені строки та з чітко окресленими цілями. У практиці IT-компаній та дотичних галузей проєкти часто мають високий ступінь невизначеності, реалізуються розподіленими командами та потребують постійної адаптації до змін вимог замовника та зовнішнього середовища.

Предметна область управління проєктами охоплює сукупність процесів, ролей, артефактів та інструментів, за допомогою яких організація планує, координує та контролює роботу над цими проєктами. У загальному вигляді життєвий цикл проєкту можна подати як послідовність етапів: ініціація, планування, виконання, моніторинг і контроль, завершення. На кожному з цих етапів виникають специфічні задачі, які потребують підтримки з боку інформаційних систем.

На етапі ініціації проєкту формулюється його мета, попередній обсяг робіт, очікувані результати, оцінюються ризики та доцільність реалізації. Сформовані на цьому етапі артефакти - візія продукту, опис бізнес-проблеми, чернеткові вимоги, базова дорожня карта - надалі визначають контекст усього управління. Важливо, щоб ці дані були структурованими, доступними всім ключовим стейкхолдерам та могли еволюціонувати в міру уточнення вимог.

Етап планування передбачає деталізацію цілей у конкретні задачі, визначення обсягу робіт, розподіл ролей та відповідальності, оцінку тривалості задач і ресурсів, формування графіка, бюджету та критеріїв успіху. У сучасній

IT-практиці планування зазвичай реалізується або в рамках гнучких методологій (Scrum, Kanban, їх гібриди), або в рамках більш класичних підходів (waterfall, stage-gate), або ж через гібридні моделі. У будь-якому з цих випадків платформа управління проєктами має підтримувати:

1. розбиття роботи на задачі, підзадачі, епіки та релізи;
2. визначення пріоритетів і залежностей між задачами;
3. планування ітерацій (спринтів), релізів або фаз;
4. оцінку трудомісткості та завантаженості команди;
5. постановку цілей і ключових результатів (OKR) або інших KPI.

На етапі виконання основна увага переноситься на оперативну координацію роботи команди: створення та оновлення задач, фіксація прогресу, комунікація між учасниками, узгодження змін, робота з інцидентами. У цій фазі виникає значний обсяг неструктурованого текстового контенту: коментарі до задач, листування в месенджерах, нотатки з мітингів, обговорення в каналах проєкту, технічна документація. Саме тут особливо гостро постає проблема втрати контексту: важливі рішення та домовленості можуть залишатися лише в чатах або усних розмовах, що ускладнює їхнє відстеження та повторне використання.

Етап моніторингу й контролю передбачає постійне відстеження статусу задач, прогресу щодо плану, показників якості, фінансових метрик, ризиків та змін у вимогах. Менеджер проєкту має відповідати на запитання:

1. чи встигає команда в заплановані строки;
2. які задачі блокують інші;
3. де зосереджені основні ризики;
4. як змінюється завантаженість ресурсів;
5. чи не накопичується технічний борг.

Ця діяльність потребує систематичного збору й аналізу даних, побудови дашбордів, звітів, прогнозів. У багатьох компаніях такі звіти формуються вручну, шляхом експорту даних з різних систем та їхньої подальшої обробки, що є трудомістким та схильним до помилок процесом.

Нарешті, етап завершення включає підбиття підсумків, фіксацію досвіду (lessons learned), оцінку досягнення цілей і показників, передачу продукту в супровід або експлуатацію. Важливо не лише формально закрити задачі, а й зберегти знання, отримані в процесі реалізації проєкту, щоб організація могла використати їх у наступних ініціативах. Це вимагає від системи управління проєктами підтримки механізмів документування, структурованого зберігання та повторного використання інформації.

Важливою характеристикою сучасної предметної області є розподіленість команд і використання різноманітних інструментів. Типова IT-команда може паралельно використовувати таск-менеджери, системи контролю версій, CI/CD-платформи, месенджери, вікі, окремі засоби для тайм-трекінгу та звітності. Дані про один і той самий проєкт розподілені між різними сервісами, не завжди пов’язаними між собою. Це створює низку проблем:

- відсутність єдиного «джерела правди» (single source of truth) щодо статусу проєкту;
- дублювання інформації та розбіжності між різними системами;
- складність у відстеженні повного «ланцюга» від вимоги до реалізації та тестування;
- втрати контексту при переході між інструментами;
- значні витрати часу менеджера на ручну агрегацію даних.

З огляду на це, платформи для управління проєктами прагнуть об’єднати в одному середовищі щонайменше базові функції: управління задачами, планування ітерацій, фіксацію обговорень, побудову звітів та інтеграції з іншими системами (Git, CI/CD, календарі тощо). Проте зі зростанням складності проєктів та обсягу інформації навіть централізована платформа не завжди забезпечує достатню підтримку для прийняття рішень: менеджеру все ще доводиться вручну аналізувати великі масиви тексту, історію змін і сигнали про можливі ризики.

У межах предметної області управління проєктами можна виділити кілька ключових класів задач, які мають безпосереднє відношення до функціональності розроблюваної платформи:

1. Задачі планування та пріоритизації.

   Сюди належать: розбиття високорівневих цілей на задачі, оцінка їхньої складності, визначення пріоритетів, формування спринтів і релізів, балансування між короткостроковими та довгостроковими цілями. У реальній практиці ці задачі ускладнюються невизначеністю вимог, різним рівнем деталізації задач, суперечливими очікуваннями різних стейкхолдерів. Платформа має допомагати структурувати цей процес, надавати зручні інструменти для пріоритизації та забезпечувати видимість наслідків тих чи інших рішень (наприклад, вплив перенесення задач на дедлайни).

2. Задачі оперативного управління роботою команди.

   Це щоденна робота з задачами: створення, оновлення статусів, призначення відповідальних, фіксація перешкод (blockers), зміна пріоритетів, координація між різними ролями. Для цих задач критичними є простота взаємодії з системою, швидкість оновлення даних, підтримка роботи в реальному часі та наочність відображення стану проєкту (дошки, списки, календарі, діаграми). Також важливо, щоб платформа допомагала команді не забувати про важливі, але відкладені задачі, і вчасно реагувати на блокери.

3. Задачі комунікації й узгодження.

   Значна частина рішень у проєкті приймається в ході зустрічей, дзвінків, обговорень у чатах. Виникає потреба зберігати результати цих обговорень у структурованому вигляді: перелік домовленостей, список дій (action items), уточнення вимог, зафіксовані ризики. Якщо система управління проєктами не підтримує зручні механізми фіксації та подальшого пошуку такого контенту, виникають типові проблеми: команда забуває про

домовленості, одна й та сама інформація повторюється в різних каналах, новим учасникам важко «увійти в контекст».

4. Задачі моніторингу, звітності та аналітики.

   Менеджмент і стейкхолдери очікують від системи зрозумілої картини стану проєкту: відсоток виконання, динаміка закриття задач, дотримання дедлайнів, завантаженість учасників, показники якості (кількість дефектів, швидкість реагування на інциденти тощо). У типових інструментах ці показники або надаються у вигляді стандартних дашбордів, або потребують ручного налаштування. Значною проблемою є й те, що кількісні метрики не завжди відображають «якість» прогресу (наприклад, закриття великої кількості дрібних задач може маскувати стагнацію ключового епіка).

5. Задачі управління ризиками та змінами.

   У реальних проєктах постійно виникають нові ризики: затримка з боку замовника, нестача ресурсів, технічні обмеження, зміни законодавства тощо. Управління ризиками передбачає їхнє виявлення, оцінку впливу, розробку планів реагування, моніторинг тригерів. Аналогічно, управління змінами включає аналіз впливу змін вимог на обсяг робіт, строки та бюджет. Більшість популярних інструментів надають лише базові можливості фіксації ризиків і змін, залишаючи їхній аналіз на розсуд менеджера.

6. Задачі накопичення та повторного використання знань.

   Кожен проєкт генерує значний обсяг корисного досвіду: шаблони рішень, типові проблеми та способи їхнього вирішення, контекстні нюанси взаємодії з певними замовниками, внутрішні правила команди. Якщо цей досвід не структурується й не зберігається в єдиному середовищі, організація змушена «винаходити велосипед» у кожному наступному проєкті. Системи управління проєктами можуть відігравати роль «пам’яті організації», але для цього потрібні інструменти не лише для зберігання, а й для швидкого пошуку та узагальнення знань.

Окремо варто відзначити, що предметна область управління проєктами характеризується високою часткою текстових, напівструктурованих та неструктурованих даних. Це описи задач, коментарі, протоколи зустрічей, листування, технічні специфікації. На відміну від суто транзакційних систем, де домінують чітко структуровані записи, тут значна частина корисної інформації прихована саме в текстах. Класичні інструменти проєктного менеджменту здебільшого працюють з метаданими (статус, пріоритет, виконавець, дедлайн) [3], тоді як семантика текстових описів залишається майже невикористаною.

Усе це створює передумови для появи нового покоління платформ управління проєктами, які поєднують традиційні можливості (постановка задач, планування, дашборди) із засобами автоматизованого аналізу текстового та історичного контенту. Зокрема, актуальними стають:

1. інтелектуальний пошук по задачах та документації з урахуванням значення, а не лише ключових слів;
2. автоматичне підсумовування обговорень і протоколів зустрічей;
3. виявлення прихованих зв’язків між задачами та ризиками;
4. прогнозування можливих затримок на основі історичних даних;
5. рекомендації щодо пріоритизації задач з урахуванням контексту.

Таким чином, аналіз предметної області показує, що класичні задачі управління проєктами виходять далеко за межі простого «обліку задач» і потребують систем, здатних працювати з великими обсягами різнорідних даних, у тому числі текстових, забезпечувати прозорість стану проєкту для різних ролей та підтримувати прийняття рішень у умовах невизначеності. Саме в цьому контексті виникає потреба в розробці платформи управління проєктами з інтеграцією засобів штучного інтелекту, яка буде розглянута у подальших розділах роботи.

1.2. Цільові користувачі та сценарії використання платформи

Розроблювана платформа для управління проєктами з інтеграцією засобів штучного інтелекту орієнтована на декілька груп користувачів, які виконують різні ролі в життєвому циклі проєкту. Визначення цих ролей та типових сценаріїв їхньої взаємодії із системою є важливим кроком для формування вимог до функціональності, інтерфейсу та архітектури платформи.

До основних цільових груп користувачів належать:

A. менеджер проєкту (Project Manager, PM);
B. учасники команди (розробники, тестувальники, дизайнери, аналітики тощо);
C. стейкхолдери (представники замовника, керівництво компанії, власники продукту);

   адміністратори системи (відповідальні за підтримку та налаштування платформи).

Кожна з цих груп використовує платформу з різною глибиною залученості та з фокусом на власні задачі, проте всі вони працюють з єдиним інформаційним простором проєкту.

Менеджер проєкту є ключовим користувачем системи з погляду управління. Для нього платформа виступає інструментом планування, моніторингу та прийняття рішень. Типові сценарії використання включають:

- формування структури проєкту (проєкти, епіки, задачі, спринти, релізи);
- визначення пріоритетів та розподіл задач між учасниками команди;
- перегляд оглядових дашбордів стану проєкту (прогрес, дедлайни, завантаженість, ризики);

  аналіз результатів спринтів, підготовка звітів для стейкхолдерів;
- роботу з ризиками та змінами.

Інтеграція ШІ для менеджера передбачає сценарії, у яких система виступає «асистентом з аналітики»: пропонує автоматично згенеровані підсумки мітингів, виділяє перелік дій за результатами обговорень, сигналізує про

потенційні затримки на основі історичних даних, рекомендує задачі, які доцільно підняти в пріоритеті [1]. Це дозволяє зменшити обсяг ручної роботи з інформацією та сфокусуватися на прийнятті рішень, а не на їхньому підготовчому етапі.

Учасники команди використовують платформу переважно на операційному рівні. Для них система слугує центральною точкою взаємодії з задачами та контекстом проєкту. Типові сценарії включають:

- перегляд переліку власних задач, спринтів та пріоритетів;
- створення та уточнення описів задач, додавання коментарів, вкладень, технічних деталей;
- фіксацію прогресу (оновлення статусів, оцінок, позначення блокерів);
- уточнення вимог через обговорення в рамках задачі;
- пошук інформації про попередні рішення, приклади реалізацій, пов’язані задачі.

У межах цих сценаріїв ШІ може виконувати роль «контекстного помічника»: допомагати швидко знайти релевантні задачі чи документи за семантичним пошуком, пропонувати узагальнення довгих обговорень, формувати чернетки описів задач на основі коротких нотаток, підказувати можливі технічні ризики чи залежності, які вже зустрічались у попередніх частинах проєкту.

Стейкхолдери, зокрема представники замовника та керівництво, взаємодіють із платформою переважно на оглядовому рівні . Їхні потреби зосереджені навколо прозорості стану робіт та доступу до консолідованої інформації без необхідності занурюватися в операційні деталі. Типові сценарії:

- перегляд високорівневих дашбордів по проєкту або портфелю проєктів;
- ознайомлення з узагальненими звітами про прогрес, ризики, досягнення цілей;
- доступ до підсумків ключових мітингів та прийнятих рішень;

- аналіз виконання зобов’язань за строками та обсягом робіт.

Для цієї групи користувачів AI-компоненти платформи дозволяють отримувати короткі, структуровані аналітичні звіти з великих масивів даних, сформованих у ході роботи команди, а також задавати запити «природною мовою» (наприклад, «які основні ризики на найближчі два спринти?») і отримувати зрозумілі, агреговані відповіді.

Адміністратори системи відповідають за налаштування та підтримку платформи. Їхні сценарії використання включають:

- створення та керування обліковими записами, ролями та правами доступу;
- конфігурацію інтеграцій з зовнішніми сервісами (репозиторії коду, CI/CD, календарі, AI-провайдери тощо);
- контроль за коректністю роботи системи, моніторинг технічних показників;
- управління політиками безпеки та зберігання даних.

Для цієї ролі важливими є інструменти прозорої конфігурації AI-інтеграцій (налаштування ключів, лімітів, політик доступу до моделей та даних), а також можливість відстеження використання AI-функцій з точки зору навантаження та вартості.

Окремої уваги заслуговує узагальнений сценарій взаємодії користувачів із AI-асистентом у межах платформи. На відміну від класичних систем, де всі операції виконуються через статичні інтерфейси (форми, фільтри, звіти), у запропонованій платформі користувачі можуть формулювати запити у вигляді природномовних інструкцій: попросити систему підсумувати обговорення за певний період, знайти задачі, пов’язані з конкретним ризиком, запропонувати порядок виконання задач заданого набору, згенерувати чернетку статус-репорту тощо. Це відкриває додатковий вимір зручності та гнучкості роботи з даними проєкту.

Таким чином, сукупність цільових користувачів та їхніх сценаріїв використання демонструє необхідність створення платформи, яка поєднує:

1. класичний функціонал управління задачами та проєктами;
2. зручні оглядові інтерфейси для різних рівнів відповідальності; гнучкі механізми конфігурації та інтеграцій;
3. а також AI-компоненту, що виступає універсальним помічником для аналізу, узагальнення та пошуку інформації.

Це визначає як загальну концепцію системи, так і вимоги до її технічної реалізації, що будуть розглянуті в наступних підрозділах.

1.3. Вимоги до системи (функціональні та нефункціональні)

З огляду на те що в межах дипломної роботи створюється не промисловий продукт, а дослідницький прототип (MVP) платформи управління проєктами з інтегрованими AI-можливостями, вимоги до системи формулюються зі зміщеним акцентом: не на повноті покриття усіх можливих сценаріїв, а на демонстрації ключових ідей та можливості експериментувати з інтеграцією штучного інтелекту. Частина вимог має концептуальний характер і задає цільовий образ системи, тоді як реалізація охоплює мінімально необхідний обсяг функцій для перевірки основних сценаріїв.

1.3.1. Функціональні вимоги

Функціональні вимоги визначають, що саме платформа повинна «вміти» з точки зору користувача.

Передусім система має підтримувати базові механізми роботи з користувачами. Передбачається, що користувач може зареєструватися в системі та виконати вхід, наприклад, за допомогою електронної пошти та пароля або через інтеграцію з готовим сервісом аутентифікації. Для цілей MVP достатньо простої рольової моделі, у якій хоча б розрізняються менеджер проєкту та учасник команди, а також існує можливість прив’язати користувачів до одного чи

декількох проєктів. Цього достатньо, щоб продемонструвати розмежування доступу та базову безпеку.

Другий блок функціональних вимог стосується управління проєктами та задачами. Користувач повинен мати змогу створювати нові проєкти, змінювати їх основні параметри та за потреби видаляти. Усередині проєкту мають створюватися задачі з ключовими атрибутами: назвою, коротким описом, виконавцем, статусом, пріоритетом, строком виконання. Система повинна дозволяти змінювати статус задачі (наприклад, від «Заплановано» до «В роботі» та «Завершено»), відображати перелік задач певного проєкту й окремо - задачі, призначені конкретному користувачеві. Базові можливості фільтрації та сортування (за статусом, виконавцем, дедлайном) необхідні для того, щоб користувачі могли організовано працювати з беклогом навіть у рамках мінімального прототипу.

Важливим аспектом є підтримка контексту задачі та комунікації навколо неї. Платформа повинна надавати можливість додавати текстові коментарі до задач, зберігати історію ключових змін (хто і коли змінив статус або основні поля), а також відображати цей контекст у зручному для користувача вигляді. Це дозволяє відновити хід прийняття рішень і зрозуміти, чому задача перебуває в тому чи іншому стані.

Для управління проєктами навіть у дослідницькому форматі важливі прості засоби огляду стану. Система має забезпечувати хоча б базовий огляд по кожному проєкту: кількість задач у різних статусах, певний індикатор прогресу, а також уявлення про завантаженість учасників (наприклад, через перегляд переліку їхніх активних задач). Доцільно також передбачити можливість сформувати простий статус-огляд за певний період, наприклад список задач, які були завершені протягом тижня.

Окремий блок функціональних вимог пов’язаний з AI-можливостями, які й становлять дослідницьке ядро роботи. У межах MVP планується реалізувати щонайменше кілька показових сценаріїв. По-перше, це автоматичне формування коротких підсумків для довгих текстів: описів задач, обговорень у

коментарях, нотаток із зустрічей. По-друге, це виділення зі змісту обговорення конкретних дій (action items), які можна надалі перетворити на окремі задачі в системі. По-третє, це інтелектуальний пошук по задачах і коментарях, коли користувач формулює запит природною мовою, а система намагається знайти релевантні об’єкти не лише за ключовими словами, а й за змістом. Нарешті, корисним є сценарій генерування чернеток описів задач на основі коротких підказок, коли користувач задає загальну ідею, а система допомагає її розгорнути.

Усі ці AI-функції в контексті MVP розглядаються як демонстраційні: вони працюють на обмеженому обсязі даних та для обмеженої кількості сценаріїв, однак цього достатньо, щоб дослідити, як ШІ може підсилити процеси управління проєктами та які переваги (і обмеження) він дає користувачам.

1.3.2. Нефункціональні вимоги

Нефункціональні вимоги задають рамки якості роботи системи й відображають специфіку саме дослідницького прототипу, а не готового комерційного продукту.

По-перше, важливою є простота та зрозумілість інтерфейсу. Платформа має бути такою, щоб новий користувач міг без тривалої адаптації створити проєкт, додати задачі, змінити їхні статуси та скористатися основними AI-функціями. Ключові дії (перегляд списку задач, оновлення інформації, виклик AI-асистента) повинні виконуватися в мінімальну кількість кроків. Це важливо не лише з точки зору зручності, а й для коректного проведення подальших експериментів: якщо інтерфейс надто складний, користувачі будуть оцінювати насамперед його, а не власне AI-можливості.

По-друге, архітектура системи має бути прозорою та розширюваною. Логіка роботи з даними проєктів і задач повинна бути відокремлена від логіки інтеграції зі службами штучного інтелекту. Такий поділ дозволяє змінювати або доповнювати AI-компонент (наприклад, підключати інші моделі, змінювати параметри викликів) без суттєвого втручання в основні модулі системи. Бажано, щоб прототип можна було розгорнути в стандартному середовищі (наприклад, у

контейнерах), що полегшує відтворення результатів дослідження на іншому обладнанні чи в іншій організації.

Продуктивність і масштабованість у цьому випадку розглядаються в поміркованому ключі. Система повинна стабільно працювати для невеликої кількості користувачів і проєктів (наприклад, кількох команд із десятками чи сотнями задач), без спеціальної оптимізації під високі навантаження. Час відповіді на основні операції не повинен створювати відчутних затримок у роботі користувача, однак немає вимоги до жорстких SLA, характерних для промислових систем. Важливіше те, щоб обрані технологічні рішення не блокували потенційну можливість масштабування в майбутньому, навіть якщо в межах дипломної роботи це масштабування не реалізується.

Ще одним аспектом є надійність та обробка помилок. Система має коректно реагувати на типові ситуації: некоректні вхідні дані, тимчасову недоступність AI-сервісу, проблеми з валідацією форм. У таких випадках користувач повинен отримувати зрозуміле повідомлення, а основна частина функціоналу (робота з задачами та проєктами) має залишатися доступною. Бажано також вести базове логування помилок та ключових подій, щоб мати можливість проаналізувати роботу прототипу та вдосконалити його в наступних ітераціях.

Питання безпеки та конфіденційності розглядаються на рівні мінімально необхідних вимог. Доступ до даних проєкту має бути обмежений авторизованими користувачами; передача даних між клієнтом і сервером - здійснюватися захищеним каналом. При інтеграції із зовнішніми AI-сервісами необхідно уникати надсилання чутливої інформації, а ключі доступу до таких сервісів мають зберігатися безпечно.

Окрема група нефункціональних вимог пов’язана з дослідницьким характером роботи. Система повинна дозволяти експериментувати з різними варіантами AI-сценаріїв - змінювати промпти, параметри моделей, типи вхідних даних - без повної перебудови архітектури. Результати роботи AI-функцій доцільно представляти так, щоб користувач міг їх оцінити, за потреби відкоригувати й

зафіксувати своє ставлення до якості відповіді. Це створює основу для подальшого аналізу та можливого вдосконалення інтеграції ШІ.

У підсумку вимоги до системи в цій дипломній роботі окреслюють прототип платформи, який, з одного боку, реалізує базові сценарії управління проєктами, а з іншого - дозволяє на практиці дослідити потенціал і обмеження AI-інтеграцій у цій предметній області. Такий баланс між функціональністю та дослідницькою гнучкістю є принциповим для досягнення поставленої мети роботи.

1.4. Архітектурний підхід до побудови платформи

Архітектура розроблюваної платформи базується на класичній клієнт-серверній моделі вебзастосунку: на стороні клієнта працює інтерфейс користувача, реалізований засобами Next.js, на стороні сервера - прикладна логіка та доступ до даних, реалізовані на основі Nest.js. Взаємодія між клієнтом і сервером здійснюється через HTTP з використанням REST-подібного API, де дані передаються у форматі JSON. Такий підхід є типовим для сучасних вебсистем і забезпечує відносну простоту реалізації, прозорість меж між компонентами та можливість подальшого масштабування.

У загальному вигляді архітектуру можна розглядати як трирівневу. На презентаційному рівні (frontend) Next.js відповідає за відображення інтерфейсу, обробку дій користувача та формування запитів до серверної частини. Клієнтський застосунок виконує роль «тонкого» клієнта в тому сенсі, що він не містить критичної бізнес-логіки, а переважно викликає відповідні API-методи бекенду та відображає отримані результати. При цьому використання Next.js дозволяє комбінувати клієнтський рендеринг з серверним рендерингом окремих сторінок, що покращує початкову швидкість завантаження та зручність роботи.

Прикладний рівень реалізовано у вигляді серверного застосунку на Nest.js. Цей фреймворк пропонує модульну, багатошарову структуру, в якій виділяються контролери (відповідають за обробку HTTP-запитів), сервіси (вміщують

бізнес-логіку) та шари доступу до даних. Завдяки такому поділу відповідальностей досягається краща керованість коду, можливість повторного використання компонентів та спрощення тестування. У межах прототипу передбачається окреме логічне виділення модулів управління користувачами, проєктами й задачами, а також модуля інтеграції з AI-сервісами.

На рівні доступу до даних серверний застосунок звертається до реляційної бази даних (наприклад, PostgreSQL) через ORM-або аналогічний інструмент [31]. Такий підхід дозволяє описувати структуру даних на рівні типів і моделей, отримувати від цього додаткові гарантії цілісності та зменшувати кількість ручних SQL-запитів. Для клієнта всі операції з даними (створення, читання, оновлення, видалення задач, проєктів, користувачів) представлені у вигляді чітко визначених REST-ендпойнтів.

Особливу роль відіграє модуль інтеграції зі службами штучного інтелекту. У архітектурному сенсі він розглядається як окремий сервісний шар усередині бекенду, який інкапсулює роботу з зовнішніми AI-API. Клієнтський застосунок не звертається безпосередньо до AI-провайдера: усі запити проходять через сервер, де відбувається формування промптів [5], виклики до моделей, застосування додаткової бізнес-логіки та постобробка результатів. Це дозволяє централізовано керувати політиками безпеки, логуванням, обмеженням використання та спрощує заміну або доповнення моделей у майбутньому.

Комунікація між рівнями побудована за принципом чітко визначених інтерфейсів. Frontend працює лише з документацією REST-API й не має знань про внутрішню реалізацію бекенду. У свою чергу, серверна частина оперує моделями домену (проєкти, задачі, користувачі) та не залежить від деталей відображення у клієнті. Такий поділ відповідає загальновизнаним принципам розділення відповідальностей (Separation of Concerns) та полегшує як еволюцію окремих частин системи, так і подальше перенесення прототипу на інші платформи.

Обраний архітектурний підхід є характерним для більшості сучасних вебзастосунків: клієнт-серверна модель, REST-орієнтований API, модульна серверна логіка, шар доступу до даних та додаткові інтеграційні модулі.

Висновки до розділу 1

У першому розділі дипломної роботи було проведено ґрунтовний аналіз предметної області управління проєктами та обґрунтовано актуальність створення інтелектуальної платформи, що поєднує класичні інструменти проєктного менеджменту з можливостями штучного інтелекту.

Було визначено, що сучасні IT-проєкти характеризуються високим рівнем складності, динамічними вимогами та значними обсягами неструктурованих даних. Проаналізовано ключові етапи життєвого циклу проєкту - від ініціації до завершення - та основні класи задач, які потребують автоматизації: планування, комунікація, моніторинг, аналітика, управління ризиками й накопичення знань. Показано, що традиційні системи управління проєктами не забезпечують достатньої підтримки для роботи з текстовими даними та прийняття рішень у складних, багатофакторних умовах, що створює передумови для впровадження AI-рішень.

Також було визначено цільові групи користувачів платформи - менеджери проєктів, учасники команд, стейкхолдери та адміністратори - і описано типові сценарії їхньої взаємодії з системою. Для кожної ролі наведено приклади того, як AI може підвищити ефективність роботи: від аналітичної підтримки менеджера до автоматизованого підсумовування мітингів і семантичного пошуку задач.

У підрозділі вимог до системи сформульовано функціональні та нефункціональні характеристики платформи. Функціональні вимоги охоплюють створення проєктів і задач, ведення історії змін, додавання коментарів, а також ключові AI-функції - підсумовування текстів, виділення дій, інтелектуальний пошук і генерацію описів задач. Нефункціональні вимоги визначають принципи

зручності інтерфейсу, модульності архітектури, гнучкості у проведенні експериментів із різними AI-сценаріями та базові критерії безпеки.

Завершальний підрозділ описує архітектурний підхід, заснований на клієнт-серверній моделі з використанням Next.js для фронтенду та Nest.js для бекенду. Передбачено чітке розділення логіки між рівнями, модульну структуру, REST-орієнтовану взаємодію та окремий сервіс для інтеграції зі сторонніми AI-провайдерами.

Отже, у першому розділі було сформовано теоретичну, функціональну та архітектурну основу для створення прототипу системи управління проєктами з інтегрованими можливостями штучного інтелекту, що стане предметом подальших досліджень і реалізації в наступних розділах.

РОЗДІЛ 2
ДОСЛІДЖЕННЯ ТА СПОСОБИ ІНТЕГРАЦІЇ ШІ

2.1. Роль штучного інтелекту в управлінні проєктами

Штучний інтелект (ШІ) посідає все більш помітне місце в сучасних інформаційних системах, переходячи від ролі допоміжного інструмента до статусу повноцінного компонента, що впливає на якість управлінських рішень. Управління проєктами, як сфера, у якій постійно поєднуються невизначеність, обмежені ресурси та високі вимоги до координації, є природним полем для застосування AI-технологій. Зростання складності проєктів, розподіленість команд, динамічність вимог та значний обсяг інформації, що генерується у процесі роботи, створюють ситуацію, коли традиційні інструменти планування та контролю дедалі частіше виявляються недостатніми.

Класичні системи управління проєктами зосереджені переважно на фіксації структурованих даних: задач, термінів, ресурсів, статусів, показників виконання. Вони забезпечують базовий рівень прозорості та контрольованості, дозволяючи відстежувати хід робіт, формувати звіти, узгоджувати відповідальність. Однак значна частина релевантної інформації при цьому залишається поза межами формальних структур - це тексти листування, коментарі до задач, протоколи зустрічей, неформальні домовленості, контекстні пояснення причин тих чи інших рішень. Саме в опрацюванні такого роду даних штучний інтелект, особливо в сегменті методів оброблення природної мови (NLP) та генеративних моделей, відкриває нові можливості.

Роль ШІ в управлінні проєктами можна описати через декілька вимірів. По-перше, це автоматизація аналітичних задач, які раніше виконувалися вручну. Йдеться не лише про побудову дашбордів або звітів, а й про узагальнення текстової інформації, виявлення ключових тез у великих масивах даних, формування коротких підсумків обговорень, виокремлення дій (action items), ризиків чи залежностей, що не були явно формалізовані. Така автоматизація не

замінює менеджера чи команду, але зменшує обсяг рутинної роботи та скорочує час від «сирих» даних до інтерпретованих висновків.

По-друге, ШІ виступає як інструмент підтримки прийняття рішень. На основі історичних даних про попередні проєкти, виконані задачі, затримки, зміни в обсязі робіт та інші фактори, моделі машинного навчання можуть допомагати оцінювати ймовірність вчасного завершення задач, прогнозувати завантаженість учасників, виявляти патерни, що передують виникненню проблем. У цьому контексті AI-компонент виконує роль «другого погляду» на дані, пропонуючи менеджеру альтернативні сценарії, попередження або рекомендації. Остаточне рішення залишається за людиною [2], але воно ґрунтується на глибшому аналізі, ніж той, який зазвичай можливо провести вручну в обмежений час.

По-третє, інтеграція ШІ змінює спосіб взаємодії користувача із системою управління проєктами. Якщо традиційні платформи передбачають роботу через форми, меню та фільтри, то AI-доповнені системи дозволяють звертатися до них у більш природній формі, наближеній до звичайної комунікації [7]. Користувач може сформулювати запит у вигляді питання («які задачі наразі є критичними для релізу?», «які основні ризики на найближчий спринт?») або інструкції («підсумуй результати останньої зустрічі по проєкту», «запропонуй порядок виконання оцих задач»), а система відповідає узагальненням, підбіркою задач чи рекомендованими діями. Таким чином, ШІ виступає своєрідним «інтерфейсом поверх інтерфейсу», роблячи доступ до інформації гнучкішим і менш залежним від конкретних екранів чи форм.

Важливою складовою ролі штучного інтелекту в проєктному менеджменті є перехід від суто описових і діагностичних характеристик до прогнозних і рекомендаційних. Якщо раніше система, умовно кажучи, відповідала на питання «що відбувається зараз» і «що вже сталося», то з появою AI-компонент з’являється можливість відповідати на питання «що може статися надалі» та «що доцільно зробити, щоб уникнути негативних сценаріїв». Це проявляється, зокрема, в прогнозуванні строків, оцінці ймовірності

перевищення бюджету, ідентифікації задач, які з високою імовірністю стануть критичними, якщо їх відкласти, а також у формуванні варіантів перерозподілу ресурсів.

Разом з тим роль ШІ не слід абсолютизувати. Моделі, що лежать в основі сучасних AI-систем, навчаються на даних, які можуть бути неповними, зашумленими або не репрезентативними для конкретного проєкту. Вони не мають «розуміння» в людському сенсі цього слова, а оперують статистичними закономірностями. Тому AI-компоненти доцільно розглядати не як автономних агентів, що самостійно приймають рішення, а як інструменти підтримки менеджера та команди. Людина зберігає контроль над ключовими управлінськими рішеннями, використовуючи ШІ як засіб пришвидшення аналізу, розширення огляду та перевірки власних гіпотез.

У цьому контексті особливої ваги набуває концепція «людина-в-циклі» (human-in-the-loop) [30]. Вона передбачає, що результати роботи AI-системи не є остаточними, а підлягають перевірці, коригуванню й доповненню з боку користувача. З одного боку, це дозволяє зменшити ризики помилок, які можуть виникати через неточність моделей або відсутність контексту. З іншого - відкриває можливості для поступового вдосконалення AI-компонента на основі зворотного зв’язку: система може «вчитися» на виправленнях користувачів, коригуванні запропонованих підсумків чи рекомендацій.

Окремо варто відзначити, що впровадження ШІ в управління проєктами має також організаційний та культурний вимір. Використання AI-інструментів змінює спосіб роботи команди, вимоги до компетенцій, очікування щодо прозорості даних та відповідальності за рішення. З одного боку, менеджери отримують доступ до нових інструментів аналізу і планування, з іншого - мають навчитися критично інтерпретувати результати AI-моделей, розуміти їхні обмеження, пояснювати команді можливі ризики й переваги. Таким чином, роль ШІ виходить за рамки суто технічного компонента і стає елементом організаційної трансформації.

Узагальнюючи, можна стверджувати, що штучний інтелект у сфері управління проєктами виконує щонайменше три взаємопов’язані ролі: інструмента автоматизації аналізу, засобу підтримки прийняття рішень та нового рівня взаємодії користувачів із системами проєктного менеджменту. У сукупності ці ролі створюють передумови для зменшення рутинних операцій, підвищення якості управлінських рішень та кращого використання накопичених даних і знань. Подальші підрозділи цього розділу будуть присвячені конкретнішим напрямкам застосування ШІ в проєктному менеджменті та підходам до інтеграції таких можливостей у вебплатформи.

2.2. Потенційні сфери застосування штучного інтелекту в управлінні проєктами

Можливості застосування штучного інтелекту в управлінні проєктами охоплюють широкий спектр завдань, однак їх доцільно розглядати не як набір розрізнених функцій, а як сукупність напрямів, у межах яких AI доповнює вже наявні процеси. Такий підхід дозволяє уникнути прив’язки до конкретних реалізацій і зосередитися на тому, які саме типи діяльності потенційно можуть бути підсилені інтелектуальними методами.

Першим важливим напрямом є опрацювання та узагальнення інформації. У проєктах постійно накопичуються значні обсяги текстових даних: описи задач, коментарі, протоколи зустрічей, листування з замовниками, внутрішня документація. Штучний інтелект може виконувати роль «фільтра» та «агрегатора» цієї інформації, допомагаючи виділяти головне, структурувати неформальні записи, формувати короткі огляди стану справ. У цьому контексті AI слугує засобом зменшення когнітивного навантаження на учасників команди, скорочуючи час, необхідний для ознайомлення з контекстом.

Другий напрям пов’язаний із підтримкою прийняття рішень та прогнозуванням. На основі історичних даних про виконання задач, строки, обсяги змін і типові ризики моделі можуть будувати оцінки ймовірності

затримок, перевантаження окремих учасників, виникнення «вузьких місць» у процесі. У теоретичному плані йдеться не стільки про точний прогноз, скільки про формування додаткових сигналів для менеджера, які вказують на потенційно проблемні ділянки проєкту й дозволяють завчасно розглянути альтернативні сценарії планування.

Третім важливим блоком є підтримка комунікації та координації роботи команди. ШІ може виступати посередником між різними каналами взаємодії, узгоджуючи інформацію, що надходить з мітингів, чатів і систем управління задачами. На концептуальному рівні це означає можливість переходу від фрагментованого інформаційного поля до більш цілісної картини [2], де домовленості, прийняті в одному контексті, стають видимими та в інших (наприклад, у вигляді оновлених задач або пояснень до них).

Четвертий напрям стосується автоматизації рутинних операцій та робочих процесів. Йдеться про ті дії, які за своєю природою є повторюваними й шаблонними, але все ще виконуються вручну: створення типових задач, заповнення окремих полів, формування стандартних звітів, підготовка нагадувань. Інтелектуальні системи можуть або повністю брати на себе виконання таких операцій, або виступати у ролі асистента, пропонуючи користувачу попередньо сформовані варіанти, які потребують лише мінімального затвердження чи коригування.

Окремо варто виокремити сферу управління знаннями та навчання організації. Кожен проєкт генерує не лише артефакти у вигляді коду, документів чи звітів, а й досвід: вдалих рішень, помилок, специфічних обмежень. Штучний інтелект потенційно може допомагати виявляти, структурувати й повторно використовувати ці знання, перетворюючи розрізнені фрагменти інформації на узагальнені шаблони, рекомендації або «best practices», доступні для майбутніх проєктів.

Зазначені напрями не вичерпують усіх можливих сценаріїв застосування ШІ в управлінні проєктами, але задають рамку, в межах якої можна описувати як вже наявні інструменти, так і потенційні функції майбутніх систем. У

подальших розділах окремі з цих сфер будуть розглянуті детальніше на прикладі конкретних AI-функцій та їхнього впливу на ефективність проєктного менеджменту.

2.3. Підходи до інтеграції AI у вебплатформи управління проєктами

Окреслені у попередніх підрозділах напрями застосування штучного інтелекту в управлінні проєктами потребують технічної реалізації у вигляді конкретних механізмів інтеграції AI-моделей у програмні системи. На практиці розробник платформи стоїть перед вибором не лише конкретного постачальника моделей, а й архітектурного підходу: де саме розміщувати AI-логіку, яким чином організовувати обмін даними, як поєднувати наявні сервіси з новими інтелектуальними компонентами. У загальному вигляді можна виокремити декілька типових стратегій інтеграції. Перша передбачає використання зовнішніх інтеграційних платформ автоматизації, які беруть на себе значну частину рутинної роботи з оркестрації запитів (наприклад, n8n, Zapier, Make) [4]. Друга ґрунтується на прямій взаємодії бекенд-застосунку з AI-провайдерами через їхні API, що дає більше контролю та гнучкості ціною більшої складності реалізації. Третя, перспективна з огляду на конфіденційність і автономність, пов’язана з розгортанням локальних моделей або використанням гібридних RAG-підходів.

У межах цієї роботи основну увагу приділено першій і другій стратегіям як найбільш релевантним для MVP-платформи. Далі розглядається приклад інтеграції через платформу n8n як представника класу інструментів автоматизації, що можуть виконувати роль проміжної ланки між системою управління проєктами та зовнішніми AI-сервісами.

2.3.1 Підходи до інтеграції AI у вебплатформи управління проєктами

Одним із можливих підходів до інтеграції штучного інтелекту у платформу управління проєктами є використання зовнішніх інструментів автоматизації робочих процесів. Характерним прикладом такого класу рішень є

n8n - платформа з відкритим вихідним кодом, яку часто порівнюють із комерційними сервісами на кшталт Zapier чи Make. На відміну від суто «no-code» рішень, n8n поєднує в собі візуальний редактор сценаріїв (воркфлоу) з можливістю використання власного коду, що робить її придатною не лише для простих інтеграцій, а й для побудови відносно складних мікросервісних ланцюжків оброблення даних.

У контексті платформи управління проєктами n8n може виконувати роль проміжної ланки між бекендом системи та зовнішніми AI-сервісами. Наприклад, сценарій роботи може виглядати так: користувач фіксує усне обговорення у вигляді голосового повідомлення; це повідомлення надходить до сервісу розпізнавання мовлення (Whisper, Google Speech-to-Text чи аналогічного), після чого отриманий текст обробляється мовною моделлю, яка формує структурований опис задачі. Уже у n8n результат цієї обробки може бути приведений до формату, зручного для платформи проєктного менеджменту (наприклад, JSON зі зрозумілими полями) і переданий назад у систему через REST-API.

З погляду архітектури такий підхід дозволяє розділити складну інтеграційну задачу на послідовність логічних кроків. На вході n8n отримує дані з PMS (текст, аудіо, параметри запиту) через вузол типу *Webhook*. Далі один чи кілька вузлів відповідають за взаємодію з AI-сервісами (наприклад, виклик мовної моделі через модуль OpenAI або інший провайдер) [25], після чого проміжні результати можуть додатково трансформуватися, фільтруватися або збагачуватися. На завершальному етапі спеціалізований вузол (*HTTP Request*, *Database* тощо) надсилає уже структуровані дані назад у платформу управління проєктами, де вони зберігаються як нова задача або оновлення наявного запису.
На рис. 1 схематично зображено приклад такого воркфлоу у візуальному редакторі n8n [4]. Ланцюжок вузлів демонструє типовий сценарій: отримання HTTP-запиту з бекенду PMS, передача даних до AI-моделі, оброблення й форматування відповіді, а також зворотне надсилання результату у систему.

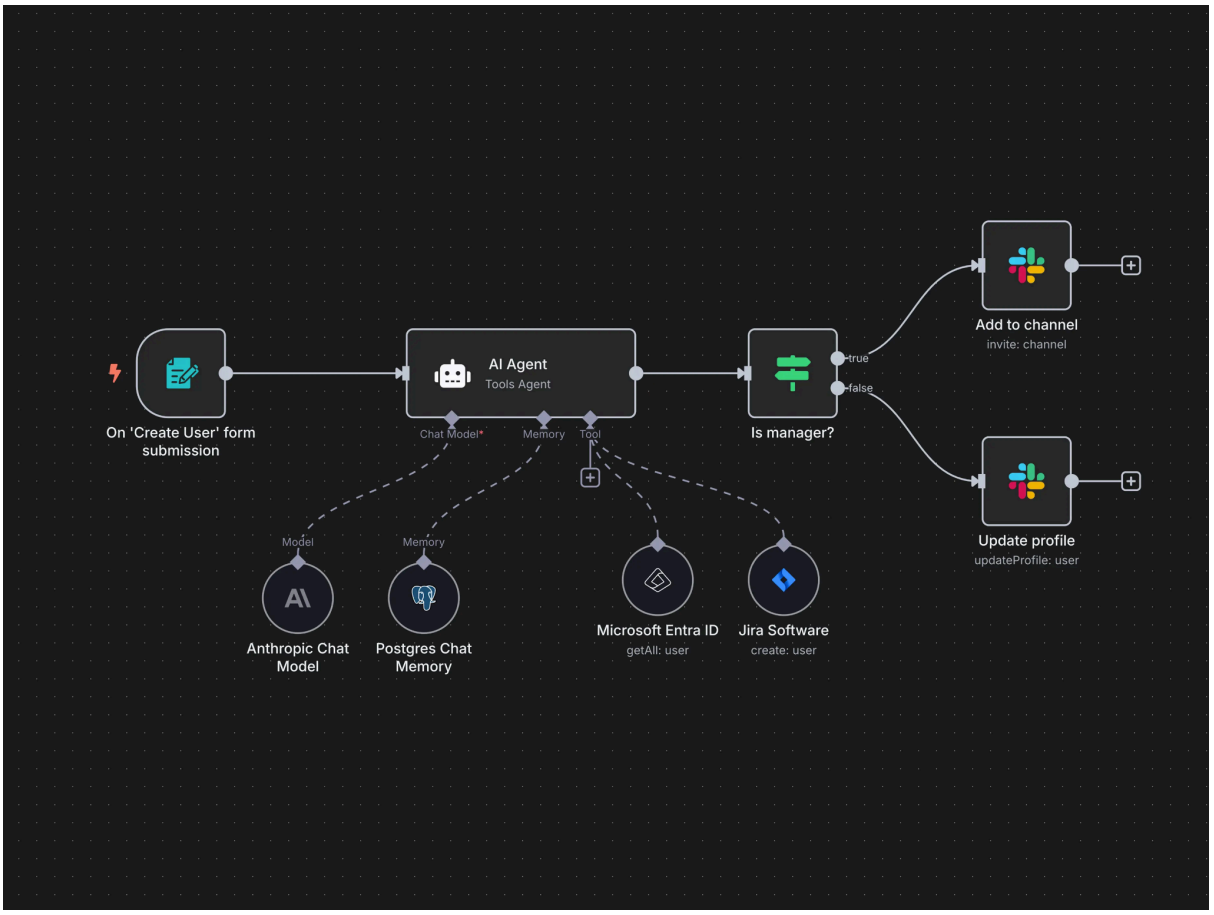

Рис. 2.1. n8n схема

Джерело: документація n8n

Перевагою використання n8n є те, що значну частину інтеграційної логіки можна реалізувати без безпосередньої модифікації основного коду бекенду. Це спрощує експериментування з різними AI-сценаріями: зміна послідовності кроків, заміна моделі, додавання нових умов відбувається на рівні візуального редактора, а не завдяки переписуванню серверного застосунку. Водночас такий підхід має і свої обмеження: з'являється додатковий компонент у системі, який необхідно розгортати, налаштовувати й супроводжувати; також зростає кількість місць, де може виникати затримка або помилка, що потребує належної організації моніторингу та логування [26].

2.3.2. Інтеграція через хмарні сервіси автоматизації

Окрім рішень із відкритим кодом на зразок n8n, у практиці розробки бізнес-застосунків широко використовуються хмарні платформи автоматизації, такі як Zapier чи Make (раніше Integromat). Вони надають користувачам

можливість будувати зв’язки між різними вебсервісами за принципом «тригери → дії», часто без необхідності писати код або з його мінімальним використанням. У більшості випадків інтеграції реалізуються у вигляді зв’язок на кшталт «якщо в системі A сталася подія X - зробити Y у системі B».

У контексті платформи управління проєктами такі сервіси можуть використовуватися для створення допоміжних ланцюжків, що розширюють функціональність основної системи без зміни її коду. Наприклад, за допомогою Zapier можна налаштувати сценарій, за яким створення нової задачі у PMS призводить до автоматичного звернення до AI-сервісу для генерації більш розгорнутого опису або формування початкового переліку підзадач. Аналогічно, зміна статусу задачі може автоматично породжувати коротке резюме прогресу, яке надсилається в корпоративний чат чи електронною поштою відповідним стейкхолдерам.

Сильна сторона таких платформ полягає у великій кількості готових інтеграцій із популярними AI-провайдерами, системами управління задачами, календарями, месенджерами та іншими сервісами. Це дозволяє будувати складні сценарії, комбінуючи кілька інструментів без створення додаткової серверної логіки. Для MVP-рішень або невеликих команд це може бути швидким шляхом до запровадження перших AI-можливостей: достатньо налаштувати кілька «zap’ів» чи сценаріїв без глибокого втручання в архітектуру платформи.

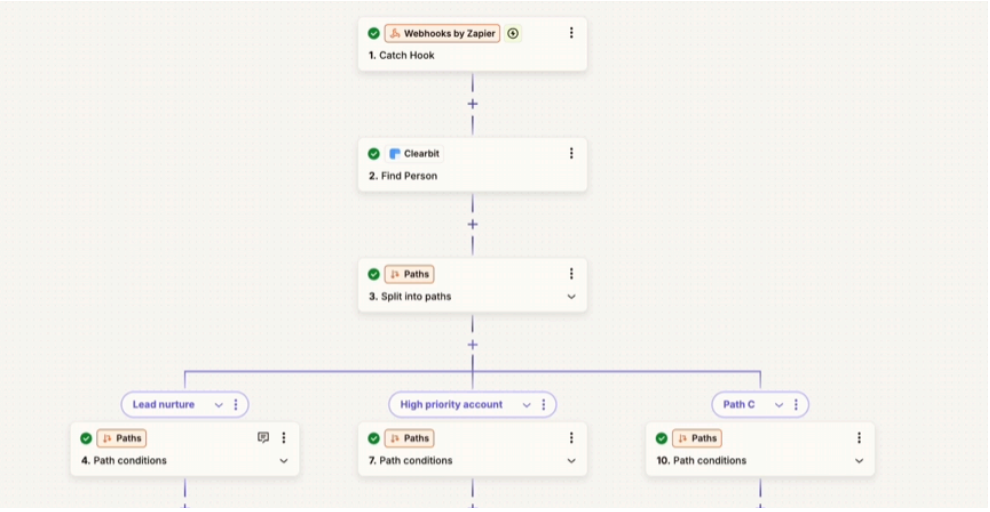

Рис. 2.2. Схема zap

Джерело: документація zapier

Водночас використання хмарних сервісів автоматизації має і низку обмежень. По-перше, значна частина логіки опиняється поза межами основного коду системи, що ускладнює її централізоване документування, тестування і версіонування. По-друге, розробник залежить від моделі ліцензування, обмежень на кількість запитів та затримок, притаманних зовнішній платформі. По-третє, детальний контроль над тим, як саме відбувається оброблення даних, обмежений функціоналом інструмента, що не завжди відповідає потребам дослідницьких або високоспецифічних рішень. Унаслідок цього Zapier та подібні сервіси доцільно розглядати як важливий, але не універсальний варіант інтеграції AI з платформами управління проєктами.

2.3.3. Пряме підключення до AI-сервісів через API

Альтернативою використанню зовнішніх платформ автоматизації є пряме підключення бекенд-застосунку до AI-провайдерів через їхні програмні інтерфейси (API). У цьому підході виклики до мовних моделей, сервісів розпізнавання мовлення чи інших інтелектуальних компонентів здійснюються безпосередньо з серверної частини платформи, наприклад, з модулів Nest.js [25]. Така інтеграція передбачає, що вся логіка формування запитів, оброблення відповідей, обробки помилок та застосування бізнес-правил реалізується у власному коді системи.

Головною перевагою прямої інтеграції є високий рівень контролю. Розробник має можливість гнучко керувати структурою промптів, параметрами моделей, політиками повторних спроб у разі помилок, обсягами та форматом даних, що передаються. Це спрощує реалізацію складніших сценаріїв, де AI-відповідь є лише однією зі складових більшого бізнес-процесу (наприклад, генерація пропозицій по пріоритизації задач із подальшим збереженням їх у базі даних і логуванням рішень користувача). Крім того, пряме підключення дає змогу централізовано реалізувати механізми кешування, обмеження частоти запитів та анонімізації даних.

З архітектурної думки пряме підключення зазвичай оформлюється у вигляді окремого сервісного шару в бекенді. Для платформи управління проєктами це може бути модуль, відповідальний за взаємодію з мовними моделями: він приймає структуровані запити від бізнес-логіки (наприклад, «створити підсумок для цього тексту», «згенерувати перелік задач на основі опису мітингу»), формує відповідні звернення до зовнішнього API [25], обробляє результат і повертає його в уніфікованому для решти системи форматі. Інші підсистеми (модуль задач, модуль звітності тощо) сприймають цей AI-шар як «чорну скриньку» з чітко визначеним інтерфейсом.

Пряме використання API також спрощує проведення досліджень, пов’язаних із порівнянням різних моделей або конфігурацій. Оскільки логіка виклику зосереджена в одному місці, розробник може варіювати типи моделей, параметри генерації чи навіть постачальників (наприклад, змінювати моделі в межах одного AI-провайдера або перемикатися між кількома), не змінюючи загальну архітектуру системи. Це особливо важливо в рамках дипломної роботи, де необхідно оцінити, як різні моделі справляються з тими самими вхідними даними та які наслідки це має для якості підтримки управлінських рішень.

Серед недоліків прямого підходу можна відзначити більшу початкову складність реалізації порівняно з «готовими» інтеграційними платформами. Розробник бере на себе відповідальність за всі аспекти комунікації з AI-сервісом, включно з обробкою помилок мережі, зберіганням ключів доступу, логуванням, тестуванням. Однак у випадках, коли система має дослідницький характер і передбачає гнучке налаштування AI-логіки, саме такий підхід дає найбільшу свободу для експериментування.

2.3.4. Обґрунтування обраного підходу в межах даної роботи

З огляду на дослідницьку спрямованість дипломної роботи та необхідність гнучко експериментувати з різними AI-моделями, параметрами та сценаріями їх використання, у межах даного проєкту обрано саме підхід прямого підключення до AI-сервісів через API з боку серверної частини

(Nest.js). Інструменти автоматизації на кшталт n8n, Zapier чи Make розглядаються як важливі елементи екосистеми інтеграцій, однак у такому випадку вони відіграють радше роль альтернативних варіантів для подальшого розвитку, ніж основної платформи взаємодії з AI.

Такий вибір зумовлений кількома факторами. По-перше, пряме використання API дозволяє зосередити логіку оброблення запитів до моделей у межах коду самої платформи, що спрощує її аналіз, документування та верифікацію в контексті дослідження. По-друге, цей підхід забезпечує можливість точного контролю над тим, які дані передаються в AI-сервіс, у якому вигляді вони повертаються та як саме результати вбудовуються в бізнес-процеси управління проєктами. По-третє, пряме підключення створює технічну основу для проведення порівняльних експериментів: одна й та сама функція платформи (наприклад, створення підсумку) може бути реалізована з використанням різних моделей без зміни решти системи.

У подальших розділах, присвячених опису технологічного стека та потенційних AI-функцій, буде детальніше показано, як обраний підхід інтеграції реалізується з практичного боку та яким чином він використовується для дослідження можливостей штучного інтелекту в управлінні проєктами.

Висновки до розділу 2

У другому розділі було розглянуто роль штучного інтелекту в управлінні проєктами та проаналізовано можливі підходи до його інтеграції у вебплатформи. Показано, що ШІ може виступати не лише як інструмент автоматизації окремих операцій, а як повноцінний компонент системи підтримки прийняття рішень. Зокрема, було окреслено ключові напрями його застосування: узагальнення текстової інформації, підтримка планування та прогнозування, підсилення комунікації й координації в команді, автоматизація рутинних дій та управління знаннями. Таке використання AI дозволяє зменшити когнітивне навантаження на користувачів, прискорити доступ до релевантної інформації та підвищити якість управлінських рішень у проєктному

середовищі. Окрему увагу приділено технічним стратегіям інтеграції AI у платформу: використанню інструментів автоматизації (n8n, Zapier, Make), а також прямому підключенню до моделей через API. Було показано, що інтеграційні платформи дають змогу швидко будувати сценарії оброблення даних і пов’язувати між собою різні сервіси без істотних змін у коді основної системи, однак водночас ускладнюють централізований контроль і тестування логіки. Пряме використання API, навпаки, вимагає більше зусиль на етапі реалізації, але забезпечує вищу гнучкість, прозорість і придатність до досліджень. У межах даної роботи саме цей підхід обрано як основний, оскільки він найкраще відповідає меті - дослідити можливості різних моделей та їх вплив на процеси управління проєктами в рамках єдиної програмної платформи.

РОЗДІЛ 3
ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1. Засоби розробки

WebStorm - це професійне інтегроване середовище розробки (IDE), створене компанією JetBrains для мов JavaScript, TypeScript та фреймворків, таких як React, Next.js чи Node.js. Середовище надає зручну підтримку підсвічування синтаксису, автодоповнення коду, інтеграцію з системами контролю версій, а також потужні інструменти для налагодження (debugging) та тестування застосунків. Завдяки вбудованим інструментам роботи з базами даних, npm-пакетами та середовищами виконання Node.js, WebStorm дозволяє розробнику ефективно працювати без потреби перемикатися між різними програмами. Крім того, IDE має вбудовану підтримку Git, що дає змогу виконувати коміти, переглядати зміни та синхронізувати код безпосередньо з інтерфейсу середовища. Висока продуктивність, інтелектуальний аналіз коду та зручність інтерфейсу роблять WebStorm одним із найпотужніших інструментів для розробки сучасних вебзастосунків.

Git є системою розподіленого контролю версій, яка дозволяє відстежувати зміни у вихідному коді, працювати з різними гілками (branches) та забезпечує можливість командної роботи над одним проєктом. Застосування Git гарантує збереження історії змін, швидке повернення до попередніх версій коду та ефективне розв’язання конфліктів при злитті гілок. Особливістю Git є його розподілена архітектура - кожен розробник має повну копію репозиторію з усією історією змін, що підвищує надійність та автономність роботи. Це дозволяє продовжувати розробку навіть без підключення до мережі, а згодом синхронізувати зміни з центральним сховищем. Такий підхід сприяє стабільному командному розвитку проєкту, а також підвищує безпеку даних.

GitHub виступає як хмарна платформа для зберігання та спільної роботи з репозиторіями Git. Вона надає зручний вебінтерфейс для керування версіями,

перегляду комітів, створення pull request’ів та контролю якості коду через систему рев’ю. GitHub дозволяє організувати командну взаємодію через систему обговорень, управління задачами (Issues) та планування розробки через Projects або Milestones. Крім того, GitHub підтримує інтеграцію з великою кількістю зовнішніх сервісів, зокрема для автоматизації тестування, розгортання чи сповіщень у робочих месенджерах. Таким чином, платформа виконує роль не лише сховища коду, а й повноцінного центру командної співпраці в процесі розробки програмного забезпечення.

Окрім цього, у роботі застосовувалася система автоматизації GitHub Actions, яка дозволяє створювати CI/CD-конвеєри (Continuous Integration / Continuous Deployment). Вона автоматизує процеси тестування, побудови (build) та розгортання застосунку після кожного коміту або pull request’а. Наприклад, при кожному оновленні коду GitHub Actions може автоматично запускати тести, перевіряти відповідність коду вимогам форматування або збирати проєкт для розгортання на сервері. Це значно знижує кількість людських помилок і пришвидшує цикл розробки. Завдяки автоматизації GitHub Actions забезпечує стабільність, передбачуваність та високу якість продукту протягом усього життєвого циклу системи.

3.2. Бібліотеки серверної частини

Для серверної та клієнтської частини, як мову програмування було обрано Typescript. TypeScript - це мова програмування, розроблена компанією Microsoft як надбудова над JavaScript. Вона розширює можливості JavaScript, додаючи статичну типізацію, інтерфейси, модулі та об'єктноорієнтовані конструкції, що робить код більш передбачуваним, структурованим і легшим для підтримки. TypeScript компілюється у стандартний JavaScript, який може виконуватись у будь-якому браузері або середовищі Node.js.

Основна перевага TypeScript полягає в типізації - розробник може вказати типи даних для змінних, функцій та об’єктів. Це дозволяє виявляти помилки ще

на етапі компіляції, до запуску програми, що значно підвищує надійність коду. TypeScript підтримує інтерфейси, класи та узагальнення (generics), що дозволяє будувати масштабовану архітектуру застосунку. Це особливо важливо для великих проєктів, де необхідна чітка структура даних і взаємодія між компонентами.

Інтерфейси допомагають створювати узгоджені структури даних, а узагальнення (<T>) - розробляти універсальні функції або класи, що можуть працювати з будь-якими типами, не втрачаючи безпеки типів. TypeScript також надає низку утилітних типів, які суттєво спрощують роботу з даними. Наприклад, Partial<T> [22] робить усі поля об’єкта необов’язковими, Pick<T, K> вибирає лише певні властивості, а Omit<T, K> - навпаки, виключає непотрібні. Це дозволяє гнучко модифікувати типи, не створюючи нових структур вручну:

Лістинг 3.1. Використання утиліт типів

```
type User = {
  id: string;
  name: string;
  email: string;
  password: string;
};

type PublicUser = Omit<User, "password">;

const user: PublicUser = {
  id: "1",
  name: "Alice",
  email: "alice@example.com",
};
```

У цьому прикладі тип PublicUser автоматично створюється на основі User, але без поля password. Такий підхід широко використовувався у дипломному проєкті для передачі даних між сервером і клієнтом, щоб уникнути витоку конфіденційної інформації.

Крім того, завдяки можливостям типів, таких як Record, Readonly, Pick, Exclude та власним користувацьким type-оголошенням, код стає більш само

документованим. Це значно полегшує роботу в команді, оскільки кожен розробник може швидко зрозуміти, яку структуру мають дані та які поля доступні в тій чи іншій частині системи.

TypeScript підтримує асинхронні операції, деструктуризацію, імпорт модулів, а також усі сучасні можливості ECMAScript [13], що дозволяє легко працювати з API-запитами, базами даних або обробкою даних на сервері:

Лістинг 3.2. Приклад асинхронної функції

```
async function fetchProjects(): Promise<Project[]> {
  const response = await fetch("/api/projects");
  return await response.json();
}
```

На бекенді, у поєднанні з Nest.js, TypeScript забезпечував строгий контроль типів у сервісах, контролерах і DTO (Data Transfer Objects), дозволяючи легко описувати структуру вхідних та вихідних даних. А на фронтенді (у React) він допомагав типізувати props компонентів і хуки, зменшуючи кількість можливих помилок під час рендерингу. Загалом використання TypeScript у цьому проєкті дозволило створити надійний, читабельний і масштабований код як на фронтенді, так і на бекенді. Завдяки чіткій типізації, утилітним типам [22], автоматичній перевірці коду та глибокій інтеграції з IDE (зокрема WebStorm), процес розробки став більш продуктивним і передбачуваним. Це зробило TypeScript ключовим елементом технологічного стека системи управління проєктами з інтеграцією штучного інтелекту.

Nest.js - це сучасний фреймворк для створення серверних застосунків на базі Node.js з використанням TypeScript. Він поєднує у собі простоту Express.js із потужною модульною архітектурою, подібною до Angular, і забезпечує розробникам високий рівень структурованості, масштабованості та підтримуваності коду. Nest.js є універсальним рішенням, що підходить для побудови REST API, GraphQL-сервісів, мікросервісів та систем із використанням WebSocket-з’єднань [23].

В основі Nest.js лежить архітектурна модель, яка поділяє застосунок на окремі, логічно незалежні частини:

1. Модулі (Modules) - основні будівельні блоки застосунку, які групують пов’язані елементи (контролери, сервіси, провайдери) за певною бізнес-логікою. Кожен модуль відповідає за певну функціональність системи, наприклад, ProjectsModule, UsersModule, AuthModule.
2. Контролери (Controllers) - приймають HTTP-запити від клієнта, обробляють їх і повертають відповідь. Вони реалізують логіку маршрутизації та викликають сервіси, не містячи бізнес-логіки безпосередньо.
3. Сервіси (Providers) - містять бізнес-логіку застосунку. Використовуються для взаємодії з базою даних, зовнішніми API або внутрішніми компонентами.
4. Декоратори (Decorators) - є ключовим елементом Nest.js, що дозволяє легко визначати поведінку класів і методів. Наприклад, @Controller(), @Get(), @Post(), @Injectable() додають метадані, які Nest.js використовує для побудови маршрутизації та ін’єкції залежностей.
5. Ін’єкція залежностей (Dependency Injection, DI) - один із центральних принципів фреймворку. Вона дає змогу автоматично передавати необхідні залежності класам через конструктор, що зменшує зв’язність і підвищує гнучкість системи.

Nest.js побудований за принципом модульності, що дозволяє ізолювати функціональні частини системи. Головний модуль AppModule імпортує всі інші модулі, створюючи ієрархію компонентів, яка чітко відображає логічну структуру програми.

Лістинг 3.3. Приклад модуля Nest

```
@Module({
  imports: [ProjectsModule, UsersModule, AuthModule],
})
export class AppModule {}
```

Кожен модуль визначає власні контролери та сервіси. Така архітектура значно спрощує масштабування та повторне використання коду. У дипломному проєкті

подібна структура використовувалася для розділення логіки між управлінням проєктами, задачами користувачів і AI-модулем.

Контролери визначають маршрути HTTP-запитів і описують, як система реагує на ті чи інші виклики [23]. Наприклад, методи з декораторами @Get() або @Post() відповідають за отримання або створення даних відповідно.

Лістинг 3.4. Декоратор для контролера

```
@Controller('projects')
export class ProjectsController {
  constructor(private readonly projectsService: ProjectsService) {}

  @Get()
  findAll() {
    return this.projectsService.findAll();
  }
}
```

Кожен метод контролера є окремою кінцевою точкою (endpoint), що відповідає за певну операцію. Це дозволяє чітко відділити рівень API від бізнес-логіки.

Сервіси - це ядро будь-якого застосунку на Nest.js. Вони містять логіку, пов’язану з обробкою даних, і взаємодіють із базою даних або іншими модулями. Класи-сервіси позначаються декоратором @Injectable(), що дає змогу Nest.js автоматично створювати їхні екземпляри та передавати залежності.

Лістинг 3.5. Injectable сервіс

```
@Injectable()
export class ProjectsService {
  findAll() {
    return [{ id: 1, title: 'AI Integration Project' }];
  }
}
```

У дипломному проєкті сервіси використовувалися для обробки запитів користувачів, створення нових задач, збереження AI-згенерованих задач і взаємодії з ORM-бібліотекою Prisma.

Для передачі інформації між клієнтом і сервером у Nest.js застосовуються Data Transfer Objects (DTO) - спеціальні класи, які описують структуру вхідних

даних. Це дозволяє виконувати типізацію та автоматичну перевірку правильності запитів ще до обробки на сервері.

Наприклад, DTO для створення нового проєкту може виглядати так:

Лістинг 3.6. Приклад DTO

```
export class CreateProjectDto {
  title: string;
  description?: string;
}
```

У поєднанні з бібліотекою class-validator DTO дозволяють перевіряти коректність даних, що надходять із фронтенду, забезпечуючи більшу безпеку та стабільність бекенду.

Щодо взаємодії з базами даних і зовнішніми сервісами.Nest.js має гнучку систему інтеграції з ORM-бібліотеками (наприклад, Prisma, TypeORM, Sequelize) і зовнішніми API. У межах дипломного проєкту Nest.js взаємодіяв із PostgreSQL через Prisma [24], використовуючи типізовані запити для CRUD-операцій. Це дозволило забезпечити повну узгодженість типів між сервером і клієнтом. Крім роботи з базою даних, фреймворк також забезпечував взаємодію зі штучним інтелектом через REST-запити до зовнішнього API, що дозволило інтегрувати AI-функціональність у систему управління проєктами.

Як висновок можна скласти список наступних плюсів:

- Модульність та структурованість. Код легко розділяється на логічні частини, що сприяє масштабуванню системи.
- Типізація через TypeScript. Усі частини застосунку (модулі, DTO, сервіси) суворо типізовані, що мінімізує ризик помилок.
- Ін’єкція залежностей. Забезпечує слабке зв’язування між компонентами та підвищує гнучкість архітектури.
- Вбудована підтримка middleware, pipes, guards та filters. Це дозволяє легко реалізовувати автентифікацію, авторизацію, логування або обробку помилок.

- Універсальність. Nest.js підтримує не лише REST API, але й WebSocket-комунікацію, мікросервіси та черги повідомлень, що робить його придатним для створення комплексних систем.

Таким чином, Nest.js став центральним елементом серверної частини дипломного проєкту. Його архітектура, побудована на принципах модульності, ін’єкції залежностей та строгій типізації, забезпечила гнучкість, надійність і простоту подальшого розвитку системи. Завдяки використанню Nest.js вдалося створити стабільний і безпечний бекенд, який ефективно поєднує бізнес-логіку, роботу з базою даних та інтеграцію з AI-сервісами.

1.3 Prisma ORM

Prisma ORM - це сучасний інструмент для роботи з базами даних у середовищі Node.js, який забезпечує зручний, безпечний і типізований доступ до даних. Prisma не є класичною ORM у традиційному розумінні - вона поєднує переваги автоматизованого відображення об’єктів на таблиці (Object-Relational Mapping) із декларативним підходом до моделювання схеми бази даних [24]. У дипломному проєкті Prisma виконувала роль основного шару доступу до даних у бекенді, розробленому на Nest.js. Вона забезпечувала зручну взаємодію між бізнес-логікою застосунку та базою даних PostgreSQL [31], дозволяючи виконувати операції створення, читання, оновлення та видалення (CRUD) через безпечні, типізовані запити.

Робота з Prisma базується на трьох ключових компонентах:

Prisma Schema - головний файл конфігурації (зазвичай schema.prisma), у якому описуються моделі бази даних, зв’язки між ними, типи полів і використовуваний драйвер (наприклад, PostgreSQL, MySQL чи SQLite). Цей файл виступає єдиним джерелом істини для структури даних.

Prisma Client - автоматично згенерована бібліотека, яка надає розробнику типізований API для роботи з моделями бази даних. Це означає, що всі запити перевіряються ще на етапі компіляції, а отже, ризик помилок під час виконання значно зменшується.

Prisma Migrate - система для управління міграціями бази даних. Вона дозволяє вносити зміни до структури таблиць у контрольований спосіб, синхронізуючи кодову модель із реальною базою даних.

Лістинг 3.7. Модель у Prisma

```
model Project {
  id          Int      @id @default(autoincrement())
  title       String
  description String?
  createdAt   DateTime @default(now())
  tasks       Task[]
}
```

Кожна модель відповідає таблиці в базі даних, а поля - її стовпцям. Зв'язки (relations) між моделями описуються через типи - у такому випадку Project може мати кілька пов'язаних Task. Такий опис дає змогу Prisma автоматично згенерувати запити для отримання проєктів разом із їхніми задачами без додаткової логіки.

Після генерації клієнта Prisma створює об'єкт prisma, через який здійснюється доступ до таблиць. Наприклад, запит для отримання всіх проєктів або створення нового виглядає так:

Лістинг 3.8. Використання Prisma

```
const projects = await prisma.project.findMany();
const newProject = await prisma.project.create({ data: { title: 'AI
Assistant' } });
```

Головною перевагою такого підходу є повна підтримка типів TypeScript. Усі моделі, поля, параметри та повернені значення мають чітко визначені типи, що дозволяє IDE (наприклад, WebStorm) підказувати можливі властивості й перевіряти правильність запитів у реальному часі.

Prisma надає потужні можливості для роботи з реляційними зв'язками - one-to-many, many-to-one, many-to-many. Наприклад, один користувач може мати багато проєктів, або одна задача може належати до певного проєкту. Такі зв'язки задаються декларативно в схемі [24], а Prisma автоматично генерує методи для роботи з ними (наприклад, include, select, connect, disconnect).

Це дозволяє реалізовувати складні вибірки - наприклад, отримати всі задачі певного користувача разом з інформацією про відповідний проєкт - без необхідності писати довгі SQL-запити. У процесі розробки структура бази даних часто змінюється - додаються нові поля, таблиці або зв’язки. Prisma розв'язує цю задачу за допомогою Prisma Migrate - інструменту, який автоматично створює й застосовує міграції. Розробник описує зміни у файлі schema.prisma, після чого виконує команду:

Лістинг 3.9. Команда для створення міграції

```
npx prisma migrate dev --name add-task-model
```

У результаті створюється SQL-скрипт із точними змінами, які буде застосовано до бази. Це забезпечує прозорість і контроль над еволюцією структури даних, що особливо важливо для командної роботи. Prisma ідеально інтегрується з Nest.js, оскільки обидва інструменти базуються на TypeScript і підтримують ін’єкцію залежностей. У дипломному проєкті Prisma було реалізовано як окремий сервіс (PrismaService), який імпортувався у бізнес-логіку відповідних модулів (наприклад, ProjectsService, TasksService). Таким чином, Prisma ORM відіграла ключову роль у серверній частині системи управління проєктами. Вона забезпечила надійний, гнучкий і безпечний доступ до бази даних, дозволила централізовано керувати моделями та міграціями, а також гарантувала узгодженість типів між бекендом і фронтендом. Завдяки Prisma вдалося побудувати ефективний шар взаємодії з даними, який поєднує простоту використання, продуктивність і строгість типізації - критично важливі характеристики для сучасних систем корпоративного рівня.

CASL - це сучасна бібліотека для реалізації системи контролю доступу (access control) у вебзастосунках. Вона дозволяє гнучко визначати, хто і що може робити в системі, використовуючи декларативний підхід до опису прав користувачів. CASL забезпечує централізоване управління дозволами та може бути використана як на бекенді (Nest.js), так і на фронтенді (React), що робить її зручною для побудови узгодженої політики доступу в усьому застосунку.

Основна ідея CASL полягає у створенні об’єкта Ability - набору правил, що описують дії, які користувач може або не може виконувати. Ці правила визначаються на основі ролей, атрибутів користувача або стану даних, і можуть застосовуватися до будь-яких сутностей у системі: користувачів, проєктів, задач тощо.

Основні концепти CASL

1. Ability (можливість) - ядро системи доступу, яке описує дії користувача відносно певного типу об’єкта. Наприклад, “користувач може редагувати власний проєкт” або “адміністратор може видаляти будь-які задачі”.
2. Action (дія) - операція, яку може виконувати користувач (наприклад, create, read, update, delete).
3. Subject (об’єкт доступу) - сутність, над якою виконується дія (наприклад, Project, Task, User).
4. Conditions (умови) - правила, які обмежують дію певними обставинами, наприклад: користувач може редагувати лише ті об’єкти, де він є власником.

Для кожної ролі визначалися різні можливості доступу:

Лістинг 3.10. Приклад створення abilities

```
import { AbilityBuilder, Ability } from '@casl/ability';

export function defineAbilitiesFor(role: string) {
  const { can, cannot, build } = new AbilityBuilder(Ability);

  if (role === 'admin') {
    can('manage', 'all'); // може виконувати будь-які дії
  } else if (role === 'manager') {
    can(['read', 'update'], 'Project');
    can('create', 'Task');
    cannot('delete', 'User');
  } else {
    can('read', 'Project');
    can('update', 'User', { id: 'self' }); // може оновлювати лише
власний профіль
  }
```

```
  return build();
}
```

Ця конфігурація дозволяє централізовано визначати політику доступу й у подальшому перевіряти дозволи у будь-якій частині застосунку - як під час обробки HTTP-запитів на сервері, так і при відображенні інтерфейсу на клієнті.

CASL легко інтегрується з Nest.js через кастомні Guards (механізми контролю доступу). Guard перевіряє, чи має користувач дозвіл на виконання певної дії, перш ніж контролер обробить запит. Це дозволяє реалізувати role-based access control (RBAC) або навіть attribute-based access control (ABAC).

Лістинг 3.11. Використання Ability Guard

```
@UseGuards(AbilitiesGuard)
@CheckAbilities({ action: 'update', subject: 'Project' })
@Put(':id')
updateProject(@Param('id') id: string) {
  return this.projectsService.update(id);
}
```

Переваги використання CASL:

- Єдина політика доступу для всіх частин системи.
- Декларативність. Правила доступу описуються у зрозумілому, форматі.
- Гнучкість.
- Можна комбінувати різні типи контролю: ролі, атрибути користувачів, динамічні умови.
- Безпека. CASL дозволяє централізовано перевіряти дозволи, запобігаючи несанкціонованому доступу.
- Простота інтеграції. Бібліотека сумісна з Nest.js, React, Vue та іншими фреймворками.

CASL забезпечила зручний і безпечний механізм контролю доступу в системі управління проєктами. Її використання дозволило ефективно розмежувати права користувачів, підтримувати єдину політику безпеки між клієнтською та

серверною частинами, а також підвищити загальну надійність і керованість системи.

Для забезпечення безпечної автентифікації та авторизації користувачів у системі було використано технологію JWT (JSON Web Token). Вона є одним із найпоширеніших і найефективніших методів керування доступом у сучасних вебзастосунках, особливо тих, що побудовані на архітектурі REST API. JWT дозволяє передавати дані користувача між клієнтом і сервером у компактному, безпечному та самодостатньому форматі.

JWT складається з трьох частин, розділених крапками:

header.payload.signature

1. Header (заголовок) містить інформацію про тип токена та алгоритм підпису (наприклад, HS256).
2. Payload (тіло токена) містить основні дані користувача - claims, наприклад id, email, role або час дії (exp).
3. Signature (підпис) - криптографічний підпис, який сервер формує за допомогою секретного ключа. Він гарантує, що токен не був змінений після видачі.

Коли користувач успішно проходить автентифікацію (наприклад, вводить правильний логін і пароль), сервер створює JWT і відправляє його клієнту. Надалі клієнт додає цей токен у заголовок кожного запиту (Authorization: Bearer <token>), що дозволяє серверу підтвердити його особу без необхідності постійного зберігання сесій. До основних етапів реалізації можна віднести:

1. Процес входу (Login)

   Користувач надсилає свої облікові дані, після чого сервер перевіряє їх у базі даних. Якщо дані коректні, формується JWT із вказаними даними користувача.

2. Захист маршрутів (Guards)

   Для контролю доступу до приватних ресурсів використовується AuthGuard, який перевіряє наявність і дійсність токена. Якщо токен

прострочений або відсутній, запит блокується до моменту повторного входу користувача.

3. Витяг даних користувача з токена

   Після валідації JWT сервер може отримати ідентифікатор користувача та роль, щоб визначити права доступу. Це зручно поєднується з бібліотекою CASL, яка визначає конкретні дозволи для кожного типу користувача.

4. Термін дії та оновлення

   Токени мають обмежений час життя (expiresIn), після чого користувачу потрібно пройти повторну автентифікацію або використати refresh token для оновлення доступу. Це підвищує безпеку системи та запобігає зловживанням при компрометації токена.

JWT має вигляд:

Лістинг 3.12. Токен jwt

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxIiwidXNlcm5hbWUiOiJtYXh1cyIsInJvbGUiOiJhZG1pbiJ9.
VfQkM2sS1gY6W5bS6bJqWlm4f5W9cE4K0kz1vYj3T4s
```

Декодувавши його, можна побачити зручну структуру JSON:

Лістинг 3.13. Приклад payload

```
{
  "sub": "1",
  "username": "John",
  "role": "admin",
  "iat": 1730123402,
  "exp": 1730127002
}
```

Для реалізації функціональності, пов’язаної зі штучним інтелектом, у серверній частині системи було використано офіційний пакет openai. Він забезпечує зручний інтерфейс для взаємодії з моделями OpenAI (зокрема GPT-4 та GPT-4o) [25], дозволяючи інтегрувати генерацію тексту, автоматичне створення описів, рекомендацій та інших інтелектуальних функцій безпосередньо у бекенд-застосунок. Пакет OpenAI є офіційною бібліотекою для роботи з API

компанії OpenAI. Його головна мета - спростити процес надсилання запитів до моделей і отримання структурованих відповідей.

У дипломному проєкті цей пакет використовувався у модулі, що відповідав за AI-підказки та автоматичну генерацію контенту. Наприклад, система могла пропонувати рекомендації щодо формулювання завдань, створювати короткі описи проєктів або формувати текстові підсумки на основі даних користувача.

Для використання пакета спочатку здійснюється його підключення та ініціалізація клієнта з використанням API-ключа, який зберігається у середовищі змінних (.env):

Лістинг 3.14. Створення instance

```
import OpenAI from "openai";

const openai = new OpenAI({
  apiKey: process.env.OPENAI_API_KEY,
});
```

Такий підхід відповідає вимогам безпеки, оскільки дозволяє не розкривати конфіденційні дані у вихідному коді.

У межах модуля AiService створювався метод, який надсилає запит до моделі GPT-4 і повертає згенеровану відповідь у вигляді рядка.

Лістинг 3.15. Використання пакету

```
import OpenAI from "openai";
const client = new OpenAI();

const response = await client.responses.create({
  model: "gpt-5",
  input: "Write a short bedtime story about a unicorn.",
});
console.log(response.output_text);
```

Асинхронна робота. Усі запити до OpenAI виконуються асинхронно, що не блокує основний потік Nest.js-застосунку. Гнучкість у налаштуванні. Можна змінювати параметри моделі - температуру (temperature), максимальну довжину відповіді (max_tokens) та стиль відповіді через змінні оточення.

Розширюваність. Завдяки модульній архітектурі Nest.js, AI-сервіс можна масштабувати, додаючи нові сценарії використання без зміни базової структури. Безпека даних. Усі ключі та конфігураційні параметри зберігаються у .env-файлі, що унеможливлює несанкціонований доступ до API. Типізований інтерфейс. Завдяки TypeScript, усі відповіді від OpenAI мають визначену структуру, що полегшує обробку результатів у бізнес-логіці.

Для документування та тестування REST API у серверній частині системи управління проєктами було використано Swagger - популярний інструмент для створення інтерактивної документації API. У фреймворку Nest.js Swagger інтегрований офіційно через пакет @nestjs/swagger, що дозволяє автоматично генерувати документацію на основі декораторів у коді.

Swagger забезпечує розробників і тестувальників зручним інтерфейсом для перегляду доступних ендпойнтів, параметрів запитів, типів відповідей і можливих статусів помилок. Це значно спрощує комунікацію між бекендом і фронтендом, а також прискорює процес розробки та налагодження API.

Swagger дозволяє автоматично зчитувати метадані з контролерів, DTO та сервісів, формуючи на їхній основі детальну OpenAPI-специфікацію. Ця специфікація описує структуру всіх маршрутів, методів (GET, POST, PUT, DELETE), типи вхідних і вихідних даних, приклади запитів і можливі коди відповідей. Документація доступна у вигляді вебінтерфейсу, де можна не лише переглянути інформацію, а й виконати реальні запити безпосередньо з браузера. Це робить Swagger надзвичайно зручним інструментом для інтерактивного тестування API.

Інтеграція Swagger у застосунок виконується у файлі main.ts, де після ініціалізації Nest-застосунку створюється Swagger-документація:

Лістинг 3.16. Додавання документації через swagger

```
import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';

const config = new DocumentBuilder()
  .setTitle('Project Management API')
  .setDescription('Документація REST API системи управління проєктами')
```

```
  .setVersion('1.0')
  .addBearerAuth()
  .build();

const document = SwaggerModule.createDocument(app, config);
SwaggerModule.setup('api/docs', app, document);
```

У результаті документація стає доступною за шляхом http://localhost:3000/api/docs, де користувач може переглядати всі маршрути, вводити параметри та отримувати реальні відповіді від сервера.

Swagger у Nest.js базується на декораторах, що додаються до контролерів і DTO. Наприклад:

Лістинг 3.17. Документування endpoint

```
@ApiTags('Projects')
@ApiBearerAuth()
@Controller('projects')
export class ProjectsController {
  @ApiOperation({ summary: 'Отримати список проєктів' })
  @ApiResponse({ status: 200, description: 'Список проєктів успішно отримано' })
  @Get()
  findAll() {
    return this.projectsService.findAll();
  }
}
```

Ці декоратори дозволяють Swagger автоматично включати методи контролера до документації, додаючи опис, статусні коди, типи відповідей та вимоги до авторизації [23]. Swagger став важливим інструментом під час розробки серверної частини, оскільки дозволив швидко перевіряти запити, документувати структуру системи та забезпечити узгодженість між усіма учасниками розробки.

3.3. Бібліотеки клієнтської частини

PNPM (Performant NPM) - це сучасний менеджер пакетів для JavaScript та TypeScript проєктів, який використовується як альтернатива стандартному npm або yarn. Його головна мета - підвищити швидкість встановлення залежностей, зменшити обсяг дискового простору та забезпечити стабільне, детерміноване середовище розробки. На відміну від npm або yarn, PNPM не копіює кожен пакет у папку node_modules кожного проєкту окремо. Замість цього він використовує глобальний кеш (store), де всі завантажені пакети зберігаються лише один раз. Під час встановлення залежностей PNPM не створює дублікати, а створює symbolic links (символічні посилання) на потрібні пакети з кешу. Це суттєво зменшує використання дискового простору, особливо у великих проєктах із численними залежностями або монорепозиторіями.

PNPM створює власний файл блокування - pnpm-lock.yaml, який зберігає точні версії всіх встановлених пакетів. Це гарантує, що всі розробники команди отримують ідентичне середовище незалежно від платформи чи часу встановлення. Файл pnpm-lock.yaml відрізняється від npm’s package-lock.json тим, що він зберігає повну структуру залежностей і використовує оптимізований формат, який прискорює перевірку консистентності.

React - це популярна JavaScript-бібліотека для створення користувацьких інтерфейсів, розроблена компанією Meta. Вона базується на компонентному підході, що дозволяє будувати масштабовані інтерфейси з окремих, незалежних елементів. У дипломному проєкті React використовувався для розробки клієнтської частини системи управління проєктами, забезпечуючи високу продуктивність і зручність взаємодії користувача із системою. React ґрунтується на концепції компонентів, які є ізольованими частинами інтерфейсу, що можуть повторно використовуватись у різних частинах застосунку [9]. Компоненти можуть бути функціональними або класовими, однак у сучасних застосунках переважає функціональний підхід із використанням React Hooks для керування станом і життєвим циклом. Кожен компонент повертає JSX - спеціальний синтаксис, що поєднує JavaScript і HTML, роблячи код більш декларативним і зручним для сприйняття. Наприклад:

Лістинг 3.18. JSX

```
function ProjectCard({ title, description }: { title: string;
description: string }) {
  return (
    <div className="project-card">
      <h3>{title}</h3>
      <p>{description}</p>
    </div>
  );
}
```

JSX дозволяє розробнику інтуїтивно описувати структуру інтерфейсу, а React самостійно оптимізує його оновлення через віртуальний DOM, що значно підвищує швидкість рендерингу.

Для управління станом React використовує Hooks - спеціальні функції, що додають компонентам можливість працювати зі станом або життєвим циклом [9] без написання класів. Основні хуки, які активно застосовувались у проєкті:

- useState() - зберігає локальний стан компонента;
- useEffect() - виконує побічні ефекти (запити до API, підписки, очищення ресурсів);
- useContext() - дозволяє передавати дані між компонентами без "перекидування" пропсів;
- useMemo() та useCallback() - оптимізують повторні обчислення та функції (у React 19 більшість таких оптимізацій уже виконується автоматично компілятором).

Наприклад:

Лістинг 3.19. Використання hooks

```
const [projects, setProjects] = useState<Project[]>([]);

useEffect(() => {
  fetch('/api/projects')
    .then((res) => res.json())
    .then((data) => setProjects(data));
}, []);
```

Однією з ключових технологій React є Virtual DOM - внутрішнє уявлення дерева компонентів, яке оновлюється лише тоді, коли дійсно змінюється стан інтерфейсу. Завдяки цьому React виконує мінімальну кількість реальних змін у DOM, що значно підвищує продуктивність навіть у великих застосунках. Основними причинами вибору React як основної технології стали :

1. Велика екосистема бібліотек (React Query, Zustand, Shadcn/UI, TailwindCSS).
2. Простота інтеграції з бекендом через REST або GraphQL.
3. Актуальна підтримка й розвиток з боку спільноти.

Next.js - це сучасний фреймворк для розробки вебзастосунків на основі React, який надає розширені можливості для рендерингу, маршрутизації та оптимізації продуктивності. Розроблений компанією Vercel, Next.js поєднує гнучкість React із серверними технологіями, забезпечуючи Server-Side Rendering (SSR), Static Site Generation (SSG) та Incremental Static Regeneration (ISR) [16]. Завдяки цьому фреймворк став одним із найпопулярніших рішень для створення масштабованих, високопродуктивних і SEO-дружніх застосунків. У межах дипломного проєкту Next.js використовувався для побудови клієнтської частини системи управління проєктами, інтегрованої з бекендом на Nest.js. Фреймворк забезпечив чітку структуру сторінок, швидке завантаження, можливість часткового рендерингу даних на сервері й високу інтерактивність інтерфейсу.

До основних концепцій можна віднести:

File-based routing (маршрутизація на основі файлової структури).

Next.js автоматично створює маршрути на основі структури папок у директорії app/ або pages/. Це усуває потребу в ручній конфігурації маршрутів і спрощує навігацію між сторінками.

Наприклад, файл app/projects/page.tsx автоматично відповідає за маршрут /projects.

App Router (новий підхід до архітектури).

Починаючи з Next.js 13, фреймворк перейшов до нової системи - App Router, яка замінила стару директорію pages/. Вона ґрунтується на React Server Components (RSC) [12], що дозволяє частину логіки виконувати на сервері без потреби у великих клієнтських бандлах. Це значно покращує продуктивність та час першого завантаження сторінки.

Data fetching (отримання даних). Next.js пропонує кілька способів отримання даних - fetch(), getServerSideProps(), getStaticProps(), getStaticPaths() або Server Actions (у версії 15).

Нові Server Actions дозволяють викликати серверні функції безпосередньо з компонентів, що спрощує роботу з бекендом.

Наприклад:

Лістинг 3.20. Приклад server action

```
"use server";
export async function createProject(data: FormData) {
  await fetch(`${process.env.API_URL}/projects`, {
    method: "POST",
    body: data,
  });
}
```

Layouts і templates. Next.js надає можливість створювати багаторівневі макети (layout.tsx) [16], які повторно використовуються між сторінками. Це дозволяє легко створювати спільні елементи інтерфейсу - наприклад, навігаційну панель, шапку або футер.

Кожна сторінка автоматично “вкладається” у відповідний макет, що забезпечує узгоджений дизайн і логіку по всьому застосунку.

Dynamic routing (динамічні маршрути). Сторінки можуть містити параметри (наприклад, app/projects/[id]/page.tsx), що дає змогу генерувати контент на основі конкретних даних. Це зручно для перегляду деталей проєктів, задач або користувачів.

Next.js дозволяє обирати оптимальний метод рендерингу для кожної сторінки:

1. SSR (Server-Side Rendering): сторінка генерується на сервері під час кожного запиту. Це гарантує актуальні дані, особливо важливо для динамічних сторінок, як-от панель управління проєктами.
2. SSG (Static Site Generation): сторінка створюється під час збірки застосунку і може віддаватися з CDN, що забезпечує максимальну швидкість.
3. ISR (Incremental Static Regeneration): поєднує переваги двох попередніх підходів - сторінки оновлюються періодично без повної перебудови всього сайту.

Такі можливості дозволяють Next.js ефективно балансувати між продуктивністю, актуальністю даних і навантаженням на сервер.

Next.js має вбудовані механізми оптимізації - Image Optimization, Font Optimization, Script Strategy - які зменшують розмір сторінки, скорочують час завантаження і покращують індексацію пошуковими системами.

Також фреймворк надає спеціальні метадані через API metadata або next/head, що дозволяє гнучко керувати SEO-тегами для кожної сторінки (title, description, canonical URL, Open Graph-теги тощо).

Next.js у цьому проєкті виконує роль Frontend Gateway, який безпосередньо взаємодіє з Nest.js через REST API або внутрішні Server Actions. Усі запити до бекенду надсилаються через API-модулі, що інкапсулюють логіку роботи з сервером, забезпечуючи єдиний шар взаємодії.

Tailwind CSS - це сучасний CSS-фреймворк, заснований на утилітарному підході до стилізації інтерфейсів. Його концепція полягає не в заздалегідь визначених компонентах або темах, як у традиційних бібліотеках (Bootstrap, Material UI), а у використанні коротких утилітарних класів, що дозволяють швидко створювати адаптивний, уніфікований дизайн без написання додаткового CSS-коду [15]. Tailwind CSS використовується у поєднанні з React та Next.js, забезпечуючи швидку розробку та високу узгодженість візуального стилю системи управління проєктами. Tailwind надає розробнику великий набір

коротких CSS-класів, кожен з яких відповідає за конкретну властивість - колір, відступ, розмір, тінь, шрифт, вирівнювання тощо. Наприклад:

Лістинг 3.21. Приклад tailwind класів

```
<button className="bg-blue-600 hover:bg-blue-700 text-white
font-semibold py-2 px-4 rounded-lg">
  Add
</button>
```

У цьому прикладі повністю описано зовнішній вигляд кнопки без використання окремого CSS-файлу. Такий підхід дозволяє зберігати дизайн безпосередньо поруч із логікою компонента, що спрощує підтримку та пришвидшує розробку.

Основна ідея Tailwind полягає у принципі “composition over customization” - розробник не створює нові CSS-класи, а складає існуючі утиліти для формування потрібного вигляду. Це зменшує кількість дублювань, уніфікує дизайн і дозволяє швидко вносити зміни без ризику зламати стилі інших елементів. Це створює чисту, передбачувану структуру, яку легко змінювати без пошуку відповідних CSS-правил у файлах стилів. Tailwind має вбудовану систему breakpoints для створення адаптивного дизайну. Достатньо додати префікси (sm:, md:, lg:, xl:) [15], щоб задавати стилі для різних розмірів екрана.

Крім того, Next.js автоматично оптимізує стилі під час збірки, видаляючи невикористані класи (Tree-shaking через PostCSS), що суттєво зменшує розмір фінального CSS-файлу. Таким чином, навіть у великому проєкті з сотнями компонентів Tailwind забезпечує мінімальний обсяг стилів і швидке завантаження сторінок. Tailwind має гнучку екосистему плагінів, що дозволяє додавати готові рішення для типографії, анімацій, форм, скролбарів або кольорових схем.

Наприклад, у проєкті можуть використовуватися плагіни:

1. @tailwindcss/forms - для уніфікованого оформлення форм;
2. @tailwindcss/typography - для покращення стилю текстового контенту;
3. @tailwindcss/aspect-ratio - для керування пропорціями зображень або відео.

Tailwind також підтримує роботу з UI-бібліотеками (наприклад, ShadCN UI або Headless UI), які базуються на його утилітарних класах і надають набір готових доступних компонентів - кнопок, модальних вікон, меню що випадають тощо.

ShadCN UI - це сучасна бібліотека інтерфейсних компонентів, побудована на основі React, Tailwind CSS та Radix UI. Вона поєднує мінімалістичний дизайн, високу кастомізованість і повну інтеграцію з Tailwind, надаючи розробнику готові до використання, але гнучкі компоненти. На відміну від традиційних UI-бібліотек (наприклад, Material UI чи Chakra UI), ShadCN UI не встановлюється як окремий пакет - її компоненти додаються безпосередньо у проєктний код, що робить їх повністю контрольованими розробником. Основна ідея бібліотеки полягає у принципі “copy and own” - тобто замість того, щоб використовувати заздалегідь скомпільовану бібліотеку, розробник копіює вихідний код компонентів до свого проєкту. Це дає змогу:

- повністю змінювати зовнішній вигляд компонентів під власний стиль;
- оптимізувати продуктивність шляхом видалення непотрібних залежностей;
- забезпечити повну сумісність із власною системою дизайну.

Усі компоненти ShadCN UI побудовані на Tailwind CSS, тому легко поєднуються з утилітарними класами, створеними вручну. Водночас для складних елементів (модальні вікна, діалоги, селекти, меню тощо) використовується Radix UI - набір низькорівневих, доступних (accessible) компонентів, що відповідають стандартам WAI-ARIA. Компоненти ShadCN UI організовані у папці components/ui/, що забезпечує зручну структуру для реюзу в усьому проєкті. Кожен компонент є незалежним і може бути налаштований або розширений під конкретні потреби. Інтеграція з Tailwind відбувається природно - усі класи стилів застосовуються безпосередньо через JSX. Крім того, ShadCN UI підтримує темізацію, дозволяючи швидко перемикатися між світлою та темною темами за допомогою CSS-змінних (--foreground, --background, --primary тощо). Наприклад, система кольорів у проєкті може

базуватись на визначених у Tailwind токенах і розповсюджуватись на всі компоненти ShadCN UI через глобальні стилі.

React Query - це потужна бібліотека для управління серверним станом у React-застосунках. Вона спрощує роботу з даними, які отримуються з API, забезпечує кешування, автоматичне оновлення, синхронізацію та оптимістичне оновлення інтерфейсу. Основна ідея React Query полягає в тому, щоб зняти з розробника необхідність вручну реалізовувати логіку запитів, обробку помилок і зберігання даних - усе це відбувається автоматично через централізований Query Client [27]. Традиційно в React- або Next.js-застосунках розробник виконує запити через fetch або axios, зберігаючи отримані дані у стані (useState або useReducer). Такий підхід призводить до дублювання коду, проблем з оновленням даних і відсутності централізованого кешу. React Query розв'язує ці проблеми, впроваджуючи "data synchronization layer" - шар синхронізації даних, який автоматично керує отриманням, зберіганням і повторним використанням результатів запитів.

Основні поняття бібліотеки:

1. Query - запит до API, який може кешуватись і повторно використовуватись.
2. Mutation - зміна даних на сервері (створення, оновлення або видалення).
3. QueryClient - головний об’єкт, що зберігає усі активні запити, їхній стан та кеш.
4. Invalidate Queries - механізм оновлення даних у кеші після змін.

Чому React query була обраною:

- Автоматичне кешування.

  Усі дані зберігаються у кеші, і якщо користувач повторно відкриє сторінку, бібліотека миттєво відобразить кешовану інформацію без повторного запиту до сервера.

- Автоматичне оновлення даних (Background Refetching).

  React Query періодично перевіряє актуальність кешованих даних і

автоматично оновлює їх у фоновому режимі. Це забезпечує постійно актуальний стан без ручного втручання [27].

- Оптимістичне оновлення (Optimistic Updates).

  Під час відправлення mutation (наприклад, створення або редагування задачі) React Query може миттєво оновити інтерфейс без очікування відповіді від сервера. Якщо сервер повертає помилку, зміни автоматично скасовуються.

- Інтеграція з React Suspense й Error Boundaries.

  Завдяки підтримці React 19 бібліотека працює з Suspense API, що дозволяє реалізувати більш гнучке завантаження компонентів і відображення помилок без додаткового коду.

- Підтримка автоматичного повтору (Retries).

  Якщо запит до сервера не вдався, React Query автоматично повторить його кілька разів із затримкою, що підвищує стійкість програми до мережевих збоїв.

- Invalidate та Refetch механізми.

  Після створення або зміни даних можна invalidate певний кеш, примушуючи бібліотеку оновити дані.

React Hook Form - це популярна бібліотека для керування формами у React-застосунках. Її головна мета - спростити роботу з формами, підвищити продуктивність і зменшити кількість повторюваного коду [14]. На відміну від багатьох альтернатив (Formik, Redux Form), React Hook Form не зберігає стан форми у React state, що дозволяє уникнути зайвих рендерів і підвищити ефективність. React Hook Form побудований на React Hooks і використовує принцип неконтрольованих компонентів. Це означає, що значення полів форми зчитуються безпосередньо з DOM, а не через постійне оновлення стану React. Такий підхід забезпечує мінімальні перерендери та значно підвищує продуктивність навіть у складних формах.

Основні хуки бібліотеки:

1. useForm() - створює об’єкт керування формою (register, handleSubmit, reset, errors тощо);
2. register() - зв’язує елементи форми з внутрішнім механізмом бібліотеки;
3. handleSubmit() - обробляє подання форми та викликає callback із валідованими даними;
4. watch() - дозволяє відстежувати зміни у конкретних полях у режимі реального часу;
5. reset() / setValue() - змінюють або очищують значення полів вручну.

Лістинг 3.22. Обробка форми через useForm

```
 const { register, handleSubmit, formState: { errors } } =
useForm<ProjectForm>();

 const onSubmit = (data: ProjectForm) => {
    console.log("Submitted:", data);
 };
```

React Hook Form має вбудовану підтримку валідації полів, яка виконується як синхронно, так і асинхронно [14]. Можна задавати обмеження безпосередньо в register() або використовувати інтеграцію з бібліотеками схем валідації, такими як Yup або Zod.

Zod - це сучасна бібліотека для перевірки (валідації) та типізації даних у TypeScript-застосунках. Вона дозволяє створювати гнучкі схеми даних, які одночасно виконують роль валідатора і типового визначення, забезпечуючи узгодженість між клієнтською та серверною частинами застосунку. На відміну від класичних рішень (як-от Yup), Zod тісно інтегрується з TypeScript, що дозволяє автоматично виводити типи (infer) на основі описаної схеми без дублювання коду. Це робить Zod ідеальним інструментом для проектів, де важлива сувора типізація та валідація даних, зокрема у поєднанні з React Hook Form [18].

Лістинг 3.23. Zod схема

```
import { z } from "zod";

const ProjectSchema = z.object({
```

```
  title: z.string().min(3, "Назва має містити мінімум 3 символи"),
  description: z.string().optional(),
  priority: z.enum(["low", "medium", "high"]),
});

type ProjectForm = z.infer<typeof ProjectSchema>;
```

У поєднанні з React Hook Form застосовується спеціальний zodResolver, що дозволяє виконувати валідацію безпосередньо під час сабміту форми [14].

Висновки до розділу 3

У цьому розділу було описано основні інструменти для роботи з кодом та його зберігання для колаборaційної роботи. Також обрано основні бібліотеки на фреймворки для серверної та клієнтської частин, й показане їх використання.

РОЗДІЛ 4
РОЗРОБКА ТА ВПРОВАДЖЕННЯ

4.1. Засоби розробки

Розроблювана платформа для управління проєктами має MVP-характер і передусім реалізує базовий набір функцій, притаманних сучасним системам проєктного менеджменту: реєстрацію та автентифікацію користувачів, створення й супровід проєктів, ведення задач із основними атрибутами, а також огляд поточного стану робіт. Така основа є необхідною передумовою для впровадження інтелектуальних функцій, оскільки саме на ці дані спираються AI-моделі при аналізі, узагальненні та формуванні рекомендацій.

У межах подальших підрозділів розділу розглядаються приклади можливих AI-інтеграцій, які доповнюють наявний функціонал платформи. Частина запропонованих можливостей може бути реалізована в рамках прототипу, інші розглядаються як перспективні напрями розвитку. Метою є не стільки створення вичерпного набору фіч, скільки демонстрація того, як саме штучний інтелект може підсилити окремі елементи процесу управління проєктами.

Як раніше була зазначено в основі застосунку лежить Next та Nest фреймворки відповідно. Проєкт побудований за класичною схемою frontend + backend + database [32]. Почнемо з першого кроку загального user-flow - тобто вхід у систему. Тут використовується auth модуль. Модуль auth реалізує JWT-автентифікацію [23] на базі NestJS Passport. Токен формується під час реєстрації/логіну і містить sub (ID користувача), email, role. Перевірка токена відбувається глобально через guard. Публічні маршрути позначаються декоратором @AllowAnonymous(). Актуальний користувач доступний через декоратор @CurrentUser(). Далі працює service який відповідає за логіку реєстрації/логіну, хешування паролів, формування JWT із роллю. Відповідно як і кожен модуль, будь-який input запиту проходить через class-validator. Це бібліотека TypeScript/JavaScript, що забезпечує декларативну валідацію об'єктів

за допомогою декораторів. Дозволяє визначати правила перевірки даних безпосередньо у класах, використовуючи анотації на основі стандарту validator.js. Також для відповіді у деяких запитах використовується DTO (Data Transfer Object) - патерн проєктування, що представляє об'єкт для передачі даних між підсистемами або шарами додатка. DTO інкапсулює дані без бізнес-логіки, слугуючи контрактом для структури даних і часто поєднується з валідацією для забезпечення цілісності вхідних/вихідних даних. У комбінації class-validator з DTO створюють type-safe систему валідації, де класи DTO анотуються декораторами валідації (@IsString(), @IsEmail(), @Min() тощо), що особливо поширено у застосунках для автоматичної валідації HTTP-запитів.
На клієнтській частині за допомогою react-query запит на відповідний endpoint.

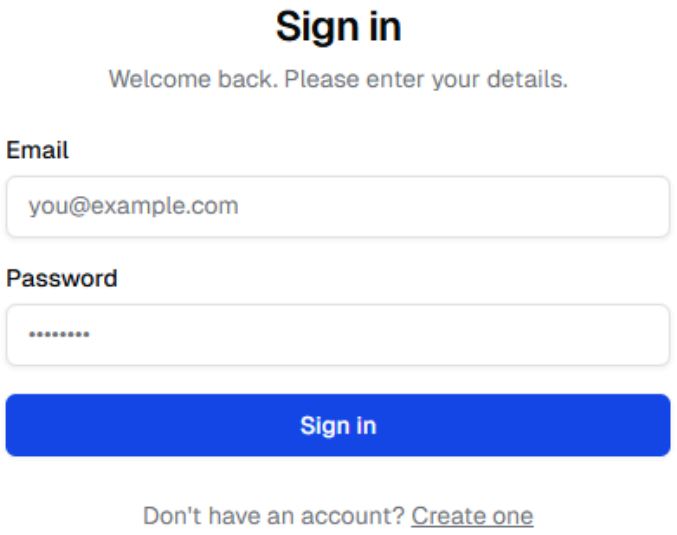

Рис. 4.1. Sign in сторінка

Джерело: створено автором

Також для забезпечення коректної роботи auth-логіки, тобто щоб неавторизований користувач не міг відвідати сторінку де необхідний access-token чи це й же токен вичерпався у валідності було використано routing middleware від Next.js - дозволяє виконувати код до завершення запиту [16]. Потім, на основі вхідного запиту, можна змінити відповідь шляхом перезапису,

перенаправлення, зміни headers запиту або відповіді, або безпосередньо самого response.

Лістинг 4.1. Next middleware

```
export function middleware(req: NextRequest) {
  const { pathname } = req.nextUrl
  if (isPublicPath(pathname)) {
    return NextResponse.next()
  }
  const token = req.cookies.get('accessToken')?.value
  if (!token) {
    const loginUrl = new URL('/login', req.url)
    loginUrl.searchParams.set('redirect', pathname)
    return NextResponse.redirect(loginUrl)
  }
  return NextResponse.next()
}
```

Наступне перейдемо до дашборду - він реалізований за допомогою use-case на отримання статистики які пізніше зарендерена на відповідному маршруті використовуючи page.tsx

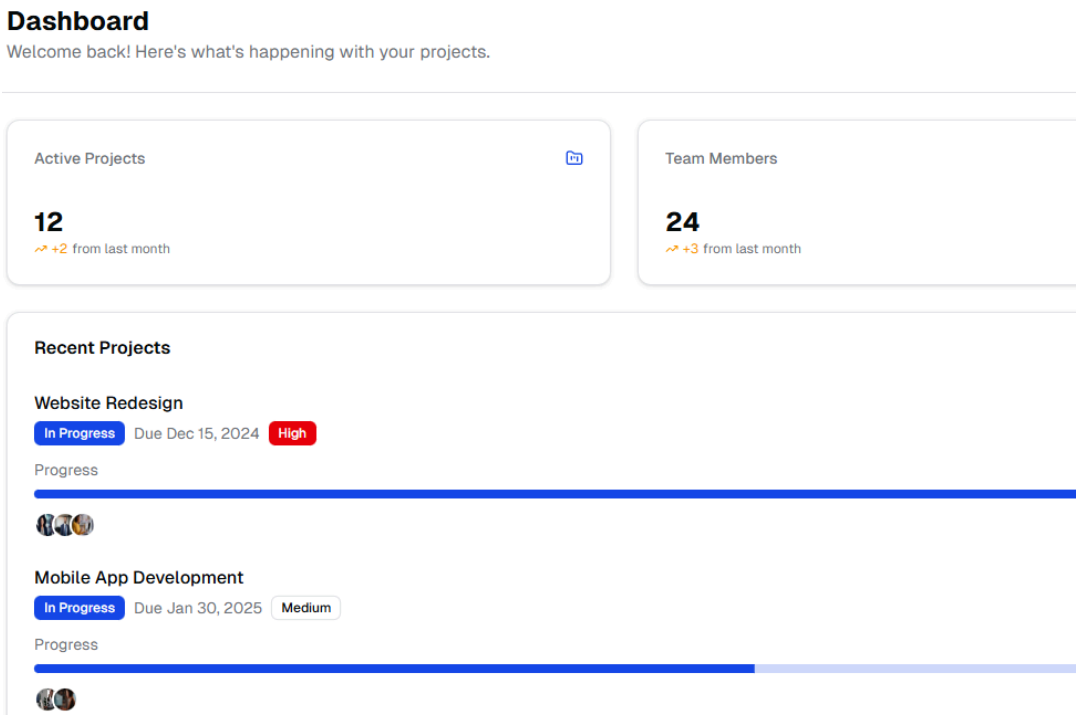

Рис. 4.2. Dashboard

Джерело: створено автором

Наступна основна сторінка це grid-сітка з доступними для авторизованого користувача проєктами. Створена використовуючи shadcn як бібліотеку компонентів. Зі сторони сервера створений відповідний модуль для CRUD операцій по проєктах.

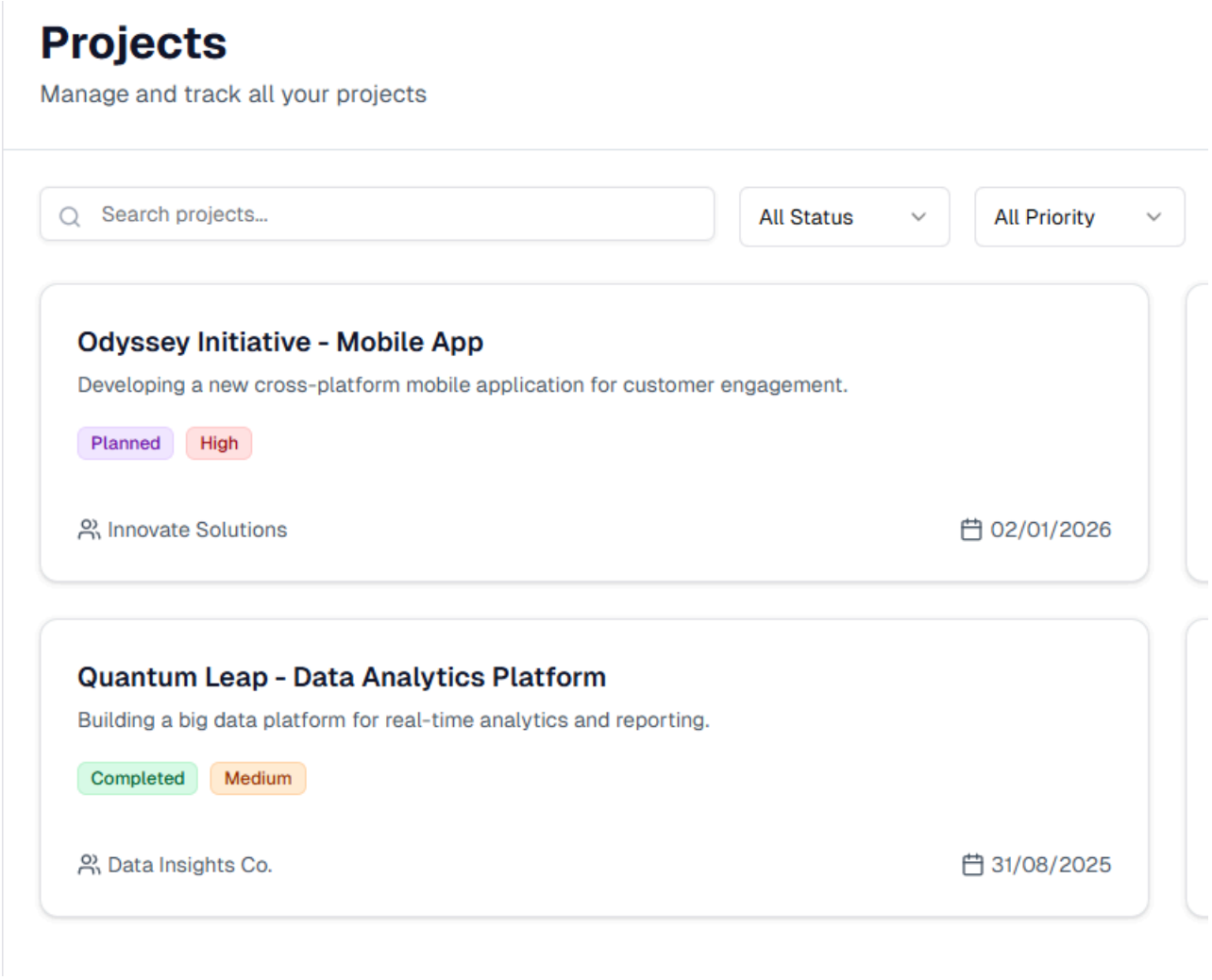

Рис. 4.3. Перелік наявних проєктів

Джерело: створено автором

Наступний важливий базовий елемент це детальна сторінка кожного проєкту де знаходиться kanban-дошка з відповідними колонками-статусами для керування завданнями проєкту.

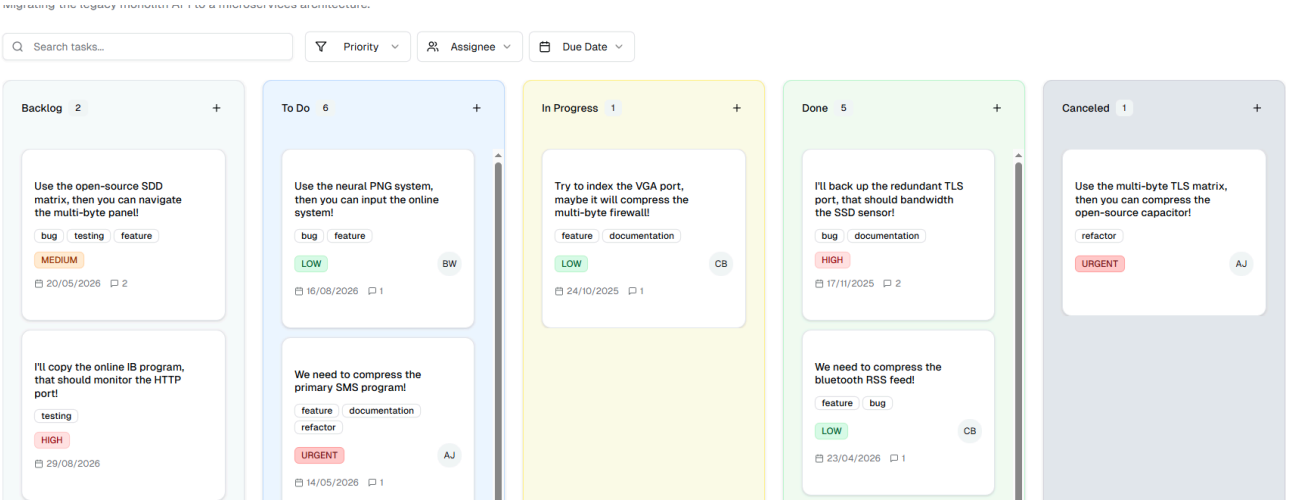

Рис. 4.4. Kanban дошка

Джерело: створено автором

На стороні сервера відповідний метод для отримання необхідних даних

Лістинг 4.2. Отримання усіх завданнь

```
async findAll(filters: QueryTaskDto): Promise<TaskListItemDto[]> {
    const tasks = await this.prisma.task.findMany({
      where: filters,
      select: {
        id: true,
        title: true,
        status: true,
        tags: true,
        dueDate: true,
        priority: true,
        assignee: {
          select: {
            id: true,
            name: true,
          },
        },
        _count: {
          select: { comments: true },
        },
      },
    });

    return tasks.map((t) => ({
      id: t.id,
```

```
      title: t.title,
      status: t.status as unknown as TaskListItemDto['status'],
      tags: t.tags,
      dueDate: t.dueDate?.toISOString(),
      priority: t.priority as unknown as TaskListItemDto['priority'],
      assignee: t.assignee
        ? { id: t.assignee.id, name: t.assignee.name }
        : undefined,
      commentsCount: t._count.comments,
    }));
  }
```

Відповідно для створення задач та інших відповідних дій наявні впливаючи вікнами

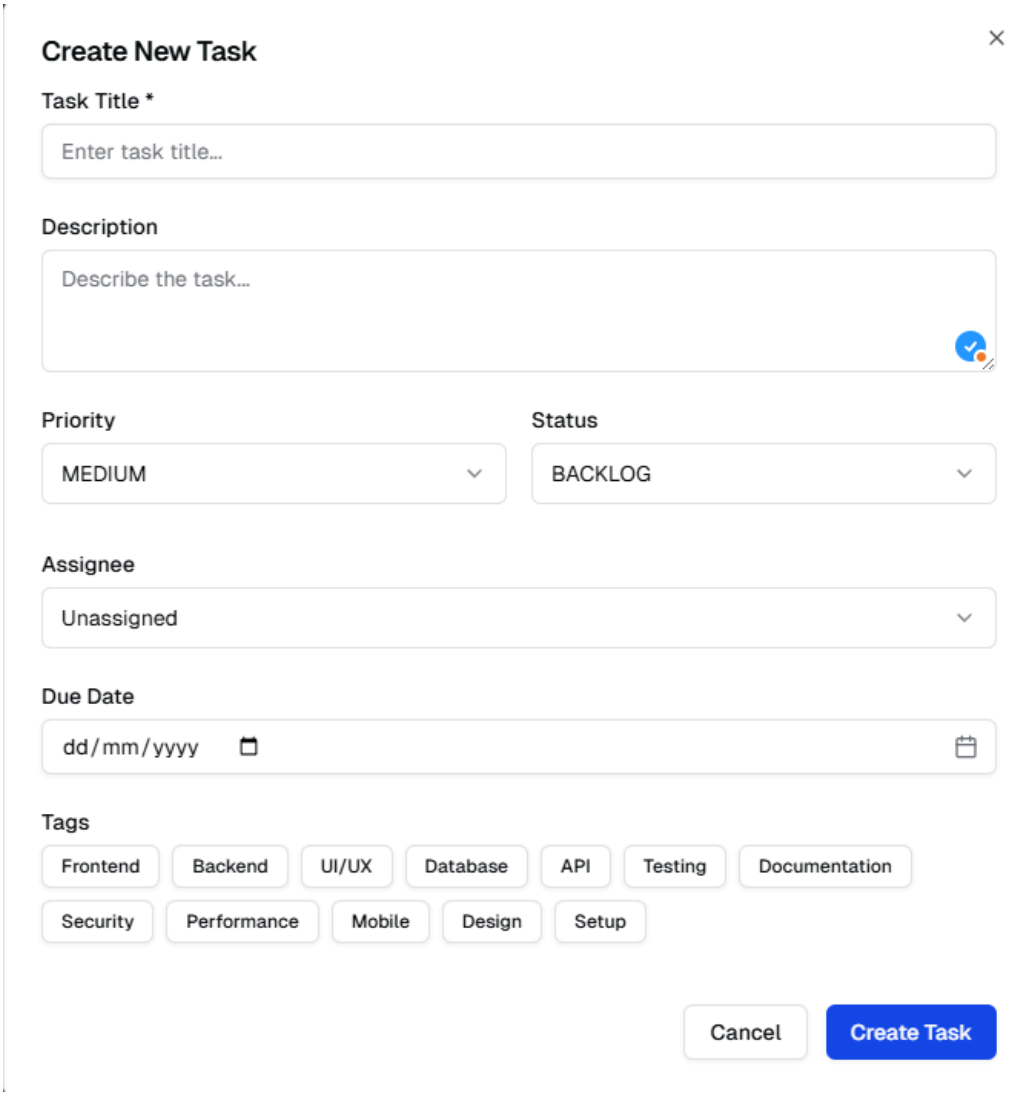

Рис. 4.5. Вікно для створення завдання

Джерело: створено автором

Не вдаючись в подробиці - ці та інші елементи будуть основою для впровадження AI функціоналу надалі.

4.2. Приклади AI-функцій платформи та їхній вплив на управління проєктами

На основі проаналізованих у попередніх розділах підходів до інтеграції штучного інтелекту можна виокремити низку функцій, які доцільно розглядати як кандидатів для впровадження в платформу управління проєктами. Нижче подано орієнтовний перелік таких функцій та коротко окреслено, як саме вони можуть підсилити роботу користувачів. Частина з них може бути реалізована в MVP-варіанті, інші - слугувати концептуальною основою для подальших ітерацій розвитку системи. Першою, однією з найочевидніших функцій є автоматичне узагальнення текстових обговорень та мітингів. У процесі роботи над проєктом накопичуються численні коментарі до задач, нотатки з зустрічей, дискусії в чатах. AI-модуль, інтегрований у платформу, може на вимогу користувача сформувати короткий підсумок за обраний період або за конкретну задачу: виділити ключові тези, основні рішення та невирішені питання. Для менеджера це означає суттєве скорочення часу на ознайомлення з історією обговорень, а для нових учасників команди - швидший вхід у контекст. У перспективі саме на цій функції зручно будувати порівняння різних моделей (наприклад, за повнотою та структурованістю підсумку). Друга потенційна функція - автоматичне виділення дій (action items) та формування задач. На основі тексту протоколу зустрічі або довгої дискусії в коментарях AI може виявити фрагменти, які містять формулювання на кшталт «потрібно зробити», «варто перевірити», «відповідальний N», і перетворити їх на пропозиції задач з попередньо заповненими полями (назва, короткий опис, ініціальний виконавець). Користувачеві залишається лише відкоригувати ці пропозиції та підтвердити їх створення. Така можливість допомагає зменшити кількість

«забутих» домовленостей та забезпечує більш дисципліновану фіксацію дій за результатами обговорень. Третьою важливою можливістю є інтелектуальний пошук по задачах і документації. Замість того щоб шукати інформацію лише за ключовими словами в заголовках чи описах, користувач може формулювати запити природною мовою: наприклад, «задачі, пов’язані з міграцією на нову базу даних» або «усі інциденти, де були проблеми з продуктивністю». AI-компонент аналізує зміст описів, коментарів і пов’язаних артефактів, щоб знайти семантично близькі записи. Це особливо корисно в довготривалих проєктах, де обсяг накопиченої інформації ускладнює навігацію традиційними засобами фільтрації. Четверта функція пов’язана з підтримкою пріоритизації задач [26]. Мовна модель може аналізувати описи задач, їхні зв’язки з іншими елементами проєкту, дедлайни, згадки в обговореннях і пропонувати менеджеру орієнтовний рейтинг пріоритетності або виділяти задачі, які потенційно є «критичними» для досягнення найближчих цілей. Такий механізм не замінює остаточного рішення менеджера, але надає додатковий шар аналітики, який допомагає не пропустити важливі елементи серед великої кількості дрібніших задач. У більш розвиненому варіанті ця функція може поєднуватися з історичними даними про затримки або інциденти. П’ятою функцією, що має значний потенціал для підтримки управління, є автоматизоване формування статус-репортів і коротких оглядів прогресу. На основі інформації про зміну статусів задач, нові коментарі, завершені етапи й зафіксовані ризики AI може згенерувати звіт у форматі, наближеному до очікувань стейкхолдерів: з розподілом на розділи «що зроблено», «що в роботі», «ризики й блокери», «плани на наступний період». Це дозволяє зменшити ручну працю менеджера зі складання регулярних оновлень і підвищує стандартизацію таких звітів у межах організації. Шостою, більш перспективною, але важливою з погляду дослідження, є функція аналізу ризиків та раннього виявлення проблемних задач. На основі патернів у текстах (часті згадки про блокери, негативні оцінки, повторювані перенесення дедлайнів), а також на основі історичних даних про аналогічні задачі AI-модуль може сигналізувати про підвищену ймовірність

затримки або ускладнень. Попри те, що побудова надійних прогнозних моделей потребує значних масивів даних, уже на рівні прототипу можна оцінити здатність різних мовних моделей виявляти «ризикові» сигнали на рівні текстових описів і коментарів.

Зазначені функції не є вичерпним переліком можливостей, але вони охоплюють ключові напрями, де інтеграція AI особливо природно вписується в платформу управління проєктами: робота з текстом, підтримка прийняття рішень, автоматизація рутинних дій та підвищення прозорості поточного стану. Надалі для окремих із цих функцій може бути проведено порівняльний аналіз роботи різних моделей, що дозволить емпірично оцінити їхню придатність до використання в проєктному менеджменті. Окремим перспективним напрямом є використання голосової взаємодії для створення задач. У цьому сценарії користувач формулює задачу усно (наприклад, під час мітингу або впродовж робочого дня), а система, використовуючи поєднання сервісу розпізнавання мовлення та мовної моделі, перетворює аудіозапис на структурований запис на канбан-дошці. На виході голосове повідомлення трансформується у задачу з назвою, коротким описом, а за можливості - з попередньо визначеними тегами, пріоритетом або навіть запропонованим виконавцем. Такий підхід знижує бар’єр для фіксації нових елементів роботи: задачі не «зависають» у голові чи особистих нотатках, а швидко потрапляють у спільний простір платформи. Для менеджера та команди це означає більш повний беклог та зменшення ризику втрати важливих ідей, які виникають у динамічному робочому середовищі.

4.3 Приклад реалізації AI-функціоналу

Оскільки весь раніше названий функціонал побудований за принципом third-party API + server + client, візьмемо для прикладу імплементацію функціоналу Voice-to-task. На стороні front-end використовується SpeechRecognition з нативно Chrome Web Speech API.

Лістинг 4.3. Використання Speech API

```
const recognition = new SpeechRecognition();
recognition.continuous = false;
recognition.lang = "en-US";
recognition.interimResults = false;
recognition.maxAlternatives = 1;

recognition.onresult = (event) => {
  const color = event.results[0][0].transcript;
  diagnostic.textContent = `Result received: ${color}`;
  bg.style.backgroundColor = color;
};
```

Вище згадане API дозволяє обробити SpeechRecognitionEvent, у результаті якого отримуємо готовий текст. Наступний етап це надсилання отриманого результату на специфічний endpoint, який займатиметься обробкою цього тексту. На backend стороні використовуючи пакет openai, де вказавши необхідну модель, та використовуючи потрібний промт [25], отримуємо результат у вигляді JSON, з якого пізніше створюємо готову сутність у базі даних, при необхідності використовуючи парсинг для зберігання єдино оголошених enums.

Лістинг 4.4. Запит до openai

```
const response = await this.openai.chat.completions.create({
        model: 'gpt-4o-mini',
        messages: [
          { role: 'system', content: system },
          {
            role: 'user',
            content:
              'Enums: priority=[LOW, MEDIUM, HIGH, URGENT],
status=[BACKLOG, TODO, IN_PROGRESS, DONE, CANCELED]. ' +
              'Output keys: title, description, priority, status,
dueDateIso, tags, assigneeName. ' +
              'Text: ' +
              text,
          },
        ],
        response_format: { type: 'json_object' },
        temperature: 0,
      });
```

4.4. Порівняльний аналіз моделей

Для оцінки ефективності використання різних мовних моделей у межах розроблюваної платформи було проведено експериментальне порівняння їх роботи на прикладі однієї з найбільш універсальних функцій - автоматичного узагальнення текстових обговорень (AI-summary). Дана функція є показовою, оскільки поєднує основні аспекти взаємодії людини з моделлю: сприйняття неструктурованого тексту, розуміння контексту та формування чіткої, стисненої відповіді, зрозумілої користувачу. Мета експерименту - перевірити, наскільки різні моделі здатні узагальнити одне й те саме текстове обговорення, зберігаючи ключові деталі, логіку подій і управлінські висновки.

Вхідні дані - фрагмент обговорення задачі в межах команди (кілька повідомлень із зазначенням проблеми, рішень і домовленостей).

Очікуваний результат - короткий підсумок у 5–7 речень із виокремленням основних тез, дій і ризиків.

Модель	**Формат відповіді**	**Повнота змісту (1–5)**	**Лаконічність (1–5)**	**Узгодженість і логіка (1–5)**
GPT-4o	JSON	4,5	5	4
Claude 4	Текст	5	4,5	5
Gemini 2.5 Flash	Текст	4	4,5	5

Таблиця 4.1 - Порівняння результатів роботи моделей при формуванні AI-summary

На основі проведеного аналізу можна зробити висновок, що всі протестовані моделі продемонстрували здатність виконувати завдання узагальнення, з суб’єктивної точки зори, усі моделі впорались із завдання вправно без особливих помилок, які могли б сильно вплинути на проєкт. Наявна певна

різниця у стилях написаннях, але вона корегується промптом. Зі сторони DE, використання моделей від open ai, надає можливість зразу отримувати відповідь у форматі JSON, що економить зусилля з парсингу при використанні інших моделей.

Висновки до розділу 4

У цьому розділі було представлено практичну частину дипломного дослідження, спрямовану на демонстрацію можливостей інтеграції штучного інтелекту у платформу для управління проєктами. На основі створеного MVP-рішення описано базовий набір функцій системи, який охоплює автентифікацію користувачів, створення та ведення проєктів, управління задачами, формування коментарів і перегляд стану виконання робіт. Цей фундамент забезпечує необхідну основу для впровадження інтелектуальних модулів, що працюють із наявними даними та розширюють традиційний функціонал системи проєктного менеджменту.

Було розглянуто низку AI-функцій, які потенційно підсилюють окремі елементи процесу управління. Зокрема, запропоновано автоматичне узагальнення обговорень та формування коротких звітів (AI-summary), що дозволяє швидко отримувати ключові тези й рішення з великих масивів коментарів або протоколів мітингів. Функція виділення дій (Action Item Extraction) забезпечує автоматичне формування задач на основі текстових обговорень, перетворюючи домовленості команди у конкретні дії. Описано також інтелектуальний пошук за змістом задач і коментарів, що спрощує навігацію в межах великих проєктів, а також AI-підтримку пріоритизації, яка допомагає менеджеру виявляти критичні або найбільш значущі задачі. Додатково запропоновано функцію створення задач за допомогою голосу, що відкриває новий спосіб взаємодії з системою та знижує бар’єр для фіксації нових ідей.

У межах дослідження проведено порівняльний аналіз роботи мовних моделей на прикладі завдання автоматичного узагальнення обговорень.

Порівняння дало змогу оцінити якість результатів за такими критеріями, як повнота, лаконічність, логічна послідовність і структурованість тексту. На основі отриманих спостережень можна стверджувати, що інтеграція штучного інтелекту у системи управління проєктами має значний потенціал у кількох вимірах. По-перше, AI сприяє скороченню рутинних дій і підвищенню продуктивності команди шляхом автоматизації підготовки звітів, фіксації задач і оброблення коментарів. По-друге, використання мовних моделей створює передумови для покращення якості управлінських рішень, оскільки користувачі отримують швидкий доступ до аналітичних і контекстних підсумків. По-третє, інтеграція AI відкриває нові форми взаємодії - від семантичного пошуку до голосового введення - що робить платформу більш адаптивною та інтуїтивною.

Водночас результати дослідження показали, що впровадження AI-компонентів вимагає уважного ставлення до питань якості даних, етичних аспектів, стабільності моделей і захисту конфіденційної інформації. Тому інтеграція штучного інтелекту повинна розглядатися не як одноразова функція, а як поступовий процес розвитку системи, що передбачає тестування, оцінку й постійне вдосконалення моделей відповідно до потреб користувачів.

У підсумку, результати роботи над цим розділом підтверджують практичну доцільність використання AI у платформах управління проєктами. Поєднання базових інструментів організації роботи з інтелектуальними можливостями аналізу та узагальнення створює основу для нової генерації систем, які не лише відображають стан проєкту, а й активно допомагають у прийнятті рішень, координації команди та підвищенні ефективності всієї організації.

ВИСНОВКИ

У магістерській роботі було здійснено комплексне дослідження проблеми інтеграції технологій штучного інтелекту у процеси управління проєктами та розроблено концептуальну модель платформи, що поєднує класичні механізми проєктного менеджменту з інтелектуальними сервісами аналізу, узагальнення та підтримки прийняття рішень.

У теоретичній частині проведено огляд сучасних підходів до управління проєктами, визначено основні труднощі, з якими стикаються команди - зокрема надлишок інформації, складність координації між учасниками, потреба у швидкому прийнятті рішень на основі неповних даних. Було проаналізовано потенціал використання AI для розв’язання цих проблем, а також описано напрями, у яких такі технології можуть забезпечити найбільшу користь - автоматизація звітності, аналітика текстових обговорень, прогнозування ризиків, пріоритизація задач.

У другому розділі дослідження зосереджено увагу на наявних підходах до інтеграції штучного інтелекту у вебплатформи. Проаналізовано доступні інструменти та рішення - n8n, Zapier, пряме підключення до API моделей - і обґрунтовано вибір архітектури, що поєднує гнучкість REST-підходу з можливістю подальшого масштабування. Визначено ключові вимоги до системи, її користувачів і сценарії взаємодії, що створюють основу для розроблення мінімально життєздатного продукту (MVP).

Практична частина роботи (розділ 4) продемонструвала, як на базі сучасного стека технологій (Nest.js, Next.js, PostgreSQL, Prisma) можна реалізувати платформу для управління проєктами з подальшим розширенням шляхом AI-модулів. Було описано основний функціонал системи - створення користувачів, проєктів, задач, управління статусами - та запропоновано шість напрямів розвитку з використанням штучного інтелекту. Серед них:

- автоматичне узагальнення обговорень і мітингів (AI-summary);
- генерація задач на основі тексту або голосу (Action Item / Voice-to-Task);

- семантичний пошук по контенту;
- пріоритизація задач;
- формування звітів;
- прогнозування ризиків виконання.

У рамках дослідження проведено експеримент із порівняння результатів роботи різних мовних моделей для задачі текстового узагальнення. Загалом результати роботи засвідчили, що інтеграція штучного інтелекту у системи управління проєктами здатна суттєво підвищити ефективність командної взаємодії, зменшити навантаження на менеджерів, покращити якість звітності та аналітики. Така платформа може не лише відображати стан виконання завдань, а й проактивно допомагати користувачам у прийнятті рішень, виявленні ризиків та оптимізації процесів.

Подальші дослідження можуть бути спрямовані на вдосконалення інтеграційного шару, впровадження механізмів навчання моделей на власних даних, а також розроблення інтерфейсів, які забезпечать природну взаємодію з користувачем - зокрема через голосові команди або персоналізованих AI-асистентів. Реалізація таких напрямів сприятиме формуванню нової генерації інтелектуальних систем управління проєктами, де технології штучного інтелекту виступатимуть не допоміжним інструментом, а активним учасником процесу планування, аналізу та контролю.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Rajabion, Ladan, et al. “An investigation of project management software tools: AI-based analysis.” *Journal of Ambient Intelligence and Humanized Computing* 11 (2020): 2319–2331.
   URL: link.springer.com/article/10.1007/s12652-019-01381-7
2. Ahmed, Jamil, et al. “Artificial intelligence techniques for project management: A review.” *IEEE Access* 8 (2020): 163962–163975.
   URL: ieeexplore.ieee.org/document/9166714
3. “AI in Project Management: Opportunities and Challenges.” Project Management Institute (PMI).
   URL:pmi.org/learning/library/ai-project-management-opportunities-challenges-11809
4. n8n: Which Automation Tool Should You Choose?” n8n Blog.
   URL: blog.n8n.io/tag/guide/
5. Brown, Tom, et al. “Language Models are Few-Shot Learners.” *arXiv preprint* arXiv:2005.14165 (2020).
   URL: arxiv.org/abs/2005.14165
6. “How to Run LLMs Locally: A Practical Guide.” Hugging Face Blog.
   URL: huggingface.co/blog/run-llms-locally
7. Touvron, Hugo, et al. “LLaMA: Open and Efficient Foundation Language Models.” *arXiv preprint* arXiv:2302.13971 (2023).
   URL: arxiv.org/abs/2302.13971
8. “LangChain for Developers: Building with LLMs.” LangChain Documentation. URL: python.langchain.com/docs
9. Документація для бібліотеки React.js. URL: https://react.dev/reference/react.
10. Документація для інструменту ESLint. URL: https://eslint.org/docs.
11. Конфігурація для Prettier. URL: https://prettier.io/docs/en/configuration.html.
12. How To Structure React Projects From Beginner To Advanced. URL: https://blog.webdevsimplified.com/2022-07/react-folder-structure/.

13. MDN web docs for JavaScript. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript.
14. Документація для інструменту react-hook-form. URL: https://react-hook-form.com/docs.
15. Документація та для технології Tailwind CSS. URL: https://tailwindcss.com/docs/configuration.
16. Документація для метафреймворку Next.js. URL: https://nextjs.org/docs
17. Репозиторій бібліотеки tailwind-merge. URL: https://github.com/dcastil/tailwind-merge.
18. Документація для бібліотеки Zod. URL: https://zod.dev. High performance
19. GitHub репозиторій по PostCSS. URL: https://github.com/postcss/postcss.
20. GitHub репозиторій для плагіну prettier-plugin-tailwindcss. URL: https://github.com/tailwindlabs/prettier-plugin-tailwindcss.
21. Документація для препроцесора SASS. URL: https://sass-lang.com/documentation.
22. Utility types in TypeScript. URL: https://typescriptlang.org/docs/handbook/utility-types.html.
23. Nest.js Документація. URL: https://docs.nestjs.com/.
24. Prisma ORM Документація. URL: https://www.prisma.io/docs.
25. OpenAI API Reference. URL: https://platform.openai.com/docs/api-reference.
26. “AI-Assisted Project Management: A Systematic Review.” Journal of Intelligent & Fuzzy Systems, 2022. URL: https://content.iospress.com/articles/journal-of-intelligent-and-fuzzy-systems/ifs223523.
27. TanStack Query (React Query) Документація. URL: https://tanstack.com/query/latest/docs/react/overview.
28. Zustand State Manager Документація. URL: https://zustand-demo.pmnd.rs/.
29. Vercel Deployment Документація. URL: https://vercel.com/docs.
30. “Human-in-the-loop Machine Learning: Challenges and Opportunities.” arXiv preprint (2021). URL: https://arxiv.org/abs/2108.00076.

31. PostgreSQL Документація. URL: https://www.postgresql.org/docs/.
32. “A Review of Web Application Architectures.” ACM Computing Surveys, 2020. URL: https://dl.acm.org/doi/10.1145/3380873.