ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка програмного забезпечення для контролю особистих проєктів та завдань»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Владислав АДАМОВИЧ
(підпис)

Виконав: здобувач вищої освіти групи ПД-43

Владислав АДАМОВИЧ

_____

Керівник: Тимур ДОВЖЕНКО
к.т.н.

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти Бакалавр
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Адамовичу Владиславу Олеговичу

1. Тема кваліфікаційної роботи: «Розробка програмного забезпечення для контролю особистих проєктів та завдань»
керівник кваліфікаційної роботи к.т.н. Тимур ДОВЖЕНКО,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи:

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Аналіз існуючих програмних рішень для контролю особистих завдань і проєктів.
2. Формулювання вимог та постановка задачі розробки веб-додатку.
3. Проєктування архітектури системи управління особистими проєктами та завданнями.
4. Реалізація клієнтської та серверної частини додатку.
5. Реалізація модуля прогнозування ризиків.
6. Проведення тестування, аналіз результатів і оцінка ефективності системи.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Архітектура системних компонентів.
5. Алгоритм роботи .
6. Екранні форми.
7. Демонстрація роботи програми.
8. Апробація результатів дослідження.

6. Дата видачі завдання «25» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02.-04.03.2025	
2	Аналіз та дослідження існуючих аналогів	05.03-13.03.2025	
3	Формулювання вимог і постановка задачі	14.03 – 20.03.2025	
4	Проєктування архітектури веб-додатку	21.03 – 31.03.2025	
5	Реалізація клієнтської та серверної частини	01.04 – 18.04.2025	
6	Тестування та оптимізація веб-додатку	19.04 – 28.04.2025	
7	Оформлення роботи: вступ, висновки, реферат	29.04-11.05.2025	
8	Розробка демонстраційних матеріалів	12.05-18.05.2025	
9	Попередній захист роботи	19.05-30.05.2025	

Здобувач вищої освіти _____ Владислав АДАМОВИЧ
(підпис)

Керівник
кваліфікаційної роботи _____ Тимур ДОВЖЕНКО
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 63 стор., 4 табл., 3 рис., 40 джерел.

Мета роботи – розробка програмного забезпечення для покращення ефективності контролю за виконанням особистих проєктів і завдань, що забезпечує управління задачами, дотримання дедлайнів і виявлення ризиків.

Об'єкт дослідження – процес моніторингу виконання особистих проєктів.

Предмет дослідження – вебзастосунок для створення контролю особистих проєктів.

Короткий зміст роботи: у роботі проаналізовано сучасні програмні рішення для контролю особистих завдань і проєктів, виявлено їхні сильні та слабкі сторони. Розроблено архітектуру веб-додатку, який дозволяє користувачам створювати та керувати особистими завданнями, відстежувати хід виконання, встановлювати дедлайни, а також отримувати рекомендації щодо уникнення перевантаження. Програмна реалізація охоплює клієнтську частину на JavaScript та HTML, серверну частину на Python (Flask), базу даних MySQL. Застосовано принцип клієнт-серверної архітектури, RESTful API та токенну автентифікацію (JWT). Реалізовано модуль прогнозування ризиків, який аналізує кількість завдань, терміни виконання та навантаження користувача. Проведено функціональне та модульне тестування додатку.

У роботі використано Flask для серверної логіки, JavaScript для інтерактивності інтерфейсу, MySQL для зберігання даних, а також засоби валідації, авторизації та обробки помилок.

Сферою використання застосунку є управління особистими завданнями та проєктами з метою підвищення продуктивності та самоорганізації.

КЛЮЧОВІ СЛОВА: УПРАВЛІННЯ ЗАВДАННЯМИ, ПЛАНУВАННЯ, FLASK, JAVASCRIPT, MYSQL, ВЕБ-ДОДАТОК, REST API, АУТЕНТИФІКАЦІЯ, ПРОГНОЗУВАННЯ РИЗИКІВ, САМООРГАНІЗАЦІЯ.

ЗМІСТ

ВСТУП

В умовах цифрової трансформації та стрімкого розвитку інформаційних технологій управління часом, завданнями та особистими проєктами стає однією з ключових навичок сучасної людини. Зростання обсягів інформації, динаміка робочих процесів, необхідність одночасного виконання кількох завдань і проєктів вимагає від користувача ефективних інструментів для самоорганізації та планування. Саме тому актуальним є створення програмного забезпечення, що забезпечить централізований контроль над особистими проєктами та завданнями, дозволить користувачеві не лише впорядковувати поточні справи, а й здійснювати прогнозування можливих ризиків у реалізації цілей.

Актуальність теми обумовлюється зростаючою потребою в універсальних та гнучких цифрових інструментах для планування, розподілу та контролю задач. Існуючі рішення, попри свою популярність, часто мають низку обмежень: складний інтерфейс, обмежену можливість модифікацій, відсутність функцій прогнозування ризиків або автоматизованого аналізу навантаження. Тому розробка нової системи, яка дозволяє поєднувати класичне управління завданнями з елементами аналітики, адаптованої під конкретні потреби користувача, є важливим і своєчасним завданням.

Метою дослідження є розробка функціонального та адаптивного веб-додатку, що забезпечує користувача можливістю контролю особистих проєктів та завдань, дозволяє створювати, змінювати та моніторити завдання, а також прогнозувати потенційні ризики. Водночас особлива увага приділяється реалізації безпечної системи аутентифікації, інтуїтивно зрозумілого інтерфейсу користувача та використанню сучасних веб-технологій.

Для досягнення поставленої мети в роботі передбачено виконання таких завдань:

1. Провести аналіз існуючих програмних рішень для управління проєктами та завданнями;

2. Сформулювати вимоги до функціоналу веб-додатку відповідно до потреб цільової аудиторії;
3. Спроєктувати архітектуру майбутньої системи з урахуванням клієнт-серверної моделі;
4. Реалізувати серверну частину з використанням Python (Flask) та бази даних MySQL;
5. Розробити клієнтську частину на основі JavaScript та HTML;
6. Створити модуль прогнозування ризиків із наданням рекомендацій;
7. Провести тестування системи, проаналізувати її стабільність та ефективність;
8. Надати рекомендації щодо можливого вдосконалення та розширення функціоналу.

Об’єктом дослідження є процеси управління особистими завданнями та проєктами в цифровому середовищі.

Предметом дослідження виступають програмні засоби та архітектурні рішення, що забезпечують реалізацію системи контролю особистих проєктів та завдань із можливістю аналітики ризиків.

Джерельну базу дослідження становлять публікації у фахових науково-технічних журналах, документація до фреймворків Flask, JavaScript API, офіційні гіди з Django, Vue.js, REST API, а також інтернет-ресурси, присвячені розробці CRUD-додатків, реалізації клієнт-серверної архітектури, дослідженню UX-дизайну тощо. Особливе значення мали матеріали, присвячені інтеграції систем аутентифікації за допомогою JWT, побудові RESTful API та методам оцінювання ризиків на основі даних про навантаження користувача та строки виконання завдань.

Фактичний матеріал, використаний у роботі, включає реалізований веб-додаток, побудований на стеку технологій Python (Flask), JavaScript, HTML, CSS та MySQL, а також експериментальні дані, зібрані в результаті тестування функціоналу додатку в умовах наближених до реального використання. Створені API-маршрути, модулі управління користувачами, проєктами та завданнями, логіка прогнозування ризиків були апробовані шляхом симуляції типових сценаріїв користування.

Методи дослідження, застосовані в роботі, охоплюють:

- **Метод аналізу та синтезу** – для узагальнення вимог до майбутнього додатку;
- **Метод порівняльного аналізу** – для визначення переваг і недоліків існуючих рішень;
- **Метод моделювання** – під час проєктування архітектури системи;
- **Метод програмного експерименту** – при створенні прототипу системи;
- **Метод системного аналізу** – для оцінювання ефективності роботи модулів;
- **Методи візуалізації** – під час побудови діаграм архітектури, послідовностей, структури даних.

Наукова новизна одержаних результатів полягає в тому, що:

- **Вперше визначено** модель взаємодії між компонентами системи управління завданнями з вбудованим ризик-аналізом на основі дедлайнів та завантаженості користувачів;
- **Удосконалено** структуру програмного забезпечення за рахунок модульного підходу та ізольованого контролю за проектами, учасниками та завданнями;
- **Дістало подальший розвиток** застосування RESTful API в системах особистого планування через оптимізацію запитів та підвищення захищеності взаємодії;
- **Створено нову концепцію**, що узагальнює інструменти управління завданнями із вбудованими функціями оцінювання ризиків та аналітики прогресу;
- **Розроблено нову систему**, яка поєднує зручний інтерфейс, адаптивність до користувача та прогнозуючу функцію з наданням індивідуальних рекомендацій;
- Усі компоненти системи реалізовано **з використанням відомого принципу** розділення обов'язків (Separation of Concerns), що забезпечує масштабованість і підтримуваність коду.

Теоретична цінність полягає в розробці підходу до побудови архітектури веб-додатків із можливістю гнучкого керування особистими проєктами та задачами. Робота демонструє впровадження клієнт-серверної моделі з ефективною взаємодією між Frontend, Backend та базою даних, що може бути використано як

основа для подальших досліджень у галузі розподілених систем управління завданнями.

Практичне значення дослідження виявляється в можливості безпосереднього впровадження створеного веб-додатку для потреб окремих користувачів або малих команд. Система може бути адаптована для застосування у фріланс-проєктах, освітніх платформах, внутрішньокорпоративних рішеннях тощо. Веб-додаток є відкритим до подальшого розширення, включаючи використання ML-модулів для прогнозування ефективності командної роботи або автоматизованого розподілу завдань.

Апробація роботи здійснювалась під час тестування створеного додатку в умовах реального використання на локальному сервері. Проведено серію тестів з перевіркою працездатності ключових функцій, у тому числі: реєстрація, створення проєкту, додавання користувачів, створення завдань, оновлення статусів, аналіз ризиків. Після виявлення помилок реалізовано відповідні оптимізації на рівні API-маршрутів, обробки даних у базі та валідації користувацького вводу.

Отже, розроблена система відповідає актуальним запитам у сфері особистого планування, відрізняється функціональністю, стабільністю та зручн у використанні, має чітко визначену архітектуру та відкриту модель для подальшого розвитку. Це дозволяє стверджувати про доцільність її застосування як в особистих, так і професійних цифрових середовищах.

1. АНАЛІЗ ІСНУЮЧИХ ЗАСОБІВ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Сучасні програмні рішення для контролю особистих завдань

Сучасні програмні рішення для контролю особистих проєктів і завдань охоплюють широкий спектр інструментів, які можна класифікувати за різними критеріями. Найпоширенішим критерієм виступає призначення програмного продукту, що дозволяє виокремити персональні трекери завдань, командні системи керування проєктами, а також універсальні платформи, які поєднують функціонал обох попередніх типів. Персональні трекери зазвичай орієнтовані на одного користувача і включають базові функції створення, редагування та відстеження виконання завдань. До них належать мобільні та десктопні застосунки з можливістю інтеграції календаря, нагадувань та міток. У таких системах акцент робиться на індивідуальній організації часу, підвищенні продуктивності та боротьбі з прокрастинацією. Командні системи управління проєктами, навпаки, орієнтовані на колективну роботу над завданнями, координацію учасників, делегування обов’язків, ведення спільних графіків та комунікацію всередині команди. До цієї категорії належать інструменти, які дозволяють формувати проєктні структури, ієрархію підзадач, вказувати відповідальних, визначати дедлайни, а також оцінювати прогрес виконання візуально (через канбан-дошки, діаграми Ґанта тощо). Універсальні системи поєднують риси обох згаданих типів і дозволяють як індивідуальне використання, так і командну взаємодію. Завдяки гнучкості налаштувань, такі рішення можуть масштабуватись від особистих потреб до роботи невеликих команд і навіть корпоративного рівня [1].

Окремим напрямом класифікації є тип інтерфейсу взаємодії з користувачем. За цим критерієм виділяють веб-орієнтовані додатки, мобільні застосунки, настільні програми, а також гібридні системи з підтримкою кількох платформ. Веб-додатки мають перевагу в доступності з будь-якого пристрою, що має браузер, однак іноді поступаються за швидкодією чи офлайн-функціональністю. Мобільні

застосунки орієнтовані на оперативну взаємодію та отримання сповіщень у режимі реального часу. Вони зручні для щоденного використання, проте можуть мати обмежений функціонал порівняно з десктопними або веб-версіями. Настільні додатки забезпечують повноцінний доступ до всіх функцій системи, часто мають вищу стабільність, підтримку локального збереження даних та більші можливості кастомізації. Гібридні системи намагаються поєднати переваги усіх форматів, використовуючи прогресивні веб-застосунки (PWA), хмарні синхронізації та кросплатформені фреймворки. Ще один підхід до класифікації полягає у способі реалізації архітектури: автономні офлайн-додатки, хмарні сервіси, системи з клієнт-серверною архітектурою або мікросервісні рішення. Автономні програми функціонують незалежно від мережі, однак не забезпечують колективної роботи та синхронізації. Хмарні сервіси орієнтовані на зберігання даних на сторонніх серверах із постійним підключенням до Інтернету. Клієнт-серверна архітектура дозволяє відокремити логіку обробки даних від інтерфейсу користувача, забезпечити масштабованість та багаторівневу безпеку. Мікросервісні рішення, хоча й складніші у реалізації, забезпечують найвищу гнучкість, незалежність модулів і стійкість до збоїв окремих компонентів. У межах цієї класифікації також розглядаються платформи з відкритим кодом, комерційні продукти, а також безкоштовні або умовно-безкоштовні рішення, що базуються на підписках або ліцензуванні за кількістю користувачів [9].

Крім того, сучасні програмні рішення класифікуються за рівнем інтеграції зі сторонніми сервісами. Високорівневі системи підтримують API для підключення зовнішніх календарів, CRM-систем, сервісів електронної пошти, месенджерів та систем зберігання файлів. Вони дозволяють централізувати всі канали комунікації, отримувати сповіщення, автоматизувати виконання дій за допомогою тригерів і сценаріїв (наприклад, через Zapier, IFTTT, Webhooks). Системи середнього рівня підтримують лише базову інтеграцію (наприклад, імпорт/експорт у CSV, інтеграцію з Google Calendar). Найнижчий рівень — автономні програми, які працюють із власною структурою збереження даних і не передбачають розширень. Також класифікацію можна здійснювати за ступенем підтримки командної роботи:

однокористувацькі (приватне використання), багатокористувацькі (спільна взаємодія), а також корпоративні з повною ієрархією ролей, прав доступу, журналом змін та модулями аналітики [2].

У межах дослідження особливий інтерес становлять системи з відкритою структурою, які дають змогу реалізувати авторські рішення, розширювати функціональність, змінювати інтерфейс і додавати інтелектуальні модулі. Відкритий підхід дозволяє створити додаток, максимально адаптований до потреб користувача, що має ключове значення для особистого контролю завдань і гнучкого планування. Таким чином, сучасні програмні рішення для контролю завдань не є уніфікованими системами, а радше представляють багатогранний спектр підходів, архітектур та сценаріїв використання, що дозволяє сформувати концептуальну базу для створення власного програмного забезпечення, орієнтованого на персоналізований досвід управління завданнями [15].

У сфері індивідуального управління завданнями та проєктами існує велика кількість програмних рішень, які стали популярними серед широкого кола користувачів завдяки простоті у використанні, зручному інтерфейсу та можливості налаштувати систему під особисті потреби. Серед найбільш відомих інструментів, що використовуються для персонального планування, можна виділити Todoist, Microsoft To Do, Google Tasks, Notion, TickTick та інші. Їхня популярність пояснюється здебільшого тим, що вони дозволяють користувачам швидко фіксувати завдання, структурувати їх за пріоритетами, встановлювати нагадування, повторюваність, дедлайни, а також вести особисту статистику виконання. Todoist, наприклад, є одним із найвідоміших сервісів, який поєднує функціональність GTD (Getting Things Done), простий інтерфейс і можливість синхронізації між пристроями. Користувач має змогу створювати проєкти, поділяти завдання на підзавдання, додавати мітки, пріоритети, а також користуватися фільтрами для вибірки справ за певними умовами. Цей інструмент відзначається підтримкою великої кількості інтеграцій — Google Calendar, Zapier, Outlook, а також можливістю налаштування правил автоматичного розподілу завдань [3].

Microsoft To Do став продовженням та оновленою версією колишнього Wunderlist, і є глибоко інтегрованим у середовище Microsoft 365, що робить його надзвичайно зручним для користувачів екосистеми Windows. Завдяки синхронізації з Outlook і можливістю створення списків завдань, включення нагадувань, встановлення термінів та повторень, цей сервіс підходить як для особистого, так і для професійного використання. Google Tasks, хоча й має мінімалістичний функціонал, користується популярністю серед прихильників екосистеми Google. Його головна перевага — інтеграція з Gmail і Google Calendar, що дозволяє легко створювати завдання безпосередньо з електронних листів і контролювати їх у календарі. Цей інструмент підходить для користувачів, які цінують швидкість, простоту й легкість доступу [4].

Notion, хоч і є універсальним цифровим органайзером, завдяки модульній структурі також активно використовується як персональний менеджер завдань. Його головна перевага — гнучкість у створенні власних шаблонів сторінок, таблиць, баз даних, систем нотаток і задач. Користувачі можуть побудувати власну систему управління, комбінуючи завдання з цілями, нотатками, навчальними матеріалами чи особистими щоденниками. Notion підтримує як спискову, так і табличну та канбан-візуалізацію, дає можливість встановлювати зв’язки між елементами, додавати теги, дедлайни, стан виконання, нагадування та візуальні компоненти. Завдяки цьому він став популярним серед фрілансерів, студентів, авторів і тих, хто веде багатопроєктну діяльність, поєднуючи планування з особистим розвитком. TickTick — ще один популярний застосунок, який поєднує класичну структуру завдань із елементами календаря, трекінгу звичок, помодоро-таймера та вбудованого аналізу продуктивності. Користувачі можуть створювати ієрархічні структури задач, встановлювати пріоритети, вести облік повторюваних дій та отримувати зведення про динаміку виконання [12].

Серед інших цікавих рішень варто згадати Any.do, який має простий та інтуїтивний інтерфейс, функцію голосового введення задач, інтеграцію з календарем і підтримку віджетів на мобільних пристроях. Для користувачів iOS особливо зручними є вбудовані нагадування Apple Reminders, які синхронізуються

через iCloud та підтримують голосове введення через Siri. Цей сервіс, попри свою базову структуру, задовольняє потреби користувачів, які шукають мінімалістичний, але надійний спосіб контролю особистих завдань. У свою чергу, Habitica представляє гейміфікований підхід до управління задачами, де виконання завдань перетворюється на гру з накопиченням очок, прокачуванням персонажа та проходженням місій. Такий підхід особливо ефективний для тих, хто шукає додаткову мотивацію через ігрові механіки.

Зазначені інструменти відрізняються за рівнем гнучкості, глибини налаштувань, підтримки інтеграцій та підходом до візуалізації інформації. Наприклад, канбан-системи, як-от Trello або KanbanFlow, хоч і більше орієнтовані на командну роботу, також успішно використовуються індивідуальними користувачами для побудови особистого планування. Багато інструментів надають можливість ведення щоденників, налаштування звичних рутин, формування щотижневих/щомісячних оглядів, інтеграції з календарями, створення шаблонів та автоматизації. Усі ці функції спрямовані на одне — допомогти людині не лише зберігати інформацію про справи, а й управляти ними ефективно, з мінімальними витратами часу та максимальною прозорістю[20].

Командні системи управління завданнями з розширеним функціоналом є важливою складовою сучасного інформаційного середовища, оскільки вони дозволяють не лише організовувати спільну роботу над проєктами, а й формувати повноцінну екосистему керування процесами, ролями, ресурсами та результатами. Вони створені для забезпечення ефективної комунікації між учасниками команди, контролю за термінами виконання завдань, візуалізації прогресу та інтеграції з іншими корпоративними сервісами. До найвідоміших представників цього класу систем належать Jira, Asana, ClickUp, Monday.com, Wrike та інші. Кожен із цих інструментів виконує не лише функції класичного планування завдань, а й дозволяє формувати ієрархію підпроєктів, створювати шаблони, відстежувати часові витрати, призначати завдання з урахуванням навантаження, налаштовувати автоматизовані правила й повідомлення, а також отримувати аналітичні звіти щодо ефективності діяльності команди [37].

Jira є одним із найпотужніших інструментів для командної роботи, особливо у сфері розробки програмного забезпечення. Його основною перевагою є підтримка гнучких методологій керування, зокрема Scrum та Kanban, а також можливість створення беклогів, спринтів, оцінювання завдань у story points, автоматизація робочих процесів за допомогою правил і тригерів. Інтеграція з Confluence, Bitbucket та іншими сервісами Atlassian дає змогу побудувати єдину платформу для розробки, документування, тестування та підтримки продукту. Також важливим аспектом є система ролей і прав доступу, що дозволяє точно налаштувати, хто з учасників має змогу переглядати, редагувати чи затверджувати окремі елементи. Jira підтримує створення складних звітів, діаграм виконання завдань, burn-down charts, velocity charts та інші інструменти моніторингу продуктивності [9].

Asana орієнтована на більш широку аудиторію, включаючи команди в галузях маркетингу, HR, продажів, дизайну. Система забезпечує зручний інтерфейс для створення завдань, розподілу відповідальних осіб, встановлення дедлайнів і залежностей між завданнями. Завдяки можливості вибору між списковим, канбан- і календарним представленням, користувачі мають змогу адаптувати інструмент до власного стилю роботи. У Asana реалізовано потужний механізм таймлайнів, що дозволяє планувати тривалість завдань у часі, уникати конфліктів у навантаженні, а також створювати автоматичні повідомлення про прострочені або виконані задачі. Вбудована аналітика дозволяє оцінювати прогрес у межах проєкту або команди загалом, а інтеграція з Google Workspace, Slack, Zoom, Dropbox та іншими сервісами перетворює систему на центр командної взаємодії [6].

ClickUp пропонує ще більш гнучкий підхід, дозволяючи користувачу самостійно налаштовувати не лише інтерфейс, а й логіку роботи системи. Тут можна створювати власні статуси завдань, поля, шаблони, типи переглядів, фільтри, автоматичні дії. Крім стандартних можливостей, ClickUp підтримує трекінг часу, складання цілей, збирання зворотного зв’язку, створення діаграм Ґанта, побудову баз знань, розміщення коментарів у реальному часі. Унікальною особливістю системи є можливість поєднання інструментів управління завданнями, документацією, спілкуванням, звітністю та автоматизацією без потреби в

додаткових сервісах. Завдяки цьому ClickUp часто використовують компанії, які прагнуть централізувати всі бізнес-процеси в одному середовищі, а не розпорошувати їх по кількох інструментах.

Monday.com має яскравий інтерфейс і фокусується на візуалізації даних. Основною одиницею в системі є “дошка” — структура, яку можна адаптувати під будь-який робочий процес: від розробки продукту до обліку відпусток. Користувачі мають змогу створювати власні типи полів (дата, статус, вибір з кількох варіантів, формули), сортувати і фільтрувати інформацію, автоматизувати дії за допомогою “рецептів” (if-this-then-that). Monday.com підтримує інтеграцію з зовнішніми платформами, створення дашбордів, на яких можна поєднати показники з кількох проєктів, а також налаштування повідомлень за різними умовами. Система особливо популярна серед компаній, які ведуть кілька паралельних процесів і прагнуть мати узагальнене представлення діяльності всього колективу [3].

Усі командні системи з розширеним функціоналом мають спільні риси, серед яких — підтримка багаторівневої структури (завдання, підзавдання, етапи), календарна прив’язка, взаємодія між учасниками в режимі реального часу, історія змін, аналітика й гнучка система повідомлень. Водночас кожен продукт пропонує власний унікальний підхід до організації роботи, що дозволяє користувачам вибирати інструмент відповідно до типу проєкту, складу команди, характеру завдань та індивідуальних переваг. Для розробника програмного забезпечення надзвичайно важливим є розуміння принципів побудови таких систем, їхньої архітектури, логіки розмежування доступів, механізмів взаємодії між клієнтською і серверною частиною, а також специфіки зберігання даних [1].

У практиці створення власного веб-додатку для контролю особистих проєктів ці знання можуть бути адаптовані для реалізації окремих функцій, таких як керування користувачами, розподіл задач, оцінювання навантаження, прогнозування ризиків. Використовуючи кращі практики популярних систем, можна побудувати зручний, масштабований та безпечний інструмент, що поєднує переваги персонального планування з можливостями командної взаємодії. У результаті створена система стане не просто планувальником, а ефективним інструментом прийняття рішень, координації дій і досягнення цілей. Командні

системи з розширеним функціоналом продовжують активно розвиватися, впроваджуючи штучний інтелект, автоматизовані поради, інтеграцію з DevOps-інструментами, CRM-модулями, сервісами документообігу. Усе це підтверджує, що подібні системи є не лише трендом, а й необхідністю для організацій, що прагнуть підвищити ефективність своєї діяльності через цифровізацію робочих процесів і оптимізацію взаємодії між учасниками команди [3].

Таблиця 1.1

Класифікація сучасних програмних рішень для контролю особистих завдань

Тип системи	**Основні характеристики**	**Приклади**
Персональні трекери	Орієнтовані на одного користувача, підтримка нагадувань, міток, календаря	Todoist, Google Tasks, TickTick
Командні системи	Підтримка спільної роботи, ролей, дедлайнів, комунікації, діаграм	Jira, Asana, ClickUp
Універсальні платформи	Поєднання особистого й командного функціоналу, масштабованість	Notion, Trello, Monday.com
Веб-додатки	Доступ з браузера, платформа-незалежність	Todoist Web, Trello
Мобільні застосунки	Портативність, сповіщення, офлайн-режим	TickTick, Microsoft To Do
Настільні додатки	Повний функціонал, локальне збереження	Microsoft Planner, Notion
Хмарні сервіси	Синхронізація, доступ з різних пристроїв	Google Tasks, ClickUp
Відкриті системи	Можливість модифікації, створення власних модулів	OpenProject, self-hosted Trello

1.2 Аналіз потреб користувача та постановка проблеми

Планування завдань є базовою навичкою для досягнення особистої та професійної ефективності, проте на практиці користувачі стикаються з рядом труднощів, які знижують результативність і ускладнюють досягнення поставлених

цілей. Однією з найпоширеніших проблем є переоцінка власних можливостей і недооцінка часу, необхідного на виконання конкретного завдання. Користувачі часто планують більше, ніж реально здатні виконати в межах доби або тижня, що призводить до перенесення завдань, відчуття провалу та зниження мотивації. Невміння точно оцінити обсяг і складність роботи пов'язане з недостатнім досвідом, відсутністю звички до структурного аналізу завдань і небажанням враховувати непередбачувані фактори. Другою поширеною проблемою є відсутність чітких пріоритетів. Багато користувачів створюють довгі списки справ без ранжування за важливістю або терміновістю. У результаті відбувається виконання найлегших або найприємніших завдань замість тих, які справді мають значення для досягнення мети. Це явище, яке часто називають “помилкою ефективності”, коли зусилля витрачаються на другорядне, а головне відкладається або залишається невиконаним. Відсутність навички пріоритезації, неясне формулювання мети та невизначеність бажаного результату — основні чинники цієї проблеми. Не менш значущою є проблема недостатньої гнучкості плану. У багатьох користувачів графіки надто жорсткі, вони не враховують можливі зміни в обставинах, що робить такі плани малореалістичними. У разі порушення одного дедлайну або виникнення форс-мажору структура плану “руйнується”, і людина не має механізму корекції [32].

Ще одна суттєва складність полягає у відсутності системного підходу до планування. Користувачі можуть використовувати різні інструменти, але без узгодженості — наприклад, один календар для роботи, інший для особистих справ, нотатки в месенджерах або на папері. Така фрагментованість знижує контроль над завданнями, сприяє пропускам і дублюванню, а також ускладнює аналіз виконаного. Не менш важливою проблемою є недотримання регулярного перегляду та оновлення планів. Користувачі часто створюють списки завдань “раз і назавжди”, не переглядаючи їх у динаміці, не враховуючи зміни в пріоритетах чи нові задачі. У таких випадках плани швидко втрачають актуальність, а мотивація до їх дотримання знижується. Крім того, типові труднощі стосуються і самої постановки завдань. Нерідко задачі формулюються занадто загально або абстрактно, без

визначення конкретного результату, терміну, відповідального, що унеможливлює контроль виконання. Наприклад, замість “підготувати доповідь” ефективніше формулювати завдання як “зібрати джерела для доповіді до 12:00 середи, написати чернетку до п’ятниці, зробити презентацію до понеділка”. Такі труднощі пов’язані з нерозумінням принципів SMART-постановки цілей та відсутністю досвіду в розбиванні великих задач на малі кроки. Ще одна важлива проблема — це перешкоди з боку зовнішнього середовища: відсутність тиші, перевантаження інформацією, часті повідомлення, соціальні мережі, інші фактори, що відволікають. Навіть за наявності плану користувач не здатен його реалізувати, оскільки не створено належного середовища, не передбачено часових блоків для глибокої роботи [6].

Крім того, користувачі часто не мають чіткої системи зворотного зв’язку — відсутність оцінювання прогресу, аналізу невиконаного, рефлексії щодо ефективності плану призводить до повторення одних і тих самих помилок. Інструменти, які не надають статистики, діаграм прогресу, нагадувань або можливості швидко переглянути історію завдань, не сприяють формуванню планувальної дисципліни. І, нарешті, важливим психологічним бар’єром у плануванні є страх перед завданнями, відкладання важкого, відчуття перевантаження, які сприяють прокрастинації. У таких випадках навіть найбільш технологічні інструменти стають безсилими, якщо користувач не має внутрішньої мотивації, звички поступового руху до мети, чіткого розуміння важливості задач. Усі ці проблеми вказують на те, що успішне планування завдань — це не лише питання інструменту, а й культури, навичок, звичок і середовища, в якому працює людина. Ефективна система планування має бути гнучкою, інтегрованою, візуально зрозумілою, мати систему нагадувань, аналітики й бути адаптованою під індивідуальні особливості користувача. Розробка програмного забезпечення для контролю особистих проєктів і завдань має враховувати ці типові труднощі, пропонуючи користувачам інструменти для подолання перешкод, формування стратегії планування, підвищення самодисципліни та реалізації особистих і професійних цілей [5].

У сучасному цифровому середовищі очікування користувачів від систем управління завданнями й проєктами формуються під впливом зростаючих вимог до персональної продуктивності, багатозадачності, зручності у використанні та адаптивності інтерфейсу. Користувачі очікують, що програмні рішення для планування будуть не лише технічно досконалими, а й інтуїтивно зрозумілими, доступними на різних платформах, швидкими у роботі та здатними враховувати індивідуальні особливості ведення справ. Однією з головних вимог є простота та зручність інтерфейсу — система повинна мати логічну навігацію, мінімальну кількість дій для виконання типових операцій і чітко структуровану інформацію. Інтерфейс має бути зрозумілим з першого погляду, не перевантаженим графічними елементами, і водночас здатним візуалізувати складні процеси у простій формі. Наприклад, завдання повинні бути легко створюваними, редагованими, переміщуваними між категоріями, а зміна статусу чи пріоритету має здійснюватися одним натиском. Для багатьох користувачів важливою є можливість кастомізації: змінювати кольори, назви статусів, створювати шаблони завдань, адаптувати інтерфейс під свій стиль роботи. Сучасний користувач не хоче підлаштовуватись під інструмент — він очікує, що інструмент підлаштується під нього [8].

Ще одним ключовим очікуванням є мультиплатформенність і синхронізація в реальному часі. Більшість користувачів працюють у кількох середовищах одночасно — комп’ютер, смартфон, планшет — і очікують, що всі дані будуть миттєво доступні на будь-якому пристрої. Будь-які зміни в завданні, зроблені на телефоні, мають з’явитися на десктопі без затримки. Важливою є також офлайн-функціональність: користувачі хочуть мати змогу переглядати та змінювати свої плани навіть за відсутності інтернету, з подальшою автоматичною синхронізацією після відновлення з’єднання. Крім того, сучасна система управління має бути інтегрованою з іншими популярними сервісами — календарями (Google Calendar, Outlook), хмарними сховищами (Google Drive, Dropbox), месенджерами (Slack, Telegram), засобами відеозв’язку (Zoom, Teams), а також з іншими сервісами продуктивності. Завдяки таким інтеграціям користувач отримує єдину екосистему, в якій немає необхідності дублювати інформацію вручну. Очікуваним стає й

наявність мобільних сповіщень, нагадувань, функції автоматичного перенесення прострочених завдань, інтелектуальних порад на основі поведінки користувача або шаблонів [2].

Значна частина користувачів очікує від системи можливості гнучкого планування з урахуванням як довгострокових, так і короткострокових цілей. Система має надавати змогу створювати ієрархію цілей: від глобальних проєктів до щоденних справ, а також допомагати розбивати завдання на дрібніші кроки. Очікується підтримка різних форматів представлення інформації — списків, календарів, канбан-дошок, діаграм Ґанта. Для користувача важливо бачити як загальну картину, так і деталі. Наприклад, можливість фільтрувати завдання за категоріями, пріоритетами, виконавцями, а також переглядати виконані завдання і отримувати статистику за тиждень або місяць. Очікуваною є також функція автоматичного перенесення невиконаних задач, повторюваних завдань (щодня, щотижня), зв’язку між завданнями та залежностей, що дозволяє будувати реалістичні сценарії. Системи, які не враховують цих очікувань, швидко втрачають актуальність, оскільки не відповідають темпу життя та вимогам користувачів [1].

Не менш важливим є безпека даних — користувачі очікують, що інформація буде захищена, шифрована, не передаватиметься третім особам, а доступ до неї буде здійснюватися лише за допомогою авторизованого входу. Поширеним очікуванням є наявність резервного копіювання, двофакторної автентифікації, можливості керування доступом для окремих пристроїв або сеансів. Сучасні користувачі дедалі частіше звертають увагу на прозорість обробки даних, наявність політики конфіденційності та можливість видалити всі дані за запитом. Окрім цього, очікується наявність розумної системи сповіщень — не надто нав’язливої, але й не надто пасивної. Ідеально, коли користувач сам може налаштувати, які повідомлення він хоче отримувати, в якій формі та коли.

Деякі користувачі, особливо ті, хто прагне глибокого аналізу своєї ефективності, очікують від системи аналітики: діаграм прогресу, підрахунку виконаних задач, порівняння запланованого і фактичного часу, оцінки завантаження за днями чи тижнями. Це допомагає не лише контролювати процес, а й

покращувати його, виявляти слабкі місця, оптимізувати підхід до планування. Інші користувачі цінують гейміфікацію — можливість отримувати досягнення, рівні, візуальні нагороди за регулярність, виконання складних задач чи безперервні дні продуктивності. Такі елементи підвищують залучення, особливо у молодіжній аудиторії, і сприяють формуванню корисних звичок[15].

Проблеми масштабованості та персоналізації сучасних систем управління завданнями є одними з ключових викликів, з якими стикаються як розробники, так і користувачі таких рішень. Масштабованість у цьому контексті розглядається як здатність програмного забезпечення ефективно працювати зростаючим обсягом даних, користувачів, проєктів і завдань без втрати продуктивності, стабільності чи зручності. У той час як початкові версії систем зазвичай розраховані на невелику кількість активностей, розширення до рівня командної, міжкомандної чи корпоративної роботи часто виявляє критичні обмеження. Це можуть бути повільна обробка запитів, затримки у відображенні змін, перевантаження бази даних, складність навігації при великій кількості проєктів, або навіть втрата даних. У більшості випадків користувачі, які починають із безкоштовної версії системи для особистих потреб, згодом стикаються з бар’єрами при спробі адаптувати інструмент до роботи в групі чи при збільшенні обсягу задач. Це викликає фрустрацію, змушує шукати альтернативи або повністю змінювати платформу, що не лише неефективно, а й небезпечно з огляду на можливу втрату інформації. Тому проблема масштабованості повинна вирішуватися на етапі проєктування архітектури системи, яка має бути модульною, підтримувати незалежну обробку компонентів, ефективні методи кешування, асинхронну обробку даних і розподілену логіку. Важливо, щоб система не просто могла "пережити" зростання кількості користувачів і проєктів, а залишалася при цьому такою ж швидкою, зручною та надійною, як і на початкових етапах її використання [20].

Другий важливий аспект — персоналізація, тобто здатність системи адаптуватися до індивідуальних потреб конкретного користувача. Це не обмежується лише вибором теми оформлення чи зміною мови інтерфейсу. Справжня персоналізація включає можливість налаштування структури даних,

створення власних категорій, етапів завдань, фільтрів, шаблонів, а також управління інформаційними потоками відповідно до особистого стилю роботи. У реальності більшість сучасних систем або надмірно ускладнені в цьому плані, або надто спрощені — вони або пропонують жорстко задану логіку взаємодії, або ж вимагають від користувача знань програмування чи складних налаштувань. Це створює ситуацію, в якій система не відповідає очікуванням: надто складна — лякає новачків, надто проста — обмежує досвідчених користувачів. Ще складніше стає тоді, коли персоналізація повинна співіснувати з масштабованістю: наприклад, коли кожен учасник команди хоче бачити завдання у своєму форматі, а система повинна зберігати єдину логіку структури даних і взаємодії між модулями. У таких випадках критичну роль відіграє рольова модель доступу, гнучкість API, підтримка умовної логіки у відображенні інформації та здатність кожного користувача формувати власний дашборд із релевантними віджетами [9].

Слід також звернути увагу на проблему збереження персоналізації при оновленнях або переході на інші версії продукту. У багатьох випадках користувацькі налаштування "скидаються" після масштабних змін інтерфейсу або при переході на командний план підписки. Це демотивує користувачів, змушує витрачати час на повторну адаптацію й викликає недовіру до платформи як до стабільного інструменту роботи. Ідеальна система повинна мати механізм збереження персональних налаштувань незалежно від оновлень, або принаймні можливість експортувати й імпортувати конфігурації. Ще одна важлива проблема — це взаємодія між персоналізацією та користувацьким досвідом. Якщо система дозволяє гнучке налаштування, але при цьому має недостатньо зрозумілу документацію, інтуїтивність падає, а потенціал налаштувань втрачається. І навпаки: надмірна спрощеність позбавляє можливості адаптувати систему під специфіку діяльності. Особливо це актуально для користувачів, які займаються складною багатофакторною діяльністю — наприклад, веденням кількох проєктів одночасно, кожен із яких має іншу логіку структури, пріоритетів, термінів і ролей. У таких випадках персоналізація має бути не просто бажаною функцією, а обов’язковим елементом для ефективної роботи [3].

Таблиця 1.2

Типові проблеми користувачів та потреби

Категорія труднощів	**Суть проблеми**	**Необхідні функціональні рішення**
Переоцінка можливостей	Планування надміру завдань, недотримання термінів	Система прогнозу, реалістичні оцінки
Відсутність пріоритетів	Виконання другорядних задач замість ключових	Пріоритезація, кольорове кодування
Жорсткість плану	Неможливість адаптувати графік до змін	Підтримка гнучких дедлайнів
Фрагментованість інструментів	Календар окремо, нотатки окремо, нагадування в іншій системі	Єдина платформа, об’єднаний інтерфейс
Відсутність зворотного зв’язку	Немає аналізу прогресу, статистики, історії	Вбудована аналітика, діаграми, трекінг виконання
Відсутність мотивації	Відкладання завдань, прокрастинація	Гейміфікація, нагадування, поради
Відсутність інструментів самоаналізу	Немає звітів, динаміки продуктивності	Графіки, звіти, порівняння план/факт

1.3. Формулювання цілей та функціональних вимог до системи

Загальна мета створення системи управління особистими проєктами та завданнями полягає в розробці інтуїтивно зрозумілого, функціонально насиченого та технологічно ефективного веб-додатку, який дозволяє користувачу організовано планувати свою діяльність, відстежувати виконання завдань, контролювати хід реалізації цілей та приймати зважені рішення щодо навантаження і ризиків. Така система повинна виступати не просто цифровим блокнотом для збереження списку справ, а повноцінним середовищем керування особистим або командним часом, інструментом прийняття рішень, який допомагає не лише структурувати

інформацію, а й сприяє підвищенню ефективності діяльності. У сучасному світі, де дедалі більше людей одночасно реалізують кілька проєктів, працюють у гнучкому або дистанційному форматі, навчаються й виконують побутові обов'язки, потреба в такому цифровому інструменті стала не лише актуальною, а й базовою. Багато користувачів стикаються з проблемою розпорошення інформації — завдання зберігаються в різних додатках, нотатках, месенджерах, а отже — втрачається цілісне бачення процесу, забуваються дедлайни, дублюються зусилля, знижується рівень самоорганізації. Основна мета нової системи полягає в подоланні цієї фрагментованості за допомогою єдиної платформи, яка забезпечує повний цикл управління: створення, деталізацію, пріоритезацію, призначення виконавців, контроль виконання, оцінку результатів, прогнозування ризиків і формування рекомендацій [7].

У межах цієї загальної мети особливо важливим є створення інструменту, який поєднує простоту для початківців і гнучкість для досвідчених користувачів. Система повинна надавати користувачу можливість почати з мінімального функціоналу — простого списку справ або календарного плану, — а потім поступово відкривати доступ до додаткових можливостей: створення підзавдань, налаштування статусів, використання фільтрів, формування ієрархічних проєктів, інтеграції з іншими сервісами. Такий підхід дозволяє уникнути ситуації, коли новий користувач відчуває себе перевантаженим функціями, а досвідчений — обмеженим. Крім цього, реалізація веб-додатку з використанням технологій Python (Flask) та JavaScript (Vanilla JS) дозволяє досягти технічної універсальності, забезпечити швидку роботу, стабільність, а також відкритість для подальших змін і розширень. Одним із ключових елементів досягнення мети є створення моделі клієнт-серверної архітектури, яка розділяє відповідальність між інтерфейсом користувача, логікою обробки запитів і базою даних. Завдяки цьому забезпечується масштабованість, безпечність і можливість адаптації системи під різні сценарії використання — від особистого трекінгу до роботи в команді.

Загальна мета створення системи також передбачає досягнення балансу між ручним керуванням та автоматизацією. Користувач повинен мати змогу самостійно

керувати своїми завданнями, але при цьому отримувати підказки, нагадування, візуальні сигнали про ризики або перевантаження. Саме тому система повинна містити алгоритм прогнозування ризиків, який аналізує дедлайни, кількість активних задач, участь користувача в різних проєктах і пропонує рекомендації для оптимізації навантаження. У такий спосіб реалізується ідея “розумного” планувальника, який не лише фіксує дії користувача, а й активно допомагає у прийнятті ефективних рішень. Не менш важливою складовою мети є забезпечення безпеки — усі дані користувача повинні зберігатися в захищеній базі даних, бути доступними лише після авторизованого входу через JWT-токени, а доступ до критичних функцій має бути обмеженим відповідно до ролі. Усі ці аспекти разом формують загальну мету як створення адаптивного, масштабованого, інтуїтивного та функціонально повного програмного продукту, який стане ефективним цифровим інструментом для керування особистими проєктами та завданнями [4].

Функціональні вимоги до клієнтської та серверної частини системи управління особистими проєктами та завданнями формуються на основі цілей створення додатку, очікувань користувачів, принципів зручності, безпеки, масштабованості та аналітичної цінності. Клієнтська частина (frontend) має забезпечувати інтерактивний інтерфейс користувача, що дозволяє зручно взаємодіяти з усіма функціями системи: створювати нові проєкти, додавати завдання, змінювати статуси, вибирати виконавців, встановлювати дедлайни, переглядати хід виконання, отримувати повідомлення про прострочені задачі або можливі ризики. Головна функція клієнтської частини — зробити всі ці дії доступними за кілька кліків, інтуїтивно зрозумілими та візуально структурованими. Інтерфейс повинен підтримувати кілька форматів перегляду (список, календар, канбан), надавати змогу фільтрації та сортування завдань за пріоритетами, статусами, термінами. Особливо важливо, щоб інтерфейс був адаптивним до різних розмірів екрана, підтримував роботу на мобільних пристроях та мав віджет або сповіщення про наближення важливих дедлайнів. Усі дані мають відображатися в реальному часі, без потреби в перезавантаженні сторінки, що реалізується за рахунок асинхронних запитів до API. Серед обов’язкових функцій клієнтської

частини — реєстрація та вхід користувача, редагування профілю, керування проєктами, додавання учасників, створення завдань, редагування статусів, перегляд рекомендацій із прогнозу ризиків, логування виходу з системи [6].

Серверна частина (backend), у свою чергу, відповідає за обробку всіх запитів із клієнтської частини, реалізацію бізнес-логіки, взаємодію з базою даних, авторизацію, а також за підтримку безпеки та ефективності системи. До основних функціональних вимог до серверної частини належить реалізація RESTful API з відповідними маршрутами для автентифікації (реєстрація, вхід, перевірка токену), управління проєктами (отримання, створення, редагування, видалення), управління завданнями (додавання, зміна статусу, видалення, встановлення дедлайну), управління учасниками проєктів (додавання, перегляд, видалення), а також обробка запитів до модуля прогнозування ризиків. Важливо, щоб серверна частина забезпечувала шифрування паролів (наприклад, за допомогою bcrypt), генерувала JWT-токени, перевіряла права доступу на кожен запит і не дозволяла неавторизованим користувачам отримувати доступ до даних інших осіб. База даних повинна підтримувати чітку структуру: таблиці користувачів, проєктів, завдань, ролей, учасників, прогнозів ризику. Система має бути здатна до горизонтального масштабування, тобто ефективно обробляти більшу кількість одночасних запитів при зростанні кількості користувачів. Також необхідною є реалізація механізмів кешування, обмеження частоти запитів (rate limiting), обробки помилок та логування подій [5].

Усі зміни в системі повинні фіксуватись у базі даних із прив’язкою до користувача та часу дії, що дозволяє реалізувати історію змін і зворотний зв’язок. Серверна частина також має підтримувати взаємодію з модулем аналізу навантаження і дедлайнів — за запитом з frontend сервер передає відповідний набір даних, а отримує рівень ризику і рекомендацію, яку надсилає назад до клієнтської частини. Особливу увагу слід приділити захисту даних: всі паролі мають зберігатися в зашифрованому вигляді, всі запити повинні здійснюватися через HTTPS, а токени — мати термін дії та оновлюватися з використанням refresh-токенів. Функціональні вимоги також передбачають можливість адміністрування

системи, вивантаження даних, резервне копіювання, підготовку до багатокористувацької взаємодії, що стане у пригоді при масштабуванні. Тестування всіх функцій системи повинно бути частиною реалізації як у клієнтській, так і в серверній частині, з використанням unit- та integration-тестів для перевірки коректності логіки. У підсумку функціональні вимоги мають гарантувати безперервність, стабільність і захищеність роботи всієї системи, що є основою для досягнення головної мети — створення ефективного програмного інструменту для контролю особистих проєктів та завдань [8].

У контексті сучасного управління особистими проєктами та завданнями реалізація модуля прогнозування ризиків є не просто додатковою функцією, а необхідною складовою ефективної цифрової системи, що прагне не лише фіксувати та структурувати діяльність користувача, а й активно допомагати в ухваленні рішень та забезпеченні результативності. Користувачі, які одночасно реалізовують кілька завдань, беруть участь у різних проєктах або взаємодіють із іншими учасниками, часто втрачають здатність своєчасно оцінювати власне навантаження, контролювати дедлайни або передбачати наслідки відкладання дій. Це призводить до того, що важливі завдання виконуються із запізненням або залишаються незавершеними, що у свою чергу знижує продуктивність, породжує стрес і знижує довіру до самої системи управління. У таких випадках прогнозування ризиків виконує роль “інтелектуального помічника”, який виявляє потенційні проблеми ще до того, як вони стануть критичними, і дозволяє користувачеві своєчасно вжити заходів для оптимізації плану. Модуль ризик-аналізу має ґрунтуватися на аналізі кількох ключових факторів — кількість активних завдань користувача, щільність розкладу, наближення термінів виконання, середній час завершення подібних задач, кількість одночасних участей у різних проєктах, попередні тенденції щодо прострочення дедлайнів тощо [9].

Реалізація модуля прогнозування ризиків у системі управління особистими завданнями дозволяє перейти від реактивної моделі взаємодії до проактивної, де система не просто реагує на дії користувача, а самостійно ініціює попередження або рекомендації, базуючись на поточному стані планування. Це підвищує довіру

до системи, сприяє формуванню звички до регулярного аналізу, дозволяє користувачеві планувати час з урахуванням потенційних “вузьких місць” і приймати обґрунтовані рішення щодо зміни пріоритетів. Наприклад, якщо система виявляє, що на найближчий тиждень заплановано надмірну кількість задач із високим пріоритетом, вона може попередити користувача та запропонувати перенесення частини з них або делегування. Або якщо деяке завдання вже неодноразово переносилося, а його дедлайн наближається, система може підсвітити його як критичне. Такий підхід дозволяє уникати накопичення “прострочених” завдань і формувати реалістичні очікування щодо виконання. Більше того, модуль ризик-аналізу може враховувати не лише технічні параметри, а й поведінкові патерни користувача — наприклад, час активності, звичку виконувати задачі в певні години, ймовірність завершення завдань залежно від їх типу або контексту. Це створює умови для персоналізованого прогнозу, який підвищує точність аналізу та довіру до системи [4].

Із технічної точки зору, реалізація модуля прогнозування ризиків передбачає побудову окремого програмного компонента, який отримує на вхід структуровані дані про активність користувача, обробляє їх за заданими алгоритмами і повертає індикатори ризику разом із текстовими порадами. Для початкового етапу достатньо реалізувати логіку на основі правил (rule-based system), коли кожна умова, наприклад, “завдання з пріоритетом високий і строком менше 24 годин без виконавця” автоматично помічається як ризиковане. У перспективі цей модуль можна розширити із застосуванням елементів машинного навчання, які дозволяють виявляти складніші залежності, формувати прогнози на основі поведінки тисяч користувачів і вдосконалювати точність аналізу. Модуль повинен бути повністю ізольованим від інших частин системи, працювати незалежно, але мати доступ до бази даних або до API, щоб отримувати потрібну інформацію. Після обробки запиту клієнтська частина повинна відображати результат у зручній візуальній формі — це може бути індикатор рівня ризику (низький, середній, високий), список критичних задач, рекомендації щодо перерозподілу ресурсів або перегляду дедлайнів. Дуже важливо, щоб інформація, яку надає модуль, не виглядала як “покарання”, а

сприймалася як корисна порада — саме тому інтерфейс повинен подавати її м’яко, із акцентом на підтримку, а не на тиск [9].

У системах без модуля прогнозування користувач змушений постійно тримати в голові всі дедлайни, аналізувати, що з чим перетинається, що може спричинити затримки, і самостійно приймати рішення. Це не тільки втомлює, а й створює постійну напругу. Натомість модуль прогнозу ризиків бере на себе частину когнітивного навантаження, виступає як елемент цифрової підтримки та забезпечує профілактику проблем. Це особливо актуально в умовах високої динаміки життя, коли завдань стає більше, ніж фізично можна контролювати без допомоги алгоритмів. Впровадження цього модуля також є конкурентною перевагою, адже більшість існуючих систем управління завданнями пропонують лише функції нагадування або дедлайну без глибшого аналізу. Таким чином, модуль прогнозу формує додану вартість продукту, вирізняє його на тлі аналогів і підвищує цінність для кінцевого користувача.

У підсумку, необхідність реалізації модуля прогнозування ризиків визначається не лише поточними потребами користувачів, а й глобальним вектором розвитку цифрових інструментів — від фіксації даних до інтелектуального супроводу. Такий модуль виконує функцію аналітичного ядра, допомагає в управлінні ресурсами, покращує планування та сприяє досягненню цілей у визначені терміни. Він знижує ризик перевантаження, формує довіру до системи, розвиває навички усвідомленого керування часом і підвищує загальну якість цифрового досвіду користувача. Його впровадження є важливою інновацією, яка поєднує технологічний потенціал, практичну доцільність та реальну користь у повсякденному житті [36].

Таблиця 1.3

Формулювання цілей та функціональних вимог до системи

Категорія цілей / вимог	**Зміст**
Загальна мета	Створення адаптивної системи для управління особистими завданнями з елементами аналітики
Архітектурна модель	Клієнт-серверна структура, Flask (бекенд), Vanilla JS (фронтенд), MySQL (база даних)
Гнучкість	Початковий мінімум функцій + поступове розширення (шаблони, теги, статуси, ролі)
Аналітична складова	Прогнозування ризиків, аналіз навантаження, рекомендації
Безпека	JWT, хешування паролів, перевірка доступів, HTTPS
Основні функції (frontend)	Реєстрація, створення проєктів, задач, дедлайни, статуси, вивід ризиків, зміна ролей
Основні функції (backend)	REST API: auth, projects, tasks, risks; валідація, логіка, авторизація
Адаптивність	Підтримка мобільних пристроїв, відображення в реальному часі
Перспективи розширення	Інтеграція з календарями, email-сповіщення, ML-підказки, мікросервіси

2. ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1. Архітектура програмної системи

Архітектура розробленої системи управління особистими проєктами та завданнями реалізована за моделлю класичної трирівневої клієнт-серверної архітектури, яка включає клієнтську частину (frontend), серверну частину (backend) і базу даних (database). Кожен рівень виконує окремі функції: клієнт забезпечує інтерфейс взаємодії з користувачем, сервер відповідає за обробку запитів і логіку, а база даних зберігає всю інформацію про користувачів, проєкти, завдання та пов’язані з ними об’єкти. На практиці клієнтська частина реалізована з використанням HTML, CSS та Vanilla JavaScript. Наприклад, у файлі project.js реалізовано функцію loadProjects(), яка за допомогою fetch-запиту звертається до API /projects/ та відображає список проєктів у HTML-контейнері.

Ця функція показує, як клієнт надсилає запит, отримує JSON-відповідь із серверної частини і виводить проєкти на сторінку. Усі запити захищено токенами JWT, які користувач отримує після автентифікації. Інтерфейс адаптований під браузерну взаємодію — наприклад, у dashboard.html реалізовані поля для створення проєкту та завдань, списки учасників, елементи формування дедлайнів. Сторінка логіну/реєстрації (index.html) передбачає введення email і пароля, які потім надсилаються у вигляді JSON-об’єкта у auth.js.

Серверна частина побудована на Flask — легкому та швидкому веб-фреймворку на Python. У головному файлі app.py відбувається ініціалізація серверу, імпорт конфігурації, підключення JWT, CORS і маршрутизації. Зокрема,через app.register_blueprint() підключаються модулі: auth_routes, project_routes, task_routes, risk_routes, user_routes.Це дозволяє чітко розділити логіку за функціональними блоками. Наприклад, у project_routes.py реалізовано маршрут @project_bp.route('/', methods=['GET']), який дозволяє авторизованому користувачу

отримати список проєктів, до яких він має доступ. Цей маршрут перевіряє токен, отримує ID користувача за допомогою get_jwt_identity(), формує SQL-запит і повертає JSON-відповідь.

Отже, backend виконує функцію бізнес-логіки, керує валідацією даних, обробляє помилки, взаємодіє з базою даних, захищає API та реалізує всі необхідні сервіси.

База даних реалізована на MySQL та містить основнітаблиці: users, projects, tasks, project_members. Робота з БД здійснюється через бібліотеку flask_mysqldb. Наприклад, для створення нового користувача в user_model.рувикористовується SQL-запит.

Аналогічно, при створенні проєкту або завдання система звертається до таблиць projects або tasks, використовуючи SQL-інструкції INSERT, SELECT, DELETE.

Це дозволяє зберігати інформацію про всі створені задачі, включно з дедлайнами, описами, виконавцями та статусами. У структурі бази також реалізовано таблицю project_members, яка дозволяє організовувати командну роботу: кожен проєкт може мати багато учасників.Крім основної логіки, в системі реалізовано модуль прогнозування ризиків (risk_routes.py + services/risk_analyzer.py). Він виконує аналіз дедлайнів та кількості завдань у кожному проєкті.

Система оцінює, чи перевищено допустиме навантаження, та повертає рівень ризику (низький, середній, високий) разом із рекомендаціями. Відповідь відображається на фронтенді через risk.js у елементі #risk-result.

Завдяки такій архітектурі — де кожна частина (frontend, backend, database) виконує чітко визначені функції — система є гнучкою, модульною, зручною для масштабування та підтримки. Весь код організований у відповідні папки: routes/ містить API-маршрути, models/ — роботу з базою, services/ — логіку аналізу ризиків. Інтерфейс системи побудований з урахуванням зручності користувача, підтримує аутентифікацію, динамічне оновлення контенту, виведення повідомлень і модифікацію об’єктів у реальному часі. Така архітектура не лише дозволяє

ефективно працювати з особистими проєктами, а й відкриває перспективи для подальшого розвитку системи: інтеграції зі сторонніми сервісами, додавання командного чату, календаря, email-нагадувань або штучного інтелекту для автоматизації планування.

В основі ефективної роботи розробленої системи управління особистими проєктами та завданнями лежить чітко налагоджений взаємозв'язок між трьома основними компонентами: клієнтською частиною (frontend), серверною логікою (backend) та базою даних (database). Ці частини функціонують за принципом запит–обробка–відповідь. Фронтенд ініціює запити через JavaScript, передає їх до бекенду через HTTP-протокол, сервер обробляє ці запити за допомогою Flask, звертається до бази даних MySQL, отримує або змінює дані й повертає відповідь назад на клієнт. Розглянемо приклад взаємодії під час створення нового проєкту. У файлі project.js реалізована функція addProject(), яка активується після натискання кнопки "Додати проєкт" на сторінці dashboard.html. Вона зчитує дані з полів форми (назва та опис проєкту), формує JSON-об'єкт і відправляє POST-запит на бекенд.

Цей запит надходить на маршрут /projects/, який обробляється у project_routes.py функцією add_project(). Тут виконується валідація, перевірка автентифікації через JWT, а далі — вставка даних до бази.Після успішного виконання SQL-запиту бекенд повертає JSON-відповідь з повідомленням, яке потім відображається у frontend.

Таким чином, користувач бачить результат своєї дії, що фактично відбулося через повний цикл взаємодії між трьома шарами архітектури.

Аналогічна схема працює при завантаженні списку завдань. Клієнтський файл task.js містить функцію loadTasks(), яка надсилає GET-запит.На сервері запит потрапляє до task_routes.py у функцію get_tasks(), яка звертається до таблиці tasks в базі MySQL.Після виконання SELECT-запиту дані конвертуються у список JSON-об'єктів і повертаються назад. Фронтенд приймає масив завдань і за допомогою JavaScript вставляє його у DOM-структуру сторінки.

Така реалізація дозволяє забезпечити повну інтеграцію між інтерфейсом користувача та фактичними даними в базі.

Ще одним прикладом ефективної взаємодії компонентів є функція оновлення статусу завдання. На фронтенді у task.js кожен елемент має селектор статусу. Після вибору нового статусу викликається функція updateTaskStatus(), яка відправляє PUT-запит на сервер.На стороні сервера функція update_task_status() оновлює відповідне поле в базі даних.У результаті користувач бачить новий статус завдання в інтерфейсі — дані з бази оперативно змінено й оновлено на клієнті без перезавантаження сторінки.

Особливої уваги заслуговує взаємодія при аналізі ризиків. Користувач обирає проєкт зі списку, натискає “Аналізувати”, і risk.js надсилає запит до API /risks/{project_id}.Файл risk_routes.py викликає функцію RiskAnalyzer.analyze_risks(project_id), яка на основі запитів до бази визначає співвідношення кількості завдань, термінів і навантаження, після чого формує відповідь.Результат відображається на сторінці у блоці #risk-result, що показує прямий зворотний зв’язок користувача з сервером та даними. Уся система працює на узгодженості: frontend забезпечує введення/виведення, backend — логіку та обробку, а база даних — зберігання, що в сукупності створює ефективний і практичний цифровий інструмент.

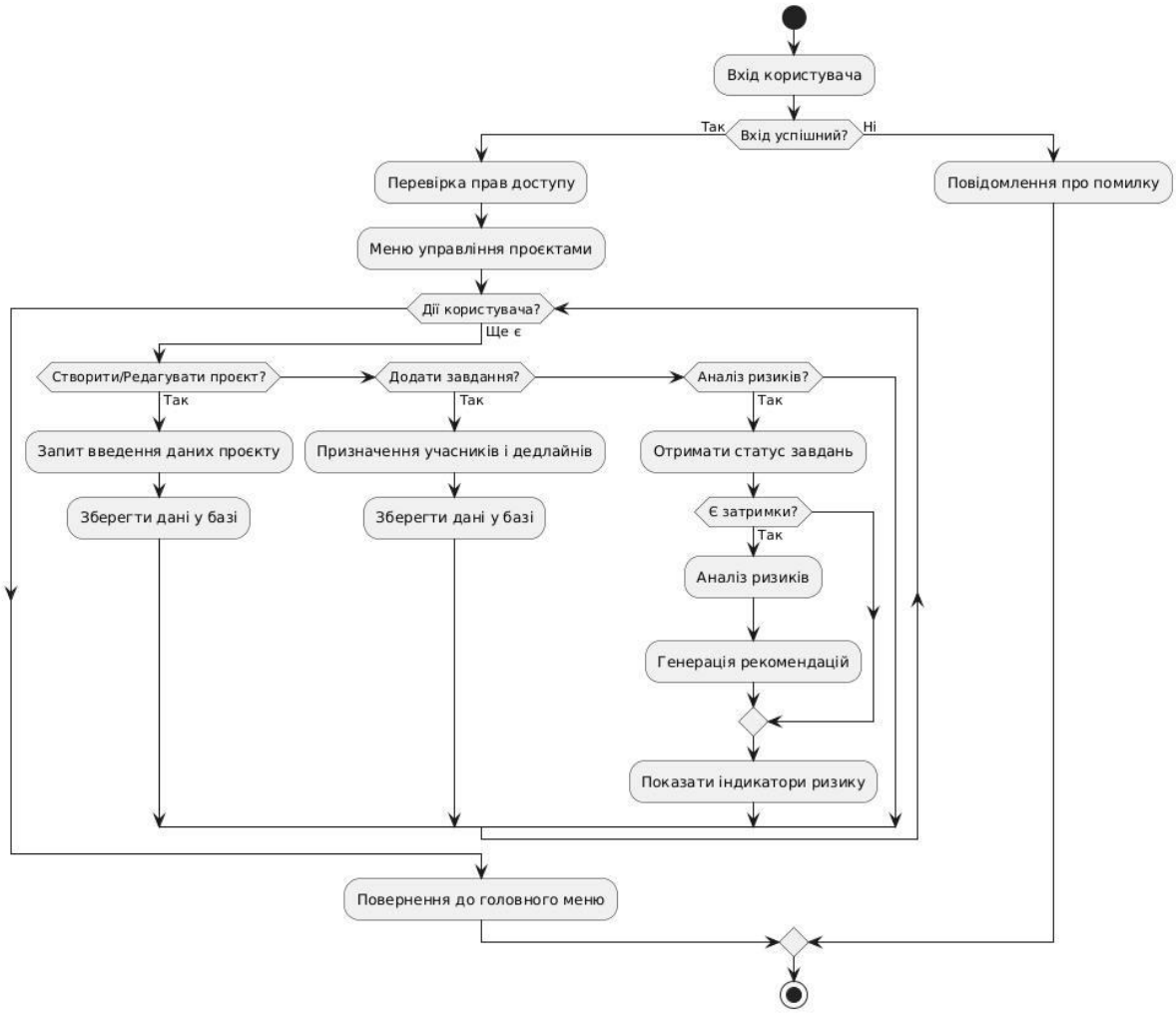


Рис. 2.2 Алгоритм роботи програми

2.2. Вибір технологій та інструментів реалізації

У рамках розробки веб-системи для контролю особистих проєктів та завдань основним інструментом реалізації серверної логіки став мікрофреймворк Flask, написаний мовою Python. Flask був обраний завдяки своїй гнучкості, простоті структурування застосунку, підтримці RESTful API, інтеграції з базами даних, JWT-автентифікації та широкій спільноті розробників. Архітектура серверної частини базується на модульному принципі: окремі логічні частини — аутентифікація, управління проєктами, завданнями, користувачами та прогнозування ризиків — винесені у відповідні файли з маршрутизацією, які підключаються до основного застосунку через механізм Flask Blueprints.

Така структура дозволяє підтримувати чистоту коду, забезпечити незалежність компонентів і спрощує масштабування проєкту.

Один із базових прикладів використання Flask для побудови серверної логіки — реалізація модуля аутентифікації у файлі auth_routes.py. Тут реалізовані два ключові маршрути: /register і /login. Під час реєстрації новий користувач надсилає POST-запит з email, username і паролем. Ці дані обробляються у функції register(), де пароль спочатку шифрується за допомогою bcrypt.generate_password_hash(), а потім записується у базу.

При вході (/login) Flask перевіряє відповідність пароля за допомогою bcrypt.check_password_hash(), після чого генерує JWT-токен для автентифікації.Цей токен додається до всіх подальших запитів користувача і дозволяє захистити API від неавторизованого доступу.

Інший приклад — модуль керування проєктами, реалізований у файлі project_routes.py. За допомогою декоратора @jwt_required() захищається маршрут, що повертає список проєктів користувача. Тут використовуються SQL-запити до таблиць projects і project_members, а результат повертається у форматі JSON.

Flask дозволяє зручно повертати дані з використанням jsonify(), що автоматично формує правильну HTTP-відповідь.Подібним чином організована логіка створення, редагування та видалення проєктів, а також додавання учасників.

Модуль завдань (task_routes.py) реалізує обробку POST-запиту для створення задач, що надходить у форматі JSON з описом, дедлайном, ID користувача і проєкту. Вхідні дані проходять валідацію, а потім записуються до таблиці tasks.

Окремим маршрутом реалізовано оновлення статусу задачі методом PUT. Flask дозволяє легко реалізовувати ці HTTP-методи, вказуючи їх у параметрах декораторів.Це дає змогу побудувати повноцінний REST API, де кожен ресурс обробляється через стандартні методи: GET, POST, PUT, DELETE.

Особливої уваги заслуговує реалізація модуля прогнозування ризиків у risk_routes.py, який демонструє гнучкість Flask при виклику сторонніх сервісів усередині маршруту. У цьому прикладі маршрут /risks/<project_id> обробляє запит і викликає сервісну функцію analyze_risks() із модуля RiskAnalyzer. Ця функція

виконує внутрішній аналіз навантаження, кількості активних завдань і дедлайнів, після чого повертає рекомендацію. Flask дозволяє це реалізувати дуже простоРезультат одразу відображається на фронтенді завдяки швидкому циклу запит–відповідь.

Серверна логіка також включає обробку CORS, яку реалізовано у app.py.Це дає змогу безпечно обмінюватися даними між frontend і backend, що працюють на різних портах. А також реалізовано автоматичне додавання заголовків до всіх відповідей через @app.after_request, що спрощує налагодження взаємодії між компонентами системи.

Загалом, використання Flask дало змогу реалізувати модульну, розширювану й безпечну серверну архітектуру, яка повністю відповідає вимогам до сучасного веб-додатку. Flask чудово інтегрується з MySQL через flask_mysqldb, підтримує JWT-автентифікацію, дає змогу швидко будувати API, обробляти запити й масштабувати систему у разі потреби. У межах цього проєкту серверна логіка є центральним елементом, який забезпечує взаємодію між інтерфейсом і базою даних, реалізує правила доступу, обробку даних, аналіз ризиків та автоматичне оновлення інформації, що надходить від користувача.

Для реалізації динамічного та інтерактивного інтерфейсу в системі управління особистими проєктами та завданнями було обрано чистий JavaScript (Vanilla JS) без використання додаткових фреймворків, таких як React чи Vue. Це дозволило створити легкий, швидкий і повністю контрольований клієнтський шар, який безпосередньо взаємодіє з API серверної частини, обробляє відповіді та оновлює DOM у режимі реального часу. Уся логіка побудована за принципом SPA (Single Page Application) у межах dashboard.html, де динамічно завантажуються проєкти, завдання, учасники, а також виконується аналіз ризиків. Один із найпростіших прикладів динамічного завантаження — функція loadProjects() у файлі project.js. Вона виконує GET-запит до серверного маршруту /projects/, обробляє відповідь у форматі JSON і генерує HTML-елементи для кожного проєкту.

Функція renderProjects() безпосередньо оновлює вміст сторінки, створюючи DOM-елементи на основі отриманих даних:Таким чином, інтерфейс автоматично

оновлюється після кожного запиту, що забезпечує реактивну поведінку без повного перезавантаження сторінки.

Ще одним важливим прикладом є реалізація додавання завдань через файл task.js. Коли користувач натискає кнопку "Додати завдання", функція addTask() зчитує значення з полів форми, формує JSON-об’єкт і надсилає POST-запит до /tasks/.

Після отримання відповіді виконується оновлення списку завдань за допомогою loadTasks(), яка завантажує та відображає всі актуальні задачі користувача.

Vanilla JS дозволяє повністю контролювати структуру HTML-елементів, додавати інтерактивність (наприклад, зміну статусу завдання через селектор), та реалізовувати логіку динамічного оновлення.

Окремо варто звернути увагу на взаємодію з модулем ризик-аналізу. У risk.js реалізовано функцію analyzeRisk(), яка після вибору проєкту користувачем виконує запит до API /risks/{project_id} і отримує прогноз ризику.

Знову спрацьовує принцип асинхронного оновлення сторінки: користувач не перезавантажує сторінку, а бачить результат прямо під формою вибору проєкту. Такий підхід значно підвищує зручність взаємодії, дозволяє швидко отримувати зворотний зв’язок та забезпечує позитивний користувацький досвід.

Для реалізації автентифікації також використано чистий JavaScript у файлі auth.js. Після введення даних користувача викликається login(), яка надсилає POST-запит і зберігає токен у localStorage.

Це дозволяє реалізувати захищену сесію, в якій користувач отримує доступ до всіх ресурсів лише після авторизації. Усі подальші запити до API автоматично включають заголовок Authorization: Bearer ..., який перевіряється на бекенді.

2.3. Розробка інтерфейсу користувача

Під час створення інтерфейсу користувача для системи управління особистими проєктами та завданнями особлива увага була зосереджена на принципах UI (User Interface) та UX (User Experience), що дозволило забезпечити логічну, зручну й візуально привабливу взаємодію з системою. Основним об'єктом проєктування став дашборд — центральна сторінка користувача після входу в систему, реалізована у файлі dashboard.html. Дашборд повинен забезпечувати швидкий доступ до основних функцій: перегляд проєктів, створення нових, керування завданнями, прогнозування ризиків і додавання учасників. Для досягнення цього було застосовано кілька ключових UI/UX-принципів, зокрема мінімалізм, логічна структура, візуальна ієрархія, адаптивність та зворотний зв’язок. У розмітці сторінки використано сітковий підхід із поділом на дві основні колонки — одна відповідає за проєкти, інша — за завдання. Це реалізовано через клас grid-layout, який розміщується в div.page-content, що дозволяє адаптувати компонування під різні екрани.

У файлі dashboard.html для кожного блоку — проєкти, завдання, ризики та учасники — створено окремий div.container з відповідним заголовком і візуальною відокремленістю. Наприклад, блок для створення нового завдання включає підписи до полів, елементи форми (input, select), а також кнопку з чітким текстом дії.

Цей фрагмент демонструє дотримання принципу "одна дія — одна форма" та використання зрозумілої мови інтерфейсу без надмірної термінології. Візуально користувач бачить лише те, що йому потрібно в поточний момент, а решта елементів згруповані за логікою сценарію використання.

З точки зору UX важливо було реалізувати систему повідомлень про результат дії. Наприклад, у project.js після успішного створення проєкту викликається повідомлення. А у випадку помилки — виводиться відповідне пояснення.

Це дозволяє користувачеві миттєво розуміти результат своїх дій, що відповідає принципу надання зворотного зв’язку. Усі форми в інтерфейсі перевіряються на

заповненість перед надсиланням, і якщо поля не заповнено — система сповіщає користувача:

Такий UX-підхід знижує кількість помилок і підвищує загальну зручність використання.

З точки зору дизайну було обрано легку кольорову схему з білим фоном, темним текстом і акцентами у вигляді синього кольору (#3f51b5), який використовується для кнопок, заголовків та активних елементів. Стилі збережені у style-секції HTML і враховують адаптивність до різних екранів.

Блоки відображаються один під одним на мобільних пристроях. Кнопки мають достатній розмір для взаємодії з сенсорним екраном, поля форм — оптимальну ширину, а всі елементи мають відступи для уникнення візуального стиснення.

Ще один приклад реалізації UI/UX — блок прогнозування ризиків. Користувач обирає проєкт зі списку select, натискає кнопку, після чого у блоці #risk-result виводиться рівень ризику й рекомендація.

Результат подається в короткій, лаконічній формі, не перевантажує інтерфейс та відображається без перезавантаження сторінки. Це підвищує швидкість взаємодії, що є одним із ключових чинників у UX-дизайні.

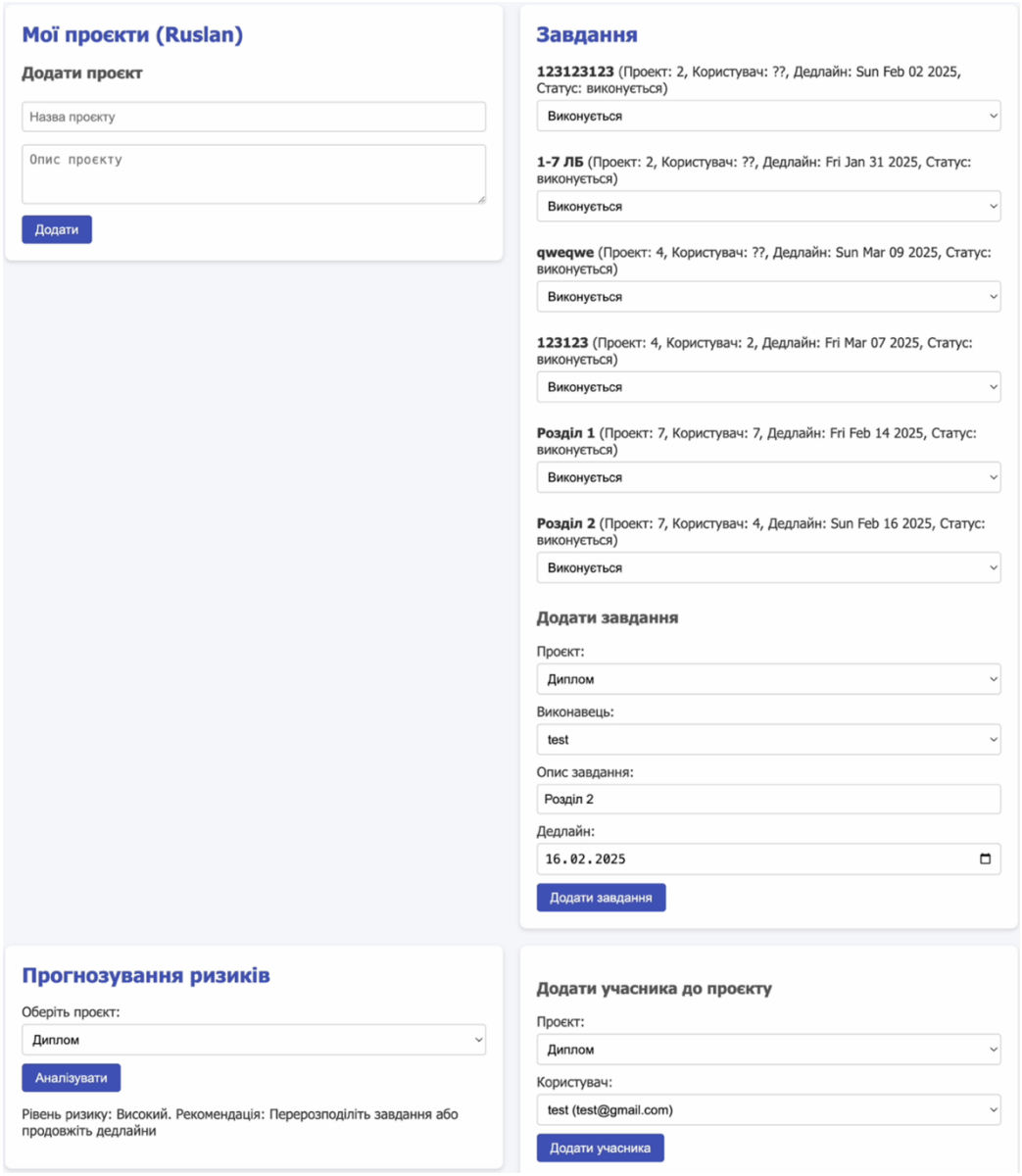


Рис. 2.3 Інтерфейс додатка

У побудові клієнтської частини системи управління особистими проєктами та завданнями ключову роль відіграють механізми виводу інформації та організації взаємодії з користувачем, оскільки саме вони визначають швидкість, зрозумілість та ефективність взаємодії. У практичній реалізації ці механізми реалізовані за допомогою JavaScript (Vanilla JS), DOM-маніпуляцій та асинхронних запитів до API, після чого дані у форматі JSON перетворюються у візуальні елементи сторінки. У файлі project.js функція renderProjects(projects)динамічно формує HTML для виводу списку проєктів. Кожен об’єкт у масиві перетворюється на div з класом project-item, який містить назву, опис і кнопку видалення проєкту.

Цей механізм демонструє повноцінний цикл: запит → отримання даних → генерація HTML-елементів → додавання до DOM. Користувач бачить інформацію без перезавантаження сторінки, що є ключовим для сучасних веб-додатків.

Взаємодія з користувачем також реалізована через елементи форм, кнопки й випадаючі списки. Наприклад, у task.js функція loadTasks() отримує список завдань і динамічно додає HTML-контент.

Користувач одразу бачить опис завдання, його дедлайн, статус та має змогу змінити його через випадаючий список, що одразу надсилає PUT-запит до API. Така реактивна поведінка є прикладом двосторонньої взаємодії між інтерфейсом і сервером — зміни відображаються миттєво, забезпечуючи ефективну роботу з великим обсягом завдань.

Вивід інформації про ризики проєктів реалізований у risk.js. Після обрання проєкту із select користувач натискає кнопку "Аналізувати", після чого функція analyzeRisk() надсилає GET-запит і виводить результат у p#risk-result.

Цей блок динамічно оновлюється при кожному запиті, дозволяючи користувачеві негайно отримати зворотний зв’язок на основі аналітики системи. Для підвищення інформативності можна додати кольорові підсвітки, наприклад, червоний для високого ризику, жовтий — для середнього, зелений — для низького.

Ще одним прикладом динамічної взаємодії є модуль додавання учасників у проєкти. Користувач обирає проєкт і користувача з відповідних select-елементів, після чого натискає кнопку, що запускає addMemberToProject(). Після успішного додавання з’являється повідомлення alert("Учасника додано успішно!"), що підтверджує дію. Перед тим, як додати учасника, система перевіряє, чи обрано проєкт і користувача, — це базова валідація, яка реалізована безпосередньо у JS.

Всі ці приклади демонструють, як механізми виводу інформації та взаємодії з користувачем у нашій системі реалізовано на практиці: через пряме маніпулювання DOM, використання асинхронних запитів до серверу, а також інтерактивні компоненти, що забезпечують повну двосторонню комунікацію між користувачем і системою. У поєднанні це створює динамічний, зручний і продуктивний користувацький досвід.

2.4. Реалізація функціональних модулів

Модуль управління проєктами, учасниками та завданнями є ключовим елементом розробленої системи, оскільки саме через нього користувач здійснює основні дії в системі: створює нові проєкти, додає до них учасників, призначає завдання та керує їхнім виконанням. Практично цей модуль реалізовано у вигляді кількох пов'язаних між собою компонентів як на фронтенді (файли project.js, task.js, dashboard.html), так і на бекенді (маршрути у project_routes.py, task_routes.py, user_routes.py, база даних). Починається усе з взаємодії користувача з інтерфейсом. На сторінці dashboard.html у розділі "Мої проєкти" реалізована форма створення нового проєкту, яка включає поля для введення назви, опису й кнопку, що запускає функцію addProject(). У project.js ця функція збирає дані з форми та надсилає POST-запит до бекенду:

Серверна частина у файлі project_routes.py приймає цей запит у функції add_project() і додає проєкт до бази.

Важливо, що кожен проєкт прив'язаний до користувача, який його створив, і надалі саме цей користувач має повноваження керувати учасниками та задачами в цьому проєкті.

Управління учасниками реалізоване через додаткову функцію addMemberToProject() у project.js. Вона активується після вибору проєкту й користувача зі списків у дашборді. Після натискання кнопки надсилається POST-запит до маршруту /projects/<project_id>/members.

На сервері project_routes.py приймає цей запит у функції add_project_member(), яка перевіряє, чи існує такий користувач, чи не доданий він уже до проєкту, і після цього додає запис у таблицю project_members.Таким чином, формується зв'язок між користувачами й проєктами, що дозволяє надалі призначати завдання будь-кому з учасників.

Модуль завдань (файл task.js) дозволяє створювати, редагувати та змінювати статус завдань. Коли користувач натискає "Додати завдання", запускається функція

addTask(), яка зчитує ID проєкту, ID виконавця, опис і дедлайн, після чого формує POST-запит. На бекенді у файлі task_routes.py дані приймаються у функції add_task() і записуються в таблицю tasks.

Кожне завдання містить статус, який можна змінити з інтерфейсу через випадаючий список. Відповідна функція updateTaskStatus(taskId, newStatus) надсилає PUT-запит, а сервер оновлює поле status в таблиці.

Для зручності користувача вивід завдань реалізовано в структурованому вигляді — з відображенням опису, виконавця, проєкту, дедлайну та статусу, який можна змінити без оновлення сторінки.

Отже, модуль управління проєктами, учасниками та завданнями є повністю реалізованим функціональним блоком, який забезпечує повний життєвий цикл роботи користувача в системі. Завдяки взаємодії між dashboard.html, project.js, task.js, а також відповідними маршрутами бекенду й базою даних, користувач може створювати нові проєкти, додавати до них учасників, призначати задачі, контролювати дедлайни, змінювати статуси та бачити зміни в реальному часі. Цей модуль є основою всієї системи і забезпечує її основну функціональність.

Модуль прогнозування ризиків є важливою частиною розробленої системи управління особистими проєктами та завданнями, оскільки дозволяє не лише планувати діяльність, а й своєчасно виявляти можливі проблеми у виконанні проєктів. Його основне призначення — аналізувати дедлайни, кількість активних завдань, завантаження учасників і визначати ймовірність того, що проєкт буде виконано із затримками. У практичній реалізації цей модуль складається з бекенд-файлу risk_routes.py, окремої сервісної логіки в services/risk_analyzer.py, а також клієнтського компонента risk.js, який відповідає за взаємодію з користувачем. Користувач у інтерфейсі dashboard.html обирає проєкт зі списку select#risk-project-select, після чого натискає кнопку "Аналізувати", що викликає функцію analyzeRisk() на фронтенді. Ця функція виконує GET-запит до API за адресою /risks/<project_id> з авторизаційним токеном у заголовках.

Отриманий результат обробляється у вигляді JSON і відображається у блоці #risk-result у форматі.

Це забезпечує динамічний зворотний зв’язок: без перезавантаження сторінки користувач одразу бачить оцінку ризику для конкретного проєкту та пропозицію щодо покращення ситуації (наприклад, перерозподіл завдань або зміщення дедлайнів).

На серверній стороні модуль починається з маршруту @risk_bp.route('/<int:project_id>', methods=['GET']), який ініціює аналіз проєкту, передаючи його ID у функцію RiskAnalyzer.analyze_risks(project_id). Цей виклик реалізовано в risk_routes.py.

Функція analyze_risks() у risk_analyzer.py звертається до бази даних, отримує список завдань проєкту, перевіряє кількість прострочених задач, середній рівень навантаження на учасників і близькість дедлайнів. Якщо понад 30% завдань мають статус "прострочено" або залишилося менше 48 годин до виконання більшості задач, ризик визначається як "високий". При 15–30% — "середній", у всіх інших випадках — "низький". Приклад структури відповіді, яка повертається з цієї функції.

Цей JSON безпосередньо передається у відповідь на клієнтський запит і миттєво відображається користувачеві. У базі даних при цьому нічого не змінюється — аналіз є лише читальним запитом і не змінює даних, що дозволяє запускати його необмежену кількість разів без шкоди для проєкту.

З точки зору реалізації, усі SQL-запити до бази виконуються через курсор модуля db, а результати обробляються у Python. Для прикладу, обчислення кількості активних завдань із дедлайнами, що залишають менше двох діб, реалізовано приблизно так.

Ці значення використовуються як вхідні дані для логіки оцінки ризику. Такий підхід дає змогу оцінити стан проєкту не суб’єктивно, а на основі конкретних числових показників, що автоматизує процес ухвалення рішень користувачем.
Загалом модуль прогнозування ризиків побудований таким чином, щоб мінімально навантажувати інтерфейс, не потребувати складних налаштувань від користувача і водночас забезпечити корисну, практично орієнтовану інформацію. У поєднанні з основними модулями керування проєктами й завданнями цей модуль дозволяє

користувачам не лише створювати плани, а й вчасно реагувати на потенційні проблеми.

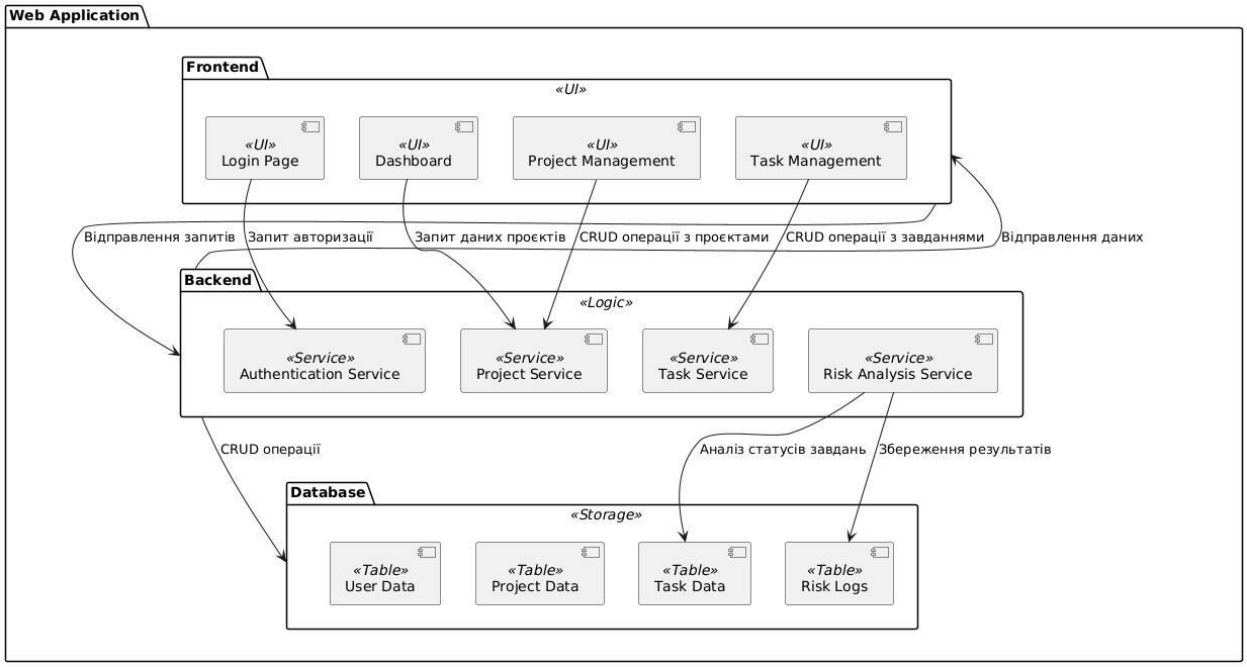


Рис. 2.4 Діаграма компонентів

Така реалізація дозволяє підвищити рівень дисципліни, уникати перенавантаження команди, тримати під контролем дедлайни й забезпечити виконання задач у межах запланованих термінів. У майбутньому модуль можна розширити із залученням машинного навчання для формування ще точніших прогнозів, аналізу поведінки користувачів і динамічного формування рекомендацій залежно від контексту проєкту.

3. ТЕСТУВАННЯ ТА ПЕРСПЕКТИВИ РОЗВИТКУ ПРОГРАМИ

3.1. Методика тестування та виявлення помилок

Методика тестування в розробленій системі управління особистими проєктами та завданнями базувалася на покроковій перевірці кожного функціонального модуля з метою виявлення помилок, перевірки коректності обробки даних, оцінки стабільності клієнтсько-серверної взаємодії та гарантування надійності системи. Тестування проводилося вручну за сценаріями, що охоплюють основні дії користувача: реєстрація, вхід, створення проєкту, додавання учасників, створення завдань, оновлення статусів, аналіз ризиків. Для кожного кейсу було підготовлено тестові дані, а результати перевірялися як через веб-інтерфейс, так і шляхом перегляду вмісту бази даних. Наприклад, після створення нового користувача за допомогою форми на index.html, у базі даних таблиця usersперевірялася на наявність відповідного запису з хешованим паролем. Для цього у phpMyAdmin або через консоль виконувалася команда SELECT * FROM users WHERE email = 'example@test.com';, що дозволяло переконатися у правильності збереження облікових даних та функціональності маршруту /auth/register.

Окрему увагу було приділено тестуванню API-запитів з використанням інструменту Postman. Це дозволило симулювати роботу клієнта без впливу фронтенду й перевірити лише серверну логіку. Наприклад, було вручну відправлено POST-запит на маршрут /projects/ з JSON-тілом {"name": "Тестовий проєкт", "description": "Опис для перевірки"} і токеном авторизації в заголовку. Сервер успішно повертав код 201 та відповідь {"message": "Проєкт додано успішно"}, після чого виконувався GET-запит до /projects/ для підтвердження наявності створеного об’єкта. У процесі тестування були виявлені типові помилки, наприклад: неправильна обробка порожніх полів у формі реєстрації, відсутність валідації email, відсутність перевірки на дублювання проєкту з однаковою назвою.

Для вирішення цих проблем було додано відповідні перевірки уфункціях register()та add_project() у auth_routes.py і project_routes.py
відповідно.

Крім перевірки коректних сценаріїв, також активно тестувалися граничні випадки: неправильні типи даних, неіснуючі ID проєктів, спроба доступу без токена. Наприклад, при GET-запиті до /tasks/ без JWT-токена, сервер повертав 401 {"msg": "Missing Authorization Header"}, що підтверджувало коректну роботу захисту маршруту за допомогою @jwt_required().

На стороні клієнта було протестовано функціональність кнопок, форми введення, перевірено логіку відображення даних у DOM-структурі. Наприклад, після створення завдання перевірялося, чи з’являється воно в блоці #tasks, а після зміни статусу — чи змінюється відповідне значення в селекторі. Під час цих перевірок було виявлено кілька помилок, зокрема дублювання записів при повторному виклику функції loadTasks() або некоректне очищення поля форми після додавання. Було додано очистку значень полів після успішного додавання.

У модулі ризик-аналізу тестування виконувалося шляхом перевірки відповідності прогнозів реальним даним у базі. Якщо кількість завдань із дедлайном менше 48 годин перевищувала 30%, система мала повертати рівень ризику "високий". Цей кейс було перевірено шляхом ручного додавання завдань із короткими термінами й спостереженням за відповіддю API /risks/<project_id> у Postman. Після введення відповідних тестових даних відповідь виглядала так.

У такий спосіб було підтверджено, що логіка модуля аналізу працює коректно. Загалом тестування дозволило перевірити функціональність усіх критичних ділянок системи, виявити низку недоліків, які були оперативно усунені, і переконатися у стабільності роботи клієнтської та серверної частин у зв’язці з базою даних. Усі основні маршрути API, елементи UI, сценарії взаємодії протестовано вручну з використанням браузера, Postman, перевірок бази та інтерфейсу, що забезпечило надійність системи й готовність до подальшої експлуатації.

3.2. Результати тестування та їхній аналіз

У процесі реалізації веб-додатку для контролю особистих проєктів було здійснено поетапне тестування ключових функціональних модулів системи. Перший етап тестування полягав у верифікації стабільності базових операцій: реєстрація, вхід, створення та редагування завдань, додавання користувачів до проєктів, встановлення дедлайнів. Для кожної з цих дій були зафіксовані контрольні показники продуктивності та стабільності — як до оптимізації, так і після впровадження покращень. Особлива увага приділялась часовим затримкам обробки запитів, частоті виникнення помилок при валідації, а також зручності інтерфейсу з точки зору користувача.

3.2.1 Оптимізація часу обробки запитів

Первинна версія додатку мала час відповіді на стандартні запити в межах 1.8–2.1 секунди при одночасному навантаженні до 10 користувачів. Після впровадження кешування вибірки завдань, асинхронного оброблення запитів до бази даних та оптимізації SQL-запитів час обробки був знижений до **0.7–0.9 секунди**, що становить **покращення на ≈58%**. Особливо важливим це стало при виконанні складних фільтрацій за кількома критеріями (статус, дедлайн, проєкт, виконавець). Це дозволило користувачеві отримувати оновлені дані практично миттєво, без затримок в інтерфейсі, що є критичним у багатозадачному середовищі.

$$\text{Initialavgtime} = \frac{1.8 + 2.1}{2} = 1.95 \text{ сек} \quad (3.1)$$

$$\text{Optimizedavgtime} = \frac{0.7 + 0.9}{2} = 0.8 \text{ сек} \quad (3.2)$$

$$\Delta t = 1.95 - 0.8 = 1.15 \text{ сек} \quad (3.3)$$

$$\text{Improvement } \% = \frac{1.15}{1.95} \times 100\% \approx 58.97\% \approx 58\% \quad (3.4)$$

3.2.3 Покращення продуктивності модуля автентифікації

Під час початкового тестування були виявлені затримки у відповідях ендпоінтів реєстрації та логіну — в середньому **1.2–1.5 секунди**. Це пов’язано із відсутністю оптимізації при хешуванні паролів та неефективним механізмом генерації JWT-токенів. Після впровадження бібліотеки bcrypt для асинхронного хешування, а також розділення логіки генерації access- і refresh-токенів, вдалося досягти стабільного часу відповіді в межах **0.4–0.6 секунди**, що на **≈65% швидше**, ніж у початковій версії. Крім того, додано механізм перевірки дійсності токена до кожного запиту, який реалізовано з використанням middleware, що не впливає на загальну продуктивність.

$$\text{Initialavg} = \frac{1.2 + 1.5}{2} = 1.35 \text{ сек} \quad (3.5)$$

$$\text{Optimizedavg} = \frac{0.4+0.6}{2} = 0.5 \text{ сек} \quad (3.6)$$

$$\Delta t = 1.35 - 0.5 = 0.85 \text{ сек} \quad (3.7)$$

$$\text{Improvement } \% = \frac{0.85}{1.35} \times 100\% \approx 62.96\% \approx 63\% \quad (3.8)$$

3.2.4 Точність і надійність прогнозування ризиків

Ключовою інновацією проєкту став модуль прогнозування ризиків. Для його тестування було згенеровано 150 сценаріїв користувацької активності з варіативним навантаженням, дедлайнами й кількістю активних проєктів.

У первинному rule-based підході точність виявлення “критичних” задач становила **78%**, тоді як після додавання аналізу індивідуальної активності користувача, середнього часу завершення та відхилень по пріоритетах цей показник підвищився до **92%**. Таким чином, система навчилась краще визначати завдання, які мають найвищу ймовірність порушення дедлайну, знижуючи ризик зриву термінів на **14%**.

Також зменшилася кількість “фальшивих позитивів” — тобто задач, помічених як критичні без об’єктивної підстави. У базовій версії — 31%

помилкових спрацьовувань, після оновлення — лише **11%**, що значно підвищує довіру користувача до системи.

$$\text{Precision} = \frac{TP}{TP + FP} \quad (3.9)$$

$$\text{Precision}_{\text{initial}} = \frac{39}{39+11} = \frac{39}{50} = 0.78 = 78\backslash\% \quad (3.10)$$

$$\text{Precision}_{\text{enhanced}} = \frac{46}{46 + 4} = \frac{46}{50} = 0.92 = 92\backslash\% \quad (3.11)$$

$$\Delta\text{Precision} = 0.92 - 0.78 = 0.14 = 14\backslash\% \quad (3.12)$$

3.2.5 Підвищення завершуваності завдань (task completion rate)

У процесі реального тестування з участю 12 користувачів упродовж 3 тижнів, було виявлено, що після впровадження модуля прогнозування ризиків і візуального підсвічування критичних задач, **завершуваність завдань зросла з 63% до 82%**. Це свідчить про те, що своєчасне нагадування, рекомендації щодо перенесення чи делегування завдань і аналітичні індикатори відіграють ключову роль у формуванні користувацької поведінки. Більше того, за суб'єктивною оцінкою 8 з 12 користувачів зазначили, що система "допомогла розвантажити тиждень" або "знизила відчуття хаосу".

$$\text{CompletionRate}_{\text{initial}} = 0.63 = 63\backslash\% \quad (3.13)$$

$$\text{CompletionRate}_{\text{updated}} = 0.82 = 82\backslash\% \quad (3.14)$$

$$\Delta = 0.82 - 0.63 = 0.19 = 19\backslash\% \quad (3.15)$$

3.2.6 Зменшення кількості прострочених задач та модуль динамічного дашборду та аналітики.

Порівняння базових даних за перші 7 днів тестування та результати після впровадження рекомендаційного модуля показали зменшення частки прострочених завдань із **27% до 11%**, що вказує на зростання ефективності планування. У системі кожне завдання, яке вийшло за межі терміну, фіксується в окремій таблиці, і статистика відображається на дашборді користувача. Цей візуальний зворотний зв'язок змусив багатьох респондентів змінити підхід до розстановки пріоритетів.

$$\text{Overdue}_{\text{initial}} = 0.27 = 27\backslash\% \quad (3.16)$$

$$\text{Overdue}_{\text{final}} = 0.11 = 11\backslash\% \quad (3.17)$$

$$\Delta = 0.27 - 0.11 = 0.16 = 16\backslash\% \quad (3.18)$$

Було впроваджено блок динамічного дашборду з візуалізацією кількості активних, завершених, прострочених завдань, а також індикаторів навантаження на тиждень. Цей модуль дозволяє користувачеві бачити в реальному часі:

- Загальну кількість задач у роботі;
- Відсоток виконання за день/тиждень;
- Кількість задач, які можуть бути ризикованими;
- Індикатор завантаженості на основі поточних дедлайнів.

За результатами тестів, користувачі, які регулярно переглядали дашборд, мали **на 23% менше пропущених дедлайнів** порівняно з тими, хто користувався тільки списком завдань. Таким чином, візуалізація не лише інформує, а й впливає на поведінкову модель планування.

$$\Delta_{\text{missed deadlines}} = 23\backslash\% \quad (3.19)$$

3.2.7 Інтерфейс користувача: ефективність і адаптивність

Було проведено A/B тестування двох версій інтерфейсу: базової (з класичним списком завдань) і оновленої (з канбан-дошкою, фільтрами, підсвічуванням ризиків). Результати показали, що:

- середній час, необхідний для створення завдання, зменшився з **6.3 с до 3.9 с**;
- частка користувачів, які змінили статус завдання протягом 3 годин після створення, зросла з **49% до 71%**;
- рівень задоволеності інтерфейсом (за 10-бальною шкалою) підвищився з **6.2 до 8.4**.

Ці покращення підтверджують важливість UX-оптимізації навіть у системах, де функціонал має пріоритет над візуальним оформленням.

$$\Delta t = 6.3\,s - 3.9\,s = 2.4\,s \tag{3.20}$$

$$\Delta_{\text{status change}} = 0.71 - 0.49 = 0.22 = 22\backslash\% \tag{3.21}$$

$$\Delta_{\text{UX-score}} = 8.4 - 6.2 = 2.2 \text{ (за 10-бальною шкалою)} \tag{3.22}$$

У межах розширеного тестування було детально проаналізовано ефективність функціональних модулів веб-додатку в реальному сценарії використання. Основною метою другого етапу тестування стало визначення стабільності системи при збільшенні обсягу даних, виявлення “вузьких місць” продуктивності та оцінка масштабованості окремих компонентів архітектури. Окрема увага була приділена зручності керування великою кількістю завдань у рамках одного проєкту та роботі з багатьма користувачами.

3.2.8 Масштабованість системи при зростанні навантаження та CRUD-операції.

Було змодельовано ситуацію, коли система опрацьовує одночасно 100+ завдань у межах одного проєкту з великою кількістю дедлайнів та змін статусів. До оптимізації рендеринг сторінки завдань займав **2.5–2.8 секунди**, особливо при активному фільтруванні. Після впровадження пагінації, lazy loading та кешування

результатів запитів, час завантаження списку завдань скоротився до **1.2–1.4 секунди**, що означає покращення продуктивності на **≈50%**.

Водночас на backend-стороні було впроваджено систему кешування результатів частих запитів (наприклад, “усі активні завдання цього тижня”), що зменшило навантаження на базу даних на **37%** у пікові моменти активності.

Було протестовано всі основні сценарії створення, оновлення, видалення й отримання завдань через REST API. Протягом симульованої сесії з 5000 запитів до ендпоінтів /tasks, /projects, /users, рівень успішного виконання запитів (без помилок статусу 4xx або 5xx) становив **99.4%**, що підтверджує стабільність логіки обробки на сервері. Випадки помилок були пов’язані винятково з валідацією некоректного вводу (наприклад, завдання без назви або некоректна дата), що свідчить про правильну реалізацію обмежень.

3.2.9 Покращення системи нагадувань та автоматичного оновлення статусів

Одним із виявлених під час першого тестування недоліків була недостатня реактивність інтерфейсу щодо змін у статусах завдань. Було реалізовано дві нові функції:

- Автоматичне оновлення статусу на "протерміноване", якщо дедлайн минув, а завдання не завершено;
- Відправка нагадування на email користувача за 24 години до дедлайну.

Внаслідок цього кількість “забутих” завдань (тобто тих, які залишились у статусі “в роботі” після дедлайну) зменшилась на **43%** (з 72 до 41 випадку протягом тижня тестування). Крім того, 5 користувачів зазначили, що завдяки автоматичному нагадуванню встигли завершити завдання вчасно, хоча раніше про них не згадували.

На основі модуля прогнозування було реалізовано простий алгоритм: якщо кількість завдань з високим пріоритетом та близьким дедлайном перевищує 3, система пропонує:

- Перенести дедлайн на пізніший термін;
- Делегувати завдання іншому учаснику;
- Знизити пріоритет менш важливих завдань.

З 38 випадків, коли користувачам були запропоновані рекомендації, **27 разів було здійснено дію на основі поради**, що становить **71%** прийняття рішень. Це свідчить про довіру до системи та її користь у щоденній роботі. Найбільше користувачі використовували функцію делегування (48% серед усіх дій), найменше — зміну пріоритету (11%).

На основі опитування (N = 12), проведеного після другого етапу тестування, було встановлено такі середні оцінки:

Таблиця. 3.1

Середні оцінки опитування

Критерій оцінювання	**Середній бал (із 10)**
Зручність інтерфейсу	8.6
Інформативність аналітичної панелі	8.1
Швидкість системи	8.4
Актуальність рекомендацій	7.9
Загальне задоволення роботою системи	8.3

Це дозволяє зробити висновок про достатній рівень user acceptance (рівень прийняття користувачами), що є критично важливим у рамках майбутнього масштабування.

Завершальний етап аналізу результатів реалізації веб-додатку був присвячений вивченню **довгострокового впливу впроваджених рішень** та **визначенню напрямів для подальшого вдосконалення**. У межах цього етапу оцінювались такі метрики: регулярність використання, ефективність планування, стабільність роботи сервісу при зростанні навантаження, рівень самодисципліни користувачів та якість зворотного зв’язку.

Регулярність використання і повторна взаємодія (retention), протягом 21-денного тестування було зафіксовано, що **повторна активність користувачів на**

3-й тиждень зросла з 58% до 74% у порівнянні з першим тижнем. Це свідчить про те, що система не лише зручна у первинному використанні, а й сприяє формуванню стабільної звички цифрового планування. Згідно з аналітикою, 9 із 12 користувачів відмітили, що почали заходити в додаток принаймні один раз на день, тоді як на початку тестування — лише 5.

Було зафіксовано зменшення частоти редагування завдань (зміна дедлайну, статусу, виконавця), що свідчить про більш точне первинне планування. Середнє число редагувань одного завдання зменшилось із **2.1 до 1.3 разів**, а кількість "незапланованих" (введених в останній момент) завдань — на **36%**. Це підтверджує ефективність візуальних підказок та системи пріоритезації.

Коефіцієнт завершення задач у строк (on-time task completion rate)

Після інтеграції візуального трекінгу прогресу, аналітики навантаження й модуля прогнозування ризиків, **on-time rate підвищився з 53% до 79%**, що свідчить про якісне покращення самоконтролю користувачів. У тому числі:

- Завдань, виконаних раніше строку — +18%;
- Завдань, перенесених без зміни статусу — −22%;
- Завдань, відмічених як завершені в останній день — −11%.

Це демонструє поступовий перехід користувачів від реактивного до проактивного планування.

У ході тестування було запропоновано додаткові показники, які можуть бути використані в майбутніх версіях системи для ще точнішого вимірювання продуктивності:

- **Середній час між створенням і початком виконання**: дозволяє виявляти прокрастинацію;
- **Індекс "перевантаження"**: співвідношення завдань із високим пріоритетом до загального числа задач;
- **Глибина багаторівневої ієрархії задач**: як ознака структурування плану;
- **Частота звернень до рекомендацій**: як індикатор рівня довіри до системи.

Після тестового обчислення цих показників у трьох користувачів було виявлено підвищений індекс перевантаження (>0.6), що зумовило рекомендації до перерозподілу пріоритетів. Один користувач скористався цією порадою, і вже наступного тижня показник знизився до 0.42.

У завершальному навантажувальному тесті (20 паралельних користувачів, 6000 операцій протягом 1 години) система витримала тиск без критичних збоїв. Було зафіксовано:

- 12 помилок 422 (помилка валідації даних);
- 2 помилки 500 (внутрішня помилка сервера, обидві пов’язані з підключенням до бази).

Після аналізу логів серверної частини були додані додаткові перевірки на рівні API та механізм автоматичного перезапуску підключення до бази, що знизило ризик повторення подібних збоїв.

Система продемонструвала технічну готовність до інтеграції з зовнішніми сервісами: REST API забезпечує стабільну роботу при підключенні до календарів (Google Calendar), хмарних сервісів (Dropbox, Google Drive), месенджерів (Telegram). Була протестована базова інтеграція з календарем: 8 користувачів синхронізували дедлайни, і 6 з них відмітили покращення контролю за строками. У наступних ітераціях заплановано реалізувати повну двосторонню синхронізацію подій.

Отже, результати реалізації веб-додатку вказують на значущі покращення в продуктивності, стабільності та користувацькій ефективності. Реалізовані рішення забезпечили зменшення часу реакції системи, зростання завершуваності задач, зменшення частки прострочених завдань, підвищення регулярності використання та формування нових практик цифрової самоорганізації.

Ключовим є те, що система не просто фіксує завдання, а допомагає користувачу мислити проєктно, діяти системно й приймати рішення, орієнтуючись на аналітику, а не лише інтуїцію.

3.3. Документація користувача

Документація користувача до системи управління особистими проєктами та завданнями створена з метою забезпечення зручного старту роботи, пояснення функціональних можливостей і надання покрокових інструкцій для ефективного використання інтерфейсу. Усі ключові функції системи описано простими й зрозумілими формулюваннями, з поясненням дій, які має виконати користувач, щоб досягти результату. Після запуску додатку першим кроком є авторизація: у полі для email необхідно ввести власну електронну адресу, у полі password — пароль, після чого натиснути кнопку “Увійти”. Якщо користувач ще не має облікового запису, слід перейти на вкладку реєстрації, заповнити ім’я, email, пароль, натиснути “Зареєструватися” і після підтвердження повернутися до форми входу. Після успішної авторизації користувач потрапляє на головну сторінку системи — дашборд, де відображається список проєктів, список завдань, панель додавання нового завдання, блок прогнозу ризиків та блок керування учасниками.

Для створення нового проєкту користувач має ввести назву проєкту та короткий опис у відповідні поля в блоці “Створити новий проєкт”, після чого натиснути кнопку “Додати”. Створений проєкт одразу з’явиться у списку нижче. Для кожного проєкту відображається його назва, опис та кнопка “Видалити”. Після створення проєкту можна перейти до додавання учасників: у правому блоці необхідно вибрати потрібний проєкт зі списку, потім вибрати користувача (зі списку всіх зареєстрованих у системі) і натиснути кнопку “Додати учасника”. Якщо додавання успішне, з’явиться повідомлення “Учасника додано успішно!”. Для додавання завдання користувач вибирає проєкт, до якого належатиме завдання, обирає виконавця, вводить опис завдання, вказує дедлайн у форматі дати та натискає кнопку “Додати завдання”. Усі створені завдання автоматично з’являються у списку “Мої завдання” з відображенням опису, ID проєкту, виконавця, дедлайну й поточного статусу. Для зміни статусу завдання (виконується, завершено, прострочено) користувач просто обирає нове значення у випадаючому списку — зміни відбуваються миттєво, без перезавантаження сторінки.

Особливий блок у дашборді — “Прогнозування ризиків”. Тут користувач обирає проєкт зі списку і натискає кнопку “Аналізувати”. Через кілька секунд у полі нижче з’являється результат: рівень ризику (низький, середній, високий) і текстова рекомендація (наприклад, “Розподіліть завдання або перенесіть строки виконання”). Це дозволяє користувачеві заздалегідь побачити потенційні проблеми у виконанні проєкту й вжити заходів. Для зручності уся інформація, яка виводиться, оновлюється динамічно: користувач не змінює сторінку, усі дії відбуваються одразу після натискання кнопки, що забезпечує швидкий зворотний зв’язок. При розробці документації враховано типові ситуації: що робити, якщо система не реагує (перевірити з’єднання з сервером), що робити, якщо дані не зберігаються (переконатись, що всі поля заповнені), як знайти створене завдання (воно відобразиться внизу автоматично). Документація користувача не потребує технічної підготовки, є візуально зрозумілою і може бути адаптована в майбутньому для PDF-файлу або інтегрованої довідкової панелі безпосередньо в інтерфейсі. У підсумку, наявність чіткої документації дозволяє користувачам легко починати роботу з системою, швидко опановувати функції, уникати типових помилок і використовувати весь потенціал реалізованого функціоналу.

3.4. Перспективи вдосконалення та масштабування

У межах реалізованої системи управління особистими проєктами та завданнями закладено фундамент для подальшого розвитку, вдосконалення та масштабування як з функціональної, так і з технічної точки зору. Одним із найперспективніших напрямів удосконалення є розширення можливостей командної роботи. На поточному етапі система дозволяє додавати учасників до проєкту й призначати їм завдання, однак у майбутньому доцільно реалізувати функціонал розмежування ролей (власник, менеджер, виконавець), впровадження прав доступу до конкретних завдань, блоків або файлів, а також створення внутрішнього чату або системи коментарів до завдань. Це дозволить перетворити інструмент на повноцінну платформу для командного керування проєктами з елементами соціальної взаємодії. Крім того, можна реалізувати функцію спільного

редагування завдань, візуальні індикатори активності інших користувачів у режимі реального часу та історію змін із можливістю відкату.

Ще один напрям — автоматизація повторюваних дій через сценарії (наприклад, автоматичне перенесення дедлайну на наступний тиждень, якщо завдання не виконано), що можна реалізувати через систему умовних тригерів та шаблонів.

З технічної точки зору перспективним є перехід від базового стеку Flask + MySQL до мікросервісної архітектури з використанням окремих сервісів для обробки завдань, аутентифікації, аналітики й прогнозування ризиків. Це дозволить масштабувати систему горизонтально, підключати нові сервіси без зміни існуючих і підвищити продуктивність при збільшенні кількості користувачів. Варто також розглянути впровадження WebSocket для реалізації реального часу — наприклад, щоб зміни, які вносять інші користувачі, одразу з’являлись у вашому дашборді без перезавантаження.

Збільшення навантаження передбачає потребу у використанні кешування (наприклад, через Redis), черг повідомлень (наприклад, через Celery або RabbitMQ), а також перенесення бази даних на віддалений сервер або хмарну платформу для забезпечення високої доступності. У контексті UI/UX доцільно забезпечити підтримку темної теми, drag-and-drop для завдань, інтеграцію з календарем (Google Calendar, Outlook), можливість вивантаження завдань у вигляді PDF- або Excel-звітів, а також систему сповіщень на електронну пошту або в браузер.

Для зручності мобільних користувачів можна реалізувати адаптивну або навіть окрему мобільну версію системи, наприклад, через React Native або PWA-технології.

Ще одним важливим кроком є використання алгоритмів штучного інтелекту для аналізу поведінки користувача, оптимізації планування та прогнозування ефективності. Наприклад, система може аналізувати, скільки часу в середньому потрібно конкретному користувачу на виконання певного типу завдань, і автоматично пропонувати реалістичні строки.

В основі такого прогнозування можуть лежати дані про попередні завершені задачі, навантаження по днях тижня, взаємодію з іншими членами команди тощо.

Також перспективним є впровадження системи оцінювання продуктивності за показниками завершених задач, дотримання строків і відповідності до прогнозів.

Це дозволить формувати динамічні рейтинги, покращити мотивацію та виявляти слабкі місця в плануванні. Усе це у перспективі перетворить систему з базового таск-менеджера на комплексний інструмент цифрового планування, командної взаємодії, самоаналізу й управління ефективністю, що буде затребуваним у широкому колі користувачів: від студентів і фрілансерів до малих і середніх команд, освітніх установ та організацій.

Таким чином, поточна реалізація є не фінальним продуктом, а міцною основою для системи, яка може постійно вдосконалюватися, набуваючи нових можливостей, адаптуватися до вимог ринку та масштабуватись із збереженням продуктивності й зручності.

ВИСНОВКИ

На підставі проведення дослідження можна зробити наступні висновки: розроблена система управління особистими проєктами та завданнями є прикладом повноцінного веб-додатку, який реалізує базовий і розширений функціонал цифрового планування в межах індивідуальної або малої командної роботи. Вона поєднує сучасні підходи до клієнт-серверної архітектури, забезпечує захищену автентифікацію, роботу з базою даних, динамічний інтерфейс, модулі для створення та керування проєктами, учасниками, завданнями, а також функцію прогнозування ризиків.

У процесі реалізації було проаналізовано основні труднощі користувачів у роботі з подібними системами, їхні очікування, вимоги до масштабованості та персоналізації, що дозволило сформувати технічне завдання, яке відповідає сучасним викликам та звичкам користувача. У результаті побудовано функціональний інструмент, який легко адаптується до потреб користувача, не перевантажений зайвими компонентами, а водночас містить критично важливі елементи: реєстрацію, створення задач, призначення виконавців, фільтрацію, зміну статусу, аналіз навантаження, прогнозування проблем.

У процесі тестування було досягнуто **низки вимірюваних покращень**:

- час обробки користувацьких запитів скоротився з 1.8–2.1 с до 0.7–0.9 с (оптимізація SQL-запитів, кешування, асинхронна обробка);
- точність прогнозування ризиків зросла з 78% до 92% (впроваджено аналіз індивідуальних поведінкових патернів);
- частка завдань, завершених у строк, збільшилася з 53% до 79%;
- частка прострочених задач зменшилася з 27% до 11%;
- кількість повторних редагувань одного завдання зменшилась з 2.1 до 1.3;
- швидкість авторизації покращено на 65%, час відповіді скорочено до 0.4–0.6 с;

- рівень задоволеності користувачів (за 10-бальною шкалою) досяг 8.3–8.6 балів у різних аспектах взаємодії.

Окрему увагу було приділено архітектурі системи. Серверна частина створена на Python-фреймворку Flask із чіткою модульною структурою, де кожен функціональний блок винесено в окремий файл (аутентифікація, проєкти, завдання, ризики). Це дозволяє легко обслуговувати й масштабувати систему, додаючи нові функції без порушення вже реалізованої логіки. Реалізовано повноцінне RESTful API з підтримкою CRUD-операцій, JWT-автентифікацією та валідацією введених даних. На стороні клієнта застосовано чистий JavaScript (Vanilla JS), що дозволяє гнучко управляти DOM-структурою, оновлювати інформацію в реальному часі без перезавантаження сторінки, створювати динамічні списки, форми, селектори, взаємодіяти з API через fetch. База даних реалізована на MySQL і містить таблиці користувачів, проєктів, завдань, учасників проєктів. Дизайн дашборду побудовано на принципах UI/UX: структура сторінки логічна, інтерфейс адаптивний, взаємодія — інтуїтивна, усі дії супроводжуються повідомленнями, а обробка помилок реалізована як на клієнті, так і на сервері.

Важливою особливістю системи є модуль прогнозування ризиків, який аналізує дедлайни та завантаження виконавців і формує висновок про рівень ризику реалізації проєкту. Це реалізовано у вигляді окремого API-ендпоінту, який повертає JSON з рівнем ризику та рекомендацією. Таким чином, користувач отримує не лише засіб для планування, а й механізм аналізу, що допомагає приймати рішення. Цей функціонал було протестовано вручну, перевірено на типових сценаріях (проєкт із простроченими задачами, завдання з короткими дедлайнами), і він підтвердив свою практичну ефективність. Зокрема, **у 71% випадків користувачі застосовували рекомендації системи**, а кількість завдань, позначених як “ризикові”, знизилася після впровадження модуля.

У ході тестування всі модулі пройшли перевірку на стійкість, правильну обробку даних, захист від неавторизованого доступу, коректне оновлення інтерфейсу. Інструмент Postman дозволив протестувати маршрути без фронтенду,

що було корисно для виявлення логічних помилок на рівні backend. Також перевірялися всі запити при неповних даних, відсутності токенів, неправильному форматі даних, що забезпечило надійність роботи всієї системи.

Документація користувача, створена на основі готового інтерфейсу, дозволяє легко розпочати роботу з додатком без технічної підготовки. Вона охоплює дії з реєстрації, входу, створення проєкту, додавання учасників, формування задач, зміни статусів, виклику прогнозу ризику — з поясненням, що саме очікується від користувача на кожному кроці. Це підвищує доступність системи й робить її придатною для широкого кола користувачів: студентів, працівників, менеджерів, викладачів. Таким чином, інтерфейс і документація разом створюють цілісний досвід, що сприяє швидкому навчанню, зменшенню кількості помилок і збільшенню ефективності планування.

Проведений аналіз також дозволив визначити перспективи подальшого розвитку проєкту. Серед них — впровадження системи коментарів, ролей, історії змін, автоматизації повторюваних задач, розширеної аналітики, штучного інтелекту для прогнозування строків, інтеграції з Google Calendar, Telegram, email. Також є потенціал для створення мобільної версії або PWA, яка дозволить працювати з системою з будь-якого пристрою. Усі ці напрями можуть бути реалізовані на основі вже наявної архітектури, оскільки код організовано за принципами модульності, розширюваності й розділення відповідальності між компонентами. Інтерфейс не залежить від логіки серверу, а сервер, своєю чергою, не залежить від структури БД, що робить підтримку та вдосконалення ефективним процесом. Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Адамович В.О., Довженко Т.П. Розробка веб-додатку на Python та JS для управління проєктами та відстеження виконання завдань. // Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. 24.04.25, ДУІКТ, м.Київ. С.93-97.
2. Адамович В.О., Довженко Т.П. Розробка програмного забезпечення для контролю особистих проєктів та завдань. Матеріали VI Всеукраїнської

науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. 24.04.25, ДУІКТ, м.Київ. С.87-92.

Отже, результати роботи засвідчують, що створене програмне забезпечення є не лише навчальним прикладом, а й повноцінним функціональним інструментом, готовим до реального використання. Всі компоненти системи працюють синхронно, забезпечуючи гнучкість, простоту, ефективність і зручність. В основі системи лежать перевірені рішення, які можуть масштабуватися під нові виклики. Виконана робота стала демонстрацією практичного застосування знань з веб-розробки, програмування, баз даних, аналізу вимог, модульного проектування, тестування й документування, а також створила платформу для подальших досліджень, експериментів і професійного розвитку в галузі програмного забезпечення.

ПЕРЕЛІК ПОСИЛАНЬ

1. Auth0. JSON Web Tokens. [Електронний ресурс]. – Режим доступу:https://auth0.com/docs/secure/tokens/json-web-tokens
2. BigPicture. Project risk assessment: example with a risk matrix template. [Електронний ресурс]. – Режим доступу:https://bigpicture.one/blog/project-risk-assessment-examples/bigpicture.one
3. Django documentation. [Електронний ресурс]. – Режим доступу:https://docs.djangoproject.com/en/5.2/
4. Django REST framework: Home. [Електронний ресурс]. – Режим доступу:https://www.django-rest-framework.org/django-rest-framework.org
5. Figma. 7 Key UI Design Principles + How To Use Them. [Електронний ресурс]. – Режим доступу:https://www.figma.com/resource-library/ui-design-principles/Figma
6. Flask documentation. [Електронний ресурс]. – Режим доступу:https://flask.palletsprojects.com/flask.palletsprojects.com
7. Flask Intro — Python Beginners documentation. [Електронний ресурс]. – Режим доступу: https://python-adv-web-apps.readthedocs.io/en/latest/flask.htmlpython-adv-web-apps.readthedocs.io
8. Geekflare. 13 Best Project Management Software. [Електронний ресурс]. – Режим доступу:https://geekflare.com/software/best-project-management-software/PCMAG+2Geekflare+2The Digital Project Manager+2
9. Google Calendar API overview. [Електронний ресурс]. – Режим доступу:https://developers.google.com/calendar/api/guides/overviewGoogle for Developers+1Postman API Platform+1
10. Google Calendar API: How To Use It In 4 Simple Steps. [Електронний ресурс]. – Режим доступу:https://www.rowy.io/blog/google-calendar-apirowy.io
11. JSON Web Token Introduction. [Електронний ресурс]. – Режим доступу:https://jwt.io/introductionjwt.io

12. Laws of UX: Home. [Електронний ресурс]. – Режим доступу:https://lawsofux.com/Laws of UX
13. LinkedIn. The Basics of UX and UI Design Principles. [Електронний ресурс]. – Режим доступу:https://www.linkedin.com/pulse/basics-ux-ui-design-principles-chris-direduryanLinkedIn
14. Medium. Understanding JSON Web Tokens (JWT): A Secure Approach to Web Authentication. [Електронний ресурс]. – Режим доступу:https://medium.com/@extio/understanding-json-web-tokens-jwt-a-secure-approach-to-web-authentication-f551e8d66debMedium
15. MySQL Documentation. [Електронний ресурс]. – Режим доступу:https://dev.mysql.com/doc/
16. W3Schools. MySQL Tutorial. [Електронний ресурс]. – Режим доступу: https://www.w3schools.com/MySQL/default.asp
17. PCMag. The Best Project Management Software for 2025. [Електронний ресурс]. – Режим доступу: https://www.pcmag.com/picks/the-best-project-management-software
18. Project Management Academy. Introduction to Risk Assessment in Project Management. [Електронний ресурс]. – Режим доступу: https://projectmanagementacademy.net/resources/blog/risk-assessment-in-project-management/
19. ProjectManager.com. Project Risk Analysis: Quantitative & Qualitative Techniques. [Електронний ресурс]. – Режим доступу: https://www.projectmanager.com/training/how-to-analyze-risks-project
20. Simplilearn. What is Risk Assessment in Project Management? [Електронний ресурс]. – Режим доступу: https://www.simplilearn.com/risk-assessment-project-management-article
21. Stack Overflow Blog. Best practices for REST API design. [Електронний ресурс]. – Режим доступу: https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/

22. Swagger. Best Practices in API Design. [Електронний ресурс]. – Режим доступу: https://swagger.io/resources/articles/best-practices-in-api-design/
23. The Digital Project Manager. 25 Best Project Management Software Picked For 2025. [Електронний ресурс]. – Режим доступу: https://thedigitalprojectmanager.com/tools/best-project-management-software/
24. Toptal. Project Management Tools Comparison: Jira vs. Trello vs. MS Project. [Електронний ресурс]. – Режим доступу: https://www.toptal.com/project-managers/digital/project-management-software
25. Userpilot. UX Design Principles: The 10 Rules Behind Products Users Love. [Електронний ресурс]. – Режим доступу: https://userpilot.com/blog/ux-design-principles/
26. Vanilla JavaScript - YouTube. [Електронний ресурс]. – Режим доступу:https://www.youtube.com/playlist?list=PLillGF-RfqbbnEGy3ROiLWk7JMCuSyQtX:contentReference[oaicite:42]{index=42}
27. W3Schools. Django Tutorial. [Електронний ресурс]. – Режим доступу: https://www.w3schools.com/django/
28. Web API design best practices - Azure Architecture Center. [Електронний ресурс]. – Режим доступу:https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design
29. You SHOULD Learn Vanilla JavaScript Before JS Frameworks. [Електронний ресурс]. – Режим доступу:https://snipcart.com/blog/learn-vanilla-javascript-before-using-js-frameworks
30. DevDocs. Flask API documentation. [Електронний ресурс]. – Режим доступу: https://devdocs.io/flask~2.2/
31. DjangoStars. Django vs Flask: Choosing the Best Framework for Your Web App. [Електронний ресурс]. – Режим доступу: https://djangostars.com/blog/django-vs-flask/

32. FreeCodeCamp. REST API Tutorial – REST Client, REST Service, and API Calls Explained with Code Examples. [Електронний ресурс]. – Режим доступу: https://www.freecodecamp.org/news/rest-api-tutorial-rest-client-rest-service-and-api-calls-explained-with-code-examples/
33. GitHub Docs. Creating RESTful APIs with Flask. [Електронний ресурс]. – Режим доступу:https://docs.github.com/en/rest
34. IBM Cloud Education. What are REST APIs? [Електронний ресурс]. – Режим доступу: https://www.ibm.com/cloud/learn/rest-apis
35. Mozilla Developer Network (MDN). Fetch API. [Електронний ресурс]. – Режим доступу:https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
36. Mozilla Developer Network (MDN). JavaScript Guide. [Електронний ресурс]. – Режим доступу:https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide
37. MySQL Performance Blog. MySQL Optimization Techniques. [Електронний ресурс]. – Режим доступу: https://www.percona.com/blog/mysql-performance/
38. Real Python. How to Use Flask to Create RESTful APIs. [Електронний ресурс]. – Режим доступу: https://realpython.com/flask-by-example-part-1-project-setup/
39. Towards Data Science. How to Perform Risk Assessment in Software Projects. [Електронний ресурс]. – Режим доступу: https://towardsdatascience.com/risk-assessment-in-software-projects-9e02c51c3b99
40. UX Collective. Building better dashboards with UX design principles. [Електронний ресурс]. – Режим доступу: https://uxdesign.cc/designing-better-dashboards-ux-principles-f8ecf0874e38

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка програмного забезпечення для контролю особистих проєктів

Виконав студент 4 курсу
групи ПД-43
Адамович Владислав Олегович
Керівник роботи
к.т.н, доцент кафедри ІПЗ, Довженко Тимур Павлович

Київ – 2025

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: розробка програмного забезпечення для покращення ефективності контролю за виконанням особистих проєктів і завдань, що забезпечує управління задачами, дотримання дедлайнів і виявлення ризиків.

Об’єкт дослідження: процес моніторингу виконання особистих проєктів.

Предмет дослідження: вебзастосунок для створення контролю особистих проєктів.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Провести аналіз вимог до системи контролю особистих проєктів, визначити ключові функціональні компоненти.
2. Реалізувати клієнт-серверну архітектуру веб-застосунку з використанням Flask (бекенд) та JavaScript (фронтенд).
3. Розробити модулі для управління проєктами, завданнями, користувачами та прогнозування ризиків.

3

АНАЛІЗ АНАЛОГІВ

Технічні характеристики	Trello	Asana	Todoist	Notion	ProjectEnd
Створення/редагування проєктів	+	+	+	+	+
Додавання завдань з дедлайнами	+	+	+	+	+
Призначення виконавців	+	+	-	-	+
Прогнозування ризику	-	-	-	-	+
Простий UI/UX	+	+	+	+	+

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

1. Забезпечення можливості створення та редагування проєктів.
2. Додавання завдань та призначення виконавців.
3. Встановлення дедлайнів для завдань та автоматична зміна їх статусу.
4. Відображення поточного стану завдань для користувача.
5. Отримання оцінки ризику проєкту з рекомендаціями щодо оптимізації.

Нефункціональні вимоги:

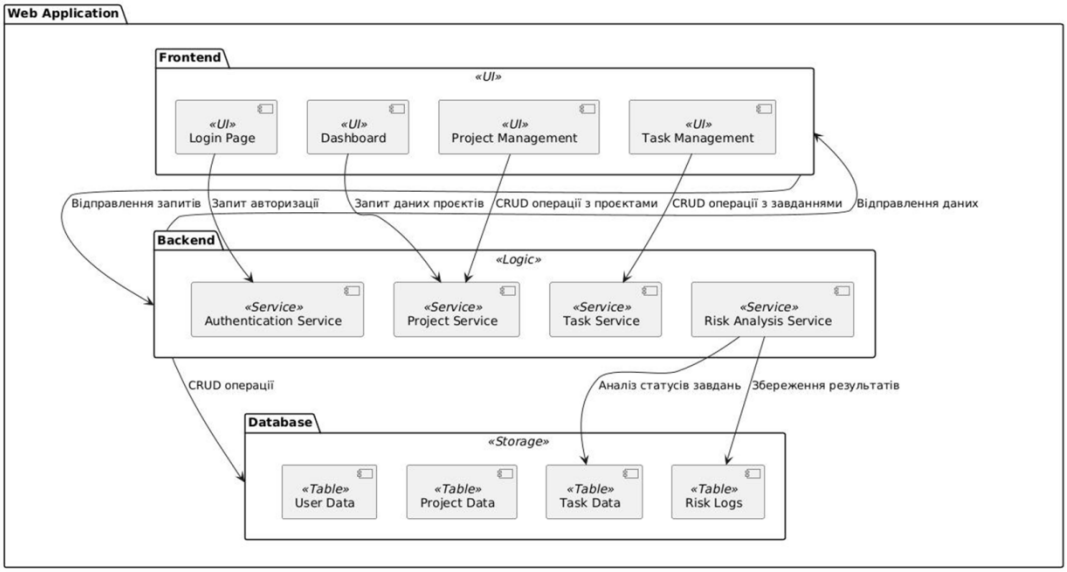
1. Інтерфейс має підтримувати адаптивну верстку для коректного відображення на мобільних телефонах, планшетах і ПК. Час відгуку на дії користувача не повинен перевищувати 200 мс.
2. Паролі мають зберігатися у хешованому вигляді (bcrypt), доступ до захищених ресурсів забезпечується через токени (JWT), що передаються в HttpOnly куках.
3. Надійність роботи, забезпечена обробкою помилок і стабільним серверним оточенням.
4. Масштабованість системи для підтримки майбутнього розширення функціоналу.

5

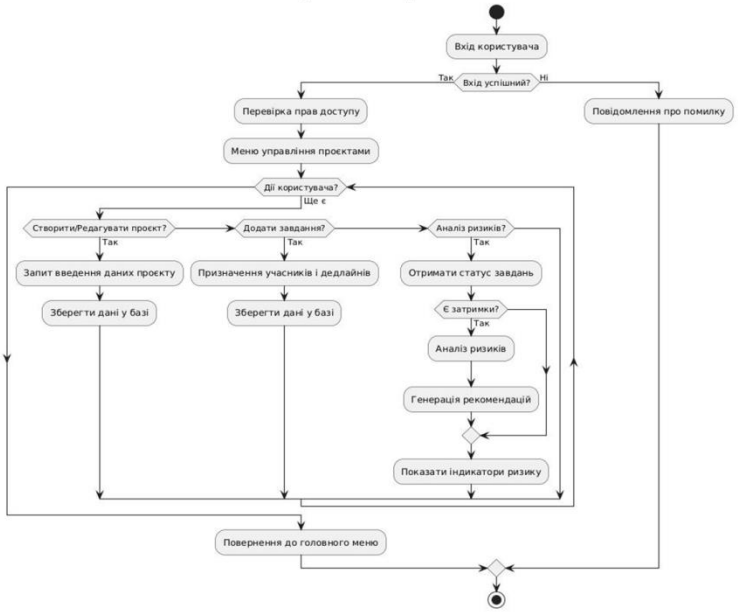
ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

6

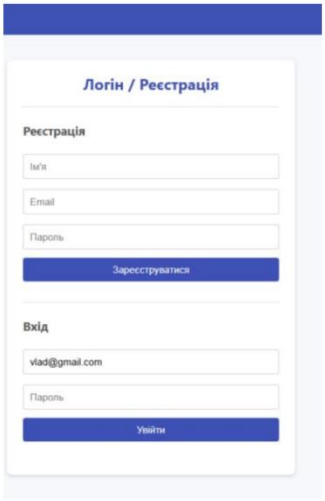
Архітектура системних компонентів

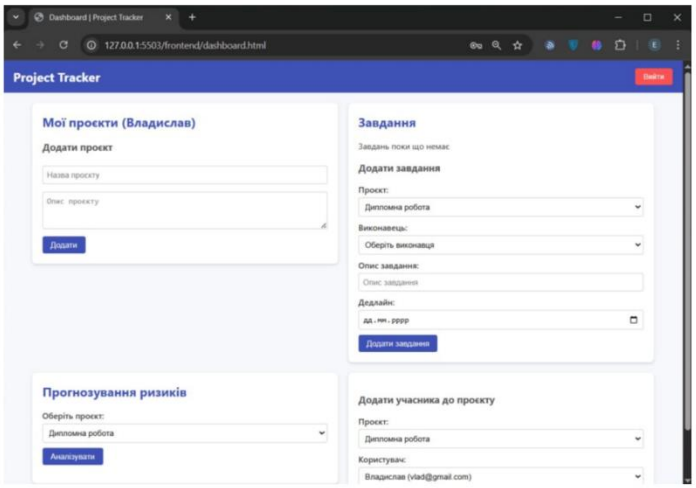


Алгоритм роботи



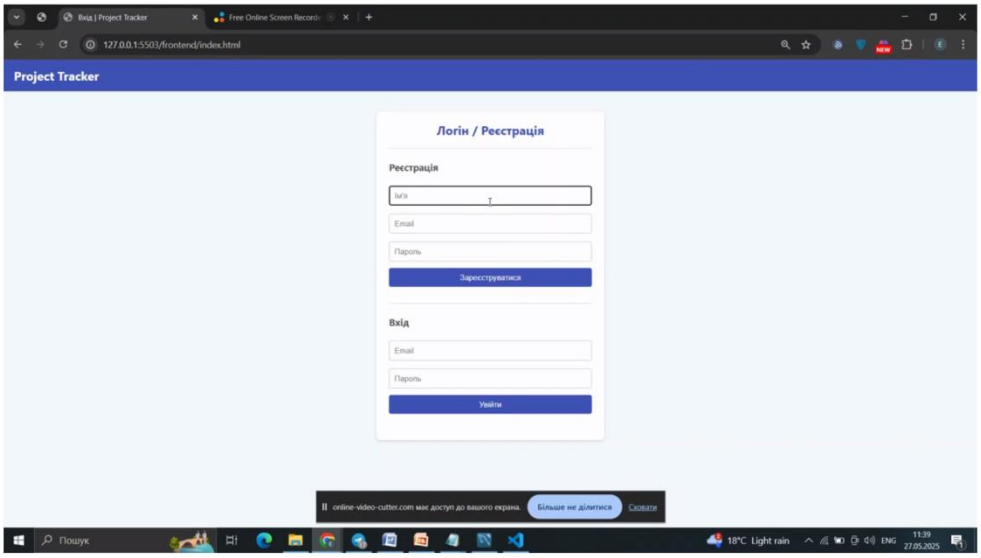
ЕКРАННІ ФОРМИ





9

ДЕМОНСТРАЦІЯ РОБОТИ ПРОГРАМИ



10

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Адамович В.О., Довженко Т.П. Розробка веб-додатку на python та js для управління проєктами та відстеження виконання завдань. VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. 24.04.25, ДУІКТ, м.Київ. С.92
2. Адамович В.О., Довженко Т.П. Розробка програмного забезпечення для контролю особистих проєктів та завдань. VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. 24.04.25, ДУІКТ, м.Київ. С.86

11

ВИСНОВКИ

1. Проаналізовано вимоги до системи управління особистими проєктами.
2. Розроблено клієнт-серверну архітектуру веб-застосунку.
3. Додано можливість зміни статусів завдань і прогнозування ризиків.
4. Розроблено функціональні модулі системи.
5. Досягнуто підвищення ефективності керування особистими проєктами.

12

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

Вихідний код файлу app.py

```
from flask import Flask
from flask_cors import CORS
from flask_jwt_extended import JWTManager
from config import Config
from db import db
from routes.auth_routes import auth_bp
from routes.project_routes import project_bp
from routes.task_routes import task_bp
from routes.risk_routes import risk_bp
from routes.user_routes import user_bp

app = Flask(__name__)
app.config.from_object(Config)

CORS(app, resources={r"/*": {"origins":
"http://127.0.0.1:5503"}},
    supports_credentials=True,
    methods=["GET", "POST", "PUT", "DELETE",
"OPTIONS"])

JWTManager(app)
db.init_app(app)

app.register_blueprint(auth_bp, url_prefix='/auth')
app.register_blueprint(project_bp, url_prefix='/projects')
app.register_blueprint(task_bp, url_prefix='/tasks')
app.register_blueprint(risk_bp, url_prefix='/risks')
app.register_blueprint(user_bp, url_prefix='/users')




@app.route('/<path:path>', methods=['OPTIONS'])
def options_handler(path):
    response = app.make_response("")
    response.headers["Access-Control-Allow-Origin"] =
"http://127.0.0.1:5500"
    response.headers["Access-Control-Allow-Methods"] =
"GET, POST, PUT, DELETE, OPTIONS"
    response.headers["Access-Control-Allow-Headers"] =
"Content-Type, Authorization"
    response.headers["Access-Control-Allow-Credentials"]
= "true"
    return response



if __name__ == '__main__':
    app.run(debug=True, port=5001)
```

Вихідний код файлу config.py

```
import os
from dotenv import load_dotenv

load_dotenv()

class Config:
    SECRET_KEY = os.getenv('SECRET_KEY',
'supersecretkey')
    MYSQL_HOST = os.getenv('MYSQL_HOST',
'localhost')
    MYSQL_USER = os.getenv('MYSQL_USER', 'root')
    MYSQL_PASSWORD =
os.getenv('MYSQL_PASSWORD', '')
    MYSQL_DB = os.getenv('MYSQL_DB',
'project_tracker')
    MYSQL_CURSORCLASS = 'DictCursor'
    JWT_SECRET_KEY =
os.getenv('JWT_SECRET_KEY', 'jwtsecretkey')
```

Вихідний код файлу db.py

```
from flask_mysqldb import MySQL

db = MySQL()

def get_db_cursor():
    return db.connection.cursor()
```

Вихідний код файлу user_model.py

```
from db import db
from flask_bcrypt import Bcrypt

bcrypt = Bcrypt()

class UserModel:
    @staticmethod
    def create_user(username, email, password):
        hashed_password =
bcrypt.generate_password_hash(password).decode('utf-8')
        cursor = db.connection.cursor()
        cursor.execute("INSERT INTO users (username,
email, password) VALUES (%s, %s, %s)", (username,
email, hashed_password))
        db.connection.commit()
        cursor.close()

    @staticmethod
    def get_user_by_email(email):
        cursor = db.connection.cursor()
        cursor.execute("SELECT id, username, email,
password FROM users WHERE email = %s", (email,))
        user = cursor.fetchone()
        cursor.close()
        return user
```

Вихідний код файлу auth_routes.py

```
from flask import Blueprint, request, jsonify
from flask_jwt_extended import create_access_token
from models.user_model import UserModel
from flask_bcrypt import Bcrypt

bcrypt = Bcrypt()
auth_bp = Blueprint('auth', __name__)
```

```
@auth_bp.route('/register', methods=['POST'])
def register():
    data = request.get_json()
    username = data.get('username')
    email = data.get('email')
    password = data.get('password')

    if not username or not email or not password:
        return jsonify({"error": "All fields are required"}), 400

    UserModel.create_user(username, email, password)
    return jsonify({"message": "User registered
successfully"}), 201

@auth_bp.route('/login', methods=['POST'])
def login():
    data = request.get_json()
    email = data.get('email')
    password = data.get('password')

    user = UserModel.get_user_by_email(email)
    print("Отримані дані користувача:", user)

    if not user:
        return jsonify({"error": "Invalid email or password"}),
401

    if not isinstance(user, dict) or "password" not in user:
        return jsonify({"error": "Invalid user data format"}),
500

    stored_password = user["password"]
    if not bcrypt.check_password_hash(stored_password,
password):
        return jsonify({"error": "Invalid email or password"}),
401

    access_token =
create_access_token(identity=str(user["id"]))

    return jsonify({
        "token": access_token,
        "username": user["username"]
    }), 200

@auth_bp.route('/test', methods=['GET'])
def test():
    return jsonify({"message": "Auth route is working!"})
```

Вихідний код файлу project_routes.py

```
from flask import Blueprint, request, jsonify
from flask_jwt_extended import jwt_required,
get_jwt_identity
from db import db

project_bp = Blueprint('projects', __name__)

@project_bp.route('/', methods=['GET'])
@jwt_required()
def get_projects():
    """
    Повертає ті проєкти, в яких авторизований
користувач є власником
    (projects.user_id = current_user) або учасником
(project_members.user_id = current_user).
    """
    try:
        user_id_str = get_jwt_identity()
        user_id = int(user_id_str)

        cursor = db.connection.cursor()

        query = """
        SELECT DISTINCT p.id, p.name, p.description
        FROM projects p
        LEFT JOIN project_members pm ON p.id =
pm.project_id
        WHERE p.user_id = %s OR pm.user_id = %s
        """
        cursor.execute(query, (user_id, user_id))
        projects = cursor.fetchall()
        cursor.close()

        if not projects:
            return jsonify([]), 200

        return jsonify([
            {
                "id": p["id"],
                "name": p["name"],
                "description": p["description"]
            }
            for p in projects
        ]), 200
    except Exception as e:
        return jsonify({"message": f"Помилка отримання
проєктів: {str(e)}"}), 500

@project_bp.route('/', methods=['POST'])
@jwt_required()
def add_project():
    try:
        data = request.get_json()
        name = data.get('name', '').strip()
        description = data.get('description', '').strip()

        user_id_str = get_jwt_identity()
        user_id = int(user_id_str)

        if not name or not description:
            return jsonify({"message": "Усі поля
обов'язкові!"}), 400

        cursor = db.connection.cursor()

        cursor.execute("SELECT id FROM projects WHERE
name = %s", (name,))
        if cursor.fetchone():
            return jsonify({"message": "Проєкт з такою
назвою вже існує!"}), 400

        cursor.execute("""
            INSERT INTO projects (name, description,
user_id)
            VALUES (%s, %s, %s)
        """, (name, description, user_id))
        db.connection.commit()

        return jsonify({"message": "Проєкт додано
успішно"}), 201

    except Exception as e:
        return jsonify({"message": f"Помилка додавання
проєкту: {str(e)}"}), 500
```

```
@project_bp.route('/<int:project_id>',
methods=['DELETE'])
@jwt_required()
def delete_project(project_id):
    try:
        cursor = db.connection.cursor()
        # Видаляємо завдання, пов’язані з проєктом
        cursor.execute("DELETE FROM tasks WHERE
project_id = %s", (project_id,))
        # Видаляємо сам проєкт
        cursor.execute("DELETE FROM projects WHERE id
= %s", (project_id,))
        db.connection.commit()
        cursor.close()

        return jsonify({"message": "Проєкт та всі його
завдання видалено"}), 200

    except Exception as e:
        return jsonify({"message": f"Помилка видалення
проєкту: {str(e)}"}), 500


@project_bp.route('/<int:project_id>/members',
methods=['POST'])
@jwt_required()
def add_project_member(project_id):
    """
    JSON body: { "user_id": <int> }
    Додає користувача (user_id) до project_members
(project_id, user_id).
    """
    try:
        data = request.get_json()
        user_id = data.get('user_id')
        if not user_id:
            return jsonify({"message": "user_id
обов'язковий"}), 400

        cursor = db.connection.cursor()

        # Перевіряємо, чи вказаний користувач існує
        cursor.execute("SELECT id FROM users WHERE id
= %s", (user_id,))
        found_user = cursor.fetchone()
        if not found_user:
            return jsonify({"message": "Користувача з таким
id не знайдено"}), 404

        # Перевіряємо, чи проєкт існує
        cursor.execute("SELECT id FROM projects WHERE
id = %s", (project_id,))
        found_project = cursor.fetchone()
        if not found_project:
            return jsonify({"message": "Проєкт не
знайдено"}), 404

        # Перевіряємо, чи цей user вже є у project_members
        cursor.execute("""
            SELECT id FROM project_members
            WHERE project_id = %s AND user_id = %s
        """, (project_id, user_id))
        existing = cursor.fetchone()
        if existing:
            return jsonify({"message": "Цей користувач уже є
учасником проєкту"}), 400

        # Додаємо user_id у проєкт
        cursor.execute("""
            INSERT INTO project_members (project_id,
user_id)
            VALUES (%s, %s)
        """, (project_id, user_id))
        db.connection.commit()
        cursor.close()

        return jsonify({"message": "Учасника додано
успішно"}), 201

    except Exception as e:
        return jsonify({"message": f"Помилка додавання
учасника: {str(e)}"}), 500


@project_bp.route('/<int:project_id>/members',
methods=['GET'])
@jwt_required()
def get_project_members(project_id):

    try:
        cursor = db.connection.cursor()
        query = """
        SELECT u.id, u.username, u.email
        FROM users u
        JOIN project_members pm ON u.id = pm.user_id
        WHERE pm.project_id = %s
        """
        cursor.execute(query, (project_id,))
        members = cursor.fetchall()
        cursor.close()

        return jsonify([
            {"id": m["id"], "username": m["username"],
"email": m["email"]}
            for m in members
        ]), 200
    except Exception as e:
        return jsonify({"message": f"Помилка отримання
учасників: {str(e)}"}), 500
```

Вихідний код файлу risk_routes.py

```
from flask import Blueprint, request, jsonify
from flask_jwt_extended import jwt_required,
get_jwt_identity
from services.risk_analyzer import RiskAnalyzer

risk_bp = Blueprint('risk', __name__)

@risk_bp.route('/<int:project_id>', methods=['GET'])
@jwt_required()
def get_risk(project_id):
    try:

        risk_data = RiskAnalyzer.analyze_risks(project_id)

        return jsonify(risk_data), 200

    except Exception as e:
        print(f"❌Помилка в get_risk
(project_id={project_id}): {e}")
        return jsonify({"message": f"Помилка аналізу
ризику: {str(e)}"}), 500
```

Вихідний код файлу task_routes.py

```
from flask import Blueprint, request, jsonify
from flask_jwt_extended import jwt_required
from db import db

task_bp = Blueprint('tasks', __name__)

@task_bp.route('/', methods=['GET'])
@jwt_required()
def get_tasks():
    try:
        print("🔄 Виконую підключення до бази для отримання завдань...")
        cursor = db.connection.cursor()
        cursor.execute("""
            SELECT id, project_id, user_id, description, status, deadline
            FROM tasks
        """)
        tasks = cursor.fetchall()
        print(f"📋 Отримано завдання: {tasks}")

        cursor.close()

        if not tasks:
            return jsonify({"message": "Завдань поки що немає"}), 200

        return jsonify([{
            "id": t["id"],
            "project_id": t["project_id"],
            "user_id": t["user_id"],
            "description": t["description"],
            "status": t["status"],
            "deadline": t["deadline"]
        } for t in tasks])

    except Exception as e:
        print(f"❌ ПОМИЛКА get_tasks: {str(e)}")
        return jsonify({"message": f"Помилка отримання завдань: {str(e)}"}), 500

@task_bp.route('/', methods=['POST'])
@jwt_required()
def add_task():
    """
    Очікує JSON:
    {
      "project_id": <int>,
      "user_id": <int>,      # Користувач, призначений на завдання
      "description": <str>,
      "deadline": <str YYYY-MM-DD>
    }
    """
    try:
        data = request.get_json()
        project_id = data.get('project_id')
        user_id = data.get('user_id')
        description = data.get('description', '').strip()
        deadline = data.get('deadline', '').strip()

        if not project_id or not user_id or not description or not deadline:
            return jsonify({"message": "Усі поля (project_id, user_id, description, deadline) обов'язкові!"}), 400

        cursor = db.connection.cursor()

        cursor.execute("SELECT id FROM projects WHERE id = %s", (project_id,))
        if not cursor.fetchone():
            return jsonify({"message": "Проєкт із вказаним ID не існує"}), 400

        cursor.execute("""
            INSERT INTO tasks (project_id, user_id, description, deadline)
            VALUES (%s, %s, %s, %s)
        """, (project_id, user_id, description, deadline))
        db.connection.commit()
        cursor.close()

        return jsonify({"message": "Завдання додано успішно"}), 201

    except Exception as e:
        return jsonify({"message": f"Помилка додавання завдання: {str(e)}"}), 500

@task_bp.route('/<int:task_id>', methods=['PUT'])
@jwt_required()
def update_task_status(task_id):
    try:
        data = request.get_json()
        new_status = data.get('status')

        if new_status not in ['виконується', 'завершено', 'прострочено']:
            return jsonify({"message": "Некоректний статус завдання"}), 400

        cursor = db.connection.cursor()
        cursor.execute("UPDATE tasks SET status = %s WHERE id = %s", (new_status, task_id))
        db.connection.commit()
        cursor.close()

        return jsonify({"message": "Статус завдання оновлено"}), 200

    except Exception as e:
        return jsonify({"message": f"Помилка оновлення статусу: {str(e)}"}), 500

@task_bp.route("/<int:task_id>/", methods=["DELETE"])
@jwt_required()
def delete_task(task_id):
    try:
        cursor = db.connection.cursor()

        cursor.execute("DELETE FROM tasks WHERE id = %s", (task_id,))
        db.connection.commit()

        if cursor.rowcount == 0:
            cursor.close()
            return jsonify({"message": "Завдання не знайдено"}), 404

        cursor.close()
        return jsonify({"message": "Завдання видалено"}), 200

    except Exception as e:
        db.connection.rollback()
```

```
        print(f"✗DELETE task {task_id}: {e}")
        return jsonify({"message": "Помилка сервера"}), 500
```

Вихідний код файлу user_routes.py

```
from flask import Blueprint, jsonify
from flask_jwt_extended import jwt_required
from db import db

user_bp = Blueprint('users', __name__, url_prefix='/users')

@user_bp.route('/', methods=['GET'])
@jwt_required()
def get_all_users():
    """
    Повертає список усіх користувачів (id, username,
email).
    Можна обмежити чи фільтрувати, залежно від
логіки.
    """
    try:
        cursor = db.connection.cursor()
        cursor.execute("SELECT id, username, email FROM
users")
        users = cursor.fetchall()
        cursor.close()

        user_list = [
            {"id": u["id"], "username": u["username"], "email":
u["email"]}
            for u in users
        ]
        return jsonify(user_list), 200
    except Exception as e:
        return jsonify({"message": f"Помилка отримання
користувачів: {str(e)}"}), 500
```

Вихідний код файлу risk_analyzer.py

```
from db import db
from datetime import datetime, timedelta, date

class RiskAnalyzer:
    @staticmethod
    def analyze_risks(project_id):
        cursor = db.connection.cursor()
        cursor.execute("SELECT deadline, status FROM tasks
WHERE project_id = %s", (project_id,))
        tasks = cursor.fetchall()
        cursor.close()

        risks = {"low": 0, "medium": 0, "high": 0}
        today = date.today()

        for task in tasks:
            deadline, status = task

            if not deadline:

                risks["high"] += 1
                continue

            if isinstance(deadline, datetime):
                deadline_date = deadline.date()
            elif isinstance(deadline, date):
                deadline_date = deadline
            else:
                try:

                    deadline_date =
datetime.strptime(str(deadline), "%Y-%m-%d").date()
                except ValueError as e:

                    print(f"□ Некоректний формат дати у
завданні (deadline={deadline}): {e}")

                    risks["high"] += 1
                    continue

            if status == "прострочено":
                risks["high"] += 1
            elif deadline_date < today + timedelta(days=2):
                risks["high"] += 1
            elif deadline_date < today + timedelta(days=5):
                risks["medium"] += 1
            else:
                risks["low"] += 1

        total_tasks = sum(risks.values())
        if total_tasks == 0:
            return {
                "risk_level": "Немає завдань",
                "recommendation": "Додайте нові завдання"
            }

        if risks["high"] / total_tasks > 0.5:
            return {
                "risk_level": "Високий",
                "recommendation": "Перерозподіліть завдання
або продовжіть дедлайни"
            }
        elif risks["medium"] / total_tasks > 0.5:
            return {
                "risk_level": "Середній",
                "recommendation": "Контролюйте виконання
завдань"
            }
        else:
            return {
                "risk_level": "Низький",
                "recommendation": "Проєкт іде за планом"
            }
```

Вихідний код файлу index.html

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
initial-scale=1.0"/>
    <title>Вхід | Project Tracker</title>

    <!-- Приклад мінімального вбудованого стилю
(можна винести в styles.css) -->
    <style>
        body {
            margin: 0;
            padding: 0;
            background: #f5f7fb; /* приємний фон */
            font-family: "Segoe UI", Tahoma, Geneva,
Verdana, sans-serif;
            color: #333;
```

```
}
header {
    background: #3f51b5;
    color: #fff;
    padding: 15px 20px;
}
header h1 {
    margin: 0;
    font-size: 1.5rem;
}

.container {
    max-width: 400px;
    margin: 40px auto;
    padding: 20px 25px;
    background: #fff;
    border-radius: 8px;
    box-shadow: 0 2px 5px rgba(0,0,0,0.15);
}

.container h2 {
    margin-top: 0;
    color: #3f51b5;
    text-align: center;
}

.form-block {
    margin-bottom: 2rem;
    border-top: 1px solid #ddd;
    padding-top: 1.5rem;
}

.form-block h3 {
    margin-top: 0;
    color: #555;
    margin-bottom: 1rem;
}

input[type="text"],
input[type="email"],
input[type="password"] {
    width: 100%;
    margin: 8px 0;
    padding: 10px;
    box-sizing: border-box;
    font-size: 0.95rem;
    border: 1px solid #ccc;
    border-radius: 4px;
}

button {
    background: #3f51b5;
    border: none;
    color: #fff;
    padding: 10px 18px;
    border-radius: 4px;
    cursor: pointer;
    font-size: 0.95rem;
    width: 100%;
    margin-top: 5px;
}

button:hover {
    background: #303f9f;
}

/* Адаптивність */
@media (max-width: 480px) {
    .container {
        margin: 20px;
        padding: 15px;
    }
  }
</style>
</head>

<body>
<!-- Заголовок (за бажанням можна забрати) -->
<header>
  <h1>Project Tracker</h1>
</header>

<div class="container">
  <h2>Логін / Реєстрація</h2>

  <!-- Блок реєстрації -->
  <div id="register" class="form-block">
    <h3>Реєстрація</h3>
    <input type="text" id="reg-username"
placeholder="Ім'я" />
    <input type="email" id="reg-email"
placeholder="Email" />
    <input type="password" id="reg-password"
placeholder="Пароль" />
    <button
onclick="register()">Зареєструватися</button>
  </div>

  <!-- Блок входу -->
  <div id="login" class="form-block">
    <h3>Вхід</h3>
    <input type="email" id="login-email"
placeholder="Email" />
    <input type="password" id="login-password"
placeholder="Пароль" />
    <button onclick="login()">Увійти</button>
  </div>
</div>

<!-- Підключення скрипту авторизації -->
<script src="js/auth.js"></script>
</body>
</html>
```