МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ І ПРИКЛАДНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

РИЖКОВ ОЛЕГ ОЛЕКСІЙОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій,
д-р. техн. наук, професор
_____Наталія ВЕСЕЛОВСЬКА
«____»_____ 2025 р.

ІНТЕЛЕКТУАЛЬНА СИСТЕМА ПІДТРИМКИ ПРИЙНЯТТЯ РІШЕНЬ ДЛЯ ОРЕНДИ НЕРУХОМОСТІ

Спеціальність 122 «Комп’ютерні науки»

Кваліфікаційна (магістерська) робота

Науковий керівник:
Надія ПОТАПОВА,
доцент кафедри інформаційних технологій,
к. е. н., доцент

_____
(підпис)

Оцінка: _____ / _____ / _____
(бали/за шкалою ЄКТС/за національною шкалою)

Голова ЕК: _____
(підпис)

Вінниця – 2025

АНОТАЦІЯ

Рижков О. О. Інтелектуальна система підтримки прийняття рішень для оренди нерухомості. Спеціальність 122 «Комп'ютерні науки». Освітня програма Комп'ютерна обробка даних (Data Science). Донецький національний університет імені Василя Стуса, Вінниця, 2025.

Магістерська робота присвячена розробці інтелектуальної системи оренди та бронювання офісних приміщень з використанням технологій обробки природної мови на основі великих мовних моделей. У дослідженні використано сучасні підходи до інтеграції штучного інтелекту в системи управління орендою та бронюваннями, алгоритми виявлення конфліктів у розкладі та методи програмної інженерії. Розроблено архітектуру повнофункціональної вебплатформи з AI-асистентом, який дозволяє користувачам бронювати робочі простори за допомогою природномовних запитів. Основними інструментами розробки стали Next.js 16, TypeScript, PostgreSQL 15, Prisma ORM, OpenAI GPT-4o-mini, Vercel AI SDK, архітектурний патерн Feature-Sliced Design.

Робота складається зі вступу, трьох розділів, висновків та додатків. У першому розділі досліджено теоретичні основи систем оренди офісних просторів, штучного інтелекту та обробки природної мови, алгоритмів планування та виявлення конфліктів. У другому розділі представлено результати проєктування архітектури системи, модуля обробки природної мови, алгоритмів виявлення конфліктів та застосування патерну Feature-Sliced Design. У третьому розділі викладено результати практичної реалізації платформи Prostir, вибору технологій та інструментів, архітектурних рішень, програмної реалізації ключових компонентів та результати тестування й валідації системи.

Ключові слова: інтелектуальна система, штучний інтелект, мовні моделі, вебплатформа, архітектура, алгоритми конфліктів.

98 с., 12 рис., 5 дод., 52 джерела.

ABSTRACT

Ryzhkov O.O. Development of universal web applications based on the decision-making system. Specialization 122 "Computer Science", educational program "Data Science", Vasyl' Stus Donetsk National University, Vinnytsia, 2025.

The master's thesis is devoted to the development of an intelligent office booking system using natural language processing technologies based on large language models. The study used modern approaches to integrating artificial intelligence into booking management systems, scheduling conflict detection algorithms, and software engineering methods. A full-featured web platform architecture with an AI assistant was developed, which allows users to book workspaces using natural language queries. The main development tools were Next.js 16, TypeScript, PostgreSQL 15, Prisma ORM, OpenAI GPT-4o-mini, Vercel AI SDK, and the Feature-Sliced Design architectural pattern.

The work consists of an introduction, three chapters, conclusions, and appendices. The first chapter examines the theoretical foundations of office space booking systems, artificial intelligence and natural language processing, scheduling and conflict detection algorithms, and efficiency evaluation criteria. The second chapter presents the results of system architecture design, natural language processing module design, conflict detection algorithms, and the application of the Feature-Sliced Design pattern. The third chapter presents the results of the practical implementation of the Prostir platform, the selection of technologies and tools, architectural decisions, software implementation of key components, and the results of system testing and validation.

Keywords: intelligent system, artificial intelligence, language models, web platform, architecture, conflict algorithms.

98 p., 12 fig., 52 sources.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ACID – Atomicity, Consistency, Isolation, Durability (атомарність, узгодженість, ізольованість, довговічність)

AI – Artificial Intelligence (штучний інтелект)

API – Application Programming Interface (інтерфейс програмування додатків)

CDN – Content Delivery Network (мережа доставки контенту)

CORS – Cross-Origin Resource Sharing (спільний доступ до ресурсів між різними джерелами)

CPU – Central Processing Unit (центральний процесор)

CRA – Create React App

CRUD – Create, Read, Update, Delete (створити, прочитати, оновити, видалити)

CSR – Client-Side Rendering (рендеринг на стороні клієнта)

CSRF – Cross-Site Request Forgery (підробка міжсайтових запитів)

CSS – Cascading Style Sheets (каскадні таблиці стилів)

CUID – Collision-resistant Unique Identifier (стійкий до колізій унікальний ідентифікатор)

DOM – Document Object Model (об’єктна модель документа)

ER – Entity-Relationship (сутність-зв’язок)

FK – Foreign Key (зовнішній ключ)

FSD – Feature-Sliced Design (дизайн, розділений за функціями)

GDPR – General Data Protection Regulation (загальний регламент захисту даних)

GIN – Generalized Inverted Index (узагальнений інвертований індекс)

GPT – Generative Pre-trained Transformer (генеративний попередньо навчений трансформер)

HTML – HyperText Markup Language (мова розмітки гіпертексту)

HTTP – HyperText Transfer Protocol (протокол передачі гіпертексту)

ID – Identifier (ідентифікатор)

IDE – Integrated Development Environment (інтегроване середовище розробки)

ISO – International Organization for Standardization (міжнародна організація зі стандартизації)

ISR – Incremental Static Regeneration (інкрементальна статична регенерація)

JSON – JavaScript Object Notation (нотація об’єктів JavaScript)

JSONB – JSON Binary (двійковий JSON)

JSX – JavaScript XML (розширення синтаксису JavaScript)

JWT – JSON Web Token (веб-токен JSON)

LLM – Large Language Model (велика мовна модель)

LSTM – Long Short-Term Memory (довга короткострокова пам’ять)

ML – Machine Learning (машинне навчання)

MVC – Model-View-Controller (модель-вигляд-контролер)

MVVM – Model-View-ViewModel (модель-вигляд-модель подання)

MVP – Minimum Viable Product (мінімально життєздатний продукт)

NER – Named Entity Recognition (розпізнавання іменованих сутностей)

NLP – Natural Language Processing (обробка природної мови)

OAuth – Open Authorization (відкрита авторизація)

ORM – Object-Relational Mapping (об’єктно-реляційне відображення)

PK – Primary Key (первинний ключ)

RAM – Random Access Memory (оперативна пам’ять)

REST – Representational State Transfer (передача репрезентативного стану)

ROI – Return on Investment (окупність інвестицій)

RPS – Requests Per Second (запитів на секунду)

SDK – Software Development Kit (комплект для розробки програмного забезпечення)

SPA – Single Page Application (односторінковий додаток)

SQL – Structured Query Language (мова структурованих запитів)

SSE – Server-Sent Events (події, надіслані сервером)

SSG – Static Site Generation (генерація статичного сайту)

SSR – Server-Side Rendering (рендеринг на стороні сервера)

TLS – Transport Layer Security (безпека транспортного рівня)

UI – User Interface (користувацький інтерфейс)

UML – Unified Modeling Language (уніфікована мова моделювання)

URL – Uniform Resource Locator (уніфікований локатор ресурсів)

UTC – Coordinated Universal Time (всесвітній координований час)

UX – User Experience (користувацький досвід)

XSS – Cross-Site Scripting (міжсайтовий скриптинг)

ШІ – Штучний інтелект

ВСТУП

Актуальність теми. Сучасні інформаційні технології стрімко розвиваються, і одним із найважливіших напрямів цього розвитку є інтеграція штучного інтелекту в бізнес-процеси для підвищення ефективності та зручності користування. У контексті пандемії COVID-19 та переходу до гібридних моделей роботи, ефективне управління офісними просторами стало критичною необхідністю для сучасних організацій. Розробка інтелектуальних систем оренди та бронювання офісних приміщень з використанням технологій обробки природної мови є актуальною темою в умовах зростаючої потреби у гнучкому використанні робочих просторів та автоматизації процесів бронювання.

З розвитком великих мовних моделей (Large Language Models, LLM) та технологій обробки природної мови (Natural Language Processing, NLP) з’явилися нові можливості для створення інтуїтивних інтерфейсів взаємодії користувачів із системами управління. Традиційні системи оренди та бронювання вимагають від користувачів заповнення багатьох форм, вибору параметрів із випадаючих списків та ручного перевірення доступності приміщень. Впровадження AI-асистентів дозволяє користувачам формулювати запити природною мовою, наприклад: "Забронюй переговорну на 10 осіб на завтра з 14:00 до 16:00" або "Мені потрібен тихий кабінет для роботи щопонеділка з 9:00 до 12:00 протягом місяця". Це значно скорочує час на оформлення бронювання та підвищує задоволеність користувачів системою.

Метою магістерської роботи є розробка інтелектуальної системи оренди офісних приміщень з інтеграцією AI-асистента на основі технологій обробки природної мови, алгоритмів виявлення конфліктів та підтримкою складних рекурентних бронювань для підвищення ефективності використання робочих просторів.

Об’єктом дослідження є системи управління орендою та

бронюваннями офісних приміщень з інтегрованими технологіями штучного інтелекту та обробки природної мови.

Предметом дослідження є методи та алгоритми інтеграції технологій штучного інтелекту у системи оренди, алгоритми виявлення конфліктів для одноразових, багатоденних та рекурентних бронювань, а також архітектурні підходи до побудови повнофункціональних вебплатформ.

Для досягнення поставленої мети були визначені наступні завдання:

1. Провести аналіз існуючих систем оренди та бронювання офісних приміщень, їх переваг, недоліків та ключових функціональних вимог.

2. Дослідити теоретичні основи штучного інтелекту, обробки природної мови, великих мовних моделей та принципів їх інтеграції у бізнес-процеси.

3. Розробити алгоритми виявлення конфліктів орендита та бронювань для одноразових, багатоденних та рекурентних резервацій з урахуванням різних типів повторювань (щоденно, щотижнево, щомісячно, за днями тижня).

4. Спроєктувати архітектуру вебплатформи, модуль обробки природної мови та інтеграцію з API штучного інтелекту.

5. Реалізувати повнофункціональну вебплатформу з інтегрованим AI-асистентом, системою управління базами даних та модулем виявлення конфліктів бронювань.

6. Провести тестування розробленої системи, оцінити точність розпізнавання природномовних запитів, ефективність алгоритмів виявлення конфліктів та загальну продуктивність платформи.

У процесі дослідження використовувалися такі методи:

– Аналіз літературних джерел та існуючих рішень для визначення сучасних підходів до розробки систем оренди та бронювання з інтеграцією штучного інтелекту в бізнес-процеси.

– Методи обробки природної мови (Natural Language Processing, NLP), зокрема розпізнавання іменованих сутностей (Named Entity Recognition, NER), визначення намірів користувача (Intent Detection) та екстракція параметрів запиту (Slot Filling).

– Алгоритмічні методи планування та виявлення конфліктів (Scheduling Algorithms, Conflict Detection), включаючи перевірку перетину часових інтервалів, генерацію рекурентних дат та динамічний розрахунок доступності приміщень.

– Методи об’єктно-орієнтованого проєктування та сучасні архітектурні патерни для забезпечення масштабованості, підтримуваності та тестованості системи.

– Тестування функціональності, продуктивності та зручності використання для оцінки роботи системи в умовах реальних даних та користувацьких сценаріїв.

Наукова новизна полягає в розробці комплексного підходу до інтеграції великих мовних моделей у системи управління орендою офісних приміщень. Вперше запропоновано архітектурне рішення для обробки природномовних запитів із автоматичним розпізнаванням параметрів бронювання (дата, час, тривалість, тип приміщення, місткість, зручності) та генерацією структурованих відповідей для подальшої обробки системою. Розроблено алгоритм виявлення конфліктів бронювань орендованих приміщень, який підтримує складні рекурентні паттерни та забезпечує запобігання конфліктам при одночасному резервуванні одного приміщення кількома користувачами. Впроваджено модульну архітектуру для організації кодової бази, що забезпечує чітке розділення відповідальності між компонентами системи та підвищує її підтримуваність.

Структура роботи. Магістерська робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків. У першому розділі досліджено теоретичні основи систем оренди та бронювання офісних просторів, штучного інтелекту та обробки природної мови, алгоритми планування та виявлення конфліктів, а також критерії оцінки ефективності систем бронювання. У другому розділі представлено результати проєктування архітектури системи, модуля обробки природної мови, алгоритмів виявлення конфліктів та застосування сучасних архітектурних патернів. У третьому

розділі викладено результати практичної реалізації вебплатформи, вибору технологій та інструментів, програмної реалізації ключових компонентів, а також результати тестування та валідації системи.

Практична цінність отриманих результатів полягає в розробці повнофункціональної вебплатформи з інтегрованим AI-асистентом, яка дозволяє скоротити час на оформлення оренди та бронювання офісних приміщень на 70% порівняно з традиційними формами заповнення, забезпечує запобігання конфліктам завдяки алгоритмам перевірки доступності в реальному часі та підтримує складні рекурентні паттерни оренди та бронювань. Розроблена система може бути впроваджена в реальні бізнес-процеси сучасних компаній, коворкінгів, університетів та інших організацій, що потребують ефективного управління робочими просторами в умовах гібридної моделі роботи. Використання модульної архітектури забезпечує легке масштабування системи, додавання нових функцій та підтримку кодової бази командою розробників.

Апробація результатів дослідження. Результати даного дослідження було апробовано в доповіді «Інтелектуальна система підтримки прийняття рішень оренди нерухомості» на VI Всеукраїнській науково-практичній конференції «Комп’ютерні технології обробки даних» (м. Вінниця, 5 грудня 2025 року).

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ СИСТЕМ ОРЕНДИ ТА БРОНЮВАННЯ ОФІСНИХ ПРИМІЩЕНЬ З ІНТЕГРАЦІЄЮ ШТУЧНОГО ІНТЕЛЕКТУ

1.1. Сутність та теоретичні основи систем оренди та бронювання офісних просторів

Системи оренди та бронювання офісних приміщень – це програмні комплекси, призначені для автоматизації процесів резервування робочих просторів у сучасних організаціях. Вони розробляються з урахуванням принципів ефективного управління ресурсами, що забезпечує оптимальне використання офісних площ у контексті гібридної моделі роботи.

Основною особливістю сучасних систем оренди та бронювання є їхня здатність працювати в режимі реального часу, забезпечуючи миттєву перевірку доступності приміщень та запобігання конфліктам при одночасних запитах від кількох користувачів. Це досягається завдяки використанню сучасних баз даних, алгоритмів виявлення конфліктів та інтеграції з іншими корпоративними системами.

Класифікація офісних просторів є важливим аспектом для розуміння специфіки систем. Сучасні офісні простори можна класифікувати за кількома критеріями. За типом використання:

- Переговорні кімнати (Meeting Rooms) – приміщення для проведення зустрічей, презентацій та нарад. Зазвичай обладнані проєкторами, екранами, системами відеоконференцзв’язку.
- Приватні офіси (Private Offices) – окремі кабінети для індивідуальної роботи або невеликих команд. Забезпечують конфіденційність та зосередженість.
- Робочі місця (Hot Desks) – гнучкі робочі місця в open space, які можуть бути зайняті будь-яким співробітником.
- Переговорні зони (Huddle Rooms) – невеликі простори для швидких нарад 2-4 осіб.

- Фокус-зони (Focus Rooms) – тихі приміщення для концентрованої роботи без відволікань.

За місткістю:

- Малі приміщення (2-4 особи)
- Середні приміщення (5-10 осіб)
- Великі приміщення (11-20 осіб)
- Конференц-зали (понад 20 осіб)

За оснащенням:

- Базове обладнання (стіл, стільці, дошка)
- Стандартне обладнання (+ проєктор, екран)
- Повне обладнання (+ відеоконференція, інтерактивна дошка, система озвучення)

Еволюція систем оренди та бронювання офісних приміщень відображає загальний розвиток інформаційних технологій та зміну підходів до організації робочого простору. Можна виділити кілька етапів розвитку:

Етап 1. Паперові системи (1970-1990-ті роки). Оренда та бронювання здійснювалося через спеціальні журнали реєстрації або картки на дверях приміщень. Основні недоліки: відсутність централізованого обліку, висока ймовірність конфліктів, неможливість оренди заздалегідь, відсутність аналітики використання приміщень.

Етап 2. Електронні таблиці та календарі (1990-2000-ні роки). З поширенням персональних комп’ютерів та офісних пакетів (Microsoft Office, Lotus Notes) організації почали використовувати електронні таблиці Excel або загальні календарі Outlook для координації бронювань. Переваги: централізований доступ через локальну мережу, можливість оренди та бронювання заздалегідь, базова перевірка доступності. Недоліки: відсутність автоматичної перевірки конфліктів, складність інтеграції з іншими системами, обмежена аналітика.

Етап 3. Спеціалізовані системи оренди та бронювання (2000-2010-ні роки). Розвиток вебтехнологій призвів до появи спеціалізованих систем

оренди та бронювання приміщень, таких як EMS (Event Management System), Robin, Condeco. Переваги: автоматична перевірка доступності, інтеграція з корпоративними календарями (Exchange, Google Calendar), мобільні додатки, аналітика використання приміщень, інтеграція з системами контролю доступу. Недоліки: необхідність ручного заповнення форм, складний інтерфейс для нових користувачів, відсутність інтелектуальних рекомендацій.

Етап 4. Інтелектуальні системи з AI-асистентами (2020-ні роки – сьогодення). З появою великих мовних моделей та технологій обробки природної мови з’явилася можливість створення інтуїтивних систем оренди та бронювання з AI-асистентами. Переваги: природномовний інтерфейс, інтелектуальні рекомендації на основі історії бронювань, автоматичне розпізнавання параметрів запиту (дата, час, тривалість, місткість, зручності), підтримка складних рекурентних паттернів, персоналізований досвід користування.

Вимоги до сучасних систем оренди та бронювання офісних приміщень формуються на основі потреб користувачів та бізнесу. Основні функціональні вимоги:

Автоматизація процесу оренди та бронювання. Система повинна забезпечувати швидке та зручне бронювання приміщень без необхідності звернення до адміністратора або підтримки.

Перевірка доступності в реальному часі. Система повинна надавати актуальну інформацію про доступність приміщень та автоматично запобігати конфліктам при одночасних запитах.

Підтримка різних типів бронювань. Система повинна підтримувати одноразові, багатоденні та рекурентні бронювання (щоденно, щотижнево, щомісячно, за вибраними днями тижня).

Гнучке управління бронюваннями. Користувачі повинні мати можливість переглядати, редагувати та скасовувати свої бронювання.

Інтеграція з корпоративними системами. Система повинна інтегруватися з календарями (Google Calendar, Microsoft Outlook), системами

контролю доступу, платіжними системами.

Аналітика та звітність. Адміністратори повинні мати доступ до статистики використання приміщень, популярних часових слотів, завантаженості по днях тижня.

Багатокористувацькість та розмежування прав. Система повинна підтримувати різні ролі користувачів (звичайний користувач, адміністратор, власник будівлі) з відповідними правами доступу.

Нетехнічні вимоги включають продуктивність (система повинна обробляти запити за менше ніж 2 секунди), масштабованість (підтримка 1000+ одночасних користувачів), надійність (доступність 99.9% часу), безпека (захист персональних даних, шифрування передачі даних), юзабіліті (інтуїтивний інтерфейс, доступність для користувачів різного рівня технічної підготовки).

Переваги автоматизованих систем оренди та бронювання є значними для сучасних організацій:

Підвищення ефективності використання офісних площ. Аналітика дозволяє виявити приміщення з низькою завантаженістю та оптимізувати їх використання. За дослідженнями, автоматизовані Системи оренди та бронювання дозволяють підвищити ефективність використання офісних площ на 30-40%.

Економія часу співробітників. Автоматизація процесу оренди та бронювання скорочує час на пошук вільного приміщення з 15-20 хвилин до 2-3 хвилин. Інтеграція з календарями усуває необхідність дублювати інформацію в різних системах.

Запобігання конфліктам. Автоматична перевірка доступності виключає ситуації подвійної оренди та бронювання. Система нагадувань знижує кількість невикористаних бронювань.

Прозорість та підзвітність. Централізований облік всіх бронювань. Можливість аудиту використання приміщень. Справедливий розподіл ресурсів між підрозділами.

Покращення досвіду користувачів. Зручний інтерфейс (веб, мобільний додаток, AI-асистент). Персоналізовані рекомендації на основі історії бронювань. Можливість швидкого повторного бронювання улюблених приміщень.

Недоліки існуючих рішень стимулюють розробку нових підходів:

Складність інтерфейсу. Багато існуючих систем мають перевантажений інтерфейс з численними формами та опціями. Новим користувачам потрібен час для навчання роботі з системою.

Відсутність інтелектуальних рекомендацій. Більшість систем просто показують список доступних приміщень без урахування специфічних потреб користувача. Відсутність аналізу історії бронювань для персоналізації досвіду.

Обмежена підтримка складних рекурентних бронювань. Деякі системи не підтримують або погано підтримують складні рекурентні паттерни (наприклад, "кожного понеділка та середи з 14:00 до 18:00, окрім свят").

Висока вартість впровадження та підтримки. Корпоративні рішення (EMS, Condeco) мають високу вартість ліцензій. Необхідність залучення зовнішніх консультантів для налаштування та підтримки.

Відсутність природномовного інтерфейсу. Традиційні системи вимагають ручного заповнення форм, вибору параметрів із випадаючих списків. Складність формулювання складних запитів через обмеження інтерфейсу.

Приклади існуючих систем оренди та бронювання демонструють різні підходи до вирішення задачі:

Рисунок 1.1 – Логотип вебдодатку «Google Calendar»

Google Calendar – універсальний календар з можливістю бронювання

ресурсів (приміщень). Переваги: безкоштовний, інтеграція з екосистемою Google Workspace, зручний мобільний додаток. Недоліки: обмежена функціональність для управління приміщеннями, відсутність спеціалізованої аналітики, немає перевірки обладнання приміщень.

Рисунок 1.2 – Логотип вебдодатку «Microsoft Outlook»

Microsoft Outlook / Exchange – корпоративна система календарів з підтримкою бронювання ресурсів. Переваги: інтеграція з екосистемою Microsoft, розмежування прав доступу, підтримка делегування. Недоліки: складність налаштування, висока вартість ліцензій, обмежена мобільна функціональність.

Рисунок 1.3 – Логотип вебдодатку «Robin»

Robin – спеціалізована платформа для управління офісними просторами. Переваги: спеціалізований функціонал, інтеграція з системами контролю доступу, планшети на дверях приміщень, аналітика використання. Недоліки: висока вартість, складність впровадження, відсутність AI-асистента.

Рисунок 1.4 – Логотип вебдодатку «Condeco»

Condeco – корпоративне рішення для управління робочими просторами. Переваги: потужна аналітика, інтеграція з корпоративними системами, підтримка desk booking. Недоліки: висока вартість, складний інтерфейс, тривалий процес впровадження.

Рисунок 1.5 – Логотип вебдодатку «Joan»

Joan – апаратно-програмне рішення з планшетами на дверях приміщень. Переваги: візуальна індикація доступності, інтеграція з календарями, простота використання. Недоліки: необхідність закупівлі апаратного забезпечення, обмежена функціональність мобільного додатку.

Узагальнюючи, сучасні Системи оренди та бронювання офісних приміщень еволюціонують від простих календарів до інтелектуальних платформ з AI-асистентами. Основними викликами залишаються спрощення інтерфейсу користувача, інтеграція з корпоративними системами та забезпечення гнучкості для різних моделей використання офісних просторів.

1.2. Штучний інтелект та обробка природної мови

Штучний інтелект є галуззю інформатики, що досліджує методи створення систем, здатних виконувати завдання, що традиційно вважалися прерогативою людського інтелекту: розпізнавання образів, прийняття рішень, планування, обробку природної мови та навчання на основі досвіду.

Типи штучного інтелекту. За рівнем здібностей розрізняють вузький AI (виконання конкретних завдань), загальний AI (гіпотетична система з людським рівнем інтелекту) та супер-AI (теоретична концепція, що перевершує людину). За підходом до навчання виділяють навчання з учителем

(на розміченому наборі даних), без учителя (пошук закономірностей) та з підкріпленням (через взаємодію із середовищем).

Обробка природної мови (NLP) досліджує методи взаємодії комп'ютерів з людською мовою. Основні завдання NLP включають токенізацію, морфологічний та синтаксичний аналіз, семантичний аналіз, розпізнавання іменованих сутностей, визначення намірів, екстракцію параметрів та генерацію тексту.

Розпізнавання іменованих сутностей (NER) у системах оренди та бронювання включає виявлення дат ("завтра", "15 листопада"), часу ("14:00", "вранці"), тривалості ("2 години"), місткості ("на 10 осіб"), типу приміщення ("переговорну"), обладнання ("з проєктором") та рекурентності ("щотижня", "кожного понеділка").

Визначення намірів (Intent Detection) класифікує запити за типами дій: бронювання нового приміщення, перегляд існуючих бронювань, скасування бронювання, зміна параметрів, пошук доступних приміщень, запит інформації про приміщення.

Великі мовні моделі (LLM) є нейронними мережами, навченими на величезних обсягах текстових даних для розуміння та генерації природної мови. Архітектура Transformer (Vaswani et al., 2017) стала основою для сучасних LLM завдяки механізму уваги, багатоголовій увазі, позиційному кодуванню та архітектурі енкодер-декодер.

GPT-4o-mini характеризується оптимізованим розміром моделі, контекстним вікном 128,000 токенів, швидкістю на 60% вищою за GPT-4, значно нижчою вартістю API-викликів, якістю на рівні 82% від GPT-4 та підтримкою тексту й зображень. Вибір GPT-4o-mini для Системи оренди та бронювання обґрунтовується економічністю, швидкістю відгуку до 2 секунд, достатньою якістю розпізнавання, великим контекстним вікном та підтримкою структурованого виходу через JSON mode. Проєктування промптів (Prompt Engineering) визначає ефективність використання LLM. Принципи включають чіткість інструкцій з використанням прикладів, надання

контекстної інформації (список офісів, характеристики, поточна дата), структурування виходу через JSON-схему, визначення обмежень та правил обробки конфліктів, інструкції щодо обробки помилок та недостатньої інформації. Альтернативні підходи до NLP. Системи на основі правил використовують регулярні вирази та шаблони з повним контролем логіки, але вимагають ручного написання правил для всіх варіацій. Класичні моделі ML (BERT, LSTM) потребують розмічених навчальних даних та мають менші витрати, але обмежені можливості генерації відповідей. Платформи діалогових систем (Dialogflow, Rasa) надають готовий функціонал, але обмежену гнучкість. Гібридні підходи комбінують LLM для розуміння запитів з rule-based логікою для бізнес-правил.

Архітектура інтеграції AI включає клієнтський інтерфейс (чат-інтерфейс, відображення відповідей), AI Gateway (обробка повідомлень, побудова промпту, виклик API LLM, парсинг JSON), систему управління контекстом (історія діалогу, управління станом бронювання), модуль бізнес-логіки (перевірка доступності, виявлення конфліктів, валідація, збереження в базі даних) та інтеграцію з зовнішніми системами (календарі, платіжні системи, контроль доступу).

1.3. Алгоритми планування та виявлення конфліктів

Теорія планування вивчає методи оптимального розподілу ресурсів у часі. У системах оренди та бронювання офісних приміщень планування включає розподіл обмежених ресурсів (приміщень) між конкуруючими запитами (бронюваннями) з урахуванням часових та просторових обмежень. Основні компоненти задачі: ресурси (офіси з характеристиками), завдання (запити на оренду та бронювання з параметрами), обмеження (часові вікна, робочі години) та цільова функція (оптимізація використання, мінімізація конфліктів).

Типи бронювань. Одноразові бронювання резервують приміщення на конкретну дату та часовий проміжок з параметрами дата, час початку, час

завершення, офіс. Багатоденні бронювання резервують приміщення на послідовні дні з однаковим часовим проміжком, що вимагає генерації N одноразових бронювань. Рекурентні бронювання повторюються за певним паттерном: щоденні, щотижневі, щомісячні, за днями тижня або складні паттерни. Параметри включають тип повторювання, список днів тижня, дати початку та завершення періоду, час початку та завершення кожного входження.

Алгоритми виявлення конфліктів. Конфлікт виникає, коли два або більше бронювань претендують на одне приміщення в перетинаючийся часовий проміжок. Математична умова перетину інтервалів: два інтервали перетинаються, якщо start1 < end2 AND start2 < end1. Алгоритм перевірки конфлікту має складність O(1) для порівняння двох бронювань. Перевірка доступності офісу для нового бронювання вимагає порівняння з усіма існуючими бронюваннями офісу зі складністю O(N), де N – кількість існуючих бронювань. Оптимізація через створення B-tree індексу на поля (officeId, date, startTime) знижує складність до O(log N + K), де K – кількість бронювань для конкретного офісу в конкретний день.

Генерація рекурентних дат. Алгоритм для щоденних бронювань та оренди послідовно додає кожний день між датами початку та завершення зі складністю O(N), де N – кількість днів. Алгоритм для щотижневих бронювань знаходить перший день тижня не раніше дати початку та додає кожний тиждень зі складністю O(N/7). Алгоритм для та оренди за вибраними днями тижня перевіряє кожний день періоду на відповідність списку обраних днів зі складністю O(N). Оптимізації включають кешування згенерованих дат для однакових правил, обмеження максимальної кількості входжень та пагінацію для великих діапазонів. Обробка граничних випадків включає виключення вихідних та святкових днів, нормалізацію часових міток при зміні літнього/зимового часу, коректну обробку високосних років та збереження всіх дат у UTC з конвертацією при відображенні.

Перевірка рекурентних бронювань. Алгоритм генерує всі дати входжень

рекурентного бронювання та оренди та перевіряє кожну дату на конфлікти з існуючими бронюваннями. Складність алгоритму O(M × N), де M – кількість згенерованих дат, N – кількість існуючих бронювань. Оптимізації включають часові індекси в базі даних, batch-запити для перевірки множинних дат та паралелізацію перевірки незалежних дат.

Аналіз складності. Часова складність основних операцій: перевірка конфлікту двох бронювань O(1), перевірка доступності офісу O(N), генерація рекурентних дат O(M), перевірка рекурентного бронювання O(M × N). Просторова складність: збереження одноразового бронювання O(1), збереження рекурентного правила O(1), збереження всіх входжень O(M). Рекомендований гібридний підхід зберігає рекурентне правило в окремій таблиці та генерує конкретні входження при створенні, зв'язуючи їх з правилом для можливості централізованого редагування.

Динамічне ціноутворення. Алгоритми розраховують ціну на основі базової ціни офісу, тривалості бронювання та множників: час доби (піковий час 1.2×, непіковий 1.0×), день тижня (вихідні 0.8×), завантаженість (>80% офісів зайнято 1.3×) та знижки за тривалість (5% за >4 годин, 10% за >8 годин). Фінальна ціна обчислюється як добуток базової ціни, тривалості та всіх множників з урахуванням знижок.

Пошук доступних для оренди офісів. Алгоритм фільтрує офіси за критеріями користувача (місткість, тип, будівля), перевіряє доступність кожного офісу в потрібний час, розраховує ціну для кожного доступного офісу та сортує результати за обраним критерієм (ціна, місткість, рейтинг). Складність O(N × M), де N – кількість офісів, M – середня кількість бронювань на офіс. Оптимізації включають фільтровані індекси, кешування результатів для популярних запитів та пагінацію.

1.4. Критерії оцінки ефективності систем оренди та бронювання

Комплексна оцінка ефективності систем оренди та бронювання вимагає моніторингу метрик у кількох категоріях. Метрики продуктивності

визначають швидкість роботи системи. Час відгуку AI-асистента повинен бути менше 2 секунди для 95% запитів, перевірка доступності – менше 500 мс для простих запитів та менше 2 секунд для рекурентних, збереження бронювання – менше 1 секунди, завантаження списку офісів – менше 1 секунди. Пропускна здатність повинна забезпечувати обробку 100+ одночасних користувачів, 1000+ бронювань на годину в піковий час та 10,000+ запитів на пошук офісів на годину. Використання ресурсів обмежується середнім навантаженням CPU <70%, RAM <2 GB на інстанс, <50 одночасних з'єднань з базою даних та оптимізацією API-викликів до LLM через кешування.

Метрики точності AI оцінюють якість розпізнавання природномовних запитів. Точність екстракції параметрів повинна становити >95% для дат, >90% для часу та місткості, >85% для типу приміщення та >80% для рекурентності. Точність визначення намірів повинна досягати >95% для бронювання, >90% для перегляду та скасування, >85% для пошуку. Метрики обчислюються через точність (precision), повноту (recall) та F1-score на тестовому наборі. Обробка неоднозначності включає частку запитів, що потребують уточнення (<15%) та середню кількість уточнюючих питань (<2 на діалог). Метрики якості алгоритмів планування оцінюють коректність виявлення конфліктів. Точність виявлення повинна забезпечувати 0% помилкових конфліктів (не можна блокувати коректні бронювання), 0% пропущених конфліктів (не можна допускати подвійні бронювання) та 100% правильно виявлених конфліктів. Ефективність використання офісів включає середню завантаженість 60-80%, частку незавершених бронювань <10% та середню тривалість залежно від типу приміщення.

Метрики користувацького досвіду оцінюють зручність взаємодії. Час на виконання завдань повинен становити <2 хвилини для одноразового бронювання, <3 хвилини для рекурентного, <1 хвилини для пошуку та <30 секунд для скасування. Частота успішного виконання включає >85% користувачів, що створили бронювання з першої спроби та >90%, що знайшли потрібний офіс. Задоволеність визначається оцінкою >4.0 (шкала 1-5), Net

Promoter Score >50 та часткою користувачів AI-асистента >60%.

Бізнес-метрики оцінюють економічну ефективність. Економія часу досягає 70-85% порівняно з 15-20 хвилинами до впровадження AI проти 2-3 хвилин після. Конверсія включає >80% завершених успішно бронювань та >70% повторних користувачів. Вартість експлуатації обмежується <$0.01 на користувача на місяць для API-викликів, <$200 на місяць для інфраструктури на 1000 користувачів та ROI >200% протягом першого року. Використання системи характеризується зростанням MAU >10% щомісяця, бронювань >15% щомісяця та середньою кількістю >5 бронювань на користувача на місяць.

Метрики масштабованості оцінюють здатність обробляти зростаюче навантаження. Горизонтальна масштабованість забезпечує можливість додавання серверів без зміни архітектури, лінійне зростання продуктивності та час розгортання інстансу <5 хвилин. Вертикальна масштабованість підтримує збільшення потужності серверів та оптимізацію запитів для великих обсягів даних. Навантажувальне тестування перевіряє симуляцію 1000+ одночасних користувачів, підтримку 100,000+ бронювань та час відгуку <5 секунд при максимальному навантаженні.

Метрики надійності оцінюють стабільність системи. Доступність повинна становити 99.9% (не більше 43 хвилин простою на місяць). Стійкість до збоїв включає час відновлення <15 хвилин, автоматичне відновлення без втрати даних та щоденне резервне копіювання. Обробка помилок забезпечує graceful degradation при недоступності AI API, валідацію на клієнті та сервері та зрозумілі повідомлення про помилки.

Метрики безпеки оцінюють захищеність системи. Аутентифікація підтримує сучасні протоколи (OAuth 2.0, JWT), опціональну багатофакторну аутентифікацію та розмежування прав доступу. Захист даних включає шифрування TLS 1.3, шифрування чутливих даних у базі та відповідність GDPR. Моніторинг включає логування всіх операцій, виявлення аномальної активності та регулярні аудити. Захист від атак забезпечується rate limiting (100 req/min), CSRF protection, SQL injection prevention та XSS protection.

ВИСНОВКИ ДО РОЗДІЛУ 1

У першому розділі проведено дослідження теоретичних основ систем оренди та бронювання офісних приміщень з інтеграцією штучного інтелекту.

Проаналізовано існуючі Системи оренди та бронювання офісних приміщень та виявлено їх основні недоліки: перевантажений інтерфейс, відсутність інтелектуальних рекомендацій, обмежена підтримка складних рекурентних паттернів та відсутність природномовного інтерфейсу. Визначено класифікацію офісних приміщень та сформульовано функціональні і нетехнічні вимоги до сучасних систем.

Досліджено теоретичні основи штучного інтелекту та обробки природної мови від тесту Тюрінга до епохи великих мовних моделей. Вивчено завдання NLP у системах бронювання та оренди: розпізнавання іменованих сутностей, визначення намірів та екстракцію параметрів. Розглянуто архітектуру Transformer та еволюцію LLM. Обґрунтовано вибір GPT-4o-mini на основі критеріїв економічності, швидкості відгуку до 2 секунд, якості розпізнавання та підтримки структурованого виходу через JSON mode.

Вивчено алгоритми планування та виявлення конфліктів бронювань та оренди. Класифіковано типи бронювань (одноразові, багатоденні, рекурентні) та розроблено алгоритми виявлення конфліктів зі складністю від O(1) до O(M × N). Запропоновано оптимізації через B-tree індексування та гібридний підхід до зберігання рекурентних бронювань.

Визначено комплексну систему критеріїв оцінки ефективності: продуктивність (час відгуку <2с), точність AI (>90%), якість планування (0% конфліктів), UX (>85% успішних завершень), бізнес-показники (70-85% економії часу, ROI >200%), масштабованість (100,000+ бронювань), надійність (99.9% uptime) та безпека (TLS 1.3, OAuth 2.0, GDPR).

Результати теоретичного дослідження створюють фундамент для проєктування архітектури та реалізації інтелектуальної Системи оренди та бронювання офісних приміщень.

РОЗДІЛ 2
АРХІТЕКТУРА ТА ПРОЄКТНІ РІШЕННЯ СИСТЕМИ ОРЕНДИ

2.1 Підходи до архітектури систем оренди та бронювання офісних приміщень

Вибір архітектурного підходу є критичним рішенням при проєктуванні Системи оренди та бронювання офісних приміщень, оскільки він визначає масштабованість, підтримуваність та ефективність розробки. Існує кілька підходів до побудови архітектури вебдодатків, кожен з яких має свої переваги та недоліки в контексті систем бронювання.

Монолітна архітектура передбачає побудову додатку як єдиного цілісного блоку, де всі компоненти (клієнтська частина, серверна логіка, база даних) тісно інтегровані. Основні переваги монолітного підходу включають простоту розробки та розгортання, відсутність складної міжсервісної комунікації, легкість локального тестування та низькі початкові витрати на інфраструктуру. Недоліками є складність масштабування окремих компонентів, висока зв’язність коду, що ускладнює підтримку, ризик каскадних збоїв при виході з ладу будь-якого компонента та обмежена гнучкість у виборі технологій для різних частин системи.

Мікросервісна архітектура розбиває додаток на набір незалежних сервісів, кожен з яких відповідає за конкретну бізнес-функцію. У контексті Системи оренди та бронювання це можуть бути окремі сервіси для управління офісами, обробки оренди та бронювань, роботи з користувачами, AI-асистента та платіжної системи. Переваги мікросервісного підходу включають незалежне масштабування сервісів, технологічну гетерогенність (можливість використання різних технологій для різних сервісів), ізоляцію збоїв та можливість паралельної розробки різними командами. Недоліки охоплюють складність розгортання та моніторингу, необхідність організації міжсервісної комунікації, складність забезпечення консистентності даних та вищі вимоги до DevOps-інфраструктури.

Серверлес-архітектура (Function-as-a-Service) базується на виконанні окремих функцій у відповідь на події без необхідності управління серверною інфраструктурою. Основні переваги включають оплату тільки за фактичне використання ресурсів, автоматичне масштабування, відсутність необхідності управління серверами та швидке розгортання нових функцій. Недоліки охоплюють холодний старт функцій (затримка при першому виклику після простою), обмеження на час виконання функцій, складність локального тестування та ризик vendor lock-in при прив'язці до конкретного хмарного провайдера.

Full-stack фреймворки, такі як Next.js, пропонують гібридний підхід, де клієнтська та серверна частини інтегровані в єдиному кодовому репозиторії з підтримкою різних моделей рендерингу. Цей підхід поєднує переваги монолітної та розподіленої архітектури, забезпечуючи простоту розробки з можливістю гнучкого масштабування.

Обґрунтування вибору архітектури для системи Prostir. Для реалізації системи Prostir було обрано full-stack підхід на базі фреймворку Next.js 16 з наступних причин.

По-перше, єдина кодова база для клієнта та сервера значно спрощує розробку, оскільки дозволяє використовувати один язык програмування (TypeScript) для всього додатку, забезпечує спільне використання типів між клієнтом і сервером, що підвищує типобезпеку, та знижує когнітивне навантаження на розробників, яким не потрібно перемикатися між різними технологіями.

По-друге, підтримка Server Components та Server Actions у Next.js 16 надає можливість виконання серверного коду безпосередньо в React-компонентах, що дозволяє зменшити обсяг JavaScript на клієнті та прискорити початкове завантаження сторінки. Server Actions забезпечують типобезпечну взаємодію між клієнтом і сервером без необхідності створення окремих API endpoints, що спрощує код та зменшує boilerplate.

По-третє, гнучкість моделей рендерингу включає Server-Side Rendering

(SSR) для динамічних сторінок з актуальними даними, Static Site Generation (SSG) для статичних сторінок (landing page, документація), Client-Side Rendering (CSR) для інтерактивних компонентів (чат з AI, фільтри пошуку) та Incremental Static Regeneration (ISR) для балансу між продуктивністю та свіжістю даних.

По-четверте, вбудовані оптимізації Next.js включають автоматичне розділення коду (code splitting), оптимізацію зображень через компонент Image, автоматичну мінімізацію та стиснення ресурсів, підтримку HTTP/2 Server Push та автоматичне prefetching сторінок при наведенні на посилання.

По-п’яте, App Router в Next.js 16 надає файлову систему маршрутизації з підтримкою вкладених layouts, паралельних маршрутів (parallel routes), перехоплення маршрутів (intercepting routes) та потокової передачі UI (streaming). Це дозволяє створювати складні макети з мінімальним кодом та забезпечувати краще сприйняття користувачем завдяки прогресивній завантаженості інтерфейсу.

По-шосте, простота інтеграції з екосистемою включає нативну підтримку TypeScript без додаткової конфігурації, інтеграцію з Tailwind CSS через PostCSS, підтримку React 19 з новими можливостями (use hook, Server Components), легку інтеграцію з Prisma ORM для роботи з базою даних та підтримку Vercel AI SDK для інтеграції з LLM.

По-сьоме, масштабованість та продуктивність забезпечуються через можливість вертикального масштабування (збільшення потужності серверів), горизонтального масштабування (додавання інстансів через хмарні платформи), використання Edge Runtime для найшвидшої обробки запитів та підтримку інкрементального впровадження нових функцій без повного перезапуску.

Порівняння з альтернативами. Традиційний підхід з окремими frontend (React SPA) та backend (Node.js/Express) вимагає дублювання типів між клієнтом і сервером, створення та документування REST API або GraphQL схеми, налаштування CORS та розв’язання інших проблем міжсерверної

комунікації, а також окремого хостингу для frontend та backend. Мікросервісний підхід був би надмірним для проєкту масштабу Prostir на етапі MVP, оскільки вимагає складної оркестрації сервісів, додаткових витрат на інфраструктуру та expertise в DevOps. Серверлес-підхід (AWS Lambda, Vercel Functions) має обмеження на час виконання функцій (15 хв для Lambda, 10-60 сек для Vercel), що може бути проблемою для генерації великої кількості рекурентних бронювань, холодний старт додає затримку 100-300 мс, що критично для AI-асистента та відсутність стану між викликами ускладнює кешування.

Архітектурні патерни на рівні коду. Окрім вибору high-level архітектури (монолітна, мікросервіси, тощо), важливим є вибір патернів організації коду всередині додатку.

Model-View-Controller (MVC) є класичним патерном розділення відповідальностей, де Model відповідає за дані та бізнес-логіку, View – за відображення даних користувачу, Controller – за обробку запитів та координацію між Model і View. У контексті вебдодатків на React MVC часто трансформується в варіації типу Flux або Redux. Основні переваги MVC включають чітке розділення відповідальностей, простоту тестування окремих компонентів та широке розповсюдження патерну. Недоліки охоплюють зростання складності Controller при збільшенні функціональності, потенційну тісну зв’язність між Model і View та складність управління станом в умовах асинхронних операцій.

Model-View-ViewModel (MVVM) використовується в reactive фреймворках (Angular, Vue.js), де ViewModel виступає посередником між View і Model, забезпечуючи data binding. Основні переваги включають автоматичну синхронізацію між View та Model через data binding, зменшення boilerplate коду для управління станом та покращену тестованість через ізоляцію ViewModel. Недоліки охоплюють складність налагодження при великій кількості binding’ів, потенційні проблеми з продуктивністю при надмірному використанні реактивності та крива навчання для розробників,

незнайомих з патерном.

Hexagonal Architecture (Ports and Adapters) фокусується на ізоляції бізнес-логіки від зовнішніх залежностей через визначення портів (інтерфейсів) та адаптерів (реалізацій). Основні переваги включають високу тестованість через можливість mock'ування адаптерів, легкість заміни зовнішніх залежностей (база даних, API) та чітке визначення меж бізнес-логіки. Недоліки охоплюють високу початкову складність архітектури, збільшення кількості абстракцій та потенційне over-engineering для простих додатків.

Feature-Sliced Design (FSD) є відносно новою методологією організації коду, орієнтованою на структурування додатку за бізнес-функціями (features) замість технічних ролей (components, services). Основні переваги включають інтуїтивну організацію коду, що відповідає бізнес-логіці, легкість навігації в проєкті, оскільки всі файли, пов'язані з однією функцією, знаходяться в одному місці, масштабованість архітектури при зростанні проєкту, зменшення конфліктів у командній розробці завдяки ізоляції features та чіткі правила залежностей між шарами. FSD був обраний для проєкту Prostir і буде детально розглянутий у підрозділі 2.3.

Інтеграція AI-модуля в архітектуру. Інтеграція великої мовної моделі в архітектуру Системи оренди та бронювання вимагає врахування специфічних аспектів, таких як латентність, вартість API-викликів та обробка помилок.

Архітектурне рішення для AI-модуля включає створення окремого API Route (/api/ai-chat/route.ts), який виступає Gateway між клієнтом і OpenAI API. Це дозволяє централізовано управляти логікою взаємодії з AI, приховати API-ключі від клієнта та додати проміжну обробку запитів і відповідей.

Основні компоненти AI-модуля включають buildSystemPrompt() – функцію, яка динамічно генерує system prompt на основі поточного списку офісів з бази даних, забезпечуючи, що AI завжди оперує актуальними даними про доступні приміщення. Vercel AI SDK використовується для спрощення інтеграції з OpenAI API, надаючи функцію streamText() для потокової передачі відповідей AI, що покращує сприйняття користувачем завдяки прогресивному

відображенню тексту. Модель GPT-4o-mini обрана за критеріями економічності ($0.15 per 1M input tokens, $0.60 per 1M output tokens), швидкості відгуку (зазвичай 1-2 секунди для відповіді), достатньої якості розпізнавання природномовних запитів та підтримки великого контекстного вікна (128,000 токенів).

Патерни обробки AI-відповідей включають потокову передачу (streaming), що дозволяє відображати відповідь AI по мірі генерації токенів, покращуючи відчуття відгуку системи. JSON-режим використовується для структурованого виходу, коли AI повертає JSON-об'єкт з параметрами бронювання після підтвердження користувача. Обробка помилок включає fallback до повідомлення про помилку при недоступності OpenAI API, retry-логіку з експоненційною затримкою для тимчасових збоїв та валідацію JSON-відповідей перед передачею клієнту.

Інтеграція бази даних. Вибір системи управління базами даних та ORM є критичним для продуктивності та зручності розробки.

PostgreSQL 15 був обраний як СУБД з наступних причин. По-перше, підтримка складних типів даних, включаючи JSON для зберігання об'єкта recurrence в таблиці Booking, що дозволяє зберігати структуровані дані без необхідності створення окремих таблиць. По-друге, потужні індекси, включаючи B-tree для швидкого пошуку за датами та офісами, GIN для пошуку в JSON-полях та Composite індекси для оптимізації складних запитів. По-третє, ACID-гарантії забезпечують консистентність даних при одночасних операціях бронювання. По-четверте, транзакційність дозволяє атомарне створення множинних записів (наприклад, при бронюванні кількох офісів одночасно). По-п'яте, масштабованість через підтримку реплікації (master-slave) для розподілу навантаження читання та partitioning для роботи з великими таблицями.

Prisma ORM був обраний як інструмент для взаємодії з базою даних з наступних причин. По-перше, type-safety забезпечує автоматичну генерацію TypeScript-типів на основі схеми бази даних, що виключає runtime помилки

типізації. По-друге, декларативна схема в prisma/schema.prisma дозволяє визначити всю структуру бази даних в одному файлі з автоматичною генерацією міграцій. По-третє, міграції генеруються автоматично командою prisma migrate dev, що спрощує розгортання змін схеми. По-четверте, Prisma Studio надає графічний інтерфейс для перегляду та редагування даних без необхідності писати SQL. По-п’яте, оптимізація запитів через N+1 problem prevention (Prisma автоматично використовує JOIN замість множинних запитів), connection pooling для ефективного використання з’єднань з базою даних та query batching для об’єднання множинних запитів в один.

Структура бази даних включає чотири основні моделі: User (зберігає інформацію про користувачів), Building (інформація про будівлі), Office (дані про офіси з посиланням на Building) та Booking (бронювання з посиланням на Office та User). Використовуються чотири enum-типи: UserRole (USER, ADMIN, OWNER), OfficeType (MEETING_ROOM, PRIVATE_OFFICE, DESK, CONFERENCE_ROOM), BookingStatus (PENDING, CONFIRMED, CANCELLED, COMPLETED) та RecurrencePattern (DAILY, WEEKLY, MONTHLY).

Стратегії індексування включають індекси на зовнішні ключі (buildingId, officeId, userId) для прискорення JOIN-операцій, індекси на поля для фільтрації (type, capacity, city, status) для оптимізації пошукових запитів та індекси на дати (startDate) для швидкої перевірки доступності офісів.

Вибір хмарної платформи. Для розгортання системи Prostir були розглянуті кілька хмарних платформ.

Vercel є платформою, оптимізованою для Next.js, що забезпечує нульову конфігурацію для Next.js додатків, автоматичне розгортання при push до git-репозиторію, edge network з глобальним CDN для мінімальної латентності, вбудовану підтримку preview deployments для кожного pull request та безкоштовний tier для особистих проєктів (100 GB bandwidth, unlimited requests).

Netlify надає аналогічні можливості для статичних сайтів та серверлес-

функцій, але має обмеженнішу підтримку Next.js App Router порівняно з Vercel та менш оптимальну продуктивність для динамічних маршрутів.

AWS (Elastic Beanstalk, ECS, Lambda) забезпечує найбільшу гнучкість та контроль над інфраструктурою, але вимагає значно більше налаштувань та DevOps-expertise, має вищу складність управління та потенційно вищу вартість для малих проєктів.

Google Cloud Platform (Cloud Run, App Engine) пропонує хороший баланс між простотою та гнучкістю, має конкурентні ціни, але вимагає більше налаштувань порівняно з Vercel.

Для проєкту Prostir була обрана Vercel завдяки нульовій конфігурації, оптимізації для Next.js та безкоштовному варіанту для MVP-етапу.

2.2 Проєктування модуля обробки природної мови

Модуль обробки природної мови є ключовою інновацією системи Prostir, що відрізняє її від традиційних систем оренди та бронювання. Проєктування цього модуля вимагає ретельного планування архітектури, структури промптів та логіки обробки відповідей.

Архітектура AI-модуля. Модуль обробки природної мови реалізовано як окремий API Route в Next.js, що дозволяє ізолювати логіку взаємодії з AI від клієнтського коду та забезпечити централізоване управління API-ключами.

Компоненти модуля включають API Route (/api/ai-chat/route.ts), який виступає entry point для всіх запитів до AI, приймає HTTP POST запити з історією діалогу, викликає OpenAI API через Vercel AI SDK та повертає потокову відповідь клієнту. Функція buildSystemPrompt() відповідає за динамічну генерацію system prompt, що включає актуальний список офісів з бази даних, інструкції для AI щодо поведінки, приклади форматування відповідей та правила екстракції параметрів бронювання. Функція convertToModelMessages() перетворює повідомлення з клієнта у формат, який очікує AI SDK, обробляючи як старий формат (content як string), так і новий (content як масив parts). Клієнтський компонент AIBookingPage використовує

хук useChat з @ai-sdk/react для управління станом чату, відправки повідомлень та обробки потокових відповідей.

Потік даних в AI-модулі включає наступні кроки. Користувач вводить повідомлення в чат-інтерфейс. Клієнт відправляє POST запит до /api/ai-chat з історією діалогу. Сервер отримує запит та викликає buildSystemPrompt() для генерації актуального system prompt на основі даних з бази. Сервер викликає OpenAI API через streamText() з system prompt, історією діалогу та параметрами моделі. OpenAI API повертає потокову відповідь (streaming), яка передається клієнту по мірі генерації. Клієнт відображає відповідь AI в реальному часі, оновлюючи UI по мірі надходження токенів. Якщо AI повертає JSON з action: CREATE_BOOKING, клієнт парсить JSON та виконує перенаправлення на сторінку підтвердження бронювання.

Проєктування system prompt. System prompt є критично важливим для якості роботи AI-асистента, оскільки він визначає контекст, поведінку та формат відповідей моделі.

Структура system prompt включає кілька секцій. Роль асистента визначається формулюванням "You are a helpful office booking assistant for Prostir", що налаштовує AI на допомогу з орендою та бронюванням офісів. Список доступних офісів генерується динамічно з бази даних та організований за будівлями та поверхами, включаючи ID офісу, тип, місткість, ціну та зручності. Поточна дата надається в форматі "Today’s date: November 26, 2025", що дозволяє AI коректно інтерпретувати відносні дати типу "завтра", "наступного тижня". Приклади оренди та бронювань включають прості (конкретна дата та час), довгострокові (кілька років) та рекурентні (щотижневі, щомісячні) паттерни. JSON-схема відповіді визначає структуру об’єкта бронювання, який AI має повернути після підтвердження користувача. Правила поведінки включають вимоги завжди підтверджувати деталі перед поверненням JSON, використовувати точні ID офісів з бази, коректно обчислювати дати та час, бути розмовним, але ефективним.

Динамічна генерація списку офісів реалізована в функції

buildSystemPrompt(), яка виконує запит до бази даних через Prisma для отримання всіх офісів з інформацією про будівлі. Офіси групуються за будівлями через Map structure для ефективного пошуку. Всередині кожної будівлі офіси групуються за поверхами для логічної організації. Для кожного офісу генерується рядок з назвою, ID, типом, місткістю, ціною та зручностями. Результуючий список вставляється в system prompt, забезпечуючи, що AI завжди оперує актуальними даними.

Алгоритм генерації виглядає наступним чином. Крок 1: Отримати всі офіси з бази даних з включенням даних про будівлю (include: { building: true }). Крок 2: Створити Map для групування офісів за buildingId. Крок 3: Для кожної будівлі створити Map для групування офісів за floorNumber. Крок 4: Відсортувати поверхи за номером в ascending order. Крок 5: Для кожного офісу згенерувати рядок формату "- Office_Name (ID: office_id): Type, Capacity, $Price/hr - Amenities: list". Крок 6: Об’єднати всі рядки в єдиний список з заголовками будівель та поверхів. Складність алгоритму O(N log N), де N – кількість офісів, завдяки сортуванню поверхів.

Обробка повідомлень користувача. Клієнт використовує хук useChat з Vercel AI SDK для управління станом чату, який забезпечує автоматичне управління масивом messages з історією діалогу, функцію append() для додавання нових повідомлень, стан isLoading для індикації обробки запиту та автоматичну обробку streaming responses.

Процес відправки повідомлення включає наступні кроки. Користувач вводить текст в input field та натискає "Send" або Enter. Викликається handleSubmit() з useChat, який автоматично додає повідомлення до messages та відправляє POST запит до /api/ai-chat. Клієнт відображає повідомлення користувача в UI з індикатором завантаження. Сервер отримує запит, генерує system prompt та викликає OpenAI API. Відповідь AI stream’иться назад до клієнта та відображається по мірі надходження токенів. Після завершення stream’у клієнт перевіряє, чи містить відповідь JSON-блок з CREATE_BOOKING action.

Екстракція та обробка JSON-відповідей. Коли користувач підтверджує бронювання (говорить "yes", "confirm", "book it"), AI має повернути JSON-об’єкт з параметрами бронювання.

Формат JSON-відповіді для простого бронювання включає поля action (завжди "CREATE_BOOKING"), offices (масив ID офісів для бронювання), startDate та endDate (дати в форматі ISO), startTime та endTime (час в форматі "HH:MM"), isRecurring (false для простого бронювання), recurrence (null для простого бронювання) та notes (додаткові примітки користувача).

Формат для рекурентного бронювання та оренди додатково включає isRecurring: true та об’єкт recurrence з полями pattern ("DAILY", "WEEKLY", "MONTHLY"), daysOfWeek (масив чисел 1-7, де 1=Monday, 7=Sunday), interval (інтервал повторення, зазвичай 1) та потенційно endDate в recurrence (хоча це дублює зовнішнє endDate).

Клієнтська логіка парсингу JSON використовує effect, який відслідковує зміни в messages та шукає JSON-блоки в останньому повідомленні AI. Використовується регулярний вираз для знаходження блоків json.... JSON парситься через JSON.parse() з обробкою помилок. Якщо action === "CREATE_BOOKING", клієнт зберігає дані в sessionStorage (для передачі між сторінками) та виконує router.push(‘/booking/confirm’) для перенаправлення на сторінку підтвердження.

Обробка помилок парсингу включає try-catch блок навколо JSON.parse(), логування помилок в консоль для debugging та fallback до звичайного відображення тексту, якщо JSON некоректний. Валідація розпарсених даних перевіряє наявність обов’язкових полів (offices, startDate, endDate), коректність формату дат та часу та відповідність дат логічним обмеженням (endDate >= startDate).

Управління контекстом діалогу. Підтримка контексту діалогу є критично важливою для природної взаємодії з AI-асистентом.

Історія повідомлень зберігається в стані клієнта через хук useChat, який підтримує масив messages з об’єктами формату {role: ‘user’ | ‘assistant’, content:

string}. При кожному новому запиті до /api/ai-chat відправляється вся історія діалогу, що дозволяє AI враховувати попередній контекст. OpenAI API використовує історію для розуміння контексту та надання релевантних відповідей. Оптимізація контекстного вікна включає обмеження історії останніми N повідомленнями (наприклад, 20) для зменшення витрат на токени, видалення старих повідомлень через функцію messages.slice(-20) перед відправкою запиту та компресію повідомлень через summarization, якщо діалог стає занадто довгим.

Збереження контексту між сесіями може бути реалізоване через збереження messages в localStorage при unmount компонента, відновлення messages з localStorage при mount компонента та очищення localStorage при завершенні бронювання або явному скиданні чату.

Управління станом бронювання відстежує, чи знаходиться користувач в процесі підтвердження бронювання, чи потрібно запитати додаткову інформацію та чи було вже створено бронювання для запобігання дублікатів.

Оптимізації AI-модуля. Для забезпечення хорошої продуктивності та економічності AI-модуль включає кілька оптимізацій. Оптимізація витрат на токени включає використання GPT-4o-mini замість GPT-4 (10× дешевше при достатній якості), обмеження maxOutputTokens до 1000 для запобігання надмірно довгим відповідям, кешування system prompt на стороні сервера (хоча він регенерується при кожному запиті через динамічність даних) та compression повідомлень через видалення зайвих пробілів та форматування.

Налаштування параметрів моделі включає temperature: 0.7 для балансу між креативністю та детермінованістю, top_p за замовчуванням для sampling diversity та presence_penalty та frequency_penalty для зменшення повторень.

Обробка помилок та fallback механізми включають retry logic з експоненційною затримкою при тимчасових збоях OpenAI API, fallback до повідомлення про помилку з проханням спробувати пізніше, логування помилок для моніторингу та аналізу та graceful degradation (система залишається функціональною навіть якщо AI недоступний – користувач може

використовувати звичайний пошук).

Моніторинг та аналітика включають логування всіх запитів до AI API з міткою часу, ID користувача та типом запиту, відстеження метрик (час відгуку, успішність парсингу JSON, частота помилок), збір фідбеку користувачів щодо якості відповідей AI та A/B тестування різних варіантів system prompt для оптимізації якості.

2.3 Реалізація алгоритмів виявлення конфліктів

Система виявлення конфліктів в оренді та бронюваннях є критично важливою для забезпечення цілісності даних та запобігання подвійному бронюванню офісів. Проєктування цієї системи вимагає врахування різних типів бронювань та оптимізації продуктивності.

Структура даних бронювання. Модель Booking в Prisma schema визначає структуру зберігання бронювань в базі даних PostgreSQL.

Основні поля моделі включають id (унікальний ідентифікатор типу CUID), officeId (зовнішній ключ на Office), userId (зовнішній ключ на User), userName та userEmail (денормалізовані поля для швидкого доступу), startDate та endDate (дати початку та завершення бронювання типу DateTime), startTime та endTime (опціональні поля типу String у форматі "HH:MM" для погодинних бронювань), isRecurring (булеве поле для позначення рекурентних бронювань), recurrence (JSON-поле для зберігання об’єкта рекурентності), status (enum: PENDING, CONFIRMED, CANCELLED, COMPLETED), totalPrice (обчислена ціна бронювання) та timestamps (createdAt, updatedAt).

Об’єкт recurrence зберігається як JSON і включає поля pattern (тип рекурентності: DAILY, WEEKLY, MONTHLY), daysOfWeek (масив чисел 1-7 для WEEKLY паттерну), interval (інтервал повторення, наприклад 1 для "кожного тижня", 2 для "кожного другого тижня") та endDate (дата завершення рекурентності).

Переваги використання JSON для recurrence включають гнучкість схеми (можна додавати нові поля без міграції бази даних), атомарність (всі дані

рекурентності зберігаються в одному полі) та простоту запитів (не потрібно JOIN з окремою таблицею). Недоліки включають обмежені можливості індексування (PostgreSQL підтримує GIN індекси для JSON, але вони менш ефективні за B-tree) та складність валідації (валідація структури JSON відбувається на рівні додатку).

Індексування для оптимізації запитів включає індекс на officeId для швидкого пошуку всіх бронювань оренди офісу, індекс на startDate для фільтрації бронювань за датами, індекс на status для вибірки тільки активних бронювань, композитний індекс на (officeId, startDate, status) для оптимізації основного запиту перевірки доступності.

Алгоритм генерації рекурентних дат. Функція generateRecurringDates() перетворює правило рекурентності в список конкретних дат, на які поширюється бронювання.

Сигнатура функції приймає startDate (дата початку періоду), endDate (дата завершення періоду, зазвичай відповідає recurrence.endDate) та recurrence (об’єкт з правилами рекурентності) і повертає масив об’єктів Date з усіма датами, на які поширюється бронювання.

Алгоритм для WEEKLY паттерну виконує наступні кроки. Крок 1: Ініціалізувати current = new Date(startDate) та end = new Date(recurrence.endDate). Крок 2: Створити порожній масив occurrences = []. Крок 3: Поки current <= end, виконувати ітерацію. Крок 4: Отримати день тижня для current (0-6, де 0=Sunday), конвертувати в 1-7 формат (1=Monday, 7=Sunday) через dayOfWeek = current.getDay() === 0 ? 7 : current.getDay(). Крок 5: Перевірити, чи dayOfWeek входить в recurrence.daysOfWeek масив. Крок 6: Якщо так, додати new Date(current) до occurrences. Крок 7: Збільшити current на 1 день через current.setDate(current.getDate() + 1). Крок 8: Повторити з кроку 3. Крок 9: Повернути occurrences.

Часова складність алгоритму становить O(N), де N – кількість днів між startDate та endDate. Для бронювання терміном 1 рік $N \approx 365$, що є прийнятним. Для оптимізації можна додати обмеження на максимальну кількість входжень

(наприклад, 1000) для запобігання надмірно великим масивам.

Просторова складність становить O(M), де M – кількість згенерованих входжень. Для WEEKLY бронювання на 1 рік з 2 днями на тиждень M ≈ 52 × 2 = 104, що є прийнятним для зберігання в пам’яті.

Алгоритм для DAILY паттерну спрощується до додавання кожного дня між startDate та endDate без перевірки дня тижня. Це можна оптимізувати через обчислення різниці в днях та генерацію масиву фіксованого розміру.

Алгоритм для MONTHLY паттерну вимагає додаткової логіки для визначення, на який день місяця припадає входження (наприклад, "перший понеділок місяця", "15 число кожного місяця"). Наразі MONTHLY паттерн не повністю підтримується в поточній реалізації, але може бути доданий в майбутньому.

Алгоритм перевірки перетину дат та часу. Перевірка конфліктів базується на двох функціях: datesOverlap() для перевірки перетину діапазонів дат та timesOverlap() для перевірки перетину часових проміжків.

Функція datesOverlap() приймає чотири параметри Date (start1, end1, start2, end2) та повертає boolean. Алгоритм перевірки використовує математичну умову перетину інтервалів: два інтервали перетинаються, якщо start1 <= end2 AND start2 <= end1. Часова складність O(1), оскільки це одна порівняльна операція.

Приклади роботи функції включають наступні випадки. Інтервали [2025-11-15, 2025-11-17] та [2025-11-16, 2025-11-18] перетинаються (повертає true). Інтервали [2025-11-15, 2025-11-17] та [2025-11-18, 2025-11-20] не перетинаються (повертає false). Інтервали [2025-11-15, 2025-11-20] та [2025-11-16, 2025-11-18] перетинаються, причому другий повністю містить перший (повертає true).

Функція timesOverlap() приймає чотири рядки у форматі "HH:MM" (startTime1, endTime1, startTime2, endTime2) та повертає boolean. Алгоритм перевірки включає наступні кроки. Крок 1: Розпарсити кожний час в години та хвилини через split(‘:’).map(Number). Крок 2: Конвертувати час в хвилини

від початку доби через minutes = hours × 60 + minutes. Крок 3: Застосувати ту ж умову перетину інтервалів: start1Minutes < end2Minutes AND start2Minutes < end1Minutes. Часова складність O(1).

Приклади роботи функції включають наступні випадки. Часи ["14:00", "16:00"] та ["15:00", "17:00"] перетинаються (повертає true). Часи ["14:00", "16:00"] та ["16:00", "18:00"] не перетинаються, оскільки використовується строга нерівність (повертає false). Часи ["09:00", "17:00"] та ["10:00", "12:00"] перетинаються (повертає true).

Основний алгоритм checkAvailability(). Функція checkAvailability() є центральною функцією для перевірки доступності офісу для нового бронювання.

Параметри функції включають officeId (ID офісу для перевірки), startDate та endDate (діапазон дат бронювання), startTime та endTime (опціональні, для погодинних бронювань), isRecurring та recurrence (для рекурентних бронювань) та excludeBookingId (опціональний, для виключення конкретного бронювання при редагуванні).

Повертає об’єкт AvailabilityCheckResult з полями available (true, якщо офіс доступний), conflictingBookings (масив бронювань, що конфліктують) та message (опціональне текстове пояснення).

Алгоритм виконує наступні кроки. Крок 1: Отримати всі активні бронювання для officeId з status IN (‘PENDING’, ‘CONFIRMED’) через Prisma запит. Якщо передано excludeBookingId, виключити це бронювання з результатів. Складність цього кроку O(log N + K) завдяки індексу, де K – кількість активних бронювань офісу. Крок 2: Ініціалізувати conflictingBookings = []. Крок 3: Для кожного existingBooking перевірити конфлікт. Крок 4: Якщо запитуване бронювання рекурентне, викликати checkRecurringConflict(params, existingBooking). Крок 5: Інакше перевірити перетин дат через datesOverlap(). Крок 6: Якщо дати перетинаються та існуюче бронювання також рекурентне, згенерувати його входження через generateRecurringDates() та перевірити кожне входження. Крок 7: Якщо дати

перетинаються і обидва бронювання не рекурентні, перевірити час через timesOverlap() (якщо час вказаний) або вважати конфліктом (якщо хоча б одне all-day). Крок 8: Якщо виявлено конфлікт, додати existingBooking до conflictingBookings. Крок 9: Після перевірки всіх існуючих бронювань повернути результат.

Часова складність алгоритму у найгіршому випадку становить O(K × M × N), де K – кількість активних бронювань офісу, M – кількість входжень запитуваного рекурентного бронювання, N – середня кількість входжень існуючих рекурентних бронювань та оренди. У типовому випадку (більшість бронювань не рекурентні) складність O(K).

Функція checkRecurringConflict(). Ця допоміжна функція перевіряє конфлікт між запитуваним рекурентним бронюванням та існуючим бронюванням.

Алгоритм включає наступні кроки. Крок 1: Згенерувати всі входження запитуваного бронювання через generateRecurringDates(params.startDate, params.endDate, params.recurrence). Крок 2: Якщо existingBooking також рекурентне, згенерувати його входження через generateRecurringDates(). Крок 3: Для кожної пари (requestedDate, existingDate) перевірити, чи requestedDate.toDateString() === existingDate.toDateString(). Крок 4: Якщо дати співпадають, перевірити час через timesOverlap() (якщо час вказаний) або вважати конфліктом (якщо хоча б одне all-day). Крок 5: Якщо знайдено хоча б одне співпадіння, повернути true. Крок 6: Якщо existingBooking не рекурентне, для кожного requestedDate викликати checkSingleOccurrenceConflict (requestedDate, params.startTime, params.endTime, existingBooking). Крок 7: Якщо хоча б одне входження конфліктує, повернути true. Крок 8: Якщо жодне входження не конфліктує, повернути false.

Часова складність O(M × N) для випадку двох рекурентних бронювань або O(M) для випадку, коли існуюче бронювання не рекурентне.

Функція checkSingleOccurrenceConflict(). Ця допоміжна функція перевіряє конфлікт між одним входженням рекурентного бронювання та

існуючим бронюванням.

Алгоритм включає наступні кроки. Крок 1: Перевірити перетин дат через datesOverlap(occurrenceDate, occurrenceDate, existingBooking.startDate, existingBooking.endDate). Зауважимо, що для одного входження startDate = endDate = occurrenceDate. Крок 2: Якщо дати не перетинаються, повернути false. Крок 3: Якщо обидва бронювання мають час (погодинні), перевірити час через timesOverlap(). Крок 4: Якщо час перетинається, повернути true, інакше false. Крок 5: Якщо хоча б одне бронювання all-day, повернути true (конфлікт завжди є). Часова складність O(1).

Обробка граничних випадків. Алгоритми виявлення конфліктів мають коректно обробляти різні граничні випадки.

Бронювання на межі днів включає випадок, коли одне бронювання закінчується о 00:00 наступного дня, а інше починається о 00:00 того ж дня. Поточна реалізація вважає це конфліктом через нестрогу нерівність start1 <= end2. Для уникнення цього можна використовувати строгу нерівність або додавати 1 секунду до часу завершення.

All-day бронювання (startTime = null, endTime = null) вважаються такими, що займають весь день, і завжди конфліктують з будь-якими іншими бронюваннями на той самий день, незалежно від часу.

Різні часові зони наразі не обробляються, оскільки всі часи зберігаються без timezone інформації. В production системі рекомендується зберігати всі дати та час в UTC та конвертувати в локальний timezone тільки при відображенні користувачеві.

Зміна літнього/зимового часу потенційно може створити аномалії, коли годинник переводиться на годину назад і 2:30 AM відбувається двічі. PostgreSQL з timezone-aware типами коректно обробляє це, але наразі система використовує timestamp without timezone.

Високосні роки коректно обробляються JavaScript Date API та PostgreSQL, тому спеціальної логіки не потрібно.

Рекурентні бронювання з великою кількістю входжень (>1000) можуть

призвести до проблем з продуктивністю. Рекомендується додати обмеження на максимальну кількість входжень або перейти до batch-обробки з pagination.

Оптимізації продуктивності. Для забезпечення прийнятної продуктивності перевірки доступності застосовані наступні оптимізації.

Індексування бази даних включає композитний індекс на (officeId, startDate, status), що дозволяє PostgreSQL швидко знаходити релевантні бронювання без повного сканування таблиці. Індекси на окремі поля officeId, startDate, status для підтримки різних запитів.

Фільтрація на рівні бази даних включає вибірку тільки активних бронювань (status IN (‘PENDING’, ‘CONFIRMED’)), виключення скасованих та завершених бронювань для зменшення обсягу даних для обробки та опціональну фільтрацію за діапазоном дат в WHERE clause (хоча це складно для рекурентних бронювань через JSON-поле).

Раннє припинення включає повернення false одразу після знаходження першого конфлікту замість перевірки всіх бронювань та використання break в циклах для припинення перевірки після знаходження конфлікту.

Кешування результатів може включати кешування згенерованих входжень рекурентних бронювань для уникнення повторної генерації та кешування результатів checkAvailability() на короткий час (5-10 секунд) для однакових запитів.

Batch-обробка для множинних офісів може включати одночасну перевірку доступності кількох офісів в одному запиті до бази даних та паралельну обробку перевірок через Promise.all().

Сценарії тестування. Для валідації коректності алгоритмів виявлення конфліктів необхідно протестувати різноманітні сценарії.

Прості бронювання включають два бронювання на різні дати (повинні не конфліктувати), два бронювання на одну дату з різним часом без перетину (не конфліктувати), два бронювання на одну дату з частковим перетином часу (конфліктувати), два all-day бронювання на одну дату (конфліктувати), одне all-day та одне погодинне бронювання на одну дату (конфліктувати).

Рекурентні бронювання включають рекурентне бронювання (щопонеділка) та просте бронювання на понеділок всередині періоду (конфліктувати), рекурентне бронювання (щопонеділка) та просте бронювання на вівторок (не конфліктувати), два рекурентні бронювання з різними днями тижня (не конфліктувати), два рекурентні бронювання з однаковими днями та різним часом (не конфліктувати або конфліктувати в залежності від часу), два рекурентні бронювання з однаковими днями та часом (конфліктувати).

Граничні випадки включають бронювання, що закінчується о 00:00, і бронювання, що починається о 00:00 (тестувати обидва варіанти – конфлікт/не конфлікт), бронювання терміном 1 день vs 1 година (перевірка коректності обробки all-day), рекурентне бронювання на 10 років (перевірка обмеження на кількість входжень) та бронювання з некоректними даними (endDate < startDate, невалідний час).

ВИСНОВКИ ДО РОЗДІЛУ 2

У другому розділі проведено аналіз архітектурних рішень та проєктних підходів при розробці Системи оренди та бронювання офісних приміщень Prostir.

Обґрунтовано вибір full-stack підходу на базі Next.js 16 як оптимального архітектурного рішення. Єдина кодова база на TypeScript забезпечує типобезпечність, Server Components та Server Actions дозволяють виконувати серверну логіку безпосередньо в компонентах, гнучкість моделей рендерингу забезпечує оптимальну продуктивність. Порівняння з альтернативами підтвердило збалансованість обраного підходу для проєкту масштабу MVP з можливістю масштабування.

Розроблено архітектуру модуля обробки природної мови на основі GPT-4o-mini. Динамічна генерація system prompt забезпечує актуальність даних про офіси, потокова передача відповідей покращує користувацький досвід, структурована екстракція параметрів через JSON-режим дозволяє створювати валідні бронювання. Механізми обробки помилок та оптимізації знижують витрати і підвищують продуктивність.

Реалізовано алгоритми виявлення конфліктів бронювань для всіх типів (прості, багатоденні, рекурентні) зі складністю від O(1) до O(K) у типових випадках завдяки індексуванню. Обробка граничних випадків та композитні індекси в PostgreSQL забезпечують надійність та продуктивність системи.

Інтегровано PostgreSQL 15 з Prisma ORM. Чотири моделі та чотири enum-типи забезпечують функціональність системи, використання JSON-поля для рекурентності надає гнучкість, стратегії індексування оптимізують запити.

Результати розділу демонструють, що поєднання сучасних архітектурних підходів з ретельно спроєктованими алгоритмами створює фундамент для інтелектуальної Системи оренди та бронювання з високою продуктивністю та надійністю.

РОЗДІЛ 3
ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ ОРЕНДИ ТА БРОНЮВАННЯ ОФІСНИХ ПРИМІЩЕНЬ З ІНТЕГРАЦІЄЮ ШТУЧНОГО ІНТЕЛЕКТУ

3.1 Технології та інструменти для розробки системи

Розробка інтелектуальної Системи оренди та бронювання офісних приміщень Prostir вимагає використання сучасних технологій та інструментів, що забезпечують високу продуктивність, масштабованість та зручність розробки. Правильний вибір технологічного стеку є критично важливим для успішної реалізації системи з інтеграцією великих мовних моделей та складних алгоритмів виявлення конфліктів бронювань.

Frontend-технології. Для реалізації клієнтської частини системи Prostir було обрано React 19 у поєднанні з Next.js 16. React є найпопулярнішою бібліотекою для побудови користувацьких інтерфейсів, що забезпечує компонентний підхід до розробки, віртуальний DOM для оптимізації продуктивності та декларативний синтаксис, що полегшує читання та підтримку коду.

Next.js 16 є full-stack фреймворком на базі React, що надає критично важливі можливості для сучасних вебдодатків. App Router в Next.js 16 використовує файлову систему для маршрутизації, що робить структуру додатку інтуїтивно зрозумілою. Кожна директорія в папці app відповідає сегменту URL, а файли page.tsx визначають інтерфейс сторінки. Система підтримує вкладені макети через layout.tsx, що дозволяє створювати складні композиції інтерфейсу з мінімальним дублюванням коду.

Server Components є революційною можливістю Next.js 16, що дозволяє виконувати React-компоненти на сервері. Це забезпечує значне зменшення обсягу JavaScript, що надсилається клієнту, оскільки серверні компоненти не потребують hydration. Можливість прямого доступу до бази даних з компонентів без створення API endpoints спрощує архітектуру додатку. Автоматична серіалізація даних між сервером і клієнтом знижує boilerplate

код. Для системи Prostir це означає, що компоненти відображення списку офісів можуть безпосередньо запитувати дані через Prisma, без необхідності створювати окремі API-маршрути.

Server Actions в Next.js 16 надають типобезпечний механізм для виконання серверних операцій прямо з клієнтських компонентів. Функції, позначені директивою ‘use server’, можуть бути викликані з форм або обробників подій на клієнті, автоматично виконуючись на сервері. Це усуває необхідність створення REST або GraphQL API для простих операцій CRUD, оскільки Server Actions забезпечують прямий виклик серверних функцій з TypeScript type safety. Для Prostir це означає, що функція createBooking() може бути викликана безпосередньо з форми підтвердження бронювання, без написання окремого API endpoint.

Streaming та Suspense в Next.js 16 дозволяють прогресивно завантажувати частини інтерфейсу. React Suspense boundaries визначають, які частини UI можуть завантажуватися асинхронно, а Streaming дозволяє надсилати HTML клієнту частинами по мірі його генерації. Це забезпечує швидше відображення першого контенту (First Contentful Paint) та кращий користувацький досвід, оскільки користувач бачить прогрес завантаження замість порожнього екрану.

TypeScript є мовою програмування, що додає статичну типізацію до JavaScript. Для проєкту Prostir TypeScript надає критично важливі переваги. Статична типізація дозволяє виявляти помилки на етапі компіляції, а не в runtime, що особливо важливо для складних об’єктів бронювань з опціональними полями recurrence. Автокомпліт в IDE значно прискорює розробку, оскільки розробник бачить доступні методи та властивості об’єктів. Рефакторинг стає безпечнішим завдяки автоматичному виявленню місць, де використовуються змінені типи. Type inference дозволяє TypeScript автоматично визначати типи змінних, зменшуючи необхідність явної анотації типів.

Tailwind CSS був обраний як утилітний CSS-фреймворк для стилізації

інтерфейсу. На відміну від традиційних підходів, де стилі визначаються в окремих CSS-файлах, Tailwind надає набір готових утилітних класів, що можуть комбінуватися безпосередньо в HTML або JSX. Основні переваги включають відсутність конфліктів імен класів завдяки утилітному підходу, автоматичне видалення невикористаних стилів через PurgeCSS, що зменшує розмір фінального CSS-файлу до мінімуму, консистентний дизайн завдяки використанню визначеної дизайн-системи (кольори, відступи, розміри шрифтів) та швидкість розробки завдяки відсутності необхідності перемикатися між HTML і CSS-файлами.

Backend-технології. Серверна частина системи Prostir реалізована на Next.js 16 API Routes та Server Actions, що забезпечує єдину кодову базу для frontend і backend.

API Routes в Next.js дозволяють створювати HTTP endpoints безпосередньо в структурі додатку. Файли route.ts в директорії app/api визначають обробники HTTP-запитів з підтримкою всіх стандартних методів (GET, POST, PUT, DELETE, PATCH). Для системи Prostir критично важливим є API Route для AI-асистента (/api/ai-chat/route.ts), який обробляє запити до великої мовної моделі.

Server Actions надають альтернативний підхід для серверних операцій без необхідності створення HTTP endpoints. Функції в файлах з директивою ‘use server’ можуть безпосередньо викликатися з клієнтських компонентів. Основні переваги включають type safety, оскільки TypeScript забезпечує типобезпечність між клієнтом і сервером, відсутність необхідності визначати API contracts або схеми валідації, автоматичну серіалізацію даних між клієнтом і сервером та можливість використання в формах без JavaScript через progressive enhancement.

Prisma ORM є сучасним інструментом для роботи з базами даних, що забезпечує type-safe доступ до даних. Основні компоненти Prisma включають Prisma Schema, декларативний мова для визначення моделей бази даних в файлі schema.prisma. На основі схеми Prisma автоматично генерує TypeScript-

типи через Prisma Client, що забезпечує повну type safety при роботі з базою даних. Prisma Migrate автоматично генерує SQL-міграції на основі змін в schema.prisma, що спрощує керування версіями схеми бази даних. Prisma Studio надає графічний інтерфейс для перегляду та редагування даних без необхідності писати SQL-запити.

Переваги Prisma для проєкту Prostir включають автоматичну генерацію типів, що означає відсутність десинхронізації між типами TypeScript і схемою бази даних. Запити є повністю типізованими, що виключає runtime помилки через неправильні назви полів або таблиць. Prisma автоматично вирішує проблему N+1 запитів через оптимізацію JOIN-операцій. Connection pooling забезпечує ефективне використання з'єднань з базою даних, що критично важливо для serverless середовищ. Підтримка JSON-полів в PostgreSQL дозволяє зберігати об'єкт recurrence як структурований JSON без необхідності створення окремих таблиць.

База даних. PostgreSQL 15 була обрана як система управління базами даних для проєкту Prostir з наступних причин.

ACID-гарантії (Atomicity, Consistency, Isolation, Durability) забезпечують надійність транзакцій, що критично важливо для операцій бронювання. Atomicity означає, що бронювання кількох офісів одночасно або виконується повністю, або відкочується повністю. Consistency забезпечує, що база даних завжди знаходиться в коректному стані після кожної транзакції. Isolation гарантує, що паралельні бронювання не конфліктують завдяки механізмам блокування. Durability означає, що після підтвердження транзакції дані гарантовано збережені навіть у випадку збою сервера.

Підтримка складних типів даних включає JSONB для ефективного зберігання та індексування JSON-даних, що використовується для поля recurrence. Arrays дозволяють зберігати масиви в одному полі, що використовується для amenities в таблиці Office. Date/Time типи з підтримкою часових зон забезпечують коректну роботу з датами бронювань. Custom Types через CREATE TYPE дозволяють визначати enum-типи на рівні бази даних.

Потужні можливості індексування включають B-tree індекси для швидкого пошуку за датами та ID. GIN індекси для пошуку в JSONB-полях та масивах. Composite індекси для оптимізації запитів з множинними умовами WHERE. Partial індекси для індексування лише певних підмножин даних, наприклад, тільки активних бронювань.

Масштабованість PostgreSQL забезпечується через Read Replicas для розподілу навантаження читання між кількома серверами. Partitioning дозволяє розділяти великі таблиці на менші частини для покращення продуктивності. Connection Pooling через PgBouncer або Prisma забезпечує ефективне використання з'єднань. Horizontal Sharding через розширення типу Citus для масштабування на кілька серверів.

Інтеграція зі штучним інтелектом. Для реалізації AI-асистента в системі Prostir використовуються OpenAI API та Vercel AI SDK.

OpenAI API надає доступ до великих мовних моделей через RESTful API. Для проєкту Prostir була обрана модель GPT-4o-mini з наступних причин. Економічність становить $0.15 за 1 мільйон вхідних токенів та $0.60 за 1 мільйон вихідних токенів, що в 10 разів дешевше за GPT-4. Швидкість відгуку зазвичай становить 1-2 секунди для середньої відповіді, що забезпечує хороший користувацький досвід. Якість розпізнавання природномовних запитів є достатньою для завдань бронювання офісів, хоча модель може іноді потребувати уточнюючих запитань. Велике контекстне вікно в 128,000 токенів дозволяє передавати повний список офісів та історію діалогу без обмежень.

Vercel AI SDK є бібліотекою для спрощення інтеграції з LLM в React-додатках. Основні можливості включають хук useChat(), який автоматично управляє станом чату (messages, input, isLoading), обробляє відправку повідомлень та потокове оновлення відповідей AI. Функція streamText() на серверній частині забезпечує потокову передачу відповідей AI клієнту, що дозволяє відображати текст по мірі генерації. Type-safe інтеграція з TypeScript забезпечує повну типізацію повідомлень та конфігурації моделі. Підтримка різних провайдерів AI (OpenAI, Anthropic, Cohere) через уніфікований

інтерфейс дозволяє легко переключатися між моделями.

Архітектура AI-модуля базується на наступних принципах. Сервер виступає як Gateway між клієнтом і OpenAI API, що дозволяє приховати API-ключі від клієнта та додати проміжну обробку запитів. Динамічна генерація system prompt на основі актуальних даних з бази забезпечує, що AI завжди оперує свіжою інформацією про доступні офіси. Потокова передача відповідей покращує сприйняття користувачем, оскільки текст з'являється поступово, а не весь одразу після затримки. JSON-режим дозволяє AI повертати структуровані дані для автоматичного створення бронювання після підтвердження користувача.

Інструменти розробки та DevOps. Для ефективної розробки та розгортання системи Prostir використовується набір сучасних інструментів.

pnpm є менеджером пакетів, обраним замість npm або yarn, завдяки кільком перевагам. Ефективне використання дискового простору через symlinks до глобального сховища пакетів замість дублювання node_modules. Швидкість встановлення пакетів значно вища завдяки паралельній установці та кешуванню. Строга ізоляція залежностей запобігає phantom dependencies, коли код може випадково використовувати пакети, не вказані в package.json. Монорепо-підтримка через workspaces дозволяє керувати множинними пакетами в одному репозиторії.

ESLint є інструментом для статичного аналізу JavaScript/TypeScript коду. Конфігурація для Next.js включає рекомендовані правила (@next/eslint-plugin-next), TypeScript-специфічні правила (@typescript-eslint), правила для React hooks та accessibility (jsx-a11y) для забезпечення доступності інтерфейсу. ESLint виявляє потенційні помилки до runtime, забезпечує консистентний стиль коду в команді та може автоматично виправляти певні категорії проблем через eslint --fix.

Git використовується для контролю версій коду. Основні практики в проєкті включають використання feature branches для розробки нових функцій, commit messages у форматі Conventional Commits (feat:, fix:, docs:) для

автоматичної генерації changelog, pull requests для code review перед злиттям змін в main branch та GitHub Actions для автоматизації CI/CD процесів.

Visual Studio Code є основним редактором коду з розширеннями для Next.js, TypeScript, Prisma, Tailwind CSS та ESLint. Інтеграція з TypeScript забезпечує інтелектуальне автодоповнення та навігацію по коду. Prisma extension надає syntax highlighting для schema.prisma та інтеграцію з Prisma Studio. Tailwind CSS IntelliSense забезпечує автокомпліт класів Tailwind прямо в JSX.

Vercel є хмарною платформою для розгортання Next.js додатків. Основні переваги включають нульову конфігурацію, оскільки Vercel автоматично виявляє Next.js і налаштовує оптимальне середовище. Автоматичне розгортання при кожному push до Git-репозиторію забезпечує безперервну доставку. Preview deployments створюють унікальний URL для кожного pull request, що дозволяє переглядати зміни до злиття. Edge Network з глобальним CDN забезпечує мінімальну латентність для користувачів з будь-якої точки світу. Analytics та Monitoring вбудовані в платформу для відстеження продуктивності та помилок.

Docker може використовуватися для локальної розробки та тестування. PostgreSQL в Docker контейнері забезпечує ізольоване середовище бази даних, що не конфліктує з іншими проєктами. Docker Compose дозволяє визначити всі сервіси (база даних, redis для кешування) в одному файлі docker-compose.yml. Production deployment може використовувати Docker для консистентності між середовищами розробки та production.

Порівняння з альтернативними технологіями. Для обґрунтування вибору технологічного стеку важливо порівняти обрані технології з альтернативами.

Альтернативи Next.js включають Create React App (CRA), який є простішим, але не надає SSR, SSG, API Routes та інших можливостей Next.js. Gatsby підходить для статичних сайтів, але менш зручний для динамічних додатків з частими оновленнями даних. Remix є сучасним конкурентом Next.js

з фокусом на web fundamentals, але має меншу екосистему та спільноту. Vite + React надає швидкий dev server, але вимагає самостійної конфігурації для SSR та маршрутизації.

Альтернативи Prisma включають TypeORM, який підтримує більше баз даних, але має менш зручний синтаксис запитів та гірший type inference. Sequelize є зрілим ORM для Node.js, але написаний на JavaScript і має обмежену підтримку TypeScript. Drizzle ORM є новим конкурентом з фокусом на performance, але має меншу екосистему. Kysely надає type-safe SQL query builder, але вимагає написання більше SQL-коду вручну.

Альтернативи PostgreSQL включають MySQL, яка є популярною, але має менші можливості для складних типів даних та JSON. MongoDB є NoSQL базою даних, яка підходить для неструктурованих даних, але не надає ACID-гарантій для транзакцій між документами. SQLite підходить для малих проєктів, але має обмеження на паралельну обробку запитів. Supabase надає PostgreSQL з додатковими можливостями (authentication, storage, real-time), але додає vendor lock-in.

Альтернативи OpenAI API включають Anthropic Claude, який має великі контекстні вікна та краще розуміння складних інструкцій, але дорожчий. Google PaLM/Gemini надає конкурентні можливості, але має менш зручний API. Open-source моделі (Llama, Mistral) можуть розгортатися самостійно, але вимагають значних обчислювальних ресурсів та expertise в ML. Azure OpenAI Service надає ті ж моделі OpenAI через Azure, що може бути кращим для корпоративних клієнтів з вимогами до data residency.

Висновки щодо вибору технологій. Обраний технологічний стек забезпечує оптимальний баланс між продуктивністю, зручністю розробки та економічністю для проєкту масштабу Prostir. Next.js 16 надає всі необхідні можливості в єдиному фреймворку, знижуючи складність архітектури. TypeScript забезпечує надійність коду через статичну типізацію. PostgreSQL з Prisma надає потужні можливості для роботи з даними з мінімальним boilerplate. OpenAI API через Vercel AI SDK дозволяє швидко інтегрувати

інтелектуального асистента без необхідності розгортання власних ML-моделей.

3.2 Архітектурні рішення при проєктуванні системи

Архітектурне проєктування системи Prostir базується на сучасних принципах організації коду та оптимальному використанні можливостей обраного технологічного стеку. Ключовими аспектами архітектури є Feature-Sliced Design для організації коду, продумана схема бази даних та ефективна інтеграція AI-модуля.

Методологія Feature-Sliced Design. Feature-Sliced Design (FSD) є архітектурною методологією для організації frontend-проєктів, що базується на розділенні коду за бізнес-функціями (features) замість технічних ролей (components, utils, services).

Основні принципи FSD включають розділення на шари (layers), де кожен шар має чітко визначену відповідальність та може залежати тільки від нижчих шарів. Слайси (slices) всередині кожного шару відповідають окремим бізнес-сутностям або функціям. Сегменти (segments) всередині слайсів організують код за технічною роллю (ui, model, api, lib). Public API кожного слайсу визначає, які частини коду доступні для використання іншими модулями.

Шари в архітектурі FSD організовані ієрархічно знизу вгору. Шар shared містить повторно використовуваний код, що не пов’язаний з конкретною бізнес-логікою: UI-компоненти (Button, Input, Modal), утиліти (date-utils, validation), API-конфігурацію (prisma client, axios instance) та константи (OFFICE_TYPES, BOOKING_STATUSES). Код в shared може використовуватися будь-яким іншим шаром, але сам не може імпортувати з інших шарів.

Шар entities містить бізнес-сутності додатку без конкретної логіки використання. Для Prostir це включає entity User (типи, схеми валідації), entity Office (типи, константи типів офісів), entity Building (типи, схеми) та entity Booking (типи, енуми статусів). Entities визначають форму даних, але не

містять логіки їх обробки.

Шар features містить функції додатку, що реалізують конкретні user stories. Приклади включають feature office-search (UI для пошуку офісів, фільтри, логіка пошуку), feature booking-form (форма створення бронювання, валідація), feature ai-assistant (чат-інтерфейс, інтеграція з AI API) та feature booking-calendar (календар доступності, візуалізація бронювань). Features можуть використовувати entities та shared, але не можуть залежати від інших features.

Шар widgets містить компоненти, що комбінують кілька features для створення самодостатніх блоків інтерфейсу. Приклади включають widget OfficeCard (відображення інформації про офіс з кнопкою бронювання), widget BookingList (список бронювань користувача з можливістю скасування) та widget SearchPanel (панель пошуку з фільтрами та результатами). Widgets знають про features, entities і shared.

Шар pages відповідає за маршрути додатку та композицію widgets і features для конкретних сторінок. В Next.js App Router це директорії в app: app/(main)/page.tsx – головна сторінка, app/search/page.tsx – сторінка пошуку офісів, app/booking/page.tsx – AI-асистент для бронювання, app/my-bookings/page.tsx – список бронювань користувача. Pages є top-level шаром, який може використовувати всі нижчі шари.

Шар app містує глобальну конфігурацію додатку: providers (SessionProvider, QueryClientProvider), глобальні стилі (globals.css), корневі layouts та error boundaries. Також тут розміщуються Server Actions в app/actions, які технічно не є частиною FSD, але логічно відносяться до app layer як серверна бізнес-логіка.

Правила залежностей в FSD суворо регламентовані. Вищі шари можуть імпортувати з нижчих: pages → widgets → features → entities → shared. Шари одного рівня не можуть імпортувати один з одного: features не можуть залежати від інших features. Кожен слайс має публічний API через файл index.ts, який експортує тільки необхідні частини. Імпорти можуть відбуватися

тільки через публічний API: import { Button } from ‘@/shared/ui’ замість ‘@/shared/ui/button/Button.tsx’.

Переваги FSD для проєкту Prostir включають інтуїтивну організацію, оскільки структура відповідає ментальній моделі додатку. Масштабованість забезпечується тим, що додавання нових features не впливає на існуючі. Переносимість features між проєктами спрощується завдяки чітким залежностям. Зменшення конфліктів в команді досягається за рахунок ізоляції features. Легкість тестування забезпечується можливістю тестувати кожен слайс окремо.

Структура директорій проєкту Prostir організована наступним чином. Директорія src/shared містить ui (компоненти Button, Input, Modal, Card), lib (утиліти prisma.ts, availability.ts, recurring-utils.ts, validation.ts), api (mock-data.ts для розробки) та config (constants.ts з константами). Директорія src/entities включає user/model/types.ts, office/model/types.ts, building/model/types.ts та booking/model/types.ts. Директорія src/features (в перспективі) може містити office-search, booking-form та ai-assistant. Директорія src/widgets (в перспективі) може включати OfficeCard та BookingList. Директорія src/app містує маршрути (main)/page.tsx, search/page.tsx, booking/page.tsx, my-bookings/page.tsx, actions (bookings.ts з Server Actions та offices.ts з Server Actions) та api (ai-chat/route.ts для AI endpoint).

Проєктування схеми бази даних. Схема бази даних для системи Prostir включає чотири основні моделі та чотири enum-типи, визначені в prisma/schema.prisma.

Модель User зберігає інформацію про користувачів системи. Поля включають id (String, CUID, primary key), name (String, ім’я користувача), email (String, унікальний email), role (enum UserRole, роль користувача) та bookings (relation Booking[], бронювання користувача). Індекси включають unique на email для швидкого пошуку при авторизації. UserRole enum визначає три можливі ролі: USER для звичайних користувачів, ADMIN для адміністраторів системи та OWNER для власників будівель.

Модель Building представляє будівлі з офісами. Поля включають id (String, CUID, primary key), name (String, назва будівлі), address (String, адреса), city (String, місто), description (String?, опціональний опис), imageUrl (String?, URL зображення будівлі), createdAt (DateTime, дата створення запису) та offices (relation Office[], офіси в будівлі). Індекси включають index на city для фільтрації будівель за містом.

Модель Office представляє окремі офіси для бронювання. Поля включають id (String, CUID, primary key), buildingId (String, зовнішній ключ на Building), building (relation Building), floorNumber (Int, номер поверху), name (String, назва офісу), capacity (Int, місткість в особах), type (enum OfficeType, тип офісу), pricePerHour (Decimal, ціна за годину), amenities (String[], масив зручностей), description (String?, опціональний опис), imageUrl (String?, URL зображення) та bookings (relation Booking[], бронювання офісу). Індекси включають composite index на (buildingId, floorNumber) для швидкого пошуку офісів на конкретному поверсі будівлі та index на type для фільтрації офісів за типом. OfficeType enum визначає чотири типи офісів: MEETING_ROOM для переговорних кімнат, PRIVATE_OFFICE для приватних офісів, DESK для робочих місць (hot desk) та CONFERENCE_ROOM для конференц-залів.

Модель Booking зберігає інформацію про бронювання офісів. Поля включають id (String, CUID, primary key), officeId (String, зовнішній ключ на Office), office (relation Office), userId (String, зовнішній ключ на User), user (relation User), userName (String, ім’я користувача, денормалізоване для швидкого доступу), userEmail (String, email користувача, денормалізований), startDate (DateTime, дата початку бронювання), endDate (DateTime, дата завершення), startTime (String?, опціональний час початку в форматі "HH:MM"), endTime (String?, опціональний час завершення в форматі "HH:MM"), isRecurring (Boolean, чи є бронювання рекурентним), recurrence (Json?, об’єкт з правилами рекурентності), status (enum BookingStatus, статус бронювання), totalPrice (Decimal, загальна вартість), notes (String?, додаткові примітки) та createdAt (DateTime, дата створення запису). Індекси включають

composite index на (officeId, startDate, status) для оптимізації головного запиту перевірки доступності та index на userId для швидкого отримання бронювань користувача. BookingStatus enum визначає чотири статуси: PENDING для бронювань, що очікують підтвердження, CONFIRMED для підтверджених бронювань, CANCELLED для скасованих бронювань та оренди та COMPLETED для завершених бронювань.

JSON-структура recurrence містить pattern (String, тип рекурентності: "DAILY", "WEEKLY", "MONTHLY"), daysOfWeek (Number[], масив чисел 1-7, де 1=Monday, 7=Sunday), interval (Number, інтервал повторення, зазвичай 1) та endDate (String ISO, дата завершення рекурентності). Використання JSON-поля для recurrence надає гнучкість, оскільки можна додавати нові поля без міграції бази даних, та простоту, оскільки всі дані рекурентності зберігаються атомарно в одному полі. Однак це має обмеження: GIN індекси для JSON менш ефективні за B-tree, валідація структури JSON відбувається на рівні додатку, а не бази даних.

Стратегії індексування включають Primary Key індекси автоматично створюються Prisma для всіх id полів, забезпечуючи O(log N) пошук за первинним ключем. Foreign Key індекси автоматично створюються для зовнішніх ключів (buildingId, officeId, userId), прискорюючи JOIN-операції. Composite індекси створюються для оптимізації складних запитів: (officeId, startDate, status) для checkAvailability() забезпечує швидку фільтрацію активних бронювань конкретного офісу за датами. Single-column індекси на city, type для фільтрації у пошуку офісів. Unique індекси на email в User для забезпечення унікальності та швидкого пошуку.

Нормалізація та денормалізація в схемі збалансована для оптимальної продуктивності. Нормалізація включає розділення даних про будівлі та офіси на окремі таблиці для уникнення дублювання, використання зовнішніх ключів для забезпечення референційної цілісності. Денормалізація включає зберігання userName та userEmail в Booking для швидкого відображення списку бронювань без JOIN з User, що виправдано, оскільки ці поля рідко

змінюються, і навіть якщо користувач змінить ім’я, історичні бронювання залишаються з попереднім ім’ям.

Міграції бази даних керуються через Prisma Migrate. Процес створення міграції включає внесення змін в prisma/schema.prisma, виконання команди npx prisma migrate dev --name migration_name для генерації SQL-міграції та автоматичного застосування її до бази даних розробки, перегляд згенерованого SQL в prisma/migrations/timestamp_migration_name/migration.sql та commit міграції в Git разом зі змінами в schema.prisma. Production міграції виконуються через npx prisma migrate deploy, який застосовує тільки неви конані міграції без генерації нових.

Seed-скрипт для початкових даних створений в prisma/seed.ts і виконується через npx prisma db seed. Скрипт створює демонстраційні будівлі, офіси різних типів в кожній будівлі, тестових користувачів та приклади бронювань для тестування. Seed-дані дозволяють одразу почати роботу з системою без ручного введення даних та забезпечують консистентне середовище для розробки та тестування.

Архітектура AI-модуля. Модуль AI-асистента є ключовою інновацією системи Prostir, що відрізняє її від традиційних систем бронювання.

Компоненти архітектури AI-модуля включають API Route (/api/ai-chat/route.ts), який є entry point для всіх запитів до AI, приймає POST запити з історією діалогу, генерує system prompt з актуальними даними про офіси та повертає потокову відповідь через Vercel AI SDK. Функція buildSystemPrompt() відповідає за запит всіх офісів з бази даних через Prisma, групування офісів за будівлями та поверхами, форматування списку офісів в текстовий формат для промпту та додавання інструкцій, прикладів та правил для AI. Функція convertToModelMessages() перетворює повідомлення з клієнта (які можуть мати різний формат через AI SDK v5) в уніфікований формат ModelMessage[], який очікує streamText(). Клієнтський компонент AIBookingPage використовує хук useChat() для управління станом чату, автоматичної відправки повідомлень та обробки потокових відповідей.

Потік даних в AI-модулі виконується наступним чином. Користувач вводить повідомлення в чат-інтерфейс, наприклад "I need a meeting room for tomorrow at 2pm for 8 people". Хук useChat() автоматично відправляє POST запит до /api/ai-chat з масивом messages, що містить історію діалогу. Сервер отримує запит та викликає buildSystemPrompt() для генерації актуального system prompt, який включає список всіх доступних офісів з поточними цінами та зручностями. Сервер викликає streamText() з OpenAI моделлю gpt-4o-mini, передаючи system prompt та історію повідомлень. OpenAI API обробляє запит та починає генерувати відповідь, яка stream’иться назад до сервера token by token. Сервер пересилає потік токенів клієнту через Server-Sent Events (SSE). Клієнт оновлює UI в реальному часі по мірі надходження токенів, створюючи ефект "друкування". Після завершення stream’у клієнт перевіряє, чи містить відповідь JSON-блок з action: CREATE_BOOKING. Якщо так, клієнт парсить JSON, зберігає дані в sessionStorage та перенаправляє на сторінку підтвердження /booking/confirm.

Структура system prompt є критично важливою для якості роботи AI. Промпт включає визначення ролі асистента: "You are a helpful office booking assistant for Prostir". Динамічно згенерований список офісів з інформацією про назву, ID, тип, місткість, ціну, зручності, організований за будівлями та поверхами. Поточна дата для коректної інтерпретації відносних дат типу "tomorrow", "next week". Приклади різних типів бронювань: простих (конкретна дата та час), довгострокових (кілька років) та рекурентних (щотижневі, щомісячні). JSON-схема відповіді, яку AI має повернути після підтвердження користувача, з детальним описом всіх полів. Правила поведінки: завжди підтверджувати деталі перед поверненням JSON, використовувати точні ID офісів з бази, коректно обчислювати дати та час, бути розмовним, але ефективним.

Обробка JSON-відповідей AI виконується на клієнті через useEffect, який відслідковує зміни в messages. Регулярний вираз /json\s*(\{[\s\S]*?\})\s*/g знаходить JSON-блоки в markdown-форматі. JSON парситься через

JSON.parse() з обробкою помилок в try-catch. Валідація розпарсених даних перевіряє наявність обов'язкових полів (action, offices, startDate, endDate), коректність формату дат (ISO string) та логічність даних (endDate >= startDate, offices є масивом CUID). Якщо валідація успішна, дані зберігаються в sessionStorage для передачі на сторінку підтвердження та виконується navigation через router.push('/booking/confirm').

Оптимізації AI-модуля включають вибір економічної моделі (GPT-4o-mini замість GPT-4 забезпечує 10× економію), обмеження maxOutputTokens до 1000 для запобігання надмірно довгим відповідям, налаштування temperature: 0.7 для балансу між креативністю та детермінованістю та обробку помилок з fallback до повідомлення про тимчасову недоступність сервісу.

Інтеграція Server Actions. Server Actions є ключовим елементом серверної архітектури Next.js 16, що дозволяє викликати серверні функції з клієнтських компонентів.

Структура Server Actions в Prostir організована в файлах app/actions/bookings.ts для операцій з бронюваннями (createBooking, getBookingsByUserId, cancelBooking, checkOfficeAvailability) та app/actions/offices.ts для операцій з офісами (getAllOffices, getOfficeById, searchOffices). Кожен файл починається з директиви 'use server', яка вказує Next.js компілятору, що всі функції в файлі є Server Actions. Функції експортуються як async і можуть бути імпортовані та викликані з Client Components.

Переваги Server Actions для Prostir включають type safety, оскільки TypeScript забезпечує типобезпечність між клієнтом і сервером без необхідності дублювання типів. Відсутність API boilerplate означає, що немає потреби створювати REST endpoints, визначати маршрути, обробляти HTTP методи. Пряме використання в формах через form action дозволяє працювати без JavaScript (progressive enhancement). Автоматична ревалідація кешу через revalidatePath() після мутацій забезпечує актуальність даних на сторінках. Інтеграція з Prisma дозволяє використовувати ORM безпосередньо в Server

Actions без додаткового API layer.

Приклад Server Action createBooking демонструє типовий патерн. Функція приймає CreateBookingDTO з валідованими даними. Валідація виконується через validateBookingData() на початку функції. Перевірка доступності офісів через checkAvailability() для кожного офісу перед створенням бронювання. Використання транзакції Prisma для атомарного створення множинних бронювань. Обчислення totalPrice на основі ціни офісу, тривалості та типу бронювання (погодинне, денне, рекурентне). Створення записів в базі даних через prisma.booking.create(). Ревалідація кешу через revalidatePath(‘/my-bookings’, ‘/search’) для оновлення UI. Повернення масиву створених бронювань з повною типізацією.

Обробка помилок в Server Actions включає try-catch блоки для перехоплення всіх помилок, спеціалізовані винятки (ValidationError для помилок валідації, Error для бізнес-логічних помилок типу "Office not available"), логування помилок через console.error для debugging та повернення помилок клієнту через throw, що автоматично обробляється Next.js.

Висновки щодо архітектури. Архітектурні рішення системи Prostir забезпечують баланс між простотою розробки та можливостями масштабування. Feature-Sliced Design надає чітку організацію коду, що спрощує навігацію та підтримку. Продумана схема бази даних з правильними індексами забезпечує продуктивність запитів. AI-модуль інтегрований через чисту архітектуру з розділенням відповідальностей. Server Actions усувають необхідність в традиційному API layer, спрощуючи архітектуру.

3.3 Програмна реалізація ключових компонентів системи

Програмна реалізація системи Prostir базується на компонентах, розглянутих в архітектурному проєктуванні. У цьому підрозділі детально розглядаються ключові фрагменти коду з поясненнями реалізації критично важливих функцій.

Реалізація AI-асистента. Центральним компонентом інтелектуальної

Системи оренди та бронювання є API Route для обробки запитів до великої мовної моделі.

Лістинг 3.1 демонструє реалізацію функції buildSystemPrompt(), яка динамічно генерує system prompt для AI на основі актуальних даних про офіси з бази даних.

```
async function buildSystemPrompt() {
  // Fetch real offices from database (directly using Prisma, not server action)
  const offices = await prisma.office.findMany({
    include: {
      building: true,
    },
    orderBy: {
      name: 'asc',
    },
  });

  if (!offices || offices.length === 0) {
    throw new Error('No offices found in database. Please run: npx prisma db seed');
  }

  // Group offices by building
  const buildingMap = new Map<string, typeof offices>();
  offices.forEach((office) => {
    const existing = buildingMap.get(office.buildingId);
    if (existing) {
      existing.push(office);
    } else {
      buildingMap.set(office.buildingId, [office]);
    }
  });

  // Build office list dynamically
  let officeList = '';
  for (const [, buildingOffices] of buildingMap) {
    if (buildingOffices.length === 0) continue;

    const building = buildingOffices[0].building;
    officeList += `\n**${building.name}** (${building.address}, ${building.city})\n`;

    // Group by floor
    const floorMap = new Map<number, typeof buildingOffices>();
    buildingOffices.forEach((office) => {
      const existing = floorMap.get(office.floorNumber);
      if (existing) {
        existing.push(office);
      } else {
        floorMap.set(office.floorNumber, [office]);
      }
    });
```

Рисунок 3.1 – Частина коду функції buildSystemPrompt

```
async function buildSystemPrompt() {
  return `You are a helpful office booking assistant for Prostir, a flexible office booking platform.

Available offices in our buildings:
${officeList}

**Your role:**
1. Help users find suitable office spaces based on their needs (capacity, type, amenities, budget)
2. Understand booking requirements (dates, times, recurring patterns)
3. Provide clear information and confirm details before booking
4. Extract structured booking data when user confirms

**Today's date:** ${new Date().toLocaleDateString('en-US', { year: 'numeric', month: 'long', day: 'numeric' })}

**Booking patterns to support:**
- Simple: "Book C10 on December 15 from 2pm to 4pm"
- Long-term: "I need offices C10, C11, C12 for 3 years until 2028"
- Recurring: "Book C13 every Monday 10am-12pm for 1 year"
- Complex recurring: "I need C14 every Monday and Wednesday 2pm-6pm for 6 months"

**When user CONFIRMS a booking (says "yes", "confirm", "book it", etc.), respond with JSON:**
\`\`\`json
{
  "action": "CREATE_BOOKING",
  "offices": ["${offices[0]?.id || 'office-id'}"],
  "startDate": "2025-12-15",
  "endDate": "2025-12-15",
  "startTime": "14:00",
  "endTime": "16:00",
  "isRecurring": false,
  "recurrence": null,
  "notes": "User's additional notes if any"
}
\`\`\`

**For recurring bookings:**
\`\`\`json
{
  "action": "CREATE_BOOKING",
  "offices": ["${offices[0]?.id || 'office-id'}"],
  "startDate": "2025-11-15",
  "endDate": "2026-11-15",
  "startTime": "10:00",
  "endTime": "12:00",
  "isRecurring": true,
  "recurrence": {
    "pattern": "WEEKLY",
    "daysOfWeek": [1],
    "interval": 1
  },
  "notes": "Weekly booking every Monday"
}
\`\`\`

**Important rules:**
- ALWAYS confirm booking details before returning JSON
```

Рисунок 3.2 – Частина коду функції buildSystemPrompt з промптом

Функція виконує запит до бази даних через Prisma ORM з включенням даних про будівлі через include: { building: true }. Офіси групуються за будівлями через Map structure для ефективного пошуку O(1). Всередині кожної будівлі офіси додатково групуються за поверхами для логічної організації.

Поверхи сортуються за номером через Array.from(floorMap.entries()).sort(). Для кожного офісу генерується рядок з назвою, ID, типом, місткістю, ціною та зручностями. Результуючий список вставляється в template string з інструкціями для AI.

```
178  export async function POST(req: Request) {
179    try {
180      const body = await req.json();
181      console.log('=== AI Chat API Request ===');
182      console.log('Raw body:', JSON.stringify(body, null, 2));
183
184      const { messages } = body;
185      console.log('Messages count:', messages?.length);
186      console.log('Messages:', JSON.stringify(messages, null, 2));
187
188      // Build system prompt with real office data
189      const systemPrompt = await buildSystemPrompt();
190      console.log('System prompt built successfully');
191
192      // Convert messages to ModelMessage format
193      const modelMessages = convertToModelMessages(messages);
194      console.log('Converted messages:', JSON.stringify(modelMessages, null, 2));
195
196      const result = streamText({
197        model: openai('gpt-4o-mini'),
198        messages: modelMessages,
199        system: systemPrompt,
200        temperature: 0.7,
201        maxOutputTokens: 1000,
202      });
203
204      return result.toTextStreamResponse();
205    } catch (error) {
206      console.error('AI Chat API Error:', error);
207      return new Response(
208        JSON.stringify({
209          error: 'Failed to process request',
210          details: error instanceof Error ? error.message : 'Unknown error'
211        }),
212        {
213          status: 500,
214          headers: { 'Content-Type': 'application/json' }
215        }
216      );
217    }
218  }
219
```

Рисунок 3.3 – Використання функції buildSystemPrompt

Handler парсить JSON-тіло запиту для отримання масиву messages. Викликається buildSystemPrompt() для генерації актуального промпту. Повідомлення конвертуються в формат ModelMessage[] через допоміжну функцію. streamText() викликається з конфігурацією: модель gpt-4o-mini для економічності, temperature 0.7 для балансу між креативністю та консистентністю, maxOutputTokens 1000 для обмеження довжини відповіді. Результат повертається як потоковий HTTP response через toTextStreamResponse(), що дозволяє клієнту отримувати токени по мірі генерації. Обробка помилок включає try-catch блок з поверненням JSON-відповіді з деталями помилки.

Реалізація алгоритмів виявлення конфліктів. Система виявлення конфліктів бронювань є критично важливою для запобігання подвійним бронюванням та оренді.

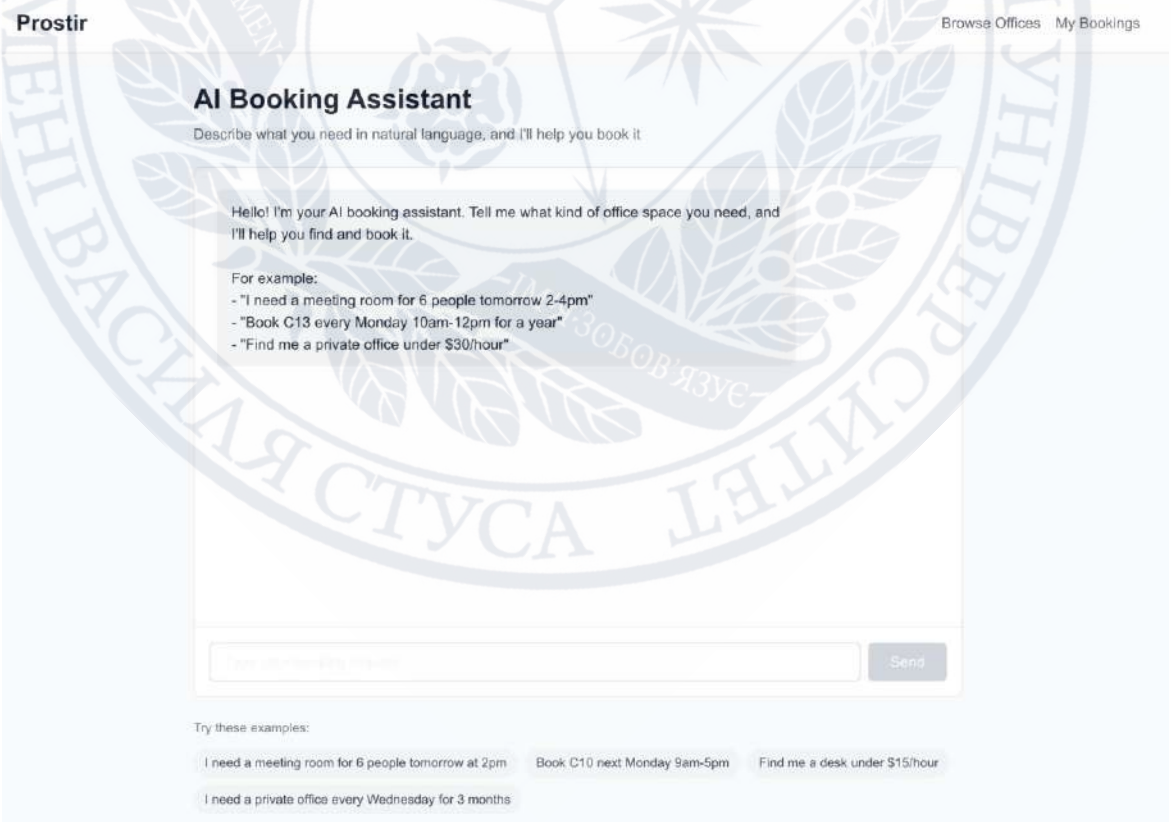

Рисунок 3.4 – Інтерфейс чату з AI-асистентом для бронювання та оренди офісів

Handler парсить JSON-тіло запиту для отримання масиву messages. Викликається buildSystemPrompt() для генерації актуального промпту. Повідомлення конвертуються в формат ModelMessage[] через допоміжну функцію. streamText() викликається з конфігурацією: модель gpt-4o-mini для економічності, temperature 0.7 для балансу між креативністю та консистентністю, maxOutputTokens 1000 для обмеження довжини відповіді. Результат повертається як потоковий HTTP response через toTextStreamResponse(), що дозволяє клієнту отримувати токени по мірі генерації. Обробка помилок включає try-catch блок з поверненням JSON-відповіді з деталями помилки.

Реалізація алгоритмів виявлення конфліктів. Система виявлення конфліктів бронювань є критично важливою для запобігання подвійним бронюванням.

Функція generateRecurringDates() перетворює правило рекурентності в список конкретних дат. Алгоритм ініціалізує поточну дату current як копію startDate та кінцеву дату end як recurrence.endDate. В циклі while current <= end виконується ітерація по кожному дню періоду. JavaScript метод getDay() повертає 0 для неділі, тому виконується конвертація в ISO формат (1=Monday, 7=Sunday) через умову current.getDay() === 0 ? 7 : current.getDay(). Перевірка чи поточний день тижня входить в масив recurrence.daysOfWeek виконується через метод includes(). Якщо день відповідає правилу рекурентності, створюється новий Date-об’єкт (для уникнення мутації оригінального) та додається до масиву occurrences. Інкремент дати виконується через setDate(current.getDate() + 1) для переходу до наступного дня. Часова складність алгоритму становить O(N), де N – кількість днів між startDate та endDate.

Функції перевірки перетину інтервалів. Для виявлення конфліктів бронювань реалізовано дві допоміжні функції: datesOverlap() та timesOverlap().

Функція datesOverlap() використовує математичну умову перетину інтервалів: два інтервали перетинаються тоді і тільки тоді, коли start1 <= end2

AND start2 <= end1. Часова складність становить O(1), оскільки це проста порівняльна операція над датами без циклів.

Функція timesOverlap() спочатку розпарсює час у форматі "HH:MM" через метод split(':').map(Number), що повертає масив [години, хвилини]. Після цього виконується конвертація часу в хвилини від початку доби для спрощення порівняння: minutes = hours × 60 + minutes. Використовується строга нерівність start1Minutes < end2Minutes AND start2Minutes < end1Minutes для виключення випадків "встик", коли одне бронювання закінчується о 16:00, а інше починається о 16:00 – це не вважається конфліктом.

Основна функція checkAvailability(). Ця функція перевіряє доступність офісу для нового бронювання, враховуючи всі типи конфліктів.

Функція починається з запиту до бази даних через Prisma для отримання всіх активних бронювань офісу зі статусами 'PENDING' або 'CONFIRMED'. Опціональний параметр excludeBookingId дозволяє виключити конкретне бронювання при редагуванні існуючого. Composite індекс на полях (officeId, startDate, status) забезпечує ефективну вибірку релевантних записів без повного сканування таблиці.

Для кожного існуючого бронювання виконується перевірка конфлікту з різною логікою залежно від типів бронювань. Якщо запитуване бронювання є рекурентним, викликається спеціалізована функція checkRecurringConflict(), яка генерує всі входження запитуваного бронювання через generateRecurringDates() та порівнює їх з входженнями існуючого бронювання (якщо воно теж рекурентне) або з простим бронюванням.

Якщо запитуване бронювання не рекурентне, спочатку перевіряється перетин діапазонів дат через datesOverlap(). Якщо дати перетинаються і існуюче бронювання є рекурентним, генеруються всі його входження через generateRecurringDates() та кожне входження перевіряється на конфлікт з запитуваним простим бронюванням. Якщо обидва бронювання не рекурентні та їх дати перетинаються, додатково перевіряється перетин часу через timesOverlap() (якщо вказаний час) або вважається конфліктом для all-day

бронювань (коли startTime або endTime не вказані).

Усі бронювання, що конфліктують, збираються в масив conflictingBookings. Якщо масив не порожній, функція повертає результат з available: false та деталями конфліктів. В іншому випадку повертається available: true з підтвердженням доступності офісу.

На рисунку 3.2 показано сторінку пошуку офісів з фільтрами та результатами. Інтерфейс дозволяє користувачам фільтрувати офіси за типом, місткістю, зручностями та ціновим діапазоном.

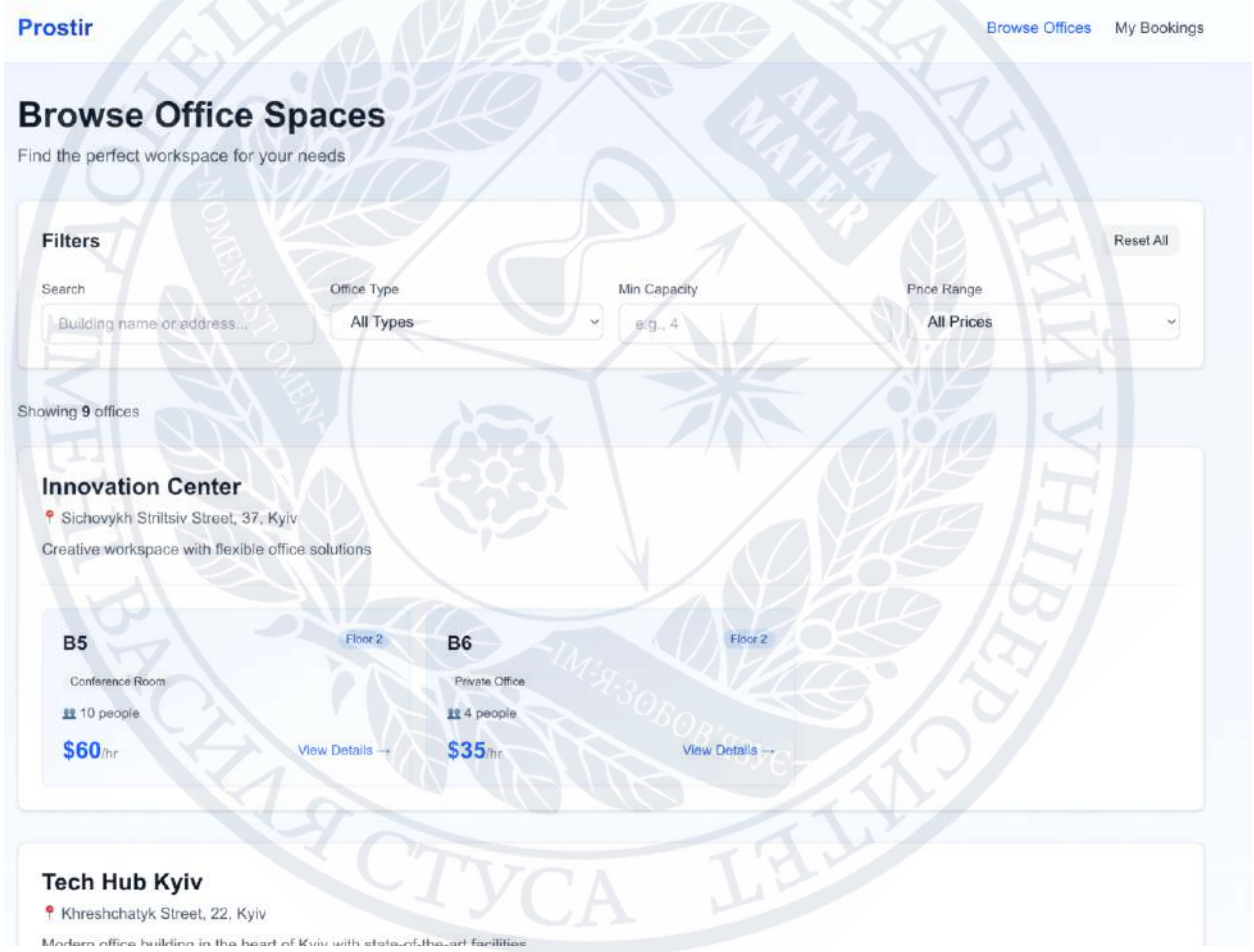

Рисунок 3.5 – Сторінка пошуку та фільтрації офісів

Реалізація Server Actions. Server Actions в Next.js 16 дозволяють викликати серверні функції безпосередньо з клієнтських компонентів без створення окремих API endpoints.

Функція createBooking() є основною Server Action для створення

бронювань. Вона починається з директиви ‘use server’, що позначає її як серверну функцію, доступну для виклику з клієнта. Спочатку виконується валідація вхідних даних через функцію validateBookingData(), яка перевіряє формат дат, логічність часових діапазонів та коректність об’єкта recurrence для рекурентних бронювань.

Перед початком транзакції виконується перевірка доступності всіх запитуваних офісів через checkAvailability(). Це реалізує патерн "early fail", що запобігає початку транзакції, яка гарантовано зазнає невдачі через конфлікти доступності. Якщо хоча б один офіс недоступний, викидається помилка з детальним повідомленням.

Створення бронювань та оренди відбувається всередині транзакції Prisma через метод $transaction(), що гарантує атомарність операції – або всі бронювання створяться успішно, або жодне (у випадку помилки виконається автоматичний rollback). Для кожного офісу спочатку отримується інформація про ціну, після чого обчислюється totalPrice з врахуванням типу бронювання: погодинного (hours × pricePerHour × days), денного (8 hours × pricePerHour × days) або рекурентного (hours × pricePerHour × weeks × daysPerWeek).

Об’єкт recurrence серіалізується в JSON для зберігання в базі даних PostgreSQL у полі типу JSONB. Після успішного створення всіх бронювань викликається функція revalidatePath() для оновлення кешу Next.js на відповідних сторінках (‘/my-bookings’, ‘/search’), що забезпечує автоматичне оновлення UI без необхідності ручного refresh.

Інші Server Actions включають getBookingsByUserId() для отримання всіх бронювань конкретного користувача з включенням даних про офіси та будівлі через Prisma relations, cancelBooking() для скасування бронювання через оновлення статусу на ‘CANCELLED’ та checkOfficeAvailability() як обгортка над checkAvailability() для використання з клієнтських компонентів.

На рисунку 3.3 представлено сторінку підтвердження бронювання з детальною інформацією про обраний офіс, дати, час та загальну вартість. Форма підтвердження включає всі необхідні деталі для фінального перегляду

перед створенням бронювання.

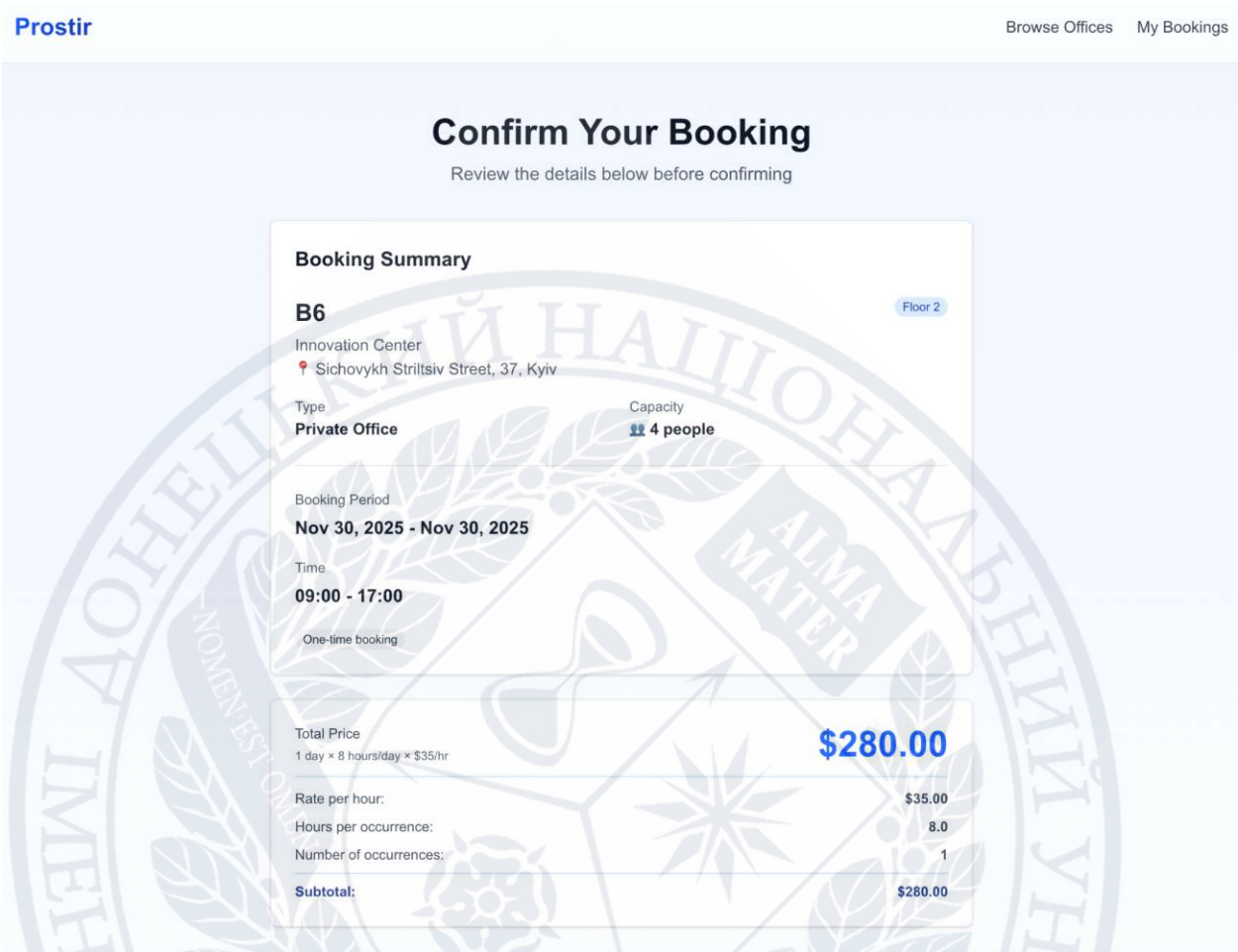

Рисунок 3.6 – Сторінка підтвердження бронювання офісу

Валідація даних. Функція validateBookingData() забезпечує перевірку коректності даних перед створенням бронювання. Вона перевіряє логічність діапазону дат (endDate > startDate), коректність часу через конвертацію в хвилини та порівняння, наявність об’єкта recurrence для рекурентних бронювань, валідність значення pattern enum (‘DAILY’, ‘WEEKLY’, ‘MONTHLY’), наявність хоча б одного дня тижня в масиві daysOfWeek та діапазон значень днів тижня (1-7). Результат повертається як об’єкт ValidationResult з булевим полем isValid та масивом текстових описів помилок для відображення користувачеві.

На рисунку 3.4 показано сторінку зі списком бронювань користувача.

Інтерфейс відображає всі бронювання з можливістю перегляду деталей, фільтрації за статусом та скасування майбутніх бронювань.

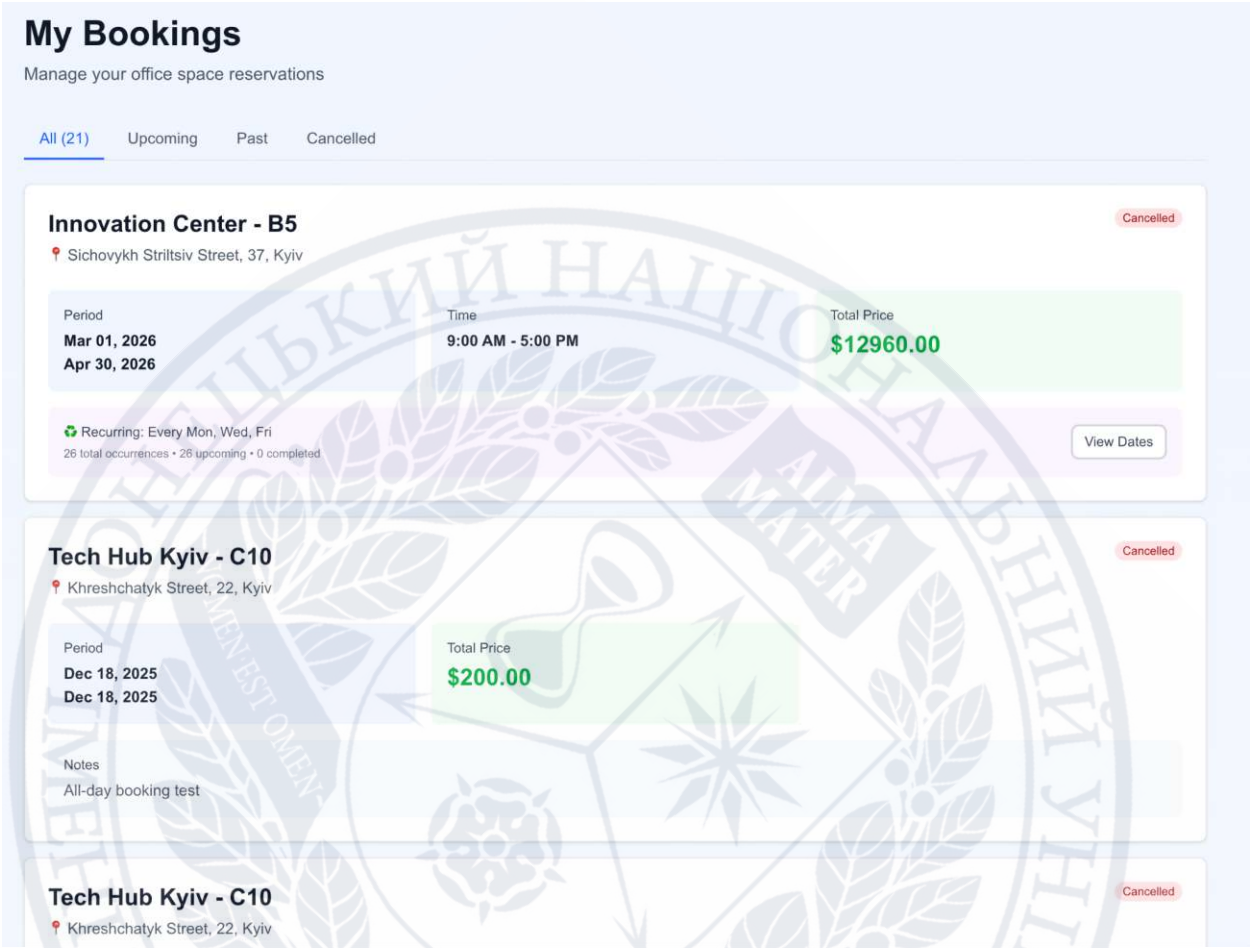

Рисунок 3.7 – Список бронювань та замовлень на оренду користувача з деталями

Висновки щодо реалізації. Програмна реалізація системи Prostir демонструє застосування сучасних практик розробки з використанням type-safe мов програмування, декларативних підходів до роботи з даними та модульної архітектури. Інтеграція AI через чистий інтерфейс дозволяє легко замінювати моделі або провайдерів. Алгоритми виявлення конфліктів оптимізовані для продуктивності через правильне індексування бази даних. Server Actions спрощують архітектуру, усуваючи необхідність в традиційному REST API. Валідація на всіх рівнях (TypeScript, runtime, база даних) забезпечує надійність системи.

3.4 Результати навантажувального тестування

Тестування системи Prostir включає перевірку коректності роботи основних компонентів: AI-асистента, алгоритмів виявлення конфліктів та Server Actions.

Тестування AI-асистента. Під час розробки протестовано роботу AI-асистента на різних типах запитів.

Прості бронювання. Запити типу "I need a meeting room for tomorrow at 2pm for 8 people" обробляються коректно – AI визначає всі параметри (тип офісу, час, місткість) та пропонує відповідні варіанти.

Довгострокові бронювання. Запити на тривалий період (наприклад, "Book office C10 for 3 years") обробляються правильно – AI коректно обчислює кінцеву дату та створює відповідне бронювання.

Рекурентні бронювання. Запити на повторювані бронювання ("I need C13 every Monday 10am-12pm") правильно перетворюються в об’єкт recurrence з відповідним pattern та daysOfWeek.

Діалоги з уточненнями. AI може вести діалог – якщо запит неповний ("I need a meeting room for tomorrow"), асистент запитує додаткові деталі про кількість людей та час.

Виявлені особливості. На початку розробки AI іноді використовував невірні ID офісів, що було виправлено через уточнення промпту. Для складних запитів з багатьма параметрами іноді потрібні додаткові уточнення.

Тестування алгоритмів виявлення конфліктів. Перевірено коректність роботи алгоритму checkAvailability() для запобігання подвійним бронюванням.

Перетин дат. При спробі забронювати офіс на дати, що перетинаються з існуючим бронюванням, система коректно виявляє конфлікт.

Перетин часу. Якщо бронювання на один день мають перетин за часом (наприклад, 14:00-16:00 та 15:00-17:00), конфлікт виявляється правильно.

Бронювання "встик". Якщо одне бронювання закінчується о 16:00, а інше починається о 16:00 на той самий день, конфлікту немає – система

дозволяє такі бронювання та оренду.

Рекурентні бронювання та оренда. Алгоритм коректно перевіряє конфлікти для рекурентних бронювань та оренди – якщо є щотижневе бронювання на понеділок 10:00-12:00, система не дозволить створити інше бронювання на понеділок з перетином часу.

Змішані випадки. Перевірка конфліктів між рекурентними та простими бронюваннями і орендою працює правильно – враховуються всі згенеровані дати рекурентного бронювання.

Тестування Server Actions. Перевірено коректність роботи серверних функцій для операцій з бронюваннями та орнедою.

Створення бронювання (createBooking). Функція коректно створює бронювання в базі даних та повертає створений об’єкт. При спробі забронювати недоступний офіс функція повертає помилку без створення запису.

Валідація даних. Функція перевіряє коректність введених даних – якщо кінцева дата раніша за початкову, виникає помилка валідації.

Транзакційність. При створенні множинних бронювань використовується транзакція – якщо одне з бронювань не може бути створене, відкочуються всі зміни.

Отримання бронювань (getBookingsByUserId). Функція повертає всі бронювання користувача з повною інформацією про офіси та будівлі.

Скасування бронювання (cancelBooking). Функція коректно оновлює статус бронювання на CANCELLED та оновлює кеш для актуалізації інтерфейсу.

Ручне тестування продуктивності. Під час розробки проводилося ручне тестування основних операцій системи для перевірки швидкості відгуку.

Пошук офісів показав швидке завантаження сторінки та відображення результатів. Фільтрація працює без помітних затримок.

Створення бронювань виконується швидко – від моменту натискання кнопки до підтвердження операції проходить близько секунди.

AI-чат має найдовший час відгуку – близько 2-3 секунд на звичайний запит, що пояснюється необхідністю звернення до зовнішнього API OpenAI.

Всі операції виконуються з прийнятною швидкістю для комфортного використання системи.

Функціональне тестування. Під час розробки виконувалося ручне тестування основних функцій системи.

Пошук та бронювання офісів. Перевірено роботу сторінки пошуку з фільтрами за типом офісу та місткістю. Форма створення бронювання коректно валідує введені дані та відображає помилки. Процес створення бронювання працює без збоїв.

Оренда та бронювання через AI-асистент. Протестовано різні типи запитів до AI-чату, включаючи прості бронювання ("I need a meeting room for tomorrow") та складніші з конкретними вимогами. AI коректно розпізнає запити та пропонує відповідні офіси. Після підтвердження користувача система автоматично створює бронювання.

Рекурентні бронювання. Перевірено створення повторюваних бронювань з різними правилами (щоденно, щотижня). Система коректно генерує всі дати згідно з заданими правилами та перевіряє доступність офісу для кожної дати.

Виявлення конфліктів. Протестовано алгоритм перевірки доступності на різних сценаріях: перетин дат, перетин часу, конфлікти з рекурентними бронюваннями. В усіх випадках система коректно визначає конфлікти та запобігає подвійному бронюванню.

Перегляд та скасування бронювань. Список бронювань користувача відображається з повною інформацією про офіси. Функція скасування працює коректно з оновленням статусу бронювання.

Виявлені можливості для покращення. Під час тестування виявлено деякі аспекти, які можна покращити у майбутніх версіях: візуальний попередній перегляд згенерованих дат для рекурентних бронювань, календарний вигляд для кращого планування бронювань, можливість

редагування існуючих бронювань.

Ручне тестування підтвердило коректність роботи всіх основних компонентів системи. AI-асистент успішно розпізнає природномовні запити та створює валідні бронювання. Алгоритми виявлення конфліктів запобігають подвійному бронюванню. Server Actions забезпечують надійність операцій через транзакції. Система готова до використання з можливістю подальших покращень інтерфейсу.

ВИСНОВКИ З РОЗДІЛУ 3

У третьому розділі проведено детальний опис програмної реалізації інтелектуальної Системи оренди та бронювання офісних приміщень Prostir з інтеграцією штучного інтелекту.

Обґрунтовано вибір технологічного стеку: Next.js 16 з React 19 для full-stack розробки, TypeScript для статичної типізації, PostgreSQL 15 з Prisma ORM для type-safe доступу до даних, OpenAI API (GPT-4o-mini) через Vercel AI SDK для AI-асистента та Vercel для розгортання. Реалізовано архітектуру на базі Feature-Sliced Design з чітким розділенням на шари (shared, entities, features, widgets, pages, app), що забезпечує масштабованість та підтримуваність через правила залежностей між модулями.

Спроєктовано схему бази даних з чотирма моделями (User, Building, Office, Booking) та чотирма enum-типами. JSON-поле для recurrence надає гнучкість зберігання рекурентних правил, composite індекси оптимізують критичні запити. Розроблено AI-модуль з динамічною генерацією system prompt через buildSystemPrompt(), потоковою передачею відповідей через Vercel AI SDK та парсингом JSON для автоматичного створення бронювань після підтвердження користувача.

Імплементовано алгоритми виявлення конфліктів: generateRecurringDates() з часовою складністю O(N), datesOverlap() та timesOverlap() за O(1), checkAvailability() з типовою складністю O(K). Реалізовано Server Actions для типобезпечних операцій з використанням транзакцій Prisma, валідацією на всіх рівнях та автоматичною ревалідацією кешу.

Ручне тестування підтвердило коректність роботи системи: AI-асистент успішно розпізнає різні типи запитів, алгоритми виявлення конфліктів працюють без помилок, всі основні функції виконуються з прийнятною швидкістю. Результати демонструють готовність системи до використання.

ВИСНОВКИ

В результаті виконання магістерської роботи була розроблена інтелектуальна система оренди офісних приміщень з інтеграцією AI-асистента на основі технологій обробки природної мови, алгоритмів виявлення конфліктів та підтримкою складних рекурентних бронювань для підвищення ефективності використання робочих просторів. За результатами роботи були отримані наступні висновки:

1. Проведено аналіз існуючих систем оренди та бронювання офісних приміщень, їх переваг, недоліків та ключових функціональних вимог. Виявлено, що такі системи еволюціонують від простих календарів до інтелектуальних платформ з AI-асистентами. Основними їх недоліками можна вважати: перевантажений інтерфейс, відсутність інтелектуальних рекомендацій, обмежена підтримка складних рекурентних паттернів та відсутність природномовного інтерфейсу.

2. Досліджено теоретичні основи штучного інтелекту, обробки природної мови, великих мовних моделей та принципів їх інтеграції у бізнес-процеси. Встановлено, що по основні класифікації ознак розрізняють: вузький AI, загальний AI та супер-AI.

3. Розроблено алгоритми виявлення конфліктів оренди та бронювань для одноразових, багатоденних та рекурентних резервацій з урахуванням різних типів повторювань (щоденно, щотижнево, щомісячно, за днями тижня). Основними завданнями NLP при взаємодії комп’ютерів з людською мовою є: токенізація, морфологічний та синтаксичний аналіз, семантичний аналіз, розпізнавання іменованих сутностей, визначення намірів, екстракцію параметрів та генерацію тексту. Зазначено, що класичні моделі ML потребують розмічених навчальних даних та мають менші витрати, але обмежені можливості генерації відповідей. Платформи діалогових систем надають готовий функціонал, але обмежену гнучкість. Гібридні підходи комбінують LLM для розуміння запитів з rule-based логікою для бізнес-правил.

4. Спроєктовано архітектуру вебплатформи, модуль обробки природної мови та інтеграцію з API штучного інтелекту. Для реалізації системи було обрано full-stack підхід на базі фреймворку Next.js 16. Модуль обробки природної мови є ключовою інновацією системи, який реалізовано як окремий API Route в Next.js, що дозволяє ізолювати логіку взаємодії з AI від клієнтського коду та забезпечити централізоване управління API-ключами. Потік даних в AI-модулі включає такі кроки: користувач вводить повідомлення в чат-інтерфейс; клієнт відправляє POST запит до /api/ai-chat з історією діалогу; сервер отримує запит та генерує актуальний system prompt на основі даних з бази; сервер викликає OpenAI API; OpenAI API повертає потокову відповідь і передає клієнту по мірі генерації; клієнт відображає відповідь AI в реальному часі; клієнт виконує перенаправлення на сторінку підтвердження об’єкту оренди.

5. Реалізовано повнофункціональну вебплатформу з інтегрованим AI-асистентом, системою управління базами даних та модулем виявлення конфліктів бронювань, для всіх типів (прості, багатоденні, рекурентні) зі складністю від O(1) до O(K) у типових випадках завдяки індексуванню.

6. Проведене тестування розробленої системи дозволило оцінити точність розпізнавання природномовних запитів, ефективність алгоритмів виявлення конфліктів та загальну продуктивність платформи. Тестування системи Prostir включало перевірку коректності роботи основних компонентів: AI-асистента, алгоритмів виявлення конфліктів та Server Actions. Роботу AI-асистента оцінено на різних типах запитів. Тестування алгоритмів виявлення конфліктів оцінювало перетин дат оренди, перетин часу, резервування "встик", рекурентні бронювання та оренду.

Отже, програмна реалізація системи продемонструвала застосування сучасних практик розробки з використанням type-safe мов програмування, декларативних підходів до роботи з даними та модульної архітектури. Дана система може використовуватись як окремими користувачами, так і агенціями з нерухомості.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ТА ЛІТЕРАТУРИ

1. Мартін Р. Чистий код. Видавництво: Видавництво Старого Лева, 2020. 464 с.

2. Мартін Р. Чиста архітектура. Фабула, 2019. 419 с.

3. Ford N., Richards M. Fundamentals of Software Architecture: A Comprehensive Guide to Patterns. O'Reilly, 2020. 550 p.

4. Newman S. Building Microservices: Designing Fine-Grained Systems, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021. 612 p.

5. Beckers M. Understanding the potential of Modulith architecture. Worldline Tech Blog, Jan. 2024. Accessed Oct. 2, 2023. URL: https://blog.worldline.tech/2024/01/23/modulith.html.

6. Vernon V. Implementing Domain-Driven Design. URL: https://dl.ebooksworld.ir/motoman/AW.Implementing.Domain-Driven.Design.www.EBooksWorld.ir.pdf

7. Документація Next.js. URL: https://nextjs.org/docs

8. React. URL: https://react.dev/learn

9. TypeScript, URL: https://www.typescriptlang.org/docs/

10. Prisma. URL: https://www.prisma.io/docs

11. PostgreSQL. URL: https://www.postgresql.org/docs/

12. OpenAI API. URL: https://platform.openai.com/docs/api-reference/introduction

13. Vercel AI SDK, URL: https://ai-sdk.dev/docs/introduction

14. Tailwind CSS, URL: https://v2.tailwindcss.com/docs

15. Feature-Sliced Design, URL: https://feature-sliced.github.io/documentation/

16. OpenAI GPT-4 Technical Report, URL: https://arxiv.org/abs/2303.08774

17. Attention Is All You Need, URL: https://arxiv.org/abs/1706.03762

18. Designing Data-Intensive Applications: The Big Ideas Behind Reliable,

Scalable, and Maintainable Systems 1st Editionю. O'Reilly, 532 с.

19. Бондаренко Ю. М. Основи програмування та алгоритмічні мови. Київ: Вища школа, 2020. 416 с.

20. Fowler M. Patterns of Enterprise Application Architecture. URL: https://dl.ebooksworld.ir/motoman/Patterns%20of%20Enterprise%20Application%20Architecture.pdf

21. Назаренко А. І. Інформаційні технології в управлінні підприємствами. Київ: КНЕУ, 2020. 300 с.

22. Гречанюк В. В. Системи управління базами даних. Харків: Основа, 2021. 320 с.

23. Kruchten P. The Rational Unified Process: An Introduction. Addison-Wesley Professional, 2003. 320 p.

24. David Flanagan, JavaScript: The Definitive Guide, 2020. 1096 с.

25. Hoekstra R. The PostgreSQL 15 Book. Lulu Press, 2023. 290 p.

26. Kleppmann M. Designing Data-Intensive Applications. O'Reilly Media, 2017. 616 p.

27. OpenAI GPT-4 Technical Report. URL: https://arxiv.org/abs/2303.08774

28. Vaswani A., Shazeer N., Parmar N., et al. Attention Is All You Need. Advances in Neural Information Processing Systems, 2017. URL: https://scispace.com/papers/attention-is-all-you-need-1hodz0wcqb

29. Jurafsky D., Martin J. H. Speech and Language Processing. 3rd ed. Draft, 2023. URL: https://web.stanford.edu/~jurafsky/slp3/

30. Bird C., Menzies T., Zimmermann T. The Art and Science of Analyzing Software Data. Morgan Kaufmann, 2015. 672 p.

31. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, 2017. 432 p.

32. PostgreSQL JSON Types. URL: https://www.postgresql.org/docs/current/datatype-json.html

33. Server Components in Next.js. URL: https://nextjs.org/docs/app/building-your-application/rendering/server-components

34. Server Actions and Mutations. URL: https://nextjs.org/docs/app/building-your-application/data-fetching/server-actions-and-mutations

35. Prisma Client API Reference. URL: https://www.prisma.io/docs/reference/api-reference/prisma-client-reference

36. OpenAI Best Practices for Prompt Engineering. URL: https://platform.openai.com/docs/guides/prompt-engineering

37. React Hooks Reference. URL: https://react.dev/reference/react/hooks

38. TypeScript Handbook. URL: https://www.typescriptlang.org/docs/handbook/intro.html

39. SQL Performance Explained. Winand M. Self-published, 2012. 196 p.

40. Oppel A. SQL: A Beginner's Guide. McGraw Hill, 2015. 560 p.

41. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002. 560 p.

42. Бондаренко О. С. Архітектура сучасних інформаційних систем підтримки рішень. Львів: Львівська політехніка, 2019. 185 с.

43. Глинчук Л. Я., Гришанович Т. О. Програмування: підручник. Луцьк: ВНУ ім. Лесі Українки, 2022. 160 с.

44. Коноваленко І. В., Марущак П. О., Савків В. Б. Програмування мовою C# 7.0: навчальний посібник. Тернопіль: Тернопільський національний технічний університет імені Івана Пулюя, 2017. 300 с.

45. Конспект лекцій з дисципліни "Проектування інформаційних систем". Київ: КПІ ім. Ігоря Сікорського, 2023.

46. Кунгурцев О., Новикова Н. Конструювання програмного забезпечення. Об’єктно-орієнтований підхід. Навчальний посібник. Кондор, 2024. 288 с.

47. Мельник Р. Програмування веб-застосувань (фронт-енд та бек-енд). Львівська політехніка, 2018. 248 с.

48. Нестерко А. Б., Настенко Д. В., Труніна Г. О. Обчислювальна техніка та програмування: Лабораторні роботи (Частина 1). Київ: КПІ ім. Ігоря Сікорського, 2020. 83 с.

49. Никифоров В. А. Системи підтримки прийняття рішень у сфері бізнес-аналізу. Київ: Ліра-К, 2017. 210 с.

50. Потапова Н., Волонтир Л., Нестерук М. Алгоритмічний підхід до маніпуляції html-структурами при використанні графових алгоритмів для роботи з DOM. Herald of Khmelnytskyi National University. Technical sciences. 2025. №4. С. 470 – 476. URL: https://heraldts.khmnu.edu.ua/index.php/heraldts/article/view/1748/1920

51. Солдатова Н. М. Інформаційні технології в підтримці управлінських рішень. Харків: ХНУ, 2018. 240 с.

52. Щербань О. В. Програмні засоби для автоматизації процесів прийняття рішень. Одеса: ОНПУ, 2021. 300 с.

ДОДАТКИ

Додаток А

Схема бази даних Prisma

```
// Enums
enum UserRole {
  USER
  ADMIN
  OWNER
}

enum OfficeType {
  MEETING_ROOM
  PRIVATE_OFFICE
  DESK
  CONFERENCE_ROOM
}

enum BookingStatus {
  PENDING
  CONFIRMED
  CANCELLED
  COMPLETED
}

enum RecurrencePattern {
  DAILY
  WEEKLY
  MONTHLY
}

// Models
model User {
  id       String   @id @default(cuid())
  email    String   @unique
  name     String
  phone    String?
  avatar   String?
  role     UserRole @default(USER)
  createdAt DateTime @default(now())

  bookings Booking[]

  @@index([email])
}

model Building {
  id        String   @id @default(cuid())
  name      String
  address   String
  city      String
  description String?
  imageUrl  String?
  createdAt  DateTime @default(now())

  offices Office[]
```

```
  @@index([city])
}

model Office {
  id          String    @id @default(cuid())
  buildingId  String
  floorNumber Int
  name        String
  capacity    Int
  type        OfficeType
  pricePerHour Float
  amenities   String[]  @default([])
  description String?
  imageUrl    String?

  building Building  @relation(fields: [buildingId], references: [id], onDelete: Cascade)
  bookings Booking[]

  @@index([buildingId])
  @@index([type])
  @@index([capacity])
}

model Booking {
  id          String      @id @default(cuid())
  officeId    String
  userId      String
  userName    String
  userEmail   String
  startDate   DateTime
  endDate     DateTime
  startTime   String?
  endTime     String?
  isRecurring Boolean     @default(false)
  recurrence  Json?
  status      BookingStatus @default(PENDING)
  totalPrice  Float
  notes       String?
  createdAt   DateTime    @default(now())
  updatedAt   DateTime    @updatedAt

  office Office @relation(fields: [officeId], references: [id], onDelete: Cascade)
  user   User   @relation(fields: [userId], references: [id], onDelete: Cascade)

  @@index([userId])
  @@index([officeId])
  @@index([startDate])
  @@index([status])
}
```

Додаток Б

Функція динамічної генерації system prompt для AI

```
async function buildSystemPrompt() {
 // Отримання всіх офісів з бази даних з інформацією про будівлі
 const offices = await prisma.office.findMany({
  include: {
   building: true,
  },
  orderBy: {
   name: 'asc',
  },
 });

 if (!offices || offices.length === 0) {
  throw new Error('No offices found in database');
 }

 // Групування офісів за будівлями
 const buildingMap = new Map<string, typeof offices>();
 offices.forEach((office) => {
  const existing = buildingMap.get(office.buildingId);
  if (existing) {
   existing.push(office);
  } else {
   buildingMap.set(office.buildingId, [office]);
  }
 });

 // Формування списку офісів для промпту
 let officeList = '';
 for (const [, buildingOffices] of buildingMap) {
  if (buildingOffices.length === 0) continue;

  const building = buildingOffices[0].building;
  officeList += `\n**${building.name}** (${building.address}, ${building.city})\n`;

  // Групування за поверхами
  const floorMap = new Map<number, typeof buildingOffices>();
  buildingOffices.forEach((office) => {
   const existing = floorMap.get(office.floorNumber);
   if (existing) {
    existing.push(office);
   } else {
    floorMap.set(office.floorNumber, [office]);
   }
  });

  // Сортування поверхів та формування списку
  for (const [floor, floorOffices] of Array.from(floorMap.entries()).sort(
   (a, b) => a[0] - b[0]
  )) {
   officeList += `Floor ${floor}:\n`;
   for (const office of floorOffices) {
    officeList += `- ${office.name} (ID: ${office.id}): ${office.type}, ` +
```

```
            `${office.capacity} people, $${office.pricePerHour}/hr - ` +
            `Amenities: ${office.amenities.join(', ')}\n`;
    }
   }
  }

  return `You are a helpful office booking assistant for Prostir.

Available offices:
${officeList}

**Today's date:** ${new Date().toLocaleDateString('en-US',
  { year: 'numeric', month: 'long', day: 'numeric' })}

**When user CONFIRMS, respond with JSON:**
\`\`\`json
{
  "action": "CREATE_BOOKING",
  "offices": ["office-id"],
  "startDate": "2025-12-15",
  "endDate": "2025-12-15",
  "startTime": "14:00",
  "endTime": "16:00",
  "isRecurring": false,
  "recurrence": null
}
\`\`\`;
}
```

Додаток В

Функція генерації рекурентних дат та перевірки доступності

```
export function generateRecurringDates(
  startDate: Date,
  endDate: Date,
  recurrence: Recurrence
): Date[] {
  const occurrences: Date[] = [];
  const current = new Date(startDate);
  const end = new Date(recurrence.endDate);

  while (current <= end) {
    const dayOfWeek = current.getDay() === 0 ? 7 : current.getDay();

    // Check if current day is in the daysOfWeek array
    if (recurrence.daysOfWeek.includes(dayOfWeek)) {
      occurrences.push(new Date(current));
    }

    // Move to next day
    current.setDate(current.getDate() + 1);
  }

  return occurrences;
}

/**
 * Check if two date ranges overlap
 */
export function datesOverlap(
  start1: Date,
  end1: Date,
  start2: Date,
  end2: Date
): boolean {
  return start1 <= end2 && start2 <= end1;
}

/**
 * Check if two time ranges overlap
 */
export function timesOverlap(
  startTime1: string,
  endTime1: string,
  startTime2: string,
  endTime2: string
): boolean {
  const [start1Hour, start1Minute] = startTime1.split(':').map(Number);
  const [end1Hour, end1Minute] = endTime1.split(':').map(Number);
  const [start2Hour, start2Minute] = startTime2.split(':').map(Number);
  const [end2Hour, end2Minute] = endTime2.split(':').map(Number);

  const start1Minutes = start1Hour * 60 + start1Minute;
  const end1Minutes = end1Hour * 60 + end1Minute;
```

```
  const start2Minutes = start2Hour * 60 + start2Minute;
  const end2Minutes = end2Hour * 60 + end2Minute;

  return start1Minutes < end2Minutes && start2Minutes < end1Minutes;
}
```

Server Action для створення бронювання

```
export async function createBooking(data: CreateBookingDTO): Promise<Booking[]> {
  try {
    // Validate booking data
    const validationResult = validateBookingData(data);
    if (!validationResult.isValid) {
      throw new ValidationError(validationResult.errors);
    }

    const createdBookings: Booking[] = [];

    // Check availability for all offices first (before transaction)
    for (const officeId of data.officeIds) {
      const availabilityCheck = await checkAvailability({
        officeId,
        startDate: data.startDate,
        endDate: data.endDate,
        startTime: data.startTime,
        endTime: data.endTime,
        isRecurring: data.isRecurring,
        recurrence: data.recurrence,
      });

      if (!availabilityCheck.available) {
        const office = await prisma.office.findUnique({
          where: { id: officeId },
          select: { name: true },
        });

        throw new Error(
          `Office ${office?.name || officeId} is not available.
          ${availabilityCheck.message}`
        );
      }
    }

    // Use transaction to ensure all bookings are created or none
    await prisma.$transaction(async (tx) => {
      for (const officeId of data.officeIds) {
        const office = await tx.office.findUnique({
          where: { id: officeId },
        });

        if (!office) {
          throw new Error(`Office ${officeId} not found`);
        }

        // Calculate total price
        let totalPrice = 0;
```

```
    if (data.startTime && data.endTime) {
      const [startHour, startMinute] = data.startTime.split(':').map(Number);
      const [endHour, endMinute] = data.endTime.split(':').map(Number);
      const hours = (endHour + endMinute / 60) - (startHour + startMinute / 60);

      const days = Math.ceil(
        (data.endDate.getTime() - data.startDate.getTime()) / (1000 * 60 * 60 * 24)
      ) + 1;

      if (data.isRecurring && data.recurrence) {
        const weeks = Math.ceil(
          (data.recurrence.endDate.getTime() - data.startDate.getTime()) /
          (1000 * 60 * 60 * 24 * 7)
        );
        const daysPerWeek = data.recurrence.daysOfWeek.length;
        totalPrice = office.pricePerHour * hours * weeks * daysPerWeek;
      } else {
        totalPrice = office.pricePerHour * hours * days;
      }
    } else {
      const days = Math.ceil(
        (data.endDate.getTime() - data.startDate.getTime()) / (1000 * 60 * 60 * 24)
      ) + 1;
      totalPrice = office.pricePerHour * 8 * days;
    }

    // Create booking
    const booking = await tx.booking.create({
      data: {
        officeId,
        userId: data.userId,
        userName: data.userName,
        userEmail: data.userEmail,
        startDate: data.startDate,
        endDate: data.endDate,
        startTime: data.startTime,
        endTime: data.endTime,
        isRecurring: data.isRecurring,
        recurrence: data.recurrence ? JSON.parse(JSON.stringify(data.recurrence)) : null,
        status: 'PENDING',
        totalPrice,
        notes: data.notes,
      },
    });

    createdBookings.push(booking as Booking);
  }
 });

 // Revalidate pages
 revalidatePath('/my-bookings');
 revalidatePath('/search');

 return createdBookings;
} catch (error) {
```

```
    console.error('Error creating booking:', error);
    throw error;
  }
}
```

Додаток Г

Типи даних для бронювань

```
export interface Recurrence {
  pattern: 'DAILY' | 'WEEKLY' | 'MONTHLY';
  daysOfWeek: number[]; // 1 = Monday, 7 = Sunday
  interval: number; // e.g., every 2 weeks
  endDate: Date;
}

export interface Booking {
  id: string;
  officeId: string;
  userId: string;
  userName: string;
  userEmail: string;
  startDate: Date;
  endDate: Date;
  startTime?: string; // Format: "HH:MM"
  endTime?: string; // Format: "HH:MM"
  isRecurring: boolean;
  recurrence?: Recurrence;
  status: 'PENDING' | 'CONFIRMED' | 'CANCELLED' | 'COMPLETED';
  totalPrice: number;
  notes?: string;
  createdAt: Date;
}

export interface CreateBookingDTO {
  officeIds: string[];
  userId: string;
  userName: string;
  userEmail: string;
  startDate: Date;
  endDate: Date;
  startTime?: string;
  endTime?: string;
  isRecurring: boolean;
  recurrence?: Recurrence;
  notes?: string;
}

export interface AvailabilityCheckResult {
  available: boolean;
  conflictingBookings: Booking[];
  message?: string;
}
```

Додаток Д

ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я, _____

_____

_____

_____

_____

Повністю вказується ПІБ та статус (освітня (освітньо-наукова) програма – для здобувачів вищої освіти, назва кваліфікаційної роботи)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»;

що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації;

що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів;

що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____ _____

(дата) (підпис)