Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)
Кафедра Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Розробка інтелектуальної автоматизованої системи з бронювання квитків на авіаперельоти
(тема)

Виконав:
здобувач четвертого року навчання,
групи ІТШІ-22-2
Дмитро АВІЛОВ
(власне ім'я, прізвище)
Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Штучний інтелект
(повна назва освітньої програми)
Керівник ас. Микола ЧЕРНЕНКО
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ШІ _____ Лариса ЧАЛА
(підпис) (власне ім'я, прізвище)

2026 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Штучного інтелекту

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Штучний інтелект
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«_____» _____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Авілову Дмитру Юрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка інтелектуальної автоматизованої системи з бронювання квитків на авіаперельоти

затверджена наказом університету від 5 червня 20 26 р. № 488Ст

2. Термін подання студентом роботи до екзаменаційної комісії 7 липня 20 26 р.

3. Вихідні дані до роботи відкриті авіаційні набори даних OpenFlights та OurAirports, що містять інформацію про авіакомпанії, аеропорти та маршрути; мова програмування Python, фреймворки FastAPI та React, мова TypeScript, СУБД PostgreSQL, середовище розробки Visual Studio Code, інструменти тестування pytest.

4. Перелік питань, що потрібно опрацювати в роботі

1) Теоретичні основи побудови систем бронювання в авіаційній галузі

2) Постановка задачі та загальна концепція реалізації системи

3) Проєктування та реалізація системи бронювання

4) Тестування та аналіз результатів

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	08.06.2026	виконано
2	Аналіз предметної галузі	12.06.2026	виконано
3	Постановка задачі	16.06.2026	виконано
4	Обґрунтування вибору використаних технологій	20.06.2026	виконано
5	Програмна реалізація	25.06.2026	виконано
6	Оформлення пояснювальної записки	30.06.2026	виконано
7	Підготовка до захисту	04.07.2026	виконано
8	Захист перед ЕК	07.07.2026	виконано

Дата видачі завдання 8 червня 20 26 р.

Здобувач _____
(підпис)

Керівник роботи _____ ас. Микола ЧЕРНЕНКО
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ

Пояснювальна записка: 105 с., 18 рис., 2 табл., 1 дод., 17 джерел.

БРОНЮВАННЯ, ПЕРСОНАЛІЗАЦІЯ, РАНЖУВАННЯ, API, ETL, JSON, ML, ORM, REST, SGD, SPA, SQL.

Об’єкт дослідження – процес автоматизованого відбору та ранжування авіамаршрутів в умовах багатокритеріального вибору.

Предмет дослідження – методи та алгоритми інтелектуального ранжування результатів пошуку авіарейсів з урахуванням персональних вподобань користувача.

Мета роботи – проєктування та реалізація інтелектуальної системи бронювання авіаквитків із гібридним підходом до ранжування, що поєднує rule-based фільтрацію та логістичну регресію з персоналізацією.

Методи дослідження – rule-based фільтрація, машинне навчання, багатокритеріальний аналіз, об’єктно-реляційне відображення, модульне та інтеграційне тестування.

Реалізовано трирівневу клієнт-серверну систему на FastAPI, React і PostgreSQL. Розроблено ETL-модуль для імпорту даних OpenFlights та OurAirports і генерації понад 108 тисяч рейсів. Реалізовано AI-пайплайн ранжування з витягом ознак, логістичною регресією та поясненням результатів. Механізм персоналізації динамічно перераховує вагові коефіцієнти без перенавчання моделі. Результати підтверджено автоматизованим тестуванням.

ABSTRACT

Bachelor’s thesis contains: 105 pp., 18 fig., 2 tabl., 1 ann., 17 references.

API, BOOKING, ETL, JSON, ML, ORM, PERSONALIZATION, RANKING, REST, SGD, SPA, SQL.

The object of the research is the process of automated selection and ranking of airline routes under conditions of multi-criteria decision-making.

The subject of the research is methods and algorithms for intelligent ranking of flight search results based on users’ personal preferences and travel requirements.

The aim of the work is to design and implement an intelligent airline ticket booking system using a hybrid ranking approach that combines rule-based filtering with logistic regression and personalization mechanisms. The system is intended to improve the relevance of search results, enhance user experience, and support informed decision-making during flight selection.

The research methods include rule-based filtering, machine learning techniques, multi-criteria analysis, object-relational mapping, as well as modular and integration testing.

A three-tier client-server system was implemented using FastAPI, React, and PostgreSQL. An ETL module was developed for importing data from OpenFlights and OurAirports datasets and generating more than 108,000 flight records. An AI-driven ranking pipeline was implemented, including feature extraction, logistic regression, and result explanation mechanisms. The personalization component dynamically recalculates weighting coefficients according to user preferences without requiring model retraining. The effectiveness and reliability of the proposed solution were validated through automated testing, demonstrating the system’s capability to provide accurate, transparent, and personalized flight recommendations.

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів 9
Вступ 10
1 Теоретичні основи побудови систем бронювання в авіаційній галузі 12
1.1 Сучасні виклики у сфері бронювання авіаквитків 12
1.2 Поняття та роль ІС бронювання в цифровому середовищі 14
1.2.1 Визначення та функціональні можливості 14
1.2.2 Роль інтелектуальної системи бронювання у процесі вибору та придбання авіаквитків 16
1.3 Моделі та підходи штучного інтелекту в інтелектуальній системі бронювання авіаквитків 17
1.3.1 Rule-based підхід для формування та фільтрації рейсів 17
1.3.2 Рекомендаційні алгоритми 19
1.3.3 Гібридний підхід: поєднання правил і рекомендацій 21
1.4 Переваги впровадження ІС бронювання 23
2 Постановка задачі та проєктування системи 25
2.1 Мета та завдання кваліфікаційної роботи 25
2.2 Архітектура системи, структурна специфікація функціональних модулів системи 28
2.3 Типи користувачів та сценарії використання системи 33
2.4 Результати впровадження та практичне значення системи 37
3 Проєктування та реалізація системи бронювання 40
3.1 Загальна архітектура програмної системи 40
3.1.1 Архітектура клієнт-серверного застосунку 40
3.1.2 Архітектура серверної частини 43
3.1.3 Архітектура клієнтської частини 46
3.2 Проєктування бази даних 48
3.2.1 Архітектура ETL-модуля 50
3.2.2 Проєктування логічної моделі даних 51

3.2.3 Реалізація моделей з використанням SQLAlchemy 53
3.2.4 Реалізація таблиць користувачів та уподобань 55
3.2.5 Реалізація таблиць рейсів та маршрутів 56
3.2.6 Реалізація таблиць бронювання 57
3.3 Реалізація модулів та REST API 59
3.3.1 Реалізація REST API 59
3.3.2 Реалізація системи керування користувачами та їхніми уподобаннями 60
3.3.3 Реалізація модуля пошуку рейсів 63
3.3.4 Реалізація модуля бронювання 66
3.3.5 Реалізація системи логування та обробки помилок 68
3.4 Реалізація клієнтської частини системи 70
3.4.1 Реалізація структури frontend-застосунку 70
3.4.2 Реалізація сторінки пошуку рейсів 72
3.4.3 Реалізація сторінки бронювання 74
3.4.4 Реалізація сторінки налаштувань користувача 75
3.4.5 Взаємодія frontend із backend 77
3.5 Реалізація алгоритмів пошуку та персоналізації 79
3.5.1 Реалізація алгоритму пошуку рейсів 79
3.5.2 Реалізація алгоритму пошуку пересадок 81
3.5.3 Реалізація алгоритму ранжування результатів 83
3.5.4 Реалізація модулю персоналізації результатів 86
4 Практична реалізація та апробація результатів 88
4.1 Організація процесу тестування 88
4.2 Апробація серверної частини 89
4.2.1 Апробація REST API та модулів користувачів 89
4.2.2 Апробація пошуку рейсів 91
4.2.3 Апробація бронювання 93
4.3 Апробація клієнтської частини 94
4.4 Апробація алгоритмів пошуку та персоналізації 97

4.4.1 Перевірка пошуку прямих рейсів та рейсів із пересадками 97
4.4.2 Перевірка алгоритму ранжування 98
4.4.3 Аналіз персоналізації результатів 100
Висновки 101
Перелік джерел посилання 103
Додаток А Відомість кваліфікаційної роботи 105

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

БД – база даних;

СУБД – система управління базами даних;

ШІ – штучний інтелект;

API – Application Programming Interface – інтерфейс програмування застосунків;

DTO – Data Transfer Object – об’єкт передачі даних;

ETL – Extract, Transform, Load – процес вилучення, перетворення та завантаження даних;

GDS – Global Distribution System – глобальна дистрибутивна система;

IATA – International Air Transport Association – Міжнародна асоціація повітряного транспорту;

ICAO – International Civil Aviation Organization – Міжнародна організація цивільної авіації;

JSON – JavaScript Object Notation – формат обміну даними;

JWT – JSON Web Token – токен для автентифікації;

ML – Machine Learning – машинне навчання;

ORM – Object-Relational Mapping – об’єктно-реляційне відображення;

REST – Representational State Transfer – архітектурний стиль побудови API;

SGD – Stochastic Gradient Descent – стохастичний градієнтний спуск;

SPA – Single Page Application – односторінковий застосунок;

SQL – Structured Query Language – мова структурованих запитів;

URL – Uniform Resource Locator – уніфікований локатор ресурсів.

ВСТУП

Розвиток цифрових технологій у сфері авіаційних перевезень призвів до суттєвого збільшення кількості доступних маршрутів, тарифних пропозицій і комбінацій пересадок. За таких умов процес вибору авіарейсу перетворився на складне багатокритеріальне завдання, яке вимагає від користувача значних аналітичних зусиль. Традиційні системи бронювання надають доступ до великих масивів інформації, однак не беруть участі у процесі прийняття рішень – весь аналітичний тягар лишається на плечах пасажира. Це зумовлює когнітивне перевантаження, подовжує час вибору та підвищує ймовірність прийняття неоптимального рішення.

Відповіддю на ці виклики є інтелектуальні системи підтримки прийняття рішень, що здатні автоматично фільтрувати, ранжувати та пояснювати результати пошуку з урахуванням індивідуальних вподобань користувача. Поєднання rule-based підходу, що забезпечує формалізований контроль над допустимістю маршрутів, із рекомендаційними алгоритмами на основі машинного навчання дозволяє будувати гнучкі та прозорі системи, здатні адаптуватись до потреб конкретного користувача.

Метою кваліфікаційної роботи є проєктування та реалізація інтелектуальної системи бронювання авіаквитків, що забезпечує автоматизоване ранжування маршрутів на основі гібридного підходу – поєднання правил відбору та моделі логістичної регресії з персоналізацією за ваговими коефіцієнтами користувача.

Для досягнення мети в роботі вирішуються такі завдання: аналіз предметної галузі та існуючих підходів до інтелектуалізації систем бронювання; проєктування архітектури системи та бази даних; реалізація серверної частини на основі FastAPI з модулями пошуку, ранжування та персоналізації; реалізація клієнтської частини на основі React із підтримкою налаштування вподобань; імпорт реальних авіаційних даних через ETL-модуль; тестування та аналіз результатів роботи системи.

Об’єктом дослідження є процес автоматизованого відбору та ранжування авіамаршрутів в умовах багатокритеріального вибору. Предметом дослідження є методи та алгоритми інтелектуального ранжування результатів пошуку рейсів з урахуванням персональних вподобань користувача.

Практична цінність роботи полягає у реалізації повноцінного прототипу системи бронювання авіаквитків із власним AI-модулем, що включає пояснюваність рішень, механізм персоналізації та інтеграцію з реальними авіаційними даними OpenFlights та OurAirports.

1 ТЕОРЕТИЧНІ ОСНОВИ ПОБУДОВИ СИСТЕМ БРОНЮВАННЯ В АВІАЦІЙНІЙ ГАЛУЗІ

1.1 Сучасні виклики у сфері бронювання авіаквитків

Сфера бронювання авіаквитків впродовж останнього десятиліття зазнала істотних змін, зумовлених цифровізацією авіаційної індустрії, зростанням пасажиропотоку та ускладненням ринкової структури авіаперевезень. Відповідно до звітів міжнародних авіаційних організацій, кількість авіапасажирів у світі стабільно зростала до пандемічного періоду, а після 2021 року відновлення супроводжується ще більшою диверсифікацією маршрутів і тарифних моделей, що призвело до значного збільшення кількості доступних пропозицій для одного й того самого напрямку, що безпосередньо впливає на складність процесу вибору рейсу з боку користувача [1].

Одним із ключових викликів сучасних систем бронювання є фрагментованість і неоднорідність авіаційних даних. Інформація про рейси, тарифи, додаткові послуги та правила перевезення формується й зберігається в різних інформаційних середовищах: у внутрішніх системах авіакомпаній, глобальних дистрибутивних системах (GDS), метапошукових платформах і незалежних агрегаторах.

Навіть у межах одного каналу продажу дані можуть відрізнятися за структурою та рівнем деталізації. У звітах Міжнародної асоціації повітряного транспорту (IATA) наголошується, що такого роду фрагментація ускладнює інтеграцію інформації та створює бар’єри для прозорого й узгодженого відображення пропозицій кінцевому користувачеві [1].

Додатковою складністю є еволюція бізнес-моделей авіакомпаній. Поширення лоукост-перевізників, динамічне ціноутворення, продаж додаткових послуг окремими пакетами та персоналізовані тарифи призвели

до того, що ціна квитка більше не є єдиним або навіть основним показником вартості подорожі. Кінцева вартість залежить від набору опцій, умов багажу, можливості зміни дати, вибору місця та інших факторів. За подібних умов користувачеві стає складно порівнювати пропозиції різних перевізників, оскільки формально дешевший квиток може виявитися менш вигідним з урахуванням додаткових витрат.

Суттєвим викликом залишається багатокритеріальний характер вибору авіарейсу. Рішення про бронювання формується на основі сукупності параметрів, серед яких вартість, тривалість маршруту, кількість пересадок, час вильоту та прильоту, а також індивідуальні обмеження користувача. Для різних пасажирів ці критерії мають різну вагу, що унеможливлює застосування універсального підходу до ранжування рейсів. Більшість традиційних систем бронювання реалізують обмежений набір фільтрів що, не враховують складних компромісів між параметрами та змушує користувача виконувати аналітичну роботу самостійно.

Окрему проблему становить інформаційне перевантаження, яке виникає внаслідок надмірної кількості варіантів і супровідної інформації. Сучасні сервіси бронювання поєднують результати пошуку з рекламними елементами, акційними пропозиціями та візуальними маркерами, що впливають на сприйняття інформації. Дослідження у сфері рекомендаційних систем і поведінки користувачів у цифрових сервісах показують, що за умов надлишку альтернатив користувачі схильні приймати рішення на основі спрощених евристик, а не раціонального аналізу всіх доступних варіантів [2]. Це знижує ефективність процесу вибору й може призводити до менш оптимальних рішень.

Важливим аспектом є також відсутність глибокої персоналізації в більшості наявних систем бронювання. Незважаючи на доступність великих обсягів даних про попередні запити, бронювання та поведінку користувачів, результати пошуку часто залишаються однаковими для різних категорій пасажирів. Це обмежує можливості адаптації сервісу до індивідуальних

потреб і не дозволяє враховувати контекст подорожі, наприклад, ділову чи туристичну мету поїздки, часові обмеження або переваги щодо комфорту.

У таких умовах зростає потреба в автоматизованих інтелектуальних рішеннях, здатних працювати з неоднорідними джерелами даних, виконувати багатокритеріальний аналіз і враховувати користувацькі вподобання. Саме ці виклики формують підґрунтя для застосування рекомендаційних систем і rule-based підходів у сфері бронювання авіаквитків, де автоматизоване ранжування та пояснюваний вибір варіантів можуть суттєво підвищити ефективність взаємодії користувача з сервісом.

1.2 Поняття та роль ІС бронювання в цифровому середовищі

1.2.1 Визначення та функціональні можливості

Інтелектуальна система бронювання авіаквитків розглядається як спеціалізована інформаційна система, що реалізує функції підтримки прийняття рішень у процесі вибору та бронювання авіарейсів. На відміну від традиційних систем пошуку, які виконують запити за заданими параметрами та відображають результати у вигляді неструктурованого списку, інтелектуальні системи здійснюють аналітичну обробку доступних варіантів і беруть участь у формуванні рекомендацій для користувача. У працях з бізнес-аналітики та інформаційних систем такі рішення відносять до класу decision support systems, орієнтованих на зменшення складності вибору в умовах багатокритеріальних задач [3].

Головна відмінність між розумною системою бронювання та звичайними пошуковими інструментами полягає у зміні ролі, яку система відіграє у взаємодії з клієнтом. Якщо традиційний сервіс виступає як пасивний посередник між користувачем та джерелом інформації, то інтелектуальний механізм активно аналізує наявні варіанти, бере до уваги усі обмеження та пріоритети, а потім пропонує вже структуруйовані

результати, знаменуючи перехід від простого пошуку до автоматизованого відбору й оцінки можливих альтернатив, що є критичним для завдань пошуку авіаперельотів, де кількість опцій може бути надзвичайно великою.

Функціональна основа інтелектуальної системи бронювання побудована на інтеграції рекомендаційного та rule-based (на основі правил) підходів. Рекомендаційна частина використовується для індивідуального підбору рейсів та мінімізації надлишкової інформації, формуючи лише ті варіанти, що є найбільш значущими. Рекомендаційні системи описуються як засоби, що залучають дані про клієнта, поточні умови та загальні тенденції для прогнозування найбільш підходящих виборів [2].

У свою чергу, система, що оперує правилами, втілює формалізований механізм відбору через набір фіксованих умов та обмежень, дозволяючи чітко встановити критерії виключення або надання переваги певним рейсам, скажімо, за числом пересадок, часовими рамками або максимально допустимою тривалістю подорожі. Сильною стороною rule-based моделей є їхня прозорість та можливість контролю, що є важливим для сервісів, які безпосередньо впливають на рішення користувача.

У сучасних системах ці дві методології часто комбінуються у гібридні архітектури, де правила встановлюють базові рамки, а алгоритми рекомендацій здійснюють фінальне сортування серед дозволених варіантів [2]. До головних можливостей, які має інтелектуальна система пошуку квитків на літак, відносять:

– автоматизований вибір рейсів, ґрунтуючись на заданих вимогах та лімітах;

– ретельне виключення невідповідних пропозицій з огляду на часові, фінансові та логістичні параметри;

– багатокритеріальне впорядкування рейсів з урахуванням коефіцієнтів важливості;

– врахування індивідуальних клієнтських вподобань, які виражені явно або виявлені на основі попередніх сесій;

– адаптація алгоритму відбору під типові сценарії використання, зокрема для відряджень чи турпоїздок.

Врахування сценаріїв користувача є суттєвою ознакою розумної системи, оскільки це дає змогу брати до уваги контекст застосування сервісу. У дослідженнях, сфокусованих на індивідуалізованих транспортних послугах, наголошується, що аналіз зразків поведінки та сценаріїв суттєво покращує релевантність пропозицій та зменшує кількість непотрібних результатів [4]. У сервісах бронювання це реалізується через збереження профілю клієнта, аналіз історії його замовлень та застосування відповідних моделей ранжування чи правил.

1.2.2 Роль інтелектуальної системи бронювання у процесі вибору та придбання авіаквитків

Процес вибору авіаквитка в онлайн-середовищі характеризується необхідністю одночасного врахування фінансових, часових та індивідуальних параметрів. Користувач змушений опрацьовувати значний масив варіантів, які за зовнішньою схожістю можуть суттєво різнитися за своїм впливом на якість подорожі. В умовах такої складності інтелектуальна система бронювання виступає інструментом підтримки прийняття рішень, що забезпечує структуризацію доступної інформації та перетворення неупорядкованої множини пропозицій на ранжований перелік релевантних альтернатив.

Підтримка користувача на етапі фіналізації вибору означає здатність системи аналізувати запит не окремо, а беручи до уваги обставини і встановлені межі. Розумний механізм може самостійно порівнювати характеристики авіарейсів, впроваджувати критерії відбору та формувати організований перелік альтернатив, що задовольняють ключові вимоги клієнта. Дослідницькі роботи у галузі підтримки рішень за численними критеріями доводять, що використання подібних методів сприяє

зменшенню невизначеності вибору та підвищує його обґрунтованість при складних завданнях з розмаїттям опцій [5].

Важливим моментом у функції інтелектуальної системи є зниження інтелектуального навантаження при роботі з платформою бронювання. Коли людина сама розглядає десятки схожих варіантів, зростає ймовірність перевтоми, прийняття невірних рішень та залишення процесу бронювання незавершеним. Розумні механізми сортування та фільтрації дають змогу зменшити обсяг даних, які треба опрацювати, і представити їх у форматі, зручному для швидкої звірки. У працях, присвячених цифровим мандрівним сервісам, наголошується, що послаблення розумового напруження позитивно впливає на швидкість взаємодії та загальне враження користувача [6].

Застосування алгоритмів ранжування та формалізованих правил дозволяє відсіювати неактуальні пропозиції вже на початкових етапах пошуку, скорочуючи кількість повторень між оглядом результатів та ухваленням рішення – що особливо важливо в умовах обмеженого часу клієнта. Окрему роль відіграє автоматизація типових послідовностей дій – пошуку маршрутів, порівняння схожих рейсів та переходу до оформлення квитка. Попередньо визначені схеми дій мінімізують ручні втручання і забезпечують злагодженість між етапами вибору та придбання, що в сукупності спрощує взаємодію користувача з системою бронювання.

1.3 Моделі та підходи штучного інтелекту в інтелектуальній системі бронювання авіаквитків

1.3.1 Rule-based підхід для формування та фільтрації рейсів

Підхід, що ґрунтується на правилах, у системах бронювання авіаперевезень спирається на використання чітко визначених норм, які окреслюють умови – дозволені, заборонені чи пріоритетні – для вибору

конкретних рейсів. Ці норми встановлюються явно і відображають комерційну логіку сервісу, ліміти предметної сфери, а також головні вимоги кінцевого користувача. На практиці, механізми, керовані правилами, залучаються на початкових стадіях обробки результатів пошуку, ще до застосування складніших технологій машинного навчання, що дає змогу звузити діапазон потенційних рішень та гарантувати коректність сформованих пропозицій.

Норми, котрі стосуються відсіювання, застосовуються для автоматичного виключення тих рейсів, які не відповідають встановленим критеріям або є об’єктивно неприйнятними. У системах бронювання до таких норм можуть належати:

– перевищення дозволеної кількості стикувань;

– надмірна загальна тривалість подорожі;

– виліт або прибуття поза заздалегідь визначеним часовим вікном;

– невідповідність запитуваному рівню обслуговування чи авіаперевізнику;

– відсутність вільних місць за вказаною ціновою категорією.

Використання правил відсіювання дозволяє суттєво скоротити кількість доступних альтернатив на самому початку і пришвидшити подальшу обробку запиту.

Логічні обмеження у парадигмі, що базується на правилах, відповідають за забезпечення внутрішньої узгодженості усіх параметрів маршруту між собою, а також із загальним контекстом подорожі. Вони дають змогу формалізувати практичні вимоги, які непросто охопити простими фільтрами. Серед прикладів таких обмежень варто виділити:

– мінімально необхідний інтервал часу між послідовними сегментами перельоту;

– заборона комбінування несумісних умов тарифу;

– обмеження щодо зміни аеропорту для пересадки;

– залежності між загальною тривалістю перельоту та максимально допустимою кількістю зупинок.

Подібні норми унеможливлюють формування маршрутів, які є формально правильними, але практично нездійсненними, чим забезпечують реалістичність пропонованих результатів пошуку.

Окрему важливість у підході, що оперує правилами, має можливість пояснення ухвалених рішень. Оскільки кожне рішення має під собою конкретну норму чи сукупність норм, система здатна надати чітке обґрунтування, чому саме цей рейс було відхилено або, навпаки, увімкнено у список пропозицій. Це сприяє:

– збільшенню рівня довіри користувачів до запропонованих варіантів;

– демонстрації прозорості архітектури роботи сервісу;

– полегшенню аудиту та тонкого налаштування бізнес-правил;

– оперативній адаптації системи до змін у вимогах чи регуляторних приписів.

У рамках інтелектуальної системи бронювання, підхід на базі правил слугує контрольованим та передбачуваним механізмом для початкового формування добірки рішень. Саме на цьому рівні задаються фундаментальні обмеження та умови, після чого може бути застосовано рекомендаційні чи статистичні моделі для подальшого ранжування та персоналізації знайдених варіантів.

1.3.2 Рекомендаційні алгоритми

У системах бронювання авіаквитків алгоритми, що надають рекомендації, залучаються для впорядкування та відбору доступних рейсів тоді, коли формально прийнятних варіантів є надзвичайно багато. На відміну від суворих, жорстко заданих правил, рекомендаційний метод зосереджується на роботі з порівняльними перевагами, ваговими коефіцієнтами та пошуком балансу між різними параметрами. Це дає змогу

системі відсіювати невідповідні маршрути, а також пропонувати клієнту найбільш слушні альтернативи, зважаючи на обставини його запиту.

Одне з центральних завдань для алгоритмів, що рекомендують, полягає у сортуванні рейсів на основі комплексу їхніх характеристик. Кожен здійснений маршрут описується набором як числових, так і якісних ознак, на підставі яких формується впорядкований список результатів. У сфері бронювання авіаперельотів до таких даних зазвичай зараховують:

– вартість квитка та умови, закладені у тариф;

– загальний час мандрівки, включаючи час очікування під час стикувань;

– кількість етапів, з яких складається шлях;

– часові рамки вильоту з початкової точки та прибуття до кінцевої;

– додаткові умови, що визначають рівень зручності.

Сортування здійснюється або через розрахунок зведеної оцінки, або шляхом послідовної організації за кількома ключовими параметрами. Такий підхід дає можливість міняти послідовність видачі результатів залежно від того, які саме аспекти вважаються найбільш важливими у конкретному випадку пошуку [8].

Важливою рисою систем, що пропонують рекомендації, є здатність підтримувати індивідуалізований вибір. При роботі авіаційних сервісів персоналізація не зводиться лише до фіксації збережених вподобань користувача, а реалізується через аналіз його попередніх дій. Серед таких даних можуть бути історія переглянутих варіантів, часто повторювані напрямки, характерні часові рамки для бронювання, а також реакція клієнта на ті пропозиції, що були йому подані. На основі цієї інформації алгоритми рекомендацій налаштовують порядок видачі, переглядаючи вагові коефіцієнти критеріїв або посилюючи значущість певних груп рейсів.

Персоналізований метод дає можливість системі зменшити надмірність інформації, оскільки клієнту демонструються ті варіанти, що з вищою ймовірністю відповідають його типовим потребам у подорожах. У

дослідженнях, присвячених рекомендаційним системам, наголошується, що навіть мінімальна персоналізація значно покращує якість вибору, коли коло альтернатив є дуже широким [8].

Окреме місце у рекомендаційних механізмах посідає інтеграція різних критеріїв для вибору. У задачах, пов’язаних із замовленням авіаквитків, практично неможливо досягти ідеалу за всіма параметрами одночасно, адже рейси, які коштують менше, часто є тривалішими або вимагають пересадок, тоді як найшвидші подорожі асоціюються з вищою платою. Для врахування таких протиріч використовуються методи, що дають змогу:

– звести кілька критеріїв в одну загальну функцію оцінки;

– створювати набори альтернатив, які не мають явної переваги одна над одною;

– впроваджувати моделі рекомендацій, керуючись певними обмеженнями, встановленими правилами бізнесу.

1.3.3 Гібридний підхід: поєднання правил і рекомендацій

У сфері автоматичного підбору авіарейсів є сенс у впровадженні комбінованого методу, який інтегрує механізми на основі правил та алгоритми формування рекомендацій. Така структура дає змогу об’єднати зафіксовані бізнес-вимоги з гнучкими методами упорядкування, сфокусованими на індивідуальних вподобаннях користувача. На відміну від систем, що ґрунтуються лише на рекомендаціях, гібридна будова забезпечує нагляд за допустимістю прийнятих рішень, а порівняно з повністю регламентованими підходами – покращує гнучкість, пристосованість та релевантність отриманих варіантів.

У рамках гібридної структури регламентні норми та алгоритми рекомендацій виконують ролі, що доповнюють одна одну. Модуль, що базується на правилах, здебільшого залучається на початкових етапах опрацювання запиту та відповідає за суворе відсіювання доступних опцій.

До категорії таких правил відносяться: перевірка часових рамок, допустимість маршрутів, обмеження на кількість пересадок, а також відповідність вимогам щодо тарифу або візового режиму. Після первинної фільтрації до роботи підключається блок рекомендацій, який виконує такі функції:

– здійснює ранжування дозволених до розгляду рейсів;

– бере до уваги вагомість критеріїв (вартість, тривалість подорожі, рівень комфорту);

– персоналізує порядок результатів відповідно до профілю клієнта.

Каскадний, алгоритм роботи сприяє зниженню обчислювальних навантажень для рекомендаційної частини та водночас запобігає появі в остаточному переліку рішень, що є невідповідними чи неприйнятними.

Об’єднання логічних правил та алгоритмів рекомендацій надає низку відчутних переваг, особливо значущих у площині авіаційних бронювань. Насамперед, гібридна тактика дає змогу чітко кодифікувати регуляторні та юридичні обмеження, не делегуючи їх обробку виключно статистичним або евристичним моделям. Серед ключових переваг такого поєднання виділяють:

– розмежування комерційної логіки від інструментів індивідуалізації;

– скорочення кількості рекомендацій, що не відповідають потребам;

– стабільність результативності у складних ситуаціях;

– можливість незалежного налаштування як правил, так і методів упорядкування.

Важливою рисою гібридного механізму є здатність контролювати якість наданих рекомендацій. Складові, засновані на правилах, надають можливість чітко встановити причини: чому той чи інший рейс було або не було внесено до результатів пошуку. Це є критично важливим у разі оскарження чи проведення внутрішнього аудиту. Контроль якості забезпечується через:

– простежування застосованих регламентів;

– верифікацію збігу рекомендацій із встановленими лімітами;

– аналіз впливу певних параметрів на кінцеве ранжування.

Завдяки наявності логічних правил система зберігає можливість пояснюваності, яка часто втрачається у моделях, що повністю ґрунтуються на статистичних даних. Це має особливе значення для авіаційних платформ, де помилкові чи незрозумілі рішення можуть мати фінансові або правові наслідки. Гібридний підхід дає змогу поєднати інтерпретованість правил із адаптивністю рекомендаційних алгоритмів, не приносячи в жертву стійкість функціонування системи.

1.4 Переваги впровадження ІС бронювання

Впровадження інтелектуальної системи бронювання авіаквитків на основі методів штучного інтелекту забезпечує суттєві переваги насамперед для кінцевого користувача. У традиційних пошукових системах користувач змушений самостійно опрацьовувати десятки або сотні варіантів, порівнюючи вартість квитків, тривалість перельоту, кількість та тривалість пересадок.

Розумна система перекладає більшу частину цього мисленнєвого навантаження на програмний рівень, пропонуючи вже впорядкований та дійсно необхідний набір рейсів. Користувач одержує не просто перелік доступного, а добірки, скомпоновані з огляду на його запит, попередні вподобання або обраний сценарій подорожі, що значно пришвидшує ухвалення рішення та мінімізує ризик помилитися з вибором.

Стосовно сервісу-посередника, інтелектуальний механізм бронювання слугує інструментом для покращення ефективності контакту з клієнтом та усунення недоліків у внутрішніх робочих процесах. Завдяки застосуванню правил і алгоритмів, що дають поради, посередник спроможний точніше керувати тим, які результати показувати, зменшуючи кількість невідповідних пошуків і повторних звернень. Це позитивно

позначається на показниках залучення, здійснення покупок і лояльності клієнтів. Додатково, система дозволяє уніфікувати управління логікою розташування та сортування рейсів, не спираючись на заздалегідь жорстко прописані схеми, що спрощує коригування бізнес-правил та припасування послуги до мінливих умов ринкової кон'юнктури [9].

З перспективи авіаперевізників, системи бронювання на базі ШІ виступають каналом для більш сфокусованої презентації їхніх пропозицій. Алгоритмічне сортування дає змогу демонструвати польоти тим клієнтам, для яких вони дійсно можуть бути цікавими, враховуючи фінансові обмеження, часові рамки чи необхідний маршрут, що зменшує частку випадкових показів і підвищує шанс придбання певних сегментів квитків.

Крім того, зведені відомості про вибір користувачів можуть бути використані для аналізу попиту, планування мережі сполучень та корекції цінової стратегії, що відповідає сучасним парадигмам керування в галузі авіасполучення, базованого на даних [1].

Інтелектуальна система бронювання забезпечує обробку великих обсягів інформації без пропорційного зростання витрат завдяки автоматичному фільтруванню, сортуванню та формуванню рекомендацій. Це дозволяє підтримувати стабільну роботу при збільшенні кількості користувачів, рейсів і підключених авіаліній, а масштабування системи здійснюється шляхом додавання нових джерел даних.

2 ПОСТАНОВКА ЗАДАЧІ ТА ПРОЄКТУВАННЯ СИСТЕМИ

2.1 Мета та завдання кваліфікаційної роботи

Інтенсивний розвиток цифрових послуг у секторі авіаперевезень зумовлює зростання кількості доступних рейсів, розширення тарифних планів та маршрутних комбінацій, що ускладнює для кінцевого користувача процес вибору оптимального варіанту. Сучасні системи бронювання та платформи-агрегатори авіаквитків здатні опрацьовувати значні обсяги даних із різних джерел, однак значна частина аналітичної роботи з їх інтерпретації залишається на боці користувача.

За подібних обставин зростає когнітивне навантаження, подовжується часовий інтервал для ухвалення остаточного рішення, а ризик обрання неоптимального маршруту підвищується. Саме тут набувають особливої ваги інтелектуальні системи автоматизованого типу, які можуть не тільки збирати інформацію про рейси, але й виконувати функції аналізу, селекції, зіставлення та систематизації (ранжування) варіантів для бронювання.

Головною метою кваліфікаційної роботи є розробка інтелектуальної автоматизованої системи бронювання авіаквитків, спрямованої на підтримку прийняття рішень користувачем в умовах багатокритеріального вибору. Розроблювана система має забезпечити автоматичне виділення та впорядкування доступних авіарейсів із застосуванням підходів, заснованих на правилах (rule-based), та рекомендаційних алгоритмів, що дозволяє мінімізувати рутинні операції з опрацювання результатів пошуку.

Йдеться не лише про технічну реалізацію механізмів бронювання, а про формування інтелектуального рівня логіки, що бере участь у прийнятті рішень на етапах фільтрації, оцінки та порівняння альтернатив.

Центральною темою роботи є автоматизоване прийняття рішень у межах інтелектуальної системи бронювання авіаквитків. Автоматичний процес прийняття рішень розглядається як послідовність дій, у якій система

самостійно застосовує визначені обмеження, логічні правила та критерії оптимізації з метою виключення неприйнятних маршрутів і формування впорядкованого переліку допустимих варіантів. Такий підхід дозволяє мінімізувати залучення користувача до рутинних процедур і зосередити його увагу на виборі остаточного рішення серед обмеженого та релевантного набору пропозицій. Поєднання логіки, заснованої на правилах, із рекомендаційними механізмами забезпечує баланс між формалізованим контролем і гнучкістю системи.

Окремим завданням роботи є проєктування масштабованої та гнучкої архітектури системи бронювання. Розроблювана інтелектуальна система має функціонувати в умовах зростання кількості користувачів, рейсів та підключених джерел даних без необхідності кардинальної зміни її внутрішньої структури. Адаптивність системи полягає у можливості модифікації правил відбору, параметрів ранжування та моделей взаємодії без повної реорганізації архітектури, що є критично важливим для агрегаторів, що функціонують у динамічному ринковому середовищі.

Система, що розробляється, позиціонується як агрегатор, здатний інтегрувати інформацію від різних авіаліній або зовнішніх платформ та забезпечувати єдину логіку для опрацювання запитів. Інтеграційний потенціал системи передбачає взаємодію з базою даних рейсів, модулями, що керують бронюванням, а також із зовнішніми сервісами, зокрема платіжними шлюзами та API постачальників авіаційних даних. Це закладає основи для створення цілісного цифрового простору для бронювання, у центрі якого інтелектуальна система виступатиме ключовим елементом у процесі ухвалення рішень.

Для досягнення поставленої цілі в рамках цього практичного дослідження планується виконати низку взаємопов’язаних завдань, що охоплюють аналітичний етап, етап проєктування та експериментальну фазу. На початковому етапі буде проведено аналіз сфери авіаційного бронювання, включаючи вивчення типових сценаріїв використання, існуючих методів

пошуку та шляхів автоматизації вибору рейсів. Особлива увага буде приділена дослідженню чинників, що впливають на процес прийняття рішень користувачем, та можливостей їхньої формалізації у вигляді чітких правил і критеріїв.

Наступний етап полягає у проєктуванні архітектури інтелектуальної системи бронювання із чітким визначенням ключових функціональних блоків, таких як інтерфейс користувача (клієнтська частина), серверна бізнес-логіка, модуль на основі правил, модуль рекомендацій та рівень зберігання даних. На цьому етапі буде визначено порядок обробки запитів, налагоджено взаємодію між окремими компонентами та точно визначено місце автоматизованого прийняття рішень у загальному робочому циклі системи.

Подальше завдання включає практичну реалізацію спроєктованої архітектури з використанням обраних засобів програмування та систем для зберігання інформації. Реалізація передбачає створення робочих механізмів фільтрування та присвоєння пріоритетів рейсам, інтегрування з базою даних, а також моделювання типових ситуацій, що виникають під час бронювання. Окрема увага буде приділена забезпеченню коректності обробки даних та стабільності роботи системи в умовах різного рівня навантаження.

Завершальною стадією практичного дослідження є тестування розробленої інтелектуальної системи бронювання. Під час тестування буде здійснено перевірку функціональної правильності впроваджених алгоритмів, оцінку дієвості автоматизованого процесу прийняття рішень та аналіз поведінки системи у стандартних користувацьких сценаріях. Отримані результати дозволять оцінити доцільність застосування інтелектуального підходу до бронювання авіаквитків загалом і розглядати цю систему як складовий елемент ширшого цифрового середовища для управління авіаційними послугами.

2.2 Архітектура системи, структурна специфікація функціональних модулів системи

Формальне визначення функціональних складових інтелектуальної системи бронювання авіаквитків є необхідним кроком для переходу від концептуального опису до практичної реалізації. Декомпозиція системи на відокремлені функціональні блоки дозволяє чітко розподілити зони відповідальності, визначити послідовність обробки даних та точки прийняття автоматизованих рішень. Такий підхід забезпечує модульність архітектурного рішення, спрощує подальше розширення системи та уможливлює незалежний розвиток окремих компонентів без порушення цілісності загальної структури.

Побудова архітектури інтелектуальної системи, призначеної для купівлі авіаквитків, відбувається у форматі багатошарового програмного рішення. У цій моделі кожен шар володіє набором чітко окреслених завдань і є інкапсульованим від інших щодо зони своєї відповідальності. Такий підхід дає змогу одночасно опрацьовувати значний потік запитів від користувачів, впроваджувати розумні методики вибору авіарейсів та забезпечувати надійне сполучення із зовнішніми інформаційними джерелами. Поділ обов’язків між цими шарами також спрощує процес розширення функціоналу системи та її подальше масштабування.

Компонент, орієнтований на клієнта, відповідає за пряму взаємодію кінцевого користувача з сервісом бронювання. Тут реалізується інтерфейс для введення критеріїв пошуку, демонстрація отриманих результатів, представлення рекомендованих опцій та ініціація процесу резервування.

Фронт-енд частина не займається складними обчисленнями; її завдання – передати сформований запит на серверне ядро й представити повернені дані у формі, максимально зручній для сприйняття користувачем. Значна увага акцентується на гнучкості інтерфейсу та методах візуалізації

результатів, адже саме цей шар формує уявлення клієнта про рівень інтелектуальності застосованих рішень.

Серверний рівень втілює ключову бізнес-логіку системи та є центральним вузлом для інтелектуального опрацювання звернень. Тут відбувається збір та первинна стандартизація даних про доступні рейси, застосування фільтраційних критеріїв, а також активізація рекомендаційних та аналітичних алгоритмів. Компоненти, засновані на визначених правилах (rule-based), дозволяють формалізувати обмеження та пріоритети, як-от усунення рейсів із надмірною кількістю пересадок чи урахування часових інтервалів. Паралельно будуть задіяні моделі машинного навчання (ML) для ранжування доступних варіантів, зокрема ті, що базуються на градієнтному бустингу або логістичній регресії, які оцінюють ймовірність вибору конкретного маршруту користувачем. Зведення детермінованих правил і статистичних моделей забезпечує баланс між повною прозорістю прийнятих рішень та адаптивністю всієї системи.

Окремий підрозділ серверної логіки сфокусований на аналізі та обробці даних про поведінку користувачів. На цьому етапі система досліджує історію пошукових запитів, параметри попередніх резервувань та типові патерни взаємодії користувача з сервісом. Ця інформація може слугувати для тренування алгоритмів ранжування, формування індивідуальних пропозицій або коригування значущості критеріїв у системі багатофакторного аналізу. Важливо, що алгоритми машинного навчання в даній структурі не функціонують ізольовано, а тісно інтегровані з бізнес-правилами, що гарантує передбачуваність функціонування системи.

Рівень роботи з даними відповідає за надійне зберігання та забезпечення доступу до інформаційних активів, необхідних для коректної роботи системи бронювання. У рамках цього проекту оптимальним вибором є використання реляційної СУБД PostgreSQL, яка підтримує складні SQL-запити, гарантує транзакційність та має вбудовані механізми для оптимізації доступу до інформації. Тут зберігаються зведені дані про маршрути,

перевізників, аеропорти, вартість квитків, а також профілі клієнтів і записи їхніх сесій взаємодії. Структура бази даних проектується з огляду на потребу у високій швидкості вибіркового звернення до значних обсягів даних і можливості виконання аналітичних запитів (рисунок 2.1).

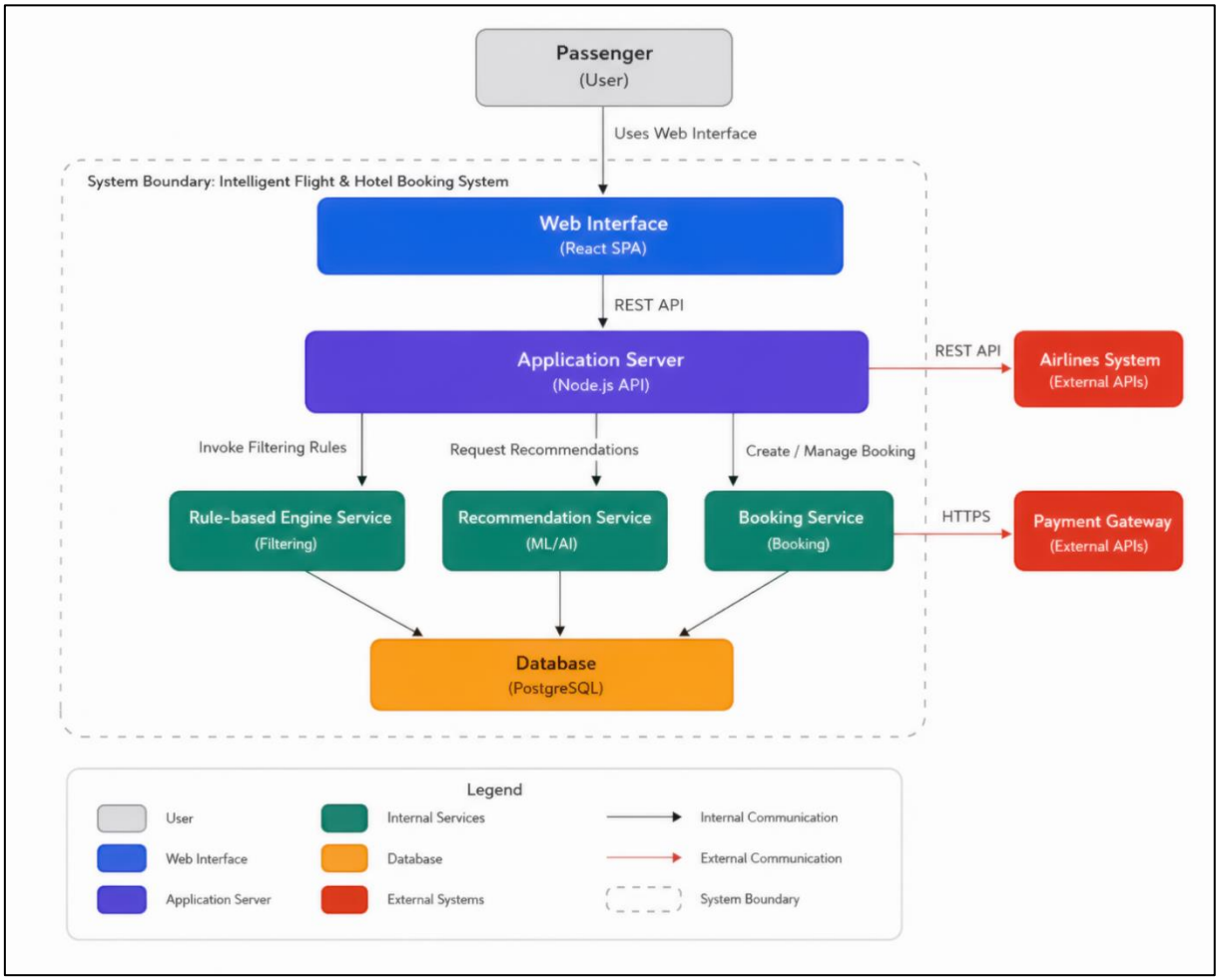


Рисунок 2.1 – Діаграма контейнерів інтелектуальної системи бронювання авіаквитків

Для наочного відображення внутрішньої структури системи та взаємодії між її компонентами використано діаграму контейнерів (рисунок 2.1). Діаграма відображає сім основних контейнерів системи в межах системної границі: клієнтський інтерфейс (Web Interface), сервер застосунку (Application Server), модуль фільтрації на основі правил (Rule-based Engine), рекомендаційний модуль (Recommendation

Engine), модуль бронювання (Booking Module) та базу даних (Database). Зовнішніми системами, з якими взаємодіє платформа, є агрегатори авіаційних даних (Airline APIs) та платіжний шлюз (Payment Gateway).

Ключовим компонентом системи є блок збору та зберігання даних про рейси. Його основне завдання полягає в акумуляції даних, отриманих із зовнішніх джерел, та приведенні їх до єдиного внутрішнього формату.

У межах цього блоку відбувається фіксація даних щодо маршрутів, часових параметрів, вартості, кількості транзитів та додаткових обмежень, що можуть впливати на процес резервування. Централізоване зберігання відомостей у реляційній базі даних гарантує їхню консистентність, можливість виконання складних запитів та подальшу обробку іншими ланцюжками системи. Блок збору даних не виносить самостійних рішень, однак слугує фундаментом для коректного функціонування усіх інтелектуальних механізмів.

Далі йде критично важливий блок, заснований на правилах (rule-based логіці), який відповідає за формалізоване опрацювання умов та встановлених критеріїв для вибору рейсів. Тут імплементуються логічні верифікації, що визначають, чи є певний маршрут прийнятним чи неприйнятним у заданих обставинах.

До таких верифікацій відносяться часові рамки, гранична кількість зупинок, відповідність узгодженому шляху чи інші умови, які мають директивний характер і не підлягають ігноруванню. Блок rule-based виконує функцію первинної фільтрації, виключаючи варіанти, які не відповідають визначеним вимогам, ще до застосування алгоритмів надання рекомендацій, що забезпечує стабільність поведінки системи та прозорість здійснених висновків.

Рекомендаційний блок втілює інтелектуальний рівень системи, сфокусований на роботі з відносними перевагами та багатокритеріальним відбором. Його задачею є ієрархічне впорядкування допустимих рейсів,

визначених після роботи rule-based логіки, з огляду на критерії оптимізації та сценарії взаємодії з користувачем.

На цьому етапі система не усуває варіанти категорично, а формує впорядкований список альтернатив, що різняться між собою вартістю, загальною тривалістю подорожі чи рівнем комфорту. Рекомендаційний блок може модифікувати свою роботу залежно від встановлених параметрів пошуку або історії взаємодій користувача з платформою, що сприяє зростанню релевантності результатів та зменшенню інформаційного навантаження.

Блок резервування виконує функцію переходу від аналітичного вибору до практичного ухвалення рішення. Його призначення полягає у фіксації обраного варіанту, перевірці актуальності відомостей та формуванні запиту на бронювання. На цьому етапі система вже не займається оптимізацією чи сортуванням, а забезпечує правильність операційної логіки, пов’язаної із закріпленням місць та збереженням параметрів замовлення. Блок резервування контактує з базою даних та зовнішніми сервісами, підтверджуючи узгодженість між внутрішнім станом системи та фактичним статусом рейсу.

Платіжний блок, у контексті даної роботи, розглядається як допоміжна складова, яка не належить до суті інтелектуальної логіки, проте є необхідною для завершення послідовності дій користувача. Його функція полягає в опрацюванні фінансових транзакцій та інтеграції із зовнішніми платіжними системами. Платіжний блок відокремлено від механізмів прийняття рішень і активується після того, як процес вибору та резервування було завершено. Поділ дозволяє чітко відділити фінансові операції від інтелектуальних складових системи та знизити складність їхньої реалізації.

Зовнішня комунікація з іншими ресурсами реалізується як самостійний функціональний шар, що забезпечує обмін даними з агрегаторами авіаперевезень, програмними інтерфейсами (API)

авіакомпаній та платіжними шлюзами. Через ці канали система отримує актуальні відомості про наявність місць, ціноутворення та умови тарифів, а також передає необхідні дані для здійснення фінансових операцій та фіксації бронювання.

Інтеграційний рівень мусить брати до уваги асинхронний характер комунікації із зовнішніми системами, можливі затримки у відповідях та потенційні збої у роботі сторонніх сервісів. Для мінімізації ризиків застосовуються механізми буферизації даних, автоматичного повтору запитів і детального логування, що підтримує стійкість роботи системи навіть у разі непередбачуваної поведінки зовнішніх джерел.

Завдяки такій побудові, інтелектуальні алгоритми, встановлені бізнес-принципи й зовнішні джерела даних об’єднуються в єдиний технічний контур, де кожен рівень виконує свою специфічну роль без надмірного дублювання функціоналу. Це формує міцну технічну основу для реалізації складних сценаріїв вибору та оформлення авіаквитків, використовуючи комбінацію методів машинного навчання, формалізованих правил і інтеграції з реальними авіаційними та фінансовими екосистемами.

Сукупність описаних функціональних складових формує логічно згуртовану структуру інтелектуальної системи бронювання авіаквитків, де кожна частина виконує чітко визначене завдання. Взаємодія між блоками побудована так, що автоматичне формування висновків сконцентровано в rule-based та рекомендаційному компонентах, тоді як інші блоки забезпечують підтримку, комунікацію та фіналізацію сценаріїв користувача. Така формалізація закладає основу для подальшого впровадження системи та емпіричної перевірки її ефективності.

2.3 Типи користувачів та сценарії використання системи

Робота системи інтелектуального планування авіаперельотів вимагає взаємодії з різними групами користувачів, кожна з яких має свої завдання,

рівні доступу й типові сценарії застосування. Правильне розмежування цих ролей є обов'язковою умовою для забезпечення стабільності системи, захисту інформації та узгодженості бізнес-правил.

Для визначення функціональних вимог до системи та відображення взаємодії користувачів із її компонентами використано діаграму варіантів використання у нотації UML (рисунок 2.2). Діаграма визначає двох акторів: пасажира та адміністратора. Пасажир взаємодіє з системою через п'ять сценаріїв — пошук рейсів, перегляд рекомендацій, вибір рейсу, бронювання квитка та оплату, між якими встановлено зв'язки типу «include», що відображають обов'язкову послідовність виконання. Адміністратор має доступ до чотирьох сценаріїв управління: керування правилами фільтрації, керування даними рейсів, моніторинг системи та налаштування алгоритмів рекомендацій.

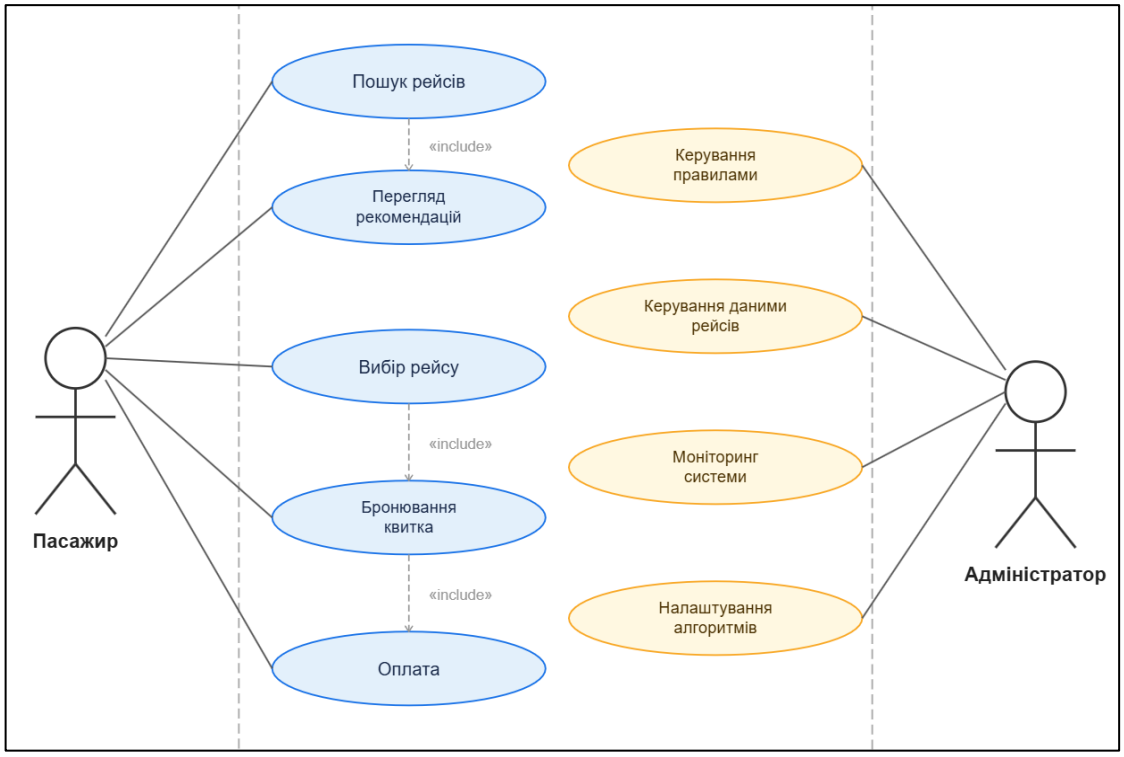


Рисунок 2.2 – Діаграма варіантів використання інтелектуальної системи бронювання авіаквитків (UML Use Case Diagram)

У контексті цього дослідження виділяють дві головні групи – це кінцевий споживач послуг, тобто сам пасажир, та адміністратор, який відповідає за технічне обслуговування й управління функціоналом сервісу.

Головним суб'єктом взаємодії з інтелектуальним механізмом бронювання є пасажир, що використовує його для пошуку, вибору та резервування місць на рейсах. Для цієї аудиторії система слугує інструментом підтримки ухвалення рішень, спрощуючи вибір і автоматизуючи більшу частину необхідних аналітичних розрахунків.

Пасажир не має доступу до внутрішньої архітектури програми, але він безпосередньо споживає результати роботи модулів, заснованих на правилах та рекомендаціях, у вигляді структурованих пропозицій. Під час сесії користувач вказує критерії пошуку, отримує пропоновані варіанти й приймає остаточне рішення про купівлю.

Послідовність дій користувача-пасажира стартує з етапу пошукового запиту, де клієнт фіксує ключові аспекти майбутньої подорожі: місце відправлення, кінцевий пункт, дати вильоту та будь-які додаткові обмеження. На цьому етапі система запускає обробку запиту, звертаючись до бази даних рейсів, після чого активується логіка, заснована на правилах (rule-based), для попереднього відбору. Ті рейси, що не відповідають заданим параметрам, виключаються до того, як почнеться етап рекомендацій, що суттєво звужує обсяг аналізу та підвищує якість фінальних пропозицій.

Процес взаємодії користувача з системою бронювання формалізовано у вигляді діаграми активності у нотації UML (рисунок 2.3). Діаграма побудована з використанням трьох доріжок (swimlanes): пасажир, система та зовнішні сервіси, що дозволяє чітко розмежувати зони відповідальності кожного учасника процесу. Процес розпочинається з введення пасажиром критеріїв пошуку, після чого система формує запит до бази даних рейсів та отримує актуальні дані через Airline API.

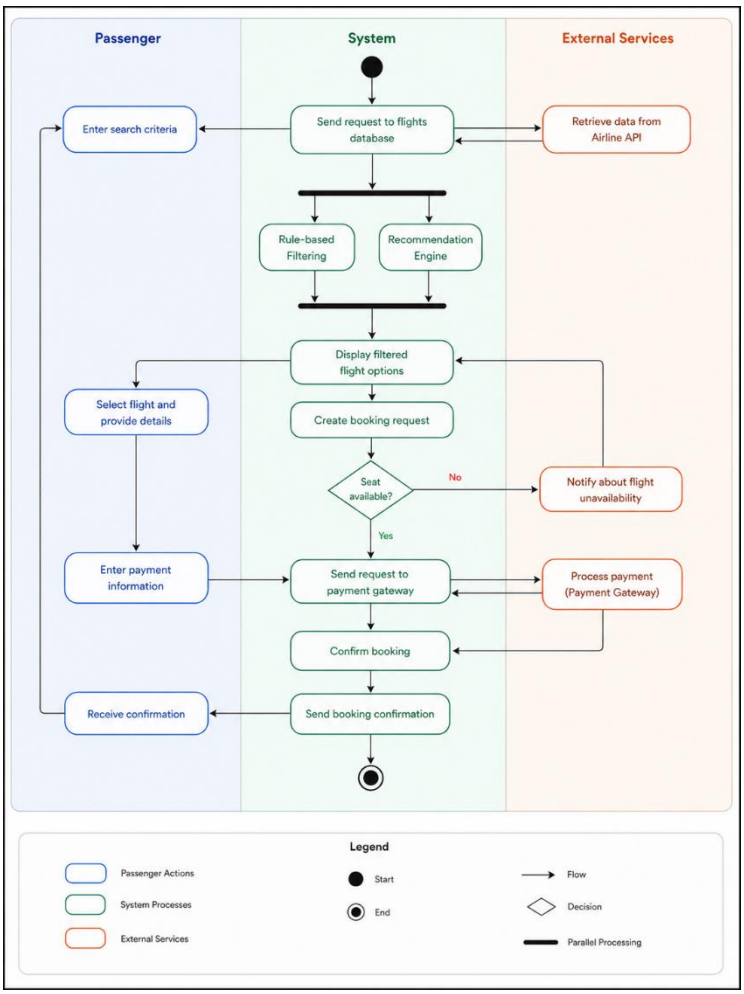


Рисунок 2.3 – Діаграма активності процесу бронювання авіаквитка (UML Activity Diagram)

Після завершення первинного фільтрування система переходить до фази формування рекомендацій, де задіюється відповідний модуль. У межах цієї стадії здійснюється ранжування доступних альтернатив згідно з низкою факторів, як-от вартість квитка, загальна тривалість перельоту чи кількість необхідних стикувань. Користувачеві надається вже впорядкований перелік варіантів, що спрощує порівняння та фокусування на найбільш вигідних із них. Таким чином, система функціонує як інтелектуальний посередник між величезним обсягом інформації та споживачем, мінімізуючи рутинну роботу з аналізу.

Наступний крок у сценарії взаємодії – це власне бронювання, яке ініціюється після того, як пасажир обрав бажану комбінацію. На цій стадії система закріплює деталі обраного маршруту, перевіряє їхню актуальність і генерує вимогу на резервування. Модуль резервування гарантує відповідність вибору клієнта поточному стану системи, запобігаючи будь-яким накладенням чи подвоєним операціям. В рамках цього дослідження процес бронювання трактується як логічне завершення етапу вибору, а не як окремий аналітичний етап.

Завершальною дією у сценарії користувача є здійснення оплати, яке реалізується через спеціалізований платіжний сервіс. Фінансова транзакція не має впливу на інтелектуальні алгоритми вибору напрямку, але є вирішальною для фіналізації циклу використання системи. Взаємодія з платіжним компонентом відбувається після підтвердження резервування й забезпечує коректне проведення грошових операцій без втручання у принципи роботи рекомендаційних чи фільтраційних механізмів.

Окрему категорію становлять адміністратори системи, відповідальні за її технічне забезпечення, налагодження й контроль правильності функціонування. На відміну від пасажира, адміністратор має доступ до внутрішніх функціональних блоків і може змінювати роботу системи.

До обов’язків адміністратора входить управління правилами відбору рейсів, оновлення параметрів алгоритмів пропонування варіантів, контроль джерел даних та моніторинг сценаріїв використання. Хоча адміністратор не бере участі у процесі бронювання як покупець, його робота є запорукою надійності та передбачуваності роботи сервісу для всіх пасажирів.

2.4 Результати впровадження та практичне значення системи

Очікувані наслідки впровадження системи штучного інтелекту для резервування авіаквитків насамперед стосуються підвищення якості етапу вибору польотів у складній ситуації, де треба зважувати безліч різних

критеріїв. Завдяки об’єднанню логіки, заснованої на правилах, з алгоритмами формування рекомендацій, система здатна автоматично обробляти великі обсяги відомостей та видавати впорядковані пропозиції, які відповідають заданим умовам та типовим сценаріям використання. Це трансформує роль споживача з активного аналітика на учасника фінальної стадії ухвалення рішення, що сприятливо позначається на загальному досвіді взаємодії з послугою.

Одним із головних передбачуваних результатів є збільшення ефективності процесу вибору авіарейсів. У звичайних системах продуктивність вибору значною мірою визначається спроможністю клієнта самостійно аналізувати та зіставляти варіанти. Розумна система переносить це завдання на рівень алгоритмів, забезпечуючи:

– вилучення маршрутів, які не відповідають критеріям;

– сортування рейсів за ступенем їхньої значущості;

– мінімізацію числа нерелевантних результатів у виводі.

Такий підхід дає змогу споживачеві сконцентруватися на кількох найбільш доцільних варіантах, замість того, щоб продивлятися увесь масив доступних пропозицій.

Важливим практичним наслідком є скорочення часу, необхідного для прийняття рішення. Методом поетапної автоматизації послідовності «пошук → порада → резервування» система зменшує термін, потрібний для планування поїздки. Користувач одразу отримує систематизований результат, що знижує потребу в багаторазових пошуках та ручному зіставленні параметрів. Це особливо цінно в умовах, коли ціни постійно змінюються, адже швидкість прийняття рішення може напряму впливати на остаточну вартість квитка.

Щодо організації роботи самого сервісу, інтелектуальна система забезпечує автоматизацію основних кроків, пов’язаних з опрацюванням запитів від користувачів. До таких кроків відносяться:

– первинне відсіювання рейсів відповідно до формалізованих приписів;

– автоматизоване визначення пріоритетності альтернатив;

– узгодження даних на етапі бронювання;

– зменшення обсягу ручних операцій з боку платформи-агрегатора.

Автоматизація сприяє зниженню навантаження на системи підтримки та гарантує стабільну роботу системи навіть при збільшенні кількості клієнтів та їхніх запитів.

Практична цінність даної розробки також полягає у можливості її подальшого розширення та модернізації. Модульна будова та чітке розділення функціональних частин створюють сприятливі умови для:

– підключення нових джерел інформації про рейси;

– розширення набору правил та критеріїв відбору;

– удосконалення механізмів формування рекомендацій;

– інтегрування з додатковими сервісами без порушення базової логіки функціонування системи.

Таким чином, у другому розділі обґрунтовано концептуальні засади побудови інтелектуальної системи бронювання авіаквитків, визначено її функціональні можливості та практичну цінність. Наступний розділ присвячено безпосередньому проєктуванню та реалізації системи.

3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ БРОНЮВАННЯ

3.1 Загальна архітектура програмної системи

3.1.1 Архітектура клієнт-серверного застосунку

Для реалізації інтелектуальної системи бронювання авіаквитків було обрано клієнт-серверну архітектуру, яка забезпечує розділення інтерфейсу користувача, серверної логіки та рівня зберігання даних. Такий підхід дозволяє створити масштабовану та гнучку систему, у якій фронтенд та бекенд можуть розвиватися незалежно один від одного.

Вибір клієнт-серверної моделі обумовлений необхідністю централізованої обробки пошукових запитів, роботи з базою даних рейсів, виконання алгоритмів персоналізованого ранжування та підтримки взаємодії між користувачем і сервером через REST API. Крім того, така архітектура дозволяє підключати різні клієнтські застосунки до одного серверного ядра без зміни бізнес-логіки системи.

У розробленому застосунку клієнтська частина реалізована за допомогою бібліотеки React та мови TypeScript. Вона відповідає за відображення інтерфейсу, введення параметрів пошуку, налаштування користувацьких уподобань та взаємодію із сервером через HTTP-запити. Використання React дозволило реалізувати компонентний підхід до побудови інтерфейсу, що спрощує підтримку та повторне використання окремих елементів системи.

Серверна частина побудована на основі фреймворку FastAPI мовою Python. Бекенд забезпечує обробку запитів, логіку пошуку рейсів, побудову маршрутів з пересадками, збереження бронювань, а також персоналізацію результатів пошуку відповідно до вподобань користувача. Для організації структури проєкту було використано модульний підхід із розділенням на маршрути API, моделі бази даних, репозиторії та ETL-компоненти.

Для взаємодії з базою даних використовується ORM SQLAlchemy. Це дозволяє працювати з таблицями бази даних через Python-моделі без написання великої кількості SQL-коду вручну. Як систему управління базами даних обрано PostgreSQL, оскільки вона підтримує складні SQL-запити, ефективну роботу з великими обсягами даних та забезпечує високу продуктивність при побудові маршрутів із пересадками.

Загальна структура системи складається з трьох основних рівнів (взаємодія висвітлена на рисунку 3.1):

– клієнтський рівень (Frontend);

– серверний рівень (Backend API);

– рівень зберігання даних (Database).

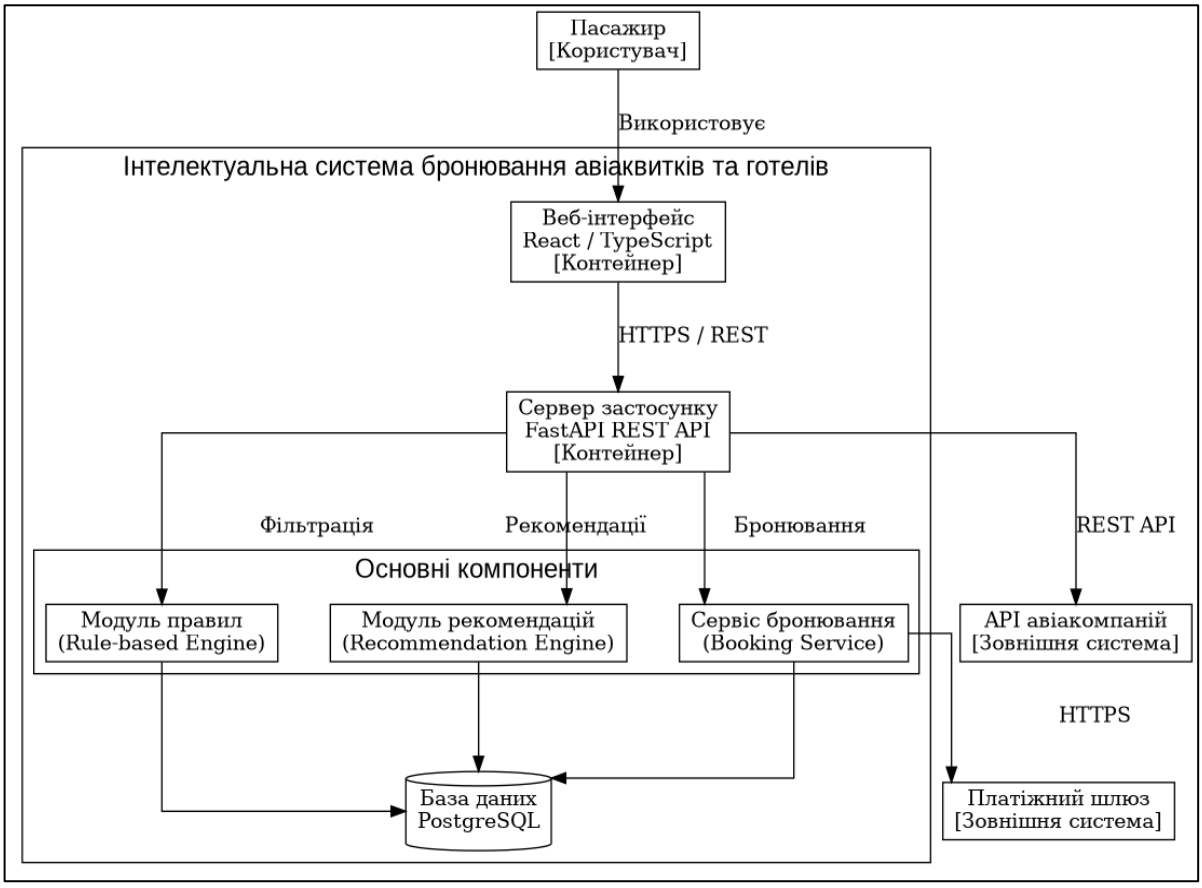


Рисунок 3.1 – Схема побудови та взаємодії компонентів системи

Клієнтський застосунок надсилає HTTP-запити до REST API сервера. Сервер обробляє отримані дані, виконує необхідні алгоритми пошуку та ранжування, після чого звертається до бази даних PostgreSQL для отримання або збереження інформації. Результати повертаються у форматі JSON та відображаються у вебінтерфейсі.

Особливістю реалізованої архітектури є використання ETL-модуля для автоматичного імпорту та генерації рейсів. Система імпортує дані з відкритих авіаційних наборів даних OpenFlights та OurAirports, після чого автоматично генерує екземпляри рейсів із реалістичними датами, часом відправлення, тривалістю польоту та вартістю квитків. Для оптимізації швидкодії було реалізовано bulk insert записів у базу даних.

У серверній частині також реалізовано механізм побудови маршрутів із пересадками. Алгоритм аналізує можливі стикування рейсів за часом прибуття та відправлення, враховуючи мінімальний час пересадки між рейсами. Це дозволяє системі формувати складні авіамаршрути та оцінювати їх за критеріями вартості, тривалості та кількості пересадок.

Взаємодія компонентів системи відбувається за схемою, зображеною на рисунку 3.2.

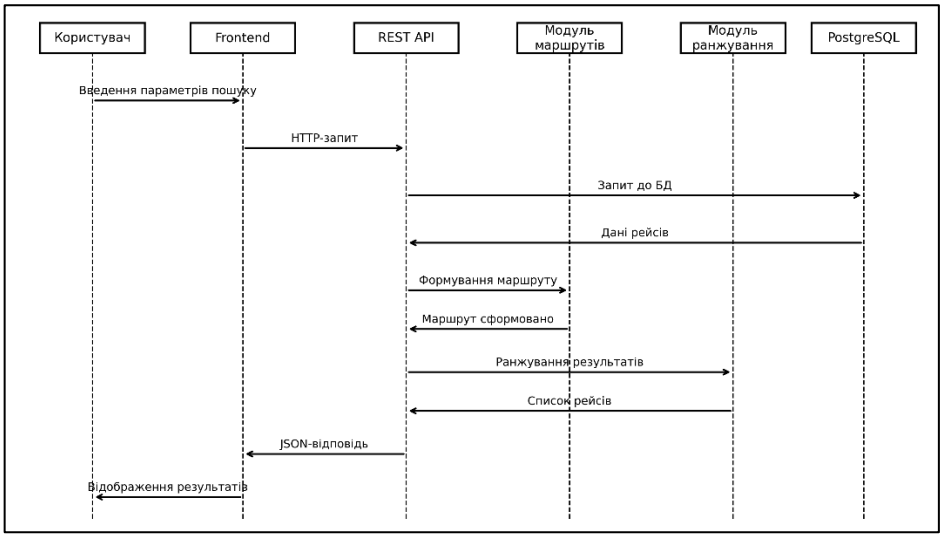


Рисунок 3.2 – Загальний сценарій роботи системи бронювання

Розроблена архітектура забезпечує модульність, масштабованість та можливість подальшого розширення функціоналу системи, зокрема інтеграції машинного навчання, зовнішніх API авіакомпаній та багатокористувацької авторизації.

3.1.2 Архітектура серверної частини

Серверна частина інтелектуальної системи бронювання авіаквитків реалізована з використанням фреймворку FastAPI мовою Python. Вибір цього фреймворку обумовлений його підтримкою асинхронної обробки запитів, автоматичною генерацією інтерактивної документації на основі стандарту OpenAPI, а також строгою типізацією вхідних та вихідних даних через бібліотеку Pydantic, що суттєво знижує ймовірність помилок при передачі даних між шарами системи.

Структура бекенд-проєкту організована за принципом розділення відповідальностей, де кожен каталог відповідає за чітко визначену функціональну роль (рисунок 3.3).

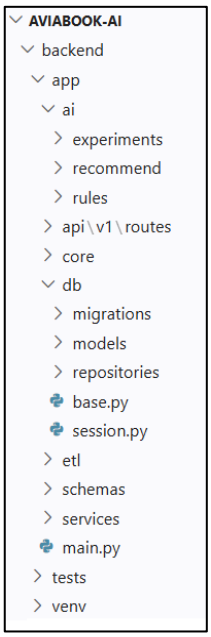


Рисунок 3.3 – Структура проєкту реалізації backend

Верхній рівень представлений модулем main.py, у якому відбувається ініціалізація застосунку FastAPI, підключення CORS-middleware, реєстрація маршрутів та запуск lifespan-контексту, що відповідає за підготовку бази даних і виконання ETL-модуля при старті системи. Налаштування середовища винесені до окремого модуля core/config.py, де реалізовано клас Settings на основі pydantic_settings.BaseSettings, що дозволяє отримувати конфігураційні параметри як з файлу .env, так і зі змінних середовища операційної системи.

Маршрути API згруповані у каталозі api/v1/routes/ та розподілені за функціональними доменами: search.py – пошук рейсів, bookings.py – управління бронюваннями, users.py – реєстрація та налаштування уподобань користувача, admin.py – адміністративні операції, health.py – перевірка стану сервісу. Кожен роутер реєструється в main.py з відповідним префіксом /api/v1 та тегом для групування в документації Swagger.

Бізнес-логіка системи зосереджена у каталозі services/, де реалізовано чотири основні сервіси. SearchService відповідає за побудову маршрутів – прямих рейсів, рейсів з однією та двома пересадками – і передає сформовані варіанти до модуля AI-ранжування. BookingService реалізує логіку створення бронювання: перевірку наявності місць, формування записів квитків і платіжної заглушки. PricingService обчислює загальну вартість маршруту як суму базових цін усіх сегментів. ExplainService формує текстове пояснення для кожного варіанту маршруту, яке відображається користувачеві поруч з результатом пошуку.

Для передачі даних між шарами системи використовуються об’єкти передачі даних (DTO), що реалізовані через Pydantic-схеми у каталозі schemas/. Зокрема, SearchRequestDTO описує параметри запиту від клієнта – аеропорт відправлення, аеропорт призначення, дату вильоту та обмеження пошуку, а SearchResponseDTO містить список ітинераріїв із сегментами, оцінкою, ціною та поясненням. Застосування Pydantic-схем забезпечує

автоматичну валідацію вхідних даних і серіалізацію відповідей у формат JSON без необхідності ручного перетворення.

FastAPI автоматично генерує документацію Swagger UI за адресою /docs та ReDoc за адресою /redoc (рисунок 3.4). Ці інтерфейси дозволяють переглядати всі наявні ендпоінти, їхні схеми запиту та відповіді, а також виконувати тестові HTTP-запити безпосередньо у браузері без додаткових інструментів. Це суттєво спрощує інтеграцію між фронтендом і бекендом на етапі розробки.

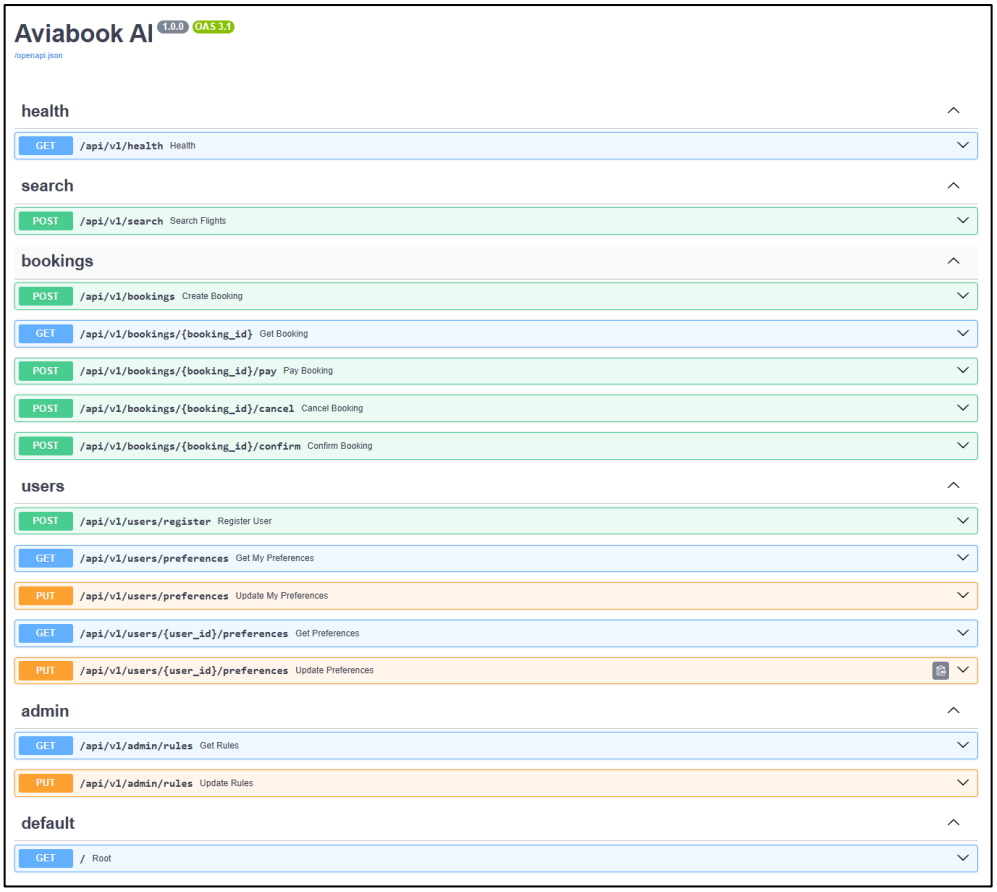


Рисунок 3.4 – Автоматично згенерована документація API (Swagger UI)

Обробка помилок реалізована на двох рівнях. На рівні конкретного ендпоінту перехоплюються ValueError (некоректні параметри запиту) та HTTPException (бізнес-логічні відмови), які перетворюються на відповідні HTTP-статуси з текстовим описом. На рівні застосунку зареєстровано

глобальний обробник усіх непередбачених винятків (global_exception_handler), який фіксує деталі помилки у системному журналі та повертає клієнту відповідь зі статусом 500 без розкриття внутрішньої реалізації. Система логування налаштована через окремий модуль core/logging.py із підтримкою рівнів деталізації, що задаються через конфігурацію.

3.1.3 Архітектура клієнтської частини

Клієнтська частина інтелектуальної системи бронювання авіаквитків реалізована як односторінковий застосунок (SPA – Single Page Application) з використанням бібліотеки React та мови TypeScript. Вибір React обумовлений компонентною моделлю побудови інтерфейсу, що дозволяє розбивати складний UI на незалежні, повторно використовувані елементи.

TypeScript забезпечує строгу типізацію на рівні фронтенду, що суттєво зменшує кількість помилок при роботі з даними від API. Збірка проєкту виконується через Vite, який забезпечує швидкий запуск у режимі розробки та оптимізований production-бандл.

Структура проєкту фронтенду організована у каталозі src/app/ і поділена на три функціональні групи (рисунок 3.5). Каталог pages/ містить п’ять сторінок застосунку: HomeSearchPage – головна сторінка з формою пошуку рейсів, ResultsPage – сторінка відображення результатів пошуку з фільтрацією та сортуванням, BookingPage – сторінка підтвердження та перегляду бронювання, ProfilePreferencesPage – сторінка налаштування вагових коефіцієнтів користувача, AdminRulesPage – сторінка адміністративного управління правилами. Каталог components/ містить перевикористовувані компоненти: SearchForm, FlightCard, FiltersPanel, ItineraryDetailsDrawer, ExplanationBadge та LoadingSkeleton. Каталог api/ відповідає за взаємодію з бекендом і містить модулі client.ts, search.ts та bookings.ts.



Рисунок 3.5 – Структура frontend-проєкту

Маршрутизація між сторінками реалізована за допомогою бібліотеки React Router v6 із використанням createBrowserRouter. Кожен маршрут обгорнутий у компонент AppShell, який містить спільну для всіх сторінок верхню навігаційну панель та футер, що дозволяє уникнути дублювання розмітки. Перехід між сторінками здійснюється без перезавантаження сторінки, що є ключовою характеристикою SPA-архітектури. При введенні невідомого маршруту користувач автоматично перенаправляється на головну сторінку через компонент Navigate.

Управління станом застосунку реалізовано засобами вбудованих хуків React (useState та useEffect) без підключення зовнішніх бібліотек управління станом. Кожна сторінка самостійно керує власним локальним станом: завантаженням даних, відображенням помилок, поточними фільтрами та обраним елементом, що є виправданим для застосунку з відносно простою структурою даних, де відсутня потреба у глобальному стані, що розділяється між багатьма компонентами одночасно.

Взаємодія з бекендом централізована через функцію apiFetch, реалізовану у модулі api/client.ts. Ця функція є обгорткою над стандартним браузерним fetch API і забезпечує: автоматичне формування заголовків запиту, додавання JWT-токена авторизації з localStorage, обробку JSON-відповіді та перетворення HTTP-помилок у типізований об’єкт ApiError зі статусом і текстом повідомлення.

Базова URL-адреса бекенду задається через змінну середовища VITE_API_BASE_URL, що дозволяє без змін у коді перемикатися між середовищем розробки та production. Поверх apiFetch у модулях search.ts та bookings.ts реалізовано типізовані функції для конкретних ендпоінтів (postSearch та createBooking) які приймають і повертають строго типізовані DTO, що відповідають схемам бекенду.

3.2 Проєктування бази даних

Як систему управління базами даних у проєкті використовується PostgreSQL. Вибір обумовлений підтримкою складних SQL-запитів із багатотабличними JOIN-операціями, які є необхідними при побудові маршрутів із пересадками, а також надійною транзакційністю при виконанні операцій бронювання. Взаємодія з базою даних реалізована через ORM SQLAlchemy з використанням декларативного підходу - усі таблиці описані як Python-класи, що наслідують спільний базовий клас Base.

База даних містить дев’ять таблиць, які можна розподілити на три групи за функціональним призначенням. До групи авіаційних даних належать таблиці airlines (авіакомпанії), airports (аеропорти), routes (маршрути між аеропортами) та flight_instances (конкретні рейси з датою, часом і ціною). До групи користувацьких даних – таблиці users та user_preferences, де остання зберігає вагові коефіцієнти для персоналізації ранжування. До групи бронювання – таблиці bookings, tickets та payment_stub.

Центральною сутністю авіаційної частини є flight_instances, яка містить зовнішній ключ на таблицю routes, а також денормалізовані поля source_iata та dest_iata для прискорення пошукових запитів без додаткового JOIN. Кожен рейс має атрибути часу відправлення та прибуття, базову ціну, валюту та кількість доступних місць. Таблиця routes у свою чергу посилається на airlines та на два записи airports – аеропорт відправлення і аеропорт призначення.

Зв’язки між сутностями реалізовані через relationship SQLAlchemy із налаштованим каскадним видаленням (cascade=«all, delete-orphan»), що висвітлено на рисунку 3.6.

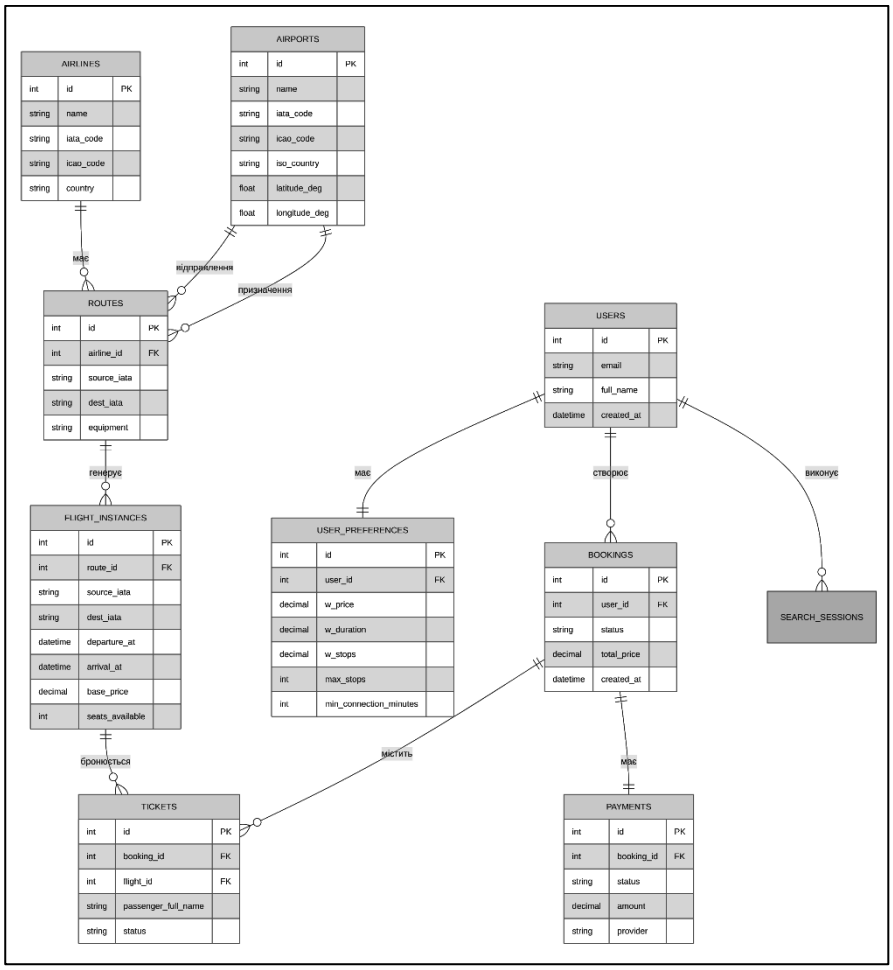


Рисунок 3.6 – Діаграма сутностей бази даних

Це означає, що при видаленні користувача автоматично видаляються його бронювання, квитки та платіжні записи, а при видаленні рейсу – усі прив’язані до нього квитки. Модель User має три відношення: один-до-одного з UserPreference, один-до-багатьох з Booking та один-до-багатьох з SearchSession. Модель Booking пов’язана з Ticket та PaymentStub через аналогічні каскадні відношення.

3.2.1 Архітектура ETL-модуля

ETL-модуль є необхідним, оскільки система пошуку та рекомендації авіаквитків потребує великого обсягу актуальних даних про авіакомпанії, аеропорти та маршрути. Ручне заповнення бази даних є трудомістким і схильним до помилок, тому для автоматизації процесу збору, перетворення та завантаження даних використовується ETL-модуль. Його застосування забезпечує цілісність даних, скорочує час підготовки системи до роботи та дозволяє регулярно оновлювати інформацію з зовнішніх джерел.

Перший скрипт – import_openflights.py – виконує імпорт даних із відкритого набору OpenFlights. З файлу airlines.dat завантажуються записи авіакомпаній із полями назви, IATA-коду, ICAO-коду, позивного та країни реєстрації. З файлу routes.dat імпортуються маршрути між аеропортами із прив’язкою до відповідної авіакомпанії. Формат OpenFlights використовує CSV із роздільником-комою, де відсутні значення позначаються рядком \N – для їх коректної обробки реалізовано функцію нормалізації _norm(), яка перетворює такі значення на None. Імпорт виконується порційно з фіксацією транзакції кожні 2000 записів, що дозволяє уникнути надмірного споживання пам’яті при обробці великих файлів.

Другий скрипт – import_ourairports.py – завантажує дані про аеропорти з набору OurAirports, який містить більш повну інформацію порівняно з OpenFlights: координати (широта та довгота), висота над рівнем моря, код країни та регіону, тип аеропорту. Для збереження даних

використовується upsert-логіка - якщо аеропорт із даним IATA-кодом вже існує в базі, його запис оновлюється, а не дублюється. Це дозволяє безпечно запускати імпорт повторно без порушення цілісності даних.

Третій скрипт – build_flights.py – генерує конкретні екземпляри рейсів на основі імпортованих маршрутів. Для кожного маршруту на задану кількість днів уперед створюється кілька рейсів на добу із розрахованими часом відправлення, тривалістю польоту та ціною квитка. Тривалість розраховується через формулу Гаверсіна – відстань між координатами аеропортів ділиться на крейсерську швидкість 820 км/год із додаванням 35 хвилин на розворот. Ціна формується на основі відстані маршруту (базовий коефіцієнт 0,12 USD/км) із застосуванням пікового множника 1,10 для ранкових та вечірніх годин вильоту, а також випадкового шуму в діапазоні 0,85–1,25 для імітації реальної варіативності тарифів.

Для оптимізації швидкодії запис згенерованих рейсів до бази виконується через bulk insert – замість послідовного виклику session.add() для кожного запису використовується метод insert() із передачею списку словників. Це дозволяє скоротити час наповнення бази при генерації десятків тисяч рейсів. Дані обробляються чанками фіксованого розміру, після кожного з яких виконується commit(), що запобігає накопиченню великих транзакцій у пам’яті.

3.2.2 Проєктування логічної моделі даних

Логічна модель даних інтелектуальної системи бронювання авіаквитків побудована навколо трьох функціональних доменів: авіаційні довідникові дані, дані користувачів та дані транзакцій бронювання. Кожен домен представлений групою взаємопов’язаних сутностей, між якими визначено зв’язки із зазначенням типу кардинальності та умов референційної цілісності.

Домен авіаційних даних включає чотири сутності. Сутність airlines є довідником авіакомпаній і містить атрибути назви, IATA- та ICAO-кодів, позивного та країни реєстрації. Сутність airports є довідником аеропортів із атрибутами назви, IATA- та ICAO-кодів, географічних координат, країни та регіону.

Сутність routes описує маршрут між двома аеропортами в рамках конкретної авіакомпанії і містить коди аеропорту відправлення та призначення, тип повітряного судна та ознаку кодшерингу. Між airlines та routes визначено зв'язок один-до-багатьох – одна авіакомпанія може мати довільну кількість маршрутів, тоді як кожен маршрут належить рівно одній авіакомпанії.

Сутність flight_instances є центральною в авіаційному домені та описує конкретний рейс – екземпляр маршруту з фіксованими датою та часом відправлення і прибуття, базовою ціною та кількістю доступних місць. Між routes та flight_instances визначено зв'язок один-до-багатьох, оскільки один маршрут може мати безліч рейсів у різні дати.

Домен користувачів включає дві сутності. Сутність users зберігає облікові дані користувача: електронну пошту, повне ім'я, хешований пароль та дату реєстрації. Сутність user_preferences містить персональні налаштування ранжування – вагові коефіцієнти для критеріїв ціни, тривалості та кількості пересадок, а також обмеження за замовчуванням для пошуку. Між users та user_preferences визначено зв'язок один-до-одного із забезпеченням унікальності через обмеження unique=True на зовнішньому ключі. Окремо виділено сутність search_sessions, яка фіксує кожен виконаний пошуковий запит у вигляді JSON-рядків запиту та відповіді - це дозволяє в майбутньому аналізувати поведінку користувачів та дотренувати модель ранжування на реальних даних.

Домен бронювання включає три сутності. Сутність bookings є агрегатним коренем транзакції бронювання та зберігає посилання на користувача, статус (CREATED/CONFIRMED/CANCELLED), загальну

вартість та дату створення. Сутність tickets представляє окремий квиток у рамках бронювання – посилання на конкретний рейс, ім’я пасажира, документ та статус квитка (RESERVED/ISSUED/CANCELLED).

Між bookings та tickets визначено зв’язок один-до-багатьох, оскільки одне бронювання може включати квитки на кілька сегментів маршруту. Зв’язок між tickets та flight_instances також є багато-до-одного – на один рейс може бути куплено кілька квитків у рамках різних бронювань. Сутність payments є заглушкою платіжної системи і пов’язана з bookings зв’язком один-до-одного.

Всі зовнішні ключі в моделі визначені з явною стратегією видалення. Для більшості залежних сутностей використовується CASCADE – при видаленні батьківського запису автоматично видаляються всі дочірні. Виняток становить зв’язок tickets – flight_instances, де застосовується стратегія RESTRICT, яка забороняє видалення рейсу, якщо на нього вже куплено хоча б один квиток, що захищає від втрати даних про здійснені бронювання.

3.2.3 Реалізація моделей з використанням SQLAlchemy

Усі моделі бази даних реалізовані з використанням декларативного підходу ORM SQLAlchemy, де кожна таблиця описується як Python-клас, що наслідує спільний базовий клас Base, визначений у модулі app/db/base.py. Базовий клас створюється через DeclarativeBase із SQLAlchemy 2.x, що забезпечує сумісність із сучасним синтаксисом та підтримку типізації. Реєстрація всіх моделей у метаданих відбувається автоматично в момент їх імпорту, тому в main.py виконується явний імпорт усіх модулів моделей перед викликом Base.metadata.create_all().

Кожна модель визначає атрибут __tablename__ із назвою відповідної таблиці та набір колонок через Column. Первинний ключ у всіх моделях реалізований як автоінкрементний цілочисельний атрибут id із параметрами

primary_key=True та index=True. Зовнішні ключі описуються через ForeignKey із явним зазначенням цільової таблиці та стратегії видалення ondelete. Для полів, які часто використовуються у фільтрах пошукових запитів, визначено індекси через параметр index=True безпосередньо у Column або через __table_args__ для складених індексів.

Зв’язки між моделями реалізовані через директиву relationship, яка дозволяє звертатися до пов’язаних об’єктів через атрибути Python без написання явних SQL-запитів. У всіх залежних моделях визначено двосторонні відношення через параметр back_populates – наприклад, User.bookings повертає список бронювань користувача, а Booking.user повертає об’єкт користувача. Параметр uselist=False використовується для відношень один-до-одного - зокрема у User.preference та Booking.payment - щоб замість списку повертався єдиний об’єкт або None.

Каскадне видалення налаштовано через параметр cascade=«all, delete-orphan» у відношеннях relationship. Це означає, що при видаленні батьківського запису через сесію SQLAlchemy дочірні записи видаляються автоматично без необхідності писати додаткову логіку.

Паралельно на рівні бази даних через ondelete=«CASCADE» у ForeignKey забезпечується каскадне видалення навіть при прямих SQL-операціях в обхід ORM. Виняток становить зовнішній ключ у Ticket.flight_id, де встановлено ondelete= «RESTRICT» - це забороняє видалення рейсу на рівні бази даних, якщо на нього існують квитки.

Для оптимізації запитів у моделі Route визначено складений індекс на полях source_iata та dest_iata через __table_args__ (лістинг 3.1).

Лістинг 3.1 – Реалізація моделі FlightInstance засобами SQLAlchemy

```
__table_args__ = (
    Index("ix_routes_src_dst", "source_iata", "dest_iata"),
)
```

Аналогічно у моделі FlightInstance окремі індекси визначено на полях source_iata, dest_iata, departure_at та arrival_at, оскільки саме за цими полями виконується основна фільтрація при пошуку рейсів. Денормалізація кодів аеропортів безпосередньо у FlightInstance є свідомим архітектурним рішенням - вона дозволяє виконувати пошукові запити без додаткового JOIN до таблиці routes, що суттєво прискорює роботу при великій кількості записів.

3.2.4 Реалізація таблиць користувачів та уподобань

Таблиця користувачів реалізована у моделі User і є центральною сутністю домену авіаційних бронювань з точки зору персоналізації. Модель містить поля електронної пошти з обмеженням унікальності (unique=True) та індексом для швидкого пошуку при автентифікації, хешованого паролю, повного імені та дати реєстрації. Поле created_at використовує server_default=func.now(), що означає автоматичне встановлення поточного часу на рівні СУБД без участі Python-коду - це гарантує коректне значення навіть при прямих SQL-вставках в обхід ORM.

Таблиця уподобань користувача реалізована у моделі UserPreference і зберігає два типи персональних налаштувань. Перший тип – вагові коефіцієнти для алгоритму ранжування: w_price, w_duration та w_stops, кожен із яких має тип Numeric(6, 3) та значення за замовчуванням 1.000. Ці коефіцієнти визначають відносну важливість відповідного критерію при обчисленні загальної оцінки маршруту – чим вище значення, тим більший вплив критерію на результат ранжування.

Другий тип – обмеження для пошуку за замовчуванням: max_stops задає максимально допустиму кількість пересадок, а min_connection_minutes – мінімальний час стикування між рейсами в хвилинах. Ці параметри автоматично підставляються у пошуковий запит, якщо користувач не вказав їх явно.

Зв’язок між User та UserPreference реалізований як один-до-одного через зовнішній ключ user_id із обмеженням unique=True у таблиці user_preferences. На рівні ORM відношення визначено з параметром uselist=False, що дозволяє звертатися до уподобань через атрибут user.preference як до єдиного об’єкта, а не списку. Каскадне видалення гарантує, що при видаленні облікового запису користувача його уподобання також автоматично видаляються.

У поточній реалізації система підтримує роботу без обов’язкової авторизації – для демонстраційних цілей у users.py реалізовано функцію ensure_default_user(), яка при кожному зверненні до ендпоінту уподобань перевіряє наявність демонстраційного користувача з id=1 і створює його якщо він відсутній. Це дозволяє фронтенду взаємодіяти з API персоналізації без реалізації повного циклу реєстрації та автентифікації, що є доцільним у контексті демонстраційної версії системи.

3.2.5 Реалізація таблиць рейсів та маршрутів

Таблиця авіакомпаній реалізована у моделі Airline і є довідниковою сутністю, що не змінюється в процесі роботи системи. Модель містить поля назви авіакомпанії, IATA- та ICAO-кодів з індексами для швидкого пошуку, позивного та країни реєстрації. Поле is_active зберігає значення у форматі OpenFlights – рядок ‘Y’ або ‘N’ – що відповідає оригінальному формату імпортованих даних і дозволяє уникнути перетворень при завантаженні.

Таблиця аеропортів реалізована у моделі Airport і також є довідниковою. Окрім стандартних ідентифікаторів (IATA- та ICAO-кодів) модель зберігає географічні координати у вигляді полів latitude_deg та longitude_deg типу Float. Ці координати використовуються в ETL-модулі при генерації рейсів для розрахунку відстані між аеропортами за формулою Гаверсіна, що є основою для обчислення тривалості польоту та базової ціни квитка. Поле ident відповідає внутрішньому ідентифікатору OurAirports і

використовується як резервний ключ при upsert-імпорті у випадках, коли IATA-код відсутній.

Таблиця маршрутів реалізована у моделі Route і описує статичний зв’язок між двома аеропортами в рамках конкретної авіакомпанії. Зовнішній ключ airline_id посилається на таблицю airlines із каскадним видаленням.

Поля source_iata та dest_iata зберігають коди аеропортів відправлення та призначення і проіндексовані як окремо, так і разом через складений індекс ix_routes_src_dst, що прискорює запити на зразок «знайти всі маршрути з аеропорту KBP» та «знайти маршрут між KBP і WAW», які є найпоширенішими при побудові графа пересадок. Поле equipment зберігає коди типів повітряних суден у форматі рядка, що відповідає формату OpenFlights.

Таблиця екземплярів рейсів реалізована у моделі FlightInstance і є найбільш навантаженою з точки зору пошукових запитів. Зовнішній ключ route_id пов’язує рейс із відповідним маршрутом, однак для уникнення зайвих JOIN-операцій поля source_iata та dest_iata денормалізовані безпосередньо у цій таблиці. Поля departure_at та arrival_at типу DateTime зберігають точний час відправлення та прибуття і проіндексовані для прискорення фільтрації за датою вильоту.

Поле base_price має тип Numeric(10, 2), що гарантує точність фінансових розрахунків без втрат через особливості представлення чисел із плаваючою точкою. Поля seats_total та seats_available дозволяють контролювати доступність місць – при бронюванні значення seats_available зменшується на кількість придбаних квитків, що запобігає надмірному продажу місць.

3.2.6 Реалізація таблиць бронювання

Таблиця бронювань реалізована у моделі Booking і є агрегатним коренем транзакції придбання квитків. Модель містить зовнішній ключ

user_id із посиланням на таблицю users та каскадним видаленням, поле статусу status із можливими значеннями CREATED, CONFIRMED та CANCELLED, загальну вартість бронювання total_price типу Numeric(10, 2), валюту та дату створення.

Статусна модель відображає життєвий цикл бронювання - після створення воно переходить у стан підтвердження після успішної оплати або скасовується у разі відмови користувача чи неуспішної транзакції. Поле total_price обчислюється на рівні PricingService як сума базових цін усіх сегментів маршруту і зберігається у таблиці для уникнення повторних обчислень при кожному зверненні до запису бронювання.

Таблиця квитків реалізована у моделі Ticket і описує окремий квиток у рамках бронювання. Модель містить два зовнішніх ключі - booking_id із посиланням на bookings та каскадним видаленням, і flight_id із посиланням на flight_instances із стратегією RESTRICT. Така різниця у стратегіях видалення є принциповою: квиток є частиною бронювання і не має сенсу без нього, тому каскадне видалення тут логічне; натомість рейс існує незалежно від квитків, і його заборонено видаляти доки на нього є хоча б один виданий квиток.

Поля passenger_full_name та passenger_doc зберігають дані пасажира безпосередньо у квитку, а не через посилання на профіль користувача – це дозволяє купувати квитки на інших осіб у рамках одного бронювання. Статус квитка status має власний цикл значень – RESERVED, ISSUED та CANCELLED – незалежний від статусу батьківського бронювання.

Таблиця платежів реалізована у моделі PaymentStub і в контексті даної роботи виступає заглушкою платіжної системи. Модель пов’язана з bookings зв’язком один-до-одного через зовнішній ключ booking_id із обмеженням unique=True. Поле status відображає стан платіжної транзакції – PENDING, PAID або FAILED.

Поле provider зберігає назву платіжного провайдера зі значенням за замовчуванням MOCK, що явно вказує на імітаційний характер платіжного

модуля та використання тестового механізму обробки платежів без інтеграції з реальними фінансовими сервісами. Такий підхід спрощує розробку, тестування та налагодження системи на етапі проєктування. Поля amount та currency дублюють відповідні атрибути з таблиці bookings, що є поширеною практикою при зберіганні платіжних операцій. Це дозволяє зафіксувати фактичні параметри транзакції на момент її створення та забезпечити цілісність фінансової інформації навіть у випадку подальших змін даних бронювання, вартості послуг або умов оплати.

3.3 Реалізація модулів та REST API

3.3.1 Реалізація REST API

REST API серверної частини системи реалізований засобами FastAPI і доступний за базовим префіксом /api/v1. Версіонування у шляху є свідомим архітектурним рішенням – воно дозволяє в майбутньому додати нову версію API без порушення сумісності з існуючими клієнтами. Усі ендпоінти повертають дані у форматі JSON, а вхідні дані автоматично валідуються через Pydantic-схеми ще до виклику бізнес-логіки. Перелік реалізованих ендпоінтів наведено у таблиці 3.1.

Таблиця 3.1 – Перелік ендпоінтів REST API

Метод	URL	Призначення
GET	/api/v1/health	Перевірка стану сервісу
POST	/api/v1/search	Пошук рейсів
POST	/api/v1/bookings	Створення бронювання
GET	/api/v1/bookings/{id}	Отримання бронювання
POST	/api/v1/bookings/{id}/confirm	Підтвердження бронювання
POST	/api/v1/bookings/{id}/pay	Оплата бронювання
POST	/api/v1/bookings/{id}/cancel	Скасування бронювання
POST	/api/v1/users/register	Реєстрація користувача
GET	/api/v1/users/preferences	Отримання уподобань
PUT	/api/v1/users/preferences	Оновлення уподобань

Передача даних між клієнтом і сервером здійснюється через типізовані DTO, реалізовані як Pydantic-моделі у каталозі schemas/. Для кожного ендпоінту визначено окрему схему запиту та відповіді – SearchRequestDTO описує параметри пошукового запиту, а SearchResponseDTO містить список ітинераріїв із усіма необхідними полями для відображення на фронтенді. Використання окремих схем дозволяє чітко розмежувати вхідний та вихідний контракт API і уникнути випадкового витоку внутрішніх полів моделей бази даних у відповідь клієнту.

Кожен роутер реєструється в main.py через метод app.include_router() із зазначенням префіксу та тегу для групування в документації. Залежність від сесії бази даних передається в ендпоінти через механізм Dependency Injection FastAPI – функція get_db() із модуля app/db/session.py створює нову сесію для кожного запиту і гарантує її закриття після завершення обробки незалежно від результату.

Обробка помилок реалізована на двох рівнях – на рівні ендпоінту перехоплюються ValueError та HTTPException, а на глобальному рівні зареєстровано обробник непередбачуваних винятків, який фіксує деталі у журналі подій та повертає клієнту відповідь зі статусом 500.

3.3.2 Реалізація системи керування користувачами та їхніми уподобаннями

Модуль користувачів реалізований у файлі api/v1/routes/users.py і забезпечує операції реєстрації, отримання та оновлення персональних уподобань. Робота з даними користувачів здійснюється через репозиторій UserRepository, який інкапсулює всі звернення до бази даних і надає роутеру типізований інтерфейс без прямих SQL-запитів.

Ендпоінт реєстрації приймає схему RegisterUserRequestDTO із полями електронної пошти, повного імені та паролю. Перед збереженням

виконується перевірка унікальності через метод get_by_email() репозиторію – якщо користувач із таким email вже існує, повертається відповідь зі статусом 400. Після створення користувача автоматично ініціалізується запис уподобань із значеннями за замовчуванням. Реалізацію ендпоінту наведено у лістингу 3.2.

Лістинг 3.2 – Реалізація ендпоінту реєстрації користувача

```
@router.post("/users/register",
response_model=RegisterUserResponseDTO)
def register_user(payload: RegisterUserRequestDTO, db: Session
= Depends(get_db)):
    repo = UserRepository(db)
    if repo.get_by_email(payload.email):
        raise HTTPException(status_code=400, detail="Email
already registered")
    user = User(
        email=payload.email.lower(),
        full_name=payload.full_name,
        hashed_password=payload.password,)
    user = repo.create(user)
    pref = UserPreference(user_id=user.id)
    db.add(pref)
    db.commit()
    return RegisterUserResponseDTO(
        user=UserDTO(id=user.id, email=user.email,
full_name=user.full_name))
```

Кожен новий користувач одразу після реєстрації має повністю ініціалізований профіль уподобань і може використовувати персоналізований пошук без додаткових налаштувань.

Ендпоінти уподобань GET та PUT /api/v1/users/preferences забезпечують отримання та оновлення вагових коефіцієнтів і обмежень пошуку. Схема UpdatePreferenceRequestDTO містить поля w_price, w_duration, w_stops, max_stops та min_connection_minutes. Реалізацію оновлення уподобань наведено у лістингу 3.3.

Лістинг 3.3 – Реалізація ендпоінту оновлення уподобань користувача

```
@router.put("/users/preferences",
response_model=GetPreferenceResponseDTO)
def update_my_preferences(
    payload: UpdatePreferenceRequestDTO,
    db: Session = Depends(get_db)):
    user = ensure_default_user(db)
    pref = user.preference
    pref.w_price = payload.w_price
    pref.w_duration = payload.w_duration
    pref.w_stops = payload.w_stops
    pref.max_stops = payload.max_stops
    pref.min_connection_minutes =
payload.min_connection_minutes
    db.commit()
    return GetPreferenceResponseDTO(preference=payload)
```

Оновлення виконується як повне перезаписування всіх полів уподобань – часткове оновлення окремих коефіцієнтів не передбачено, що спрощує логіку як на сервері так і на фронтенді.

У поточній реалізації ендпоінти уподобань працюють із демонстраційним користувачем із фіксованим id=1 без перевірки токена авторизації. Функція ensure_default_user() при кожному виклику перевіряє наявність цього користувача в базі і створює його у разі відсутності – разом із початковим записом уподобань. Реалізацію цієї функції наведено у лістингу 3.4.

Лістинг 3.4 – Функція ініціалізації демонстраційного користувача

```
def ensure_default_user(db: Session) -> User:
    user = db.get(User, DEFAULT_USER_ID)
    if user is None:
        user             =             User(id=DEFAULT_USER_ID,
email="demo@aviabook.local",       full_name="Demo       User",
hashed_password="demo-demo",)
        db.add(user)
        db.commit()
        db.refresh(user)
```

Продовження лістингу 3.4

```
  pref = getattr(user, "preference", None)
    if pref is None:
        pref = UserPreference(user_id=user.id)
        db.add(pref)
        db.commit()
        db.refresh(pref)
    return user
```

Це дозволяє фронтенду повноцінно взаємодіяти з модулем персоналізації, забезпечуючи коректну роботу системи в демонстраційному режимі без необхідності проходження повного циклу автентифікації при кожному запиті.

3.3.3 Реалізація модуля пошуку рейсів

Модуль пошуку рейсів реалізований у класі SearchService і є центральним компонентом серверної частини системи. Сервіс отримує параметри пошуку через схему SearchRequestDTO, будує маршрути трьох типів – прямі рейси, рейси з однією пересадкою та рейси з двома пересадками – після чого передає сформовані варіанти на фільтрацію та ранжування. Результат повертається у вигляді схеми SearchResponseDTO із розподілом на прийнятні та відхилені маршрути.

Пошук прямих рейсів реалізований у методі _build_direct(), який звертається до репозиторію FlightRepository через метод search_flights_leg(). Запит фільтрує рейси за кодами аеропортів відправлення та призначення, часовим вікном у межах заданої дати вильоту та наявністю вільних місць. Реалізацію методу наведено у лістингу 3.5.

Лістинг 3.5 – Реалізація пошуку прямих рейсів

```
def _build_direct(self, req: SearchRequestDTO, day_start:
datetime, day_end: datetime) -> List[ItineraryDTO]:
```

Продовження лістингу 3.5

```
    flights =
self.flight_repo.search_flights_leg(source_iata=req.source_iat
a, dest_iata=req.dest_iata, departure_from=day_start,
departure_to=day_end, limit=300,)
    items: List[ItineraryDTO] = []
    for f in flights:
        seg = self._to_segment(f)
        items.append(self._to_itinerary([seg]))
    return items
```

Кожен знайдений рейс перетворюється на об’єкт FlightSegmentDTO через метод _to_segment(), після чого з нього формується ItineraryDTO із одним сегментом. Це уніфікує представлення прямих рейсів та маршрутів із пересадками – в обох випадках результат є об’єктом ItineraryDTO із списком сегментів.

Пошук рейсів з однією пересадкою реалізований у методі _build_one_stop(). Алгоритм спочатку отримує всі рейси, що відправляються з аеропорту відправлення у задану дату, після чого для кожного з них шукає другий сегмент із проміжного аеропорту до аеропорту призначення. При цьому перевіряється часове вікно стикування – другий рейс має відправлятися не раніше ніж через min_connection_minutes після прибуття першого і не пізніше ніж через MAX_LAYOVER_HOURS годин. Реалізацію алгоритму наведено у лістингу 3.6.

Лістинг 3.6 – Реалізація пошуку рейсів з однією пересадкою

```
def _build_one_stop(self, req, day_start, day_end,
                    min_conn, max_layover) ->
List[ItineraryDTO]:
    first_legs = self.flight_repo.search_flights_departing(
        source_iata=req.source_iata, departure_from=day_start,
        departure_to=day_end, limit=self.MAX_FIRST_LEGS,)
    items: List[ItineraryDTO] = []
    for f1 in first_legs:
```

Продовження лістингу 3.6

```
        if not f1.dest_iata or f1.dest_iata.upper() ==
req.source_iata.upper(): continue
        if f1.dest_iata.upper() == req.dest_iata.upper():
            continue
        dep_from = f1.arrival_at + min_conn
        dep_to = f1.arrival_at + max_layover
        second_legs = self.flight_repo.search_flights_leg(
source_iata=f1.dest_iata, dest_iata=req.dest_iata,
departure_from=dep_from, departure_to=dep_to,
limit=self.MAX_SECOND_LEGS_PER_FIRST,)
        for f2 in second_legs:
            segs = [self._to_segment(f1),self._to_segment(f2)]
            items.append(self._to_itinerary(segs))
    return items
```

Аналогічна логіка застосована у методі _build_two_stops() для маршрутів із двома пересадками – до алгоритму додається третій вкладений цикл із пошуком фінального сегменту з другого проміжного аеропорту до аеропорту призначення. При цьому система перевіряє що жоден із проміжних аеропортів не збігається з аеропортом відправлення, призначення або один з одним, що виключає циклічні маршрути.

Після побудови всіх варіантів кожен маршрут проходить через метод _apply_constraints_only(), який перевіряє відповідність обмеженням запиту – максимальній кількості пересадок, максимальній ціні та мінімальному часу стикування між сегментами. Маршрути, що не відповідають хоча б одній умові, позначаються як відхилені з текстовим описом причини.

Прийнятні маршрути передаються до модуля ранжування rank_itineraries(), відхилені – одразу отримують нульову оцінку та пояснення від ExplainService. У фінальній відповіді прийнятні маршрути розміщуються першими у порядку спадання оцінки, відхилені – в кінці списку.

3.3.4 Реалізація модуля бронювання

Модуль бронювання реалізований у класі BookingService і забезпечує повний цикл операцій із бронюванням – від створення до підтвердження та скасування. Сервіс взаємодіє з двома репозиторіями: BookingRepository для операцій із записами бронювань і квитків, та FlightRepository для перевірки наявності місць і їх резервування.

Метод create_booking() реалізує логіку створення бронювання у чотири послідовні кроки. На першому кроці для кожного переданого ідентифікатора рейсу виконується перевірка існування запису та наявності вільних місць. Реалізацію перевірки наведено у лістингу 3.7.

Лістинг 3.7 – Перевірка наявності рейсів та місць

```
for fid in req.flight_ids:
    f = self.flight_repo.get_by_id(fid)
    if not f: raise ValueError(f"Flight not found: {fid}")
    if f.seats_available <= 0:
        raise ValueError(f"No seats available: {fid}")
    flights.append(f)
```

У разі відсутності рейсу або вільних місць генерується виняток ValueError із відповідним повідомленням, який на рівні роутера перехоплюється і повертається клієнту зі статусом 400.

На другому кроці обчислюється загальна вартість бронювання як сума базових цін усіх сегментів і створюється запис Booking зі статусом CREATED. Реалізацію цього кроку наведено у лістингу 3.8.

Лістинг 3.8 – Створення запису бронювання

```
for f in flights:
    total_price += Decimal(str(f.base_price))
booking = Booking(user_id=req.user_id, status="CREATED",
total_price=total_price, currency=currency,)
booking = self.booking_repo.create_booking(booking)
```

На третьому кроці для кожного рейсу зменшується кількість доступних місць на кількість пасажирів і створюються окремі записи Ticket для кожної комбінації рейс-пасажир зі статусом RESERVED. Реалізацію наведено у лістингу 3.9.

Лістинг 3.9 – Резервування місць та створення квитків

```
for f in flights:
    self.flight_repo.decrement_seats(f.id,
qty=len(req.passengers))
    for p in req.passengers:
        t = Ticket(booking_id=booking.id, flight_id=f.id,
passenger_full_name=p.full_name, passenger_doc=p.doc,
status="RESERVED",)
        self.booking_repo.add_ticket(t)
```

На четвертому кроці створюється запис PaymentStub зі статусом PENDING, що фіксує факт очікування оплати. Така послідовність кроків гарантує атомарність операції – якщо будь-який із кроків завершується з помилкою, транзакція відкочується і жодних часткових записів у базі не залишається.

Підтвердження бронювання реалізоване у методі confirm_booking(), який змінює статус запису Booking на CONFIRMED через репозиторій. Реалізацію методу наведено у лістингу 3.10.

Лістинг 3.10 – Реалізація підтвердження бронювання

```
def confirm_booking(self, booking_id: int) ->
ConfirmBookingResponseDTO:
    booking = self.booking_repo.set_status(booking_id,
"CONFIRMED")
    return ConfirmBookingResponseDTO(booking_id=booking.id,
status=booking.status)
```

Операції скасування та оплати реалізовані аналогічно – через зміну статусів відповідних записів Booking та PaymentStub. Скасування

переводить бронювання у статус CANCELLED, оплата – змінює статус платіжного запису з PENDING на PAID і підтверджує бронювання. Такий підхід забезпечує чіткий та відстежуваний життєвий цикл кожного бронювання в системі.

3.3.5 Реалізація системи логування та обробки помилок

Система логування налаштована централізовано у модулі core/logging.py через стандартний механізм dictConfig бібліотеки logging. При старті застосунку викликається функція setup_logging(), яка конфігурує єдиний обробник виводу у стандартний потік stdout із форматом, що включає мітку часу, рівень повідомлення, назву модуля та текст. Реалізацію налаштування формату наведено у лістингу 3.11.

Лістинг 3.11 – Налаштування формату логування

```
"formatters": {"default": {"format": "%(asctime)s |
%(levelname)s | %(name)s | %(message)s",},},
```

Рівень деталізації логів визначається параметром LOG_LEVEL із конфігурації системи і може задаватись через змінну середовища без зміни коду. Окремі логери визначені для uvicorn, uvicorn.error та uvicorn.access із параметром propagate=False, що запобігає дублюванню повідомлень у кореневому логері. У кожному модулі бізнес-логіки логер отримується через допоміжну функцію get_logger(__name__), що забезпечує ієрархічне найменування логерів відповідно до структури модулів.

Обробка помилок реалізована на двох незалежних рівнях. На рівні конкретних ендпоінтів перехоплюються передбачувані винятки – ValueError із бізнес-логіки та HTTPException із явно заданим статусом. Реалізацію такої обробки у ендпоінті пошуку наведено у лістингу 3.12.

Лістинг 3.12 – Обробка помилок на рівні ендпоінту

```
@router.post("/search", response_model=SearchResponseDTO)
def search_flights(payload: SearchRequestDTO,
                   db: Session = Depends(get_db)):
    try:
        service = SearchService(db)
        return service.search(payload)
    except ValueError as e:
        raise HTTPException(status_code=400, detail=str(e))
from e
    except HTTPException:
        raise
    except Exception as e:
        logger.exception("Search failed. payload=%s",
payload.model_dump())
        raise HTTPException(status_code=500, detail=f"Search
failed: {type(e).__name__}: {e}",) from e
```

Це дозволяє повертати клієнту зрозумілі повідомлення про помилки валідації зі статусом 400, не розкриваючи деталей внутрішньої реалізації при непередбачуваних збоях.

На глобальному рівні у main.py зареєстровано обробник усіх необроблених винятків через декоратор @app.exception_handler(Exception). Реалізацію глобального обробника наведено у лістингу 3.13.

Лістинг 3.13 – Глобальний обробник необроблених винятків

```
@app.exception_handler(Exception)
async def global_exception_handler(request: Request, exc:
Exception):
    logger.error(f"Unhandled error: {request.method}
{request.url}", exc_info=True)
    return JSONResponse(status_code=500, content={"detail":
"Internal server error"})
```

Глобальний обробник фіксує метод та URL запиту, що спричинив помилку, разом із повним стектрейсом у журналі подій, повертаючи клієнту лише загальне повідомлення без розкриття внутрішніх деталей реалізації.

Перевірка підключення до бази даних виконується при кожному старті застосунку у lifespan-контексті – у разі невдачі сервер не запускається і фіксує відповідну помилку у журналі. Це дозволяє оперативно виявляти проблеми з інфраструктурою ще до початку обробки запитів.

3.4 Реалізація клієнтської частини системи

3.4.1 Реалізація структури frontend-застосунку

Фронтенд-застосунок побудований як односторінковий React-додаток із чітким розподілом відповідальності між шарами. Точкою входу є файл main.tsx, де виконується монтування кореневого компонента App у DOM-елемент з ідентифікатором root. Компонент App у свою чергу містить лише один елемент – RouterProvider із сконфігурованим роутером, що робить маршрутизацію єдиною відповідальністю цього компонента.

Маршрутизація реалізована у файлі router.tsx через createBrowserRouter із React Router v6. Кожен маршрут обгорнутий у компонент-оболонку AppShell, який містить спільну навігаційну панель із посиланнями на основні розділи та футер. Як видно з лістингу 3.14, конфігурація роутера визначає п’ять маршрутів із відповідними сторінками та перенаправлення на головну для всіх невідомих шляхів.

Лістинг 3.14 – Конфігурація маршрутизації застосунку

```
const router = createBrowserRouter([
  { path: "/", element: <AppShell><HomeSearchPage
/></AppShell> },
  { path: "/results", element: <AppShell><ResultsPage
/></AppShell> },
  { path: "/booking/:bookingId", element:
<AppShell><BookingPage /></AppShell> },
  { path: "/preferences", element:
<AppShell><ProfilePreferencesPage /></AppShell> },
```

Продовження лістингу 3.14

```
  { path: "/admin/rules", element: <AppShell><AdminRulesPage
/></AppShell> },
  { path: "*", element: <Navigate to="/" replace /> },]);
```

Структура сторінок організована у каталозі pages/ і відповідає п’яти основним сценаріям використання системи. HomeSearchPage є головною сторінкою застосунку і відповідає за відображення форми пошуку та інформаційних блоків про можливості системи. При успішному надсиланні форми сторінка не виконує HTTP-запит безпосередньо – натомість вона формує рядок параметрів і передає їх на сторінку результатів через навігацію, що дозволяє зберігати стан пошуку у URL. Детальніше цей підхід показано у лістингу 3.15.

Лістинг 3.15 – Передача параметрів пошуку через URL

```
const onSubmit = (v: SearchFormValue) => {
  const params = new URLSearchParams();
  params.set("source", v.source_iata.trim().toUpperCase());
  params.set("dest", v.dest_iata.trim().toUpperCase());
  params.set("date", v.depart_date);
  params.set("pax", String(v.passengers));
  params.set("maxStops", String(v.constraints.max_stops));
  params.set("minConn",
String(v.constraints.min_connection_minutes));
  navigate(`/results?${params.toString()}`);
};
```

Такий підхід має практичну перевагу – користувач може поділитися посиланням на конкретні результати пошуку або повернутися до них через кнопку «назад» у браузері без повторного заповнення форми.

Перевикористовувані компоненти зосереджені у каталозі components/ і охоплюють усі повторювані елементи інтерфейсу. SearchForm інкапсулює логіку форми пошуку з валідацією полів. FlightCard відображає картку одного маршруту з інформацією про сегменти, ціну та оцінку. FiltersPanel

керує фільтрами та сортуванням на сторінці результатів. ItineraryDetailsDrawer відображає деталі обраного маршруту у бічній панелі. LoadingSkeleton показує анімований заповнювач під час завантаження даних. Кожен компонент отримує необхідні дані через пропси і не має прямого доступу до API – це відповідає принципу єдиної відповідальності і спрощує тестування окремих елементів інтерфейсу.

3.4.2 Реалізація сторінки пошуку рейсів

Сторінка пошуку реалізована у компоненті HomeSearchPage і складається з двох секцій – головного інформаційного блоку із формою пошуку та інформаційної секції з трьома картками, що описують ключові можливості системи. Вся логіка введення параметрів інкапсульована у компоненті SearchForm, який отримує єдиний проп – функцію зворотного виклику onSubmit.

Компонент SearchForm керує станом семи полів форми через окремі виклики useState: коди аеропортів відправлення та призначення, дата вильоту, кількість пасажирів, максимальна кількість пересадок, мінімальний час стикування та максимальна ціна. Дата за замовчуванням встановлюється на поточний день через функцію todayISO(). Кнопка відправлення форми блокується через атрибут disabled доки поля не пройдуть валідацію – перевіряється довжина кодів аеропортів, наявність дати та додатне значення кількості пасажирів, що реалізовано через useMemo (лістинг 3.16).

Лістинг 3.16 – Валідація форми пошуку

```
const canSubmit = useMemo(() => {
  return source.trim().length >= 3
    && dest.trim().length >= 3
    && !!date
    && pax > 0;
}, [source, dest, date, pax]);
```

Сторінка результатів реалізована у компоненті ResultsPage і є найбільш складним компонентом застосунку з точки зору управління станом. При монтуванні компонент зчитує параметри пошуку з URL через useSearchParams і формує об’єкт SearchRequest для відправлення на бекенд. HTTP-запит виконується у хуку useEffect із залежністю від об’єкта payload – повторний запит виконується автоматично при зміні будь-якого параметра пошуку або фільтра. Для уникнення стану гонки при швидкій зміні параметрів використовується прапорець alive, який запобігає оновленню стану після розмонтування компонента (лістинг 3.17).

Лістинг 3.17 – Захист від стану гонки при завантаженні результатів

```
useEffect(() => {
  let alive = true;
  async function run() {
    setLoading(true);
    try {
      const res = await postSearch(payload);
      if (!alive) return;
      setItems(Array.isArray(res.items) ? res.items : []);
    } catch (e: any) {
      if (!alive) return;
      setError(extractBestMessage(e, "Помилка пошуку"));
    } finally {
      if (!alive) return;
      setLoading(false)}}
  run();
  return () => { alive = false; };}, [payload]);
```

Отримані результати проходять через клієнтську фільтрацію та сортування у хуку useMemo. Фільтрація виконується за трьома критеріями – статусом відхилення, кількістю пересадок та ціною. Сортування підтримує три режими: за оцінкою ранжування у спадному порядку, за ціною та за тривалістю у зростаючому порядку.

Клієнтська фільтрація не виконує повторного запиту до сервера – вона працює з уже отриманим масивом результатів, що забезпечує миттєву

реакцію інтерфейсу на зміну фільтрів. Під час завантаження замість карток рейсів відображається компонент LoadingSkeleton, а у разі відсутності результатів – інформаційне повідомлення з підказками щодо розширення параметрів пошуку.

3.4.3 Реалізація сторінки бронювання

Сторінка бронювання реалізована у компоненті BookingPage і відповідає за відображення деталей конкретного бронювання та управління його станом. Ідентифікатор бронювання отримується з динамічного сегменту URL через хук useParams і одразу перетворюється на числовий тип – у разі некоректного значення відображається повідомлення про помилку без виконання запиту до сервера.

Завантаження даних бронювання виконується у хуку useEffect із залежністю від ідентифікатора. Аналогічно до ResultsPage, для запобігання оновленню стану після розмонтування компонента використовується прапорець alive. Отриманий об’єкт бронювання зберігається у стані booking і відображається у вигляді картки із трьома ключовими атрибутами – ідентифікатором, поточним статусом та загальною сумою.

Управління станом бронювання реалізоване через два обробники подій – onPay та onCancel, наведені у лістингу 3.18. Обидва обробники викликають відповідні функції з модуля api/bookings, отримують оновлений об’єкт бронювання у відповіді та одразу записують його у стан компонента через setBooking, що забезпечує миттєве оновлення відображуваного статусу без повторного запиту до сервера.

Лістинг 3.18 – Обробники оплати та скасування бронювання

```
const onPay = async () => {
  if (!booking) return;
  try {
    const b = await payBooking(booking.id);
```

Продовження лістингу 3.18

```
    setBooking(b);
  } catch (e: any) {
    alert(e?.message || "Помилка оплати");}};
const onCancel = async () => {
  if (!booking) return;
  try {
    const b = await cancelBooking(booking.id);
    setBooking(b);
  } catch (e: any) {
    alert(e?.message || "Помилка скасування");}};
```

Кнопки оплати та скасування блокуються через атрибут disabled залежно від поточного статусу бронювання – кнопка оплати недоступна якщо статус вже PAID, кнопка скасування – якщо статус CANCELLED. Деталі маршруту відображаються у компоненті ItineraryDetailsDrawer, який відкривається у вигляді бічної панелі при натисканні відповідної кнопки. Стан відкриття панелі керується локальним булевим прапорцем openDetails, що дозволяє уникнути зайвих перезапитів до сервера при повторному відкритті деталей.

3.4.4 Реалізація сторінки налаштувань користувача

Сторінка налаштувань реалізована у компоненті ProfilePreferencesPage і забезпечує перегляд та редагування персональних уподобань користувача, що безпосередньо впливають на ранжування результатів пошуку. При монтуванні компонент завантажує поточні значення уподобань із сервера через GET /api/v1/users/preferences і зберігає їх у локальному стані prefs.

Нормалізація відповіді від сервера виконується через функцію normalizePrefsResponse(), яка обробляє два можливі формати відповіді – безпосередній об’єкт уподобань або обгорнутий у поле preference. Для числових полів застосовується явне приведення типів через Number() із

значеннями за замовчуванням, що захищає від ситуацій коли сервер повертає null або рядковий тип замість числового. Детальніше це показано у лістингу 3.19.

Лістинг 3.19 – Нормалізація відповіді уподобань

```
function normalizePrefsResponse(data: any): Prefs {
  const p = (data && typeof data === "object" && "preference"
in data)
    ? data.preference : data;
  return {
    w_price: Number(p?.w_price ?? 1.0),
    w_duration: Number(p?.w_duration ?? 1.0),
    w_stops: Number(p?.w_stops ?? 1.0),
    price_sensitivity: p?.price_sensitivity == null
      ? 0.5 : Number(p.price_sensitivity),
    time_sensitivity: p?.time_sensitivity == null
      ? 0.5 : Number(p.time_sensitivity),
    max_stops: p?.max_stops == null ? null :
Number(p.max_stops),
    min_connection_minutes: p?.min_connection_minutes == null
      ? null : Number(p.min_connection_minutes),
  };}
```

Форма налаштувань містить шість редагованих полів, розташованих у сітці з двох колонок: вагові коефіцієнти w_price, w_duration та w_stops із кроком 0,1. Поле бажаної авіакомпанії у форматі IATA-коду, а також коефіцієнти чутливості до ціни та часу у діапазоні від 0 до 1. Кожне поле при зміні значення оновлює стан через патерн розпакування об’єкта { ...prefs, field: value }, що гарантує незмінність решти полів.

Збереження уподобань реалізовано у функції onSave, яка відправляє поточний стан prefs на сервер через PUT /api/v1/users/preferences і при успішній відповіді оновлює стан нормалізованими даними від сервера. Для підтвердження успішного збереження на 1,5 секунди відображається повідомлення «Збережено», після чого воно автоматично зникає через setTimeout. Відповідний фрагмент коду наведено у лістингу 3.20.

Лістинг 3.20 – Збереження уподобань із тимчасовим підтвердженням

```
const onSave = async () => {
  setSaved(null);
  setError(null);
  try {
    const p = await savePrefs(prefs);
    setPrefs(p);
    setSaved("Збережено");
    setTimeout(() => setSaved(null), 1500);
  } catch (e: any) {
    setError(e?.message || "Не вдалося зберегти");}};
```

У правому верхньому куті сторінки відображається компонент ExplanationBadge із підписом «Впливає на ранжування», який візуально нагадує користувачу про зв’язок між налаштуваннями та порядком відображення рейсів у результатах пошуку.

3.4.5 Взаємодія frontend із backend

Уся мережева взаємодія фронтенду з бекендом централізована через функцію apiFetch у модулі api/client.ts. Функція є типізованою оберткою над стандартним браузерним fetch і забезпечує уніфіковану обробку запитів незалежно від конкретного ендпоінту. Базова URL-адреса бекенду визначається зі змінної середовища VITE_API_BASE_URL, що дозволяє перемикатися між середовищами розробки та production без зміни коду.

Формування заголовків запиту відбувається автоматично – функція додає Content-Type: application/json для запитів із тілом та Authorization: Bearer якщо токен присутній у localStorage. Відповідний фрагмент наведено у лістингу 3.21.

Лістинг 3.21 – Формування заголовків HTTP-запиту

```
const headers = new Headers(options?.headers || {});
headers.set("Accept", "application/json");
if (!headers.has("Content-Type") && options?.body) {
```

Продовження лістингу 3.21

```
  headers.set("Content-Type", "application/json");}
const token = getToken();
if (token && !headers.has("Authorization")) {
  headers.set("Authorization", `Bearer ${token}`);}
```

При отриманні відповіді із кодом помилки формується типізований об’єкт ApiError, як показано у лістингу 3.22. Це дозволяє компонентам обробляти помилки однотипно без розбору сирої відповіді сервера.

Лістинг 3.22 – Формування об’єкта помилки

```
if (!res.ok) {
  const err: ApiError = {
    status: res.status,
    message: (data && (data.detail || data.message))
      || `HTTP ${res.status} ${res.statusText}`,
    details: data,
  };throw err;}
```

Поверх apiFetch у модулях search.ts та bookings.ts реалізовані типізовані функції для конкретних ендпоінтів. Модуль search.ts визначає типи SearchRequest, Itinerary та SearchResponse і експортує функцію postSearch, яка приймає строго типізований об’єкт запиту та повертає масив ітинераріїв.

Модуль bookings.ts містить складнішу логіку адаптації – функція toBackendCreateBookingPayload витягує ідентифікатори рейсів із сегментів ітинерарію, як видно з лістингу 3.23.

Лістинг 3.23 – Витяг ідентифікаторів рейсів із сегментів

```
const flight_ids = payload.itinerary.segments .map((s: any) =>
s?.flight_id).filter((id: any) => Number.isFinite(id)).map((id:
any) => Number(id));
```

Паралельно формується масив об’єктів пасажирів відповідно до кількості, переданої з форми (лістинг 3.24).

Лістинг 3.24 – Генерація масиву пасажирів

```
const count = Number(payload.passengers);
const passengers = Array.from({ length: count }).map((_, i) =>
({full_name:`Passenger ${i + 1}`,doc: null as string|null,}));
```

Окремої уваги заслуговує механізм визначення ідентифікатора користувача через функцію ensureUserId. Вона реалізує триступеневу логік – спочатку перевіряється явно переданий user_id, потім значення із localStorage, і лише якщо обидва варіанти недоступні – виконується автоматична реєстрація демонстраційного користувача через POST /api/v1/users/register із збереженням отриманого ідентифікатора у localStorage. Відповідь після успішного створення бронювання нормалізується через явне приведення полів до очікуваних типів, що захищає від розбіжностей між форматами відповіді бекенду та TypeScript-типами фронтенду.

3.5 Реалізація алгоритмів пошуку та персоналізації

3.5.1 Реалізація алгоритму пошуку рейсів

Алгоритм пошуку рейсів реалізований у класі FlightRepository і є основним інструментом доступу до даних про рейси в системі. Репозиторій інкапсулює всі SQL-запити і надає сервісному рівню типізований інтерфейс із трьох методів – search_flights_leg(), search_flights_departing() та decrement_seats().

Метод search_flights_leg() шукає рейси між конкретною парою аеропортів у заданому часовому вікні. Він застосовується для пошуку прямих рейсів та фінального сегменту маршрутів із пересадками. Запит

фільтрує записи за чотирма умовами – коди аеропортів відправлення та призначення, часове вікно вильоту та наявність вільних місць. Відповідний фрагмент наведено у лістингу 3.25.

Лістинг 3.25 – Запит пошуку рейсів між двома аеропортами

```
stmt =
(select(FlightInstance).where(FlightInstance.source_iata ==
source_iata.upper(), FlightInstance.dest_iata ==
dest_iata.upper(), FlightInstance.departure_at >=
departure_from, FlightInstance.departure_at <= departure_to,
FlightInstance.seats_available >
0,).order_by(FlightInstance.departure_at.asc()).limit(limit))
```

Метод search_flights_departing() шукає всі рейси з аеропорту відправлення без обмеження на пункт призначення і застосовується для отримання першого та другого сегментів при побудові маршрутів із пересадками. Відмінність від попереднього методу полягає у відсутності умови фільтрації за dest_iata – система отримує всі можливі проміжні напрямки і самостійно перевіряє їх придатність для стикування (лістинг 3.26).

Лістинг 3.26 – Запит пошуку всіх рейсів з аеропорту відправлення

```
stmt =
(select(FlightInstance).where(FlightInstance.source_iata ==
source_iata.upper(), FlightInstance.departure_at >=
departure_from, FlightInstance.departure_at <= departure_to,
FlightInstance.seats_available >
0,).order_by(FlightInstance.departure_at.asc()).limit(limit))
```

Обидва методи використовують selectinload для завантаження пов’язаних об’єктів маршруту та авіакомпанії в рамках одного запиту. Це вирішує проблему N+1 запитів – без явного завантаження зв’язків SQLAlchemy виконував би окремий запит для кожного рейсу при зверненні

до f.route.airline.name під час формування DTO. Відповідний фрагмент показано у лістингу 3.27.

Лістинг 3.27 – Завантаження пов’язаних об’єктів через selectinload

```
stmt = stmt.options(
selectinload(FlightInstance.route).selectinload(Route.airline)
)
```

Метод decrement_seats() відповідає за зменшення кількості доступних місць при бронюванні. Перед зменшенням виконується перевірка достатності місць – у разі нестачі генерується виняток ValueError, який на рівні BookingService перехоплюється і повертається клієнту зі статусом 400 (лістинг 3.28).

Лістинг 3.28 – Резервування місць на рейсі

```
def decrement_seats(self, flight_id: int, qty: int = 1) ->
None:
    flight = self.db.get(FlightInstance, flight_id)
    if flight is None: raise ValueError("Flight not found")
    if flight.seats_available < qty: raise ValueError("Not
enough seats available")
    flight.seats_available -= qty
    self.db.commit()
```

Результати обох пошукових методів впорядковуються за часом відправлення у зростаючому порядку та обмежуються параметром limit, що запобігає надмірному навантаженню при великій кількості рейсів у базі та передачі надлишкових даних у сервісний рівень.

3.5.2 Реалізація алгоритму пошуку пересадок

Алгоритм побудови маршрутів із пересадками реалізований у методах _build_one_stop() та _build_two_stops() класу SearchService і працює

за принципом послідовного розширення графа можливих з'єднань між рейсами. На відміну від пошуку прямих рейсів, де достатньо одного запиту до бази, побудова маршрутів із пересадками вимагає вкладених запитів – для кожного першого сегменту окремо шукаються можливі продовження.

Для маршрутів з однією пересадкою алгоритм спочатку отримує всі рейси що відправляються з аеропорту відправлення у задану дату через search_flights_departing(). Для кожного з них перевіряється що проміжний аеропорт не збігається з аеропортом відправлення чи призначення – це виключає вироджені маршрути. Після цього обчислюється часове вікно стикування і виконується пошук другого сегменту (лістинг 3.29).

Лістинг 3.29 – Обчислення часового вікна стикування

```
dep_from = f1.arrival_at + min_conn
dep_to = f1.arrival_at + max_layover
second_legs =
self.flight_repo.search_flights_leg(source_iata=f1.dest_iata,
dest_iata=req.dest_iata, departure_from=dep_from,
departure_to=dep_to, limit=self.MAX_SECOND_LEGS_PER_FIRST,)
```

Нижня межа вікна – час прибуття першого рейсу плюс мінімальний час пересадки min_connection_minutes. Верхня межа складає 6, що виключає маршрути з нереалістичним очікуванням пересадки. Кожна валідна пара перетворюється на об'єкт ItineraryDTO і додається до результатів.

Для маршрутів із двома пересадками до алгоритму додається третій вкладений рівень. Після знаходження другого проміжного аеропорту виконується додаткова перевірка його унікальності відносно всіх вже відвіданих вузлів маршруту (лістинг 3.30).

Лістинг 3.30 – Перевірка унікальності проміжних аеропортів

```
b = (f2.dest_iata or "").upper()
if b in {req.source_iata.upper(), a}:continue
if b == req.dest_iata.upper():continue
```

Після проходження перевірки обчислюється нове часове вікно для третього сегменту і виконується фінальний пошук рейсів до аеропорту призначення (лістинг 3.31).

Лістинг 3.31 – Пошук третього сегменту маршруту

```
dep3_from = f2.arrival_at + min_conn
dep3_to = f2.arrival_at + max_layover
third_legs =
self.flight_repo.search_flights_leg(source_iata=f2.dest_iata,
dest_iata=req.dest_iata, departure_from=dep3_from,
departure_to=dep3_to, limit=self.MAX_THIRD_LEGS_PER_SECOND,)
```

Кількість варіантів на кожному рівні обмежена константами – MAX_FIRST_LEGS=250, MAX_SECOND_LEGS_PER_FIRST=120 та MAX_THIRD_LEGS_PER_SECOND=80. Ці обмеження запобігають комбінаторному вибуху: теоретична максимальна кількість маршрутів із двома пересадками без обмежень становить 250×120×80=2 400 000, тоді як із обмеженнями вона зводиться до прийнятного для обробки значення. Після формування всіх варіантів маршрутів кожен із них передається на перевірку обмежень і подальше ранжування.

3.5.3 Реалізація алгоритму ранжування результатів

Ранжування маршрутів реалізоване як ML-пайплайн із трьох послідовних кроків – витяг ознак, векторизація з персоналізацією та обчислення оцінки через логістичну регресію. На першому етапі з маршруту вилучаються числові та категоріальні характеристики: тривалість, кількість пересадок, тип транспорту та завантаженість ділянок. Далі ознаки векторизуються та доповнюються персоналізованими компонентами – уподобаннями користувача і контекстом запиту. Отриманий вектор подається до моделі, яка обчислює оцінку для кожного маршруту-кандидата

і впорядковує їх у фінальному списку. Загальна схема пайплайну наведена на рисунку 3.7.

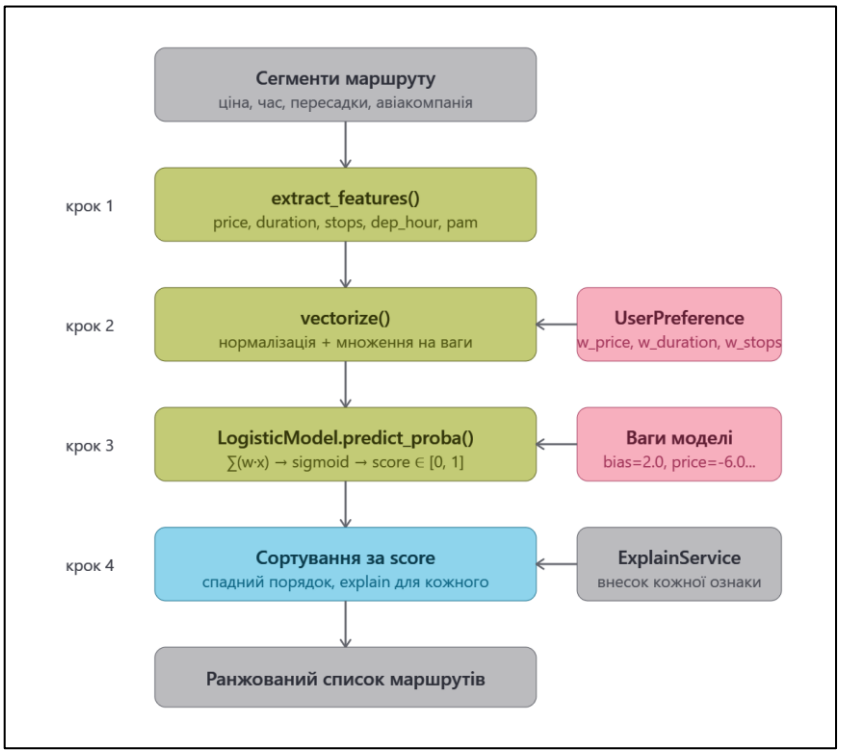


Рисунок 3.7 – Схема пайплайну ранжування маршрутів

На першому кроці функція extract_features() витягує із сегментів маршруту шість числових ознак: загальну ціну, тривалість у хвилинах, кількість пересадок, годину відправлення, годину прибуття та ознаку збігу з улюбленою авіакомпанією користувача. Ознака preferred_airline_match приймає значення 1.0 якщо IATA-код першого сегменту збігається з кодом у профілі користувача, і 0.0 у протилежному випадку (лістинг 3.32).

Лістинг 3.32 – Витяг ознак із сегментів маршруту

```
return {"price": _to_float(total_price), "duration_minutes":
float(dur_min), "stops": float(stops), "departure_hour":
dep_hour, "arrival_hour": arr_hour, "preferred_airline_match":
float(match),}
```

На другому кроці функція vectorize() нормалізує ознаки та множить їх на вагові коефіцієнти користувача. Нормалізація виконується діленням на масштабуючі константи – ціна на 1000, тривалість на 1000, кількість пересадок на 5. До нормалізованого вектора додається константний член bias=1.0, що дозволяє моделі зміщувати рівень оцінки незалежно від значень ознак (лістинг 3.33).

Лістинг 3.33 – Нормалізація та зважування вектора ознак

```
return {"bias": 1.0, "price_n": n["price_n"] * w_price,
"duration_n": n["duration_n"] * w_dur, "stops_n": n["stops_n"]
* w_stops, "pam": n["pam"] * w_air, "dep_h_n": n["dep_h_n"],
"arr_h_n": n["arr_h_n"],}
```

На третьому кроці клас LogisticModel обчислює скалярний добуток вектора ознак та вагового вектора моделі і передає результат через сигмоїд-функцію. Сигмоїд перетворює довільне дійсне число на значення у діапазоні від 0 до 1, що інтерпретується як оцінка привабливості маршруту (лістинг 3.34).

Лістинг 3.34 – Обчислення оцінки через логістичну регресію

```
def predict_proba(self, x: Dict[str, float]) -> float: s = 0.0
    for k, v in x.items(): s += float(self.weights.get(k,
0.0)) * float(v)
    return float(_sigmoid(s))
```

Від’ємні ваги моделі для ознак price_n (-6.0), duration_n (-2.2) та stops_n (-1.6) забезпечують що дешевші, швидші та з меншою кількістю пересадок маршрути отримують вищу оцінку. Метод explain() класу Scorer паралельно обчислює внесок кожної ознаки у фінальну оцінку – це значення передаються у ExplainService для формування текстового пояснення, яке відображається користувачеві поруч із картками результатів. Після

обчислення оцінки для кожного маршруту список сортується у порядку спадання через стандартний list.sort() із ключем за полем score.

3.5.4 Реалізація модулю персоналізації результатів

Механізм персоналізації реалізований у класі PersonalizationEngine і забезпечує адаптацію результатів ранжування до індивідуального профілю кожного користувача без перенавчання моделі. Персоналізація відбувається шляхом динамічного перерахування вагових коефіцієнтів на основі параметрів чутливості, збережених у таблиці user_preferences.

Клас отримує на вхід словник уподобань із трьома параметрами чутливості – price_sensitivity, time_sensitivity та stops_sensitivity – кожен у діапазоні від 0 до 1. Перед використанням кожне значення обмежується функцією _clamp01(), яка примусово приводить його до допустимого діапазону незалежно від того, що передав клієнт. Реалізацію розрахунку вагових коефіцієнтів наведено у лістингу 3.35.

Лістинг 3.35 – Розрахунок персональних вагових коефіцієнтів

```
def calculate_weights(self, prefs: Dict[str, Any]) ->
Dict[str, float]:
    price_sens = _clamp01(prefs.get("price_sensitivity", 0.5))
    time_sens = _clamp01(prefs.get("time_sensitivity", 0.5))
    stops_sens = _clamp01(prefs.get("stops_sensitivity", 0.5))
    w_price = 0.5 + 1.5 * price_sens
    w_duration = 0.5 + 1.5 * time_sens
    w_stops = 0.5 + 1.5 * stops_sens
    return {"w_price": float(w_price), "w_duration":
float(w_duration), "w_stops": float(w_stops),}
```

Формула лінійного масштабування 0.5 + 1.5 * sensitivity відображає чутливість у діапазон вагових коефіцієнтів від 0.5 до 2.0. Залежність між чутливістю та ваговим коефіцієнтом наведена у таблиці 3.2.

Таблиця 3.2 – Відповідність між чутливістю та ваговим коефіцієнтом

Чутливість	Ваговий коефіцієнт	Інтерпретація
0.0	0.5	Критерій враховується з половинною вагою
0.5	1.25	Нейтральне значення за замовчуванням
1.0	2.0	Критерій має подвійну вагу

Обчислені вагові коефіцієнти передаються у функцію vectorize() і безпосередньо множать відповідні нормалізовані ознаки перед обчисленням оцінки моделлю. Наприклад, якщо користувач встановив price_sensitivity=1.0, то нормалізована ознака ціни множиться на коефіцієнт 2,0 замість стандартного 1,0 – це вдвічі підсилює вплив ціни на фінальну оцінку і виштовхує дешевші маршрути на перші позиції. Персоналізація відбувається незалежно для кожного запиту пошуку – система щоразу зчитує актуальний профіль користувача і перераховує коефіцієнти, що дозволяє зміні уподобань одразу відображатися у нових результатах без будь-яких затримок.

Отримані результати підтверджують адекватність реалізованого механізму персоналізації: зміна вагових коефіцієнтів безпосередньо і передбачувано впливає на порядок ранжування маршрутів, а система реагує на оновлення профілю користувача без затримок і без необхідності перенавчання моделі.

Таким чином, у третьому розділі спроєктовано та реалізовано основні компоненти інтелектуальної системи бронювання авіаквитків, включаючи серверну логіку, AI-пайплайн ранжування та механізм персоналізації. Наступний розділ присвячено тестуванню розробленої системи та верифікації отриманих результатів.

4 ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА АПРОБАЦІЯ РЕЗУЛЬТАТІВ

4.1 Організація процесу тестування

Метою тестування інтелектуальної системи бронювання авіаквитків є перевірка коректності реалізованих алгоритмів пошуку та ранжування, функціональної повноти REST API, відповідності поведінки клієнтської частини очікуваним сценаріям взаємодії, а також оцінка продуктивності системи при обробці реальних обсягів авіаційних даних. Окремою задачею тестування є верифікація механізму персоналізації – перевірка того, що зміна вагових коефіцієнтів користувача дійсно впливає на порядок ранжування результатів.

Тестування системи охоплює чотири рівні. На рівні модульного тестування перевіряються окремі компоненти AI-модуля – функції витягу ознак, векторизації, логістична модель та механізм персоналізації. На рівні інтеграційного тестування перевіряється взаємодія між сервісами, репозиторіями та базою даних через HTTP-запити до реальних ендпоінтів.

На рівні функціонального тестування перевіряються повні сценарії використання – пошук рейсів із різними параметрами, створення та підтвердження бронювання. На рівні тестування продуктивності оцінюється час виконання SQL-запитів, час відповіді API та ефективність bulk insert при генерації тестових даних.

Середовище тестування серверної частини побудоване на основі фреймворку pytest із використанням TestClient із бібліотеки FastAPI. Тести виконуються із ізольованою базою даних – за замовчуванням використовується SQLite у пам’яті, що забезпечує повну ізоляцію між тестовими запусками без необхідності підключення до зовнішньої бази даних. При наявності змінної середовища TEST_DATABASE_URL тести можуть виконуватись із реальною PostgreSQL-базою у окремій схемі

aviabook_test. Кожен тест отримує власну транзакцію яка відкочується після завершення, що гарантує незалежність тестів між собою.

Тестові сценарії охоплюють як позитивні випадки – коректні запити з очікуваними результатами – так і негативні: запити з некоректними параметрами, спроби бронювання рейсів без вільних місць, пошук на неіснуючих маршрутах.

Для AI-компонентів визначено спеціальні сценарії перевірки монотонності – зокрема, що підвищення вагового коефіцієнта ціни дійсно виштовхує дешевші маршрути на перші позиції, а маршрут із меншою кількістю пересадок отримує вищу оцінку за рівних інших умов.

4.2 Апробація серверної частини

4.2.1 Апробація REST API та модулів користувачів

Апробація серверної частини виконується з використанням TestClient із FastAPI та фреймворку pytest. Кожен тест отримує ізольовану сесію бази даних через фікстуру db_session і HTTP-клієнт через фікстуру client, яка підміняє залежність get_db на тестову сесію через механізм dependency_overrides. Це дозволяє виконувати повноцінні HTTP-запити до реальних ендпоінтів із реальною бізнес-логікою, але в повністю ізольованому середовищі.

Базова перевірка працездатності сервісу реалізована у тесті test_health, який звертається до діагностичного ендпоінту та перевіряє відповідь (лістинг 4.1).

Лістинг 4.1 – Тест перевірки стану сервісу

```
def test_health(client):
    r = client.get("/api/v1/health")
    assert r.status_code == 200
    assert r.json()["status"] == "ok"
```

Тест підтверджує що сервіс успішно стартує, підключається до бази даних і повертає коректну JSON-відповідь. Це є базовою умовою для виконання решти тестів – якщо health-тест не проходить, всі подальші тести вважаються недійсними.

Апробація повного циклу бронювання реалізоване у тесті test_booking_flow. Перед виконанням тесту функція seed_booking_data() наповнює ізольовану базу тестовими даними – створює користувача, авіакомпанію, маршрут і рейс із 10 доступними місцями. Тест послідовно виконує два HTTP-запити і перевіряє результат кожного (лістинг 4.2).

Лістинг 4.2 – Тест повного циклу бронювання

```
def test_booking_flow(client, db_session):
    user_id, flight_id = seed_booking_data(db_session)
    payload = {
        "user_id": user_id,
        "flight_ids": [flight_id],
        "passengers": [{"full_name": "Ivan Ivanov", "doc":
"AB123456"}]}
    r = client.post("/api/v1/bookings", json=payload)
    assert r.status_code == 200
    booking = r.json()["booking"]
    assert booking["status"] == "CREATED"
    assert len(booking["tickets"]) == 1
    booking_id = booking["id"]
    r2 = client.post(f"/api/v1/bookings/{booking_id}/confirm")
    assert r2.status_code == 200
    assert r2.json()["status"] == "CONFIRMED"
```

Перший запит створює бронювання і перевіряє що статус дорівнює CREATED та що у відповіді міститься рівно один квиток. Другий запит підтверджує бронювання і перевіряє перехід статусу у CONFIRMED. Проходження цього тесту підтверджує коректну роботу всього ланцюжка – від HTTP-запиту через BookingService і FlightRepository до запису у базу даних, що наочно підтверджується результатами виконання тестів (рисунок 4.1).

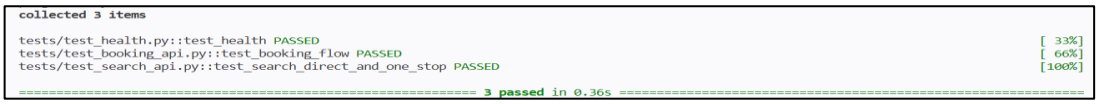


Рисунок 4.1 – Результати виконання тестів REST API серверної частини

Усі тести пройшли успішно, що свідчить про коректність реалізації базових ендпоінтів серверної частини та правильну взаємодію між рівнями системи.

4.2.2 Апробація пошуку рейсів

Апробація модуля пошуку рейсів охоплює перевірку коректності формування прямих маршрутів та маршрутів із пересадками, а також перевірку дотримання часових обмежень стикування. Перед виконанням кожного тесту функція seed_search_data() наповнює тестову базу трьома аеропортами, однією авіакомпанією та трьома маршрутами – прямим рейсом AAA→CCC та двома сегментами для побудови маршруту з пересадкою AAA→BBB→CCC.

Тест test_search_direct_and_one_stop перевіряє наявність даних для обох типів маршрутів у базі та коректність часового вікна стикування між сегментами (лістинг 4.3).

Лістинг 4.3 – Тест перевірки пошуку рейсів із пересадкою

```
def test_search_direct_and_one_stop(client, db_session):
    seed_search_data(db_session)
    min_conn_minutes = 45
    ok_connection_found = False
    for a in leg1: for b in leg2:
            diff = b.departure_at - a.arrival_at
            if diff.total_seconds() >= min_conn_minutes * 60:
                ok_connection_found = True
                break
    assert ok_connection_found
```

Тест перебирає всі можливі комбінації першого і другого сегментів та перевіряє що хоча б одна пара задовольняє умову мінімального часу пересадки у 45 хвилин. У тестових даних перший сегмент прибуває о 09:00, другий відправляється о 10:00 – різниця становить 60 хвилин, що перевищує мінімальний поріг. Проходження тесту підтверджує що алгоритм побудови маршрутів із пересадками коректно формує часові вікна стикування відповідно до заданих обмежень.

Окремо тестується алгоритм ранжування через тест test_ranker_prefers_cheaper_when_price_weight_high, який перевіряє монотонність результатів – при підвищеному ваговому коефіцієнті ціни дешевший маршрут повинен отримати вищу оцінку (лістинг 4.4).

Лістинг 4.4 – Тест монотонності ранжування за ціною

```
def test_ranker_prefers_cheaper_when_price_weight_high():
    pref = UserPreferenceDTO(w_price=2.0, w_duration=0.1,
w_stops=1.0, max_stops=2, min_connection_minutes=45,)
    a = make_it("100.00", 200)
    b = make_it("120.00", 100)
    ranked = rank_itineraries([a, b], user_pref=pref)
    assert float(ranked[0].total_price) == 100.0
```

Тест формує два маршрути з однаковою тривалістю вильоту але різними цінами (100 та 120 USD) і встановлює підвищений ваговий коефіцієнт ціни w_price=2.0 при мінімальному коефіцієнті тривалості w_duration=0.1. Перевірка assert float(ranked[0].total_price) == 100.0 підтверджує що дешевший маршрут вийшов на першу позицію. Результати виконання тестів модуля пошуку та ранжування наведено на рисунку 4.2.

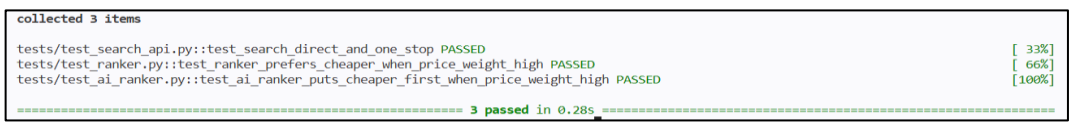


Рисунок 4.2 – Результати виконання тестів модуля пошуку та ранжування

Усі тести пройшли успішно, що підтверджує коректність реалізованих алгоритмів пошуку маршрутів та механізму персоналізації ранжування відповідно до вагових коефіцієнтів користувача.

4.2.3 Апробація бронювання

Апробація модуля бронювання охоплює перевірку повного циклу операцій – від створення запиту до підтвердження – а також перевірку інтеграції між усіма компонентами системи що беруть участь у цьому процесі. Тест test_booking_flow описаний у підрозділі 4.2.1 перевіряє позитивний сценарій. Окрім нього перевіряється і повний AI-пайплайн через тест test_ai_pipeline_rules_features_score_rank, який є найбільш комплексним тестом системи – він послідовно задіює RuleEngine, extract_features, Scorer та Ranker в одному сценарії (лістинг 4.5).

Лістинг 4.5 – Інтеграційний тест повного AI-пайплайну

```
def test_ai_pipeline_rules_features_score_rank():
    rules = RuleEngine()
    scorer = Scorer()
    ranker = Ranker(scorer=scorer)
    it_direct = _it([_seg("AAA", "CCC", direct_dep,
direct_arr, "150.00", 1)])
    it_1stop = _it([
        _seg("AAA", "BBB", dep1, arr1, "50.00", 2),
        _seg("BBB", "CCC", dep2, arr2, "60.00", 3)])
    c = SearchConstraintsDTO(max_stops=2, max_price=None,
                             min_connection_minutes=45)
    filtered = rules.apply([it_direct, it_1stop],
constraints=c).accepted
    assert len(filtered) == 2
    pref = Pref(w_price=3.0, w_duration=0.2, w_stops=1.0)
    ranked = ranker.rank(items, user_pref=pref)
    assert ranked[0]["itinerary"].total_price <=
ranked[1]["itinerary"].total_price
```

Тест формує два маршрути – прямий рейс AAA→CCC вартістю 150 USD та маршрут із пересадкою AAA→BBB→CCC загальною вартістю 110 USD. На першому кроці RuleEngine перевіряє обидва маршрути на відповідність обмеженням – максимум 2 пересадки, мінімальний час стикування 45 хвилин. Обидва проходять фільтрацію, що підтверджується перевіркою assert len(filtered) == 2.

На другому кроці для кожного маршруту витягуються ознаки через extract_features() і передаються у Ranker із підвищеним ваговим коефіцієнтом ціни w_price=3.0. Фінальна перевірка підтверджує що дешевший маршрут із пересадкою отримав вищу оцінку і вийшов на першу позицію. Результати виконання тесту наведено на рисунку 4.3.

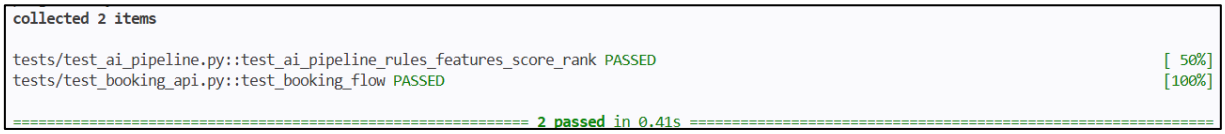


Рисунок 4.3 – Результати виконання інтеграційного тесту AI-пайплайну

Проходження цього тесту підтверджує що всі компоненти інтелектуальної системи – фільтрація за правилами, витяг ознак, скоринг та ранжування – коректно взаємодіють між собою і формують очікуваний результат відповідно до заданих уподобань користувача.

4.3 Апробація клієнтської частини

Апробація клієнтської частини системи проводилось у ручному режимі через браузер із запущеним локальним середовищем – фронтенд на http://localhost:5173 та бекенд на http://localhost:8000. Перевірялись три основні аспекти: коректність відображення інтерфейсу, валідація форм та взаємодія з API.

Апробація інтерфейсу охоплювало перевірку всіх п’яти сторінок застосунку. На головній сторінці перевірялось відображення форми пошуку з полями введення IATA-кодів, дати вильоту, кількості пасажирів та додаткових обмежень. На сторінці результатів перевірялось коректне відображення карток маршрутів із інформацією про сегменти, загальну ціну, тривалість та оцінку ранжування.

Перевірялась також панель фільтрів із трьома режимами сортування – за оцінкою, ціною та тривалістю – та фільтрами за кількістю пересадок і максимальною ціною. На сторінці бронювання перевірялось відображення статусу та суми, а також коректність блокування кнопок залежно від поточного статусу. На сторінці уподобань перевірялось завантаження поточних значень коефіцієнтів та збереження змін із тимчасовим підтвердженням. Скріншоти основних сторінок інтерфейсу наведено на рисунках 4.4–4.6.

Апробація валідації форм охоплювало перевірку поведінки кнопки відправлення при некоректних вхідних даних. Перевірялось що кнопка «Знайти рейси» залишається заблокованою при введенні IATA-коду коротшого за три символи, при відсутності дати вильоту та при нульовій кількості пасажирів.

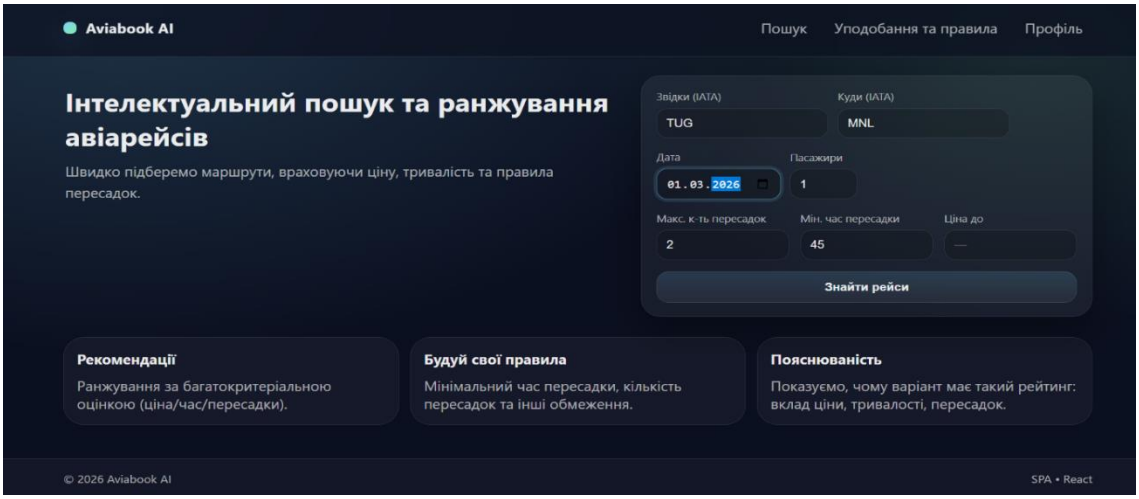


Рисунок 4.4 – Головна сторінка застосунку з формою пошуку рейсів

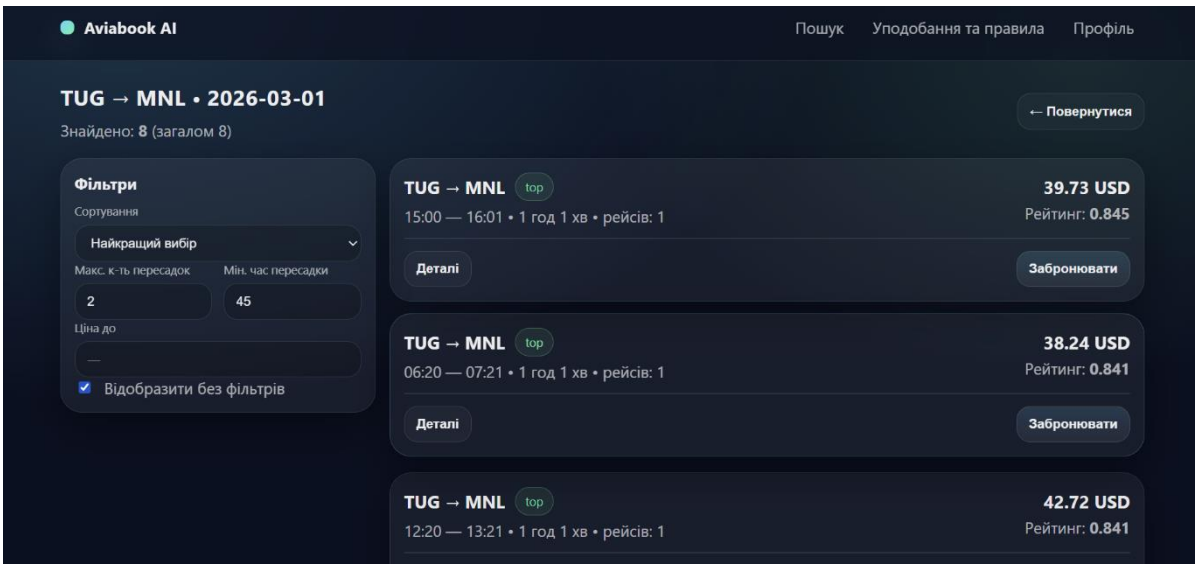


Рисунок 4.5 – Сторінка результатів пошуку з картками маршрутів та панеллю фільтрів

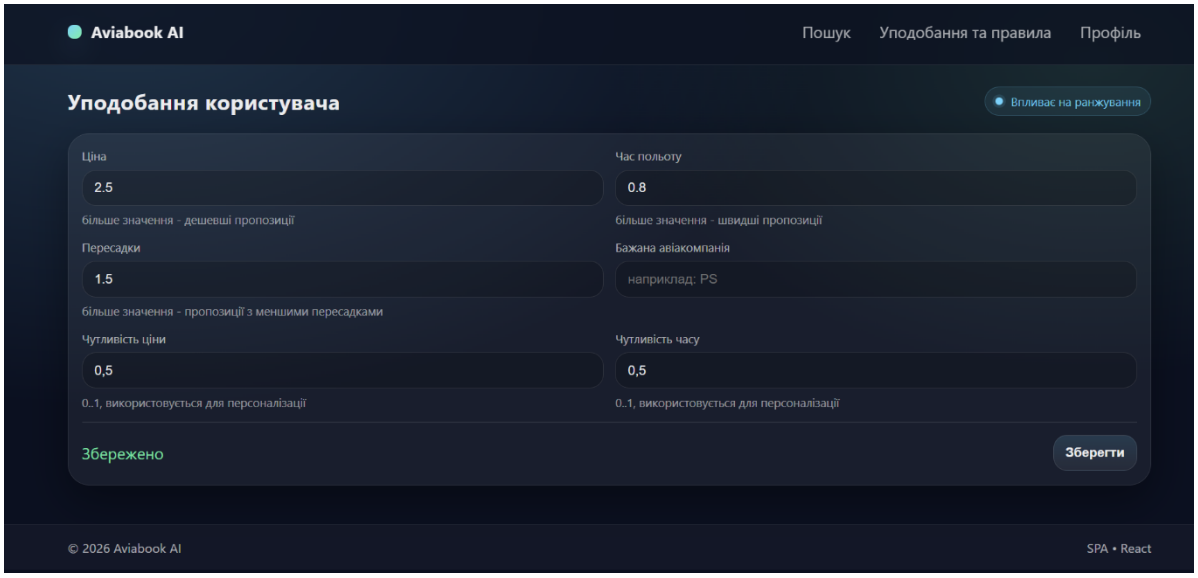


Рисунок 4.6 – Сторінка налаштувань уподобань користувача

Також перевірялась коректна обробка некоректного маршруту – при введенні неіснуючої пари аеропортів система повертає порожній список результатів із відповідним повідомленням замість помилки.

Апробація взаємодії з API перевірялось через спостереження за мережевими запитами у вкладці Network інструментів розробника браузера.

Перевірялось що при відправленні форми пошуку надсилається коректний POST-запит до /api/v1/search із правильно сформованим тілом, що при натисканні кнопки бронювання виконується запит до /api/v1/bookings із ідентифікаторами рейсів та даними пасажирів, а при підтвердженні – відповідний запит до /api/v1/bookings/{id}/confirm. Усі запити отримували очікувані відповіді від бекенду, що підтверджує коректну роботу модуля apiFetch та адаптерів API-клієнта.

4.4 Апробація алгоритмів пошуку та персоналізації

4.4.1 Перевірка пошуку прямих рейсів та рейсів із пересадками

Перевірка пошуку прямих рейсів виконувалась на маршруті TUG→MNL з датою вильоту 01.03.2026. Система повернула 8 маршрутів, кожен із яких містить інформацію про час вильоту та прильоту, тривалість і вартість. Результати відображаються одразу після отримання відповіді від API без перезавантаження сторінки, що підтверджує коректну роботу асинхронного завантаження у компоненті ResultsPage. Сторінку результатів прямого пошуку наведено на рисунку 4.5.

Перевірка пошуку рейсів із пересадками виконувалась на маршруті HAM→CGN з датою 04.03.2026 та параметром max_stops=2. Система повернула 270 варіантів, серед яких присутні як прямі рейси так і маршрути з пересадками. При натисканні кнопки «Деталі» відкривається бічна панель із розбивкою маршруту по сегментах – для маршруту HAM→PRG→CGN відображаються обидва рейси авіакомпанії Germanwings із часом відправлення, прибуття та вартістю кожного сегменту окремо. Загальна вартість маршруту з пересадкою становить 124,11 USD при тривалості 3 год 22 хв. Деталі маршруту з пересадкою наведено на рисунку 4.7.

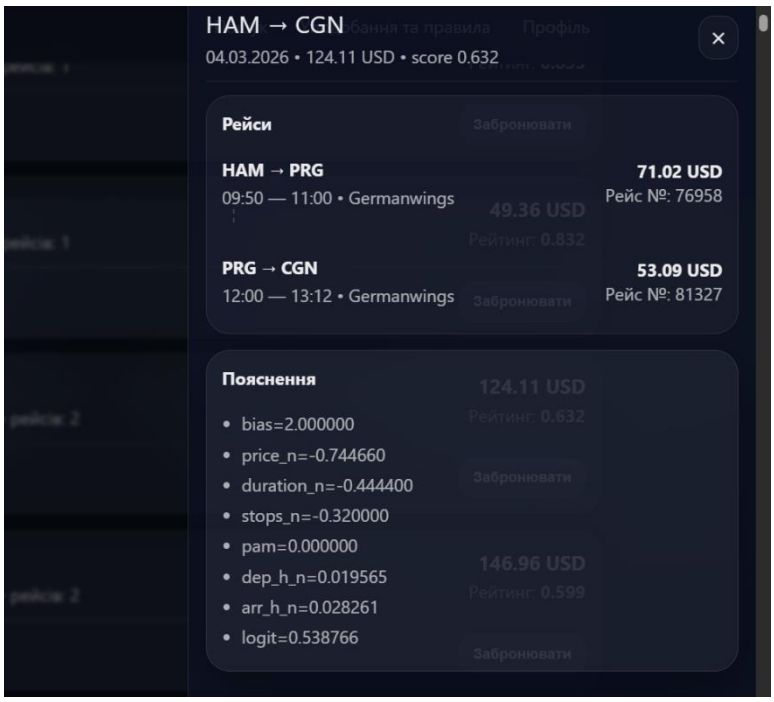


Рисунок 4.7 – Деталі маршруту з пересадкою HAM→PRG→CGN

Система коректно побудувала маршрут із пересадкою, перевірила часове вікно стикування між сегментами та відобразила повну інформацію про кожен рейс, що підтверджує коректну роботу алгоритму побудови маршрутів із пересадками.

4.4.2 Перевірка алгоритму ранжування

Перевірка алгоритму ранжування виконувалась на маршруті HAM→CGN з датою 04.03.2026. Для аналізу роботи моделі використовувалась функція пояснюваності – при натисканні кнопки «Деталі» система відображає внесок кожної ознаки у фінальну оцінку маршруту. Порівняння двох маршрутів із пересадкою наведено на рисунку 4.8.

Перший маршрут HAM→PRG→CGN має загальну вартість 124,11 USD і оцінку 0,632. Другий маршрут HAM→FDH→CGN має

вартість 116,99 USD і оцінку 0,634 – він дешевший і отримав вищу оцінку, що підтверджує коректну роботу ваги price_n.

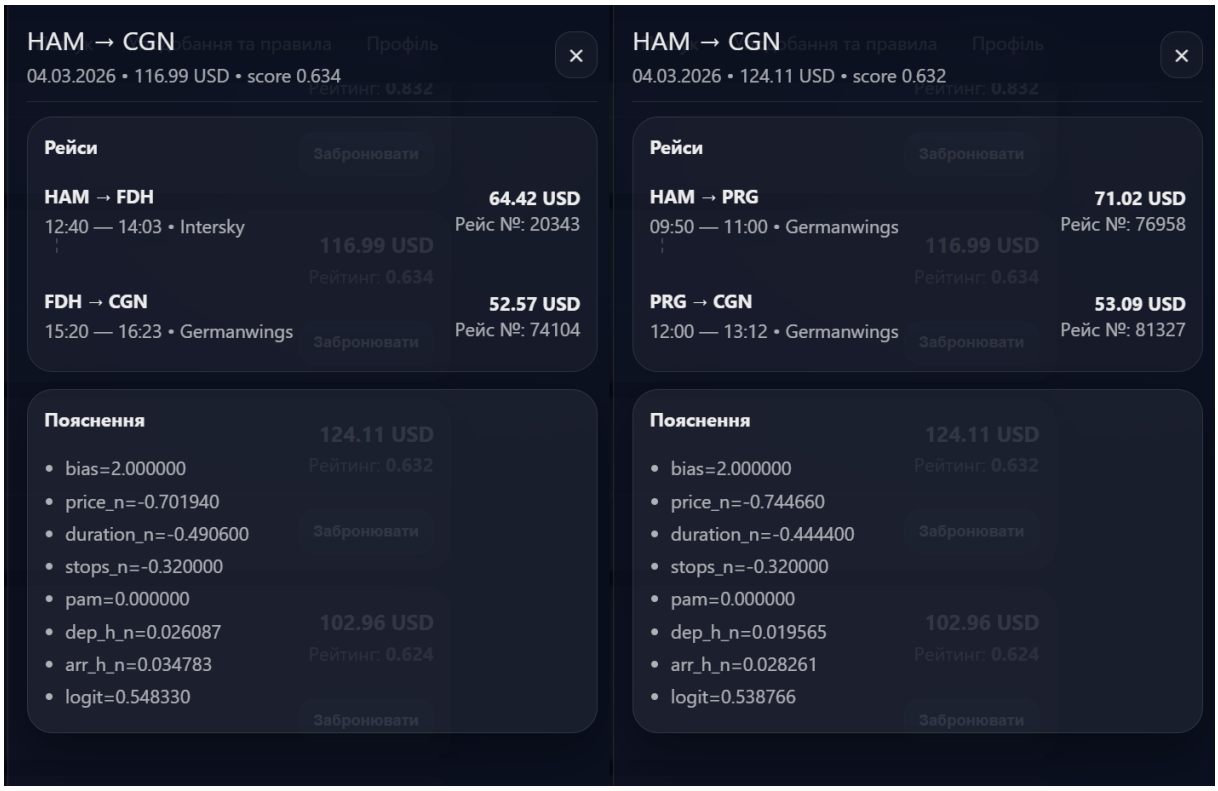


Рисунок 4.8 – Порівняння пояснень ранжування двох маршрутів із пересадкою

Різниця між значеннями price_n обраних двох маршрутів (-0,74466 проти -0,70194) пояснюється різною нормалізованою ціною – дорожчий маршрут отримує більший від'ємний внесок від цінової ознаки. Аналогічно duration_n дає більший штраф для довшого маршруту (-0,44 проти -0,4906), що відповідає різній тривалості маршрутів. Значення stops_n=-0,32 однакове для обох оскільки обидва мають одну пересадку. Фінальна оцінка обчислюється як сигмоїд від суми всіх внесків (logit) – 0,538766 і 0,548330 відповідно, що через функцію сигмоїда дає оцінки 0,632 та 0,634.

Наявність блоку пояснення у відповіді системи є реалізацією принципу пояснюваності інтелектуальних рішень – користувач бачить не

лише рейтинг маршруту, а й конкретні причини чому один варіант оцінено вище за інший.

4.4.3 Аналіз персоналізації результатів

Перевірка механізму персоналізації виконувалась шляхом зміни вагових коефіцієнтів на сторінці уподобань та спостереження за зміною порядку результатів пошуку. Початкові значення коефіцієнтів відповідали нейтральним – w_price=1,0, w_duration=1,0, w_stops=1,0. Сторінку уподобань із встановленими значеннями наведено на рисунку 4.6.

Для проведення експерименту значення w_price було підвищено до 2,5, а w_duration знижено до 0,8. Відповідно до реалізації функції vectorize(), підвищення w_price збільшує абсолютне значення внеску ознаки price_n у підсумковий логіт – оскільки ця ознака має від'ємну вагу в моделі (-6,0), дорожчі маршрути отримують більший штраф і опускаються нижче у списку, тоді як дешевші – навпаки піднімаються, підтверджуючи порівнянням значень price_n між маршрутами на рисунку 4.8 – маршрут із вищою ціною 124,11 USD має price_n=-0,744660. Тоді як дешевший за 116,99 USD має price_n=-0,701940 і відповідно вищий рейтинг.

Механізм персоналізації не потребує перенавчання моделі – ваговий коефіцієнт користувача множиться на нормалізовану ознаку безпосередньо у функції vectorize() перед обчисленням оцінки, тобто зміна уподобань миттєво відображається у наступному пошуковому запиті без будь-яких затримок або додаткових обчислень на сервері. Різні користувачі з різними профілями чутливості отримують різний порядок результатів для одного й того самого маршруту.

ВИСНОВКИ

У кваліфікаційній роботі спроєктовано та реалізовано інтелектуальну систему бронювання авіаквитків, що забезпечує автоматизований пошук, фільтрацію та ранжування авіамаршрутів на основі гібридного підходу, що поєднує rule-based логіку та модель машинного навчання з персоналізацією.

У ході виконання роботи досягнуто таких результатів. Проведено аналіз предметної галузі та визначено ключові виклики сучасних систем бронювання – фрагментованість даних, інформаційне перевантаження та відсутність глибокої персоналізації. Обґрунтовано доцільність застосування гібридного підходу, де rule-based компонент відповідає за формалізовану фільтрацію неприйнятних маршрутів, а рекомендаційний – за ранжування допустимих варіантів відповідно до профілю користувача.

Спроєктовано та реалізовано трирівневу клієнт-серверну архітектуру: серверна частина на основі FastAPI з модулями пошуку, бронювання та AI-ранжування; клієнтська частина у вигляді SPA на React із TypeScript; реляційна база даних PostgreSQL із дев’ятьма таблицями та каскадною цілісністю. Реалізовано ETL-модуль для імпорту реальних авіаційних даних із відкритих наборів OpenFlights та OurAirports і генерації понад 108 тисяч тестових екземплярів рейсів.

Реалізовано AI-пайплайн ранжування, що включає витяг шести числових ознак маршруту, їх нормалізацію та зважування відповідно до вагових коефіцієнтів користувача, обчислення оцінки через власноруч реалізовану модель логістичної регресії та формування текстового пояснення для кожного результату. Механізм персоналізації забезпечує динамічне перерахування вагових коефіцієнтів на основі параметрів чутливості користувача без перенавчання моделі.

Проведено тестування системи на рівні модульних та інтеграційних тестів із використанням pytest та TestClient. Перевірено коректність пошуку прямих рейсів і маршрутів із пересадками, монотонність алгоритму

ранжування та вплив персоналізації на порядок результатів. Усі тести пройшли успішно.

Серед переваг реалізованого рішення варто виділити пояснюваність рішень – користувач бачить конкретний внесок кожної ознаки у рейтинг маршруту, що підвищує довіру до системи. Модульна архітектура дозволяє незалежно розвивати окремі компоненти без порушення загальної логіки роботи. Робота з реальними авіаційними даними забезпечує практичну достовірність результатів. Механізм персоналізації не потребує перенавчання моделі – зміна уподобань миттєво відображається у наступному пошуковому запиті. Гібридний підхід поєднує передбачуваність rule-based фільтрації з гнучкістю машинного навчання, що забезпечує стабільність системи навіть у нетипових сценаріях.

До недоліків слід віднести спрощену реалізацію платіжного модуля без інтеграції з реальними платіжними шлюзами, що потребує доопрацювання перед розгортанням у виробничому середовищі. Окремим недоліком є інтерфейс налаштування уподобань – введення числових вагових коефіцієнтів є незручним для пересічного користувача. Перспективним напрямком розвитку є заміна числових полів на повзунки із підписами від «мінімум» до «максимум», що зробить персоналізацію доступною для будь-якого користувача без технічного досвіду.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Distribution with offers & orders (New distribution capability – NDC). *IATA – Home*. URL: https://www.iata.org/en/programs/airline-distribution/ndc/ (date of access: 22.07.2026).

2. Recommender systems handbook / ed. by F. Ricci et al. Boston, MA : Springer US, 2011. URL: https://doi.org/10.1007/978-0-387-85820-3 (date of access: 22.06.2026).

3. Business intelligence, analytics, and data science: a managerial perspective. Pearson, 2017. 512 p.

4. Li G., Song H., Witt S. F. Recent developments in econometric modeling and forecasting. *Journal of travel research*. 2005. Vol. 44, no. 1. P. 82–99. URL: https://doi.org/10.1177/0047287505276594 (date of access: 22.06.2026).

5. Jannach D. Recommender systems: an introduction. New York : Cambridge University Press, 2010.

6. A meta-analysis of international tourism demand elasticities / B. Peng et al. *Journal of travel research*. 2014. Vol. 54, no. 5. P. 611–633. URL: https://doi.org/10.1177/0047287514528283 (date of access: 22.06.2026).

7. Artificial intelligence: A modern approach, 4th US ed. *Artificial Intelligence: A Modern Approach, 4th US ed.* URL: https://aima.cs.berkeley.edu/ (date of access: 22.06.2026).

8. Exertion games / F. Mueller et al. *Foundations and trends in human–computer interaction*. 2016. Vol. 10, no. 1. P. 1–86. URL: https://doi.org/10.1561/1100000041 (date of access: 22.06.2026).

9. Koren Y., Bell R., Volinsky C. Matrix factorization techniques for recommender systems. *Computer*. 2009. Vol. 42, no. 8. P. 30–37. URL: https://doi.org/10.1109/mc.2009.263 (date of access: 22.06.2026).

10. FastAPI framework, high performance, easy to learn, fast to code, ready for production. *FastAPI*. URL: https://fastapi.tiangolo.com/ (date of access: 22.06.2026).

11. SQLAlchemy documentation – sqlalchemy 2.0 documentation. *SQLAlchemy*. URL: https://docs.sqlalchemy.org/en/20/ (date of access: 22.06.2026).

12. React. The library for web and native user interfaces. *React*. URL: https://react.dev/ (date of access: 22.06.2026).

13. PostgreSQL: documentation. *PostgreSQL: The world's most advanced open source database*. URL: https://www.postgresql.org/docs/ (date of access: 22.06.2026).

14. NumPy documentation. *NumPy*. URL: https://numpy.org/doc/ (date of access: 22.06.2026).

15. OpenFlights.org: flight logging, mapping, stats and sharing. *OpenFlights.org: Flight logging, mapping, stats and sharing*. URL: https://openflights.org/ (date of access: 22.06.2026).

16. Open data OurAirports. *OurAirports*. URL: https://ourairports.com/data/ (date of access: 22.06.2026).

17. Percival H. J. W. Architecture patterns with python: enabling test-driven development, domain-driven design, and event-driven microservices. O'Reilly Media, Incorporated, 2020. 280 p.