НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету	**Завідувач кафедри комп'ютерних наук**
_____ Ігор БОЛБОТ	**_____ Белла ГОЛУБ**
(підпис)	(підпис)
«___» _____ 2026 р.	«___» _____ 2026 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Програмне забезпечення системи моніторингу кліматичних параметрів для планування повсякденних активностей»

Спеціальність «121 Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення»

Гарант освітньої програми
к.т.н., доцент _____ **Ганна ВАЙГАНГ**
(підпис)

Керівник бакалаврської кваліфікаційної роботи
к.т.н., доцент _____ **Ганна ВАЙГАНГ**
(підпис)

Виконав _____ **Павло ПРОКОПЕНКО**
(підпис)

КИЇВ – 2026

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних наук
к.т.н., доцент _____ Белла ГОЛУБ
(підпис)
«___» _____ 2025 р.

ЗАВДАННЯ ДО ВИКОНАННЯ БАКАЛАВРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ ЗДОБУВАЧУ

Прокопенку Павлу Геннадійовичу

Спеціальність «121 Інженерія програмного забезпечення»
Освітньо-професійна програма «Інженерія програмного забезпечення»

Тема бакалаврської кваліфікаційної роботи:
«Програмне забезпечення системи моніторингу кліматичних параметрів для планування повсякденних активностей»
затверджена наказом від «19» грудня 2025 р. № 3204 «С»

Термін подання завершеної роботи на кафедру «_22__» травня 2026 р.

Вихідні дані до роботи: опис програмного забезпечення системи моніторингу кліматичних параметрів; вебзастосунок для перегляду погоди з AI-рекомендаціями; технології React, TypeScript, Node.js, PostgreSQL, Supabase, REST API та OpenWeather API; наукові й методичні джерела з проєктування веборієнтованих інформаційних систем.

Перелік питань що розглядаються: системний аналіз предметної області інформаційної системи; проєктування інформаційного та програмного забезпечення; розробка інформаційного та програмного забезпечення; рекомендації щодо впровадження та експлуатації системи

Перелік графічних документів (за необхідності).

Дата видачі завдання «_22_» ___грудня_____ 2025 р.

Керівник бакалаврської кваліфікаційної роботи _____ **Ганна ВАЙГАНГ**
(підпис)

Завдання взяв до виконання _____ **Павло ПРОКОПЕНКО**
(підпис)

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AI – Artificial Intelligence (штучний інтелект)

Anthropic API – сервіс генерації тексту на основі AI

API – Application Programming Interface (інтерфейс програмування застосунків)

Auth – Authentication (автентифікація)

DB – Database (база даних)

HTTP – HyperText Transfer Protocol (протокол передачі гіпертексту)

HTTPS – HyperText Transfer Protocol Secure (захищений протокол передачі даних)

JSON – JavaScript Object Notation (формат обміну даними)

JWT – JSON Web Token (токен автентифікації)

LLM – Large Language Model (велика мовна модель)

OpenWeather API – сервіс отримання метеоданих

REST – Representational State Transfer (архітектурний стиль взаємодії)

SDK – Software Development Kit (набір засобів розробки)

SQL – Structured Query Language (мова структурованих запитів)

Supabase – хмарна платформа для роботи з БД та автентифікацією

TTL – Time To Live (час життя даних/кешу)

UI – User Interface (інтерфейс користувача)

UML – Unified Modeling Language (уніфікована мова моделювання)

UX – User Experience (користувацький досвід)

ВСТУП

У сучасному інформаційному суспільстві метеорологічна інформація є одним із найбільш затребуваних видів даних для щоденного планування. Людина щодня стикається з необхідністю оцінювати погодні умови: обирати одяг, плануючи прогулянку, визначати маршрут для спортивного тренування, приймати рішення про поїздку або роботу на свіжому повітрі. Водночас більшість наявних погодних застосунків надають лише «сирі» числові показники - температуру, вологість, швидкість вітру - без персональної інтерпретації цих даних для конкретного користувача.

технологічний прогрес у сфері великих мовних моделей (LLM) відкриває принципово нові можливості для контекстуальної інтерпретації погодних даних. Сучасні AI-моделі здатні генерувати зв'язний, змістовний і персоналізований текст, орієнтуючись одночасно на числові метеопоказники, профіль користувача, його спосіб життя і часовий контекст. Такий підхід суттєво підвищує практичну цінність погодного застосунку, перетворюючи його з інформаційної довідки на персонального помічника.

Об'єктом дослідження є процеси отримання, обробки та персоналізованого представлення метеорологічної інформації в веб-середовищі.

Предметом дослідження є методи та засоби розробки веб-застосунку прогнозу погоди з модулем персоналізованих рекомендацій на основі великих мовних моделей та хмарних технологій.

Метою роботи є проєктування й реалізація веб-застосунку Climfy, що забезпечує отримання актуальних метеоданих із зовнішнього API, їх наочне відображення та генерацію персоналізованих рекомендацій засобами Claude AI з урахуванням профілю й способу активності користувача.

Для досягнення поставленої мети в роботі необхідно розв'язати такі завдання:

- проаналізувати предметну область та особливості розробки веб-орієнтованих систем подання метеорологічної інформації;

- дослідити існуючі погодні застосунки та визначити їх переваги й недоліки;
- сформулювати функціональні та нефункціональні вимоги до системи Climfy;
- спроєктувати архітектуру повного стека (full-stack) рішення, включаючи компонентну структуру та модель даних;
- реалізувати клієнтську частину на React 19 з модульною організацією та підтримкою трьох мов;
- реалізувати серверну частину на Hono/Node.js із інтеграцією Supabase та Anthropic Claude API;
- впровадити модуль AI-рекомендацій із кешуванням та детерміністичним fallback-алгоритмом;
- проаналізувати переваги, обмеження та перспективи подальшого розвитку запропонованого рішення.

У роботі застосовано методи системного аналізу, компонентного моделювання, UML-діаграмування, проєктування REST API, розробки реактивних веб-інтерфейсів та методи тестування веб-застосунків.

Структура роботи. Структурні елементи роботи розміщені в такій послідовності.

У першому розділі розглядається предметна область, проведено огляд існуючих рішень та сформульована постановка задачі дослідження.

Другий розділ присвячено моделюванню предметної області. Побудовані діаграми прецедентів використання, послідовності, активності, класів.

Третій розділ присвячено проєктуванню програмної системи. Були розроблені логічна модель даних, фізична модель даних, архітектура системи, вибір та обґрунтування інструментарію.

В четвертому розділі розглянуто впровадження та експлуатація системи, описані вимоги та проведено перевірку якості програмного продукту.

1 СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ІНФОРМАЦІЙНОЇ СИСТЕМИ

1.1 Опис предметної області

Предметна область дослідження охоплює перетин двох сфер: інформаційних систем реального часу та персоналізованого подання даних із залученням генеративного штучного інтелекту. Метеорологічні дані - це динамічна, постійно змінювана інформація, яка має безпосередній вплив на щоденне планування людини. Проте числові значення температури, вологості або швидкості вітру самі по собі є малоінформативними без контекстуальної інтерпретації.

Виникає принципова проблема «останньої милі» подання погодних даних: користувач отримує числа, але не отримує відповідь на практичне питання - «Що мені вдягнути?» або «Чи варто йти бігати?». Ця проблема особливо гостро проявляється для людей з різним способом життя, напирклад: офісного працівника, спортсмена, велосипедиста або туриста. Кожному з них потрібна різна інтерпретація тих самих погодних умов.

Зростання доступності та якості великих мовних моделей (LLM) дає можливість вирішити цю проблему елегантним способом: генерувати змістовний і персоналізований текстовий опис погодної ситуації, враховуючи водночас числові метеопоказники, особисті характеристики користувача (ім'я, вік, тип активності), географічний контекст і час доби. Такий підхід трансформує традиційний погодний застосунок у повноцінного персонального асистента.

Технічна реалізація системи такого класу передбачає поєднання кількох ключових компонентів: зовнішнього API для отримання актуальних метеоданих, реактивного клієнтського інтерфейсу для наочного відображення, серверного рівня для безпечної оркестрації запитів до зовнішніх сервісів, хмарної бази даних для зберігання профілів і персоналізованих результатів, а також інтеграції з AI-API для генерації рекомендацій.

Важливим аспектом предметної області є багатомовність. Погодні застосунки використовуються глобально, і надання рекомендацій рідною мовою користувача є ключовою вимогою зручності. Для даної системи визначено підтримку трьох мов: англійської, німецької та української - що відповідає цільовій аудиторії продукту.

Окремим аспектом є проблема ефективності та вартості AI-запитів. Генерація тексту за допомогою LLM є відносно дорогою операцією, тому пряме звернення до Claude API при кожному запиті користувача є недоцільним. Необхідно реалізувати механізм кешування результатів із обмеженим терміном дії (TTL - time to live), а також детерміністичний алгоритм формування рекомендацій як резервний варіант для неавтентифікованих користувачів або при збоях AI-сервісу.

1.2 Аналіз вимог до програмної системи

Для забезпечення ефективної роботи програмної системи необхідно сформувати сукупність функціональних і нефункціональних вимог, які визначають основні можливості застосунку, особливості взаємодії користувача із системою, а також вимоги до швидкодії, надійності та безпеки. Формування вимог дозволяє визначити межі функціонування програмного забезпечення та забезпечує основу для подальшого проєктування архітектури, структури даних і алгоритмів роботи системи.

З огляду на цілі системи доцільно виокремити такі основні функціональні вимоги, які наведено в табл. 1.1.

Наведені функціональні вимоги демонструють, що система орієнтована не лише на відображення погодних даних, а й на формування персоналізованого користувацького середовища з підтримкою AI-рекомендацій, мультимовності та індивідуальних налаштувань.

Таблиця 1.1

Функціональні вимоги до системи

№	Вимога	Опис вимог
1	Реєстрація та авторизація користувача з підтримкою скидання пароля	Користувач повинен мати змогу зареєструватись, увійти у застосунок та скинути пароль у разі, якщо користувач його забув.
2	Онбординг нового користувача	Програмне забезпечення повинно мати можливість введення певних персональних даних користувача після реєстрації задля надання персоналізованих рекомендацій
3	Відображення поточної погоди за геолокацією або пошуком по назві міста	Програмне забезпечення має надавати метеорологічні дані користувача за його геолокацією (у разі, якщо користувач надає доступ на отримання даних про його місцезнаходження) або за пошуком міста за його назвою.
4	Почасовий графік температури	Для зручного користування застосунком, користувач має мати можливість переглядати графік зміни температури протягом дня.
5	П'ятиденний прогноз погоди	Застосунок повинен надавати користувачу прогноз погоди на наступні 5 днів. Такий формат було обрано базуючись на майбутній цільовій аудиторії продукту - жителі міст, студенти та офісні працівники.
6	Інтернаціоналізація	Програмне забезпечення має підтримувати декілька мов - українську, англійську і німецьку
7	Додавання міст до списку улюблених	Користувач повинен мати змогу додавати міст до улюблених з метою пришвидшення доступу до даних міст, які для нього цікаві.
8	Персоналізована AI-рекомендація	Авторизований користувач повинен мати змогу отримувати персоналізовані рекомендації, базуючись на його персональних даних (вік, тип активності) та теперішніх погодних умов.
8	Одиниці вимірювання температури	Користувач має мати змогу одиницю вимірювання температури - цельсій або фаренгейт.
10	Перемикання теми	Застосунок повинен підтримувати вибір кольорової теми, а саме темна або світла.
11	Кешування AI-рекомендацій	Задля зменшення витрат на API, запити авторизованих користувачів мають проходити процес кешування на термін 1 години.

Особливу увагу приділено інтеграції із зовнішніми сервісами, механізмам кешування та підтримці персоналізації, що визначає систему як сучасний вебзастосунок із елементами інтелектуального аналізу даних.

Для забезпечення стабільної та зручної роботи програмного забезпечення також необхідно врахувати низку нефункціональних вимог, наведених у табл. 1.2.

Таблиця 1.2

Нефункціональні вимоги до системи

№	Вимога	Опис вимог
1	Зручність використання	Інтерфейс повинен бути простим, зрозумілим і не перевантаженим, оскільки система має знижувати, а не підвищувати когнітивне навантаження.
2	Швидкодія	Надання метеоданих повинно відбуватися за прийнятний час, достатній для того, щоб користувач не сприймав застосунок як повільний. Особливо важливими є швидкість отримання персоналізованої рекомендації та пошук міста.
3	Надійність	Система повинна коректно обробляти повторні сповіщення, дублікати, тимчасову недоступність зовнішніх API та інші помилки інтеграції.
4	Безпека	Доступ до персональних даних та даних, які були сформовані під час використання застосунку повинен бути захищений. Система повинна дотримуватися принципу мінімально необхідних прав доступу.
5	Масштабованість	Архітектура повинна допускати майбутнє розширення: створення нового функціоналу, додавання аналітики та одночасну розробку декількох розробників.

У результаті функціонування користувач повинен отримати таке середовище роботи:

- користувач заходить на сайт застосунку та надає доступ до локації.
- метеодані надходять до користувача.
- користувач реєструється в системі.
- користувач бачить підказку про необхідність перевірки пошти для підтвердження створення акаунту.

- користувач проходить онбординг шляхом надання персональних даних.
- користувач бачить згенеровану рекомендацію.

Така постановка задачі визначає систему не як звичайний сайт для перегляду погодних даних, а як повноцінний застосунок, який вирішує щоденні питання на основі контексту користувача та погоди.

1.3 Моделювання предметної області

Для опису взаємодій між користувачем, зовнішніми сервісами та внутрішніми компонентами системи доцільно використати UML-підхід. Моделювання дозволяє формалізувати сценарії використання, показати точки інтеграції та краще описати межі системи.

У запропонованій системі доцільно виділити ряд основних акторів.

Користувач взаємодіє з вебінтерфейсом: здійснює пошук міста, отримує актуальні кліматичні дані та переглядає сформовані рекомендації.

Синоптик виступає умовним експертним актором, який ініціює процес формування рекомендацій на основі отриманих метеорологічних даних.

Датчик надання інформації про погоду є зовнішнім актором, що автоматично постачає до системи первинні кліматичні параметри: температуру повітря, вологість, атмосферний тиск, швидкість вітру, рівень опадів та якість повітря.

Основні прецеденти системи наведено в таблиці 1.3.

Таблиця 1.3

Основні прецеденти системи

№	Прецедент	Опис
1	Пошук міста	Користувач вводить назву населеного пункту для отримання відповідних кліматичних даних.
2	Надання даних про погоду	Зовнішній датчик передає до системи структурований набір метеорологічних параметрів, що є первинним джерелом інформації.
3	Отримання даних про погоду	Система отримує та опрацьовує кліматичні параметри, передані датчиком, і готує їх до відображення або подальшого аналізу.

№	Прецедент	Опис
4	Формування рекомендацій	Синоптик на основі отриманих даних формує рекомендації для користувачів. Цей прецедент є розширенням (*extend*) сценарію «Отримання даних про погоду» - він активується лише після успішного отримання кліматичної інформації.
5	Перегляд рекомендацій	Користувач переглядає актуальні рекомендації, сформовані на підставі поточних кліматичних даних.

У межах діаграми прецедентів система представлена як центральний об'єкт взаємодії. Користувач та синоптик взаємодіють з вебінтерфейсом безпосередньо, тоді як датчик є зовнішнім джерелом, що взаємодіє із серверною частиною. Залежність між прецедентами «Формування рекомендацій» та «Отримання даних про погоду» підкреслює реактивну природу системи.

Для наочного відображення взаємодії користувачів і зовнішніх сервісів із програмною системою доцільно побудувати діаграму прецедентів використання. Дана UML-діаграма дозволяє формалізувати основні сценарії роботи системи, визначити ключових акторів та відобразити логіку взаємодії між ними.

На рис. 1.1 наведено діаграму прецедентів системи моніторингу кліматичних параметрів, яка демонструє основні функції програмного забезпечення та взаємозв’язки між процесами отримання погодних даних і формування рекомендацій.

Як показано на рис. 1.1, центральне місце в системі займає процес отримання та аналізу метеорологічної інформації. Користувач взаємодіє із системою через функції пошуку міста та перегляду рекомендацій, тоді як зовнішній сервіс надання погодних даних забезпечує надходження кліматичних параметрів. Важливим елементом моделі є залежність між процесами отримання погодних даних і формування рекомендацій, що підкреслює реактивний характер роботи системи та використання погодного контексту для генерації персоналізованих порад. Побудована діаграма дозволяє визначити межі функціонування системи та слугує основою для подальшого проєктування її архітектури й алгоритмів роботи.

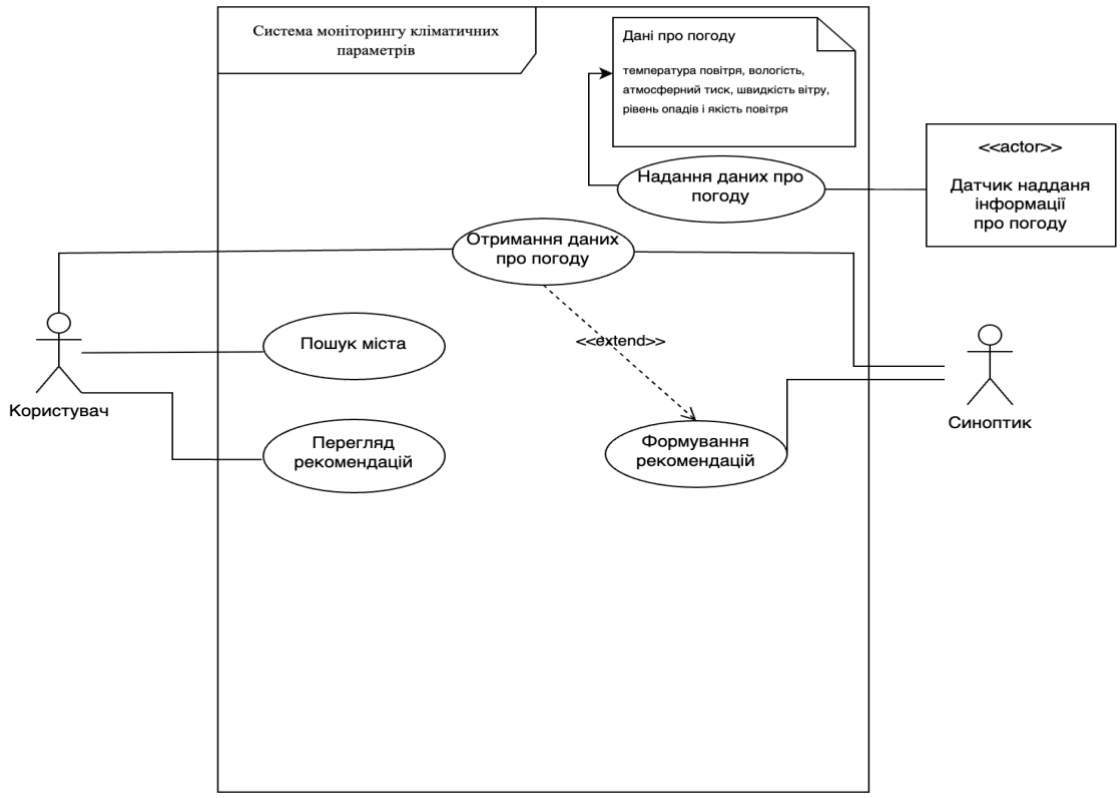

Рис. 1.1 Діаграма прецедентів системи моніторингу кліматичних параметрів

Отже, виконане моделювання предметної області дозволило формалізувати основні процеси функціонування системи, визначити ключових акторів, прецеденти використання та взаємозв’язки між компонентами програмного забезпечення. Побудовані моделі забезпечують цілісне уявлення про логіку роботи системи моніторингу кліматичних параметрів і створюють основу для подальшого проєктування інформаційного та програмного забезпечення, структури даних і алгоритмів реалізації функціоналу застосунку.

1.4 Огляд інформаційних джерел та існуючих рішень

На ринку представлено значну кількість погодних застосунків і веб-сервісів. Аналіз найбільш поширених рішень дозволяє виявити типові недоліки наявних підходів і обґрунтувати доцільність розробки.

Доцільно виділити кілька класів існуючих рішень.

Традиційні погодні сервіси (Weather.com, AccuWeather, Gismeteo). Провідні метеосервіси - Weather.com, AccuWeather, Gismeteo - пропонують детальні числові прогнози з хорошою точністю, але обмежені у персоналізації. Інтерфейс орієнтований на відображення числових показників: температури, вітру, опадів і хмарності. Жоден із цих сервісів не враховує особистий профіль користувача при поданні інформації. Незважаючи на наявність деяких загальних підказок (наприклад, «Візьміть парасольку»), вони є статичними правилами, а не динамічними персоналізованими рекомендаціями.

Мобільні погодні застосунки (Dark Sky, Carrot Weather). Dark Sky (придбана Apple і закрита для сторонніх розробників) відрізнялась точністю прогнозів на мінімальні проміжки часу та лаконічними текстовими підказками. Carrot Weather використовує ігровий підхід із характерними коментарями, але без справжньої персоналізації. Обидва застосунки прив'язані до мобільних платформ і не пропонують веб-версії з профілем користувача та адаптацією рекомендацій до способу життя.

Застосунки з AI-підтримкою (Microsoft Weather, Google Weather). Великі технологічні компанії інтегрують елементи AI у свої погодні сервіси. Microsoft Weather використовує вбудований Copilot для відповідей на погодні питання, Google Weather відображає контекстні підказки. Проте ці рішення є надбудовами над базовим функціоналом і не пропонують глибокої персоналізації: система не знає про спосіб активності користувача, його вік або мовні переваги, а рекомендації генеруються без урахування накопиченої історії взаємодії.

Аналіз наявних рішень дозволяє сформулювати такі принципові висновки.

1. Традиційні метеосервіси надають точні дані, але не персоналізують їх інтерпретацію.
2. Мобільні застосунки орієнтовані на конкретну платформу і не підтримують профіль із характеристиками активності.
3. AI-підтримка наявних продуктів є поверхневою: відсутній зв'язок із профілем користувача та кешування персоналізованих результатів.

4. Жоден із розглянутих сервісів не пропонує повного циклу: реєстрація, онбординг, персоналізована AI-рекомендація з урахуванням способу активності, мови та частих локацій.

Отже, існує потреба у створенні рішення, яка поєднує переваги кількох класів інструментів:

- отримання актуальних погодних даних відповідно до геолокації користувача.
- надання доцільних рекомендацій базуючись на контексті користувача.
- зручний інтерфейс перегляду погоди, редагування налаштувань користувача та пошуку погодних даних.

Саме цей незаповнений простір і є основою для розробки даного програмного забезпечення - відкритого веб-застосунку, що поєднує якісний метеодата-провайдер, реактивний інтерфейс і справжню AI-персоналізацію через Anthropic Claude.

1.5 Постановка завдання

Метою проєктування є створення програмного забезпечення, здатної надавати дані про погоду у зручному вигляді, аналізувати їх, виділяти з них найважливіші аспекти та надавати рекомендації для користувача базуючись на цьому контексті.

Система повинна забезпечувати зручну взаємодію користувача з погодними даними, зменшення часу на відповіді на повсякденні питання шляхом отримання згенерованих рекомендацій, які сформовані на персональному контексті користувача. Водночас вона має залишатися прозорою для користувача: інтерфейс зрозумілий, не перенасичений та зрозумілий для сприйняття.

2 ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Логічна модель даних у вигляді ER-діаграми

Логічна модель системи охоплює п'ять сутностей. В центрі - таблиця автентифікації з ідентифікатором, електронною адресою та датою реєстрації користувача.

Чотири прикладні таблиці посилаються на неї через зовнішній ключ:

- налаштування профілю - ім'я, вік, рівень фізичної активності, мова, одиниця температури, прапорець онбордингу та дата оновлення;
- обрані міста - назва, координати, країна, регіон і дата додавання;
- історія пошуку - назва міста, координати, країна та часова мітка запиту;
- рекомендації - текст поради, мова генерації, знімок погодних даних, термін дії та дата створення.

Профіль прив'язаний до облікового запису за схемою один до одного - він створюється під час онбордингу і надалі лише оновлюється. Обрані міста, пошукові запити й рекомендації можуть накопичуватися без обмежень: один користувач - багато записів.

Автентифікаційні дані відокремлені від прикладних навмисно. Завдяки цьому структуру профілю або формат рекомендацій можна змінювати, не торкаючись логіки входу.

Проєктування логічної моделі даних є одним із ключових етапів розробки програмної системи, оскільки саме структура даних визначає спосіб зберігання, обробки та взаємодії між основними компонентами застосунку. Для формалізації структури бази даних та опису зв’язків між сутностями доцільно використати ER-модель, яка дозволяє наочно представити об’єкти предметної області, їх атрибути та типи взаємозв’язків.

На рис. 2.1 наведено ER-діаграму логічної моделі даних системи моніторингу кліматичних параметрів, що відображає основні сутності, необхідні для реалізації функціоналу застосунку.

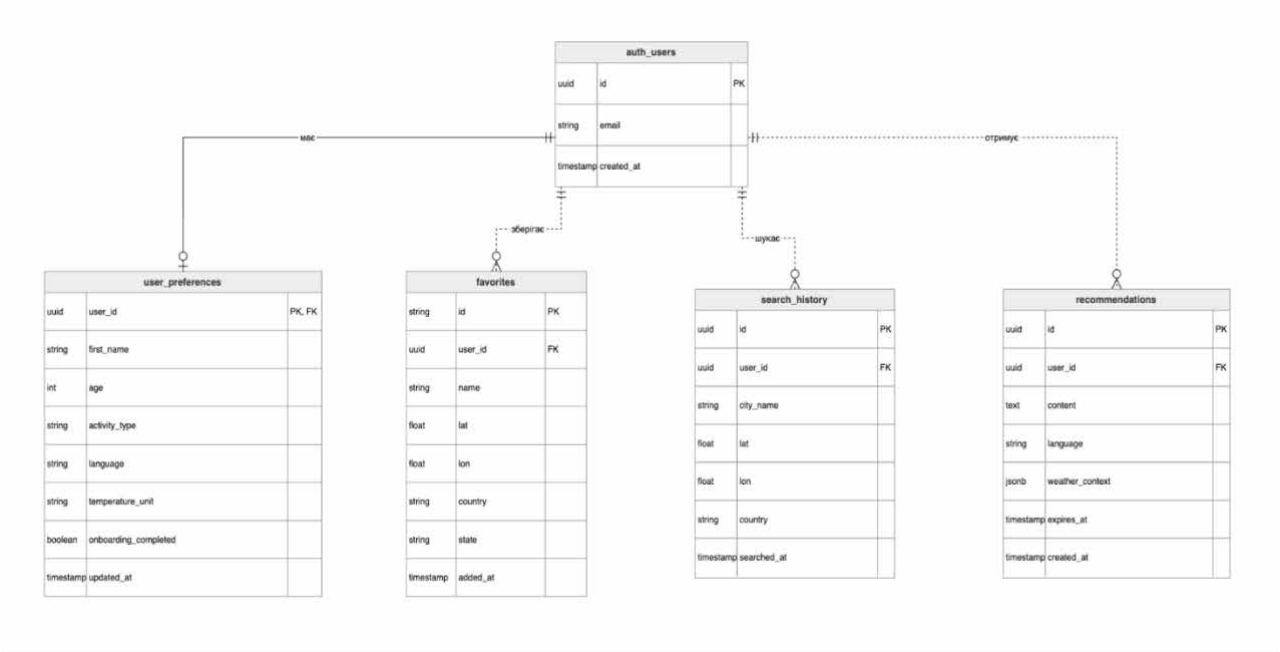

Рис. 2.1 ER-діаграма логічної моделі даних системи

Як показано на рис. 2.1, центральним елементом логічної моделі є користувач та пов’язані з ним сутності налаштувань профілю, історії пошуку, списку обраних міст і згенерованих рекомендацій. Така структура дозволяє забезпечити персоналізацію роботи системи, зберігати результати взаємодії користувача із застосунком та реалізувати механізми кешування AI-рекомендацій. Використання реляційних зв’язків між таблицями забезпечує цілісність даних, а розподіл інформації за окремими сутностями спрощує масштабування системи та подальше розширення функціоналу програмного забезпечення.

Отже, розроблена логічна модель даних забезпечує структуроване зберігання інформації про користувачів, погодні параметри, результати пошуку та персоналізовані рекомендації. Запропонована ER-модель враховує вимоги до цілісності, масштабованості та безпеки даних, а також створює основу для подальшої реалізації фізичної структури бази даних і програмної логіки системи.

2.2 Діаграма класів та кооперації

Діаграма класів фіксує структуру предметної області: п'ять класів із атрибутами, операціями та зв'язками між ними.

Центральний клас – Погода. Він зберігає температуру, вологість, атмосферний тиск, швидкість вітру, рівень опадів, якість повітря, прив'язку до міста та дату спостереження. Єдина операція - оновлення даних. Навколо нього вибудовані решта класів:

- Синоптик - ім'я, кваліфікація, досвід роботи; виконує аналіз погоди та формування рекомендацій;
- Рекомендація - опис, рівень важливості, тип активності;
- Місто - назва, країна, широта та довгота;
- Датчик - індифікатор, тип та місце розташування.

Синоптик аналізує Погоду та формує одну або кілька Рекомендацій. Погода пов'язана з одним Містом і використовує дані від одного або кількох Датчиків.

Три діаграми кооперації показують, як ці класи взаємодіють у конкретних сценаріях.

«Отримання погодних даних» - Місто агрегує об'єкт Погода, Датчик передає виміряні значення через залежність. Результат - заповнений екземпляр класу Погода.

«Аналіз погодних умов» - Синоптик звертається до об'єкту Погода. Напрямок залежності однозначний: ініціатор аналізу - Синоптик, джерело даних - Погода.

«Рекомендації для міста» - усі класи в одній кооперації. Місто агрегує Погоду, Синоптик її аналізує та формує Рекомендацію. Це замкнений цикл від вхідних даних до результату.

Для формалізації структури об’єктів предметної області та взаємозв’язків між ними доцільно використати UML-діаграму класів. Вона дозволяє описати основні сутності системи, їх атрибути, методи та типи взаємодії між об’єктами.

На рис. 2.2 наведено діаграму класів системи моніторингу кліматичних параметрів, яка відображає логічну структуру програмного забезпечення та взаємозв’язки між ключовими компонентами системи.

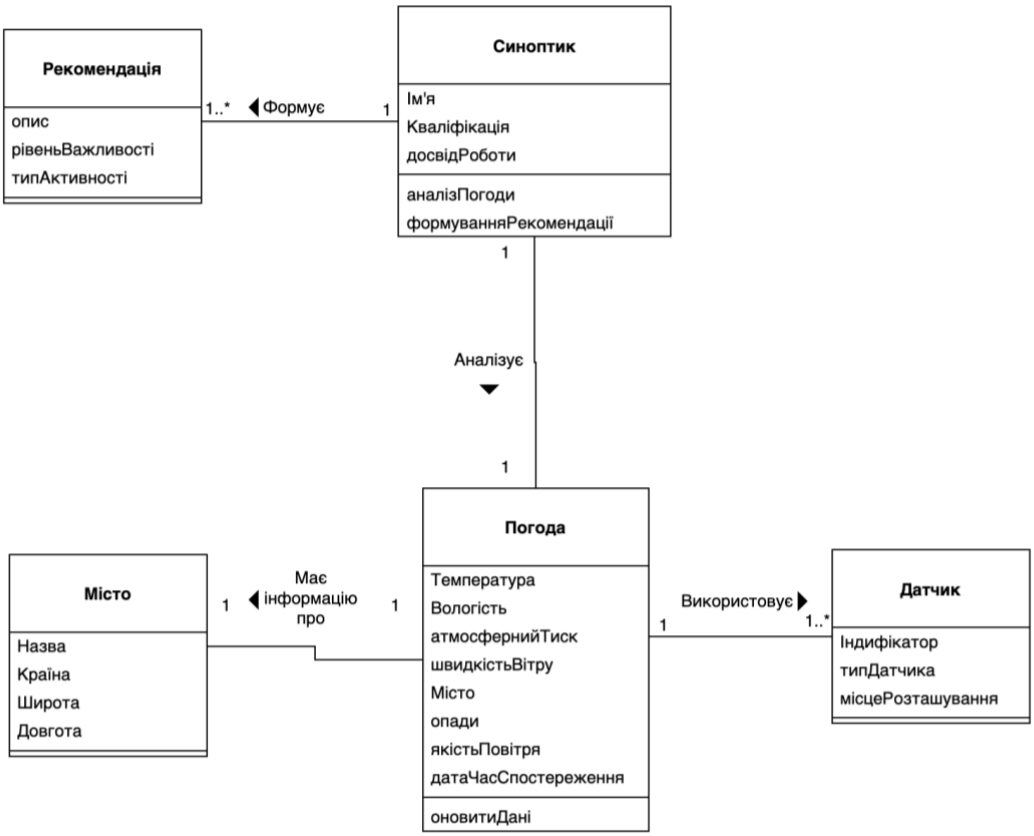

Рис. 2.2 Діаграма класів системи

Як показано на рис. 2.2, центральним елементом моделі є клас «Погода», з яким взаємодіють інші класи системи. Виділення окремих класів для рекомендацій, міста, синоптика та датчиків дозволяє структуровано описати логіку функціонування застосунку та забезпечити розподіл відповідальності між компонентами системи. Побудована діаграма створює основу для подальшої реалізації програмної архітектури та бізнес-логіки застосунку.

Для детальнішого опису процесу отримання кліматичних даних доцільно використати діаграму кооперації, яка відображає послідовність взаємодії між об’єктами системи. На рис. 2.3 наведено кооперацію «Отримання погодних

даних», що демонструє механізм передачі метеорологічної інформації від зовнішнього джерела до внутрішніх компонентів системи.

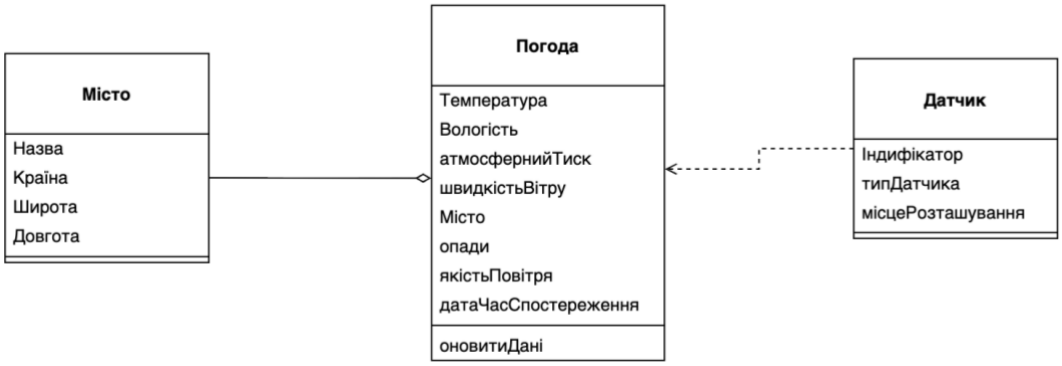

Рис. 2.3 Кооперація «Отримання погодних даних»

Як видно з рис. 2.3, ключову роль у процесі отримання даних відіграє взаємодія між датчиком, містом та об’єктом погоди. Така кооперація дозволяє забезпечити централізоване отримання та обробку кліматичних параметрів, що надалі використовуються для формування рекомендацій та відображення інформації користувачу.

Наступним важливим етапом функціонування системи є аналіз отриманих метеорологічних даних та їх інтерпретація для користувача. Для відображення взаємодії між компонентами, які беруть участь у цьому процесі, побудовано діаграму кооперації «Аналіз погодних умов», наведену на рис. 2.4.

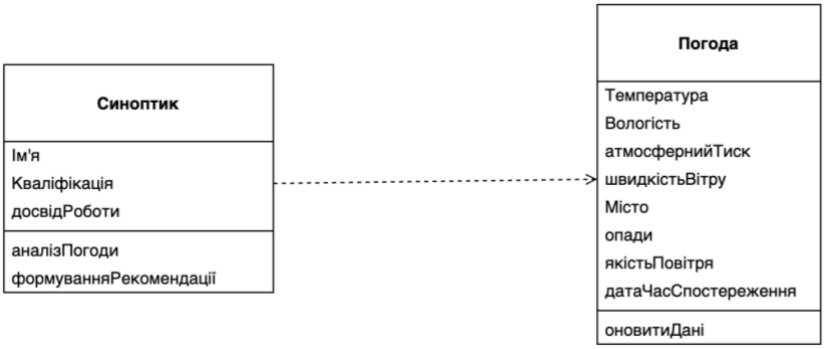

Рис. 2.4 Кооперація «Аналіз погодних умов»

Згідно з рис. 2.4, аналіз погодних умов здійснюється шляхом взаємодії між об’єктом «Синоптик» та класом «Погода». Такий підхід демонструє логіку обробки кліматичних параметрів та формування висновків щодо поточних погодних умов, які використовуються під час генерації рекомендацій для користувача.

Для комплексного відображення процесу формування рекомендацій у системі доцільно розглянути взаємодію між усіма основними об’єктами предметної області. На рис. 2.5 наведено кооперацію «Рекомендації для міста», яка демонструє взаємозв’язки між містом, погодними даними, синоптиком та рекомендаціями.

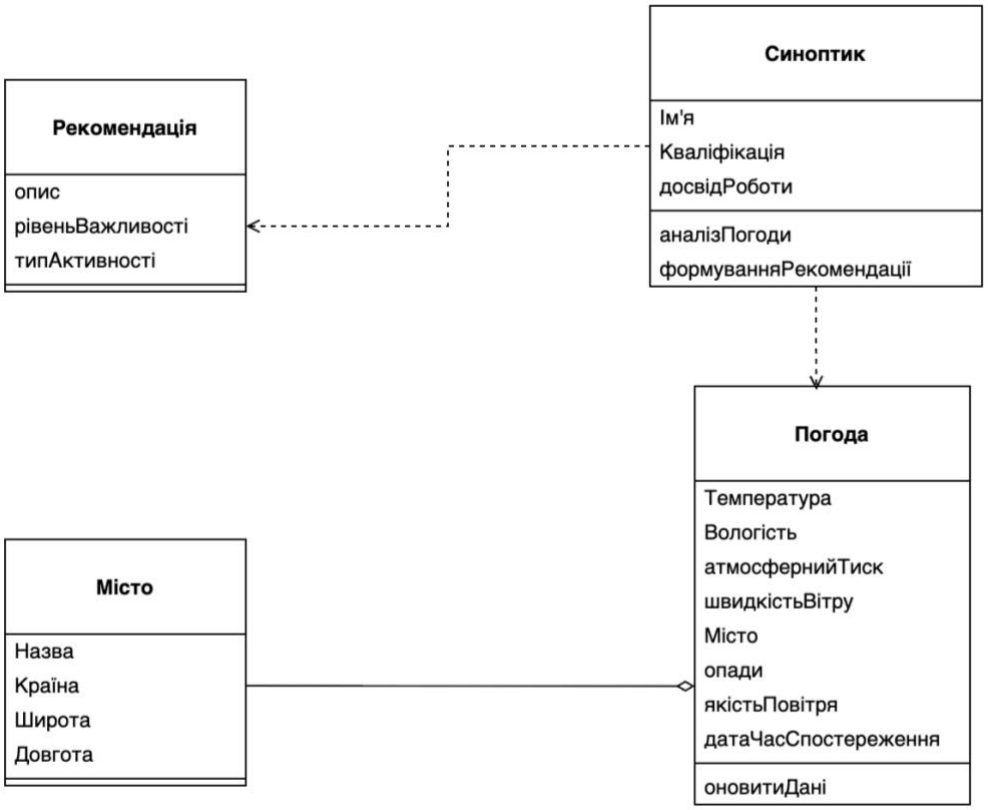

Рис. 2.5 Кооперація «Рекомендації для міста»

Як показано на рис. 2.5, формування рекомендацій є результатом послідовної взаємодії між кількома компонентами системи. Отримані погодні параметри аналізуються, після чого створюються рекомендації, адаптовані до

поточних умов. Така модель кооперації забезпечує цілісність процесу обробки інформації та реалізацію персоналізованого підходу до взаємодії з користувачем.

Отже, побудовані діаграми класів та кооперації дозволили формалізувати структуру програмної системи, визначити основні об’єкти предметної області та описати механізми їх взаємодії. Використання UML-моделювання забезпечує цілісне уявлення про логіку функціонування застосунку, спрощує подальше проєктування архітектури та створює основу для реалізації програмних компонентів системи моніторингу кліматичних параметрів.

2.3 Діаграма пакетів та компонентів

Діаграма пакетів показує логічну структуру кодової бази. Система розділена на два верхньорівневі пакети.

Frontend складається з двох внутрішніх шарів. У features зібрані функціональні модулі: weather, search, auth, favorites та recommendations. У shared - спільна інфраструктура: layout, locales, components, hooks/useSessionStorage, resources та context. Напрямок залежностей однозначний: features використовують shared, але не навпаки. Бекенд компактніший - middleware та routeHandler. Перший перехоплює вхідні запити, другий маршрутизує та виконує логіку.

Зовнішні підсистеми - Sensors, WebAPI зі SessionStorage, PostgreSQL та Claude - підключені до відповідних модулів через залежності.

Для опису логічної організації програмного забезпечення та структури модулів системи доцільно використати UML-діаграму пакетів. Вона дозволяє відобразити розподіл функціональності між окремими частинами застосунку, визначити залежності між модулями та продемонструвати загальну організацію програмної архітектури.

На рис. 2.6 наведено діаграму пакетів системи, яка демонструє структуру клієнтської та серверної частин застосунку, а також їх взаємодію із зовнішніми сервісами та компонентами інфраструктури.

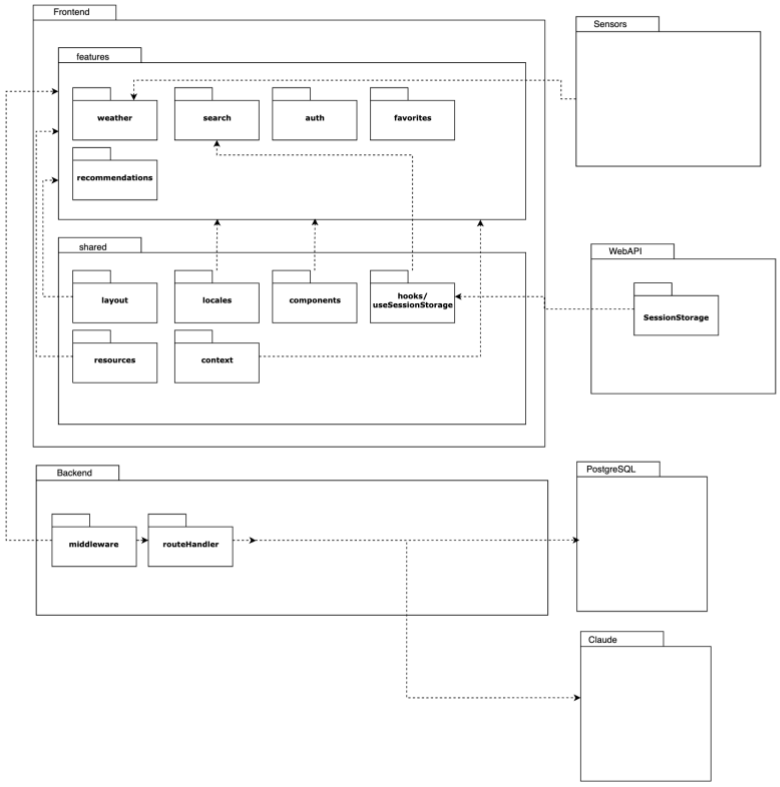

Рис. 2.6 Діаграма пакетів системи

Як показано на рис. 2.6, система побудована за принципом розділення на незалежні функціональні пакети, що забезпечує модульність та спрощує подальший розвиток програмного забезпечення. Виділення окремих пакетів для фронтенду, бекенду, спільних компонентів і зовнішніх сервісів дозволяє чітко розмежувати відповідальність між модулями та забезпечити масштабованість архітектури. Такий підхід також спрощує підтримку коду, інтеграцію нових функціональних можливостей і командну розробку застосунку

В свою чергу, діаграма компонентів деталізує те, що діаграма пакетів лише окреслює: конкретні компоненти, їхні інтерфейси та з'єднання між ними.

Backend містить Auth Middleware, який стоїть перед усіма маршрутами, чотири маршрутизатори - Auth Routes, Preferences Routes, Search History Routes та Favorites Routes - і Supabase Admin Client як єдину точку доступу до бази даних. Окремо виділено Recommendation routes, який делегує роботу двом компонентам: Rule Engine та AI Service.

Frontend побудований із п'яти незалежних можливостей - Favorites, Recommendation, Search, Authentication та Weather. Кожна з них підключена до бекенду через іменований інтерфейс: Favorites cities, Recommendation, Search history, User, User Preferences, Weather and Forecast.

Зовнішні сервіси підключені через два інтерфейси. Supabase Admin API обслуговує Supabase. Claude API - Anthropic. OpenWeather отримує та повертає дані через Weather and Forecast.

Для деталізації внутрішньої структури програмної системи та опису взаємодії між окремими програмними компонентами доцільно використати UML-діаграму компонентів. Така діаграма дозволяє відобразити основні програмні модулі, інтерфейси взаємодії між ними та підключення зовнішніх сервісів, що використовуються у процесі функціонування застосунку.

На рис. 2.7 наведено діаграму компонентів системи моніторингу кліматичних параметрів, яка демонструє взаємозв’язки між фронтенд- та бекенд-компонентами, а також інтеграцію із зовнішніми API та базою даних.

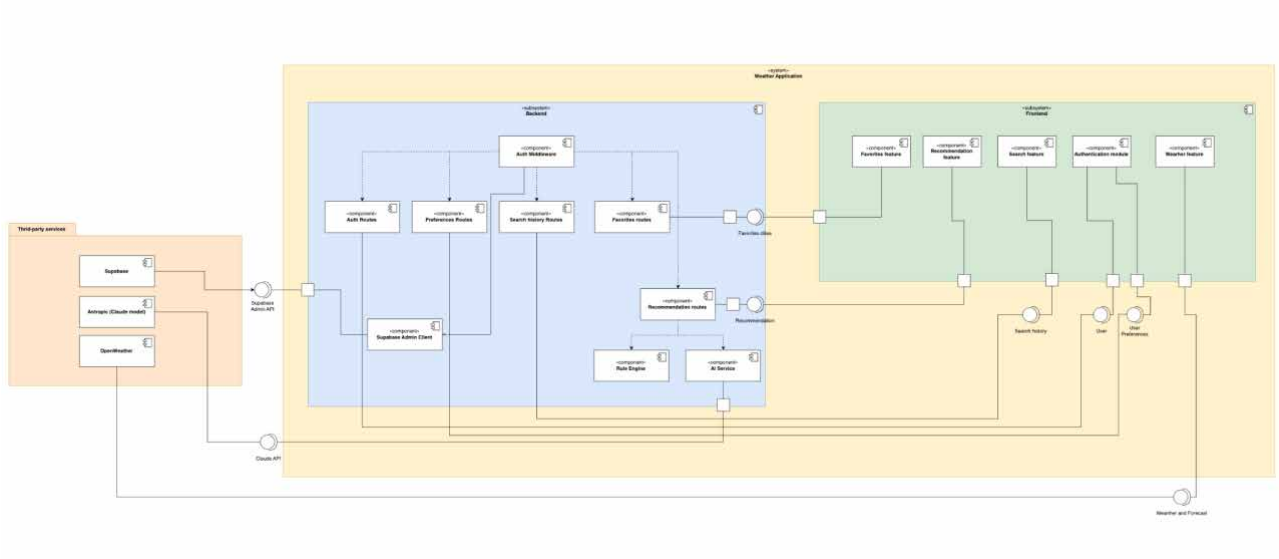

Рис. 2.7 Діаграма компонентів системи

Як показано на рис. 2.7, архітектура системи базується на чіткому розподілі функціональності між компонентами клієнтської та серверної частин. Фронтенд-компоненти відповідають за взаємодію з користувачем та відображення інформації, тоді як бекенд забезпечує обробку запитів, роботу з

базою даних і інтеграцію з OpenWeather та Anthropic API. Використання окремих компонентів для авторизації, рекомендацій, пошуку та роботи з погодними даними підвищує модульність системи, спрощує підтримку коду та забезпечує можливість подальшого масштабування програмного забезпечення.

2.4 Діаграма розгортання

Діаграма розгортання відображає фізичні та логічні вузли, на яких функціонують компоненти системи, а також канали обміну даними між ними. Для реалізованого рішення виокремлено такі вузли::

- клієнтський пристрій користувача (ПК, смартфон тощо) з браузером та клієнтським артефактом Website;
- веб-сервер фронтенду з артефактом Website, що обслуговує HTTP/HTTPS-запити від браузера;
- веб-сервер застосунку з артефактом App, що реалізує бізнес-логіку та взаємодії із зовнішніми сервісами;
- хмарну платформу Supabase із об’єктно-реляційною базою даних PostgreSQL та сервісом аутентифікації Auth Service;
- зовнішній сервіс OpenWeather як джерело метеорологічних даних;
- зовнішній сервіс Anthropic як платформу генерації рекомендацій на основі штучного інтелекту.

Логіка взаємодії між вузлами організована таким чином. Користувач працює через браузер із клієнтською частиною застосунку, яка обмінюється даними з першим веб-сервером за протоколом HTTP/HTTPS. Фронтенд-сервер, у свою чергу, взаємодіє з бекендом через захищений канал HTTPS/REST із авторизацією на основі Bearer JWT-токенів, а з базою даних - через Supabase JS SDK. Бекенд-сервер звертається до OpenWeather API для отримання актуальних кліматичних параметрів, до Anthropic API - для генерації рекомендацій за допомогою мовної моделі, а до Supabase - через Supabase Admin SDK для керування даними та автентифікацією.

Перевагою такої архітектури є чіткий розподіл відповідальності між вузлами. OpenWeather забезпечує надходження метеорологічних даних, Anthropic виконує їх смисловий аналіз та формування рекомендацій, Supabase відповідає за зберігання даних та автентифікацію, а два рівні веб-серверів забезпечують розмежування між клієнтськими запитами та внутрішньою логікою. Такий підхід спрощує масштабування та дозволяє незалежно розвивати окремі компоненти системи.

Для відображення фізичної структури програмної системи та способів взаємодії між її програмними й апаратними компонентами доцільно використати UML-діаграму розгортання. Така діаграма дозволяє описати розміщення програмних модулів на окремих вузлах інфраструктури, канали обміну даними та інтеграцію із зовнішніми сервісами. На рис. 2.8 наведено діаграму розгортання системи моніторингу кліматичних параметрів, яка демонструє взаємодію між клієнтським пристроєм, серверною частиною, базою даних та зовнішніми API-сервісами.

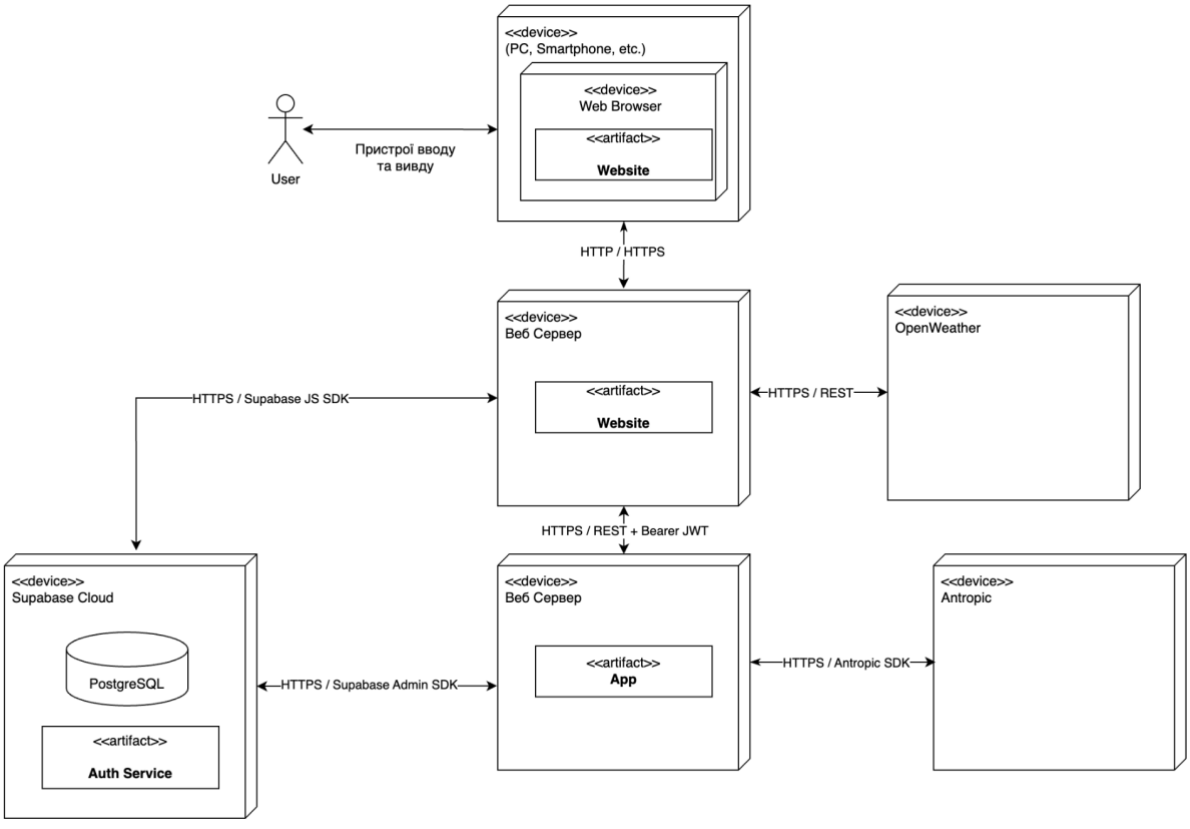

Рис. 2.8 Діаграма розгортання системи

Як показано на рис. 2.8, програмна система реалізована за клієнт-серверною архітектурою з використанням хмарної інфраструктури та зовнішніх сервісів обробки даних. Користувач взаємодіє із вебзастосунком через браузер, тоді як серверна частина забезпечує обробку запитів, доступ до бази даних та інтеграцію з OpenWeather і Anthropic API. Такий підхід дозволяє забезпечити масштабованість, розподіл навантаження та незалежний розвиток окремих компонентів системи, а також спрощує підтримку й подальше розширення функціональних можливостей застосунку.

Отже, побудована діаграма розгортання дозволила визначити фізичну структуру програмної системи, особливості взаємодії між клієнтськими та серверними компонентами, а також способи інтеграції із зовнішніми сервісами. Запропонована архітектура забезпечує гнучкість, масштабованість та можливість ефективного розподілу функціональних навантажень між окремими вузлами системи.

3 РОЗРОБКА ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЗЕПЕЧЕННЯ

3.1 Реалізація системи управління даними

Архітектура запропонованої системи базується на поєднанні сучасного вебстеку та зовнішніх API, кожен з яких вирішує окрему прикладну задачу. Такий підхід дозволяє не будувати інфраструктуру з нуля, а зосередитися на бізнес-логіці.

В основі системи лежить підхід, за якого клієнтська та серверна частини реалізуються як окремі застосунки. Фронтенд побудовано на базі React, що забезпечує динамічний інтерфейс користувача з реактивним оновленням стану. Серверна частина реалізована як самостійний бекенд-застосунок, який відповідає за обробку запитів, інтеграцію із зовнішніми сервісами та бізнес-логіку системи. Такий розподіл є доцільним, оскільки дозволяє незалежно масштабувати клієнтську та серверну частини, а також чітко розмежовує зони відповідальності в межах проєкту.

Supabase виконує роль хмарної платформи з базою даних PostgreSQL та вбудованим сервісом автентифікації. Це дозволяє зберігати профілі користувачів, налаштування, улюблені міста, історію пошуку та згенеровані рекомендації у реляційній моделі. Supabase Auth забезпечує безпечне управління сесіями без необхідності реалізовувати власну систему автентифікації.

OpenWeather API використовується для отримання актуальних метеорологічних даних. Система звертається до API у режимі polling - тобто виконує регулярні HTTP-запити для отримання поточних показників погоди за координатами обраного міста. Отримані дані включають температуру, вологість, атмосферний тиск, швидкість вітру та інші параметри, які використовуються як вхідні дані для подальшого аналізу.

Anthropic API використовується як інтелектуальний модуль генерації персоналізованих рекомендацій. На відміну від автоматичного фонового

процесу, рекомендації формуються за запитом авторизованого користувача. Підставою для запиту є поєднання актуального знімку погодних даних та індивідуальних налаштувань користувача: одиниць вимірювання температури, мови інтерфейсу, віку, типу фізичної активності. Це забезпечує практичну цінність рекомендацій і виключає генерацію нерелевантного контенту.

У межах запропонованої архітектури виділяються чотири логічні шари.

Рівень представлення. Відповідає за інтерфейс користувача: пошук міста, відображення поточних кліматичних параметрів, перегляд рекомендацій, управління списком улюблених міст, сторінку онбордингу та налаштувань профілю.

Прикладний рівень. Реалізує бізнес-логіку: перевірку сесії користувача, формування запитів до зовнішніх API, валідацію вхідних даних, збереження результатів до бази даних та управління станом рекомендацій.

Інтеграційний рівень. Забезпечує взаємодію із зовнішніми сервісами. Саме на цьому рівні виконуються polling-запити до OpenWeather API, звернення до Anthropic API з контекстом погоди та профілю користувача, а також нормалізація отриманих даних перед збереженням.

Рівень даних. Містить реляційну базу даних PostgreSQL для зберігання профілів користувачів, їхніх налаштувань, улюблених міст, історії пошуку та рекомендацій. На цьому ж рівні функціонує сервіс автентифікації Supabase Auth, пов'язаний із системною таблицею *auth.users*.

3.2 Розробка інформаційної бази

Розробка інформаційної бази є важливим етапом реалізації програмної системи, оскільки саме база даних забезпечує зберігання, обробку та цілісність інформації, необхідної для функціонування застосунку. У межах запропонованої системи інформаційна база повинна підтримувати роботу з профілями користувачів, історією пошуку, списком обраних міст та результатами генерації персоналізованих рекомендацій. Проєктування структури бази даних здійснюється з урахуванням вимог до масштабованості, безпеки та ефективної

взаємодії із зовнішніми API-сервісами. Основні сутності інформаційної бази наведено в табл. 3.1.

Таблиця 3.1

Основні сутності

№	Сутність	Функціональний опис
1	user_preferences	Зберігає індивідуальні налаштування авторизованого користувача. Містить посилання на системний запис автентифікації (*user_id*), одиниці вимірювання температури, мову інтерфейсу, ім'я, вік, тип фізичної активності та прапорець завершення онбордингу. Саме ця сутність є ключовим контекстом для персоналізації запитів до Anthropic API.
2	recommendations	Є основною прикладною сутністю системи. Містить координати міста, для якого сформовано рекомендацію, мову відповіді, знімок погодних даних на момент генерації (*weather_snapshot* у форматі JSONB), текстовий зміст рекомендації, прапорець типу генерації (*is_ai*), час створення та час завершення дії рекомендації (*expires_at*).
3	favorites	Зберігає перелік міст, які користувач додав до обраних. Містить назву міста, країну, регіон та географічні координати. Дозволяє швидко отримати погоду для збережених локацій без повторного пошуку.
4	search_history	Фіксує історію пошукових запитів користувача. Структура аналогічна до *favorites*, але записи створюються автоматично при кожному пошуку та мають часову мітку *searched_at*. Це дозволяє відображати нещодавно переглянуті міста та аналізувати поведінку користувача.

Наведені в табл. 3.1 сутності охоплюють основні функціональні аспекти роботи системи та забезпечують збереження персоналізованого контексту користувача. Особливу роль відіграють таблиці налаштувань профілю та рекомендацій, оскільки саме вони формують основу для роботи механізму AI-персоналізації. Запропонована структура бази даних дозволяє забезпечити логічний розподіл інформації між сутностями, підтримати цілісність даних та створити основу для подальшої реалізації серверної логіки застосунку.

Між сутностями встановлюються такі логічні зв'язки:

- один користувач має один запис налаштувань.
- один користувач може мати багато рекомендацій.
- один користувач може мати багато улюблених міст.
- один користувач може мати багато записів в історії пошуку.
- одна інтеграція має один набір облікових даних (credentials).
- кожна рекомендація належить лише одному користувачу.
- кожен запис в обраному належить лише одному користувачу.
- кожен запис в історії пошуку належить лише одному користувачу.
- усі сутності пов'язані з системною таблицею автентифікації *auth.users* через поле *user_id*, що гарантує розмежування даних між користувачами на рівні бази даних.

Ключовим є зв'язок між налаштуваннями користувача та рекомендаціями, оскільки саме налаштування профілю формують персоналізований контекст для генерації рекомендацій і визначають їхню релевантність для конкретного користувача.

Для рекомендацій доцільно розрізняти два стани на основі поля *expires_at:*

- активна - рекомендація діє та відображається користувачу;
- застаріла - термін дії минув, рекомендація більше не є актуальною.

Поле *is_ai* додатково дозволяє відрізняти рекомендації, згенеровані мовною моделлю, від потенційно заздалегідь підготовлених статичних відповідей. Такий підхід є мінімально достатнім для поточної реалізації та може бути розширений у майбутньому.

Проєктні рішення щодо безпеки даних. Оскільки система працює з персональними даними та профілями користувачів, питання безпеки є принциповим. Під час проєктування враховано такі положення:

- автентифікація реалізована через Supabase Auth з управлінням сесіями на рівні платформи.
- доступ до даних кожного користувача розмежований на рівні бази даних через прив'язку до *auth.users.id*.

- знімок погодних даних зберігається у форматі JSONB безпосередньо в записі рекомендації, що виключає необхідність повторних зовнішніх запитів та зменшує поверхню атаки.
- термін дії рекомендацій обмежено полем expires_at, що унеможливлює використання застарілих даних.

Для практичної реалізації інформаційної бази системи було використано хмарну платформу Supabase, яка забезпечує роботу з реляційною базою даних PostgreSQL та підтримує механізми автентифікації користувачів. Побудована модель бази даних відображає структуру таблиць, їх атрибути та взаємозв’язки між сутностями, необхідними для функціонування програмної системи. На рис. 3.1 наведено модель бази даних у СУБД Supabase, яка реалізує зберігання інформації про користувачів, історію пошуку, обрані міста та персоналізовані рекомендації.

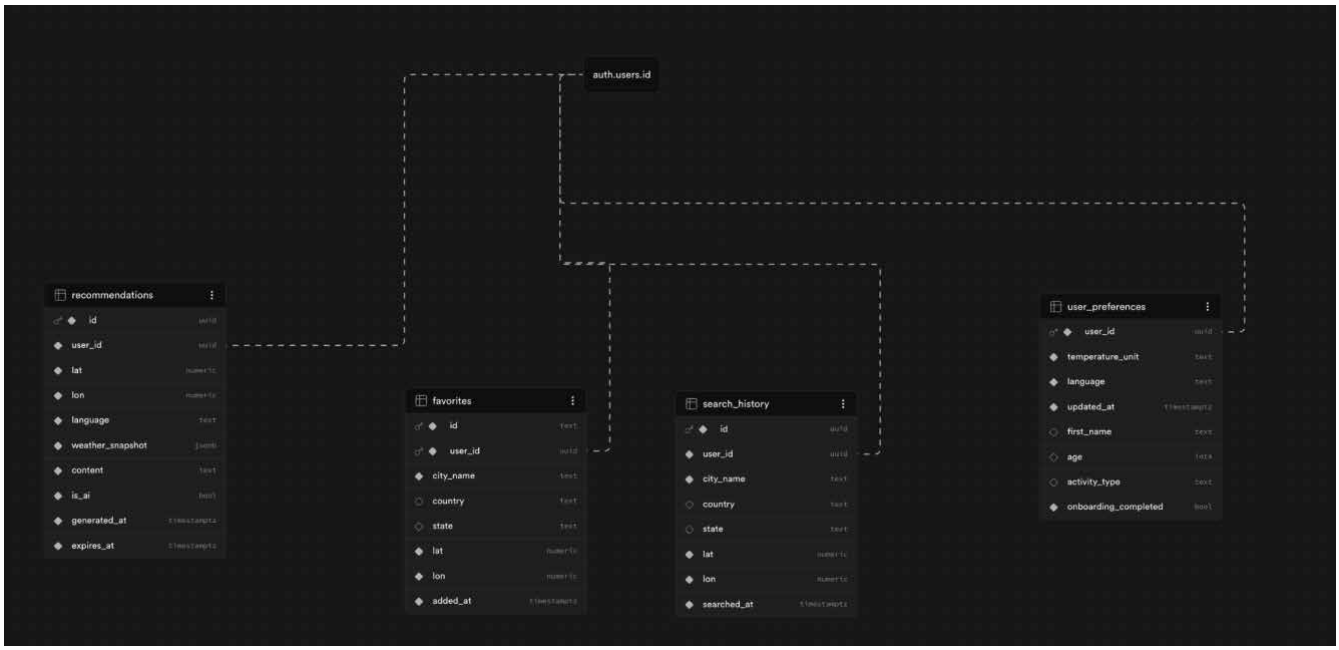

Рис. 3.1 Модель БД у СУБД Supabase

Як показано на рис. 3.1, структура бази даних побудована за принципом централізованого зв’язку всіх прикладних сутностей із таблицею автентифікації користувачів. Такий підхід дозволяє забезпечити розмежування доступу до даних, підтримати персоналізацію роботи системи та організувати зберігання інформації відповідно до логіки функціонування застосунку. Використання

Supabase у поєднанні з PostgreSQL спрощує реалізацію серверної частини системи, забезпечує підтримку механізмів безпеки та створює основу для масштабування програмного забезпечення в майбутньому.

Запропонована архітектура є доцільною з кількох причин. По-перше, розподіл на окремий React-фронтенд та бекенд-сервер відповідає сучасному підходу до побудови вебзастосунків і дозволяє чітко розмежувати зони відповідальності. По-друге, використання Supabase як єдиної платформи для зберігання даних та автентифікації суттєво спрощує інфраструктуру проєкту. По-третє, поєднання OpenWeather API як джерела даних та Anthropic API як аналітичного модуля забезпечує практичну цінність системи: користувач отримує не лише сухі метеорологічні показники, а й персоналізовані рекомендації, адаптовані до його профілю та поточних умов.

3.3 Алгоритм функціонування системи

Алгоритм функціонування системи описує послідовність дій від моменту реєстрації користувача до отримання персоналізованої кліматичної рекомендації. Цей алгоритм є центральною частиною дослідження, оскільки саме він поєднує всі компоненти архітектури в єдиний робочий процес.

Загальна послідовність роботи. У спрощеному вигляді алгоритм можна подати так:

1. Користувач відкриває застосунок та надає дозвіл на доступ до геолокації пристрою.
2. Користувач реєструється в системі та підтверджує електронну пошту.
3. Система ініціює онбординг - користувач заповнює профіль із персональними параметрами.
4. Користувач вводить назву міста у пошуковий рядок або система автоматично визначає локацію.
5. Система надсилає запит до OpenWeather API та отримує актуальні метеорологічні дані.
6. Дані про місто зберігаються в історії пошуку.

7. Система перевіряє наявність актуальної рекомендації для даного міста та користувача.
8. Якщо актуальна рекомендація відсутня або застаріла - формується запит до Anthropic API.
9. До запиту включаються поточні погодні дані та профіль користувача.
10. Система отримує відповідь і зберігає рекомендацію в базі даних.
11. Користувач бачить погодні дані та персоналізовану рекомендацію на одному екрані.
12. Користувач може додати місто до обраних для швидкого доступу в майбутньому.

Етап реєстрації та формування профілю. Перш ніж отримати персоналізовані рекомендації, користувач повинен пройти процедуру реєстрації та налаштування профілю. На першому кроці користувач надає доступ до геолокації, після чого створює обліковий запис через електронну пошту та підтверджує її. Після успішної верифікації система ініціює обов'язковий етап онбордингу.

Під час онбордингу користувач заповнює персональні параметри: ім'я, вік, тип фізичної активності, бажану мову інтерфейсу та одиниці вимірювання температури. Ці дані зберігаються в базі даних і надалі слугують контекстом для генерації рекомендацій. Даний етап є одноразовим, проте користувач може змінити налаштування в будь-який момент через сторінку профілю.

Принципово важливим є те, що без заповненого профілю система не може сформувати релевантну рекомендацію - адже саме вік, тип активності та мова визначають зміст і тон відповіді мовної моделі.

Пошук міста та отримання метеорологічних даних. Отримання погодних даних є доступним для всіх відвідувачів застосунку без необхідності реєстрації чи авторизації. Будь-який користувач може ввести назву міста та переглянути актуальні кліматичні показники. Проте персоналізовані рекомендації на основі профілю доступні виключно авторизованим користувачам із заповненим

профілем - саме авторизація надає системі необхідний контекст для звернення до Anthropic API.

Після завершення онбордингу користувач потрапляє на головний екран із пошуковим рядком. Він вводить назву населеного пункту, після чого система надсилає запит до OpenWeather API. У відповідь надходить структурований набір метеорологічних параметрів: температура, відчутна температура, вологість, атмосферний тиск, швидкість та напрямок вітру, хмарність, опис погодних умов.

Паралельно із відображенням погоди система автоматично зберігає місто в історії пошуку із часовою міткою, що дозволяє формувати список нещодавно переглянутих локацій без додаткових дій з боку користувача. Якщо місто вже присутнє в списку обраних, інтерфейс відповідно це відображає.

Перевірка актуальності рекомендації. До звернення до Anthropic API система виконує перевірку в базі даних: чи існує для поточного користувача та поточного міста рекомендація, термін дії якої ще не минув. Якщо така рекомендація знайдена, вона повертається користувачу безпосередньо з бази даних без зайвого звернення до зовнішнього API.

Такий підхід дозволяє суттєво скоротити кількість запитів до Anthropic API, зменшити затримку відповіді та оптимізувати витрати на використання мовної моделі. Окремо фіксується ознака того, чи була рекомендація згенерована моделлю, чи є статичною підстановкою.

Формування запиту до Anthropic API. Якщо актуальна рекомендація відсутня, система формує запит до Anthropic API. Промпт містить два блоки даних: поточні погодні параметри на момент запиту та профіль користувача. До профільного контексту включаються вік, тип фізичної активності та мова, якою має бути сформована відповідь.

Логіка промпту обмежує модель такими правилами: формулювати конкретні практичні рекомендації; враховувати фізичні особливості та рівень активності користувача; не генерувати загальні фрази без прив'язки до реальних погодних умов; відповідати мовою, вказаною в профілі користувача.

Збереження результату та відображення користувачу. Після отримання відповіді від Anthropic API система зберігає рекомендацію в базі даних. Запис містить інформацію про користувача, місто, мову відповіді, знімок погодних даних на момент генерації, текстовий зміст рекомендації, ознаку автоматичної генерації, час створення та час завершення дії.

Після успішного збереження інтерфейс відображає рекомендацію безпосередньо під блоком погодних даних. Користувач в одному перегляді бачить актуальні метеорологічні показники та персоналізовану пораду щодо одягу, фізичної активності або інших аспектів, що залежать від поточних кліматичних умов.

Обробка винятків і нестандартних ситуацій. У реальній роботі система повинна враховувати типові проблемні сценарії: тимчасова недоступність OpenWeather API; повернення некоректної або неповної відповіді від Anthropic; відсутність заповненого профілю користувача; введення назви міста, яке не розпізнається зовнішнім сервісом; перевищення лімітів зовнішніх API.

У разі недоступності OpenWeather API система повідомляє користувача про неможливість отримати дані та не ініціює запит до Anthropic - оскільки без актуального знімку погоди рекомендація позбавлена практичного змісту. У разі помилки на боці Anthropic API рекомендація не зберігається, а користувач отримує відповідне повідомлення без порушення загального потоку роботи застосунку.

Отже, алгоритм функціонування запропонованої системи є не просто лінійною послідовністю викликів API. Це інтегрований процес, у якому поєднуються регулярне отримання метеорологічних даних, формування персоналізованого контексту на основі профілю користувача, інтелектуальна генерація рекомендацій та ефективне кешування результатів для мінімізації зовнішніх звернень.

3.4 Реалізація інтеграції з OpenWeather API

Одним із ключових практичних компонентів системи є інтеграція з OpenWeather API. Саме вона забезпечує надходження актуальних метеорологічних даних, без яких неможливе як відображення погодних показників, так і подальша генерація персоналізованих рекомендацій. Реалізація цього компонента охоплює кілька підсистем: пошук міста за назвою, отримання поточної погоди та прогнозу, а також визначення назви місця за географічними координатами.

Архітектура модуля роботи з погодним API. Для взаємодії з OpenWeather API реалізовано окремий клас, який інкапсулює всю логіку формування запитів та обробки відповідей. Такий підхід дозволяє централізувати конфігурацію - базову адресу сервісу, ключ доступу та параметри за замовчуванням - і уникнути дублювання коду в різних частинах застосунку.

Клас містить два допоміжні приватні методи. Перший відповідає за формування повної URL-адреси запиту: він приймає назву ендпоінту та набір параметрів, додає ключ API і повертає готовий рядок запиту. Другий виконує безпосередньо HTTP-запит, перевіряє статус відповіді та повертає розібрані дані у вигляді типізованого об'єкту. У разі неуспішної відповіді від сервера генерується виняток із відповідним повідомленням. Завдяки такому розподілу публічні методи класу залишаються лаконічними та зосередженими виключно на бізнес-логіці.

Реалізація інтеграції з OpenWeather API передбачає створення окремого програмного модуля, відповідального за формування HTTP-запитів, обробку відповідей сервера та централізовану взаємодію із зовнішнім метеорологічним сервісом. Використання окремого класу для роботи з API дозволяє ізолювати логіку обробки погодних даних та забезпечити повторне використання коду в різних компонентах системи. На рис. 3.2 наведено фрагмент коду класу WeatherAPI з методами формування запиту до OpenWeather API.

```
async getCurrentWeather({ lat, lon }: Coordinates): Promise<WeatherData> {
  const url = this.createUrl(`${API_CONFIG.BASE_URL}/weather`, {
    lat: lat.toString(),
    lon: lon.toString(),
    units: API_CONFIG.DEFAULT_PARAMS.units,
  })

  return this.fetchData<WeatherData>(url)
}
async getForecast({ lat, lon }: Coordinates): Promise<ForecastData> {
  const url = this.createUrl(`${API_CONFIG.BASE_URL}/forecast`, {
    lat: lat.toString(),
    lon: lon.toString(),
    units: API_CONFIG.DEFAULT_PARAMS.units,
  })

  return this.fetchData<ForecastData>(url)
}
async reverseGeocode({
  lat,
  lon,
}: Coordinates): Promise<GeocodingResponse[]> {
  const url = this.createUrl(`${API_CONFIG.GEO}/reverse`, {
    lat: lat.toString(),
    lon: lon.toString(),
    limit: 1,
  })

  return this.fetchData<GeocodingResponse[]>(url)
}

async searchLocations(query: string): Promise<GeocodingResponse[]> {
  const url = this.createUrl(`${API_CONFIG.GEO}/direct`, {
    q: query,
    limit: '5',
  })
  return this.fetchData<GeocodingResponse[]>(url)
}
```

Рис. 3.2 Фрагмент коду класу WeatherAPI з методами формування запиту

Як показано на рис. 3.2, реалізований клас інкапсулює логіку побудови URL-запитів та обробки відповідей від зовнішнього сервісу. Такий підхід дозволяє централізувати налаштування API, спростити підтримку програмного коду та забезпечити уніфікований механізм взаємодії із сервісом OpenWeather. Крім того, використання окремих методів для формування та виконання запитів підвищує модульність системи та спрощує подальше розширення функціоналу інтеграції.

Пошук міста за назвою. Для реалізації текстового пошуку міста використовується геокодинг - процес перетворення текстового запиту користувача на географічні координати. Відповідний метод надсилає запит до

геокодингового ендпоінту OpenWeather та повертає список із до п'яти найбільш відповідних результатів. Кожен результат містить назву міста, країну, регіон та координати.

Повернення кількох варіантів є принциповим рішенням: воно дозволяє користувачу обрати потрібне місто зі списку у випадку, коли однакові назви зустрічаються в різних країнах. Наприклад, запит «Львів» може повернути однойменні населені пункти з різних регіонів. Вибрані координати передаються в наступні запити для отримання погоди.

Отримання поточних погодних даних. Після того як координати міста визначено, система надсилає запит до ендпоінту поточної погоди. Запит формується на основі широти та довготи обраного міста, а одиниці вимірювання задаються глобальним параметром конфігурації. У відповідь система отримує структурований об'єкт із повним набором метеорологічних показників.

На інтерфейсі користувачу відображаються чотири ключові параметри: поточна температура, відчутна температура з урахуванням вітру та вологості, відсоток вологості повітря та іконка, що візуально відображає загальний стан погоди. Такий набір є достатнім для швидкої оцінки умов та слугує вхідними даними для генерації рекомендацій.

Отримання прогнозу погоди. Окрім поточних даних, система реалізує отримання прогнозу погоди. Відповідний метод звертається до прогнозного ендпоінту з тими самими координатами та повертає набір метеорологічних показників на найближчі дні з кроком у кілька годин. Ці дані дозволяють відображати короткостроковий прогноз в інтерфейсі та можуть використовуватися для розширення контексту рекомендацій у майбутніх версіях системи.

Зворотне геокодування. Для підтримки сценарію автоматичного визначення місця за геолокацією пристрою реалізовано метод зворотного геокодування. Він приймає координати, отримані від браузера користувача, та повертає назву відповідного населеного пункту і країну. Результат обмежується одним найбільш точним варіантом.

Це дозволяє системі автоматично показати погоду для поточного місцезнаходження користувача без необхідності вручну вводити назву міста. Після визначення назви система діє за тим самим сценарієм, що і при текстовому пошуку.

Обробка помилок інтеграції. У процесі реалізації інтеграції з OpenWeather API необхідно враховувати низку типових проблемних ситуацій:

- тимчасова недоступність зовнішнього сервісу;
- введення назви міста, яке не розпізнається геокодером;
- некоректна або неповна відповідь від API;
- перевищення ліміту запитів при інтенсивному використанні.

У реалізованому модулі будь-яка неуспішна HTTP-відповідь призводить до генерації виключення з відповідним повідомленням. Це дозволяє обробляти помилки централізовано на рівні компонентів інтерфейсу та відображати користувачу зрозуміле повідомлення без порушення загального потоку роботи застосунку. У разі недоступності OpenWeather API система не ініціює запит до Anthropic, оскільки без актуального знімку погодних даних генерація рекомендації позбавлена практичного змісту.

3.5 Реалізація генерації рекомендацій через Anthropic API

Генерація персоналізованих рекомендацій є центральним і найбільш технічно складним компонентом системи. Саме цей модуль відповідає за перетворення сухих метеорологічних показників на конкретні практичні поради, адаптовані до профілю конкретного користувача. Реалізація охоплює формування контекстного промпту, звернення до мовної моделі, обробку відповіді та збереження результату в базі даних.

Вибір моделі та конфігурація клієнта. Для генерації рекомендацій використовується модель Claude Haiku через офіційний Anthropic TypeScript SDK. Вибір саме цієї моделі обумовлений балансом між якістю відповідей та швидкістю генерації: Claude Haiku забезпечує зв'язний і природній текст при

мінімальній затримці відповіді, що є критичним для інтерактивного вебзастосунку.

Клієнт ініціалізується з ключем доступу, що зчитується зі змінних середовища і ніколи не передається на клієнтську сторону. Запит налаштовується з обмеженням у 500 токенів на відповідь та температурою 0.7, що забезпечує достатню варіативність формулювань без втрати змістовності та точності.

Для реалізації механізму генерації персоналізованих рекомендацій у системі використовується інтеграція з Anthropic API та мовною моделлю Claude Haiku. Серверна частина застосунку виконує ініціалізацію клієнта API, формує параметри запиту та забезпечує передачу контекстної інформації до моделі штучного інтелекту. На рис. 3.3 наведено фрагмент коду ініціалізації Anthropic-клієнта та виклику моделі для генерації рекомендацій.

```
const client = new Anthropic({
  apiKey: process.env.ANTHROPIC_API_KEY,
})

const message = await client.messages.create({
  model: 'claude-haiku-4-5-20251001',
  max_tokens: 500,
  temperature: 0.7,
  messages: [{ role: 'user', content: prompt }],
})
```

Рис. 3.3 Фрагмент коду ініціалізації Anthropic клієнта та виклику моделі

Як показано на рис. 3.3, ініціалізація клієнта здійснюється з використанням ключа доступу, що зберігається у змінних середовища, а виклик моделі виконується через офіційний SDK Anthropic. Такий підхід забезпечує безпечну інтеграцію із зовнішнім AI-сервісом, централізоване управління параметрами генерації та можливість подальшого масштабування функціоналу інтелектуальних рекомендацій у системі.

Визначення контекстних параметрів. Перед формуванням промпту система автоматично визначає два додаткові контекстні параметри, які суттєво підвищують релевантність рекомендації: пору року та час доби.

Пора року визначається на основі поточного місяця: березень–травень відповідає весні, червень–серпень літу, вересень–листопад осені, решта місяців - зимі відповідно. Час доби ділиться на чотири проміжки: ранок, день, вечір та ніч. Ці параметри передаються до промпту у текстовому вигляді та дозволяють моделі формулювати рекомендації з урахуванням сезонної специфіки та актуального часового контексту - наприклад, порадити взяти парасольку ввечері перед дощем або вийти на пробіжку вранці при комфортній температурі.

Формування персоналізованого промпту. Промпт є ключовим елементом усього модуля, оскільки саме він визначає якість, тон та релевантність згенерованої рекомендації. Він формується динамічно на основі двох блоків даних.

Перший блок містить поточні погодні умови: температуру, відчутну температуру, текстовий опис стану погоди, вологість, швидкість вітру, а також визначені автоматично пору року та час доби. Другий блок містить профіль користувача: ім'я, вік, тип фізичної активності та список міст, погоду в яких користувач переглядає найчастіше.

Тип фізичної активності передається у розгорнутому текстовому описі: наприклад, значення *active* перетворюється на active (gym, hiking), а *athletic* - на athletic (sports, intensive training). Це дозволяє моделі точніше інтерпретувати рівень активності без необхідності самостійно розшифровувати технічні коди.

Промпт містить чіткі вимоги до формату відповіді: починати з імені користувача, написати не менше трьох повних речень, уникати будь-якого Markdown-форматування, прив'язувати поради до конкретного стилю життя та місцевого контексту. Відповідь генерується мовою, вказаною в профілі користувача - підтримуються українська, англійська та німецька.

Результатом роботи інтеграції з Anthropic API є формування персоналізованої текстової рекомендації, адаптованої до погодних умов та

профілю користувача. Згенерований текст враховує температуру, вологість, швидкість вітру, час доби, сезон та рівень фізичної активності користувача, що дозволяє забезпечити контекстну взаємодію із системою. На рис. 3.4 наведено приклад згенерованої рекомендації, сформованої мовною моделлю Claude Haiku.

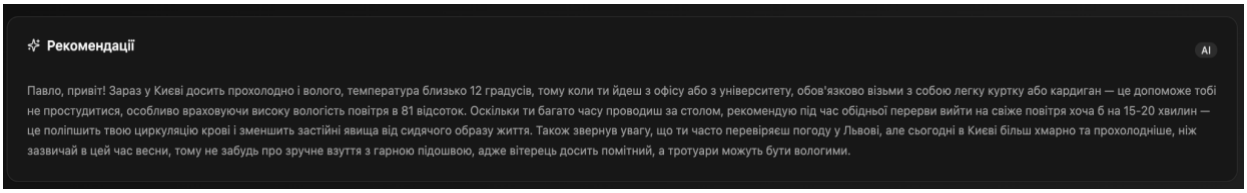

Рис. 3.4 Приклад згенерованої рекомендації

Як показано на рис. 3.4, рекомендація формується у вигляді зв’язного тексту природною мовою та містить практичні поради щодо повсякденних активностей відповідно до поточних погодних умов. Такий підхід підвищує інформативність системи та забезпечує більш зручне сприйняття кліматичних даних користувачем порівняно зі звичайним відображенням числових показників погоди.

Обробка відповіді моделі. Після отримання відповіді від Anthropic API система перевіряє коректність результату. Насамперед перевіряється тип першого блоку відповіді - очікується текстовий вміст. Якщо модель повернула блок іншого типу або порожній рядок, генерується виняток, що запобігає збереженню некоректних даних до бази даних.

У разі успішної відповіді текст очищується від зайвих пробілів та передається для збереження. Такий підхід відокремлює логіку звернення до зовнішнього API від логіки збереження даних та дозволяє незалежно обробляти помилки на кожному з цих етапів.

Кешування рекомендацій. Для уникнення надмірних звернень до Anthropic API реалізовано механізм кешування на рівні бази даних. Перед кожним зверненням до мовної моделі система перевіряє наявність актуальної рекомендації для поточного користувача та міста. Якщо рекомендація існує і термін її дії ще не минув - вона повертається користувачу безпосередньо з бази даних без нового запиту.

Разом із текстом рекомендації зберігається знімок погодних даних на момент генерації. Це дозволяє в майбутньому аналізувати, за яких умов були сформовані ті чи інші поради, а також забезпечує відтворюваність результату навіть у разі зміни зовнішніх умов.

Обробка помилок генерації. У процесі роботи модуля можуть виникати такі нестандартні ситуації: тимчасова недоступність Anthropic API; перевищення ліміту токенів або денної квоти запитів; повернення відповіді некоректного типу; відсутність обов'язкових параметрів профілю користувача.

У всіх цих випадках система генерує виняток, який перехоплюється на рівні обробника запиту. Рекомендація не зберігається до бази даних, а користувач отримує відповідне повідомлення без порушення загального потоку роботи застосунку. Погодні дані при цьому залишаються доступними - помилка генерації рекомендації не впливає на відображення метеорологічних показників.

3.6 Реалізація користувацького інтерфейсу

Технічно коректне отримання даних та генерація рекомендацій самі по собі ще не гарантують практичної цінності системи. Не менш важливо, щоб користувач міг швидко отримати всю необхідну інформацію в зручному вигляді. Тому реалізація інтерфейсу є повноцінним компонентом системи, а не другорядною візуальною оболонкою.

Принципи побудови інтерфейсу. Інтерфейс програмного забезпечення побудований на кількох базових принципах. Першим є інформаційна щільність без перевантаження: користувач бачить усі ключові дані на одному екрані без необхідності переходити між сторінками. Другим є контекстність: погодні показники, графік температури, деталізовані параметри, прогноз та персоналізована рекомендація розташовані в межах одного перегляду і логічно доповнюють одне одного. Третім є адаптивність: інтерфейс підтримує світлу та темну теми, що дозволяє комфортно користуватися застосунком у різних умовах освітлення.

Оскільки система орієнтована на щоденне використання, дизайн уникає декоративної надлишковості. Візуальна ієрархія побудована таким чином, що погодна інформація сприймається миттєво, а рекомендація - як природне продовження перегляду.

Навігація та загальна структура. Навігація реалізована через хедер, який присутній на всіх сторінках застосунку. Він містить логотип системи, пошуковий рядок для введення назви міста, перемикач теми оформлення, вибір мови інтерфейсу та аватар користувача з доступом до налаштувань профілю. Така структура забезпечує доступ до всіх ключових функцій із будь-якої точки застосунку без потреби у бічному меню чи додатковій навігаційній панелі.

Головний екран. Головний екран є єдиним основним екраном застосунку і поєднує всі функціональні блоки в одному перегляді (рис. 3.5). У верхній частині розташована картка поточної погоди, що містить назву міста та країни, поточну температуру, відчутну температуру, мінімальне та максимальне значення за добу, відсоток вологості, швидкість вітру та іконку з текстовим описом погодних умов. Праворуч від картки відображається графік зміни температури протягом доби, що дозволяє швидко оцінити температурну динаміку без перегляду детального прогнозу.

Нижче розташований блок рекомендацій, згенерованих мовною моделлю. Він відображається одразу під основними погодними даними та позначений міткою, що вказує на автоматичну генерацію. Рекомендація подається у вигляді суцільного тексту без форматування, що відповідає природному стилю персональної поради.

Головний екран є центральним елементом користувацького інтерфейсу системи, оскільки саме через нього здійснюється взаємодія користувача з основним функціоналом застосунку. На одному екрані поєднано перегляд поточних погодних умов, прогнозу, графічної візуалізації температури та персоналізованих AI-рекомендацій, що дозволяє забезпечити швидкий доступ до всієї необхідної інформації. На рис. 3.5 наведено головний екран застосунку моніторингу кліматичних параметрів.

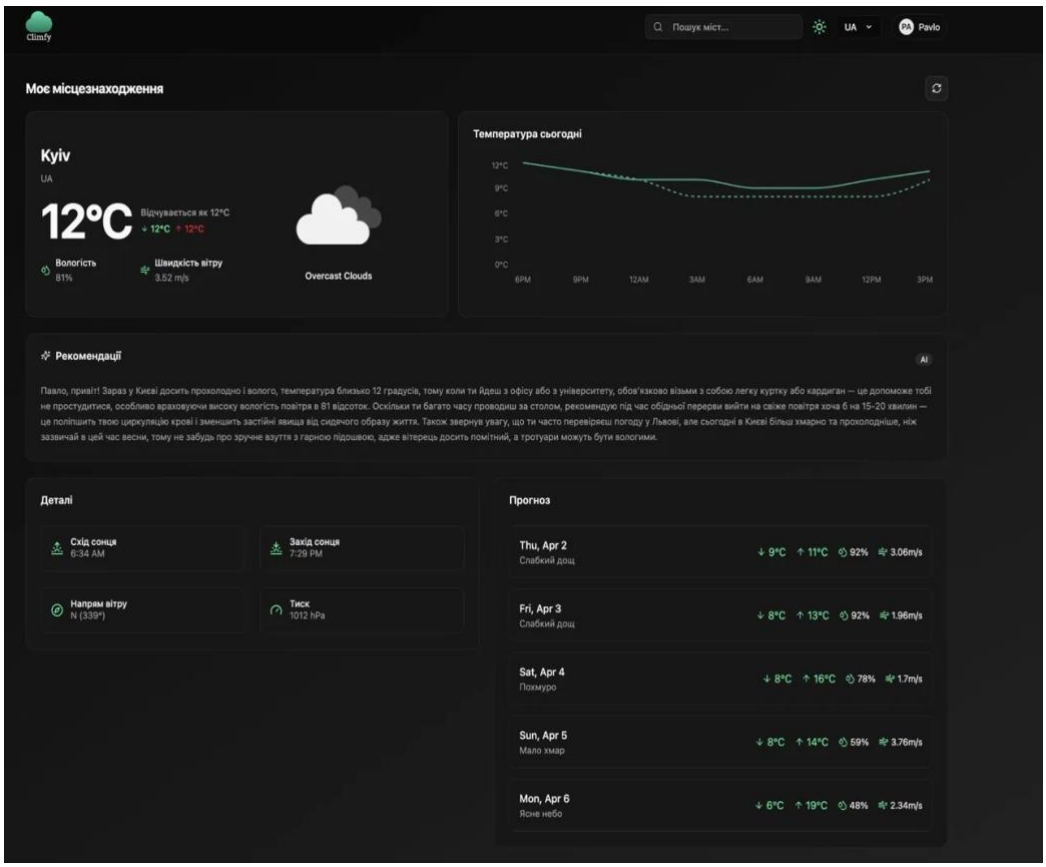

Рис. 3.5 Головний екран застосунку

Як показано на рис. 3.5, інтерфейс застосунку побудований за принципом компактного розміщення основних інформаційних блоків на одному екрані. Центральне місце займає відображення поточних погодних параметрів та персоналізованої рекомендації, тоді як додаткові блоки прогнозу та графіків забезпечують користувачу більш детальний аналіз погодної ситуації. Такий підхід дозволяє знизити когнітивне навантаження та підвищити зручність повсякденного використання системи.

У нижній частині екрану розміщені два додаткові блоки: деталізовані погодні параметри - схід та захід сонця, напрям і тиск вітру, атмосферний тиск - та прогноз погоди на п'ять днів із мінімальною та максимальною температурою, рівнем вологості та швидкістю вітру для кожного дня.

Онбординг та налаштування профілю. При першому вході до системи після реєстрації користувач проходить обов'язковий етап онбордингу у форматі модального вікна. Форма містить поле для введення імені, поле для введення

віку, блок вибору рівня фізичної активності у вигляді чотирьох кнопок - офіс або навчання, прогулянки, спортзал або походи, спорт або тренування - а також вибір мови інтерфейсу та одиниць вимірювання температури. Усі параметри активності та мови реалізовані у вигляді кнопок із візуальним підсвічуванням обраного варіанту, що спрощує взаємодію порівняно зі стандартними випадаючими списками.

Ті самі налаштування доступні в будь-який момент через модальне вікно профілю, яке відкривається з хедера. Це дозволяє користувачу оновити свої дані без необхідності проходити повторний онбординг.

Для забезпечення персоналізації роботи системи реалізовано модальне вікно налаштувань користувача, яке дозволяє змінювати основні параметри профілю без переходу на окремі сторінки застосунку. Через даний інтерфейс користувач може налаштовувати мову, одиниці вимірювання температури та рівень фізичної активності, що безпосередньо впливає на формування персоналізованих рекомендацій. На рис. 3.6 наведено модальне вікно налаштувань системи.

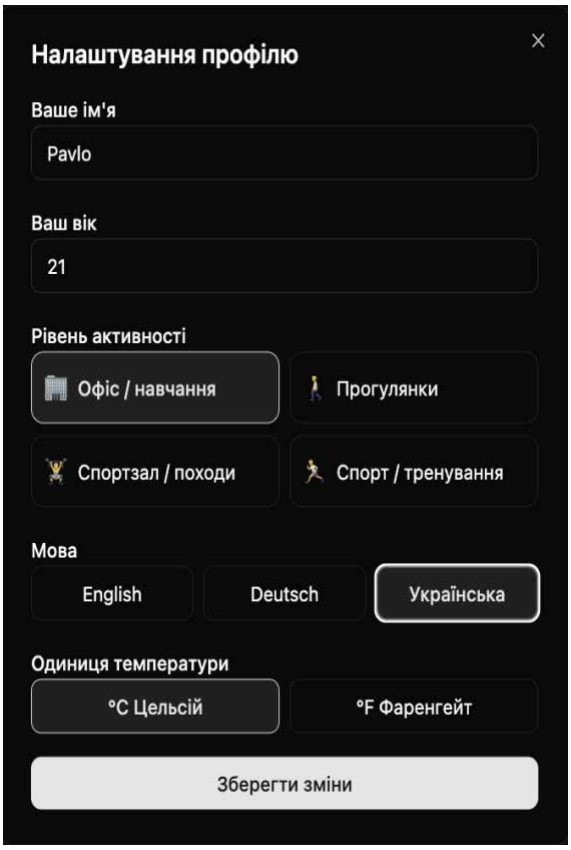

Рис. 3.6 Модальне вікно налаштувань

Як показано на рис. 3.6, налаштування реалізовані у вигляді компактного модального інтерфейсу з використанням інтерактивних елементів вибору параметрів. Такий підхід дозволяє спростити взаємодію користувача із системою, забезпечити швидке оновлення персональних даних та підвищити зручність використання застосунку без необхідності переходу між окремими сторінками.

Дизайн як засіб зниження когнітивного навантаження. У контексті цієї системи дизайн виконує функціональну роль. Концентрація всіх даних на одному екрані усуває необхідність навігації між розділами під час звичайного перегляду погоди. Темна тема знижує зорову втому при використанні в умовах слабкого освітлення. Чітка візуальна ієрархія - великий показник температури, другорядні параметри дрібнішим шрифтом, окремі картки для кожного блоку - дозволяє користувачу миттєво знайти потрібну інформацію без зайвого сканування екрану.

Адаптивна верстка забезпечує коректне відображення як на десктопі, так і на мобільних пристроях, що є важливим для застосунку, орієнтованого на щоденне використання в різних умовах.

4 РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ ПРОГРАМНОЇ СИСТЕМИ

4.1 Аналіз переваг, обмежень і напрямів розвитку системи

Оцінювання запропонованої системи в межах цієї роботи має переважно якісний характер, оскільки основна увага зосереджена на архітектурі, логіці функціонування та реалізації прототипу. Водночас уже на цьому етапі можна проаналізувати практичні переваги підходу, його обмеження та можливі напрями подальшого розвитку.

Розглянемо основні переваги системи.

Персоналізація на основі профілю користувача. Ключовою відмінністю системи від стандартних погодних застосунків є те, що рекомендації формуються не як універсальні поради, а як відповідь на конкретний профіль: вік, рівень фізичної активності та мову користувача. Це перетворює погодні дані на практично значущу інформацію, адаптовану до реального способу життя конкретної людини.

Автоматична генерація рекомендацій без зайвих дій. Користувач не виконує жодних додаткових дій для отримання рекомендації - вона з'являється автоматично при завантаженні погоди для авторизованого користувача. Це усуває бар'єр між отриманням даних та їх практичним застосуванням.

Кешування результатів. Система не звертається до Anthropic API при кожному перегляді погоди. Якщо актуальна рекомендація вже існує і термін її дії не минув, вона повертається з бази даних. Це суттєво знижує витрати на використання зовнішнього API та зменшує час відповіді для користувача.

Адаптивний інтерфейс. Підтримка світлої та темної тем дозволяє комфортно користуватися застосунком у різних умовах освітлення, а адаптивна верстка забезпечує коректну роботу як на десктопі, так і на мобільних пристроях.

Підтримка кількох мов. Рекомендації генеруються мовою, вказаною в профілі користувача - українською, англійською або німецькою. Це робить

систему придатною до використання в міжнародному контексті без необхідності реалізовувати окремий модуль перекладу.

Попри переваги, система має низку обмежень, які необхідно враховувати.

Залежність від зовнішніх API. Робота системи повністю прив'язана до доступності OpenWeather API та Anthropic API. Будь-які збої, зміни тарифних планів, лімітів або умов використання цих сервісів безпосередньо впливають на стабільність функціонування застосунку.

Рекомендації на основі лише поточної погоди. У поточній реалізації мовна модель отримує знімок погодних даних на момент запиту без урахування прогнозу. Це означає, що рекомендація може не враховувати погіршення умов, яке очікується протягом дня.

Залежність від зовнішніх сервісів. Робота системи прив’язана до доступності Gmail API, Google Cloud та OpenAI API. Будь-які обмеження, зміни політик, тарифів або лімітів можуть впливати на стабільність функціонування.

Відсутність push-сповіщень. Система не інформує користувача про різкі зміни погодних умов у реальному часі. Для отримання актуальних даних користувач повинен самостійно відкрити застосунок.

Навіть без проведення масштабного експериментального дослідження можна визначити набір критеріїв, за якими доцільно оцінювати запропоновану систему:

- релевантність рекомендацій - наскільки поради відповідають реальним погодним умовам та профілю користувача;
- швидкість відповіді - час від введення міста до відображення рекомендації;
- ефективність кешування - частка запитів, оброблених без звернення до Anthropic API;
- коректність роботи мультимовності - відповідність мови відповіді мові профілю;
- зручність інтерфейсу - наскільки швидко користувач може отримати всі необхідні дані за одне відкриття застосунку.

У подальшому ці критерії можуть стати основою для формального тестування системи в межах користувацького дослідження.

Розроблений застосунок гнучкий до масштабування. Ціла низка напрямів подальшого розвитку, які зроблять застосунок більш ефективним.

Рекомендації на основі прогнозу. Найбільш природнім розширенням є врахування прогнозних даних при формуванні рекомендацій. Це дозволить попереджати користувача про погіршення умов протягом дня та давати більш практичні поради щодо планування активності.

Push-сповіщення про зміну погоди. Реалізація механізму сповіщень дозволить інформувати користувача про різкі зміни погодних умов у містах зі списку обраних без необхідності самостійно відкривати застосунок.

Розширення профілю користувача. Додавання таких параметрів, як наявність хронічних захворювань, алергій або особливих медичних умов, дозволило б суттєво підвищити практичну цінність рекомендацій для окремих категорій користувачів.

Мобільний застосунок. Розробка нативного мобільного застосунку на основі наявної бекенд-інфраструктури є логічним кроком, зважаючи на те, що перегляд погоди є типовим сценарієм використання мобільних пристроїв.

Додавання чату з AI-синоптиком. Деякі ситуації можуть бути не покриті рекомендацію, яку ми формуємо, тому доцільно додати чату з AI асистентом, який буде надавати поради до більш детальних ситуацій користувачів.

Навіть у базовій конфігурації система демонструє важливий принцип: метеорологічні дані можуть бути не просто відображені, а й автоматично перетворені на персоналізовані практичні рекомендації з урахуванням індивідуальних характеристик користувача. Це зміщує фокус із пасивного перегляду погодних показників на активне отримання дієвої інформації.

Користувач більше не змушений самостійно інтерпретувати набір цифр і вирішувати, що вони означають для його конкретного способу життя. Натомість він отримує готову пораду, сформовану з урахуванням його профілю, поточного часу доби та сезону. У довгостроковій перспективі такий підхід може суттєво

підвищити практичну цінність погодних сервісів і стати основою для ширшої платформи персоналізованих рекомендацій.

4.2 Тестування системи

Тестування веб-застосунку організоване на двох рівнях: статична перевірка під час збірки та end-to-end тести у реальних браузерах.

Статичний контроль запускається автоматично - перед кожним комітом і під час збірки. TypeScript у строгому режимі перевіряє типи. ESLint із плагінами react-hooks, @tanstack/eslint-plugin-query та prettier стежить за правилами компонентів, коректністю хуків і форматуванням. vite-plugin-checker дублює ці перевірки в реальному часі під час розробки. Husky та lint-staged просто не пропускають коміт, якщо щось не так.

End-to-end тести написані на Playwright і живуть у tests/e2e/. Кожен запуск йде одночасно у Chromium, Firefox та WebKit. Конфіг автоматично піднімає dev-сервер на порту 3000, дозволяє геолокацію, фіксує локаль en-US і темну тему. Щоб тести не залежали від реального місцезнаходження машини, кожен сценарій на старті отримує координати Києва через допоміжну функцію.

Сценарії розбиті на дві групи залежно від стану входу.

Гість. Перевіряється наявність усіх блоків головного екрана, робота пошуку на прикладі Берліна - введення, поява підказок, перехід і заголовок сторінки. Картка рекомендацій у гостьовому сеансі має показувати запрошення увійти, а кнопка «Вибране» - бути прихованою.

Авторизований користувач. Сценарій починається з loginTestUser, який заповнює форму тестовими даними. Далі перевіряється, що картка рекомендацій змінила вміст, кнопка «Вибране» з'явилася, а після додавання міста колір кнопки стає жовтим і місто відображається у списку на головній. Окремо тестуються некоректні облікові дані - діалог має залишатися відкритим - та вихід із системи.

Елементи шукаються виключно через data-testid із переліку WeatherTestId. Зміни в розмітці чи CSS тести не ламають. Звіт формується у HTML, скриншоти зберігаються лише при невдачах, трасування - при повторних запусках. У CI

один потік і дві спроби повтору - про всяк випадок, якщо OpenWeatherMap затримає відповідь.

4.3 Вимоги до апаратного та програмного забезпечення

Робота являється клієнт-серверний застосунком, того вимоги до системи залежать від того, хто і де з ним працює.

Користувач у браузері. Нічого не встановлюється - застосунок повністю живе у браузері. Підійде будь-який пристрій з процесором від 1 ГГц та 1 ГБ оперативної пам'яті. Потрібне стабільне з'єднання від 1 Мбіт/с - без нього ні погода, ні рекомендації не завантажаться. Браузер має підтримувати ECMAScript 2022 і дозволяти JavaScript та localStorage: Chrome 110+, Firefox 110+, Safari 16+, Edge 110+. Геолокація не обов'язкова - лише для автоматичного визначення міста. ОС значення не має: Windows 10/11, macOS 12+, Linux, Android 9+, iOS 16+ - головне, щоб браузер був сучасний.

Сервер у хмарі. Розгортається на Railway. Мінімальний екземпляр тягне 1 ядро, 512 МБ RAM і 1 ГБ диску - цього вистачає для старту. Під реальним навантаженням достатньо стандартного тарифу з 2 vCPU і 1 ГБ RAM. База даних - керований PostgreSQL на Supabase, жодних окремих серверів не потрібно. Зі стеку: Node.js 20+ із прапорцем --env-file, Linux на базі Debian. Із зовнішніх сервісів - OpenWeatherMap і Anthropic Claude (claude-haiku-4-5-20251001). Перед запуском у Supabase мають бути застосовані всі міграції і згенерований service-role ключ.

Локальна машина розробника. Node.js 20+, npm, Git. Редактор - будь-який із підтримкою TypeScript, але VS Code із ESLint і Prettier заощадить час. Для Playwright потрібні три браузери - Chromium, Firefox і WebKit; завантажуються однією командою npx playwright install. Апаратний мінімум: 4 ядра, 8 ГБ RAM, 5 ГБ вільного місця під залежності та артефакти збірки.

4.4 Склад інсталяційного пакету

Climfy складається з двох незалежних частин - клієнта і сервера. Класичного інсталятора немає: обидві розгортаються автоматично з Git через Vercel і Railway.

Клієнт - статична SPA-збірка. npm run build формує директорію climfy/dist/ із точкою входу index.html та набором чанків: основний бандл, окремі файли для великих залежностей (vendor, query, supabase, ui, i18n, charts) і стилі. Браузер кешує залежності незалежно від коду застосунку - так перезбірка після змін не скидає весь кеш. vercel.json описує правила переадресації для React Router.

Перед збіркою потрібен .env із чотирма ключами: VITE_OPENWEATHER_API_KEY, VITE_API_BASE_URL, VITE_SUPABASE_URL та VITE_SUPABASE_ANON_KEY. Шаблон лежить у .env.example.

Сервер - директорія climfy-server/ із TypeScript-кодом, який tsx запускає напряму без компіляції. Структура: src/index.ts - Hono-застосунок із налаштуваннями CORS; routes/ - п'ять модулів (auth, favorites, history, preferences, recommendations); middleware/ - перевірка JWT-токена Supabase; lib/ - клієнти Supabase і Claude та логіка генерації рекомендацій; types/ - спільні інтерфейси. Поряд - railway.json для автодеплою і tsconfig.json.

Серверні змінні: PORT (3001 за замовчуванням), SUPABASE_URL, SUPABASE_SERVICE_ROLE_KEY - цей ключ лишається тільки на сервері і ніколи не потрапляє в браузер, OPENWEATHER_API_KEY, ANTHROPIC_API_KEY та ALLOWED_ORIGINS для CORS.

База даних - файл supabase/migrations.sql. Один запуск у SQL Editor Supabase створює чотири таблиці та вмикає Row Level Security: кожен рядок бачить тільки той, хто його створив.

У репозиторії також є README.md із інструкцією запуску, шаблони листів для Supabase Auth, Playwright-тести зі звітами та документація з діаграмами у docs/.

Для зручності аналізу структури інсталяційного пакету доцільно узагальнити основні компоненти системи, їх призначення та технологічне наповнення у вигляді таблиці. Це дозволяє наочно представити склад програмного забезпечення та взаємозв'язки між його основними частинами. Основні компоненти інсталяційного пакету наведено в табл. 4.1.

Таблиця 4.1

Основні компоненти інсталяційного пакету

№	Компонент	Призначення	Основні технології
1	Клієнтська частина (Frontend)	Забезпечує взаємодію користувача із системою, відображення погодних даних та рекомендацій	React, TypeScript, Vite, Tailwind CSS
2	Серверна частина (Backend)	Обробка запитів, взаємодія з API та бізнес-логіка системи	Node.js, Hono
3	База даних	Зберігання профілів користувачів, історії пошуку та рекомендацій	PostgreSQL, Supabase
4	Модуль AI-рекомендацій	Генерація персоналізованих рекомендацій на основі погодних даних	Anthropic Claude API
5	Модуль погодних даних	Отримання поточних метеорологічних параметрів та прогнозу	OpenWeather API
6	Модуль тестування	Перевірка працездатності системи та автоматизація тестування	Playwright, ESLint
7	Конфігураційні файли	Налаштування середовища та автоматизація розгортання	.env, railway.json, vercel.json

Наведена в табл. 4.1 структура демонструє модульний підхід до побудови програмної системи. Розподіл функціональності між окремими компонентами забезпечує гнучкість архітектури, спрощує підтримку програмного забезпечення та дозволяє незалежно масштабувати окремі частини застосунку. Використання сучасних вебтехнологій та хмарних сервісів також забезпечує можливість автоматизованого розгортання й подальшого розвитку системи.

Порядок першого розгортання простий: репозиторій на GitHub, Vercel із кореневою директорією climfy/ і клієнтськими змінними, Railway із climfy-server/

і серверними, Supabase із виконаною міграцією. Далі кожен push у головну гілку оновлює обидва сервіси автоматично - адреса Railway вписується у VITE_API_BASE_URL один раз і більше не змінюється.

Отже, сформований склад інсталяційного пакету охоплює всі необхідні програмні компоненти для розгортання та стабільного функціонування системи моніторингу кліматичних параметрів. Використання сучасних інструментів автоматизованого деплою, хмарної інфраструктури та модульної організації програмного забезпечення забезпечує зручність впровадження, спрощує супровід системи та створює передумови для подальшого розвитку функціональних можливостей застосунку.

ВИСНОВКИ

У процесі виконання кваліфікаційної бакалаврської роботи досліджено актуальну проблему обмеженої практичної цінності сучасних погодних сервісів та запропоновано підхід до її розв'язання шляхом розробки інформаційної системи моніторингу кліматичних параметрів з персоналізованими рекомендаціями на основі штучного інтелекту.

У ході роботи встановлено, що більшість наявних погодних застосунків надає користувачу лише набір метеорологічних показників без урахування його індивідуальних характеристик. Це створює практичну проблему: користувач змушений самостійно інтерпретувати дані та приймати рішення щодо одягу, активності та планування дня, не маючи жодної персоналізованої підтримки. Предметна область потребує інструментів, які поєднують отримання кліматичних даних з інтелектуальним аналізом та адаптацією до конкретного профілю користувача.

Основні результати виконаної роботи полягають у такому:

- проаналізовано предметну область та визначено ключову проблему - розрив між отриманням метеорологічних даних та їх перетворенням на практично корисну персоналізовану інформацію.
- розглянуто існуючі підходи та виявлено, що більшість наявних рішень не враховує індивідуальних характеристик користувача при формуванні порад;
- сформульовано функціональні та нефункціональні вимоги до системи, орієнтованої на персоналізовану генерацію рекомендацій.
- спроєктовано архітектуру, що поєднує React-фронтенд, окремий бекенд-сервер, Supabase, OpenWeather API та Anthropic API.
- описано модель взаємодії компонентів системи, діаграму прецедентів, діаграму розгортання та алгоритм функціонування від введення міста до отримання персоналізованої рекомендації.

- розглянуто структуру даних, необхідну для збереження профілів користувачів, їхніх налаштувань, улюблених міст, історії пошуку та згенерованих рекомендацій.
- описано реалізацію інтеграції з OpenWeather API, включно з геокодингом, отриманням поточної погоди, прогнозу та зворотним геокодуванням за координатами геолокації.
- наведено принципи реалізації модуля генерації рекомендацій через Anthropic API з використанням моделі Claude Haiku, динамічного промпту на основі погодних даних і профілю користувача та механізму кешування результатів.
- охарактеризовано користувацький інтерфейс системи, включно з головним екраном, онбордингом та налаштуваннями профілю; виконано якісний аналіз переваг, обмежень і перспектив розвитку системи.

Запропонована система має практичну цінність як сучасний інструмент персоналізованого моніторингу кліматичних параметрів. Її головна перевага полягає в тому, що вона перенаправляє фокус із пасивного перегляду метеорологічних показників на автоматичне отримання конкретних практичних порад, адаптованих до способу життя, рівня активності та мови конкретного користувача. Це дозволяє суттєво підвищити практичну цінність погодних даних і зробити щоденне планування більш усвідомленим.

Водночас результати роботи показують, що подальший розвиток системи доцільно спрямовувати на розширення профілю користувача, врахування прогнозних даних при генерації рекомендацій, реалізацію push-сповіщень про різкі зміни погоди, розробку мобільного застосунку та підтримку додаткових мов інтерфейсу. У такому вигляді система може стати важливим кроком у розвитку інтелектуальних персоналізованих сервісів підтримки повсякденного планування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Anthropic. Claude API Overview [Електронний ресурс]. – Режим доступу: https://platform.claude.com/docs/en/api/overview.
2. Claude API Overview. URL: https://platform.claude.com/docs/en/api/overview.
3. Hono Documentation [Електронний ресурс]. – Режим доступу: https://hono.dev/.
4. Mozilla Developer Network. HTTP Overview [Електронний ресурс]. – Режим доступу: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.
5. OpenWeatherMap API Documentation [Електронний ресурс]. – Режим доступу: https://openweathermap.org/api/.
6. Playwright Documentation [Електронний ресурс]. – Режим доступу: https://playwright.dev/.
7. PostgreSQL Global Development Group. PostgreSQL Documentation [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/
8. React Documentation [Електронний ресурс]. – Режим доступу: https://react.dev/
9. Recharts Documentation [Електронний ресурс]. – Режим доступу: https://recharts.org/
10. Sommerville I. Software Engineering. – 11th ed. – Boston : Pearson, 2020. – 811p.
11. Supabase Auth Documentation. [Електронний ресурс]. – Режим доступу: https://supabase.com/docs/guides/auth.
12. Supabase Documentation [Електронний ресурс]. – Режим доступу: https://supabase.com/docs
13. Supabase Row Level Security Guide [Електронний ресурс]. – Режим доступу: https://supabase.com/docs/guides/auth/row-level-security/
14. TanStack Query v5 Documentation [Електронний ресурс]. – Режим доступу: https://tanstack.com/query/v5/
15. UML Diagrams. [Електронний ресурс]. – Режим доступу: https://www.uml-diagrams.org/.
16. Vite Documentation [Електронний ресурс]. – Режим доступу: https://vitejs.dev/

17. Wieruch R. The Road to React. – Berlin : Self-Published Developer Edition, 2024. – 514 p.
18. Williams A. Modern Web Development with React and TypeScript. – Birmingham : Packt Publishing, 2023. – 620 p.
19. Zhang H., Kumar S. Cloud-Based Information Systems Architecture and API Integration // International Journal of Information Systems and Software Engineering. – 2024. – Vol. 10, No. 2. – P. 55–69.
20. Zhao Y., Li X. Artificial Intelligence Integration in Modern Web Applications // Journal of Web Engineering. – 2023. – Vol. 22, No. 5. – P. 901–918.
21. Биков В. Ю., Спірін О. М., Пінчук О. П. Цифрова трансформація освіти і сучасні інформаційні технології // Інформаційні технології і засоби навчання. – 2022. – Т. 89, № 3. – С. 6–27.
22. Гриб’юк О. О. Інтелектуальні інформаційні системи підтримки прийняття рішень : монографія. – Київ : НПУ імені М. П. Драгоманова, 2021. – 312 с.
23. Жалдак М. І., Морзе Н. В. Інформаційні системи та технології : навчальний посібник. – Київ : Видавництво Ліра-К, 2021. – 384 с.
24. Литвин В. В., Пасічник В. В. Інтелектуальні системи підтримки прийняття рішень : навчальний посібник. – Львів : Новий Світ-2000, 2022. – 368 с.
25. Морзе Н. В., Вембер В. П. Хмарні технології в інформаційних системах : навчальний посібник. – Київ : Університет імені Бориса Грінченка, 2021. – 276 с.
26. Пелещишин А. М., Слобода Н. Б. Методи інтеграції зовнішніх API у веборієнтованих інформаційних системах // Вісник Національного університету «Львівська політехніка». Серія: Інформаційні системи та мережі. – 2024. – № 11. – С. 44–52.
27. Про затвердження стандарту вищої освіти за спеціальністю 121 «Інженерія програмного забезпечення» для першого (бакалаврського) рівня вищої освіти : наказ МОН України № 1166 від 29.10.2018 (зі змінами станом на 2024 р.).

ДОДАТОК А

Опис таблиць бази даних додатку

```sql
-- 1. User preferences
CREATE TABLE IF NOT EXISTS user_preferences (
 user_id          UUID PRIMARY KEY REFERENCES auth.users(id) ON DELETE CASCADE,
 temperature_unit TEXT NOT NULL DEFAULT 'celsius',
 language         TEXT NOT NULL DEFAULT 'en',
 updated_at       TIMESTAMPTZ NOT NULL DEFAULT now()
);

ALTER TABLE user_preferences ENABLE ROW LEVEL SECURITY;
CREATE POLICY "Users manage own preferences"
 ON user_preferences FOR ALL
 USING (auth.uid() = user_id)
 WITH CHECK (auth.uid() = user_id);

-- 2. Favorites
CREATE TABLE IF NOT EXISTS favorites (
 id        TEXT NOT NULL,
 user_id   UUID NOT NULL REFERENCES auth.users(id) ON DELETE CASCADE,
 city_name TEXT NOT NULL,
 country   TEXT,
 state     TEXT,
 lat       DECIMAL NOT NULL,
 lon       DECIMAL NOT NULL,
 added_at  TIMESTAMPTZ NOT NULL DEFAULT now(),
 PRIMARY KEY (id, user_id)
);

ALTER TABLE favorites ENABLE ROW LEVEL SECURITY;
CREATE POLICY "Users manage own favorites"
 ON favorites FOR ALL
 USING (auth.uid() = user_id)
 WITH CHECK (auth.uid() = user_id);

-- 3. Search history
CREATE TABLE IF NOT EXISTS search_history (
 id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
 user_id     UUID NOT NULL REFERENCES auth.users(id) ON DELETE CASCADE,
 city_name   TEXT NOT NULL,
 country     TEXT,
 state       TEXT,
 lat         DECIMAL NOT NULL,
 lon         DECIMAL NOT NULL,
 searched_at TIMESTAMPTZ NOT NULL DEFAULT now()
);

ALTER TABLE search_history ENABLE ROW LEVEL SECURITY;
CREATE POLICY "Users manage own history"
 ON search_history FOR ALL
 USING (auth.uid() = user_id)
 WITH CHECK (auth.uid() = user_id);
```

```sql
CREATE INDEX search_history_user_id_searched_at
 ON search_history(user_id, searched_at DESC);

-- 4. Recommendations cache
CREATE TABLE IF NOT EXISTS recommendations (
 id               UUID PRIMARY KEY DEFAULT gen_random_uuid(),
 user_id          UUID NOT NULL REFERENCES auth.users(id) ON DELETE CASCADE,
 lat              DECIMAL NOT NULL,
 lon              DECIMAL NOT NULL,
 language         TEXT NOT NULL DEFAULT 'en',
 weather_snapshot JSONB NOT NULL,
 content          TEXT NOT NULL,
 is_ai            BOOLEAN NOT NULL DEFAULT false,
 generated_at     TIMESTAMPTZ NOT NULL DEFAULT now(),
 expires_at       TIMESTAMPTZ NOT NULL
);

ALTER TABLE recommendations ENABLE ROW LEVEL SECURITY;
CREATE POLICY "Users manage own recommendations"
 ON recommendations FOR ALL
 USING (auth.uid() = user_id)
 WITH CHECK (auth.uid() = user_id);

CREATE INDEX recommendations_lookup
 ON recommendations(user_id, language, expires_at DESC);

ALTER TABLE user_preferences
 ADD COLUMN IF NOT EXISTS first_name           TEXT,
 ADD COLUMN IF NOT EXISTS age                  INTEGER,
 ADD COLUMN IF NOT EXISTS activity_type        TEXT DEFAULT 'light',
 ADD COLUMN IF NOT EXISTS onboarding_completed BOOLEAN NOT NULL DEFAULT false;
```

ДОДАТОК Б

Код програмних рішень додатку

Лістинг Б.1 Додавання міста до вибраного

На фронт частині (кнопка)

```
import { Star } from 'lucide-react'
import { Button } from '@/shared/components/ui/button'
import type { WeatherData } from '@/features/weather/api/types'
import { useFavorites } from '@/features/favorites/hooks/use-favorite'
import { useAuth } from '@/features/auth/context/auth-context'
import { toast } from 'sonner'
import { WeatherTestId } from 'tests/resources/enums'
import { useTranslation } from 'react-i18next'

interface FavoriteButtonProps {
 data: WeatherData
}

export function FavoriteButton({ data }: FavoriteButtonProps) {
 const { user } = useAuth()
 const { addFavorite, removeFavorite, isFavorite } = useFavorites()
 const { t } = useTranslation()
 const isCurrentlyFavorite = isFavorite(data.coord.lat, data.coord.lon)

 if (!user) return null

 const handleToggleFavorite = () => {
   if (isCurrentlyFavorite) {
     removeFavorite.mutate(`${data.coord.lat}-${data.coord.lon}`)
     toast.error(t('favorites.remove', { city: data.name }))
   } else {
     addFavorite.mutate({
       name: data.name,
       lat: data.coord.lat,
       lon: data.coord.lon,
       country: data.sys.country,
     })
     toast.success(t('favorites.add', { city: data.name }))
   }
 }

 return (
   <Button
     variant={isCurrentlyFavorite ? 'default' : 'outline'}
     size="icon"
     onClick={handleToggleFavorite}
     className={isCurrentlyFavorite ? 'bg-yellow-500 hover:bg-yellow-600' :
''}
```

```
      data-testid={WeatherTestId.FavoriteButton}
    >
      <Star
        className={`h-4 w-4 ${isCurrentlyFavorite ? 'fill-current' : ''}`}
      />
    </Button>
  )
}
```

На фронт частині (хук з мутаціями)

```
import { useQuery, useMutation, useQueryClient } from '@tanstack/react-query'
import { useAuth } from '@/features/auth/context/auth-context'
import { supabase } from '@/lib/supabase'

export interface FavoriteCity {
  id: string
  name: string
  lat: number
  lon: number
  country: string
  state?: string
  addedAt: number
}

const API_BASE = import.meta.env.VITE_API_BASE_URL as string

async function getToken(): Promise<string | null> {
  const { data } = await supabase.auth.getSession()
  return data.session?.access_token ?? null
}

export function useFavorites() {
  const { user } = useAuth()
  const queryClient = useQueryClient()
  const queryKey = ['favorites', user?.id]

  const favoritesQuery = useQuery({
    queryKey,
    queryFn: async () => {
      const token = await getToken()
      if (!token) return []
      const res = await fetch(`${API_BASE}/favorites`, {
        headers: { Authorization: `Bearer ${token}` },
      })
      if (!res.ok) return []
      const data = await res.json()
      return data.map(
        (item: {
          id: string
          city_name: string
          lat: number
          lon: number
```

```
        country: string
        state?: string
        added_at: string
      }) => ({
        id: item.id,
        name: item.city_name,
        lat: item.lat,
        lon: item.lon,
        country: item.country,
        state: item.state,
        addedAt: Date.parse(item.added_at),
      }),
    ) as FavoriteCity[]
  },
  enabled: !!user,
  staleTime: 5 * 60 * 1000,
})

const addFavorite = useMutation({
  mutationFn: async (city: Omit<FavoriteCity, 'id' | 'addedAt'>) => {
    const token = await getToken()
    if (!token) return
    await fetch(`${API_BASE}/favorites`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        Authorization: `Bearer ${token}`,
      },
      body: JSON.stringify({
        id: `${city.lat}-${city.lon}`,
        city_name: city.name,
        lat: city.lat,
        lon: city.lon,
        country: city.country,
        state: city.state,
      }),
    })
  },
  onMutate: async (city) => {
    await queryClient.cancelQueries({ queryKey })
    const previous = queryClient.getQueryData<FavoriteCity[]>(queryKey)

    const optimistic: FavoriteCity = {
      ...city,
      id: `${city.lat}-${city.lon}`,
      addedAt: Date.now(),
    }
    queryClient.setQueryData<FavoriteCity[]>(queryKey, (old = []) => [
      ...old,
      optimistic,
    ])
```

```
      return { previous }
    },
    onError: (_err, _city, context) => {
      if (context?.previous !== undefined) {
        queryClient.setQueryData(queryKey, context.previous)
      }
    },
    onSettled: () => {
      queryClient.invalidateQueries({ queryKey })
    },
  })

  const removeFavorite = useMutation({
    mutationFn: async (cityId: string) => {
      const token = await getToken()
      if (!token) return
      await fetch(`${API_BASE}/favorites/${cityId}`, {
        method: 'DELETE',
        headers: { Authorization: `Bearer ${token}` },
      })
    },
    onMutate: async (cityId) => {
      await queryClient.cancelQueries({ queryKey })
      const previous = queryClient.getQueryData<FavoriteCity[]>(queryKey)

      queryClient.setQueryData<FavoriteCity[]>(queryKey, (old = []) =>
        old.filter((city) => city.id !== cityId),
      )

      return { previous }
    },
    onError: (_err, _cityId, context) => {
      if (context?.previous !== undefined) {
        queryClient.setQueryData(queryKey, context.previous)
      }
    },
    onSettled: () => {
      queryClient.invalidateQueries({ queryKey })
    },
  })

  const favorites = favoritesQuery.data ?? []

  return {
    favorites,
    addFavorite,
    removeFavorite,
    isFavorite: (lat: number, lon: number) =>
      favorites.some((city) => city.lat === lat && city.lon === lon),
  }
}
```

На бек частині

```
import { Hono } from 'hono'
import { createUserClient } from '../lib/supabase.ts'
import { authMiddleware } from '../middleware/auth.ts'
import type { FavoriteCity } from '../types/index.ts'
import type { AppEnv } from '../types/hono.ts'

const favorites = new Hono<AppEnv>()

favorites.use('/*', authMiddleware)

favorites.get('/', async (c) => {
 const user = c.get('user')!
 const db = createUserClient(c.get('token')!)

 const { data, error } = await db
   .from('favorites')
   .select('*')
   .eq('user_id', user.id)
   .order('added_at', { ascending: false })

 if (error) return c.json({ error: error.message }, 500)

 return c.json(data as FavoriteCity[])
})

favorites.post('/', async (c) => {
 const user = c.get('user')!
 const db = createUserClient(c.get('token')!)
 const body = await c.req.json<Omit<FavoriteCity, 'added_at'>>()

 if (!body.id || !body.city_name || body.lat == null || body.lon == null) {
   return c.json({ error: 'id, city_name, lat, lon are required' }, 400)
 }

 const { data, error } = await db
   .from('favorites')
   .upsert(
     {
       id: body.id,
       user_id: user.id,
       city_name: body.city_name,
       country: body.country,
       state: body.state,
       lat: body.lat,
       lon: body.lon,
     },
     { onConflict: 'id,user_id' },
   )
   .select()
   .single()
```

```
 if (error) return c.json({ error: error.message }, 500)

 return c.json(data, 201)
})

favorites.delete('/:id', async (c) => {
 const user = c.get('user')!
 const db = createUserClient(c.get('token')!)
 const id = c.req.param('id')

 const { error } = await db
   .from('favorites')
   .delete()
   .eq('id', id)
   .eq('user_id', user.id)

 if (error) return c.json({ error: error.message }, 500)

 return c.json({ message: 'Removed' })
})

export default favorites
```

Лістинг Б.2 Пошук міста та збереження історії

На фронт частині (компонент пошуку)

```
import { useState } from 'react'
import { useNavigate } from 'react-router'
import { format } from 'date-fns'
import { Search, Loader2, Clock, Star, XCircle } from 'lucide-react'
import { useLocationSearch } from '@/features/weather/hooks/use-weather'
import {
 useSearchHistory,
 type SearchHistoryItem,
} from '@/features/search/hooks/use-search-history'
import {
 Command,
 CommandDialog,
 CommandEmpty,
 CommandGroup,
 CommandInput,
 CommandItem,
 CommandList,
 CommandSeparator,
} from '@/shared/components/ui/command'
import { Button } from '@/shared/components/ui/button'
import { useFavorites } from '@/features/favorites/hooks/use-favorite'
import { WeatherTestId } from 'tests/resources/enums'
import { useTranslation } from 'react-i18next'

export function CitySearch() {
```

```
  const [open, setOpen] = useState(false)
  const [query, setQuery] = useState('')
  const navigate = useNavigate()
  const { t } = useTranslation()

  const { data: locations, isLoading } = useLocationSearch(query)
  const { favorites } = useFavorites()
  const { history, clearHistory, addToHistory } = useSearchHistory()

  const handleSelect = (cityData: string) => {
    const [lat, lon, name, country] = cityData.split('|')

    // Add to search history
    addToHistory.mutate({
      query,
      name,
      lat: parseFloat(lat),
      lon: parseFloat(lon),
      country,
    })

    setOpen(false)
    navigate(`/city/${name}?lat=${lat}&lon=${lon}`)
  }

  return (
    <>
      <Button
        variant="outline"
        className="relative justify-start text-sm text-muted-foreground md:w-
40 lg:w-64"
        onClick={() => setOpen(true)}
        data-testid={WeatherTestId.SearchBar}
      >
        <Search className="h-4 w-4 md:mr-2" />
        <span className="hidden md:inline">{t('search.placeholder')}</span>
      </Button>
      <CommandDialog
        open={open}
        onOpenChange={() => {
          setOpen(false)
          setQuery('')
        }}
      >
        <Command>
          <CommandInput
            placeholder={t('search.placeholder')}
            value={query}
            onValueChange={setQuery}
            data-testid={WeatherTestId.SearchBarInput}
          />
          <CommandList data-testid={WeatherTestId.SearchBarResultList}>
```

```
            {query.length > 2 && !isLoading && (
              <CommandEmpty>{t('search.noResults')}</CommandEmpty>
            )}

            {/* Favorites Section */}
            {favorites.length > 0 && (
              <CommandGroup heading={t('favorites.title')}>
                {favorites.map((city) => (
                  <CommandItem
                    key={city.id}
value={`${city.lat}|${city.lon}|${city.name}|${city.country}`}
                    onSelect={handleSelect}
                    data-testid={WeatherTestId.FavoriteItem}
                  >
                    <Star className="mr-2 h-4 w-4 text-yellow-500" />
                    <span>{city.name}</span>
                    {city.state && (
                      <span className="text-sm text-muted-foreground">
                        , {city.state}
                      </span>
                    )}
                    <span className="text-sm text-muted-foreground">
                      , {city.country}
                    </span>
                  </CommandItem>
                ))}
              </CommandGroup>
            )}

            {/* Search History Section */}
            {history.length > 0 && (
              <>
                <CommandSeparator />
                <CommandGroup>
                  <div className="flex items-center justify-between px-2 my-
2">
                    <p className="text-xs text-muted-foreground">
                      {t('search.recent')}
                    </p>
                    <Button
                      variant="ghost"
                      size="sm"
                      onClick={() => clearHistory.mutate()}
                    >
                      <XCircle className="h-4 w-4" />
                      {t('search.clear')}
                    </Button>
                  </div>
                  {history.map((item: SearchHistoryItem) => (
                    <CommandItem
                      key={item.id}
```

```
value={`${item.lat}|${item.lon}|${item.name}|${item.country}`}
                    onSelect={handleSelect}
                  >
                    <Clock className="mr-2 h-4 w-4 text-muted-foreground" />
                    <span>{item.name}</span>
                    {item.state && (
                      <span className="text-sm text-muted-foreground">
                        , {item.state}
                      </span>
                    )}
                    <span className="text-sm text-muted-foreground">
                      , {item.country}
                    </span>
                    <span className="ml-auto text-xs text-muted-foreground">
                      {format(item.searchedAt, 'MMM d, h:mm a')}
                    </span>
                  </CommandItem>
                ))}
              </CommandGroup>
            </>
          )}

          {/* Search Results */}
          <CommandSeparator />
          {locations && locations.length > 0 && (
            <CommandGroup heading={t('search.suggestions')}>
              {isLoading && (
                <div className="flex items-center justify-center p-4">
                  <Loader2 className="h-4 w-4 animate-spin" />
                </div>
              )}
              {locations?.map((location) => (
                <CommandItem
                  key={`${location.lat}-${location.lon}`}

value={`${location.lat}|${location.lon}|${location.name}|${location.country}
`}
                  onSelect={handleSelect}
                  data-testid={WeatherTestId.SearchResultItem}
                >
                  <Search className="mr-2 h-4 w-4" />
                  <span>{location.name}</span>
                  {location.state && (
                    <span className="text-sm text-muted-foreground">
                      , {location.state}
                    </span>
                  )}
                  <span className="text-sm text-muted-foreground">
                    , {location.country}
                  </span>
                </CommandItem>
```

```
              ))}
            </CommandGroup>
          )}
        </CommandList>
      </Command>
    </CommandDialog>
  </>
 )
}
```

На фронт частині (хук історії)

```
import { useQuery, useMutation, useQueryClient } from '@tanstack/react-query'
import { useAuth } from '@/features/auth/context/auth-context'
import { useSessionStorage } from '@/shared/hooks/use-session-storage'
import { supabase } from '@/lib/supabase'

export interface SearchHistoryItem {
 id: string
 query: string
 lat: number
 lon: number
 name: string
 country: string
 state?: string
 searchedAt: number
}

const API_BASE = import.meta.env.VITE_API_BASE_URL as string

async function getToken(): Promise<string | null> {
 const { data } = await supabase.auth.getSession()
 return data.session?.access_token ?? null
}

export function useSearchHistory() {
 const { user } = useAuth()
 const queryClient = useQueryClient()

 // Always initialize sessionStorage hook (used for anonymous path)
 const [sessionHistory, setSessionHistory] = useSessionStorage<
   SearchHistoryItem[]
 >('search-history', [])

 // Backend query — only enabled when logged in
 const backendQuery = useQuery({
   queryKey: ['search-history', user?.id],
   queryFn: async () => {
     const token = await getToken()
     if (!token) return []
     const res = await fetch(`${API_BASE}/history`, {
       headers: { Authorization: `Bearer ${token}` },
```

```
    })
    if (!res.ok) return []
    const data = await res.json()
    return data.map(
      (item: {
        id: string
        city_name: string
        lat: number
        lon: number
        country: string
        state?: string
        searched_at: string
      }) => ({
        id: item.id,
        query: item.city_name,
        name: item.city_name,
        lat: item.lat,
        lon: item.lon,
        country: item.country,
        state: item.state,
        searchedAt: Date.parse(item.searched_at),
      }),
    ) as SearchHistoryItem[]
  },
  enabled: !!user,
  staleTime: 5 * 60 * 1000,
})

const history = user ? (backendQuery.data ?? []) : sessionHistory

const addToHistory = useMutation({
  mutationFn: async (
    search: Omit<SearchHistoryItem, 'id' | 'searchedAt'>,
  ) => {
    if (!user) {
      // Anonymous: sessionStorage
      const newSearch: SearchHistoryItem = {
        ...search,
        id: `${search.lat}-${search.lon}-${Date.now()}`,
        searchedAt: Date.now(),
      }
      const filtered = sessionHistory.filter(
        (item) => !(item.lat === search.lat && item.lon === search.lon),
      )
      setSessionHistory([newSearch, ...filtered].slice(0, 10))
      return
    }

    // Authenticated: backend
    const token = await getToken()
    if (!token) return
    await fetch(`${API_BASE}/history`, {
```

```
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
          Authorization: `Bearer ${token}`,
        },
        body: JSON.stringify({
          city_name: search.name,
          lat: search.lat,
          lon: search.lon,
          country: search.country,
          state: search.state,
        }),
      })
    },
    onSuccess: () => {
      if (user) {
        queryClient.invalidateQueries({ queryKey: ['search-history', user.id]
})
      }
    },
  })

  const clearHistory = useMutation({
    mutationFn: async () => {
      if (!user) {
        setSessionHistory([])
        return
      }
      const token = await getToken()
      if (!token) return
      await fetch(`${API_BASE}/history`, {
        method: 'DELETE',
        headers: { Authorization: `Bearer ${token}` },
      })
    },
    onSuccess: () => {
      if (user) {
        queryClient.invalidateQueries({ queryKey: ['search-history', user.id]
})
      }
    },
  })

  return {
    history,
    addToHistory,
    clearHistory,
  }
}
```

На бек частині

```
import { Hono } from 'hono'
import { createUserClient } from '../lib/supabase.ts'
```

```
import { authMiddleware } from '../middleware/auth.ts'
import type { SearchHistoryItem } from '../types/index.ts'
import type { AppEnv } from '../types/hono.ts'

const history = new Hono<AppEnv>()

history.use('/*', authMiddleware)

history.get('/', async (c) => {
 const user = c.get('user')!
 const db = createUserClient(c.get('token')!)

 const { data, error } = await db
   .from('search_history')
   .select('*')
   .eq('user_id', user.id)
   .order('searched_at', { ascending: false })
   .limit(10)

 if (error) return c.json({ error: error.message }, 500)

 return c.json(data as SearchHistoryItem[])
})

history.post('/', async (c) => {
 const user = c.get('user')!
 const db = createUserClient(c.get('token')!)
 const    body    =    await    c.req.json<Omit<SearchHistoryItem,    'id'    |
'searched_at'>>()

 if (!body.city_name || body.lat == null || body.lon == null) {
   return c.json({ error: 'city_name, lat, lon are required' }, 400)
 }

 // Remove duplicate entry for same coordinates before inserting
 await db
   .from('search_history')
   .delete()
   .eq('user_id', user.id)
   .eq('lat', body.lat)
   .eq('lon', body.lon)

 const { data, error } = await db
   .from('search_history')
   .insert({
     user_id: user.id,
     city_name: body.city_name,
     country: body.country,
     state: body.state,
     lat: body.lat,
     lon: body.lon,
   })
```

```
    .select()
    .single()

  if (error) return c.json({ error: error.message }, 500)

  // Keep only the 10 most recent
  const { data: allRows } = await db
    .from('search_history')
    .select('id')
    .eq('user_id', user.id)
    .order('searched_at', { ascending: false })

  if (allRows && allRows.length > 10) {
    const toDelete = allRows.slice(10).map((r: { id: string }) => r.id)
    await db.from('search_history').delete().in('id', toDelete)
  }

  return c.json(data, 201)
})

history.delete('/', async (c) => {
  const user = c.get('user')!
  const db = createUserClient(c.get('token')!)

  const { error } = await db
    .from('search_history')
    .delete()
    .eq('user_id', user.id)

  if (error) return c.json({ error: error.message }, 500)

  return c.json({ message: 'History cleared' })
})

export default history
```

Лістинг Б.3 AI-рекомендації одягу та активностей

На фронт частині (компонент картки)

```
import { useTranslation } from 'react-i18next'
import { Sparkles, LogIn } from 'lucide-react'
import {
  Card,
  CardHeader,
  CardTitle,
  CardAction,
  CardContent,
} from '@/shared/components/ui/card'
import { Skeleton } from '@/shared/components/ui/skeleton'
import { useRecommendations } from '../hooks/use-recommendations'
import { useAuth } from '@/features/auth/context/auth-context'
```

```
import type { Coordinates, WeatherData } from '@/features/weather/api/types'

interface RecommendationCardProps {
 coords: Coordinates | null
 weather: WeatherData | null | undefined
 cityName: string
 country?: string
}

export function RecommendationCard({
 coords,
 weather,
 cityName,
 country,
}: RecommendationCardProps) {
 const { t } = useTranslation()
 const { user } = useAuth()
 const { data, isLoading, isError } = useRecommendations({
   coords,
   weather,
   cityName,
   country,
 })

 if (isError) return null

 return (
   <Card data-testid="RecommendationCard">
     <CardHeader>
       <CardTitle className="flex items-center gap-2">
         <Sparkles className="h-4 w-4 text-primary" />
         {t('recommendations.title')}
       </CardTitle>
       {data?.is_ai && (
         <CardAction>
           <span  className="rounded-full  bg-primary/10  px-2  py-0.5  text-xs
font-medium text-primary">
             {t('recommendations.aiPowered')}
           </span>
         </CardAction>
       )}
     </CardHeader>

     <CardContent>
       {!user ? (
         <p
           className="flex items-center gap-2 text-sm text-muted-foreground"
           data-testid="RecommendationLoginPrompt"
         >
           <LogIn className="h-4 w-4 shrink-0" />
           {t('recommendations.loginPrompt')}
         </p>
```

```
        ) : isLoading ? (
          <div className="space-y-2">
            <Skeleton className="h-4 w-full" />
            <Skeleton className="h-4 w-5/6" />
            <Skeleton className="h-4 w-4/6" />
          </div>
        ) : (
          <p className="text-sm leading-relaxed text-muted-foreground">
            {data?.content || t('recommendations.unavailable')}
          </p>
        )}
      </CardContent>
    </Card>
  )
}
```

На фронт частині (хук та API-клієнт)

```
import { useQuery } from '@tanstack/react-query'
import { useTranslation } from 'react-i18next'
import {
 fetchRecommendations,
 type RecommendationParams,
} from '../api/recommendations'
import type { Coordinates } from '@/features/weather/api/types'
import type { WeatherData } from '@/features/weather/api/types'
import { useAuth } from '@/features/auth/context/auth-context'

interface UseRecommendationsParams {
 coords: Coordinates | null
 weather: WeatherData | null | undefined
 cityName: string
 country?: string
}

export function useRecommendations({
 coords,
 weather,
 cityName,
 country,
}: UseRecommendationsParams) {
 const { i18n } = useTranslation()
 const { user, preferences } = useAuth()
 const lang = i18n.language as string

 const params: RecommendationParams | null =
   coords && weather
     ? {
         lat: coords.lat,
         lon: coords.lon,
         lang,
         temp: weather.main.temp,
```

```
          feels_like: weather.main.feels_like,
          humidity: weather.main.humidity,
          wind_speed: weather.wind.speed,
          description: weather.weather[0]?.description ?? '',
          city: cityName,
          country,
        }
      : null

 return useQuery({
   queryKey: ['recommendation', params],
   queryFn: () => fetchRecommendations(params!),
   // Only fetch when logged in AND onboarding is complete
   enabled: !!params && !!user && preferences?.onboarding_completed === true,
   staleTime: 60 * 60 * 1000, // 1 hour — matches backend cache
   retry: false,
 })
}


import { supabase } from '@/lib/supabase'

const API_BASE = import.meta.env.VITE_API_BASE_URL as string

export interface RecommendationResponse {
 content: string
 is_ai: boolean
 generated_at: string
 expires_at: string
 cached: boolean
}

export interface RecommendationParams {
 lat: number
 lon: number
 lang: string
 temp: number
 feels_like: number
 humidity: number
 wind_speed: number
 description: string
 city: string
 country?: string
}

export async function fetchRecommendations(
 params: RecommendationParams,
): Promise<RecommendationResponse> {
 const query = new URLSearchParams({
   lat: params.lat.toString(),
   lon: params.lon.toString(),
   lang: params.lang,
```

```
    temp: params.temp.toString(),
    feels_like: params.feels_like.toString(),
    humidity: params.humidity.toString(),
    wind_speed: params.wind_speed.toString(),
    description: params.description,
    city: params.city,
    ...(params.country ? { country: params.country } : {}),
  })

  const headers: HeadersInit = {}

  // Attach auth token if user is logged in
  const { data } = await supabase.auth.getSession()
  if (data.session?.access_token) {
    headers['Authorization'] = `Bearer ${data.session.access_token}`
  }

  const res = await fetch(`${API_BASE}/recommendations?${query.toString()}`, {
    headers,
  })

  if (!res.ok) {
    throw new Error(`Recommendations request failed: ${res.status}`)
  }

  return res.json()
}
```

На бек частині (маршрут із кешуванням)

```
import { Hono } from 'hono'
import { createUserClient } from '../lib/supabase.ts'
import { generateAIRecommendation } from '../lib/claude.ts'
import { getRuleBasedRecommendation } from '../lib/recommendations.ts'
import { optionalAuthMiddleware } from '../middleware/auth.ts'
import type { WeatherSnapshot } from '../types/index.ts'
import type { AppEnv } from '../types/hono.ts'

const recommendations = new Hono<AppEnv>()

recommendations.get('/', optionalAuthMiddleware, async (c) => {
  const user = c.get('user')

  const lat = parseFloat(c.req.query('lat') ?? '')
  const lon = parseFloat(c.req.query('lon') ?? '')
  const lang = (c.req.query('lang') ?? 'en') as 'en' | 'de' | 'ua'
  const temp = parseFloat(c.req.query('temp') ?? '0')
  const feelsLike = parseFloat(c.req.query('feels_like') ?? '0')
  const humidity = parseInt(c.req.query('humidity') ?? '50')
  const windSpeed = parseFloat(c.req.query('wind_speed') ?? '0')
  const description = c.req.query('description') ?? ''
  const icon = c.req.query('icon') ?? ''
```

```
const cityName = c.req.query('city') ?? 'your city'
const country = c.req.query('country') ?? ''

if (isNaN(lat) || isNaN(lon)) {
  return c.json({ error: 'lat and lon query params are required' }, 400)
}

const weather: WeatherSnapshot = {
  temp,
  feels_like: feelsLike,
  humidity,
  wind_speed: windSpeed,
  description,
  icon,
}

const now = new Date()
const expires = new Date(now.getTime() + 60 * 60 * 1000) // +1 hour

// Anonymous users → rule-based, no caching
if (!user) {
  const content = getRuleBasedRecommendation({
    ...weather,
    hour: now.getHours(),
    language: lang,
  })
  return c.json({
    content,
    is_ai: false,
    generated_at: now.toISOString(),
    expires_at: expires.toISOString(),
    cached: false,
  })
}

const db = createUserClient(c.get('token')!)

// Fetch user profile + top cities in parallel
const [{ data: prefsRow }, { data: historyRows }] = await Promise.all([
  db
    .from('user_preferences')
    .select('first_name, age, activity_type, onboarding_completed')
    .eq('user_id', user.id)
    .single(),
  db
    .from('search_history')
    .select('city_name')
    .eq('user_id', user.id)
    .order('searched_at', { ascending: false })
    .limit(20),
])
```

```
const topCities = [
  ...new Set((historyRows ?? []).map((r: { city_name: string }) =>
r.city_name)),
].slice(0, 3)

// Don't generate until onboarding is complete — profile is incomplete
if (!prefsRow?.onboarding_completed) {
  return c.json({ error: 'Onboarding not completed' }, 403)
}

// Check cache only after confirming the user is fully onboarded
const { data: cached } = await db
  .from('recommendations')
  .select('content, is_ai, generated_at, expires_at')
  .eq('user_id', user.id)
  .eq('language', lang)
  .gt('expires_at', now.toISOString())
  .not('content', 'is', null)
  .neq('content', '')
  .order('generated_at', { ascending: false })
  .limit(1)
  .single()

if (cached) {
  return c.json({ ...cached, cached: true })
}

const userProfile = {
  firstName: prefsRow.first_name ?? undefined,
  age: prefsRow.age ?? undefined,
  activityType: prefsRow.activity_type ?? undefined,
}

// Generate AI recommendation
let content: string
let isAI = true

try {
  content = await generateAIRecommendation({
    weather,
    cityName,
    country,
    language: lang,
    topCities,
    userProfile,
  })
} catch (err) {
  console.error('[recommendations] Claude generation failed:', err)
  // Fallback to rule-based if Claude fails
  content = getRuleBasedRecommendation({
    ...weather,
    hour: now.getHours(),
```

```
      language: lang,
    })
    isAI = false
  }

  // Persist to cache (only if content is non-empty)
  if (content) await db.from('recommendations').insert({
    user_id: user.id,
    lat,
    lon,
    language: lang,
    weather_snapshot: weather,
    content,
    is_ai: isAI,
    generated_at: now.toISOString(),
    expires_at: expires.toISOString(),
  })

  return c.json({
    content,
    is_ai: isAI,
    generated_at: now.toISOString(),
    expires_at: expires.toISOString(),
    cached: false,
  })
})

// Invalidate the recommendation cache for the current user
recommendations.delete('/', optionalAuthMiddleware, async (c) => {
  const user = c.get('user')
  if (!user) return c.json({ error: 'Unauthorized' }, 401)

  const db = createUserClient(c.get('token')!)
  await db.from('recommendations').delete().eq('user_id', user.id)

  return c.json({ message: 'Cache cleared' })
})

export default recommendations
```

На бек частині (виклик Claude API)

```
import Anthropic from '@anthropic-ai/sdk'
import type { WeatherSnapshot } from '../types/index.ts'

const client = new Anthropic({
  apiKey: process.env.ANTHROPIC_API_KEY,
})

const LANGUAGE_NAMES: Record<string, string> = {
  en: 'English',
  de: 'German',
```

```
 ua: 'Ukrainian',
}

function getSeason(month: number): string {
 if (month >= 3 && month <= 5) return 'spring'
 if (month >= 6 && month <= 8) return 'summer'
 if (month >= 9 && month <= 11) return 'autumn'
 return 'winter'
}

function getTimeOfDay(hour: number): string {
 if (hour >= 6 && hour < 12) return 'morning'
 if (hour >= 12 && hour < 17) return 'afternoon'
 if (hour >= 17 && hour < 21) return 'evening'
 return 'night'
}

const ACTIVITY_LABELS: Record<string, string> = {
 sedentary: 'sedentary (office/studying)',
 light: 'light (casual walks, cycling)',
 active: 'active (gym, hiking)',
 athletic: 'athletic (sports, intensive training)',
}

export async function generateAIRecommendation(params: {
 weather: WeatherSnapshot
 cityName: string
 country: string
 language: string
 topCities: string[]
 userProfile: { firstName?: string; age?: number; activityType?: string }
}): Promise<string> {
 const { weather, cityName, country, language, topCities, userProfile } =
params
 const now = new Date()
 const season = getSeason(now.getMonth() + 1)
 const timeOfDay = getTimeOfDay(now.getHours())
 const langName = LANGUAGE_NAMES[language] ?? 'English'

 const locationLabel = country ? `${cityName}, ${country}` : cityName
 const activityLabel = ACTIVITY_LABELS[userProfile.activityType ?? ''] ??
userProfile.activityType ?? 'general'
 const ageInfo = userProfile.age ? ` (${userProfile.age} years old)` : ''
 const citiesInfo =
   topCities.length > 0
     ? `\n- Frequently checks weather in: ${topCities.join(', ')}`
     : ''

 const prompt = `You are a personal weather advisor. Write a warm, personalized
recommendation for ${userProfile.firstName ?? 'the user'}.

Current conditions in ${locationLabel}:
```

```
- Temperature: ${weather.temp}°C (feels like ${weather.feels_like}°C)
- Weather: ${weather.description}
- Humidity: ${weather.humidity}%
- Wind speed: ${weather.wind_speed} m/s
- Time of day: ${timeOfDay}, ${season}

User profile:
- Name: ${userProfile.firstName ?? 'unknown'}${ageInfo}
- Lifestyle: ${activityLabel}${citiesInfo}

Requirements:
- Open with the user's first name (e.g. "${userProfile.firstName}," or "Hi
${userProfile.firstName},")
- Write at least 3 complete sentences
- Tailor the advice specifically to a ${activityLabel} lifestyle — mention
concrete activity-relevant tips
- Reference the local context of ${country || cityName} and the current
${season} season
- Cover clothing/gear and any activity or health tips relevant to conditions
- Do NOT use markdown — no headings, no bullet points, no bold or italic text.
Plain text only.

Respond in ${langName} only.`

 const message = await client.messages.create({
   model: 'claude-haiku-4-5-20251001',
   max_tokens: 500,
   temperature: 0.7,
   messages: [{ role: 'user', content: prompt }],
 })

 const block = message.content[0]
 if (block.type !== 'text') {
   throw new Error('Unexpected response type from Claude')
 }

 const text = block.text.trim()
 if (!text) {
   throw new Error('Claude returned empty response text')
 }
 return text
}
```

ДОДАТОК В

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

Декларація про академічну доброчесність та використання ШІ

Я, Прокопенко Павло Геннадійович, здобувач НУБіП України, усвідомлюючи відповідальність за порушення академічної доброчесності, цією заявою підтверджую, що:

1. Моя бакалаврська кваліфікаційна робота на тему «Програмне забезпечення системи моніторингу кліматичних параметрів для планування повсякденних активностей» є результатом моєї особистої праці.
2. Усі запозичення з текстів інших авторів мають відповідні посилання згідно з чинними стандартами.
3. Щодо використання інструментів ШІ зазначаю, що (обрати необхідне):

- [ ] інструменти генеративного ШІ не використовувалися.
- [ ] інструменти ШІ (вказати назву: ChatGPT, Gemini, Claude, DeepL, _____інші (вказати назву)) були використані для:
  - [ ] перекладу іншомовних джерел;
  - [ ] стилістичного редагування та перевірки граматики;
  - [ ] допомоги у написанні/відлагодженні програмного коду;
  - [ ] інше: _____.

Я розумію, що надання недостовірної інформації може призвести до скасування результатів захисту та відрахування з числа здобувачів НУБіП України.

« » 2026 р. _____ **Павло ПРОКОПЕНКО**
(підпис)