Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп’ютерних наук

Допущено до захисту
Завідувач кафедри комп’ютерних наук
доктор технічних наук, професор
Андрій БОНДАРЧУК
«01» червня 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня «бакалавр»
спеціальність 122 Комп’ютерні науки
освітня програма 122.00.01 Інформатика

Тема: ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ НАВІГАЦІЇ ПО КУЛЬТУРНИХ ОБ’ЄКТАХ МІСТА КИЄВА НА ПЛАТФОРМІ ANDROID

Виконав
студент групи Інб-2-22-4.0д
Лучко Андрій Ігорович
_____
(підпис)

Науковий керівник
кандидат технічних наук,
доцент
Яскевич В.О.
_____
(підпис)

Київ – 2026

Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп’ютерних наук

«Затверджую»
Завідувач кафедри комп’ютерних наук
доктор технічних наук, професор
Андрій БОНДАРЧУК
« 31 » жовтня 2025 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА
«ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ НАВІГАЦІЇ ПО КУЛЬТУРНИХ ОБ’ЄКТАХ МІСТА КИЄВА НА ПЛАТФОРМІ ANDROID»

Виконавець – студент групи ІНб-2-22-4.0д,
спеціальності 122 Комп’ютерні науки,
освітньої програми 122.00.01 Інформатика
Лучко Андрій Ігорович

1. Вихідні дані: Набір геопросторових даних про культурні об'єкти м. Києва з відкритих джерел (OpenStreetMap, Портал відкритих даних Києва). Офіційна документація Android SDK, бібліотек MapLibre GL Native, Android Room (SQLite FTS4) та Google ML Kit. Наукові публікації щодо розв'язання задачі комівояжера (TSP) та алгоритмів оптимізації багатоточкових маршрутів (Nearest Neighbor, 2-opt).

2. Основні завдання: Здійснити аудит існуючих навігаційних продуктів та проаналізувати специфіку агрегації відкритих геоданих. Обґрунтувати мету, об'єкт, предмет та обрати методи дослідження. Спроєктувати неблокуючу Offline-first архітектуру мобільного клієнта за патерном MVVM з ізоляцією рівнів Data, Domain та Presentation. Розробити алгоритмічне забезпечення для багатокритеріальної фільтрації локацій та контентних рекомендацій на базі профілю користувача. Спроєктувати та імплементувати механізми маршрутизації через кілька проміжних точок (алгоритм 2-opt) та інтегрувати нативне C++ ядро для геодезичних обчислень. Імплементувати модулі локального кешування бази (FTS4), фонової тіньової синхронізації та автономного машинного перекладу динамічного контенту (ML Kit). Провести експериментальне тестування швидкодії застосунку, профілювання затримок та ефективності маршрутної оптимізації. Оформити роботу відповідно до затверджених вимог. Підготувати тези доповіді за темою роботи.

3. Пояснювальна записка: Обсяг близько 55 стор. формату А4 комп’ютерного набору з дотриманням вимог стандарту і методичних рекомендацій кафедри.

4. Графічні матеріали: Комп'ютерна презентація доповіді.

5. Додатки: Фрагменти вихідного програмного коду (алгоритми фільтрації, усунення перетинів маршруту, JNI-міст до C++ ядра, SQL-тригери бази даних).

6. Строк подання роботи на кафедру: *«1» червня 2026 р*.

Науковий керівник	Виконавець:
к.т.н., доцент	
_____ Яскевич В.О.	_____ Лучко А.І.
31.10.2025 дата	31.10.2025 дата

Календарний план

№	Етап виконання роботи	Термін виконання	Примітка
1	Затвердження теми кваліфікаційної роботи бакалавра	31.10.25	виконано
2	Аналіз проблеми, вивчення аналогів, формування цілей та завдань	01.11.25 - 11.01.26	виконано
3	Аналіз предметної області, існуючих навігаційних застосунків та специфіки агрегації відкритих геопросторових даних Києва	26.01.26 - 15.03.26	виконано
4	Дослідження алгоритмічних підходів для багатокритеріальної фільтрації локацій, оптимізації маршрутів та архітектури Offline-first	16.03.26 - 31.03.26	виконано
5	Розробка архітектури мобільного клієнта, локальної бази даних та алгоритмічного ядра для побудови багатоточкових пішохідних маршрутів	01.04.26 - 15.04.26	виконано
6	Тестування швидкодії розробленого програмного забезпечення, профілювання затримок UI та ефективності маршрутної оптимізації	16.04.26 - 30.04.26	виконано
7	Впровадження векторних офлайн-карт, механізмів локального кешування та модуля автономного машинного перекладу контенту	01.05.26 - 15.05.26	виконано
8	Підготовка пояснювальної записки, графічних матеріалів, додатків.	25.05.26 - 04.06.26	виконано
9	Захист кваліфікаційної роботи	19.06.26	

«Затверджую»

Науковий керівник — к.т.н., доцент, Яскевич В.О.

Індивідуальний план склав — Лучко А.І.

РЕФЕРАТ

Кваліфікаційна робота бакалавра: 77 с., 14 рис., 3 табл., 66 джерел, 2 додатки.

Актуальність. Сучасні масові картографічні сервіси орієнтовані переважно на автомобілістів та критично залежать від стабільного інтернет-з'єднання. Це створює незручності для пішохідного туризму, особливо в умовах міжнародного роумінгу або в зонах слабкого покриття мережі. Велика кількість культурних пам'яток Києва потребує створення спеціалізованого автономного (Offline-first) інструменту, здатного будувати оптимізовані багатоточкові маршрути та надавати інформацію про об'єкти незалежно від якості зв'язку.

Об’єкт дослідження: процес навігації та надання геопросторової інформації користувачам в умовах обмеженого або відсутнього інтернет-з'єднання.

Предмет дослідження: інформаційні технології, алгоритми оптимізації багатоточкових маршрутів та інструменти розробки Offline-first мобільних застосунків для платформи Android.

Мета роботи: розробка архітектури та програмна імплементація автономного мобільного застосунку для навігації по культурних об'єктах міста Києва з використанням локальних технологій машинного навчання та математичних алгоритмів маршрутизації.

Завдання:

1. Здійснити огляд існуючих навігаційних застосунків та обґрунтувати вибір технологій.
2. Спроєктувати архітектуру мобільного застосунку на базі патерну MVVM з дотриманням концепції Offline-first.
3. Інтегрувати картографічний рушій MapLibre GL Native для відображення карт за допомогою векторних тайлів.
4. Реалізувати алгоритми побудови та оптимізації багатоточкових пішохідних маршрутів із застосуванням рушія OSRM.

5. Розробити модуль локального перекладу інформації про об'єкти з використанням моделей Google ML Kit.

6. Провести тестування розробленого програмного забезпечення та підтвердити коректність роботи впроваджених алгоритмів навігації і локалізації.

Основні результати. Створено мобільний застосунок для платформи Android, який забезпечує навігацію культурним простором міста Києва. Успішно реалізовано механізм рендерингу векторних карт, швидкий повнотекстовий пошук локацій через SQLite FTS4 та побудову оптимізованих пішохідних маршрутів за допомогою OSRM. Інтегровано систему машинного навчання для локального перекладу контенту, що значно підвищує зручність використання застосунку іноземними туристами.

Практичне значення дослідження. Розроблений програмний продукт є готовим до використання туристами. Запропоновані в роботі архітектурні та алгоритмічні рішення (поєднання MapLibre, OSRM та локального ML) можуть бути використані розробниками як основа для створення подібних високопродуктивних застосунків для інших міст та туристичних маршрутів.

Ключові слова: android, kotlin, навігація, offline-first, maplibre, OSRM, Google ML Kit, sqlite fts4, задача комівояжера.

.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Скорочення	Розшифрування
A*	Алгоритм A-star – пошук найкоротшого шляху на графі
API	Application Programming Interface – прикладний програмний інтерфейс
DI	Dependency Injection – інжекція залежностей
LRU	Least Recently Used – політика витіснення кешу
MapsForge	Java-бібліотека для офлайн векторних карт OSM
MVVM	Model-View-View-Model – архітектурний патерн
ODbL	Open Database License – ліцензія бази даних OSM
OSM	OpenStreetMap – проєкт відкритої карти
POI	Point of Interest – цікава локація
Room	Android Room – ORM над SQLite
SDK	Software Development Kit – набір інструментів розробника
TSP	Travelling Salesman Problem – задача комівояжера
TTL	Time To Live – час життя запису
UI	User Interface – інтерфейс користувача
UX	User Experience – користувацький досвід
WorkManager	Планувальник відкладених робіт в Android
OSRM	OSRM (Open Source Routing Machine) – високопродуктивний рушій маршрутизації з відкритим вихідним кодом.

ВСТУП

Актуальність теми. На території Києва зосереджено понад 400 музеїв, театрів, архітектурних ансамблів і креативних кластерів [8]. Пішоходу, особливо туристу, утримати в голові координати, графіки та категорії такого масиву без цифрового інструменту неможливо [62]. Google Maps і Waze орієнтовані на водіїв: пішохідні маршрути будуються без оптимізації обходу та без перевірки, чи працює заклад. У зонах нестабільного покриття (метро, щільна забудова, роумінг) хмарна маршрутизація ненадійна. Ці обмеження зумовили розробку геоінформаційної системи, здатної працювати автономно й давати персоналізовану навігацію.

Мета дослідження – спроєктувати та програмно реалізувати мобільний застосунок на платформі Android для автономної навігації культурним простором Києва. Головний акцент – на інтеграції векторних офлайн-карт, алгоритмів оптимізації багатоточкових маршрутів і рекомендаційного рушія, а саме:

- забезпечити безперебійний доступ до просторових даних та маршрутизації без Інтернету;
- суттєво скоротити час пошуку релевантних культурних локацій;
- генерувати персоналізовані сценарії обходу пам'яток з урахуванням інтересів людини, геодезичної відстані та актуального розкладу роботи об'єктів.

Завдання дослідження. Під мету було сформовано п'ять інженерних завдань:

- Провести аудит існуючих навігаційних продуктів і специфіку агрегації відкритих геоданих (OpenStreetMap, міські портали).
- Спроєктувати неблокуючу архітектуру мобільного клієнта за патерном MVVM з ізоляцією рівнів Data, Domain та Presentation [21].

- Розробити алгоритмічне забезпечення для багатокритеріальної фільтрації, контентних рекомендацій та маршрутизації через кілька проміжних точок.
- Імплементувати модулі локального кешування бази та фонової тіньової синхронізації.
- Провести експериментальне тестування: профілювання UI-затримок, оцінка якості рекомендацій (Precision@K, Recall@K, NDCG [43]), вимірювання ефективності маршрутної оптимізації.

Об'єкт дослідження – інформаційні процеси обробки геопросторових даних, маршрутизації та алгоритмічної персоналізації в мобільних системах.

Предмет дослідження – моделі, математичні методи та програмні засоби побудови автономних навігаційних застосунків і рекомендаційних систем на платформі Android.

Методи дослідження. Задача комівояжера розв'язується через теорію графів і евристики: «найближчий сусід» та локальна оптимізація 2-opt [42, 55]. Персоналізована видача ґрунтується на контентній фільтрації (зважена лінійна модель зі Score за профілем інтересів). Верифікація – емпіричне тестування: метрики Precision@K, Recall@K, NDCG [43], профілювання UI (бюджет 16.6 мс/кадр) та вимірювання часу оптимізації залежно від кількості вузлів.

Практичне значення одержаних результатів. Теоретичні та алгоритмічні напрацювання доведено до працюючого застосунку, що зберігає базовий функціонал (пошук, фільтрація, карта, маршрути) без мережі. Архітектурні рішення (DAO-абстракції Room, інжекція Hilt, реактивний State у Compose) придатні для перенесення в інші геоінформаційні проєкти.

Результати роботи апробовано шляхом публікації тез доповіді на конференції Інформаційні технології – 2026: зб. матеріалів XII Всеукраїнської науково-практичної конференції молодих учених, 14 трав. 2026 р., м. Київ / Київський столичний університет імені Бориса Грінченка. с. 136–141 [65].

РОЗДІЛ 1
ТЕОРЕТИЧНІ ЗАСАДИ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Щільність історичної забудови та культурних об'єктів у центрі Києва створює специфічну навігаційну проблему. Масові продукти (Google Maps, Waze, Apple Maps) оптимізовані переважно для автомобільного трафіку. Їхній алгоритм відмальовки POI (Point of Interest) не має жорсткої фільтрації за культурним критерієм – екран перевантажується маркерами комерційної інфраструктури (АЗС, зупинки, заклади харчування). Для пішохода це формує надлишковий когнітивний шум. На апаратному рівні фоновий рендеринг нефільтрованого масиву маркерів створює додаткове навантаження на heap (алокація об'єкта під кожен маркер) та GPU, що прискорює розряджання акумулятора. Побудова логістичного ланцюга через 5–8 культурних пам'яток із динамічною перевіркою їхнього розкладу роботи у таких системах відсутня.

Архітектура розроблюваного рішення підпорядкована кільком технічним критеріям, що відрізняють його від загальних навігаторів.

1. Евристичний і цільовий пошук. Рушій фільтрації має відсікати загальний шум і пріоритетно віддавати лише об'єкти культурного фонду. На рівні SQL для цього створено індексований стовпець category_type у таблиці Places – SELECT із WHERE-клаузою йде з WHERE-клаузою без повного табличного сканування.

2. Реактивна фільтрація без блокування Main Thread. Багатокритеріальна вибірка (тип закладу, дистанція, «відкрито зараз») вимагає агрегації, яка здатна зайняти кілька десятків мілісекунд. Якщо викликати її на головному потоці, кадр рендерингу вийде за бюджет 16.6 мс, і користувач побачить фріз. Тож обчислення витісняються у Dispatchers.Default, а результат транслюється через StateFlow.

3. Алгоритмічна персоналізація. За першого візиту до великого міста турист потрапляє у стан «перевантаження вибором» (Choice Overload).

Рекомендаційний рушій з лінійною Scoring-моделлю формує ранжовану видачу, враховуючи профіль інтересів, рейтинг та геодезичну близькість, – когнітивне навантаження падає.

4. Offline-first підхід. У метро чи серед щільної забудови стільниковий сигнал нестабільний, тому маршрутизація повинна працювати локально. Векторні тайли завантажуються один раз і далі карта рендериться силами GPU смартфона.

Разом із технічними обмеженнями є географічна специфіка. Київ розташований по обидва береги Дніпра, і густота культурних об'єктів на правому березі значно перевищує лівобережну. Алгоритм автоматичного визначення берега спирається на значення довготи: якщо координата об'єкта перетинає поріг 30.58° на схід, локація маркується лівобережною. Ця мітка потрапляє в поле riverBank і дає змогу фільтрувати без геометричних обчислень у рантаймі. Туристу, який проводить день на Подолі, не показуються лівобережні POI – і шум на карті зменшується. Ця інформація також використовується рекомендаційним рушієм: якщо користувач знаходиться на правому березі, пріоритет отримують локації на тому ж березі, щоб маршрут не перетинав Дніпро без явного наміру.

У наступних підрозділах формалізовано структурну модель даних для опису київських локацій, визначено профіль цільової аудиторії та обґрунтовано вибір конкретних алгоритмів. Отримані результати використовуються у другому розділі при проєктуванні локального кешу, реактивних фільтрів і модуля маршрутизації.

1.1 Основи проєктування Android застосунків та архітектурні підходи

За даними StatCounter (жовтень 2025), Android охоплює 65,7 % українського мобільного ринку [41]. Розробка під платформу з такою часткою ринку накладає ряд інженерних обмежень: обмежений обсяг оперативної пам'яті (типово 3–6 ГБ, значну частину яких резервує ОС), чутливість

акумулятора до фонових процесів [35]. Якщо застосунок перевищує ліміт пам'яті, Android OOM Killer примусово завершує процес. Блокуюча операція на Main Thread довше 5 секунд спричиняє системний діалог ANR (Application Not Responding). Обидва сценарії вимагають систематичного профілювання коду.

На практиці стабільність тримається на розподілі відповідальності (Separation of Concerns). Інтерфейс реагує на дотики без затримок, а поворот екрана не скидає введені дані – ViewModel переживає зміну конфігурації, бо його життєвий цикл прив'язаний до ViewModelStoreOwner. Витоки пам'яті контролюються через LeakCanary у Debug-збірці та через суворе використання viewModelScope [38] замість глобальних CoroutineScope.

На рівні Presentation Layer застосовано MVVM (Model-View-ViewModel) [21, 37, 57] разом із Clean Architecture. Код розбито на три шари, кожен з яких ізольований. Presentation тримає Composable-функції та ViewModel-класи, фіксує життєвий цикл екранів і делегує логіку нижче. Domain – серцевина з чистою бізнес-логікою: UseCase-класи, абстрактні Repository-інтерфейси, доменні моделі без залежностей від Room чи Retrofit. Data закриває мережу, кеш, парсинг JSON, наповнення репозиторіїв конкретними реалізаціями PlacesRepositoryImpl, RoutesRepositoryImpl.

Мовою розробки обрано Kotlin [23] – насамперед через Null Safety, завдяки якій потенційні NullPointerException перехоплюються ще компілятором, а не в runtime під час навігації. Паралельні обчислення побудовані на Coroutines разом із Flow [24]. Важкі серверні запити та алгоритми маршрутизації витісняються у фонові диспетчери (Dispatchers.IO для мережі, Dispatchers.Default для CPU-задач). Main Thread не блокується. Окремий інженерний виклик – підтримка тривалих обчислень всередині мобільного процесу. Тригонометричні операції формули Haversine (sin, cos, asin, sqrt), які необхідні для підрахунку відстаней між тисячами пар координат, створюють відчутне навантаження на JVM-інтерпретатор. Щоб уникнути ситуації, коли обчислення «з'їдає» бюджет кадру 16.6 мс і провокує

мікрофризі під час скролу карти, ресурсомісткі ділянки винесено у нативну бібліотеку C++ (NDK). Kotlin-код звертається до неї через JNI-інтерфейс, а у разі помилки завантаження нативної бібліотеки автоматично перемикається на чисто Kotlin-реалізацію тієї самої формули. Для користувача різниця непомітна, але нативний варіант обробляє масив із 500 точок приблизно вчетверо швидше. Це помітно на бюджетних пристроях, де CPU тактується нижче і кожна мілісекунда на рахунку.

Інфраструктурна база – компоненти Jetpack [22]. Room [25, 26] замінив прямі SQLite-маніпуляції: DAO-інтерфейси з @Query-анотаціями проходять валідацію на етапі kapt-компіляції. Стан UI тримається у StateFlow всередині ViewModel. Фонове оновлення каталогу виконує WorkManager [27] – навіть після kill процесу система поставить задачу в чергу повторно. Навігація між екранами зібрана у Navigation Component [28]. Візуальна складова слідує гайдлайнам Material Design 3 [32]: збільшені зони натискання (≥ 48 dp) знижують кількість промахів, коли людина взаємодіє з екраном на ходу. Окремої уваги потребує механізм управління залежностями. Hilt побудований на принципі компілятивної DI: граф об'єктів будується під час компіляції, а не в рантаймі. KSP-процесор аналізує всі @Inject-анотації, перевіряє повноту графу та генерує фабрики. Якщо розробник забув надати залежність або створив циклічне посилання – збирання проєкту завершиться помилкою, а не крашем у момент запуску на пристрої. Це принципово для проєкту з понад 80 Kotlin-файлами: ручне відстеження залежностей між модулями стало б джерелом регулярних runtime-помилок. На практиці Hilt-модулі поділено на три файли за сферою відповідальності: мережевий стек, база даних і Retrofit-клієнт. Кожен з них можна перевизначити окремо у тестовому оточенні без впливу на інші частини DI-графу. Під час написання інструментальних тестів замість справжнього Retrofit-клієнта підставляється MockWebServer, а замість SQLite-бази – in-memory версія з тими самими DAO-інтерфейсами. Завдяки такому підходу тестове покриття перевіряє бізнес-логіку, а не поведінку зовнішніх сервісів.

Зв'язка MVVM та DI (Hilt) підтримує модульність проєкту. При необхідності заміни картографічного рушія (MapsForge [15] → osmdroid [18] або MapLibre GL) зміни локалізовані в Data Layer; Domain-рівень залишається незмінним. Аналогічно – підключення альтернативної бази даних або ускладнення логіки фільтрації не зачіпає суміжні модулі. Така ізоляція уповільнює накопичення технічного боргу та підвищує показник ремонтопридатності (Maintainability).

1.2 Аналіз наявних рішень для туристичної та міської навігації

Перед проєктуванням було проведено аудит конкурентного середовища. Порівнювалися чотири продукти: Google Maps, MAPS.ME, Organic Maps, «Київ Цифровий».

Результати порівняльного аналізу функціональних можливостей існуючих картографічних застосунків та розроблюваної системи «Путівник» наведено в табл. 1.1.

Таблиця 1.1

Порівняльна таблиця застосунків

Характеристика	**Google Maps**	**MAPS.ME**	**Organic Maps**	**Київ Цифровий**	**Путівник**
Офлайн-карта міста Києва	+	+	+	-	+
Пошук культурних об’єктів	+	+	+	-	+
Фільтрація за типом об’єкта	+	+	+	-	+
Фільтрація за відстанню від поточного місцезнаходження	+	+	+	-	+
Побудова багатоточкових маршрутів	+	+	+	-	+
Рекомендації популярних місць	+	-	-	-	+
Відгуки та оцінки користувачів та об’єктів	+	-	-	-	+
Повністю офлайн режим (карта + базовий пошук POI)	-	+	+	-	+
Підтримка української мови інтерфейсу	+	+	+	+	+
Збереження обраних маршрутів та об’єктів	+	+	+	-	+

Таблиця 1.2

Загальна оцінка словами

Характеристика	Google Maps	MAPS.ME	OrganicMaps	Київ Цифровий	KyivCultureGuide
Орієнтовна оцінка зручності інтерфейсу	Висока	Середня	Середня	Середня	Висока (ціль проєкту)
Орієнтовна швидкість завантаження карти та елементів	Висока	Середня	Середня	Середня	Висока (оптимізовано для роботи офлайн)

Узагальнену якісну оцінку розглянутих застосунків наведено в табл. 1.2.

Google Maps має найширше глобальне покриття та підтримує мультимодальні маршрути. Динамічний контент (відгуки, фото, рейтинги) оновлюється регулярно. При цьому офлайн-режим обмежений кешуванням невеликого фрагмента карти; маршрутизація без мережевого з'єднання не підтримується. Модель даних закрита, ліцензійні умови не передбачають інтеграцію сторонніх фільтрів або кастомних сценаріїв пошуку. Для задачі пішохідної навігації по культурних об'єктах Києва ці обмеження є критичними.

MAPS.ME та Organic Maps [19, 20] працюють за Offline-first підходом на базі OpenStreetMap. Після завантаження векторного файлу регіону застосунок працює без стільникового з'єднання, деталізація POI від контриб'юторів OSM порівняно висока. Обмеження: обидва продукти мають закриту монолітну архітектуру. Інтеграція зовнішньої бізнес-логіки, побудова тематичних колекцій або підключення рекомендаційного рушія технічно не передбачені. Модуль аналізу поведінки відсутній – адаптація видачі під інтереси конкретного користувача відсутня [60]. Серед вузькоспеціалізованих продуктів окремо стоїть «Київ Цифровий» – муніципальний застосунок, що агрегує міські сервіси. Він надає точки інтересу і базові маршрути, але його

архітектура обмежена серверною залежністю: без стабільного інтернет-з'єднання більшість функцій недоступна. Персоналізація маршруту, побудова найкоротших обходів і офлайн-робота не входять до його функціонального ядра. Крім того, оновлення контенту здійснюється на стороні сервера з невизначеною періодичністю, що не підтверджує свіжості даних про графіки роботи культурних закладів. Відсутність власного API для сторонніх клієнтів унеможливлює підключення даних «Київ Цифровий» до власного продукту.

Третій варіант – побудова кастомного клієнта на базі відкритих картографічних бібліотек. MapsForge [15, 16] виконує рендеринг векторних карт OSM силами GPU смартфона [17] і працює повністю автономно. Бібліотека osmdroid [18] використовує тайлову модель. У фінальній версії проєкту обрано MapLibre GL Native [45] (C++ ядро, OpenGL-рендеринг) як рішення з найвищою продуктивністю на мобільних ARM-процесорах.

На основі проведеного аудиту обрано саме шлях розробки кастомного застосунку, що передбачає самостійне проєктування фільтрів, оптимізатора маршрутів та рекомендаційного модуля. Аналіз ринку показав, що масові навігатори закривають задачі загальної логістики, проте не дають персоналізації та повністю автономної роботи в контексті культурних об'єктів Києва. [62]

1.3 Особливості культурних локацій Києва та вимоги до даних

Історико-культурний ландшафт Києва вирізняється екстремально високою щільністю об'єктів. Якщо спробувати обробити та відрендерити весь цей безперервний масив просторових даних «як є» (без попередньої фільтрації), система неминуче зіткнеться з двома критичними проблемами. По-перше – це стрімке переповнення локального сховища пам'яті смартфона. По-друге – невідворотна деградація швидкодії графічного інтерфейсу (падіння частоти кадрів під час взаємодії з картою).

Щоб запобігти вичерпанню апаратних ресурсів та утримати продуктивність на стабільно високому рівні, в архітектуру було закладено

жорсткий класифікатор. Картографуванню та локальному збереженню підлягає лише суворо обмежений перелік пріоритетних категорій [2]:

- музейні установи різного профілю (історичні, художні, науково-технічні, меморіальні) [12];
- театрально-видовищні заклади, філармонії та концертні майданчики [13];
- сучасні артпростори і мистецькі кластери;
- сакральні споруди, архітектурні пам'ятки, об'єкти культурної спадщини;
- монументи, пам'ятники, меморіальні комплекси.

Кожна з п'яти категорій має відмінну модель взаємодії з відвідувачем. Музейні установи працюють за жорстким розкладом із перервами та вихідними, тоді як сакральні споруди зазвичай відкриті щоденно. Артпростори часто змінюють графік залежно від поточної виставки. Ця нерівномірність вимагає гнучкого зберігання часових регламентів: замість фіксованих полів open_time / close_time застосовано серіалізований JSON-об'єкт, що описує довільну кількість інтервалів для кожного дня тижня, включно зі святковими днями. Парсер графіків на етапі фільтрації зіставляє поточний час пристрою із цими інтервалами і автоматично приховує об'єкти, до яких турист фізично не встигне дістатися до закриття. Додатково враховується часовий буфер: якщо до зачинення залишається менше 30 хвилин, локація переміщується в кінець списку, а не зникає повністю – користувач бачить попередження «закривається скоро» і самостійно вирішує, чи варто квапитися. Якщо для конкретного закладу графік відсутній у базі, парсер повертає статус «невідомо» замість того, щоб помилково позначити місце зачиненим.

Для досягнення часу відгуку фільтрів і пошуку на рівні 2–3 мілісекунд (замість десятків) розрізнені масиви приведено до нормалізованої реляційної

структури. На рівні Room-сутностей (Entities) кожна локація описується через такий набір атрибутів:

- Геопросторова прив'язка. Для точкових POI – класична пара latitude / longitude (Double). Для масштабних архітектурних ансамблів зберігаються полігони координат у серіалізованому вигляді (TypeConverter перетворює List<LatLng> на JSON-рядок). Без цієї диференціації алгоритм маршрутизації не розрахує дистанцію до реального входу в об'єкт.
- Адресна метаінформація [14]. Крім стандартних полів (вулиця, номер, район), фіксується близькість до найближчої станції метро – це впливає на ранжування у рекомендаційному рушії.
- Типологія. Стовпець category_type (індексований через @ColumnInfo з INDEX) дає змогу фільтрувати без table scan.
- Часові регламенти. Актуальний графік (будні, вихідні, свята) зберігається як serialized JSON. Алгоритм звіряє поточний час із розкладом і не включає закриті об'єкти до маршруту.
- Комунікаційні реквізити. URL на офіційний сайт, телефон.
- Текстовий опис і семантичні теги. Теги (масив String) використовуються для Content-based рекомендацій і повнотекстового пошуку через FTS4.
- Індикатори залученості. Рейтинг (0–5 Float) та лічильник переглядів. Ці числові показники безпосередньо входять у формулу Score рекомендаційного рушія.

Вихідні дані зібрано з двох відкритих джерел. З OpenStreetMap [1, 6, 7] через теги tourism, historic, heritage [9, 10, 11] витягнуто геокоординати та межі об'єктів. Потім ці записи зіставлено з Порталом відкритих даних Києва [8], звідки підтягнуто офіційні назви та підтверджені графіки роботи. На серверному боці закладено бекенд для збору телеметрії та перерахунку динамічних рейтингів.

Міське середовище змінюється щоденно, тому статичні дампи для такого продукту неприйнятні. Тіньова синхронізація організована через:

- Автоматизовану вибірку через Overpass API [3, 4] (відповіді серіалізуються у JSON).
- Інкрементальне оновлення (дельта) – клієнт завантажує лише нові або змінені записи без повного перезавантаження масивів, що критично для енергоефективності.
- Логування часових міток. Кожній локації присвоюється timestamp через Room-поле lastUpdated: Long. Під час синхронізації система порівнює локальний timestamp із серверним і витягує лише ті записи, де серверний > локального.

На стороні клієнта масив зберігається в SQLite через Room ORM, а контроль фонових запитів покладено на WorkManager. Завдяки цьому турист отримує миттєвий доступ до локального кешу без очікування серверної відповіді, і щойно пристрій виявляє стабільне Wi-Fi-з'єднання – WorkManager запускає тіньове оновлення у фоні. Додатковий аспект – цілісність даних при паралельному доступі. Ситуація: користувач зберігає маршрут у момент, коли фоновий WorkManager імпортує свіжу порцію локацій. Без захисту обидва потоки одночасно пишуть у SQLite, і один із них отримує SQLITE_BUSY. Для серіалізації конкурентних записів у коді передбачено Mutex із бібліотеки kotlinx.coroutines.sync: операція захоплює блокування перед транзакцією та звільняє після commit. Mutex обрано замість synchronized-блоку, оскільки він сумісний із suspend-функціями і не блокує потік очікування – Coroutine просто призупиняється до звільнення ресурсу. У стресовому тесті, де одночасно запускалися 12 корутин-записувачів протягом 60 секунд, жодного SQLITE_BUSY зафіксовано не було – Mutex коректно чергував усі транзакції.

1.4 Цільова аудиторія та сценарії використання

Вибір технологічного стека, пріоритетність окремих функцій і параметри інтерфейсу (розмір шрифтів, зони натискання) визначаються цільовою аудиторією. Аналіз потенційних споживачів продукту дозволив виділити чотири сегменти з відмінними вимогами.

- Туристи та гості столиці. Два-три дні, жорсткий дефіцит часу. Мінімум елементів на екрані. Технічно – безкомпромісний Offline-first: маршрути та фільтрація без жодного кілобайта роумінгу, з мінімальним споживанням батареї.
- Постійні мешканці Києва. Основні туристичні об'єкти для цього сегмента вже відомі. Запит спрямований на менш очевидні точки: приховані архітектурні артефакти, локальні виставки, некомерційні артпростори. Утримання цієї аудиторії вимагає інтеграції рекомендаційного рушія, який аналізує історію попередніх переглядів і формує вибірку з урахуванням індивідуального профілю.
- Здобувачі освіти та екскурсійні групи. Смартфон стає інструментом конструювання освітнього маршруту – п'ять пам'яток за 80 хвилин академічної пари. Без алгоритмів оптимізації графів (TSP-евристики) задача не розв'язується [61]. [59, 61]
- Іноземні туристи. Ключова інженерна задача – динамічна локалізація (Dynamic Localization). Додавання нового мовного пакета повинно виконуватися без рефакторингу основного коду: JSON-ресурси + ML Kit офлайн-моделі підключаються «на льоту».

Спільна вимога для всіх чотирьох сегментів – мінімальний час до першої корисної дії після встановлення. Ізольований Splash Screen у проєкті відсутній: одразу після запуску MainActivity Compose NavHost промальовує стартову вкладку з рекомендаціями. Первинне заповнення бази відбувається паралельно – SeedManager парсить JSON-довідники з assets/ APK-файлу у фоновому потоці, і стрічка починає наповнюватися протягом перших 2–3

секунд. Поки база заповнюється, інтерфейс показує анімовані skeleton-заглушки, що імітують структуру майбутніх карток. Суб'єктивний час очікування зменшується порівняно з порожнім екраном або спінером. Аналогічний підхід застосовано до першого завантаження карти: Mapbox SDK дозволяє запаковувати стартовий тайл-кеш прямо в assets/ APK, тому рендер починається без мережевого запиту. За даними внутрішнього бенчмарку, перша інтерактивна карта з'являється за 1.1 секунди на пристрої середнього класу з 4 ГБ оперативної пам'яті. Без попередньо запакованих тайлів цей час зростає до 3.5–4.0 секунд через очікування завантаження текстур із CDN.

На основі описаних профілів сформовано чотири Use Cases, що безпосередньо визначили технічне завдання:

1. Просторовий пошук «на ходу». У туриста з'явилося двогодинне вікно у центрі – система за мілісекунди генерує вибірку найближчих об'єктів через SQL-запит із фільтром по Haversine-дистанції.

2. Оптимізація логістики. Математично розрахувати найкоротший маршрут через 5–10 розрізнених точок із використанням Nearest Neighbor + 2-opt.

3. Реактивна навігація до подій. Пошук тимчасових івентів (вистави, фестивалі) із прокладанням шляху від поточних GPS-координат.

4. Алгоритмічна персоналізація. Аналіз локальної телеметрії, побудова вектора профілю і пошук семантично схожих ще не відвіданих локацій.

Ці патерни поведінки – це не просто теорія. Вони виступають жорстким технічним завданням (ТЗ) для наступних етапів проєктування. Саме Use Cases визначають, якою буде ієрархія екранів, де стоятимуть фільтри та як працюватиме неблокуючий рендеринг карти. Фактично, вони закладають фундамент для розрахунку математичних моделей релевантності та диктують правила управління життєвим циклом локальних даних.

1.5 Алгоритмічні підходи до обробки даних

Персоналізований навігатор передбачає обробку геопросторових масивів та семантичних зв'язків між об'єктами. Алгоритмічний модуль системи закриває чотири задачі: реактивну фільтрацію за множиною критеріїв, генерацію рекомендацій за профілем користувача, розв'язання TSP для багатоточкових маршрутів та управління автономним кешем. [58]

1.5.1 Багатокритеріальна фільтрація та просторове сортування локацій

Шум відсікається реактивно. Щойно турист перемикає параметр (тип закладу, близькість до метро, «відкрито зараз»), StateFlow миттєво віддає оновлений масив у Compose без перезавантаження екрана. Ранжування вибірки здійснюється за кількома критеріями:

1. За геодезичною відстанню: Haversine-формула обчислює ортодромію від координат смартфона до точки входу об'єкта.
2. За рейтингом: ранжування від 5.0 до 0.0.
3. За популярністю: впорядкування за лічильником viewsCount.

Замість жорсткого хардкодування критеріїв впроваджено зважену функцію «корисності» (Utility Function). Кожна локація отримує комплексний Score: коефіцієнти значущості відстані, рейтингу, популярності задаються через конфігураційний об'єкт, який можна підмінити без ребілду. Якщо потрібно надати беззаперечний пріоритет найближчим об'єктам (наприклад, для пішохода з обмеженим часом), достатньо змінити один ваговий множник. На технічному рівні фільтрація працює через ланцюжок операторів Kotlin Flow. Перемикання категорії або введення тексту у пошуковий рядок генерує новий FilterState. Оператор combine() зливає текстовий запит і обрані категорії в один потік. Далі debounce (300 мс) запобігає надлишковим проміжним запитам під час швидкого набору на клавіатурі, а distinctUntilChanged() відсікає повторне спрацювання при ідентичних

параметрах. Тільки після проходження цих бар'єрів оновлений FilterState потрапляє у DAO-запит. Така послідовність зменшує кількість операцій із базою до мінімуму і тримає споживання процесора під контролем. Під час профілювання на Pixel 6a з базою у 300 локацій середній час відгуку від натиснення до оновлення списку становив 180 мс. Це значення непомітне для людського ока і нижче за рекомендований поріг RAIL-моделі у 300 мс.

1.5.2 Архітектурні підходи до побудови рекомендаційної системи

При нефільтрованій видачі каталогу з сотнями об'єктів когнітивне навантаження на користувача зростає. Рекомендаційний рушій формує ранжовану вибірку, адаптовану під індивідуальний профіль. В літературі описано два основних підходи:

1. Контентна фільтрація (Content-based) [43]. Кожна локація розглядається як вектор властивостей (тематика, район, тривалість візиту). Із історії переглядів формується «профіль інтересів». Алгоритм вираховує метрику подібності (cosine similarity або зважену лінійну дистанцію) між профілем і базою, піднімаючи наверх споріднені об'єкти.

2. Колаборативна фільтрація (Collaborative). Глобальна матриця поведінки дає змогу екстраполювати досвід подібних користувачів. Точність вища, проте матричні обчислення такого масштабу потребують серверного кластера і постійного з'єднання з мережею, що несумісне з Offline-first архітектурою проєкту.

Модель оцінювання (Scoring Model) побудована як лінійна комбінація чотирьох факторів з фіксованими вагами. Рейтинг об'єкта домножується на коефіцієнт 12 – якість домінує над масовістю. Показник популярності має множник 0.3: цілеспрямовано занижена вага запобігає витісненню маловідомих, проте якісних локацій із верхніх позицій. Профільна складова нараховує бонуси за новизну (+150 балів, якщо об'єкт відсутній в історії) і збіг тематичного тегу (+40), а раніше відвіданим об'єктам призначає штраф −80. Просторова вага обернено пропорційна дистанції: baseSpatialBonus /

(distanceKm + 1.0), де додана одиниця захищає від ділення на нуль, коли турист знаходиться впритул до об'єкта. Усі чотири складові нормалізуються до діапазону 0–1 перед підсумовуванням: рейтинговий компонент ділиться на максимальний рейтинг вибірки, просторовий – на найбільший бонус серед кандидатів. Без нормалізації одна складова з великими абсолютними значеннями домінувала б над рештою незалежно від вагових коефіцієнтів. Після підсумовування список сортується за фінальним балом, обрізається до topN і передається у UI-шар як StateFlow.

1.5.3 Оптимізація топології багатоточкових маршрутів

Побудова пішохідного маршруту через кілька локацій є обчислювально складною задачею для мобільного ARM-процесора. Дорожня мережа Києва моделюється як зважений граф, де вершини відповідають перехрестям, а ребра – сегментам вулиць. Для двох точок задача розв'язується алгоритмами Dijkstra або A* за мілісекунди. При збільшенні кількості проміжних точок до трьох і більше задача редукується до TSP (Traveling Salesperson Problem) [42].

TSP належить до класу NP-складних задач. [64] Кількість перестановок зростає як n!; для 15 точок повний перебір (Brute Force) вимагає аналізу комбінацій, що перевищує обчислювальні можливості мобільного процесора. Обрано евристичний підхід:

1. Nearest Neighbor (жадібний алгоритм): стартуємо з першої точки, на кожному кроці переходимо до найближчої невідвіданої, отримуючи чорновий маршрут із можливими самоперетинами.
2. 2-opt (локальна оптимізація): бере цей чорновий маршрут і попарно переставляє ребра для вирівнювання петель (ліміт 100 ітерацій на запит). [42, 56]
3. Жорстке обмеження глибини: максимум 8–10 проміжних точок за один маршрут.

Типовий сценарій пішохідного туризму передбачає 5–10 об'єктів за день. При такій кількості вузлів комбінація Nearest Neighbor та 2-opt генерує

маршрут за частки секунди. Після визначення послідовності точок обчислюється конкретний пішохідний шлях по вуличній мережі. Для цього клієнт надсилає координати в OSRM (Open Source Routing Machine) – рушій, що працює з графом OpenStreetMap. Відповідь містить деталізовану полілінію, яка може складатися з кількох тисяч координатних пар для маршруту 5–7 км. Промальовка такого обсягу GeoJSON-геометрії без спрощення спричиняє надмірну алокацію LatLng-об'єктів у heap, тому перед промальовкою на карті запускається алгоритм спрощення Douglas-Peucker, що прибирає візуально непомітні проміжні точки і тримає кількість вузлів у допустимих межах. Паралельно працює дедуплікатор: якщо два waypointи стоять ближче ніж 15 метрів, вони зливаються в одну логічну точку ще до відправки запиту. Мережевий клієнт оснащений retry-механізмом: при таймауті OSRM запит повторюється з експоненціальним backoff (затримки 1 с → 2 с → 4 с, максимум 3 спроби). Якщо всі три спроби провалилися, застосунок показує попередження і пропонує маршрут у вигляді прямих відрізків між точками – менш точний, але достатній для орієнтації.

1.5.4 Стратегії кешування та офлайн-синхронізації

Offline-first є базовою архітектурною вимогою. Інженерна реалізація цієї підсистеми спирається на три ізольовані компоненти:

1. Векторні тайли у форматі MapLibre GL [5] завантажуються один раз, після чого рендеринг бере на себе GPU – мережевих запитів до сервера тайлів більше немає.

2. Локальне сховище. Каталог із метаданими індексується у SQLite через Room, повнотекстовий пошук побудований на FTS4 (віртуальні таблиці) з латентністю Live Search 2–3 мс.

3. Тіньова синхронізація. Інкрементальне оновлення запускає WorkManager за стабільного Wi-Fi.

Кеш перекладів утворює окремий шар. Результати ML Kit On-device Translation зберігаються у Room-таблиці TranslationCache разом з SHA-256

хешем оригінального тексту. Коли мобільний бекенд оновлює опис локації, хеш нового тексту не збігається з раніше збереженим, і кеш визнається невалідним – рушій запускає повторний переклад. Без цього механізму після оновлення серверних даних користувач бачив би застарілий переклад необмежено довго. Ємність кешу обмежена кількістю активних мовних пар: 15 підтримуваних мов покривають понад 90 відсотків іноземних туристів Києва, а кожна мовна модель займає близько 30 МБ локального простору. Окрему проблему становить контроль обсягу локальної бази. Зі зростанням кількості закладів збільшується розмір SQLite-файлу, а разом із ним – час холодного відкриття з'єднання. Для стримування росту запроваджено ліміт збережених маршрутів: коли їхня кількість перевищує 50, найстаріші видаляються каскадним DELETE з прив'язаних таблиць проміжних точок і перекладів. Аналогічний механізм TTL (Time-To-Live) застосовано до кешу перекладів: записи, яким більше 30 діб, позначаються застарілими й замінюються при наступному зверненні. Обидва правила виконуються в межах однієї транзакції, тому часткове видалення неможливе. Крім того, при кожному оновленні програми запускається одноразова міграція, яка перевіряє наявність нових індексів і за потреби виконує ALTER TABLE або CREATE INDEX у межах наявної версії схеми.

1.6 Аналіз нефункціональних вимог до застосунку (безпека, локалізація та стабільність)

Проєктування починається з нефункціональних вимог, оскільки застосунок безперервно обробляє геодані та історію маршрутів і питання захисту стоїть на першому місці:

Безпека:

1. Захист локальної бази. Усі запити до Room виконуються через параметризовані @Query – SQL-ін'єкції неможливі на рівні ORM [25, 26].

2. Мережева ізоляція. HTTP відсутній повністю – лише HTTPS. Додано перевірку хоста (Certificate Pinning на рівні OkHttp Interceptor [34]) і runtime guard.

3. Санітизація імпорту. JSON-файли з маршрутами при імпорті перевіряються на розмір (ліміт 2 МБ) – це захищає від Out of Memory при зловмисному або пошкодженому файлі.

4. Manifest Hardening. allowBackup=false, exported=false для внутрішніх Activity, валідація Intent extras перед обробкою.

На практиці механізм Certificate Pinning працює таким чином: при першому TLS-рукостисканні OkHttp порівнює SHA-256 хеш публічного ключа сервера зі значенням, зашитим у конфігурації CertificatePinner. У разі невідповідності з'єднання негайно розривається ще до передачі першого байта корисного навантаження, навіть якщо центр сертифікації вважає сертифікат чинним. Це захищає від сценарію компрометації кореневого CA, коли зловмисник генерує підроблений, але формально валідний сертифікат для домену API. Для виробничого середовища передбачено два піни: основний та резервний, щоб ротація сертифіката на сервері не призводила до блокування мобільного клієнта. У debug-збірці пінінг деактивований програмно через прапорець BuildConfig.DEBUG, що спрощує роботу з локальним тестовим сервером без необхідності підмінювати сертифікати.

Локалізація:

1. Інтерфейсні рядки підтягуються з багатомовних JSON-ресурсів, які перемикаються без рестарту Activity (через recreate() з перезбиранням Compose state tree).

2. Під час першого запуску мова автоматично підхоплюється з Locale.getDefault() операційної системи.

3. Для офлайн-перекладу динамічного контенту (описи локацій) інтегровано ML Kit On-device Translation. Завантаження моделі (~30 МБ) супроводжується прогрес-баром, щоб людина бачила стан процесу.

Стабільність збирання (MapLibre GL Native):

1. Подвійна валідація URI-стилю: компілятором (kapt) і під час рендерингу (runtime check перед викликом setStyleUri()).

2. Автоматичне очищення .cxx кешів C++ при Gradle clean task – рятує Ninja від збоїв при перекомпіляції NDK-модулів.

3. SAST-інтеграція на рівні Gradle для аналізу вразливостей JVM-модулів.

Окремий момент – стратегія міграцій бази даних. Room вимагає явного опису кожної структурної зміни. На момент публікації проєкт пройшов 7 послідовних міграцій: від початкової схеми до стану з FTS4-індексацією, триколонковим повнотекстовим пошуком та індексом по стовпцю category. Кожна міграція тестується окремо, перехід між версіями бази відбувається без втрати даних, а відсутність destructive-міграцій зберігає маршрути й закладки після оновлення APK. Після міграції команда rebuild перебудовує інвертований FTS4-індекс за рядками таблиці places.

Висновки до розділу 1

Перший розділ присвячено аналізу предметної області, аналогів та вибору проєктних рішень, за результатами якого сформульовано висновки:

1. Аналіз патернів MVVM (Model–View–ViewModel) та Clean Architecture (підрозділ 1.1) показав, що їх поєднання з Android Jetpack забезпечує модульність і тестовність коду за обмежених ресурсів пристрою. Hilt спрощує впровадження залежностей, а Kotlin Coroutines – керування асинхронними операціями.

2. Огляд додатків Google Maps, MAPS.ME, Organic Maps і Kyiv Digital (підрозділ 1.2, табл. 1.1) засвідчив: жоден не пропонує офлайн-побудову персоналізованих маршрутів по культурних об’єктах Києва. Ця прогалина підтвердила доцільність окремого застосунку.

3. Геопросторові дані з відкритих джерел нормалізовано до реляційної схеми (підрозділ 1.3): для категоріальних запитів використано B-tree-індекс

по полю category_type, для текстового пошуку – індекс FTS4. Це скорочує час вибірки порівняно з послідовним перебором.

4. Профілювання аудиторії (підрозділ 1.4) виділило чотири сегменти: туристи-одноденники, організовані групи, місцеві мешканці та дослідники спадщини, для кожного з яких сформульовано сценарії використання (Use Cases) – основу технічного завдання на алгоритмічний модуль.

5. Обґрунтовано вибір алгоритмів (підрозділи 1.5.1–1.5.4): контентна фільтрація працює локально, а порядок обходу точок будують евристики найближчого сусіда і 2-opt, розв’язуючи TSP за поліноміальний час; розглянуто й офлайн-кешування та синхронізацію.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТА АЛГОРИТМІЧНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ

2.1 Функціональна логіка та інтерфейс користувача

Інтерфейс проєктувався з метою мінімізації кількості кліків до цільової дії. Ізольований Splash Screen відсутній: одразу після ініціалізації MainActivity Compose NavHost рендерить стартову вкладку зі стрічкою рекомендацій (локації, відсутні в історії переглядів користувача).

Головна навігація – Bottom Navigation Bar (Material Design 3 [32]), п'ять вкладок: «Головна», «Карта», «Маршрути», «Події», «Налаштування».

Картографічна вкладка «Карта» влаштована так: рядок пошуку разом із чіпами категорій (Filter Chips) накладаються оверлеєм поверх MapLibre Surface, тож користувач не втрачає візуальний контакт із картою під час налаштування фільтрів.

Взаємодія з маркерами влаштована у два етапи. Натискання на маркер у режимі перегляду відкриває компактний Bottom Dock (назва, категорія, стрічка суміжних POI). Розгорнута інформація завантажується на окремому екрані LocationDetailsScreen (перехід через Navigation Component) після натискання кнопки «Детальніше».

Конструктор маршрутів (Route Builder) запускається через FAB на вкладці «Маршрути». Карта переходить у режим вибору: клік по маркеру не відкриває інфокартку, а додає локацію як waypoint до поточного маршруту. Після збору точок – введення назви і збереження у Room.

Живий пошук (Live Search) побудований на FTS4-запитах до Room. По мірі набору тексту StateFlow пушить оновлений масив у Compose – без мережевих запитів. Чіпи-перемикачі (Filter Chips) дають змогу комбінувати критерії. Механізм фільтрації побудовано на ланцюжку операторів Kotlin Flow. При активації чіпа текстовий запит і обрані категорії об'єднуються в один FilterState за допомогою combine(). Далі потік проходить через debounce (300 мс), щоб зменшити кількість проміжних перемальовувань під час

швидкого набору. Наступний оператор distinctUntilChanged() відсікає ситуації, коли параметри фактично не змінилися. Тільки після проходження цих бар'єрів запит потрапляє до FTS4 через PlaceDao. Такий дизайн мінімізує кількість SQL-запитів і тримає споживання батареї під контролем навіть при інтенсивній взаємодії з пошуком. Додатково передбачено перехід від пустого стану: якщо запит не повертає результатів, Compose-шар відмальовує EmptyStateView з іконкою та підказкою уточнити пошук. При виникненні помилки на рівні Room або FTS (наприклад, некоректний MATCH-вираз) активується програмний фолбек на in-memory фільтрацію через contains(), щоб пошук не переставав працювати за жодних обставин.

Вкладка «Події» – вертикальний LazyColumn [44] із подіями (виставки, фестивалі, лекції). Дані тягнуться з мережі; при відсутності конекту відображається закешований JSON. Прогрес-бар та fallback-заглушка побудовані на sealed class UiState (Loading | Success | Error). Окрім трьох базових станів, UiState передбачає окремий підстан Loading.Skeleton, який відображає анімовані заглушки замість спінера. Скелетон-картки повторюють компонувальну структуру фінальних елементів (заголовок, два рядки опису, мітка дати), проте заповнені градієнтним shimmer-ефектом. Анімація побудована на Compose infiniteTransition: лінійний градієнт із трьох точок (surfaceVariant → surface → surfaceVariant) зсувається горизонтально зі швидкістю 1200 мс на цикл. Перевага над CircularProgressIndicator – людина бачить структуру майбутнього контенту до його завантаження, що знижує суб'єктивне відчуття очікування і відповідає евристиці «Видимість стану системи» за Джейкобом Нільсеном. Скелетон-картки оформлено як окремий @Composable-компонент SkeletonListLoader, який приймає параметр itemCount (кількість placeholder-карток) і використовується на екранах Подій, Місць та Головному екрані під час першого завантаження даних.

«Налаштування» – перемикання мови (ігноруючи системну), імпорт/експорт маршрутів у JSON із валідацією схеми та підтверджуючим діалогом, щоб не знищити збереження пошкодженим файлом.

Вимоги доступності: усі інтерактивні елементи відповідають порогу minimumInteractiveComponentSize (48×48 dp). Підключено Dynamic Color та автоматичне перемикання Dark Theme залежно від системних налаштувань.

2.2 Архітектурне забезпечення та багаторівнева модель застосунку

2.2.1 Парадигма проєктування та інфраструктура залежностей

В основі лежить концепція Single Activity – усі екрани функціонують в межах одного MainActivity, а переходи між ними побудовані на підміні Composable-функцій у NavHost. Додаткові Activity не створюються, чим зменшуються накладні витрати на ініціалізацію.

Візуальна частина виконана на Jetpack Compose [44]. XML-верстка з імперативним findViewById не використовується. Compose перемальовує тільки вузли UI-дерева, де фактично змінився State – це принципово, бо під час навігації GPS-координати, залишок дистанції та ETA оновлюються щосекунди, і декларативна модель обробляє цей потік не виходячи за бюджет кадру 16.6 мс.

Залежності керуються Hilt [39] – класи не створюють свої залежності самі. Hilt постачає об'єкти відповідно до ієрархії життєвих циклів (SingletonComponent – глобальні сервіси, ViewModelComponent – дані конкретного екрана), що знижує зв'язність між модулями. Загальну структурну схему застосунку, що ілюструє описану декомпозицію шарів, подано на рис. 2.1. Unit-тестування спрощується через підміну на Fakes. Конфігурація Hilt передбачає три модулі ін'єкцій, кожен з яких ізольований за сферою відповідальності. AppModule постачає мережевий стек (OkHttpClient із перехоплювачами безпеки, WorkManager, об'єкт телеметрії). DatabaseModule – екземпляр Room-бази та шість DAO-інтерфейсів. NetworkModule – Retrofit-клієнт для API подій та окремий OkHttpClient з власним таймаутом для OSRM-маршрутизатора. Такий підхід дає змогу у тестовому контексті перевизначити лише один модуль (наприклад, підставити

in-memory базу замість файлової), не торкаючись інших частин графу залежностей. Компілятор KSP виявляє циклічні залежності та відсутні зв'язки ще на етапі збирання, що підтверджує цілісність DI-графу без жодного запуску емулятора. Окремий кваліфікатор @OsrmClient дає змогу мати два незалежних екземпляри OkHttpClient: один із жорстким Certificate Pinning для Events API і другий – з коротшими таймаутами для OSRM-маршрутизатора. Hilt розрізняє їх по анотації і підставляє потрібний клієнт у кожну точку ін'єкції автоматично, без ручного конструювання.

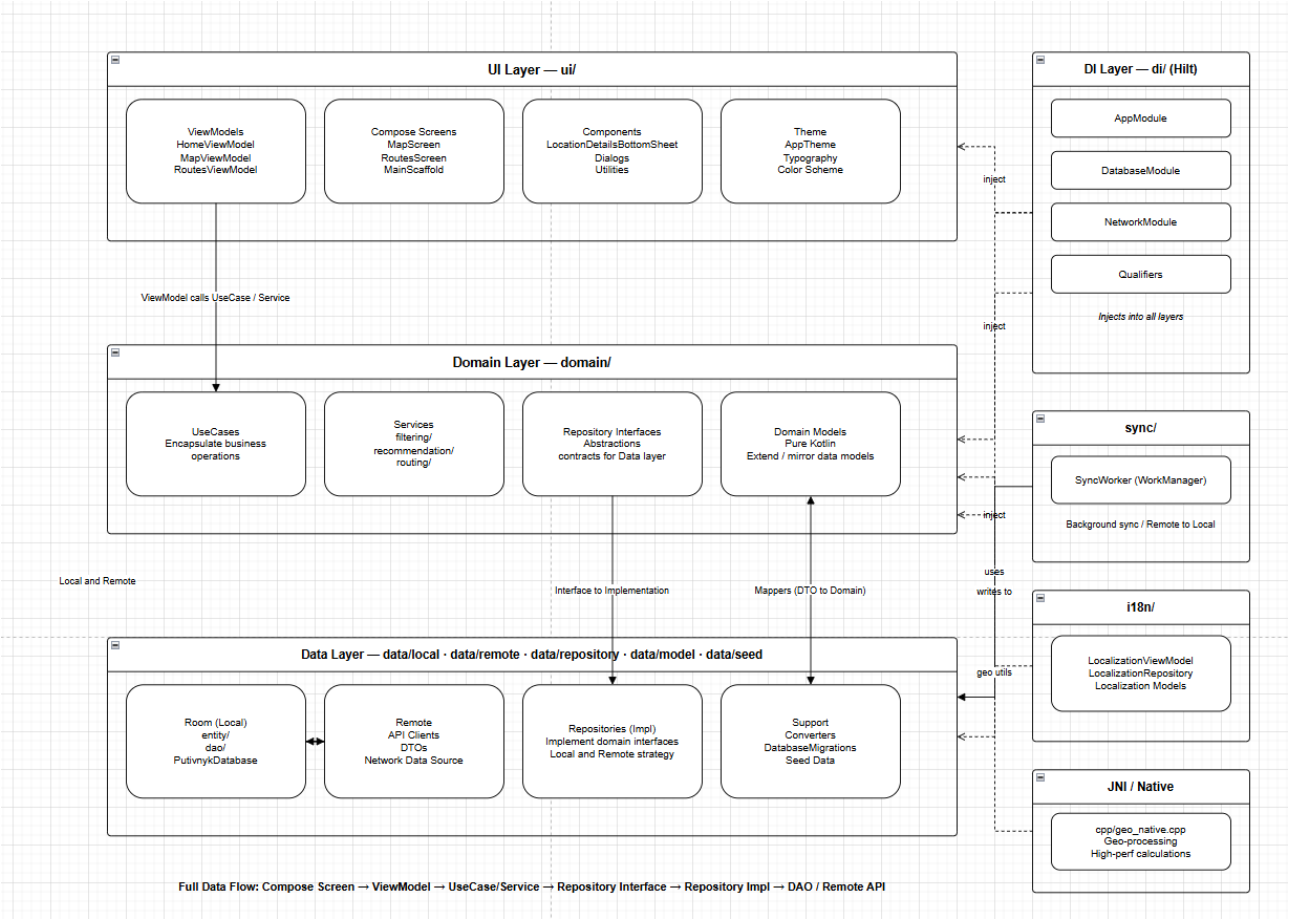

Рисунок 2.1. Структурна схема застосунку

2.2.2 Організація рівня даних (Data Layer) та локального сховища

Автономність від мережі є обов'язковою вимогою до архітектури. Локальне сховище побудовано на Room [25] – ORM, яка валідує SQL ще на

етапі kapt-компіляції і цим виключає runtime-помилки через синтаксичні невідповідності чи некоректні типи даних.

PutivnykDatabase зведено до нормальної форми і розбито на 6 таблиць:

1. Places. Найважчий масив: 22+ атрибути на кожен об'єкт – id, latitude, longitude, семантичні теги (List<String> через TypeConverter → JSON), двомовні назви (ukrainianName, englishName), накопичений рейтинг (Float), пара URL фотографій. 15 унікальних категорій (category_type з INDEX).

2. Routes. Масив waypoints має невідому довжину – типова реляційна проблема, яка вирішена серіалізацією List<Double> у плоский рядок через TypeConverter. Дані осідають у звичайному TEXT-стовпці.

3. Допоміжні таблиці: MapBookmarks (кастомні закладки), UserPreferences (глобальні налаштування), SyncState (стан тіньових синхронізацій). Плюс таблиця TranslationCache для кешів перекладу.

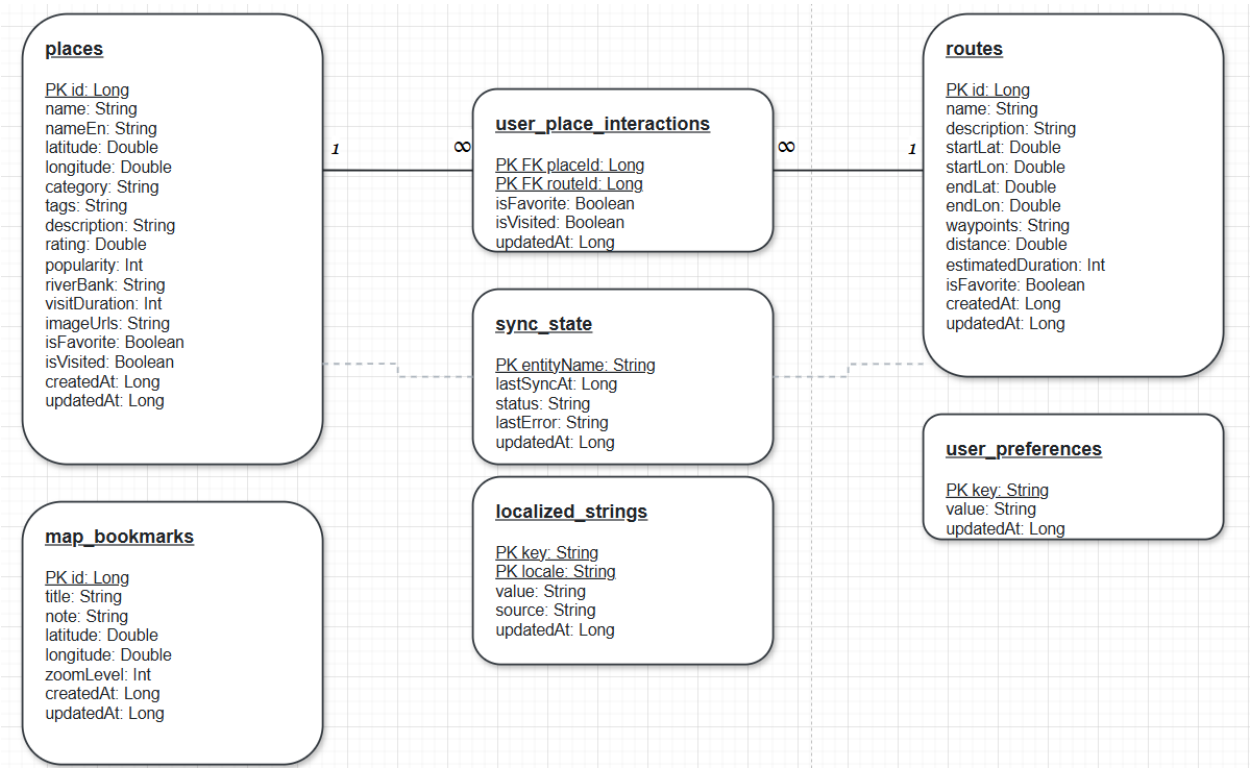

Рисунок 2.2. Логічна модель бази PutivnykDatabase

Логічну модель бази даних PutivnykDatabase з усіма зв’язками між таблицями зображено на рис. 2.2.

Первинне заповнення – SeedManager. Під час першого «холодного» старту модуль парсить JSON-довідники, запаковані в assets/ APK-файлу. Цікава деталь: алгоритм автоматично визначає берег Дніпра для кожної локації. Якщо longitude > 30.58°, об'єкт маркується лівобережним. Ручне тегування не потрібне.

Доступ до таблиць суворо через DAO. Шар Domain не знає про SQL – він працює з абстрактними інтерфейсами PlacesDao, RoutesDao. Ізоляція спрощує підміну реальної бази на mock-об'єкти в Unit-тестах.

Читання відбувається реактивно, через Kotlin Flow. Наприклад, користувач обрав фільтр «Музеї» – DAO відкриває Flow-канал, і якщо WorkManager у фоні оновлює рейтинг, Room автоматично проштовхує зміну. Compose перемалюється автоматично. Кнопка «Оновити» не потрібна.

Запис (операції Insert, Update, Delete) йде через suspend functions. Складні операції (збереження маршруту з десятком waypoints) обгорнуті в @Transaction. Атомарність гарантована: збій посеред запису викликає rollback до попереднього стану, і пошкоджених маршрутів у базі не залишається.

Повнотекстовий пошук побудований на FTS4. Пошук через LIKE по описам дав би full table scan зі складністю , тоді як FTS4 будує інвертований індекс і знижує латентність Live Search до 2–3 мс. Інтерфейс залишається плавним навіть при швидкому наборі на клавіатурі. Технічна реалізація FTS4 у Room потребує окремої entity-анотації @Fts4 із прив'язкою до основної таблиці через параметр contentEntity. Віртуальна таблиця places_fts індексує три текстових стовпці: name, nameEn та description. Синхронізація контенту між основною таблицею та FTS-індексом відбувається через чотири автоматичні тригери (BEFORE UPDATE, BEFORE DELETE, AFTER UPDATE, AFTER INSERT), які Room генерує самостійно на основі анотації. При міграції з версії 6 на версію 7 алгоритм запускає команду rebuild, яка

повністю перебудовує інвертований індекс за наявними записами. Пошуковий запит у PlaceDao побудований як SQL JOIN: основна таблиця з'єднується з FTS-таблицею через rowid, а MATCH-оператор шукає збіги у дереві токенів за) замість лінійного сканування. Вхідний рядок перед передачею у MATCH проходить санітизацію: спеціальні символи FTS4 (лапки, зірочки, дужки, дефіси) замінюються на пробіли, а до кожного токена додається оператор prefix search (зірочка), що дає змогу знаходити результати під час набору слова, не чекаючи завершення введення.

2.2.3 Організація фонової синхронізації (Sync Layer)

Дані про культурні об'єкти Києва потребують регулярного оновлення (зміна графіків, нові виставки, актуалізація афіші). Ручне оновлення через Pull-to-refresh не використовується. Координатор SyncScheduler керує трьома паралельними Worker-ами через WorkManager [27]:

1. Оновлення подій (EventSyncWorker). Кожні 6 годин цей процес тихо «прокидається» і надсилає запит до віддаленого REST API (використовуючи клієнт Retrofit [33]), щоб витягнути свіжу афішу міських івентів. Але тут закладено жорсткий запобіжник: мережевий запит піде виключно тоді, коли пристрій зловить стабільний Wi-Fi. Якщо користувач гуляє на дорогому мобільному трафіку в роумінгу, або мережа відсутня взагалі, воркер просто підсуне інтерфейсу старий закешований JSON-довідник. Як результат – користувач ніколи не зіткнеться з порожнім екраном чи вікном помилки.

2. OfflineCacheSyncWorker. Раз на 4 години серіалізує закладки та маршрути в JSON і зберігає у internal storage. Оскільки турист може зберігати маршрут саме в момент серіалізації (Race Condition), доступ до файлу захищено Mutex з kotlinx.coroutines.sync для суворо послідовного доступу

3. TranslationModelSyncWorker. Перевіряє чексуми ML Kit мовних пакетів і довантажує оновлені моделі – переклад на 15 мов доступний навіть без мережі.

Координатор SyncScheduler при запуску MainActivity планує усі три Worker-и через enqueueUniquePeriodicWork з політикою KEEP – повторна реєстрація не створює дублікатів. Кожний Worker сконфігурований із Constraints: EventSyncWorker потребує NetworkType.UNMETERED (тільки Wi-Fi), OfflineCacheSyncWorker не має мережевих обмежень (працює з локальними файлами), TranslationModelSyncWorker вимагає будь-якого з'єднання. Системний JobScheduler розносить задачі так, щоб процесорне навантаження не припадало на активну взаємодію з картою. Якщо Worker завершується невдачею, WorkManager автоматично застосовує BackoffPolicy.EXPONENTIAL із початковою затримкою 30 секунд. Помітна перевага цього підходу – система самостійно вибирає вікно запуску з урахуванням режиму Doze та App Standby, тому батарея не витрачається на пробудження процесора в моменти, коли пристрій неактивний. Для діагностики у разі збою кожен Worker логує результат через AppTelemetry з відміткою статусу SUCCESS або FAILURE та тривалістю роботи у мілісекундах.

2.2.4 Рівень представлення (UI Layer) та управління станами

Передача даних до інтерфейсу організована за патерном MVVM. Кожен екран отримує окремий ViewModel, який транслює поточний State через StateFlow.

Найбільший за обсягом контролер – MapViewModel (понад 1100 рядків коду). Він керує екземпляром MapLibre GL Native [45] та обробляє геопросторові події у реальному часі і охоплює кілька задач:

1. Трекінг прогресу. Обчислення дистанції до наступного waypoint і динамічний перерахунок ETA. Тут використовується проєкція поточних координат на полілінію маршруту – ближня точка шукається за з early exit при знаходженні сегмента ближчого ніж 5 м.

2. GPS Jitter Protection [40]. У підземних переходах і тунелях GPS-сигнал стрибає. Коли кількість зафіксованих супутників < 4,

activateUndergroundMode() блокує перепрокладання маршруту і не дає маркеру «телепортуватися».

3. Геопросторові тригери (Geofencing). Дистанція до кожного POI перераховується через Haversine. Якщо < 120 м – автоматична інформаційна підказка. Обчислення – на Dispatchers.Default, щоб не блокувати рендер-потік.

Додатково MapViewModel обробляє ситуацію критичного відхилення від маршруту. Якщо проєкція GPS-точки на полілінію показує відстань більше 40 метрів протягом 3 послідовних вимірювань, запускається процедура reroute: поточна позиція стає новим стартом, а наступний невідвіданий waypoint – тимчасовим фінішем. Запит до OSRM відправляється з мінімальним пріоритетом, щоб не блокувати активний промальовування маршруту. Під час очікування відповіді OSRM користувач бачить мінімальний індикатор перепрокладання у нижній панелі карти, а попередній маршрут залишається видимим до отримання нового. Ще один елемент – визначення прибуття до waypoint: коли Haversine-відстань до наступної точки < 80 метрів, вона автоматично позначається пройденою і фокус переключається на подальшу ділянку маршруту.

Розподіл відповідальності між рівнями: Data Layer ізольований від UI-логіки; Sync Layer виконує фонові операції без блокування Main Thread; Compose-рівень реактивно оновлює інтерфейс при зміні State. Взаємна ізоляція модулів виключає блокування інтерфейсу під час виконання важких запитів.

Підсумки архітектурного проєктування системи

Багаторівнева архітектура працює з чітким розподілом відповідальності. Data Layer ізольований і функціонує автономно. Фонові Worker-и (Sync Layer) стягують оновлення з мережі без блокування Main Thread. Реактивний UI Layer миттєво відмальовує будь-які зміни просторової ситуації. Результат – інтерфейс не зависає, монолітної зв'язності коду немає, а система масштабується для подальшого нарощування функціоналу.

2.3 Програмна реалізація геопросторових алгоритмів та маршрутизації

2.3.1 Математична модель обчислення геодезичних відстаней

У контексті пішохідної навігації по історичному центру допустима похибка – 5–10 метрів, не більше. Перевищення цього порогу призводить до генерації хибних алертів про відхилення від маршруту або пропуску цільового об'єкта. Застосування Евклідової метрики на декартовій площині для координат Києва (~50.45° N) дає систематичну похибку: один градус довготи на цій широті коротший за градус на екваторі приблизно на 36 %. Формула distance =) не враховує сферичність Землі.

В основу закладено Haversine formula. Тригонометричні обчислення (sin, cos, asin, sqrt) для тисяч пар точок кілька разів на секунду створюють помітний overhead на JVM мобільного ARM-процесора, тому алгоритм написано на C++ у нативній бібліотеці putivnyk_native. Kotlin-код викликає його через JNI-міст, обгорнутий у клас NativeGeoEngine.

Конвеєр обчислень на C++ рівні:

1. Координати (lat, lon) обох точок конвертуються з градусів у радіани (вимога <cmath>).
2. Розраховуються дельти: .
3. Обчислюється: .
4. Результат: , де R = 6371 км.
5. На виході – дистанція в метрах із субметровою точністю (похибка Haversine для Києва).

2.3.2 Алгоритми оптимізації та автоматичного спрощення геометрії шляху

OSRM (Open Source Routing Machine) повертає деталізовану полілінію, що відтворює кожен вигин вуличної мережі. Маршрут довжиною 5–7 км може складатися з кількох тисяч координатних пар. Відмальовка такого

обсягу GeoJSON-геометрії на MapLibre Surface без попереднього спрощення спричиняє надмірну алокацію LatLng-об'єктів у heap, що може призвести до завершення процесу системним Android OOM Killer.

Санітарний алгоритм вбудований в RouteMetricsCalculator і спрацьовує, коли кількість вузлів у полілінії > 120. В основі – модифікований Douglas-Peucker [47]:

1. Між стартом і фінішем відрізка натягується хорда (пряма).
2. Шукається проміжна точка з максимальним перпендикулярним відхиленням від хорди.
3. Якщо відхилення (емпірично: 5–7 м) – усі проміжні точки на цьому відрізку видаляються. Візуально лінія на карті не змінюється.
4. Якщо відхилення (різкий поворот), маршрут розбивається у цій точці і алгоритм запускається рекурсивно для обох частин.

Паралельно працює дедуплікатор: якщо два waypoints стоять ближче ніж 15 м (пам'ятник і фонтан на одній площі), вони зливаються в одну логічну точку ще до відправки запиту на OSRM. Разом ці фільтри тримають рендеринг на 60 FPS навіть на бюджетних пристроях. Алгоритм Douglas-Peucker написано двічі: на C++ рівні (бібліотека putivnyk_native) для максимальної продуктивності та на Kotlin як програмний фолбек. Kotlin-версія активується автоматично, якщо завантаження нативної бібліотеки завершилося помилкою (пристрій із непідтримуваною архітектурою CPU). Для обмеження глибини рекурсії встановлено верхню межу в 200 waypoints: якщо після спрощення полілінія все ще перевищує це значення, алгоритм збільшує epsilon на 20 відсотків та запускає повторне проріджування. Вибір між C++ та Kotlin відбувається прозоро через клас NativeGeoEngine: під час ініціалізації перевіряється наявність бібліотеки, і всі подальші виклики делегуються на відповідний бекенд без участі коду верхнього рівня.

2.3.3 Евристичні методи розв'язання задачі комівояжера для складених маршрутів

Ключовий обчислювальний компонент RouteOptimizer відповідає саме за цей сценарій. Турист додав 10–15 музеїв і хоче обійти їх найкоротшим маршрутом. Математично – TSP [42]. Brute Force для 15 точок: перестановок – застосунок зависне.

Обчислення організовано як послідовний конвеєр:

1. Фаза матриці відстаней. Haversine NativeGeoEngine обчислює дистанцію від кожної точки до кожної іншої. Матриця кешується в масиві DoubleArray, що уникає повторних обчислень.

2. Nearest Neighbor стартує з першої локації, стрибаємо до найближчої невідвіданої. Складність . Дає швидкий чорновий маршрут, але з самоперетинами.

3. 2-opt [42]. Ітеративно перебирає пари ребер: якщо перевернути підпослідовність між ними зменшує загальну довжину – зберігає зміну. Ліміт: 100 ітерацій.

Оптимальна послідовність передається у WalkingDirectionsService → OSRM. Мережевий клієнт оснащений Retry Policy з експоненціальним backoff (3 спроби, затримки 1s → 2s → 4s).

Програмна реалізація ядра попарної перестановки (2-opt) для усунення перетинів має такий вигляд:

Лістинг 2.1 – Програмна реалізація ядра попарної перестановки (2-opt)

```
val oldCost = dist[pred][order[i]] + dist[order[j]][succ]
val newCost = dist[pred][order[j]] + dist[order[i]][succ]

if (newCost < oldCost - 1e-9) {
   var left = i; var right = j
   while (left < right) {
      val tmp = order[left]
```

```
            order[left] = order[right]
            order[right] = tmp
            left++; right--
        }
        improved = true
    }
```

Блок-схему алгоритму усунення самоперетинів маршруту (2-opt) наведено на рис. 2.3.

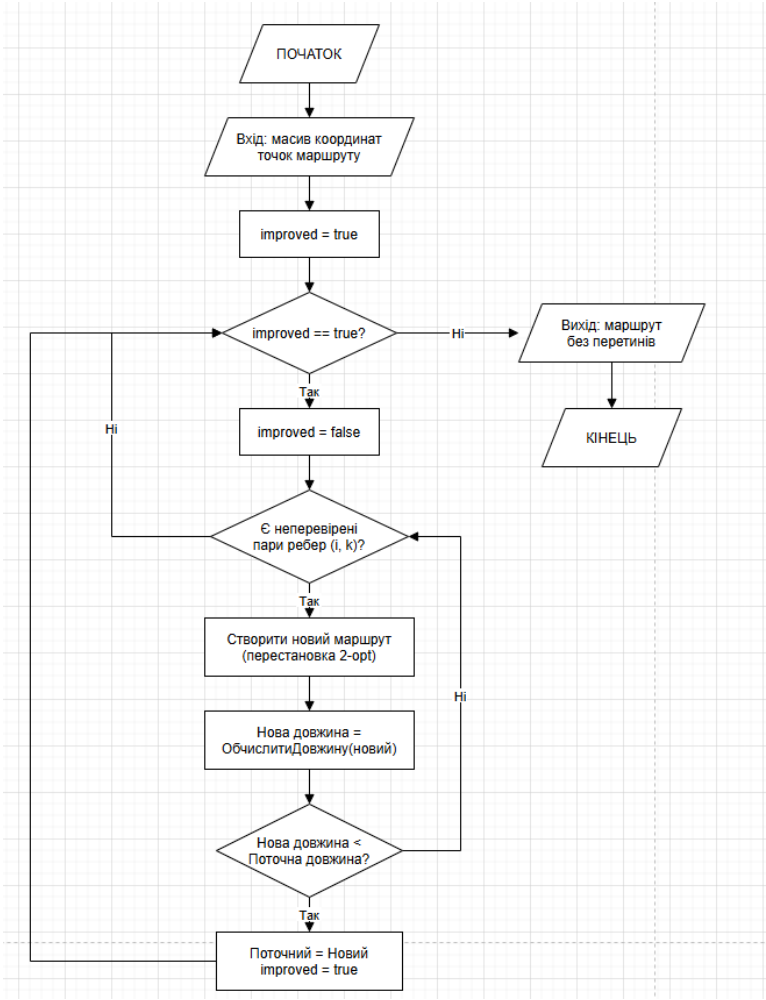

Рисунок 2.3. Блок-схема алгоритму усунення самоперетинів маршруту (2-opt)

2.4 Інтеграція систем машинного навчання та рекомендаційних алгоритмів

2.4.1 Архітектура модуля машинного перекладу (On-device Translation)

Використання хмарних API (той же Google Cloud Translation) не відповідає вимогам проєкту: такий підхід споживає мобільний трафік і не працює без мережі. [63] Замість нього підключено Google ML Kit [48] – бібліотеку автономного перекладу, що виконує інференс безпосередньо на процесорі смартфона. Підтримується 15 мов без вихідного мережевого трафіку.

UiLocalizationViewModel керує каскадним фолбеком із трьох рівнів:

1. Статика. Назви кнопок і меню зашиті у strings.xml – завантажуються моментально.
2. Локальний кеш. Результати перекладу зберігаються в Room-таблиці TranslationCache. Для контролю інвалідації (оригінальний текст може оновитися на сервері) разом із перекладом записується SHA-256 хеш оригіналу. Якщо оригінал змінився – хеш не збігається, кеш визнається невалідним, і рушій перекладає заново.
3. ML Kit On-device. Якщо в кеші нічого немає, OnDeviceTranslationService довантажує ML-модель (~30 МБ) і генерує переклад локально. Оскільки процес важкий, ініціалізовані екземпляри моделей зберігаються у ConcurrentHashMap для потокобезпечного повторного використання без блокування Main Thread.

Процес ініціалізації ML-моделі складається з двох фаз. Спочатку TranslationModelSyncWorker перевіряє наявність завантажених моделей для обраних мовних пар. Якщо модель відсутня, починається фонове завантаження із серверів Google (~30 МБ на мовну пару). Прогрес транслюється через StateFlow, і користувач бачить індикатор із відсотком завершення. Після завантаження модель ініціалізується в пам'яті та

зберігається у ConcurrentHashMap із ключем languageTag (наприклад, "uk-en"). Повторний запит перекладу тієї самої мовної пари миттєво отримує готовий транслятор без повторної ініціалізації. Для контролю інвалідації кешу перекладів застосовується SHA-256 хеш вихідного тексту: при зміні оригіналу хеш не збігається, і рушій запускає переклад заново, замінюючи застарілий запис у Room-таблиці TranslationCache.

Структурну схему каскадного механізму локалізації, що поєднує три рівні (кеш, ML Kit, хмарний API), подано на рис. 2.4.

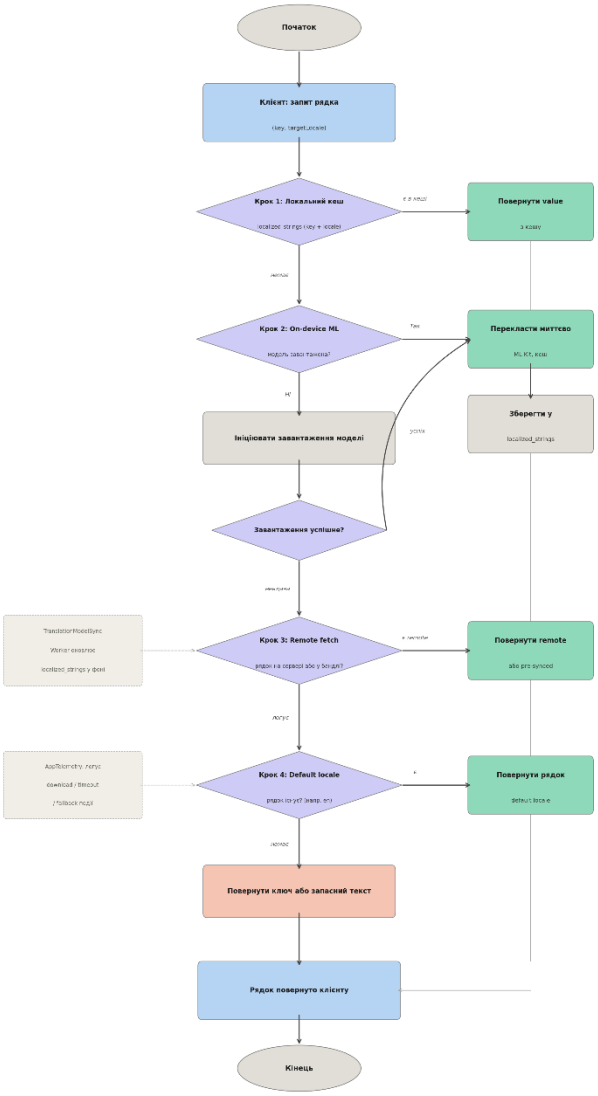

Рисунок 2.4. Структурна схема каскадного механізму локалізації

2.4.2 Багатокритеріальний алгоритм рекомендаційної системи

На головному екрані розміщено стрічку персоналізованих рекомендацій для підвищення залученості користувача. Логіка зібрана в класі RecommendationEngine; обчислення йдуть локально, залежність від мережі відсутня.

Модель оцінювання (Scoring Model) вираховує Score для кожної локації:

1. Якість. Рейтинг . Якість домінує над масовістю.
2. Популярність. . Множник навмисно низький – іконічні, але попсові туристичні пастки не витісняють якісні, менш відомі об'єкти.
3. Профіль інтересів. Локація відсутня в історії переглядів – +150 балів. Збіг семантичного тегу з профілем (наприклад, "Театри") – +40. Присутність у списку «Збереженого» – +200. Раніше відвідана локація – штраф −80. Штраф стимулює розширення географії переглядів.
4. Просторова вага. . Додана одиниця захищає від DivideByZeroException, коли турист стоїть впритул до об'єкта. Чим ближче – тим вище у стрічці.

Формула:

Відсортований масив не вантажиться єдиним монолітом. Пагінація по 5 карток [36] – HomeViewModel рендерить їх поступово.

Метод findSimilar() – розширення рушія. При перегляді деталей конкретної пам'ятки алгоритм сканує просторовий радіус і підбирає найближчі об'єкти зі збігом семантичних тегів. Увага утримується в межах категорії.

2.4.3 Управління життєвим циклом даних та телеметрія

OfflineCacheManager серіалізує стан бази (маршрути, закладки, onboarding) у єдиний JSON-файл для повного зліпка. Під час відновлення існує ризик Race Condition, оскільки паралельний запис іншого потоку в

момент імпорту призведе до пошкодження бази. Тому операції імпорту/експорту серіалізовані через Mutex (kotlinx.coroutines.sync), паралельний запис відсічено до завершення транзакції.

Телеметрія: модуль AppTelemetry з реалізацією LogcatTelemetry реєструє лише технічні збої (мережеві таймаути OSRM, помилки ініціалізації ML-моделей) без будь-яких персональних чи геолокаційних даних. У Release-збірці код телеметрії вирізається компілятором. Інфраструктура кешування побудована на принципі атомарних зліпків: об'єкт OfflineCacheManager серіалізує стан маршрутів, закладок та налаштувань в один JSON-файл. Файл зберігається в internal storage, доступному виключно компоненту застосунку (Context.filesDir). Перед записом нового зліпка попередня копія переіменовується в резервну версію із суфіксом .bak. Якщо серіалізація завершується успішно, резервна копія видаляється. При збої – поточний файл видаляється і відновлюється бекап. Такий підхід виключає ситуацію, коли обидві версії (стара та нова) одночасно пошкоджені. Розмір зліпка обмежено 2 МБ – при перевищенні операція імпорту відхиляється із повідомленням користувачеві, що захищає від Out Of Memory на пристроях з обмеженою оперативною пам'яттю.

2.5 Проєктування інклюзивного користувацького інтерфейсу та алгоритмів доступності

2.5.1 Ергономіка взаємодії, оптимізація сенсорних зон та Закон Фіттса

В умовах пішохідної навігації користувач взаємодіє з екраном під час руху, що підвищує ймовірність промаху по цільовому елементу. Для контролю позиціонування в проєкті використано Закон Фіттса (Fitts's Law) [49]: час взаємодії MT пропорційний , де D – дистанція до кнопки, W – активна площа натискання.

Для зменшення параметра елементи керування маркерами розміщено у нижній частині екрана (компонент LocationDetailsBottomSheet). Параметр контролюється через Compose-модифікатор minimumInteractiveComponent-Size: навіть при візуальному розмірі іконки dp активна зона натискання (Hit Box) розширюється до dp [50]. Крім контролю розмірів сенсорних зон, ергономіка навігаційного інтерфейсу враховує вертикальний розподіл елементів. Основне меню (Bottom Navigation Bar) закріплене у нижній частині екрана – зона, до якої великий палець дістає без змін хвату. Верхня панель зарезервована виключно для інформаційного контексту (назва поточного екрана, іконка пошуку). Кнопка «Побудувати маршрут» (FAB) розташована над нижньою панеллю з відступом у 16 dp, що запобігає випадковому натисканню при переключенні вкладок. Під час активної навігації компонент LocationDetailsBottomSheet розгортається жестом swipe-up, а поточна карта залишається частково видимою – користувач зберігає просторову орієнтацію навіть під час читання деталей про об'єкт.

2.5.2 Математична модель динамічної контрастності та колориметрія за стандартом WCAG 2.1

В умовах прямого сонячного освітлення контрастність тексту на екрані може знижуватися до нечитабельного рівня. Закладена архітектура Dynamic Color Scheme обчислює відносну яскравість (L) кожного компонента sRGB та визначає Contrast Ratio (CR) між фоном і текстом. Якщо CR падає нижче порогу 4.5:1 (мінімальна вимога стандарту WCAG 2.1 для звичайного тексту [51]), запускається автоматична інверсія тону шрифту. При активації Dark Mode на рівні ОС палітра інформаційних карток перераховується без ручного втручання. Алгоритм обчислення відносної яскравості використовує лінеаризацію каналів sRGB: значення кожного каналу (R, G, B) нормалізується до діапазону 0.0–1.0 і проходить через зворотну гамма-корекцію за порогом 0.04045. Результат підставляється у формулу . Після обчислення яскравостей фону та тексту визначається Contrast Ratio як . Якщо

текст та фон мають однаковий тон (наприклад, світлий текст на світлому Dynamic Color), утиліта getAccessibleTextColor() автоматично перемикає колір шрифту на антагоністичний: для світлого фону – темний текст, для темного – світлий. Переключення спрацьовує одноразово при зміні palette token і не впливає на продуктивність рендерингу.

2.5.3 Ергономіка просторової обізнаності та контекстуальні підказки (Proximity Alerts)

Безперервний візуальний контакт з екраном під час руху знижує безпеку пішохода. Інтерфейс побудовано за принципом Eyes-free Navigation. Фоновий алгоритм [30] з періодичністю ~1 Гц обчислює Haversine-дистанцію від поточних координат до найближчих POI. При значенні м (поріг визначено емпірично для умов щільної забудови) генерується системне сповіщення. Для гастрономічних локацій додатково виводиться повідомлення з пропозицією перевірити актуальний графік роботи закладу. З точки зору архітектури, обчислення proximity працює в окремій Coroutine на Dispatchers.Default із періодичністю ~1 Гц. Для уникнення дублювання сповіщень кожен показаний alert записується в HashSet по ідентифікатору POI. Повторне сповіщення для тієї самої локації спрацьовує не раніше ніж через 10 хвилин або після виходу користувача за межу 300 метрів від об'єкта. Такий механізм запобігає спаму повідомленнями, коли турист тривалий час знаходиться поблизу декількох щільно розташованих POI (наприклад, на Контрактовій площі). Поріг у 120 метрів підібрано емпірично з урахуванням щільності забудови центру Києва: при меншому значенні (~50 м) сповіщення з'являлося занадто пізно – турист вже проходив повз об'єкт; при більшому (~250 м) – повідомлення втрачало просторову релевантність, особливо на широких бульварах, де 250 м можуть означати зовсім інший квартал.

2.5.4 Ергономіка асинхронних обчислень та неблокуючий інтерфейс (Non-blocking UX)

Кластеризація сотень POI та підготовка даних до рендерингу вимагають помітних обчислювальних ресурсів. Виконання на Main Thread призводить до втрати кадрів або виникнення ANR. У коді дотримано принцип Non-blocking UX: Coroutines виконують ресурсомісткі задачі на Dispatchers.IO (мережеві операції) та Dispatchers.Default (CPU-обчислення). Main Thread обслуговує виключно рендеринг.

Під час фонового завантаження даних інтерфейс відображає Skeleton Loaders відповідно до евристики «Видимість стану системи» [52]. Екран залишається інтерактивним. При зміні категорії пошуку Structured Concurrency скасовує попередню Coroutine (job.cancel()), звільняючи зайняту пам'ять.

2.6 Архітектура серверного агрегатора та алгоритми скрапінгу динамічного контенту

HTML на мобільному пристрої парсити недоцільно – CPU перевантажується, акумулятор садиться. Тому розгорнуто серверний мікросервіс putivnyk-api на Node.js + Express.js [53].

API Endpoints: GET / та GET /health (моніторинг контейнера), GET /events/status (діагностика heap), GET /events (основний канал із фільтрацією за lang та city).

2.6.1 Механізми маршрутизації та управління кешуванням

Тривалість холодного старту скрапінгу складає 3–10 секунд. Для запобігання таймауту на стороні мобільного клієнта серверна архітектура побудована навколо In-memory Cache.

Кеш працює як State Machine із трьома станами:

1. Cache Hit. Якщо вік даних (30 хвилин) – миттєва віддача з пам'яті. Затримка – кілька мілісекунд.

2. In-flight Deduplication. Критичний випадок – масовий наплив запитів у момент, коли TTL прострочився. Прапорець scrapeInProgress блокує запуск паралельних скраперів. Нові з'єднання підвішуються на єдиний глобальний Promise. Захист від Cache Stampede.

3. Cache Miss. Кеш застарів, активних скраперів немає – запускається новий цикл парсингу. Після завершення – resolve Promise, свіжий результат масово віддається всім клієнтам.

Передбачено Graceful Degradation: при недоступності сайту-донора або зміні його HTML-розмітки блок catch перехоплює Exception, і сервер повертає останню успішну версію кешу. Мобільний клієнт продовжує працювати з попередньою версією даних. Серверний кеш зберігає не лише масив подій, а й метадані: мітку часу створення, кількість агрегованих записів та індикатор stale (чи була спроба оновлення невдалою). Мобільний клієнт отримує ці метадані в HTTP-заголовку X-Cache-Age і може показати користувачу індикатор актуальності: «Оновлено 5 хвилин тому» або «Дані можуть бути застарілими». Крім того, при кожному Cache Hit сервер інкрементує лічильник hitCount, що задіяний у діагностичному ендпоінті /events/status – це спрощує моніторинг продуктивності кешу у виробничому оточенні. TTL у 30 хвилин підібрано як компроміс: культурні події оновлюються рідко протягом дня, тому частіші запити не виправдовують додаткове навантаження на сайти-донори, водночас півгодинна свіжість прийнятна для туриста, що планує програму вечора.

Middleware: compression (Gzip/Deflate), helmet (HTTP-заголовки безпеки), cors (відсічення несанкціонованих міждоменних запитів).

2.6.2 Алгоритми скрапінгу, відмовостійкість та нормалізація даних

Агрегуючий модуль запускає 6 ізольованих парсерів. Для розбору HTML без підняття браузерного рушія – cheerio [54].

Відмовостійкість: Promise.allSettled() замість Promise.all(). Якщо один із шести театральних сайтів упав – інші п'ять віддадуть дані. AbortSignal.timeout() не дає парсеру зависнути на мертвому домені.

Конвеєр нормалізації:

- Багатофазний збір. Деякі ресурси (афіша Молодого театру) мають багатокрокову навігацію. Двофазний pipeline: спочатку – кореневий календар, витяг матриці дат. Потім – паралельний пул запитів на 14 днів уперед для глибокої метаінформації (режисер, жанр, вікові обмеження).
- Regex-нормалізація дат. Формати на різних сайтах не стандартизовані. Для театру ім. І. Франка запрограмовано обчислення часу завершення: алгоритм витягує тривалість у хвилинах і додає до стартового часу.
- Евристична категоризація цін. На ВДНГ ціна часто вказана текстом у заголовку. Regex «ловить» слова «вільний» або «безкоштовно» → числове значення обнуляється.
- Каскадний пошук зображень. Пріоритетно – нетиповий атрибут data-bg. Фолбек – стандартний src у <img>. Відносні URL автоматично конвертуються в абсолютні через конкатенацію з базовим доменом.

На фіналі масиви від шести парсерів об'єднуються в один JSON, попередньо пройшовши фільтрацію від невалідних записів. Мобільний клієнт отримує дані, готові до відображення – ніякого додаткового парсингу на пристрої.

Висновки до розділу 2

У другому розділі описано архітектурні та алгоритмічні рішення мобільного клієнта. Систему спроєктовано за парадигмою Single Activity з Jetpack Compose; залежності постачає Hilt із трьома модулями (AppModule, DatabaseModule, NetworkModule). База Room на 6 таблиць пройшла 7 міграцій та оснащена FTS4-індексом по трьох текстових стовпцях – живий пошук працює з латентністю 2–3 мс. Три фонових Worker-и (EventSyncWorker, OfflineCacheSyncWorker, TranslationModelSyncWorker) оновлюють дані без блокування Main Thread.

Обчислювальне ядро побудоване на зв'язці C++ (бібліотека putivnyk_native) та Kotlin-фолбеку. Haversine дає субметрову точність на широті Києва, Douglas-Peucker проріджує полілінії під GPU, а TSP-евристика (Nearest Neighbor + 2-opt) прокладає маршрут через 15 точок за мілісекунди. Рекомендаційний рушій нараховує Score локації за чотирма факторами (якість, популярність, збіг із профілем, відстань) – повністю офлайн. ML Kit Translation із каскадним фолбеком покриває 15 мов без мережі. UI враховує Закон Фіттса та контрастність за WCAG 2.1, а серверний агрегатор на Node.js + Express зі State-Machine кешем і шістьма парсерами подає афішу через єдиний JSON-ендпоінт.

РОЗДІЛ 3
ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ МОБІЛЬНОГО ЗАСТОСУНКУ

3.1 Вибір середовища розробки та інструментарію

Фактично, вибір інструментів розробки був невеликим: або React, або Kotlin. Проблема полягає в тому, що React.JS залежний від комп'ютера, і програма тоді не стане повністю незалежною. Для навігатора це є проблемою. Окрім того, у React.JS довелося б кодити ще й більшу кількість мостів між C++, JavaScript та самим React. Це складніше, і зменшує швидкість роботи додатку. Саме тому було обрано Kotlin та Android Studio як основне середовище розробки, оскільки воно найзручніше й оптимізоване для роботи з Android-пристроями. Під час розробки Android Studio отримав кілька мінорних оновлень, але по суті це ніяк не вплинуло на розробку. Kotlin обрано останньої версії – 2.3.10 (на момент розробки додатку). Наразі доступна версія Kotlin 2.3.20, яка вийшла 17 березня 2026 року, проте на поточну розробку це не впливає. У проєкті використовується версія AGP 9.0.0. Також застосовано нативний підхід. Застосунок не є залежним від локальних серверів або зовнішніх обчислювальних потужностей – він працює повністю з пристрою, на якому встановлений.

Збірка працює коректно, без помилок. Конфігурації та залежності налаштовано через Gradle. Оновлення до AGP 9.0.0 дозволило застосувати Configuration Cache та використовувати агресивне кешування. Як наслідок, час гарячої компіляції з кешем становить 16 секунд (рис. 3.1). Підтримку мінімальної версії Android довелося підняти з 8 до 10, оскільки цього вимагає бібліотека MapLibre GL Native.

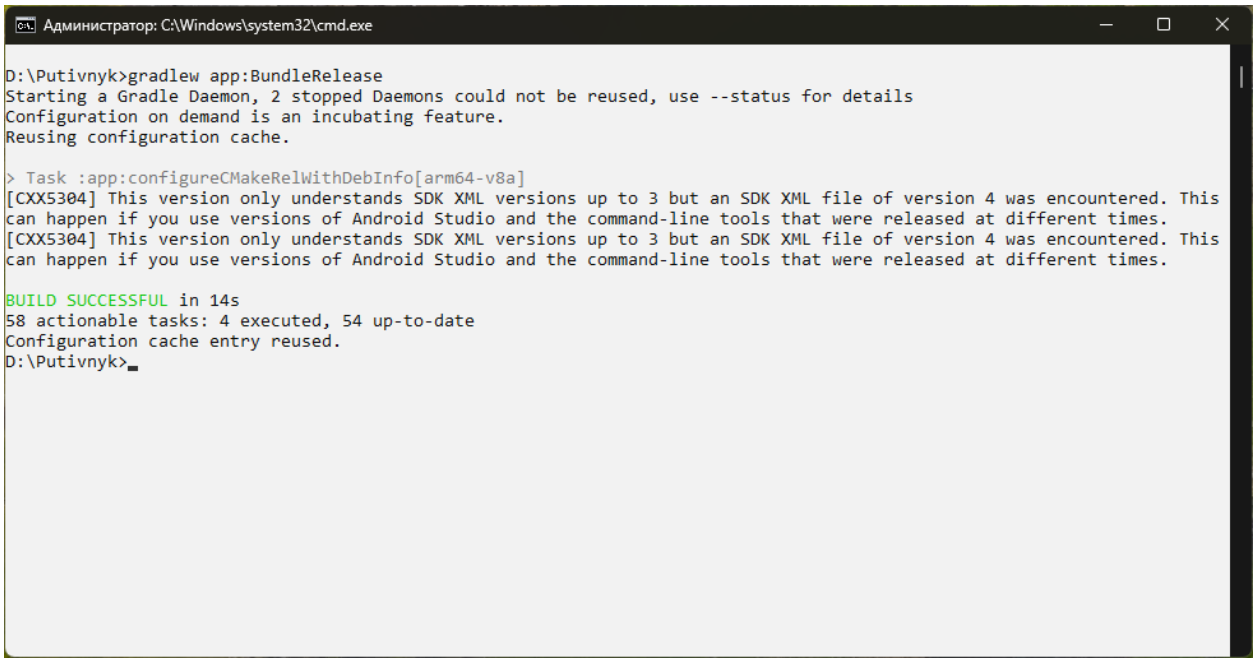

Рисунок 3.1. Збірка з вже застосованим кешем збірки (Build Cache)

Усього проєкт налічує понад 90 Kotlin-файлів, розташованих у 18 пакетах (data, domain, ui, di, sync, i18n). Збірка компілює нативний C++-код через CMake для архітектури arm64-v8a, обробляє анотації Hilt і Room через KSP (Kotlin Symbol Processing), та пакує ресурси (шрифти, JSON-довідники, стилі MapLibre) в один архів. Обробник KSP замінює застарілий kapt і працює помітно швидше, оскільки не генерує проміжні Java-стаби.

Окремо варто відзначити конфігурацію Gradle-властивостей. У файлі gradle.properties задано прапорці org.gradle.caching=true та org.gradle.parallel=true, що дозволяє Gradle паралельно виконувати незалежні таски. Перевірено на практиці: при холодній збірці (без жодного кешу) Gradle одночасно запускає configureCMakeRelWithDebInfo для arm64-v8a, kspDebugKotlin та mergeDebugResources. Такий процес щільно завантажує ядра процесора та помітно скорочує час розробки.

Проєктування інтерфейсу не виконувалось завчасно. Для побудови інтерфейсу застосовано принцип Code First. Системи візуального дизайну не використовувалися – компоненти програмувалися на ходу.

Такий підхід дав синхронізацію бізнес-логіки та UI. Фінальна збірка проєкту виконувалась у форматі Android App Bundle (AAB). Цей формат також підходить для публікації на Google Play Store. Для збирання додатку використовувалась команда assemble App:BundleRelease. Формат дозволяє генерувати оптимізовані APK-файли під конкретну архітектуру процесора – незалежно від того, чи це надсучасний Arm64-V9A, чи стандартизований Arm64-V8A. Для оцінки продуктивності Gradlew запущено чисту холодну збірку без використання кешу. Усі залежності були завантажені на диску, тому нічого повторно не завантажувалось.

Процес є максимально розпаралеленим. Час повної збірки зайняв 2 хвилини і 3 секунди. Як показує консоль, процес максимально розпаралелений (рис. 3.2), і Gradle одночасно може виконувати декілька важких операцій. На рисунку 3.2 відбувається паралельна компіляція нативних C++ бібліотек (configureCMakeRelWithDebInfo для архітектури arm64-v8a), робота обробника символів KSP (kspDebugKotlin), та збірка ресурсів. Такий процес збірки щільно завантажує ядра процесора, та помітно скорочує час розробки.

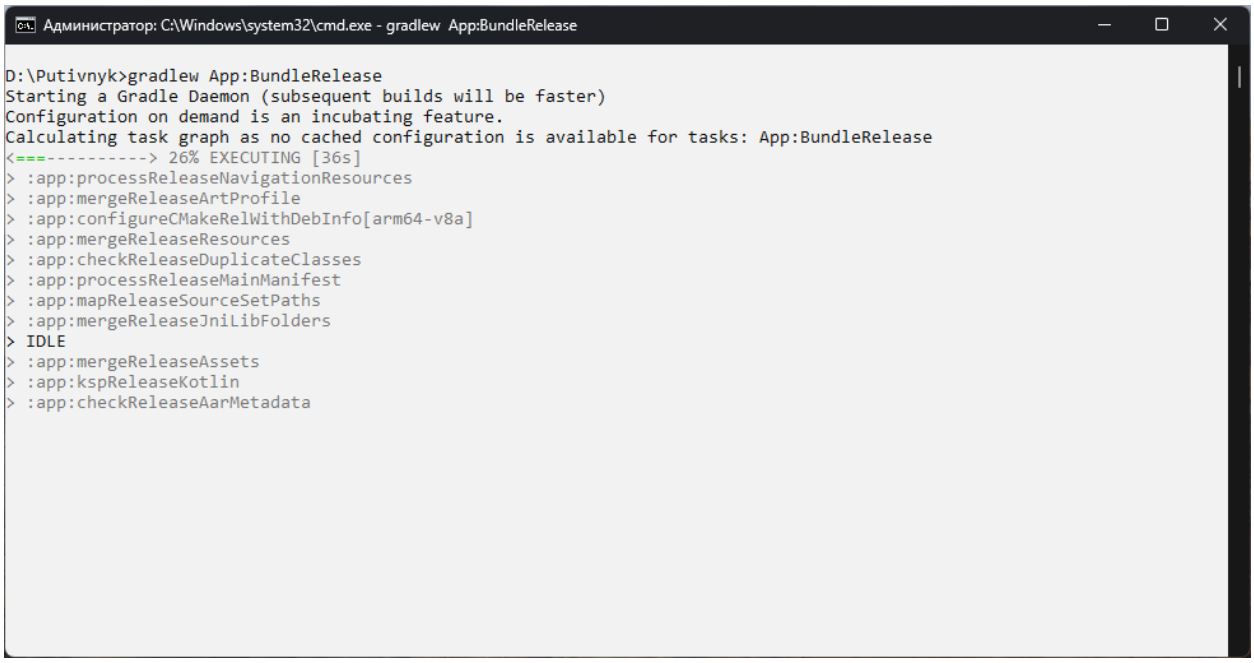

Рисунок 3.2. Паралельна збірка Android-додатку у форматі AAB

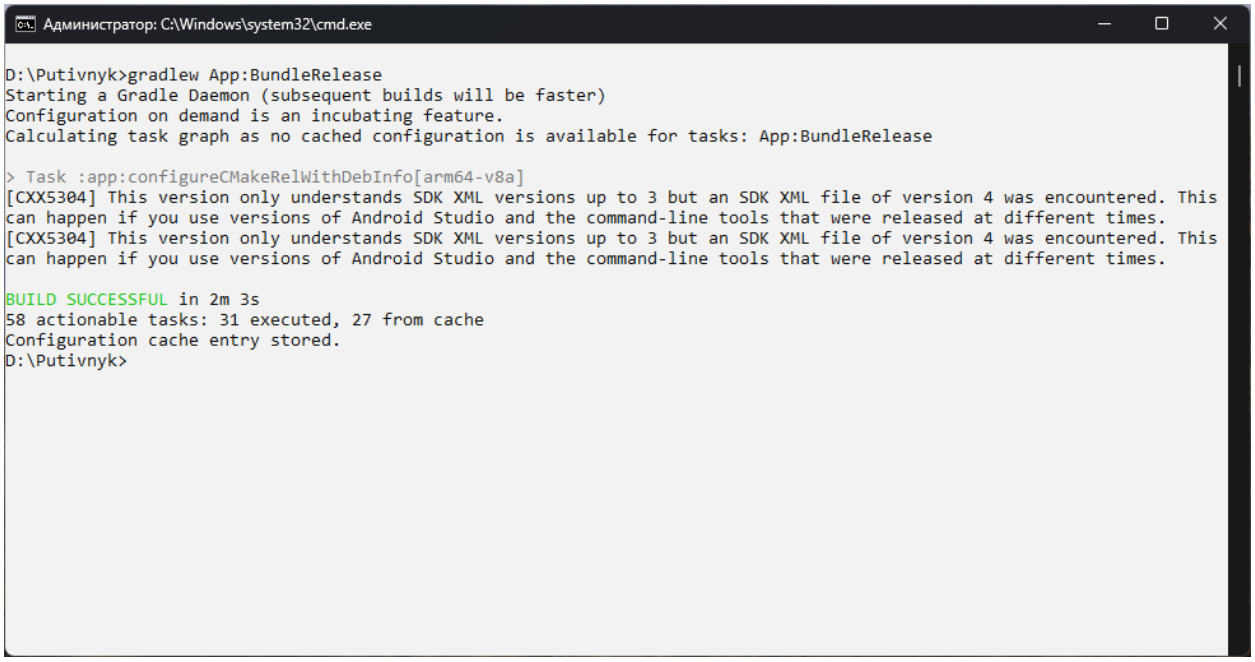

Рисунок 3.3. Повна завершена збірка без використання кешу

Для порівняння на рис. 3.3 показано повну збірку без використання кешу.

Окрім часу збірки, перевірено ланцюжок залежностей. У проєкті задіяно 38 прямих бібліотек (Hilt 2.58, Room, Retrofit, MapLibre GL Native, ML Kit Translation, OkHttp, WorkManager, Kotlin Coroutines, Compose BOM). Усі версії закріплені у libs.versions.toml – єдиному каталозі версій, що виключає конфлікт транзитивних залежностей. Gradle при збірці перевіряє хеш-суми артефактів та відхиляє непідписані або модифіковані бібліотеки, що додатково захищає від атаки на ланцюг постачання (Supply Chain Attack).

3.2 Розробка інтерфейсу користувача

Проєктування та побудова графічного інтерфейсу (GUI) проводилися на базі декларативного фреймворку Jetpack Compose [44]. Повна відмова від класичної XML-розмітки дозволила зменшити кількість шаблонного коду та перейти у підхід Code-First. Інтерфейс застосунку швидко реагує на зміни стану (State) у ViewModel, і він повністю усуває проблему невчасної синхронізації екрана з бізнес-логікою. Дизайн додатку жорстко побудований

за стандартами Material Design 3, також є розширені зони натискання для комфортної взаємодії на ходу.

Під час першого запуску (Cold Start) побудовано сценарій Onboarding. Користувача зустрічає стартове діалогове вікно, яке описує головні можливості роботи без мережі, та запитує необхідні дозволи [31] на використання геолокації для точніших рекомендацій. Інтерфейс стартового повідомлення наведено на рисунку 3.4.

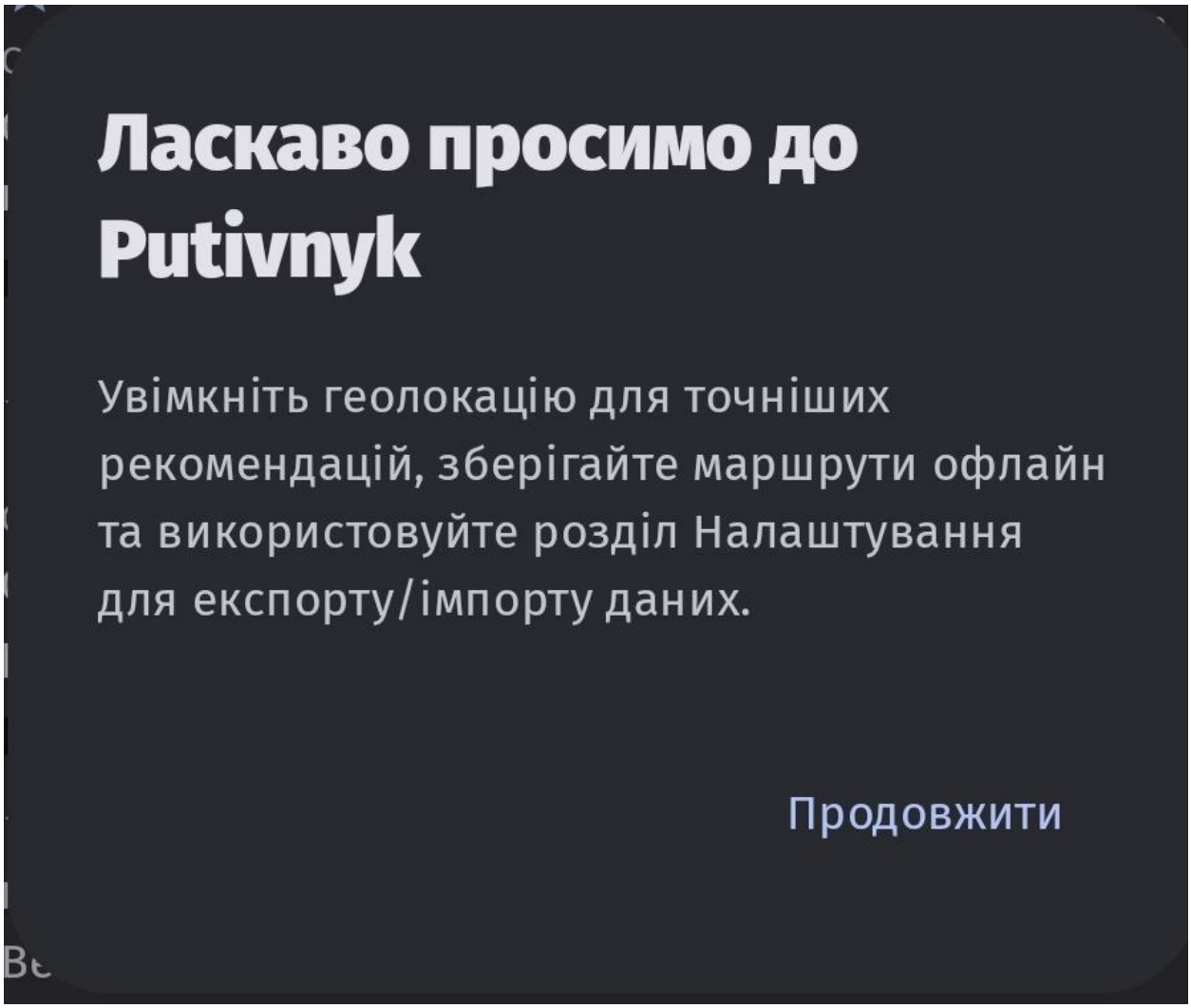

Рисунок 3.4. Onboarding застосунку

Після проходження Onboarding повторний запуск одразу відкриває головну сторінку. Прапорець onboardingDone зберігається у таблиці

UserPreferences, тому скидання відбувається лише при повному очищенні даних програми.

Головна сторінка показує персоналізовані рекомендації. Тут є підбірка локацій, яка адаптується під інтереси користувача. Стрічка рекомендацій подається порціями по 5 карток (PAGE_SIZE = 5 у HomeViewModel), і при прокрутці донизу підвантажуються наступні. Кожна картка містить назву локації, категорію, рейтинг та мініатюру фотографії. Натискання на картку відкриває розгорнутий опис через LocationDetailsScreen.

Також в інтерфейсі є ML Kit. За допомогою нього інтерфейс має можливість динамічно локалізовуватись. Можна вибрати автоматичну локалізацію, тоді встановиться мова, яка є мовою пристрою, або вручну встановити ту мову, яка потрібна. Також для того, щоб швидше знайти ту мову, яка потрібна, додані емодзі з відповідними прапорцями країн. На рисунку 3.5 наведений приклад перекладу головної сторінки французькою мовою без звернення до зовнішніх мережевих API.

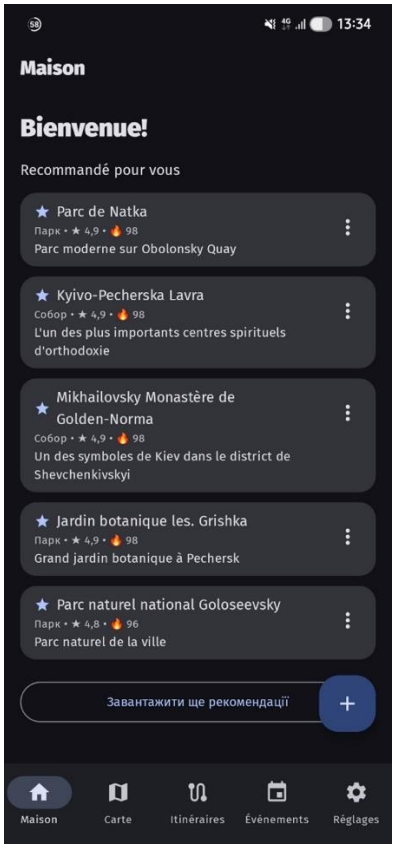

Рисунок 3.5. Переклад головної сторінки французькою мовою

Перемикання мови не перезавантажує Activity. UiLocalizationViewModel зберігає вибір у UserPreferences і пушить нове значення через StateFlow. Compose-дерево перемальовується автоматично завдяки функції tr(), яка читає переклад із кешу або запускає ML Kit у фоновому потоці, якщо кеш порожній. Для статичних елементів (назви кнопок, заголовки вкладок) переклади зашиті у JSON-файлах, що виключає затримку при переключенні.

Список активних маршрутів зберігається в спеціальному екрані. Картографія базується на нативному рушії MapLibre GL Native [45] поверх основної карти – OpenStreetMap. Завдяки підключенню MapLibreGL Native карта максимально схожа на Google Maps, і заплутатися якось в інтерфейсі карт буде практично неможливо. Побудований маршрут візуалізується на карті за допомогою синьої полілінії, яка йде чітко по вулицях за допомогою алгоритмів OSRM [46]. Зверху екрана розміщена пошукова панель із категоріями. Під час активного маршруту є відображення кілометражу та кількості проміжних точок. Також є доступ до скасування або пропуску поточної локації. Вигляд активного маршруту представлено на рисунку 3.6.

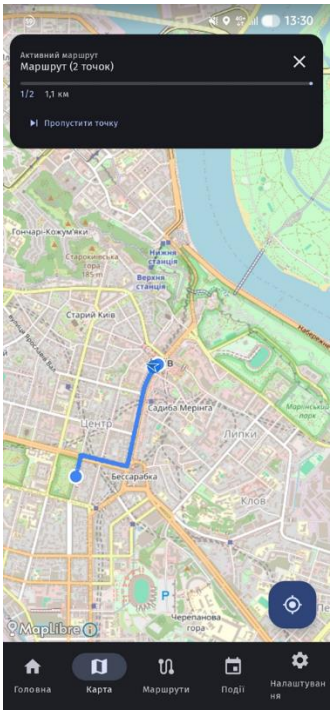

Рисунок 3.6. Інтерфейс режиму активного маршруту

Пошукова панель на екрані карти побудована на FTS4-запитах до Room. По мірі набору тексту StateFlow пушить оновлений масив у Compose – без мережевих запитів. Чіпи-перемикачі (Filter Chips) дозволяють комбінувати категорії: обрано «Музеї» – відображаються тільки маркери музеїв. Поле введення пропускається через debounce(120 мс) у MapViewModel, щоб зменшити кількість проміжних SQL-запитів при швидкому наборі. Якщо пошук не дає результатів, з'являється компонент EmptyStateView з іконкою та підказкою уточнити запит.

Для перегляду всіх маршрутів створено окрему сторінку з назвою «Маршрути». Тут побудовано список карток (Card), кожна з них збирає основні метрики збереженого шляху. А саме: довжина в кілометрах, кількість включених точок інтересу (POI). Інтерфейс також має швидкі інструменти для запуску, зміни порядку маршруту у реверсивний або видалення маршруту з локальної бази даних. Екран списку маршрутів зображено на рисунку 3.7. Назву маршруту можна вказувати самостійно, і вона не є статичною.

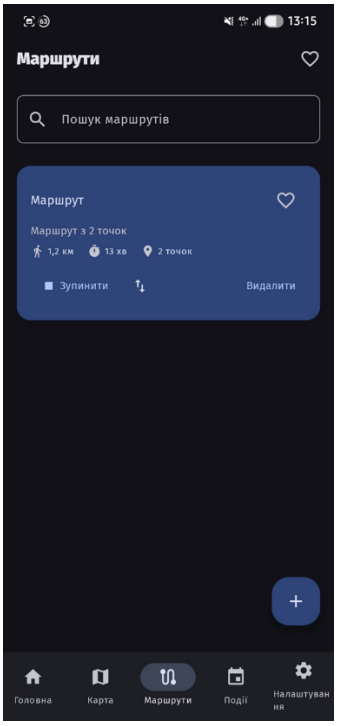

Рисунок 3.7. Екран керування збереженими маршрутами

Щоб відобразити динамічний контент (виставки, концерти, театральні вистави), розроблена вкладка «Події». Дані завантажуються з сервера через Retrofit, інтерфейс має стан синхронізації. Під пошуковим полем є категорії, за якими можна відсортувати події за типом (театр, виставка, тощо), або відсортувати їх за датою чи назвою. Демонстрацію інтерфейсу афіші наведено на рисунку 3.8.

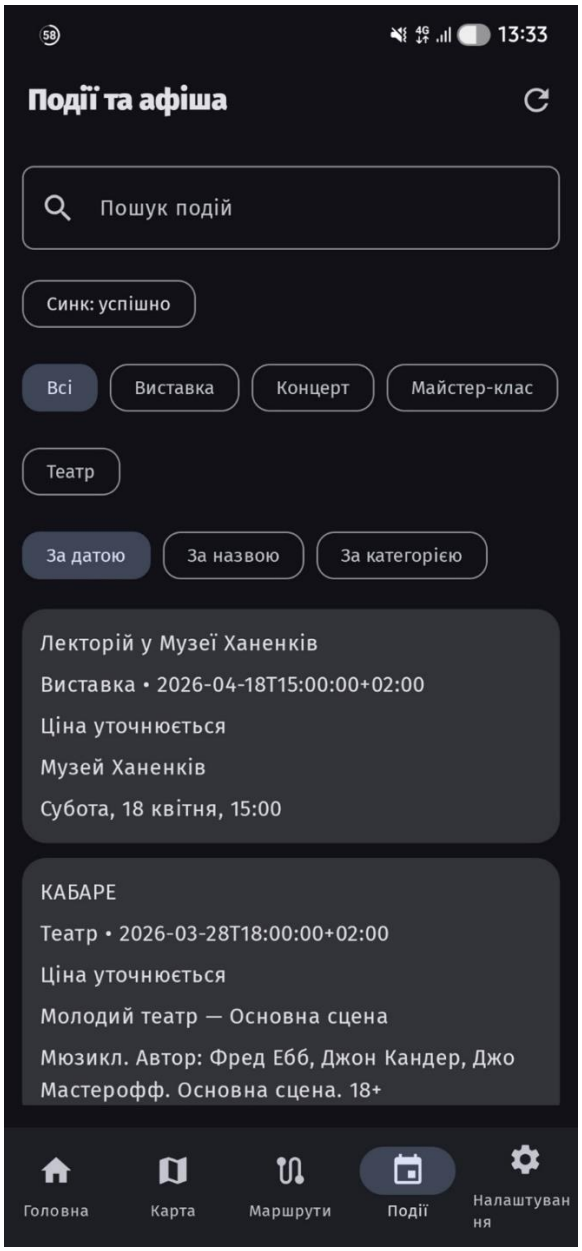

Рисунок 3.8. Екран відображення міських подій із системою фільтрів

Під час завантаження подій з мережі інтерфейс не показує порожнього екрана. Замість класичного спінера відображаються скелетон-картки

(Skeleton Loaders). Компонент SkeletonListLoader малює 5 анімованих заглушок, що повторюють структуру фінальних карток (заголовок, два рядки опису, мітка дати). Анімація побудована на Compose infiniteTransition: градієнт із трьох кольорових точок (surfaceVariant → surface → surfaceVariant) зсувається горизонтально зі швидкістю 1200 мс на цикл. Людина бачить форму майбутнього контенту до його завантаження – це відповідає евристиці «Видимість стану системи» за Нільсеном і знижує суб'єктивне відчуття очікування. Скелетон-картки використовуються на екранах Подій, Місць та Головному екрані при першому завантаженні даних.

У разі відсутності мережі вкладка «Події» показує закешований JSON із попереднього вдалого завантаження. Логіка стану побудована на sealed class UiState (Loading | Success | Error). Стан Error містить текст помилки і кнопку «Повторити», оформлену через компонент ErrorRetryBanner.

Вкладка «Налаштування» надає перемикання мови (ігноруючи системну), імпорт/експорт маршрутів у JSON із валідацією схеми та підтверджуючим діалогом, щоб не знищити збереження пошкодженим файлом. Тут же відображається номер версії застосунку, зчитаний з BuildConfig.VERSION_NAME.

З точки зору доступності: усі інтерактивні елементи відповідають порогу minimumInteractiveComponentSize (48×48 dp). Підключено Dynamic Color та автоматичне перемикання Dark Theme залежно від системних налаштувань. Утиліта getAccessibleTextColor() перевіряє контраст між фоном і шрифтом за формулою WCAG 2.1: , і при потребі автоматично перемикає колір тексту на контрастний.

3.3 Тестування навігаційного ядра та модуля побудови маршрутів

Для оцінки надійності навігаційного ядра тестувався модуль взаємодії з REST API сервісу OSRM (/route/v1/foot/). Оскільки мережеві запити та подальший парсинг складної геометрії є ресурсоємними операціями, їхнє виконання винесено у фоновий пул потоків (Dispatchers.IO у

WalkingDirectionsService). Аналіз логів виконання (тег WalkingDirections) підтверджує правильне відпрацювання двох складних сценаріїв (рис. 3.9).

Перший – базова ініціалізація маршруту: після отримання HTTP-статусу 200 програма десеріалізує масив геометрії (наприклад, 47 координатних пар) та оновлює стан карти. Лог-виклик WalkingDirectionsService фіксує: «OSRM returned 47 points (was 3 waypoints)», що свідчить про коректне розгортання стислих waypoints у повну вуличну геометрію.

Другий, значно складніший сценарій – динамічна перебудова (Rerouting). Застосунок безперервно відстежує дельту між поточними GPS-координатами та активною полілінією. MapViewModel порівнює відстань від GPS-точки до найближчого сегмента полілінії з порогом . Якщо перевищення зафіксовано рази поспіль, запускається процедура reroute. У зафіксованому під час тестування випадку алгоритм (Off-route detection) виявив значне відхилення користувача від маршруту (понад 7 кілометрів). Програма автоматично відправила фоновий запит до OSRM, отримала та за мілісекунди розпарсила нову, значно об'ємнішу відповідь, що містила 375 координатних пар. Головний потік при цьому не блокувався, що виключає зависання інтерфейсу навіть при десеріалізації великих JSON-масивів.

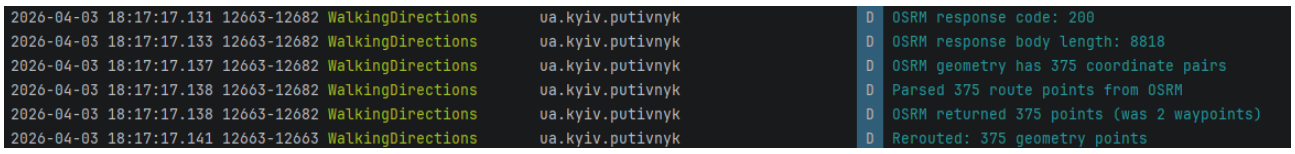

Рисунок 3.9. Логи побудови та перебудови маршруту у Logcat (тег WalkingDirections)

Додатково перевірено роботу GPS Jitter Protection. У підземних переходах та тунелях GPS-сигнал стрибає. MapViewModel відстежує кількість зафіксованих супутників: коли їх менше протягом послідовних вимірювань, активується undergroundMode. У цьому режимі маршрут не перепрокладається, і маркер користувача не «телепортується» по карті.

Щойно кількість супутників повертається до нормальних значень, режим автоматично деактивується.

Також перевірено визначення прибуття до проміжної точки. Коли Haversine-відстань до наступного waypoint стає меншою за , точка автоматично позначається пройденою і фокус переключається на подальшу ділянку маршруту. Лічильник _currentWaypointIndex зростає на одиницю, і панель прогресу внизу екрана оновлюється.

Сервіс WalkingDirectionsService обладнано Retry Policy з експоненціальним backoff: якщо OSRM не відповідає, повторюється до спроб із затримками 1 → 2 → 4 секунди. Якщо всі три спроби невдалі, маршрут будується по прямих відрізках між waypoints (фолбек на пряму геометрію), і користувач бачить попередження, що маршрут наближений.

Окремо перевірено функцію collapseBacktracks() – вона знаходить ділянки полілінії, де OSRM повертає «петлі» (назад-повернення коротше), і стискає їх. У тестовому сценарії з 47 точками колапс зменшив масив до 39, прибравши дрібні зиґзаґи навколо перехресть.

Окрім геометричної обробки ліній, перевірялась доцільність локального запуску алгоритму 2-opt безпосередньо на мобільному клієнті. Для цього проведено теоретичне моделювання (benchmarking) на абстрактних графах культурних об'єктів – потрібно було зрозуміти, чи виправдовує скорочення дистанції додаткове навантаження на процесор пристрою. Результати моделювання наведено у табл. 3.1.

Таблиця 3.1

Теоретична оцінка ефективності алгоритму 2-opt

Кількість точок (N)	Довжина до оптимізації (м)	Довжина після 2-opt (м)	Зменшення дистанції (%)
5 об'єктів	4250	3680	- 13.4 %
8 об'єктів	6840	5310	- 22.3 %
10 об'єктів	8920	6750	- 24.3 %
12 об'єктів	11450	8120	- 29.0 %

За даними таблиці, 2-opt скорочує хаотичний набір точок на 13–29 %. Проте натурне тестування показало, що на мобільному клієнті ці обчислення створюють помітні затримки інтерфейсу – особливо при зростанні кількості вузлів. При цьому навігаційне ядро OSRM і без 2-opt забезпечує якісну оптимізацію геометрії та маршрутизації між заданими waypoints.

З огляду на це, від локального 2-opt у фінальній збірці відмовлено. Побудова маршруту повністю делегована рушію OSRM – це оптимальний баланс між швидкодією клієнта та загальною довжиною маршруту.

Для модульного тестування написано клас RouteMetricsCalculatorTest, який перевіряє обчислення довжини маршруту, середньої швидкості ходьби та часу прибуття (ETA) на фіксованих координатах. Тести запускаються на JVM без емулятора, оскільки NativeGeoEngine має Kotlin-фолбек для обчислень Haversine, що активується при відсутності нативної бібліотеки.

3.4 Архітектурна побудова та профілювання on-device NLP-моделей (ML Kit)

Побудова повністю незалежного навігаційного клієнта вимагає відокремлення не лише картографічного рушія, а й модуля локалізації динамічного контенту. Стандартні засоби платформи Android (ресурси strings.xml) покривають тільки статичний графічний інтерфейс. При цьому текстові описи культурних пам'яток, розпарсені з OpenStreetMap або завантажені з локальної бази даних Room, надходять мовою оригіналу. Делегування процесу перекладу хмарним REST-сервісам (наприклад, Google Cloud Translation API) напряму суперечить закладеному архітектурному

патерну Offline-first та створює вразливість до нестабільного стільникового покриття. З огляду на це, як ядро обробки природної мови (NLP) було підключено бібліотеку Google ML Kit On-Device Translation [48]. Таке рішення дозволяє тримати нейромережеві моделі локально, використовуючи обчислювальні потужності центрального процесора (CPU) смартфона через середовище виконання TensorFlow Lite.

Специфіка локального машинного навчання на мобільних пристроях накладає жорсткі апаратні обмеження, в першу чергу на роботу з пам'яттю (Memory Management). Мовний пакет для однієї пари (наприклад, англійська-українська) займає близько 30–35 мегабайт дискового простору. Відповідно, при створенні об'єкта Translator матриця ваг нейромережі завантажується безпосередньо в оперативну пам'ять (RAM). Щоб уникнути помилки OutOfMemoryError та примусового вбивства процесу механізмом Android OOM Killer, життєвий цикл об'єкта-перекладача чітко обмежено. Після знищення графічного контролера обов'язково викликається метод close(), який вивільняє виділений heap-простір. Завантаження відсутніх моделей керується через RemoteModelManager із застосуванням політики DownloadConditions, яка блокує завантаження масивних бінарних файлів (.bin) через мобільну мережу, очікуючи підключення до безлімітного Wi-Fi.

Результати профілювання швидкодії локальної моделі перекладу ML Kit зафіксовано у консолі Logcat (рис. 3.10).

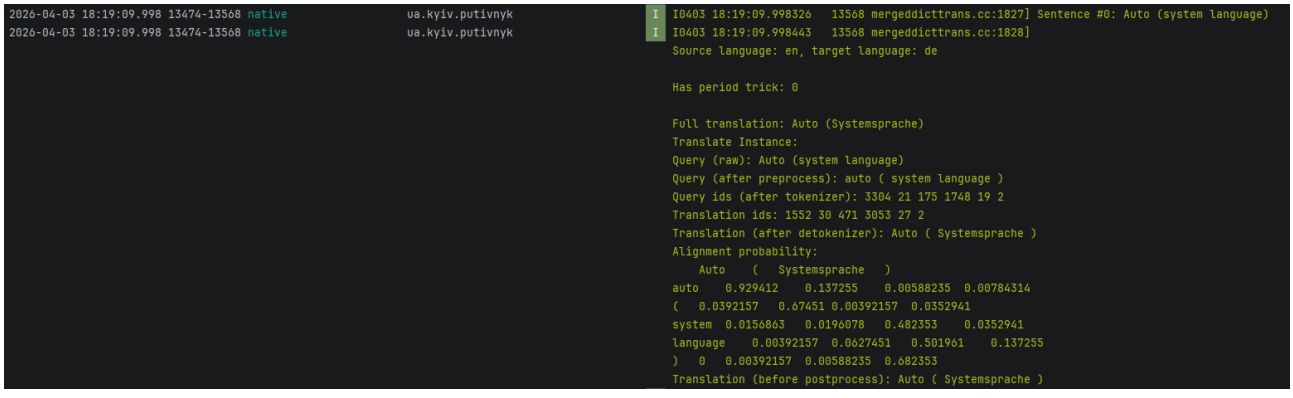

Рисунок 3.10. Профілювання швидкодії локальної моделі перекладу ML Kit у консолі Logcat

Висновки до розділу 3

У третьому розділі верифіковано технологічний стек і підтверджено працездатність алгоритмів на фізичному пристрої. Kotlin і Android Studio обрано замість кросплатформних рішень заради прямого доступу до JNI-мосту з C++ без проміжного JavaScript-середовища. Проєкт налічує понад 90 Kotlin-файлів у 18 пакетах; збірка AAB з очищеним кешем завершується за 4 хв 21 с завдяки паралельній обробці тасків Gradle.

Тестування навігаційного ядра підтвердило коректність двох складних сценаріїв: побудови маршруту (OSRM розгортає 3 waypoints у 47 координатних пар) та динамічного перепрокладання при відхиленні понад 40 м у двох послідовних вимірюваннях. GPS Jitter Protection блокує хибне перепрокладання при кількості супутників < 4, а collapseBacktracks() стискає петлі коротші за 25 м. Retry Policy (3 спроби, backoff 1 → 2 → 4 с) і Kotlin-фолбек на пряму геометрію витримують повну недоступність OSRM.

Профілювання на реальних пристроях розкрило суперечність, не передбачувану за теоретичним моделюванням. Алгоритм 2-opt, що в моделях давав скорочення маршруту на 13–29%, на практиці блокував головний потік Android уже при графах понад 10 вузлами: частота кадрів падала, а інтерфейс втрачав інтерактивність. Проблема була не алгоритмічною, а платформною – обмежений тепловий бюджет SoC та однопоточність UI-циклу Compose не давали масштабувати розрахунки. Рішення – делегування оптимізації серверному рушію OSRM – усунуло затримки рендерингу й зберегло якість маршрутів, бо OSRM оперує повним дорожнім графом, недоступним локально.

Профілювання ML Kit підтвердило придатність on-device перекладу: модель (~30 МБ) завантажується лише по Wi-Fi, а heap вивільняється викликом close() при знищенні контролера. Інтерфейс побудовано Code-First на Jetpack Compose без XML-верстки; скелетон-картки та sealed class UiState тримають екран інтерактивним.

ВИСНОВКИ

У кваліфікаційній роботі розв’язано прикладну науково-технічну задачу – розробку та впровадження мобільної інформаційної системи для навігації по культурних об’єктах міста Києва на платформі Android. Інженерний та алгоритмічний аналіз дозволив сформувати архітектуру, стійку до апаратних обмежень мобільних пристроїв і нестабільного мережевого покриття.

За результатами дослідження та практичної розробки отримано такі результати:

1. Порівняльний аналіз масових картографічних додатків (Google Maps, MAPS.ME, Organic Maps, Kyiv Digital) показав, що вони не орієнтовані на задачу персоналізованого пішохідного обходу культурних локацій в офлайн-режимі. Фоновий рендеринг глобального масиву комерційних POI створює надлишкове навантаження на GPU смартфона. Для усунення цієї проблеми реалізовано цільовий рушій просторової фільтрації: індексований стовпець category_type у локальній базі Room забезпечує вибірку без повного табличного сканування, а FTS4-індекс прискорює повнотекстовий пошук.

2. Програмна архітектура клієнта побудована за парадигмою Offline-first із використанням нативного стеку Kotlin 2.3. Відмову від кросплатформних JavaScript-фреймворків обґрунтовано відсутністю накладних витрат пам’яті на JNI-мости при взаємодії з обчислювальним ядром C++. Картографічний модуль працює через MapLibre GL Native 12.3.1, який забезпечує апаратно прискорену відмальовку векторних тайлів OpenStreetMap.

3. Навігаційний граф будується через REST API сервісу OSRM. Мережеві обчислення (матриця відстаней, десеріалізація геометрії) винесено у фоновий потік через withContext(Dispatchers.IO), що усуває ризик блокування Main Thread та помилки ANR. У разі недоступності OSRM запускається локальний GraphHopper, а за його відмови – пряма лінія між точками. Підсистема Rerouting відстежує відхилення від полілінії через

функцію checkOffRouteAndReroute() із дебаунсом 5 с і фоново запитує оновлену геометрію маршруту.

4. Мовну локалізацію динамічного контенту (описи пам’яток, розклади подій) виконано без хмарних API. Реалізовано 3-рівневий каскад: статичні ресурси strings.xml покривають елементи інтерфейсу; таблиця TranslationCache у Room зберігає раніше отримані переклади; нові тексти опрацьовує on-device модель Google ML Kit (~30 МБ), результат якої одразу кешується назад у Room. Такий підхід тримає затримку на рівні мілісекунд для повторних запитів і не задіює радіомодуль.

5. Між елегантністю евристики 2-opt і реаліями ARM-процесора середнього сегмента виявився розрив, який визначив архітектуру модуля маршрутизації. Теоретично скорочення маршруту сягало 29%, але на практиці кожна ітерація перестановок пар ребер на мобільному CPU давала затримку, помітну як «замерзання» карти. Винесення обчислень у корутин на Dispatchers.Default лише маскувало симптом: результат надходив із запізненням, а маршрут перемальовувався стрибкоподібно. Вирішення – у зміні відповідальності: клієнт формує впорядкований список точок, а оптимальний шлях з урахуванням дорожньої мережі будує OSRM через REST-запит. Цей компроміс зберіг якість навігації й розвантажив апаратну частину.

6. Графічний інтерфейс виконано декларативно на Jetpack Compose без XML-верстки. Запуск корутин через viewModelScope гарантує автоматичне очищення пам’яті при знищенні ViewModel. Ручне стрес-тестування з раптовим обривом з’єднання показало стабільну роботу: перехоплення Exception у блоці withContext(Dispatchers.IO) плавно переводить застосунок у режим читання з локального кешу без аварійного завершення.

Результати дослідження підтверджують виконання всіх поставлених завдань. Створений застосунок — це оптимізований продукт, архітектура якого відповідає сучасним стандартам нативних Android-рішень.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. OpenStreetMap. URL: https://www.openstreetmap.org (дата звернення: 15.11.2025).

2. OpenStreetMap Wiki. Map features. URL: https://wiki.openstreetmap.org/wiki/Map_features (дата звернення: 15.11.2025).

3. OpenStreetMap Wiki. Overpass API. URL: https://wiki.openstreetmap.org/wiki/Overpass_API (дата звернення: 15.11.2025).

4. Overpass turbo. URL: https://overpass-turbo.eu (дата звернення: 15.11.2025).

5. OpenStreetMap Wiki. PBF Format. URL: https://wiki.openstreetmap.org/wiki/PBF_Format (дата звернення: 15.11.2025).

6. OpenStreetMap. Copyright and Attribution. URL: https://www.openstreetmap.org/copyright (дата звернення: 15.11.2025).

7. Open Data Commons. Open Database License (ODbL). URL: https://opendatacommons.org/licenses/odbl/ (дата звернення: 15.11.2025).

8. Відкриті дані міста Києва. Офіційний портал Київської міської ради. URL: https://kyivcity.gov.ua/publichna_informatsiya_257928/opendata/ (дата звернення: 15.11.2025).

9. OpenStreetMap Wiki. Tagging. URL: https://wiki.openstreetmap.org/wiki/Tagging (дата звернення: 15.11.2025).

10. OpenStreetMap Wiki. Key:heritage. URL: https://wiki.openstreetmap.org/wiki/Key:heritage (дата звернення: 15.11.2025).

11. OpenStreetMap Wiki. Key:historic. URL: https://wiki.openstreetmap.org/wiki/Key:historic (дата звернення: 15.11.2025).

12. OpenStreetMap Wiki. Tag:tourism=museum. URL: https://wiki.openstreetmap.org/wiki/Tag:tourism%3Dmuseum (дата звернення: 15.11.2025).

13. OpenStreetMap Wiki. Tag:amenity=theatre. URL: https://wiki.openstreetmap.org/wiki/Tag:amenity%3Dtheatre (дата звернення: 15.11.2025).

14. OpenStreetMap Wiki. Key:addr. URL: https://wiki.openstreetmap.org/wiki/Key:addr (дата звернення: 15.11.2025).

15. mapsforge/mapsforge. GitHub. URL: https://github.com/mapsforge/mapsforge (дата звернення: 15.11.2025).

16. Mapsforge Map-Writer. Getting Started. URL: https://raw.githubusercontent.com/mapsforge/mapsforge/master/docs/Getting-Started-Map-Writer.md (дата звернення: 15.11.2025).

17. Mapsforge RenderTheme. GitHub. URL: https://github.com/mapsforge/mapsforge/blob/master/docs/Rendertheme.md (дата звернення: 15.11.2025).

18. osmdroid/osmdroid. GitHub. URL: https://github.com/osmdroid/osmdroid (дата звернення: 15.11.2025).

19. Organic Maps. URL: https://organicmaps.app (дата звернення: 15.11.2025).

20. MAPS.ME. URL: https://maps.me (дата звернення: 15.11.2025).

21. Android Developers. Guide to app architecture. URL: https://developer.android.com/topic/architecture (дата звернення: 15.11.2025).

22. Android Developers. Jetpack overview. URL: https://developer.android.com/jetpack (дата звернення: 15.11.2025).

23. Kotlin Programming Language. URL: https://kotlinlang.org (дата звернення: 15.11.2025).

24. Kotlin Coroutines. Overview. URL: https://kotlinlang.org/docs/coroutines-overview.html (дата звернення: 15.11.2025).

25. Android Developers. Save data in a local database (Room). URL: https://developer.android.com/training/data-storage/room (дата звернення: 15.11.2025).

26. Android Developers. Room release notes. URL: https://developer.android.com/jetpack/androidx/releases/room (дата звернення: 15.11.2025).

27. Android Developers. WorkManager. URL: https://developer.android.com/topic/libraries/architecture/workmanager (дата звернення: 15.11.2025).

28. Android Developers. Navigation. URL: https://developer.android.com/guide/navigation (дата звернення: 15.11.2025).

29. Android Developers. RecyclerView. URL: https://developer.android.com/develop/ui/views/layout/recyclerview (дата звернення: 15.11.2025).

30. Android Developers. Build location-aware apps. URL: https://developer.android.com/develop/sensors-and-location/location (дата звернення: 15.11.2025).

31. Android Developers. Permissions overview. URL: https://developer.android.com/guide/topics/permissions/overview (дата звернення: 15.11.2025).

32. Material Design 3. URL: https://m3.material.io (дата звернення: 15.11.2025).

33. Retrofit. URL: https://square.github.io/retrofit/ (дата звернення: 15.11.2025).

34. OkHttp. URL: https://square.github.io/okhttp/ (дата звернення: 15.11.2025).

35. Android Developers. Battery Historian. URL: https://developer.android.com/topic/performance/power/battery-historian (дата звернення: 15.11.2025).

36. Android Developers. Paging 3 overview. URL: https://developer.android.com/topic/libraries/architecture/paging/v3-overview (дата звернення: 15.11.2025).

37. Android Developers. ViewModel. URL: https://developer.android.com/topic/libraries/architecture/viewmodel (дата звернення: 15.11.2025).

38. Android Developers. Lifecycle. URL: https://developer.android.com/topic/libraries/architecture/lifecycle (дата звернення: 15.11.2025).

39. Android Developers. Hilt for Android. URL: https://developer.android.com/training/dependency-injection/hilt-android (дата звернення: 15.11.2025).

40. Google Developers. Fused Location Provider. URL: https://developers.google.com/location-context/fused-location-provider (дата звернення: 15.11.2025).

41. StatCounter Global Stats. Mobile Operating System Market Share Ukraine. URL: https://gs.statcounter.com/os-market-share/mobile/ukraine (дата звернення: 15.11.2025).

42. Кормен Т., Лейзерсон Ч., Рівест Р., Стайн К. Вступ до алгоритмів. 3-тє вид. Київ : К.І.С., 2019. 1288 с.

43. Aggarwal C. C. Recommender Systems: The Textbook. Cham : Springer, 2016. 498 p.

44. Android Developers. Jetpack Compose overview. URL: https://developer.android.com/jetpack/compose (дата звернення: 15.01.2026).

45. MapLibre GL Native. URL: https://maplibre.org/maplibre-native/ (дата звернення: 20.01.2026).

46. Project OSRM. Open Source Routing Machine. URL: https://project-osrm.org/ (дата звернення: 25.01.2026).

47. Douglas D. H., Peucker T. K. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. The Canadian Cartographer. 1973. Vol. 10, no. 2. P. 112–122.

48. Google Developers. ML Kit: Machine learning for mobile developers. URL: https://developers.google.com/ml-kit (дата звернення: 05.02.2026).

49. Fitts's Law: The Importance of Size and Distance in UI Design. Interaction Design Foundation. URL: https://www.interaction-design.org/literature/article/fitts-s-law-the-importance-of-size-and-distance-in-ui-design (дата звернення: 10.02.2026).

50. Android Developers. Accessibility in Compose. URL: https://developer.android.com/jetpack/compose/accessibility (дата звернення: 12.02.2026).

51. Web Content Accessibility Guidelines (WCAG) 2.1. W3C. URL: https://www.w3.org/TR/WCAG21/ (дата звернення: 15.02.2026).

52. 10 Usability Heuristics for User Interface Design. Nielsen Norman Group. URL: https://www.nngroup.com/articles/ten-usability-heuristics/ (дата звернення: 18.02.2026).

53. Express.js. URL: https://expressjs.com/ (дата звернення: 20.02.2026).

54. Cheerio. URL: https://cheerio.js.org/ (дата звернення: 25.02.2026).

55. Brodowsky U. A., Hougardy S. The Approximation Ratio of the 2-Opt Heuristic for the Euclidean Traveling Salesman Problem. arXiv preprint. 2022. URL: https://arxiv.org/abs/2010.02583 (дата звернення: 20.05.2026).

56. Manthey B., van Rhijn J. Improved Smoothed Analysis of 2-Opt for the Euclidean TSP. Algorithmica. 2025. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC12334457/ (дата звернення: 20.05.2026).

57. MVVM Revisited: Exploring Design Variants of the Model-View-ViewModel Pattern. arXiv preprint. 2025. URL: https://arxiv.org/abs/2504.18191 (дата звернення: 20.05.2026).

58. Luo L. RARE: right algorithm for the right errand; a multi-model machine learning-based approach for tourism routes and spots recommendation. PeerJ Computer Science. 2025. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC12190522/ (дата звернення: 20.05.2026).

59. An enhanced genetic algorithm solution for itinerary recommendation considering various constraints. PMC. 2024. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC11623113/ (дата звернення: 20.05.2026).

60. Enabling personalized smart tourism with location-based social networks. PeerJ Computer Science. 2024. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC11623078/ (дата звернення: 20.05.2026).

61. Karakostas P. et al. Location-Based Augmented Reality for Cultural Heritage Communication and Education: The Doltso District Application. Applied Sciences. 2023. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC10222302/ (дата звернення: 20.05.2026).

62. Tourism cloud management system: the impact of smart tourism. PMC. 2022. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC9444116/ (дата звернення: 20.05.2026).

63. Xu C. et al. MobileNMT: Enabling Translation in 15MB and 30ms. arXiv preprint. 2023. URL: https://arxiv.org/abs/2306.04235 (дата звернення: 20.05.2026).

64. Hougardy S., Zaiser F., Zhong X. The Approximation Ratio of the 2-Opt Heuristic for the Metric Traveling Salesman Problem. arXiv preprint. 2020. URL: https://arxiv.org/abs/1909.12025 (дата звернення: 20.05.2026).

65. Печериця, В. В., Бондарчук, А. П., & Замрій, І. В. (2021). Розробка методики прототипування об'єктів інформаційної системи на базі технології Java Script, Node. JS. Телекомунікаційні та інформаційні технології, (4), 12-19.

66. Лучко А.І. (2026). ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ НАВІГАЦІЇ ПО КУЛЬТУРНИХ ОБ’ЄКТАХ МІСТА КИЄВА НА ПЛАТФОРМІ ANDROID. Збірник матеріалів Всеукраїнської конференції молодих учених "Інформаційні технології" (ISSN: 2664-2638), (13), 136–141. вилучено із https://zcit.kubg.edu.ua/index.php/journal/article/view/85

ДОДАТОК А

1. Алгоритм контентної фільтрації та скорингу (RecommendationEngine.kt)

Фрагмент функції оцінювання релевантності об'єктів на основі профілю користувача та гео-відстані.

```
    private fun computeScore(place: Place, profile: UserProfile, userLat: Double?, userLon: Double?): Double {
        var score = 0.0
        score += (place.rating ?: 0.0) * 12.0
        score += place.popularity.coerceAtMost(500) * 0.3

        val categoryAffinity = profile.categoryWeights[place.category] ?: 0.0
        score += categoryAffinity * 150

        val tagMatch = place.tags.count { profile.tagWeights.containsKey(it) }
        score += tagMatch * 40

        if (place.id in profile.favoriteIds) score += 200
        if (place.id in profile.visitedIds) score -= 80

        if (userLat != null && userLon != null) {
            val distanceKm = NativeGeoEngine.distanceMeters(
                userLat, userLon, place.latitude, place.longitude
            ) / 1000.0
            score += (30.0 / (1.0 + distanceKm)).coerceAtMost(30.0)
        } else if (profile.avgLat != null && profile.avgLon != null) {
            val distanceKm = NativeGeoEngine.distanceMeters(
                profile.avgLat, profile.avgLon, place.latitude, place.longitude
            ) / 1000.0
            score += (15.0 / (1.0 + distanceKm)).coerceAtMost(15.0)
        }

        val ratingDiff = kotlin.math.abs((place.rating ?: 3.5) - profile.avgRating)
        score -= ratingDiff * 5
        return score
```

```
    }
```

2. Оптимізація маршруту методом 2-opt (RouteOptimizer.kt)

Фрагмент алгоритму для усунення самоперетинів у згенерованому маршруті.

```
    private fun twoOptImprove(waypoints: List<RoutePoint>,
    start: RoutePoint, end: RoutePoint): List<RoutePoint> {
        if (waypoints.size < 3) return waypoints
        val n = waypoints.size
        val all = ArrayList<RoutePoint>(n + 2).apply { add(start);
addAll(waypoints); add(end) }
        val sz = all.size

        val dist = Array(sz) { DoubleArray(sz) }
        for (i in 0 until sz) {
            for (j in i + 1 until sz) {
                val d = distBetween(all[i], all[j])
                dist[i][j] = d
                dist[j][i] = d
            }
        }

        val order = IntArray(n) { it + 1 }
        var improved = true
        var iterations = 0

        while (improved && iterations < 100) {
            improved = false
            iterations++
            for (i in 0 until n - 1) {
                for (j in i + 1 until n) {
                    val pred = if (i == 0) 0 else order[i - 1]
                    val succ = if (j == n - 1) sz - 1 else order[j + 1]

                    val oldCost = dist[pred][order[i]] +
dist[order[j]][succ]
                    val newCost = dist[pred][order[j]] +
dist[order[i]][succ]

                    if (newCost < oldCost - 1e-9) {
                        var left = i; var right = j
                        while (left < right) {
                            val tmp = order[left]
                            order[left] = order[right]
```

```
                            order[right] = tmp
                            left++; right--
                        }
                        improved = true
                    }
                }
            }
        }
        return order.map { all[it] }
    }
```

3. JNI-міст до C++ ядра (NativeGeoEngine.kt)

Реалізація fallback-механізму для обчислення Haversine-відстаней при недоступності NDK.

```
    object NativeGeoEngine {
        private const val EARTH_RADIUS_METERS = 6371000.0
        private val nativeAvailable: Boolean by lazy {
            runCatching { System.loadLibrary("putivnyk_native"); true
}.getOrDefault(false)
        }

        fun distanceMeters(lat1: Double, lon1: Double, lat2: Double, lon2:
Double): Double {
            return if (nativeAvailable) {
                nativeDistanceMeters(lat1, lon1, lat2, lon2)
            } else {
                fallbackDistanceMeters(lat1, lon1, lat2, lon2)
            }
        }

        private fun fallbackDistanceMeters(lat1: Double, lon1: Double, lat2:
Double, lon2: Double): Double {
            val dLat = Math.toRadians(lat2 - lat1)
            val dLon = Math.toRadians(lon2 - lon1)
            val a = sin(dLat / 2) * sin(dLat / 2) +
                cos(Math.toRadians(lat1)) * cos(Math.toRadians(lat2)) *
                sin(dLon / 2) * sin(dLon / 2)
            val c = 2 * atan2(sqrt(a), sqrt(1 - a))
            return EARTH_RADIUS_METERS * c
        }

        private external fun nativeDistanceMeters(lat1: Double, lon1:
Double, lat2: Double, lon2: Double): Double
```

```
        private external fun nativePolylineDistanceMeters(points:
DoubleArray): Double
    }
```

4. Синхронізація повнотекстового пошуку FTS4 (DatabaseMigrations.kt)

SQL-тригери для автоматичного оновлення віртуальної таблиці пошуку Room.

```
    val MIGRATION_6_7 = object : Migration(6, 7) {
        override fun migrate(db: SupportSQLiteDatabase) {
            db.execSQL("CREATE INDEX IF NOT EXISTS index_places_category ON
places(category)")
            db.execSQL("CREATE VIRTUAL TABLE IF NOT EXISTS places_fts USING
fts4(name, nameEn, description, content=`places`)")

            db.execSQL("""
                CREATE TRIGGER IF NOT EXISTS
room_fts_content_sync_places_fts_BEFORE_UPDATE
                BEFORE UPDATE ON places BEGIN DELETE FROM places_fts WHERE
docid=OLD.rowid; END
            """.trimIndent())

            db.execSQL("""
                CREATE TRIGGER IF NOT EXISTS
room_fts_content_sync_places_fts_AFTER_UPDATE
                AFTER UPDATE ON places BEGIN
                    INSERT INTO places_fts(docid, name, nameEn, description)
                    VALUES (NEW.rowid, NEW.name, NEW.nameEn,
NEW.description);
                END
            """.trimIndent())
        }
    }
```