МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА МОБІЛЬНОГО ЗАСТОСУНКУ ПРОГНОЗУ ПОГОДИ З ВИКОРИСТАННЯМ МЕТОДІВ МАШИННОГО НАВЧАННЯ»**

Виконав: студент 4 курсу, групи 6.1211-2пі
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
А.С. Зарубін
(ініціали та прізвище)
Керівник доцент кафедри програмної інженерії, доцент, к.ф.-м.н. Горбенко В.І.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Решевська К.С.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ “ _____ 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Зарубіну Артему Сергійовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка мобільного застосунку прогнозу погоди з використанням методів машинного навчання

керівник роботи Горбенко Віталій Іванович, к.ф.-м.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 20 » грудня 2024 року № 2001-с

2. Строк подання студентом роботи 02.06.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.
3. Результати попереднього аналізу та проєктні рішення.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Розгляд аналогічних варіантів та визначення вимог.
2. Розробка архітектури, бази даних та ML-елемента програми.
3. Розробка інтерфейсу, реалізація основних функцій та інтеграція ML-моделі.
4. Перевірка програми, оцінка здобутих результатів та підсумки.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 26.12.2024 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1.	Створення плану, дослідження предметної області.	20.01.2025	
2.	Збір первинних даних, вивчення аналогів, деталізація вимог.	03.02.2025	
3.	Проєктування архітектури, UML-діаграм та бази даних.	17.02.2025	
4.	Розробка та тренування ML-моделі, конвертація в TFLite.	03.03.2025	
5.	Реалізація мережевого шару та шару даних.	24.03.2025	
6.	Розробка UI (Jetpack Compose, піксель-арт), навігації та екранів.	14.04.2025	
7.	Інтеграція ML-моделі та локальної бази даних (Room).	05.05.2025	
8.	Перевірка, виправлення помилок, завершення коду.	19.05.2025	
9.	Складання та оформлення пояснювальної записки.	22.05.2025	
10.	Нормоконтроль.	27.05.2025	
11.	Подання остаточної роботи на кафедру.	02.06.2025	
12.	Захист кваліфікаційної роботи.	18.06.2025	

Студент _____ А.С. Зарубін
(підпис) (ініціали та прізвище)

Керівник роботи _____ В.І. Горбенко
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка мобільного застосунку прогнозу погоди з використанням методів машинного навчання»: 79 с., 20 рис., 2 табл., 29 джерел, 2 додатки.

МАШИННЕ НАВЧАННЯ, ПРОГНОЗ ПОГОДИ, ANDROID, JETPACK COMPOSE, KOTLIN, TENSORFLOW LITE.

Об’єкт дослідження – архітектурні методи, програмні рішення та технології для розробки мобільного застосунку на платформі Android.

Мета цієї роботи полягає в створенні функціонального прототипу мобільного додатку для прогнозу погоди на платформі Android, що забезпечує користувачів актуальною метеорологічною інформацією та уточненими даними температури «відчувається як» за допомогою локальної моделі машинного навчання.

Методи дослідження – аналіз науково-технічної літератури, об’єктно-орієнтоване проєктування, техніки тестування програмного забезпечення, емпіричні методи.

У роботі показано процес створення мобільного застосунку, включаючи аналіз вимог, проєктування архітектури MVVM та вибір технологій. У роботі застосовано методи отримання та обробки метеорологічних даних з API, локального зберігання даних за допомогою Room та підготовки ознак для моделі машинного навчання.

У роботі розроблено інтерфейс у стилі піксель-арт, що відображає поточну погоду, погодинний та денний прогнози. Реалізовано функціонал геолокації, пошуку та збереження обраних міст з навігацією між ними за допомогою пейджера, а також інтегровано ML-модель для уточнення «відчувається як» температури. Передбачено налаштування одиниць виміру та теми, проведено ручне тестування.

SUMMARY

Bachelor’s qualifying paper “Development of a Weather Forecast Mobile Application Using Machine Learning Methods”: 79 pages, 20 figures, 2 tables, 29 references, 2 supplements.

ANDROID, JETPACK COMPOSE, KOTLIN, MACHINE LEARNING, TENSORFLOW LITE, WEATHER FORECAST.

The object of the study is architectural methods, software solutions, and technologies for the development of a mobile application on the Android platform.

The aim of this work is to create a functional prototype of a mobile weather forecast application for the Android platform, which provides users with up-to-date meteorological information and refined “feels like” temperature data using a local machine learning model.

Research methods include analysis of scientific and technical literature, object-oriented design, software testing techniques, and empirical methods.

The work demonstrates the process of creating a mobile application, including requirements analysis, MVVM architecture design, and technology selection. The work applies methods for obtaining and processing meteorological data from APIs, local data storage using Room, and feature preparation for a machine learning model.

A pixel art style user interface was developed, displaying current weather, hourly, and daily forecasts. Functionality for geolocation, location search, and saving selected cities with navigation between them using a pager was implemented, along with the integration of an ML model to refine the “feels like” temperature. Settings for measurement units and themes are provided, and manual testing has been conducted.

ЗМІСТ

Завдання на кваліфікаційну роботу……..2
Реферат 4
Summary 5
Вступ..... 8
1 Розгляд аналогічних варіантів та визначення вимог 10
1.1 Огляд предметної області прогнозування погоди 10
1.2 Аналіз існуючих мобільних застосунків прогнозу погоди..... 11
1.3 Обґрунтування машинного навчання в уточненні прогнозу..... 14
1.4 Висновки до розділу 1 16
2 Розробка архітектури, бази даних та ML-елемента програми 18
2.1 Формулювання вимог до системи..... 18
2.1.1 Функціональні вимоги..... 18
2.1.2 Нефункціональні вимоги..... 20
2.2 Обґрунтування вибору архітектури та технологій 22
2.3 Розробка моделей та діаграм системи 25
2.3.1 Функціональна схема та структурна схема модулів 25
2.3.2 Діаграма компонентів..... 27
2.3.3 Діаграма класів..... 29
2.3.4 Діаграма послідовностей..... 31
2.4 Проєктування компонента машинного навчання..... 33
2.4.1 Постановка задачі машинного навчання та вибір цільової змінної 33
2.4.2 Вибір та підготовка даних для тренування моделі..... 34
2.4.3 Вибір ознак для моделі..... 35
2.4.4 Архітектура моделі, тренування та оцінка..... 36
2.4.5 Конвертація моделі у формат TensorFlow Lite та підготовка для інтеграції 37
2.5 Висновки до розділу 2 38

3 Розробка інтерфейсу, реалізація основних функцій та інтеграція ML-моделі 40
3.1 Опис середовища розробки та інструментів 40
3.2 Реалізація шарів застосунку 40
3.2.1 Мережевий шар та робота з API 41
3.2.2 Шар даних: репозиторій, локальне сховище та перетворення даних 43
3.2.3 Шар бізнес-логіки та ViewModel 45
3.3 Реалізація користувацького інтерфейсу у стилі піксель-арт 47
3.3.1 Головний екран з пейджером погоди (WeatherScreen) 48
3.3.2 Екран управління збереженими локаціями (ManageCitiesScreen) 50
3.4 Інтеграція моделі машинного навчання TensorFlow Lite 55
3.5 Висновки до розділу 3 58
4 Перевірка програми, оцінка здобутих результатів та підсумки 60
4.1 Стратегія та види тестування 60
4.2 Ручне тестування основного функціоналу та сценаріїв 61
4.3 Unit-тестування компонентів viewmodel 62
4.4 Аналіз результатів роботи моделі машинного навчання 65
4.5 Висновки до розділу 4 66
Висновки 68
Перелік посилань 70
Додаток А Скриншоти інтерфейсу мобільного застосунку 73
Додаток Б Фрагменти програмного коду 78
Б.1 Лістинг фрагменту класу mainviewmodel.kt – управління станом пейджера та завантаження даних 78
Б.2 Лістинг фрагменту класу weathermodelinterpreterimpl.kt – підготовка ознак та отримання передбачення 79

ВСТУП

Стрімкий розвиток інформаційних технологій зумовив глибоке проникнення мобільних застосунків у всі сфери людського життя, включаючи отримання оперативної інформації, такої як прогноз погоди. Актуальні та точні метеорологічні дані є важливими для ефективного планування, забезпечення безпеки та комфорту. Хоча ринок мобільних погодних сервісів є досить насиченим, більшість існуючих рішень характеризуються стандартизованим функціоналом та дизайном. Це створює простір для розробки застосунків, що пропонують не лише прогноз, а й унікальний досвід, зокрема шляхом використання оригінальних візуальних стилів та інноваційних підходів до представлення даних.

Актуальність даної кваліфікаційної роботи визначається кількома аспектами. По-перше, зберігається стабільний попит користувачів на зручні та точні погодні інформери. По-друге, інтеграція методів машинного навчання (МН) відкриває перспективи для підвищення точності локальних прогнозів шляхом врахування мікрокліматичних особливостей, що не завжди повною мірою реалізовано у глобальних моделях, які використовуються стандартними API. По-третє, оригінальність інтерфейсу, у стилі піксель-арт, може стати вагомою перевагою для залучення та утримання цільової аудиторії, пропонуючи свіжий погляд на звичний функціонал.

Метою даної кваліфікаційної роботи є розробка функціонального прототипу мобільного застосунку прогнозу погоди “Pixel Weather” для платформи Android. Ключовими особливостями застосунку є надання актуальної метеорологічної інформації з відкритого API, уточнення показника температури, що відчувається як (англ. Feels Like Temperature, далі – FLT) за допомогою локально інтегрованої моделі машинного навчання, а також реалізація інтерфейсу в унікальному стилі піксель-арт.

Для досягнення поставленої мети в роботі вирішувались наступні завдання: аналіз предметної області та існуючих аналогів; формулювання

вимог до застосунку; обґрунтування вибору архітектури та технологічного стеку; проєктування бази даних; розробка та інтеграція ML-моделі; реалізація основного та додаткового функціоналу; розробка інтерфейсу у стилі піксель-арт; проведення тестування.

Об’єктом дослідження є процес розробки мобільного застосунку для прогнозування погоди з використанням методів машинного навчання. Предметом дослідження виступають архітектурні рішення, програмні засоби та технології, застосовані при створенні застосунку “Pixel Weather”. Методологія дослідження включала аналіз літератури, системний аналіз, об’єктно-орієнтоване проєктування з використанням UML, методи машинного навчання та тестування програмного забезпечення.

Наукова новизна полягає у підході до інтеграції локальної ML-моделі для уточнення прогнозу “FLT” в мобільному застосунку з піксель-арт інтерфейсом. Практична цінність роботи полягає у створенні функціонального прототипу застосунку “Pixel Weather”, що демонструє можливості поєднання сучасних технологій Android-розробки, машинного навчання та оригінального дизайну для покращення досвіду.

Робота складається зі вступу, чотирьох розділів, висновків, переліку посилань та додатків. Перший розділ присвячено аналізу предметної області, існуючих рішень та формулюванню вимог. У другому розділі описано проєктування архітектури, бази даних та ML-компонента. Третій розділ розкриває процес реалізації програмного продукту. Четвертий розділ містить опис тестування та аналіз результатів.

1 РОЗГЛЯД АНАЛОГІЧНИХ ВАРІАНТІВ ТА ВИЗНАЧЕННЯ ВИМОГ

1.1 Огляд предметної області прогнозування погоди

Прогнозування погодних умов є невіддільною складовою сучасного інформаційного простору, що впливає на планування діяльності людини в різноманітних сферах – від повсякденного життя до функціонування цілих галузей економіки, таких як сільське господарство, транспорт, енергетика та туризм. Своєчасне отримання достовірної метеорологічної інформації дозволяє не тільки оптимізувати щоденні завдання, але й мінімізувати потенційні ризики, пов’язані з несприятливими атмосферними явищами, забезпечуючи таким чином підвищений рівень безпеки та комфорту. У цьому контексті мобільні застосунки стали одним із ключових інструментів доступу до погодних даних завдяки їх повсюдній доступності, оперативності оновлення інформації та інтерактивним можливостям.

Сучасні користувачі очікують від погодних інформерів не лише базових показників, таких як температура повітря, атмосферний тиск, вологість, швидкість та напрямок вітру, але й більш специфічних даних, наприклад, «відчуваної» температури, яка враховує комбінований вплив температури, вологості та вітру на тепловідчуття людини. Також важливим є надання прогнозів різної строковості: короткострокових (погодинних) для найближчої доби та середньострокових (на кілька днів) для більш тривалого планування. Основними джерелами даних для таких застосунків виступають спеціалізовані вебсервіси (API), що надають оброблену інформацію з глобальних та регіональних метеорологічних моделей [10-12].

Незважаючи на велику кількість існуючих на ринку мобільних погодних застосунків, більшість з них пропонують схожий набір функцій та дотримуються стандартизованих підходів до дизайну користувацького інтерфейсу [21]. Це створює певну одноманітність та відкриває можливості для розробки нішевих продуктів, які можуть запропонувати користувачам не

тільки функціональність, але й унікальний візуальний досвід. Крім того, актуальною залишається задача підвищення точності та персоналізації прогнозів. Стандартні API, хоча й надають великий обсяг даних, не завжди можуть врахувати локальні мікрокліматичні особливості або надати максимально точний прогноз для конкретного користувача. Застосування методів машинного навчання для локального уточнення прогнозів, отриманих з API, є перспективним напрямком, що може підвищити цінність мобільного застосунку. Використання нетрадиційних стилів оформлення, як-от піксель-арт, може привернути увагу певної аудиторії та зробити взаємодію із застосунком більш цікавою та емоційно забарвленою. Таким чином, розробка мобільного застосунку прогнозу погоди, що поєднує стандартний функціонал з інноваційними підходами до уточнення даних за допомогою машинного навчання та оригінальним візуальним стилем, є актуальним та практично значущим завданням.

1.2 Аналіз існуючих мобільних застосунків прогнозу погоди

Для визначення конкурентних переваг та функціональних особливостей майбутнього програмного продукту було проведено детальний аналіз існуючих на ринку мобільних застосунків, призначених для прогнозування погоди. Дослідження охоплювало популярні та високо оцінені користувачами рішення, що дозволило сформувати уявлення про поточні тенденції та очікування цільової аудиторії. Основними об’єктами для порівняльного аналізу було обрано чотири відомі застосунки: AccuWeather [22], Weather Channel [23], Yahoo Weather [24] та Google Weather (вбудований функціонал пошукової системи Google) [25].

Порівняння аналогів здійснювалося за набором попередньо визначених критеріїв, що відображають ключові аспекти якості та функціональності погодних інформерів. До таких критеріїв було віднесено: точність прогнозування (A1), детальність наданої метеорологічної інформації (A2),

якість користувацького інтерфейсу та досвіду взаємодії (UI/UX) (А3), наявність додаткових функцій (наприклад, погодні карти, сповіщення, віджети) (А4), та відсутність або мінімальний обсяг рекламних оголошень (А5).

Для об’єктивної оцінки та визначення пріоритетності цих критеріїв було застосовано метод експертних оцінок, а саме метод аналізу ієрархій Томаса Сааті [26]. Цей метод дозволяє структурувати проблему вибору, визначити відносну важливість кожного критерію шляхом попарних порівнянь та розрахувати відповідні вагові коефіцієнти. У ході експертного оцінювання було встановлено, що найвищий пріоритет для користувачів мають точність прогнозу (А1), зручний та привабливий інтерфейс (А3), а також відсутність нав’язливої реклами (А5). Вагові коефіцієнти для критеріїв, отримані в результаті розрахунків, склали: А1 (Точність) – 0,48; А2 (Деталізація) – 0,14; А3 (UI/UX) – 0,28; А4 (Додаткові функції) – 0,04; А5 (Без реклами) – 0,07.

На основі визначених критеріїв та їх вагових коефіцієнтів було проведено оцінку кожного з обраних застосунків-аналогів. Кожен застосунок отримав оцінку за кожним критерієм за 7-бальною шкалою (де 1 – найгірше значення, 7 – найкраще). Результати цих оцінок представлено у таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика застосунків-аналогів

Назва застосунку	**А1 (Точність)**	**А2 (Деталізація)**	**А3 (UI/UX)**	**А4 (Дод. функції)**	**А5 (Без реклами)**
AccuWeather	**6**	**7**	**4**	**6**	**2**
Weather Channel	**6**	**7**	**4**	**6**	**2**
Yahoo Weather	**5**	**4**	**7**	**4**	**5**
Google Weather	**6**	**3**	**6**	**2**	**7**

Після отримання експертних оцінок для кожного застосунку-аналога за всіма критеріями, було розраховано інтегральні показники їх якості (Qj). Розрахунок здійснювався за адитивною формулою згортки, яка враховує вагові коефіцієнти кожного критерію [27]:

$$Q_J = \sum(w_i * P_{ij}),$$

де w_i – ваговий коефіцієнт і-го критерію важливості;

P_{ij} – оцінка j-го програмного продукту за і-м критерієм.

Результати розрахунку інтегральних показників якості для проаналізованих мобільних застосунків прогнозу погоди, а також очікуваний показник для розроблюваного застосунку “Pixel Weather”, наведені в таблиці 1.2. Розрахунки проводилися на основі оцінок, представлених у таблиці 1.1, та вагових коефіцієнтів, визначених раніше.

Таблиця 1.2 – Інтегральні показники якості застосунків-аналогів та розроблюваного продукту

№ п/п	**Назва програмного продукту**	**Інтегральний показник якості (Qj)**
1	AccuWeather	**5,2947**
2	Weather Channel	**5,2947**
3	Yahoo Weather	**5,3831**
4	Google Weather	**5,5168**
5	Pixel Weather	**5,6533**

Аналіз розрахованих інтегральних показників якості (див. табл. 1.2) свідчить про високий рівень конкуренції на ринку мобільних погодних застосунків. Найвищий показник якості серед розглянутих аналогів має Google Weather (Qj = 5,5168), що зумовлено збалансованими оцінками за більшістю критеріїв, зокрема високою оцінкою за відсутність реклами та хороший інтерфейс. Застосунок Yahoo Weather (Qj = 5,3831) також демонструє конкурентний результат, маючи перевагу в UI/UX. Програмні продукти AccuWeather та Weather Channel отримали однакові інтегральні оцінки (Qj = 5,2947); їх сильні сторони полягають у високій точності та деталізації прогнозу, проте вони поступаються іншим аналогам через менш зручний інтерфейс та наявність реклами.

Незважаючи на високі показники існуючих лідерів, розроблюваний застосунок “Pixel Weather” має потенціал для успішного позиціонування. Це обґрунтовується очікуваним інтегральним показником якості (Qj = 5,6533), який є найвищим серед усіх розглянутих варіантів. Такий результат прогнозується завдяки акценту на ключових для користувача критеріях: високій точності прогнозу (оцінка 6), що буде підсилена інтеграцією локальної моделі машинного навчання для уточнення показника “FLT”; унікальному та зручному інтерфейсі в стилі піксель-арт (оцінка 6); а також повній відсутності рекламних оголошень (оцінка 7). Хоча за критеріями деталізації (оцінка 4) та наявності додаткових функцій (оцінка 2) розроблюваний застосунок може поступатися деяким аналогам, пріоритет надано саме тим аспектам, які, згідно з експертною оцінкою, мають найбільшу вагу для кінцевого користувача. Таким чином, прогнозований показник якості свідчить про доцільність розробки та потенційну конкурентоспроможність застосунку “Pixel Weather” за умови успішної реалізації всіх запланованих функціональних та нефункціональних характеристик.

1.3 Обґрунтування машинного навчання в уточненні прогнозу

Сучасні мобільні застосунки для прогнозування погоди переважно покладаються на дані, що надаються великими метеорологічними службами через програмні інтерфейси (API). Ці дані, хоча й охоплюють значні території та базуються на складних глобальних моделях прогнозування, не завжди можуть з абсолютною точністю відображати реальні погодні умови для кожної конкретної, часто невеликої, географічної локації. Глобальні моделі оперують на досить великій просторовій сітці, що може призводити до усереднення показників та ігнорування локальних мікрокліматичних особливостей, зумовлених рельєфом, близькістю водойм, міською забудовою тощо. Як наслідок, користувач може спостерігати розбіжності між прогнозованими та фактичними погодними умовами.

Особливо це стосується такого комплексного показника, як “FLT”. Цей параметр є суб’єктивним для людини і залежить не лише від температури повітря, але й від швидкості вітру, вологості, а іноді й сонячної радіації. Формули, що використовуються для розрахунку цього показника в стандартних API, можуть бути узагальненими і не завжди ідеально відповідати індивідуальним відчуттям користувача в конкретному місці [10].

Саме для вирішення подібних задач та підвищення точності певних аспектів прогнозу доцільним видається застосування методів машинного навчання. Замість того, щоб повністю замінювати дані з API, локальна модель машинного навчання, інтегрована безпосередньо в мобільний застосунок, може виступати як інструмент для їх уточнення або надання альтернативного, потенційно більш точного, прогнозу для окремих параметрів [7, 13, 14].

У рамках даної кваліфікаційної роботи було прийнято рішення розробити та інтегрувати модель машинного навчання для прогнозування саме “FLT”. Такий вибір зумовлений кількома причинами:

– важливість для користувача: ”FLT” часто є більш значущим показником для планування активностей та вибору одягу, ніж просто температура повітря;
– можливість уточнення: як зазначалось, цей параметр залежить від багатьох факторів, і модель, натренована на історичних даних, що включають ці фактори, потенційно може краще вловлювати їх взаємозв’язок для конкретних умов, ніж узагальнені формули;
– доступність даних для тренування: існують відкриті датасети з історичними метеорологічними даними, які містять як фактичну температуру, так і “FLT”, а також інші необхідні ознаки (вологість, швидкість вітру, тиск тощо), що дозволяє натренувати відповідну регресійну модель;
– реалістичність інтеграції: сучасні технології, зокрема TensorFlow Lite, дозволяють ефективно запускати попередньо натреновані моделі безпосередньо на мобільних пристроях без значного впливу на продуктивність чи споживання ресурсів.

Таким чином, використання локальної моделі машинного навчання для прогнозування “FLT” в мобільному застосунку “Pixel Weather” розглядається як спосіб надати користувачеві додаткову, потенційно більш точну та персоналізовану інформацію, що підвищить загальну цінність та корисність розроблюваного програмного продукту. Це дозволить не просто дублювати дані з API, а запропонувати унікальну функцію, спрямовану на покращення користувацького досвіду.

1.4 Висновки до розділу 1

У даному розділі було проведено комплексний аналіз предметної області розробки мобільних застосунків для прогнозування погоди. Розглянуто актуальність та важливість таких програмних продуктів для сучасних користувачів, а також визначено ключові метеорологічні параметри та джерела даних, що використовуються у погодних інформерах.

Здійснено порівняльний аналіз чотирьох популярних застосунків-аналогів (AccuWeather, The Weather Channel, Yahoo Weather та Google Weather) за допомогою методу експертних оцінок Томаса Сааті. На основі розрахованих вагових коефіцієнтів критеріїв якості (точність, деталізація, UI/UX, додаткові функції, відсутність реклами) та інтегральних показників було виявлено сильні та слабкі сторони існуючих рішень. Аналіз показав, що розроблюваний застосунок “Pixel Weather” має потенціал конкурентоспроможності завдяки акценту на точності прогнозу за допомогою машинного навчання, унікальному піксель-арт інтерфейсі та відсутності реклами, що підтверджується найвищим прогнозованим інтегральним показником якості.

Також було обґрунтовано доцільність використання методів машинного навчання для уточнення прогнозу “FLT”. Визначено, що інтеграція локальної ML-моделі може підвищити точність та персоналізацію цього важливого для

користувача параметра, враховуючи доступність даних для тренування та можливість ефективної реалізації на мобільних пристроях за допомогою TensorFlow Lite. Результати аналізу, проведеного в цьому розділі, слугуватимуть основою для подальшого проєктування та розробки програмного застосунку.

2 РОЗРОБКА АРХІТЕКТУРИ, БАЗИ ДАНИХ ТА ML-ЕЛЕМЕНТА ПРОГРАМИ

Проєктування програмного забезпечення є ключовим етапом життєвого циклу розробки, що визначає структуру, функціональність та характеристики майбутнього продукту. Якісне проєктування дозволяє мінімізувати ризики на етапах реалізації та тестування, забезпечити відповідність системи поставленим завданням та очікуванням користувачів, а також закласти основи для подальшого розвитку та супроводу програмного забезпечення.

2.1 Формулювання вимог до системи

На основі аналізу предметної області, розглянутих аналогів та поставленої мети кваліфікаційної роботи було сформульовано комплекс вимог до мобільного застосунку “Pixel Weather”. Ці вимоги поділяються на функціональні, що описують основні можливості та дії, які користувач може виконувати за допомогою застосунку, та нефункціональні, що визначають атрибути якості системи, такі як продуктивність, надійність, зручність використання та інші.

2.1.1 Функціональні вимоги

До основних функціональних можливостей, які має надавати мобільний застосунок “Pixel Weather”, належать:

а) відображення поточної погоди:
   1) температура повітря (фактична);
   2) “FLT” (значення з API та значення, розраховане ML-моделлю);
   3) загальний стан погоди (наприклад, «ясно», «хмарно», «дощ»);

4) іконка, що відповідає поточному стану погоди;
5) швидкість та напрямок вітру;
6) атмосферний тиск;
7) вологість повітря;
8) видимість;
9) час сходу та заходу сонця;

б) відображення погодинного прогнозу:
1) прогноз щонайменше на найближчі 24 години з інтервалом в 1 або 3 години;
2) для кожної години: температура, іконка стану погоди, ймовірність опадів (POP);

в) відображення денного прогнозу:
1) прогноз щонайменше на 5-7 днів вперед;
2) для кожного дня: дата, іконка стану погоди, максимальна та мінімальна температура, ймовірність опадів (POP), текстовий опис стану погоди;

г) отримання даних з відкритого погодного API (OpenWeatherMap) для забезпечення актуальною метеорологічною інформацією;

д) автоматичне визначення геолокації користувача (за його згодою) для відображення погоди у поточному місці перебування;

е) можливість ручного пошуку локації за назвою міста з використанням API геокодингу (Geoapify) та відображенням списку підказок;

ж) збереження обраних користувачем локацій у локальній базі даних (Room);

з) можливість перегляду погоди для збережених локацій за допомогою горизонтального пейджера на головному екрані;

и) управління списком збережених локацій: додавання нових локацій, видалення існуючих, зміна їх порядку;

к) інтеграція локальної моделі машинного навчання (TensorFlow Lite) для уточнення/розрахунку показника “FLT” на основі даних, отриманих з API;

л) візуалізація результату роботи ML-моделі для користувача (відображення “ML Feels Like” поруч з “API Feels Like”);

м) налаштування одиниць виміру температури (Цельсій/Фаренгейт);

н) налаштування теми оформлення застосунку (системна/світла/темна);

о) наявність екрану «Про застосунок» з інформацією про розробника та використані ресурси;

п) обробка відсутності інтернет-з’єднання та відображення відповідних повідомлень.

2.1.2 Нефункціональні вимоги

До нефункціональних вимог, що визначають якісні характеристики застосунку, належать:

а) продуктивність:

1) час завантаження початкових даних погоди при запуску застосунку не повинен перевищувати 4 секунд (за умови стабільного інтернет-з’єднання);
2) навігація між основними екранами та сторінками пейджера має відбуватися з мінімальною затримкою (менше 0,5 секунди);
3) оновлення даних за допомогою “pull-to-refresh” має бути швидким та відгукуватися миттєво;

б) надійність:

1) застосунок повинен стабільно працювати без збоїв та непередбачуваних помилок;
2) має бути реалізована коректна обробка помилок мережі (відсутність з’єднання, тайм-аути сервера, помилки API) з відображенням зрозумілих повідомлень користувачеві;
3) забезпечення стабільної роботи ML-моделі на пристрої;

в) зручність використання (Usability):

1) інтерфейс користувача має бути інтуїтивно зрозумілим та легким у навігації;
2) усі елементи інтерфейсу мають бути виконані в єдиному піксель-арт стилі, забезпечуючи унікальний візуальний досвід;
3) застосунок має відповідати принципам Material Design, адаптованим під піксель-арт, для забезпечення звичних патернів взаємодії;

г) точність:
1) застосунок має використовувати надійне та перевірене погодне API для отримання базового прогнозу;
2) інтегрована ML-модель має сприяти підвищенню точності або наданню альтернативного значення для показника “FLT”;

д) сумісність: застосунок має підтримуватися на пристроях під управлінням операційної системи Android версії 8.0 (API level 26) та вище;

е) безпека:
1) API ключі для доступу до зовнішніх сервісів (OpenWeatherMap, Geoapify) мають зберігатися безпечним способом, не допускаючи їх потрапляння у відкритий вихідний код;
2) усі мережеві запити до API мають здійснюватися через захищене з’єднання HTTPS;

ж) готовність до підтримки:
1) код застосунку має бути добре структурованим, читабельним та документованим;
2) має бути наявність коментарів до ключових блоків коду;
3) дотримання обраної архітектури (MVVM) та принципів чистого коду;
4) розширюваність: архітектура застосунку має дозволяти відносно легко додавати нові джерела даних, нові функції або оновлювати ML-модель у майбутньому.

2.2 Обґрунтування вибору архітектури та технологій

Вибір правильної архітектури та відповідного технологічного стеку є фундаментальним для успішної розробки будь-якого програмного продукту, особливо мобільного застосунку, де важливі як функціональність, так і продуктивність та зручність використання. Для мобільного застосунку “Pixel Weather” було прийнято рішення використовувати сучасні, перевірені часом підходи та технології, що забезпечують гнучкість, масштабованість та високу якість кінцевого продукту.

Основним архітектурним шаблоном, обраним для побудови користувацького інтерфейсу та управління станом, є MVVM (Model-View-ViewModel) [28]. Цей шаблон добре зарекомендував себе в Android-розробці завдяки чіткому розділенню відповідальностей.

Першим компонентом є Model (модель). Вона представляє дані та бізнес-логіку застосунку. До її складу входять джерела даних, такі як API та локальна база даних, репозиторії, що абстрагують доступ до цих джерел, а також різні моделі даних, зокрема DTO (Data Transfer Objects), Domain Models та Entities.

Другий компонент, View (представлення), відповідає за відображення даних користувачеві та передачу його дій до ViewModel. У контексті розробки на Jetpack Compose, роль Представлення виконують Composable-функції, які декларативно описують користувацький інтерфейс. View реагує на зміни даних у ViewModel, підписуючись на них, і відповідно оновлює відображення.

Третім важливим елементом є ViewModel (модель Представлення). Вона виступає як посередник між Model та View, інкапсулюючи логіку представлення. ViewModel обробляє дії користувача, отримує дані з Model, готує їх для відображення та надає View. Важливою характеристикою ViewModel є відсутність прямих посилань на View, що значно полегшує процес тестування та підвищує її незалежність від конкретної реалізації UI. Окрім того, ViewModel відповідає за збереження стану інтерфейсу та здатна переживати зміни конфігурації пристрою, наприклад, поворот екрана.

Для доступу до даних та управління їх джерелами було обрано шаблон Repository Pattern [28]. Репозиторій виступає як єдина точка доступу до даних для ViewModel, абстрагує джерела даних (мережеві запити, локальна база даних). Це дозволяє легко змінювати або додавати джерела даних без значних змін у бізнес-логіці чи ViewModel, а також спрощує тестування, оскільки можна замінювати реальний репозиторій на тестовий варіант, що імітує його поведінку.

Для управління залежностями та забезпечення слабкої зв’язності компонентів системи було використано принцип Dependency Injection (DI), реалізований за допомогою бібліотеки Hilt [3]. Hilt, розроблений Google, значно спрощує впровадження DI в Android-застосунках, автоматизуючи генерацію коду та інтегруючись з компонентами життєвого циклу Android. Використання Hilt сприяє підвищенню модульності, готовності коду до тестів та підтримки.

Технологічний стек для розробки застосунку “Pixel Weather” було обрано з урахуванням сучасних тенденцій та рекомендацій для Android-розробки:

Основною мовою програмування для розробки застосунку є Kotlin – офіційно підтримувана Google мова для Android. Вона вирізняється лаконічним і зручним для розробника синтаксисом, високим рівнем безпеки типів та чудовою інтеграцією з бібліотеками Java. Крім того, Kotlin підтримує сучасний підхід до асинхронного програмування завдяки так званим корутинам – механізму, який дозволяє легко виконувати фонові операції (наприклад, завантаження даних з інтернету) без блокування основного інтерфейсу користувача, забезпечуючи плавну та стабільну роботу застосунку. Для розробки інтерфейсу користувача використовується Jetpack Compose – сучасний декларативний фреймворк, офіційно рекомендований Google для створення нативних інтерфейсів Android [2]. Його впровадження дозволяє значно спростити процес розробки UI, пришвидшує внесення змін до коду, а також покращує його підтримуваність. Compose надає гнучкі інструменти для створення унікального візуального оформлення, що особливо важливо для

реалізації стилістики в піксель-арті, де кожен елемент інтерфейсу має відповідати загальному художньому задуму застосунку.

Асинхронне програмування реалізується за допомогою Kotlin Coroutines та Flow. Це забезпечує ефективне виконання фонових операцій, таких як мережеві запити чи доступ до бази даних, без блокування основного потоку. Окрім цього, використання Flow дозволяє організувати зручну реактивну передачу даних між різними шарами застосунку.

Для мережевої взаємодії використовується бібліотека Retrofit – типізований HTTP-клієнт для Android і Java, що значно спрощує роботу з REST API [5]. У ролі низькорівневого клієнта, який безпосередньо обробляє HTTP-запити, використовується OkHttp, що інтегрується з Retrofit для забезпечення надійного та ефективного обміну даними.

Kotlinx.Serialization – це рідна бібліотека для мови Kotlin, яка призначена для роботи з форматом JSON. Вона забезпечує безпечне і ефективне перетворення даних між об’єктами програми та їх JSON-представленням.

Room Persistence Library [4] – це рівень абстракції над SQLite, який значно полегшує роботу з локальною базою даних, дозволяючи зберігати обрані користувачем локації та налаштування.

TensorFlow Lite [7] – це бібліотека, яка дає змогу запускати моделі машинного навчання на мобільних та вбудованих пристроях. Вона дозволяє безпосередньо в додатку використовувати заздалегідь навчені моделі для підвищення точності прогнозів.

Зовнішні API:

– OpenWeatherMap API [10]: як основне джерело метеорологічних даних (поточна погода, погодинний та денний прогнози);
– Geoapify API [11, 12]: для реалізації функціоналу автозаповнення при пошуку міст та для реверс-геокодингу (перетворення координат на назву місцевості).

Обраний набір архітектурних рішень та технологій забезпечує створення сучасного, ефективного та підтримуваного мобільного застосунку, що відповідає поставленим функціональним та нефункціональним вимогам.

2.3 Розробка моделей та діаграм системи

Для наочного представлення архітектури, структури та взаємодії компонентів мобільного застосунку “Pixel Weather” було розроблено набір діаграм з використанням уніфікованої мови моделювання UML (Unified Modeling Language). Нижче наведено та описано ключові діаграми, що відображають різні аспекти спроєктованої системи.

2.3.1 Функціональна схема та структурна схема модулів

Загальне уявлення про функціональні можливості застосунку та його основні компоненти надає функціональна схема. Вона ілюструє ключові функції системи та потоки даних між ними (див. рис. 2.1).

Як видно з рисунка, основними функціональними блоками є отримання даних про погоду, їх обробка (включаючи уточнення за допомогою ML-моделі), відображення користувачеві, управління локаціями та налаштуваннями.

Для деталізації архітектури на вищому рівні було розроблено структурну схему модулів, яка відображає розподіл системи на основні логічні шари відповідно до обраного архітектурного підходу (MVVM та принципи Clean Architecture) (рис. 2.2). Ця схема показує три основні шари: шар представлення (UI Layer), шар домену або бізнес-логіки (Domain/Use Case Layer) та шар даних (Data Layer).

Шар представлення (UI Layer) відповідає за відображення інформації користувачеві та обробку його взаємодії. Він включає елементи інтерфейсу (Composable-функції), ViewModel для кожного екрану, яка керує станом UI та взаємодіє з шаром домену.

Шар домену (Domain/Use Case Layer) містить бізнес-логіку застосунку, незалежну від конкретної реалізації UI або джерел даних. Тут знаходяться моделі даних предметної області (Domain Models) та Use Cases

(інтерактори), що інкапсулюють окремі бізнес-операції.

Шар даних (Data Layer) відповідає за отримання та зберігання даних. Він включає реалізації репозиторіїв, які абстрагують джерела даних (мережеві API, локальна база даних Room), DTO (Data Transfer Objects) для роботи з API та маппери для перетворення даних між шарами.

Такий поділ на шари забезпечує слабку зв’язність компонентів, підвищує тестуємість та полегшує подальшу підтримку та модифікацію застосунку.

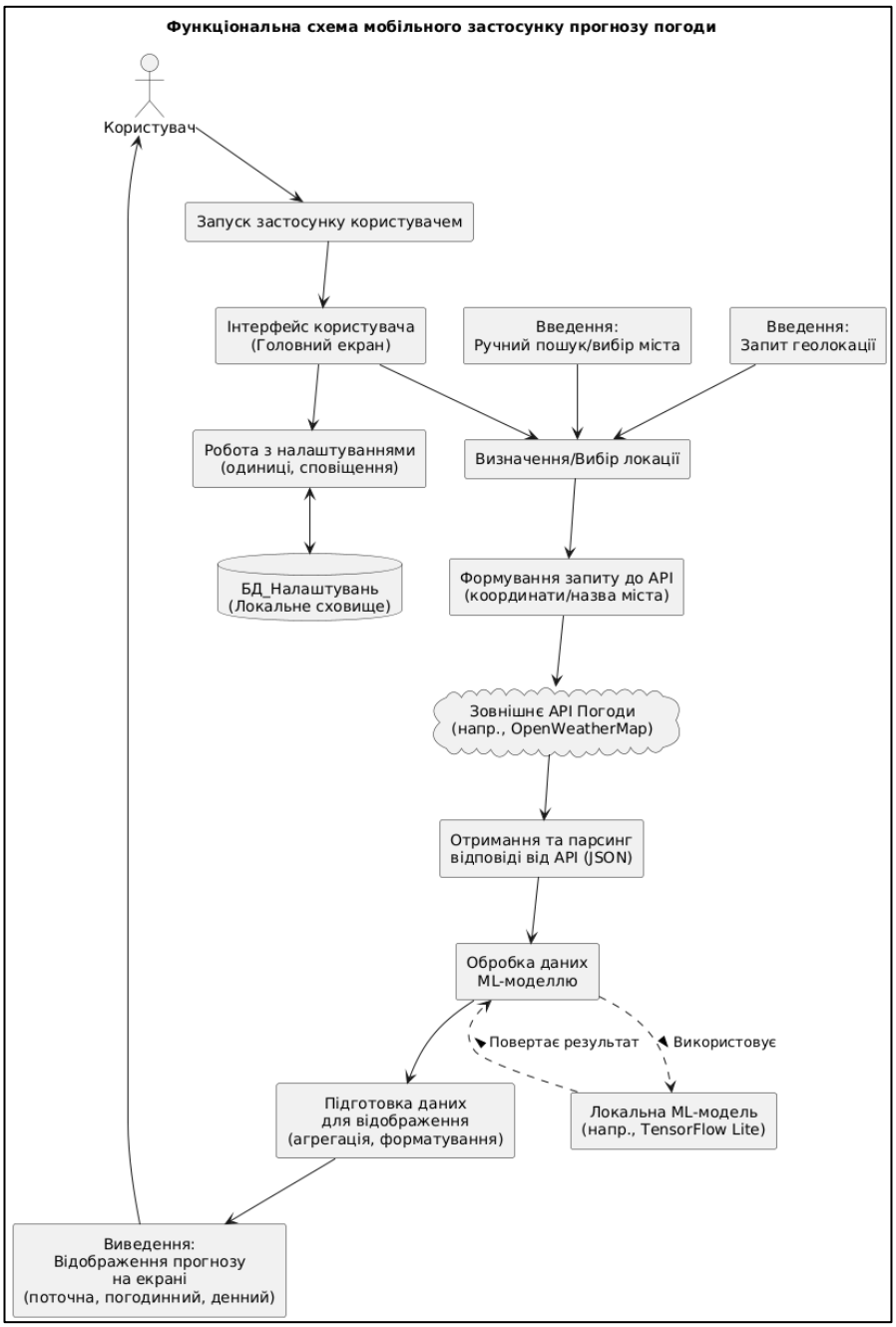


Рисунок 2.1 – Функціональна схема мобільного застосунку “Pixel Weather”

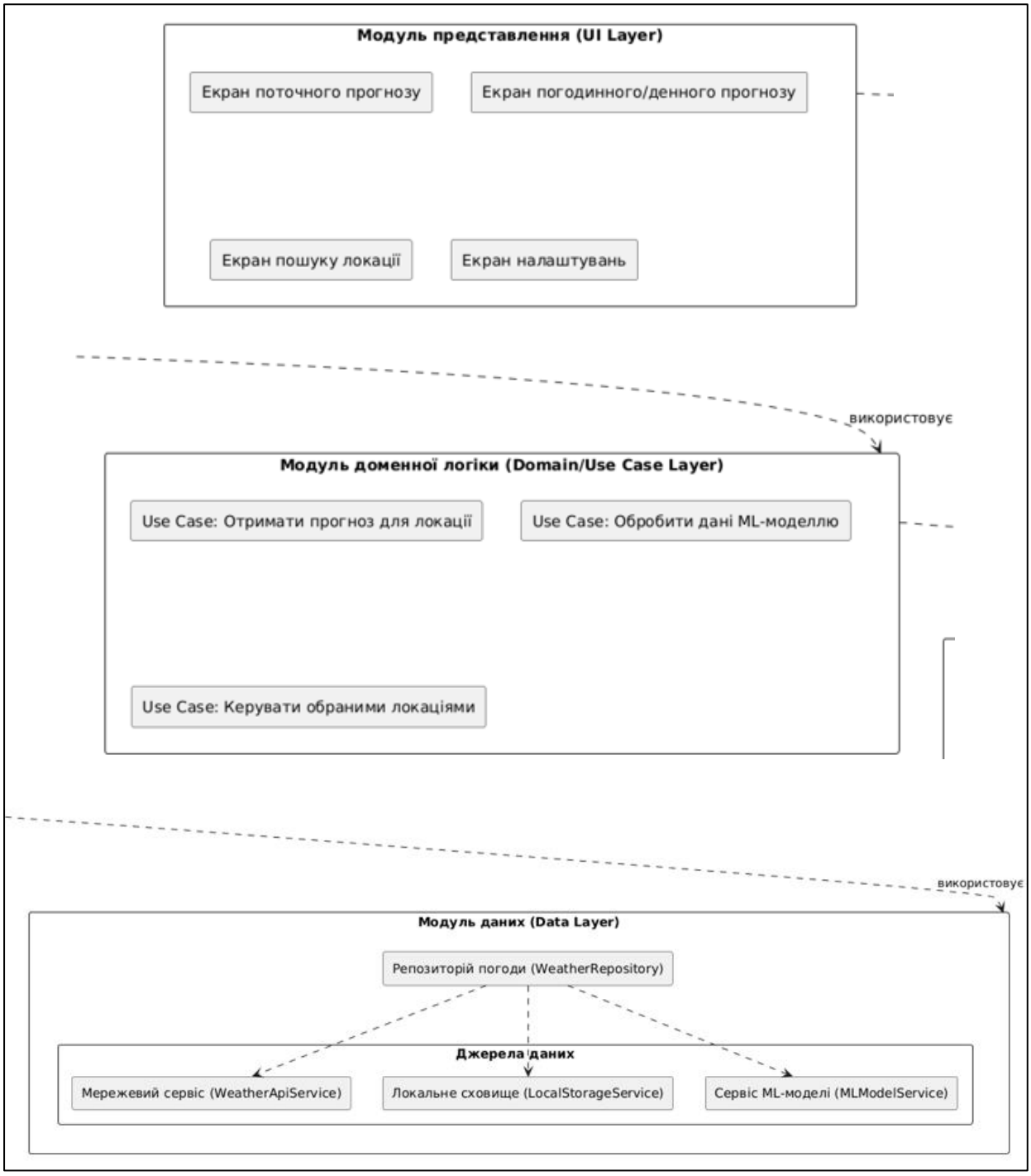


Рисунок 2.2 – Структурна схема модулів застосунку “Pixel Weather”

2.3.2 Діаграма компонентів

Діаграма компонентів (рис. 2.3) деталізує основні програмні компоненти системи та залежності між ними. Вона показує, як різні частини застосунку взаємодіють на більш високому рівні абстракції, ніж діаграма класів.

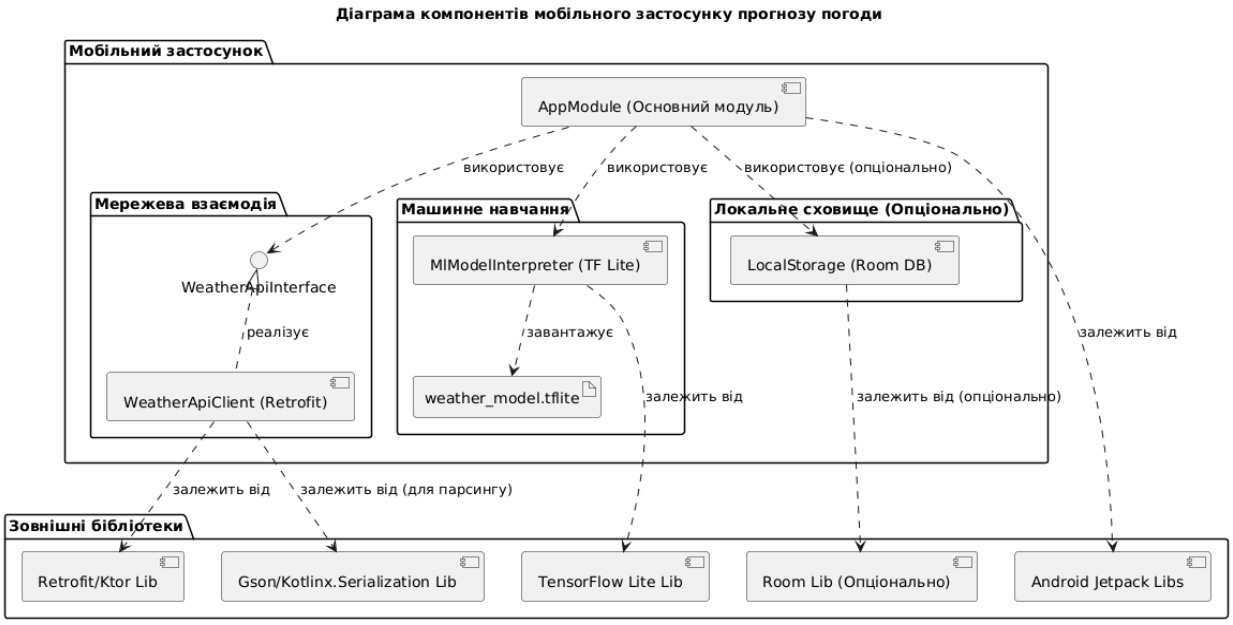


Рисунок 2.3 – Діаграма компонентів застосунку “Pixel Weather”

На рисунку 2.3 представлена діаграма компонентів мобільного застосунку “Pixel Weather”, що ілюструє основні програмні блоки та взаємозв’язки між ними. Центральним елементом є «Мобільний застосунок», який інкапсулює ключову функціональність.

Усередині застосунку виділено кілька основних компонентів. Основний модуль – AppModule – відповідає за надання залежностей за допомогою Hilt, налаштовуючи екземпляри сервісів для мережевої взаємодії, машинного навчання та роботи з локальним сховищем. Мережева взаємодія реалізується через інтерфейс WeatherApiInterface та його клієнт WeatherApiClient, який використовує бібліотеку Retrofit для виконання HTTP-запитів до зовнішніх погодних API. Для роботи з моделлю машинного навчання в застосунку застосовується компонент MlModelInterpreter, що базується на TensorFlow Lite і відповідає за завантаження та виконання локальної моделі (weather_model.tflite) для уточнення прогнозу. Опціонально використовується локальне сховище LocalStorage, яке реалізує взаємодію з базою даних Room для збереження даних, таких як обрані локації.

Взаємодія між цими компонентами базується на використанні зовнішніх бібліотек. Зокрема, WeatherApiClient залежить від бібліотек Retrofit та Ktor для мережевих операцій, а також Gson або Kotlinx.Serialization для обробки JSON-

даних. MlModelInterpreter покладається на TensorFlow Lite для роботи з моделлю машинного навчання, тоді як LocalStorage використовує бібліотеку Room для операцій з базою даних. Крім того, усі компоненти застосунку застосовують бібліотеки Android Jetpack для реалізації базової функціональності Android-застосунку, користувацького інтерфейсу та управління життєвим циклом.

Стрілки на діаграмі відображають залежності типу «використовує», «реалізує», «завантажує» та «залежить від», що показує загальну архітектуру взаємодії програмних блоків.

2.3.3 Діаграма класів

Для детального представлення статичної структури системи, її ключових класів, їх атрибутів, методів та взаємозв'язків було розроблено діаграму класів. Зважаючи на загальний обсяг проєкту, на рисунках 2.4 та 2.5 представлено фрагменти цієї діаграми, що ілюструють відповідно структуру шару домену та репозиторію, а також структуру об'єктів передачі даних (DTO), які використовуються для обробки відповідей від зовнішнього API.

На рисунку 2.4 зображено взаємодію між репозиторієм погоди, сервісом API та моделями даних предметної області.

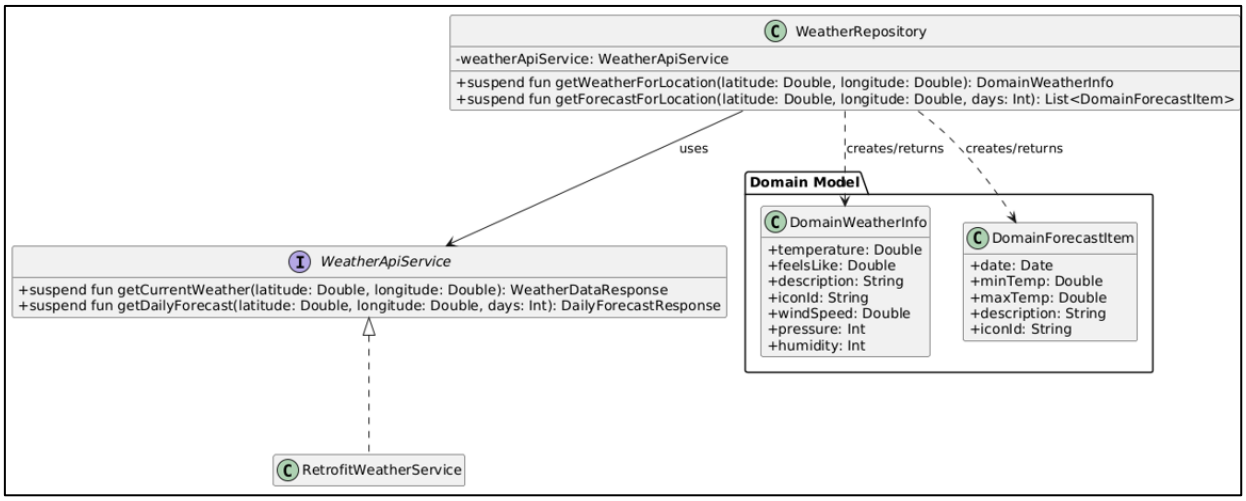


Рисунок 2.4 – Діаграма класів: шар домену та репозиторій

Клас WeatherRepository відповідає за надання даних про погоду іншим частинам застосунку. Для отримання цих даних він використовує інтерфейс WeatherApiService, конкретною реалізацією якого є RetrofitWeatherService, що інкапсулює логіку мережевих запитів. У результаті своєї роботи WeatherRepository оперує об’єктами з Domain Model, такими як DomainWeatherInfo (для поточної погоди) та DomainForecastItem (для елементів прогнозу).

Структура об’єктів передачі даних (DTO), що використовуються для десеріалізації відповіді від погодного API, представлена на рисунку 2.5. Вона включає основні класи, такі як WeatherDataResponse для поточної погоди та DailyForecastResponse для прогнозу, а також вкладені DTO (DailyForecastItemDto, WindDto, MainInfoDto, WeatherDescriptionDto, TempDto), що відображають ієрархію отриманих JSON-даних.

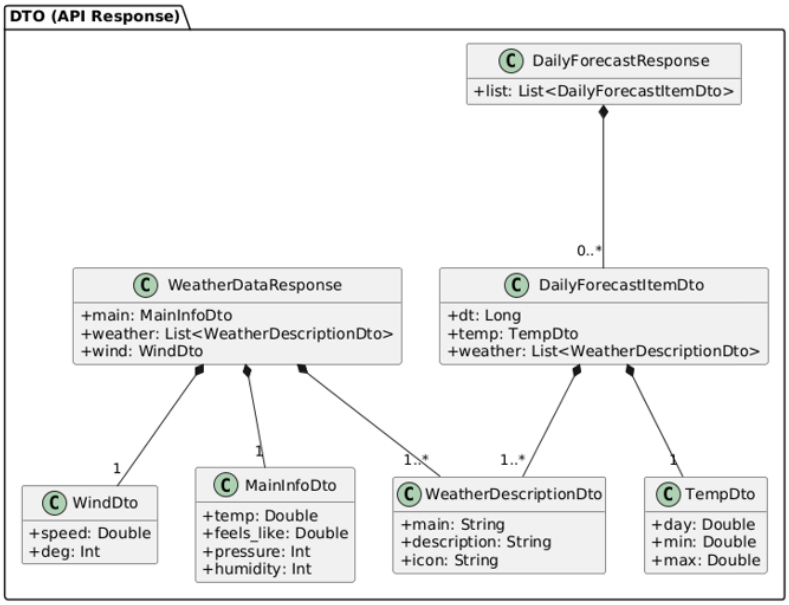


Рисунок 2.5 – Діаграма класів: структура DTO для відповіді API

Таке розділення на DTO та Domain Model дозволяє забезпечити незалежність бізнес-логіки застосунку від конкретного формату даних, що надається зовнішнім API, та підвищує гнучкість системи при можливих змінах у структурі відповіді API. Зв’язки між класами на діаграмах показують агрегацію, композицію, реалізацію інтерфейсу та залежності використання.

2.3.4 Діаграма послідовностей

Для ілюстрації динамічної взаємодії об’єктів у часі для виконання ключового сценарію – отримання та відображення даних про погоду – було розроблено діаграму послідовностей (рис. 2.6). Ця діаграма деталізує кроки обміну повідомленнями між основними компонентами системи.

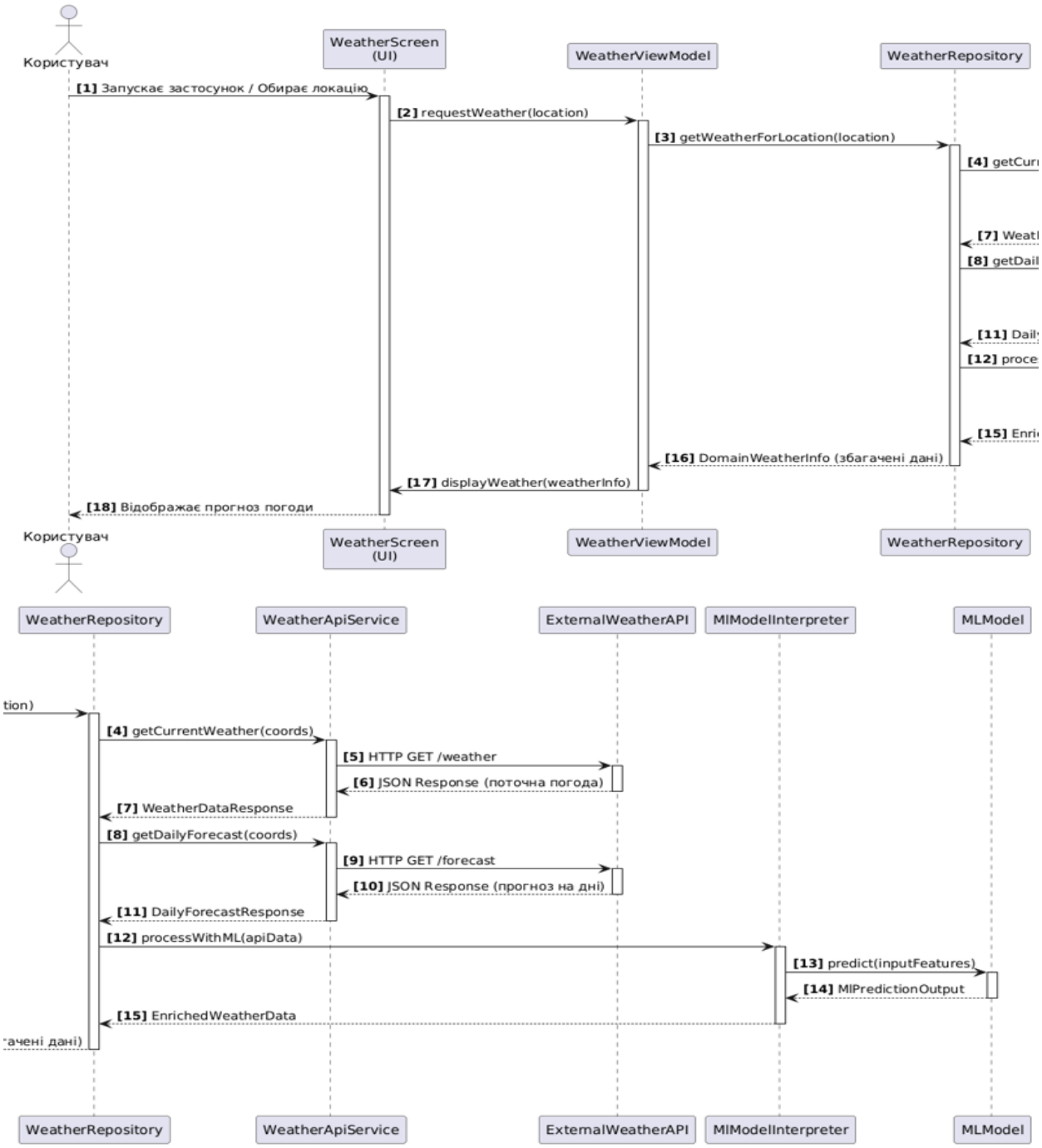


Рисунок 2.6 – Діаграма послідовностей для сценарію «Отримання та відображення даних про погоду»

Процес взаємодії, зображений на рисунку 2.6, починається з ініціації дії користувачем:

- користувач запускає застосунок або обирає локацію в інтерфейсі (WeatherScreen (UI));
- WeatherScreen (UI) надсилає запит на отримання погоди (requestWeather(location)) до WeatherViewModel (у тебе в коді MainViewModel);
- WeatherViewModel делегує цей запит до WeatherRepository, викликаючи метод getWeatherForLocation(location);
- WeatherRepository ініціює запит на отримання поточної погоди (getCurrentWeather(coords)) до WeatherApiService;
- WeatherApiService виконує HTTP GET запит до зовнішнього погодного API (ендпоінт /weather);
- зовнішнє API повертає JSON відповідь з даними поточної погоди;
- WeatherApiService передає ці дані (WeatherDataResponse) назад до WeatherRepository;
- паралельно або послідовно WeatherRepository запитує прогноз на кілька днів (getDailyForecast(coords)) у WeatherApiService;
- WeatherApiService виконує HTTP GET запит до зовнішнього API (ендпоінт /forecast);
- зовнішнє API повертає JSON відповідь з даними прогнозу;
- WeatherApiService передає ці дані (DailyForecastResponse) до WeatherRepository;
- WeatherRepository обробляє отримані дані від API та передає їх (apiData) до MlModelInterpreter для отримання передбачення від моделі машинного навчання (викликаючи, наприклад, processWithML(apiData));
- MlModelInterpreter викликає метод predict(inputFeatures) у локальної MLModel (TensorFlow Lite);
- MLModel повертає результат передбачення (MlPredictionOutput) до MlModelInterpreter;

- MlModelInterpreter повертає оброблені дані, збагачені передбаченням ML, (EnrichedWeatherData) назад до WeatherRepository;
- WeatherRepository формує фінальний об’єкт DomainWeatherInfo (або WeatherDataBundle у твоїй реалізації), що містить усі необхідні дані, включаючи результат ML, та повертає його до WeatherViewModel;
- WeatherViewModel отримує ці збагачені дані та оновлює свій стан, що призводить до виклику displayWeather(weatherInfo) у WeatherScreen (UI);
- WeatherScreen (UI) відображає отриманий прогноз погоди користувачеві.

Ця послідовність демонструє чіткий розподіл відповідальностей між компонентами та потік даних від користувацького інтерфейсу через ViewModel та Repository до джерел даних (API та ML-модель) і назад до UI.

2.4 Проєктування компонента машинного навчання

Однією з ключових особливостей мобільного застосунку “Pixel Weather” є використання методів машинного навчання для уточнення прогнозу погоди, зокрема для розрахунку показника “FLT”. Цей підрозділ описує основні етапи проєктування та підготовки компонента машинного навчання для інтеграції в застосунок.

2.4.1 Постановка задачі машинного навчання та вибір цільової змінної

Основною задачею, яку вирішує компонент машинного навчання, є прогнозування “FLT” користувачем, на основі набору поточних метеорологічних параметрів. Стандартні формули розрахунку цього

показника, що використовуються в API, можуть бути узагальненими [29]. Метою інтеграції ML-моделі є надання альтернативного, потенційно більш точного або адаптованого значення цього параметра.

В якості цільової змінної для моделі було обрано Apparent Temperature (C) (відчувана температура в градусах Цельсія) з використовуваного історичного набору даних. Це задача регресії, оскільки необхідно передбачити неперервне числове значення. Модель має на меті навчитися на історичних даних, де відомі як фактична температура та інші погодні умови, так і відповідна їм відчувана температура, для різних географічних локацій.

2.4.2 Вибір та підготовка даних для тренування моделі

Для тренування моделі машинного навчання було використано відкритий набір даних “Historical Weather Data for different cities around the world”, отриманий з платформи Kaggle. Цей датасет містить погодинні записи метеорологічних спостережень для різноманітних міст світу за тривалий період, включаючи такі параметри як: Formatted Date (дата та час), Summary (загальний опис погоди), Precip Type (тип опадів), Temperature (C) (температура повітря), Apparent Temperature (C) (відчувана температура), Humidity (вологість), Wind Speed (km/h) (швидкість вітру), Wind Bearing (degrees) (напрямок вітру), Visibility (km) (видимість), Loud Cover (малоінформативна ознака, що зазвичай видаляється) та Pressure (millibars) (атмосферний тиск).

Попередня обробка даних включала наступні ключові кроки:

а) очищення даних: видалення малоінформативних стовпців (Loud Cover, Daily Summary) (обробка пропущених значень, зокрема для стовпця Precip Type, де пропуски були замінені на категорію “None” для подальшого кодування);

б) робота з датою та часом: стовпець Formatted Date було перетворено у формат datetime для виділення часових ознак;

в) генерація ознак:
   1) створення циклічних часових ознак (синус та косинус для години доби, місяця та дня року) для адекватного представлення періодичності погодних явищ;
   2) перетворення напрямку вітру (Wind Bearing (degrees)) на дві циклічні ознаки (синус та косинус кута) для уникнення проблем з розривом шкали (0 та 360 градусів);
   3) кодування категоріальної ознаки Precip Type (тип опадів: “rain”, “snow”, “None”) методом One-Hot Encoding з видаленням першої категорії (drop='first') для запобігання мультиколінеарності;

г) конвертація одиниць виміру: швидкість вітру, що у датасеті надана в км/год, була залишена в цих одиницях для тренування, оскільки дані з OpenWeatherMap API (м/с) будуть відповідно конвертуватися в км/год перед подачею на вхід моделі в мобільному застосунку для забезпечення узгодженості.

2.4.3 Вибір ознак для моделі

На основі аналізу доступних даних у датасеті “Historical Weather Data for different cities around the world” та даних, які можна отримати з OpenWeatherMap API для використання в мобільному застосунку, було відібрано наступний набір з 15 вхідних ознак для моделі (згідно з файлом feels_like_final_feature_names.txt):

– num__Temperature (C): фактична температура повітря;
– num__Humidity: відносна вологість повітря (значення від 0 до 1, отримане діленням відсотків на 100 перед масштабуванням);
– num__Wind Speed (km/h): швидкість вітру;
– num__Visibility (km): видимість;
– num__Pressure (millibars): атмосферний тиск;

– cat__Precip Type_rain: бінарна ознака, що вказує на наявність дощу;
– cat__Precip Type_snow: бінарна ознака, що вказує на наявність снігу;
– remainder__HourSin: синус години доби;
– remainder__HourCos: косинус години доби;
– remainder__MonthSin: синус місяця;
– remainder__MonthCos: косинус місяця;
– remainder__DayOfYearSin: синус дня року;
– remainder__DayOfYearCos: косинус дня року;
– remainder__WindBearingSin: синус азимута напрямку вітру;
– remainder__WindBearingCos: косинус азимута напрямку вітру.

Префікси num__, cat__ та remainder__ були додані інструментом ColumnTransformer під час попередньої обробки даних в Python і вказують на тип обробки відповідної ознаки. Порядок цих ознак є критично важливим для коректної роботи моделі TensorFlow Lite.

2.4.4 Архітектура моделі, тренування та оцінка

Для вирішення задачі регресії – прогнозування “FLT” – було обрано архітектуру штучної нейронної мережі. Вхідні числові ознаки (після генерації циклічних та One-Hot Encoded ознак) перед подачею в модель масштабувалися за допомогою StandardScaler. Цільова змінна (Apparent Temperature (C)) також масштабувалася перед тренуванням для стабілізації процесу навчання.

Архітектура нейронної мережі включала вхідний шар, розрахований на 15 підготовлених ознак, за яким слідували три приховані повнозв’язні шари з функціями активації ReLU (128, 64 та 32 нейрони відповідно). Для покращення узагальнення та запобігання перенавчанню після кожного прихованого шару застосовувалися шари нормалізації Batch Normalization–та шари Dropout з коефіцієнтом 0.3. Вихідний шар складався з одного нейрона з лінійною функцією активації, що безпосередньо передбачав масштабоване значення відчуваної температури.

Модель компілювалася з використанням оптимізатора Adam з низькою швидкістю навчання (0.0005) та середньоквадратичної помилки (MSE) як функції втрат. В якості основної метрики для оцінки якості моделі під час тренування та на тестовій вибірці використовувалася середня абсолютна помилка (MAE). Тренування проводилося на 100 епохах з розміром батчу 256 та механізмом ранньої зупинки (EarlyStopping з patience=15), який припиняв навчання, якщо помилка на валідаційній вибірці не покращувалася протягом заданої кількості епох, та відновлював найкращі ваги моделі.

Після завершення тренування, модель продемонструвала середню абсолютну помилку на тестовій вибірці близько 0.29 градуса Цельсія при прогнозуванні відчуваної температури у фактичних одиницях після зворотного масштабування. Такий результат свідчить про високу здатність моделі апроксимувати залежність відчуваної температури від вхідних метеорологічних параметрів на основі використаного набору даних.

2.4.5 Конвертація моделі у формат TensorFlow Lite та підготовка для інтеграції

Для використання в мобільному застосунку натренована модель Keras була конвертована у формат TensorFlow Lite (.tflite) за допомогою стандартного інструментарію TensorFlow. Під час конвертації було застосовано стандартні оптимізації (tf.lite.Optimize.DEFAULT), що сприяють зменшенню розміру файлу моделі та потенційному пришвидшенню її виконання на мобільних пристроях без суттєвої втрати точності.

Разом з файлом моделі (feels_like_final_model.tflite) було збережено наступні артефакти, необхідні для коректної підготовки вхідних даних та інтерпретації вихідних значень в Android-застосунку:

– файл зі списком назв та порядком вхідних ознак, які очікує модель (feels_like_final_feature_names.txt);

– файл з параметрами масштабування (середні значення та стандартні відхилення) для тих вхідних числових ознак, що підлягали стандартизації (feels_like_scaler_X_params_final.npz);
– файл з параметрами масштабування (середнє значення та стандартне відхилення) для цільової змінної (Apparent Temperature (C)), які використовуються для зворотного перетворення передбаченого моделлю значення у реальні градуси Цельсія (feels_like_scaler_y_params_final.npz).

Ці компоненти забезпечують можливість коректної інтеграції та функціонування моделі машинного навчання безпосередньо на пристрої користувача.

2.5 Висновки до розділу 2

У даному розділі було детально описано етап проєктування мобільного застосунку прогнозу погоди “Pixel Weather”. На основі попереднього аналізу було сформульовано вичерпний перелік функціональних та нефункціональних вимог, що стали основою для подальших проєктних рішень.

Обґрунтовано вибір архітектурного шаблону MVVM у поєднанні з Repository Pattern та використанням механізму впровадження залежностей Hilt, що забезпечує модульність, готовність коду до тестування та супроводження. Також детально описано та обґрунтовано вибір технологічного стеку, включаючи мову програмування Kotlin, декларативний UI-фреймворк Jetpack Compose для реалізації унікального піксель-арт інтерфейсу, бібліотеки Retrofit та Kotlinx.Serialization для мережевої взаємодії, Room для локального зберігання даних та TensorFlow Lite для інтеграції моделі машинного навчання.

Для візуалізації структури та поведінки системи було розроблено набір UML-діаграм: функціональну схему, структурну схему модулів, діаграму компонентів, діаграми класів для шару домену, DTO та діаграму

послідовностей для ключового сценарію отримання погоди. Ці моделі дозволили чітко визначити взаємозв’язки між компонентами системи.

Окрему увагу було приділено проєктуванню компонента машинного навчання. Було визначено задачу прогнозування “FLT”, обрано та підготовлено відповідний набір даних, сформовано набір вхідних ознак. Описано архітектуру нейронної мережі, процес її тренування, оцінку якості за метрикою MAE та конвертацію у формат TensorFlow Lite для подальшої інтеграції в мобільний застосунок.

3 РОЗРОБКА ІНТЕРФЕЙСУ, РЕАЛІЗАЦІЯ ОСНОВНИХ ФУНКЦІЙ ТА ІНТЕГРАЦІЯ ML-МОДЕЛІ

3.1 Опис середовища розробки та інструментів

Розробка мобільного застосунку “Pixel Weather” здійснювалася з використанням інтегрованого середовища розробки Android Studio Meerkat | 2025.3.2.

В якості основної мови програмування було обрано Kotlin у версії 2.1.20, що забезпечує сучасний підхід до розробки під платформу Android завдяки своїй лаконічності, безпеці типів та підтримці корутин.

Система збирання проєкту базується на Gradle з використанням скриптів конфігурації на Kotlin DSL. Управління версіями коду здійснювалося за допомогою системи контролю версій Git з розміщенням репозиторію на платформі GitHub.

Для тестування (хоча й обмеженого в рамках даної роботи) було використано jUnit для Unit-тестів та інструментів Compose для Ui-тестів. Розробка та тестування проводилися переважно на віртуальних пристроях android (емуляторах) з різними версіями ОС та розмірами екранів, а також на фізичному пристрої Xiaomi Redmi Note 10 Pro.

3.2 Реалізація шарів застосунку

Програмна реалізація мобільного застосунку “Pixel Weather” здійснювалася відповідно до спроєктованої тришарової архітектури (UI Layer, Domain Layer, Data Layer) з використанням шаблону MVVM. Такий підхід дозволив чітко розділити відповідальності між компонентами системи та забезпечити її гнучкість і готовність до підтримки та оновлення.

3.2.1 Мережевий шар та робота з API

Мережевий шар відповідає за взаємодію застосунку із зовнішніми сервісами для отримання даних.

У проєкті “Pixel Weather” використовуються два основні API: OpenWeatherMap API для отримання метеорологічних даних (поточна погода, погодинний та денний прогнози) та Geoapify API для функцій геокодингу (автозаповнення назв міст при пошуку та реверс-геокодинг для визначення назви поточної локації за координатами).

Для здійснення HTTP-запитів та обробки відповідей було обрано бібліотеку Retrofit у поєднанні з OkHttp клієнтом. Retrofit дозволяє декларативно описувати API ендпоінти за допомогою інтерфейсів та анотацій, що значно спрощує процес мережевої взаємодії.

Для серіалізації та десеріалізації даних у форматі JSON, що передаються через API, використовується бібліотека Kotlinx.Serialization, яка забезпечує ефективну та безпечну роботу з Kotlin Data класами.

Ключовим елементом мережевого шару є інтерфейс ApiService, який визначає всі необхідні запити до зовнішніх API.

Приклад визначення методів для отримання поточної погоди та прогнозу з OpenWeatherMap API, а також для автозаповнення міст з Geoapify API, наведено на рисунку 3.1.

Створення та надання екземплярів Retrofit, OkHttpClient, конфігурації Json для Kotlinx.Serialization, а також самої реалізації ApiService іншим компонентам застосунку здійснюється за допомогою механізму впровадження залежностей Hilt.

Відповідні провайдери визначені у Hilt-модулі NetworkModule. Такий підхід забезпечує створення синглтон-екземплярів цих важливих сервісів та їх централізоване конфігурування, що підвищує загальну зручність підтримки та простоту тестування коду.

```
package com.artemzarubin.weatherml.data.remote

import com.artemzarubin.weatherml.data.remote.dto.CurrentWeatherResponseDto
import com.artemzarubin.weatherml.data.remote.dto.ForecastResponseDto
import com.artemzarubin.weatherml.data.remote.dto.GeoapifyAutocompleteResponseDto
import com.artemzarubin.weatherml.data.remote.dto.GeoapifyReverseGeocodeResponseDto
import retrofit2.http.GET
import retrofit2.http.Query

interface ApiService {

    @GET("weather")
    suspend fun getCurrentWeather(
        @Query("lat") latitude: Double,
        @Query("lon") longitude: Double,
        @Query("appid") apikey: String,
        @Query("units") units: String = "metric",
        @Query("lang") lang: String = "en"
    ): CurrentWeatherResponseDto

    @GET("forecast")
    suspend fun getForecast(
        @Query("lat") latitude: Double,
        @Query("lon") longitude: Double,
        @Query("appid") apikey: String,
        @Query("units") units: String = "metric",
        @Query("lang") lang: String = "en",
        @Query("cnt") count: Int = 40
    ): ForecastResponseDto

    @GET("https://api.geoapify.com/v1/geocode/autocomplete")
    suspend fun getCityAutocomplete(
        @Query("text") text: String,
        @Query("apikey") apikey: String,
        @Query("limit") limit: Int = 10,
        @Query("lang") lang: String = "en",
        @Query("type") type: String = "city"
    ): GeoapifyAutocompleteResponseDto

    @GET("https://api.geoapify.com/v1/geocode/reverse")
    suspend fun reverseGeocode(
        @Query("lat") latitude: Double,
        @Query("lon") longitude: Double,
        @Query("apikey") apikey: String,
        @Query("type") type: String = "city",
        @Query("lang") language: String = "en"
    ): GeoapifyReverseGeocodeResponseDto
}
```

Рисунок 3.1 – Фрагмент інтерфейсу ApiService для визначення запитів до API

3.2.2 Шар даних: репозиторій, локальне сховище та перетворення даних

Шар даних у застосунку “Pixel Weather” відповідає за надання узгодженого доступу до даних з різних джерел, таких як мережеві API та локальна база даних, а також за перетворення цих даних у формат, зручний для використання шаром бізнес-логіки та ViewModel. Ключовим компонентом цього шару є репозиторій, реалізований за шаблоном Repository Pattern.

Репозиторій (WeatherRepository та WeatherRepositoryImpl) виступає як єдина точка взаємодії для ViewModel при запиті погодних даних, інформації про локації чи налаштувань. Він абстрагує логіку отримання даних, приховуючи від ViewModel деталі того, чи дані завантажуються з мережі, чи беруться з локального кешу або бази даних. Інтерфейс WeatherRepository визначає контракт для отримання всіх необхідних даних, включаючи поточну погоду, прогнози, список збережених локацій, а також для виконання операцій збереження, оновлення та видалення локацій. Конкретна реалізація, WeatherRepositoryImpl, інкапсулює логіку взаємодії з ApiService для отримання даних з мережі, з SavedLocationDao для операцій з локальною базою даних Room, та з WeatherModelInterpreter для отримання передбачень від ML-моделі. Важливим аспектом реалізації репозиторію є обробка можливих помилок та повернення результатів у вигляді об’єкта Resource, що дозволяє ViewModel адекватно реагувати на різні стани завантаження даних (успіх, помилка, завантаження).

Для локального зберігання даних, таких як список обраних користувачем локацій та їх порядок, а також налаштування застосунку (одиниці виміру, тема), використовується бібліотека Room Persistence Library. Було визначено сутність SavedLocationEntity, що представляє таблицю збережених локацій у базі даних SQLite. Для взаємодії з цією таблицею створено об’єкт доступу до даних (DAO) – SavedLocationDao, який містить методи для виконання CRUD-операцій (Create, Read, Update, Delete) та інших специфічних запитів. Наприклад, SavedLocationDao надає методи для

отримання всіх збережених локацій або поточної активної локації у вигляді kotlinx.coroutines.flow.Flow, що забезпечує реактивне оновлення даних в UI при їх зміні в базі. Клас AppDatabase, що успадковує RoomDatabase, визначає саму базу даних та надає доступ до DAO. Екземпляри AppDatabase та SavedLocationDao надаються іншим компонентам за допомогою Hilt. Фрагмент коду, що демонструє визначення сутності SavedLocationEntity та частини інтерфейсу SavedLocationDao, наведено на рисунку 3.2.

```
// Фрагмент SavedLocationEntity.kt
package com.artemzarubin.weatherml.data.local

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = “saved_locations”)
data class SavedLocationEntity(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    val cityName: String,
    val countryCode: String?,
    val latitude: Double,
    val longitude: Double,
    val isCurrentLocation: Boolean = false,
    val orderIndex: Int = 0
)

// Фрагмент SavedLocationDao.kt
package com.artemzarubin.weatherml.data.local

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import kotlinx.coroutines.flow.Flow

@Dao
interface SavedLocationDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertLocation(location: SavedLocationEntity): Long

    @Query(“SELECT * FROM saved_locations ORDER BY orderIndex ASC, cityName ASC”)
    fun getAllSavedLocations(): Flow<List<SavedLocationEntity>>

    @Query(“SELECT * FROM saved_locations WHERE isCurrentLocation = 1 LIMIT 1”)
    fun getCurrentActiveLocation(): Flow<SavedLocationEntity?>

    // ... інші методи DAO ...
}
```

Рисунок 3.2 – Приклад визначення сутності та DAO для збережених локацій

Важливою частиною шару даних є перетворення (відображення) даних між різними представленнями: DTO (Data Transfer Objects), отримані від API, перетворюються на моделі даних предметної області (Domain Models), які використовуються у ViewModel та UI; а Domain Models, у свою чергу, можуть перетворюватися на сутності бази даних (Entities) для збереження, і навпаки. Це відображення реалізовано за допомогою функцій-розширень, згрупованих у файлі WeatherMappers.kt. Такий підхід забезпечує чітке розділення моделей даних для різних шарів та запобігає «протіканню» деталей реалізації одного шару в інший. Наприклад, CurrentWeatherResponseDto перетворюється на CurrentWeather (Domain Model), а SavedLocation (Domain Model) трансформується у SavedLocationEntity для збереження в Room.

3.2.3 Шар бізнес-логіки та ViewModel

Шар бізнес-логіки (Domain Layer) у застосунку “Pixel Weather” інкапсулює основні правила та операції, пов’язані з обробкою даних, незалежно від їх джерела чи способу відображення. Хоча в даному проєкті окремі UseCase-класи для кожної операції не створювалися для спрощення, логіка, яку вони б виконували, частково реалізована в WeatherRepositoryImpl та MainViewModel. Основними сутностями цього шару є моделі даних предметної області (CurrentWeather, HourlyForecast, DailyForecast, SavedLocation, WeatherDataBundle), які представляють інформацію в чистому, незалежному від платформи вигляді.

Ключовим компонентом, що поєднує шар даних і шар представлення, є MainViewModel. Вона реалізована за шаблоном MVVM і відповідає за управління станом інтерфейсу з використанням StateFlow для відображення динамічних даних (поточна погода, збережені локації, результати пошуку, стан завантаження).

MainViewModel взаємодіє з WeatherRepository для отримання необхідних даних, обробляє дії користувача (вибір міста, оновлення погоди,

зміна налаштувань), керує логікою пейджера між геолокацією та збереженими містами, ініціалізує отримання геолокації через LocationTracker з обробкою дозволів, а також працює з UserPreferencesRepository для збереження й завантаження користувацьких налаштувань (одиниці виміру, тема).

Дані і стани передаються у вигляді StateFlow, на які підписується UI (Composable-функції), забезпечуючи реактивне оновлення інтерфейсу при зміні інформації.

Для керування асинхронними операціями та життєвим циклом ViewModel використовується viewModelScope з бібліотеки AndroidX Lifecycle. Запити до репозиторію та інші тривалі операції виконуються у фоновому режимі за допомогою легковагових потоків, що дозволяють не блокувати основний UI-потік. Результати таких операцій, які можуть бути успішними або містити помилки, упаковуються у клас Resource, що забезпечує передачу стану (Loading, Success, Error) до інтерфейсу користувача. Приклад коду, який ілюструє отримання даних для пейджера та ініціалізацію ViewModel, наведено на рисунку 3.3.

```
// У MainViewModel.kt
@HiltViewModel
class MainViewModel @Inject constructor(
    private val weatherRepository: WeatherRepository,
    private val locationTracker: LocationTracker,
    private val userPreferencesRepository: UserPreferencesRepository,
    private val application: Application
) : ViewModel() {

    private val _weatherDataStateMap = MutableStateFlow<Map<String,
Resource<WeatherDataBundle>>>(emptyMap())
    val weatherDataStateMap: StateFlow<Map<String, Resource<WeatherDataBundle>>> =
_weatherDataStateMap.asStateFlow()

    private val _savedLocationsFromDbFlow =
weatherRepository.getSavedLocations().distinctUntilChanged()
    private val _geolocationPagerItemState =
MutableStateFlow(PagerItem.GeolocationPage(isLoadingDetails = false))

    private val _currentPagerIndex = MutableStateFlow(0)
```

Рисунок 3.3 – Фрагмент класу MainViewModel: управління станом та ініціалізація (аркуш 1)

```
    val currentPagerIndex: StateFlow<Int> = _currentPagerIndex.asStateFlow()

    val pagerItems: StateFlow<List<PagerItem>> =
        combine(
            _savedLocationsFromDbFlow,
            _geolocationPagerItemState,
            hasLocationPermissionFlow().distinctUntilChanged()
        ) { savedDbList, geoPageState, hasPermission ->
            val list = mutableListOf<PagerItem>()
            if (hasPermission) { list.add(geoPageState) }
            list.addAll(savedDbList.map { PagerItem.SavedPage(it) })
            list.distinctBy { it.id }
        }.stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000), emptyList())

    val currentActivePagerItem: StateFlow<PagerItem?> =
        combine(pagerItems, _currentPagerIndex) { items, index ->
            items.getOrNull(index)
        }.stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000), null)

    init {
        Log.d("MainViewModel", "ViewModel initialized.")
        observeActivePagerItemToFetchWeather()
        tryInitialSetupOrLoadActive()
    }

    private fun tryInitialSetupOrLoadActive() {
        viewModelScope.launch {
            // ... (логіка визначення початкової сторінки пейджера) ...
        }
    }
    // ... (інші ключові методи, onPageChanged або onCitySuggestionSelected)
}
```

Рисунок 3.3 – Фрагмент класу MainViewModel: управління станом та ініціалізація (аркуш 2)

Така організація ViewModel дозволяє відокремити логіку управління станом від UI, що робить код більш чистим, зрозумілим та легким для тестування.

3.3 Реалізація користувацького інтерфейсу у стилі піксель-арт

Інтерфейс (UI) мобільного застосунку "Pixel Weather" розроблено з використанням декларативного підходу Jetpack Compose, що дозволило

ефективно реалізувати унікальний візуальний стиль піксель-арт. Цей стиль є ключовою особливістю застосунку, спрямованою на створення оригінального та привабливого для користувача досвіду. Усі візуальні елементи, включаючи фони, іконки, шрифти та компоненти відображення даних, були підібрані або створені для підтримки єдиної піксельної естетики.

Для структурування UI та забезпечення його відповідності архітектурі MVVM, кожен екран або значний логічний блок інтерфейсу представлений набором Composable-функцій. Ці функції отримують необхідні дані та стан з відповідної MainViewModel (через StateFlow, зібрані за допомогою collectAsState()) та реагують на дії користувача, передаючи їх назад у ViewModel для обробки.

3.3.1 Головний екран з пейджером погоди (WeatherScreen)

Головний екран застосунку (WeatherScreen) є центральним елементом взаємодії з користувачем. Він реалізований з використанням HorizontalPager для забезпечення можливості перегортання між сторінками, кожна з яких відображає погоду для окремої локації (поточна геолокація або одна зі збережених). Стан пейджера синхронізується з MainViewModel для коректного завантаження та відображення даних при зміні сторінки.

Кожна сторінка пейджера містить наступні ключові інформаційні блоки, реалізовані як окремі Composable-функції та обгорнуті у власний компонент PixelArtCard для стилістичної єдності.

Компоненти інтерфейсу включають CurrentWeatherMainSection, який відображає основну інформацію про поточну погоду: велику іконку стану, поточну температуру, словесний опис та «FLT» – значення з API і передбачення ML-моделі.

Далі йде секція погодинного прогнозу, що представляє горизонтальний список із прогнозом на найближчі години, де для кожної години вказано час, іконку та температуру.

Секція денного прогнозу відображає вертикальний список з прогнозом на кілька днів вперед, включно з датою, іконкою, максимальними та мінімальними температурами і ймовірністю опадів.

CurrentWeatherDetailsSection надає детальну інформацію про поточну погоду: швидкість і напрямок вітру, вологість, тиск, видимість та час сходу і заходу сонця.

Для відображення стану завантаження даних використовується анімований PixelatedSunLoader, а у випадку помилок або відсутності даних – відповідні текстові повідомлення. У верхній частині екрану розташований TopAppBar, що відображає назву поточної локації та містить кнопки для переходу на екран управління містами та екран налаштувань. Приклад реалізації компонента PixelArtCard та фрагмент коду головного екрану з використанням HorizontalPager наведено на рисунку 3.4.

```
// Фрагмент PixelArtCard.kt (з ui.common)
package com.artemzarubin.weatherml.ui.common

import androidx.compose.foundation.BorderStroke
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.ColumnScope
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Card
import androidx.compose.material3.CardDefaults
import androidx.compose.material3.MaterialTheme
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.RectangleShape // Для гострих кутів
import androidx.compose.ui.unit.Dp
import androidx.compose.ui.unit.dp

@Composable
fun PixelArtCard(
    modifier: Modifier = Modifier,
    internalPadding: Dp = 16.dp, // Внутрішній відступ за замовчуванням
    borderWidth: Dp = 2.dp,      // Ширина рамки за замовчуванням
    content: @Composable ColumnScope.() -> Unit
) {
    Card(
        modifier = modifier,
```

Рисунок 3.4 – Приклад реалізації кастомного компонента PixelArtCard та використання HorizontalPager (аркуш 1)

```
        shape = RectangleShape, // Гострі кути для піксельного стилю
        colors = CardDefaults.cardColors(
            containerColor = MaterialTheme.colorScheme.surfaceVariant.copy(alpha = 0.5f) //
Напівпрозорий фон
        ),
        border = BorderStroke(borderWidth, MaterialTheme.colorScheme.outline.copy(alpha =
0.7f)) // Рамка
    ) {
        Column(
            modifier = Modifier.padding(internalPadding)
        ) {
            content()
        }
    }
}

// Фрагмент WeatherScreen.kt
// ...
    HorizontalPager(
        state = pagerState,
        modifier = Modifier.fillMaxSize()
    ) { pageIndex ->
        val pagerItem = pagerItemsList.getOrNull(pageIndex)
        if (pagerItem != null) {
            val weatherStateForThisPage by remember(pagerItem.id, weatherDataMap) {
                derivedStateOf { weatherDataMap[pagerItem.id] ?: Resource.Loading() }
            }
            WeatherPageContent(
                weatherState = weatherStateForThisPage,
                temperatureUnit = userPreferences.temperatureUnit,
                isGeolocationPageAndLoadingDetails = (pagerItem is PagerItem.GeolocationPage
&& pagerItem.isLoadingDetails)
            )
        } // ...
    }
// ...
```

Рисунок 3.4 – Приклад реалізації кастомного компонента PixelArtCard та використання HorizontalPager (аркуш 2)

3.3.2 Екран управління збереженими локаціями (ManageCitiesScreen)

Екран «Управління локаціями» (ManageCitiesScreen) надає користувачеві інструменти для персоналізації списку міст, для яких він бажає отримувати прогноз погоди. Цей екран є важливою частиною функціоналу, що

підвищує зручність використання застосунку та дозволяє швидко перемикатися між прогнозами для різних географічних точок. Навігація на цей екран здійснюється з головного екрану WeatherScreen за допомогою відповідної іконки у TopAppBar.

На екрані **ManageCitiesScreen** реалізовані основні функціональні можливості для роботи з локаціями. Користувач може шукати та додавати нові міста, вводячи назву у текстове поле. Після введення трьох символів застосунок звертається до Geoapify API для отримання списку відповідних міст, які відображаються для вибору. Обране місто додається до локальної бази та автоматично стає активною сторінкою пейджера на головному екрані.

Всі збережені локації відображаються у вигляді вертикального списку з назвами міст та кодами країн.

Користувач може видаляти локації через іконку кошика; якщо видаляється активна сторінка, пейджер переходить на іншу доступну локацію. Також можливо змінювати порядок локацій за допомогою кнопок «Вгору» та «Вниз», що зберігається в базі та впливає на порядок сторінок у пейджері. Вибір локації в списку встановлює її як активну для перегляду погоди та повертає на головний екран.

Для управління станом цього екрану – пошуковим запитом, результатами автозаповнення та списком збережених локацій – використовується та сама MainViewModel, що й для головного екрану. Це забезпечує централізоване управління даними та логікою. Фрагмент коду, який ілюструє реалізацію списку збережених локацій із можливістю видалення та зміни порядку, наведено на рисунку 3.5.

Для надання користувачеві можливості персоналізувати досвід використання застосунку “Pixel Weather” було реалізовано екран налаштувань (SettingsScreen). Доступ до цього екрану здійснюється через відповідну іконку в TopAppBar на головному екрані WeatherScreen. Екран налаштувань дозволяє користувачеві змінювати ключові параметри відображення даних та вигляд самого застосунку.

```
// У ManageCitiesScreen.kt
// ...
        if (savedLocationPages.isEmpty()) {
            Text("No locations saved yet.", style = MaterialTheme.typography.bodyMedium)
        } else {
            LazyColumn(modifier = Modifier.fillMaxWidth().weight(1f)) {
                itemsIndexed(
                    items = savedLocationPages,
                    key = { _, savedPageItem -> savedPageItem.id }
                ) { index, savedPageItem ->
                    SavedLocationRow(
                        location = savedPageItem.location,
                        onMoveUp = {
                            if (index > 0) {
                                viewModel.moveSavedLocation(index, index - 1)
                            }
                        },
                        onMoveDown = {
                            if (index < savedLocationPages.size - 1) {
                                viewModel.moveSavedLocation(index, index + 1)
                            }
                        },
                        onSelect = {
                            val targetIndexInFullPager = pagerItemsList.indexOfFirst { it.id ==
savedPageItem.id }
                            if (targetIndexInFullPager != -1) {
                                viewModel.onPageChanged(targetIndexInFullPager)
                            }
                            onNavigateBack()
                        },
                        onDelete = {
                            viewModel.deleteLocationAndUpdatePager(savedPageItem.location.id)
                        },
                        canMoveUp = (index > 0),
                        canMoveDown = (index < savedLocationPages.size - 1)
                    )
                    HorizontalDivider(color = MaterialTheme.colorScheme.outline.copy(alpha =
0.3f))
                }
            }
        }
// ...

// Фрагмент SavedLocationRow
@Composable
fun SavedLocationRow(
    location: SavedLocation,
    onMoveUp: () -> Unit,
    onMoveDown: () -> Unit,
    onSelect: () -> Unit,
```

Рисунок 3.5 – Фрагмент екрану ManageCitiesScreen: список збережених локацій (аркуш 1)

```
    onDelete: () -> Unit,
    canMoveUp: Boolean,
    canMoveDown: Boolean
) {
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 4.dp),
        verticalAlignment = Alignment.CenterVertically
    ) {
        Column(horizontalAlignment = Alignment.CenterHorizontally) {
            IconButton(onClick = onMoveUp, enabled = canMoveUp, modifier =
Modifier.size(32.dp)) {
                Image(painter = painterResource(id = R.drawable.ic_pixel_arrow_up), /*...*/)
            }
            IconButton(onClick = onMoveDown, enabled = canMoveDown, modifier =
Modifier.size(32.dp)) {
                Image(painter = painterResource(id = R.drawable.ic_pixel_arrow_down), /*...*/)
            }
        }
        Spacer(modifier = Modifier.width(8.dp))
        Text(
            text = "${location.cityName}${location.countryCode?.let { ", $it" } ?: ""}",
            modifier = Modifier.weight(1f).clickable(onClick = onSelect).padding(vertical = 8.dp,
horizontal = 8.dp),
            /*...*/
        )
        IconButton(onClick = onDelete, modifier = Modifier.size(32.dp)) {
            Image(painter = painterResource(id = R.drawable.trash_bin), /*...*/)
        }
    }
}
```

Рисунок 3.5 – Фрагмент екрану ManageCitiesScreen: список збережених локацій (аркуш 2)

На даний момент на екрані налаштувань реалізовано кілька ключових можливостей. По-перше, користувач може обрати одиниці виміру температури – градуси Цельсія (°C) або Фаренгейта (°F). Вибір здійснюється за допомогою радіокнопок, а обране значення зберігається в локальному сховищі DataStore Preferences через UserPreferencesRepository. MainViewModel підписана на зміни цих налаштувань і при оновленні ініціює перезавантаження погодних даних з API з відповідним параметром units.

По-друге, доступний вибір теми оформлення: системна (за замовчуванням, залежно від налаштувань пристрою), світла або темна. Вибір

також зберігається в DataStore Preferences і застосовується до інтерфейсу «на льоту» за допомогою оновлення MaterialTheme в кореневій Composable-функції.

Окрім того, на екрані налаштувань присутній пункт меню для переходу на окремий екран «Про застосунок» (AboutScreen).

Екран «Про застосунок» (AboutScreen) містить загальну інформацію про програмний продукт “Pixel Weather”, його призначення, дані про розробника та наукового керівника. Також на цьому екрані наведено перелік використаних зовнішніх ресурсів, таких як погодні API (OpenWeatherMap, Geoapify), джерела піксельних шрифтів та іконок, із зазначенням авторів та джерел, що відповідає принципам академічної доброчесності та поваги до авторських прав. Для відображення інформації в оригінальному вигляді реалізовано анімацію тексту, який плавно рухається знизу вгору, схожу на рух титрів у кінофільмах. У нижній частині екрану налаштувань також показується поточна версія застосунку. Фрагмент коду, що демонструє реалізацію перемикачів одиниць виміру температури на екрані налаштувань, наведено на рисунку 3.6.

```
// У SettingsScreen.kt
// ... (Column та Text(“Temperature Units”)) ...
        Row(
            modifier = Modifier
                .fillMaxWidth()
                .clickable { viewModel.updateTemperatureUnit(TemperatureUnit.CELSIUS) }
                .padding(vertical = 4.dp),
            verticalAlignment = Alignment.CenterVertically
        ) {
            Image(
                painter = painterResource(
                    id = if (userPreferences.temperatureUnit == TemperatureUnit.CELSIUS)
R.drawable.ic_pixel_radio_checked
                        else R.drawable.ic_pixel_radio_unchecked
                ),
                contentDescription = “Celsius radio button”,
                modifier = Modifier.size(24.dp),
                colorFilter = ColorFilter.tint(MaterialTheme.colorScheme.onSurface)
```

Рисунок 3.6 – Фрагмент екрану SettingsScreen: вибір одиниць виміру температури (аркуш 1)

```
            )
            Spacer(modifier = Modifier.width(8.dp))
            Text(
                text = stringResource(id = R.string.celsius), // Використання stringResource
                style = MaterialTheme.typography.bodyLarge
            )
        }

        Row(
            modifier = Modifier
                .fillMaxWidth()
                .clickable {
viewModel.updateTemperatureUnit(TemperatureUnit.FAHRENHEIT) }
                .padding(vertical = 4.dp),
            verticalAlignment = Alignment.CenterVertically
        ) {
            Image(
                painter = painterResource(
                    id = if (userPreferences.temperatureUnit == TemperatureUnit.FAHRENHEIT)
R.drawable.ic_pixel_radio_checked
                        else R.drawable.ic_pixel_radio_unchecked
                ),
                contentDescription = “Fahrenheit radio button”,
                modifier = Modifier.size(24.dp),
                colorFilter = ColorFilter.tint(MaterialTheme.colorScheme.onSurface)
            )
            Spacer(modifier = Modifier.width(8.dp))
            Text(
                text = stringResource(id = R.string.fahrenheit), // Використання stringResource
                style = MaterialTheme.typography.bodyLarge
            )
        }
```

Рисунок 3.6 – Фрагмент екрану SettingsScreen: вибір одиниць виміру температури (аркуш 2)

Реалізація цих екранів дозволяє підвищити гнучкість застосунку та надати користувачеві можливість налаштувати його відповідно до своїх вподобань, а також ознайомитися з інформацією про продукт.

3.4 Інтеграція моделі машинного навчання TensorFlow Lite

Одним із ключових завдань при розробці мобільного застосунку була інтеграція попередньо натренованої моделі машинного навчання для

уточнення прогнозу “FLT”. Для цього використали бібліотеку TensorFlow Lite, яка дозволяє запускати ML-моделі безпосередньо на мобільних пристроях.

Спочатку модель була натренована у Python із застосуванням Keras/TensorFlow та конвертована у формат .tflite. Разом з нею у проєкт були додані файли: список вхідних ознак, а також параметри масштабування, які зберігаються у теці assets і використовуються під час виконання застосунку.

Для роботи з моделлю було реалізовано клас WeatherModelInterpreterImpl, що відповідає за її ініціалізацію, завантаження моделі та файлів із параметрами масштабування, а також створення інтерпретатора TensorFlow Lite. Основна задача цього класу – підготовка вхідних даних на основі актуальних метеоданих (CurrentWeather), що включає вибір та конвертацію числових ознак, масштабування за допомогою стандартних параметрів, а також генерацію додаткових циклічних ознак для часу та напрямку вітру, а також кодування типу опадів.

Після підготовки вхідного масиву виконується передбачення моделі, яке потім коригується за допомогою зворотнього масштабування для отримання кінцевого значення “FLT” у градусах Цельсія. Для коректного завершення роботи передбачено звільнення ресурсів інтерпретатора.

Фрагмент коду, що ілюструє ключову логіку підготовки даних і отримання передбачення, наведено на рисунку 3.7.

```
// У WeatherModelInterpreterImpl.kt
// ... (частина initialize(), де завантажуються scalerXMean, scalerXScale, scalerYMean, scalerYScale, fullFeatureOrderFromTraining)

override fun prepareAndScaleFeatures(currentWeather: CurrentWeather): FloatArray? {
    // ... (перевірки fullFeatureOrderFromTraining.isEmpty(), numTotalFeaturesExpectedByModel == -1) ...

    val featureMap = mutableMapOf<String, Float>()

    // 1. Числові ознаки - отримання та масштабування
    val rawTemperature = currentWeather.temperatureCelsius.toFloat()
    // ... (отримання інших raw значень: вологість, швидкість вітру в км/год, видимість, тиск)
```

Рисунок 3.7 – Фрагмент класу WeatherModelInterpreterImpl: робота з ML-моделлю (аркуш 1)

```
    val rawNumericValues = listOf(rawTemperature, rawHumidity / 100.0f /* ... інші ... */)

    for (i in numericFeatureNamesForXScaling.indices) {
        val simpleName = numericFeatureNamesForXScaling[i]
        val fullNameInModel = fullFeatureOrderFromTraining.find { /* ... */ }
        if (fullNameInModel != null) {
            val rawValue = rawNumericValues[i]
            val mean = scalerXMean[i]; val scale = scalerXScale[i]
            featureMap[fullNameInModel] = (rawValue - mean) / scale
        }
    }

    // 2. Часові ознаки (Sin/Cos)
    val (localHour, localMonth, dayOfYear) = getLocalTimeFeatures(/*...*/)
    featureMap["remainder__HourSin"] = sin(2 * PI * localHour / 24.0).toFloat()
    // ... (інші Sin/Cos для часу)

    // 3. Напрямок вітру (Sin/Cos)
    // ... (розрахунок та додавання remainder__WindBearingSin/Cos)

    // 4. One-Hot Encoded 'Precip Type'
    val currentPrecipType = currentWeather.weatherCondition
    featureMap["cat__Precip Type_rain"] = if (currentPrecipType.contains("rain", ignoreCase
= true)) 1.0f else 0.0f
    featureMap["cat__Precip Type_snow"] = if (currentPrecipType.contains("snow",
ignoreCase = true)) 1.0f else 0.0f
    // Якщо є cat__Precip Type_None, додати його обробку

    // Формування фінального масиву
    val featuresArray = FloatArray(numTotalFeaturesExpectedByModel)
    for (i in fullFeatureOrderFromTraining.indices) {
        featuresArray[i] = featureMap[fullFeatureOrderFromTraining[i]] ?: 0.0f
    }
    return featuresArray
}

override fun getPrediction(input: ModelInput): ModelOutput? {
    // ... (перевірки isInitialized, interpreter != null, input.features.size) ...
    return try {
        val inputBuffer = ByteBuffer.allocateDirect(1 * numTotalFeaturesExpectedByModel *
numBytesPerFloat)
            .apply { order(ByteOrder.nativeOrder()); rewind(); asFloatBuffer().put(input.features)
}
        val outputArray = Array(1) { FloatArray(1) } // Припускаємо, що модель повертає
одне значення
        interpreter?.run(inputBuffer, outputArray)

        val predictedFeelsLikeScaled = outputArray[0][0]
        val predictedFeelsLikeActual = (predictedFeelsLikeScaled * scalerYScale) +
scalerYMean
        ModelOutput(predictedFeelsLikeTemp = predictedFeelsLikeActual)
    } catch (e: Exception) { /* ... логування помилки ... */ null }
}
```

Рисунок 3.7 – Фрагмент класу WeatherModelInterpreterImpl: робота з ML-моделлю (аркуш 2)

Реалізований WeatherModelInterpreterImpl потім інтегрується в загальний потік даних застосунку. Розглянемо цей процес.

Інтеграція в репозиторій: екземпляр WeatherModelInterpreter ін’єктується за допомогою Hilt у WeatherRepositoryImpl. При отриманні даних про поточну погоду з API, WeatherRepositoryImpl викликає методи modelInterpreter.prepareAndScaleFeatures() для підготовки вхідного вектора ознак та modelInterpreter.getPrediction() для отримання прогнозу “FLT”. Це передбачене значення потім додається до об'єкта CurrentWeather (у спеціально створене поле mlFeelsLikeCelsius) перед тим, як фінальний WeatherDataBundle буде передано у MainViewModel.

Відображення в UI: на головному екрані WeatherScreen, у секції CurrentWeatherMainSection, значення currentWeather.mlFeelsLikeCelsius відображається поруч зі значенням “FLT”, отриманим безпосередньо з API. Це дозволяє користувачеві візуально порівняти обидва показники та оцінити результат роботи локальної ML-моделі.

Така архітектура інтеграції дозволяє виконувати ML-передбачення повністю локально на пристрої користувача, забезпечуючи швидку відповідь та незалежність від додаткових мережевих запитів для отримання уточненого прогнозу. Це також відкриває можливості для подальшого розширення функціоналу, наприклад, шляхом додавання інших локальних моделей для прогнозування інших погодних параметрів.

3.5 Висновки до розділу 3

У даному розділі було детально висвітлено процес програмної реалізації мобільного застосунку прогнозу погоди “Pixel Weather”. Розробка здійснювалася на основі проєктних рішень, визначених у попередньому розділі, з використанням інтегрованого середовища Android Studio, мови програмування Kotlin та сучасного набору технологій Android Jetpack.

Було послідовно реалізовано ключові архітектурні шари застосунку відповідно до шаблону MVVM. Зокрема, детально описано створення мережевого шару для ефективної взаємодії з зовнішніми API OpenWeatherMap та Geoapify за допомогою бібліотеки Retrofit. Для обробки форматів даних, що надходять у форматі JSON, застосовано Kotlinx.Serialization – інструмент, який дозволяє зручно перетворювати ці дані у Kotlin-об’єкти та навпаки.

Розглянуто реалізацію шару даних, що включає WeatherRepository для абстрагування джерел даних, та використання локальної бази даних Room для збереження обраних користувачем локацій та їх налаштувань. Особливу увагу приділено логіці MainViewModel, яка керує станом користувацького інтерфейсу, обробляє дії користувача, взаємодіє з репозиторієм та забезпечує дані для пейджера навігації між прогнозами для різних локацій.

У роботі реалізовано основну структуру інтерфейсу, що охоплює перегляд прогнозу погоди, керування локаціями та налаштування застосунку. Описано процес інтеграції попередньо натренованої моделі машинного навчання TensorFlow Lite для прогнозування “FLT” та відображення її результатів в інтерфейсі користувача.

4 ПЕРЕВІРКА ПРОГРАМИ, ОЦІНКА ЗДОБУТИХ РЕЗУЛЬТАТІВ ТА ПІДСУМКИ

Етап тестування є невіддільною частиною життєвого циклу розробки програмного забезпечення, метою якого є перевірка відповідності розробленого продукту функціональним та нефункціональним вимогам, виявлення та усунення дефектів, а також оцінка загальної якості та надійності системи. Для мобільного застосунку “Pixel Weather” було проведено комплексне тестування, що включало ручну перевірку сценаріїв, спроби написання автоматизованих Unit-тестів для ключових компонентів, а також аналіз результатів роботи інтегрованої моделі машинного навчання.

4.1 Стратегія та види тестування

Для мобільного застосунку була обрана комбінована стратегія тестування, яка поєднувала кілька підходів для максимально повного охоплення функціоналу та перевірки якості на різних рівнях. Основну увагу приділено ручному функціональному тестуванню, що дало змогу оцінити роботу застосунку з точки зору кінцевого користувача та перевірити зручність інтерфейсу.

Проводилось тестування UI/UX для оцінки навігації, читабельності та адаптації дизайну до різних розмірів екранів. Було перевірено стабільність роботи, обробку типових помилок (наприклад, відсутність інтернет-з’єднання або відмови у доступі до дозволів), а також частково реалізовано модульні тести для логічних компонентів, зокрема MainViewModel. Окрему увагу приділено перевірці інтеграції моделі машинного навчання: коректності її завантаження, підготовки вхідних даних, отримання результатів і їх відповідності очікуванням на основі API.

Тестування проводилося на віртуальних пристроях Android (емуляторах) з різними версіями ОС (Android 8.0 – Android 13) та на фізичному пристрої Xiaomi Redmi Note 10 Pro для перевірки роботи в реальних умовах.

4.2 Ручне тестування основного функціоналу та сценаріїв

Ручне тестування стало основним методом перевірки функціональності та зручності використання мобільного застосунку. Було розроблено та виконано набір тест-кейсів, що охоплюють ключові сценарії відповідно до функціональних вимог.

Особливу увагу виділено початковому запуску та налаштуванню: перевірялося коректне відображення екрана завантаження, запит на доступ до геолокації з різними варіантами відповіді (дозвіл, тимчасова та постійна відмова), а також подальше отримання погоди для поточного розташування або коректне повідомлення про помилку.

Поглиблене тестування охопило відображення погодних даних на головному екрані – перевірялася повнота і точність показників, таких як температура, відчувається як (API та ML), опис погоди, вітер, тиск, вологість, видимість, а також погодинні та денні прогнози. Усі значення мали відповідати обраним у налаштуваннях одиницям виміру.

Було протестовано пейджер: його можливість перемикання між геолокаційною сторінкою та збереженими містами, коректне оновлення даних, заголовка та індикаторів при зміні сторінок. Також перевірявся функціонал «потягни, щоб оновити», що дозволяє оновлювати дані шляхом прокручування екрана вниз.

Особлива увага була приділена екрану керування локаціями. Перевірялась робота пошуку міст, додавання нових у список збережених з автоматичним переходом на відповідну сторінку, обробка обмеження у 10 міст, а також правильність відображення списку, можливість видалення та

зміни порядку міст. Важливим аспектом було оновлення інтерфейсу пейджера згідно з діями користувача.

У налаштуваннях тестувалися перемикання одиниць виміру температури (Цельсій/Фаренгейт) з миттєвим оновленням показників, вибір теми оформлення (світла, темна, системна) та коректність візуального відображення змін. Також перевірявся перехід на екран «Про застосунок» і повернення назад.

Окремо проводилась перевірка поведінки застосунку в умовах помилок: нестабільне або відсутнє інтернет-з’єднання, неправильні API-ключі, некоректні координати, порожній список збережених локацій. Усі ці ситуації мали відповідно оброблятися та відображатися в інтерфейсі.

У ході проведення ручного тестування було виявлено та усунуто низку недоліків, пов’язаних переважно з логікою управління станами при асинхронних операціях, обробкою життєвого циклу та дозволів, а також з коректністю відображення даних у різних сценаріях.

4.3 Unit-тестування компонентів ViewModel

Для забезпечення якості коду на більш низькому рівні та перевірки коректності роботи окремих логічних компонентів застосунку, зокрема MainViewModel, було здійснено написання автоматизованих Unit-тестів. Основною метою цього етапу було забезпечення правильності обробки станів, взаємодії з залежностями (такими як WeatherRepository, LocationTracker, UserPreferencesRepository) та коректності бізнес-логіки, інкапсульованої у ViewModel.

Для написання Unit-тестів використовувався фреймворк JUnit4. Для ізоляції MainViewModel від її реальних залежностей та забезпечення контрольованої поведінки цих залежностей у тестовому середовищі використовувалася бібліотека **MockK**, яка дозволяє створювати фейкові версії об’єктів для тестування [1]. Це дозволило створювати фейкові репозиторії та

сервіси й задавати їхню поведінку для конкретних тестів. Для тестування асинхронних потоків даних (kotlinx.coroutines.flow.StateFlow), які активно використовуються у MainViewModel для управління станом, застосовувалася бібліотека Turbine, що надає зручні інструменти для перевірки значень, які емітуються потоками. Для перевірки асинхронного коду використовувався StandardTestDispatcher з бібліотеки kotlinx-coroutines-test.

Було розроблено декілька тестових сценаріїв, що покривають ключові аспекти роботи MainViewModel. Перший приклад тесту, що перевіряє початковий стан ViewModel за відсутності дозволів на геолокацію та збережених локацій, наведено на рисунку 4.1.

```
// У MainViewModelTest.kt
@Test
fun `initial state - no permission, no saved locations - shows permission error for geo page`() =
runTest(testDispatcher) {
    // Arrange: дозволів немає (встановлено в setUp, де setPermissions(granted = false))
    // ViewModel ініціалізується в setUp з відповідними моками (порожні списки локацій)
    // every { mockWeatherRepository.getSavedLocations() } returns flowOf(emptyList()) в setUp
    // every { mockWeatherRepository.getCurrentActiveWeatherLocation() } returns flowOf(null) в
setUp

    viewModel = MainViewModel( // viewModel створюється з цими умовами
        weatherRepository = mockWeatherRepository,
        locationTracker = mockLocationTracker,
        application = mockApplication,
        userPreferencesRepository = mockUserPreferencesRepository
    )
    testScheduler.advanceUntilIdle() // Даємо час на виконання init блоку ViewModel

    // Assert
    val geoPageId = PagerItem.GeolocationPage().id
    val state = viewModel.weatherDataStateMap.value[geoPageId]

    assertTrue("State for geo page should be Error, but was $state", state is Resource.Error)
    val errorMessage = (state as Resource.Error).message
    assertTrue(
        "Error message should indicate permission issue. Actual: '$errorMessage'",
        errorMessage?.contains("permission", ignoreCase = true) == true ||
        errorMessage?.contains("Location permission is required to start.", ignoreCase = true) == true || //
Можливе повідомлення з tryInitialSetupOrLoadActive
        errorMessage?.contains("Location permission needed to set initial location.", ignoreCase = true)
== true || // Можливе повідомлення з fetchInitialLocationViaGpsAndSave
        errorMessage?.contains("Location permission needed for geolocation.", ignoreCase = true) ==
true // Можливе повідомлення з fetchDetailsForGeolocationPage
    )
}
```

Рисунок 4.1 – Приклад Unit-тесту для MainViewModel: початковий стан без дозволів

Інший важливий сценарій стосується поведінки ViewModel, коли дозволи на геолокацію надані, але збережених локацій у користувача ще немає. У цьому випадку очікується, що застосунок спробує отримати поточну геолокацію, визначити назву місцевості та завантажити для неї погодні дані. Відповідний тест наведено на рисунку 4.2.

```
// У MainViewModelTest.kt
@Test
fun `initial state - permission granted, no saved locations - fetches geo details and weather`() =
runTest(testDispatcher) {
    // Arrange
    setPermissions(granted = true) // Встановлюємо, що дозволи надані
    every { mockWeatherRepository.getSavedLocations() } returns flowOf(emptyList()) //
Немає збережених локацій
    every { mockWeatherRepository.getCurrentActiveWeatherLocation() } returns
flowOf(null) // Немає активної в БД
    coEvery { mockLocationTracker.getCurrentLocation() } returns mockLocation // Імітуємо
успішне отримання координат (mockLocation визначено в @Before)
    coEvery {
mockWeatherRepository.getLocationDetailsByCoordinates(mockLocation.latitude,
mockLocation.longitude, any()) } returns Resource.Success(mockGeoDetails) // Імітуємо
успішний реверс-геокодинг
    // Імітуємо успішне завантаження погоди для отриманих координат
    coEvery { mockWeatherRepository.getAllWeatherData(eq(mockLocation.latitude),
eq(mockLocation.longitude), any(), any()) } returns
Resource.Success(mockWeatherDataBundle.copy(currentWeather =
mockWeatherDataBundle.currentWeather.copy(cityName = mockGeoDetails.properties?.city
?: “Test City”)))


    // Ініціалізуємо ViewModel з цими налаштуваннями
    viewModel = MainViewModel(mockWeatherRepository, mockLocationTracker,
mockApplication, mockUserPreferencesRepository)
    testScheduler.advanceUntilIdle() // Даємо час на виконання init та наступних корутин

    // Assert
    // 1. Перевіряємо, що деталі геолокації були завантажені та відображені
    val geoItemState = viewModel.pagerItems.value.find { it is PagerItem.GeolocationPage }
as? PagerItem.GeolocationPage
    assertNotNull(“GeolocationPage item should exist in pagerItems”, geoItemState)
    assertEquals(“isLoadingDetails should be false after fetching”, false,
geoItemState?.isLoadingDetails)
    assertEquals(“Fetched city name should match mockGeoDetails”,
mockGeoDetails.properties?.city, geoItemState?.fetchedCityName)
```

Рисунок 4.2 – Приклад Unit-тесту для `MainViewModel`: початковий стан з наданими дозволами (аркуш 1)

```
    // 2. Перевіряємо, що погода для геолокації завантажена
    val geoPageId = PagerItem.GeolocationPage().id
    val weatherState = viewModel.weatherDataStateMap.value[geoPageId]
    assertTrue("Weather for geo page should be Success, but was $weatherState", weatherState
is Resource.Success)
    assertEquals("Weather data city name should match", mockGeoDetails.properties?.city,
(weatherState as Resource.Success).data?.currentWeather?.cityName)

    // 3. Перевіряємо, що геолокація стала активною сторінкою пейджера (індекс 0)
    assertEquals("Current pager index should be 0 (for geolocation)", 0,
viewModel.currentPagerIndex.value)
    assertTrue("Active PagerItem should be GeolocationPage",
viewModel.currentActivePagerItem.value is PagerItem.GeolocationPage)
}
```

Рисунок 4.2 – Приклад Unit-тесту для `MainViewModel`: початковий стан з наданими дозволами (аркуш 2)

Базові тести дозволили перевірити ключові аспекти логіки MainViewModel та отримати досвід у застосуванні сучасних інструментів тестування для застосунків Android. Подальше розширення тестового покриття, включаючи тестування сценаріїв додавання, видалення, зміни порядку локацій, а також зміни налаштувань, є важливим напрямком для майбутнього розвитку та забезпечення довгострокової якості програмного продукту.

4.4 Аналіз результатів роботи моделі машинного навчання

Інтегрована в застосунок "Pixel Weather" модель машинного навчання призначена для прогнозування "FLT" на основі поточних метеорологічних даних. Модель була натренована на історичних даних і конвертована у формат TensorFlow Lite для використання на мобільних пристроях.

Оцінка якості моделі проводилася на етапі її тренування в середовищі Python з використанням метрики середньої абсолютної помилки (MAE). На тестовій вибірці, що не використовувалася під час навчання, модель продемонструвала MAE близько 0.29 градуса Цельсія. Це означає, що в

середньому передбачення моделі відхиляються від реальних значень “FLT” з тестового датасету менш ніж на третину градуса, що є досить високим показником точності для даної задачі.

У мобільному застосунку результат роботи ML-моделі (mlFeelsLikeCelsius) відображається поруч зі значенням «FLT», отриманим з OpenWeatherMap API (feelsLikeCelsius). Користувач мав можливість порівнювати ці значення в різних погодних умовах та для різних локацій. У більшості випадків прогноз моделі був дуже близьким до значення API, що може свідчити про те, що модель навчилася наближено повторювати логіку розрахунку температури «за відчуттям», подібну до тієї, що використовується в OpenWeatherMap. Проте іноді спостерігалися невеликі відхилення на 1-2 градуси. Це можна пояснити тим, що модель, натренована на конкретному наборі даних, може по-своєму інтерпретувати вплив вологості, вітру та температури. Для остаточних висновків щодо якості таких прогнозів потрібен більш тривалий та системний аналіз – наприклад, шляхом порівняння з даними з метеостанцій або реальними відчуттями користувачів.

Інтеграція ML-моделі показала, що можливе локальне обчислення додаткових погодних параметрів прямо на пристрої. У майбутньому цей компонент можна розвивати – наприклад, використовуючи більш різноманітні дані для навчання, експериментуючи з архітектурою моделей або додаючи нові погодні характеристики, які також варто прогнозувати.

4.5 Висновки до розділу 4

У даному розділі було представлено опис процесу тестування мобільного застосунку “Pixel Weather” та проведено аналіз отриманих результатів, зокрема щодо роботи інтегрованої моделі машинного навчання. Обрана комбінована стратегія тестування, дозволила комплексно перевірити відповідність розробленого продукту поставленим вимогам.

Ручне тестування, що охоплювало всі ключові сценарії, дозволило виявити та усунути низку недоліків, що сприяло підвищенню стабільності та якості користувацького досвіду.

Для MainViewModel було написано Unit-тести з використанням фреймворку JUnit4, бібліотеки MockK для створення замінників залежностей і Turbine для перевірки асинхронних потоків даних. Аналіз результатів роботи інтегрованої ML-моделі для прогнозування “FLT” показав, що модель, натренована на історичних даних з MAE близько 0.29°C, у більшості випадків надає передбачення, близькі до значень, отриманих з API OpenWeatherMap.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було розроблено функціональний прототип мобільного застосунку прогнозу погоди “Pixel Weather” для платформи Android, який надає користувачам актуальну метеорологічну інформацію, уточнені дані “FLT” за допомогою локальної моделі машинного навчання та реалізований в унікальному інтерфейсі у стилі піксель-арт.

Для досягнення мети було вирішено наступні завдання:

- **проведено аналіз предметної області**, розглянуто існуючі мобільні застосунки прогнозу погоди та обґрунтовано актуальність розробки власного рішення з унікальними характеристиками;
- **сформульовано детальні функціональні та нефункціональні вимоги** до програмного продукту;
- **обґрунтовано та обрано сучасний технологічний стек** та архітектурні рішення, зокрема шаблон MVVM, Repository Pattern та Dependency Injection за допомогою Hilt;
- **розроблено та інтегровано модель машинного навчання** на основі нейронної мережі, конвертовану у формат TensorFlow Lite, для прогнозування “FLT”;
- **реалізовано основний функціонал застосунку**, включаючи отримання та відображення поточної погоди, погодинного та денного прогнозів для обраної локації, пошук міст, визначення геолокації, збереження обраних локацій у локальній базі даних Room та навігацію між ними за допомогою пейджера;
- **розроблено користувацький інтерфейс** у стилі піксель-арт з використанням Jetpack Compose;
- **реалізовано додатковий функціонал**, такий як налаштування одиниць виміру температури, вибір теми оформлення та екран «Про застосунок»;

– **проведено ручне тестування основного функціоналу та здійснено написання Unit-тестів** для ключових компонентів ViewModel.

Інтегрована ML-модель показала задовільну точність (MAE ≈ 0.29°C на етапі тренування) у прогнозуванні “FLT”, надаючи користувачеві альтернативне значення цього параметра.

ПЕРЕЛІК ПОСИЛАНЬ

1. Kotlin Programming Language. *Офіційна документація kotlinlang.org*. URL: https://kotlinlang.org/docs/home.html (дата звернення: 15.01.2025).

2. Android Developers. Jetpack Compose. *Android Developers*. URL: https://developer.android.com/jetpack/compose (дата звернення: 28.03.2025).

3. Android Developers. Hilt – Dependency Injection. *Android Developers*. URL: https://developer.android.com/training/dependency-injection/hilt-android (дата звернення: 10.03.2025).

4. Android Developers. Room Persistence Library. *Android Developers*. URL: https://developer.android.com/training/data-storage/room (дата звернення: 18.04.2025).

5. Square. Retrofit. *Офіційна документація Square*. URL: https://square.github.io/retrofit/ (дата звернення: 05.03.2025).

6. Kotlin. Kotlinx.serialization. *GitHub репозиторій та документація*. URL: https://github.com/Kotlin/kotlinx.serialization (дата звернення: 06.03.2025).

7. TensorFlow. TensorFlow Lite for mobile and embedded devices. *Офіційна документація TensorFlow*. URL: https://www.tensorflow.org/lite/guide (дата звернення: 20.02.2025).

8. Android Developers. DataStore. *Android Developers*. URL: https://developer.android.com/topic/libraries/architecture/datastore (дата звернення: 22.04.2025).

9. Android Developers. Navigation Compose. *Android Developers*. URL: https://developer.android.com/jetpack/compose/navigation (дата звернення: 01.04.2025).

10. OpenWeatherMap. API documentation. *Офіційний сайт OpenWeatherMap*. URL: https://openweathermap.org/api (дата звернення: 25.01.2025).

11. Geoapify. Geocoding API documentation. *Офіційний сайт Geoapify*. URL: https://www.geoapify.com/geocoding-api/ (дата звернення: 26.01.2025).

12. Geoapify. Reverse Geocoding API documentation. *Офіційний сайт Geoapify*. URL: https://www.geoapify.com/reverse-geocoding-api/ (дата звернення: 20.04.2025).

13. Historical Weather Data for different cities around the world. *Kaggle Dataset*. URL: https://www.kaggle.com/datasets/muthuj7/weather-dataset (дата звернення: 28.01.2025).

14. Goodfellow I., Bengio Y., Courville A. Deep Learning. Cambridge, MA : MIT Press, 2016. 800 p.

15. Martin R. C. Clean Architecture: A Craftsman’s Guide to Software Structure and Design. Upper Saddle River, NJ : Prentice Hall, 2017. 432 p.

16. Freepik Company S.L. *Flaticon. Вебсайт*. URL: https://www.flaticon.com (дата звернення: 10.04.2025).

17. Leal R. VCR OSD Mono Font. *DaFont Вебсайт*. URL: https://www.dafont.com/vcr-osd-mono.font (дата звернення: 26.03.2025).

18. Yasha T. “TY – Fictional Computer OS – 32” Color Palette. *Lospec Вебсайт*. URL: https://lospec.com/palette-list/ty-fictional-computer-os-32 (дата звернення: 27.03.2025).

19. Baranova O. App Icon Original Art. *Vecteezy Вебсайт*. URL: https://www.vecteezy.com/vector-art/45828662-the-sun-icon-for-summer-holidays-in-pixel-art-style-inspired-by-the-games-of-the-80s-90s (дата звернення: 23.04.2025).

20. Loading.io – Ajax loader, loading gif, loading bar, progress bar. *Вебсайт loading.io*. URL: https://loading.io (дата звернення: 01.04.2025).

21. KBV Research. Weather App Market Size, Share & Top Key Players by 2030. *Вебсайт KBV Research*. URL: https://www.kbvresearch.com/weather-app-market/ (дата звернення: 04.04.2025).

22. AccuWeather: Weather Radar. *Google Play Store*. URL: https://play.google.com/store/apps/details?id=com.accuweather.android (дата звернення: 12.03.2025).

23. The Weather Channel – Radar. *Google Play Store*. URL: https://play.google.com/store/apps/details?id=com.weather.Weather (дата звернення: 12.03.2025).

24. Yahoo Weather. *Google Play Store*. URL: https://play.google.com/store/apps/details?id=com.yahoo.mobile.client.android.weather (дата звернення: 12.03.2025).

25. Google Weather – інтегрований сервіс. *Офіційна довідка Google*. URL: https://support.google.com/websearch/answer/13692898?hl=uk (дата звернення: 12.03.2025).

26. Saaty T. L. Decision Making with the Analytic Hierarchy Process. *International Journal of Services Sciences*. 2008. Vol. 1, No. 1. P. 83–98. URL: https://www.rafikulislam.com/uploads/resourses/197245512559a37aadea6d.pdf (дата звернення: 18.04.2025).

27. Triantaphyllou E. Multi-Criteria Decision Making Methods: A Comparative Study. Dordrecht : Springer Science & Business Media, 2000. 320 p. DOI: 10.1007/978-1-4757-3157-6 (дата звернення: 19.04.2025).

28. Android Developers. Guide to app architecture: MVVM. *Android Developers*. URL: https://developer.android.com/topic/architecture (дата звернення: 21.04.2025).

29. OpenWeatherMap API documentation – Current Weather Data. *Офіційний сайт OpenWeatherMap*. URL: https://openweathermap.org/current#data (дата звернення: 21.03.2025).

ДОДАТОК А

Скриншоти інтерфейсу мобільного застосунку

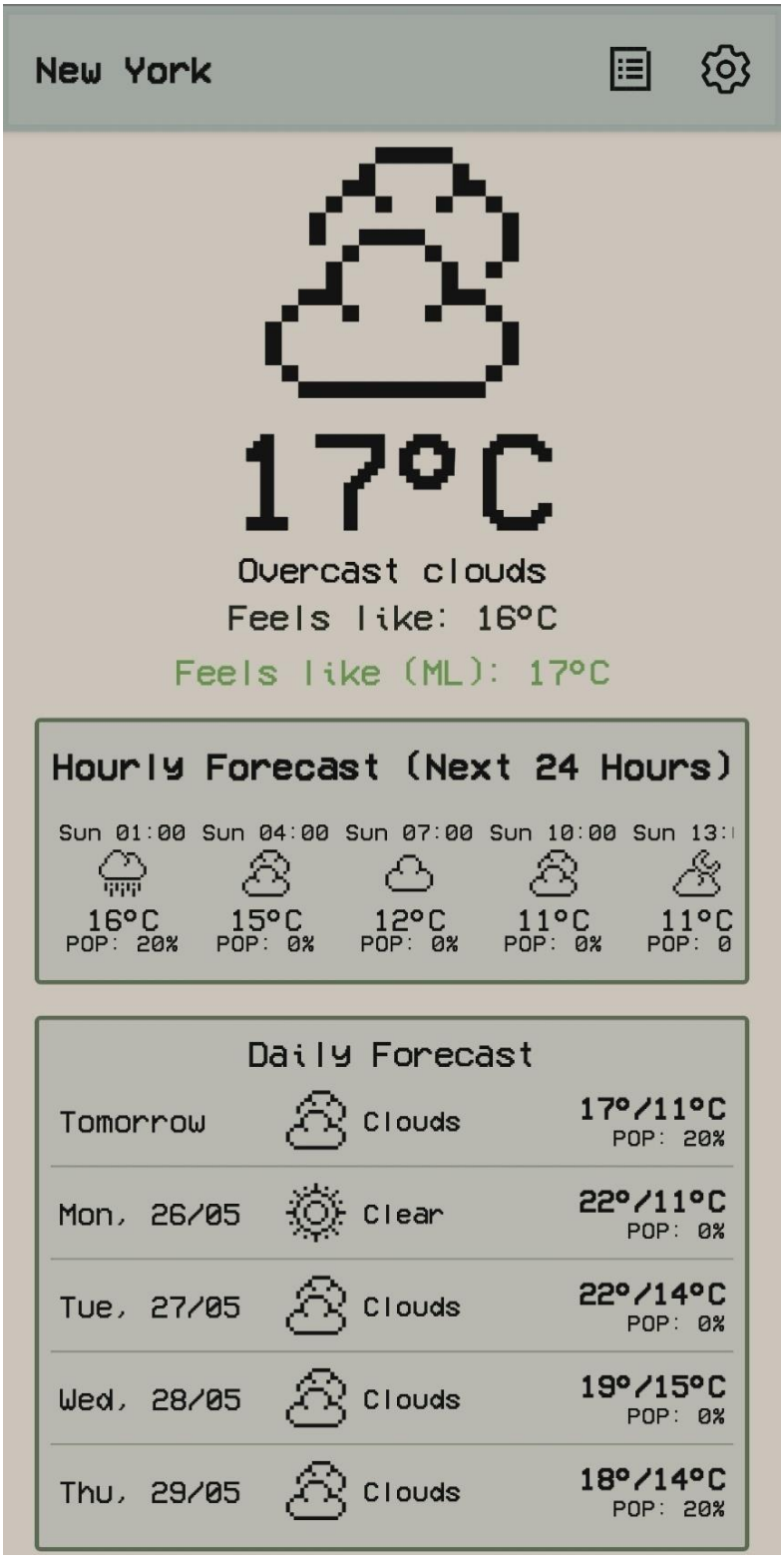


Рисунок А.1 – Головний екран застосунку: відображення погоди для поточної геолокації

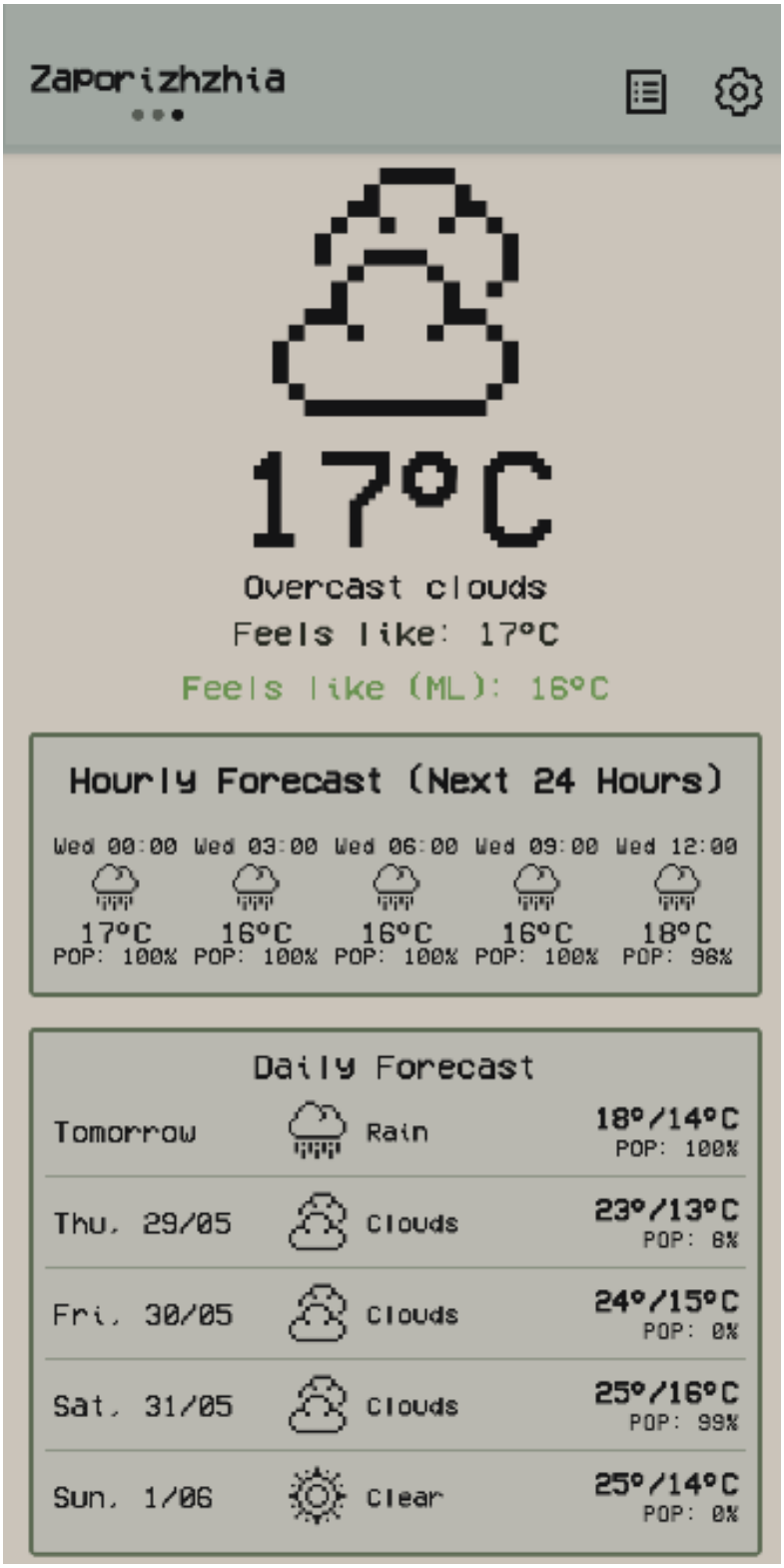


Рисунок А.2 – Головний екран застосунку: відображення погоди для збереженого міста

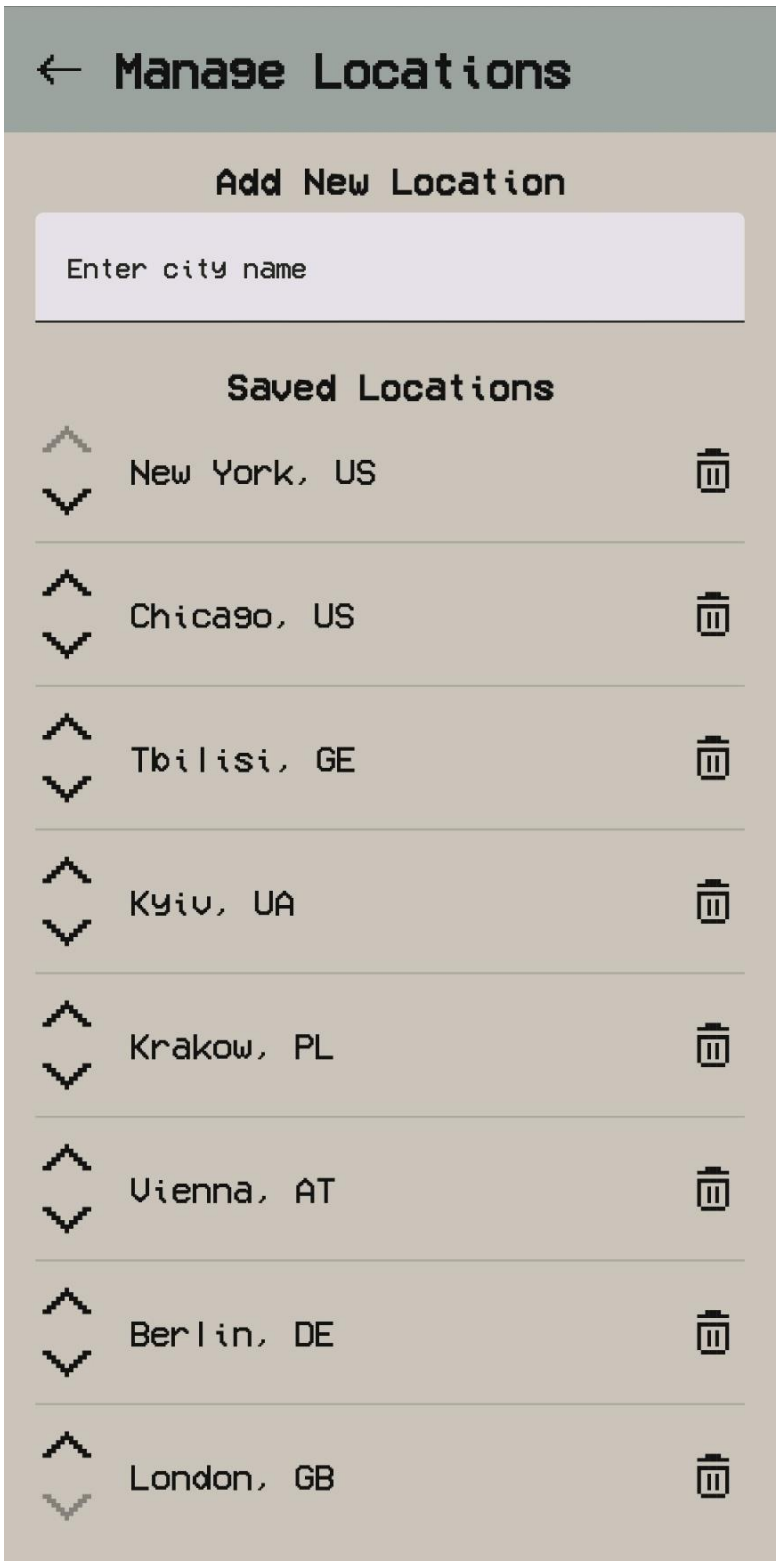


Рисунок А.3 – Екран управління локаціями (ManageCitiesScreen): пошук та список збережених міст



Рисунок А.4 – Екран налаштувань (SettingsScreen): вибір одиниць виміру та теми



Рисунок А.5 – Екран «Про застосунок» (AboutScreen)

ДОДАТОК Б

Фрагменти програмного коду

Б.1 Лістинг фрагменту класу MainViewModel.kt – управління станом пейджера та завантаження даних

```
// com.artemzarubin.weatherml.ui.mainscreen.MainViewModel.kt
// Фрагмент, що показує combine для pagerItems
// та метод onPageChanged або observeActivePagerItemToFetchWeather
val pagerItems: StateFlow<List<PagerItem>> =
    combine(
        _savedLocationsFromDbFlow,
        _geolocationPagerItemState,
        hasLocationPermissionFlow().distinctUntilChanged()
    ) { savedDbList, geoPageState, hasPermission ->
        val list = mutableListOf<PagerItem>()
        if (hasPermission) {
            list.add(geoPageState)
        }
        list.addAll(savedDbList.map { PagerItem.SavedPage(it) })
        list.distinctBy { it.id }
    }.stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000), emptyList())

fun onPageChanged(pageIndex: Int) {
    viewModelScope.launch {
        val items = pagerItems.value
        if (pageIndex >= 0 && pageIndex < items.size) {
            val selectedPagerItem = items[pageIndex]
            if (_currentPagerIndex.value != pageIndex) {
                _currentPagerIndex.value = pageIndex
            }
            if (selectedPagerItem is PagerItem.SavedPage) {
                weatherRepository.setActiveLocation(selectedPagerItem.location.id)
            } else if (selectedPagerItem is PagerItem.GeolocationPage) {
                weatherRepository.setActiveLocation(0)
            }
        }
    }
}
```

Б.2 Лістинг фрагменту класу WeatherModelInterpreterImpl.kt – підготовка ознак та отримання передбачення

```
// com.artemzarubin.weatherml.data.ml.WeatherModelInterpreterImpl.kt
// Фрагмент методу prepareAndScaleFeatures та getPrediction
override fun prepareAndScaleFeatures(currentWeather: CurrentWeather): FloatArray? {
    // ... (ключова логіка масштабування числових ознак
    //      та генерації циклічних/категоріальних) ...
    // Приклад:
    // val rawTemperature = currentWeather.temperatureCelsius.toFloat()
    // val scaledTemp = (rawTemperature - scalerXMean[0]) / scalerXScale[0]
    // featureMap[fullFeatureOrderFromTraining.find { it.contains("Temperature (C)") }!!] =
scaledTemp
    // ...
    return featuresArray
}

override fun getPrediction(input: ModelInput): ModelOutput? {
    // ... (виклик interpreter.run() та зворотне масштабування) ...
    // val predictedFeelsLikeActual = (predictedFeelsLikeScaled * scalerYScale) + scalerYMean
    // return ModelOutput(predictedFeelsLikeTemp = predictedFeelsLikeActual)
}
```