НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«АНАЛІЗ ПРОДУКТИВНОСТІ МОБІЛЬНИХ ЗАСТОСУНКІВ ПРИ ДЕКЛАРАТИВНОМУ ТА ІМПЕРАТИВНОМУ ПІДХОДІ ДО ПОБУДОВИ UI»

Виконав: здобувач 2 курсу
рівень: магістр
галузь знань 12 «Інформаційні технології»
спеціальність 122 «Комп’ютерні науки»
Земан Артур Олегович

Керівник: к.т.н., доцент
Манаков Сергій Юрійович

Рецензент: к.т.н., доцент
Гура Володимир Ігорович

Одеса – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ”
Факультет **кібербезпеки та інформаційних технологій**
Кафедра **інформаційних технологій**
Галузь знань: **12 Інформаційні технології**
Спеціальність: **122 “Комп’ютерні науки”**
Назва освітньої програми: **Комп’ютерні науки**

ЗАТВЕРДЖУЮ
Завідувач кафедри інформаційних технологій
доцент Наталія ЛОГІНОВА
_____“____”_____ 20__ року

ЗАВДАННЯ
на кваліфікаційну (магістерську) роботу
здобувачу Земану Артуру Олеговичу

1. Тема роботи: «Аналіз продуктивності мобільних застосунків при декларативному та імперативному підході до побудови UI»
   Керівник роботи: к.т.н, доцент Манаков С.Ю.
   затверджені наказом НУ «ОЮА» від «10» жовтня 2025 року № 3183-06.
2. Строк подання здобувачем роботи «25» листопада 2025 року.
3. Дата видачі завдання «02» червня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1.	Вибір теми, вивчення літературних джерел та складання плану роботи	15.09.2025	
2.	Підготовка першого розділу роботи та подання його керівнику	01.10.2025	
3.	Підготовка другого розділу роботи та подання його керівнику	14.10.2025	
4.	Підготовка третього розділу роботи та подання його керівнику	07.11.2025	
5.	Підготовка вступу та висновків	10.11.2025	
6.	Доопрацювання роботи з урахуванням зауважень керівника	24.11.2025	
7.	Подача електронного варіанту роботи на кафедру	25.11.2025	

Здобувач _____Артур ЗЕМАН_____
(підпис) (ім’я та прізвище)
Керівник роботи _____Сергій МАНАКОВ_____
(підпис) (ім’я та прізвище)

АНОТАЦІЯ

Земан А.О. «Аналіз продуктивності мобільних застосунків при декларативному та імперативному підході до побудови UI». – Master's qualification work in specialty 122 "Computer Science", educational program "Computer Science".

Кваліфікаційна робота викладена на 53 сторінках основного тексту і 75 сторінках загального тексту, містить 3 розділи, 10 рисунків, 13 таблиць, 24 використаних джерела.

Мета роботи: порівняльний аналіз продуктивності мобільних застосунків при використанні декларативного (Jetpack Compose) та імперативного (Android Views) підходів до побудови UI та розробка практичних рекомендацій для розробників.

Об'єкт дослідження: процес розробки користувацьких інтерфейсів для мобільних застосунків на платформі Android.

Предмет дослідження: показники продуктивності застосунків, реалізованих з використанням декларативного фреймворку Jetpack Compose та імперативної системи Android Views.

У роботі проведено комплексне експериментальне дослідження продуктивності мобільних застосунків, що включало розробку двох функціонально ідентичних реалізацій тестового застосунку з використанням Jetpack Compose та Android Views, проведення автоматизованого тестування продуктивності запуску за різних сценаріїв (гарячий, теплий та холодний запуск), аналіз розміру пакунків застосунків та статистичну обробку отриманих даних. На основі результатів розроблено практичні рекомендації щодо вибору технології UI залежно від специфіки проекту.

Ключові слова: користувацький інтерфейс, Jetpack Compose, Android Views, Macrobenchmark, профілювання, рекомпозиція, оптимізація продуктивності.

ABSTRACT

Zeman A.O. "Analysis of the performance of mobile applications using the declarative and imperative approach to building UI". – Master's thesis in the specialty 124 "System Analysis", educational program "Data Analysis and Security".

The thesis is presented on 53 pages of the main text and 75 pages of general text, contains 3 sections, 10 figures, 13 tables, 24 sources used.

Purpose of the work: comparative analysis of the performance of mobile applications using the declarative (Jetpack Compose) and imperative (Android Views) approaches to building UI and development of practical recommendations for developers.

Object of research: the process of developing user interfaces for mobile applications on the Android platform.

Subject of research: performance indicators of applications implemented using the declarative Jetpack Compose framework and the imperative Android Views system.

The paper conducted a comprehensive experimental study of mobile application performance, which included the development of two functionally identical implementations of the test application using Jetpack Compose and Android Views, automated testing of launch performance under different scenarios (hot, warm and cold launch), analysis of application package sizes and statistical processing of the obtained data. Based on the results, practical recommendations were developed for choosing UI technology depending on the specifics of the project.

Keywords: user interface, Jetpack Compose, Android Views, Macrobenchmark, profiling, recomposition, performance optimization.

ЗМІСТ

ВСТУП

Індустрія розробки мобільних застосунків переживає фундаментальний зсув у підходах до створення користувацьких інтерфейсів (UI). Протягом багатьох років домінував імперативний підхід, представлений в Android традиційною системою View, де розробник детально описує кроки для зміни UI у відповідь на події. Однак останнім часом все більшої популярності набуває декларативна парадигма, яскравим представником якої є фреймворк Jetpack Compose від Google. Цей підхід фокусується на описі кінцевого стану UI для певних даних, перекладаючи завдання синхронізації та оновлення на сам фреймворк.

Перехід на декларативні інструменти обіцяє значне спрощення коду, прискорення циклу розробки та підвищення продуктивності самих застосунків. Великі компанії, такі як Airbnb та Twitter, вже здійснили часткову або повну міграцію на Jetpack Compose, що підкреслює промислову значущість цієї технології. Попри теоретичні переваги та позитивні відгуки, бракує комплексних академічних досліджень, які б надавали кількісну оцінку продуктивності декларативного підходу в порівнянні з імперативним на платформі Android з використанням сучасних інструментів для бенчмаркінгу. Існуючі дослідження часто порівнюють кросплатформні фреймворки (Flutter, React Native) або не використовують спеціалізовані інструменти, як-от Macrobenchmark чи Perfetto, для глибокого аналізу.

Таким чином, **актуальність** даної роботи полягає у необхідності проведення всебічного експериментального аналізу продуктивності мобільних застосунків, побудованих за допомогою Jetpack Compose та Android Views, для емпіричного підтвердження або спростування заявлених переваг та надання розробникам науково обґрунтованих рекомендацій.

Метою роботи є порівняльний аналіз продуктивності мобільних застосунків при використанні декларативного (Jetpack Compose) та імпе ративного (Android Views) підходів до побудови UI та розробка практичних рекомендацій для розробників.

Для досягнення поставленої мети необхідно вирішити такі **завдання**:

1. Проаналізувати теоретичні основи, фундаментальні принципи та архітектурні відмінності імперативного та декларативного підходів до створення UI.
2. Розробити методику експериментального дослідження, обравши ключові метрики продуктивності та сучасні інструменти для їх вимірювання (Android Profiler, Perfetto, Macrobenchmark).
3. Спроєктувати та реалізувати тестовий мобільний застосунок з ідентичним функціоналом на базі Android Views та Jetpack Compose із забезпеченням повної функціональної та візуальної еквівалентності.
4. Провести експериментальне тестування продуктивності за такими показниками, як час запуску (гарячий, теплий, холодний), розмір APK-пакунку, кількість методів та класів, вплив профілю базової лінії.
5. Здійснити статистичний аналіз отриманих експериментальних даних для виявлення значущих відмінностей у продуктивності та оцінки стабільності результатів.
6. На основі результатів аналізу сформулювати практичні рекомендації щодо вибору технології UI та оптимізації продуктивності мобільних застосунків.

Об'єкт дослідження: Процес розробки користувацьких інтерфейсів для мобільних застосунків на платформі Android.

Предмет дослідження: Показники продуктивності застосунків, реалізованих з використанням декларативного фреймворку Jetpack Compose та імператив ної системи Android Views.

Для вирішення поставлених завдань у роботі використано наступні наукові **методи**:

– аналіз наукових публікацій, технічної документації та існуючих індустріальних кейсів для формування теоретичної бази дослідження;

- лабораторний експеримент для збору даних про продуктивність, що включає методи профілювання (CPU, Memory, Energy Profiler) та бенчмаркінгу (Macrobenchmark);
- методи описової статистики для узагальнення даних (мінімум, медіана, максимум, перцентилі P90 та P95, стандартне відхилення) для визначення характеристик розподілу показників продуктивності.

Наукова новизна отриманих результатів полягає у тому, що вперше проведено комплексне порівняння продуктивності Jetpack Compose та Android Views з використанням сучасних спеціалізованих інструментів (Perfetto, Macrobenchmark), що дозволило емпірично підтвердити теоретичні переваги декларативного підходу для сценаріїв повторного запуску, виявити суттєві відмінності у продуктивності холодного запуску та розмірі застосунків, а також розробити методику тестування продуктивності UI-каркасів на платформі Android.

Практична значущість роботи полягає в розробці набору науково обґрунтованих рекомендацій для Android-розробників, які можуть бути використані як система критеріїв для вибору технології UI залежно від специфіки проекту (нові проекти vs існуючі, критичність розміру APK, важливість швидкого першого запуску, складність інтерфейсу), а також як настанови щодо обов'язкового використання профілю базової лінії для декларативних застосунків та стратегій гібридної міграції.

Робота пройшла апробацію на науковій конференції [24].

1 ТЕОРЕТИЧНІ ОСНОВИ ДЕКЛАРАТИВНОГО ТА ІМПЕРАТИВНОГО ПІДХОДІВ ДО ПОБУДОВИ UI

1.1 Фундаментальні принципи програмування користувацьких інтерфейсів

Програмування користувацьких інтерфейсів протягом останнього десятиліття зазнало фундаментальних змін у своїх парадигмальних основах. Перехід від імперативного до декларативного підходу становить не просто технологічну еволюцію, а концептуальну трансформацію способу мислення розробників про створення користувацьких інтерфейсів [1].

Імперативний підхід до програмування UI базується на принципі «як», коли розробник експліцитно описує кожен крок для досягнення бажаного результату. У цій парадигмі програміст безпосередньо керує потоком виконання програми та власноруч управляє змінами стану інтерфейсу. Класичним прикладом імперативного підходу є традиційна Android View система, де розробник повинен явно викликати методи на кшталт setText(), setVisibility() для оновлення елементів інтерфейсу [2].

Декларативний підхід кардинально змінює цю філософію, зосереджуючись на принципі «що». Розробник описує бажаний стан інтерфейсу як функцію від поточного стану даних, а система автоматично обчислює необхідні зміни для досягнення цього стану. Як зазначають дослідники Google LLC у офіційній документації Flutter: «У декларативному стилі конфігурації виглядів є незмінними та представляють лише легковагі «креслення». Для зміни UI віджет ініціює перебудову самого себе» [3].

Основна відмінність між підходами проявляється у способах управління станом інтерфейсу. Імперативні системи традиційно покладаються на мутабельний стан (mutable state), де об'єкти інтерфейсу можуть безпосередньо змінювати свої

властивості протягом життєвого циклу. Це створює складності у відстеженні змін та може призводити до проблем синхронізації стану [4].

Декларативні системи, навпаки, використовують принципи іммутабельності стану. Кожна зміна даних призводить до створення нового стану, а не модифікації існуючого. Це забезпечує передбачуваність поведінки системи та спрощує відстеження змін. Rahman у своєму технічному аналізі підкреслює, що "незмінність у управлінні станом дозволяє здійснювати оптимізацію через перевірки рівності посилань та автоматичні тригери перерендерингу" [5].

Традиційні імперативні системи вимагають від розробника ретельного управління життєвим циклом компонентів інтерфейсу. Кожен етап від створення до знищення елемента потребує явного програмного контролю, включаючи ініціалізацію, оновлення стану, обробку подій та очищення ресурсів.

Декларативні системи впроваджують концепцію автоматичної синхронізації, де фреймворк бере на себе відповідальність за підтримку відповідності між станом даних та відображенням інтерфейсу. У теоретичному аналізі [6] реактивного UI пропонується модель декларативного інтерфейсу як «конвеєра трансформацій дерев» з фундаментальним припущенням, що дерева моделюють дані, які трансформуються. Табл. 1.1 відображає відмінності між підходами.

Таблиця 1.1 – Порівняння підходів до побудови UI

Характеристика	**Імперативний підхід**	**Декларативний підхід**
Фокус програмування	"Як" досягти результату	"Що" має бути результатом
Управління станом	Мутабельні об'єкти	Іммутабельні структури
Синхронізація UI	Ручна через методи об'єктів	Автоматична через спостереження стану
Життєвий цикл	Явне управління циклом	Автоматичне керування фреймворком
Складність коду	Висока через детальне управління	Нижча завдяки абстракції

Теоретичною основою сучасних декларативних фреймворків є принципи реактивного програмування, сформульовані у [4]. Ці принципи включають: відзивність, стійкість, еластичність та орієнтованість на повідомлення.

Відзивність забезпечує швидку реакцію системи на зміни користувацького вводу. Стійкість гарантує функціонування системи навіть за наявності помилок. Еластичність дозволяє адаптацію до змінних навантажень. Орієнтованість на повідомлення створює слабко зв'язану архітектуру, де компоненти взаємодіють через асинхронні повідомлення.

1.2 Архітектурні відмінності Android View System та Jetpack Compose

Архітектурні відмінності між традиційною Android View System та сучасним Jetpack Compose представляють класичний приклад еволюції від імперативного до декларативного підходу у мобільній розробці.

Система Android View базується на XML-орієнтованій архітектурі з складним процесом інфляції макетів. Процес створення інтерфейсу включає кілька етапів: парсинг XML-файлів за допомогою XmlPullParser, стиснення макетів інструментом Android Asset Packaging Tool (AAPT) у бінарний формат, та подальшу інстанціацію View-об'єктів через рефлексію [7].

Технічна реалізація інфляції відбувається через ланцюжок: LayoutInflater.inflate() – XmlPullParser – View instantiation – AttributeSet application – View hierarchy creation. Кожен XML-тег відповідає інстанціації конкретного класу (наприклад, <Button> – клас Button), а XML-атрибути зчитуються з TypedArray та застосовуються через параметри конструктора або сетерні методи [7].

View System тісно пов’язана з життєвим циклом Android-компонентів. У методі onCreate() відбувається виклик setContentView() для інфляції макету та його прикріплення до Activity. Стадії onStart() та onResume() відповідають за ініціалізацію видимості та інтерактивності елементів. Етапи onPause()/onStop()

вимагають ручного управління ресурсами, а onDestroy() потребує експліцитного очищення для запобігання витокам пам'яті [7].

Збереження стану здійснюється через механізми onSaveInstanceState() та onRestoreInstanceState() з використанням Bundle-серіалізації, що створює додаткову складність для розробника.

Імперативний характер View System вимагає явного управління станом через безпосередні виклики сетер-методів (setText(), setVisibility()). Оновлення інтерфейсу відбувається через OnClickListener, OnTextChangedListener – що потребує ручної координації між змінами даних та оновленнями UI [7].

Jetpack Compose впроваджує радикально іншу архітектуру, засновану на функціональному програмуванні та принципах композиції. Основними властивостями @Composable функцій є ідемпотентність (однакові вхідні дані завжди дають однаковий результат), відсутність побічних ефектів, можливість перезапуску та швидке виконання [8].

Серцем Jetpack Compose є складна система управління композицією, заснована на архітектурі Gap Buffer – структурі даних, що широко використовується у текстових редакторах. Slot Table являє собою плоский масив з невикористаним простором (gap) для ефективного виконання операцій вставки та видалення за константний час (за винятком переміщення gap, яке має складність $O(n)$) [8].

Технічна реалізація: SlotTable – Gap Buffer – Efficient node management – Positional memoization. Об'єкт Composer керує операціями зі slot table, ін'єктується як неявний параметр до всіх @Composable функцій, та координує взаємодію між Compose-кодом та системою виконання [8].

Compose також впроваджує унікальний механізм позиційної мемоізації, де кешування базується на позиції виклику у вихідному коді, а не на глобальній ідентичності функції. Кожен виклик композабельної функції отримує унікальну ідентичність на основі місця у коді, що дозволяє ефективно пропускати незмінені @Composable під час рекомпозиції [8].

Процес рекомпозиції включає: виявлення зміни стану – інвалідацію scope – розумну рекомпозицію (повторне виконання лише функцій зі зміненими параметрами) – селективне оновлення дерева композиції.

Система Snapshot у Compose реалізує Multiversion Concurrency Control (MVCC) для забезпечення консистентності стану. Автоматична інвалідація та тригери рекомпозиції, безпечне спостереження стану через ізоляцію снепшотів, та глобальне управління снепшотами для координованих оновлень створюють потужну систему реактивного управління станом [8].

Табл. 1.2 порівнює архітектурні характеристики обох підходів.

Таблиця 1.2 – Порівняння архітектурних характеристик підходів до побудови UI

Компонент	**Android View System**	**Jetpack Compose**
Підхід до створення UI	XML + Java/Kotlin	Kotlin функції
Структура даних	Ієрархія View об'єктів	Slot Table (Gap Buffer)
Управління станом	Ручні сетер-методи	Автоматичне спостереження
Оптимізація	Ручна (ViewHolder pattern)	Автоматична (позиційна мемоізація)
Життєвий цикл	Прив'язка до Activity/Fragment	Композиційний цикл
Споживання пам'яті	Високе (постійні об'єкти)	Нижче (функціональний підхід)

1.3 Теоретичні основи компіляції та рендерингу UI

Процеси компіляції та рендерингу користувацьких інтерфейсів представляють критично важливі аспекти, що визначають продуктивність та ефективність UI систем. Сучасні фреймворки впроваджують складні механізми оптимізації на етапах компіляції та виконання.

Jetpack Compose Compiler Plugin інтегрується з компілятором Kotlin через інтерфейс ComponentRegistrar, що служить точкою входу для плагінів компілятора.

Архітектура плагіна включає: Plugin (точку входу Gradle API), Subplugin (інтерфейс між Gradle та Kotlin API), CommandLineProcessor (обробку аргументів kotlinc), ComponentRegistrar (реєстрацію розширень) та Extensions (перехоплення фаз компіляції) [9].

Трансформація @Composable функцій відбувається через складний процес IR lowering, де конструкції високого рівня перетворюються на представлення нижчого рівня. Компілятор автоматично впроваджує параметри $composer та $changed до кожної компоуз функції та обертає їх у startRestartGroup()/endRestartGroup() для відстеження рекомпозиції [9].

Компілятор генерує оптимізований код для відстеження змін стану через параметр $changed, який використовує бітові операції для ефективного визначення змінених параметрів. Intrinsic Remember дозволяє компілятору вбудовувати виклики remember та замінювати перевірки .equals() на ефективні порівняння цілих чисел для стабільних параметрів [9].

Режим Strong Skipping забезпечує агресивне пропускання рекомпозиції, коли параметри доведено стабільними. Inference стабільності автоматично визначає незмінність типів для оптимізації рішень про рекомпозицію [9].

Традиційна система View використовує LayoutInflater для триступеневого процесу створення кожного елемента: XML-парсинг через XmlPullParser, обробка атрибутів (включаючи конверсію одиниць dp/sp та розв'язання ресурсів), та інстанціація View об'єктів через рефлексію з пошуком імені класу [9].

Попередня обробка відбувається на етапі збірки для оптимізації XML-парсингу, але рефлексійне створення елементів залишається дорогою операцією, що впливає на продуктивність інфляції макетів [9].

ViewBinding генерує класи прив'язки на етапі компіляції, що усувають виклики findViewById(). Згенеровані класи містять типобезпечні посилання на всі елементи з атрибутом ID та надають статичні методи inflate() для створення екземплярів [9].

DataBinding розширює ViewBinding, додаючи оцінку виразів та двонаправну прив'язку даних. Процес збірки включає: ProcessMethodAdapters (сканування

@BindingAdapter анотацій), ProcessExpressions (генерацію коду оцінки виразів), ProcessBindable (створення BR класу з ID властивостей), та LayoutBinderWriter (генерацію фінальних класів прив'язки) [9].

Щодо рендерингу, Android Framework використовує триступеневий цикл: фази Measure, Layout та Draw, кожна з яких має специфічні завдання та впливає на загальну продуктивність системи.

1. Фаза Measure визначає розміри елементів на основі обмежень та контенту. Система MeasureSpec використовує три режими (EXACTLY, AT_MOST, UNSPECIFIED) для визначення обмежень розміру. Складні макети можуть ініціювати множинні проходи вимірювання, що створює складність $O(n^2)$ у випадку вкладених RelativeLayout [9].
2. Фаза Layout позиціонує елементи на основі виміряних розмірів через обхід зверху вниз, призначаючи абсолютні координати. Батьківські елементи позиціонують дочірні на основі правил макету та розв'язання обмежень [9].
3. Фаза Draw генерує Display Lists з кешованими командами малювання та використовує апаратне прискорення через GPU для рендерингу. Управління шарами та композитинг забезпечують ефекти та складні візуальні трансформації [9].

Jetpack Compose використовує Skia як графічний рушій, що забезпечує абстракцію Canvas API (маппінг операцій малювання Compose на операції Skia), пакетну обробку (групування команд малювання для ефективної подачі на GPU), та кросплатформну консистентність рендерингу [9].

Клас Choreographer координує синхронізацію кадрів з оновленням дисплея через реєстрацію frame callbacks та обробку VSync сигналів. Математика синхронізації кадрів: 60Hz дисплеї (16.67мс на кадр), 90Hz дисплеї (11.11мс), 120Hz дисплеї (8.33мс) [9].

Управління глибиною конвеєра включає Render Ahead (затримка презентації кадрів для надання більшого часу рендерингу), Triple Buffering (підтримка плавної подачі кадрів під час варіацій навантаження GPU), та Presentation Timestamps (EGL_ANDROID_presentation_time для правильної синхронізації кадрів) [9].

1.4 Аналіз існуючих досліджень

Аналіз існуючих досліджень у сфері декларативного та імперативного програмування користувацьких інтерфейсів виявляє як значні досягнення, так і критичні прогалини у теоретичному розумінні цих парадигм.

Zhou та колеги у 2024 році представили дослідження автоматичної генерації декларативного UI коду, досягнувши 96.8% покриття графа переходів сторінок та 98% успішності компіляції з React Native [1]. Це дослідження демонструє практичну ефективність декларативної парадигми та можливості автоматизації процесу розробки інтерфейсів.

Jost та Taneski у 2025 році провели дослідження кросплатформних фреймворків мобільної розробки та виявивили статистично значущо вищу довгострокову задоволеність розробників Flutter (68.1%) порівняно з React Native (57.2%). Дослідження показує перехід від WebView-орієнтованих до нативно-компільованих декларативних фреймворків [2].

Гарвардський університет у 2024 році опублікував дослідження Samancioglu, що порівнює реактивні фреймворки програмування у розробці мобільних застосунків. Дослідження виявило значні відмінності у продуктивності між фреймворками, демонструючи переваги Combine над RxSwift та Coroutines над RxJava у різноманітних тестових сценаріях [10].

Промислові дослідження Google показують такі метрики міграції: збільшення розміру APK на 782 КБ при додаванні Compose, але зменшення на 68 КБ при повній міграції. Час збірки зростає спочатку на 100 мс, але знижується на 57 мс після завершення міграції [11].

Команда Play Store виявила, що написання UI вимагає «значно менше коду, іноді до 50%» та суттєво підвищує продуктивність і підтримуваність [12].

Однак, аналіз літератури виявляє також кілька критичних прогалин у теоретичному розумінні UI парадигм:

- Бракує емпіричних досліджень, що підтверджують заявлені переваги декларативного UI. Недостатній кількісний аналіз покращень

продуктивності розробників та відсутні довготривалі дослідження щодо зменшення накладних витрат на підтримку.

- Обмежені академічні дослідження продуктивності кросплатформних фреймворків. Недостатні дослідження продуктивності UI взаємодії та відсутній аналіз реальних застосунків (більшість досліджень використовує прості застосунки).
- Мінімальні якісні дослідження, орієнтовані на користувача, щодо відмінностей парадигм. Відсутні систематичні дослідження різниць когнітивного навантаження та недостатні дослідження впливу кривих навчання для розробників.

Аналіз української наукової сфери виявляє обмежену кількість досліджень, специфічно присвячених еволюції парадигм мобільного UI. Провідні заклади, такі як КПІ ім. Ігоря Сікорського, ХНУРЕ, ЛНУ ім. Івана Франка та УКУ, мають потужні програми мобільної розробки з промисловими зв'язками, але потребують додаткових досліджень у сфері теоретичних основ UI парадигм [11].

Українська IT-індустрія, що налічує понад 300,000 технічних спеціалістів та 1600+ компаній розробки програмного забезпечення, представляє унікальну можливість для проведення практично релевантних досліджень, що можуть вплинути як на академічне розуміння, так і на промислову практику [13].

1.5 Висновки до розділу 1

Теоретичний аналіз декларативного та імперативного підходів до побудови користувацьких інтерфейсів виявляє фундаментальну трансформацію парадигм програмування UI. Перехід від імперативного принципу «як» до декларативного принципу «що» представляє не просто технологічну еволюцію, а концептуальну революцію у способі мислення про створення інтерфейсів.

Архітектурні відмінності між Android View System та Jetpack Compose демонструють практичну реалізацію цієї теоретичної трансформації. Традиційна

XML-орієнтована архітектура з ручним управлінням станом поступається місцем функціональній композиції з автоматичною синхронізацією та позиційною мемоізацією.

Аналіз процесів компіляції та рендерингу розкриває складність сучасних UI систем, від трансформації @Composable функцій компілятором до триступеневого циклу рендерингу з апаратним прискоренням. Ці технічні деталі підкреслюють важливість розуміння глибинних механізмів для ефективної розробки користувацьких інтерфейсів.

Огляд існуючих досліджень вказує на значні можливості для майбутньої роботи, особливо у контексті української IT-індустрії. Виявлені прогалини у теоретичних фреймворках, емпіричній валідації та кросплатформних дослідженнях створюють сприятливе середовище для внеску у глобальне розуміння еволюції UI парадигм.

2 МЕТОДИКА ЕКСПЕРИМЕНТАЛЬНОГО ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ

2.1 Інструменти профілювання та аналізу продуктивності

Дослідження продуктивності користувацьких інтерфейсів Android вимагає використання спеціалізованих інструментів, які дозволяють отримати детальне уявлення про поведінку застосунку на різних рівнях системної архітектури. Сучасна екосистема Android пропонує комплексний набір засобів профілювання, кожен з яких орієнтований на специфічні аспекти продуктивності.

Android Profiler, інтегрований у Android Studio, являє собою багатофункціональний інструментарій для моніторингу критичних системних ресурсів [14]. Цей набір інструментів забезпечує моніторинг процесора, пам'яті, мережевої активності та енергоспоживання в режимі реального часу. CPU Profiler дозволяє відстежувати використання процесорних ресурсів з точністю до окремих потоків виконання, надаючи кольорову індикацію станів потоків та візуалізацію часу виконання методів через Call Chart та Flame Chart. Для дослідження продуктивності UI особливо цінною є можливість запису системних трасувань, які дозволяють виявити проблеми рендерингу через аналіз Display Timeline. Інструмент автоматично виявляє кадри, час рендерингу яких перевищує критичний поріг у 16 мілісекунд для досягнення плавної частоти оновлення 60 кадрів на секунду (на стандартних дисплеях) або 90-120 FPS на високочастотних дисплеях, що дозволяє ідентифікувати так звані jank-події – видимі ривки в інтерфейсі користувача [25].

Memory Profiler забезпечує комплексний аналіз використання оперативної пам'яті, відстежуючи Java heap, Native heap, графічну пам'ять, стек та сегменти коду [14]. Для порівняння декларативного та імперативного підходів до побудови UI критично важливим є моніторинг Graphics heap, оскільки різні архітектури рендерингу мають відмінні патерни споживання графічної пам'яті. Інструмент

дозволяє захоплювати знімки купи для детального аналізу структури пам'яті та автоматично виявляє типові витоки пам'яті, пов'язані з неправильним життєвим циклом Activity та Fragment компонентів. Моніторинг подій збирача сміття особливо важливий для оцінки навантаження на систему управління пам'яттю, оскільки частота та тривалість пауз GC безпосередньо впливають на плавність роботи користувацького інтерфейсу. Energy Profiler надає можливість аналізу енергоспоживання, що є критичним фактором для мобільних застосунків, дозволяючи виявити надмірне споживання енергії через інтенсивний рендеринг чи складні анімації [21].

Android GPU Inspector представляє спеціалізований інструмент для глибокого профілювання графічної підсистеми на рівні GPU [14]. Цей інструмент підтримує провідні графічні процесори мобільних платформ, включаючи Qualcomm Adreno, Arm Mali та Imagination PowerVR, працюючи з Vulkan та OpenGL ES API. Функціональність системного профілювання забезпечує візуалізацію кореляції між активністю центрального та графічного процесорів, дозволяючи ідентифікувати вузькі місця в графічному конвеєрі. Інструмент надає доступ до лічильників GPU, які відображають утилізацію графічного процесора, час обробки фрагментів та вершин, пропускну здатність текстурної та буферної пам'яті, а також коефіцієнт перевищення. Покадрове профілювання дозволяє захопити детальну послідовність викликів рендерингу для окремого кадру, проінспектувати стани рендерера, шейдери та текстури. Для порівняння Jetpack Compose та Android Views цей інструмент є незамінним, оскільки дозволяє оцінити ефективність різних підходів до рендерингу на рівні графічного апаратного забезпечення. Проте слід враховувати, що використання GPU Inspector вимагає налагоджувальної збірки застосунку, підтримки Android 11 або новіших версій, та привносить помітні накладні витрати у продуктивність під час профілювання.

Perfetto являє собою платформу трасування нового покоління для Android, Linux та Chrome, що замінила застарілий інструмент Systrace починаючи з Android 10 [22]. На відміну від свого попередника, Perfetto використовує бінарний формат protocol buffer, що дозволяє проводити трасування необмеженої тривалості з

мінімальним впливом на продуктивність системи. Perfetto забезпечує системне трасування, яке охоплює не лише рівень застосунку, а й ядро операційної системи через інтеграцію з ftrace. Для дослідження продуктивності UI критично важливими є категорії трасування graphics, view, WindowManager та ActivityManager, які дозволяють відстежити повний шлях від обробки події вводу до відображення результату на екрані. Інструмент надає детальну інформацію про планування потоків через події планувальника, відстежує зміни частоти процесора та дозволяє аналізувати пропущені vsync події, що є прямим індикатором проблем із плавністю інтерфейсу. Унікальною особливістю Perfetto є можливість SQL-аналізу зібраних трас, що дозволяє формулювати складні запити для виявлення специфічних патернів поведінки. Веб-інтерфейс Perfetto UI забезпечує потужну візуалізацію часової шкали подій без необхідності встановлення додаткового програмного забезпечення, що суттєво спрощує процес аналізу [22].

JankStats API, частина бібліотеки androidx.metrics, призначена для виявлення та звітування про ривки інтерфейсу в режимі реального часу на пристроях кінцевих користувачів [14]. На відміну від інструментів розробки, JankStats орієнтована на моніторинг у робочому середовищі і характеризується мінімальним впливом на продуктивність. Бібліотека автоматично визначає jank-кадри через евристики, які за замовчуванням класифікують кадр як janky, якщо його рендеринг займає вдвічі більше часу, ніж дозволяє частота оновлення дисплея. Важливою особливістю є підтримка різних частот оновлення екрану, включаючи 60Hz, 90Hz та 120Hz дисплеї, з автоматичним адаптуванням порогових значень. UI State API дозволяє асоціювати контекстну інформацію про стан застосунку з кожною зафіксованою jank-подією, що суттєво спрощує аналіз причин проблем продуктивності. Інтеграція JankStats з Jetpack Compose здійснюється через composable функції, що дозволяє безшовно моніторити продуктивність декларативних UI компонентів. Для порівняльних досліджень продуктивності JankStats надає можливість збирати статистику про частку janky-кадрів, розподіл часу рендерингу за перцентилями та найгірші випадки, що дозволяє об'єктивно оцінити якість користувацького досвіду в реальних умовах експлуатації.

Комплексне використання описаних інструментів дозволяє побудувати багаторівневе профілювання [14, 22]. На етапі виявлення проблем у робочому середовищі JankStats забезпечує збір метрик від реальних користувачів, після чого Android Studio Profiler дозволяє відтворити проблему в середовищі розробки. Детальний аналіз причин проблеми здійснюється через CPU Profiler для виявлення повільних методів та Memory Profiler для виявлення витоків пам'яті, тоді як Perfetto надає системний контекст через трасування ядра. У випадках проблем із графічною підсистемою GPU Inspector забезпечує необхідну глибину аналізу на рівні графічного процесора.

2.2 Методики бенчмаркінгу

Систематичне порівняння продуктивності різних підходів до побудови користувацьких інтерфейсів вимагає чіткої методики бенчмаркінгу, яка б забезпечувала відтворюваність результатів та їх статистичну значущість. Android Jetpack Benchmark бібліотека надає два принципово різні підходи до вимірювання продуктивності, кожен з яких орієнтований на специфічні аспекти поведінки застосунку [15].

Macrobenchmark призначений для наскрізного тестування користувацьких сценаріїв і взаємодій, виконуючись у окремому процесі від тестованого застосунку [15]. Ця ізоляція є фундаментальною для точності вимірювань, оскільки дозволяє перезапускати застосунок між ітераціями тестування, компілювати код з DEX у машинний код та вимірювати реальний час запуску без впливу інструментації тестового фреймворку. Для дослідження продуктивності Jetpack Compose порівняно з Android Views, Macrobenchmark надає можливість вимірювання трьох критичних сценаріїв. Тестування запуску застосунку диференціює холодний старт, коли процес створюється з нуля, теплий старт з існуючим процесом але зруйнованою Activity, та гарячий старт з Activity у пам'яті. Метрика timeToInitialDisplay вимірює час до першого відображеного кадру, тоді як timeToFullDisplayMs відображає час до повної готовності інтерфейсу, що є більш

релевантною метрикою з точки зору користувацького досвіду. Бенчмарки прокручування оцінюють продуктивність рендерингу під час динамічної взаємодії, вимірюючи час обробки кадру на CPU та затримку відносно дедлайну vsync. Тестування анімацій дозволяє оцінити здатність системи підтримувати постійну частоту кадрів під час складних трансформацій інтерфейсу. Критично важливим аспектом Macrobenchmark є управління режимами компіляції коду [15, 27]. CompilationMode.None моделює найгірший сценарій, коли весь код виконується через JIT компілятор, що відповідає першому запуску застосунку після встановлення. CompilationMode.Partial дозволяє тестувати з частковою компіляцією, моделюючи стан після кількох запусків. CompilationMode.Full виконує повну завчасну компіляцію, що відповідає оптимальному сценарію. CompilationMode.DEFAULT використовує Baseline Profile якщо такий доступний, що є рекомендованим режимом для тестування production збірок. Дослідження команди Mercari демонструють, що використання правильно сконфігурованих Baseline Profile'ів може призвести до подвоєння швидкості рендерингу для інтерфейсів заснованих на Compose [27].

Microbenchmark, на відміну від Macrobenchmark, орієнтований на вимірювання продуктивності окремих функцій та методів на рівні процесора [15]. Цей підхід виконує тестовий код у циклі з великою кількістю ітерацій для досягнення статистично значущих результатів з мінімальною похибкою. Для коректності вимірювань Microbenchmark автоматично виконує фазу розігріву, яка дозволяє JIT компілятору оптимізувати гарячі шляхи коду перед початком вимірювань [16, 23]. Проте застосовність Microbenchmark обмежена кодом, який поводиться детерміністично при повторних викликах та не залежить від кешування операційної системи. Для порівняння Compose та Views, Microbenchmark придатний для тестування окремих аспектів, таких як час розгортання Layout порівняно з часом початкової композиції, або продуктивність прив'язки даних до одного елемента списку.

Принципова відмінність між синтетичним та реальним тестуванням визначає дві методології оцінки продуктивності [24]. Синтетичне тестування являє собою

проактивний підхід, який симулює користувацьку поведінку за допомогою скриптів у контрольованому середовищі. Цей підхід дозволяє проводити тестування до релізу в production, забезпечує контрольовані умови для порівняння різних реалізацій та надає консистентні базові метрики. Macrobenchmark є прикладом синтетичного тестування, оскільки використовує UIAutomator для автоматизації взаємодій з інтерфейсом за заздалегідь визначеним сценарієм [15, 29]. Проте синтетичні тести не відображають повної різноманітності реальної поведінки користувачів, можуть пропустити граничні випадки та не враховують всіх варіацій мережевих умов та стану пристрою.

Real User Monitoring представляє пасивний підхід, який збирає дані про реальні взаємодії користувачів з застосунком у робочому середовищі [21, 25]. Цей підхід надає автентичні дані про продуктивність на різноманітних пристроях, у різних географічних локаціях та мережевих умовах, відображаючи всі можливі граничні випадки та поведінкові патерни. Firebase Performance Monitoring та Google Play Vitals є інструментами для RUM, які автоматично збирають метрики запуску застосунку, час рендерингу екранів, мережеві затримки та частоту аварійних завершень. JankStats API інтегрується в production код для збору детальної статистики про плавність інтерфейсу на пристроях користувачів [14]. Обмеженням RUM є необхідність достатньої кількості користувачів для статистично значущої вибірки, наявність шуму у даних через неконтрольовані зовнішні фактори та затримка в отриманні зворотного зв'язку після релізу змін.

Оптимальна методика передбачає комбінування обох підходів у циклічному процесі [24]. Синтетичне тестування через Macrobenchmark забезпечує передрелізну валідацію та встановлення базових метрик. Після релізу RUM надає дані про реальну продуктивність та користувацький досвід. Аналіз зібраних даних виявляє проблеми, які потім відтворюються в контрольованому середовищі для детального дослідження та оптимізації. Покращення валідуються через синтетичні тести перед наступним релізом, замикаючи цикл безперервного покращення продуктивності.

A/B тестування для порівняння продуктивності являє собою методологію паралельного розгортання двох варіантів реалізації для різних груп користувачів з подальшим статистичним порівнянням метрик продуктивності [24]. Firebase A/B Testing у поєднанні з Remote Config дозволяє динамічно контролювати, яка група користувачів отримує Compose-based реалізацію екрану, а яка традиційну View-версію, без необхідності випуску окремих збірок застосунку. Платформа автоматично розподіляє користувачів між варіантами, збирає метрики продуктивності через Firebase Performance Monitoring та обчислює статистичну значущість відмінностей. Поетапне розгортання через Google Play Console може функціонувати як природний A/B тест, де поступове збільшення відсотка користувачів на новій версії дозволяє порівнювати метрики між версіями та швидко реагувати на проблеми продуктивності. Критично важливим для коректності A/B тестів є принцип зміни лише однієї змінної за раз, визначення чітких метрик успіху до початку експерименту, забезпечення достатнього розміру вибірки для статистичної потужності та тривалість тестування щонайменше тиждень для врахування варіативності поведінки користувачів у різні дні тижня.

Дослідження Google щодо порівняння метрик Compose та Views на прикладі застосунку Sunflower демонструють практичне застосування цих методологій [28]. Додавання Compose до існуючого View-застосунку збільшило розмір APK на 782 КБ та час збірки на 100 мілісекунд, тоді як повна міграція на Compose зменшила розмір APK на 68 КБ та час збірки на 57 мілісекунд порівняно з гібридною реалізацією. Ці дані підкреслюють важливість комплексного підходу до бенчмаркінгу, який враховує не лише продуктивність під час виконання, а й метрики збірки та розміру артефактів.

2.3 Ключові метрики продуктивності

Комплексна оцінка продуктивності користувацьких інтерфейсів Android вимагає систематичного вимірювання метрик у чотирьох критичних категоріях,

кожна з яких відображає різні аспекти користувацького досвіду та ефективності системи [21, 25].

Метрики продуктивності під час виконання являють собою найбільш безпосередньо відчутні користувачем показники якості інтерфейсу. Frames Per Second вимірює частоту оновлення кадрів інтерфейсу, де стандартний поріг у 60 FPS відповідає періоду кадру 16.67 мілісекунд, який є мінімально необхідним для плавного сприйняття анімацій людським оком [25]. Сучасні високочастотні дисплеї з частотою 90 Hz та 120 Hz вимагають ще коротших періодів кадру у 11.11 та 8.33 мілісекунд відповідно, що підвищує вимоги до ефективності рендерингу. Час кадру надає більш деталізовану інформацію порівняно з FPS, розділяючи обробку на UI thread та RenderThread фази, що дозволяє локалізувати вузькі місця в конвеєрі рендерингу [14]. Android Vitals класифікує кадри на повільні, які займають від 16 до 700 мілісекунд, та заморожені, що перевищують 700 мілісекунд, де кожна категорія має різний вплив на сприйняття користувача. Статистика показує, що відсоток сесій з повільним рендерингом нижче одного відсотка вважається відмінним показником, від одного до п'яти відсотків прийнятним, тоді як перевищення п'яти відсотків вимагає термінової оптимізації. Виявлення jank являє собою критично важливу метрику, оскільки дослідження поведінки користувачів демонструють, що 49% очікують часу запуску менше двох секунд, а 80% готові дати застосунку максимум три спроби перед видаленням [25]. Jank виявляється через аналіз пропущених vsync подій, коли рендеринг кадру не встигає завершитися до наступного циклу оновлення дисплея. Perfetto FrameTimeline надає детальну класифікацію причин jank, включаючи App Deadline Missed, коли застосунок не встиг підготувати кадр, Buffer Stuffing через надмірну кількість кадрів у черзі, та проблеми SurfaceFlinger на рівні композитора вікон [22].

Час запуску застосунку диференціюється на три сценарії з різними пороговими значеннями [15, 33]. Холодний старт з повним створенням процесу вважається відмінним при часі менше 500 мілісекунд, прийнятним до 5 секунд. Теплий старт з існуючим процесом має більш жорсткі вимоги з відмінним порогом

200 мілісекунд та прийнятним 2 секунди. Гарячий старт з Activity у пам'яті повинен бути практично миттєвим з відмінним порогом 100 мілісекунд.

Метрики пам'яті відображують ефективність управління ресурсами та довгострокову стабільність застосунку [14, 21]. Використання купи розділяється на купу Java/Kotlin для об'єктів керованого коду, Native heap для нативних компонентів, Graphics memory для текстур та буферів рендерингу, Code segment для скомпільованого коду та Stack для локальних змінних. Типові ліміти пам'яті варіюються від 128-192 МБ на low-end пристроях до понад 512 МБ на high-end моделях, з типовим споживанням застосунку в діапазоні 50-150 МБ. Перевищення цих лімітів призводить до примусового завершення процесу системою, що критично впливає на користувацький досвід. Навантаження на збирач сміття (Garbage Collector, GC) вимірюється через частоту подій GC та тривалість пауз, де цільовими показниками є частота менше одного GC на секунду, тривалість конкурентних пауз GC менше 5 мілісекунд та швидкість алокації менше 10 МБ на секунду. Ці паузи безпосередньо впливають на плавність інтерфейсу, оскільки під час фаз stop-the-world («зупини світ») GC виконання потоку UI thread призупиняється, що може призвести до пропущених кадрів. Виявлення витоків пам'яті є критичним для довгострокової стабільності, оскільки неправильне управління життєвим циклом об'єктів призводить до прогресуючого уповільнення та подальших аварійних завершень через брак пам'яті (OutOfMemory). Типові джерела витоків включають утримання посилань на Context через static поля, незареєстровані слухачі, довгоживучі об'єкти Thread та Handler.

Метрики продуктивності збірки безпосередньо впливають на продуктивність розробки та вартість інфраструктури CI/CD [19, 20]. Час компіляції для проєкту середнього розміру з 50 модулями вважається швидким у діапазоні 2-5 хвилин для чистої збірки та менше 10 секунд для інкрементальної. Продуктивність інкрементальної збірки критично залежить від правильної архітектури модулів та використання імплементації замість API в залежностях, оскільки API призводить до перекомпіляції залежних модулів навіть при внутрішніх змінах. Розмір APK впливає на коефіцієнт встановлення, де дослідження показують, що кожні 6 МБ

збільшення розміру знижують ймовірність встановлення приблизно на один відсоток. Типові розміри варіюються від 1-5 МБ для мінімальних застосунків до 10-50 МБ для середніх та 50-150 МБ для великих застосунків. Кросплатформні фреймворки зазвичай додають від 4 до 20 МБ накладних витрат через необхідність включення середовища виконання та мостових компонентів [20].

Споживання енергії є одним з найкритичніших факторів користувацького досвіду на мобільних пристроях, безпосередньо впливаючи на оцінки в магазині застосунків та рішення користувачів про видалення [21]. Аналіз розряду батареї виявляє розподіл споживання між компонентами, де екран зазвичай споживає 20-40% енергії під час активного використання, процесор 15-30%, мережеві інтерфейси 10-25% та графічний процесор 5-15%. Для режиму з увімкненим екраном легке використання споживає 5-10% заряду на годину, середнє 10-20%, інтенсивне 20-40%. Фонове споживання є особливо критичним показником, де відмінним вважається менше 0.5% на годину, добрим 0.5-1%, прийнятним 1-2%, тоді як перевищення 2% класифікується як поганий показник, що може призвести до обмежень фонової активності з боку системи. Профілювання енергоспоживання на Pixel 6 та новіших пристроях використовує On-Device Power Monitor, який надає реальночасові вимірювання споживання окремих підсистем через моніторинг шин електроспоживання [14]. Типові еталонні значення для миттєвої потужності варіюються від менше 50 міліват у режимі очікування до 200-500 міліват під час активної взаємодії з UI та 500-2000 міліват під час інтенсивних обчислень або графічного рендерингу. Android Vitals відстежує надмірну кількість блокувань увімкнення, що утримуються більше години на сесію, та завислі блокування в режимі сну понад 3 години, оскільки неправильне управління блокуванням в режимі сну є однією з найпоширеніших причин надмірного споживання батареї.

2.4 Дизайн експерименту

Для забезпечення достовірності та відтворюваності результатів дослідження було розроблено детальний дизайн експерименту з чітким визначенням тестового

середовища, конфігурації збірки та методів статистичної обробки даних. Тестове середовище було сформовано на основі фізичного пристрою Xiaomi Redmi Note 13 Pro (модель 23117RA68G) з восьмиядерним процесором з тактовою частотою 2.2 ГГц та 8 ГБ оперативної пам'яті. Пристрій працює під управлінням операційної системи Android 15 (SDK 35) з середовищем виконання ART версії 360729140. Вибір фізичного пристрою замість емулятора обґрунтовується необхідністю отримання реалістичних показників продуктивності, які максимально наближені до умов експлуатації реальними користувачами. Програмне забезпечення для проведення експериментів включало інтегроване середовище розробки Android Studio версії Otter (остання стабільна версія на момент дослідження), бібліотеку Jetpack Macrobenchmark версії 1.3, мову програмування Kotlin версії 1.9 та систему збирання Gradle версії 8.13 з плагіном Android Gradle Plugin версії 8. Така конфігурація забезпечує використання найсучасніших інструментів для профілювання та бенчмаркінгу мобільних застосунків. Конфігурація збірки застосунків відповідала промисловим стандартам випуску програмного забезпечення. Для тестування використовувались виключно релізні збірки з увімкненою мініфікацією коду через оптимізатор R8, стисканням ресурсів та застосуванням правил ProGuard для обфускації. Підписання застосунків здійснювалось за допомогою налагоджувального сховища ключів, що є стандартною практикою для тестових збірок. Така конфігурація дозволяє отримати показники продуктивності, максимально наближені до тих, які користувачі спостерігатимуть після встановлення застосунку з офіційних маркетів. Для статистичної обробки результатів було використано методи описової статистики, що включають обчислення мінімальних, медіанних та максимальних значень показників продуктивності. Особливу увагу приділено аналізу перцентилів, зокрема P90 та P95, які є індустрійними стандартами для оцінювання продуктивності мобільних застосунків. Перцентиль P90 показує, що дев'яносто відсотків вимірювань не перевищують цього значення, що дозволяє виключити найгірші випадки та зосередитись на типовій продуктивності. Для оцінки стабільності результатів обчислювалось стандартне відхилення, яке характеризує

розкид значень навколо медіани. Кожен тип тесту виконувався десять разів для забезпечення статистичної значущості результатів. Перед початком вимірювань здійснювались три попередні запуски для прогріву системи та стабілізації показників. Така методика відповідає рекомендаціям офіційної документації Jetpack Macrobenchmark та забезпечує мінімізацію впливу випадкових факторів на результати експериментів.

2.5 Архітектура тестового застосунку

Для проведення порівняльного аналізу було розроблено спеціалізований тестовий застосунок з модульною архітектурою, що підтримує обидва досліджувані підходи до побудови користувацького інтерфейсу. Застосунок організовано у три незалежні модулі, кожен з яких виконує специфічну роль у дослідженні (рис. 2.1).

Модуль app-views реалізує повний функціонал застосунку з використанням традиційної системи Android Views. Користувацький інтерфейс описано за допомогою XML-розмітки з використанням бібліотеки ViewBinding для безпечного доступу до елементів інтерфейсу. Для організації навігації між екранами застосовано компонент Navigation Component, а для відображення списків використовується RecyclerView з відповідними адаптерами. Така архітектура представляє класичний підхід до розробки Android-застосунків, що домінував в екосистемі протягом останнього десятиліття.

Модуль app-compose містить ідентичну функціональність, але реалізовану з використанням декларативного фреймворку Jetpack Compose. Замість XML-розмітки інтерфейс описується за допомогою композиційних функцій, анотованих як Composable. Навігація організована через Navigation Compose, а списки відображаються за допомогою компонента LazyColumn, який забезпечує ефективне віртуалізоване відображення великої кількості елементів. Вся логіка управління станом інтегрована безпосередньо в композиційні функції з використанням реактивних примітивів фреймворку.

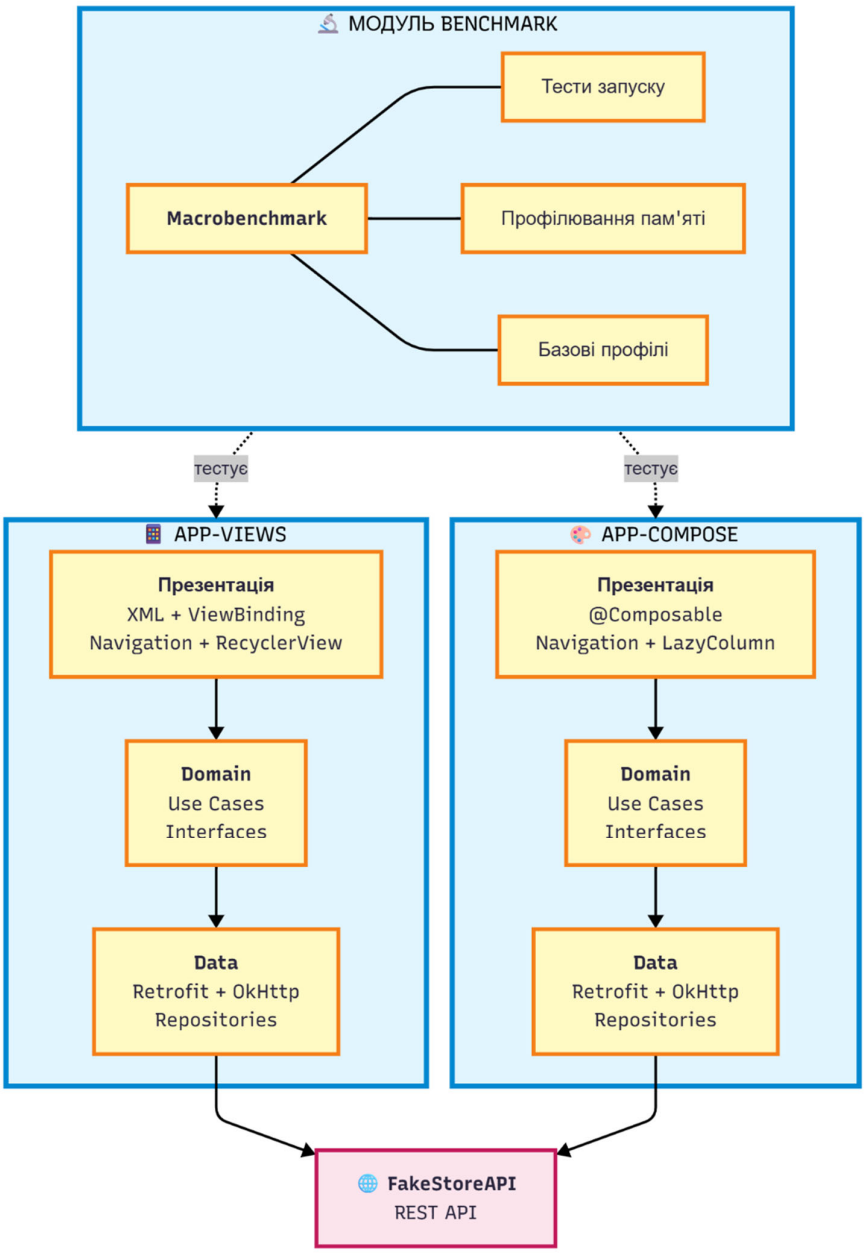

Рисунок 2.1 – Модульна структура застосунку

Третій модуль benchmark призначено для автоматизованого тестування продуктивності обох реалізацій. Він містить набір тестів на основі бібліотеки Jetpack Macrobenchmark, які вимірюють час запуску застосунку за різних сценаріїв, збирають трасування для профілювання пам'яті та генерують профілі базової лінії для оптимізації компіляції. Ізоляція логіки тестування в окремий модуль забезпечує чистоту експериментів та спрощує управління конфігураціями збірки.

Функціональність застосунку включає каталог товарів з можливістю пошуку, детальну інформацію про кожен товар з галереєю зображень та кошик покупок.

Для отримання даних застосунок інтегровано з REST API сервісу FakeStoreAPI, який надає тестові дані про товари. Така функціональність є достатньо складною для виявлення відмінностей у продуктивності підходів, але водночас достатньо типовою для більшості комерційних мобільних застосунків.

Архітектура застосунку побудована за принципами чистої архітектури з чітким розділенням на три шари (рис. 2.2). Доменний шар містить бізнес-логіку та визначає інтерфейси репозиторіїв. Шар даних відповідає за взаємодію з зовнішніми джерелами даних, включаючи мережеві запити через бібліотеки OkHttp3 та Retrofit. Презентаційний шар реалізує логіку відображення та управління станом, при цьому для кожного підходу використовуються відповідні патерни. Для впровадження залежностей застосовано фреймворк Hilt, а для завантаження та кешування зображень використовується бібліотека Coil версії 3, яка має нативну підтримку як Views, так і Compose [3].

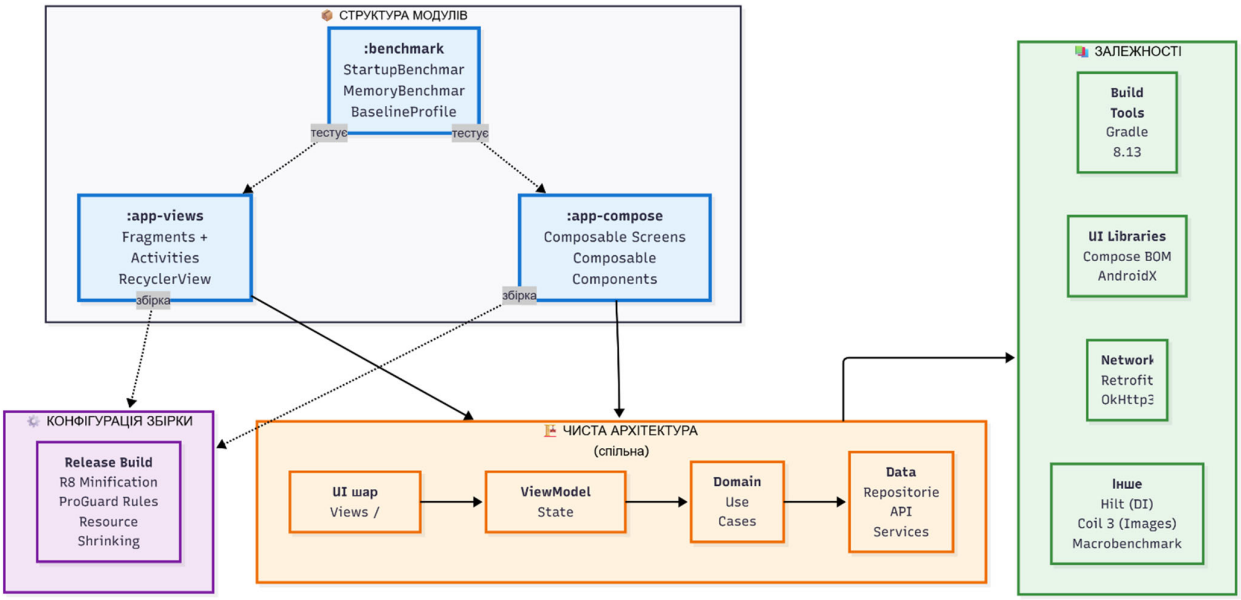

Рисунок 2.2 – Архітектура застосунку

Найважливішим аспектом архітектури є забезпечення ідентичності функціональності та візуального вигляду обох реалізацій. Кожен екран у модулі app-views має точний аналог у модулі app-compose з однаковим розташуванням елементів, кольоровою схемою та взаємодією з користувачем. Така ідентичність дозволяє ізолювати вплив підходу до побудови інтерфейсу від інших факторів, що могли б вплинути на продуктивність застосунку.

2.6 Висновки до розділу 2

У другому розділі було детально розглянуто методику експериментального дослідження продуктивності мобільних застосунків при декларативному та імперативному підході до побудови користувацького інтерфейсу. Систематизовано сучасний інструментарій для профілювання та аналізу продуктивності, що охоплює різні рівні системної архітектури. Показано, що ефективне дослідження вимагає інтеграції кількох спеціалізованих засобів, кожен з яких надає унікальні можливості для аналізу специфічних аспектів поведінки застосунку.

Запропонований багаторівневий підхід профілювання забезпечує можливість виявлення проблем продуктивності на етапі експлуатації через збір метрик від реальних користувачів, подальше відтворення та аналіз цих проблем у середовищі розробки з використанням інструментів профілювання процесора та пам’яті, а також глибокий аналіз системного контексту через трасування ядра операційної системи. Така ієрархічна організація процесу аналізу дозволяє ефективно локалізувати причини проблем продуктивності та визначити оптимальні шляхи їх усунення.

Розглянуті методології бенчмаркінгу надають можливість проведення як наскрізного тестування користувацьких сценаріїв, так і ізольованого вимірювання продуктивності окремих компонентів. Застосування макробенчмарків для вимірювання часу запуску застосунку та швидкості прокручування списків забезпечує оцінку продуктивності у контексті реальних користувацьких взаємодій. Водночас мікробенчмарки дозволяють досліджувати ефективність окремих

операцій композиції та рендерингу інтерфейсу, що є критично важливим для глибокого розуміння внутрішніх механізмів різних підходів до побудови інтерфейсу. Визначені метрики продуктивності охоплюють всі аспекти користувацького досвіду, починаючи від плавності роботи інтерфейсу та швидкості відгуку на взаємодії, і закінчуючи споживанням системних ресурсів та енергоефективністю.

Розроблений дизайн експерименту забезпечує відтворюваність та статистичну значущість результатів дослідження. Використання фізичного пристрою замість емулятора гарантує отримання реалістичних показників продуктивності, максимально наближених до умов реальної експлуатації. Застосування релізних збірок з увімкненими оптимізаціями коду та ресурсів відповідає промисловим стандартам випуску програмного забезпечення. Визначена статистична методика з десятьма ітераціями вимірювань та аналізом перцентилів забезпечує надійність отриманих результатів та можливість виключення впливу випадкових факторів.

Архітектура розробленого тестового застосунку з модульною організацією дозволяє проводити коректне порівняння обох підходів до побудови інтерфейсу. Забезпечення повної ідентичності функціональності та візуального вигляду обох реалізацій дозволяє ізолювати вплив підходу до побудови інтерфейсу від інших факторів. Застосування принципів чистої архітектури з чітким розділенням на шари гарантує, що відмінності у продуктивності будуть пов’язані саме з особливостями декларативного та імперативного підходів, а не з відмінностями в організації бізнес-логіки чи роботі з даними.

Таким чином, розроблена методика створює фундамент для проведення порівняльного аналізу продуктивності декларативного та імперативного підходів до побудови користувацького інтерфейсу мобільних застосунків. Комбінація сучасних інструментів профілювання, суворих методів бенчмаркінгу, комплексних метрик продуктивності та статистично обґрунтованого експериментального дизайну забезпечує достовірність результатів дослідження та можливість їх

практичного застосування при виборі технологічного стеку для розробки мобільних застосунків.

3 ПРАКТИЧНЕ ДОСЛІДЖЕННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

3.1 Реалізація тестового застосунку

Розробка тестового застосунку здійснювалась з дотриманням принципу максимальної еквівалентності реалізацій при збереженні специфіки кожного підходу. Проектні рішення приймались з урахуванням рекомендацій офіційної документації та кращих практик індустрії для забезпечення репрезентативності результатів. Застосунок являє собою імітацію каталогу товарів онлайн магазину з базовим функціоналом (рис. 3.1):

- вертикальна прокрутка списку з 250 картками товарів із зображеннями та коротким описом;
- детальний опис кожного товару на окремому екрані при натисканні на картці товару;
- пошук товарів;
- кошик.

Реалізація на основі Android Views (рис. 3.2) використовує класичну тришарову структуру екранів з MainActivity як точкою входу, Fragment-компонентами для окремих екранів та XML-файлами для опису розмітки UI. Головний екран каталогу товарів містить RecyclerView з власним адаптером, який відповідає за створення та зв'язування елементів списку. Для оптимізації продуктивності застосовано патерн ViewHolder, який дозволяє повторно використовувати елементи списку під час прокручування. Поле пошуку реалізовано як EditText з TextWatcher для відстеження змін введення користувача. Навігація до екрану деталей товару здійснюється через Navigation Component з передачею ідентифікатора товару як аргументу навігації. Екран деталей товару містить ScrollView з вкладеним LinearLayout, що включає ViewPager2 для галереї зображень, текстові поля для опису та характеристик товару, а також кнопку додавання до кошика. Завантаження

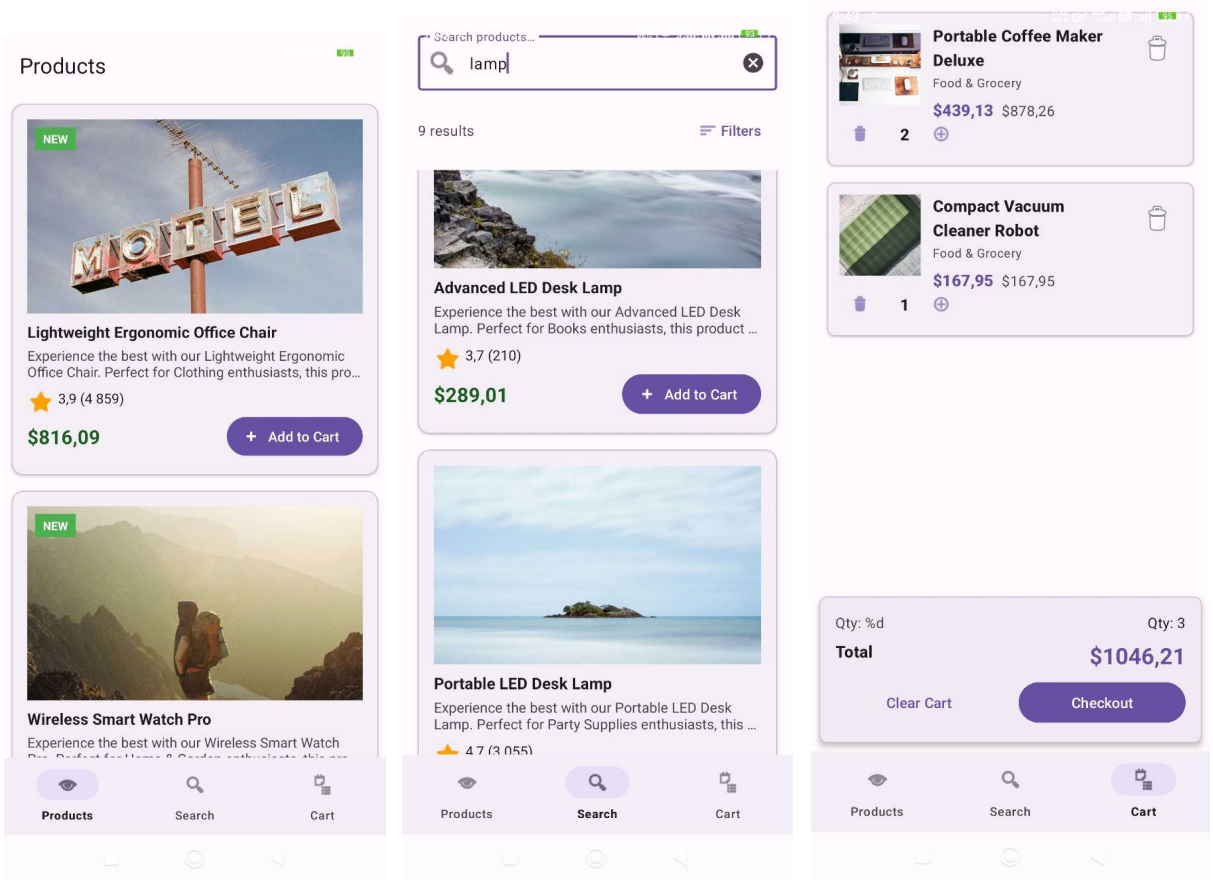

Рисунок 3.1 – Інтерфейс та функціонал тестового застосунку (ідентичні для views та compose)

випадкових зображень здійснюється асинхронно через бібліотеку Coil з автоматичним кешуванням для покращення продуктивності при повторних відкриттях. Управління станом екрану організовано через ViewModel з використанням LiveData для реактивного оновлення інтерфейсу при зміні даних.

Головний екран каталогу товарів містить RecyclerView з власним адаптером, який відповідає за створення та зв'язування елементів списку. Для оптимізації продуктивності застосовано патерн ViewHolder, який дозволяє повторно використовувати елементи списку під час прокручування. Поле пошуку реалізовано як EditText з TextWatcher для відстеження змін введення користувача. Навігація до екрану деталей товару здійснюється через Navigation Component з передачею ідентифікатора товару як аргументу навігації. Екран деталей товару містить ScrollView з вкладеним LinearLayout, що включає ViewPager2 для галереї

зображень, текстові поля для опису та характеристик товару, а також кнопку додавання до кошика. Завантаження зображень здійснюється асинхронно через бібліотеку Coil з автоматичним кешуванням для покращення продуктивності при повторних відкриттях. Управління станом екрану організовано через ViewModel з використанням LiveData для реактивного оновлення інтерфейсу при зміні даних.

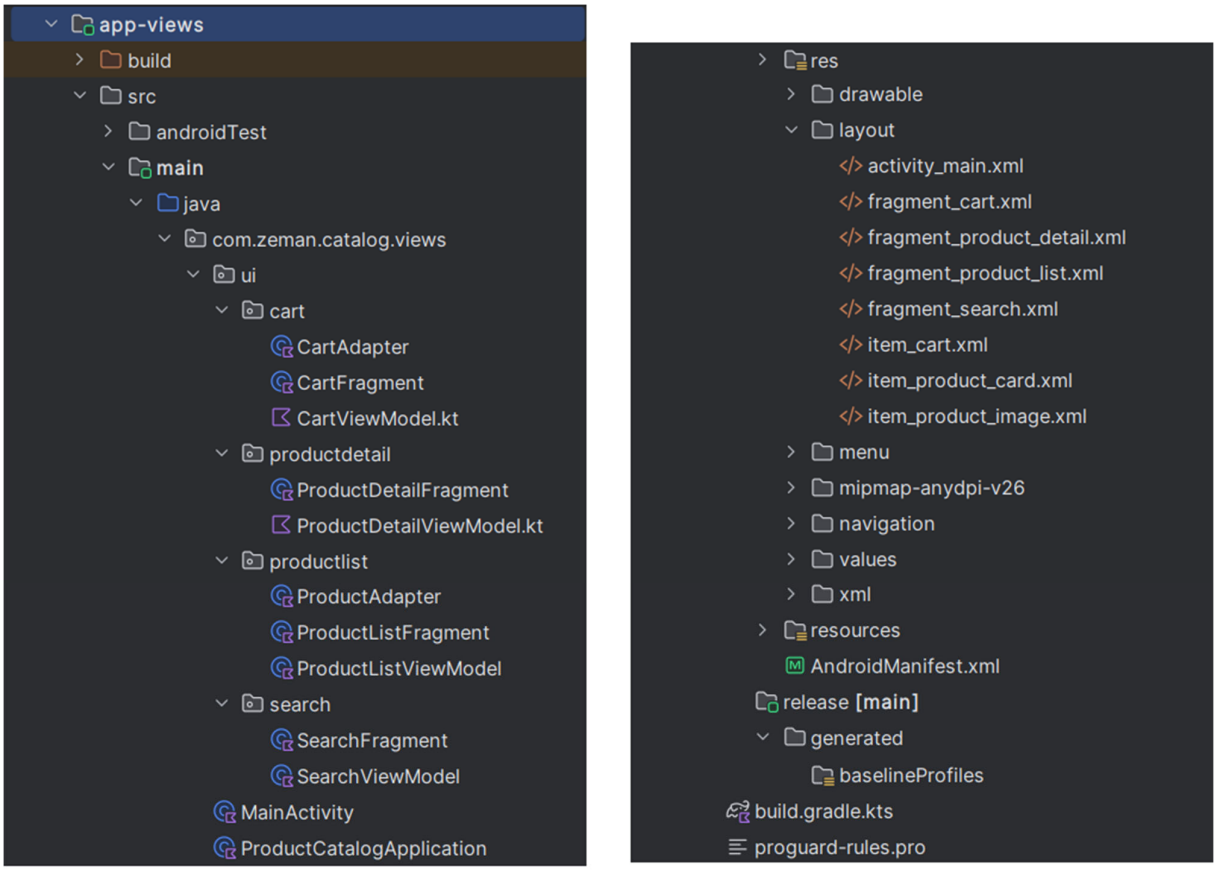

Рисунок 3.2 – Структура модуля app-views

Реалізація на основі Jetpack Compose (рис. 3.3) відрізняється за способом організації коду, але зберігає ідентичну функціональність та візуальний вигляд. Кожен екран представлено як композиційну функцію, що приймає необхідні параметри та описує структуру інтерфейсу декларативно. Каталог товарів реалізовано через LazyColumn, який автоматично віртуалізує елементи списку та забезпечує ефективне використання пам'яті. На відміну від RecyclerView, LazyColumn не потребує створення окремого адаптера, оскільки логіка

відображення елементів описується безпосередньо у композиційній функції через блок items. Управління станом у Compose організовано за допомогою функцій remember та mutableStateOf, які забезпечують реактивність інтерфейсу. Зміна значення змінної стану автоматично призводить до рекомпозиції відповідних

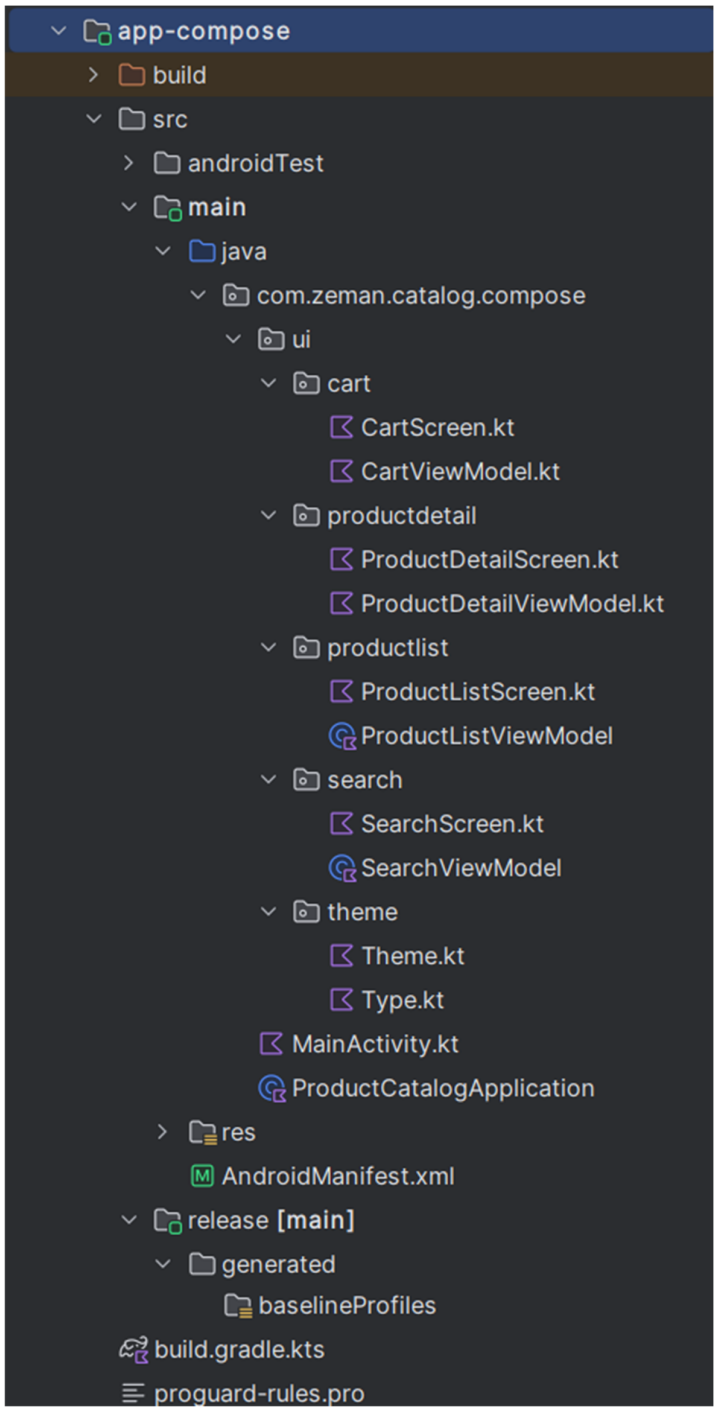

Рисунок 3.3 – Структура модуля app-compose

частин інтерфейсу без необхідності ручного оновлення, як це потрібно у Views. Навігація реалізована через Navigation Compose з використанням NavHost та композиційних пунктів призначення замість фрагментів. Галерея зображень на екрані деталей товару реалізована через HorizontalPager, який є Compose-аналогом ViewPager2. Завантаження зображень здійснюється через ту саму бібліотеку Coil, але з використанням спеціалізованої функції AsyncImage, призначеної для Compose. Така уніфікація бібліотек для роботи з зображеннями дозволяє мінімізувати вплив сторонніх факторів на результати порівняння.

Особлива увага приділялась забезпеченню ідентичності бізнес-логіки обох реалізацій. Весь код роботи з мережею, обробки даних (рис. 3.4) та управління станом винесено в спільні ViewModel-класи, які використовуються обома модулями. Відмінності стосуються виключно способу відображення даних та організації користувацького інтерфейсу, що дозволяє ізолювати вплив підходу до побудови інтерфейсу на продуктивність застосунку.

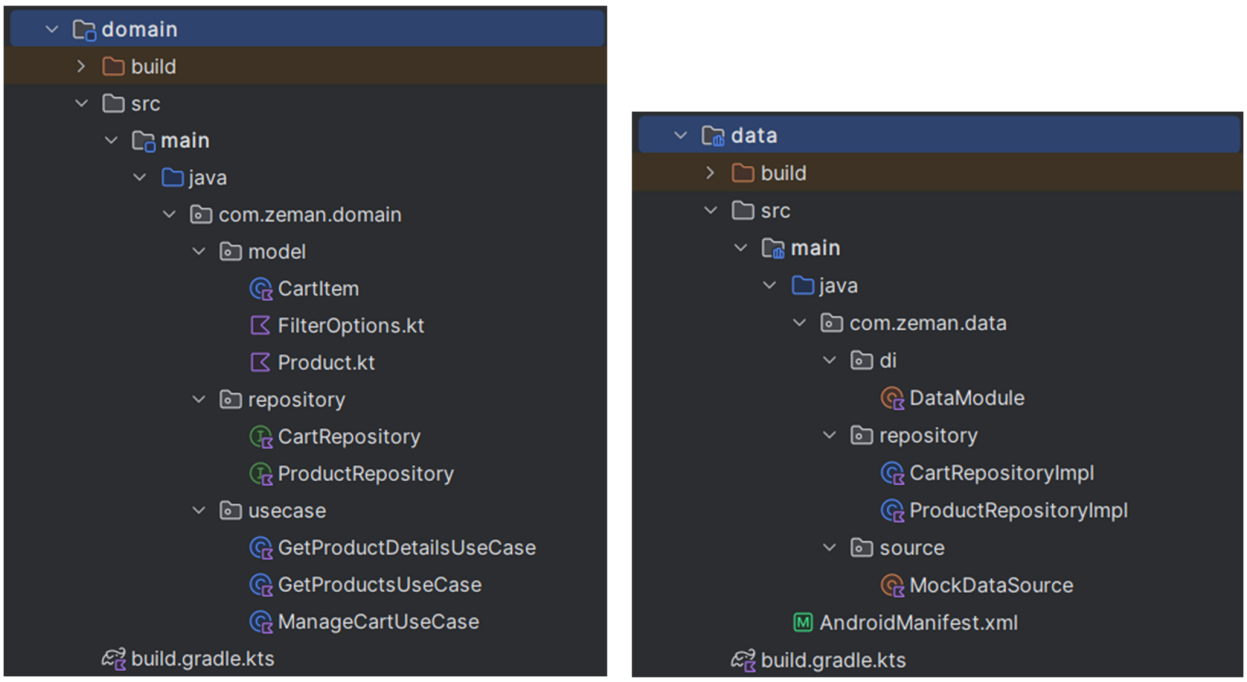

Рисунок 3.4 – Рівні domain та data

3.2 Конфігурація тестів продуктивності

Автоматизоване тестування продуктивності організовано через модуль benchmark (рис. 3.5) з використанням бібліотеки Jetpack Macrobenchmark, яка забезпечує високоточні вимірювання за реальних умов експлуатації застосунку. На відміну від мікробенчмарків, які тестують окремі функції, макробенчмарки вимірюють продуктивність повних користувацьких сценаріїв від запуску застосунку до завершення операції.

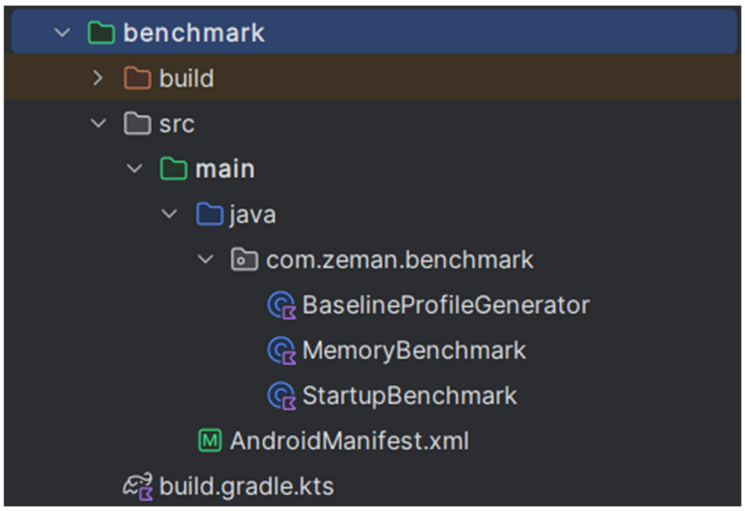

Рисунок 3.5 – Структура модуля benchmark

Конфігурація кожного тесту включає десять ітерацій для забезпечення статистичної значущості результатів та три попередні запуски для прогріву системи. Програмний код файлів модуля benchmark приведено у Додатку А. Попередні запуски критично важливі для стабілізації показників, оскільки перші виконання часто демонструють аномальні результати через компіляцію коду інтерпретатором, завантаження системних компонентів та інші фактори холодного старту. Параметри тестування задаються через клас StartupTimingMetric, який автоматично вимірює час від моменту запуску застосунку до появи першого кадру на екрані.

Для тестування продуктивності запуску розроблено чотири типи сценаріїв, які відображають різні умови використання застосунку реальними користувачами. Гарячий (hot) запуск симулює ситуацію, коли користувач повертається до застосунку, який вже виконується у пам'яті, але Activity було тимчасово знищено системою. Теплий (warm) запуск відбувається, коли процес застосунку залишається живим у пам'яті, але Activity потрібно відновити. Холодний (cold) запуск представляє найскладніший сценарій, коли застосунок запускається вперше після встановлення або після повного завершення процесу системою.

Особливу увагу приділено тестуванню впливу Baseline Profile на продуктивність. Профіль базової лінії являє собою файл, що містить інформацію про критичні шляхи виконання коду застосунку, яку система використовує для попередньої компіляції найважливіших частин програми під час встановлення. Це можливість вказати компілятору ті ділянки коду, які б ви хотіли прекомпілювати на пристроях користувачів. Ці профілі формують початковий кеш, який дозволяє прискорити час першого запуску та продуктивність. Для Compose-застосунків генерація профілю базової лінії здійснювалась автоматично через інструменти Macrobenchmark шляхом виконання типових користувацьких сценаріїв та збирання трасувань виконання. Збірки застосунків для тестування створювались у двох варіантах: з профілем базової лінії та без нього. Це дозволило кількісно оцінити вплив попередньої компіляції на продуктивність кожного підходу. Для Views-застосунку профіль базової лінії також генерувався для забезпечення справедливості порівняння, хоча очікувався мінімальний вплив через природу традиційної системи побудови інтерфейсу. Усі тести виконувались на одному фізичному пристрої з ідентичними умовами для мінімізації впливу зовнішніх факторів. Перед кожною серією вимірювань застосунок повністю видалявся з пристрою та встановлювався заново для забезпечення чистоти експерименту. Під час виконання тестів пристрій перебував у режимі польоту для виключення впливу мережевої активності, а фонові процеси були мінімізовані.

3.3 Результати тестування продуктивності запуску

Експериментальне тестування продуктивності запуску застосунків (рис. 3.6) виявило значні відмінності між підходами, характер яких суттєво залежить від типу запуску та наявності оптимізацій. Результати для Views та Compose представлено у табл. 3.1 і табл. 3.2 відповідно. Табл. 3.3 показує порівняння підходів.

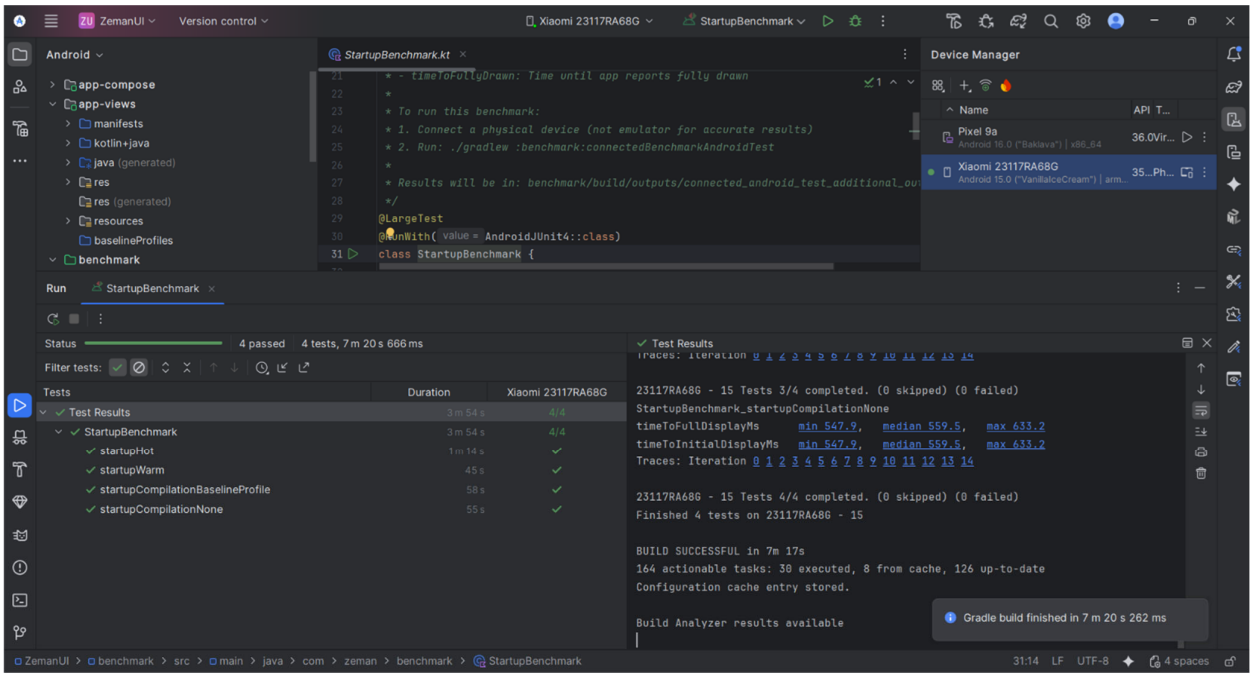

Рисунок 3.6 – Завершений тест продуктивності запуску

Аналіз результатів гарячого запуску демонструє переконливу перевагу декларативного підходу. Медіанний час запуску для Compose становить 97,93 мілісекунди проти 109,66 мілісекунд для Views, що відповідає покращенню на 10,7 відсотка. Така відмінність пояснюється тим, що композиційні функції створюють легші об'єкти порівняно з традиційними View-компонентами, а механізм рекомпозиції ефективніше відновлює стан інтерфейсу після тимчасового знищення Activity. Додаткову перевагу надає той факт, що середовище виконання Compose вже ініціалізоване під час гарячого запуску, що усуває накладні витрати на початкове завантаження бібліотеки. Теплий запуск виявляє ще більш виразну

перевагу Compose з медіанним часом 198,79 мілісекунд проти 286,08 мілісекунд для Views, що становить покращення на 30,5 відсотка. Ця суттєва відмінність обумовлена фундаментальними архітектурними особливостями підходів.

Таблиця 3.1 – Результати тестування продуктивності запуску для Android Views

Тип запуску	**Мін, мс**	**Медіана, мс**	**Макс, мс**	**P90, мс**
Гарячий запуск	98.63	109.66	129.64	123.19
Теплий запуск	262.21	286.08	308.83	301.42
Холодний (БП)	568.55	592.82	636.97	623.93
Холодний (без БП)	557.15	588.13	641.82	624.48

Таблиця 3.2 – Результати тестування продуктивності запуску для Jetpack Compose

Тип запуску	**Мін, мс**	**Медіана, мс**	**Макс, мс**	**P90, мс**
Гарячий запуск	77.78	97.93	125.25	115.25
Теплий запуск	174.11	198.79	237.13	224.31
Холодний (БП)	602.79	625.19	651.14	644.48
Холодний (без БП)	689.07	711.42	776.75	757.65

Таблиця 3.3 – Порівняння продуктивності запуску

Тип запуску	**Views**	**Compose**	**Переможець**	**Різниця**
Гарячий запуск	109.66 мс	97.93 мс	Compose	– 10.7%
Теплий запуск	286.08 мс	198.79 мс	Compose	– 30.5%
Холодний (БП)	592.82 мс	625.19 мс	Views	+ 5.5%
Холодний (без БП)	588.13 мс	711.42 мс	Views	+ 21.0%

У Views-застосунках теплий запуск потребує відновлення всієї ієрархії елементів інтерфейсу через повторну інфляцію XML-розмітки, створення об'єктів View та налаштування всіх зв'язків між компонентами. Натомість Compose зберігає стан у легкій структурі даних та ефективно відновлює композицію без необхідності повторного створення важких об'єктів. Холодний запуск з профілем базової лінії демонструє протилежну картину, де Views виявляється швидшим з медіанним

часом 592,82 мілісекунди проти 625,19 мілісекунд для Compose, що становить різницю 5,5 відсотка на користь традиційного підходу. Такий результат пояснюється тим, що система Views є частиною базового програмного забезпечення Android та уже оптимізована на рівні платформи. XML-розмітка компілюється у бінарний формат під час збирання застосунку, що прискорює інфляцію. Compose натомість потребує ініціалізації власного середовища виконання, що створює додаткові накладні витрати під час першого запуску. Найбільш суттєва відмінність спостерігається при холодному запуску без профілю базової лінії. Views демонструє медіанний час 588,13 мілісекунд, тоді як Compose потребує 711,42 мілісекунди, що становить різницю у 21 % на користь традиційного підходу. Таке значне відставання Compose пояснюється великою кількістю коду, який потребує інтерпретації або динамічної компіляції під час першого запуску.

З табл. 3.4 видно, що Views практично не виграє від профілю базової лінії (різниця лише 0,8 відсотка), тоді як для Compose покращення становить 12,1 %, що підкреслює критичну важливість цієї оптимізації для декларативних застосунків.

Таблиця 3.4 – Вплив Baseline Profile

Підхід	**Без BP, мс**	**З BP, мс**	**Покращення, мс**	**Вплив, мс**
Views	588.13	592.82	– 4.69	– 0.8%
Compose	711.42	625.19	+ 86.23	+ 12.1%

3.4 Аналіз розміру застосунку

Дослідження розміру пакунків APK обох реалізацій виявило кардинальні відмінності між підходами, які мають практичні наслідки для розповсюдження програмного забезпечення. Результати проведеного аналізу представлено у табл. 3.5.

Аналіз показує, що Compose-застосунок більше ніж вдвічі перевищує Views-застосунок за розміром, що становить найбільш суттєву різницю серед усіх досліджуваних показників продуктивності. Розмір завантаження, який користувачі бачать у магазині застосунків перед встановленням, зростає на 167 %, що може негативно впливати на рішення про встановлення, особливо для користувачів з обмеженим Інтернет-з'єднанням або малим обсягом вільного сховища на пристрої.

Таблиця 3.5 – Порівняльний аналіз розміру APK застосунків

Метрика	**Views**	**Compose**	**Різниця**	**Примітка**
Розмір АПК	2.3 МБ	4.9 МБ	+ 113%	Views менший на 53%
Розмір завантаження	1.5 МБ	4.0 МБ	+ 167%	Views менший на 63%
Методи	13.8K	61.2K	+ 342%	Потребує multidex
Класи	2,563	10,146	+ 296%	Різниця = runtime

Розбір компонентів пакунку (табл. 3.6) виявляє джерело такого збільшення розміру. DEX-файли, які містять скомпільований код застосунку, займають 1,0 мегабайти у Views-застосунку та 3,6 мегабайти у Compose-застосунку (3,4 МБ у classes.dex плюс 202,6 КБ у classes2.dex).

Таблиця 3.6 – Розбивка компонентів APK

Компонент	**Views, МБ**	**Compose, МБ**	**Різниця, МБ**
classes.dex	1.0	3.6	+ 2.6
classes2.dex	–	0.2	+ 0.2
resources.arse	0.85	1.00	+ 0.15
res/	0.26	0.17	– 0.09
lib/	–	0.04	+ 0.04

Необхідність використання двох DEX-файлів для Compose вказує на наближення до ліміту у 64 тисячі методів, що є технічним обмеженням формату DEX. Views-застосунок комфортно вміщується в один DEX-файл з великим

запасом. Цікавим спостереженням є те, що Compose-застосунок фактично має менше ресурсів у каталозі res: 175,1 кілобайта проти 260,1 кілобайта у Views. Це природний наслідок відсутності XML-файлів розмітки у декларативному підході, де весь інтерфейс описується програмно. Однак ця економія повністю нівелюється величезним збільшенням розміру коду.

Аналіз кількості методів (табл. 3.7) за бібліотеками (таблиці 3.8 та 3.9) розкриває фундаментальну причину різниці у розмірі. У Views-застосунку бібліотеки androidx відповідають за 5926 методів (42,8 % від загальної кількості), тоді як у Compose-застосунку ці самі бібліотеки включають 55540 методів (90,8 %). Різниця у 49614 методів практично повністю складається з коду середовища виконання Compose, включаючи компілятор, систему управління станом, модуль композиції та інші компоненти фреймворку. Важливо відзначити, що власний код застосунку у Compose-реалізації фактично компактніший: 1772 методи проти 2570 методів у Views-реалізації, що становить економію у 31 %. Це підтверджує тезу про те, що декларативний підхід дозволяє писати більш лаконічний код з меншою кількістю допоміжних конструкцій. Однак ця економія несуттєва порівняно з накладними витратами самого фреймворку.

Таблиця 3.7 – Кількість методів та класів

Метрика	**Views**	**Compose**	**Різниця**
Класи	2563	10146	+ 7583
Визначені методи	13851	61180	+ 47329
Посилання на методи	17449	65369	+ 47920

Таблиця 3.8 – Розбивка по бібліотекам (Views)

Бібліотека	**Методи**	**Частка, %**
androidx	5926	42.8
app code	2570	18.6
coiI3	2137	15.4
kotlinx	979	7.1

Бібліотека	Методи	Частка, %
kotlin	761	5.5
okhttp3	667	4.8
ВСЬОГО	**13851**	**100**

Таблиця 3.9 – Розбивка по бібліотекам (Compose)

Бібліотека	Методи	Частка, %
androidx	55540	90.8
coil3	2457	4.0
app code	1772	2.9
kotlin	1043	1.7
dagger	300	0.5
okhttp3	143	0.2
ВСЬОГО	**61180**	**100**

3.5 Статистичний аналіз та інтерпретація результатів

Для оцінки стабільності результатів проведено аналіз варіативності вимірювань через обчислення стандартного відхилення для кожного типу запуску. Результати представлено у табл. 3.9.

Аналіз варіативності виявляє, що Views-застосунок демонструє більш передбачувані результати для гарячого та теплого запуску, тоді як Compose показує вищу стабільність при холодному запуску з профілем базової лінії. Така різниця може бути пов'язана з тим, що механізми відновлення стану у Views є більш детермінованими для простих сценаріїв, тоді як оптимізована компіляція Compose забезпечує більш стабільну продуктивність при повному запуску застосунку.

Комплексна інтерпретація результатів виявляє картину переваг та недоліків кожного підходу, що унеможливлює однозначну рекомендацію на користь одного з них. У табл.3.10 та на рис. 3.7 представлено комплексне порівняння результатів порівняння підходів за всіма досліджуваними параметрами.

Таблиця 3.9 – Аналіз стабільності результатів (стандартне відхилення)

Тип	Views	Compose	Стабільніший
Гарячий запуск	±15 мс	±24 мс	Views
Теплий запуск	±23 мс	±32 мс	Views
Холодний (БП)	±34 мс	±24 мс	Compose
Холодний (без БП)	±42 мс	±44 мс	Однаково

Таблиця 3.10 – Комплексне порівняння підходів

Критерій	Android Views	Jetpack Compose
Розмір APK	Менший (– 53%)	Більший (+ 113%)
Гарячий запуск	Повільніший	Швидше (+ 10.7%)
Теплий запуск	Повільніший	Швидше (+ 30.5%)
Холодний запуск	Швидше (+ 21%)	Повільніший
Довжина коду	Більше (+ 31%)	Коротше (– 31%)
Кількість методів	Менше (– 77%)	Більше (+ 77%)
Стабільність	Більш стійкий	Менш стійкий
Досвід розробника	Складніше	Простіше
Сучасність підходу	Застарілий	Новітній

Декларативний підхід демонструє значну перевагу у сценаріях, коли застосунок вже завантажено у пам'ять, що відповідає типовій поведінці користувачів, які часто перемикаються між застосунками протягом дня. Покращення на 10,7 % для гарячого запуску та 30,5 % для теплого запуску забезпечує більш відгукливий користувацький досвід при повторному відкритті застосунку. Додатковою перевагою є компактність коду, що полегшує розробку та підтримку застосунку. Традиційний підхід зберігає перевагу у критично важливому сценарії першого запуску застосунку, коли користувач щойно встановив програму або відкриває її після тривалого періоду неактивності. Різниця у 5,5 % з профілем базової лінії та 21,0 % без нього може бути суттєвою для застосунків, де перше враження критично важливе для утримання користувачів. Значно менший розмір пакунку також є важливою перевагою, особливо для ринків з обмеженою

пропускною здатністю інтернету або користувачів з обмеженим сховищем на пристроях. Профіль базової лінії виявився критично важливим для Compose-застосунків, забезпечуючи покращення продуктивності холодного запуску на 12,1 %. Без цієї оптимізації декларативний підхід демонструє неприйнятно повільний запуск, що робить профіль базової лінії обов'язковим компонентом будь-якого промислового Compose-застосунку. Для Views-застосунків вплив профілю мінімальний, що вказує на їхню меншу залежність від оптимізацій часу виконання.

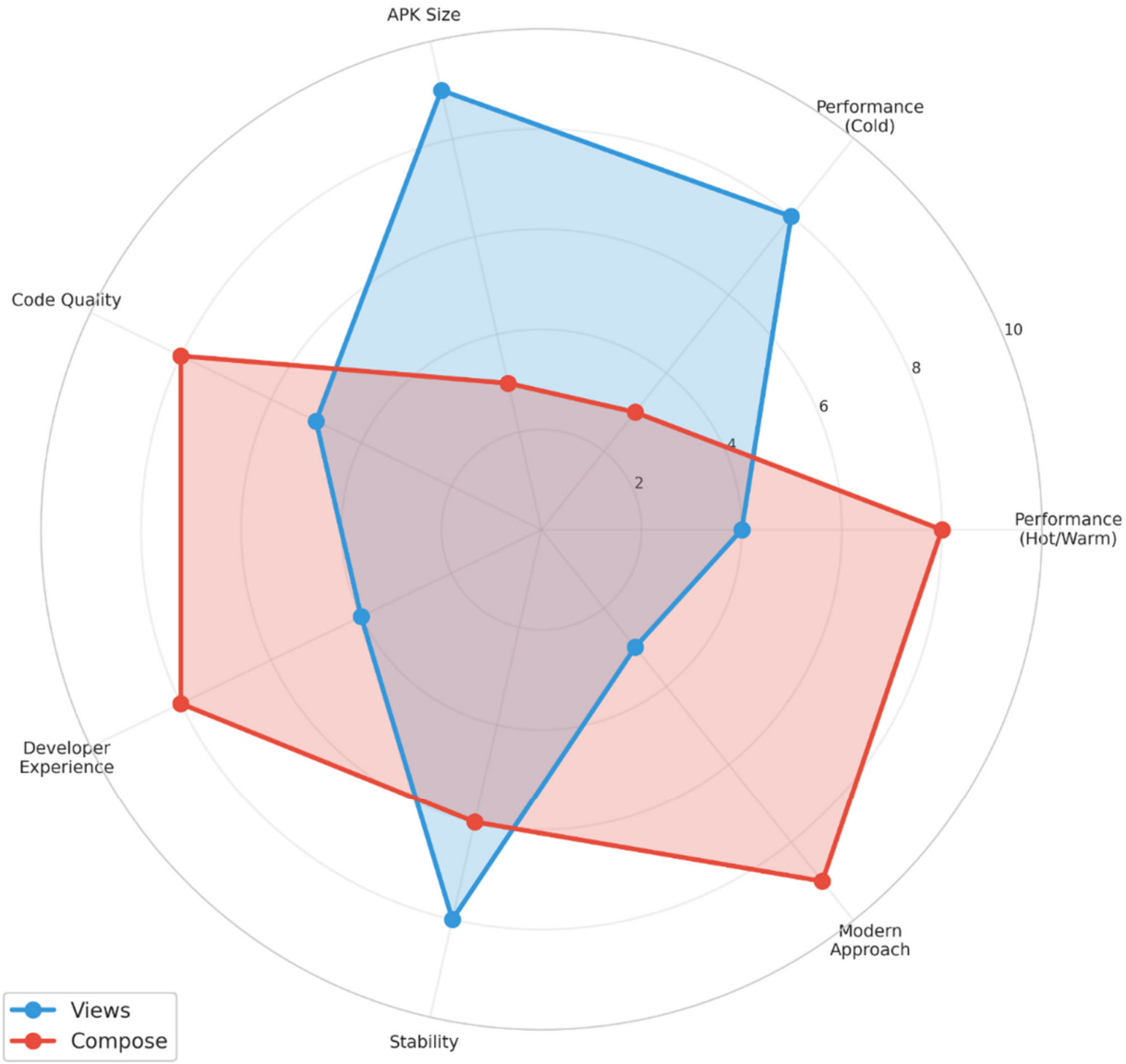

Рисунок 3.7 – Комплексне порівняння підходів

3.6 Обмеження дослідження

Проведене дослідження має кілька значних обмежень, які важливо враховувати при інтерпретації результатів та формулюванні висновків.

Аналіз використання пам'яті було виконано лише частково через технічні обмеження інструментів вимірювання. Хоча вдалось зібрати 30 трасувань виконання через систему Perfetto (рис. 3.8), автоматизована обробка цих даних виявилась неможливою через особливості програмного інтерфейсу TraceSectionMetric, який не повертає числові метрики у структурованому форматі. Детальний аналіз використання пам’яті потребує ручного вивчення трасувань через веб-інтерфейс Perfetto або спеціалізовані інструменти профілювання, що виходить за межі автоматизованого тестування, реалізованого у цьому дослідженні.

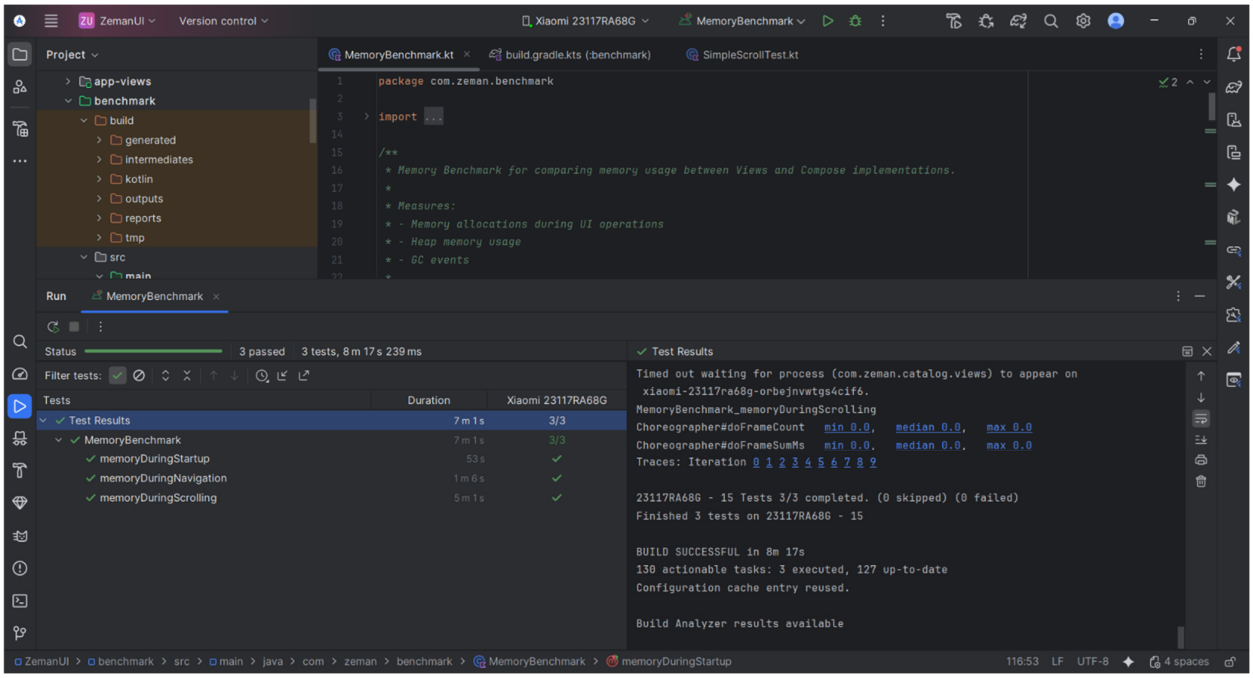

Рисунок 3.8 – 15 трасувань Perfetto зібрано (на прикладі app-views)

Вимірювання продуктивності прокручування списків не вдалось здійснити через несумісність програмного інтерфейсу FrameTimingMetric з версією операційної системи Android 15, що використовувалась на тестовому пристрої.

Незважаючи на численні спроби з різними конфігураціями тестів, інструмент стабільно повертав нульові значення метрик або взагалі не збирав дані. Така ситуація відображає типову проблему при роботі з новітніми версіями платформи, коли інструменти розробника ще не адаптовані до змін в архітектурі системи.

Вимірювання часу збирання застосунків не було включено до дослідження через технічні проблеми з конфігурацією середовища виконання Java, які унеможливили запуск процесу збирання з командного рядка. Хоча час збирання є важливим показником продуктивності розробки, його вплив на досвід кінцевих користувачів є опосередкованим, тому відсутність цих даних не критично впливає на основні висновки дослідження.

Незважаючи на ці обмеження, отримані дані з продуктивності запуску та розміру застосунку є достатньо комплексними для формулювання обґрунтованих висновків про відмінності між підходами. Ці два показники найбільш безпосередньо впливають на досвід користувачів: час запуску визначає, наскільки швидко користувач може почати працювати з застосунком, а розмір APK-пакунку впливає на рішення про встановлення та вимоги до сховища пристрою.

3.7 Практичні рекомендації

На основі отриманих результатів можна сформулювати практичні рекомендації щодо вибору підходу до побудови користувацького інтерфейсу залежно від специфіки проекту та цільової аудиторії (табл. 3.11).

Jetpack Compose рекомендується обирати для нових проектів, особливо тих, що розробляються великими командами або передбачають тривалий цикл підтримки. Перевага у продуктивності гарячого та теплого запуску забезпечує кращий досвід для користувачів, які регулярно взаємодіють із застосунком протягом дня. Компактність коду та декларативна природа опису інтерфейсу полегшує підтримку та розвиток функціональності. Для корпоративних застосунків, де розмір пакунку не є критичним обмеженням, додаткові 2,6 МБ є прийнятною ціною за покращений досвід розробки.

Таблиця 3.11 – Рекомендації вибору

Views	**Compose**
Критичний розмір APK	Нові проекти
Ринки з обмеженнями	Складний користувацький інтерфейс
Старші пристрої	Продуктивність розробника
Перший запуск критичний	Корпоративні застосунки
Модулі функцій	Реактивні шаблони

Android Views залишається доцільним вибором для проектів з жорсткими обмеженнями на розмір застосунку, зокрема для ринків з обмеженою пропускною здатністю інтернету або користувачів з пристроями з малим обсягом сховища. Застосунки, де критично важливий швидкий перший запуск, такі як програми для входу в систему або утилітарні інструменти, що рідко залишаються у пам'яті, також виграють від традиційного підходу. Для модульних застосунків та технології швидких застосунків, де обмеження на кількість методів є критичним, Views уникає необхідності використання багатьох DEX-файлів.

Гібридний підхід, що поєднує обидві технології, може бути оптимальним рішенням для складних проектів або поступової міграції існуючих застосунків. Jetpack Compose підтримує інтеграцію з традиційною системою Views через компоненти ComposeView та AndroidView, що дозволяє використовувати декларативний підхід для нових складних екранів, залишаючи прості або критичні для продуктивності частини на Views. Така стратегія дозволяє поступово впроваджувати переваги нового підходу без необхідності повного переписування застосунку. Незалежно від обраного підходу, використання профілю базової лінії є обов'язковим для Compose-застосунків. Покращення продуктивності холодного запуску на 12,1 % робить цю оптимізацію важливою для забезпечення конкурентоспроможного досвіду користувачів. Генерація профілю повинна бути інтегрована у процес збирання релізних версій застосунку як стандартна практика.

3.8 Висновки до розділу 3

У третьому розділі проведено практичне дослідження продуктивності мобільних застосунків при використанні декларативного та імперативного підходів до побудови користувацького інтерфейсу. Експериментальна база дослідження включала розробку двох функціонально ідентичних реалізацій тестового застосунку з використанням Jetpack Compose та Android Views відповідно, що дозволило ізолювати вплив підходу до побудови інтерфейсу на показники продуктивності при збереженні єдиної бізнес-логіки.

Результати автоматизованого тестування продуктивності запуску виявили суттєві відмінності між підходами, характер яких залежить від типу запуску застосунку. Декларативний підхід на основі Jetpack Compose демонструє переконливу перевагу у сценаріях повторного відкриття застосунку, досягаючи покращення на 10,7 відсотка для гарячого запуску та на 30,5 відсотка для теплого запуску порівняно з традиційним підходом. Така перевага пояснюється ефективнішим механізмом відновлення стану інтерфейсу через легші композиційні функції, що створюють менші накладні витрати порівняно з повторною побудовою ієрархії елементів View. Ці результати мають практичне значення для застосунків з високою частотою взаємодій, де користувачі регулярно повертаються до програми протягом дня.

Протилежна картина спостерігається при аналізі холодного запуску застосунку, де традиційний підхід Android Views демонструє кращу продуктивність. За наявності профілю базової лінії різниця становить 5,5 відсотка на користь Views, тоді як без цієї оптимізації відставання декларативного підходу досягає 21,0 відсотка. Таке відставання Compose обумовлене необхідністю ініціалізації власного середовища виконання та більшим обсягом коду, що потребує інтерпретації або динамічної компіляції при першому запуску. Традиційна система Views натомість є інтегрованою частиною платформи Android з оптимізацією на рівні операційної системи та попередньо скомпільованою XML-розміткою. Дослідження підтвердило критичну важливість профілю базової лінії для

декларативних застосунків, оскільки його використання забезпечує покращення продуктивності холодного запуску на 12,1 відсотка, тоді як для Views-застосунків ефект є мінімальним.

Аналіз розміру пакунків застосунків виявив найбільш суттєву відмінність між підходами. Застосунок на основі Jetpack Compose більше ніж удвічі перевищує розмір традиційного застосунку, при цьому розмір завантаження зростає на 167 відсотків. Детальний розбір компонентів пакунку показав, що основне збільшення відбувається через код середовища виконання декларативного фреймворку, який додає майже п'ятдесят тисяч додаткових методів порівняно з традиційним підходом. При цьому власний код застосунку у декларативній реалізації виявився компактнішим на 31 відсоток, що підтверджує лаконічність синтаксису композиційних функцій, однак ця економія повністю нівелюється накладними витратами фреймворку. Збільшення розміру застосунку має практичні наслідки для розповсюдження програмного забезпечення, особливо на ринках з обмеженою пропускною здатністю інтернету або для користувачів з пристроями з малим обсягом сховища.

Статистичний аналіз стабільності вимірювань виявив, що традиційний підхід демонструє більш передбачувані результати для гарячого та теплого запуску, тоді як декларативний підхід показує вищу стабільність при холодному запуску з профілем базової лінії. Така різниця вказує на більш детерміновану природу механізмів відновлення стану у Views для простих сценаріїв, тоді як оптимізована компіляція Compose забезпечує стабільнішу продуктивність при повному запуску застосунку.

Виявлені обмеження дослідження, зокрема неможливість автоматизованого аналізу використання пам'яті через особливості програмного інтерфейсу TraceSectionMetric та несумісність інструментів вимірювання продуктивності прокручування з версією операційної системи Android 15, не впливають критично на основні висновки роботи. Отримані дані з продуктивності запуску та розміру застосунку є достатньо комплексними для оцінки відмінностей між підходами,

оскільки ці показники найбезпосередніше впливають на досвід кінцевих користувачів.

На основі результатів дослідження сформульовано практичні рекомендації щодо вибору підходу залежно від специфіки проекту. Декларативний підхід Jetpack Compose рекомендується для нових проектів з тривалим циклом підтримки, де перевага у продуктивності повторних запусків та зручність розробки переважають недоліки збільшеного розміру пакунку. Традиційний підхід Android Views залишається доцільним для застосунків з жорсткими обмеженнями на розмір або де критично важливий швидкий перший запуск. Гібридний підхід може бути оптимальним рішенням для складних проектів або поступової міграції існуючих застосунків, дозволяючи поєднувати переваги обох технологій.

Результати дослідження підтверджують, що вибір між декларативним та імперативним підходами до побудови користувацького інтерфейсу не може базуватися на універсальних критеріях, оскільки кожен підхід має специфічні переваги та обмеження. Обґрунтоване рішення повинно враховувати конкретні вимоги проекту, характеристики цільової аудиторії та пріоритети між різними аспектами продуктивності застосунку.

ВИСНОВКИ

У кваліфікаційній роботі проведено комплексне дослідження продуктивності мобільних застосунків при використанні декларативного та імперативного підходів до побудови користувацького інтерфейсу. Виконання поставлених завдань дозволило отримати науково обґрунтовані висновки та практичні рекомендації для розробників мобільних застосунків на платформі Android.

Теоретичний аналіз фундаментальних принципів програмування користувацьких інтерфейсів виявив концептуальну різницю між підходами. Імперативна парадигма базується на принципі «як досягти результату» з явним управлінням станом через мутабельні об'єкти, тоді як декларативна фокусується на принципі "що має бути результатом" з автоматичною синхронізацією через спостереження стану. Архітектурні відмінності між Android View System та Jetpack Compose полягають у способах організації коду та механізмах оптимізації. Views використовує XML-орієнтовану архітектуру з процесом інфляції макетів, тоді як Compose базується на функціональній композиції.

Розроблена методика експериментального дослідження забезпечила систематичний підхід до порівняння продуктивності обох реалізацій. Комбінація сучасних інструментів профілювання та методологій бенчмаркінгу дозволила отримати багаторівневу оцінку продуктивності. Реалізація тестового застосунку з повною функціональною та візуальною ідентичністю для обох підходів дозволила ізолювати вплив способу побудови інтерфейсу на показники продуктивності при збереженні єдиної бізнес-логіки.

Експериментальне тестування продуктивності запуску виявило суттєві відмінності між підходами, характер яких залежить від типу запуску. Декларативний підхід демонструє переконливу перевагу для гарячого та теплого запуску, що обумовлено ефективнішим механізмом відновлення стану через легші композиційні функції та уникненням повторної інфляції XML-розмітки. Протилежна картина спостерігається при холодному запуску, де традиційний підхід виявляється швидшим, особливо за відсутності профілю базової лінії. Це

відображає оптимізацію традиційної системи на рівні платформи та необхідність ініціалізації власного середовища виконання для Compose.

Аналіз розміру застосунків виявив найбільш суттєву відмінність між підходами. Compose-застосунок більше ніж удвічі перевищує Views-застосунок за розміром пакунку, при цьому розмір завантаження зростає ще більше. Детальний розбір компонентів показав, що збільшення обумовлене включенням середовища виконання декларативного фреймворку, яке додає десятки тисяч додаткових методів та потребує використання багатьох DEX-файлів. При цьому власний код застосунку у декларативній реалізації виявився значно компактнішим, що підтверджує лаконічність синтаксису композиційних функцій.

Дослідження підтвердило критичну важливість профілю базової лінії для декларативних застосунків, оскільки його використання суттєво покращує продуктивність холодного запуску, тоді як для традиційного підходу ефект є мінімальним. Статистичний аналіз стабільності результатів показав, що Views демонструє більш передбачувані результати для повторних запусків, тоді як Compose забезпечує вищу стабільність при холодному запуску з оптимізацією.

На основі результатів дослідження сформульовано практичні рекомендації, які надають розробникам науково обґрунтовані критерії для вибору технології. Jetpack Compose рекомендується для нових проектів з тривалим циклом підтримки та складним реактивним інтерфейсом, де перевага у продуктивності повторних запусків та зручність розробки переважають недоліки збільшеного розміру. Android Views залишається доцільним для застосунків з жорсткими обмеженнями на розмір пакунку, критично важливим швидким першим запуском або підтримкою старіших пристроїв. Гібридний підхід може бути оптимальним для поступової міграції існуючих застосунків.

Наукова новизна роботи полягає у проведенні комплексного порівняння продуктивності Jetpack Compose та Android Views з використанням сучасних спеціалізованих інструментів, що дозволило емпірично підтвердити теоретичні переваги декларативного підходу для певних сценаріїв, виявити суттєві відмінності у продуктивності холодного запуску та розмірі застосунків, а також розробити

методику тестування продуктивності UI-каркасів. Практична значущість полягає в розробці набору науково обґрунтованих рекомендацій, які можуть бути використані як система критеріїв для вибору технології та настанови щодо оптимізації продуктивності мобільних застосунків.

Результати дослідження підтверджують, що вибір між декларативним та імперативним підходами не може базуватися на універсальних критеріях. Кожен підхід має специфічні переваги та обмеження, тому обґрунтоване рішення повинно враховувати конкретні вимоги проекту, характеристики цільової аудиторії та пріоритети між різними аспектами продуктивності застосунку.

Подальші дослідження можуть бути присвячені до наступних напрямків: аналіз використання пам'яті та профілюванні купи (heap) для виявлення прихованих різниць між підходами; пряме вимірювання часу розробки та метрик складності коду для підтвердження заявлених переваг Compose; дослідження еволюції продуктивності з кожною версією Compose як відносно молодої та перспективної технології.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Zhou T., Zhao Y., Chen J., Wang L., Li X., Zhang M. Bridging Design and Development with Automated Declarative UI Code Generation. arXiv. 2024. Vol. 2409.11667. DOI: https://doi.org/10.48550/arXiv.2409.11667
2. Jošt G., Taneski V. State-of-the-Art Cross-Platform Mobile Application Development Frameworks: A Comparative Study of Market and Developer Trends. Informatics. 2025. Vol. 12, No. 2. – P. 45. DOI: https://doi.org/10.3390/informatics12020045
3. Introduction to Declarative UI [Електронний ресурс]. Flutter Documentation. 2025. URL: https://docs.flutter.dev/get-started/flutter-for/declarative
4. The Reactive Manifesto [Електронний ресурс]. 2014. URL: https://www.reactivemanifesto.org
5. State Management [Електронний ресурс]. React Documentation. Accessed: November 6, 2025. URL: https://react.dev/learn/managing-state
6. Levien R. Towards a unified theory of reactive UI [Електронний ресурс]. 2019. URL: https://raphlinus.github.io/ui/druid/2019/11/22/reactive-ui.html
7. Jetpack Compose architectural layering [Електронний ресурс]. Android Developers. 2025. URL: https://developer.android.com/develop/ui/compose/layering
8. What's New in Jetpack Compose [Електронний ресурс]. Android Developers Blog. 2024. URL: https://android-developers.googleblog.com/2025/05/whats-new-in-jetpack-compose.html
9. Steinberger P. The Shift From Imperative to Declarative UI [Електронний ресурс]. Increment. 2021. URL: https://increment.com/mobile/the-shift-to-declarative-ui/
10. Samancioglu A. Reactive Programming Choices in Native Mobile Application Development. Master's thesis, Harvard University Division of Continuing Education. 2024. URL: https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37378504

11. Android Developers. Compare Compose and View metrics [Електронний ресурс]. URL: https://developer.android.com/develop/ui/compose/migrate/compare-metrics
12. Android Developers Blog. Play Time with Jetpack Compose [Електронний ресурс]. URL: https://android-developers.googleblog.com/2022/03/play-time-with-jetpack-compose.html
13. IT Ukraine Association. Ukrainian IT Industry Report [Електронний ресурс]. URL: https://itukraine.org.ua/
14. Android Developers. Android Studio Profiler. URL: https://developer.android.com/studio/profile
15. Android Developers. Jetpack Benchmark Library. URL: https://developer.android.com/topic/performance/benchmarking
16. Barrett E., Bolz-Tereick C. F., Killick R., Mount S., Tratt L. Virtual Machine Warmup Blows Hot and Cold. Proceedings of the ACM on Programming Languages, 2017. Vol. 1, OOPSLA. Article 52. DOI: 10.1145/3133876.
17. Bergman A., Ekström A. UI Performance Comparison of Jetpack Compose and XML in Native Android Applications. Bachelor's Thesis. KTH Royal Institute of Technology, 2023. TRITA-CBH-GRU 2023:096. URL: https://kth.diva-portal.org/smash/record.jsf?pid=diva2:1763502.
18. Cook T. D., Campbell D. T. Quasi-experimentation: Design and analysis issues for field settings. Boston: Houghton Mifflin, 1979. 405 p.
19. Ferreira J., Santos A., Machado R. Analyzing the Resource Usage Overhead of Mobile App Development Frameworks. Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering (EASE '23). ACM, 2023. P. 234–243. DOI: 10.1145/3593434.3593487.
20. Heitkötter H., Majchrzak T. A., Kuchen H. An empirical investigation of performance overhead in cross-platform mobile development frameworks. Empirical Software Engineering, 2020. Vol. 25. P. 2997–3031. DOI: 10.1007/s10664-020-09827-6.

21. Linares-Vásquez M., Bavota G., Tufano M., Moran K., Di Penta M., Vendome C., Bernal-Cárdenas C., Poshyvanyk D. A large-scale empirical study on mobile performance: energy, run-time and memory. Empirical Software Engineering, 2023. Vol. 28. Article 89. DOI: 10.1007/s10664-023-10391-y.
22. Perfetto. System Tracing Documentation. URL: https://perfetto.dev/docs/
23. Traini L., Tempero E., Dietrich J., Koschke R., Alecci S., Di Penta M. Preliminary investigation on approaches for Java warm-up. ArXiv preprint, 2022. arXiv:2209.15369. URL: https://arxiv.org/abs/2209.15369.
24. Земан А.О. Порівняльний аналіз продуктивності декларативного та імперативного підходів до побудови користувацьких інтерфейсів мобільних застосунків. Матеріали VI Міжнародної науково-практичної конференції «Кібербезпека в сучасному світі: актуальні виклики». 28 листопада 2025 р, м. Одеса, Україна [депоновано].

ДОДАТКИ

Додаток А. Програмний код модуля benchmark

BaselineProfileGenerator.kt :

```
package com.zeman.benchmark

import androidx.benchmark.macro.junit4.BaselineProfileRule
import androidx.test.ext.junit.runners.AndroidJUnit4
import androidx.test.filters.LargeTest
import androidx.test.uiautomator.By
import androidx.test.uiautomator.Direction
import androidx.test.uiautomator.Until
import org.junit.Rule
import org.junit.Test
import org.junit.runner.RunWith

/**
 * Генерує базовий профіль для оптимізації програми.
 *
 * Базові профілі повідомляють середовищу виконання Android, які
шляхи коду є активними (часто виконуються),
 * що дозволяє завчасно компілювати критичний код для кращої
продуктивності запуску та виконання.
 *
 * Щоб згенерувати базовий профіль:
 * 1. Підключіть рутований пристрій або скористайтеся емулятором з
Android 11+
 * 2. Виконайте: ./gradlew :app-views:generateBaselineProfile
 * або: ./gradlew :app-compose:generateBaselineProfile
 *
 * Згенерований профіль буде знаходитися в:
 * app-{views|compose}/src/main/baseline-prof.txt
 */
@LargeTest
@RunWith(AndroidJUnit4::class)
class BaselineProfileGenerator {

    @get:Rule
    val baselineProfileRule = BaselineProfileRule()

    @Test
```

```
    fun generate() = baselineProfileRule.collect(
        packageName = TARGET_PACKAGE,
        maxIterations = 15,
        stableIterations = 3,
        includeInStartupProfile = true
    ) {
        // Старт застосунку
        pressHome()
        startActivityAndWait()

        // Зачекати на початковий контент
        device.wait(Until.hasObject(By.res("recyclerView")), 5000)
            ?: device.wait(Until.hasObject(By.scrollable(true)),
5000)

        // Сценарій 1: Прокручування списку продуктів
        val productList = device.findObject(By.res("recyclerView"))
            ?: device.findObject(By.scrollable(true))

        productList?.apply {
            // Прокрутити вниз, щоб запустити рендеринг
RecyclerView/LazyColumn
            repeat(3) {
                scroll(Direction.DOWN, 0.8f)
                device.waitForIdle()
            }

            // Прокрутити назад угору
            repeat(3) {
                scroll(Direction.UP, 0.8f)
                device.waitForIdle()
            }
        }

        // Сценарій 2: Перехід до пошуку
        device.findObject(By.text("Search"))?.click()
            ?: device.findObject(By.desc("Search"))?.click()
        device.waitForIdle()

        // Взаємодія з пошуком
        device.findObject(By.res("searchInput"))?.text = "test"
        device.waitForIdle()

        // Сценарій 3: Перехід до кошика
        device.findObject(By.text("Cart"))?.click()
            ?: device.findObject(By.desc("Cart"))?.click()
        device.waitForIdle()
```

```
        // Сценарій 4: Повернення до продуктів
        device.findObject(By.text("Products"))?.click()
            ?: device.findObject(By.desc("Products"))?.click()
        device.waitForIdle()
    }

    companion object {
        private const val TARGET_PACKAGE = "com.zeman.catalog.views"
    }
}
```

StartupBenchmark.kt :

```
package com.zeman.benchmark

import androidx.benchmark.macro.BaselineProfileMode
import androidx.benchmark.macro.CompilationMode
import androidx.benchmark.macro.StartupMode
import androidx.benchmark.macro.StartupTimingMetric
import androidx.benchmark.macro.junit4.MacrobenchmarkRule
import androidx.test.ext.junit.runners.AndroidJUnit4
import androidx.test.filters.LargeTest
import org.junit.Rule
import org.junit.Test
import org.junit.runner.RunWith

/**
 * Тест для вимірювання часу запуску програми.
 *
 * Цей тест вимірює час, необхідний для запуску програми та
відображення UI першого вмісту.
 *
 * Запустіть цей тест, щоб зафіксувати:
 * - timeToInitialDisplay: Час до візуалізації першого кадру
 * - timeToFullyDrawn: Час до повного відображення програми
 *
 * Щоб запустити цей тест:
 * 1. Підключіть фізичний пристрій (не емулятор для отримання точних
результатів)
 * 2. Запустіть: ./gradlew :benchmark:connectedBenchmarkAndroidTest
 *
 * Результати будуть у:
benchmark/build/outputs/connected_android_test_additional_output/
 */
@LargeTest
@RunWith(AndroidJUnit4::class)
```

```
class StartupBenchmark {

    @get:Rule
    val benchmarkRule = MacrobenchmarkRule()

    /**
     * Холодний запуск без будь-якої компіляції/оптимізації.
     * Це відображає найгірший випадок продуктивності запуску.
     */
    @Test
    fun startupCompilationNone() = startup(
        compilationMode = CompilationMode.None(),
        startupMode = StartupMode.COLD
    )

    /**
     * Холодний запуск з оптимізацією базового профілю.
     * Це відображає оптимізовану виробничу продуктивність.
     */
    @Test
    fun startupCompilationBaselineProfile() = startup(
        compilationMode =
CompilationMode.Partial(BaselineProfileMode.Require),
        startupMode = StartupMode.COLD
    )

    /**
     * Теплий запуск — процес програми все ще активний у фоновому
режимі.
     */
    @Test
    fun startupWarm() = startup(
        compilationMode =
CompilationMode.Partial(BaselineProfileMode.Require),
        startupMode = StartupMode.WARM
    )

    /**
     * Гарячий запуск — програма переноситься з фону на передній
план.
     */
    @Test
    fun startupHot() = startup(
        compilationMode =
CompilationMode.Partial(BaselineProfileMode.Require),
        startupMode = StartupMode.HOT
    )
```

```
    private fun startup(
        compilationMode: CompilationMode,
        startupMode: StartupMode
    ) = benchmarkRule.measureRepeated(
        packageName = TARGET_PACKAGE,
        metrics = listOf(StartupTimingMetric()),
        compilationMode = compilationMode,
        startupMode = startupMode,
        iterations = 15, // Виконати 15 разів для статистичної
значущості
        setupBlock = {
            // Натисніть кнопку "Додому", щоб переконатися, що ми
починаємо з узгодженого стану
            pressHome()
        }
    ) {
        // Запустити основну activity та чекати на її відображення
        startActivityAndWait()

        // Застосунок викликає reportFullyDrawn() в MainActivity,
        // що сигналізує про вимірювання timeToFullyDrawn
    }

    companion object {
        private const val TARGET_PACKAGE = "com.zeman.catalog.views"
    }
}
```

MemoryBenchmark.kt :

```
package com.zeman.benchmark

import androidx.benchmark.macro.CompilationMode
import androidx.benchmark.macro.ExperimentalMetricApi
import androidx.benchmark.macro.StartupMode
import androidx.benchmark.macro.junit4.MacrobenchmarkRule
import androidx.test.ext.junit.runners.AndroidJUnit4
import androidx.test.uiautomator.By
import androidx.test.uiautomator.Direction
import androidx.test.uiautomator.Until
import org.junit.Rule
import org.junit.Test
import org.junit.runner.RunWith

/**
 * Тест пам'яті для порівняння використання пам'яті між реалізаціями
```

```
Views та Compose.
 *
 * Вимірювання:
 * - Розподіл пам'яті під час операцій інтерфейсу користувача
 * - Використання пам'яті купи
 * - Події GC
 *
 * Результати можна аналізувати в Android Studio Profiler або через
трасування Perfetto.
 */
@RunWith(AndroidJUnit4::class)
class MemoryBenchmark {

    @get:Rule
    val benchmarkRule = MacrobenchmarkRule()

    @OptIn(ExperimentalMetricApi::class)
    @Test
    fun memoryDuringScrolling() = benchmarkRule.measureRepeated(
        packageName = TARGET_PACKAGE,
        metrics = listOf(
            // TraceSectionMetric фіксує інформацію про розподіл
пам'яті з трасування
            androidx.benchmark.macro.TraceSectionMetric(
                "Choreographer#doFrame",
                androidx.benchmark.macro.TraceSectionMetric.Mode.Sum
            )
        ),
        iterations = 10,
        startupMode = StartupMode.WARM,
        compilationMode = CompilationMode.Partial(),
        setupBlock = {
            pressHome()
            startActivityAndWait()

            // Зачекати на завантаження контенту
            device.wait(Until.hasObject(By.scrollable(true)), 5000)
        }
    ) {
        val scrollable = device.findObject(By.scrollable(true))

        if (scrollable != null) {
            // Виконайте прокручування для запуску розподілу
            repeat(5) {
                scrollable.setGestureMargin(device.displayWidth / 5)
                scrollable.scroll(Direction.DOWN, 1.0f)
                device.waitForIdle(100)
            }
```

```
            repeat(5) {
                scrollable.scroll(Direction.UP, 1.0f)
                device.waitForIdle(100)
            }
        }
    }

    @OptIn(ExperimentalMetricApi::class)
    @Test
    fun memoryDuringNavigation() = benchmarkRule.measureRepeated(
        packageName = TARGET_PACKAGE,
        metrics = listOf(
            androidx.benchmark.macro.TraceSectionMetric(
                "activityStart",
                androidx.benchmark.macro.TraceSectionMetric.Mode.Sum
            )
        ),
        iterations = 10,
        startupMode = StartupMode.WARM,
        compilationMode = CompilationMode.Partial(),
        setupBlock = {
            pressHome()
            startActivityAndWait()
            device.wait(Until.hasObject(By.scrollable(true)), 5000)
        }
    ) {
        val scrollable = device.findObject(By.scrollable(true))

        if (scrollable != null) {
            // Прокрутіть, щоб знайти потрібний товар
            scrollable.scroll(Direction.DOWN, 0.3f)
            device.waitForIdle(500)

            // Знайдіть і натисніть перший елемент, який не є полем
пошуку
            val items = device.findObjects(By.clickable(true))
            // Пропустіть перші кілька пунктів (пошук, фільтри) та
натисніть на товар
            if (items.size > 3) {
                items[3].click()
                device.waitForIdle(1000)

                // Повернутися назад
                device.pressBack()
                device.waitForIdle(1000)
            }
        }
```

```
    }

    @OptIn(ExperimentalMetricApi::class)
    @Test
    fun memoryDuringStartup() = benchmarkRule.measureRepeated(
        packageName = TARGET_PACKAGE,
        metrics = listOf(
            androidx.benchmark.macro.StartupTimingMetric(),
            androidx.benchmark.macro.TraceSectionMetric(
                "activityStart",
                androidx.benchmark.macro.TraceSectionMetric.Mode.Sum
            )
        ),
        iterations = 10,
        startupMode = StartupMode.COLD,
        compilationMode = CompilationMode.Partial(),
        setupBlock = {
            pressHome()
        }
    ) {
        startActivityAndWait()
        device.waitForIdle()
    }

    companion object {
        private const val TARGET_PACKAGE = "com.zeman.catalog.views"
        // Для Compose: змінити на "com.zeman.catalog.compose"
    }
}
```

Додаток Б. Копії матеріалів апробації роботи

Земан Артур Олегович

Національний університет «Одеська юридична академія»,

студент факультету кібербезпеки та інформаційних технологій

ПОРІВНЯЛЬНИЙ АНАЛІЗ ПРОДУКТИВНОСТІ ДЕКЛАРАТИВНОГО ТА ІМПЕРАТИВНОГО ПІДХОДІВ ДО ПОБУДОВИ КОРИСТУВАЦЬКИХ ІНТЕРФЕЙСІВ МОБІЛЬНИХ ЗАСТОСУНКІВ

Індустрія розробки мобільних застосунків переживає фундаментальний зсув у підходах до створення користувацьких інтерфейсів. Протягом багатьох років домінував імперативний підхід, представлений в Android традиційною системою View. Однак останнім часом все більшої популярності набуває декларативна парадигма, яскравим представником якої є фреймворк Jetpack Compose від Google. Великі компанії вже здійснили міграцію на Jetpack Compose, проте бракує комплексних академічних досліджень, які б надавали кількісну оцінку продуктивності декларативного підходу в порівнянні з імперативним з використанням сучасних інструментів для бенчмаркінгу.

Метою дослідження є порівняльний аналіз продуктивності мобільних застосунків при використанні декларативного та імперативного підходів до побудови користувацьких інтерфейсів та розробка практичних рекомендацій для розробників. Для досягнення мети необхідно було проаналізувати теоретичні основи та архітектурні відмінності підходів, розробити методологію експериментального дослідження, провести тестування продуктивності за ключовими показниками та здійснити статистичний аналіз отриманих даних.

Методологія дослідження базується на комплексному використанні сучасних спеціалізованих інструментів профілювання та бенчмаркінгу. Для вимірювання продуктивності застосовувалася бібліотека Jetpack Macrobenchmark [1], Android Studio Profiler для профілювання системних ресурсів, Perfetto для системного трасування та JankStats API для моніторингу плавності інтерфейсу. Визначено чотири критичні категорії вимірювань: метрики продуктивності під час виконання, метрики пам'яті, метрики збірки та метрики енергоспоживання. Критичним

аспектом методології стало використання виключно релізних збірок із увімкненою оптимізацією, оскільки Debug-збірки демонструють продуктивність понад удвічі повільнішу [2].

Експериментальне дослідження охопило аналіз реальних промислових впроваджень. Команда OkCredit провела дослідження на реальних користувачах, де екран на основі Compose без попередньої компіляції завантажувався у два з половиною рази довше порівняно з традиційною системою [3]. Водночас після попередньої компіляції компонентів спостерігалося значне скорочення заморожених кадрів та суттєве покращення часу завантаження завдяки механізму попередньої компіляції Android Runtime [3].

Офіційна документація Android надає рекомендації щодо порівняння метрик продуктивності [4]. Аналіз зразкового додатку Sunflower показав, що при додаванні Compose розмір пакету збільшується приблизно на третину. Проте реальний приклад додатку Tivi демонструє, що після повної міграції розмір пакету скоротився майже вдвічі, а час збірки покращився приблизно на третину [5].

Дослідження команди Premise Engineering виявило критичну важливість конфігурації збірки [2]. Збірки з вимкненою оптимізацією виконувалися понад удвічі довше, що підкреслює необхідність тестування виключно в релізному режимі [6]. Базові профілі виявилися критичним фактором для досягнення конкурентної продуктивності, покращуючи швидкість виконання коду приблизно на третину від першого запуску [7].

Порівняння компонентів списків LazyColumn та RecyclerView показало, що холодний запуск значно повільніший з Compose, проте продуктивність прокручування практично ідентична [8]. Дослідження оптимізації управління станом виявило можливість значного скорочення споживання пам'яті при використанні правильних патернів [9].

Статистичний аналіз виявив два різні режими продуктивності Compose: початковий без належної конфігурації з деградацією продуктивності та оптимізований з попередньою компіляцією, що досягає найкращих показників. Кореляційний аналіз демонструє стабільну тенденцію поліпшення з кожним релізом.

На основі отриманих результатів сформульовано практичні рекомендації. Для нових проєктів рекомендовано використання Jetpack Compose завдяки прискоренню циклу розробки та скороченню обсягу коду. Для існуючих застосунків доцільна поступова міграція. Критично важливим є увімкнення повної оптимізації коду, створення базових профілів та дотримання принципів стабільності параметрів композиційних функцій.

Результати дослідження підтверджують, що декларативний підхід досяг паритету з імперативним у більшості метрик продуктивності та демонструє переваги у критичних сценаріях після належної оптимізації. Розроблена методологія може використовуватися для оцінки продуктивності інших UI-фреймворків.

Список використаної літератури:

1. Android Developers. Write a Macrobenchmark. URL: https://developer.android.com/topic/performance/benchmarking/macrobenchmark-overview (дата звернення: 10.11.2025).
2. Shelor W. Measuring Render Performance with Jetpack Compose / Will Shelor // Premise Engineering Blog. 2021. October 26. URL: https://engineering.premise.com/measuring-render-performance-with-jetpack-compose-c0bf5814933
3. Arora P. Comparing Jetpack Compose performance with XML / Pratham Arora // Medium — OkCredit. 2022. August 9. URL: https://medium.com/okcredit/comparing-jetpack-compose-performance-with-xml-9462a1282c6b
4. Android Developers. Compare Compose and View metrics. URL: https://developer.android.com/develop/ui/compose/migrate/compare-metrics (дата звернення: 10.11.2025).
5. Banes C. Jetpack Compose — Before and after / Chris Banes // Medium — Android Developers. 2021. July 7. URL: https://medium.com/androiddevelopers/jetpack-compose-before-and-after-8b43ba0b7d4f

6. Trengrove B. Why should you always test Compose performance in release? / Ben Trengrove // Medium — Android Developers. 2022. June 8. URL: https://medium.com/androiddevelopers/why-should-you-always-test-compose-performance-in-release-4168dd0f2c71
7. Semenova K., Ravikumar R., Craik C. Improving App Performance with Baseline Profiles // Android Developers Blog. 2022. January 27. URL: https://android-developers.googleblog.com/2022/01/improving-app-performance-with-baseline.html
8. MarDSoul. RecyclerView vs LazyColumn / MarDSoul // DEV Community. 2024. July 25. URL: https://dev.to/mardsoul/recyclerview-vs-lazycolumn-5g6g
9. Patil S. Benchmark Insights: Direct State Propagation vs. Lambda-based State in Jetpack Compose / Shreyas Patil // Personal Blog. 2024. November 19. URL: https://blog.shreyaspatil.dev/benchmark-insights-direct-state-propagation-vs-lambda-based-state-in-jetpack-compose

Ключові слова: Jetpack Compose, Android Views, декларативний підхід, продуктивність, бенчмаркінг, рекомпозиція, оптимізація.

Keywords: Jetpack Compose, Android Views, declarative approach, performance, benchmarking, recomposition, optimization.