International Science Journal of Engineering & Agriculture
2026; 5(3): 124-135
https://isg-journal.com/isjea
doi: 10.46299/j.isjea.20260503.11
ISSN: 2720-6319

INTERNATIONAL SCIENCE GROUP

Дослідження архітектурних підходів до розробки крос-платформних мобільних застосунків з використанням Kotlin Multiplatform

Іван Мороз
Факультет комп’ютерних та інформаційних технологій, Луцький національний технічний університет, Луцьк, Україна
ORCID: 0009-0009-5552-071X

Ігор Андрущак
Факультет комп’ютерних та інформаційних технологій, Луцький національний технічний університет, Луцьк, Україна
ORCID: 0000-0002-8751-4420

Анотація: У статті досліджено архітектурні підходи до розробки крос-платформних мобільних застосунків з використанням технології Kotlin Multiplatform (KMM). Сучасний стан мобільної розробки характеризується технологічним дуалізмом двох домінантних платформ – Android та iOS, які використовують принципово різні мови програмування, фреймворки та платформо-залежні інтерфейси прикладного програмування. Така ситуація зумовлює подвоєння трудовитрат, дублювання помилок та зростання вартості супроводу програмного продукту. Розглянуто еволюцію крос-платформних рішень: гібридні підходи на основі WebView (Apache Cordova, PhoneGap, Ionic), технологія Xamarin із використанням C# та середовища виконання Mono, сучасні фреймворки React Native і Flutter. Встановлено, що кожен з наведених підходів містить істотні компроміси у площинах продуктивності, природності користувацького досвіду та інтеграції з нативними інтерфейсами прикладного програмування. Технологія Kotlin Multiplatform від компанії JetBrains реалізує принципово відмінний підхід, який полягає у спільному використанні бізнес-логіки на мові Kotlin за умови збереження повністю нативних шарів користувацького інтерфейсу. У роботі запропоновано багатомодульну архітектуру KMM-застосунку із чітким розділенням на шари презентації, домену, даних та інфраструктури. Проаналізовано механізм expect/actual як основний інструмент роботи з платформо-залежним кодом. Розглянуто роль ключових інфраструктурних бібліотек екосистеми Kotlin: Ktor для виконання мережевих операцій, SQLDelight для роботи з реляційними базами даних, kotlinx.coroutines для асинхронного виконання, Koin для впровадження залежностей, kotlinx.serialization для серіалізації даних. Виконано порівняльне оцінювання чотирьох підходів (нативна розробка, KMM, Flutter, React Native) за критеріями часу холодного запуску, розміру бінарного файлу, споживання оперативної пам’яті та частки спільного коду. Отримані результати засвідчили, що Kotlin Multiplatform забезпечує продуктивність, наближену до нативної, з приростом часу холодного запуску близько 13 % за умови досягнення частки спільного коду до 75 % у типовому застосунку. На підставі отриманих даних обґрунтовано доцільність використання KMM у проєктах, для яких якість користувацького досвіду та продуктивність є критичними, а підтримка двох окремих кодових баз стає економічно невиправданою. Окремо проаналізовано перспективи подальшого розвитку напряму, зокрема впровадження Compose Multiplatform як єдиного фреймворка користувацького інтерфейсу, інтеграцію зі засобами машинного навчання на пристрої та поширення KMM у корпоративному середовищі.

Ключові слова: Kotlin Multiplatform; крос-платформна розробка; мобільні застосунки; архітектура програмного забезпечення; механізм expect/actual; спільний код; багатомодульна архітектура; продуктивність мобільних застосунків.

1. Вступ

Сучасна мобільна розробка характеризується технологічним дуалізмом двох провідних платформ – Android та iOS, кожна з яких побудована на власній екосистемі інструментів та фреймворків. Платформа Android традиційно базується на мові Kotlin (раніше – Java) у поєднанні з комплектом розробника Android SDK та фреймворком Jetpack. Платформа iOS використовує мову Swift (раніше – Objective-C) і власні фреймворки компанії Apple [1]. Відсутність спільного коду між цими платформами зумовлює необхідність дублювання значного обсягу робіт – бізнес-логіки, мережевих взаємодій, кешування, валідації даних. Як наслідок, виникає подвоєння трудовитрат на розробку, тестування й супровід програмного продукту, а також пропорційне зростання витрат на реалізацію кожної нової функціональної можливості.

Прагнення подолати такий стан речей зумовило виникнення низки крос-платформних підходів, які можна умовно розподілити на чотири покоління.

Перше покоління становлять гібридні рішення на основі WebView – Apache Cordova, PhoneGap, Ionic. Програмне забезпечення розробляється засобами HTML, CSS і JavaScript, а у застосунку виконується вбудований компонент WebView [2]. Незважаючи на спрощення процесу розробки, для таких рішень характерні низька продуктивність, неприродний вигляд інтерфейсу на нативних платформах та обмежений доступ до системних інтерфейсів прикладного програмування.

Друге покоління представлене технологією Xamarin компанії Microsoft, у межах якої програмний код на мові C# компілюється у нативний байт-код через середовище виконання Mono. Такий підхід забезпечує вищу продуктивність порівняно з WebView-рішеннями, проте призводить до суттєвого збільшення розміру бінарного файлу через необхідність включення середовища виконання Mono. Крім того, екосистема Xamarin традиційно розвивалася відокремлено від основних спільнот розробників Android та iOS [3].

Третє покоління становлять фреймворки React Native та Flutter. React Native виконує програмний код мовою JavaScript у спеціалізованому середовищі та звертається до нативних компонентів через міст-механізм. Фреймворк Flutter повністю відмовляється від нативних компонентів користувацького інтерфейсу та використовує власний рендеринговий двигун Skia. Обидва підходи забезпечують високу частку спільного коду, проте характеризуються специфічними обмеженнями: для React Native – накладні витрати міст-механізму, для Flutter – значний розмір бінарного файлу та ускладнена інтеграція з платформо-залежними інтерфейсами прикладного програмування [4].

Четверте покоління представлене технологією Kotlin Multiplatform (KMM) компанії JetBrains, яка у жовтні 2023 року отримала стабільний статус. Філософія KMM полягає у відмові від уніфікації шару користувацького інтерфейсу та зосередженні зусиль на спільному використанні виключно бізнес-логіки, реалізованої мовою Kotlin. Спільний код компілюється у байт-код віртуальної машини Java для платформи Android та у машинний код для платформи iOS засобами Kotlin/Native [5]. Шар користувацького інтерфейсу залишається повністю нативним: Jetpack Compose або XML-розмітка на платформі Android, SwiftUI або UIKit – на платформі iOS.

Ключовою перевагою такого підходу є збереження природного користувацького досвіду на кожній платформі за одночасного спільного використання шарів даних, мережевих взаємодій та бізнес-логіки. За оцінками практичних проєктів, частка спільного коду у типовому KMM-застосунку становить від 60 до 80 відсотків, причому йдеться саме про той код, супровід якого є найбільш трудомістким.

Водночас успіх KMM-проєкту критично залежить від обґрунтованості архітектурних рішень. Невдалий вибір модульної структури, нераціональне застосування механізму expect/actual, нехтування особливостями середовища Kotlin/Native можуть знівелювати переваги технології. У граничному випадку супровід KMM-проєкту може виявитися більш

складним, ніж супровід двох незалежних нативних кодових баз. Саме тому архітектурне обґрунтування є необхідною передумовою виробничого використання технології.

З огляду на швидкий розвиток екосистеми Kotlin Multiplatform та появу таких суміжних технологій, як Compose Multiplatform, дослідження архітектурних принципів побудови KMM-застосунків зберігає актуальність. Воно об'єднує одразу декілька дисциплін: інженерію програмного забезпечення, оцінювання продуктивності, проєктування модульних систем та формування практичних рекомендацій щодо вибору технологічного стека.

2. Об'єкт і предмет дослідження

Об'єктом дослідження є процес розробки крос-платформних мобільних застосунків, зокрема той його варіант, у якому спільний код мовою Kotlin поєднується з нативними реалізаціями для платформ Android та iOS. До складу об'єкта належать: проєктування модульної структури проєкту, організація шарів бізнес-логіки, налаштування системи збирання Gradle, інтеграція спеціалізованих бібліотек екосистеми Kotlin, реалізація платформо-залежного коду через механізм expect/actual, побудова нативних інтерфейсів засобами Jetpack Compose та SwiftUI, налаштування процесів безперервної інтеграції та доставки для одночасної публікації застосунку у магазинах Google Play та App Store.

У межах об'єкта розглядаються типові інженерні рішення, характерні для виробничих KMM-проєктів: рівень деталізації модульної декомпозиції, способи передачі даних між шарами, стратегії обробки помилок, методи тестування спільного коду, інтеграція засобів журналювання, аналітики та звітності про збої. Кожне з цих рішень безпосередньо впливає на якість, продуктивність та супровідність кінцевого продукту.

Окремий аспект становить взаємодія KMM-коду з платформо-залежними бібліотеками. Незважаючи на наявність у Kotlin/Native механізму прозорої взаємодії з Objective-C та Swift, на практиці існують суттєві обмеження: окремі особливості мови Swift є недоступними з Kotlin, модель пам'яті Kotlin/Native має специфічні правила, замикання працюють відмінно від очікувань Swift-розробників, асинхронний код у певних випадках має виконуватися в головному потоці. Ці обмеження визначають архітектурні рішення на рівні шарів спільного коду.

Предметом дослідження є архітектурні принципи побудови KMM-застосунків, які забезпечують максимізацію частки спільного коду без зниження якості користувацького досвіду, продуктивності та супровідності. До складу предмета належать: формалізація вимог до архітектури, обґрунтування багатомодульної декомпозиції, аналіз застосування механізму expect/actual, інтеграція інфраструктурних бібліотек, порівняльне оцінювання продуктивності щодо альтернативних крос-платформних підходів.

Окремою складовою предмета є пошук балансу між максимізацією частки спільного коду та збереженням нативного характеру користувацького досвіду. Надмірне розширення спільного модуля ускладнює інтеграцію з SwiftUI та платформо-залежними інтерфейсами прикладного програмування на стороні iOS. Натомість недостатнє розширення спільного коду не дозволяє повною мірою реалізувати переваги технології. Завданням дослідження є визначення рівня модульної декомпозиції, на якому переваги KMM реалізуються повною мірою, а недоліки залишаються керованими.

3. Мета та задачі дослідження

Метою роботи є обґрунтування архітектурних принципів побудови KMM-застосунків, що забезпечують максимальну частку спільного коду без зниження якості користувацького досвіду, продуктивності та супровідності. Окремий акцент зроблено на способах організації багатомодульної структури проєкту, раціональному використанні механізму expect/actual та інтеграції до запропонованої архітектури ключових бібліотек екосистеми Kotlin.

Для досягнення поставленої мети у роботі виконано аналіз сучасних крос-платформних підходів, сформульовано вимоги до архітектури KMM-застосунку, спроєктовано модульну структуру спільного коду, реалізовано прототип застосунку та виконано його порівняння з альтернативними рішеннями за об’єктивними метриками.

Поставлено такі задачі дослідження:

1) проаналізувати сучасний стан крос-платформної розробки та формалізувати функціональні й нефункціональні вимоги до архітектури KMM-застосунків;

2) розробити багатомодульну структуру спільного коду з обґрунтуванням взаємодії шарів, інтеграцією інфраструктурних бібліотек та раціональним застосуванням механізму expect/actual;

3) реалізувати прототип крос-платформного застосунку, виконати порівняльне оцінювання його продуктивності з альтернативними рішеннями та сформулювати рекомендації щодо розвитку напряму. У сукупності поставлені задачі охоплюють теоретичний аналіз та практичні вимірювання, що формує основу для обґрунтованих рекомендацій щодо застосування Kotlin Multiplatform у виробничих проєктах.

4. Аналіз літератури

Проблематика крос-платформної мобільної розробки досліджується у науковій літературі понад десять років. Накопичений масив результатів дозволяє простежити еволюцію підходів.

Ранні дослідження [6] зосереджувалися на гібридних рішеннях на основі WebView – Apache Cordova, PhoneGap, Ionic. Спільний код у межах цих підходів реалізовувався мовами HTML, CSS та JavaScript. Автори фіксували зростання швидкості розробки, однак відзначали низку істотних недоліків: низьку продуктивність, неприродний вигляд інтерфейсу на нативних платформах, обмежений доступ до системних інтерфейсів прикладного програмування.

Дослідження другого покоління крос-платформних рішень представлене роботами щодо технології Xamarin [7]. Компіляція коду мовою C# у нативний байт-код через середовище Mono забезпечила вищу продуктивність порівняно з WebView-рішеннями. Водночас постала проблема значного збільшення розміру бінарного файлу, зумовленого включенням середовища виконання Mono. Окремою складовою є набір Xamarin.Forms, який пропонував обмежений набір крос-платформних компонентів користувацького інтерфейсу та не покривав усієї сукупності нативних патернів проєктування.

Технологію React Native розглянуто у роботі [8]. Архітектура цього фреймворка ґрунтується на виконанні коду мовою JavaScript у спеціалізованому середовищі та виклику нативних компонентів через міст-механізм. Перевагою підходу автори вважають широку екосистему пакетів npm. Серед недоліків відзначено накладні витрати міст-механізму, складність синхронізації потоків та проблематичність налагодження міжмовних викликів. Нова архітектура (Fabric і JSI) частково усуває зазначені обмеження, проте міграція до неї пов’язана з істотними витратами.

Фреймворк Flutter проаналізовано у дослідженні [9]. Його принциповою особливістю є повна відмова від нативних компонентів користувацького інтерфейсу на користь власного рендерингового двигуна Skia. Автори фіксують високу частку спільного коду (понад 95 відсотків) та стабільну продуктивність на обох платформах. Водночас відзначено значний розмір бінарного файлу та ускладнену інтеграцію з платформо-залежними інтерфейсами прикладного програмування. Окремою проблемою є необхідність опанування мови Dart, що ускладнює залучення розробників, які мають досвід роботи з Android та iOS.

Технологію Kotlin Multiplatform розглянуто у роботі [10]. Її філософія полягає не в уніфікації шару користувацького інтерфейсу, а у спільному використанні виключно бізнес-логіки. Шар користувацького інтерфейсу залишається нативним, що забезпечує природний користувацький досвід, високу продуктивність та повну сумісність з платформо-залежними інтерфейсами прикладного програмування. Автори зазначають, що успіх KMM-проєкту

більшою мірою залежить від якості архітектурних рішень, ніж від характеристик самої технології.

Питанням архітектурних патернів для KMM присвячено дослідження [11], у якому розглянуто застосування принципів Clean Architecture у багатомодульних KMM-проєктах. Авторами рекомендовано чітке розділення шарів презентації, домену та даних, повне розміщення доменного шару у спільному модулі, а також часткове розміщення шару даних залежно від доступності платформо-залежних інтерфейсів прикладного програмування. У роботі [12] детально проаналізовано механізм expect/actual та сформульовано рекомендації щодо його застосування.

Інфраструктурні бібліотеки екосистеми Kotlin розглядаються окремо. Робота [13] присвячена бібліотеці Ktor – крос-платформному клієнту та серверу для роботи з протоколом HTTP. Дослідження [14] стосується бібліотеки SQLDelight, що генерує типобезпечний крос-платформний код для роботи з системою керування реляційними базами даних SQLite. Автори обох робіт констатують готовність зазначених бібліотек до виробничого використання.

Перспективи подальшого розвитку напряму пов’язані з технологією Compose Multiplatform [15]. Поява стабільної підтримки платформи iOS у 2024 році відкриває можливість спільного використання не лише бізнес-логіки, а й шару користувацького інтерфейсу. Водночас доцільність уніфікації UI-шару залишається предметом архітектурного аналізу, оскільки для окремих категорій застосунків нативна реалізація користувацького інтерфейсу зберігає переваги.

Огляд літератури свідчить про відсутність комплексних робіт, які б одночасно поєднували архітектурний аналіз, формалізацію модульної структури, обґрунтування використання механізму expect/actual та об’єктивне порівняльне оцінювання Kotlin Multiplatform. Цей пробіл обґрунтовує актуальність цієї роботи.

5. Методи досліджень

У роботі застосовано комбіновану методологію, яка поєднує порівняльний аналіз сучасних крос-платформних підходів, архітектурне проєктування KMM-застосунку, реалізацію прототипу та виконання об’єктивних бенчмарків продуктивності. Такий підхід забезпечує оцінювання технології Kotlin Multiplatform як з архітектурної, так і з виконавчої перспективи.

Перший етап методології становить аналіз вимог до KMM-застосунку. До функціональних вимог віднесено: спільне використання шарів даних та бізнес-логіки; підтримку асинхронних операцій засобами бібліотеки kotlinx.coroutines; інтеграцію з мережевими інтерфейсами прикладного програмування засобами бібліотеки Ktor; локальне зберігання даних засобами бібліотеки SQLDelight; керування залежностями засобами бібліотеки Koin; серіалізацію даних засобами бібліотеки kotlinx.serialization. Нефункціональні вимоги охоплюють: приріст часу холодного запуску не більше 20 відсотків щодо нативної реалізації; приріст споживання оперативної пам’яті у межах 15 відсотків; ефективну інтеграцію з нативними фреймворками користувацького інтерфейсу; можливість часткового перенесення наявних нативних проєктів на технологію KMM.

Другий етап становить архітектурне проєктування. На рис. 1 наведено запропоновану багатомодульну структуру KMM-застосунку. Архітектура поєднує платформо-залежні шари користувацького інтерфейсу, тонкі модулі-обгортки, спільні модулі бізнес-логіки та інфраструктури, а також actual-реалізації для конкретних платформ.

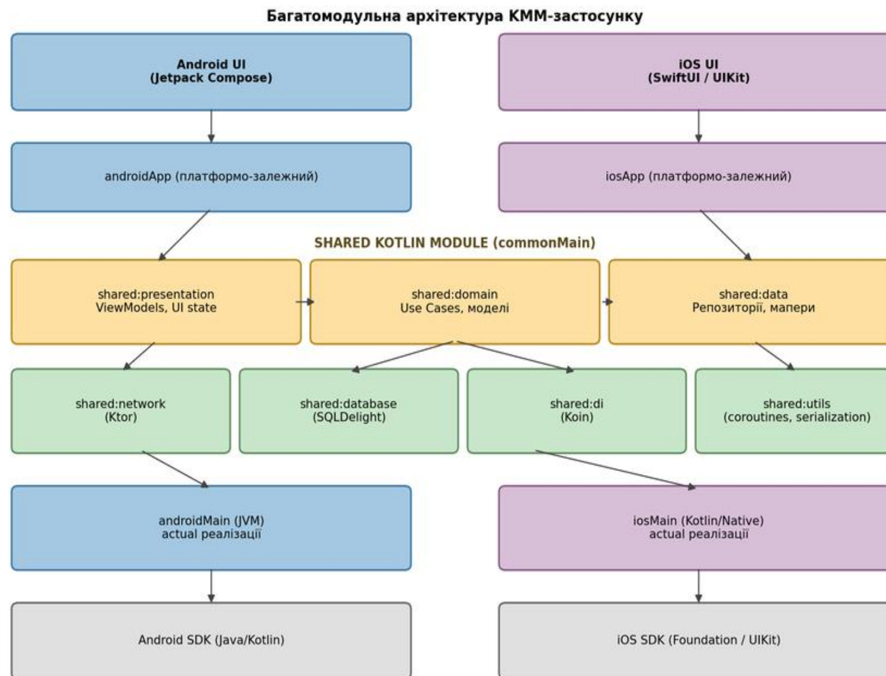

Рис. 1. Багатомодульна архітектура KMM-застосунку з нативними шарами користувацького інтерфейсу та спільним кодом бізнес-логіки

Розглянемо роль кожного з модулів у запропонованій архітектурі.

Модуль shared:presentation містить класи ViewModel та структури станів користувацького інтерфейсу. У ньому застосовано примітиви StateFlow та MutableStateFlow з бібліотеки kotlinx.coroutines. На стороні платформи Android їх можна безпосередньо інтегрувати з Jetpack Compose. На стороні платформи iOS потрібен тонкий шар-обгортка, який перетворює StateFlow на сумісний з SwiftUI Combine Publisher.

Модуль shared:domain становить шар чистої бізнес-логіки та містить класи use-case, доменні моделі та інтерфейси репозиторіїв. Модуль не залежить від платформо-залежних інтерфейсів прикладного програмування та є повністю переносним.

Модуль shared:data реалізує інтерфейси репозиторіїв, оголошені у доменному модулі, координує виклики до мережевих та локальних джерел даних і містить мапери для перетворення об’єктів передачі даних (DTO) у доменні моделі.

Інфраструктурні модулі shared:network, shared:database, shared:di та shared:utils забезпечують технічну реалізацію базових механізмів. Бібліотека Ktor у модулі shared:network реалізує крос-платформний HTTP-клієнт з підтримкою серіалізації. Бібліотека SQLDelight у модулі shared:database генерує типобезпечний крос-платформний код для роботи з системою керування реляційними базами даних SQLite на основі звичайних SQL-схем. Бібліотека Koin у модулі shared:di становить фреймворк впровадження залежностей з мінімальною кількістю платформо-залежних особливостей.

Механізм expect/actual є фундаментальним для технології Kotlin Multiplatform. У межах цього механізму у спільному коді декларується expect-сутність (інтерфейс або клас), а кожна платформа надає власну actual-реалізацію. Типовими прикладами застосування механізму є доступ до системних інтерфейсів дати та часу, робота з налаштуваннями користувача (DataStore на платформі Android, NSUserDefaults на платформі iOS), отримання інформації про пристрій, журналювання та інтеграція з нативними засобами шифрування.

Водночас механізм expect/actual не слід розглядати як універсальний інструмент подолання платформо-залежних обмежень. Надмірне його застосування у шарах бізнес-логіки призводить до виникнення неявних залежностей та ускладнює супровід проєкту. У роботі

обґрунтовано доцільність концентрації expect-декларацій у виокремленому модулі shared:platform, до якого інші модулі звертаються через чітко визначені інтерфейси.

Третій етап становить реалізація прототипу застосунку з типовим набором функціональних можливостей: автентифікація користувача, отримання списку записів з віддаленого інтерфейсу прикладного програмування, локальне кешування у базі даних, отримання детальної інформації за записом, базове редагування. Прототип реалізовано у чотирьох конфігураціях: повністю нативна (Kotlin на платформі Android, Swift на платформі iOS), KMM з нативним користувацьким інтерфейсом, Flutter, React Native.

Для кожної конфігурації виконано вимірювання чотирьох метрик: часу холодного запуску (від моменту запуску процесу до появи основного екрана); розміру бінарного файлу (APK для платформи Android, IPA для платформи iOS); споживання оперативної пам’яті у стійкому режимі роботи; частки спільного коду у проєкті. Бенчмарки виконано на мобільному пристрої середнього класу, кожне значення усереднено за результатами 10 запусків.

Четвертий етап становить аналіз отриманих результатів та формулювання рекомендацій. На основі числових показників виконано порівняння підходів, виявлено сильні та слабкі сторони KMM щодо альтернативних рішень та обґрунтовано доцільність використання технології у виробничих проєктах.

6. Результати досліджень

Експериментальне дослідження побудовано на порівнянні чотирьох підходів до крос-платформної мобільної розробки: повністю нативна розробка, Kotlin Multiplatform з нативним користувацьким інтерфейсом, Flutter та React Native. Тестовий застосунок реалізовано однотипно у кожному з варіантів за рекомендованими практиками відповідної екосистеми, що забезпечує коректність порівняння.

Узагальнені характеристики чотирьох підходів наведено у таблиці 1, у якій представлено принцип роботи, очікувані переваги та основні обмеження кожного з підходів.

Таблиця 1. Порівняльні характеристики крос-платформних підходів

Підхід	Принцип роботи	Очікувані переваги	Основні обмеження
Нативна розробка	Окремі кодові бази мовами Kotlin (Android) та Swift (iOS) з прямим доступом до комплектів розробника	Максимальна продуктивність, повний доступ до платформо-залежних інтерфейсів прикладного програмування, природний користувацький досвід	Подвоєння трудовитрат, дублювання помилок, складність синхронізації функціональних можливостей
Kotlin Multiplatform	Спільна бізнес-логіка мовою Kotlin з нативними шарами користувацького інтерфейсу; механізм expect/actual для платформо-залежних інтерфейсів	Близька до нативної продуктивність, нативний користувацький досвід, частка спільного коду 60–80 %	Складність архітектури, обмеження середовища Kotlin/Native, потреба у глибокій експертизі обох платформ

Продовження Таблиці 1

Flutter	Спільний шар користувацького інтерфейсу та логіки мовою Dart; власний рендеринговий двигун Skia	Висока частка спільного коду (понад 95 %), єдина кодова база, стабільна продуктивність	Значний розмір бінарного файлу, неприродний користувацький інтерфейс, окрема мова Dart, складна інтеграція з комплектами розробника
React Native	Спільний шар користувацького інтерфейсу та логіки мовою JavaScript; нативні компоненти через міст-механізм	Велика екосистема JavaScript, висока частка спільного коду, швидка ітерація розробки	Накладні витрати міст-механізму, складність налагодження, нижча продуктивність холодного запуску

Результати кількісних вимірювань наведено на рис. 2. Аналіз отриманих даних дозволяє сформулювати такі висновки.

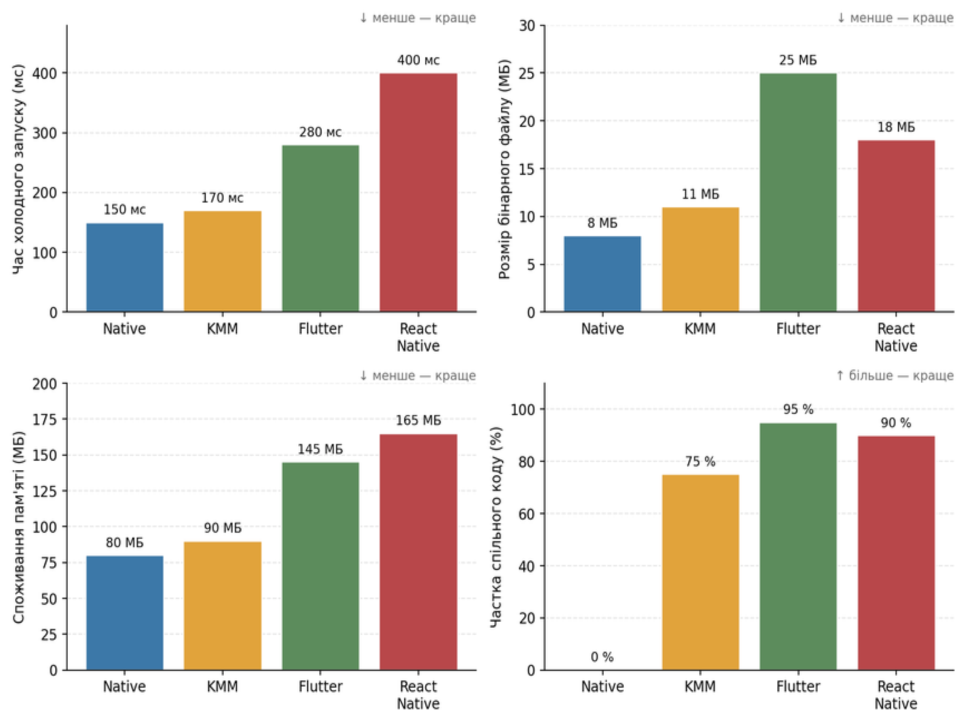

Рис. 2. Порівняльні результати бенчмарків чотирьох підходів за критеріями часу холодного запуску, розміру бінарного файлу, споживання оперативної пам'яті та частки спільного коду

За критерієм часу холодного запуску найкращий результат демонструє нативна реалізація – близько 150 мс на тестовому пристрої. Технологія Kotlin Multiplatform забезпечує приріст часу запуску близько 13 відсотків, що зумовлено ініціалізацією середовища виконання Kotlin/Native на платформі iOS та додатковим завантаженням спільних модулів. Фреймворк Flutter демонструє приріст близько 87 відсотків через необхідність ініціалізації двигуна Skia та віртуальної машини Dart. Найвищий приріст (близько 167 відсотків) спостерігається для React Native, що зумовлено ініціалізацією JavaScript-двигуна та завантаженням JS-бандла.

За критерієм розміру бінарного файлу спостерігається аналогічна тенденція. Нативна реалізація має мінімальний розмір. Kotlin Multiplatform додає близько 3 МБ за рахунок включення середовища виконання Kotlin/Native. Flutter додає близько 17 МБ через включення двигуна Skia та віртуальної машини Dart. React Native додає близько 10 МБ; менший приріст щодо Flutter зумовлений тим, що на платформі iOS JavaScript-двигун уже є частиною операційної системи, тому до бінарного файлу включаються лише JS-бандл та нативні модулі.

За критерієм споживання оперативної пам'яті Kotlin Multiplatform знову демонструє результат, наближений до нативного – приріст близько 12 відсотків. Фреймворки Flutter та React Native характеризуються істотно більшим споживанням пам'яті, що зумовлено наявністю власного рендерингового двигуна та функціонуванням міст-механізму відповідно.

За критерієм частки спільного коду Kotlin Multiplatform забезпечує близько 75 відсотків, тоді як Flutter та React Native – 95 і 90 відсотків відповідно. На перший погляд, KMM поступається конкурентам за цим показником. Однак слід враховувати, що Flutter та React Native реалізують спільний шар користувацького інтерфейсу, тоді як у KMM-застосунку шар користувацького інтерфейсу залишається нативним. Це є усвідомленим архітектурним вибором, який забезпечує природний користувацький досвід та повний доступ до платформо-залежних інтерфейсів прикладного програмування.

Узагальнюючи отримані результати, можна стверджувати, що технологія Kotlin Multiplatform є оптимальним вибором для проєктів, у яких критичними є якість користувацького досвіду та продуктивність застосунку, а істотне скорочення дублювання коду залишається важливою метою.

7. Перспективи подальшого розвитку досліджень

За результатами виконаного аналізу можна виокремити декілька важливих узагальнень. По-перше, Kotlin Multiplatform займає особливу нішу серед крос-платформних технологій: ця технологія не конкурує з нативною розробкою на рівні шару користувацького інтерфейсу, а доповнює її через спільне використання бізнес-логіки. Цей підхід є особливо привабливим для команд, які надають пріоритет якості користувацького досвіду та готові інвестувати у глибоку інтеграцію з платформо-залежними інтерфейсами прикладного програмування.

По-друге, успіх KMM-проєкту критично залежить від обґрунтованості архітектурних рішень. Невдалий вибір модульної структури, неконтрольоване застосування механізму expect/actual, нехтування особливостями середовища Kotlin/Native, відсутність чіткого розподілу між шарами – будь-який з цих факторів може зробити супровід KMM-проєкту складнішим за супровід двох незалежних нативних кодових баз. Отже, обґрунтована модульна структура та чіткі правила взаємодії між шарами є обов'язковою умовою виробничого використання.

По-третє, екосистема технології Kotlin Multiplatform є зрілою для виробничого використання. Бібліотеки Ktor, SQLDelight, kotlinx.coroutines, Koin, kotlinx.serialization покривають більшість типових інженерних потреб, активно розвиваються та мають підтримку як компанії JetBrains, так і широкої спільноти розробників. Водночас залишаються специфічні предметні області, у яких крос-платформні бібліотеки поступаються нативним альтернативам, зокрема біометрія, робота зі специфічними сенсорами, платформо-залежне шифрування.

Перспективи подальшого розвитку напряму можна розглядати у декількох ключових векторах.

Перший вектор – впровадження технології Compose Multiplatform, яка дозволяє використовувати фреймворк Jetpack Compose як єдину модель шару користувацького інтерфейсу на платформах Android, iOS, настільних операційних системах та у вебсередовищі. Стабільний випуск підтримки платформи iOS у 2024 році відкриває можливість спільного використання не лише бізнес-логіки, а й шару користувацького інтерфейсу. Це принципово змінює архітектурні рішення у KMM-проєктах. Водночас Compose Multiplatform не виключає

часткового використання нативного користувацького інтерфейсу для критичних екранів, що дозволяє будувати гібридні архітектури.

Другий вектор – розширення цільових платформ. Технологія Kotlin Multiplatform підтримує не лише Android та iOS, а й JVM, JavaScript та WebAssembly. Спільна бізнес-логіка може використовуватися на серверній стороні, у вебсередовищі та на настільних операційних системах, що є особливо цінним для команд, які прагнуть єдиної доменної моделі у межах усього стека технологій.

Третій вектор – інтеграція зі засобами машинного навчання на пристрої. На платформі Android для цього призначений TensorFlow Lite, на платформі iOS – Core ML. Спільний код може здійснювати координацію викликів до моделей машинного навчання, оброблення результатів та зберігання налаштувань користувача, тоді як платформо-залежні модулі забезпечують безпосереднє виконання обчислень. Такий підхід є особливо цінним для застосунків з функціональністю розпізнавання зображень, оброблення мовлення та формування персоналізованих рекомендацій.

Четвертий вектор – поширення технології Kotlin Multiplatform у корпоративному середовищі, зокрема у банківському, страховому та фінтех-секторах. Високі вимоги до якості користувацького досвіду, потреба швидкого випуску нових функціональних можливостей на обох платформах та складна бізнес-логіка зумовлюють природність вибору KMM у відповідних сценаріях. Кількість великих компаній, що публікують досвід впровадження технології, зростає.

Окремим напрямом подальших досліджень є формалізація стратегій міграції наявних нативних проєктів на технологію Kotlin Multiplatform. Більшість виробничих команд не починають розробку з нуля, тому стратегія поступового перенесення критичних модулів у спільний код за умови збереження стабільності продукту потребує систематизації.

8. Висновки

У роботі виконано системний аналіз технології Kotlin Multiplatform та її позиції серед сучасних крос-платформних рішень, де вона фокусується на спільному використанні бізнес-логіки при збереженні повністю нативних шарів користувацького інтерфейсу. Запропонована багатомодульна архітектура забезпечує чіткий розподіл відповідальностей, спрощує тестування й супровід, а виокремлення expect/actual-декларацій формалізує абстракцію над платформо-залежними інтерфейсами. Інтеграція ключових інфраструктурних бібліотек екосистеми Kotlin ефективно покриває більшість інженерних потреб, підтверджуючи їхню готовність до виробничого використання.

Експериментальні дослідження підтвердили, що Kotlin Multiplatform забезпечує високу частку спільного коду при збереженні загальної продуктивності, наближеної до нативної розробки. Дану технологію доцільно обирати для проєктів, де якість користувацького досвіду є критичною, а команда має досвід нативної розробки та готова до глибокого архітектурного проєктування на початкових етапах. Найперспективнішими напрямами подальших досліджень залишаються впровадження Compose Multiplatform, розширення цільових платформ та інтеграція зі засобами машинного навчання на пристрої. У підсумку, розроблений підхід доводить зрілість технології для підвищення ефективності команд мобільної розробки без зниження якості кінцевого продукту.

Список літератури:

1) Joorabchi M. E., Mesbah A., Kruchten P. (2013). Real challenges in mobile app development. ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 15–24. https://doi.org/10.1109/ESEM.2013.9

2) Heitkötter H., Hanschke S., Majchrzak T. A. (2013). Evaluating cross-platform development approaches for mobile applications. Lecture Notes in Business Information Processing, 140, 120–138. https://doi.org/10.1007/978-3-642-36608-6_8

3) Xanthopoulos S., Xinogalos S. (2013). A comparative analysis of cross-platform development approaches for mobile applications. Proceedings of the 6th Balkan Conference in Informatics, 213–220. https://doi.org/10.1145/2490257.2490292

4) Biørn-Hansen A., Majchrzak T. A., Grønli T.-M. (2017). Progressive web apps: The possible web-native unifier for mobile development. Proceedings of the 13th International Conference on Web Information Systems and Technologies, 344–351. https://doi.org/10.5220/0006353703440351

5) JetBrains. (2024). Kotlin Multiplatform: Share code on iOS, Android and other platforms. https://kotlinlang.org/docs/multiplatform.html

6) Que P., Guo X., Zhu M. (2016). A comprehensive comparison between hybrid and native app paradigms. 8th International Conference on Computational Intelligence and Communication Networks, 611–614. https://doi.org/10.1109/CICN.2016.125

7) Versluis R., Adams M. P. (2018). Xamarin.Forms projects: Build seven real-world cross-platform mobile apps with C# and Xamarin.Forms. Packt Publishing.

8) Hansson N., Vidhall T. (2016). Effects on performance and usability for cross-platform application development using React Native. Linköping University Electronic Press.

9) Wu W. (2018). React Native vs Flutter, cross-platform mobile application frameworks. Metropolia University of Applied Sciences.

10) Salin H. (2022). Mobile app architecture and security with Kotlin Multiplatform Mobile. Master’s thesis, KTH Royal Institute of Technology.

11) Ganatra N., Patel A. (2021). Comprehensive analysis of mobile architectural patterns for Kotlin Multiplatform. International Journal of Computer Applications, 183(40), 18–25. https://doi.org/10.5120/ijca2021921778

12) Stojkov M., Dragan D., Mirković M. (2023). Code sharing techniques and architectural patterns in Kotlin Multiplatform Mobile applications. Proceedings of the IEEE Zooming Innovation in Consumer Technologies Conference, 132–137. https://doi.org/10.1109/ZINC58345.2023.10173837

13) JetBrains. (2024). Ktor: A framework for building asynchronous servers and clients in connected systems using the powerful Kotlin programming language. https://ktor.io/docs/welcome.html

14) Cash App. (2024). SQLDelight: Generates typesafe Kotlin APIs from SQL. https://cashapp.github.io/sqldelight/

15) JetBrains. (2024). Compose Multiplatform: A declarative framework for sharing UIs across multiple platforms with Kotlin. https://www.jetbrains.com/lp/compose-multiplatform/

Research of architectural approaches to cross-platform mobile application development using Kotlin Multiplatform

Ivan Moroz
Lutsk National Technical University, Lutsk, Ukraine
ORCID: 0009-0009-5552-071X

Igor Andrushchak
Lutsk National Technical University, Lutsk, Ukraine
ORCID: 0000-0002-8751-4420

Abstract: The paper investigates architectural approaches to the development of cross-platform mobile applications based on the Kotlin Multiplatform (KMM) technology. The current state of

mobile development is characterised by a technological duality of two dominant platforms, Android and iOS, which rely on fundamentally different programming languages, frameworks, and platform-specific application programming interfaces. This situation leads to doubled development effort, duplicated defects, and increased product maintenance costs. The evolution of cross-platform solutions is reviewed, including hybrid WebView-based approaches (Apache Cordova, PhoneGap, Ionic), the Xamarin technology with C# and the Mono runtime, and the modern frameworks React Native and Flutter. It is established that each of these approaches entails significant trade-offs in terms of performance, naturalness of user experience, or integration with platform-specific application programming interfaces. The Kotlin Multiplatform technology developed by JetBrains implements a fundamentally different approach: sharing business logic written in Kotlin while preserving fully native user interface layers. The paper proposes a multi-module architecture for KMM applications with a clear separation into presentation, domain, data, and infrastructure layers. The expect/actual mechanism is analysed as the principal instrument for handling platform-specific code. The role of key infrastructure libraries of the Kotlin ecosystem is examined: Ktor for network operations, SQLDelight for database operations, kotlinx.coroutines for asynchronous execution, Koin for dependency injection, and kotlinx.serialization for data serialisation. A comparative evaluation of four approaches (native development, KMM, Flutter, and React Native) is performed using the criteria of cold start time, binary file size, runtime memory consumption, and shared code ratio. The obtained results demonstrate that Kotlin Multiplatform delivers performance close to native, with a cold start time overhead of approximately 13 % and a share of common code reaching 75 % in a typical application. The results substantiate the use of KMM for projects in which user experience quality and performance are critical and the maintenance of two independent codebases is no longer economically justified. The prospects for further research are also analysed, including the adoption of Compose Multiplatform as a unified user interface framework, integration with on-device machine learning, and the broader use of KMM in enterprise environments.

Keywords: Kotlin Multiplatform; cross-platform development; mobile applications; software architecture; expect/actual mechanism; shared code; multi-module architecture; mobile application performance.