Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Уніфікована модель мобільних веб-застосунків на платформі Swift
(тема)

Виконав:
здобувач 2 року навчання,
групи СПм-23-4
Владислав ОЦЕВИК
(власне ім’я, прізвище)

Спеціальність 123 «Комп’ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва освітньої програми)

Керівник: доц. Тетяна ФІЛІМОНЧУК
(посада, власне ім’я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ ____________ Андрій КОВАЛЕНКО
(підпис) (власне ім’я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп’ютерна інженерія»
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ________________
(підпис)
“____”________________ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Оцевику Владиславу Андрійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Уніфікована модель мобільних веб-застосунків на платформі Swift

затверджена наказом по університету від “ 21 ” квітня 2025 р. № 296 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 16 червня 2025 р.

3. Вхідні дані до роботи 1) мова програмування Swift; 2) фреймворк SwiftUI; 3) фреймворк Skip; 4) фреймворк Vapor; 5) середовище розробки Xcode.

4. Перелік питань, що потрібно опрацювати у роботі

1) аналіз сучасних підходів до кросплатформної розробки;
2) огляд можливостей мови Swift та пов’заних технологій (Skip, Wapor, Tokmak);
3) побудова уніфікованої моделі для мобільної та веброзробки;
4) реалізація прототипу застосунку для обліку фінансів
5) висновки.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій

Слайд-презентація – 12 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Формулювання проблематики та цілей	22.04.25-25.04.25	
2	Аналіз сучасних підходів до кросплатформної	26.04.25-30.04.25	
3	Вибір інструментів та методології реалізації	01.05.25-05.05.25	
4	Розробка уніфікованої моделі застосунку	06.05.25-11.05.25	
5	Реалізація мобільного та вебінтерфейсу	12.05.25-18.05.25	
6	Інтеграція з серверною частиною	19.05.25-24.05.25	
	та налаштування CI/CD		
7	Тестування, аналіз результатів та оптимізація	25.05.25-31.05.25	
8	Подання кваліфікаційної роботи на рецензу-	01.06.25-12.06.25	
	вання		

Дата видачі завдання “ 21 ” квітня 20 25 р.

Здобувач ____________________
(підпис)

Керівник роботи ____________________ доц. Тетяна ФІЛІМОНЧУК
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 88 с., 13 рис., 1 дод., 23 джерела.

КРОСПЛАТФОРМЕННА РОЗРОБКА, SWIFT, FULL SWIFT STACK, МОБІЛЬНІ ЗАСТОСУНКИ, IOS, SWIFTUI, SKIP, VAPOR, SWIFTWASM, TOKAMAK, УНІФІКАЦІЯ КОДОВОЇ БАЗИ, ОПТИМІЗАЦІЯ РОЗРОБКИ.

Метою кваліфікаційної роботи є розробка та аналіз ефективності кросплатформенного застосунку, побудованого на основі Full Swift Stack, що охоплює мобільні платформи, зокрема iOS.

Об'єктом дослідження є процес розробки кросплатформного застосунку з використанням мови програмування Swift. Предметом дослідження є архітектура, технологічний стек та інструменти, що використовуються для розробки таких застосунків.

Для досягнення поставленої мети застосовуються методи аналізу літератури та документації, порівняльний аналіз існуючих рішень, розробка та тестування прототипу, а також статистичний аналіз отриманих результатів.

Дане дослідження має значний практичний потенціал, оскільки його результати можуть бути використані для створення ефективних та якісних кросплатформних застосунків. Використання єдиного технологічного стеку дозволить зменшити витрати на розробку, спростити підтримку застосунків та забезпечити їх доступність на різних платформах.

ABSTRACT

Master's thesis: 88 pages, 13 figures, 1 appendice, 23 sources.

CROSS-PLATFORM DEVELOPMENT, SWIFT, FULL SWIFT STACK, MOBILE APPLICATIONS, IOS, SWIFTUI, SKIP, VAPOR, SWIFTWASM, TOKAMAK, CODEBASE UNIFICATION, DEVELOPMENT OPTIMIZATION.

The aim of the qualification thesis is the development and analysis of the effectiveness of a cross-platform application built on the basis of the Full Swift Stack, covering mobile platforms, particularly iOS.

The object of the research is the process of developing a cross-platform application using the Swift programming language. The subject of the research is the architecture, technology stack, and tools used for the development of such applications.

To achieve the stated goal, methods such as literature and documentation analysis, comparative analysis of existing solutions, prototype development and testing, as well as statistical analysis of the obtained results are used.

This research has significant practical potential, as its results can be used to create efficient and high-quality cross-platform applications. The use of a unified technology stack will reduce development costs, simplify application maintenance, and ensure their availability across different platforms.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 8
ВСТУП 9
1 АНАЛІЗ СУЧАСНИХ ПІДХОДІВ ДО КРОСПЛАТФОРМНОЇ РОЗРОБКИ 10
1.1 Фреймворк Flutter 11
1.3 Фреймворк Kotlin Multiplatform 15
1.4 Застосування мови програмування Swift 16
1.4.1 Swift у кросплатформній розробці 16
1.4.2 Swift у веброзробці 18
1.4.3 Swift у серверній розробці 21
1.4.4 Використання Swift у розробці Android-застосунків 23
1.4.5 Переваги Swift у кросплатформній розробці 24
1.5 Інструменти для Swift 26
2 АНАЛІЗ СКЛАДОВИХ МОДЕЛІ, ЯКА ОРІЄНТОВАНА НА РОЗРОБКУ ВЕБ ТА МОБІЛЬНИХ ЗАСТОСУНКІВ 30
2.1 Аналіз останніх досліджень та публікацій 30
2.2 Модель уніфікованого підходу для розробки мобільних та вебзастосунків з використанням мови Swift 33
2.3 Обґрунтування складових уніфікованої моделі для розробки мобільних та вебзастосунків 34
3 РОЗРОБКА УНІФІКОВАНОГО ПРОТОТИПУ НА ПРИКЛАДІ ЗАСТОСУНКУ ДЛЯ ОБЛІКУ ФІНАНСІВ 42
3.1 Опис уніфікованого прототипу 43
3.2 Розгортання серверної частини (Backend) 45
3.3 Мобільний застосунок 49
3.4 Вебверсія застосунку 51
3.5 Розгортання та безперервна інтеграція (CI/CD) 53

4 ПОРІВНЯННЯ ІСНУЮЧОЇ ТА РОЗРОБЛЕНОЇ МОДЕЛЕЙ ДО РОЗРОБКИ МОБІЛЬНИХ ТА ВЕБЗАСТОСУНКІВ 56
4.1 Порівняння моделей... 56
4.2 Порівняння за швидкістю розробки 59
4.3 Порівняння за кількістю розробників 61
4.4 Порівняння за вартістю розробки.. 64
4.5 Порівняння за швидкістю заміни та масштабування 66
4.6 Порівняння за безпекою та якістю 69
4.7 Висновки порівняння 71
ВИСНОВКИ... 76
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 79
ДОДАТОК А Графічний матеріал кваліфікаційної роботи 82

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

A – архітектура розробки (англ., Architecture)

AM – аналітика та моніторинг (англ., Analytics and Monitoring)

API – інтерфейс програмування додатків (англ., Application Programming Interface)

DB – база даних (англ., Database)

IDE – інтегроване середовище розробки (англ., Integrated Development Environment)

iOS – мобільна операційна система (англ., Apple's mobile operating system)

JSON – формат обміну даними (англ., JavaScript Object Notation)

MVC – модель-вид-контролер (англ., Model-View-Controller)

MVVM – модель-представлення-модель представлення (англ., Model-View-ViewModel)

NI – мережний інтерфейс (англ., Network Interface)

P – платформа розробки (англ., Platform)

PL – мова програмування (англ., Programming Language)

SC – серверна складова застосунку (англ., Server Component)

S – безпека застосунку (англ., Security)

SDK – комплект для розробки програмного забезпечення (англ., Software Development Kit)

TS – набір тестів (англ., Test Suite)

UI – інтерфейс користувача (англ., User Interface)

UX – досвід користувача (англ., User Experience)

WWW – Всесвітня павутина (англ., World Wide Web)

ВСТУП

Сучасний світ характеризується стрімким розвитком інформаційних технологій, і мобільні застосунки стали невід'ємною частиною повсякденного життя. Вони використовуються для спілкування, навчання, роботи, розваг та інших сфер. У зв'язку з цим, розробка якісних та функціональних мобільних застосунків є надзвичайно актуальною задачею.

Проте розробка під кожну платформу окремо, зокрема iOS, Android та Web, є складним та ресурсозатратним процесом, що вимагає значних витрат часу, зусиль та фінансових ресурсів. Тому кросплатформні рішення, які дозволяють використовувати один код для різних платформ, стають все більш популярними та затребуваними [1].

У цьому контексті використання мови Swift для кросплатформної розробки є перспективним напрямком. Swift – це сучасна, потужна та безпечна мова програмування, яка була розроблена компанією Apple, що забезпечує високу продуктивність, має вбудовані механізми захисту від помилок та постійно вдосконалюється відповідно до сучасних вимог розробки. Завдяки розвитку таких інструментів, як SwiftUI, Skip, Vapor, SwiftWasm та Tokamak, Swift стає привабливим варіантом для створення універсальних застосунків, які можуть працювати на різних платформах.

Метою дослідження є розробка та аналіз ефективності кросплатформного застосунку, побудованого на основі Full Swift Stack, що охоплює мобільні платформи, зокрема iOS та Android, а також Web. У межах дослідження буде проведено аналіз сучасних підходів до кросплатформної розробки, обґрунтовано вибір Full Swift Stack, створено прототип застосунку, проведено тестування та оцінено продуктивність рішення.

1 АНАЛІЗ СУЧАСНИХ ПІДХОДІВ ДО КРОСПЛАТФОРМНОЇ РОЗРОБКИ

Сучасна розробка мобільних застосунків стоїть перед викликом забезпечення їхньої доступності на різних платформах, таких як iOS, Android та Web. Традиційний підхід передбачає створення окремих нативних застосунків для кожної платформи, що дозволяє досягти найкращої продуктивності, оптимізації під конкретне середовище та доступу до всіх можливостей операційної системи. Однак такий підхід є ресурсозатратним та потребує значних зусиль, оскільки розробка, тестування та підтримка окремих версій застосунку для кожної платформи вимагають більше часу та фінансових ресурсів.

Альтернативою цьому підходу є використання кросплатформних рішень, які дозволяють створювати один код для різних платформ, що значно зменшує витрати на розробку, спрощує підтримку та оновлення застосунків, а також забезпечує швидший вихід продукту на ринок. Завдяки розвитку кросплатформних технологій розробники отримали можливість створювати застосунки, які мають нативний вигляд та більшу продуктивність, зберігаючи при цьому єдиний код для всіх платформ.

На сьогодні існує декілька основних підходів до кросплатформної розробки, серед яких найбільш поширеними є використання фреймворків Flutter, React Native та Kotlin Multiplatform, які дозволяють працювати з єдиним кодом для мобільних платформ. Кожен із цих підходів має свої переваги та недоліки, які впливають на вибір технології залежно від вимог проєкту.

Окремим напрямком є використання Swift як універсальної мови програмування не лише для ОС iOS, але й для інших платформ, включаючи Android, Web та серверну частину. Swift традиційно асоціюється з екосистемою Apple, однак завдяки розвитку інструментів, таких як SwiftUI, Skip, Vapor, SwiftWasm та Tokamak, вона отримує можливості для створення універсальних застосунків, що можуть працювати на різних платформах.

Розглянемо сучасні підходи до кросплатформної розробки, їхні переваги та обмеження, а також перспективи використання мови програмування Swift як єдиного технологічного стеку для створення застосунків, що функціонують на мобільних та вебплатформах.

1.1 Фреймворк Flutter

Flutter [2] – це фреймворк для кросплатформної розробки, створений компанією Google, який дозволяє створювати застосунки для iOS, Android, Web, macOS, Windows та Linux на основі єдиного коду. Його основною мовою програмування є Dart, а ключовою особливістю – використання власного рендерингового рушія, що забезпечує високу продуктивність та однаковий вигляд застосунку на всіх платформах.

На архітектурному рівні Flutter складається з кількох основних компонентів, які взаємодіють між собою для забезпечення швидкої роботи та зручного написання коду. Основу архітектури можна побачити на рисунку 1.1.

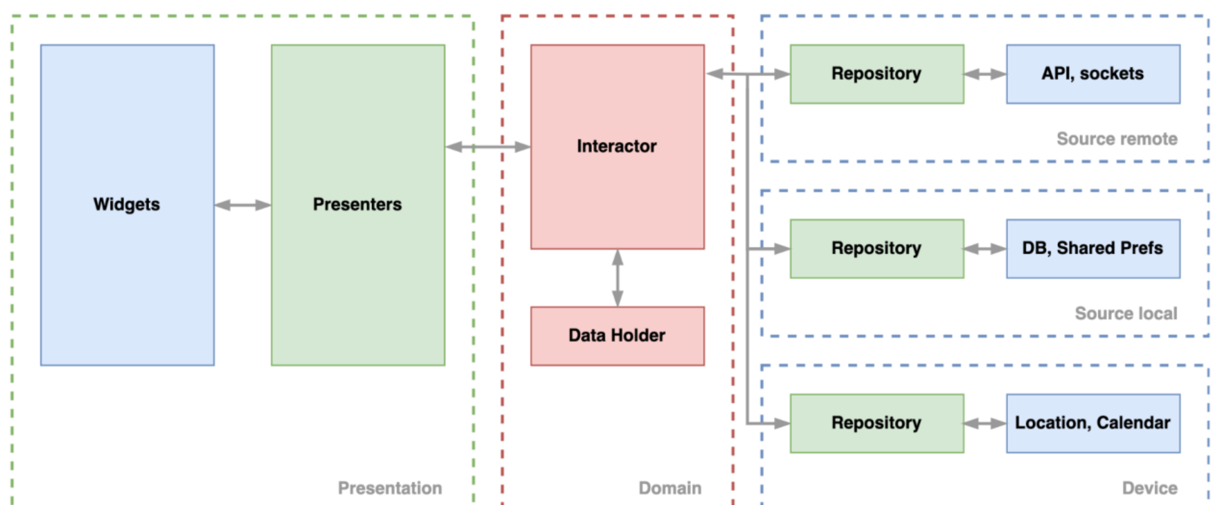

Рисунок 1.1 – Загальна архітектура Flutter-застосунку

Flutter працює на основі архітектури, що використовує віртуальну машину Dart VM для швидкої розробки та нативну компіляцію для кінцевого

виконання. Основним компонентом є рушій Skia [3], який самостійно рендерить усі елементи інтерфейсу, оминаючи нативні компоненти операційної системи, що дозволяє забезпечити однаковий вигляд застосунку незалежно від платформи, але може викликати певні труднощі у відповідності до гайдлайнів UI кожної операційної системи.

Однією з найбільших переваг фреймворку Flutter є можливість писати єдиний код для всіх платформ, що значно скорочує час та витрати на розробку. Висока продуктивність досягається завдяки компіляції Ahead-of-Time (AOT), що дозволяє виконувати код без проміжних інтерпретацій. Гнучка система віджетів дозволяє створювати сучасний адаптивний інтерфейс, який легко кастомізується. Функція Hot Reload забезпечує миттєве оновлення інтерфейсу під час розробки, що значно пришвидшує процес внесення змін.

Проте Flutter має й недоліки. Розмір застосунків, створених за допомогою цього фреймворку, зазвичай більший порівняно з нативними аналогами, оскільки до них додаються всі необхідні бібліотеки та рушій. Хоча Flutter підтримує взаємодію з нативними API через Platform Channels, деякі складні інтеграції можуть вимагати написання додаткового коду на Swift або Kotlin. Окрім цього, хоча фреймворк намагається дотримуватися стилю нативних застосунків, у деяких випадках вигляд елементів інтерфейсу може трохи відрізнятися.

Популярність Flutter зростає, і він вже використовується такими компаніями, як Google, Alibaba, BMW, а також у багатьох стартапах. Його особливо цінують за швидкість розробки, універсальність та гнучкість. Flutter є хорошим вибором для проєктів, де важлива одночасна підтримка кількох платформ, однак у випадках, коли необхідна максимальна відповідність нативним стандартам, може знадобитися глибша інтеграція з платформами або використання нативних підходів. Flutter активно розвивається, отримує нові інструменти та має велику спільноту, що робить його привабливим для сучасної кросплатформної розробки.

1.2 Фреймворк React Native

React Native [4] – це кросплатформний фреймворк, який був створений компанією Meta (Facebook). Він дозволяє розробляти мобільні застосунки для iOS та Android з використанням JavaScript та React. Основною концепцією React Native є можливість повторного використання вебтехнологій для створення нативних мобільних застосунків, що дозволяє значно скоротити витрати часу та ресурсів на розробку.

На архітектурному рівні React Native складається з 3 основних частин:

- JavaScript-код: логіка застосунку та UI, що працює у середовищі V8 або Hermes;

- Native Modules: нативні модулі iOS та Android для роботи з системними API;

- Bridge: механізм зв’язку між JavaScript-кодом та нативними API.

Ця структура добре ілюструється на рисунку 1.2.

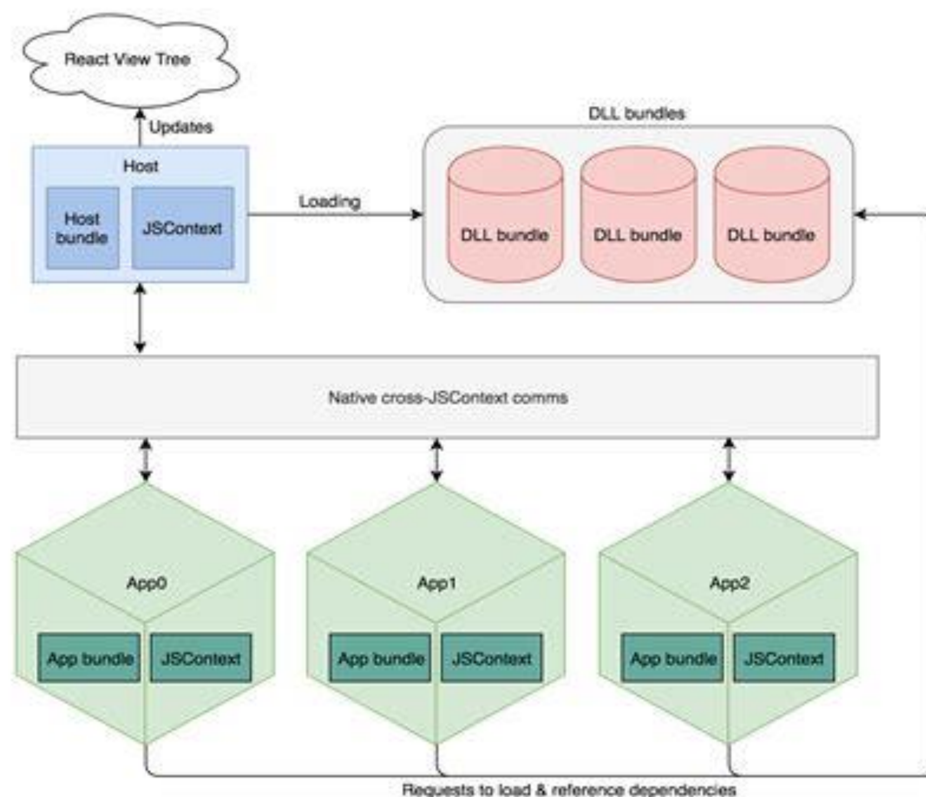

Рисунок 1.2 – Архітектура React Native

Фреймворк використовує компонентний підхід, характерний для React, де UI представлений у вигляді окремих компонентів, що управляються станом. Головною особливістю React Native є можливість взаємодії з нативними компонентами через міст (Bridge), який дозволяє JavaScript-коду викликати функції нативних API iOS та Android. Це забезпечує можливість використання нативних можливостей платформи, таких як доступ до камери, Bluetooth, геолокації та інших системних функцій.

Однією з ключових переваг React Native є можливість спільного використання значної частини коду між платформами, що дозволяє швидко адаптувати застосунок для iOS та Android. Використання React дозволяє розробникам застосовувати вже знайомі концепції, такі як компонентний підхід та управління станом за допомогою Redux або Context API. Крім того, фреймворк підтримує гаряче оновлення коду (Hot Reload), що дозволяє миттєво бачити зміни без необхідності перезавантаження застосунку.

Продуктивність React Native зазвичай вища, ніж у вебзастосунків, але нижча, ніж у нативних застосунків, оскільки фреймворк працює через міст між JavaScript та нативним кодом. Це може спричиняти затримки в обробці анімацій або виконанні складних обчислень, що є однією з основних проблем при розробці продуктивних мобільних застосунків. Для покращення продуктивності можна використовувати такі підходи, як Hermes [5] – спеціальний JavaScript-рушій, який оптимізований для мобільних пристроїв.

Попри свою гнучкість, React Native має певні обмеження. Хоча бібліотека має багато готових компонентів, деякі складні функції можуть вимагати написання нативного коду на Swift або Kotlin. Також існує ризик того, що оновлення фреймворку можуть спричиняти проблеми сумісності, особливо якщо застосунок використовує багато сторонніх бібліотек. Слід зазначити, що на даний час React Native активно використовується великими компаніями, такими як Instagram, Airbnb, Walmart, Tesla та Uber Eats. Завдяки своїй простоті, швидкому розгортанню та широкій підтримці спільноти, цей фреймворк залишається одним із найпопулярніших інструментів для

кросплатформної мобільної розробки. Він є хорошим вибором для створення застосунків, які мають простий та адаптивний інтерфейс, але вимагають мінімального використання складних нативних API.

1.3 Фреймворк Kotlin Multiplatform

Kotlin Multiplatform (KMP) – це технологія, яка була розроблена компанією JetBrains, що дозволяє створювати кросплатформні застосунки, використовуючи єдину кодову базу для бізнес-логіки, при цьому залишаючи можливість використання нативного коду для інтерфейсу та платформоспецифічних функцій. Основною перевагою Kotlin Multiplatform є можливість повторного використання коду для iOS, Android, Web та навіть серверних застосунків, що дозволяє значно зменшити час розробки та спростити підтримку проєкту.

На відміну від Flutter чи React Native, які намагаються створювати однаковий UI на різних платформах, Kotlin Multiplatform [6] фокусується на спільному використанні бізнес-логіки, залишаючи розробку інтерфейсу нативною. Це дозволяє уникнути проблем із продуктивністю та відповідністю платформним гайдлайнам. Архітектура KMP (рисунок 1.3) передбачає поділ застосунку на дві основні частини:

- Common module: містить всю бізнес-логіку (робота з мережею, обробка даних, управління станом);
- Platform modules: містять специфічний код для iOS та Android, включаючи UI та доступ до системних API.

Однією з найбільших переваг Kotlin Multiplatform є гнучкість у виборі інструментів. Оскільки застосунок використовує нативні UI-фреймворки, такі як Jetpack Compose для Android або SwiftUI для iOS, це дозволяє досягти найкращої продуктивності та відповідності дизайну. Також розробники можуть вибірково спільно використовувати код, наприклад, лише для мережної взаємодії, обробки даних чи управління станом.

Проте технологія має і свої недоліки. На момент написання цього дослідження Kotlin Multiplatform ще перебуває в активній розробці, тому може виникати нестача документації або підтримки для деяких платформ. Також процес інтеграції може бути складнішим, ніж у більш зрілих фреймворках, оскільки вимагає налаштування взаємодії між різними модулями та платформами.

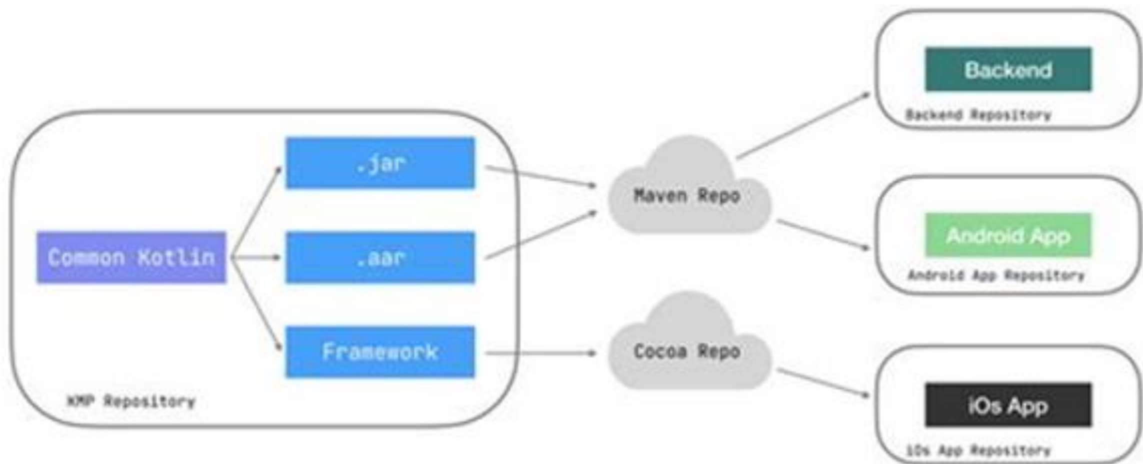

Рисунок 1.3 – Архітектура KMP

Попри ці обмеження, Kotlin Multiplatform активно використовується такими компаніями, як Netflix, Philips, Cash App та IceRock, що свідчить про його великий потенціал. Він є чудовим вибором для команд, які хочуть зменшити дублювання коду між iOS та Android, зберігаючи при цьому продуктивність та відповідність кожної платформи її нативним стандартам.

Особливо ця технологія підходить для розробки корпоративних застосунків, де важливо зберегти єдиний підхід до бізнес-логіки, але при цьому не обмежувати можливості нативного інтерфейсу.

1.4 Застосування мови програмування Swift

1.4.1 Swift у кросплатформній розробці

Мова програмування Swift [7] була представлена Apple у 2014 році як сучасна мова, що мала замінити Objective-C для розробки застосунків під

iOS, macOS, watchOS та tvOS. Основною метою її створення було підвищення продуктивності, безпеки та спрощення розробки мобільних застосунків. На відміну від Objective-C, Swift отримав сучасний синтаксис, який зробив його простішим у використанні, особливо для початківців. Важливою особливістю мови стала підтримка безпечної роботи з пам’яттю, що зменшило кількість помилок, пов’язаних із нульовими покажчиками та керуванням пам’яттю.

Починаючи з другої версії, Apple відкрила вихідний код Swift, що значно розширило його можливості. Це рішення дозволило не лише залучити нових розробників, але й створити додаткові інструменти та фреймворки, які допомогли поширити Swift за межі екосистеми Apple. Підтримка Linux стала першим важливим кроком у цьому напрямку, оскільки відкрила шлях до використання Swift у серверній розробці. У відповідь на це з’явилися фреймворки на кшталт Vapor та Kitura, які дали можливість розробляти вебзастосунки та серверні API за допомогою Swift.

Відкритий код сприяв швидшому розвитку мови та збільшенню її популярності серед розробників у всьому світі. З кожним новим релізом Swift отримував покращення продуктивності та нові можливості. Починаючи з п’ятої версії, мова стала більш стабільною, оскільки було зафіксовано ABI (Application Binary Interface), що забезпечило сумісність між версіями та зробило Swift більш привабливою для довготривалих проєктів.

Відкриття коду дало змогу використовувати Swift не лише у мобільній розробці, а й на сервері та у вебсередовищі. Проєкт SwiftWasm дозволив компілювати Swift у WebAssembly, відкриваючи можливість використання цієї мови у браузерах. Фреймворк Tokamak зробив можливим використання SwiftUI-подібного синтаксису для веброзробки, що спрощує створення UI для вебзастосунків.

Попри всі ці досягнення, найбільшим обмеженням залишається відсутність офіційної підтримки Android. Хоча існують сторонні ініціативи, такі як Skip, вони ще не отримали широкого поширення через обмежену

інтеграцію з екосистемою Google. Це значно стримує використання Swift як повноцінного кросплатформного інструменту, оскільки розробники, які створюють мобільні застосунки, змушені використовувати окремий стек для Android.

Swift має низку переваг, які роблять її привабливою для кросплатформної розробки. Завдяки строгій типізації, автоматичному керуванню пам'яттю та мінімізації нульових покажчиків мова забезпечує високу безпеку та стабільність коду. Висока продуктивність досягається завдяки компіляції у машинний код, що забезпечує швидку роботу застосунків на рівні C++. Зрозумілий синтаксис робить Swift доступною навіть для початківців, що прискорює навчання та розробку.

Можливість використання Swift не лише для мобільних застосунків, а й для серверної та веброзробки робить її універсальним інструментом. Завдяки фреймворку Vapor розробники можуть писати back-end тією ж мовою, що й клієнтську частину застосунку, що зменшує кількість технологій у проєкті, полегшує підтримку та покращує взаємодію між командами.

Swift має значний потенціал для кросплатформної розробки, і з розвитком екосистеми її можливості лише зростатимуть. Незважаючи на певні обмеження, вона залишається перспективним вибором для розробників, які хочуть працювати в єдиному технологічному стеку, створюючи ефективні та стабільні застосунки для різних платформ.

1.4.2 Swift у веброзробці

Мова Swift поступово знаходить своє застосування у веброзробці. Завдяки появі SwiftWasm, який дозволяє компілювати Swift у WebAssembly, та фреймворку Tokamak, що надає декларативний підхід до створення UI, Swift може використовуватися для розробки front-end вебзастосунків. Це відкриває можливість створення вебінтерфейсів з використанням мови, яка раніше була орієнтована виключно на мобільну та серверну розробку.

WebAssembly (Wasm) [8] – це бінарний формат, який дозволяє виконувати код високої продуктивності безпосередньо в браузері. Його основною метою є розширення можливостей вебплатформи шляхом запуску мов, які зазвичай не використовуються у браузерах, таких як C, C++ та Rust. Swift також отримала можливість компіляції в WebAssembly завдяки проєкту SwiftWasm, який дозволяє запускати Swift-код у браузері без необхідності використання JavaScript.

SwiftWasm базується на офіційній версії Swift та адаптує її для генерації коду у WebAssembly. Це дає змогу створювати продуктивні вебзастосунки без необхідності використовувати традиційні інструменти, такі як JavaScript, TypeScript або React. SwiftWasm підтримує взаємодію з JavaScript через WebAssembly System Interface (WASI), що дозволяє викликати браузерні API та взаємодіяти з DOM.

Оскільки WebAssembly забезпечує продуктивність, наближену до нативного виконання, SwiftWasm може бути ефективним для створення складних вебзастосунків, які потребують швидкого виконання. Це відкриває нові можливості для використання Swift у таких сферах, як розробка ігор, аналітичних інструментів та високонавантажених вебсервісів.

Tokamak [9] – це декларативний UI-фреймворк, який був розроблений для використання Swift у веброзробці. Його основна ідея полягає в тому, щоб зробити SwiftUI доступним у браузерах, використовуючи WebAssembly. Tokamak дозволяє розробникам створювати вебінтерфейси, використовуючи знайомий синтаксис SwiftUI, що значно спрощує розробку для тих, хто вже працює з екосистемою Swift.

Архітектурно Tokamak працює подібно до React: він використовує віртуальний DOM для оптимізації оновлення UI та декларативний підхід до створення компонентів, що дозволяє будувати складні інтерфейси, використовуючи Swift, без необхідності звертатися до JavaScript або інших традиційних вебтехнологій.

Головною перевагою Tokamak є можливість повторного використання Swift-коду між мобільними та вебзастосунками. Це особливо корисно для команд, які вже використовують Swift для iOS-розробки та хочуть розширити підтримку своїх застосунків у вебсередовищі без необхідності переходу на JavaScript.

Проте, на відміну від React, який має багату екосистему та велику кількість бібліотек, Tokamak ще знаходиться на етапі активного розвитку. Наразі підтримка браузерних API та сторонніх бібліотек у ньому обмежена, що може створювати труднощі для розробників, які хочуть інтегрувати складні функції, такі як керування станом або робота з API.

Swift у веброзробці має кілька суттєвих переваг. Головною є продуктивність WebAssembly, яка дозволяє виконувати Swift-код у браузері без значних накладних витрат. Використання Swift для веброзробки дозволяє розробникам, які вже працюють у цій екосистемі, створювати вебзастосунки, не переходячи на інші мови, такі як JavaScript або TypeScript. Повторне використання коду між iOS, macOS та вебплатформою допомагає зменшити час розробки та витрати на підтримку застосунків.

Однак використання Swift у веброзробці також має певні обмеження. Основним викликом є відносно слабка екосистема та нестача бібліотек, що значно поступається популярним вебфреймворкам, таким як React або Vue.js. Хоча WebAssembly забезпечує високу продуктивність, взаємодія з DOM через SwiftWasm поки що не така ефективна, як у JavaScript, що може призводити до додаткових затримок при оновленні інтерфейсу.

Попри ці обмеження, Swift для веброзробки є перспективним напрямком, особливо для компаній, які вже використовують Swift у своїх мобільних застосунках. З подальшим розвитком SwiftWasm та Tokamak ця технологія може стати конкурентоспроможною альтернативою традиційним вебфреймворкам, надаючи Swift-розробникам можливість працювати в єдиному технологічному стеку без необхідності залучення інших мов програмування.

1.4.3 Swift у серверній розробці

Swift відкриває нові можливості у серверній розробці, дозволяючи розробникам використовувати єдину мову як для клієнтської, так і для back-end частини застосунків. Завдяки своїй продуктивності, безпеці та сучасному синтаксису Swift стає потужним інструментом для створення серверних API та масштабованих вебсервісів. Головним рушієм популярності Swift у серверній розробці є фреймворк Vapor [10], який забезпечує високу продуктивність, простоту у використанні та підтримку сучасних технологій.

Фреймворк Vapor пропонує повноцінне рішення для створення серверних застосунків, надаючи вбудовану маршрутизацію, обробку HTTP-запитів, інтеграцію з базами даних, підтримку асинхронного програмування та розширювану систему Middleware. Завдяки використанню SwiftNIO, Vapor працює на неблокуючій асинхронній архітектурі, що забезпечує високу продуктивність та масштабованість навіть під значним навантаженням. Його структура побудована таким чином, щоб максимально використовувати ресурси сервера, зменшуючи затримки та споживання пам’яті.

Vapor підтримує розширюваність через модульну архітектуру, що дозволяє легко додавати нові функціональні можливості без зайвого ускладнення коду. Це робить його ідеальним рішенням для створення сучасних API, мікросервісів та високонавантажених вебзастосунків. Завдяки сучасному синтаксису Swift, код на Vapor є лаконічним, читабельним і легко підтримуваним, що значно покращує продуктивність розробників.

Однією з ключових особливостей Vapor є глибока інтеграція з ORM Fluent, який надає зручний та гнучкий підхід до роботи з базами даних. Fluent дозволяє визначати моделі у вигляді Swift-структур, що спрощує взаємодію з базами даних та забезпечує безпечне використання SQL-запитів. Завдяки Fluent можна легко працювати з PostgreSQL, MySQL, SQLite та іншими базами, що забезпечує широкі можливості для розробників.

Fluent підтримує автоматичну міграцію схем, що значно полегшує

процес управління базою даних, що дозволяє швидко вносити зміни у структуру даних без необхідності написання SQL-запитів вручну. ORM також використовує типізовані запити, що усуває ризики SQL-ін’єкцій та підвищує безпеку роботи з даними.

Використання Swift для серверної розробки має значні переваги у порівнянні з іншими популярними технологіями, такими як Node.js, Django та Spring. Завдяки компіляції у машинний код, Swift працює значно швидше, ніж інтерпретовані мови, такі як JavaScript або Python, що робить її ідеальним вибором для застосунків, які потребують високої продуктивності та низької затримки. Порівняно з Node.js, який базується на JavaScript, Vapor пропонує кращу продуктивність та безпеку завдяки статичній типізації Swift, що зменшує ризик помилок під час виконання, роблячи код більш надійним та передбачуваним. Мова Swift також забезпечує ефективне управління пам’яттю, що дозволяє значно оптимізувати споживання ресурсів сервера.

На відміну від Django, який використовує Python і працює на основі інтерпретованого підходу, Swift у поєднанні з Vapor забезпечує набагато швидше виконання запитів. Крім того, Swift має сучасний синтаксис і більшу продуктивність у порівнянні з Python, що робить її більш ефективнішою для масштабних проєктів.

Порівняно з Java та Spring, Vapor пропонує більш простий синтаксис та меншу складність у розробці. Слід зазначити, що мова програмування Swift позбавлена багатьох шаблонних підходів, які є у Java, що дозволяє розробникам писати більш лаконічний та зрозумілий код. Завдяки цьому, розробка на Swift потребує менше часу та зусиль, зберігаючи при цьому високу продуктивність та стабільність.

Swift у серверній розробці відкриває нові можливості для створення швидких, безпечних та гнучких вебзастосунків. Завдяки Vapor, Fluent та високій продуктивності мови, Swift стає чудовою альтернативою традиційним технологіям, дозволяючи розробникам працювати в єдиному технологічному стеку для мобільних, веб- та серверних застосунків.

1.4.4 Використання Swift у розробці Android-застосунків

Swift, який традиційно використовується для розробки застосунків в екосистемі Apple, поступово розширює свої можливості, отримуючи підтримку інших платформ, зокрема Android. Попри те, що офіційно Swift не має вбудованої підтримки для Android, завдяки зусиллям спільноти та новим технологіям стало можливим компілювати Swift-код для цієї платформи, що відкриває нові перспективи для розробників, які прагнуть використовувати єдиний стек технологій для кросплатформної розробки.

Одним із ключових напрямків адаптації Swift для Android є можливість компіляції Swift-коду у формат, сумісний із Dalvik/ART, який використовується в Android. Завдяки LLVM-компілятору Swift може генерувати нативний машинний код для архітектур ARM та x86, що дозволяє створювати високопродуктивні Android-застосунки на Swift. Це відкриває шлях до розробки нативних Android-застосунків із використанням усіх переваг Swift, включаючи його продуктивність, безпеку та сучасний синтаксис.

Важливим досягненням у цьому напрямку є проєкт Skip [11], який спрощує інтеграцію Swift у середовище Android. Skip дозволяє компілювати Swift у байт-код, сумісний із Android Runtime (ART), що дає змогу легко взаємодіяти з існуючими компонентами Android. Завдяки Skip розробники можуть використовувати Swift як основну мову програмування для Android-застосунків, зберігаючи всі переваги Swift, включаючи строгий контроль за типами, ефективне управління пам’яттю та високу швидкість виконання.

Skip також надає можливість інтеграції Swift-коду з Java та Kotlin, що дозволяє поступово впроваджувати Swift у вже існуючі Android-застосунки. Це особливо корисно для команд, які хочуть використовувати Swift у своєму мобільному стеку, але мають значний обсяг коду, написаного на інших мовах. Завдяки цьому підходу Swift може органічно доповнювати екосистему

Android, дозволяючи розробникам працювати з єдиним кодовим базисом для всіх платформ.

Swift пропонує низку переваг для Android-розробки, серед яких висока продуктивність, сучасний синтаксис та зручне управління пам’яттю за допомогою ARC (Automatic Reference Counting). Це дозволяє зменшити навантаження на процесор та оптимізувати споживання ресурсів, що робить Swift чудовим вибором для мобільних застосунків, які потребують високої продуктивності. Ще однією важливою перевагою є можливість використання Swift для кросплатформної розробки, що дозволяє зменшити витрати часу та ресурсів на створення окремих застосунків для iOS та Android. Використовуючи спільний кодовий базис, розробники можуть писати бізнес-логіку на Swift та лише адаптовувати UI під кожну платформу, що значно прискорює процес розробки.

Завдяки активному розвитку екосистеми Swift та підтримці ініціатив на кшталт Skip, перспективи використання Swift у мобільній розробці для Android виглядають надзвичайно багатообіцяючими. Swift вже довів свою ефективність у мобільних застосунках iOS, і його поступове проникнення в екосистему Android відкриває шлях до створення швидких, безпечних та продуктивних застосунків на єдиному технологічному стеку.

З подальшим розвитком інструментів та збільшенням підтримки Swift для Android можна очікувати, що ця мова стане ще більш привабливим вибором для мобільних розробників, які прагнуть працювати в уніфікованому середовищі, мінімізуючи потребу у використанні різних мов програмування для різних платформ.

1.4.5 Переваги Swift у кросплатформній розробці

Swift демонструє значний потенціал у сфері кросплатформної розробки завдяки своїм високим показникам безпеки, продуктивності та зручному синтаксису. Її строгий контроль за типами, відсутність нульових покажчиків та автоматичне управління пам’яттю значно зменшують кількість

потенційних помилок, що робить мову одним із найбезпечніших інструментів для розробки мобільних, веб- та серверних застосунків. Висока продуктивність досягається завдяки компіляції у машинний код, що забезпечує швидке виконання програм, а сучасний синтаксис спрощує читабельність та підтримку коду, сприяючи ефективнішій розробці.

Одна із головних переваг Swift у кросплатформній розробці полягає у використанні єдиного технологічного стеку. Swift може бути застосована для розробки мобільних застосунків для iOS та Android, серверних рішень за допомогою фреймворку Vapor, а також вебзастосунків через SwiftWasm та Tokamak. Така уніфікація значно зменшує необхідність використовувати кілька мов програмування для створення комплексних систем, що спрощує підтримку застосунків та знижує витрати на розробку.

Використання Swift для всіх частин програмного забезпечення дозволяє оптимізувати роботу команд розробників, оскільки їм не потрібно перемикатися між різними мовами та інструментами. Це спрощує оновлення застосунків, оскільки всі зміни можуть бути внесені в одному кодовому базисі без необхідності дублювання логіки. Такий підхід зменшує кількість помилок, покращує продуктивність розробників та прискорює вихід продукту на ринок.

Потенціал Swift як кросплатформної мови стає дедалі очевиднішим. Її інтеграція у серверну та веброзробку відкриває нові можливості для створення універсальних застосунків, а поступове розширення підтримки Android робить Swift ще привабливішим для мобільної розробки. Завдяки активному розвитку фреймворків та інструментів, Swift може стати повноцінною альтернативою для створення багатоплатформних рішень без необхідності використовувати сторонні технології.

Порівняно з іншими кросплатформними технологіями Swift має низку переваг. У порівнянні з Flutter вона пропонує нативний рендеринг UI без необхідності використання власного рушія, що забезпечує кращу продуктивність та відповідність гайдлайнам платформи. React Native

використовує міст між JavaScript та нативним кодом, що може створювати затримки в продуктивності, тоді як Swift виконується безпосередньо, зменшуючи накладні витрати. Kotlin Multiplatform забезпечує повторне використання логіки, але не підтримує універсальність Swift, яка може бути використана у веброзробці, серверних застосунках та мобільних платформах одночасно.

З огляду на всі переваги, Swift має всі шанси стати ключовою технологією для кросплатформної розробки, особливо якщо Apple або спільнота розробників продовжать розширювати її можливості на Android та інші платформи. Її висока продуктивність, безпека та сучасні інструменти роблять мову чудовим вибором для тих, хто прагне працювати в єдиній екосистемі, створюючи масштабовані та ефективні застосунки для різних платформ.

1.5 Інструменти для Swift

Swift має потужний набір інструментів, які роблять її універсальною мовою для розробки застосунків на різних платформах. Завдяки таким технологіям, як SwiftUI, Skip, Vapor, SwiftWasm та Tokamak, Swift може використовуватися для мобільної, серверної та веброзробки, що відкриває нові можливості для створення повністю кросплатформних рішень. На рисунку 1.4 наведено ключові інструменти для кросплатформної розробки на мові Swift.

SwiftUI [12] є офіційним фреймворком від Apple для створення інтерфейсів користувача на iOS, macOS, watchOS та tvOS. Його головна перевага – це декларативний підхід, який дозволяє описувати UI в простому та читабельному вигляді. SwiftUI значно спрощує розробку, оскільки розробники можуть будувати інтерфейси, використовуючи одну й ту ж саму структуру коду для всіх платформ Apple. Він підтримує функцію Hot Reload, що дає змогу миттєво переглядати зміни в інтерфейсі без необхідності повного перезапуску застосунку.

Інструмент	Призначення	Підтримувані платформи
SwiftUI	Фреймворк для створення UI на Apple-пристроях. Використовує декларативний підхід до створення інтерфейсу.	iOS, macOS, watchOS, tvOS
Skip	Компіляція Swift-коду для Android. Інтеграція Swift з екосистемою Android, підтримка Java та Kotlin.	Android
Vapor	Фреймворк для серверної розробки на Swift, заснований на асинхронній архітектурі. Використовується для створення API та вебсервісів.	Linux, macOS
Fluent	ORM для Swift, що забезпечує інтеграцію з базами даних, такими як PostgreSQL, MySQL та SQLite.	PostgreSQL, MySQL, SQLite
SwiftWasm	Компіляція Swift у WebAssembly, що дозволяє використовувати Swift для веброзробки.	Web
Tokamak	Декларативний UI-фреймворк для Web, що наслідує підхід SwiftUI. Інтегрується з WebAssembly.	Web

Рисунок 1.4 – Ключові інструменти для кросплатформної розробки на Swift

Завдяки інтеграції з Swift фреймворк забезпечує високу продуктивність і спрощує адаптацію UI до різних екранів та пристроїв. Skip є інноваційною технологією, яка дозволяє використовувати Swift для розробки Android-застосунків. Його головна задача – компілювати Swift у формат, сумісний із середовищем виконання Android (ART), що дає змогу запускати Swift-код безпосередньо на пристроях під управлінням цієї ОС. Skip інтегрується з екосистемою Android, дозволяючи взаємодіяти з Java та Kotlin, що відкриває можливість спільного використання коду між платформами. Завдяки цій технології розробники можуть використовувати Swift як основну мову для створення мобільних застосунків не тільки для iOS, але й для Android, що значно спрощує кросплатформну розробку та зменшує витрати на підтримку окремих кодових баз.

Vapor є провідним серверним фреймворком для Swift, який дозволяє створювати вебсервери та API. Завдяки використанню асинхронної архітектури на основі SwiftNIO, Vapor забезпечує високу продуктивність та масштабованість, що робить його чудовим вибором для розробки

високонавантажених систем. Він підтримує маршрутизацію HTTP-запитів, роботу з базами даних через ORM Fluent, автентифікацію користувачів та інші важливі серверні функції. Vapor дозволяє розробникам працювати в єдиному технологічному стеку, використовуючи Swift не тільки для мобільних застосунків, а й для серверної частини, що значно спрощує підтримку та розширення застосунків.

SwiftWasm відкриває можливості використання Swift у веброзробці, дозволяючи компілювати його у WebAssembly, що дає змогу запускати Swift-код безпосередньо у браузерах. SwiftWasm забезпечує високу продуктивність, оскільки виконується у WebAssembly, який працює набагато швидше, ніж інтерпретований JavaScript. Він також підтримує взаємодію з браузерними API через WebAssembly System Interface (WASI), що дозволяє інтегрувати Swift-код у сучасні вебзастосунки, що дає змогу розробникам писати вебінтерфейси, використовуючи той самий синтаксис, який вони вже знають зі Swift, і значно зменшує потребу у вивченні додаткових мов програмування.

Tokamak є потужним інструментом для створення вебінтерфейсів на основі Swift, забезпечуючи декларативний підхід, схожий на SwiftUI. Він дозволяє будувати інтерфейси, використовуючи знайомий для Swift-розробників синтаксис, що спрощує розробку кросплатформних застосунків. Tokamak працює за принципом віртуального DOM, що робить його схожим на React, але з повною інтеграцією у Swift. Завдяки Tokamak розробники можуть створювати сучасні вебінтерфейси, не використовуючи JavaScript, і легко інтегрувати вебверсію застосунку з мобільною та серверною частиною на Swift.

Поєднання SwiftUI, Skip, Vapor, SwiftWasm та Tokamak створює потужний стек технологій, який дозволяє використовувати Swift для повноцінної кросплатформної розробки. Завдяки цим інструментам можна створювати застосунки для мобільних платформ, вебінтерфейси та серверні рішення, використовуючи єдину мову програмування, що значно спрощує

розробку, зменшує витрати на підтримку різних кодових баз і забезпечує високу продуктивність застосунків. З розвитком цих технологій Swift має всі шанси стати основним вибором для кросплатформної розробки, об'єднуючи мобільні, серверні та веброзв'язання в єдиному екосистемному підході.

Крім того, Swift має сильну підтримку спільноти та відкритий код, що сприяє швидкому вдосконаленню екосистеми. Бібліотеки, такі як SwiftUI та Fluent, дають змогу створювати UI та бази даних у зрозумілий та декларативний спосіб. За допомогою SwiftWasm можлива компіляція в WebAssembly, що відкриває нові можливості для запуску Swift-застосунків у браузері. Tokamak, як інтерфейсний фреймворк, дозволяє реалізовувати UI для веба, використовуючи знайомі концепції SwiftUI. Це дає змогу розробникам переносити досвід iOS на інші платформи без суттєвих змін. Такий підхід знижує поріг входу та підвищує ефективність команд. У майбутньому варто очікувати ще більшої інтеграції між компонентами стеку та поліпшення інструментарію для розробки.

2 АНАЛІЗ СКЛАДОВИХ МОДЕЛІ, ЯКА ОРІЄНТОВАНА НА РОЗРОБКУ ВЕБ ТА МОБІЛЬНИХ ЗАСТОСУНКІВ

2.1 Аналіз останніх досліджень та публікацій

В ході дослідження було проведено аналіз сучасних підходів до розробки веб та мобільних застосунків з посиланням на низку наукових публікацій, які стосуються архітектури, інструментарію та принципів їх проєктування.

Робота [13] містить аналіз сучасних засобів розробки мобільних застосунків (React Native, Swift та Kotlin). Автори статті наводять переваги використання кросплатформних рішень для випадків, коли необхідно дуже швидко вивести продукт, що розробляється, на ринок. Також проведено аналіз нативних інструментів, і як висновок є те, що мова Swift для iOS, забезпечує найвищу продуктивність, стабільність та глибшу інтеграцію з платформою.

Публікації [14] та [15] аналізують використання архітектурних підходів (паттернів) для мобільних застосунків. Слід зазначити, що автори роблять наголос на те, що паттерни MVC та MVVM орієнтовані на оптимізацію розробки front-end мобільного застосунку. Це пов’язано з тим, що вони надають можливість розділити бізнес-логіку та інтерфейс користувача. Завдяки використанню такого підходу спрощується проєктування, розробка, тестування та масштабування кінцевого програмного застосунку. Публікація [14] містить аналіз архітектурного принципу REST API і пропонується використання його як засіб комунікації між клієнтською та серверною частинами мобільного засосунку. Використання такого підходу дозволяє полегшити інтеграцію різних компонентів застосунку та стандартизувати передачу даних між його окремими частинами.

В роботі [16] наведено підхід, який пропонує поділ мобільного Andriod-застосунку на шари, які з’єднані відповідними правилам, що

дозволяє у подальшому здійснювати масштабування проєкту без зайвих зусиль. Такий поділ автори рекомендують виконувати за допомогою мови Kotlin та набору бібліотек та компонентів підходу Clean Architecture.

Робота [17] ілюструє модель мобільного застосунку, яка будується на основі множини взаємопов'язаних між собою елементів: інтерфейсу користувача, клієнтської та серверної частин, набору інструментів безпеки та тестування. Також автори наголошують на необхідності впровадження в модель модулю моніторингу з функціоналом подальшої аналітики, отриманих результатів.

Робота [18] містить інформацію стосовно моделі фреймворку для побудови чат-ботів. Наведена модель містить архітектурні рішення, які спрямовані на вирішення специфічних задач чат-боту. Автори пропонують розширити функціональність складових фреймворку з огляду на нову бот-перспективу за рахунок розширення функціоналу модулю безпеки для шифрування тексту повідомлень, а також модифікувати модуль адміністрування для вирішення специфічних задач чат-боту.

У роботі [19] описано структуру та функціонал мобільного застосунку у зв'язці зі службами, з якими він взаємодіє. Основний наголос зроблено на розробці інтерфейсу користувача, тому що він є ключовим об'єктом при роботі як із сторонніми ресурсами, так із базою даних. Запропонований застосунок за результатами використання демонструє високу гнучкість та масштабованість завдяки додаванню сторонніх служб до застосунку.

Робота [20] ілюструє модель мобільного застосунку для ОС iOS, яка побудована на наборі шаблонів та використовує сервіси для роботи з мережею та базами даних. Запропонована модель рекомендує використання не тільки існуючих механізмів, а спробу розширення базових класів, які допомагають розробнику з навігацію, що побудована на основі координатора.

Стаття [21] містить аналіз використання JavaScript-фреймворків (Angular, React, Vue, Backbone, Ember, Knockout) для розробки front-end.

Набір інструментів, які розглянуто, може надавати розробнику ряд переваг, але при проєктуванні і подальшій розробці мобільного застосунку слід відштовхуватись від потреб замовника. В якості висновку автори говорять, що більшість фреймворків, які було розглянуто, не забезпечують можливість реалізації як клієнтської, так і серверної частин застосунків однією мовою, що ускладнює інтеграцію та збільшує витрати на їх розробку.

У роботі [22] модель веброзробки розглядається як граф та використовується у зв'язці із алгоритмом сканування сторінок, який запропонували автори. Інформація, яку отримано після сканування, можливо використовувати у подальшому для обчислення метричних характеристик, які дозволяють провести аналіз структурної зв'язності вебзастосунків різного типу складності.

Стаття [23] пропонує використовувати фреймворк Ember.js для вебзастосунків при розробці front-end. Інструментарій, яким оперує Ember.js дає розробнику можливість розгортати складні клієнтські застосунки. Але в якості недоліку автори вказують те, що даний фреймворк не забезпечує універсальність, дозволяючи реалізовувати клієнтську та серверну частини однією мовою.

Проведений порівняльний аналіз виявив, що на даний час не має уніфікованої моделі, яка може бути використана для побудови мобільних та вебзастосунків. Є деякі спроби авторів навести моделі або для веброзробок, або для мобільних застосунків. Аналіз попередніх публікацій показав, що для більшості моделей присутні складові, які наведено у (2.1):

$$M = \{UI, BL, DB, API\}. \tag{2.1}$$

де UI – інтерфейс користувача;

BL – модуль бізнес-логіки;

DB – база даних, яка орієнтована на зберігання даних;

API – інтерфейс взаємодії клієнтської та серверної частин.

2.2 Модель уніфікованого підходу для розробки мобільних та вебзастосунків з використанням мови Swift

Проведений аналіз існуючих аналогів застосунків виявив певні обмеження у розуміння сучасних підходів до їх розробки. Перш за все, більшість із них зосереджуються на використанні сторонніх фреймворків для back-end, таких як Node.js або Django, тоді як застосування нативного back-end, написаного на мові Swift із використанням фреймворку Vapor, залишається недостатньо вивченим. Такий підхід може суттєво підвищити продуктивність системи та забезпечити більш природну інтеграцію між клієнтською та серверною частинами. Крім того, у дослідженнях недостатньо уваги приділено перевагам SwiftUI як сучасного інструменту для створення адаптивного та інтерактивного інтерфейсу користувача, який значно спрощує розробку завдяки використання декларативного підходу.

Зважаючи на виявлені недоліки, було представлено модель (2.2), яка дозволяє подолати існуючі обмеження та забезпечує ефективну інтеграцію front-end та back-end компонентів, високий рівень продуктивності, безпеки та легкість масштабування. Ця модель враховує всі ключові аспекти сучасної розробки, забезпечуючи комплексний підхід до створення програмних застосунків (веб та мобільних):

$$M=\{P, A, PL, UI, UX, SC, DB, NI, S, TS, AM\}, \tag{2.2}$$

де: P (platform) – платформа розробки;

A (architecture) – архітектура розробки;

PL (programming language) – мова програмування;

UI (user interface) – інтерфейс користувача;

UX (user experience) – досвід користувача;

SC (server component) – серверна складова застосунку;

DB (database) – база даних;

NI (network interface) – мережний інтерфейс;

S (security) – безпека застосунку;

TS (test suite) – набір тестів;

AM (analytics and monitoring) – аналітика та моніторинг застосунку.

Запропонована модель дозволяє зменшити вартість розробки завдяки єдиному стеку технологій. Оскільки всі компоненти моделі можуть бути реалізовані за допомогою мови Swift, зникає потреба у використанні додаткових мов або технологій для різних частин системи, що особливо важливо в умовах обмежених ресурсів, коли кожна година розробника має значення. Крім того, цей підхід підвищує консистентність коду, оскільки всі частини проєкту розробляються в єдиному стилі та з використанням єдиних парадигм. Кожен із цих компонентів впливає на загальну структуру застосунку, створюючи стабільну, продуктивну та легко масштабовану архітектуру. Розглянемо більш детально кожен з них та його роль у запропонованій моделі.

2.3 Обґрунтування складових уніфікованої моделі для розробки мобільних та вебзастосунків

На сучасному ринку розробки програмного забезпечення основними платформами (P) є iOS, Android, веб та кросплатформні рішення. Кожна з них має свої переваги: iOS вирізняється стабільністю та глибокою інтеграцією з екосистемою Apple; Android забезпечує гнучкість та широке охоплення пристроїв; веб пропонує універсальність, а кросплатформні інструменти дозволяють одночасно створювати програми для кількох середовищ.

Запропонована модель орієнтована на платформу iOS із можливістю розширення для iPadOS. Вона забезпечує високу продуктивність, глибоку інтеграцію з екосистемою Apple (iCloud, Apple Pay, HealthKit) та підтримку

інноваційних технологій, таких як ARKit та Core ML, що робить її ідеальним вибором для створення сучасних, стабільних та масштабованих рішень.

Архітектура (А) розробки мобільних та вебзастосунків – це сукупність підходів, правил та шаблонів, які забезпечують ефективну організацію роботи програмного забезпечення. На сьогодні найбільш поширеними архітектурними підходами (патернами) є:

- MVC (Model-View-Controller);
- MVVM (Model-View-ViewModel);
- VIPER (View-Interactor-Presenter-Entity-Router);
- Clean Architecture.

MVC використовується для структурування застосунків шляхом розділення даних (Model), інтерфейсу користувача (View) та логіки управління (Controller). Даний патерн підходить для невеликих та середніх проєктів, забезпечує простоту реалізації та швидке тестування, але ускладнюється процес масштабування застосунку у разі потреби.

MVVM забезпечує чітке розділення логіки та інтерфейсу завдяки використанню ViewModel, що спрощує процеси тестування, масштабування та роботу з асинхронними даними. Використання даного підходу ідеально підходить для front-end розробки мобільних та вебзастосунків, де необхідна гнучкість та чітка структура.

VIPER слід використовувати для складних проєктів, де необхідне забезпечення модульності та високої масштабованість, що досягається завдяки розділенню відповідальностей. Як зауваження при використанні цього архітектурного підходу слід зазначити, що він є складнішим в реалізації та вимагає більшого часу на розробку.

Clean Architecture передбачає побудову системи з чітко визначеними рівнями (інтерфейс, бізнес-логіка, дані), що робить її максимально масштабованою та адаптивною. Використання даної архітектури доцільно для великих проєктів, але також вимагає додаткових ресурсів.

У запропонованій моделі використовуються архітектурний підхід

MVVM на front-end та MVC на серверній частині, оскільки вони поєднують простоту реалізації та високу ефективність:

- MVVM забезпечує відокремлення логіки від інтерфейсу, що сприяє легкому оновленню дизайну та підтримці коду, а також дозволяє ефективно працювати з SwiftUI;

- MVC у фреймворку Vapor структурує серверний код, роблячи його зрозумілим, підтримуваним та зручним для інтеграції з базою даних та API.

В якості висновку слід зазначити, що обрані архітектурні підходи дозволяють створити гнучку, масштабовану та стабільну систему, яка відповідає сучасним вимогам розробки мобільних та вебзастосунків.

Одним із популярних підходів при кросплатформній розробці є використання фреймворку Flutter та мови програмування Dart. Ця зв'язка дозволяє створювати застосунки для iOS та Android з єдиним кодом, пропонує широкий набір віджетів для інтерфейсу, підтримує плавні анімації та асинхронну обробку даних. Проте її обмеження включає знижену продуктивність порівняно з нативними рішеннями, труднощі доступу до платформних API та збільшений розмір додатку.

Запропонована модель (2.2) використовує мову Swift для клієнтської частини та фреймворк Vapor для серверної. Мова програмування Swift забезпечує високу продуктивність та сучасні можливості, такі як асинхронна обробка даних, а фреймворк Vapor дозволяє створювати REST API, керувати базами даних через ORM Fluent та забезпечувати безпеку застосунку. Завдяки єдиній мовній базі забезпечується безшовна інтеграція між компонентами, що скорочує час розробки, підвищує стабільність та спрощує масштабування застосунку у разі потреби.

Інтерфейс користувача (UI) є ключовим елементом будь-якого застосунку, і використання сучасних фреймворків значно спрощує його розробку. Наприклад, SwiftUI забезпечує декларативний підхід, що дозволяє розробникам створювати складні інтерфейси з мінімумом коду. Компоненти, створені за допомогою SwiftUI автоматично адаптуються під різні розміри

екранів, що особливо важливо для пристроїв iOS та iPadOS. Крім того, SwiftUI підтримує інтерактивні елементи, анімації та плавні переходи, покращуючи загальний досвід користувача.

Ще одним прикладом є технологія Jetpack Compose для ОС Android, яка також використовує декларативний підхід. Вона спрощує створення складних макетів та взаємодій, а також забезпечує високу швидкість розробки завдяки своїй інтеграції з іншими інструментами екосистеми Android.

У випадку використання SwiftUI головною перевагою є глибока інтеграція з екосистемою Apple, що дозволяє ефективно використовувати унікальні можливості платформи, такі як динамічні шрифти, системні кольори та доступність, що робить його ідеальним вибором для розробки інтерфейсів, які відповідають сучасним стандартам продуктивності та дизайну.

Досвід користувача (UX) визначається емоціями, враженнями та рівнем задоволення, які отримує користувач під час взаємодії з програмним продуктом. З точки зору розробника, ключовою задачею є створення такого інтерфейсу та функціоналу, які не лише задовольняють потреби користувача, але й перевищують його очікування, забезпечуючи зручність, доступність та естетичну привабливість. Дотримання рекомендацій Apple у Human Interface Guidelines (HIG) допомагає розробникам створювати інтуїтивно зрозумілі та гармонійні застосунки. Нативні компоненти, такі як системні кнопки, списки, вкладки та піктограми, забезпечують єдиний стиль та спрощують адаптацію до інтерфейсу. Принципи HIG гарантують не лише функціональність, але й узгодженість дизайну з іншими додатками екосистеми Apple.

З точки зору користувача, плавні анімації, інтерактивні елементи та логічно організовані розділи допомагають утримувати увагу та створюють відчуття комфорту під час роботи з застосунком. Використання рекомендацій HIG забезпечує зручну навігацію та естетику, що сприяє довірі до продукту. Наприклад, адаптація дизайну для різних екранів, оптимізація часу відгуку та

забезпечення доступності для людей з обмеженими можливостями роблять UX основою успішного застосунку.

Серверна частина (SC) є важливим компонентом мобільних та вебзастосунків, забезпечуючи обробку даних, реалізацію бізнес-логіки та інтеграцію з іншими сервісами. У мобільних додатках сервер обробляє запити клієнтів та зберігає дані, тоді як у вебзастосунках він часто також відповідає за динамічне формування контенту. Основними викликами серверної розробки є забезпечення продуктивності, безпеки, масштабованості та ефективної взаємодії з базами даних.

У запропонованій моделі серверна частина реалізується за допомогою фреймворку Vapor на мові Swift. Vapor дозволяє створювати REST API для взаємодії з клієнтом, ефективно працювати з базами даних через ORM Fluent та забезпечує високий рівень безпеки завдяки підтримці middleware для аутентифікації та шифрування. Асинхронна архітектура Vapor забезпечує високу продуктивність та здатність обробляти велику кількість запитів, роблячи його ідеальним вибором для сучасних застосунків. Використання єдиної мовної бази на основі Swift забезпечує природну інтеграцію між клієнтською та серверною частинами, створюючи цілісну та ефективну систему.

База даних (DB) є ключовим компонентом будь-якого сучасного застосунку, що забезпечує надійне зберігання та управління інформацією. У вебзастосунках база даних використовується для зберігання контенту, даних користувача, а також журналів транзакцій. У мобільних застосунках вона часто виступає у ролі локального сховища для тимчасового зберігання даних або кешування, що забезпечує швидкий доступ до інформації навіть без підключення до мережі.

Для зберігання даних у запропонованій моделі використовується PostgreSQL – реляційна база даних, відома своєю стабільністю, безпекою та масштабованістю. ORM Fluent у Vapor дозволяє значно спростити роботу з PostgreSQL. Він забезпечує створення моделей для роботи з даними,

управління міграціями та побудову запитів без необхідності прямого використання SQL, що сприяє скороченню часу розробки та зменшенню кількості помилок, пов’язаних із взаємодією з базою даних.

Така інтеграція дозволяє забезпечити швидку та зручну взаємодію між серверною частиною застосунку та базою даних. PostgreSQL у поєднанні з Fluent робить модель ефективною, масштабованою та готовою до роботи з великими обсягами даних та складною бізнес-логікою. Цей підхід забезпечує стабільну основу для зберігання інформації, адаптовану для мобільних та вебзастосунків.

Мережні з'єднання (NI) забезпечують обмін даними між клієнтською та серверною частинами через REST API. У сучасних застосунках це здійснюється через стандартні протоколи HTTP/HTTPS, які гарантують безпеку та простоту передачі даних. Наприклад, клієнт (мобільний або веб) формує запит до серверу за допомогою інструментів, таких як URLSession на iOS, і отримує відповіді у форматі JSON, що легко обробляється.

Використання архітектурного стилю REST API в запропонованій моделі є оптимальним вибором для мережної взаємодії завдяки його простоті, стандартизованості та ефективності. Він дозволяє використовувати зрозумілі HTTP-методи (GET, POST, PUT, DELETE) для виконання операцій над ресурсами, а передача даних у форматі JSON спрощує інтеграцію між клієнтом та сервером. У запропонованій моделі реалізація стилю REST API здійснюється за допомогою фреймворку Vapor, що дозволяє легко створювати маршрути для обробки запитів та повертати структуровані дані. Такий підхід забезпечує стабільну, швидку та безпечну комунікацію між компонентами системи.

Безпека (S) є одним із найважливіших аспектів сучасних програмних застосунків через зростаючу кількість конфіденційних даних, що обробляються. Витоки особистої інформації, фінансових даних або зломи систем можуть призводити до значних збитків, тому забезпечення захисту даних є пріоритетом для будь-якого кінцевого продукту.

У сучасних застосунках використовуються такі методи захисту, як шифрування запитів через протокол HTTPS, аутентифікація за допомогою JWT-токенів та двофакторна авторизація. Протокол HTTPS гарантує шифрування всіх переданих даних, захищаючи їх від перехоплення, тоді як JWT забезпечує безпечну ідентифікацію користувачів. Крім того, використовуються захищені протоколи для зберігання даних та валідація на стороні серверу для запобігання SQL-ін’єкціям.

У запропонованій моделі реалізація безпеки здійснюється через фреймворк Vapor, який підтримує middleware для аутентифікації, обробку JWT-токенів та захист від несанкціонованого доступу. Шифрування запитів через HTTPS забезпечує надійність передачі даних, а налаштування middleware дозволяють адаптувати систему до вимог сучасних стандартів безпеки.

Тестування (TS) є обов’язковим етапом розробки застосунку, який забезпечує стабільність, надійність та якість програмного забезпечення. Воно дозволяє виявляти та усувати помилки на ранніх етапах, що мінімізує ризик збоїв після впровадження змін чи масштабування застосунку. Тестування також гарантує, що функціонал працює відповідно до заданих специфікацій, забезпечуючи якісний досвід користувача.

У розробці застосовуються різні види тестування:

- unit тести, які перевіряють окремі модулі чи компоненти системи;
- UI-тести, які оцінюють коректність роботи інтерфейсу користувача;
- інтеграційні тести, які перевіряють взаємодію між різними компонентами системи;
- тестування API, яке гарантує стабільність та продуктивність серверної частини.

У запропонованій моделі використовується інструмент XCTest, який інтегрований у середовище розробки Xcode, що дозволяє створювати та виконувати як unit, так і UI-тести для клієнтської частини. Для серверної частини, яка розроблена на Vapor, проводяться тестування API та

інтеграційні тести, які забезпечують коректність взаємодії між сервером та клієнтом. Завдяки цьому модель гарантує стабільність застосунку навіть при внесенні змін або масштабуванні, а також зберігає високу якість роботи в реальних умовах.

Аналітика та моніторинг (АМ) є важливими складовими сучасного застосунку, які забезпечують глибоке розуміння поведінки користувачів та продуктивності системи. Інтеграція інструментів аналітики дозволяє відстежувати активність користувачів, аналізувати взаємодію з різними елементами інтерфейсу та збирати ключові метрики, такі як конверсії, час сесій та залученість.

Моніторинг продуктивності системи забезпечує можливість своєчасного виявлення та вирішення потенційних проблем, таких як високий час відгуку сервера, перевантаження бази даних чи збої в роботі API. Використання таких інструментів, як Google Analytics, Firebase Analytics, Mixpanel або Sentry, дозволяє автоматизувати процес збору та аналізу даних, забезпечуючи актуальну інформацію для прийняття рішень.

У запропонованій моделі аналітика та моніторинг інтегровані як невід'ємна частина системи, яка побудована з використанням мови Swift. Інструменти аналітики збирають дані на клієнтській стороні через SwiftUI та URLSession, а моніторинг продуктивності реалізується на серверній частині за допомогою фреймворку Vapor, що дозволяє розробникам отримувати комплексну картину роботи застосунку, враховувати реальні дані користувачів під час оптимізації функціоналу та забезпечувати стабільність системи.

Такий підхід підвищує якість досвіду користувача, дозволяє своєчасно реагувати на проблеми й адаптувати функціонал відповідно до потреб цільової аудиторії. Інтеграція аналітики та моніторингу уніфікує процес управління даними, сприяючи підвищенню ефективності розробки та підтримки застосунків.

3 РОЗРОБКА УНІФІКОВАНОГО ПРОТОТИПУ НА ПРИКЛАДІ ЗАСТОСУНКУ ДЛЯ ОБЛІКУ ФІНАНСІВ

У сучасному цифровому світі, де користувачі очікують доступу до своїх даних із будь-якого пристрою – чи то смартфон на iOS або Android, чи веббраузер, – розробка кросплатформних застосунків стає не просто трендом, а необхідністю. Одним із викликів при створенні таких рішень є забезпечення єдності коду, мінімізація дублювання логіки та підтримка консистентності між різними платформами. У цьому розділі представлено розробку уніфікованого прототипу на прикладі застосунку для обліку фінансів, який демонструє можливості Full Swift Stack як єдиного технологічного стеку для створення кросплатформних рішень.

Уніфікований прототип покликаний показати, як використання однієї мови програмування (Swift) для всіх компонентів системи (клієнтських інтерфейсів, серверної частини та вебверсії) може значно спростити розробку, знизити витрати на підтримку та прискорити виведення продукту на ринок. Прототип реалізовано на основі запропонованої моделі (2.2), яка охоплює ключові аспекти розробки: від вибору платформ (P) та архітектури (A) до забезпечення безпеки (S) та аналітики (AM).

Застосунок для обліку фінансів дозволяє користувачам реєструватися, автентифікуватися, вести облік доходів та витрат, переглядати фінансову статистику у вигляді графіків та звітів, а також синхронізувати дані між iOS, Android та вебверсією через сервер. У процесі розробки особливу увагу приділено уніфікації технологічного стеку, що включає SwiftUI та Combine для iOS, Skip для Android, Vapor та Fluent для серверної частини, а також SwiftWasm та Tokamak для вебінтерфейсу. Такий підхід не лише забезпечує консистентність коду, але й дозволяє повторно використовувати бізнес-логіку між платформами, зменшуючи час на розробку та підвищуючи якість продукту. У розділі детально описано кожен етап створення прототипу, починаючи від налаштування бекенду та бази даних, закінчуючи

розгортанням та тестуванням, із чітким акцентом на те, як уніфікований підхід сприяє ефективності, безпеці, масштабованості та зручності для кінцевого користувача. Завдяки інтеграції всіх компонентів моделі та використанню сучасних практик, таких як автоматизація через CI/CD, уніфікований прототип демонструє потенціал Full Swift Stack для створення конкурентоспроможних кросплатформних рішень.

3.1 Опис уніфікованого прототипу

Мета створення уніфікованого прототипу полягає в розробці кросплатформного застосунку для обліку фінансів, який забезпечує користувачам зручний та ефективний інструмент для управління особистими фінансами з можливістю доступу до даних із різних пристроїв. Застосунок дозволяє вести облік доходів та витрат, аналізувати фінансову поведінку через статистику та синхронізувати дані між платформами, забезпечуючи безперебійний досвід користувача незалежно від обраного пристрою. Прототип слугує практичною демонстрацією можливостей Full Swift Stack, показуючи, як єдиний технологічний стек на основі мови Swift може бути використаний для створення консистентного, продуктивного та масштабованого рішення для iOS, Android та вебсередовища.

Основні функції застосунку охоплюють повний цикл управління фінансами. Користувачі можуть пройти реєстрацію та автентифікацію для безпечного доступу до своїх даних, використовуючи email та пароль. Після входу в систему вони отримують можливість додавати фінансові транзакції, вказуючи суму, категорію (наприклад, "Продукти", "Транспорт", "Розваги"), тип (дохід або витрата) та дату операції. Застосунок також підтримує редагування та видалення транзакцій, що дозволяє користувачам виправляти помилки або оновлювати інформацію. Важливою функцією є перегляд статистики, яка включає графіки та звіти за обраними періодами (день, тиждень, місяць), наприклад, співвідношення доходів та витрат або розподіл

витрат за категоріями. Для забезпечення доступу до даних із різних пристроїв реалізовано синхронізацію через сервер, що дозволяє користувачу почати роботу на iPhone, продовжити на Android-пристрої, а потім переглянути статистику у веббраузері, не втрачаючи жодних даних.

Цільові платформи прототипу включають iOS, Android та Web, що відповідає компоненту P (платформа розробки) уніфікованої моделі 2.2. Вибір цих платформ зумовлений їхньою популярністю серед користувачів та необхідністю забезпечити максимальну доступність застосунку. Для iOS та Android розробляються нативні мобільні застосунки, які враховують особливості кожної операційної системи, а веберсія забезпечує доступ через браузер, що особливо зручно для користувачів, які працюють із настільних комп'ютерів або не мають можливості встановити застосунок.

Технологічний стек прототипу базується на концепції Full Swift Stack, що передбачає використання мови Swift як єдиної мови програмування для всіх компонентів системи (PL – мова програмування). Для iOS застосовується SwiftUI у поєднанні з Combine, що дозволяє створювати декларативний та реактивний інтерфейс із підтримкою асинхронного оновлення даних. На Android використовується Skip – інструмент, який компілює Swift-код у нативний Android-застосунок, адаптуючи інтерфейс до стандартів Material Design. Серверна частина (backend) реалізується за допомогою фреймворку Vapor, який працює на Swift, із використанням Fluent як ORM для взаємодії з базою даних PostgreSQL, що забезпечує надійне зберігання даних. Для веберсії застосовується SwiftWasm, який компілює Swift-код у WebAssembly, та Tokamak – бібліотека для створення декларативного UI, подібного до SwiftUI, але для браузера. Такий уніфікований технологічний стек дозволяє не лише зменшити складність розробки, а й повторно використовувати код між платформами, зокрема бізнес-логіку та моделі даних.

На цьому етапі розробки визначаються ключові компоненти моделі 2.2, які формують основу уніфікованого підходу. Компонент P (платформи)

охоплює iOS, Android та Web, задаючи цільові середовища для розгортання додатку. Компонент PL (мова програмування) фіксує вибір Swift як єдиної мови, що забезпечує консистентність коду та спрощує інтеграцію між клієнтськими та серверними частинами. Окрім того, вибір технологій вже частково торкається інших компонентів моделі, таких як UI (інтерфейс користувача), який буде реалізовано через SwiftUI, Skip, Tokamak та SC (серверна складова), яка представлена Vapor. Подальші етапи розробки розкриють, як інші компоненти моделі, зокрема UX, DB, S, та AM інтегруються в прототип, забезпечуючи його функціональність, безпеку та зручність використання.

3.2 Розгортання серверної частини (Backend)

Серверна частина (backend) застосунку для обліку фінансів розробляється з використанням фреймворку Vapor, який дозволяє створювати швидкі та масштабовані REST API на мові Swift. Це забезпечує уніфікацію технологічного стеку, оскільки Swift використовується як для серверної, так і для клієнтської частин, що відповідає компоненту (PL) (мова програмування) уніфікованої моделі 2.2. Backend відповідає за обробку запитів від клієнтських застосунків (iOS, Android, Web), управління даними користувачів та транзакцій, а також реалізацію механізмів безпечної взаємодії через автентифікацію та захист ендпоінтів

Для обробки запитів реалізується REST API з набором ендпоінтів, які покривають основні функції застосунку: POST/register для реєстрації, POST/login для автентифікації з видачею JWT-токена, GET/POST/PUT/DELETE/transactions для управління транзакціями та GET/statistics для отримання фінансової статистики (наприклад, сума доходів та витрат за місяць). Vapor використовує SwiftNIO для асинхронної обробки запитів, що забезпечує високу продуктивність. Дані передаються у форматі JSON, що відповідає стандартам RESTful API. Ця частина реалізації

відноситься до компоненту NI (мережний інтерфейс), який забезпечує обмін даними між клієнтом та сервером, а також до SC (серверна складова), представленого Vapor.

Для зберігання даних використовується PostgreSQL з Fluent ORM, яка інтегрується з Vapor та дозволяє працювати з базою даних на рівні об’єктів Swift. Створюються дві таблиці: user (поля: id – UUID, email – String, password_hash – String) та transactions (поля: id – UUID, user_id – UUID, amount – Double, category – String, type – Enum: income/expense, date – DateTime). Модель для таблиці user визначена в коді лістингу 3.1. Аналогічно, модель для таблиці transaction наведена в лістингу 3.2.

Лістинг 3.1 – Модель User

```
import Fluent
import Vapor
final class User: Model, Content {
    static let schema = "users"
    @ID(key: .id) var id: UUID?
    @Field(key: "email") var email: String
    @Field(key: "password_hash") var passwordHash: String
    init() {}
    init(id: UUID? = nil, email: String, passwordHash: String) {
        self.id = id
        self.email = email
        self.passwordHash = passwordHash
    }      }
```

Лістинг 3.2 – Модель Transaction

```
import Fluent
import Vapor
final class Transaction: Model, Content {
    static let schema = "transactions"
    @ID(key: .id) var id: UUID?
    @Parent(key: "user_id") var user: User
    @Field(key: "amount") var amount: Double
    @Field(key: "category") var category: String
    @Field(key: "type") var type: String
    @Field(key: "date") var date: Date
    init() {}
    init(id: UUID? = nil, userID: UUID, amount: Double,
category: String, type: String, date: Date) {
        self.id = id
        self.$user.id = userID
```

```
        self.amount = amount
        self.category = category
        self.type = type
        self.date = date
    }     }
```

Для створення таблиць у PostgreSQL використовуються міграції. Приклад міграції для таблиці user наведено у лістингу 3.3. Частина реалізації, яка наведена у лістингу 3.3, відповідає компоненту (DB) (база даних), який забезпечує надійне зберігання даних.

Лістинг 3.3 – Міграція CreateUser

```
import Fluent
struct CreateUser: Migration {
    func prepare(on database: Database) -> EventLoopFuture<Void>
{
        database.schema("users")
            .id()
            .field("email", .string, .required)
            .field("password_hash", .string, .required)
            .unique(on: "email")
            .create()
    }
    func revert(on database: Database) -> EventLoopFuture<Void>
{
        database.schema("users").delete()
    }     }
```

Автентифікація користувачів реалізується через JSON Web Token (JWT). При реєстрації користувач надсилає email та пароль, пароль хешується за допомогою BCrypt, зберігається в базі, а користувачу видається JWT-токен. При вході перевіряється хеш пароля та видається новий токен. Захищені ендпоінти, такі як /transactions, вимагають токен у заголовку Authorization: Bearer <token>. Приклад маршруту для реєстрації наведено в лістингу 3.4.

Лістинг 3.4 – Маршрут реєстрації

```
import Vapor
struct UserController {
    func register(req: Request) throws ->
```

```
EventLoopFuture<UserResponse> {
        let userData = try req.content.decode(UserData.self)
        let passwordHash = try Bcrypt.hash(userData.password)
        let user = User(email: userData.email, passwordHash:
passwordHash)
        return user.save(on: req.db).map {
            let token = try req.jwt.sign(.init(with: ["sub":
user.id!.uuidString]))
            return UserResponse(id: user.id!, email: user.email,
token: token)
        }
    }
}
struct UserData: Content {
    let email: String
    let password: String
}
struct UserResponse: Content {
    let id: UUID
    let email: String
    let token: String
}
```

Для захисту маршрутів використовується middleware, як показано в лістингу 3.5.

Лістинг 3.5 – Захищений маршрут import Vapor

```
struct TransactionController {
    func getTransactions(req: Request) throws ->
EventLoopFuture<[Transaction]> {
        let user = try req.auth.require(User.self)
        return Transaction.query(on: req.db)
            .filter(\.$user.$id == user.id!)
            .all()
    }     }
app.grouped(User.authenticator(), User.guardMiddleware())
```

В лістингу 3.5 наведена частина реалізації відповідає компоненту (S) (безпека), який забезпечує захист через автентифікацію, шифрування паролів і перевірку токенів. Таким чином, серверна частина охоплює кілька компонентів моделі:

- SC (серверна складова) реалізується через Vapor;
- DB (база даних) реалізується через PostgreSQL та Fluent;
- NI (мережний інтерфейс) реалізується через REST API;

- S (безпека) реалізується через JWT та шифрування.

Використання Swift та Vapor забезпечує уніфікацію з іншими частинами системи.

3.3 Мобільний застосунок

Мобільний застосунок для обліку фінансів розробляється з використанням MVVM-архітектури, що відповідає компоненту (A) (архітектура розробки) уніфікованої моделі 2.2. MVVM забезпечує чіткий розподіл між бізнес-логікою, представленням даних та інтерфейсом користувача, що спрощує тестування та підтримку коду. Застосунок підтримує дві платформи: iOS та Android, використовуючи уніфікований підхід на основі Full Swift Stack. Для iOS застосовується SwiftUI у поєднанні з Combine для створення декларативного інтерфейсу та реактивного оновлення даних, що дозволяє швидко реагувати на зміни, наприклад, при отриманні нових транзакцій із сервера.

Інтерфейс включає основні екрани: екран входу/реєстрації для автентифікації користувача, головний екран зі списком транзакцій, екран додавання нової транзакції (з полями для суми, категорії, типу та дати) та екран статистики, де відображаються графіки доходів та витрат за обраний період.

Для Android використовується Skip – інструмент, який компілює Swift-код у нативний Android-застосунок, адаптуючи інтерфейс до стандартів Material Design, щоб забезпечити природний вигляд та поведінку для користувачів Android. Skip дозволяє повторно використовувати той самий Swift-код, написаний для iOS, із мінімальними змінами для рендерингу UI, що значно економить час розробки. Спільна бізнес-логіка, яка включає моделі даних (наприклад, структури User та Transaction), мережні запити через URLSession для взаємодії з REST API та логіку обробки статистики (обчислення сум за категоріями чи періодами), використовується на обох

платформах без дублювання. Це усуває необхідність писати окремий код для iOS та Android, забезпечуючи консистентність та спрощуючи підтримку. Приклад ViewModel для управління транзакціями наведено в лістингу 3.6.

Лістинг 3.6 – ViewModel транзакцій

```
import Combine
import Foundation
class TransactionViewModel: ObservableObject {
    @Published var transactions: [Transaction] = []
    func fetchTransactions() {
        guard let url = URL(string:
"https://api.example.com/transactions") else { return }
        URLSession.shared.dataTask(with: url) { data, _, _ in
            if let data = data, let transactions = try?
JSONDecoder().decode([Transaction].self, from: data) {
                DispatchQueue.main.async {
                    self.transactions = transactions
                }
            }
        }.resume()
    }     }
```

Цей ViewModel використовується для асинхронного завантаження транзакцій із сервера та оновлення UI через Combine. Приклад SwiftUI-екрану для відображення списку транзакцій на iOS наведено в лістингу 3.7.

Лістинг 3.7 – Екран списку транзакцій

```
import SwiftUI
struct TransactionListView: View {
    @StateObject private var viewModel = TransactionViewModel()

    var body: some View {
        NavigationView {
            List(viewModel.transactions) { transaction in
                HStack {
                    Text(transaction.category)
                    Spacer()
                    Text("\(transaction.amount, specifier:
"%.2f")")
                }
            }
            .navigationTitle("Transactions")
            .onAppear { viewModel.fetchTransactions() }
        }
    }     }
```

На Android той самий ViewModel через Skip адаптується для рендерингу UI у стилі Material Design, зберігаючи спільну логіку. Мережні запити до REST API, які використовуються для отримання даних, наприклад, списку транзакцій чи статистики, реалізуються через URLSession, що відповідає компоненту NI (мережний інтерфейс) для зв'язку з сервером.

Інтерфейс користувача (UI) реалізується через SwiftUI для iOS та Skip для Android, забезпечуючи нативний вигляд та поведінку на кожній платформі. Досвід користувача (UX) досягається завдяки інтуїтивному дизайну, адаптивності до різних розмірів екранів та плавним анімаціям, що робить застосунок зручним у використанні. Таким чином, мобільний застосунок охоплює кілька компонентів моделі:

- UI (інтерфейс користувача) через SwiftUI та Skip;
- UX (досвід користувача) через інтуїтивний дизайн та адаптивність;
- P (платформи): iOS та Android;
- NI (мережний інтерфейс) для зв'язку з сервером;
- A (архітектура) через використання MVVM.

3.4 Вебверсія застосунку

Вебверсія застосунку для обліку фінансів розробляється з використанням SwiftWasm, який компілює Swift-код у WebAssembly, дозволяючи виконувати його в браузері, та Tokamak – бібліотеки для створення декларативного UI, подібного до SwiftUI, але адаптованого для вебсередовища. Це відповідає компоненту (PL) (мова програмування) уніфікованої моделі 2.2, оскільки Swift використовується як єдина мова для всіх платформ, включаючи веб.

Вебверсія підтримує платформу web, забезпечуючи доступ до застосунку через браузер, що особливо зручно для користувачів, які працюють із настільних комп'ютерів або не можуть встановити мобільний застосунок. Вона включає ті ж екрани, що й мобільний застосунок: екран

входу/реєстрації для автентифікації, список транзакцій для перегляду фінансових операцій, екран додавання нової транзакції (з полями для суми, категорії, типу та дати) та екран статистики, де відображаються графіки доходів та витрат за вибраний період. Однією з ключових переваг є повторне використання бізнес-логіки, розробленої для мобільних платформ (iOS та Android), зокрема моделей даних (User, Transaction), мережних запитів та логіки обробки статистики, що усуває дублювання коду та забезпечує консистентність між платформами.

Взаємодія з бекендом здійснюється через REST API за допомогою URLSession, який використовується для надсилання HTTP-запитів до сервера, наприклад, для отримання списку транзакцій (GET/transactions) або додавання нової транзакції (POST/transactions). Це відповідає компоненту NI (мережний інтерфейс) для зв’язку з сервером. Реактивне оновлення UI забезпечується Tokamak, який дозволяє автоматично оновлювати інтерфейс при зміні даних, наприклад, після завантаження нових транзакцій із сервера. Приклад коду для вебекрану списку транзакцій наведено в лістингу 3.8.

Лістинг 3.8 – Екран транзакцій для Web

```
import TokamakDOM
struct WebTransactionView: View {
    @State private var transactions: [Transaction] = []
    var body: some View {
        VStack {
            Text("Financial Transactions")
            ForEach(transactions) { transaction in
                HStack {
                    Text(transaction.category)
                    Text("\(transaction.amount)")
                }
            }
            Button("Load Transactions") {
                fetchTransactions()
            }        }       }
    func fetchTransactions() {
        guard let url = URL(string:
"https://api.example.com/transactions") else { return }
        URLSession.shared.dataTask(with: url) { data, _, _ in
            if let data = data, let transactions = try?
JSONDecoder().decode([Transaction].self, from: data) {
```

```
                self.transactions = transactions
            }
        }.resume()
    }      }
```

Цей код демонструє, як Tokamak використовується для створення декларативного UI, подібного до SwiftUI, а URLSession забезпечує взаємодію з бекендом. WebAssembly, згенерований через SwiftWasm, дозволяє виконувати цей код у браузері з високою продуктивністю. Інтерфейс користувача (UI) реалізується через Tokamak, забезпечуючи зручний та сучасний вигляд веберсії. Досвід користувача (UX) досягається завдяки реактивному дизайну, який дозволяє інтерфейсу миттєво реагувати на дії користувача, наприклад, оновлення списку транзакцій після натискання кнопки, а також адаптивності до різних розмірів екранів у браузері. Таким чином, веберсія охоплює кілька компонентів моделі:

- P (платформа): web;
- UI (інтерфейс користувача) через Tokamak;
- UX (досвід користувача) через реактивний дизайн;
- NI (мережний інтерфейс) для зв’язку з сервером.

Уніфікований підхід із використанням Swift та повторне використання бізнес-логіки підкреслюють переваги Full Swift Stack для кросплатформної розробки.

3.5 Розгортання та безперервна інтеграція (CI/CD)

Для забезпечення ефективного розгортання та підтримки застосунку для обліку фінансів застосовується автоматизація через CI/CD (Continuous Integration/Continuous Deployment) з використанням GitHub Actions, що дозволяє автоматизувати процеси тестування, збірки та розгортання для всіх компонентів системи. Це відповідає компоненту TS (набір тестів) уніфікованої моделі 2.2, оскільки автоматизоване тестування є ключовою частиною CI/CD-пайплайну.

Процес CI/CD включає кілька етапів: тестування, збірку та деплой, які охоплюють усі платформи – iOS, Android, Web та серверну частину. На етапі тестування для iOS використовується XCTest, який дозволяє перевіряти коректність роботи SwiftUI-компонентів та бізнес-логіки, наприклад, тестування ViewModel для завантаження транзакцій. Для серверної частини (Vapor) застосовуються тести, які перевіряють коректність роботи REST API, зокрема ендпоінтів для реєстрації, автентифікації та управління транзакціями.

На етапі збірки для iOS генерується бінарний файл через SwiftUI, для Android – через Skip, який компілює Swift-код у нативний Android-застосунок, а для вебверсії – через SwiftWasm, який створює WebAssembly для виконання в браузері. Етап розгортання включає розгортання серверної частини на Heroku, публікацію мобільних застосунків у App Store для iOS та Google Play для Android, а також розміщення вебверсії на статичному хостингу, наприклад, Netlify, який підтримує WebAssembly та забезпечує швидкий доступ до вебверсії. Приклад конфігурації GitHub Actions для автоматизації цього процесу наведено в лістингу 3.9.

Лістинг 3.9 – Конфігурація GitHub Actions

```
name: CI/CD Pipeline
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Test iOS
        run: xcodebuild test -scheme iOSApp
      - name: Test Backend
        run: swift test
      - name: Build Backend
        run: swift build
      - name: Deploy to Heroku
        run: heroku deploy
```

Конфігурація GitHub Actions, що наведена на лістингу 3.9 запускає пайплайн при кожному push у репозиторій, виконує тестування для iOS та

бекенду, компілює серверну частину та розгортає її на Heroku. Для хостингу серверної частини обирається між кількома варіантами: Heroku для швидкого старту, оскільки він забезпечує просте розгортання та автоматичне масштабування для прототипу; AWS для масштабування, де можна використовувати EC2 для Vapor та RDS для PostgreSQL, що підходить для production-середовища з великим навантаженням; або DigitalOcean, який пропонує баланс між ціною та продуктивністю, ідеальний для середнього навантаження. Вибір Heroku для прототипу дозволяє швидко запустити серверну частину, а в майбутньому можливий перехід на AWS для масштабування. Це відповідає компоненту SC (серверна складова), який розгортається на хостингу. Таким чином, CI/CD та розгортання охоплюють компоненти моделі:

- SC (серверна складова) через розгортання на хостингу;
- TS (набір тестів) через автоматизоване тестування в GitHub Actions, забезпечуючи стабільність і якість додатку на всіх етапах розробки.

4 ПОРІВНЯННЯ ІСНУЮЧОЇ ТА РОЗРОБЛЕНОЇ МОДЕЛЕЙ ДО РОЗРОБКИ МОБІЛЬНИХ ТА ВЕБЗАСТОСУНКІВ

Сучасна кросплатформна розробка мобільних та вебзастосунків вимагає від розробників пошуку оптимальних рішень, які б поєднували високу продуктивність, економію ресурсів та здатність швидко адаптуватися до змін у вимогах ринку. У цьому розділі представлено порівняльний аналіз двох підходів: традиційної моделі кросплатформної розробки (2.1), основні характеристики якої розглянуто в підрозділі 2.1 та нової уніфікованої моделі (2.2), детальний опис якої наведено в підрозділі 2.2. Існуюча модель відображає базовий підхід, що широко застосовується в індустрії, тоді як розроблена модель пропонує інноваційне рішення, засноване на уніфікації технологічного стеку з використанням мови Swift.

Метою цього порівняння є не лише виявлення сильних та слабких сторін обох моделей, а й обґрунтування переваг нової моделі за такими ключовими критеріями, як швидкість розробки, зменшення кількості необхідних розробників, зниження фінансових та часових витрат, спрощення процесів заміни компонентів та масштабування системи, а також підвищення загальної ефективності проєкту. Аналіз базується на теоретичних засадах, викладених у попередніх розділах і практичних результатах, отриманих під час розробки прототипу додатку для обліку фінансів (розділ 3). Порівняння дозволяє оцінити, як уніфікований підхід може змінити парадигму створення кросплатформних застосунків, забезпечивши розробникам інструменти для швидшого виведення продукту на ринок, зниження залежності від великих команд і створення більш стабільних та гнучких рішень.

4.1 Порівняння моделей

Існуюча модель, яка представлена виразом (2.1), є базовим підходом до кросплатформної розробки, який часто застосовується в популярних

фреймворках, таких як Flutter, React Native чи Xamarin. Вона складається з чотирьох ключових компонентів, кожен із яких виконує визначену функцію у структурі застосунку:

- UI (інтерфейс користувача) відповідає за візуалізацію даних та взаємодію з користувачем. У кросплатформних рішеннях UI адаптується до особливостей кожної платформи (iOS, Android, Web), але часто потребує використання різних інструментів (наприклад, Dart для Flutter або JavaScript для React Native), що ускладнює уніфікацію;

- BL (бізнес-логіка) охоплює основну функціональність застосунку, таку як обробка даних, обчислення чи правила роботи системи. У традиційному підході цей компонент реалізується окремо для кожної платформи, що призводить до дублювання коду та ускладнює його підтримку;

- DB (база даних) забезпечує зберігання даних, таких як інформація про користувачів чи транзакції. У цій моделі інтеграція бази даних із клієнтською частиною зазвичай відбувається через API, а вибір конкретної бази (наприклад, SQLite чи Firebase) залежить від платформи та потреб проєкту;

- API (інтерфейс взаємодії) слугує мостом між клієнтською та серверною частинами, найчастіше реалізується як REST API. Цей компонент забезпечує передачу даних, але його створення та підтримка вимагають окремого технологічного стеку для серверної логіки (наприклад, Node.js чи Python).

Модель (2.1) фокусується на основних аспектах розробки застосунку, що робить її простою для розуміння та реалізації в типових сценаріях. Проте вона не враховує уніфікацію технологічного стеку, що призводить до фрагментації коду, а також не включає такі важливі аспекти, як безпека, тестування чи аналітика, які доводиться додавати окремо, збільшуючи складність проєкту.

Розроблена модель (2.2) є розширеним та уніфікованим підходом, який усуває недоліки традиційної моделі за рахунок інтеграції додаткових

компонентів та використання єдиного технологічного стеку на базі мови Swift (Full Swift Stack). Її складові охоплюють ширший спектр потреб сучасної розробки:

- P (платформа) визначає цільові платформи (iOS, Android, Web), забезпечуючи їхню підтримку через адаптивні інструменти, такі як SwiftUI, Skip та Tokamak;
- A (архітектура) задає чітку структуру застосунку, використовуючи MVVM для клієнтської частини та MVC для серверної, що сприяє модульності та легкості масштабування;
- PL (мова програмування) уніфікує розробку за допомогою Swift, усуваючи потребу в різних мовах для клієнта (SwiftUI, Skip) та серверу (Vapor);
- UI (інтерфейс користувача) реалізується через декларативні фреймворки (SwiftUI для iOS, Skip для Android, Tokamak для Web), що забезпечують швидке створення адаптивних інтерфейсів;
- UX (досвід користувача) акцентує увагу на зручності та інтуїтивності взаємодії, враховуючи платформні стандарти (наприклад, Human Interface Guidelines для iOS чи Material Design для Android);
- SC (серверна складова) включає серверну логіку, реалізовану через Vapor, що дозволяє обробляти запити та інтегруватися з базою даних у межах єдиного стеку;
- DB (база даних) підтримує зберігання даних (наприклад, через PostgreSQL із Fluent ORM), інтегруючись із сервером та клієнтом через Swift;
- NI (мережний інтерфейс) забезпечує стандартизовану взаємодію через REST API, реалізовану на Vapor, із підтримкою асинхронних запитів;
- S (безпека) включає вбудовані механізми захисту, такі як JWT для автентифікації та HTTPS для шифрування, що підвищують надійність системи;
- TS (тестування) передбачає автоматизоване тестування через XCTest для клієнта та тести Vapor для сервера, забезпечуючи стабільність продукту;

- AM (аналітика та моніторинг) додає інструменти для збору даних про поведінку користувачів та продуктивність системи, що сприяє її оптимізації.

Розроблена модель базується на концепції Full Swift Stack, що забезпечує єдність технологій для всіх частин застосунку: від клієнтських інтерфейсів до серверної логіки. Вона усуває фрагментацію, характерну для традиційних підходів і пропонує комплексне рішення, яке охоплює не лише базові функції, а й додаткові аспекти, необхідні для створення конкурентоспроможних продуктів.

Оскільки обидві моделі вже були детально розглянуті в підрозділах 2.1 та 2.2, а їхня практична реалізація проілюстрована в розділі 3 на прикладі прототипу застосунку для обліку фінансів, подальший аналіз у цьому розділі зосереджується на порівнянні їхньої ефективності за ключовими критеріями, такими як швидкість розробки, економія ресурсів та гнучкість.

4.2 Порівняння за швидкістю розробки

Швидкість розробки є одним із ключових факторів, що визначають ефективність моделі кросплатформної розробки, особливо в умовах стислих термінів виведення продукту на ринок. Існуюча модель (2.1) через відсутність уніфікації технологічного стеку суттєво уповільнює процес створення застосунку. Для кожної платформи (iOS, Android, Web) та серверної частини необхідно створювати окремі кодові бази, що призводить до значних витрат часу. Наприклад, бізнес-логіка (BL), яка включає обробку транзакцій чи автентифікацію користувачів, дублюється різними мовами програмування: Swift чи Objective-C для iOS, Kotlin чи Java для Android, JavaScript для Web і, наприклад, Python чи Node.js для серверної частини. Це дублювання не лише збільшує обсяг роботи, але й ускладнює інтеграцію з API, оскільки кожен клієнтський застосунок потребує окремого налаштування для взаємодії з сервером, а різниця в технологічних стеках додає час на узгодження форматів даних і обробку помилок.

Натомість розроблена модель (2.2) завдяки компоненту (PL) (єдина мова програмування – Swift) дозволяє значно прискорити розробку за рахунок повторного використання коду між усіма платформами та сервером. У прототипі застосунку для обліку фінансів, описаному в розділі 3, моделі даних, такі як User та Transaction, а також логіка їхньої обробки (наприклад, обчислення статистики чи валідація введених даних), пишуться один раз на Swift та застосовуються без змін для iOS (SwiftUI), Android (Skip), Web (SwiftWasm + Tokamak) та серверної частини (Vapor). Це усуває необхідність переписувати код для кожної платформи, а інтеграція з REST API, реалізованим на Vapor, спрощується завдяки однаковому синтаксису та інструментам (наприклад, URLSession для клієнтів і SwiftNIO для сервера). За оцінками, заснованими на реалізації прототипу, такий підхід скорочує час розробки на 30-40% порівняно з традиційною моделлю, що досягається за рахунок уникнення дублювання логіки, спрощення налагодження та швидшого тестування.

Для наглядності порівняння швидкості розробки на рисунку 4.1 наведено інформацію, яка ілюструє приблизні витрати часу на ключові етапи створення середнього кросплатформного застосунку (наприклад, застосунку для обліку фінансів) для обох моделей.

Етап розробки	Існуюча модель (години)	Розроблена модель (години)	Різниця (%)
Розробка UI (iOS)	80	60	-25%
Розробка UI (Android)	80	50	-37.5%
Розробка UI (Web)	70	50	-28.6%
Реалізація бізнес-логіки (BL)	120 (по 40 на платформу)	50 (один раз для всіх)	-58.3%
Налаштування бази даних (DB)	30	25	-16.7%
Розробка та інтеграція API	60	40	-33.3%
Налагодження та інтеграція	50	30	-40%
Загальний час	**490**	**305**	**-37.8%**

Рисунок 4.1 – Порівняння витрат часу на розробку

Часові оцінки базуються на припущенні, що проєкт включає розробку клієнтських інтерфейсів для iOS, Android та Web, а також серверної частини з базою даних та API.

Для існуючої моделі (2.1) припускається використання Flutter (Dart) для клієнтів та Node.js для сервера. Для розробленої моделі (2.2) використовується Full Swift Stack (SwiftUI, Skip, Tokamak, Vapor). Час для UI у новій моделі (2.2) менший завдяки декларативному підходу SwiftUI та повторному використанню компонентів через Tokamak і Skip. Бізнес-логіка в новій моделі (2.2) пишеться один раз, тоді як у старій (2.1) – окремо для кожної платформи.

Як видно із рисунку 4.1, розроблена модель (2.2) значно скорочує час на всіх етапах, особливо на реалізації бізнес-логіки та інтеграції, завдяки уніфікації. Наприклад, у прототипі (розділ 3) створення ендпоінтів REST API на Vapor і їхнє підключення до клієнтів через URLSession зайняло 40 годин замість 60, оскільки не було потреби адаптувати код під різні мови. Аналогічно, повторне використання моделі Transaction дозволило скоротити час на бізнес-логіку з 120 до 50 годин.

Таким чином, розроблена модель (2.2) завдяки уніфікації через мову Swift забезпечує суттєве прискорення розробки, що робить її кращим вибором для проєктів із обмеженими термінами та ресурсами.

4.3 Порівняння за кількістю розробників

Кількість розробників, необхідних для реалізації проєкту, є важливим показником ефективності моделі, оскільки вона впливає не лише на швидкість розробки, а й на витрати, координацію та загальну продуктивність команди. Існуюча модель (2.1) через відсутність уніфікації технологічного стеку вимагає залучення команди з різноманітними компетенціями, що суттєво ускладнює процес. Для середнього кросплатформного проєкту, такого як застосунок для обліку фінансів, зазвичай потрібні окремі

спеціалісти для кожної платформи та серверної частини:

- розробник iOS, який володіє Swift або Objective-C, для створення інтерфейсу та логіки на платформі Apple;

- розробник Android, знайомий із Kotlin або Java, для адаптації застосунку під екосистему Google;

- веброзробник із навичками JavaScript (наприклад, React або Vue.js) для створення вебверсії;

- бекенд-розробник, що працює з Python (Django/Flask), Node.js або іншою мовою для реалізації серверної логіки, API та інтеграції з БД.

Такий підхід призводить до формування команди з 5-7 осіб, залежно від складності проєкту. Наприклад, для реалізації прототипу застосунку за цією моделлю можуть знадобитися: 1 iOS-розробник, 1 Android-розробник, 1 веброзробник, 1-2 бекенд-розробники та 1 спеціаліст із баз даних чи інтеграції. Це ускладнює координацію, оскільки кожен член команди працює в своєму технологічному стеку, що вимагає додаткових зусиль на узгодження інтерфейсів, форматів даних та розв’язання конфліктів між платформами. Крім того, витрати на управління зростають через необхідність залучення менеджера проєкту для синхронізації роботи команди.

Розроблена модель (2.2) завдяки уніфікації через компонент PL (мова програмування – Swift) дозволяє значно зменшити розмір команди до 2-3 розробників, які володіють однією мовою. У цій моделі один спеціаліст може охоплювати кілька компонентів одночасно: UI (SwiftUI для iOS, Skip для Android, Tokamak для Web), SC (серверна логіка на Vapor) та DB (інтеграція з PostgreSQL через Fluent). Наприклад, у прототипі застосунку для обліку фінансів (розділ 3) один розробник здатен реалізувати інтерфейс для iOS за допомогою SwiftUI, адаптувати його для Android через Skip, створити вебверсію через Tokamak та налаштувати сервер із REST API на Vapor, використовуючи той самий синтаксис та підходи. Другий розробник може зосередитися на тестуванні (TS) та безпеці (S), таких як налаштування JWT та HTTPS, а також аналітиці (AM).

Ця уніфікація спрощує комунікацію, оскільки команда працює в єдиному технологічному контексті, а необхідність у додаткових спеціалістах відпадає. Наприклад, замість трьох фронтенд-розробників для різних платформ достатньо одного, який адаптує код між SwiftUI, Skip та Tokamak, а бекенд-логіка на Vapor не потребує окремого спеціаліста, якщо фронтенд-розробник знайомий із Swift. Це також знижує навантаження на менеджмент, оскільки координація між 2-3 особами значно простіша, ніж між 5-7, і часто може бути виконана без окремого менеджера проєкту.

Для наочності порівняння кількості розробників (рисунок 4.2) наведено типовий склад команди для середнього проєкту (наприклад, застосунку для обліку фінансів) за обома моделями. Для існуючої моделі (2.1) припускається використання Flutter (Dart) для клієнтів та Node.js для бекенду. Для розробленої моделі (2.2) один універсальний Swift-розробник охоплює UI, SC та частково DB, а другий може відповідати за S, TS та AM. Тестувальник / DevOps є опціональним у обох моделях, але в новій моделі (2.2) його роль може виконувати основний розробник завдяки вбудованим інструментам тестування (XCTest).

Роль у команді	**Існуюча модель (к-сть осіб)**	**Розроблена модель (к-сть осіб)**	**Різниця**
iOS-розробник (Swift)	1	1 (універсальний Swift-розробник)	-
Android-розробник (Kotlin/Java)	1	-	-1
Веброзробник (JavaScript)	1	-	-1
Бекенд-розробник (Python/Node.js)	1-2	-	-1/-2
Спеціаліст із баз даних	1 (опціонально)	-	-1
Тестувальник/DevOps	1 (опціонально)	1 (опціонально)	-
Загальна кількість	**5-7**	**2-3**	**-3/-4**

Рисунок 4.2 – Порівняння кількості розробників для існуючої та розробленої моделей

З рисунку 4.2 видно, що розроблена модель (2.2) скорочує команду щонайменше на 3-4 особи за рахунок уніфікації через Swift. У прототипі (розділ 3) два розробники могли б повністю реалізувати проєкт: один для створення інтерфейсів та серверної логіки, другий для тестування та безпеки, тоді як у традиційній моделі знадобилося б щонайменше п’ять спеціалістів для покриття всіх платформ та бекенду. Це не лише знижує витрати на зарплати, а й спрощує комунікацію, зменшуючи ризик помилок через непорозуміння між членами команди.

Таким чином, нова модель (2.2) завдяки уніфікації технологічного стеку дозволяє суттєво зменшити кількість розробників, підвищуючи ефективність роботи команди та знижуючи управлінське навантаження.

4.4 Порівняння за вартістю розробки

Вартість розробки є критичним фактором при виборі моделі для кросплатформного проєкту, оскільки вона безпосередньо впливає на бюджет та рентабельність продукту. Існуюча модель (2.1) через свою фрагментовану природу виявляється значно дорожчою, що зумовлено потребою в кількох технологічних стеках та більшій кількості розробників. Для реалізації середнього проєкту, такого як застосунок для обліку фінансів, залучаються спеціалісти з різними навичками, кожен із яких працює зі своїм набором інструментів: Swift для iOS, Kotlin або Java для Android, JavaScript для Web і, наприклад, Python або Node.js для бекенду. Це призводить до збільшення витрат через тривалий час інтеграції, дублювання коду та необхідність узгодження між платформами.

Припустимо, що погодинні ставки розробників становлять: $50/год для Swift (iOS), $45/год для Kotlin (Android), $40/год для JavaScript (Web) та $50/год для Python (бекенд). З урахуванням типового часу розробки (рисунок 4.1), загальні витрати на проєкт за існуючою моделлю можуть коливатися від $50,000 до $70,000. Наприклад, розробка UI для iOS (80 годин × $50 = $4000), Android (80 годин × $45 = $3600), Web (70 годин × $40 =

$2800), бізнес-логіки (120 годин сумарно для трьох платформ, розподілених між розробниками), API та інтеграція (60 годин × $50 = $3000 плюс додаткові години на узгодження) швидко накопичують значну суму. До цього додаються витрати на тестування та менеджмент, які зростають через складність координації команди з 5-7 осіб.

Розроблена модель (2.2) завдяки уніфікації через єдиний технологічний стек Swift суттєво знижує витрати. Один розробник із погодинною ставкою $50/год може виконувати завдання для всіх компонентів: від створення інтерфейсів (UI) за допомогою SwiftUI, Skip та Tokamak до реалізації серверної логіки (SC) на Vapor та налаштування бази даних (DB) через Fluent. Повторне використання коду, наприклад, моделей User та Transaction у прототипі (розділ 3), скорочує час на розробку бізнес-логіки з 120 годин до 50, а уніфікований підхід до API (40 годин замість 60) зменшує витрати на інтеграцію. Загальний час розробки для аналогічного проєкту становить 305 годин (рисунок 4.1), що при ставці $50/год дає діапазон витрат від $30,000 до $40,000, залежно від додаткових завдань, таких як тестування чи аналітика.

Для наочності порівняння нижче наведено рисунок 4.3, який ілюструє витрати на розробку середнього проєкту за обома моделями з урахуванням погодинних ставок і часу, витраченого на ключові етапи.

Для існуючої моделі (2.1) використано середні ставки для різних спеціалістів ($40-$50/год), а загальна сума включає додаткові витрати на команду з 5-7 осіб і менеджмент. Для розробленої моделі (2.2) припускається єдина ставка $50/год для універсального Swift-розробника та команда з 2-3 осіб. Часові оцінки взяті з рисунку 4.1.

З рисунку 4.3 видно, що основна економія в розробленій моделі досягається за рахунок скорочення часу на бізнес-логіку (з $5400 до $2500) завдяки повторному використанню коду та зменшення витрат на інтеграцію (з $3000 до $2000) через уніфікований стек. У прототипі (розділ 3) створення єдиного REST API на Vapor та його підключення до клієнтів через Swift

скоротило витрати на 33% порівняно з традиційним підходом, де API писалося на Node.js і потребувало адаптації для кожного клієнта. Додаткове зменшення витрат забезпечується меншою кількістю розробників (2-3 замість 5-7), що знижує загальну суму на 40% – з $50,000-$70,000 до $30,000-$40,000. Таким чином, розроблена модель завдяки уніфікації через Swift і меншій команді дозволяє суттєво знизити вартість розробки, роблячи її економічно вигіднішою для проєктів різного масштабу.

Етап розробки	Існуюча модель (витрати, $)	Розроблена модель (витрати, $)	Різниця (%)
Розробка UI (iOS)	80 год × $50 = $4000	60 год × $50 = $3000	-25%
Розробка UI (Android)	80 год × $45 = $3600	50 год × $50 = $2500	-30.6%
Розробка UI (Web)	70 год × $40 = $2800	50 год × $50 = $2500	-10.7%
Реалізація бізнес-логіки (BL)	120 год × $45 (середнє) = $5400	50 год × $50 = $2500	-53.7%
Налаштування бази даних (DB)	30 год × $50 = $1500	25 год × $50 = $1250	-16.7%
Розробка та інтеграція API	60 год × $50 = $3000	40 год × $50 = $2000	-33.3%
Налагодження та інтеграція	50 год × $50 = $2500	30 год × $50 = $1500	-40%
Загальні витрати	**$22,800** (без урахування менеджменту)	**$15,250** (без урахування менеджменту)	**-33.1%**
З урахуванням команди (5 vs 2)	**$50,000-$70,000**	**$30,000-$40,000**	**-40%**

Рисунок 4.3 – Порівняння витрат на розробку для існуючої та розробленої моделей

4.5 Порівняння за швидкістю заміни та масштабування

Швидкість заміни компонентів та масштабування системи є важливими характеристиками моделі розробки, оскільки вони визначають, наскільки легко проєкт може адаптуватися до нових вимог, таких як додавання

платформ, оновлення функціональності чи заміна технологій. Існуюча модель (2.1) через різноманітність мов програмування та фреймворків створює значні труднощі в цих аспектах. Наприклад, оновлення API у традиційному підході, де серверна частина написана на Python (Django) або Node.js, а клієнтські застосунки – на Swift (iOS), Kotlin (Android) та JavaScript (Web), вимагає внесення змін у код різними мовами та подальшого тестування сумісності між платформами. Аналогічно, масштабування проєкту шляхом додавання нової платформи, наприклад macOS, потребує створення нової кодової бази з нуля, що включає переписування бізнес-логіки (BL) та адаптацію UI, займаючи тижні. Ця фрагментація стеку ускладнює підтримку консистентності та збільшує ризик помилок, особливо під час інтеграції.

Розроблена модель (2.2) завдяки компонентам A (архітектура: MVVM для фронтенду, MVC для бекенду) та PL (єдина мова Swift) забезпечує значно швидше масштабування та заміну компонентів. Уніфікований підхід дозволяє повторно використовувати код між платформами, а модульна архітектура спрощує внесення змін. У прототипі застосунку для обліку фінансів (розділ 3) додавання підтримки iPadOS чи watchOS не вимагало переписування бізнес-логіки чи серверної частини – достатньо було адаптувати UI через SwiftUI, використовуючи вже наявні моделі User та Transaction, що зайняло лише кілька годин замість днів чи тижнів. Серверна логіка на Vapor та БД через Fluent залишалися незмінними, оскільки не залежать від платформи клієнта. Заміна компонентів також прискорюється: наприклад, перехід із SQLite на PostgreSQL у прототипі потребував лише зміни конфігурації Fluent і кількох рядків коду, що було виконано за 1-2 дні, тоді як у традиційній моделі це могло б вимагати тижня через необхідність адаптації API та клієнтських частин.

Модульність нової моделі, підкріплена чіткою архітектурою (MVVM/MVC), дозволяє ізолювати зміни в одному компоненті без впливу на інші. Наприклад, оновлення API (компонент NI) у прототипі – додавання

нового ендпоінту для аналітики – вимагало лише внесення змін у Vapor та автоматично відобразилося на клієнтах завдяки однаковому стеку Swift, скоротивши час із кількох днів до кількох годин. Такий підхід контрастує з існуючою моделлю, де подібне оновлення потребувало б узгодження між бекенд-розробником (Python/Node.js) та фронтенд-командою (Swift/Kotlin/JavaScript).

Для наочності порівняння на рисунку 4.4 наведена інформація, яка ілюструє приблизний час на типові завдання заміни та масштабування для середнього проєкту в обох моделях.

Завдання	Існуюча модель (час)	Розроблена модель (час)	Різниця (%)
Додавання нової платформи (macOS)	2-3 тижні (120-180 год)	1-2 дні (8-16 год)	-90%
Оновлення API (новий ендпоінт)	3-5 днів (24-40 год)	4-6 годин	-85%
Заміна бази даних (SQLite → PostgreSQL)	5-7 днів (40-56 год)	1-2 дні (8-16 год)	-75%
Адаптація UI під нову платформу	1-2 тижні (60-120 год)	1 день (8 год)	-90%
Додавання нового модуля (аналітика)	1-2 тижні (60-120 год)	2-3 дні (16-24 год)	-80%

Рисунок 4.4 – Порівняння часу на заміну та масштабування для існуючої та розробленої моделей

Для існуючої моделі (2.1) припускається використання Flutter (Dart) для клієнтів і Node.js для бекенду, що вимагає переписування коду для кожної платформи. Для розробленої моделі (2.2) використано Full Swift Stack: SwiftUI для UI, Vapor для API, Fluent для DB. Часові оцінки базуються на досвіді реалізації прототипу (розділ 3) та типових сценаріях масштабування.

З рисунку 4.4 видно, що розроблена модель скорочує час на масштабування та заміну компонентів на 75-90% завдяки уніфікації через Swift та модульній архітектурі. Наприклад, додавання macOS у прототипі

потребувало лише адаптації SwiftUI (8 годин), тоді як у традиційній моделі це означало б створення нової кодової бази з нуля (120-180 годин). Заміна бази даних у новій моделі була спрощена завдяки Fluent, який абстрагує доступ до різних СУБД, тоді як у старій моделі це вимагало змін у бекенді та клієнтах.

Таким чином, нова модель (2.2) завдяки уніфікованому стеку та чіткій архітектурі забезпечує швидке масштабування та заміну компонентів, дозволяючи адаптувати проєкт до нових вимог за дні, а не тижні, що робить її значно гнучкішою за традиційний підхід.

4.6 Порівняння за безпекою та якістю

Безпека та якість програмного продукту є ключовими аспектами, що визначають його надійність, стабільність та здатність відповідати очікуванням користувачів. Існуюча модель (2.1) не передбачає вбудованих механізмів безпеки (S) чи тестування (TS) у своїй базовій структурі, що створює додаткові виклики для розробників. У традиційному підході ці аспекти необхідно реалізовувати вручну, причому часто різними інструментами для кожного технологічного стеку. Наприклад, для клієнтської частини на iOS (Swift) можуть використовуватися бібліотеки на кшталт Alamofire з ручним налаштуванням HTTPS, на Android (Kotlin) – OkHttp із власними конфігураціями, а для вебверсії (JavaScript) – Axios із додатковими плагінами для захисту. На серверній стороні (наприклад, Node.js чи Python) безпека, як-от автентифікація через JWT чи шифрування, також потребує окремих бібліотек та підходів, що не завжди узгоджуються між собою. Така фрагментація призводить до зниження консистентності: механізми безпеки можуть відрізнятися за якістю реалізації на різних платформах, а тестування, якщо воно проводиться, часто залежить від сторонніх фреймворків (наприклад, JUnit для Android, Jest для JavaScript), що ускладнює уніфіковану перевірку якості. Як наслідок, забезпечення

належного рівня безпеки та стабільності вимагає значних додаткових зусиль і часу, що не входять у базову модель, а також підвищує ризик помилок через неузгодженість інструментів.

Розроблена модель (2.2) має суттєву перевагу завдяки інтеграції компонентів S (безпека) та TS (тестування) у єдиний технологічний стек на базі Swift, що забезпечує централізований та консистентний підхід до цих аспектів. Компонент S включає вбудовані механізми, такі як JWT (JSON Web Tokens) для автентифікації та HTTPS для шифрування запитів, які реалізовані однаково як на сервері (Vapor), так і на клієнтських частинах (SwiftUI, Skip, Tokamak). У прототипі застосунку для обліку фінансів (розділ 3) автентифікація користувачів була налаштована централізовано: сервер на Vapor видає JWT-токени, які перевіряються клієнтами через стандартні бібліотеки Swift, такі як URLSession, без необхідності підключати сторонні інструменти. Шифрування HTTPS також було вбудовано в усі мережеві запити завдяки SwiftNIO на сервері та Transport Layer Security (TLS) на клієнтах, що гарантує захист даних без додаткових витрат часу чи ресурсів. Це контрастує з існуючою моделлю (2.1), де кожна платформа потребувала б окремого налаштування шифрування, що могло б призвести до пропуску вразливостей через людський фактор.

Компонент TS у новій моделі (2.2) забезпечує автоматизоване тестування через інтегровані інструменти, такі як XCTest для клієнтських інтерфейсів та тестовий фреймворк Vapor для серверної логіки. У прототипі (розділ 3) тести для перевірки бізнес-логіки (наприклад, коректності обчислень у моделі Transaction) та стабільності API (наприклад, обробки запитів на створення транзакцій) були написані один раз на Swift і застосовані до всіх платформ без адаптації. Це дозволило швидко виявляти помилки та забезпечити високу якість продукту, тоді як у традиційній моделі тестування для iOS, Android, Web та бекенду потребувало б різних інструментів та підходів, що знижує ефективність і підвищує ймовірність пропуску дефектів. Наприклад, у прототипі модульні тести для

автентифікації через JWT виконувалися централізовано на Vapor та покривали як сервер, так і клієнтів, скорочуючи час на верифікацію порівняно з фрагментованим тестуванням у старій моделі.

Уніфікований підхід підвищує безпеку та стабільність без додаткових витрат, оскільки розробникам не доводиться шукати, інтегрувати та підтримувати сторонні бібліотеки для кожної платформи. У прототипі централізована реалізація безпеки (JWT та HTTPS) та тестування (XCTest) дозволила уникнути типових вразливостей, таких як незахищені запити чи помилки автентифікації, які часто виникають у традиційних моделях через неузгодженість. Крім того, якість продукту зростає завдяки можливості швидкого виявлення та виправлення помилок у єдиному стеку, що робить розроблену модель більш надійною і стійкою до зовнішніх загроз.

Таким чином, нова модель (2.2) завдяки вбудованим компонентам S та TS у єдиному стеку Swift перевершує існуючу (2.1) за рівнем безпеки та якості, забезпечуючи централізований захист та стабільність із меншими зусиллями, що робить її кращим вибором для проєктів, де ці аспекти є пріоритетними.

4.7 Висновки порівняння

Проведений аналіз показує, що розроблена модель (2.2) суттєво перевершує існуючу модель (2.1) за всіма ключовими параметрами, які визначають ефективність кросплатформної розробки. Ці переваги базуються на уніфікації технологічного стеку через Swift, модульній архітектурі та інтеграції додаткових компонентів, що дозволяють оптимізувати процес створення застосунків, знизити витрати та підвищити якість кінцевого продукту. Нижче наведено основні висновки порівняння з акцентом на конкретні досягнення нової моделі (2.2). Для візуального відображення цих переваг пропонується використати стовпчикову діаграму, яка порівнює обидві моделі за п’ятьма ключовими параметрами (рисунки 4.5-4.9).

Висновок №1: швидша розробка (рисунок 4.5). Завдяки компоненту PL (єдина мова Swift) та повторному використанню коду між платформами (iOS, Android, Web) та сервером, час розробки скорочується на 30-40%. Наприклад, у прототипі застосунку для обліку фінансів (розділ 3) бізнес-логіка писалася один раз та застосовувалася всюди (120 годин проти 50), що зменшило загальний час із 490 годин до 305 годин.

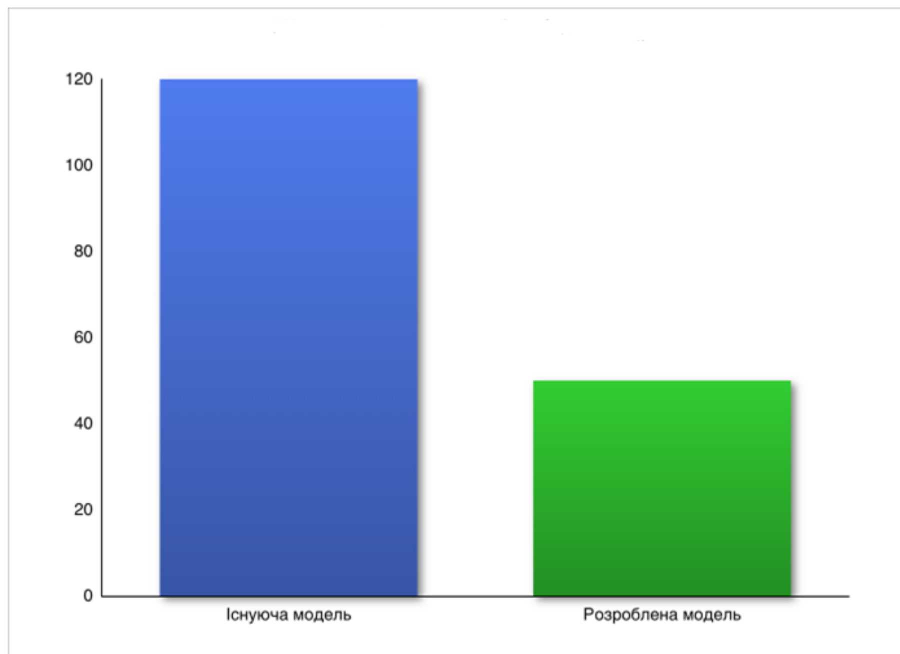

Рисунок 4.5 – Візуалізація швидкості розробки проєкту на етапі бізнес-логіки

Висновок №2: менше розробників (рисунок 4.6). Уніфікація дозволяє зменшити команду з 5-7 спеціалістів із різними компетенціями (Swift, Kotlin, JavaScript, Python) до 2-3 універсальних Swift-розробників. У прототипі два розробники могли б охопити всі аспекти – від UI до серверної логіки, тоді як традиційна модель (2.1) потребувала б щонайменше п’яти.

Висновок №3: дешевше вартість застосунку (рисунок 4.7). Завдяки меншій кількості годин та розробників витрати знижуються на 40%. Для середнього проєкту витрати за існуючою моделлю становлять $50,000-$70,000, тоді як нова модель скорочує їх до $30,000-$40,000, зокрема через економію на бізнес-логіці та інтеграції.

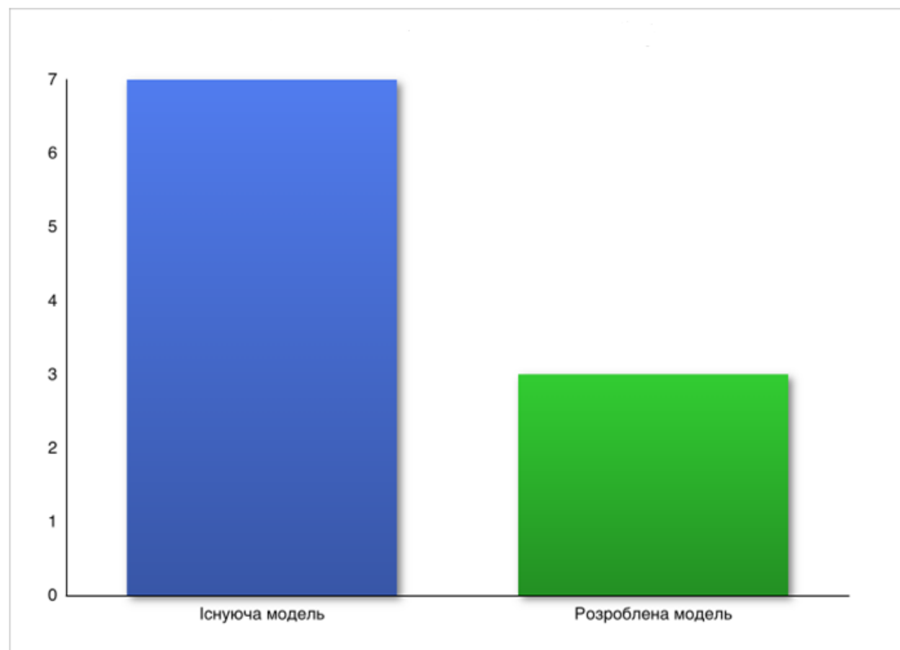

Рисунок 4.6 – Візуалізація кількості розробників на проєкті

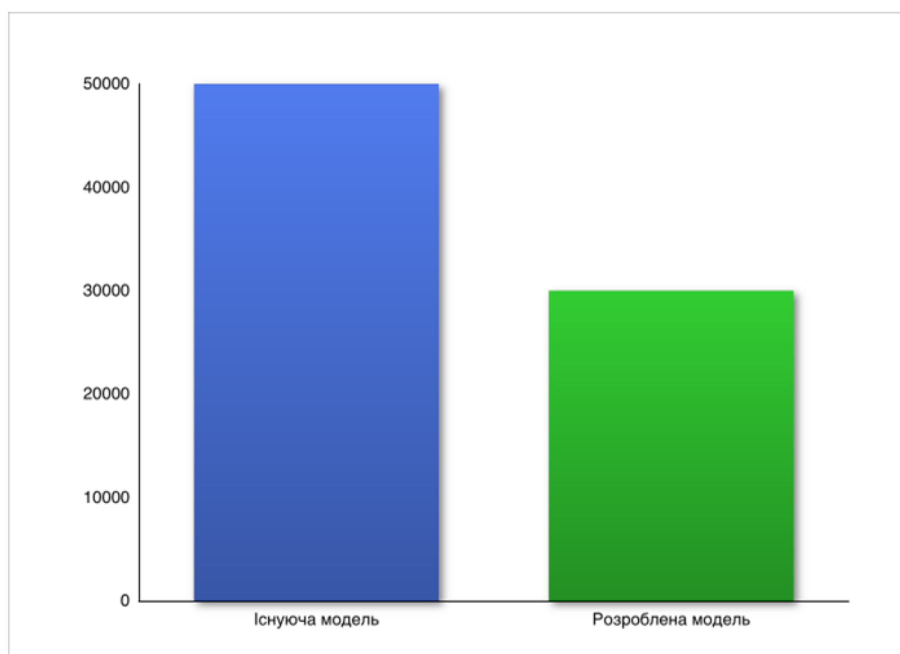

Рисунок 4.7 – Візуалізація вартості кінцевого проєкту

Висновок №4: швидша заміна та масштабування (рисунок 4.8). Модульна архітектура (A) та уніфікований стек (PL) дозволяють вносити зміни за дні, а не тижні. У прототипі додавання підтримки watchOS чи оновлення API займало 8-16 годин замість 120-180 годин у традиційній моделі (2.1).

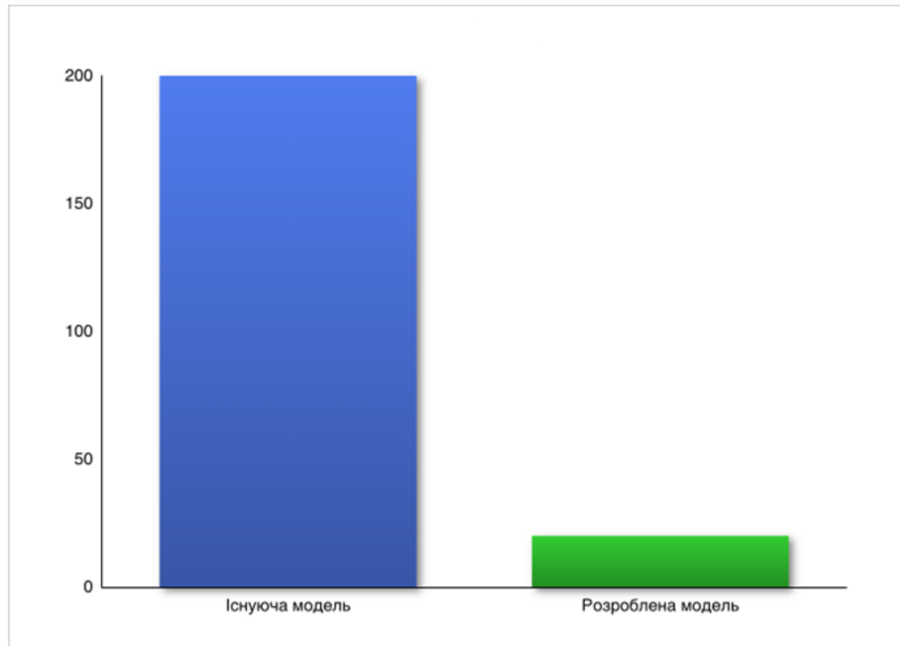

Рисунок 4.8 – Візуалізація часу масштабування проєкту

Висновок №5: вища якість застосунку (рисунок 4.9). Вбудовані компоненти S (JWT, HTTPS) та TS (XCTest, тести Vapor) забезпечують централізовану безпеку та стабільність. У прототипі автентифікація та тестування були реалізовані єдино на Swift, що усунуло неузгодженість та підвищило надійність порівняно з фрагментованим підходом старої моделі (2.1).

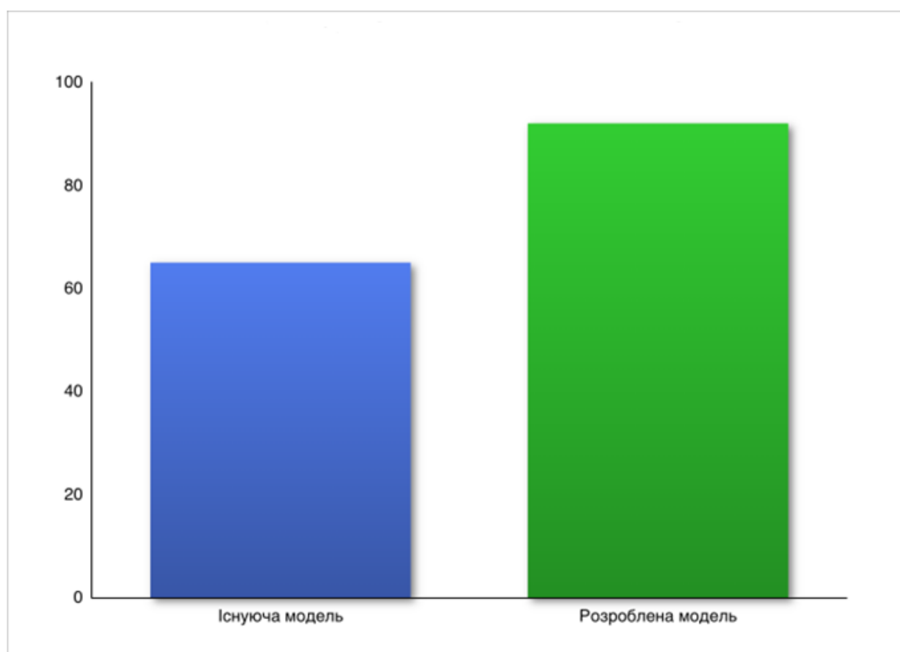

Рисунок 4.9 – Візуалізація якості проєкту

Із рисунків 4.5-4.9 можливо побачити, що розроблена модель (2.2) має перевагу за всіма параметрами: стовпчики для нової моделі будуть нижчими для часу, кількості розробників та витрат, але вищими для якості, що візуально підкреслить її ефективність. Таким чином, розроблена модель (2.2) завдяки уніфікації через Swift, модульності та вбудованим механізмам безпеки й тестування є значно ефективнішим рішенням для кросплатформної розробки. Вона дозволяє не лише скоротити ресурси й час, а й підвищити якість продукту, що робить її перспективним підходом для сучасних проєктів, де потрібна швидка адаптація та стабільність.

ВИСНОВКИ

В ході виконання кваліфікаційної роботи було проведено аналіз існуючих складових моделей, орієнтованих на розробку мобільних та вебзастосунків; виявлено переваги та недоліки існуючих моделей, зокрема їх гнучкість та масштабованість. На основі отриманих результатів було модифіковано математичну модель за рахунок додавання складових, які дозволили отримати користувачам від кінцевого продукту високої швидкодії, продуктивності, адаптивності та зручного інтерфейсу.

Уніфікована модель, яка була запропонована, використовує в якості мови програмування використовує Swift для розробки клієнтської та серверної частин. Такий підхід має численні переваги, які роблять його особливо актуальним у сучасних умовах високих вимог до продуктивності, безпеки та інтеграції застосунків.

Основними перевагами уніфікованої моделі є:

- зменшення витрат та часу на розробку: використання єдиної мовної бази дозволяє уникнути дублювання коду та інтегрувати функціональність як для клієнта, так і для серверу, що в свою чергу спрощує розробку, полегшує впровадження нових функцій та оновлень, а також знижує витрати на підтримку застосунку у подальшому;

- консистентність та узгодженість коду: спільна кодова база для мобільного інтерфейсу, бізнес-логіки та серверного шару застосунку підвищує узгодженість між його компонентами, що не лише зменшує кількість потенційних помилок, а й забезпечує єдність функціоналу для різних платформ;

- гнучкість та масштабованість: архітектура на базі використання мови Swift дозволяє легко додавати нові функції та інтегрувати зовнішні сервіси, такі як аналітика або хмарні сховища, не змінюючи основну структуру. Крім того, запропонована модель легко адаптується для підтримки різних

платформ Apple, таких як iPadOS, watchOS та macOS, що дає можливість охопити ширшу аудиторію та забезпечити мультиплатформенну підтримку;

- покращена безпека: використання сучасних методів шифрування, JWT-токенів для авторизації та двофакторної аутентифікації підвищує захист даних користувачів та запобігає несанкціонованому доступу до інформації. Слід зазначити, що мова Swift забезпечує надійну реалізацію безпеки, що є особливо важливим для застосунків, які обробляють чутливі дані;

- інтеграція аналітики та моніторингу: інструменти аналітики дозволяють відстежувати активність користувачів, оцінювати продуктивність системи та своєчасно реагувати на можливі проблеми, що дає змогу покращувати функціональність застосунку на основі реальних даних, що підвищує якість досвіду користувача;

- стабільність завдяки тестуванню: включення компонентів тестування, таких як XCTest для unit-тестування та інтеграційного тестування, підвищує стабільність роботи застосунку. Тестування дозволяє виявляти та вирішувати потенційні проблеми на ранніх етапах, зменшуючи ризик збоїв після випуску нових оновлень.

Слід зазначити, що запропонована модель може здатися надто складною для простих мобільних та вебзастосунків. Однак, навіть часткове її використання дозволяє розробникам швидко масштабувати проєкт у подальшому. Наприклад, спочатку можна створити простий застосунок, використовуючи лише необхідні компоненти, а згодом, без значних змін в основній структурі, додати нові функції.

Як ілюстрацію можливо навести приклад найпростішого мобільного застосунку для підрахунку калорій. При розробці його структури на першому кроці можливо використовувати такі компоненти, як фреймворк Flutter для клієнтської частини, а саме модулів інтерфейсу взаємодії, клієнтської частини та обробки даних, а Firebase для розгортання бази даних. Іншим прикладам можливо навести розробку трейдингової біржі, яка потребує більшої функціональності та безпеки. Для реалізації цього застосунку

розробник може використати Flutter для клієнтської частини, FastAPI для серверної та бізнес-логіки, Postgres для сховища даних, Docker для контейнеризації, Splunk для подальшого логування, Sentry для аналітики, kafka для кешування та Github Actions для автоматизації оновлення застосунку на сервері або в хмарі.

Але попри численні переваги застосування запропонованої моделі, є певні виклики, зокрема, обмежена кількість інструментів та бібліотек для серверної розробки на мові Swift у порівнянні з іншими мовами, такими як Python чи Node.js. Також розробникам може знадобитися додатковий час для вивчення фреймворків на кшталт Vapor, які менш поширені, ніж інші серверні рішення.

У цілому, уніфікована модель на базі мови Swift є перспективним рішенням для розробників, які прагнуть створити мультиплатформний застосунок з єдиною кодовою базою. Вона полегшує розробку, скорочує витрати та час, необхідні для впровадження функцій, і надає високий рівень продуктивності та безпеки. Такий підхід може стати базою для створення нових інноваційних продуктів, що відповідають сучасним вимогам ринку та дозволяють легко адаптуватися до швидкозмінного середовища технологій.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Оцевик В.А., Філімончук Т.В., Майстренко Г.В., Волк Д.М. Модель уніфікованого підходу для розроблення мобільних і вебзастосунків із використанням мови Swift // Інформаційно-керуючі системи на залізничному транспорті. Харків: УкрДУЗТ, 2025. Випуск №1 (160). С. 59-66.

2. Miola A. Flutter Complete Reference 2.0. Publisher: Independently published. 2023. 826 p.

3. Official website Flutter Docs: Flutter architectural overview. URL: https://docs.flutter.dev/resources/architectural-overview

4. Dabit N. React Native in Action, Second Edition. Publisher: Manning; First Edition. 2019. 320 p.

5. Official website React Native: Using Hermes. URL: https://reactnative.dev/docs/hermes

6. Team K., Moore K.D., Mota C., Taheri S. Kotlin Multiplatform by Tutorials (Second Edition): Build Native Apps Faster by Sharing Code Across Platforms. Publisher: Kodeco Inc. 2023. 504 p.

7. Official website Developer: The Swift Programming Language. URL: Посилання: https://developer.apple.com/swift

8. Official website SwiftWasm: Try Swift on WebAssembly now. URL: https://swiftwasm.org.

9. Official website GitHub: Tokamak: SwiftUI-Compatible Framework for WebAssembly. URL: https://github.com/TokamakUI/Tokamak

10. Official website Vapor: Swift, but on a server. URL: https://vapor.codes.

11. Official website Skip.tools: Build native apps for iPhone and Android with Skip. URL: https://skip.tools.

12. Dương Đình Bảo (James) Thăng. Ultimate SwiftUI Handbook for iOS Developers: A complete guide to native app development for iOS, macOS, watchOS, tvOS, and visionOS. Publisher: Orange Education Pvt Ltd. 2023. 278 p.

13. Ічанська Н.В., Улько С.І. Основні аспекти створення мобільних додатків та вибір інструментів їх розробки. Системи управління, навігації та зв'язку. Збірник наукових праць. Полтава: ПНТУ, 2020. Вип. 1(59). С. 74-78. doi: 10.26906/SUNZ.2020.1.074.

14. Середюк Г.В., Паламарчук Є.А. Мобільний додаток на платформі IOS з використанням архітектурного патерну MVVM. Матеріали L науково-технічної конференції підрозділів ВНТУ. Вінниця, 2021. URL: https://conferences.vntu.edu.ua/index.php/all-fksa/all-fksa-2021/paper/view/12157

15. Данілов Д.В. Дослідження основних шаблонів Model View Controller та Model View View Model для проектування Android додатків. III Всеукраїнська конференція здобувачів вищої освіти і молодих учених «Інноватика в освіті, науці та бізнесі: виклики та можливості». 2022. С. 129-132.

16. Козуб Г.О., Козуб Ю.Г., Могильний Г.А., Жуков А.В. Розробка мобільного Android-додатку з застосуванням принципів Clean Architecture. Вісник Східноукраїнського національного університету імені Володимира Даля. 2021. № 5 (269). С. 5-10. doi: 10.33216/1998-7927-2021-269-5-5-10.

17. Бешта В.С., Комаричев А.В., Філімончук Т.В., Баранєй Д.І. Модель мобільного додатку, яка орієнтована на обробку даних. Системи управління, навігації та зв'язку. Збірник наукових праць. Полтава: ПНТУ, 2024. Т. 3 (77). С. 80-83. doi: doi.org/10.26906/SUNZ.2024.3.080.

18. Філімончук Т.В., Хабазня Д.Ю. Специфіка чат-ботів як парадигма розробки моделі фреймворку. Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління: тези доповідей 11 Міжнародної науково-технічної конференції. Баку: ВА ЗС АР; Харків: НТУ «ХПІ»; Київ: ДП «ПДПРОНДІАВІАПРОМ»; Жиліна: УмЖ, 2021. Т.2. С. 40.

19. Петрунь В.М., Філімончук Т.В., Кравченко П.О. Мобільний застосунок для дослідження космосу з інтеграцією чат боту. Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління:

тези доповідей 14 Міжнародної науково-технічної конференції. Баку: НУО АР; Харків: НТУ «ХПІ»; Харків: ХНУРЕ; Харків: НАУ «ХАІ»; Жиліна: УмЖ, 2024. Т. 1: секція 2. С. 109.

20. Філімончук Т.В., Оцевик В.А. Модель мобільного застосунку iOS. Проблеми інформатизації: тези доповідей 9 Міжнародної науково-технічної конференції. Черкаси: ЧДТУ; Харків: НТУ «ХПІ»; Баку: ВАЗС АР; Бельсько-Бяла: УТіГН, 2021. Т.2. С. 98.

21. Танянський О., Руденко Д. Порівняльний аналіз популярних JavaScript-фреймворків та бібліотек для front-end розробки. Інформаційні системи та технології : матеріали статей 7-ї Міжнародної науково-технічної конференції, Коблеве-Харків, 2018. Харків: ХНУРЕ, 2018. С. 347-349.

22. Гук Н.А., Диханов С.В., Матющенко О.Д. Алгоритм побудови моделі веб-сайту. Вісник Харківського національного університету імені В. Н. Каразіна. 2020. Вип. 47. С. 25-34. doi: 10.26565/2304-6201-2020-47-03.

23. Прудченко А.Ю. Аналіз фреймворку Ember.js для створення високопродуктивних веб-додатків. Матеріали 78-ї студентської науково-технічної конференції «Тиждень студентської науки». Секція «Інформаційні та телекомунікаційні технології». НТУ «Дніпровська політехніка». 2023. С. 382-384.