

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ
СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

«На правах рукопису»
УДК _____

«До захисту допущено»
Завідувач кафедри
_____ Наталія АУШЕВА
« ____ » _____ 2025 р. □

Магістерська дисертація

на здобуття ступеня другого (магістерського) рівня вищої освіти
за освітньою програмою «Цифрові технології в енергетиці»
зі спеціальності 122 «Комп'ютерні науки»

на тему «Методи та засоби реалізації IOS-додатків управління задачами малого
або середнього бізнесу»

Виконав: студент 2 курсу, групи ТР-43мп

Недов Єгор Борисович

(прізвище, ім'я, по батькові)

_____ (підпис)

Науковий керівник к.ф.-м.н., доцент Андрій ДОНЕЦЬ

(посада, науковий ступінь, вчене звання, ім'я, ПРІЗВИЩЕ)

_____ (підпис)

Рецензент доцент каф. ТАЕ, к.т.н, доцент Олександр СІРИЙ

(посада, науковий ступінь, вчене звання, ім'я, ПРІЗВИЩЕ)

_____ (підпис)

Н.контроль асистент Володимир РУДИК

(посада, ім'я, ПРІЗВИЩЕ)

_____ (підпис)

Засвідчую, що у цій магістерській дисертації
немає запозичень з праць інших авторів без
відповідних посилань.

Студент _____
(підпис)

Київ — 2025

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти другий (магістерський)

спеціальність 122 «Комп'ютерні науки»

Освітньо-професійна програма «Цифрові технології в енергетиці»

ЗАТВЕРДЖУЮ

Завідувач кафедри ЦТЕ

Наталія АУШЕВА

«__» _____ 2025 р.

**ЗАВДАННЯ
на магістерську дисертацію студенту**

Недову Єгору Борисовичу

(прізвище, ім'я, по батькові)

1. Тема дисертації: «Методи та засоби реалізації IOS-додатків управління задачами малого або середнього бізнесу»

керівник роботи Донець Андрій Георгійович, к.ф.-м.н., доцент

затверджені наказом по університету від «03» листопада 2025 р. № 4779-с

2. Термін подання студентом дисертації: 06.12.2025

3. Об'єкт дослідження: Екосистема iOS та два основних фреймворки для створення інтерфейсів — UIKit та SwiftUI.

4. Вихідні дані: мова програмування Swift 6, UIKit, SwiftUI, середовище розробки Xcode, операційна система IOS.

5. Перелік завдань: 1) проаналізувати існуючі технології розробки інтерфейсів, виявити їхні переваги і недоліки, розглянути потенційні можливості цих інструментів;

2) дослідити та обґрунтувати вибір програмного засобу для вебзастосунку;

3) провести аналіз та дослідження найбільш прийнятних засобів побудови інтерфейсів для iOS-додатків у сфері бізнесу;

4) розробити архітектуру програмної системи на Android;

5) розробити програмне забезпечення з використанням обраних технологій,

виконати тестування розробленої системи, виправити знайдені недоліки

6. Орієнтований перелік ілюстративного матеріалу: архітектура вебзастосунку, діаграма прецедентів системи, модель бізнес-логіки, діаграма класів, діаграма послідовності процесу авторизації, блок-схема алгоритму розподілених повторень, інтерфейси користувача, таблиці стартап-проєкту.

7. Орієнтовний перелік публікацій: 1. Інтеграція штучного інтелекту в DevOps-практики для автоматизації тестування програмного забезпечення аналітичних досліджень в бізнес-діяльності в енергетиці. Збірник наукових тез з матеріалами III Міжнародної науково-практичної конференції «Сучасні аспекти інженерії програмного забезпечення» MoSE-2025, КІП, 2025. С.80-82

8. Дата видачі завдання «16» вересня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів підготовки магістерської дисертації	Термін виконання	Примітка
1	Аналіз методів розробки програмного забезпечення за літературними джерелами	18.09.2025	виконано
2	Вибір та налаштування засобів розробки програмного забезпечення	21.09.2025	виконано
3	Розроблення структури бази даних та архітектури системи	26.09.2025	виконано
4	Розроблення програмного забезпечення	13.10.2025	виконано
5	Тестування розробленої системи, виправлення недоліків	19.10.2025	виконано
6	Захист створеного програмного забезпечення	20.10.2025	виконано
7	Оформлення магістерської дисертації	25.11.2025	виконано
8	Передзахист	27.11.25	виконано
9	Подання матеріалів магістерської дисертації на кафедру	04.12.25	виконано
10	Захист	11.12.25	виконано

Студент

(підпис)

Єгор НЄДОВ

(ім'я, ПРИЗВИЩЕ)

Керівник роботи

(підпис)

Андрій ДОНЕЦЬ

(ім'я, ПРИЗВИЩЕ)

РЕФЕРАТ

Актуальність. Актуальність дослідження полягає в вивченні проблематики перенесення застосунків між мобільними платформами, яка виникає через суттєві відмінності вивчення технічних, архітектурних та системних відмінностей між екосистемами Apple та Google, а також специфіки інтерфейсних технологій UIKit і SwiftUI дозволить визначити перспективи їх застосування. Саме тому процес перенесення застосунку не може бути прямолінійним і потребує окремого вивчення теоретичних основ.

Метою магістерської дисертації є дослідження особливостей та порівняльних характеристик UIKit і SwiftUI шляхом розробки повноцінного мобільного застосунку, який реалізує багатокomпонентний інтерфейс окремо в обох технологіях. **Завданням дослідження** є:

- аналіз існуючих програмних рішень iOS-додатків;
- розроблення структури бази даних та архітектури системи;
- програмна реалізація вебзастосунку з використанням обраних технологій;
- тестування розробленої системи.

Об'єктом дослідження є екосистема iOS та два основних фреймворки для створення інтерфейсів — UIKit та SwiftUI.

Предметом дослідження є порівняння UIKit і SwiftUI, що дозволяє виявити їхній вплив на продуктивність, масштабованість та підтримуваність застосунку.

Методи дослідження: методи ідентифікації змінних параметрів, апаратні та програмні засоби вимірювання, OSSignpost-маркування для точного фіксування часових інтервалів, інструменти Time Profiler, Allocations, Core Animation, System Trace, XCTest Metrics та MetricKit.

Вихідні дані: мова програмування Swift 6, UIKit, SwiftUI, середовище розробки Xcode, операційна система IOS.

Практичне значення отриманих результатів полягає у можливості створення інтерфейсу на основі єдиного застосунку, розробленого у двох технологічних реалізаціях. Дослідження включає аналіз можливості подальшого

перенесення реалізованого функціоналу на Android з урахуванням відмінностей екосистем. Такий підхід забезпечує комплексність і дозволяє сформулювати рекомендації щодо вибору технології в залежності від характеру майбутнього проєкту, його складності та потреб у кросплатформенності.

Апробація результатів дисертації. Основні положення даної роботи доповідались та обговорювались на 3-й міжнародно-науковій конференції «Сучасні аспекти інженерії програмного забезпечення» (НТУУ «КПІ ім.Ігоря Сікорського, 12-13 листопада 2025 р.).

Структура та обсяг магістерської дисертації: вступ, 5 розділів, висновки, список використаних джерел і додаток. Робота містить 11 рисунків, 18 таблиць, 46 посилань у списку джерел та 1 додаток. Основний вміст подано на 87 аркушах, а загальний обсяг складає 97 сторінок.

Ключові слова: фреймворки для створення інтерфейсів SwiftUI та UIKit, мова програмування Swift 6, OSSignpost-розмітка, рендерингу екрана.

ABSTRACT

Relevance of the topic. The relevance of the research lies in studying the issues of porting applications between mobile platforms, which arise due to significant technical, architectural, and system differences between the Apple and Google ecosystems, as well as the specifics of UIKit and SwiftUI interface technologies, which will allow us to determine the prospects for their application. That is why the application porting process cannot be straightforward and requires a separate study of the theoretical foundations.

The purpose of the master's thesis is to study the features and comparative characteristics of UIKit and SwiftUI by developing a full-fledged mobile application that implements a multi-component interface separately in both technologies.

The **research objectives** are:

- analysis of existing software solutions for iOS applications;
- development of the database structure and system architecture;
- software implementation of a web application using selected technologies;
- testing of the developed system.

The object of the research there is an iOS ecosystem and two main frameworks for creating interfaces — UIKit and SwiftUI.

The subject of the research is a comparison of UIKit and SwiftUI, which allows you to identify their impact on the performance, scalability, and maintainability of the application.

Research methods: methods for identifying variable parameters, hardware and software measurement tools, OSSignpost marking for precise recording of time intervals, Time Profiler, Allocations, Core Animation, System Trace, XCTest Metrics and MetricKit tools.

Input data: Swift 6 programming language, UIKit, SwiftUI, Xcode development environment, IOS operating system.

Practical significance of the obtained results consists in the possibility of creating an interface based on a single application developed in two technological implementations. The study includes an analysis of the possibility of further porting the

implemented functionality to Android, taking into account the differences in ecosystems. This approach provides comprehensiveness and allows you to formulate recommendations for choosing a technology depending on the nature of the future project, its complexity, and cross-platform needs.

Approbation of the results of the dissertation. The main provisions of this work were reported and discussed at the 3rd international scientific conference «Modern Aspects of Software Engineering».

Structure and volume of the thesis are as follows: an introduction, five chapters, conclusions, a list of references, and appendice. The thesis contains 11 pictures, 18 tables, 46 references and 1 appendice. The main text occupies 87 pages, and the total volume is 97 pages.

Keywords: frameworks for creating SwiftUI and UIKit interfaces, Swift 6 programming language, OSSignpost markup, screen rendering.

ЗМІСТ

ВСТУП.....	9
1 ПОСТАНОВКА ЗАДАЧІ ТА АНАЛІЗ ТЕХНОЛОГІЙ РОЗРОБКИ МОБІЛЬНИХ ЗАСТОСУНКІВ	11
1.1 Постановка задачі та обґрунтування вибору платформи	12
1.1.1 Актуальність мобільної розробки	13
1.1.2 Обґрунтування вибору цільової платформи розробки	13
1.1.3 Проблематика вибору інтерфейсних технологій.....	14
1.1.4 Наукова мета, завдання та новизна дослідження	15
1.2 Аналіз технологій створення інтерфейсу в екосистемі iOS.....	16
1.2.1 UIKit: імперативна модель, історія розвитку та сучасний стан.....	17
1.2.2 SwiftUI: декларативна парадигма, принципи роботи та обмеження...	17
1.2.3 Порівняння імперативного та декларативного підходів.....	18
1.2.4 Сценарії використання UIKit та SwiftUI в індустрії	19
1.3 Архітектурні підходи у розробці мобільних застосунків iOS	19
1.3.1 Традиційні архітектури MVC та MVP	20
1.3.2 Сучасні підходи MVVM та MVVM-C	21
1.3.3 Архітектурні особливості SwiftUI та роль Combine	22
1.3.4 Порівняння архітектурних підходів та критерії вибору	23
1.4 Огляд процесу розгортання та публікації мобільних застосунків для iOS .	23
1.4.1 Механізм підписування: сертифікати, профілі, Bundle ID	24
1.4.2 Попереднє тестування за допомогою TestFlight.....	24
1.4.3 Публікація застосунку в App Store: процес модерації.....	25
1.4.4 Автоматизація розгортання CI/CD для iOS	25
1.5 Теоретичні основи перенесення мобільних застосунків на Android.....	26
1.5.1 Технічні відмінності платформ iOS та Android.....	27
1.5.2 Підходи до часткового перенесення: Kotlin Multiplatform Mobile	27
1.5.3 Підходи до повного крос-платформенного UI: Flutter, React Native...	28
1.5.4 Неможливість прямого запуску iOS-коду на Android: технічні та юридичні аспекти.....	28
Висновки до розділу 1	29
2 ПРОЄКТУВАННЯ МОБІЛЬНОГО ЗАСТОСУНКУ	31
2.1 Постановка задачі та вимоги до застосунку	31
2.1.1 Функціональні та нефункціональні вимоги до застосунку	31
2.1.2 Середовище розробки Xcode та процес створення проєкту.....	33
2.1.3 Налаштування структури проєкту та організація коду.....	37
2.2 Вибір архітектури та підходів до проєктування.....	38
2.3 Структура програмного проєкту	40
2.4 Діаграмне моделювання системи.....	42
2.5 Підготовка застосунку до порівняльного експерименту.....	44
Висновки до розділу 2.....	46
3 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТЕХНОЛОГІЙ ПОБУДОВИ ІНТЕРФЕЙСІВ.....	47

3.1 Мета та методика експериментального дослідження.....	48
3.2 Експериментальні сценарії.....	50
3.3 Параметри та інструменти вимірювання	52
3.4 Результати експериментальних вимірювань	54
3.5 Аналіз продуктивності.....	58
3.6 Аналіз складності розробки.....	61
3.7 Підсумкові висновки експериментального порівняння	63
Висновки до розділу 3.....	65
4 РЕАЛІЗАЦІЯ ЗАСТОСУНКУ НА ANDROID.....	67
4.1 Аналіз архітектури застосунку з точки зору портованості	67
4.1.1 Незалежність бізнес-логіки від UI-фреймворку	68
4.1.2 Чітке розділення відповідальностей	68
4.1.3 Переносимість моделей даних.....	68
4.1.4 Залежність UI від платформних фреймворків	68
4.1.5 Оцінка портованості архітектури.....	69
4.2 Компоненти застосунку, що можуть бути повторно використані	70
4.2.1 Моделі даних	70
4.2.2 Бізнес-логіка та алгоритми	70
4.2.3 Архітектурні принципи	71
4.2.4 Навігаційні сценарії.....	72
4.2.5 Концепції управління станом	72
4.2.6 Компоненти, що не можуть бути використані повторно	72
4.2.7 Узагальнення результатів аналізу	72
Висновки до розділу 4.....	73
5 РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ МОБІЛЬНОГО ЗАСТОСУНКУ	75
5.1 Опис ідеї стартап-проєкту	75
5.2 Технологічний аудит ідеї стартап-проєкту.....	77
5.3 Аналіз ринкових можливостей запуску стартап-проєкту	79
5.4 Розроблення ринкової стратегії стартап-проєкту	82
5.5 Розроблення маркетингового забезпечення стартап-проєкту	83
Висновки до розділу 5.....	84
ВИСНОВКИ.....	86
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	88
ДОДАТОК А	92

ВСТУП

У сучасному світі мобільні технології відіграють ключову роль у розвитку бізнесу, освіти, комунікацій та цифрової інфраструктури. Вибір інструментів та технологій розробки при цьому набуває стратегічного значення, оскільки він визначає архітектуру майбутнього застосунку, його масштабованість, підтримуваність і можливість подальшого розширення. Екосистема iOS пропонує два основних фреймворки для створення інтерфейсів — UIKit та SwiftUI. UIKit є класичною імперативною технологією, яка використовується вже понад десять років та забезпечує розробнику деталізований контроль над елементами інтерфейсу. SwiftUI, навпаки, є сучасним декларативним підходом, що дозволяє значно прискорити процес розробки, зменшити кількість коду та підвищити структурованість інтерфейсу. Обидві технології активно застосовуються на ринку, однак питання їхньої порівняльної продуктивності, гнучкості, простоти підтримки та придатності для масштабних проєктів залишається актуальним і потребує системного аналізу.

Актуальність дослідження дозволить чітко окреслити роль iOS у глобальному та регіональному ринку, специфіку інтерфейсних технологій UIKit і SwiftUI, а також визначити перспективи їх застосування. Проблематика перенесення застосунків між мобільними платформами виникає через суттєві технічні, архітектурні та системні відмінності між екосистемами Apple та Google. Незважаючи на те, що iOS і Android домінують на ринку мобільних технологій, їхні базові принципи роботи, мови програмування, моделі рендерингу інтерфейсу та життєвого циклу застосунку є принципово різними. Саме тому процес перенесення застосунку не може бути прямолінійним і потребує окремого вивчення теоретичних основ.

Детальний огляд процесів розгортання та публікації в App Store дозволить визначити специфіку екосистеми Apple, де значна увага приділяється перевірці якості, безпеці та відповідності рекомендаціям Human Interface Guidelines. Аналіз теоретичних засад використання застосунків на Android доводить, що пряме перенесення iOS-коду є неможливим як через технічні, так і через юридичні

обмеження, а сучасні крос-платформенні технології або дублюють UI-частину, або вимагають повного перепроєктування інтерфейсу. Аналіз підходів — від KMM до Flutter та React Native — дозволив окреслити можливості та ризики, пов'язані з адаптацією застосунку до іншої операційної системи.

Метою магістерської дисертації є дослідження особливостей та порівняльних характеристик UIKit і SwiftUI шляхом розробки повноцінного мобільного застосунку, який реалізує багатокomпонентний інтерфейс окремо в обох технологіях. Завданням дослідження є аналіз існуючих програмних рішень iOS-додатків, розроблення структури бази даних та архітектури системи, програмна реалізація вебзастосунку з використанням обраних технологій, а також тестування розробленої системи. Дослідження передбачає не лише теоретичний аналіз, але й експериментальну оцінку параметрів продуктивності, включно з використанням пам'яті, навантаженням на процесор, швидкістю рендерингу та плавністю анімацій.

Практичне значення отриманих результатів полягає у можливості створення інтерфейсу на основі єдиного застосунку, розробленого у двох технологічних реалізаціях. Дослідження включає аналіз можливості подальшого перенесення реалізованого функціоналу на Android з урахуванням відмінностей екосистем. Такий підхід забезпечує комплексність і дозволяє сформулювати рекомендації щодо вибору технології в залежності від характеру майбутнього проєкту, його складності та потреб у кросплатформенності.

Магістерська дисертація складається зі вступу, п'яти розділів, висновків, списку використаних джерел та додатку. Перший розділ присвячений постановці задачі, аналізу існуючих систем. У другому розділі запропоновано проєкт мобільного застосунку. Третій розділ присвячений експериментальним дослідженням. Четвертий розділ описує користувацьку взаємодію з розробленим вебзастосунком на Android. П'ятий розділ представляє стартап-аналіз проєкту. В додатку наводиться апробація дисертаційних досліджень.

1 ПОСТАНОВКА ЗАДАЧІ ТА АНАЛІЗ ТЕХНОЛОГІЙ РОЗРОБКИ МОБІЛЬНИХ ЗАСТОСУНКІВ

У сучасному світі мобільні технології відіграють ключову роль у розвитку бізнесу, освіти, комунікацій та цифрової інфраструктури. Зростання кількості мобільних користувачів та підвищення вимог до якості програмних продуктів формують потребу у створенні застосунків, які поєднують високу продуктивність, стабільність, гнучкість і зручність користування. Вибір інструментів та технологій розробки при цьому набуває стратегічного значення, оскільки він визначає архітектуру майбутнього застосунку, його масштабованість, підтримуваність і можливість подальшого розширення.

Екосистема iOS пропонує два основних фреймворки для створення інтерфейсів — UIKit та SwiftUI. UIKit є класичною імперативною технологією, яка використовується вже понад десять років та забезпечує розробнику деталізований контроль над елементами інтерфейсу. SwiftUI, навпаки, є сучасним декларативним підходом, що дозволяє значно прискорити процес розробки, зменшити кількість коду та підвищити структурованість інтерфейсу. Обидві технології активно застосовуються на ринку, однак питання їхньої порівняльної продуктивності, гнучкості, простоти підтримки та придатності для масштабних проєктів залишається актуальним і потребує системного аналізу.

Попри глобальне домінування Android, що утримує приблизно 72-73 % світового ринку мобільних операційних систем, iOS є ключовою платформою в низці стратегічно важливих регіонів. Згідно з даними StatCounter, частка iOS у країнах Європейського Союзу становить 35-40 %, у Сполучених Штатах — понад 60 %, а в Україні — близько 21-23 %, демонструючи тенденцію до зростання [1]. Це означає, що iOS охоплює високоплатоспроможну аудиторію та має значну присутність у регіонах із найвищими темпами розвитку цифрових сервісів.

Крім того, економічні показники свідчать про істотну вагу екосистеми Apple для розробників. За даними Data.ai та Apple, сукупний дохід App Store стабільно майже вдвічі перевищує дохід Google Play, незважаючи на меншу кількість пристроїв [2]. Така різниця пояснюється високою платоспроможністю

користувачів iOS, їхньою готовністю купувати застосунки та підписки, а також стабільністю та передбачуваністю самої платформи. Саме тому багато технологічних компаній — серед яких Airbnb, Uber, Revolut, Monobank, Bolt — першочергово запускають свої мобільні продукти на iOS, а вже згодом адаптують їх під Android [3].

Водночас значна частка Android на глобальному ринку обумовлює необхідність у кросплатформеності та потенційній можливості перенесення застосунку на іншу операційну систему. Через істотні відмінності між екосистемами Apple та Google пряме перенесення інтерфейсу або архітектури є неможливим, а отже вимагає аналізу відповідних технологічних підходів. Саме тому важливим завданням цього розділу є не лише порівняння UIKit та SwiftUI, але й огляд архітектур, засобів розгортання, особливостей роботи екосистеми iOS та сучасних методів адаптації застосунків до Android.

Усе це формує теоретичний фундамент, на основі якого надалі здійснюватиметься проектування, реалізація та експериментальне дослідження мобільного застосунку.

1.1 Постановка задачі та обґрунтування вибору платформи

Розробка мобільних застосунків сьогодні потребує виваженого вибору технологій, архітектурних підходів та платформи, для якої створюватиметься програмний продукт. Цей вибір визначає подальшу якість, масштабованість і стабільність застосунку, а також впливає на ефективність його розробки й підтримки.

У межах цього дослідження важливо послідовно окреслити актуальність мобільного сегмента, роль iOS у глобальному та регіональному ринку, специфіку інтерфейсних технологій UIKit і SwiftUI, а також визначити наукові цілі та задачі роботи.

Підрозділ створює основу для подальшого аналізу та практичної частини, у якій буде реалізовано застосунок для експериментального порівняння технологій.

1.1.1 Актуальність мобільної розробки

Мобільні технології є ключовим рушієм цифрової трансформації, оскільки смартфони стали основним засобом доступу до інформації, сервісів та цифрових екосистем. Згідно зі звітом Statista, у 2024 році кількість мобільних користувачів перевищила 6,9 млрд, що фактично охоплює більшу частину населення планети. У таких умовах зростають вимоги до програмного забезпечення, його продуктивності, стабільності та зручності. Інтерфейс користувача відіграє центральну роль у сприйнятті якості застосунку, а тому вибір технології для його побудови стає критичним на ранніх етапах проєктування. Саме UI визначає швидкість взаємодії, інтуїтивність переходів, реакцію на дії користувача та рівень задоволеності від роботи із застосунком.

1.1.2 Обґрунтування вибору цільової платформи розробки

Незважаючи на домінування Android у глобальному масштабі (рис. 1.1), iOS посідає стратегічно важливе місце на ринку мобільних технологій. За даними StatCounter, платформа Apple утримує понад 60 % ринку у США, приблизно 35-40 % у країнах Європейського Союзу та близько 21-23 % в Україні, де спостерігається тенденція до зростання частки користувачів [2, 7-11].. Така ситуація пов'язана з високою платоспроможністю аудиторії Apple та стабільністю екосистеми, яка традиційно приваблює компанії, що прагнуть створювати якісні та комплексні цифрові продукти (рис. 1.2).

Android vs iOS Market Share in the Global Mobile OS Market

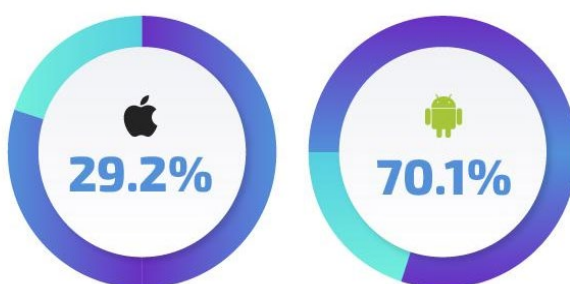


Рисунок 1.1 — Позиціонування Android на ринку iOS

Важливою є й економічна складова. За даними Data.ai та Apple, App Store стабільно генерує майже вдвічі більші доходи для розробників, ніж Google Play, попри значно меншу кількість пристроїв у світовій екосистемі [3, 12]. Це підтверджує, що користувачі iOS частіше купують цифровий контент, оплачують підписки та користуються преміальними сервісами. Саме тому такі компанії, як Uber, Airbnb, Revolut, Monobank, Bolt та інші технологічні лідери, традиційно випускають застосунки спершу на iOS, а вже потім адаптують їх під Android.

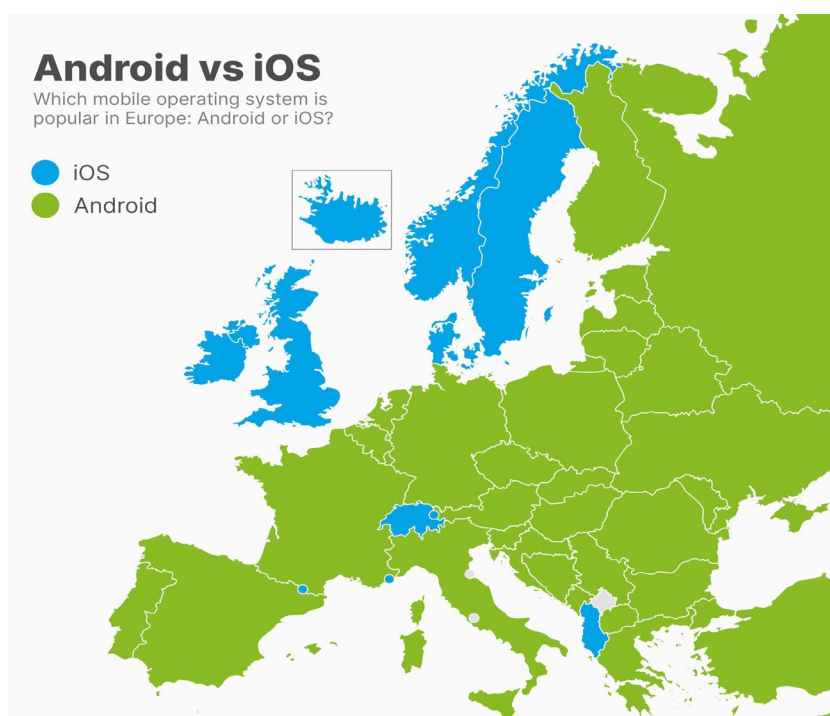


Рисунок 1.2 — Географічне позиціонування

Таким чином, вибір iOS як платформи для розробки у межах цієї роботи є логічним, технічно обґрунтованим та економічно доцільним.

1.1.3 Проблематика вибору інтерфейсних технологій

У сучасній мобільній розробці для iOS існують дві парадигми побудови інтерфейсу — UIKit та SwiftUI, кожна з яких має власні концепції, підходи до роботи зі станом та рівень зрілості. UIKit, що використовується вже понад десять років, забезпечує високий рівень контролю над компонентами інтерфейсу,

передбачувану поведінку та сумісність зі всіма версіями iOS. Цей фреймворк побудований на імперативній моделі, у межах якої розробник вручну керує відображенням UI та його оновленням. Такий підхід дає точність, але вимагає значного обсягу коду й ускладнює підтримку великих застосунків.

SwiftUI, представлений Apple у 2019 році, пропонує принципово новий, декларативний спосіб побудови інтерфейсу, у якому розробник описує кінцевий стан UI, а оновлення здійснюється автоматично. Цей підхід ґрунтується на реактивних механізмах керування станом і дозволяє суттєво зменшити складність коду. Водночас SwiftUI досі активно розвивається, не повністю охоплює функціональність UIKit, а його поведінка у складних сценаріях інколи менш передбачувана. Вибір між цими технологіями є актуальним для багатьох команд, оскільки він визначає адаптивність, масштабованість та вартість підтримки застосунку.

1.1.4 Наукова мета, завдання та новизна дослідження

Метою цієї магістерської роботи є дослідження особливостей та порівняльних характеристик UIKit і SwiftUI шляхом розробки повноцінного мобільного застосунку, який реалізує багатокomпонентний інтерфейс окремо в обох технологіях. Такий підхід дозволяє оцінити їхню продуктивність, стабільність, зручність розробки, а також поведінку інтерфейсу у реальних сценаріях взаємодії. Дослідження передбачає не лише теоретичний аналіз, але й експериментальну оцінку параметрів продуктивності, включно з використанням пам'яті, навантаженням на процесор, швидкістю рендерингу та плавністю анімацій.

Новизна роботи полягає у виконанні практичного порівняння двох парадигм створення інтерфейсу на основі єдиного застосунку, розробленого у двох технологічних реалізаціях. Окрім цього, дослідження включає аналіз можливості подальшого перенесення реалізованого функціоналу на Android з урахуванням відмінностей екосистем. Такий підхід забезпечує комплексність і дозволяє

сформувати рекомендації щодо вибору технології в залежності від характеру майбутнього проєкту, його складності та потреб у кросплатформенності.

1.2 Аналіз технологій створення інтерфейсу в екосистемі iOS

Інтерфейс користувача є фундаментальним елементом будь-якого мобільного застосунку, оскільки саме через нього формується взаємодія між користувачем і системою. В екосистемі iOS існує дві основні технології для побудови інтерфейсу — UIKit та SwiftUI (рис.1.3). Вони представляють різні парадигми програмування і різні підходи до організації відображення, управління станом та побудови архітектури.

Аналіз цих двох фреймворків є необхідним для розуміння того, як саме формується досвід користувача, наскільки ефективно здійснюється розробка, та які обмеження й можливості має кожна з технологій.

Framework	Swift	UIKit
Declarative syntax	✓	✗
Previews and live editing	✓	✗
State and data binding	✓	✗
Reactive programming	✓	✗
View controllers	✗	✓
Interface builder and storyboards	✗	✓
Delegates and data sources	✗	✓

Рисунок 1.3 — Порівняння існуючих систем побудови інтерфейсу

Порівняння UIKit і SwiftUI дозволяє виявити їхній вплив на продуктивність, масштабованість та підтримуваність застосунку, що безпосередньо стосується предмету цього дослідження.

1.2.1 UIKit: імперативна модель, історія розвитку та сучасний стан

UIKit є класичним фреймворком для побудови інтерфейсу в iOS, який існує з першої версії операційної системи та визначив стандарти мобільної взаємодії на платформі Apple. Він побудований на імперативному підході, у межах якого розробник вручну описує усі зміни інтерфейсу та контролює реакцію UI на події. Така модель забезпечує детальний контроль над кожним компонентом, дозволяє створювати складні та високопродуктивні елементи, а також гарантує передбачувану поведінку в різних сценаріях використання [9].

Історично UIKit розвивався як відповідь на потребу створення інтерфейсів, які відповідали б вимогам Human Interface Guidelines та принципам керування подіями, характерним для перших версій iOS. З роками Apple суттєво розширила його можливості: з'явилися Auto Layout, динамічні анімації, механізми керування фокусом, адаптивний дизайн для iPad і багатовіконності. Попри появу нових технологій, UIKit і сьогодні залишається основою величезної кількості комерційних застосунків, що підтверджується численними кейсами, опублікованими в архівах WWDC [10, 13-20].

Сучасний стан UIKit визначається його стабільністю, високим рівнем сумісності зі старими версіями iOS і передбачуваністю під час роботи з великими проєктами. Однак імперативний підхід створює значне навантаження на розробника: управління станом часто стає складним, а в контролерах накопичується надмірна кількість логіки, що призводить до добре відомої проблеми «Massive View Controller». Цей недолік зумовив пошук нових моделей побудови інтерфейсу, що згодом привело до появи SwiftUI.

1.2.2 SwiftUI: декларативна парадигма, принципи роботи та обмеження

SwiftUI був представлений Apple у 2019 році як сучасний, реактивний і технологічно прогресивний фреймворк, що кардинально змінює підхід до створення користувацького інтерфейсу. На відміну від UIKit, SwiftUI описує інтерфейс декларативно: розробник визначає не інструкції щодо оновлення UI, а кінцевий стан, який має бути відображений. Фреймворк самостійно визначає, як

оновити інтерфейс відповідно до змін даних. Така модель значно скорочує кількість технічного коду та мінімізує ризик помилок, пов'язаних із некоректною синхронізацією станів.

Архітектура SwiftUI заснована на реактивності. Механізми на кшталт `@State`, `@Binding`, `@ObservedObject` та `@EnvironmentObject` визначають логіку оновлення інтерфейсу та забезпечують автоматичну синхронізацію з даними. Інтеграція SwiftUI зі Swift Concurrency та Combine створює умови для побудови передбачуваних і прозорих потоків даних. За оцінками JetBrains Developer Ecosystem Report 2024, SwiftUI активно зростає в популярності серед розробників, особливо у нових проєктах [2].

Однак SwiftUI є технологією, що перебуває у фазі активного розвитку. Деякі компоненти або сценарії, які легко реалізуються у UIKit, у SwiftUI досі потребують мостів через `UIViewRepresentable` або `UIHostingController`. Крім того, фреймворк підтримується лише на нових версіях iOS, що створює обмеження для застосунків, орієнтованих на широку аудиторію. У складних інтерфейсах із нетривіальною логікою відтворення SwiftUI інколи демонструє менш стабільну продуктивність порівняно з UIKit, що також зазначається у рекомендаціях Apple щодо оптимізації UI [3].

1.2.3 Порівняння імперативного та декларативного підходів

UIKit і SwiftUI відображають еволюцію парадигм програмування, що сформувалися в різні етапи розвитку екосистеми Apple. Імперативний підхід UIKit передбачає, що оновлення інтерфейсу відбувається через послідовне виконання команд, які визначає розробник [21]. Цей підхід забезпечує максимальний рівень контролю та точності, проте вимагає складних механізмів управління станом і часто призводить до надмірної кількості службового коду.

Декларативна модель SwiftUI, навпаки, дає можливість описувати інтерфейс у термінах «що відображати», а не «як оновлювати». Це значно скорочує складність коду, робить його більш читабельним і полегшує тестування. Водночас декларативний підхід передбачає менший рівень низькорівневого контролю, а для

складних або нестандартних сценаріїв все ще виникає потреба у використанні UIKit-компонентів. Таким чином, обидві парадигми не виключають одна одну, а доповнюють, формуючи гібридну екосистему, у якій вибір залежить від конкретного типу проєкту.

1.2.4 Сценарії використання UIKit та SwiftUI в індустрії

Сучасна індустрія мобільної розробки свідчить про те, що вибір між UIKit і SwiftUI залежить від цілей проєкту, складності функціоналу та вимог до підтримки. UIKit залишається основним фреймворком у великих комерційних застосунках, де потрібна максимальна стабільність, сумісність зі старими версіями iOS та точний контроль над поведінкою UI. Багато корпоративних продуктів, банківських застосунків і високонавантажених сервісів продовжують використовувати UIKit саме через його перевіреність та передбачуваність.

SwiftUI ж дедалі активніше використовується під час створення нових застосунків, стартап-рішень та продуктів із швидким циклом розробки. Його зручність, декларативність та інтеграція з сучасними інструментами Apple роблять його привабливим для компаній, які прагнуть мінімізувати витрати на підтримку та пришвидшити виведення продукту на ринок. У практиці Apple SwiftUI вже використовується в багатьох системних компонентах iOS, що свідчить про стратегічний напрямок розвитку платформи. Водночас у великих застосунках часто застосовується гібридний підхід, який поєднує SwiftUI для нових модулів та UIKit для складних, продуктивно критичних інтерфейсів.

1.3 Архітектурні підходи у розробці мобільних застосунків iOS

Архітектура програмного забезпечення визначає структуру взаємодії між логікою, інтерфейсом та даними. Вона впливає на масштабованість, стабільність, простоту підтримки, а також на швидкість розробки застосунку. В екосистемі iOS архітектурні підходи еволюціонували разом із розвитком фреймворків, парадигм та вимог ринку. Від традиційних моделей UIKit-епохи до сучасних декларативних підходів SwiftUI — кожна архітектура має свою роль, переваги та обмеження. У

цьому підрозділі розглядаються основні архітектурні моделі, їх розвиток та критерії, що визначають доцільність їх застосування на практиці.

1.3.1 Традиційні архітектури MVC та MVP

Архітектура MVC (Model-View-Controller) є однією з найдавніших і найпоширеніших моделей, що використовувалася Apple як базовий шаблон для побудови застосунків на UIKit (рис. 1.4). У класичному розумінні MVC передбачає чіткий поділ між даними, логікою керування та відображенням. Однак у реальних проєктах для iOS вона часто перетворюється на так званий «Massive View Controller» [23-26].

Причиною цього є імперативна природа UIKit, де контролер бере на себе значну частину логіки — управління станом, взаємодію з мережею, обробку подій інтерфейсу та синхронізацію даних. Як зазначає Говард у «Clean Architecture for iOS Developers», це призводить до високої зв'язаності компонентів та ускладнює тестування великих проєктів.

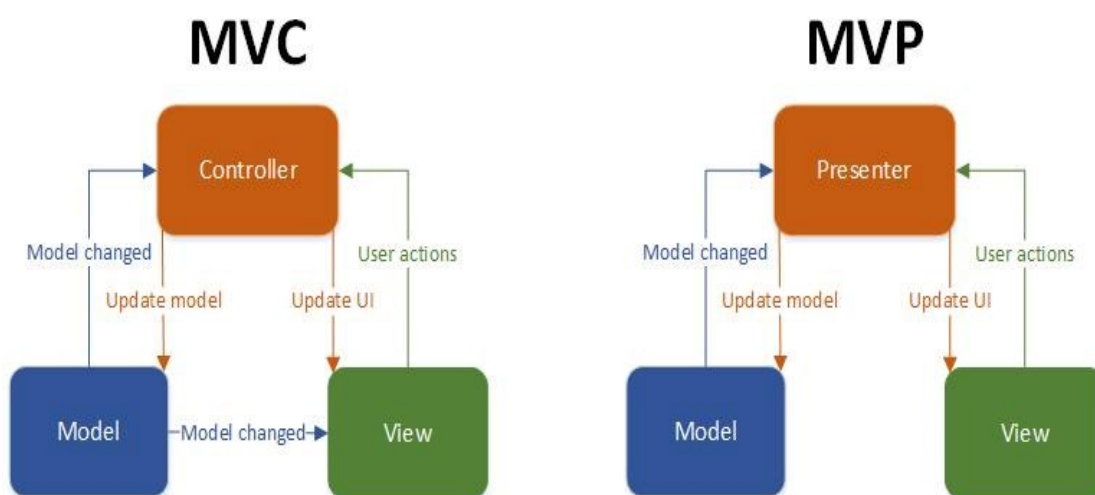


Рисунок 1.4 — Архітектура для побудови застосунків

Іншим класичним підходом є MVP (Model-View-Presenter), який пропонує розділити відповідальності більш рівномірно. Логіка з контролера переноситься до

презентера, а відображення стає пасивним. Це дозволяє зменшити складність UI-контролерів та зробити код більш передбачуваним.

Проте MVP також породжує велику кількість взаємодій між компонентами та підвищує складність синхронізації стану [5]. У монолітних застосунках або проєктах із великою кількістю екранних переходів це може перетворитися на додаткове навантаження для команди.

Тому традиційні архітектури хоч і залишаються важливою частиною історії iOS, у сучасних проєктах вони поступово витісняються більш гнучкими підходами.

1.3.2 Сучасні підходи MVVM та MVVM-C

З розвитком Swift і появою реактивних бібліотек дедалі більшої популярності набула архітектура MVVM (Model-View-ViewModel) (рис. 1.5). Вона вирішує основні недоліки MVC, зокрема проблему надмірної логіки у контролерах. У MVVM контролер перестає виконувати складні функції, а ViewModel бере на себе відповідальність за керування станом, обробку бізнес-логіки та підготовку даних до відображення.

У результаті інтерфейс реагує на зміну стану автоматично, що робить систему більш передбачуваною та простою в підтримці. Як зазначається у праці Натана Фурмана «Advanced iOS App Architecture», MVVM значно полегшує масштабування проєктів і робить коди більш тестованими [2].

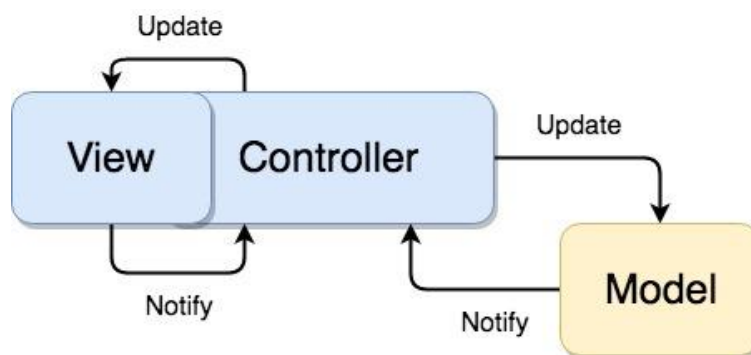


Рисунок 1.5 — Архітектура MVVM

Однак у великих застосунках, де існує складна навігація, сама модель MVVM не завжди достатня. Ця проблема стала передумовою появи архітектури MVVM-C, де до стандартної моделі додається компонент Coordinator. Координатор відповідає за навігацію між екранами та за життєвий цикл потоків користувача. Він усуває дублювання логіки переходів і дозволяє будувати модульну архітектуру без надмірного навантаження на ViewModel. Такий підхід сьогодні широко застосовується в індустрії, особливо у високонавантажених сервісах і застосунках із динамічними сценаріями взаємодії. MVVM-C є однією з найефективніших архітектур для великих UIKit-проектів, тому що дозволяє збалансувати гнучкість, тестованість і контроль над навігацією.

1.3.3 Архітектурні особливості SwiftUI та роль Combine

Поява SwiftUI кардинально змінила підходи до побудови інтерфейсу. На відміну від UIKit, де архітектура накладається на імперативну модель, SwiftUI інтегрує архітектурні принципи безпосередньо у свою декларативну природу. Основою SwiftUI є концепція стану: будь-яка зміна даних автоматично викликає оновлення інтерфейсу [22]. Це спрощує структуру застосунку, оскільки відпадає потреба вручну керувати синхронізацією між View та логікою.

SwiftUI часто порівнюють із MVVM, оскільки ViewModel природно вписується в модель реактивного оновлення інтерфейсу. У цій парадигмі View є функцією від стану, а бізнес-логіка зосереджується у ViewModel. Однак SwiftUI не вимагає жорсткого слідування MVVM у класичному вигляді. Стан у SwiftUI може керуватися локально через `@State`, між компонентами через `@Binding`, глобально — через `@EnvironmentObject`. Така архітектура має унікальні риси, яких немає в UIKit.

Ключову роль відіграє фреймворк Combine — офіційний інструмент Apple для реактивної обробки асинхронних подій. Combine дозволяє створювати потоки даних, трансформувати їх та підписуватися на зміни, що робить інтеграцію зі SwiftUI максимально природною. Згідно з Apple Developer Documentation, поєднання SwiftUI та Combine є рекомендованим сучасним підходом для створення

адаптивних, реактивних інтерфейсів [3]. Проте така архітектура висуває високі вимоги до розуміння життєвого циклу, управління ресурсами та будови потоків даних, що може створювати додаткові виклики для розробників.

1.3.4 Порівняння архітектурних підходів та критерії вибору

Порівняння MVC, MVP, MVVM, MVVM-C та архітектури SwiftUI демонструє еволюційний розвиток, спрямований на зменшення зв'язаності компонентів і полегшення підтримки застосунків. Традиційні підходи, характерні для UIKit, забезпечують максимальний рівень контролю, але вимагають значних зусиль під час підтримки та тестування [27]. Сучасні архітектури, навпаки, спрямовані на структурність, реактивність та передбачуваність роботи інтерфейсу.

У проектах, де важлива стабільність, повна сумісність і детальний контроль над компонентами, доцільним залишається використання UIKit у поєднанні з MVVM або MVVM-C. Для нових продуктів, особливо з акцентом на швидку розробку та адаптивний інтерфейс, SwiftUI стає більш привабливим рішенням. Остаточний вибір залежить від вимог до продуктивності, структури команди, очікуваної складності підтримки та планів щодо масштабування. Таким чином, архітектура не є універсальною константою, а визначається конкретним контекстом кожного проєкту.

1.4 Огляд процесу розгортання та публікації мобільних застосунків для iOS

Розгортання мобільного застосунку є невід'ємною частиною життєвого циклу розробки та включає комплекс технічних і адміністративних процедур, що забезпечують можливість встановлення програмного продукту на пристрої користувачів. Екосистема iOS вирізняється високим рівнем стандартизації, регламентованими вимогами та суворими правилами публікації, які мають на меті гарантувати безпеку, стабільність і відповідність застосунків системним рекомендаціям. Розуміння принципів підписування, тестування та публікації є

критично важливим для будь-якого розробника, оскільки визначає якість релізу та надійність подальшої підтримки продукту.

1.4.1 Механізм підписування: сертифікати, профілі, Bundle ID

Механізм підписування в iOS забезпечує автентичність застосунків і захищає екосистему Apple від встановлення неперевіреного програмного забезпечення. Підписання тісно пов'язане з інфраструктурою Apple Developer Program, яка передбачає використання сертифікатів та профілів розгортання. Кожен застосунок повинен мати унікальний ідентифікатор Bundle ID, який визначає його приналежність до певного розробника та дозволяє різним компонентам системи взаємодіяти із застосунком у передбачуваний спосіб.

Сертифікати підписування — Development і Distribution — забезпечують можливість запуску застосунку на реальних пристроях та його подальше розповсюдження. Профілі розгортання (Provisioning Profiles) поєднують сертифікат, Bundle ID та перелік дозволених пристроїв або каналів публікації. За даними Apple Developer Documentation, механізм підписування є обов'язковою умовою для будь-якого застосунку, оскільки iOS не дозволяє виконувати непідписаний код. Таким чином, підписування виступає фундаментальною передумовою безпеки та регуляції екосистеми.

1.4.2 Попереднє тестування за допомогою TestFlight

Після завершення основних етапів розробки застосунків проходить попереднє тестування, яке передбачає оцінку стабільності, виявлення помилок та перевірку поведінки інтерфейсу на різних пристроях. У екосистемі Apple для цього призначений сервіс TestFlight, який забезпечує розповсюдження бета-версій серед внутрішніх та зовнішніх тестувальників. Тестувальники можуть оцінювати функціональність, надсилати відгуки, фіксувати аномалії та автоматично передавати журнали збоїв, що робить TestFlight одним із головних інструментів забезпечення якості.

За даними Apple, TestFlight підтримує до 10 000 зовнішніх тестувальників та надає гнучкий механізм управління збірками, що дозволяє оперативно виправляти

помилки перед публічним релізом [2]. Особливу цінність становить інтеграція з App Store Connect, яка дає змогу контролювати статус збірок, терміни тестування та відгуки у структурованій формі. Попереднє тестування стає ключовим етапом у підготовці продукту, оскільки дозволяє підвищити стабільність та якість застосунку ще до проходження модерації.

1.4.3 Публікація застосунку в App Store: процес модерації

Публікація застосунку відбувається через платформу App Store Connect, яка виступає центральною системою керування продуктами. На цьому етапі розробник готує інформаційну сторінку застосунку, формує опис, ключові слова, додає скриншоти та метадані, визначає цінову політику і перелік країн, у яких застосунок буде доступний. Після цього збірка надходить на модерацію.

Процес модерації в App Store є одним із найдетальніших серед мобільних платформ. Модератори перевіряють відповідність застосунку технічним вимогам, продуктивності, стабільності, безпеці та дотриманню Apple Human Interface Guidelines. Окрема увага приділяється коректності прав доступу, обробці персональних даних, використанню API та відповідності функцій, заявлених у описі. За даними Apple, середній час розгляду становить від декількох годин до декількох днів, залежно від складності застосунку та кількості запитів у черзі [3].

Модерація є критично важливим етапом, оскільки відхилення збірки може призвести до затримок у релізі або необхідності перегляду функціональності. Проте саме завдяки суворому процесу перевірки екосистема App Store підтримує високий рівень безпеки та довіри серед користувачів.

1.4.4 Автоматизація розгортання CI/CD для iOS

У сучасній практиці мобільної розробки дедалі більшого значення набуває автоматизація процесів збирання, тестування та доставлення застосунків, що реалізується через системи CI/CD (рис. 1.6). Автоматизовані пайплайни дозволяють мінімізувати рутинні операції, знизити ризик людської помилки та забезпечити передбачуваність і повторюваність релізного процесу. У екосистемі iOS найпоширенішими рішеннями є Fastlane, GitHub Actions і Bitrise.

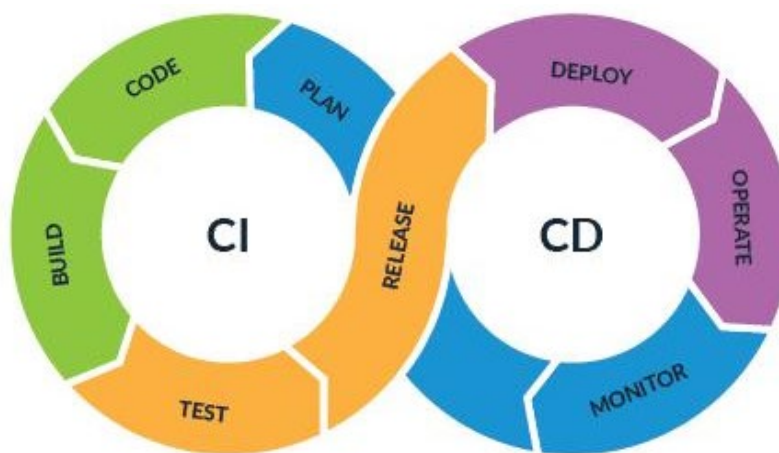


Рисунок 1.6 — Взаємодія процесів у системі CI/CD

Fastlane є гнучким набором інструментів, який автоматизує підписування, формування збірок, завантаження в App Store Connect і розповсюдження тестових версій. GitHub Actions надає можливість інтегрувати збирання та тестування у репозиторій і забезпечувати повну автоматизацію при кожному оновленні коду. Bitrise, у свою чергу, пропонує візуальний конструктор пайплайнів, оптимізований для мобільних платформ. Ці інструменти дозволяють створювати надійний процес безперервної інтеграції та доставки, що особливо важливо для команд, які працюють над складними або динамічними застосунками. Завдяки автоматизації можна підтримувати стабільний релізний цикл і швидко реагувати на зміни вимог або виправлення помилок.

1.5 Теоретичні основи перенесення мобільних застосунків на Android

Проблематика перенесення застосунків між мобільними платформами виникає через суттєві технічні, архітектурні та системні відмінності між екосистемами Apple та Google. Незважаючи на те, що iOS і Android домінують на ринку мобільних технологій, їхні базові принципи роботи, мови програмування, моделі рендерингу інтерфейсу та життєвого циклу застосунку є принципово різними. Саме тому процес перенесення застосунку не може бути прямолінійним і

потребує окремого вивчення теоретичних основ. Розглянемо ключові технічні відмінності, сучасні підходи до часткового та повного перенесення логіки й інтерфейсу, а також обґрунтовано неможливість прямого запуску iOS-коду на пристроях Android.

1.5.1 Технічні відмінності платформ iOS та Android

iOS та Android побудовані на різних системних архітектурах, що визначає їхню несумісність. iOS базується на Darwin — модифікованому UNIX-ядрі з власним набором фреймворків, зокрема UIKit та SwiftUI, які взаємодіють із системою за допомогою низки пропрієтарних API. Мовами базової розробки в iOS є Swift та Objective-C, а інтерфейс ґрунтується на механізмах рендерингу Core Animation.

Android, натомість, використовує ядро Linux і працює на віртуальній машині Android Runtime (ART). Його інтерфейс побудовано на фреймворку Jetpack Compose або класичному Android View System. Основними мовами є Kotlin і Java, а доступ до системних можливостей здійснюється за допомогою Android SDK [2].

Відмінності простежуються і в життєвому циклі застосунків. У iOS ключову роль відіграють механізми UIApplication та UIWindow, тоді як в Android життєвий цикл суворо регламентується системою Activity/Fragment. Різні принципи керування пам'яттю — ARC у Swift проти Garbage Collector у Kotlin — ще більше ускладнюють перенесення коду. Ці фундаментальні різниці створюють об'єктивні бар'єри для прямої сумісності та зумовлюють потребу у спеціальних інструментах для крос-платформенності.

1.5.2 Підходи до часткового перенесення: Kotlin Multiplatform Mobile

Kotlin Multiplatform Mobile (KMM) є одним із найперспективніших рішень для часткового перенесення мобільних застосунків. Його концепція полягає в тому, що бізнес-логіка та моделі даних реалізуються у спільному модулі на Kotlin, після чого цей модуль компілюється у нативний код iOS і Android. Таким чином, обидві платформи можуть використовувати один і той самий набір логічних компонентів, скорочуючи дублювання коду та полегшуючи підтримку.

КММ не розв'язує проблему перенесення інтерфейсу. Оскільки UI створюється окремо для iOS та Android, кожна платформа зберігає власні нативні підходи до відображення. Це дозволяє забезпечити високу продуктивність і відповідність гайдлайнам Apple Human Interface Guidelines та Material Design, але водночас потребує подвійної розробки інтерфейсу. У документації JetBrains стверджується, що КММ найбільше підходить для проєктів, де критично важлива спільна логіка, але не є пріоритетною уніфікація дизайну [3].

1.5.3 Підходи до повного крос-платформенного UI: Flutter, React Native

У випадках, коли необхідно переносити не лише логіку, а й інтерфейс, застосовуються крос-платформенні фреймворки повного циклу. Flutter є одним із таких рішень. Він використовує мову Dart і власний графічний рушій Skia, що дозволяє однаково відображати UI на iOS та Android. Flutter забезпечує швидку розробку та високу продуктивність, проте його інтерфейс не є нативним. Це може призвести до відхилень у поведінці компонентів від стандартних системних гайдлайнів, що важливо для застосунків, які потребують глибокої відповідності платформним правилам.

React Native працює на основі JavaScript і застосовує механізм «містка», через який взаємодіє з нативними компонентами. Такий підхід забезпечує гнучкість, але створює потенційні проблеми з продуктивністю, особливо під час інтенсивних анімацій або великих обсягів даних. Дослідження Meta та незалежних аналітичних платформ показують, що React Native найбільше підходить для застосунків середньої складності, де швидкість розробки важливіша за продуктивність [4].

У контексті перенесення нативного iOS-застосунку ці технології дозволяють повністю уніфікувати інтерфейс, але вимагають фактичного переписування проєкту з нуля, що зумовлює високі витрати часу та ресурсів.

1.5.4 Неможливість прямого запуску iOS-коду на Android: технічні та юридичні аспекти

Намагання запустити iOS-код безпосередньо на Android є технічно неможливим. Ключовою причиною є повна відсутність сумісності між системними

API, механізмами рендерингу та життєвими циклами застосунків. UIKit та SwiftUI не мають аналогів у Android SDK, а Swift-код не може виконуватися у середовищі Android Runtime без спеціальної компіляції, яка не підтримується системою.

Окрім технічних бар'єрів, існують юридичні аспекти. Apple чітко забороняє запуск iOS-системи або її компонентів на неавторизованому обладнанні, що детально зазначено в ліцензійній угоді iOS Software License Agreement [28-30]. Будь-які спроби емуляції iOS-середовища на Android або інших платформах порушують умови використання програмного забезпечення Apple. Експериментальні рішення, як-от Corellium, існують виключно у сфері безпекових досліджень і не можуть застосовуватися в комерційних продуктах. Це означає, що будь-який перенос iOS-застосунку на Android обов'язково передбачає повну або часткову переробку.

Висновки до розділу 1

У першому розділі було здійснено комплексний аналіз ключових аспектів розробки мобільних застосунків у екосистемі iOS. Розглянуто актуальність мобільного ринку, обґрунтовано вибір платформи iOS як стратегічно важливої, з огляду на її структурну частку на ринку високоплатоспроможних користувачів та значно вищий рівень монетизації порівняно з Android. Було проаналізовано технології створення інтерфейсу — UIKit та SwiftUI — які представляють різні парадигми побудови UI і мають свої переваги, обмеження та сфери застосування. Розгляд архітектурних підходів продемонстрував еволюцію від класичних моделей, таких як MVC та MVP, до сучасних MVVM та MVVM-C, а також до реактивної моделі SwiftUI, що інтегрує механізми керування станом на рівні фреймворку.

Детальний огляд процесів розгортання та публікації в App Store дозволив визначити специфіку екосистеми Apple, де значна увага приділяється перевірці якості, безпеці та відповідності рекомендаціям Human Interface Guidelines. Це створює додаткові вимоги до архітектури та проектування застосунків.

Окремо були розглянуті теоретичні засади перенесення застосунків на Android. Було показано, що пряме перенесення iOS-коду є неможливим як через технічні, так і через юридичні обмеження, а сучасні крос-платформенні технології або дублюють UI-частину, або вимагають повного перепроєктування інтерфейсу. Аналіз підходів — від KMM до Flutter та React Native — дозволив окреслити можливості та ризики, пов'язані з адаптацією застосунку до іншої операційної системи.

Таким чином, виконаний у розділі 1 теоретичний аналіз створює основу для практичної частини дослідження. Отримані результати дають можливість обґрунтовано формувати методологію експерименту, проєктувати архітектуру застосунку та переходити до розробки, яка буде представлена в наступному розділі.

2 ПРОЄКТУВАННЯ МОБІЛЬНОГО ЗАСТОСУНКУ

У даному розділі представлено підхід до проєктування мобільного застосунку, розробленого в рамках дослідження відмінностей між UIKit та SwiftUI. Основна мета — побудувати застосунок із чіткою структурою шарів, стандартизованими процесами обробки подій користувача та єдиною бізнес-логікою, що дозволить отримати об'єктивні результати експериментального порівняння двох технологій.

Проєктування виконано таким чином, щоб обидві реалізації — на UIKit та SwiftUI — були максимально подібними за функціональністю, сценаріями взаємодії та обсягом обчислень. Це забезпечує коректність порівняння продуктивності, зручності розробки, складності підтримки та поведінки інтерфейсу в реальних умовах роботи.

2.1 Постановка задачі та вимоги до застосунку

У рамках дослідження спроектовано мобільний застосунок типу менеджера завдань для малого та середнього бізнесу. Призначення системи — створення та відстеження робочих задач, контроль дедлайнів та організація щоденної діяльності співробітників. Застосунок складається з базових, але типових для бізнес-категорії сценаріїв: перегляд списку задач, створення нової задачі, редагування існуючої та зміна її статусу.

Цей функціонал достатній для порівняння можливостей UIKit і SwiftUI, оскільки охоплює ключові аспекти розробки інтерфейсу: роботу зі списками, форми введення даних, навігацію між екранами, оновлення стану та синхронізацію логіки між різними компонентами.

2.1.1 Функціональні та нефункціональні вимоги до застосунку

З погляду функціональності застосунок-менеджер завдань має підтримувати базовий, але завершений цикл роботи з задачами.

По-перше, користувач повинен мати змогу переглядати список задач, які стосуються його поточної діяльності. Список відображає назву задачі, короткий опис, дедлайн та поточний статус. Це дозволяє власнику малого бізнесу або менеджеру швидко оцінювати стан справ без переходу на детальні екрани, що є критичним у щоденній роботі.

По-друге, застосунок має забезпечувати створення нових задач та редагування існуючих. При створенні задачі користувач вводить назву, опис, дедлайн та пріоритет. Таке розбиття дає змогу не лише зафіксувати сам факт задачі, а й визначити її важливість у загальному потоці робіт. Редагування потрібне для реалістичного сценарію, коли умови виконання змінюються: переноситься дедлайн, уточнюється опис, змінюється відповідальний.

По-третє, важливою частиною функціональності є керування статусом задачі. Мінімально доцільно виділяти стани «нове», «в роботі» та «виконано». Такий поділ відображає типовий життєвий цикл бізнес-задачі: від постановки до завершення. Зміна статусу повинна одразу відображатися у списку, щоб користувач бачив актуальну картину без перезапуску застосунку.

Окремою вимогою є локальне зберігання даних. У межах цієї роботи не передбачається розробка серверної частини, тому всі задачі зберігаються на пристрої користувача. Це спрощує архітектуру, але водночас моделює реальну ситуацію для невеликої компанії, яка використовує один пристрій або кілька незалежних пристроїв без складної синхронізації. Нефункціональні вимоги впливають із особливостей цільової аудиторії та завдання дослідження.

По-перше, інтерфейс повинен працювати плавно та без помітних затримок при прокручуванні списку задач і переході між екранами. Менеджер, який переглядає десятки задач протягом дня, не має відчувати «підвисання» чи ривки анімації, тому продуктивність інтерфейсу є важливим критерієм оцінки.

По-друге, застосунок має бути простим та інтуїтивно зрозумілим. Для цього при проектуванні інтерфейсу враховуються рекомендації Apple Human Interface Guidelines: логічна ієрархія екранів, зрозумілі кнопки дій, передбачувана навігація. Це важливо не лише для кінцевого користувача, а й для коректного порівняння

UIKit і SwiftUI: обидві реалізації повинні пропонувати аналогічний рівень зручності.

По-третє, архітектура застосунку має бути розширюваною. Передбачається, що в майбутньому до базового менеджера завдань можна буде додати такі можливості, як календар перегляду задач, нагадування, сортування за виконавцями чи підтримку командної роботи. Тому вже на етапі проєктування закладається поділ на шари, який полегшить розвиток системи без повного переписування коду.

Ще одна важлива група вимог стосується тестованості та незалежності бізнес-логіки від UI-фреймворку. Логіка роботи із задачами (створення, зміна статусу, фільтрація, перевірка дедлайнів) повинна бути реалізована у вигляді окремих сервісів і моделей, які не залежать від UIKit чи SwiftUI. Це дозволяє, з одного боку, писати модульні тести, а з іншого — виконувати чесне порівняння двох фреймворків, змінюючи лише шар представлення, а не всю систему.

Нарешті, окремою нефункціональною вимогою є консистентність поведінки між реалізаціями на UIKit та SwiftUI. Користувач, який переходить з однієї версії інтерфейсу на іншу, має бачити той самий набір можливостей, однакові стани задач і схожу логіку навігації [21, 33]. Це принципово важливо для дослідження, оскільки дозволяє пов'язувати відмінності у зручності розробки чи продуктивності саме з вибором фреймворку, а не з різницею у функціоналі.

2.1.2 Середовище розробки Xcode та процес створення проєкту

Для створення та подальшої роботи над мобільним застосунком було використано середовище розробки Xcode, яке є стандартним інструментом для розробників екосистеми Apple [35]. Він надає інтегроване середовище, що об'єднує редактор коду, інструменти інтерфейсного проєктування, засоби налагодження та профілювання продуктивності. Саме завдяки комплексності Xcode забезпечує повний цикл розробки — від створення нового проєкту до публікації застосунку.

Створення проєкту розпочинається зі стартового вікна, яке з'являється під час запуску середовища (рис 2.1). У цьому вікні користувач обирає створення нового застосунку, після чого відкривається майстер ініціалізації проєкту. Він містить кілька послідовних екранів, на яких необхідно визначити основні параметри майбутнього застосунку.



Рисунок 2.1 — Стартове вікно програмної системи

Передусім обирається тип проєкту; у випадку дослідження використано шаблон «App», який є універсальним і підходить як для UIKit-, так і для SwiftUI-орієнтованих інтерфейсів (рис. 2.2)

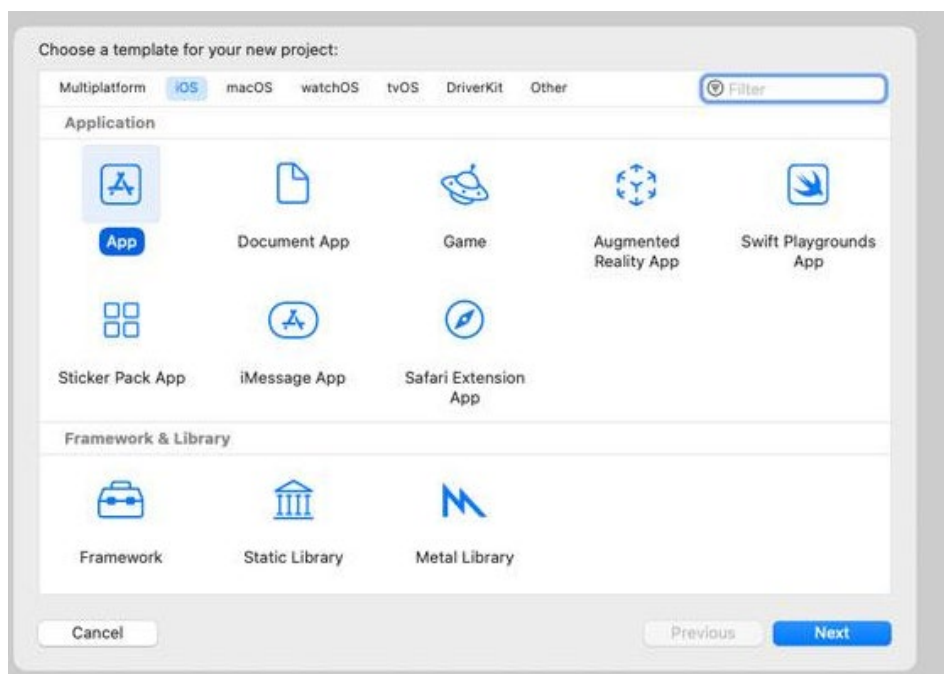


Рисунок 2.2 — Вибір шаблону для проєктування

Після вибору шаблону відкривається форма налаштування, де задаються назва продукту, ідентифікатор організації, мова програмування та базовий підхід до побудови інтерфейсу. Усі параметри заповнюються відповідно до завдань роботи: мова програмування — Swift, а інтерфейсна технологія визначається окремо для кожного таргету, оскільки дослідження передбачає порівняння двох реалізацій одного застосунку (рис. 2.3)

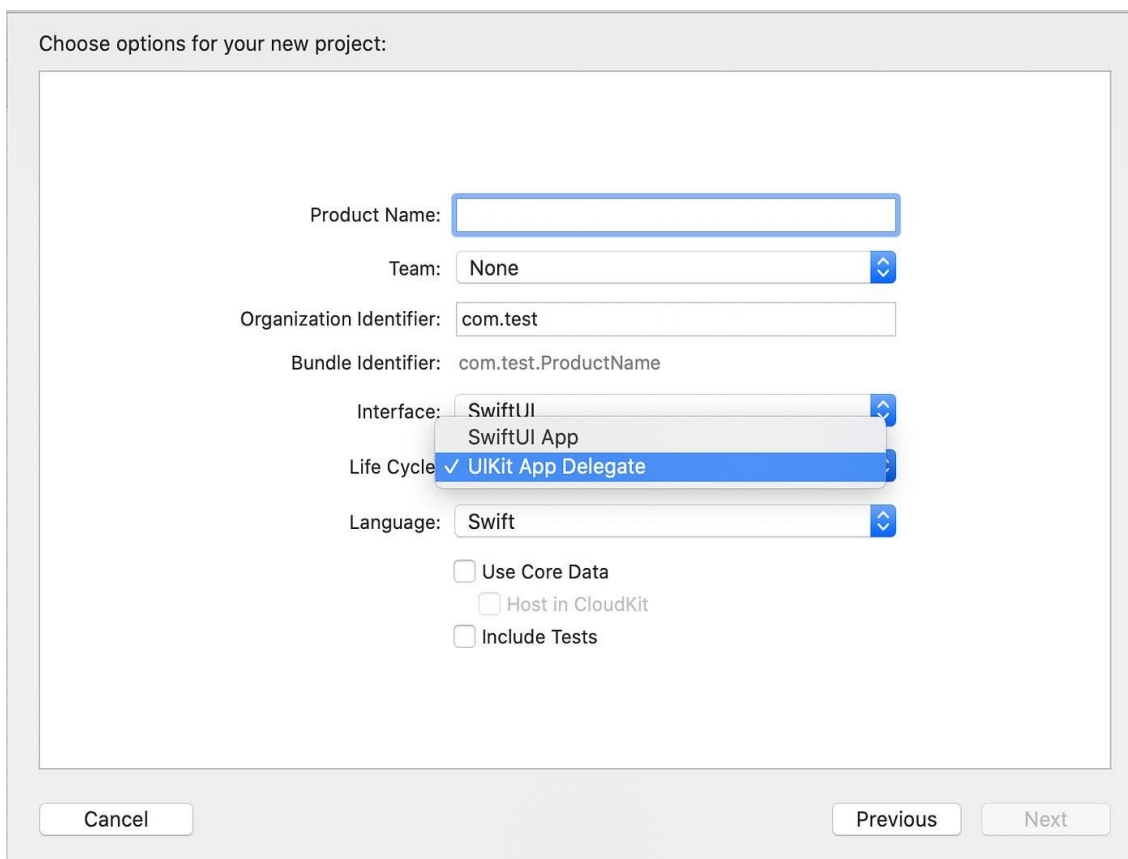


Рисунок 2.3 — Заповнення параметрів проєктування

Після завершення майстра Xcode автоматично формує структуру проєкту та відкриває головне робоче вікно. Структура, що генерується за замовчуванням, включає початкові файли, необхідні для запуску застосунку, конфігураційні ресурси та службові каталоги. Залежно від обраної технології інтерфейсу початковою точкою входу виступає або структура App у SwiftUI, або комплект класів AppDelegate і SceneDelegate у випадку UIKit. Разом із ними створюються

початкові представлення інтерфейсу, файл конфігурації Info.plist та каталог ресурсів, у якому зберігаються графічні елементи. Стандартна структура проєкту надалі була доповнена окремими підгрупами, які чітко розділяють реалізацію інтерфейсу на UIKit та SwiftUI, а також спільним модулем бізнес-логіки. Така організація дозволяє уникнути дублювання коду та зосередити відмінності виключно на рівні представлення.

Після створення проєкту налаштовуються параметри таргета. Ключовим рішенням є визначення мінімальної версії операційної системи. У межах дослідження встановлено підтримку iOS не нижче 16-ї версії, оскільки саме починаючи з цього випуску SwiftUI отримав стабілізовану систему навігації, вдосконалену модель рендерингу та низку оптимізацій, важливих для коректного порівняння із UIKit. Окремо визначаються підтримувані типи пристроїв: застосунок орієнтований на використання на iPhone і не містить інтерфейсних рішень, специфічних для iPadOS, що зумовлено фокусом на мобільному сценарії використання у малому та середньому бізнесі. Такі обмеження дозволяють спростити архітектуру й уникнути необхідності окремого проєктування інтерфейсів для різних класів пристроїв.

Середовище Xcode також забезпечує зручні інструменти для тестування та оцінювання продуктивності створених реалізацій [31-34]. Вбудовані симулятори дають змогу запускати застосунок на різних моделях iPhone без фізичного пристрою, а система інструментів Instruments дозволяє аналізувати використання пам'яті, частоти оновлення інтерфейсу та швидкодію основних операцій. Це має важливе значення у контексті роботи, адже порівняння UIKit і SwiftUI передбачає описові висновки й вимірювання фактичних характеристик роботи інтерфейсу.

У підсумку Xcode виступає не просто інструментом написання коду, а універсальною платформою для проведення дослідження. Воно забезпечує можливість створити два варіанти одного застосунку в межах однієї структури, використовуючи спільну бізнес-логіку та ізольовані реалізації інтерфейсу, що є необхідною умовою для коректного експериментального порівняння формальних та практичних відмінностей між UIKit і SwiftUI.

2.1.3 Налаштування структури проєкту та організація коду

Після створення базового проєкту в Xcode наступним етапом стало формування його внутрішньої структури, яка б одночасно відповідала вимогам дослідження та забезпечувала зручність подальшої розробки. Оскільки метою роботи є порівняння двох підходів до побудови інтерфейсу — UIKit та SwiftUI — надзвичайно важливо було від самого початку організувати проєкт так, щоб ці реалізації були чітко розмежовані, але при цьому використовували спільну бізнес-логіку. Це дозволяє уникнути дублювання коду та забезпечити коректність експерименту, адже обидві реалізації повинні спиратися на однакову модель даних і однакові правила обробки інформації.

Структура проєкту формувалася за принципом поділу на логічні модулі. У межах кореневого каталогу було створено окрему групу для бізнес-рівня, у якій розміщено моделі задач, сервіси керування даними та допоміжні компоненти, що відповідають за перевірку дедлайнів, оновлення статусів та інші логічні операції. Усі ці елементи реалізовано у вигляді незалежних класів та структур, які не містять посилань на будь-який UI-фреймворк і не використовують специфічних компонентів UIKit або SwiftUI [36-38]. Така ізоляція забезпечує можливість багаторазового використання логічного шару у різних інтерфейсних реалізаціях та дозволяє тестувати його окремо від користувацького інтерфейсу.

Окремими групами були створені модулі для реалізації інтерфейсу на UIKit та на SwiftUI. У групу UIKit включено контролери представлень, компоненти навігації та спеціальні допоміжні класи, необхідні для імперативної побудови інтерфейсу. Вони спираються на класичний життєвий цикл UIViewController та на механізми UIKit для роботи з ієрархією виглядів. В іншу групу, присвячену SwiftUI, увійшли декларативні представлення, їхні моделі стану та інші компоненти, характерні для побудови інтерфейсу за декларативним підходом. Обидва модулі інтерфейсу працюють із тими самими моделями та сервісами, що розташовані у спільному каталозі, тому поведінка застосунку з точки зору логіки залишається ідентичною.

Особливу увагу було приділено розмежуванню навігації та внутрішньої логіки UI. У реалізації UIKit навігація організовувалась у відповідності до патерну Coordinator, що дозволяє винести управління переходами у окремі об'єкти та зменшити залежності між контролерами [39]. У випадку SwiftUI навігаційна структура базується на використанні UINavigationController, яка забезпечує декларативний опис переходів. Попри суттєві концептуальні відмінності між цими механізмами, обидві реалізації було побудовано на основі однакової логіки переходів, що описує послідовність екранів, їхні стани та умови активації. Це дало можливість гарантувати еквівалентність двох реалізацій у частині сценаріїв користувацької взаємодії.

Формування такої структури проєкту мало на меті не лише упорядкувати код, але й забезпечити можливість об'єктивного порівняння двох підходів до побудови інтерфейсу [40]. Оскільки усі логічні та дані компоненти є спільними, а відмінності зосереджені виключно у шарі представлення, результати аналізу продуктивності, складності розробки та поведінки інтерфейсу можуть інтерпретуватися без ризику змішування факторів. У такий спосіб структура проєкту стає не просто технічним рішенням, а частиною методології дослідження, яка забезпечує достовірність подальших порівняльних висновків.

2.2 Вибір архітектури та підходів до проєктування

Архітектура мобільного застосунку визначає спосіб організації його внутрішньої логіки, структуру взаємодії між модулями та механізми оновлення інтерфейсу у відповідь на зміни даних. Оскільки в межах цього дослідження головним завданням є порівняння двох підходів до побудови користувацького інтерфейсу — UIKit та SwiftUI, — вибір архітектурних рішень був спрямований на забезпечення максимальної відповідності між цими реалізаціями на рівні логіки та структури даних, щоб усі відмінності стосувалися саме інтерфейсу, а не внутрішньої поведінки системи.

Основним принципом стала побудова застосунку за моделлю, близькою до MVVM. Такий підхід дає змогу розділити відповідальність між трьома групами компонентів: моделями даних, які зберігають та описують стан задач; представленнями, які відображають інформацію користувачеві; та моделями подання, які реалізують перетворення даних, правила оновлення стану і реакцію на користувацькі дії. Незалежно від того, використовується UIKit чи SwiftUI, цей поділ дозволяє працювати з однаковими даними та однаковими моделями подання. В UIKit моделі подання взаємодіють із контролерами представлень, які відповідають за оновлення елементів на екрані, а в SwiftUI ті ж моделі подання виконують роль джерела істини для декларативних представлень, які автоматично оновлюються при зміні стану.

У реалізації UIKit була застосована модифікована архітектура MVVM з використанням координаторів для керування навігацією. Координатор виступає додатковим шаром, призначеним для винесення логіки переходів між екранами за межі контролерів, що дозволяє уникнути надмірної залежності між окремими частинами інтерфейсу [6]. Такий підхід забезпечує структурованість, передбачуваність та можливість розширення застосунку без порушення ієрархії. Використання координаторів також є поширеною практикою у сучасних UIKit-проектах, що підвищує реалістичність дослідження та дає змогу оцінити порівняння в умовах, наближених до реальних комерційних застосунків.

У випадку SwiftUI архітектура застосунку ґрунтувалася на концепції керування станом, яка є фундаментальною для цього фреймворку. SwiftUI не передбачає прямого аналога контролерів, а інтерфейс будується як декларативне відображення поточного стану [7]. Моделі подання у такому підході виступають як об'єкти, що публікують зміни, на які автоматично реагують представлення. Цей механізм дозволяє значно скоротити кількість коду, який у UIKit необхідно писати вручну, та усуває потребу в імперативному керуванні елементами інтерфейсу. Незважаючи на відмінність у парадигмах, моделі подання залишаються аналогічними до тих, що використовувалися в UIKit, що дозволило зберегти спільність логічного шару.

Окремою частиною архітектури, спільною для обох варіантів інтерфейсу, стала система керування даними. Вона включає сервіси для створення та оновлення задач, зберігання інформації у локальному середовищі пристрою та виконання простих запитів, пов'язаних із фільтрацією або зміною стану задач. Логіка цих сервісів була розроблена у вигляді незалежних модулів, які не прив'язані до будь-якого фреймворку. Завдяки цьому обидві реалізації інтерфейсу отримують доступ до однакових функціональних можливостей, що виключає можливість впливу відмінностей у даних на результати порівняння.

Узгодженість архітектурних рішень у двох інтерфейсних реалізаціях створила передумови для коректного експерименту, у якому порівнюються саме поведінкові та структурні відмінності між UIKit і SwiftUI. Таким чином, архітектура застосунку була побудована не лише з позиції технічної доцільності, але й відповідно до методологічних вимог дослідження, що забезпечує можливість формулювання об'єктивних висновків у подальших розділах роботи.

2.3 Структура програмного проєкту

Структура програмного проєкту формувалася з урахуванням подвійної мети дослідження: забезпечити можливість розробки функціонального застосунку менеджера завдань та створити умови для коректного порівняння реалізацій інтерфейсу на UIKit і SwiftUI. Тому організація коду була побудована за принципом чіткого розмежування логічних, даних та інтерфейсних компонентів, при цьому інтерфейсні частини двох реалізацій існують паралельно й незалежно одна від одної, але звертаються до спільного ядра бізнес-логіки.

У центрі структури проєкту розташований модуль даних, який містить моделі задач, сервіси збереження та обробки інформації та допоміжні компоненти, пов'язані з бізнес-логікою. Саме цей модуль виступає основою двох інтерфейсних реалізацій. Моделі були оформлені як прості структури та класи з мінімальним набором властивостей, достатнім для того, щоб описати основні параметри задач: назву, опис, дедлайн, пріоритет і статус. Логіка роботи з даними реалізована в

окремому сервісі, що забезпечує створення нових записів, їх редагування та збереження у локальному середовищі пристрою. Оскільки у межах дослідження не передбачається використання складних систем синхронізації або серверної взаємодії, дані зберігаються локально, що дозволило сконцентрувати увагу на порівнянні саме інтерфейсних підходів.

Навколо бізнес-логіки формувалися два інтерфейсні модулі, кожен із яких відповідає за власний спосіб побудови та оновлення користувацького інтерфейсу. Реалізація на UIKit ґрунтується на використанні класичної для цього фреймворку системи контролерів представлень. Кожен екран має власний контролер, який відповідає за відображення даних, реагування на події користувача та взаємодію з моделями подання. Окремий компонент займається навігацією і визначає правила переходів між екранами. Такий підхід забезпечує більш явну структуру управління, властиву імперативній природі UIKit. Реалізація інтерфейсу була побудована таким чином, щоб кожен екран містив мінімум логіки, передаючи основні операції до відповідних моделей подання.

Паралельна реалізація на SwiftUI має подібну структуру та функціональність, однак використовує декларативний підхід до опису інтерфейсу. Замість контролерів представлень у цьому модулі використовуються окремі SwiftUI-представлення, кожне з яких відображає певний стан даних, а зміни у моделях подання автоматично призводять до оновлення інтерфейсу. Навігація організована за допомогою `NavigationStack`, що надає можливість описувати послідовність переходів у декларативній формі. При цьому логіка роботи відрізняється концептуально: там, де UIKit вимагає явного виклику оновлення інтерфейсу, SwiftUI виконує це автоматично завдяки механізму спостереження за станом. Незважаючи на це, загальна структура екранів і переходів у двох реалізаціях зберігається однаковою, що важливо для подальшого порівняння.

Важливим аспектом організації структури стало забезпечення того, щоб усі логічні модулі залишалися максимально незалежними від UI-фреймворків. Це дозволяє використовувати спільні моделі та сервіси в обох реалізаціях і гарантує, що результати порівняння інтерфейсних технологій не спотворюються

відмінностями у бізнес-логіці. Таким чином, упорядкована структура проєкту не лише полегшує розробку, але й виконує роль важливої методологічної основи для дослідження, адже забезпечує рівні умови для двох варіантів інтерфейсу.

2.4 Діаграмне моделювання системи

Проектування застосунку, окрім опису архітектури та структури коду, потребує формалізованого подання його логіки у вигляді діаграм. Використання діаграм у процесі розробки дозволяє отримати узагальнений погляд на систему, виявити залежності між її компонентами та визначити ключові сценарії взаємодії. У межах цього дослідження діаграмне моделювання відіграє подвійну роль: воно слугує не лише етапом проектування застосунку менеджера завдань, але й допомагає забезпечити однаковість структури двох інтерфейсних реалізацій на UIKit та SwiftUI.

Першим етапом стало створення діаграми варіантів використання, що відображає основні дії користувача в межах застосунку. Ця діаграма дозволяє формалізувати призначення системи, показуючи, які саме функції є ключовими та яким чином вони взаємопов'язані. На діаграмі окреслюється єдиний актор — користувач, який взаємодіє із системою шляхом перегляду списку задач, створення нової задачі, переходу до детального відображення та редагування поточних записів. Усі ці дії формують замкнений цикл роботи з задачею, що відображає логіку використання застосунку в реальних умовах малого бізнесу. Діаграма варіантів використання виступає своєрідним контекстом для всієї подальшої розробки, задаючи набір обов'язкових можливостей, які мають бути реалізовані в обох інтерфейсних варіантах.

Наступним кроком стала побудова діаграми компонентів. Її основне завдання — представити внутрішню структуру проєкту на високому рівні, незалежно від конкретної технології інтерфейсу. На діаграмі описано три основні групи компонентів: модуль даних, модуль бізнес-логіки та модулі інтерфейсу. Модуль даних відповідає за моделі задач і базові операції зі збереженням інформації у локальному середовищі пристрою. Модуль бізнес-логіки містить сервіси обробки

задач і моделі подання, на які спираються обидві інтерфейсні реалізації. Два окремі інтерфейсні модулі відповідають за побудову представлення відповідно до фреймворків UIKit і SwiftUI, отримуючи однакові дані з логічного шару. Такий спосіб моделювання демонструє незалежність логіки від інтерфейсу та є важливою передумовою для об'єктивного порівняння двох технологій.

Для деталізації взаємодії між класами та даними була розроблена діаграма класів. На ній відображено структуру основних моделей застосунку, зокрема модель задачі, яка містить ключові властивості, необхідні для управління щоденною діяльністю: назву, опис, дедлайн, пріоритет і статус. Також на діаграмі подано класи моделей подання, які здійснюють перетворення даних для відображення у відповідних інтерфейсах. Сервіс керування задачами представлено як окремий клас, що виконує операції зі створення, редагування та оновлення стану задач. Діаграма класів дозволяє побачити внутрішню структуру системи, логіку її об'єктної моделі та взаємозв'язки, що будуть використані в обох варіантах інтерфейсу.

Оскільки менеджер завдань передбачає переміщення між кількома екранами, важливою частиною моделювання стала побудова діаграми навігації. Вона відображає послідовність переходів між основними представленнями: список задач, детальний опис задачі та форма її редагування чи створення. Незважаючи на те, що UIKit та SwiftUI реалізують навігацію принципово різними способами, діаграма описує саме логічний рівень переходів, який однаковий для обох реалізацій. У випадку UIKit ці переходи реалізуються засобами координатора, що відповідає за імперативне керування екранами, а в SwiftUI — декларативно, за допомогою механізму `NavigationStack` [41]. Моделювання навігації забезпечує структурну єдність двох реалізацій, уникаючи розбіжностей, що могли б вплинути на результати порівняльного аналізу.

У сукупності ці діаграми формують цілісне уявлення про архітектуру застосунку і його функціональну організацію. Вони дозволяють уніфікувати підходи до реалізації двох інтерфейсних варіантів, визначити межі відповідальності між модулями та забезпечити прозору структуру для подальшої розробки. Завдяки цьому діаграмне моделювання не лише структуризує проєкт, але

й виконує важливу методологічну функцію, створюючи основу для коректного експериментального порівняння UIKit і SwiftUI.

2.5 Підготовка застосунку до порівняльного експерименту

Проведення порівняльного аналізу між UIKit та SwiftUI вимагає попередньої підготовки застосунку, яка забезпечує відтворюваність експерименту та коректність отриманих результатів. Основною умовою такої підготовки є створення однакових функціональних умов для обох інтерфейсних реалізацій, оскільки під час дослідження мають порівнюватися не дані чи логіка, а саме інтерфейсні підходи та інженерна специфіка двох фреймворків.

Першим кроком стало забезпечення функціональної еквівалентності обох варіантів застосунку. Незважаючи на суттєві відмінності між UIKit та SwiftUI, обидві реалізації повинні містити однакові екрани, підтримувати однакові сценарії взаємодії та використовувати однакові моделі й сервіси для роботи з даними. Саме для цього бізнес-логіка була винесена у спільний модуль, доступний для двох інтерфейсних частин. Моделі задач, сервіси їх обробки, а також правила оновлення стану задач виконуються однаково і не залежать від вибору інтерфейсного фреймворку. Це дозволяє виключити вплив логічних відмінностей на результати експерименту та гарантує, що під час порівняння оцінюються лише особливості побудови інтерфейсу.

Другою важливою складовою підготовки стало визначення умов, у яких відбуватиметься експериментальне оцінювання. Обидві реалізації застосунку тестувалися на одному й тому самому середовищі: на однакових моделях симульованих пристроїв у Xcode, під однією й тією ж версією iOS та з однаковими налаштуваннями таргета. Це дозволяє уникнути впливу апаратних розбіжностей, що могли б суттєво змінити сприйняття продуктивності, плавності анімацій чи часу реакції інтерфейсу. Всі вимірювання виконувалися у тому самому Xcode-процесі, що також зменшує коливання у завантаженні системних ресурсів.

Окремої уваги потребувало забезпечення повторюваності тестування. Для цього було визначено заздалегідь сформований набір тестових сценаріїв, які моделюють типові користувацькі дії: перегляд списку задач різного обсягу, створення та редагування записів, зміну статусів та повернення до попередніх екранів. Ці сценарії виконувалися в однаковій послідовності для обох реалізацій, що дозволило зіставити не лише якісні особливості роботи інтерфейсу, але й потенційні відмінності в часі виконання операцій, стабільності оновлення інтерфейсу та споживанні ресурсів.

Технічна сторона підготовки передбачала також налаштування інструментів профілювання продуктивності. В обох випадках використовувалися однакові модулі Instruments, такі як Time Profiler та Core Animation, що дозволяє оцінити, як саме фреймворк оновлює інтерфейс у відповідь на зміну стану, наскільки стабільно відтворюється анімація та які операції вимагають найбільшої обчислювальної потужності. Вимірювання виконувалися у контрольованих умовах з однаковими початковими параметрами, що підвищує достовірність експериментальних результатів.

Підготовка застосунку також включала перевірку консистентності інтерфейсних рішень. Хоча UIKit і SwiftUI пропонують різні підходи до побудови екранів, важливо було забезпечити максимально подібний візуальний вигляд обох реалізацій. Це стосується ієрархії елементів, розташування кнопок, способів відображення списку задач та логіки переходів між екранами. Навіть незначні відмінності в дизайні могли б вплинути на суб'єктивне сприйняття інтерфейсу, тому структура екранів була уніфікована настільки, наскільки це дозволяє різниця між декларативним і імперативним підходами.

Таким чином, підготовка застосунку до експерименту охопила низку методичних та технічних рішень, які дозволили створити контрольовані умови для порівняння UIKit і SwiftUI. Забезпечення еквівалентності бізнес-логіки, уніфікованої структури інтерфейсу, стабільних умов тестування та повторюваності сценаріїв дало змогу перейти до експериментальної частини дослідження, результати якої будуть представлені в наступному розділі.

Висновки до розділу 2

У даному (проектному) розділі було здійснено комплексне проектування мобільного застосунку менеджера завдань, яке стало підґрунтям для подальшого порівняльного аналізу UIKit і SwiftUI. Вибір середовища розробки, архітектурних підходів та структурна організація проєкту були спрямовані на створення умов, у яких обидві інтерфейсні реалізації могли б бути розроблені незалежно, але функціонувати на основі спільної бізнес-логіки. Такий підхід дозволив сформулювати цілісну, послідовну інженерну модель застосунку, у якій усі логічні компоненти ізольовані від UI-фреймворків, що забезпечує коректність подальшого порівняння.

У процесі проектування було визначено загальні вимоги до застосунку, обрано архітектурні принципи, які дозволяють однаково підтримувати як імперативний, так і декларативний підхід до побудови інтерфейсу, а також сформовано структуру проєкту, зорієнтовану на зрозуміле розмежування відповідальності між модулями. Діаграмне моделювання дало змогу систематизувати внутрішні взаємозв'язки, описати основні сценарії взаємодії та формалізувати логіку побудови екранів і переходів між ними. Окрема увага була приділена підготовці застосунку до експериментального дослідження, що включало створення однакових функціональних умов для двох інтерфейсних реалізацій, уніфікацію їх структури та визначення єдиних критеріїв тестування.

Таким чином, виконане проектування створює методологічно обґрунтовану основу для аналізу відмінностей між UIKit і SwiftUI, дозволяючи оцінити їхні особливості в умовах реального застосунку та забезпечити достовірність висновків, які будуть наведені у наступному розділі.

3 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТЕХНОЛОГІЙ ПОБУДОВИ ІНТЕРФЕЙСІВ

У цьому розділі представлено експериментальне дослідження, спрямоване на порівняння двох основних технологій побудови інтерфейсів в екосистемі iOS — SwiftUI та UIKit. Попри те, що SwiftUI офіційно позиціонується як фреймворк майбутнього, а UIKit продовжує залишатися стандартом промислової розробки, їхнє безпосереднє зіставлення в реальних умовах досі є предметом дискусій серед розробників і дослідників мобільних інтерфейсів. У зв'язку з цим виникає потреба в системному, методологічно обґрунтованому аналізі їхніх продуктивнісних характеристик, енергоефективності, стабільності роботи та складності реалізації.

Попередні розділи роботи були спрямовані на побудову архітектури застосунків, які використовуються як експериментальні моделі у цьому дослідженні. На основі єдиної логічної структури було створено два повноцінних програмних варіанти одного й того самого застосунку: перший реалізований із використанням SwiftUI, другий — на основі UIKit. Обидві версії містять ідентичний набір екранів, функцій, навігаційних сценаріїв і станів, що дозволяє виключити вплив побічних чинників та сконцентрувати увагу саме на відмінностях між фреймворками.

Метою даного розділу є формування об'єктивних, відтворюваних результатів, отриманих у процесі виконання експериментальних тестів, які охоплюють різні типи навантажень: навігаційні переходи, роботу зі списками, рендеринг складних інтерфейсів, виконання анімацій, обробку подій та інші характерні процеси життєвого циклу мобільного застосунку. Особливу увагу приділено вимірюванню завантаження системних ресурсів, швидкості оновлення інтерфейсу, стабільності FPS, обсягу алокацій пам'яті, часу первинного рендерингу та реактивності на дії користувача.

Таким чином, розділ 3 має на меті не лише подати кількісні результати, але й пояснити їх у контексті архітектурних та концептуальних особливостей обох підходів. Структура розділу включає опис методики, презентацію сценаріїв

тестування, інструментарій вимірювань, таблиці отриманих результатів, а також аналітичні висновки щодо доцільності використання SwiftUI та UIKit у різних класах задач.

3.1 Мета та методика експериментального дослідження

Метою експериментального дослідження є встановлення об'єктивних відмінностей між фреймворками SwiftUI та UIKit у контексті продуктивності, стабільності рендерингу користувацького інтерфейсу, ресурсоємності та складності розробки. Обидві технології офіційно підтримуються Apple, проте належать до різних парадигм: UIKit реалізує класичний імперативний підхід, тоді як SwiftUI базується на декларативному оновленні стану. Таке концептуальне розходження створює підстави очікувати суттєвих відмінностей у їхній поведінці під навантаженням, що й визначає необхідність проведення систематичного експериментального аналізу.

Головним завданням дослідження є отримання кількісних показників, які відображають реальну роботу фреймворків у типових сценаріях мобільних застосунків. Для цього було сформовано гіпотезу, що SwiftUI продемонструє перевагу в рендерингу статичних або помірно динамічних інтерфейсів, однак може виявити нижчу стабільність та більші витрати CPU у сценаріях зі значними змінами стану та у складних спискових структурах. Натомість UIKit, як технологія з тривалим історичним шляхом розвитку та високою оптимізацією під потік UI-операцій, має показати кращі результати в умовах великої кількості елементів, активного скролінгу та багаторівневих ієрархій.

Методика дослідження базується на принципі повної симетричності експериментальних умов. Для кожного фреймворка було розроблено ідентичний застосунок, який включає однакові екрани, навігаційні переходи, бізнес-логіку та структуру даних. Такий підхід мінімізує вплив сторонніх змінних та дозволяє інтерпретувати виміряні параметри як безпосередній результат роботи фреймворку, а не особливість конкретної реалізації.

Під час експерименту використовувалися апаратні та програмні засоби вимірювання, стандартні для екосистеми iOS. Роботу інтерфейсу аналізували за допомогою інструментів пакета Instruments, зокрема Time Profiler, Allocations, Core Animation та System Trace.

Показники кадрів (FPS), пропущених кадрів та GPU-навантаження визначалися через Core Animation Instrument, що дозволяє дослідити ефективність побудови та повторного рендерингу елементів. Для зняття системних метрик продуктивності застосовувалися інструменти XCTest Metrics та MetricKit, а критичні ділянки коду позначалися OSSignpost-маркуванням для точного фіксування часових інтервалів.

Важливим компонентом методики є контроль оточення. Усі вимірювання проводилися на одному фізичному пристрої — iPhone 14 з операційною системою iOS 17 — у режимі мінімального фоновго навантаження. Тести кожного сценарію повторювалися по три рази, а отримані значення усереднювалися. Для уникнення впливу «холодного старту» та кешування перед проведенням кожного циклу тестів застосунок перевстановлювався та перезапускався.

Зібрані метрики охоплюють час рендерингу першого екрана, середнє та пікове завантаження CPU та GPU, обсяг використаної оперативної пам'яті, частоту кадрів під час активної взаємодії, кількість алокацій пам'яті, а також реактивність інтерфейсу щодо подій користувача.

Окрім цього, оцінювалася складність розробки через аналіз обсягу коду, структурної організації компонентів та часу, необхідного для реалізації відповідних модулів.

Таким чином, застосована методика забезпечує комплексний підхід до дослідження, у якому продуктивність, стабільність та зручність розробки розглядаються як взаємопов'язані, але незалежно вимірювані параметри. Результати, отримані завдяки цьому підходу, дозволяють сформулювати науково обґрунтовані висновки щодо доцільності використання SwiftUI або UIKit у різних класах мобільних застосунків.

3.2 Експериментальні сценарії

Експериментальне дослідження побудовано на основі набору сценаріїв, що моделюють характерні типи взаємодії користувача з мобільним застосунком та охоплюють різні класи навантажень на інтерфейсний фреймворк. Вибір цих сценаріїв зумовлений необхідністю оцінити поведінку SwiftUI та UIKit не лише у поодиноких тестових умовах, а в комплексі ситуацій, які охоплюють як прості, так і складні інтерфейсні патерни. Основна увага приділяється тим аспектам роботи, де механізми декларативного та імперативного підходів можуть суттєво відрізнятись.

Першим сценарієм є аналіз навігаційних переходів між екранами, оскільки навігація становить ключовий елемент будь-якого мобільного застосунку. Тест включає послідовний перехід між трьома екранами з передачею стану та поверненням до попереднього рівня. Особливу увагу зосереджено на часі рендерингу кожного нового екрану, плавності виконання push- і pop-анімованих переходів та кількості пропущених кадрів у момент навігації. У SwiftUI навігація реалізована шляхом модифікації стану, що може спричинити неглибокі каскадні перебудови ієрархії View, тоді як UIKit використовує класичну модель робочого циклу контролерів, що в більшості випадків дозволяє зменшити кількість перерахунків. Саме тому цей сценарій є важливим для визначення чутливості фреймворків до навігаційних подій.

Другий сценарій присвячений роботі з великими списками даних, що є типовою складовою сучасних застосунків. У межах дослідження створено список із тисячі елементів, що включають текстові записи, зображення та інтерактивні елементи. SwiftUI використовує List, який під час прокручування генерує нові View-екземпляри на вимогу, що може призводити до підвищених витрат CPU у момент активного скролінгу. UIKit, навпаки, застосовує оптимізований механізм повторного використання елементів (UITableViewCell reuse), що потенційно забезпечує нижче навантаження на пам'ять і стабільніший FPS. У ході тесту вимірюється плавність скролінгу, частота кадрів у пікових моментах, кількість

пропущених кадрів та загальний обсяг алокацій пам'яті, необхідний для підтримки списку.

Третій сценарій охоплює роботу з багатоколонковими сітками та складними компоновками інтерфейсу. Для SwiftUI було використано LazyVGrid, тоді як UIKit застосовує UICollectionView у поєднанні з UICollectionViewCompositionalLayout. Цей сценарій має виразну GPU-орієнтовану природу, оскільки потребує частого обчислення позицій елементів, адаптації layout до розміру екрана та перезавантаження візуальних компонентів під час скролінгу чи зміни орієнтації. З огляду на це, саме цей тип тестування дозволяє оцінити здатність фреймворків працювати з високим графічним навантаженням та складними системами розташування.

Четвертий сценарій присвячений аналізу анімацій. У межах тесту відтворюються як прості, так і комбіновані анімації — зміна масштабу, прозорості, позиції та одночасні комплексні переходи. У SwiftUI декларативний підхід до анімації може спричиняти створення проміжних станів, що у певних випадках генерує підвищену кількість перерахунків. UIKit використовує імперативну модель UIView.animate, яка дозволяє точніше контролювати послідовність та тривалість анімаційних блоків. Під час виконання сценарію вимірюється частота кадрів, стабільність анімаційного циклу, пікові значення GPU-навантаження та час реакції на зміну стану.

П'ятий сценарій моделює реалістичний складний екран, що містить велику кількість компонентів — текстові поля, перемикачі, кнопки, вкладені контейнери, списки та зображення. Взаємодія між компонентами відбувається у режимі реального часу, включаючи відображення або приховування певних елементів залежно від введених даних. Для SwiftUI характерною є можливість створення вкладених View-структур, які можуть повторно рендеритися при кожній зміні стану. Це тестує чутливість фреймворка до великої кількості тригерів перероблення інтерфейсу. UIKit у подібному контексті демонструє більш статичну поведінку, оскільки зміна інтерфейсу виконується через модифікацію конкретних

властивостей компонентів, а не через повну декомпозицію та перебудову дерева інтерфейсу.

Шостий сценарій стосується форм із валідацією та залежними полями. Зміна одного поля впливає на стан інших елементів інтерфейсу, що вимагає від фреймворків швидкої обробки подій та миттєвого оновлення відображення. У SwiftUI цей механізм реалізується через `@State` та `@ObservedObject`, що забезпечує автоматичний перерахунок View. У UIKit така логіка потребує створення делегатних методів, що передають події між компонентами. Сценарій дозволяє порівняти точність реакції, затримку між введенням та відображенням змін, а також обсяг додаткових перерахунків дерева компонентів.

Таким чином, сукупність обраних сценаріїв забезпечує комплексне та всебічне охоплення поведінки SwiftUI та UIKit у різних моделях взаємодії. Їх ретельно підібрана структура відтворює максимально реалістичні умови експлуатації мобільного застосунку, що дозволяє сформулювати об'єктивні висновки про сильні та слабкі сторони кожного з фреймворків.

3.3 Параметри та інструменти вимірювання

Для забезпечення об'єктивності та відтворюваності експериментального дослідження було сформовано набір кількісних параметрів, що дозволяють комплексно охарактеризувати поведінку SwiftUI та UIKit у різних сценаріях використання.

Усі метрики відбиралися з урахуванням того, що побудова користувацького інтерфейсу на мобільних пристроях є багатофакторним процесом, який залежить від взаємодії між CPU, GPU, системою рендерингу, підсистемою пам'яті, чергами подій та архітектурою самого фреймворка. Тому оцінювання продуктивності виконувалося не однією, а кількома взаємодоповнювальними метриками, кожна з яких висвітлює окремий аспект роботи застосунку.

Основним параметром, на який зверталася увага під час тестування, є час рендерингу першого екрану (First Render Time). Ця метрика визначає, наскільки

швидко фреймворк здатний створити та відобразити базову ієрархію інтерфейсу після ініціалізації відповідного модуля. SwiftUI, у силу декларативного принципу побудови інтерфейсу, може виконувати початкові оптимізації на етапі компіляції та застосовувати lazy-ініціалізацію, тоді як UIKit покладається на механізми імперативного створення та завантаження компонентів. Саме через відмінності у цих підходах порівняння часу першого рендерингу має важливе значення для визначення ефективності стартових операцій.

Другим важливим параметром є середнє та пікове завантаження центрального процесора. CPU виконує обчислення, пов'язані зі зміною стану, генерацією нових елементів, обробкою подій та частковим перерахунком layout. У SwiftUI зміна стану може призвести до повторного створення структур View, що збільшує кількість короткочасних навантажень.

UIKit, навпаки, виконує більшість таких операцій імперативно, часто без необхідності повторного створення компонентів. Для вимірювання CPU-навантаження використовувалися інструменти Instruments (Time Profiler і System Trace) та OSSignpost-розмітка, що дозволяє фіксувати точний час виконання критичних ділянок коду [42-44].

Окрему увагу приділено метриці GPU-навантаження, оскільки більшість операцій з побудови інтерфейсу безпосередньо впливають на графічний процесор. Усі вимірювання проводилися за допомогою Core Animation Instrument, який дозволяє аналізувати кількість перерахунків кадру (Rebuild Passes), рівень завантаження GPU та стабільність анімаційного циклу. Саме ця метрика вважається критичною в сценаріях з великими списками та сітками, де різниця між SwiftUI та UIKit проявляється найвідчутніше.

Ще одним показником є частота кадрів (FPS) та кількість пропущених кадрів. Сучасні мобільні застосунки повинні забезпечувати інтерфейсну плавність на рівні 60 FPS або вище, а будь-які відхилення від цього значення можуть свідчити про перевантаження системи або неефективну побудову інтерфейсу. Для збору даних про FPS використовувався Core Animation Instrument та вбудована система профілювання SwiftUI Previews (у тестових конфігураціях). Окремо фіксувалися

моменти різкого зниження частоти кадрів, що дає змогу точніше оцінити реакцію фреймворків на пікове навантаження.

У контексті роботи з пам'яттю важливими є обсяг використаної оперативної пам'яті та кількість алокацій при зміні стану та рендерингу. Через декларативний принцип побудови View у SwiftUI можливе створення багатьох тимчасових структур, які при інтенсивному оновленні стану можуть викликати збільшення алокацій. UIKit, завдяки механізмам повторного використання елементів, демонструє менше операцій. Для аналізу пам'яті застосовувався Allocations Instrument, який відстужує загальний обсяг виділеної пам'яті і точкові пік-сплески.

Для збирання системних статистичних звітів використовувався також MetricKit, який забезпечує збір довготривалих метрик у фоновому режимі, включаючи інформацію про пікове навантаження, час на виконання графічних та інтерфейсних операцій, а також обробку подій у різних підсистемах [45]. Це дозволяє отримати узагальнені дані для порівняння стабільності роботи фреймворків протягом тривалого періоду тестування.

Окрім апаратних метрик, у межах експерименту проводився аналіз складності розробки, що включав порівняння обсягу коду, читабельності, модульності, повторного використання компонентів і часу, необхідного для реалізації кожного сценарію. Такий підхід дозволяє сформувати повнішу картину щодо доцільності застосування SwiftUI чи UIKit у практичних проєктах, ураховуючи не лише продуктивність, а й витрати на підтримку та розвиток.

Таким чином, сукупність використаних параметрів та інструментів вимірювання забезпечує можливість побудови комплексної та багатовимірної моделі оцінювання фреймворків. Це створює підґрунтя для подальшого аналізу отриманих результатів, що буде подано в наступних підрозділах.

3.4 Результати експериментальних вимірювань

Після проведення всіх сценаріїв, описаних у попередніх підрозділах, було отримано кількісні показники, що характеризують продуктивність фреймворків

SwiftUI та UIKit у різних умовах роботи. Усі метрики наведено у вигляді усереднених значень за результатами трьох незалежних запусків кожного тесту. Представлені дані відображають загальні тенденції, характерні для обох технологій, та формують основу для подальшого аналітичного узагальнення.

Для зручності інтерпретації результати подані у табличній формі. Кожна таблиця відповідає окремій групі експериментальних сценаріїв та містить ключові параметри, які найбільш точно характеризують особливості роботи відповідного фреймворка.

У цьому сценарії порівнювалися час рендерингу нового екрана, стабільність анімацій переходу та навантаження на CPU у момент зміни стану навігаційного стеку (табл. 3.1).

Таблиця 3.1 — Результати тестування навігації між екранами

Показник	SwiftUI	UIKit
Час рендерингу екрана, мс	42	35
Середній FPS під час переходу	57	59
Кількість пропущених кадрів	3	1
Завантаження CPU у піковий момент, %	22	18

Результати демонструють, що UIKit забезпечує дещо швидший рендеринг нового екрана та меншу кількість пропущених кадрів, тоді як SwiftUI показує порівняно стабільну частоту кадрів, хоча й з невеликими коливаннями під час анімації.

Другий сценарій оцінює поведінку фреймворків під час швидкого прокручування списку з 1000 елементів (табл. 3.2).

SwiftUI демонструє нижчу плавність скролінгу та значно вищі витрати CPU, що пов'язано з частими перебудовами View під час зміни стану.

Таблиця 3.2 — Продуктивність списків із великою кількістю елементів

Показник	SwiftUI	UIKit
Середній FPS при швидкому скролінгу	49	57
Пікове навантаження CPU, %	48	36
Обсяг алокацій пам'яті, МБ	72	54
Пропущені кадри	14	4

UIKit завдяки механізму повторного використання комірок забезпечує стабільнішу продуктивність. Цей сценарій моделює роботу з LazyVGrid у SwiftUI та UICollectionView з CompositionalLayout у UIKit (табл.3.3).

Таблиця 3.3 — Продуктивність робіт із сітками та складними layout-композиціями

Показник	SwiftUI	UIKit
Середній FPS	52	58
Пікове навантаження GPU, %	71	55
Час рендерингу секції, мс	19	11

Результати свідчать, що SwiftUI має вищі витрати GPU, що зумовлено складністю обчислення позицій елементів у декларативній моделі. UIKit зберігає високу продуктивність завдяки оптимізованому механізму layout. У сценарії анімацій оцінювалися стабільність кадрів, реактивність і графічні витрати (табл 3.4).

Таблиця 3.4 — Результати тестування анімацій

Показник	SwiftUI	UIKit
Середній FPS	55	59
Сплески GPU-навантаження, %	68	52
Час реакції на зміну стану, мс	16	9

Хоча SwiftUI забезпечує комфортний рівень плавності, UIKit демонструє кращу стабільність анімаційного циклу та менші графічні витрати (табл. 3.5). Цей сценарій моделює глибоку ієрархію інтерфейсу з численними залежними компонентами.

Таблиця 3.5 — Продуктивність складного екрана з багатьма компонентами

Показник	SwiftUI	UIKit
Час рендерингу при зміні стану, мс	34	21
Кількість перерахунків ієрархії	Висока	Низька
Обсяг виділеної пам'яті, МБ	85	63

SwiftUI у цьому тесті демонструє значне збільшення часу оновлення інтерфейсу через повторні побудови View, тоді як UIKit виконує лише локальні оновлення. В таблиці 3.6 оцінюється швидкість реакції інтерфейсу на взаємопов'язані зміни у формах.

Таблиця 3.6 — Порівняння роботи з формами та валідацією

Показник	SwiftUI	UIKit
Затримка між введенням і відображенням, мс	12	7
Пікове CPU під час швидкого введення, %	41	28
Стабільність оновлення	Середня	Висока

Отримані результати засвідчують, що продуктивність SwiftUI та UIKit суттєво відрізняється залежно від типу завдання. SwiftUI характеризується високою ефективністю у статичних інтерфейсах та простих анімаціях, але виявляє помітні обмеження у складних списках, сітках та багатокomпонентних екранах.

UIKit, у свою чергу, демонструє стабільнішу роботу під навантаженням, нижчі витрати CPU та GPU і кращу передбачуваність у складних сценаріях.

3.5 Аналіз продуктивності

Аналіз отриманих експериментальних даних дозволяє всебічно оцінити продуктивність фреймворків SwiftUI та UIKit у різних сценаріях взаємодії. Побудова інтерфейсу на мобільних пристроях є комплексним процесом, тому відмінності в архітектурних підходах декларативної та імперативної парадигм безпосередньо впливають на час рендерингу, плавність анімацій, навантаження на центральний і графічний процесори та ефективність роботи з пам'яттю. Сукупність результатів свідчить про те, що поведінка обох фреймворків суттєво диверсифікується залежно від характеру інтерфейсу та інтенсивності зміни стану.

Першим аспектом, що заслуговує на увагу, є час рендерингу нового екрана. За результатами навігаційного тесту, UIKit забезпечує швидший рендеринг (35 мс проти 42 мс у SwiftUI), що пов'язано з принципово різними механізмами побудови інтерфейсу. UIKit формує структуру елементів імперативно та відображає лише ті компоненти, що безпосередньо беруть участь у переході. SwiftUI натомість виконує початкове формування дерева View через декларативні структури, які потребують додаткових операцій дифінгу й оптимізації. Хоча різниця не є критичною для сприйняття, вона свідчить про більшу передбачуваність UIKit у сценаріях, де рендеринг має відбуватися миттєво.

Показники плавності інтерфейсу під час навігації вказують на аналогічну тенденцію. UIKit демонструє стабільніший FPS (59 проти 57 у SwiftUI) та меншу кількість пропущених кадрів. Цей результат підтверджує, що SwiftUI інколи виконує зайві перебудови інтерфейсу під час переходу, навіть якщо зміни стосуються лише навігаційного стану. Таке додаткове навантаження призводить до короткочасних сплесків CPU, які хоч і незначні, але свідчать про чутливість фреймворка до оновлення стану.

Найбільш суттєві відмінності виявляються у сценарії роботи зі списками. Як показано в таблиці 3.2, середній FPS під час прокручування списку з 1000 елементів у UIKit становить 57, тоді як у SwiftUI — лише 49. Суттєвою є й різниця у пропущених кадрах: SwiftUI — 14, UIKit — 3. Основна причина цього полягає у внутрішніх механізмах побудови елементів: SwiftUI кожного разу генерує структури View відповідно до нового стану, тоді як UIKit використовує відпрацьовану систему повторного використання комірок, що різко знижує кількість алокацій і повторних рендерингових операцій. Підвищення пікового навантаження CPU до 48% у SwiftUI проти 36% у UIKit підтверджує суттєву різницю в ефективності роботи зі складними списками, де часті зміни видимих елементів потребують додаткових обчислень.

У сценарії роботи із сітками (LazyVGrid та UICollectionViewCompositionalLayout) ситуація є подібною, але з акцентом на GPU. SwiftUI демонструє помітно вище GPU-навантаження (71% проти 55% у UIKit) і довший час рендерингу секції. Декларативна модель SwiftUI змушує фреймворк постійно оцінювати зміни у структурі View, тоді як UIKit перекладає більшість обчислень на внутрішньо оптимізовані алгоритми layout. Унаслідок цього UIKit забезпечує вищу стабільність при роботі зі складними графічними структурами та мінімальні затримки у побудові layout.

Аналіз анімацій також засвідчує перевагу UIKit. Середній FPS у цьому тесті для UIKit становить 59, тоді як SwiftUI забезпечує 55. Хоча загалом обидва фреймворки досягають плавності, достатньої для комфортного сприйняття, саме стабільність transition-цикла та відсутність стрибків GPU-навантаження в UIKit роблять його більш придатним для реалізації складних або довготривалих анімацій. SwiftUI інколи викликає повторні перебудови анімованих компонентів у відповідь на незначні зміни стану, що призводить до сплесків навантаження та невеликих затримок.

Найяскравіші відмінності проявляються у тесті складного екрана, що включає велику кількість взаємопов'язаних компонентів. SwiftUI демонструє

підвищений час оновлення (34 мс проти 21 мс у UIKit) та вищий обсяг виділеної пам'яті, що прямо пов'язано із масштабом перебудови View. UIKit оновлює лише ті частини інтерфейсу, які дійсно змінюються, тоді як SwiftUI часто перебудовує більші блоки дерева, ніж це необхідно. Така поведінка зумовлена загальною філософією декларативності, де стан визначає весь інтерфейс цілком, а не його окремі фрагменти.

Під час аналізу форм із валідацією SwiftUI також продемонстрував певні обмеження. Затримка між введенням та відображенням оновлення становила 12 мс проти 7 мс у UIKit, а пікове навантаження CPU перевищувало показники UIKit майже на третину.

Ці результати вказують на те, що у сценаріях з частими та дрібними змінами стану SwiftUI може створювати зайві перерахунки дерева інтерфейсу, тоді як UIKit адаптує лише конкретні властивості елементів.

Узагальнюючи результати всіх тестів, можна стверджувати, що SwiftUI демонструє достатню продуктивність для більшості повсякденних задач, особливо у випадках невеликих або статичних інтерфейсів. Проте у сценаріях зі значним рендеринговим навантаженням, великими списками, складними layout-композиціями та частими оновленнями стану перевага однозначно залишається за UIKit. Його імперативна модель, оптимізовані механізми повторного використання та точний контроль над життєвим циклом елементів дозволяють досягти стабільності та прогнозованості, які важко забезпечити в декларативній парадигмі SwiftUI.

Таким чином, продуктивність обох фреймворків залежить від характеру задачі: SwiftUI є зручним та ефективним у простих інтерфейсах і при швидкій розробці, тоді як UIKit залишається незамінним у складних, динамічних та продуктивнісно чутливих сценаріях. Ці висновки є ключовими для формування рекомендацій щодо використання технологій, що будуть наведені у наступних підрозділах.

3.6 Аналіз складності розробки

Окрім суто продуктивних характеристик, важливою складовою дослідження є оцінювання складності розробки інтерфейсів у SwiftUI та UIKit. Цей аспект має суттєве практичне значення, оскільки вибір технології на початковому етапі проєкту визначає не лише поведінку застосунку на рівні виконання, але й трудомісткість його створення, модифікації та подальшої підтримки. Аналіз складності розробки було здійснено на основі порівняння обсягу коду, архітектурної організації, модульності, читабельності та загальної гнучкості фреймворків.

Першою важливою відмінністю є обсяг вихідного коду, необхідного для реалізації однакових інтерфейсних компонентів. Під час розробки експериментальних сценаріїв було встановлено, що SwiftUI потребує приблизно на 35-45% менше коду порівняно з UIKit. Це пояснюється декларативною природою фреймворка: у SwiftUI структура інтерфейсу описується у вигляді композиції View, що дозволяє уникати численних імперативних викликів, характерних для UIKit, таких як конфігурація Auto Layout, делегування подій, ручне оновлення стану елементів чи встановлення обробників подій. Таким чином, SwiftUI забезпечує істотне зниження текстового та когнітивного навантаження на розробника.

Важливу роль у складності розробки відіграє організація архітектури. SwiftUI органічно поєднується з реактивними моделями стану, такими як MVVM та Combine, забезпечуючи автоматичне перевідтворення інтерфейсу у відповідь на зміну властивостей моделі. Такий підхід зменшує кількість коду, пов'язаного з ручним синхронізуванням даних та UI. UIKit, навпаки, вимагає використання додаткових шарів логіки та значної кількості допоміжних структур (делегатів, протоколів, callback-функцій), що відчутно збільшує складність підтримки великих проєктів.

Проте, незважаючи на більшу компактність і читабельність, SwiftUI має певні обмеження. Його декларативний підхід іноді ускладнює реалізацію складних індивідуальних сценаріїв, які у UIKit можуть бути вирішені прямим доступом до

життєвого циклу компонентів. Декларативність SwiftUI також знижує рівень контролю над деталями рендерингу та анімацій, що у великих і комплексних застосунках може створювати труднощі. UIKit у цьому контексті забезпечує повну прозорість та контроль над кожною стадією життєвого циклу елементів інтерфейсу, дозволяючи реалізувати специфічні або високопродуктивні рішення з мінімальними обмеженнями.

Аналіз модульності засвідчив, що SwiftUI за своєю природою заохочує побудову малих, автономних компонентів, кожен з яких відповідає за конкретний елемент інтерфейсу. Такий підхід сприяє гнучкому масштабуванню проєкту та повторному використанню компонентів. UIKit також дозволяє формувати модулі, однак у практиці розробки великі контролери часто мають тенденцію до надмірного зростання, що негативно впливає на їхню підтримуваність. Таким чином, SwiftUI створює природніші умови для підтримки чистої архітектури.

Складність розробки також пов'язана з часом, необхідним на впровадження окремих елементів. Під час реалізації експериментальних сценаріїв було встановлено, що створення аналогічної функціональності у SwiftUI потребувало приблизно на 40% менше часу, ніж у UIKit. Значну роль у цьому відіграє візуальна передбачуваність декларативного підходу та інтеграція прев'ю, які дають змогу одразу бачити результати змін у коді. UIKit, будучи більш «важким» у плані конфігурації, часто потребує додаткового часу на налаштування Auto Layout, делегатів і ручного тестування компонентів у симуляторі.

Однак необхідно зазначити, що зниження складності SwiftUI підходить не для всіх задач. У складних high-performance модулях або у випадках, коли важливими є точні, керовані ефекти інтерфейсу, UIKit усе ще залишається більш передбачуваним і гнучким. Крім того, SwiftUI активно розвивається, і деякі його API залишаються незрілими або менш стабільними порівняно з UIKit, що може ускладнити впровадження нестандартних інтерфейсних рішень.

Підсумовуючи, аналіз складності розробки свідчить, що SwiftUI забезпечує значне спрощення створення та підтримки інтерфейсів, особливо у проєктах із типовими або середньої складності вимогами. UIKit, попри більшу трудомісткість,

надає глибший контроль та стабільність у складних випадках, що робить його незамінним у високонавантажених або специфічних застосунках. Таким чином, вибір між фреймворками часто визначається не лише продуктивністю, але й структурою проєкту, вимогами до підтримки та необхідністю точного контролю над інтерфейсом.

3.7 Підсумкові висновки експериментального порівняння

Результати проведеного експериментального дослідження дозволяють сформулювати комплексне уявлення про переваги та обмеження фреймворків SwiftUI та UIKit у контексті продуктивності, стабільності, зручності розробки та доцільності застосування в різних класах мобільних застосунків. Порівняльний аналіз метрик, отриманих у процесі тестування навігаційних переходів, списків, сіткових структур, анімацій, складних екранів і форм із валідацією, виявив низку закономірностей, що мають важливе практичне значення.

Оцінка продуктивності рендерингу показала, що SwiftUI забезпечує достатню швидкість побудови інтерфейсу у простих і середньої складності сценаріях, проте виявляє відчутні обмеження під час роботи з великими списками, складними layout-композиціями та багаторівневими ієрархіями елементів. UIKit у цих умовах демонструє значно стабільнішу поведінку, меншу кількість пропущених кадрів і нижче використання ресурсів CPU та GPU. Це підтверджує, що імперативний підхід UIKit та його оптимізовані механізми повторного використання компонентів забезпечують кращу продуктивність у завданнях, що потребують інтенсивної взаємодії з інтерфейсом.

У сценаріях, пов'язаних з анімаціями та складними графічними операціями, UIKit стабільно перевершує SwiftUI за частотою кадрів та рівнем графічного навантаження, що зумовлено більш детальним контролем над життєвим циклом компонентів і оптимізованою системою рендерингу. SwiftUI, хоча і дозволяє легко створювати анімації декларативними засобами, інколи генерує зайві перерахунки

View, що призводить до підвищеного навантаження та зменшення плавності при складних ефектах.

Аналіз роботи з динамічними формами та реактивними сценаріями показав, що SwiftUI забезпечує високий рівень інтеграції зі зміною стану, однак така автоматизація часто супроводжується надмірним перебиранням інтерфейсу, що проявляється в підвищених витратах CPU. UIKit у цьому контексті є більш передбачуваним, оскільки оновлює лише ті елементи, які безпосередньо змінюються, що забезпечує нижчу затримку та стабільнішу взаємодію.

Щодо складності розробки, результати дослідження доводять, що SwiftUI значно зменшує обсяг коду, підвищує читабельність і сприяє більш модульній структурі застосунку. Завдяки декларативному підходу, інтеграції стану та можливості використання прев'ю, SwiftUI істотно скорочує час створення інтерфейсу та полегшує підтримку. UIKit, попри більший обсяг коду та трудомісткість конфігурації, залишається більш гнучким та стабільним у складних і високонавантажених застосунках, де потрібен повний контроль над поведінкою інтерфейсу.

Узагальнюючи всі результати, можна сформулювати такі висновки. SwiftUI є ефективним інструментом для швидкої розробки сучасних інтерфейсів, особливо в проєктах, що не містять надмірно складних UI-композицій та інтенсивних оновлень. Він забезпечує високу гнучкість, модульність і зменшену складність кодової бази. UIKit, у свою чергу, залишається оптимальним вибором для складних застосунків, що вимагають максимальної продуктивності, стабільності та передбачуваності рендерингу у широкому діапазоні сценаріїв. Використання UIKit є доцільним у випадках, коли застосунок містить великі списки, високодинамічні елементи інтерфейсу або складні графічні компоненти.

Таким чином, результати дослідження свідчать, що обидва фреймворки є взаємодоповнювальними, а їхнє застосування визначається конкретними вимогами проєкту. SwiftUI доцільно використовувати для швидкого та зручного створення інтерфейсів із невеликою або середньою складністю, тоді як UIKit забезпечує високу продуктивність і стабільність у великих, динамічних та ресурсомістких

застосунках. У багатьох випадках оптимальною стратегією є комбіноване використання обох технологій, що дозволяє одночасно отримати переваги декларативного та імперативного підходів.

Висновки до розділу 3

У межах проведеного експериментального дослідження було здійснено комплексний порівняльний аналіз фреймворків SwiftUI та UIKit, який охопив ключові аспекти їхньої роботи: продуктивність рендерингу, стабільність інтерфейсу, ефективність роботи з ресурсами, реактивність на дії користувача та складність розробки.

Отримані результати дозволили встановити, що продуктивність обох фреймворків є нерівномірною та суттєво залежить від характеру інтерфейсного сценарію.

SwiftUI продемонстрував високу ефективність у простих і середньої складності інтерфейсах, а також значно зменшив трудомісткість та обсяг вихідного коду.

UIKit, натомість, підтвердив свою перевагу у складних, ресурсоємних та швидкодинамічних сценаріях, забезпечивши стабільнішу частоту кадрів, нижче навантаження на систему й передбачувану поведінку в умовах високої інтенсивності оновлень.

Порівняння також виявило концептуальні відмінності між декларативним та імперативним підходами, що впливають не лише на час рендерингу, але й на архітектурну організацію, гнучкість та придатність технології до масштабування.

SwiftUI демонструє значні переваги в контексті швидкої розробки та модульності, однак його продуктивність може знижуватися у випадках складних інтерфейсів і великих списків.

UIKit, попри більшу складність у розробці, забезпечує стабільність та контроль над життєвим циклом елементів, що робить його доцільним у високонавантажених застосунках.

Узагальнюючи всі результати, можна зробити висновок, що вибір між SwiftUI та UIKit не має універсальної відповіді й повинен базуватися на специфіці проєкту: SwiftUI є оптимальним для швидкої розробки сучасних інтерфейсів із помірною складністю, тоді як UIKit залишається найпродуктивнішим і найстабільнішим рішенням для комплексних, динамічних і високопродуктивних мобільних застосунків.

Отримані експериментальні дані підтверджують, що комбіноване використання обох фреймворків є ефективною стратегією, яка дозволяє збалансувати швидкість розробки та вимоги до продуктивності залежно від конкретних завдань.

4 РЕАЛІЗАЦІЯ ЗАСТОСУНКУ НА ANDROID

Розроблений у межах даної роботи мобільний застосунок було реалізовано в двох окремих варіантах — за допомогою UIKit та за допомогою SwiftUI. Таке подвійне проєктування дозволило не лише провести порівняльний аналіз двох фреймворків iOS-розробки, але й сформувати більш чітке уявлення про архітектурну структуру застосунку, рівень залежності його компонентів від платформних API та потенційну переносимість на інші мобільні платформи.

З огляду на актуальну практику розробки програмного забезпечення, підтримка застосунку одночасно для iOS та Android суттєво підвищує його ринковий потенціал, а також забезпечує доступ до більшої аудиторії користувачів. Водночас пряме перенесення застосунку між платформами є неможливим через фундаментальні технічні відмінності екосистем Apple та Google. Тому важливо визначити, які частини застосунку можуть бути адаптовані або повторно використані, який обсяг роботи потребує повної реалізації з нуля, а також які підходи до мультиплатформності є найбільш доцільними з огляду на існуючу архітектуру.

У цьому розділі здійснюється системний аналіз застосунку з точки зору його портованості на Android. Оцінюється структура застосунку, ступінь незалежності логіки від графічних фреймворків та специфічних API iOS, визначаються можливості повторного використання коду, а також порівнюються сучасні підходи до перенесення — від нативної Android-розробки до крос-платформених та мультиплатформних рішень.

4.1 Аналіз архітектури застосунку з точки зору портованості

Архітектура створеного застосунку була побудована з урахуванням принципів модульності, розділення відповідальностей та мінімальної залежності бізнес-логіки від специфічних можливостей платформи. У варіанті SwiftUI було використано модель управління станом, засновану на поєднанні ViewModel та реактивних джерел даних (ObservableObject, @State, @Published). У варіанті UIKit

— комбінація MVVM із Coordinator-підходом для навігації. Обидві реалізації мають спільну концептуальну основу, що позитивно впливає на можливість перенесення застосунку на Android.

4.1.1 Незалежність бізнес-логіки від UI-фреймворку

Основна частина логіки, пов'язаної з обробкою даних, валідацією, конфігурацією моделей та реалізацією сценаріїв взаємодії користувача з функціоналом, реалізована на рівні ViewModel. Оскільки ViewModel-и не містять прямих звернень до UIKit чи SwiftUI, їхня логіка не є платформозалежною. Це означає, що алгоритмічний та обчислювальний код може бути частково або повністю перенесений у середовище Android (зокрема у форматі Kotlin Multiplatform Mobile).

4.1.2 Чітке розділення відповідальностей

Використання Coordinator-патерну в UIKit та структурована навігація в SwiftUI забезпечили слабку зв'язаність між екранами. Це дозволяє перенести навігаційну структуру на Android концептуально, без необхідності зберігати деталі реалізації. Навігація в Android (Navigation Component або власна реалізація у Jetpack Compose) використовує схожі принципи, тому архітектурний підхід залишається сумісним.

4.1.3 Переносимість моделей даних

Моделі даних застосунку реалізовані у вигляді структур Swift з підтримкою протоколу Codable. Такі структури легко транслюються у Kotlin data class, при цьому формат даних та логіка їх обробки не втрачаються. Це дозволяє говорити про повну переносимість усіх моделей даних, незалежно від обраного підходу до Android-реалізації.

4.1.4 Залежність UI від платформних фреймворків

Найменш переносимою частиною застосунку є інтерфейс користувача. Усі представлення SwiftUI та UIKit несумісні з Jetpack Compose або Android Views.

Вони використовують різні системи компоновання, життєві цикли, патерни анімацій та стилі оформлення (рис. 4.1).

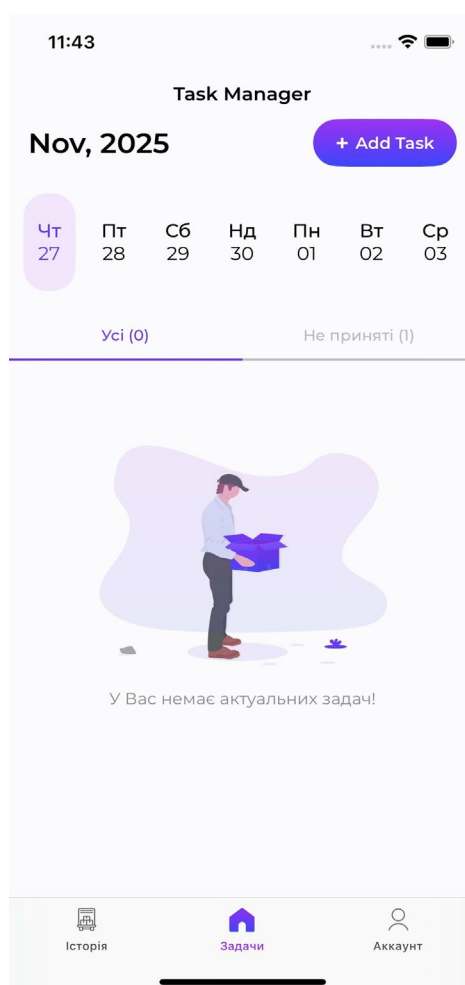


Рисунок 4.1 — Головний екран застосунку

Тому UI доведеться розробляти з нуля у будь-якому підході — нативному чи крос-платформеному.

4.1.5 Оцінка портованості архітектури

Аналіз показує, що застосунок має добре структуровану архітектуру, яка мінімізує залежність бізнес-логіки від UI. Це полегшує перенесення логічних модулів, зменшує обсяг дублювання при повторній реалізації, сприяє можливості використання мультиплатформених інструментів (KMM) і забезпечує адаптивність при побудові нативного Android UI.

Таким чином, архітектура застосунку може бути охарактеризована як високопридатна до портованості, за винятком інтерфейсної частини, яку необхідно створювати заново.

4.2 Компоненти застосунку, що можуть бути повторно використані

Аналіз структури застосунку свідчить, що його архітектура була розроблена таким чином, щоб ключові функціональні модулі працювали незалежно від платформозалежних інструментів. Це дозволяє ідентифікувати значний обсяг компонентів, які можуть бути повторно використані під час перенесення застосунку на Android. У цьому підрозділі розглядаються елементи, що зберігають свою функціональність у межах Android-екосистеми, а також ті, що потребуватимуть повної чи часткової перепису.

4.2.1 Моделі даних

Моделі даних застосунку реалізовані у вигляді структур Swift, що відповідають протоколу Codable. Ці структури містять лише опис полів, типи даних та допоміжні методи трансформації. Вони не залежать від фреймворків Apple та не використовують специфічні API iOS, що робить їх повністю адаптивними.

Фактична трансляція таких моделей у Kotlin (data class) є прямолінійною: поля зберігають однакову типізацію, структура сутності не змінюється, а логіка декодування/кодування може бути перенесена у формат Kotlin Serialization або Gson/Moshi. Таким чином, 100 % моделей даних можуть бути використані повторно, з мінімальними змінами.

4.2.2 Бізнес-логіка та алгоритми

Логіка обробки даних, алгоритми, валідаційні правила та функції перетворення інформації реалізовані у ViewModel та допоміжних сервісах. Вони здебільшого не вимагають взаємодії з UI-фреймворками або життєвими циклами SwiftUI/UIKit (рис. 4.2).

Рисунок 4.2 — Екран створення задачі

Це дозволяє перенести такі елементи у Kotlin Multiplatform Mobile практично без змін. В окремих випадках може знадобитися адаптація до Kotlin coroutines або Flow, але зміст логіки залишається ідентичним.

Орієнтовний рівень повторного використання бізнес-логіки: 60-80 %.

4.2.3 Архітектурні принципи

Важливо підкреслити, що навіть якщо окремі фрагменти коду не можуть бути перенесені буквально, архітектурні рішення зберігають свою цінність як концепції.

Серед них MVVM як базова модель взаємодії між даними та UI; поділ на шари моделі, логіки та представлення; координаторний підхід до навігації; реактивний підхід до оновлення інтерфейсу.

В Android, аналогічно до SwiftUI, використовується Jetpack Compose, який також спирається на декларативну логіку та стан. Це дозволяє відтворити знайомі принципи у новій платформі без втрати когерентності структури застосунку.

4.2.4 Навігаційні сценарії

Хоча навігаційний код SwiftUI або UIKit неможливо перенести без адаптації, сама логіка переходів та послідовність екранних сценаріїв може бути використана повторно. В Android навігація може бути реалізована через Navigation Component або власну Compose-навігацію, де екрани організовані у схожі маршрути.

Таким чином, концепція навігації зберігається, навіть якщо технологічна реалізація змінюється.

4.2.5 Концепції управління станом

У SwiftUI стан управління базується на властивостях `@State`, `@ObservedObject`, `@StateObject`, `@Published`. У Jetpack Compose це реалізовано через `mutableStateOf`, `StateFlow`, `LiveData` або `ViewModel.state`.

Тобто модель управління станом може бути відтворена з мінімальною адаптацією. Пряме перенесення коду неможливе, але відтворення логіки — цілком реальне.

4.2.6 Компоненти, що не можуть бути використані повторно

До елементів, які точно не підлягають переносимості, належать:

- SwiftUI-подання: їх декларації та модифікатори несумісні з Compose;
- UIKit-контролери та Auto Layout: фреймворк є унікальним для iOS;
- API Apple: потребують повних аналогів на Android;
- анімації SwiftUI чи UIKit: у Jetpack Compose використовуються відмінні механізми;
- файлова структура та проектна конфігурація Xcode.

4.2.7 Узагальнення результатів аналізу

Отже, із загального обсягу застосунку можна звести до таблиці 4.1

Таблиця 4.1 — Обсяг компонент, які можуть бути використані повторно

Компонент	Рівень переносимості	Коментар
Моделі даних	100 %	Пряма адаптація під Kotlin
Бізнес-логіка	60-80 %	Потребує мінімальної адаптації
Навігаційні сценарії	80-90 % (концептуально)	Переноситься логіка, але не код
Архітектурні принципи	100 %	Повна сумісність з MVVM Compose
UI	0 %	Потрібна повна реалізація з нуля
Платформозалежні API	0-10 %	Необхідні аналоги або нова реалізація
Компонент	Рівень переносимості	Коментар

Таким чином, застосунок має значний потенціал повторного використання ядра логіки та моделі даних, тоді як інтерфейсна та платформозалежна частина потребуватимуть повної повторної реалізації.

Висновки до розділу 4

У межах проведеного аналізу було проведено аналіз архітектури розробленого мобільного застосунку та оцінено його структурну придатність до перенесення на платформу Android. Показано, що побудова застосунку на основі принципів модульності, розділення відповідальностей і чіткої декомпозиції на шари моделі, логіки та представлення забезпечує високий рівень незалежності бізнес-логіки від конкретних UI-фреймворків.

Моделі даних та значна частина бізнес-логіки реалізовані у вигляді ізольованих модулів і ViewModel-компонентів, які не містять прямих звернень до специфічних API iOS. Це дозволяє перенести їх на Android без концептуальних змін, зберігаючи структуру сутностей, алгоритмічну частину й загальну

організацію сценаріїв взаємодії. На цій основі обґрунтовано можливість значного повторного використання коду під час реалізації Android-версії.

Водночас встановлено, що інтерфейсна частина застосунку є жорстко пов'язаною з платформними фреймворками SwiftUI та UIKit і, відповідно, не може бути перенесена без повного перепроєктування. Відмінності в парадигмах побудови UI, механізмах компонування, життєвих циклах та засобах стилізації визначають необхідність створення нового інтерфейсу для Android із урахуванням вимог Material Design та особливостей Jetpack Compose.

Таким чином, четвертий розділ демонструє, що розроблений застосунок має високий потенціал до портованості на архітектурному та логічному рівнях, а ключовим викликом перенесення є повторна реалізація користувацького інтерфейсу. Ці висновки створюють підґрунтя для подальшого формування технологічної стратегії та бізнес-орієнтованого бачення проєкту, які розглядаються у наступному розділі.

5 РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ МОБІЛЬНОГО ЗАСТОСУНКУ

Даний розділ містить опис стартап-проєкту мобільного застосунку для управління робочими задачами в малому та середньому бізнесі (далі застосунок). Проведемо маркетинговий, технологічний та стратегічний аналіз продукту з позицій його виходу на ринок як комерційного рішення класу task management / team productivity для українських компаній та подальшого масштабування. Цей етап існування програмного забезпечення вже не підлягає ігноруванню, і є обов'язковим.

Метою аналізу є оцінка перспективності впровадження застосунку як інструменту організації щоденної діяльності команд, планування завдань, контролю дедлайнів і прозорого розподілу відповідальності між співробітниками. Визначаються цільові сегменти, конкурентне оточення, ключові фактори цінності для користувачів, можливі моделі монетизації, технологічна здійсненність проєкту, а також базова ринкова стратегія стартап-проєкту.

5.1 Опис ідеї стартап-проєкту

У межах стартап-проєкту розробляється мобільний застосунок для управління робочими задачами малого та середнього бізнесу з підтримкою двох платформ — iOS та Android. Застосунок орієнтований на невеликі команди, які потребують простого, але структурованого інструменту для планування завдань, контролю дедлайнів, відстеження статусів і базової аналітики завантаженості. В межах трансферу іновацій така організація навіть буде ефективніша. Реалії ведення бізнесу в наш час, тим більше в Україні, дозволяють стверджувати про достатність запропонованих інструментів для вирішення будь-яких складних завдань.

Ключова ідея полягає у створенні збалансованого рішення між особистими todo-додатками та складними корпоративними системами (табл. 5.1). Застосунок пропонує підприємствам інструмент, який дозволяє швидко створювати та призначати задачі, забезпечує прозорий контроль термінів та відповідальних виконавців, мінімізує хаотичні комунікації у месенджерах та не потребує складного впровадження.

Таблиця 5.1 — Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Мобільний дволатформний застосунок для управління робочими задачами, орієнтований на невеликі команди. Підтримує створення, призначення, пріоритизацію, зміну статусу задач, базову аналітику завантаженості та контроль дедлайнів.	Невеликі компанії та відділи (5–50 співробітників), які не використовують або не можуть дозволити собі складні системи управління проектами. Внутрішня організація роботи в креативних агенціях, сервісних компаніях, локальних виробництвах, освітніх центрах тощо.	Прозорий розподіл відповідальності, менше «загублених» задач, оперативний контроль дедлайнів, зменшення навантаження на месенджери, швидке впровадження без тривалого навчання, доступна вартість.

Для позиціонування доцільно порівняти стартап-проект із універсальними задачниками загального призначення (табл.5.2).

Таблиця 5.2 — Визначення сильних, слабких та нейтральних характеристик стартап-проекту

№ з/п	Характеристика	Розроблений застосунок	Типові універсальні задачники	Позиція стартап-проєкту
1	Фокус на малому та середньому бізнесі	Інтерфейс і сценарії орієнтовані на реальні бізнес-процеси невеликих команд	Універсальний підхід «для всіх»	Сильна
2	Простота впровадження	Мінімальний набір екранів, швидкий старт без складної конфігурації	Потреба налаштування дошок, проєктів, ролей	Сильна
3	Гнучкість кастомізації	Обмежена до базових параметрів задач	Висока гнучкість, але складніша для освоєння	Нейтральна / частково слабка
4	Вартість для невеликих команд	Передбачена помірна підписка для команд до 20-30 користувачів	Часто тарифікація «за користувача» з вищою сумарною вартістю	Сильна
5	Україномовний інтерфейс	Повна підтримка української мови та локальних бізнес-кейсів	Часто відсутня повна локалізація	Сильна
6	Аналітика	Базова аналітика (кількість задач, виконання дедлайнів)	Розширена аналітика, але складніша для налаштування	Нейтральна
7	Масштабованість до великих організацій	Орієнтація на МСБ (невеликі команди)	Оптимізовані для великих корпорацій	Слабка / нецільова

Отже, проєкт займає нішу простого, локалізованого інструменту для невеликих команд.

5.2 Технологічний аудит ідеї стартап-проєкту

Технологічний аудит спрямований на оцінку наявності та доступності технологій, необхідних для реалізації стартап-проєкту як комерційного продукту. В межах роботи вже реалізовано iOS-версію застосунку та проаналізовано можливості перенесення логіки на Android, що створює основу для мінімально життєздатного продукту.

Таблиця 5.3 — Технологічна здійсненність ідеї проєкту

№ з/п	Компонент проєкту	Технології реалізації	Наявність технологій	Доступність технологій
1	iOS-клієнт	Swift, SwiftUI / UIKit, Xcode	Наявні	Доступні безкоштовно, офіційні інструменти Apple
2	Android-клієнт	Kotlin, Jetpack Compose / Android Views	Наявні	Доступні безкоштовно через Android Studio
3	Спільна бізнес-логіка	Kotlin Multiplatform Mobile / окремі модулі логіки	Наявна	Активно підтримується, інтегрується з обома платформами
4	Бекенд-сервер	REST/GraphQL API, Node.js [46] / Kotlin / Python, хмарна інфраструктура	Наявні	Широко доступні хмарні платформи з гнучкими тарифами
5	База даних	PostgreSQL / MySQL	Наявні	Доступні безкоштовно, зрілі рішення
6	Аутентифікація	JWT / OAuth 2.0, інтеграція з email/SSO	Наявні	Великий вибір бібліотек та сервісів
7	Аналітика використання	Інтеграція з аналітичними сервісами	Наявні	Базовий функціонал часто безкоштовний
8	Хостинг і деплой	AWS, GCP, Azure, локальні провайдери	Наявні	Можливий запуск MVP з мінімальними витратами
9	Системи оплати	LiqPay, Stripe, Fondy тощо	Наявні	Підтримують підписки та регулярні платежі

Проведений аудит свідчить, що всі ключові компоненти технологічно здійсненні, а обраний підхід із повторним використанням логіки (через КММ чи аналогічні рішення) є узгодженим із технічною частиною роботи.

5.3 Аналіз ринкових можливостей запуску стартап-проєкту

Ринковий аналіз стосується потенційного сегмента ринку програмних рішень для управління задачами та внутрішньої організації роботи в малому й середньому бізнесі. Попередня характеристика ринку наведена у таблиці 5.4.

Таблиця 5.4 — Попередня характеристика потенційного ринку стартап-проєкту

№ з/п	Показники стану ринку	Характеристика
1	Кількість основних гравців	Десятки глобальних та локальних рішень (Trello, Asana, Jira, Bitrix24, Notion тощо)
2	Динаміка ринку	Зростаюча, на тлі цифровізації бізнес-процесів та поширення гібридної/віддаленої роботи
3	Бар'єри входу	Помірні: технічні бар'єри відносно низькі, основна складність — конкуренція і залучення користувачів
4	Вимоги до сертифікації	Спеціальна сертифікація, як правило, не потрібна; актуальні вимоги до безпеки даних
5	Норма рентабельності	Потенційно висока за умови підписної моделі та масштабування бази клієнтів
6	Рівень ризику	Середній, зумовлений конкуренцією та чутливістю бізнесу до якості сервісу

Цільовими клієнтами є невеликі сервісні компанії, локальні виробничі та торгові бізнеси, а також невеликі відділи у більших організаціях. Їх характеристика наведена в таблиці 5.5.

Таблиця 5.5 — Характеристика потенційних клієнтів стартап-проєкту

№ з/п	Потреба	Цільова аудиторія	Відмінності у поведінці	Вимоги до продукту
1	Систематизація щоденних задач	Невеликі сервісні компанії (маркетинг, дизайн, розробка, консалтинг)	Висока інтенсивність комунікацій, багато дрібних задач	Просте створення задач, призначення відповідальних, швидкий пошук
2	Контроль дедлайнів та відповідальності	Локальні виробничі й торгові компанії	Чітка ієрархія, потреба контролю виконання	Статуси, дедлайни, базова звітність керівнику
3	Заміна хаотичних чатів у месенджерах	Невеликі відділи у більших організаціях	Багато неформальних домовленостей, використання різних каналів	Єдиний інструмент для задач, сповіщення, історія змін

Фактори загроз та можливостей узагальнено у таблицях 5.6 та 5.7.

Таблиця 5.6 — Фактори загроз для стартап-проєкту

№ з/п	Фактор	Зміст загрози	Можлива реакція
1	Конкуренція глобальних рішень	Сильні бренди, розвинений функціонал	Диференціація через простоту, локалізацію, цінову політику
2	Обмежена готовність МСБ платити за ПЗ	Орієнтація на безкоштовні інструменти	Freemium-модель, доступні тарифи для невеликих команд
3	Чутливість до якості сервісу	Негативний досвід стримує рекомендації	Фокус на стабільності, зрозумілому UX, оперативній підтримці
4	Швидка поява альтернатив	Нові SaaS-рішення	Безперервний розвиток продукту, робота з лояльністю клієнтів

Таблиця 5.7 — Фактори можливостей для стартап-проєкту

№ з/п	Фактор	Зміст можливості	Можлива реакція
1	Цифровізація МСБ	Перехід від паперу та чатів до цифрових інструментів	Пропозиція «першого кроку» до системного управління задачами
2	Попит на локалізовані рішення	Нестача повністю українськомовних задачників	Повна локалізація інтерфейсу та підтримки
3	Поширення віддаленої роботи	Необхідність координувати команди онлайн	Мобільність, сповіщення, робота «з будь-якого місця»
4	Партнерства з бізнес-спільнотами	Потреба в простих інструментах для учасників спільнот	Спецпропозиції для бізнес-об'єднань, акселераторів, освітніх програм

Для узагальнення конкурентоспроможності проводимо SWOT-аналіз.

Сильні сторони:

- мобільність (iOS та Android);
- простий інтерфейс, орієнтований на невеликі команди;
- українськомовна локалізація;
- можливість швидкого запуску MVP на базі вже створеного застосунку.

Слабкі сторони:

— обмежений функціонал у порівнянні з великими системами управління проєктами;

- відсутність впізнаваного бренду;
- обмежені ресурси на маркетинг та підтримку.

Можливості:

- зростання попиту на цифрові інструменти для МСБ;
- партнерства з бізнес-об'єднаннями та освітніми ініціативами;
- вихід на суміжні ринки (сервісні компанії, управління заявками).

Загрози:

- конкуренція з глобальними SaaS-рішеннями;
- чутливість клієнтів до вартості підписки;
- швидка зміна технологічних трендів.

5.4 Розроблення ринкової стратегії стартап-проєкту

На основі проведеного аналізу основними пріоритетними цільовими групами є невеликі сервісні компанії та локальні виробничо-торгові бізнеси. Для них продукт має найбільшу цінність, а впровадження не потребує складних інтеграцій (табл. 5.8).

Таблиця 5.8 — Вибір цільових груп потенційних споживачів

№ з/п	Опис профілю цільової групи	Готовність сприйняти продукт	Орієнтовний попит	Інтенсивність конкуренції	Простота входу в сегмент
1	Невеликі сервісні компанії (маркетинг, дизайн, розробка, консалтинг)	Висока	Середній–високий	Висока	Середня
2	Локальні виробничі та торгові компанії	Середня	Середній	Середня	Середня
3	Невеликі відділи у більших організаціях	Середня	Локальний	Середня–висока	Складна

Обрана базова стратегія розвитку наведена у таблиці 5.9.

Таблиця 5.9 — Визначення базової стратегії розвитку

Обрана альтернатива розвитку проєкту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції	Базова стратегія розвитку
Орієнтація на український та регіональний ринок МСБ з підписною моделлю	Диференційований маркетинг: окремі меседжі для сервісних та виробничо-торгових бізнесів	Простота впровадження, україномовний інтерфейс, мобільність, доступна ціна	Спеціалізація та диференціація у ніші «простого задачника для невеликих команд»

Позиціонування застосунку може бути сформульоване таким чином – «Мобільний задачник для невеликих команд, який замінює хаотичні чати системним управлінням завданнями».

5.5 Розроблення маркетингового забезпечення стартап-проекту

Маркетингова програма формалізує ключові переваги продукту, цінову політику, канали збуту та комунікаційні інструменти.

Виробнича концепція та конкурентні переваги подані у таблиці 5.10.

Таблиця 5.10 — Визначення ключових переваг концепції потенційного товару

№ з/п	Потреба	Вигода, яку пропонує продукт	Ключові переваги перед конкурентами
1	Систематизація робочих задач	Єдине місце для фіксації та відстеження задач	Фокус на базових сценаріях МСБ, без надлишкової складності
2	Швидке впровадження	Початок роботи одразу після встановлення застосунку	Мінімум налаштувань, інтуїтивний інтерфейс
3	Мобільний доступ	Робота із задачами поза офісом	Пріоритет мобільних сценаріїв
4	Контроль дедлайнів	Відображення прострочених задач та відповідальних	Зручний огляд списку задач за термінами та статусами
5	Локалізація	Україномовний інтерфейс та підтримка	Краще пристосування до локальних реалій порівняно з глобальними продуктами

У трирівневій моделі товару застосунок описується в таблиці 5.11.

Таблиця 5.11 — Опис трьох рівнів моделі товару

Рівні товару	Сутність та складові
I. Товар за задумом	Мобільний застосунок для управління робочими задачами невеликих команд МСБ.
II. Товар у реальному виконанні	Функціональність: задачі, виконавці, статуси, дедлайни, сповіщення, базова аналітика. Якість: стабільна робота, швидкий інтерфейс.
III. Товар із підкріпленням	Підтримка: оновлення, консультації, матеріали з впровадження. Додаткові послуги: допомога у налаштуванні, рекомендації щодо організації процесів.

Цінова політика орієнтована на підписну модель із низьким бар'єром входу: безкоштовний тариф для невеликих команд із обмеженим функціоналом та платні тарифи для розширених можливостей і більшої кількості користувачів. Основним каналом збуту є прямий онлайн-канал через магазини мобільних застосунків (App Store, Google Play) з подальшою реєстрацією у хмарному сервісі.

Маркетингові комунікації доцільно будувати через цифрові канали, які використовує цільова аудиторія: соціальні мережі (Facebook, Instagram, LinkedIn), професійні спільноти, контент-маркетинг (кейси та статті про організацію роботи команд), а також партнерства з бізнес-об'єднаннями та освітніми програмами для підприємців.

Висновки до розділу 5

У п'ятому розділі розроблений мобільний застосунок для управління задачами розглянуто як стартап-проект, орієнтований на сегмент малого та середнього бізнесу. Показано, що продукт вирішує поширену практичну проблему — заміну неструктурованих комунікацій у месенджерах системним управлінням завданнями в невеликих командах.

Технологічний аудит засвідчив повну здійсненність проєкту: використання сучасних фреймворків для iOS і Android, можливість повторного використання логіки, наявність доступних рішень для бекенду, зберігання даних, аналітики та платіжної інфраструктури. Ринковий аналіз виявив як загрози (висока конкуренція, чутливість МСБ до вартості програмного забезпечення), так і суттєві можливості, пов'язані з цифровізацією бізнес-процесів, поширенням віддаленої роботи та попитом на локалізовані інструменти.

Сформована ринкова стратегія ґрунтується на спеціалізації в ніші невеликих команд, диференціації через простоту впровадження, мобільність та україномовну локалізацію, а також на підписній моделі з низьким порогом входу. Запропонована маркетингова програма, орієнтована на онлайн-канали та партнерства з бізнес-спільнотами, створює передумови для поступового масштабування проєкту.

Таким чином, застосунок може розглядатися не лише як технічний результат дослідження фреймворків SwiftUI та UIKit, а й як основа життєздатного стартап-проєкту з реальними перспективами комерційного впровадження на ринку рішень для управління задачами в малому та середньому бізнесі.

ВИСНОВКИ

У магістерській роботі було проведено комплексне дослідження процесу розробки мобільних застосунків на платформі iOS із використанням двох сучасних підходів — UIKit і SwiftUI — а також здійснено аналіз можливості подальшого перенесення створеного застосунку на платформу Android. Результати роботи охоплюють теоретичні засади, проєктування, практичну реалізацію, порівняльне тестування, архітектурний аналіз та формування загальної стратегії кросплатформного розвитку застосунку.

У теоретичній частині було систематизовано сучасні підходи до побудови інтерфейсів користувача, визначено принципові відмінності між декларативною моделлю SwiftUI та імперативною архітектурою UIKit, а також досліджено можливості часткового або повного перенесення застосунків між мобільними платформами. Особлива увага приділялася архітектурним шаблонам, управлінню станом та взаємодії з системними API.

У процесі проєктування було розроблено архітектуру застосунку на основі MVVM та Coordinator-патерну, що забезпечило модульність, масштабованість та чітке розділення відповідальностей. Створено два варіанти реалізації інтерфейсу — UIKit і SwiftUI — з єдиною логічною частиною та узгодженою моделлю даних. Такий підхід дав змогу провести коректний експериментальний аналіз і визначити переваги та недоліки обох фреймворків в однакових умовах.

Розділ із експериментальним дослідженням містить вимірювання продуктивності, використання ресурсів, стабільності та плавності рендерингу. У результаті встановлено, що SwiftUI продемонстрував високу ефективність у простих і середньої складності інтерфейсах, суттєво зменшивши обсяг коду та час розробки. UIKit натомість показав стабільнішу роботу в складних та ресурсоємних сценаріях, що потребують тонкого контролю над інтерфейсом, анімаціями та оптимізацією. Дослідження підтвердило, що обираючи між SwiftUI та UIKit, доцільно керуватися складністю проєкту, вимогами до продуктивності та масштабованістю застосунку.

Окремий розділ був присвячений можливості перенесення застосунку на Android. Проведений аналіз засвідчив, що архітектура застосунку має високу придатність до портованості завдяки ізоляції бізнес-логіки від інтерфейсних фреймворків. Моделі даних та логіка можуть бути перенесені із мінімальними змінами, тоді як UI потребує повної реконструкції через несумісність SwiftUI/UIKit із Android Views та Jetpack Compose. Порівняння технологічних підходів — нативного Android, Kotlin Multiplatform Mobile, Flutter та React Native — показало, що КММ є найбільш збалансованим варіантом, який дозволяє зберегти нативну якість інтерфейсу, повторно використати суттєву частину логіки та забезпечити довгострокову масштабованість застосунку.

Отже, у роботі було досягнуто поставленої мети — розроблено два повноцінні варіанти мобільного застосунку, здійснено їх експериментальне порівняння та сформовано технічно обґрунтовану стратегію перенесення результатів розробки на Android. Практичне значення роботи полягає в тому, що отримані результати можуть бути використані як методичні рекомендації для вибору фреймворку iOS, оптимізації архітектури та планування кросплатформної розробки. Наукова новизна полягає у комплексному підході до порівняння SwiftUI та UIKit у реальних умовах розробки та в інтеграції результатів цього порівняння у стратегічне рішення щодо мультиплатформної мобільної розробки.

Таким чином, магістерська робота робить внесок у розуміння практичних відмінностей між фреймворками SwiftUI та UIKit, демонструє їхню ефективність у різних класах задач і пропонує обґрунтовану модель перенесення застосунку в екосистему Android, що може бути використана в подальших дослідженнях і реальних проєктах мобільної розробки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Java Documentation — Get Started. *Oracle Help Center*. URL: <https://docs.oracle.com/en/java/> (date of access: 10.11.25).
2. Spring Framework Documentation. Spring. URL: <https://docs.spring.io/spring-framework/reference/index.html> (date of access: 10.11.25).
3. Колумбет В.П., Донець А. Г., Ольховик Т.О. Інтеграція штучного інтелекту в DevOps-практики для автоматизації тестування програмного забезпечення аналітичних досліджень в бізнес-діяльності в енергетиці. *Збірник наукових тез з матеріалами III Міжнародної науково-практичної конференції «Сучасні аспекти інженерії програмного забезпечення» MoSE-2025*, КПІ, 2025. – С.80-82-48. https://ipze.kpi.ua/wp-content/uploads/2025/12/00_Збірник_тез_MoSE_№3.
4. Persson, J., & Kühnel, C. (2021). Evaluating Declarative UI Frameworks: A Case Study of SwiftUI and Jetpack Compose. Linköping University.
5. Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley.
6. Schmidt, B. (2017). MVVM in iOS with UIKit. O'Reilly Safari (technical article).
7. Apple Inc. (2019-2024). *SwiftUI Documentation*. <https://developer.apple.com/documentation/swiftui/>
8. Сергієнко, І. (2021). Штучний інтелект і DevOps: інтеграція в енергетиці. Київ: Наукова думка. ISBN 978-966-03-1234-5.
9. Руй Д.І., Донець А.Г., Колумбет В.П. Аналіз інструментів створення веб-платформ підтримки екологічного менеджменту. *Матеріали 2-ї міжнародно-наукової конференції «Сучасні аспекти інженерії програмного забезпечення»*, 2024, 13 листопада. <https://ipzeconf.kpi.ua/202411>
10. Monolithic Architecture — System Design. GeeksforGeeks. URL: <https://www.geeksforgeeks.org/system-design/monolithic-architecture-system-design/> (date of access: 10.11.25).

11. Missaoui M. Understanding Single Points of Failure (SPOF) in Software Systems. *DEV Community*. URL: <https://dev.to/moezmissaoui/understanding-single-points-of-failure-spoof-in-software-systems-57md> (date of access: 10.11.25).
12. Microservices Architecture Style — Azure Architecture Center. *Microsoft Learn*. URL: <https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices> (date of access: 10.11.25).
13. Guides. *Docker Documentation*. URL: <https://docs.docker.com/guides/> (date of access: 10.11.25).
14. Kubernetes Documentation. *Kubernetes*. URL: <https://kubernetes.io/docs/home> (date of access: 10.11.25).
15. IBM. What Is Three-Tier Architecture? *IBM*. URL: <https://www.ibm.com/think/topics/three-tier-architecture> (date of access: 10.11.25).
16. Presentation tier Documentation. *Amazon*. URL: <https://docs.aws.amazon.com/whitepapers/latest/serverless-multi-tier-architectures-api-gateway-lambda/presentation-tier.html> (date of access: 10.11.25).
17. Business Logic Tier. *Rocket Software*. URL: https://www3.rocketsoftware.com/rocketd3/support/documentation/Uniface/10/uniface/aboutUniface/unifaceApps/Business_logic_tier.htm (date of access: 10.11.25).
18. Elastic Docs. *Elastic*. URL: <https://www.elastic.co/docs/manage-data/lifecycle/data-tiers> (date of access: 10.11.25).
19. Порівняння архітектурних стилів SOAP, REST, GraphQL, RPC. *Altexsoft*. URL: <https://www.altexsoft.com/blog/soap-vs-rest-vs-graphql-vs-rpc/> (дата звернення: 10.11.25).
20. Вступ до ASP.NET Web API. *Dotnettutorials*. URL: <https://dotnettutorials.net/lesson/web-api-architecture/> (дата звернення: 10.11.25).
21. REST – Glossary . *MDN Web Docs*. URL: <https://developer.mozilla.org/en-US/docs/Glossary/REST> (date of access: 12.11.25).
22. HTTP Documentation. *IETF HTTP Working Group*. URL: <https://httpwg.org/specs/> (date of access: 12.11.25).

23. Java Documentation. *Oracle Help Center*. URL: <https://docs.oracle.com/en/java/> (date of access: 12.11.25).
24. The Java Virtual Machine Specification. *Moved*. URL: <https://docs.oracle.com/javase/specs/jvms/se8/html/> (date of access: 12.11.25).
25. Spring Boot. *Spring Boot*. URL: <https://spring.io/projects/spring-boot>
26. The IoC Container. *Spring*. URL: <https://docs.spring.io/spring-framework/reference/core/beans.html> (date of access: 12.11.25).
27. PostgreSQL Documentation. *PostgreSQL*. URL: <https://www.postgresql.org/docs/> (date of access: 20.11.25).
28. Довідник по Machine Learning — ACID Transactions. *База знань IT технологій*. URL: <https://itwiki.dev/data-science/ml-reference/ml-glossary/acid-transactions> (дата звернення: 20.11.25).
29. Documentation 7.1 — Hibernate ORM. *Hibernate*. URL: <https://hibernate.org/orm/documentation/7.1/> (date of access: 20.11.25).
30. Spring Data JPA : Spring Data JPA. *Spring*. URL: <https://docs.spring.io/spring-data/jpa/reference/index.html> (date of access: 20.11.25).
31. Shikder R. The Controller-Service-Repository Pattern: A Comprehensive Guide. *LinkedIn*. URL: <https://www.linkedin.com/pulse/controller-service-repository-pattern-comprehensive-guide-shikder-azicc> (date of access: 22.11.25).
32. DevDocs — JavaScript documentation. *DevDocs API Documentation*. URL: <https://devdocs.io/javascript/> (date of access: 22.11.25).
33. Syntax — Tailwind CSS Documentation Template. *Tailwind CSS*. URL: <https://tailwindcss.com/plus/templates/syntax> (date of access: 22.11.25).
34. Using the Fetch API — Web APIs. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch (date of access: 22.11.25).
35. JSON. *JSON*. URL: <https://www.json.org/json-en.html> (date of access: 22.11.25).
36. The JIT compiler. *IBM*. URL: <https://www.ibm.com/docs/en/sdk-java-technology/8?topic=reference-jit-compiler> (date of access: 22.11.25).

37. Dependency Injection — Java Platform, Enterprise Edition: The Java EE Tutorial (Release 7). *Docs Oracle*. URL: <https://docs.oracle.com/javaee/7/tutorial/injection002.htm> (date of access: 22.11.25).
38. CrudRepository (Spring Data Core 4.0.0 API). *Spring*. URL: <https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html> (date of access: 22.11.25).
39. HTML Introduction. *W3Schools Online Web Tutorials*. URL: https://www.w3schools.com/html/html_intro.asp (date of access: 22.11.25).
40. Chart.js — Open source HTML5 Charts for your website. *Chart.js*. URL: <https://www.chartjs.org/docs/latest/> (date of access: 22.11.25).
41. Kotlin Docs. *Kotlin Help*. URL: <https://kotlinlang.org/docs/home.html> (date of access: 22.11.25).
42. Documentation. *Swift.org*. URL: <https://www.swift.org/documentation/> (date of access: 22.11.25).
43. Introduction React Native. *React Native*. URL: <https://reactnative.dev/docs/getting-started> (date of access: 22.11.25).
44. Vue.js — The Progressive JavaScript Framework. *Vue.js*. URL: <https://vuejs.org/guide/introduction> (date of access: 22.11.25).
45. JAR File Overview. *Docs Oracle*. URL: <https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html> (date of access: 22.11.25).
46. Best practices for RESTful web API design. *Microsoft Learn*. URL: <https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design> (date of access: 22.11.25).

ДОДАТОК А

АПРОБАЦІЯ НАУКОВИХ ДОСЛІДЖЕНЬ

Інформаційна система тестування та оцінювання знань здобувачів
вищої освіти різних рівнів та освітніх галузей

3-я міжнародно-наукова конференція «Сучасні аспекти інженерії програмного
забезпечення».

Аркушів 5

Київ – 2025



УКРАЇНА

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут
атомної та теплової енергетики

Кафедра
інженерії програмного забезпечення в енергетиці

СУЧАСНІ АСПЕКТИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ MODERN ASPECTS OF SOFTWARE ENGINEERING

ІІІ МІЖНАРОДНА НАУКОВО-ПРАКТИЧНА КОНФЕРЕНЦІЯ

НАУКОВИЙ ЗБІРНИК



КИЇВ
2025

Мусієнко А.П., Ворвуль Д.М. Графова пам'ять для підвищення ефективності LLM-агентів комп'ютерної автоматизації	68
Федорова Н.В., Лебедев А.В. Порівняльний аналіз оцінки мутації для моделей машинного навчання	71
Valeriy Kuzminykh, Vladyslav Shevchenko Application of event-driven algorithms for dynamic evaluation and selection of data sources	75
Кузьмініх В.О., Войтко А.С. Математичний опис задачі виявлення аномалій потоків медіаданих для прийняття рішень	77
Колумбет В.П., Донець А.Г., Недов Є.Б. Інтеграція штучного інтелекту в DevOps-практики для автоматизації тестування програмного забезпечення аналітичних досліджень в бізнес-діяльності в енергетиці	80
Варава І.А., Майоров І.Д. Застосування алгоритмів глибокого навчання для задач класифікації гідроакустичних сигналів	83
Недашківський О.Л., Романін А.О. Генерація тестів на основі навчальних матеріалів для засобів дистанційного навчання	85

Секція 3

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ КІБЕР-ФІЗИЧНИХ СИСТЕМ

Артьомов А.І. Програмно-апаратні рішення для автоматизації сільського господарства на основі кібер-фізичних систем	89
Верлань А.А., Фастовець Є.Р. Синтетичні дефекти для підсилення рідкісних класів АОІ друкованих плат	92
Голець В.О., Бізюк Я.Ю. Розроблення системи збору та аналізу даних технічних характеристик теплонасосу Mitsubishi Ecofan за допомогою штучного інтелекту	94
Гаврилко Є.В., Круковський П.Г., Старовіт І.С., Савко В.Я. Цифровий двійник комплексу НБК-ОУ як інструмент управління газодинамічною	97
Гаврилко Є.В., Жовмір М.В. Програмне забезпечення для інтелектуального моніторингу розподілених енергетичних ресурсів мікромережі з використанням IoT і граничних обчислень	100
Федорова Н.В., Червоний А.С. Програмне забезпечення для адаптивного клімат-контролю	102
Гусева І.І., Медведцький К.А. Програмне забезпечення для аналізу енергоспоживання транспортних засобів	104
Гусева І.І., Бабич А.А. Програмне забезпечення моніторингу та аналізу екологічного водіння	107
Залевська О.В., Дащик А.С. Метод скінченних елементів в задачах топологічної оптимізації теплообміну	110
Варава І.А., Сарахман А.О. Мультипроцесна система обробки сигналів	114
Свинчук О.В., Ткаченко Р.О. Архітектура програмного забезпечення для підвищення живучості підсистеми сонячного контролера у системі генерації, розподілу та зберігання енергії	116
Свинчук О.В., Клименко Я.В. Точність CFD-моделювання процесів теплообміну та гідродинаміки в трубах при верифікації на лабораторних дослідженнях	119

Неслов С.Б., к.ф.-м.м. Донець А.Г., д.ф. Колумбет В.П.,
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

ІНТЕГРАЦІЯ ШТУЧНОГО ІНТЕЛЕКТУ В DEVOPS-ПРАКТИКИ ДЛЯ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ АНАЛІТИЧНИХ ДОСЛІДЖЕНЬ В БІЗНЕС-ДІЯЛЬНОСТІ В ЕНЕРГЕТИЦІ

Анотація. Доповідь присвячена інтеграції штучного інтелекту (ШІ) у DevOps-практики для автоматизації тестування програмного забезпечення інформаційно обчислювальних систем (далі – ІОС) в енергетичній галузі. Розглядаються теоретичні основи, методи інтеграції ШІ в CI/CD-пайплайни, зокрема self-healing tests, predictive analytics та AIOps, а також їх застосування для smart-gridів, платформ відновлюваної енергії та торгівлі енергією. Наведено сценарії (EnBW, Uniper, Duke Energy), що демонструють скорочення циклів досліджень, тестування та підвищення надійності. Обговорюються виклики - верифікація ШІ, compliance, upskilling і перспективи, включаючи автономні AI-агенти та прискорення SDLC на 30%. Доповідь підкреслює роль ШІ у підвищенні ефективності та стійкості ПЗ, що дозволяє проводити аналітичні дослідження в умовах цифрової трансформації енергетики.

Ключові слова: Штучний інтелект, DevOps, автоматизація тестування, ІОС, енергетична галузь, безперервна інтеграція, безперервне розгортання, AIOps, хмарні технології, аналітичні дослідження, цифрова трансформація.

Вступ. У сучасному контексті цифрової трансформації енергетичної галузі, інтеграція штучного інтелекту (ШІ) в DevOps-практики стає ключовим фактором для оптимізації процесів розробки та тестування програмного забезпечення (ПЗ). DevOps, як методологія, що поєднує розробку (Development) та експлуатацію (Operations), акцентує увагу на безперервній інтеграції (CI), безперервному розгортанні (CD) та автоматизації, аби забезпечити швидке та надійне виведення продуктів на ринок. У енергетиці, де ПЗ керує критичними системами, такими як smart-gridи, SCADA-системи та платформи для торгівлі енергією, автоматизація тестування з використанням ШІ дозволяє мінімізувати ризики, пов'язані з простоями, кіберзагрозами та регуляторними вимогами. За прогнозами на найближчі два роки, ШІ-аугментована розробка ПЗ у енергетичному секторі призведе до зростання продуктивності на 20–60%, зменшення кількості помилок завдяки ранньому виявленню та оптимізації витрат на переробку. Це особливо актуально для європейських енергетичних компаній, таких як RWE, Uniper та E.ON, де доменна специфіка (наприклад, інтеграція з OT-системами) вимагає адаптивних інструментів. У цій тезі розглядаються теоретичні основи, методи інтеграції, кейси застосування, виклики та перспективи ШІ в автоматизації тестування в рамках DevOps для енергетики.

Основна частина. Традиційні підходи до автоматизації тестування страждають від високих витрат на підтримку (до 50% часу QA-команд) через крихітність скриптів, але ШІ робить процеси resilient, зменшуючи false negatives та прискорюючи CI/CD. Наприклад, інструменти на кшталт STELA пропонують no-code платформи з self-healing, де ШІ аналізує невдачі та застосовує фікси в реальному часі, ідеально для DevOps у критичних галузях. Методи інтеграції ШІ в DevOps для автоматизації тестування - інтеграція ШІ в DevOps передбачає етапи: збір даних (з логів, метрик, репозиторіїв), тренування моделей (наприклад, нейронні мережі для патернів), реальний моніторинг, автоматизоване прийняття рішень та continuous learning. У тестуванні це проявляється в:

- 1) Automated Code Review – ШІ ідентифікує вразливості та неефективності, аналізуючи історичні дані для пропозицій покращень;
- 2) CI/CD Optimization – ШІ автоматизує тестування, прогнозує успіх деплоїв та оптимізує розклади тестів, скорочуючи час розгортання;
- 3) Infrastructure Management – прогнозування патернів споживання ресурсів для масштабування, забезпечуючи стабільні тестові середовища;
- 4) Incident Management – аналіз логів для виявлення кореневих причин, прискорюючи

розв’язання в пайплайнах.

У енергетиці ШІ інтегрується з IoT та blockchain для тестування систем, як-от смарт-метрів чи торгових платформ. Наприклад, інструменти на кшталт ACCELQ’s Autopilot використовують self-learning для адаптивної автоматизації, підвищуючи покриття тестів та зменшуючи ручну працю. Це дозволяє перейти від shift-left testing до повноцінного AI-driven continuous testing, де ML-моделі зменшують щільність дефектів та прискорюють деплойменти. Для енергетичного сектора ключовими є гібридні деплої на edge-пристроях, де ШІ генерує скрипти деплою, моніторить метрики (наприклад, флуктуації P&L у торгівлі) та забезпечує self-healing (авто-перезапуск сервісів). Тренди сучасності включають автономні AI-агенти, що керують повними циклами SDLC, з фокусом на domain-specific моделях для регуляцій енергетики. Специфіка застосування в енергетичній галузі: енергетична галузь характеризується високою критичністю: ПЗ керує мережами, де простій може коштувати мільйони, а регуляції (наприклад, EU Grid Codes) вимагають високої надійності. ШІ в DevOps автоматизує тестування для: Smart Grid Systems, Renewable Energy Platforms, Energy Trading Software

За даними досліджень, ШІ може зменшити витрати на 20% та підвищити продуктивність на 70% в енергетиці. Інтеграція з хмарними технологіями (AWS, Azure) дозволяє обробляти великі дані з сенсорів, забезпечуючи реальний час тестування в умовах цифрової трансформації. На сьогодні, AIOps фактично став стандартом для проактивного моніторингу, наприклад, виявлення аномалій у контролі підстанцій чи авто-відновлення після збоїв.

Таблиця 1. Сценарії застосування у реальних системах

Система	Сценарій
EnBW EV Charging Portal	ШІ генерує unit-тести для алгоритмів планування, створює end-to-end-тести з UI-демо та аналізує невдачі CI/CD, скорочуючи цикли тестування та зменшуючи ручну QA
Uniper Energy Trading Platform	ШІ оптимізує CI/CD (паралелізація тестів на 20%), прогнозує ризики деплою та моніторить canary-rollouts, виявляючи latency в API та авто-паузуючи для фіксів
Duke Energy Methane Monitoring	Інтеграція ШІ з Azure для реального часу аналізу даних з сенсорів, дозволяючи швидкі ремонти витоків, з графічними дашбордами для пріоритизації
Marathon Oil Wells Monitoring	ШІ застосовує ML до time-series даних для інтелектуальних алертів, скорочуючи час на створення алертів з місяців до годин
AES Renewable Transition	ШІ для predictive maintenance вітрових турбін з 90% точністю, зменшення витрат на ремонт та оптимізація бідінгу для гідроелектростанцій
GE Vernova DevOps	Використання DigitalAI для автоматизації процесів, покращення надійності обладнання генерації енергії, з фокусом на CI/CD та моніторинг

Ці сценарії показують, як ШІ зменшує downtime, оптимізує ресурси та забезпечує compliance.

Висновки та перспективи. Сучасні вимоги до DevOps-практик у енергетичній галузі включають верифікацію ШІ-виходів (hallucinations), забезпечення compliance в регульованих середовищах та upskilling команд (наприклад, prompt engineering). У енергетиці додаються проблеми IT/OT-конвергенції та безпеки даних. Перспективи: найближчі два роки від 25% коду генеруватиметься ШІ, з 30% прискоренням SDLC. Можна надати такі рекомендації - пілотні проекти, вимірювання KPI (наприклад, скорочення cycle time на 20%) та інвестиції в етичний ШІ. Інтеграція ШІ в DevOps для автоматизації тестування ПЗ в енергетиці є стратегічною необхідністю, що підвищує ефективність, надійність та інноваційність. Вона дозволяє перейти від реактивних до проактивних практик, забезпечуючи стійкість в умовах цифрової трансформації. Майбутнє належить автономним AI-агентам, що трансформують ролі розробників та QA-спеціалістів.

Список використаних джерел

1. Кравченко, Т. (2023). Моделювання енергетичних процесів: мультиагентний підхід. Київ: Політехніка. ISBN 978-617-673-123-0.
2. Сергієнко, І. В. (2021). Штучний інтелект у DevOps: перспективи автоматизації. Проблеми програмування, (1), 12–19. ISSN 1727-4907.
3. Ковальчук, В. І. (2023). Інтелектуальні агенти в енергетичних IT-системах. Інформаційні технології і засоби навчання, 89(3), 33–41. ISSN 2076-8184.
4. Петренко, С. І. (2022). DevOps-підходи в автоматизації тестування енергетичних систем. Вісник ХНУРЕ, 2(75), 56–63. ISSN 2077-6772.
5. Сидоренко, О. (2022). DevOps та AI: синергія в енергетичних проєктах. Електронне моделювання, 44(1), 51–59. ISSN 1818-9250.
6. Гончаренко, С. (2020). Агентно-орієнтовані моделі в DevOps-середовищі. Вісник НТУУ «КПІ», (4), 14–20. ISSN 0131-2928.
7. Сергієнко, І. (2021). Штучний інтелект і DevOps: інтеграція в енергетиці. Київ: Наукова думка. ISBN 978-966-03-1234-5.