

Міністерство освіти і науки України
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

магістр

(ступінь вищої освіти)

на тему ДОСЛІДЖЕННЯ ТА РЕАЛІЗАЦІЯ ТЕХНОЛОГІЙ РОЗПІЗНАВАННЯ
ПРОДУКТІВ З ВИКОРИСТАННЯМ МОДЕЛЕЙ МАШИННОГО НАВЧАННЯ
ДЛЯ ФОРМУВАННЯ КУЛІНАРНИХ РЕКОМЕНДАЦІЙ

RESEARCH AND IMPLEMENTATION OF PRODUCT RECOGNITION

TECHNOLOGY USING MACHINE LEARNING MODELS FOR CULINARY

RECOMMENDATION

Виконав: студент 2 курсу, групи КНТ-124м

Спеціальності 121 Інженерія програмного
забезпечення

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

ПАНЧЕНКО М.В.

(ПРИЗВИЩЕ та ініціали)

Керівник ГЛАДКОВА О.М.

(ПРИЗВИЩЕ та ініціали)

Рецензент ДЬЯЧУК Т.С.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти магістр
Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
“ ” 2025 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

ПАНЧЕНКО Михайла Володимировича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Дослідження та реалізація технології розпізнавання продуктів з використанням моделей машинного навчання для формування кулінарних рекомендацій. Research and Implementation of Product Recognition Technology Using Machine Learning Models for Culinary Recommendation

керівник проєкту (роботи) к.т.н., ГЛАДКОВА Ольга Миколаївна,
(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “ 30 ” вересня 2025 року № 447

2. Строк подання студентом проєкту (роботи) 8 грудня 2025 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Теоретичні відомості про розробку iOS застосунків. 2. Аналіз існуючих застосунків аналогів. Вибір архітектури застосунку. 3. Розробка iOS застосунку для сканування інгредієнтів та видачі кулінарних рекомендацій. 4. Тестування застосунку. Заповнення даних для бази рецептів.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	ГЛАДКОВА О.М., доцент		
Нормоконтролер	БЄЛОВА А.В., асистент		

7. Дата видачі завдання “ 20 ” вересня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	2-3 тижні	Розділ 1
3	Розробка та удосконалення методів, моделей й алгоритмів вирішення задачі.	4-5 тижні	Розділ 2
4	Розробка архітектури застосунку.	6 тиждень	Розділ 3
5	Розробка застосунку.	7-8 тижні	Розділ 3
6	Тестування та експериментальне дослідження програмного забезпечення.	9 тиждень	Розділ 4
7	Оформлення пояснювальної записки та документів до неї.	10-11 тижні	Додатки
8	Нормоконтроль та рецензування.	12 тиждень	
9	Захист роботи.	12 тиждень	

Студент(ка)

(підпис) Михайло ПАНЧЕНКО
(Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи)

(підпис) Ольга ГЛАДКОВА
(Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи магістра:
85 с., 4 табл., 34 рис., 2 дод., 16 джерел.

IOS, SWIFT, COREML, CREATEML, REDUX АРХІТЕКТУРА,
VISION, FIREBASE, SWIFTUI, XCODE

Об'єкт дослідження – процес інженерії програмного забезпечення для розпізнавання об'єктів в реальному часі.

Предмет дослідження – мобільні застосунки для формування кулінарних рекомендацій з наведених інгредієнтів.

Мета роботи – створення застосунку що прискорить вибір кулінарного рецепту завдяки розпізнаванню невідомих інгредієнтів в домашніх умовах та в магазині.

Матеріали, методи та технічні засоби: структурне та об'єктно-орієнтоване програмування, мова програмування Swift, персональний комп'ютер з процесором M4 Pro під управлінням операційної системи MAC OS.

Результати. Створено програмний IOS-застосунок, який за допомогою машинного навчання та комп'ютерного зору відрізняє різні продукти та видає рецепти з ними

Висновки. Розроблено мобільний застосунок який прискорює знаходження рецепту по розпізнаванню об'єкта в живу.

Галузь використання – знаходження продуктів та рецептів з них в магазині або в домашніх умовах.

ABSTRACT

Explanatory note to the diploma qualifying work of the master: 85 pages, 4 tables, 34 figures, 2 appendixes, 16 sources.

IOS, SWIFT, COREML, CREATEML, REDUX ARCHITECTURE, VISION, FIREBASE, SWIFTUI, XCODE.

The object of research is the process of software engineering for real-time object recognition.

The subject of research is mobile applications for generating culinary recommendations from the given ingredients.

The purpose of the work is to create an application that will speed up the selection of a culinary recipe by recognizing unknown ingredients at home and in the store.

Materials, methods and technical means: structural and object-oriented programming, Swift programming language, personal computer with an M4 Pro processor running the MAC OS operating system.

Results. An iOS software application has been created that, using machine learning and computer vision, distinguishes different products and issues recipes with them

Conclusions. A mobile application has been developed that speeds up finding a recipe by recognizing an object in real time.

Field of application - finding products and recipes from them in a store or at home.

ЗМІСТ

	С.
Перелік скорочень та умовних позначок	8
Вступ.....	9
1 Теоритичні відомості про розробку ios застосунків	11
1.1 Розробка iOS застосунків	11
1.2 Бібліотеки для створення UI на мові програмування Swift для iOS застосунку	14
1.3 Машинне навчання та його використання в iOS застосунках	16
1.4 Висновки за розділом.....	19
2 Аналіз існуючих застосунків аналогів. Вибір архітектури застосунку	20
2.1 Аналіз існуючих застосунків аналогів	20
2.2 Вибір архітектури застосунку	24
2.3 Висновки за розділом.....	33
3 Розробка iOS застосунку для скану ігредієнтів та видачі кулінарних рекомендацій.....	35
3.1 Створення iOS застосунку.....	35
3.2 Заповнення даних для бази рецептів та налаштування Firebase.....	37
3.3 Оформлення Redux архітектури	39
3.4 Створення DataSet для CreateML.....	41
3.5 Інтеграція CoreML та Vision до проекту.....	43
3.6 Діаграми сценаріїв роботи застосунку.....	44
3.7 Висновки за розділом.....	45
4 Тестування застосунку для розпізнавання інгредієнтів та формування кулінарних рекомендацій	47

4.1 Тестування iOS застосунку для розпізнавання інгредієнтів та формування кулінарних рекомендацій	47
4.2 Тестування в домашніх умовах.....	47
4.3 Тестування в умовах магазину.....	49
4.4 Аналіз результатів тестування	51
4.5 Взаємодія користувача з застосунком.....	52
4.6 Висновок за розділом.....	54
Висновки	55
Перелік джерел посилання	57
Додаток А Текст програми.....	59
Додаток Б Слайди презентації	79

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

- IDE – Integrated Development Environment,
ML – машинне навчання (Machine Learning);
UI – інтерфейс користувача (User Interface);

ВСТУП

Сучасний розвиток мобільних технологій сприяє створенню інтелектуальних систем, здатних спрощувати повсякденні завдання користувачів завдяки використанню технологій машинного навчання (Machine Learning, ML) та комп'ютерного зору.

Одним із перспективних напрямів є розпізнавання об'єктів, зокрема харчових продуктів, що відкриває можливість побудови систем кулінарних рекомендацій, персоналізованого харчування та контролю дієти.

Завдяки інтеграції технологій ML у мобільні застосунки користувач отримує зручний інструмент, який може аналізувати вхідні зображення продуктів, визначати їх тип, а також пропонувати рецепти приготування на основі доступних інгредієнтів.

Платформа iOS надає широкий набір засобів для побудови подібних рішень. Зокрема, фреймворки CoreML та Vision дозволяють інтегрувати моделі машинного навчання безпосередньо у мобільний застосунок, забезпечуючи обробку даних локально на пристрої користувача без необхідності постійного підключення до мережі.

Це забезпечує високу швидкодію, конфіденційність даних і незалежність від зовнішніх сервісів.

Додатково, використання сучасних архітектурних підходів, таких як Redux у поєднанні з реактивною бібліотекою Combine та декларативним інтерфейсом SwiftUI, дає змогу створити стабільний, масштабований і легкий у підтримці програмний продукт.

Метою даної магістерської кваліфікаційної роботи є дослідження, проектування та реалізація технології розпізнавання харчових продуктів на

основі ML-моделей із подальшим формуванням персоналізованих кулінарних рекомендацій у межах мобільного застосунку для iOS.

Наукова новизна роботи полягає у комплексному поєднанні фреймворків CoreML, Vision та Combine у межах архітектури Redux, що дозволяє реалізувати реактивну систему з високою продуктивністю, стабільністю та передбачуваним потоком даних.

Практичне значення роботи полягає у створенні прототипу iOS-застосунку, який може бути використаний як основа для розробки комерційних рішень у сфері кулінарних рекомендацій, контролю харчування або фуд-аналітики.

Розроблений застосунок є прикладом інтеграції сучасних технологій штучного інтелекту в повсякденне життя користувача та демонструє потенціал iOS-платформи для реалізації інноваційних ML-рішень.

1 ТЕОРИТИЧНІ ВІДОМОСТІ ПРО РОЗРОБКУ IOS ЗАСТОСУНКІВ

1.1 Розробка iOS застосунків

Розробка iOS застосунків починається з вибору інструментів для створення застосунку. Є варіанти створення iOS застосунків за допомогою крос-платформ методів таких як Flutter, що використовує мову програмування Dart та React native, що використовує мову програмування Java Script. Основною різницею між способами створення IOS застосунків, є те що Flutter та React native можуть використовувати базові фреймворки Apple, але вони все одно мають прошарки, які можуть бути не up to date. Тому нативний метод проектування, є найшвидшим та найбезпечнішим варіантом.

Є дві основні мови програмування нативної розробки, це Objective-C та Swift. Apple рекомендує використання Swift через те, що Swift оновлюється під нові iOS, а від objective-c Apple намагається повністю відійти. Swift набагато швидший та має більш простіший синтаксис за Objective-C.

До плюсів розробки на Swift порівнюючи з крос-платформ мовами:

- можливість розробки для Apple Watch та Vision Pro;
- використання нативних UI (user interface) елементів, це дозволяє виглядати в стилі iOS і при оновленні елементів від Apple бути up to date;
- Swift має доволі простий синтаксис.

Swift – багатопарадигмова компільована мова програмування, розроблена компанією Apple для того, щоб співіснувати з Objective C і бути стійкішою до помилкового коду. Swift була представлена на конференції розробників WWDC 2014. Мова побудована з LLVM компілятором, включеного у Xcode 6 beta [1].

В таблиці 1.1 зведено порівняння мов та платформ розробки.

Таблиця 1.1 – Порівняння технологій програмування

Критерій	Swift	Objective-C	Flutter	React native
Розробник	Apple	Apple	Google	Meta
Продуктивність	Вища	Висока	Висока	Середня
Зручність розробки	Сучасний синтаксис, легка розробка	Складний синтаксис	Легка розробка	Середня важкість розробки
Підтримка ML (CoreML / Vision)	Пряма підтримка, нативна інтеграція	Повна, але писати складніше	Існує лише через плагіни	ML через JS дуже обмежене
Системні можливості	Повний доступ до iOS API	Повний доступ до iOS API	Частковий, через платформенні канали	Частковий, залежність від нативних модулів
Простота інтеграції з нативними модулями	Висока	Висока	Середня (через platform channels)	Середня/низька (через native modules)
Кросплатформеність	немає	немає	є	є

Компілятор Swift побудований з використанням технологій вільного проекту LLVM. Swift успадковує найкращі елементи мов C і Objective-C, тому синтаксис звичний для знайомих з ними розробників, але водночас відрізняється використанням засобів автоматичного розподілу пам'яті і контролю переповнення змінних і масивів, що значно збільшує надійність і безпеку коду.

При цьому Swift-програми компілюються у машинний код, що дозволяє забезпечити високу швидкодію. За заявою Apple, код Swift виконується в 1.3 рази швидше коду на Objective-C. Замість збирача сміття Objective-C в Swift використовуються засоби підрахунку посилань на об'єкти, а також надані у LLVM оптимізації, такі як автовекторизація.

Мова також пропонує низку сучасних методів програмування, таких як замикання, узагальнене програмування, лямбда-вирази, кортежі і словникові типи, швидкі операції над колекціями, елементи функційного програмування. Основним застосуванням Swift є розробка користувацьких застосунків для macOS, iOS, tvOS, watchOS з використанням тулкіта Cocoa і Cocoa Touch. При цьому Swift надає об'єктну модель, сумісну з Objective-C. Сирцевий код мовою Swift може змішуватися з кодом на C і Objective-C в одному проєкті.

Swift щільно інтегровано до IDE(інтегроване середовище розробки) Xcode, проте може бути викликано з терміналу, що уможливорює її використання на операційних системах, відмінних від macOS, наприклад, на Linux та Windows [1].

Для використання мови Swift буде використовуватися інтегроване середовище розробки Xcode. Xcode - це інтегроване середовище розробки Apple (IDE) для macOS, що використовується для розробки програмного забезпечення для macOS, iOS, iPadOS, watchOS та tvOS. Вперше він вийшов у 2003 році; останній стабільний випуск - версія 12.5, випущена 26 квітня 2021

року, і доступна через Mac App Store безкоштовно для користувачів macOS Big Sur. Зареєстровані розробники можуть завантажувати попередні випуски та попередні версії набору через веб-сайт Apple Developer. Xcode включає інструменти командного рядка (CLT), які дозволяють розробляти в UNIX-подібній системі за допомогою програми Terminal у macOS. Їх також можна завантажити та встановити без основної IDE [2].

1.2 Бібліотеки для створення UI на мові програмування Swift для iOS застосунку

Існує дві нативні бібліотеки для створення UI елементів на Swift це SwiftUI та UIKit.

UIKit – це один з основних фреймворків, який надає розробникам можливість створювати графічні інтерфейси для застосунків на платформі iOS. Він був розроблений компанією Apple і став невід’ємною частиною екосистеми iOS, даючи змогу розробникам реалізовувати складні та інтерактивні користувацькі інтерфейси з мінімальними зусиллями [3].

Основні функції UIKit охоплюють управління елементами користувацького інтерфейсу, такими як кнопки, мітки, текстові поля і зображення, а також обробку дотиків і жестів, що робить взаємодію з застосунками більш інтуїтивною. Крім того, UIKit підтримує автоматичне компонування (Auto Layout), що дає змогу адаптувати інтерфейси до різних розмірів екранів і орієнтацій пристроїв, забезпечуючи тим самим універсальність застосунків.

SwiftUI відносно нова бібліотека вона має декілька плюсів над UIKit, а саме більш простий та зрозумілий синтаксис, який в своїй основі використовує паттерн Builder, що спрощує створення UI. Також не потрібно створювати

констрейнти, які прив'язують UI елементи одне до одного і має багато проблем зі складним UI. Ще одним з плюсів є Preview за допомогою чого ми можемо в режимі наживо бачити зміни UI елементів. В таблиці 1.2 наведено порівняння мов програмування.

Таблиця 1.2 – Порівняльна таблиця SwiftUI та UIKit

Критерії	SwiftUI	UIKit
Тип фреймворку	Декларативний	Імперативний
Рік появи	2019 (iOS 13)	2008 (iPhoneOS 2)
Підхід до побудови UI	Описує що має бути на екрані	Описує як це має бути реалізовано
Продуктивність	Дуже висока, оптимізована під сучасні iOS	Висока, але потребує більше ручної оптимізації
Інтерактивність та прев'ю	Live Preview у Xcode	Відсутній нативний прев'ю режим
Підтримка Apple	Основний фреймворк розвитку UI	Підтримується, але як legacy
Найкраще підходить для	Нових проєктів, швидкої розробки, ML/AR, анімацій	Складних кастомних UI, великих legacy систем
Можливість змішування	Можна вбудовувати UIKit-компоненти через UIViewRepresentable	Можна вбудовувати SwiftUI через UIHostingController
Мінімальна версія iOS	iOS 13+	Підтримує старі версії (iOS 8+)

Є і недоліки в основному пов'язаний саме з тим що деякі елементи працюють тільки через UIKit, наприклад photo picker. Для обходу цього недоліку використовується UIViewControllerRepresentable протокол який допомагає UIKit елемент переробити у SwiftUI View. Також є, проблеми з деякими елементами SwiftUI, вони можуть мати не весь функціонал в залежності від цільової версії iOS.

1.3 Машинне навчання та його використання в iOS застосунках

Машинне навчання - це галузь досліджень штучного інтелекту, основний напрямок якої розробка та вивчення статистичних алгоритмів, які здатні навчатися з даних і узагальнюватися на небачені данні, через що можуть виконувати завдання без явних інструкцій [4].

Класичний процес машинного навчання включає етапи:

- збір даних - підбір навчальних даних, які відображають реальний стан речей;
- попередня обробка - очищення, нормалізація, перетворення даних;
- навчання моделі - тренування алгоритму за допомогою наприклад CreateML на цих даних;
- оцінка моделі - перевірка точності;
- інференс - використання навченої моделі для передбачення результатів у реальних сценаріях.

Мобільні пристрої стали потужними обчислюваними платформами здатними виконувати складні моделі машинного навчання на самому пристрої, без необхідності у серверній інфраструктурі. Це стало можливим завдяки оптимізації фреймворків Apple та появі процесорів із нейронними ядрами -

Apple Neural Engine починаючи з A11 Bionic в iPhone X, iPhone 8 та iPhone 8 Plus.

Існують такі напрямки машинного навчання в iOS застосунках:

- аудіоаналіз - класифікація звуків, музичних жанрів, шумів тощо;
- обробка природної мови – автозаповнення тексту, розпізнання голосу тощо;
- комп'ютерний зір - трекінг обличчя, аналіз емоцій та AR-сценарій;
- персоналізація контенту – рекомендаційні системи, що підлаштовуються під користувача;
- розпізнавання зображень чи об'єктів – класифікація продуктів, визначення об'єктів у реальному часі за допомогою камери.

Основним способом створення машинного навчання для IOS, є CoreML.

Core ML - це framework для роботи з технологіями машинного навчання, який компанія Apple Inc. представила на конференції WWDC у 2017 році. Дана технологія доступна, починаючи з iPhone 8 та iPhone 8 Plus. Для роботи Core ML не потребує інтернет з'єднання, через те, що модель даних запускається локально на пристрої користувача, що дозволяє зберігати конфіденційність даних.

Фреймворк дозволяє оптимізувати продуктивність пристрою та мінімізувати обсяг використаної пам'яті та енергоспоживання за рахунок використання ресурсів центрального та графічного

процесорів, а також Neural Engine [5].

В проєкті для інтеграції машинного навчання будуть використовуватися CoreML, CreateML та Vision.

Для створення самої моделі використовується CreateML. CreateML - це інструмент, що дозволяє навчати моделі безпосередньо на Mac у середовищі

Swift, або через графічний інтерфейс. CreateML може створювати моделі для класифікації зображень та тексту, також рекомендацій та регресії [6].

Vision - це високорівневий фреймворк від Apple, призначений для аналізу та обробки зображень і відео з використанням технології комп'ютерного зору (Computer Vision). Він був представлений у складі iOS 11 та є частиною екосистеми машинного навчання Apple, тісно інтегрований з CoreML, AVFoundation та Metal [7].

Фреймворк дозволяє розробникам виконувати складні завдання комп'ютерного зору локально на пристрої, використовуючи вбудовані або кастомні ML-моделі. Його API спрощує роботу за нейронними мережами, автоматично обробляючи етапи попередньої підготовки зображень, масштабування, нормалізації та конвертації даних у формат, придатний для моделі.

Vision використовує CoreML модель для розпізнавання зображень та об'єктів в реальному часі.

На основі класичного процесу машинного навчання, можна виділити основні пункти використання машинного навчання в iOS застосунках за допомогою екосистеми машинного навчання Apple:

- зібрати список продуктів та їх фотографії бажана долька в різних кольорах наприклад зелені яблука та червоні;
- обрізати фото та визначити координати об'єктів у спеціальному файлі;
- завантажити данні до CreateML та створити модель машинного навчання, потім додати цю модель до проєкту і використати CoreML для інтеграції моделі з бібліотекою Vision;
- перевірити точність моделі можна в CreateML;

- для перевірки на реальних даних використовується вже IOS застосунок.

1.4 Висновки за розділом

У першому розділі були розглянуті теоретичні основи, необхідні для розробки інтелектуального iOS-застосунку з використанням технологій комп'ютерного зору та машинного навчання. Проаналізовано ключові характеристики платформи iOS, її фреймворків і сучасних підходів до створення програмних продуктів. Особливу увагу приділено принципам роботи машинного навчання, типам моделей, етапам їх тренування та інтеграції у мобільні додатки.

Було досліджено можливості фреймворку CoreML, який забезпечує виконання ML-моделей безпосередньо на пристрої користувача, а також описано функціональність фреймворку Vision, що відповідає за обробку та аналіз зображень. Разом ці технології формують ефективний інструментарій для розпізнавання об'єктів у реальному часі, що є критично важливим для задачі розпізнавання харчових продуктів.

Узагальнюючи проведений аналіз, можна зробити висновок, що сучасна екосистема Apple забезпечує всі необхідні засоби для розробки високопродуктивних ML-рішень, зокрема для задач класифікації зображень. Теоретичні відомості та інструментарій, розглянуті в цьому розділі, створюють фундамент для подальшої практичної реалізації застосунку.

2 АНАЛІЗ ІСНУЮЧИХ ЗАСТОСУНКІВ АНАЛОГІВ. ВИБІР АРХІТЕКТУРИ ЗАСТОСУНКУ

2.1 Аналіз існуючих застосунків аналогів

Так як цілю проєкту є створення застосунку для впізнавання інгредієнтів та підбір кулінарних рекомендацій для споживача.

Розглядаються застосунки аналоги, в яких присутні такі елементи, як пошук рецептів на основі інгредієнтів та використання ML з камерою.

Основними критеріями для порівняння аналогічних застосунків є:

- якість користувацького інтерфейсу(UI/UX);
- швидкість та стабільність роботи;
- використання машинного навчання;
- зручність і точність розпізнавання продуктів;
- інтеграція з базами рецептів.

Перший у списку застосунків в App Store на запит рецептів з інгредієнтів, є застосунок з найменуванням Menu planner. Він є найближчим з застосунків аналогів до цілі проєкту, але є декілька нюансів. По-перше застосунок має дуже спірне рішення, що до агресивної монетизації, хоча для безкоштовних застосунків, це доволі популярна проблема. По-друге обробка зображення проходить не в режимі “в живу”, а відсилається на серверну сторону для обробки, що сповільнює користування застосунком. З плюсів в застосунку, є планування їжі на певний термін часу, також є спеціальні питання до користувача, щоб підібрати певні рецепти, до прикладу, якщо користувач вибере, що він веган, то йому не будуть видаватись рецепти з м'ясом.

За критеріями можна дати таку оцінку:

- доволі яскравий дизайн, але ж проблема з явною рекламою та платною підпискою;
- сам інтерфейс доволі швидко працює, але дуже повільна обробка зображень;
- використовується на серверній частині, більша база, але повільніша робота та неможливість роботи в режимі “в живу”;
- доволі непогана точність;
- велика база даних рецептів та спеціальні питання до користувача, які допомагають підібрати рецепти.

Ще існують декілька застосунків такі як Cookpad, Tasty. Вони дуже схожі між собою, але в них не використовується автоматичне розпізнавання завдяки ML. Але в них доволі є доволі значні плюси особливістю Cookpad, є соціальний аспект, що дозволяє ділитися рецептами між людьми в середині застосунки для забезпечення якості таких рецептів, є оцінка від інших користувачів. А особливістю Tasty є відеоінструкції по рецептам, що спрощує користувачам сприйняття інформації, що до рецепту.

Також є застосунки, які сканують вже готові страви та підраховують кількість калорій та білків, жирів, вуглеводів. Прикладами таких застосунків є SnapDish та FoodVisior.

SnapDish має фокус на розпізнаванні готових страв за зображенням. Фреймворк Vision дозволяє виділити ключові ознаки страви, а CoreML класифікує зображення. Має добру точність, але не вміє аналізувати окремі інгредієнти для побудови кулінарних вподобань.

Оцінка по критеріям така:

- доволі посередній UI;
- інтерфейс доволі стандартний та працює без проблем;
- використовується CoreML та Vision;

- точність непогана але працює тільки з повноцінними стравами;
- не дає кулінарних рекомендацій.

FoodVisior один з найтехнологічніших застосунків, який аналізує фото їжі та пропонує дієтичні рекомендації. Розроблений із використанням глибоких нейронних мереж та інтегрований з CoreML. Відзначається високою точністю, однак орієнтований більш на фітнес харчування, ніж на кулінарію.

По критеріям така оцінка:

- доволі непоганий UI, але є проблеми з дуже великим стартом з багатьма питаннями та рекламою платної підписки;
- дуже швидка та стабільна робота;
- частина застосунку використовує CoreML та частина працює через серверну частину;
- хороша точність, але працює лише на готові страви;
- база даних невелика і більш зосереджена на фітнес харчуванні.

На основі проведеного аналізу можна зробити висновок, що найбільш ефективними у використанні технологій ML та CoreML є застосунки FoodVisior та SnapDish, які реалізують глибокі нейронні моделі для обробки зображень безпосередньо на пристрої.

Більшість популярних застосунків зосереджуються або на соціальній взаємодії (Cookpad), або на відеоконтенті (Tasty), залишаючи нішу для програм, що поєднують розпізнавання продуктів і рекомендаційні системи.

Таким чином проект є унікальним в своїй ніші. Найближчий конкурент Menu planner має деякі мінуси, які можливо відремонтувати завдяки Vision та CoreML.

У таблиці 2.1 наводиться порівняння аналогів розроблюваного застосунку.

Таблиця 2.1 – Порівняльна таблиця застосунків аналогів

Критерій	Menu Planner	Cookpad	Tasty	SnapDish	Foodvisor
UI/UX	Яскравий дизайн, але агресивна монетизація та плутана підписка	Добрий інтерфейс, соціальна взаємодія з іншими користувачами	Гарний UI, відеоінструкції полегшують роботу з рецептами	Посередній UI, мінімалістичний стиль	Непоганий UI, але є довгий вступ і реклама підписки
Швидкість роботи	Інтерфейс швидкий, але обробка фото повільна (серверна)	Загалом швидка робота	Швидкий UI, відео працюють стабільно	Швидкий та стабільний застосунок	Дуже швидка та стабільна робота
Обробка зображень	Фото надсилаються на сервер, немає режиму “в живу”	Немає ML, рецепти вводяться вручну користувачами	Немає ML, фокус на відеоконтенті	Використання CoreML + Vision для розпізнавання страв	Глибокі нейронні моделі для аналізу їжі, частково через сервер, частково локально
Тип розпізнавання	Розпізнавання інгредієнтів (повільне)	Немає	Немає	Розпізнає цілі страви, але не окремі продукти	Розпізнає готові страви та рахує калорії, але не фокусується на кулінарії
Точність ML	Доволі непогана (серверні моделі)	Немає ML	Немає ML	Непогана точність на рівні страв, але не інгредієнтів	Висока точність, одна з найкращих моделей на ринку
База рецептів	Велика, є персоналізація через питання користувачеві	Велика завдяки активній спільноті користувачів	Середня, акцент на відео	Невелика, фокус на фото страв і соц. функціях	Невелика, орієнтація на фітнес-харчування
Особливості	Персональні питання для підбору рецептів; погана монетизація	Соціальна мережа рецептів, оцінки та відгуки	Відеорецепти, зручність подачі	Працює лише з готовими стравами, не аналізує інгредієнти	Модель для підрахунку калорій та БЖВ, орієнтація на дієти

2.2 Вибір архітектури застосунку

Основним етапом розробки IOS застосунку, є вибір архітектури, яка визначає структуру коду, взаємодію компонентів і підтримку коду у належному стан у майбутньому. Для iOS розробки існує декілька архітектурних рішень таких як MVC, MVP, MVVM, VIPER, REDUX та Clean Architecture.

MVC – є класичним архітектурним паттерном, який був базовим в IOS застосунках оснований на UIKit [8].

Основними елементами, є:

- Model - відповідає за бізнес логіку та є основним сховищем даних;
- View – відповідає за те, що бачить користувач;
- Controller – є посередником між Model та View. Переробляє данні користувача та оновлює Model та View.

Якщо брати реалізацію цього паттерну від Apple (рис. 2.1).

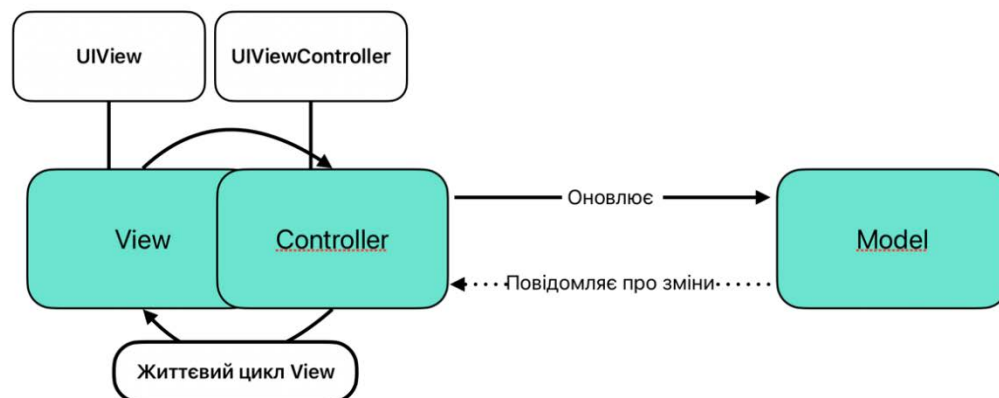


Рисунок 2.1 – MVC архітектура від Apple

Він доволі простий в основі, але має велику проблему з тим, що ViewController починає дуже сильно розширюватися і забирати на себе занадто багато, наприклад UI та бізнес логіку. Також View та View Controller мають сильний зв'язок і не можуть тестуватися окремо, тому тестування у такій реалізації доволі складне. Але існує можливість створення MVC з вирішенням проблеми величезного ViewController та можливістю тестувати. Це створення MVP. MVP – це шаблон проектування програмного забезпечення, який зазвичай використовується для організації коду в чистий та зрозумілий спосіб [9]. Основна структура MVP (рис. 2.2), де:

- Model - відповідає за бізнес логіку та є основним сховищем даних;
- Presenter - який займається логікою Controller;
- Passive View - що насправді є ViewController та View одночасно.

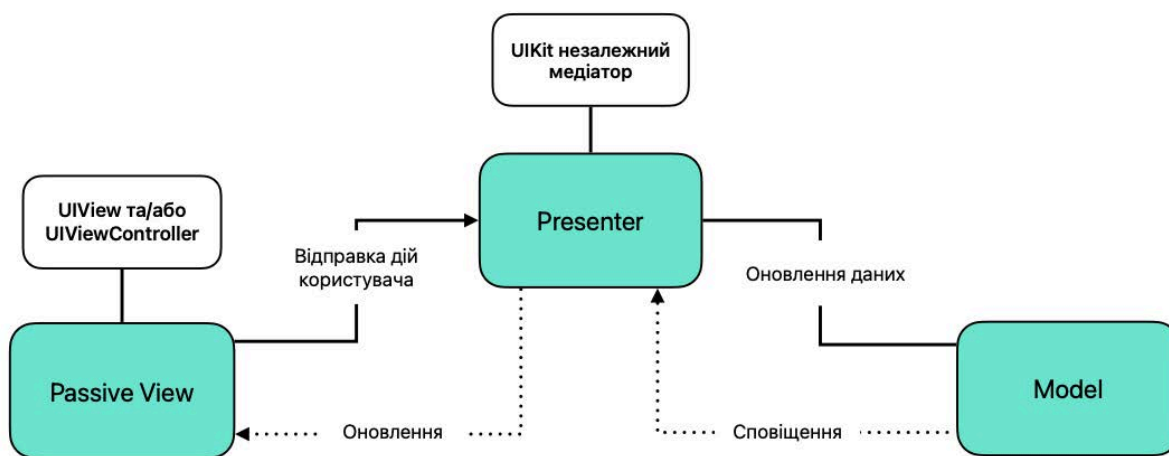


Рисунок 2.2 – MVP архітектура

Ця модульність дозволяє тестувати Presenter та Model, створюючи Mock об'єкт. Mock об'єкт – це об'єкти, що імітують поведінку справжніх об'єктів контрольованими способами, тобто, реалізують інтерфейси справжніх об'єктів, але не мають власної реальної функціональності [10]. З плюсів MVP

можна виділити простоту використання, тестувальність та розподіл обов'язків. Але один з недоліків цього підходу, є відсутність реактивного програмування, яке спрощує розробку та не дуже працює з SwiftUI, через те що SwiftUI використовує Combine та реактивні підходи. В свою чергу MVVM в своїй основі використовує реактивний підхід. Реактивний підхід оснований на патерні Спостерігач, або більш відомий як Observer. Це поведінковий патерн проектування, який створює механізм підписки, що дає змогу одним об'єктам стежити й реагувати на події, які відбуваються в інших об'єктах [11].

MVVM (рис. 2.3) має такі модулі:

- Model - яка займається бізнес логікою;
- View - відповідає за UI елементи та дії користувача, що відправляються по підписці до ViewModel;
- ViewModel - слухає зміни в Model та змінює View.

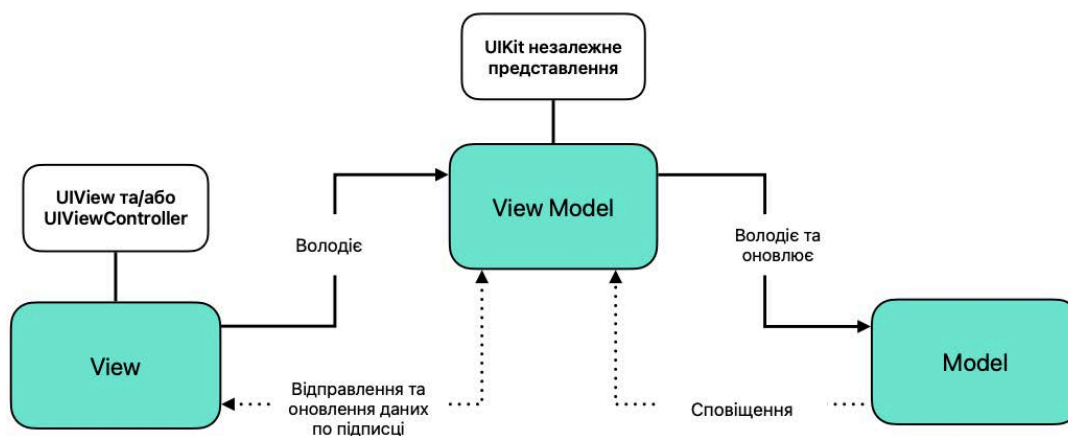


Рисунок 2.3 – MVVM архітектура

З плюсів є гарна тестувальність та швидкість розробки з SwiftUI. З мінусів не підходить до великих проєктів з багатьма екранами бо відсутня координація, хоча це вирішується додаванням, ще одного модуля Coordinator

який займається координацією в застосунку і звільняє частину логіки з ViewModel.

Для великих проектів існує VIPER (рис. 2.4) він вже не є MV архітектурою і є більш складною реалізацією архітектурного паттерну. Він також використовує реактивний підхід. І основною різницею з MV паттернами, є розділення модулю Model на Interactor та Entity, що ускладнює розробку але спрощує масштабування проекту [12]. Viper має таку структуру:

- View – займається UI елементами та відправляє дії користувача до Presenter;
- Presenter – збирає данні зі View та слухає Interactor;
- Router – займається координацією в проєкті;
- Interactor – займається бізнес логікою та змінює Entity;
- Entity – чисті данні.

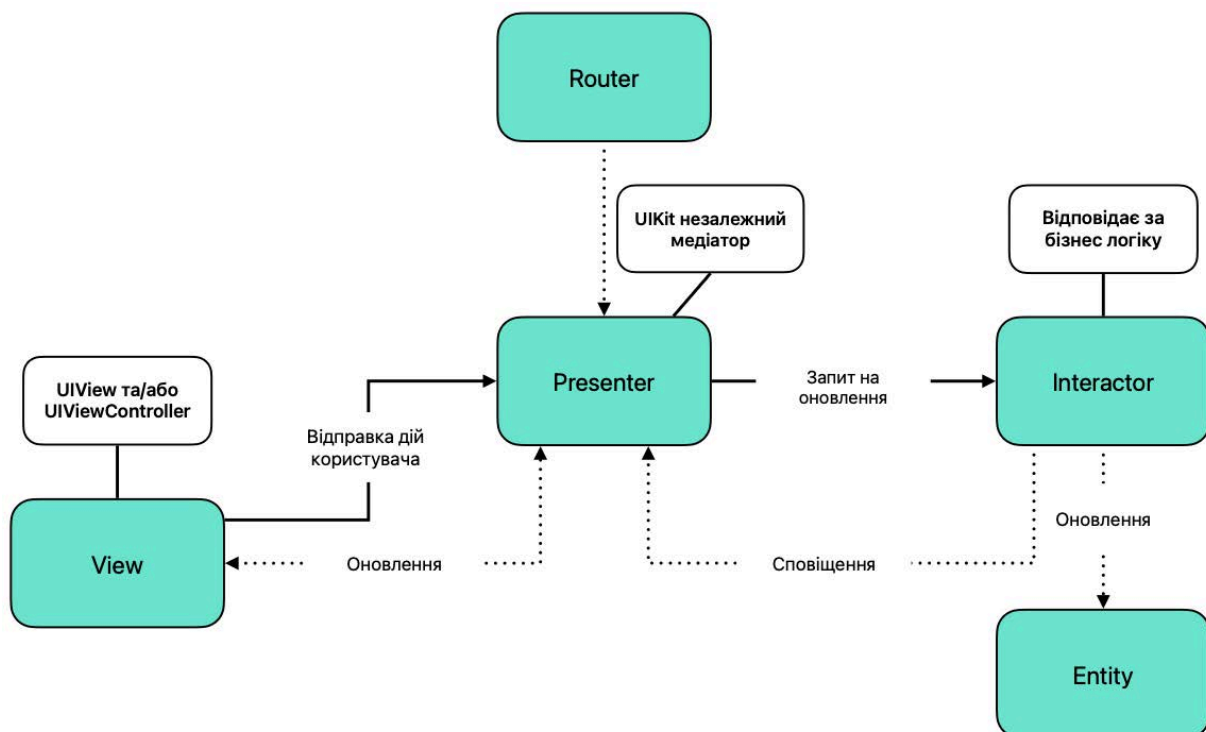


Рисунок 2.4 – Viper архітектура

З плюсів велика модульність та висока тестувальність, але доволі складна розробка. Підходить до великих комерційних продуктів, але для застосунку з розпізнаванням продуктів не дуже раціональний вибір.

Також існує Clean (рис. 2.5) архітектура, яка використовується як доповнення та заміщує в MV модуль в Model на UseCase та Entities. Цю архітектуру частно описують в колі, де Entities знаходяться посередині потім йдуть UseCase, а потім інтерфейси і в кінці бібліотеки. Вся залежність йде до центру кола [13].

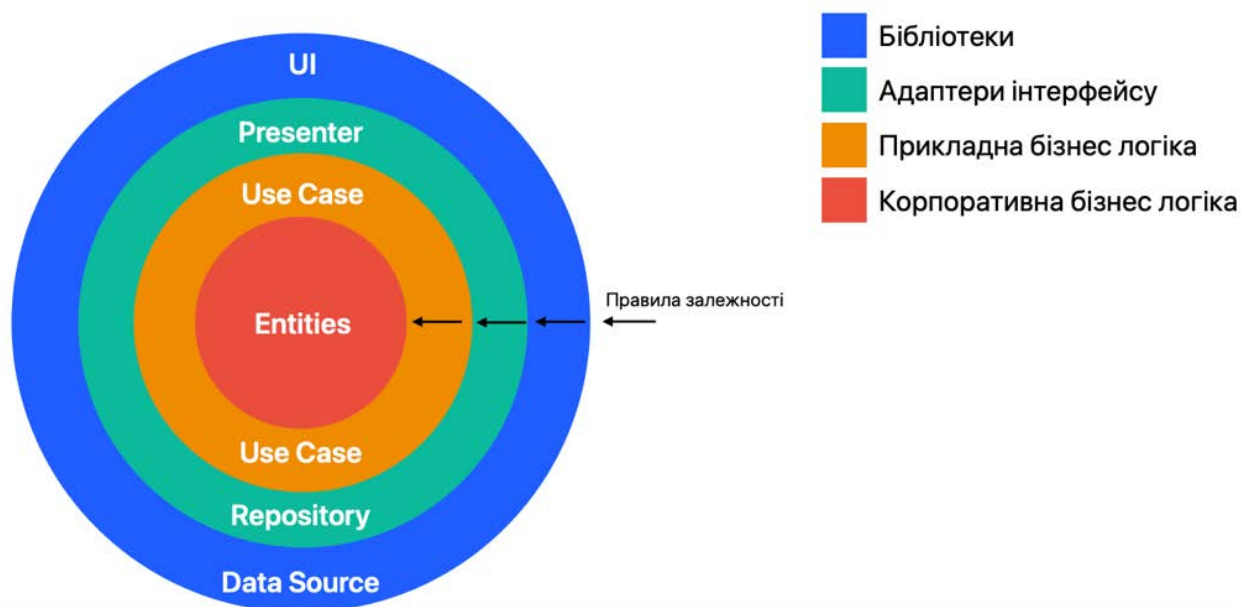


Рисунок 2.5 – Clean

Доволі складна архітектура для освоєння, але розробка с часом стає доволі швидкою. Має гарну тестувальність і варіативність у підходах.

У таблиці 2.2 наведено порівняння архітектур для побудови iOS застосунку.

Таблиця 2.2 – Порівняння архітектур

Критерії	MVC	MVP	MVVM	VIPER	Clean	Redux
1	2	3	4	5	6	7
Складність	1	2	3	4	4	4
Розділення відповідальностей	Низьке	Середнє	Високе	Дуже високе	Дуже високе	Максимальне
Тестованість	Низька	Середня	Висока	Дуже висока	Дуже висока	Дуже висока
Придатність для великих проєктів	Немає	Часткова	Так	Так	Так	Так
Керування станом	Немає	Презентер керує логікою	Реактивне через Bindings	Модульно через Interaptor	Чіткі межі між Data та Domain	Єдине джерело правди(Store)
Кількість файлів	Мала	Середня	Середня	Висока	Дуже Висока	Середня
Швидкість розробки	Найшвидша	Швидка	Середня	Низька	Низька	Висока

Кінець таблиці 2.2.

Переваги	Простота, швидкість	Краща структурування коду	Баланс простоти і чистоти	Чіткі модулі, тестованість	Чіткі шари, чистота логіки	Прозорий state flow, реактивність
Недоліки	Massive VC	Presenter виростає	ViewModel може “роздуватись”	Багато файлів, складність	Занадто важка структура	Важко дебажити великі сторінки стану
Оптимальне використання	Маленькі проекти	Середні проекти	Середні/великі проекти	Корпоративні великі продукти	Дуже великі системи	Реактивні, ML, SwiftUI-проекти

Одним з нових архітектурних патернів є Redux. Redux має основний елемент Store, що є основним джерелом для усього застосування, це дозволяє його використовувати й на невеликих проектах і на більш об’ємних. Redux (рис. 2.6) використовує в своїй основі реактивний підхід та патерн State. State - це поведінковий патерн проектування, що дає змогу об’єктам змінювати поведінку в залежності від їхнього стану [14]. Ззовні створюється враження, ніби змінився клас об’єкта.

Redux складається:

- View – UI частина яка відправляє Action;
- Action – це дія яку отримує Store;
- State – модель даних яку слухає View;
- Store – основне сховище яке в собі зберігає State та Reducer;

- Reducer – бере Action та змінює State;
- Middleware або SideEffect – блок який займається асинхронними діями які потребують час по закінченню відправляє Action до Store.

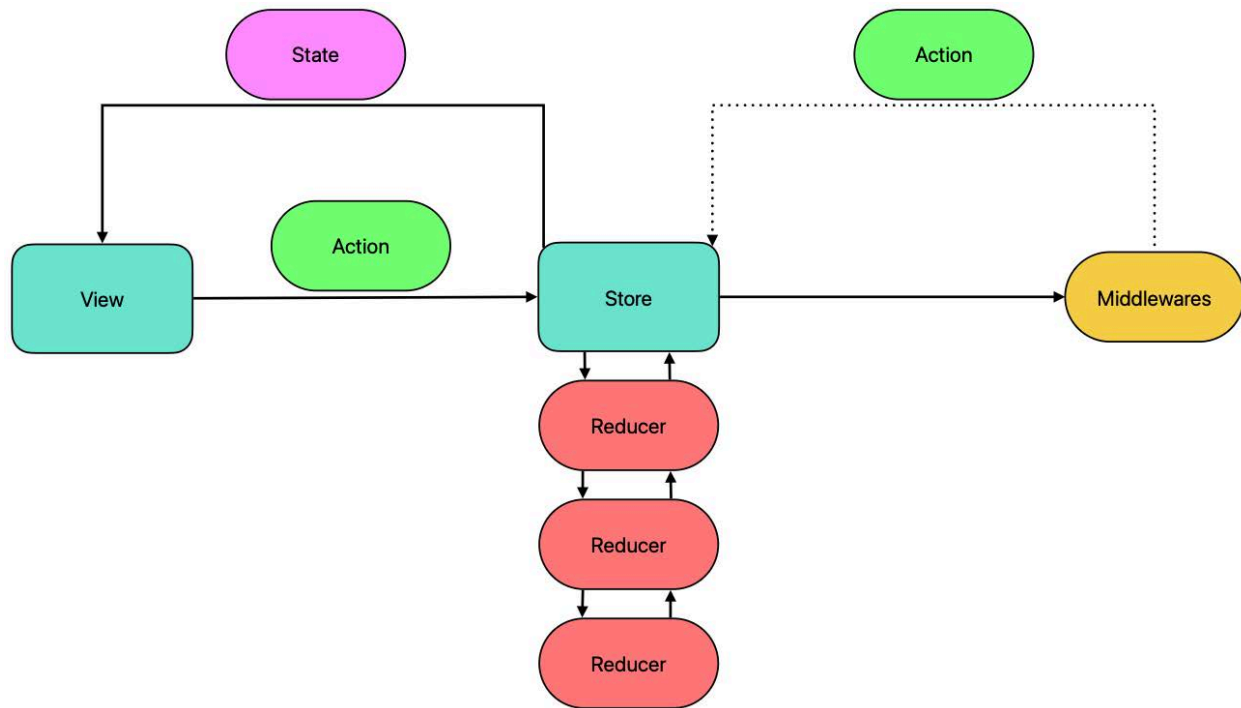


Рисунок 2.6 – Redux архітектура

З плюсів доволі швидка розробка з високою тестувальністю, непогана взаємодія зі SwiftUI. Але доволі складна для розуміння архітектура з багатьма різними реалізаціями.

Якщо порівнювати усі архітектури, то для застосунку з розпізнавання продуктів найкраще підходять MVVM з Coordinator та Redux.

Архітектура Redux є більш сучасною, структурованою та орієнтованою на уніфікацію потоків даних. Її головна перевага полягає у повній передбачуваності стану застосунку.

Такий підхід значно спрощує тестування, дебагінг і дозволяє уникнути складних сценаріїв багатопотокової взаємодії, що є актуальним при роботі з машинним навчанням, обробкою зображень та асинхронними подіями.

Попри свою простоту, MVVM із Coordinator залишається чудовим вибором для невеликих або середніх застосунків, де потрібно швидко інтегрувати UI з бізнес-логікою та навігацією.

Coordinator у такому випадку відповідає за маршрутизацію між екранами, що забезпечує кращу модульність та розділення відповідальностей.

Проте при масштабуванні застосунку MVVM може ускладнюватися через велику кількість двосторонніх зв'язків між View та ViewModel, особливо якщо застосунок має численні асинхронні процеси, як-от передача даних між Vision, CoreML і мережею.

Також розглядався варіант MVVM + Clean Architecture, який дозволяє отримати максимальну модульність, чітке розділення шарів і гнучкість у зміні джерел даних.

Однак цей підхід вимагає значного обсягу коду, підвищує складність інтеграції та не завжди виправдовує себе в невеликих мобільних проєктах, де швидкість розробки є критичною.

У підсумку вибір пав на Redux, який чудово масштабується, легко тестується і природно поєднується з реактивними фреймворками, такими як Combine.

Redux забезпечує єдине джерело істини (Single Source of Truth) для всіх даних у застосунку, що робить роботу з результатами розпізнавання стабільною та передбачуваною.

Це особливо важливо, коли стан системи формується внаслідок кількох одночасних процесів:

- зйомки зображення або відеопотоку з камери;

- розпізнавання продукту через Vision/CoreML;
- отримання додаткових даних із бази рецептів;
- формування рекомендацій для користувача.

Для використання реактивного програмування існує декілька підходів [15]:

- створення свого патерну Observer;
- використання сторонніх бібліотек таких як RxSwift та ReactiveSwift;
- Combine нативний реактивний фреймворк від Apple.

З усіх варіантів найбільше підходить Combine так як він є нативною бібліотекою Apple і його ключова перевага — повна інтеграція з системними фреймворками SwiftUI, Foundation, URLSession, CoreData, Vision, CoreML та іншими API.

2.3 Висновки за розділом

У розділі здійснено аналіз існуючих застосунків-аналогів, що використовують розпізнавання продуктів, визначення харчової цінності або формування рекомендацій. Визначено їх основні переваги, недоліки та обмеження. Особливу увагу приділено тому, як різні рішення інтегрують ML-моделі, які підходи до UX використовуються та наскільки зручні вони для користувача. Проведений аналіз дозволив сформулювати вимоги до функціоналу майбутнього застосунку.

Також було виконано детальний огляд архітектур iOS-застосунків, зокрема MVC, MVVM, VIPER, Clean Architecture та Redux. Для кожної архітектури описано переваги, недоліки та придатність до використання в ML-орієнтованих мобільних проєктах. У результаті порівняння встановлено, що саме архітектура Redux у поєднанні з реактивним програмуванням на базі

Combine і декларативним інтерфейсом SwiftUI є оптимальним вибором для реалізації стабільного, масштабованого та передбачуваного застосунку.

Таким чином, розділ 2 дав змогу сформуванню архітектурний фундамент майбутнього рішення та визначити структуру взаємодії компонентів. Отримані результати забезпечують технічну цілісність та обґрунтовують вибір технологічного стеку, який буде застосований у процесі реалізації застосунку.

3 РОЗРОБКА IOS ЗАСТОСУНКУ ДЛЯ СКАНУ ІГРЕДИЄНТІВ ТА ВИДАЧІ КУЛІНАРНИХ РЕКОМЕНДАЦІЙ

3.1 Створення IOS застосунку

Для створення IOS додатку використовується Xcode IDE. Створення починається з вибору платформи та типу проекту (рис. 3.1). Для проекту буде використовуватися ios платформа та звичайний App шаблон.

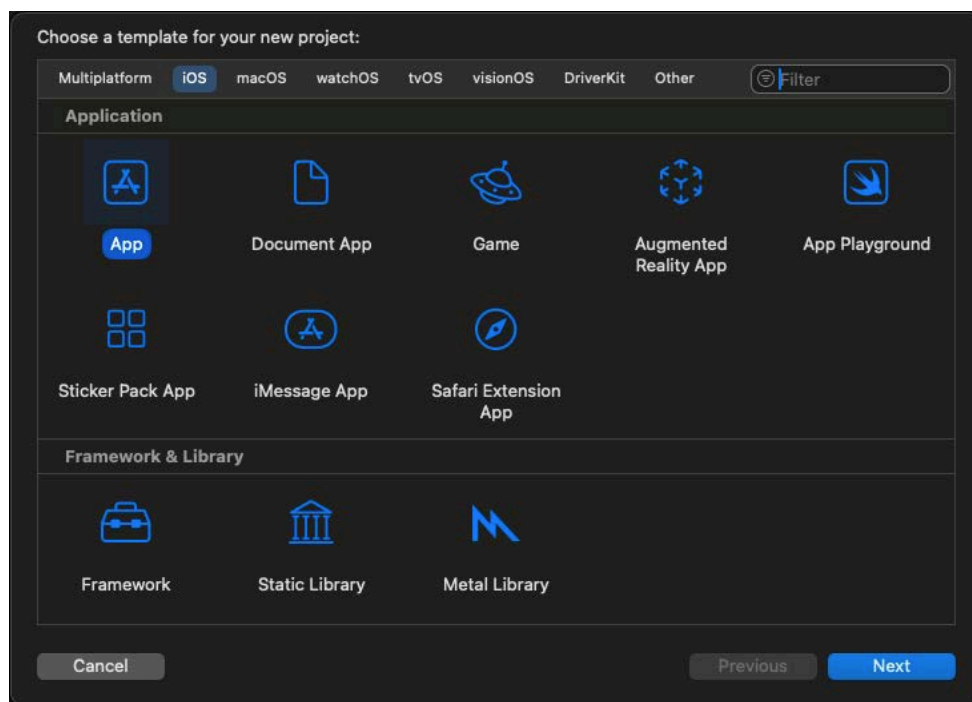


Рисунок 3.1 – Xcode вікно створення проекту

Потім з'являється вікно вибору технологій та мови програмування (рис. 3.2). Для проекту використовується інтерфейс SwiftUI та мова Swift. Також використовується моя персональна команда, створюється вона з Apple Developer аккаунту, це потрібно для тестування на пристрої бо камера не працює в симуляторі.

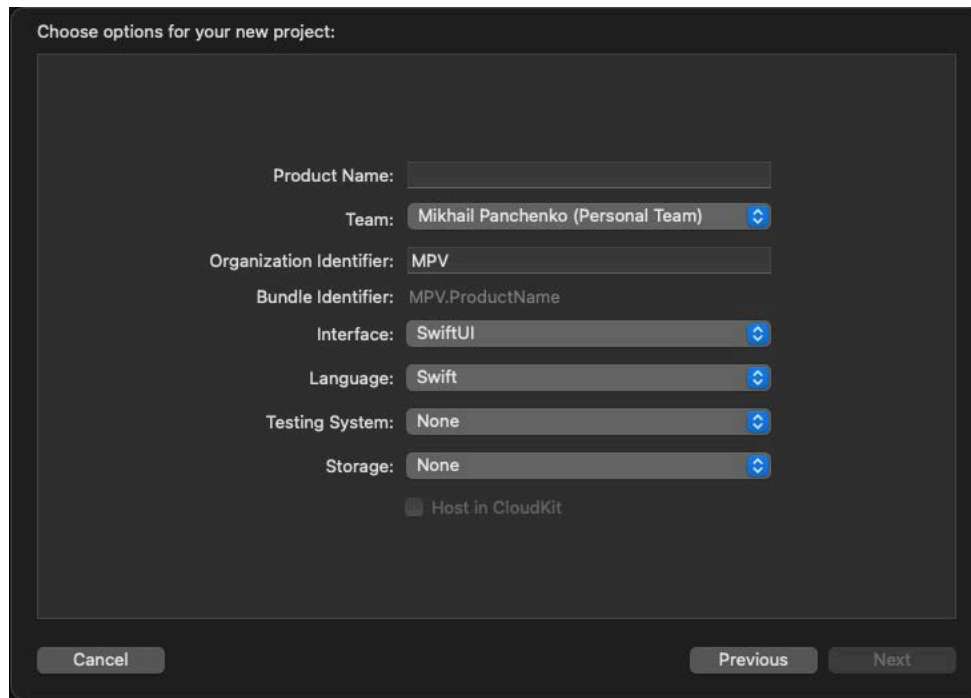


Рисунок 3.2 – Xcode вікно вибору технологій та мови програмування

Після створення проекту починається налаштування та розробка базових елементів та архітектури (рис. 3.3). По перше розроблені протоколи для Redux архітектури для її реалізації на їх основі будуть створенні Store, State, Reducer, Action та Middleware. Також для навігації між екранами створено Router він більш реалізований на стороні View, що суперечить архітектурі, але на даний момент в SwiftUI реалізувати навігацію іншим способом неможливо. Потім на SwiftUI були створенні UI сцени з камерою, списком рецептів та детальним оглядом рецепту.

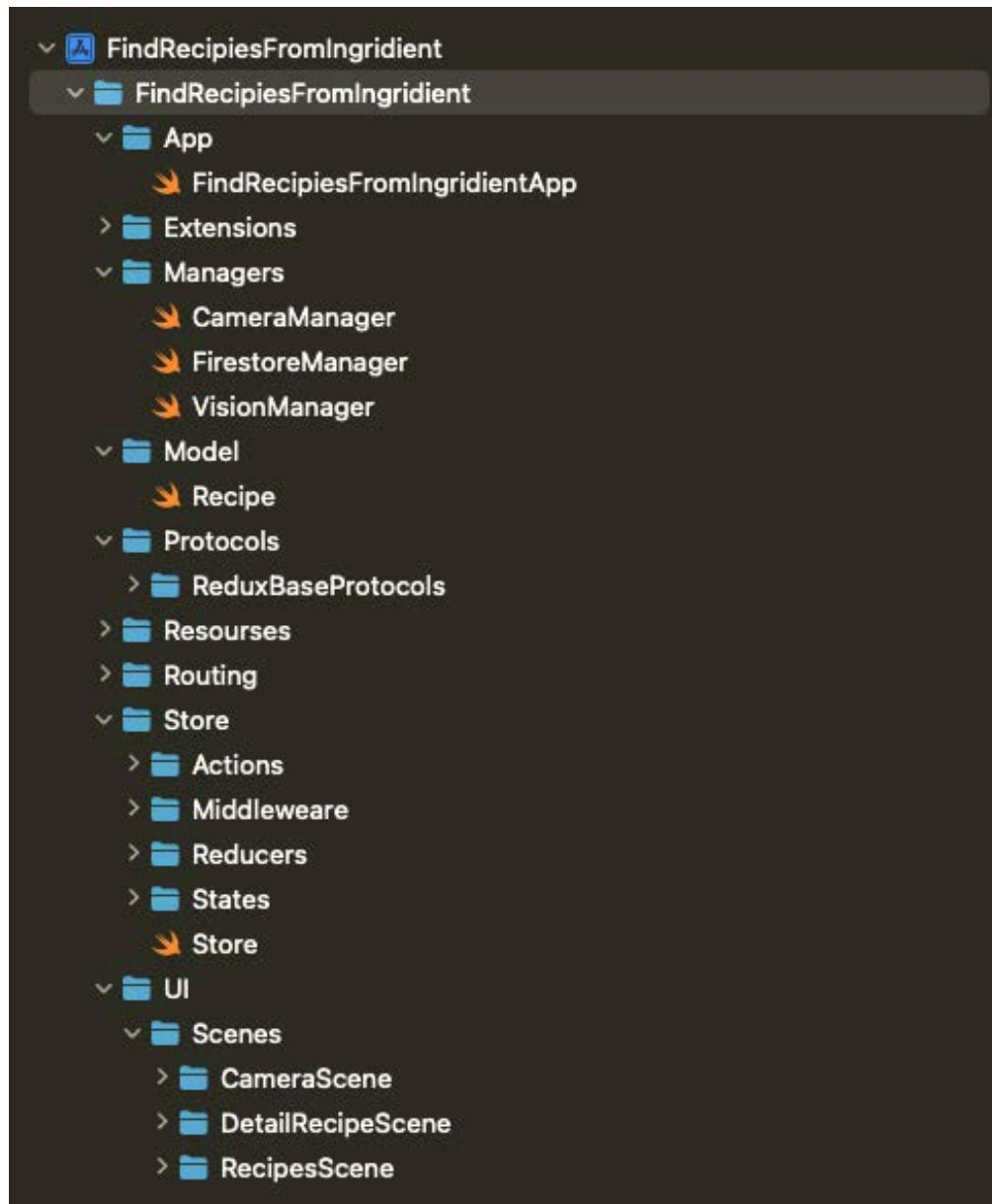


Рисунок 3.3 – Project inspector налаштованого проекту

3.2 Заповнення даних для бази рецептів та налаштування Firebase

В проекті Firebase буде використовуватися, як база даних на якій будуть розташовані рецепти, інгредієнти та спосіб приготування (рис. 3.4).



Рисунок 3.4 – Приклад двох рецептів з JSON бази

Базу даних було сформовано з 50 рецептів та завантажено у вигляді JSON в Realtime Database.

Firebase Realtime Database є хмарною NoSQL-базою даних, що забезпечує зберігання інформації у вигляді великого JSON-дерева та надає можливість синхронізації даних між клієнтами в режимі реального часу. На відміну від реляційних баз даних, Realtime Database не використовує таблиць або складних схем, що робить її особливо зручною для зберігання структурованих об'єктів, таких як рецепти чи інгредієнти. Завдяки тому, що дані представляються у форматі JSON, вони можуть бути напряму інтегровані в мобільний застосунок без додаткових перетворень або підготовки [16].

Однією з ключових особливостей Firebase Realtime Database є миттєва синхронізація оновлень. Будь-яка зміна, що відбувається у базі, негайно передається всім підключеним клієнтам через постійне сокет-з'єднання. Це дозволяє працювати з даними майже так само, як із локальним сховищем, не витрачаючи час на ручне оновлення або перезапити. Крім того, Firebase забезпечує офлайн-підтримку, кешуючи дані на пристрої та автоматично

синхронізуючи їх із сервером після відновлення з'єднання, що робить систему більш надійною у мобільних сценаріях використання.

Важливим фактором, що робить Realtime Database привабливою для iOS-розробників, є проста інтеграція через Firebase SDK. Розробник може під'єднати базу до застосунку за допомогою Swift Package Manager або CocoaPods та отримати готовий набір інструментів для читання, запису та спостереження за змінами в даних. API Firebase добре поєднується з сучасними підходами роботи у Swift, включаючи `async/await` та `Combine`, а структура JSON природно зв'язується з моделями через протокол `Codable`.

3.3 Оформлення Redux архітектури

Спочатку створюється Store в якому є `AppState`, `RootReducer`, масив `Middleware` та метод `dispatch`.

`AppState` відповідає за дані усього застосунку в ньому у свою чергу зберігаються різні State об'єкти, що відповідають за різні частини застосунку, наприклад `CameraState` який відповідає за дані, що потрібні для `CameraScene`. `AppState` обгортається `@Published`. `@Published` - це обгортка властивості з бібліотеки `Combine`, яка дозволяє View слухати зміни і вимальовувати нові дані на UI.

`RootReducer` в свою чергу відповідає за оновлення State завдяки Action та зберігає в собі інші Reducer, наприклад для екрану камери є Action `SwitchCameraRecognition` при його отриманні `CameraReducer` змінює поле `isDetecting` в `CameraState`. Все це проходить через метод `dispatch`, який використовує сама View, наприклад при натисканні кнопки «Почати розпізнавання» ми відправляємо `SwitchCameraRecognition`.

Найважливішим елементом роботи з асинхронним кодом є `Middleweare`. Вони опрацьовують `Action` та відправляють свій `EffectDispatch` котрий спрацьовує через деякий час та відправляє `Action`, наприклад коли показується екран з рецептами ми відправляємо `SelectIngridientAction`, який обробляє `RecipesListMiddleweare` та дістає результати пошуку з `Firebase`, за допомогою `FirebaseDatabaseManager`, відправляючи `SetRicipesListAction` зі списком рецептів. В свою чергу `RecipesListReducer` змінює `RecipesListState` додаючи новий список рецептів.

В готовій архітектурі (рис. 3.5) маємо виділені основні прошарки застосунку зеленим кольором позначено основний клас саме застосунку, там відбувається повне налаштування усіх початкових об'єктів архітектури. Жовтим позначено прошарок `View` який відповідає за візуальну частину та взаємодію користувача с застосунком. `View` елементи відправляють `Action` до `AppStore`. Синій прошарок це в нас основна бізнес логіка та реалізація `Redux` елементів. `AppStore` отримує, за допомогою метода `dispatch`, `Action` який перенаправляє в `RootReducer` та в масив `CameraMiddleweare` та `RecipesListMiddleweare`. `RootReducer` змінює `AppState` та відправляє `Action` у інші `Reducer`, які в свою чергу змінюють свої `State`. `Middleweare` отримують `Action` та створюють свій `Action` з даними від менеджерів, який направляється назад через час. Розовий прошарок – це менеджери котрі працюють з різними бібліотеками, наприклад `AVFoundation`, `Vision` тощо, та з базами даних наприклад `FirebaseDatabase`

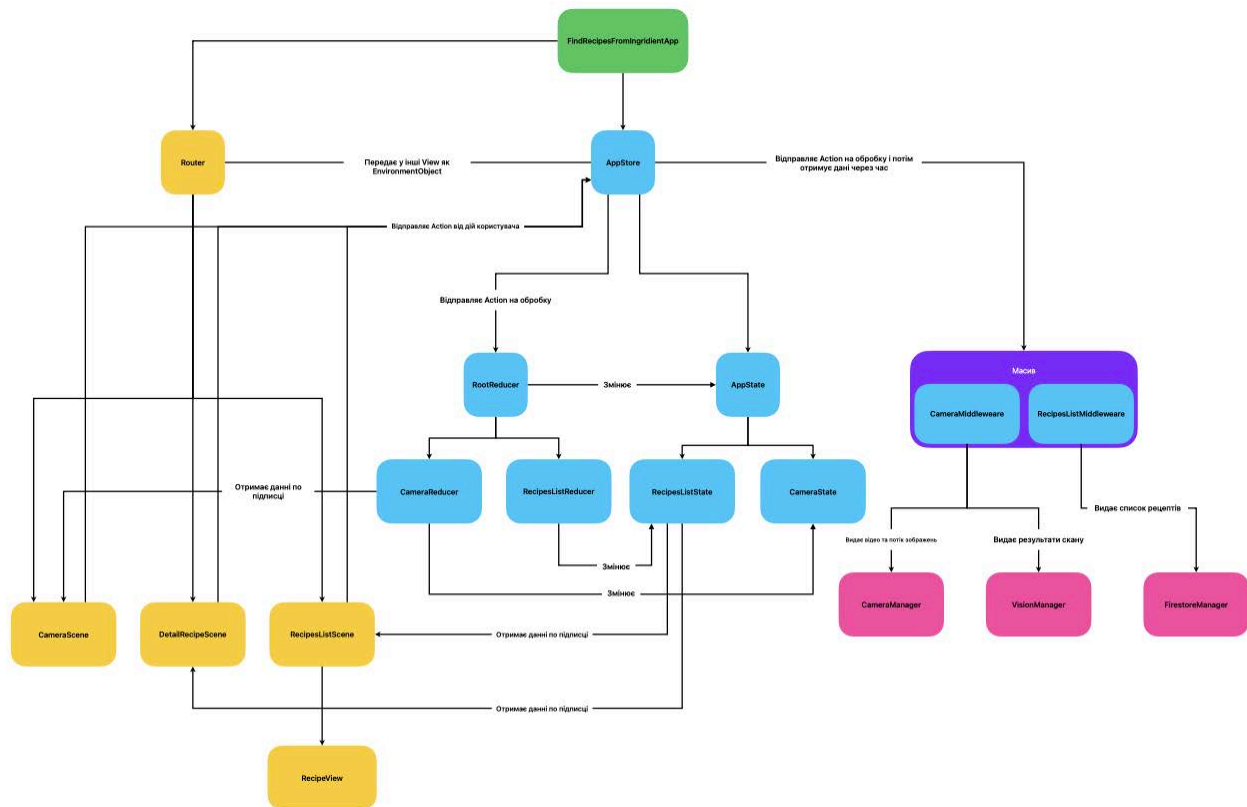


Рисунок 3.5 – Готова архітектура застосунку

3.4 Створення DataSet для CreateML

DataSet – це данні для навчання в моделі. Для CreateML в нього входять фото з об’єктами та їх координати з назвами в спеціальному JSON файлі. Для спрощення створення DataSet використовується сайт <https://app.roboflow.com>. Вже на сайті доступне виділення об’єктів за допомогою спеціальних інструментів (рис. 3.6).

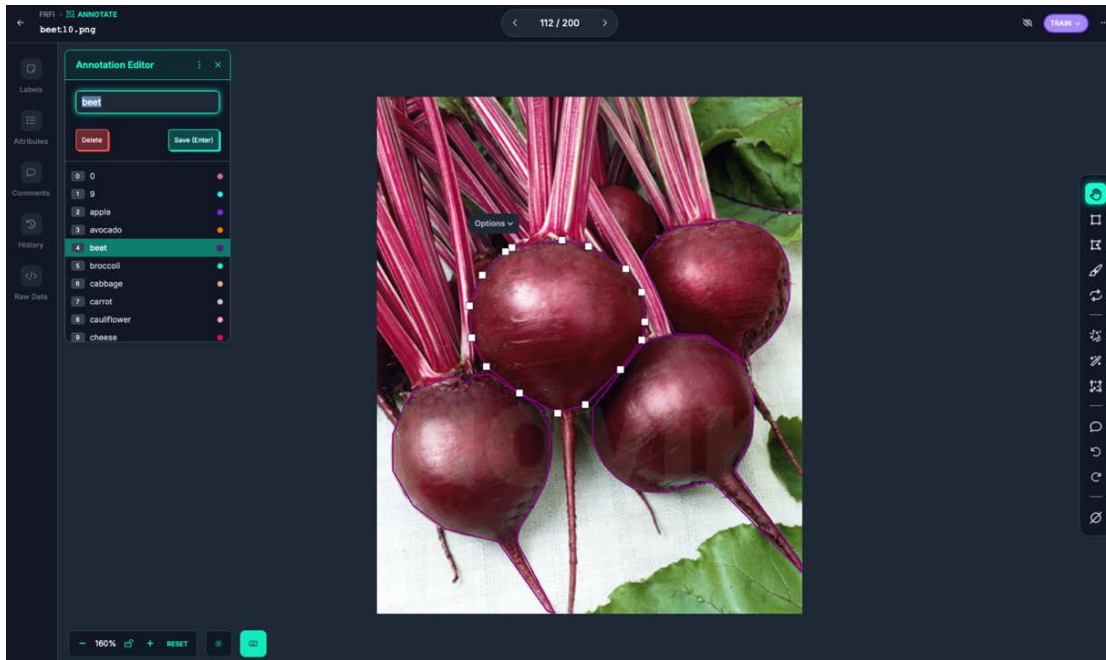


Рисунок 3.6 – Приклад виділення об'єкту на знімку

Для проекту було завантажено близько 248 фото, тобто на кожен інгредієнт вийшло 10 фото. Після виділення об'єктів можна імпортувати вибраний DataSet та зробити його більш функціональним за допомогою препроцесорних змін, наприклад додати до фото різні кути нахилу тим самим зробити більш чітку модель. Імпортувати модель можна у різні варіанти, але для проекту потрібно саме CreateML DataSet. Створений архів додається до CreateML та починається навчання моделі. На виході отримано модель .mlmodel та інтегровано до проекту (рис. 3.7).

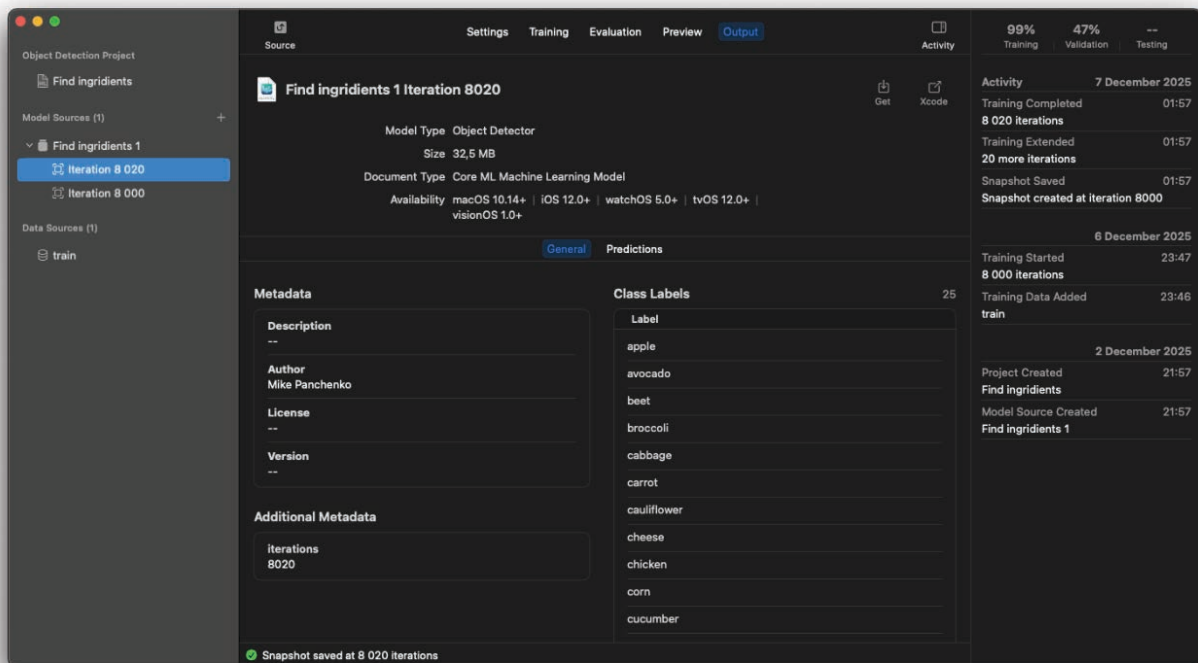


Рисунок 3.7 – Створена модель машинного навчання в CreateML

3.5 Інтеграція CoreML та Vision до проекту

До проекту додається файл `.mlmodel` і при першому Build створюється `Find_ingredients_1` клас, який інтегрується в Vision бібліотеку. Для повноцінної роботи з Redux архітектурою був створений менеджер який створює запит з CoreML моделі на Vision.

Також менеджер який відповідає за камеру відсилає в потоці теперішнє зображення, яке оброблює `VisionManager` та видає координати в `Middleware`. Потім `Middleware` передає `Action` в `State` з координатами та впевненістю в об'єктах і вже `View` їх відображає.

3.6 Діаграми сценаріїв роботи застосунку

Для використання додатку є декілька сценаріїв роботи. Перший (рис. 3.8) це коли користувач хоче дізнатися назву невідомого для користувача продукту, який знаходиться перед ним. Користувач відкриває застосунок наводить камеру на інгредієнт, натискає на кнопку почати розпізнавання та отримує квадратик з назвою невідомого продукту.



Рисунок 3.8 –Діаграма послідовності розпізнання продукту

Другий сценарій роботи (рис. 3.9) – це коли користувач бажає отримати рецепт з одного з продуктів, які є в його холодильнику. Користувач спочатку відкриває застосунок, потім наводить камеру на продукт починає розпізнавання. Коли продукт розпізнано людина натискає на квадратик, який знаходиться над продуктом та отримує перелік рецептів з бази даних. Після чого натискає на назву рецепта та бачить детальний рецепт з бази даних.



Рисунок 3.9 –Діаграма послідовності отримання рецепту

3.7 Висновки за розділом

У третьому розділі було детально розглянуто процес розробки iOS застосунку для сканування інгредієнтів та формування кулінарних рекомендацій. На початковому етапі описано створення проекту в середовищі Xcode із використанням платформи iOS, мови програмування Swift та інтерфейсу SwiftUI. Обґрунтовано вибір сучасних технологій розробки, що забезпечують високу продуктивність, декларативний підхід до побудови інтерфейсу та зручну інтеграцію з ML-фреймворками. Також зазначено необхідність використання Apple Developer аккаунту для тестування функціоналу камери на реальному пристрої.

У межах реалізації архітектури застосунку було впроваджено Redux-підхід, що дозволив чітко розділити відповідальності між станом застосунку, діями та логікою обробки. Описано структуру Store, AppState, Reducer та Middleware, а також показано, як за допомогою бібліотеки Combine забезпечується реактивне оновлення інтерфейсу при зміні стану. Особливу увагу приділено використанню Middleware для обробки асинхронних операцій, зокрема взаємодії з Firebase Realtime Database для отримання списку рецептів на основі розпізнаних інгредієнтів.

Окремо розглянуто процес створення DataSet для навчання ML-моделі з використанням CreateML. Описано підготовку набору даних, розмітку зображень інгредієнтів, застосування препроцесорних змін та навчання моделі. Отримана модель у форматі .mlmodel була успішно інтегрована в проєкт, що підтверджує доцільність використання CreateML для задач комп'ютерного зору в мобільних застосунках.

Завершальним етапом стала інтеграція CoreML та Vision у структуру застосунку. Було реалізовано механізм обробки зображень з камери в реальному часі, передачу результатів розпізнавання через Redux Middleware та відображення координат і рівня впевненості на інтерфейсі користувача. Такий підхід забезпечив чіткий потік даних від моменту отримання кадру до формування рекомендацій, що позитивно вплинуло на масштабованість та підтримуваність проєкту.

Таким чином, у третьому розділі було реалізовано повний цикл розробки iOS-застосунку — від створення проєкту та вибору архітектури до навчання ML-моделі й інтеграції CoreML та Vision. Отримане рішення підтверджує можливість ефективного використання сучасних інструментів екосистеми Apple для побудови інтелектуальних мобільних застосунків, орієнтованих на розпізнавання об'єктів і персоналізовані рекомендації.

4 ТЕСТУВАННЯ ЗАСТОСУНКУ ДЛЯ РОЗПІЗНАВАННЯ ІНГРЕДІЄНТІВ ТА ФОРМУВАННЯ КУЛІНАРНИХ РЕКОМЕНДАЦІЙ

4.1 Тестування ios застосунку для розпізнавання інгредієнтів та формування кулінарних рекомендацій

Тестування розробленого iOS-застосунку було проведено з метою перевірки коректності роботи функціоналу розпізнавання інгредієнтів, стабільності застосунку, точності ML-моделі та якості формування кулінарних рекомендацій. Особлива увага приділялася тестуванню в умовах, максимально наближених до реального використання, оскільки робота з камерою та комп'ютерним зором значною мірою залежить від зовнішніх факторів, таких як освітлення, фон та розташування об'єктів.

Для тестування було використано реальний iOS-пристрій із камерою, оскільки робота модуля розпізнавання не підтримується в симуляторі Xcode. Тестування проводилося у двох різних середовищах: у домашніх умовах та в магазині. Для кожного сценарію були записані відео з екрану пристрою, що демонструють роботу застосунку в режимі реального часу.

4.2 Тестування в домашніх умовах

Перший етап тестування здійснювався в домашньому середовищі з використанням реальних продуктів харчування, розміщених на різних поверхнях, таких як стіл або кухонна стільниця. У більшості випадків система правильно ідентифікувала продукти, які були присутні в навчальному DataSet, та відображала їх координати разом із рівнем впевненості (рис. 4.1).



Рисунок 4.1 – Приклад коректного розпізнавання інгредієнтів у домашніх умовах

Разом із тим під час тестування було зафіксовано випадки помилкового розпізнавання. Зокрема, застосунок іноді визначав неправильний тип продукту або відображав рамки розпізнавання в місцях, де фактично відсутні об'єкти. Такі помилки здебільшого виникали через недостатню кількість навчальних зображень, схожість форм або кольорів об'єктів, а також складний фон (рис. 4.2).



Рисунок 4.2 – Приклад помилкового розпізнавання продукту або появи рамки без об'єкта

4.3 Тестування в умовах магазину

Другий етап тестування проводився в магазині, де умови значно відрізняються від домашніх: велика кількість товарів, складний фон, різне освітлення та сторонні об'єкти в кадрі. Під час тестування застосунок

демонстрував нестабільні результати, що є типовим для задач комп'ютерного зору в реальних середовищах.

У деяких випадках система вірно розпізнавала продукти (рис. 4.3), які були представлені в DataSet, однак часто виникали помилкові спрацьовування (рис. 4.4). Застосунок міг визначати інгредієнти в руках, або навіть у фонових зонах, де відсутні будь-які релевантні об'єкти. Це свідчить про необхідність подальшого розширення навчальної вибірки та вдосконалення моделі.



Рисунок 4.3 – Приклад розпізнавання інгредієнтів у магазині



Рисунок 4.4 – Приклад помилкового визначення інгредієнтів у складному середовищі магазину

4.4 Аналіз результатів тестування

Тестування показало, що розроблений застосунок здатний працювати стабільно та виконувати розпізнавання інгредієнтів у реальному часі. У контрольованих умовах точність моделі є задовільною, однак у складних середовищах кількість помилок зростає. Основними причинами таких помилок є обмежений обсяг DataSet, схожість об'єктів між собою та складні фонові умови.

Попри зазначені недоліки, отримані результати підтверджують працездатність обраного підходу та правильність архітектурних рішень.

Подальше вдосконалення системи можливе шляхом збільшення кількості навчальних даних, покращення препроцесингу зображень та використання більш складних ML-моделей.

4.5 Взаємодія користувача с застосунком

Беручи другий сценарій взаємодії користувача с застосунком, користувач відкриває застосунок та наводить камеру на продукт (рис. 4.5).



Рисунок 4.5 – Відкриття застосунку

Потім користувач вмикає розпізнавання та бачить оранжевий квадрат з назвою продукту (рис. 4.6).

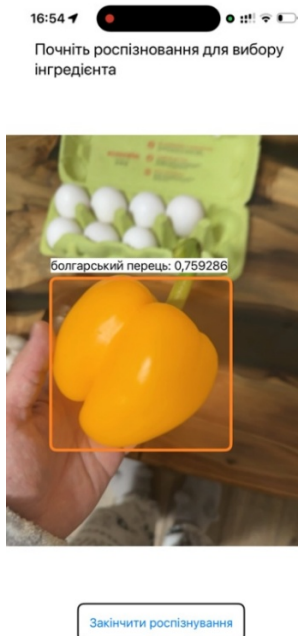


Рисунок 4.6 – Вдале розпізнавання продукту

Користувач натискає на квадрат та отримує список рецептів (рис. 4.7).

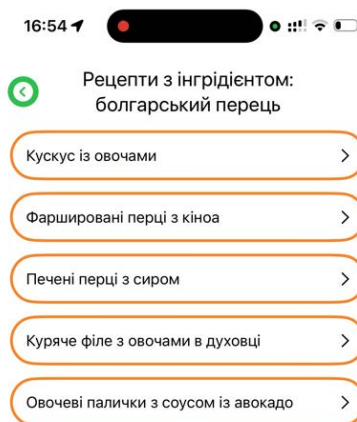


Рисунок 4.7 – Список рецептів

Вже в кінці користувач натискає на рецепт та відкриває детальний екран рецепту з описом страви та інгредієнти з їх кількістю (рис. 4.8).

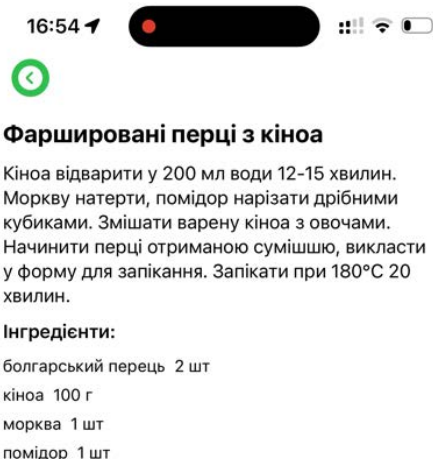


Рисунок 4.8 – Детальний екран рецепту

4.6 Висновок за розділом

Тестування застосунку в реальних умовах дозволило виявити як сильні сторони розробленого рішення, так і його обмеження. Отримані результати підтверджують доцільність використання CoreML та Vision для задач розпізнавання інгредієнтів, а також демонструють перспективність подальшого розвитку застосунку шляхом покращення якості навчальної вибірки та алгоритмів обробки зображень. Також застосунок показав доволі швидку роботу з базою даних та через те що модель машинного навчання знаходиться на самому мобільному пристрої, а не на серверній частині.

ВИСНОВКИ

У межах магістерської кваліфікаційної роботи було досліджено та реалізовано технологію розпізнавання харчових продуктів із використанням моделей машинного навчання для формування кулінарних рекомендацій у мобільному iOS-застосунку.

У ході виконання роботи проаналізовано сучасні підходи до розробки інтелектуальних мобільних застосунків на платформі iOS, а також можливості фреймворків CoreML, Vision, SwiftUI та Combine. Досліджено принципи побудови ML-рішень для задач комп'ютерного зору та визначено доцільність виконання розпізнавання об'єктів безпосередньо на пристрої користувача без залучення серверної інфраструктури.

Проведено аналіз існуючих застосунків-аналогів, що дозволило виявити їхні функціональні обмеження, зокрема відсутність роботи в режимі реального часу, залежність від серверної обробки або фокус на готових стравах замість окремих інгредієнтів. На основі цього сформовано вимоги до розроблюваного рішення та обґрунтовано вибір архітектури Redux як оптимальної для ML-орієнтованого застосунку з асинхронними процесами та складним керуванням станом.

У процесі практичної реалізації було розроблено iOS-застосунок із використанням мови програмування Swift та декларативного інтерфейсу SwiftUI. Реалізовано архітектуру Redux з єдиним джерелом істини для керування станом застосунку, що забезпечило передбачуваність поведінки, високу тестованість і зручність масштабування. Для асинхронної обробки подій та взаємодії з базою даних Firebase використано Middleware у поєднанні з реактивним фреймворком Combine.

Для задачі розпізнавання інгредієнтів було підготовлено набір навчальних даних, виконано розмітку зображень і навчання моделі машинного навчання за допомогою CreateML. Отриману ML-модель інтегровано в застосунок із використанням CoreML та Vision, що дало змогу реалізувати розпізнавання продуктів у реальному часі з відображенням результатів безпосередньо поверх зображення з камери.

Проведене тестування в домашніх умовах та в магазині підтвердило коректність роботи застосунку, стабільність розпізнавання інгредієнтів і логічність формування кулінарних рекомендацій на основі отриманих результатів. Застосунок продемонстрував достатню точність розпізнавання та зручність користування в реальних сценаріях.

Таким чином, поставлену мету магістерської роботи досягнуто. Розроблено функціональний прототип iOS-застосунку, який поєднує комп'ютерний зір, машинне навчання та сучасні архітектурні підходи для спрощення процесу пошуку кулінарних рецептів.

Практичне значення отриманих результатів полягає у можливості використання розробленого рішення як основи для створення комерційних продуктів у сфері кулінарних рекомендацій, персоналізованого харчування або фуд-аналітики. Подальший розвиток роботи може бути спрямований на розширення набору розпізнаваних інгредієнтів, підвищення точності ML-моделі, додавання персоналізації рекомендацій, підтримку декількох інгредієнтів одночасно та інтеграцію з дієтичними або медичними сервісами.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Swift мова програмування [Електронний ресурс]. – Режим доступу: [https://uk.wikipedia.org/wiki/Swift_\(%D0%BC%D0%BE%D0%B2%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F\)](https://uk.wikipedia.org/wiki/Swift_(%D0%BC%D0%BE%D0%B2%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F))
2. Xcode IDE [Електронний ресурс]. – Режим доступу: <https://uk.wikipedia.org/wiki/Xcode>.
3. Введення в UIKit [Електронний ресурс]. – Режим доступу: <https://foxminded.ua/uikit-tse>.
4. Машинне навчання [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/Машинне_навчання.
5. Apple CoreML [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/Apple_CoreML#:~:text=Core%20ML%20—%20це%20framework%20для,8%20та%20iPhone%208%20Plus.
6. CreateML [Electronic resource]. – Mode of access: <https://developer.apple.com/machine-learning/create-ml>.
7. Vision framework [Electronic resource]. – Mode of access: <https://www.hackingwithswift.com/example-code/vision>.
8. MVC architecture pattern [Electronic resource]. – Mode of access: <https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html>.
9. Swift MVP a step by step guide for clean code [Electronic resource]. – Mode of access: <https://medium.com/@gbrigens/swift-mvp-a-step-by-step-guide-for-clean-code-869ec8eb4b22>.
10. Mock object [Electronic resource]. – Mode of access: https://en.wikipedia.org/wiki/Mock_object.

11. An overview of the MVVM design pattern in swift [Electronic resource].
– Mode of access: <https://medium.com/clean-code-channel/an-overview-of-the-mvvm-design-pattern-in-swift-85627f2bd2b7>.

12. Getting started with the Viper architecture pattern [Electronic resource].
– Mode of access: <https://www.kodeco.com/8440907-getting-started-with-the-viper-architecture-pattern>.

13. Clean architecture in ios development a comprehensive guide [Electronic resource]. – Mode of access: <https://medium.com/@dyaremyshyn/clean-architecture-in-ios-development-a-comprehensive-guide-7e3d5f851e79>.

14. Getting started with redux in swift [Electronic resource]. – Mode of access: <https://medium.com/mackmobile/getting-started-with-redux-in-swift-54e00f323e2b>.

15. Hello Combine [Electronic resource]. – Mode of access: <https://www.kodeco.com/books/combine-asynchronous-programming-with-swift/v1.0/chapters/1-hello-combine>.

16. Realtime Database [Electronic resource]. – Mode of access: <https://firebase.google.com/docs/database>.

ДОДАТОК А
Текст програми

```
// FindRecipiesFromIngridientApp.swift
```

```
import SwiftUI
import FirebaseCore
```

```
class AppDelegate: NSObject, UIApplicationDelegate {
    func application(_ application: UIApplication,
                     didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey : Any]?
                     = nil) -> Bool {
        FirebaseApp.configure()
        return true
    }
}
```

```
typealias AppStore = Store<AppState, RootReducer>
```

```
@main
struct FindRecipiesFromIngridientApp: App {
    private var store: AppStore = {
        let isDetectingDefault: Bool = false
        return Store(
            initialState: AppState(cameraState: CameraState(isDetecting: isDetectingDefault),
                                recipesListState: RecipesListState()),
            rootReducer: RootReducer(cameraReducer: CameraReducer(),
                                    recipesListReducer: RecipesListReducer()),
            middlewares: [CameraMiddleware(isDetecting: isDetectingDefault),
                        RecipesListMiddleware()])
    }()
}
```

```
@UIApplicationDelegateAdaptor(AppDelegate.self) var delegate

var body: some Scene {
    WindowGroup {
        rootView().environmentObject(store)
    }
}
```

```
// ReduxStore.swift
import Foundation
```

```
public protocol ReduxStore: ObservableObject {
    associatedtype State: ReduxState

    var state: State { get }

    func dispatch(_ action: ReduxAction)
}
```

```

// ReduxState.swift

public protocol ReduxState {

}

// ReduxReducer.swift

public protocol ReduxReducer {
    associatedtype State: ReduxState

    func reduce(state: State?, action: ReduxAction?) -> State
}

// ReduxMiddleware.swift

protocol ReduxMiddleware {

    func dispatch(state: ReduxState,
                  action: ReduxAction,
                  effectDispatch: @escaping (ReduxAction) -> Void)
}

// ReduxAction.swift

public protocol ReduxAction {

}

// Store.swift

import Foundation
import Combine

class Store<AppState, RootReducer>: ReduxStore
    where RootReducer: ReduxReducer,
          RootReducer.State == AppState {

    @Published
    private(set) public var state: AppState

    private let rootReducer: RootReducer
    private var middlewares: [ReduxMiddleware]

    init(
        initialState: State,
        rootReducer: RootReducer,
        middlewares: [ReduxMiddleware]
    ) {
        self.state = initialState
        self.rootReducer = rootReducer
    }

```

```

        self.middlewares = middlewares
    }

    public func dispatch(_ action: ReduxAction) {
        DispatchQueue.main.async {
            self.state = self.rootReducer.reduce(
                state: self.state,
                action: action
            )
        }
        self.middlewares.forEach {
            $0.dispatch(state: self.state,
                action: action,
                effectDispatch: self.dispatch
            )
        }
    }
}

// AppState.swift

struct AppState: ReduxState {
    var isLoading = false
    var selectedIngridient: String?
    var cameraState: CameraState
    var recipesListState: RecipesListState
    var selectedRecipe: Recipe?
}

// CameraState.swift

import SwiftUI
import Vision

struct CameraState: ReduxState {
    var observations: [any VisionObservation] = []
    var previewImage: UIImage? = nil
    var isDetecting: Bool
}

// RecipesListState.swift

struct RecipesListState: ReduxState {
    var recipeList: [Recipe] = []
}

// RootReducer.swift

struct RootReducer: ReduxReducer {
    let cameraReducer: CameraReducer
    let recipesListReducer: RecipesListReducer

```

```

func reduce(state: AppState?,
            action: ReduxAction?) -> AppState {
    var state = state ?? AppState(cameraState: CameraState(isDetecting: false),
                                   recipesListState: RecipesListState())
    state.cameraState = cameraReducer.reduce(state: state.cameraState,
                                              action: action)
    state.recipesListState = recipesListReducer.reduce(state: state.recipesListState,
                                                        action: action)

    switch action {
    case let action as SelectIngridientAction:
        state.selectedIngridient = action.ingridient
    case let action as SelectRecipeAction:
        state.selectedRecipe = action.recipe
    default:
        break
    }
    return state
}
// CameraReducer.swift

```

```

struct CameraReducer: ReduxReducer {
    func reduce(state: CameraState?, action: ReduxAction?) -> CameraState {
        var state = state ?? CameraState(isDetecting: false)
        switch action {
        case is SwitchCameraRecogision:
            state.isDetecting.toggle()
            if !state.isDetecting {
                state.observations = []
            }
        case let action as CameraRecogisionResult:
            state.observations = action.result
        case let action as CameraPreviewResult:
            state.previewImage = action.previewImage
        default:
            break
        }
        return state
    }
}

```

```

// RecipesListReducer.swift

```

```

class RecipesListReducer: ReduxReducer {
    func reduce(state: RecipesListState?, action: ReduxAction?) -> RecipesListState {
        var state = state ?? RecipesListState()
        switch action {
        case let action as SetRicipesListAction:
            state.recipeList = action.recipes
        default:

```

```

        break
    }
    return state
}
}

// CameraMiddleware.swift

import SwiftUICore

class CameraMiddleware: ReduxMiddleware {
    let camera = CameraManager()
    let vision = VisionManager()
    var isDetecting: Bool

    init(isDetecting: Bool) {
        self.isDetecting = isDetecting
    }

    func dispatch(state: ReduxState,
                  action: ReduxAction,
                  effectDispatch: @escaping (ReduxAction) -> Void) {
        guard let state = state as? AppState else { return }
        switch action {
        case is StartCameraRecognition:
            Task {
                await camera.start()
                Task {
                    await handleCameraPreviews(effectDispatch: effectDispatch)
                }
                Task {
                    await handleVisionObservations(effectDispatch: effectDispatch)
                }
            }
        case is StopCameraRecognition:
            camera.stop()
        case is SwitchCameraRecognition:
            isDetecting.toggle()
            effectDispatch(StartCameraRecognition())
        default:
            break
        }
    }

    private func handleCameraPreviews(effectDispatch: @escaping (ReduxAction) -> Void) async {
        for await ciImage in camera.previewStream {
            Task { @MainActor in
                effectDispatch(CameraPreviewResult(previewImage: ciImage))
                if isDetecting {
                    vision.processLiveDetection(ciImage)
                }
            }
        }
    }
}

```

```

    }
  }
}

private func handleVisionObservations(effectDispatch: @escaping (ReduxAction) -> Void) async {
  for await observations in vision.observationStream {
    Task { @MainActor in
      if isDetecting {
        effectDispatch(CameraRecogisionResult(result: observations))
      }
    }
  }
}
}
}

```

// RecipesListMiddleware.swift

```

class RecipesListMiddleware: ReduxMiddleware {
  let firestoreManager = FirebaseDatabaseManager()

  func dispatch(state: ReduxState,
    action: ReduxAction,
    effectDispatch: @escaping (ReduxAction) -> Void) {
    switch action {
    case let action as SelectIngridientAction:
      Task {
        do {
          let recipes = try await firestoreManager.getRecipesFromFirestore(by: action.ingridient)
          effectDispatch(SetRicipesListAction(recipes: recipes))
        } catch {
          print("Error")
        }
      }
    default:
      break
    }
  }
}

```

// CameraSceneActions.swift

```

import Vision
import CoreImage

struct SelectIngridientAction: ReduxAction {
  let ingridient: String
}

struct StartCameraRecognition: ReduxAction { }
struct StopCameraRecognition: ReduxAction { }
struct SwitchCameraRecogision: ReduxAction { }

```

```

struct CameraRecognitionResult: ReduxAction {
    let result: [any VisionObservation]
}

struct CameraPreviewResult: ReduxAction {
    let previewImage: UIImage?
}

// RecipesListActions.swift

struct FetchRecipesListAction: ReduxAction {
    let ingredient: String
}

struct SetRecipesListAction: ReduxAction {
    let recipes: [Recipe]
}

struct SelectRecipeAction: ReduxAction {
    let recipe: Recipe
}

// CameraScene.swift

import SwiftUI
import AVFoundation
import Vision

struct CameraScene: View {
    @EnvironmentObject var router: Router
    @EnvironmentObject var store: AppStore

    @State private var imageSize: CGSize = .zero

    var body: some View {

        ZStack {
            store.state.cameraState.previewImage?.image?
                .resizable()
                .scaledToFit()
                .overlay(content: {
                    GeometryReader { geometry in
                        DispatchQueue.main.async {
                            self.imageSize = geometry.size
                        }
                    }
                    return Color.clear
                })
            .overlay(content: {
                ForEach(0..

```

```

        let observation = store.state.cameraState.observations[index]

        let (text, confidence, boxSize, boxPosition) = processObservation(observation, for:
imageSize)

        RoundedRectangle(cornerRadius: 8)
            .fill(Color.orange.opacity(0.3))
            .stroke(.orange, style: .init(lineWidth: 4.0))
            .onTapGesture {
                store.dispatch(SelectIngridientAction(ingridient: text))
                router.push(.recipientScene)
            }
            .overlay(alignment: .topLeading, content: {
                Text("\(text): \(confidence)")
                    .foregroundColor(.black)
                    .background(.white)
                    .offset(y: -28)
            })
            .frame(width: boxSize.width, height: boxSize.height)
            .position(boxPosition)
    }
})

VStack {
    titleView
    Spacer()
    startDetectingButton
}
}

.onAppear() {
    store.dispatch(StartCameraRecognision())
}

.onDisappear {
    store.dispatch(StopCameraRecognision())
}
}

private var titleView: some View {
    VStack(alignment: .leading) {
        Text("Почніть розпізнавання для вибору інгредієнта")
            .font(.title2)
    }
}

private var startDetectingButton: some View {
    Button(action: {
        store.dispatch(SwitchCameraRecogision())
    }, label: {
        Text("\(store.state.cameraState.isDetecting ? "Закінчити" : "Почати") розпізнавання")
            .foregroundColor(.blue)
    })
}

```

```

        .padding(.all)
        .background(
            RoundedRectangle(cornerRadius: 8)
                .stroke(.black, lineWidth: 2.0))
    })
}

@MainActor
func processObservation(_ observation: any VisionObservation, for imageSize: CGSize) -> (text:
String, confidence: Float, size: CGSize, position: CGPoint) {
    switch observation {
    case let objectObservation as RecognizedObjectObservation:
        return processObjectObservation(objectObservation, for: imageSize)

    default:
        return ("", .zero, .zero, .zero)
    }
}

private func processObjectObservation(_ observation: RecognizedObjectObservation, for imageSize:
CGSize) -> (text: String, confidence: Float, size: CGSize, position: CGPoint) {
    let recognizedText = observation.labels.first?.identifier ?? ""
    let confidence = observation.confidence

    let boundingBox = observation.boundingBox
    let converted = boundingBox.toImageCoordinates(imageSize, origin: .upperLeft)

    let position = CGPoint(x: converted.midX, y: converted.midY)

    let text = getLocalizedString(recognizedText)
    return (text, confidence, converted.size, position)
}

func getLocalizedString(_ key: String) -> String {
    if let path = Bundle.main.path(forResource: "IngridientsLocalization", ofType: "plist"),
        let dict = NSDictionary(contentsOfFile: path) as? [String: String] {
        return dict[key] ?? key
    } else {
        return key
    }
}

// RecipesScene.swift
import SwiftUI
import Firebase

struct RecipesScene: View {
    @EnvironmentObject var router: Router
    @EnvironmentObject var store: AppStore

```

```

var body: some View {
    contentView
        .padding()
}

private var contentView: some View {
    VStack {
        navigationTab
        ScrollView {
            LazyVStack {
                ForEach(store.state.recipesListState.recipeList.indices, id: \.self) { recipeIndex in
                    let recipe = store.state.recipesListState.recipeList[recipeIndex]
                    RecipeView(recipe: recipe)
                        .onTapGesture {
                            store.dispatch(SelectRecipeAction(recipe: recipe))
                            router.push(.detailRecipeScene)
                        }.padding(4)
                }
            }
        }
        Spacer()
    }
}

private var navigationTab: some View {
    ZStack {
        Text("Рецепти з інгредієнтом:\n\((store.state.selectedIngridient ?? "невідомим інгредієнтом")")
            .font(.title2)
            .multilineTextAlignment(.center)
        HStack(alignment: .center) {
            Button {
                router.pop()
            } label: {
                Image(systemName: "chevron.backward.circle.fill")
                    .resizable(resizingMode: .tile)
                    .foregroundColor(.white)
                    .frame(width: 22, height: 22)
            }.growingButtonStyle(padding: 6)
            Spacer()
        }
    }
}

// RecipeView.swift

import SwiftUI

struct RecipeView: View {

```

```

let recipe: Recipe

var body: some View {
    contentView
        .padding(16)
        .overlay {
            Capsule().stroke(Color.orange, lineWidth: 3)
        }
}

private var contentView: some View {
    HStack {
        Text(recipe.name ?? "Something wrong with recipe")
        Spacer()
        Image(systemName: "chevron.right")
    }
}
}

// DetailRecipeScene.swift

import SwiftUI

struct DetailRecipeScene: View {
    @EnvironmentObject var store: AppStore
    @EnvironmentObject var router: Router

    var body: some View {
        VStack {
            backButton
                .padding(.leading, 20)
            contentView
                .padding(12)
            Spacer()
        }
    }

    private var contentView: some View {
        VStack(alignment: .leading, spacing: 12) {
            if let recipe = store.state.selectedRecipe {
                Text(recipe.name)
                    .font(.title2)
                    .bold()
                Text(recipe.description)
                    .font(.body)
                Text("Інгредієнти:").bold()
                ingredientsList(recipe: recipe)
            }
        }
    }
}

```

```

@ViewBuilder
private func ingredientsList(recipe: Recipe) -> some View {
    LazyVStack(spacing: 8) {
        ForEach(recipe.ingredients, id: \.name) { ingredient in
            HStack(spacing: 8) {
                Text(ingredient.name)
                    .font(.subheadline)
                if let count = ingredient.count, count != "null" {
                    Text(count)
                        .font(.subheadline)
                }
                Spacer()
            }
        }
    }
}

private var backButton: some View {
    HStack(alignment: .center) {
        Button {
            router.pop()
        } label: {
            Image(systemName: "chevron.backward.circle.fill")
                .resizable(resizingMode: .tile)
                .foregroundColor(.white)
                .frame(width: 22, height: 22)
        }.growingButtonStyle(padding: 6)
        Spacer()
    }
}

```

```
// CameraManager.swift
```

```
import SwiftUI
import AVFoundation
```

```
class CameraManager: NSObject {

    private let captureSession = AVCaptureSession()

    private var isCaptureSessionConfigured = false
    private var deviceInput: AVCaptureDeviceInput?

    private var videoOutput: AVCaptureVideoDataOutput?
    private var sessionQueue: DispatchQueue!

    var isPreviewPaused = false

    private var addToPreviewStream: ((CIImage) -> Void)?

```

```

lazy var previewStream: AsyncStream<UIImage> = {
    AsyncStream { continuation in
        addToPreviewStream = { ciImage in
            if !self.isPreviewPaused {
                continuation.yield(ciImage)
            }
        }
    }
}()

override init() {
    super.init()
    captureSession.sessionPreset = .low
    sessionQueue = DispatchQueue(label: "session queue")
}

func start() async {
    let authorized = await checkAuthorization()
    guard authorized else {
        print("Camera access was not authorized.")
        return
    }

    if isCaptureSessionConfigured {
        if !captureSession.isRunning {
            sessionQueue.async { [self] in
                self.captureSession.startRunning()
            }
        }
        return
    }

    sessionQueue.async { [self] in
        self.configureCaptureSession { success in
            guard success else { return }
            self.captureSession.startRunning()
        }
    }
}

func stop() {
    guard isCaptureSessionConfigured else { return }

    if captureSession.isRunning {
        sessionQueue.async {
            self.captureSession.stopRunning()
        }
    }
}

```

```

private func configureCaptureSession(completionHandler: (_ success: Bool) -> Void) {

    var success = false

    self.captureSession.beginConfiguration()

    defer {
        self.captureSession.commitConfiguration()
        completionHandler(success)
    }

    guard
        let captureDevice = AVCaptureDevice.DiscoverySession(deviceTypes:
[.builtInWideAngleCamera], mediaType: .video, position: .back).devices.first,
        let deviceInput = try? AVCaptureDeviceInput(device: captureDevice)
    else {
        print("Failed to obtain video input.")
        return
    }

    captureSession.sessionPreset = AVCaptureSession.Preset.vga640x480

    let videoOutput = AVCaptureVideoDataOutput()
    videoOutput.alwaysDiscardsLateVideoFrames = true
    videoOutput.videoSettings = [kCVPixelBufferPixelFormatTypeKey as String:
Int(kCVPixelFormatType_420YpCbCr8BiPlanarFullRange)]
    videoOutput.setSampleBufferDelegate(self, queue: DispatchQueue(label:
"VideoDataOutputQueue"))
    if let videoOutputConnection = videoOutput.connection(with: .video) {
        if videoOutputConnection.isVideoMirroringSupported {
            videoOutputConnection.isVideoMirrored = false
        }
    }

    guard captureSession.canAddOutput(videoOutput) else {
        print("Unable to add video output to capture session.")
        return
    }

    captureSession.addInput(deviceInput)
    captureSession.addOutput(videoOutput)

    self.deviceInput = deviceInput
    self.videoOutput = videoOutput

    isCaptureSessionConfigured = true

    success = true
}

```

```

private func checkAuthorization() async -> Bool {
    switch AVCaptureDevice.authorizationStatus(for: .video) {
    case .authorized:
        print("Camera access authorized.")
        return true
    case .notDetermined:
        print("Camera access not determined.")
        sessionQueue.suspend()
        let status = await AVCaptureDevice.requestAccess(for: .video)
        sessionQueue.resume()
        return status
    case .denied:
        print("Camera access denied.")
        return false
    case .restricted:
        print("Camera library access restricted.")
        return false
    default:
        return false
    }
}

}

extension CameraManager: AVCaptureVideoDataOutputSampleBufferDelegate {

    func captureOutput(_ output: AVCaptureOutput, didOutput sampleBuffer: CMSampleBuffer, from
connection: AVCaptureConnection) {
        guard let pixelBuffer = sampleBuffer.imageBuffer else { return }
        connection.videoRotationAngle = RotationAngle.portrait.rawValue
        addToPreviewStream?(CIImage(cvPixelBuffer: pixelBuffer))
    }

    func captureOutput(_ output: AVCaptureOutput, didDrop sampleBuffer: CMSampleBuffer, from
connection: AVCaptureConnection) {
        // print("frame dropped")
    }
}

private enum RotationAngle: CGFloat {
    case portrait = 90
    case portraitUpsideDown = 270
    case landscapeRight = 180
    case landscapeLeft = 0
}

```

```
extension UIImage {
    var image: Image? {
        let ciContext = CIContext()
        guard let cgImage = ciContext.createCGImage(self, from: self.extent) else { return nil }
        return Image(decorative: cgImage, scale: 1, orientation: .up)
    }
}
```

```
// VisionManager.swift
```

```
import SwiftUI
import Vision
```

```
class VisionManager {

    private var requests: [any VisionRequest] = []
    private var isProcessing: Bool = false

    private var addToObservationStream: (([any VisionObservation]) -> Void)?

    lazy var observationStream: AsyncStream<[any VisionObservation]> = {
        AsyncStream { continuation in
            addToObservationStream = { observations in
                continuation.yield(observations)
            }
        }
    }()

    init() {
        do {
            let request = CoreMLRequest(model: try CoreMLModelContainer(model: try
Find_ingridients_1(configuration: MLModelConfiguration()).model))
            requests = [request]
        } catch {
            fatalError("Failed to load Vision ML model: \(error)")
        }
    }

    func processLiveDetection(_ ciImage: UIImage) {
        if isProcessing { return }
        isProcessing = true

        Task {
            let imageRequestHandler = ImageRequestHandler(ciImage)
            let results = imageRequestHandler.performAll(self.requests)

            await handleVisionResults(results: results)
            isProcessing = false
        }
    }
}
```

```

    }

    @MainActor
    private func handleVisionResults(results: some AsyncSequence<VisionResult, Never>) async {
        var newObservations: [any VisionObservation] = []

        for await result in results {
            switch result {
            case .coreML(_, let observations):
                newObservations.append(contentsOf: observations)
            default:
                return
            }
        }
        addToObservationStream?(newObservations)
    }
}

// FirebaseDatabaseManager.swift

import FirebaseDatabase

class FirebaseDatabaseManager {
    func getRecipesFromFirestore(by ingredient: String) async throws -> [Recipe] {
        let dataBase = Database.database().reference().child("Recipes")
        let dataSnapshot = try? await dataBase.getData()
        if let recipeData = dataSnapshot?.value as? [Any],
           let data = try? JSONSerialization.data(withJSONObject: recipeData, options: []),
           let decodedResult = try? JSONDecoder().decode([Recipe].self, from: data) {
            let result = decodedResult.filter { recipe in
                recipe.ingredients.contains { recipeIngredient in
                    recipeIngredient.name == ingredient
                }
            }
            return result
        } else {
            return []
        }
    }
}

// Recipe.swift

import FirebaseFirestore

struct Recipe: Codable {
    let ingredients: [Ingredient]
    let name: String
    let description: String
}

```

```

struct Ingredient: Codable {
    let name: String
    let count: String?
}

// RouterView.swift
import SwiftUI

struct RouterView<Content: View>: View {

    @StateObject private var router = Router()

    private let content: Content

    init(@ViewBuilder content: @escaping () -> Content) {
        self.content = content()
    }

    var body: some View {
        NavigationStack(path: $router.path) {
            content.navigationDestination(for: Router.Route.self) {
                router.view(for: $0).navigationBarBackButtonHidden()
            }
        }
        .environmentObject(router)
    }
}

// Router.swift
import SwiftUI

@MainActor
final class Router: ObservableObject {

    @Published var path = NavigationPath()

    @ViewBuilder
    func view(for route: Route) -> some View {
        switch route {
        case .cameraScene:
            CameraScene()
        case .recipientScene:
            RecipesScene()
        case .detailRecipeScene:
            DetailRecipeScene()
        }
    }

    func push(_ appRoute: Route) {

```

```

        path.append(appRoute)
    }

    func pop() {
        guard !path.isEmpty else { return }
        path.removeLast()
    }

    func popToRoot() {
        path.removeLast(path.count)
    }
}

extension Router {
    enum Route {
        case cameraScene
        case recipientScene
        case detailRecipeScene
    }
}

// RootView
import SwiftUI

struct RootView: View {
    @EnvironmentObject var store: AppStore

    var body: some View {
        RouterView {
            CameraScene()
        }
    }
}

```

ДОДАТОК Б
Слайди презентації

Національний університет Запорізький політехніка

Кафедра програмних засобів

Дослідження та реалізація технології розпізнавання продуктів з використанням моделей машинного навчання для формування кулінарних рекомендацій

Виконав студент гр. КНТ-124м:
Керівник к.т.н., доцент:

Михайло ПАНЧЕНКО
Ольга ГЛАДКОВА

Запоріжжя 2025

Рисунок Б.1 – Слайд 1

МЕТА РОБОТИ

Мета кваліфікаційної роботи - спростити вибір кулінарного рецепту за допомогою розпізнавання інгредієнтів вдома, допомогти знайти невідомий продукт на полиці в магазині і запропонувати рецепти з нього.

Завдання, для виконання поставленої мети:

- аналіз мов програмування для створення мобільних застосунків;
- аналіз технологій для створення, навчання і використання моделі машинного навчання;
- аналіз застосунків аналогів;
- аналіз архітектур для розробки мобільних застосунків на платформі iOS з використанням мови програмування Swift;
- розробка застосунку на архітектурі Redux з використання мови Swift;
- навчання моделі машинного навчання в CreateML та додавання готової моделі до проєкту;
- створення бази даних рецептів з використання Firebase та обробка її використання в проєкті;
- тестування застосунку в різних умовах.

2

Рисунок Б.2 – Слайд 2

Вибір мови програмування та платформи

Критерій	Swift	Objective-C	Flutter	React native
Розробник	Apple	Apple	Google	Meta
Продуктивність	Вища	Висока	Висока	Середня
Зручність розробки	Сучасний синтаксис, легка розробка	Складний синтаксис	Легка розробка	Середня важкість розробки
Підтримка ML (CoreML / Vision)	Пряма підтримка, нативна інтеграція	Повна, але писати складніше	Існує лише через плагіни	ML через JS дуже обмежене
Системні можливості	Повний доступ до iOS API	Повний доступ до iOS API	Частковий, через платформенні канали	Частковий, залежність від нативних модулів
Простота інтеграції з нативними модулями	Висока	Висока	Середня (через platform channels)	Середня/низька (через native modules)
Кросплатформеність	немає	немає	є	є
Критерій	Swift	Objective-C	Flutter	React native
Розробник	Apple	Apple	Google	Meta

3

Рисунок Б.3 – Слайд 3

Вибір бібліотеки для UI

Критерій	SwiftUI	UIKit
Тип фреймворку	Декларативний	Імперативний
Рік появи	2019 (iOS 13)	2008 (iPhoneOS 2)
Підхід до побудови UI	Описує що має бути на екрані	Описує як це має бути реалізовано
Продуктивність	Дуже висока, оптимізована під сучасні iOS	Висока, але потребує більше ручної оптимізації
Інтерактивність та прев'ю	Live Preview у Xcode	Немає ML
Підтримка Apple	Основний фреймворк розвитку UI	Підтримується, але як legacy
Найкраще підходить для	Нових проєктів, швидкої розробки, ML/AR, анімацій	Складних кастомних UI, великих legacy систем
Можливість змішування	Можна вбудовувати UIKit-компоненти через UIViewRepresentable	Можна вбудовувати SwiftUI через UIHostingController
Мінімальна версія iOS	iOS 13+	Підтримує старі версії (iOS 8+)

4

Рисунок Б.4 – Слайд 4

Вибір архітектури IOS застосунку

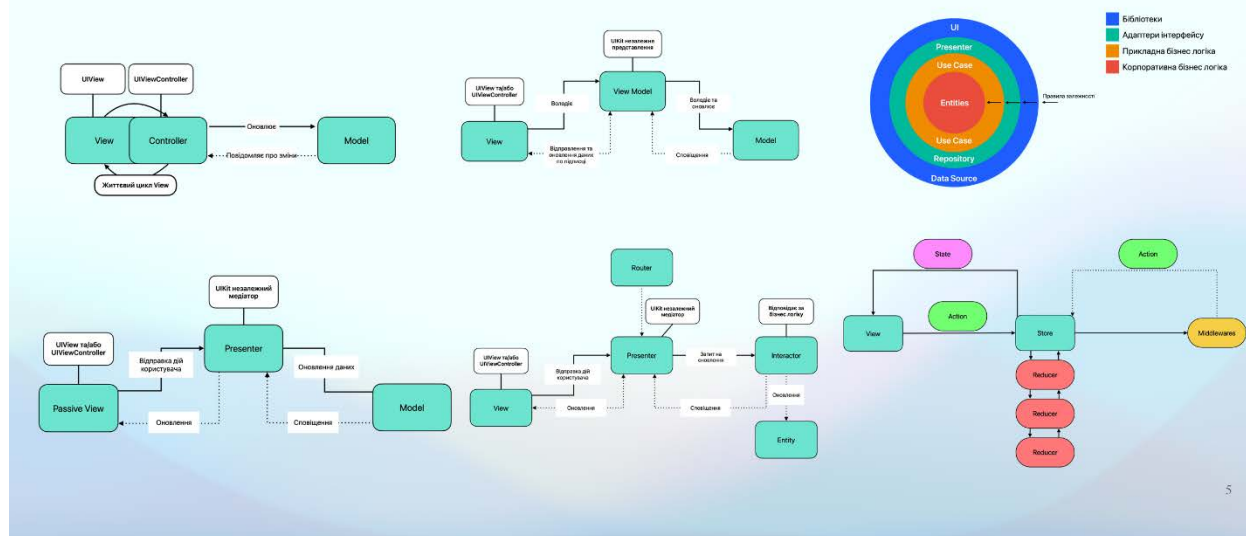


Рисунок Б.5 – Слайд 5

Аналіз застосунків аналогів

Критерії	Menu Planner	Cookpad	Tasty	SnapDish	Foodvisor
UI/UX	Яскравий дизайн, але агресивна монетизація та плутана підписка	Добрий інтерфейс, соціальна взаємодія з іншими користувачами	Гарний UI, відеоінструкції полегшують роботу з рецептами	Посередній UI, мінімалістичний стиль	Непоганий UI, але с довгий вступ і реклама підписки
Швидкість роботи	Інтерфейс швидкий, але обробка фото повільна (серверна)	Загалом швидка робота	Швидкий UI, відео працюють стабільно	Швидкий та стабільний застосунок	Дуже швидка та стабільна робота
Обробка зображень	Фото надсилаються на сервер, немає режиму "в живу"	Немає ML, рецепти вводяться вручну користувачами	Немає ML, фокус на відеоконтенті	Використання CoreML + Vision для розпізнавання страв	Глибокі нейронні моделі для аналізу їжі, частково через сервер, частково локально
Тип розпізнавання	Розпізнавання інгредієнтів (повільне)	Немає	Немає	Розпізнає цілі страви, але не окремі продукти	Розпізнає готові страви та рахує калорії, але не фокусується на кулінарії
Точність ML	Доволі непогана (серверні моделі)	Немає ML	Немає ML	Непогана точність на рівні страв, але не інгредієнтів	Висока точність, одна з найкращих моделей на ринку
База рецептів	Велика, є персоналізація через питання користувачеві	Велика завдяки активній спільноті користувачів	Середня, акцент на відео	Невелика, фокус на фото страв і соц. функціях	Невелика, орієнтація на фітнес-харчування

Рисунок Б.6 – Слайд 6

Діаграми послідовності дій

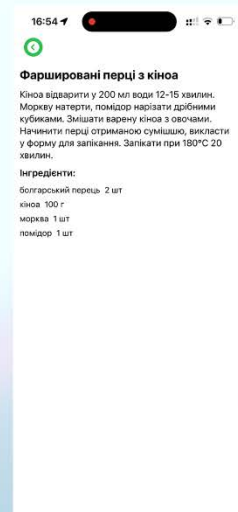
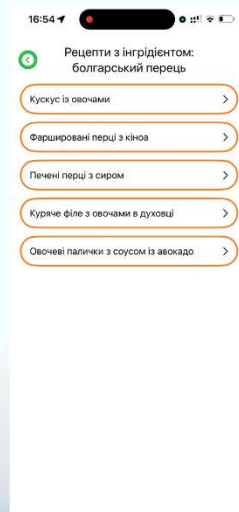
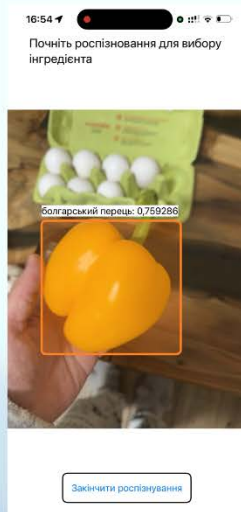


9

Рисунок Б.9 – Слайд 9

Тестування

В домашніх умовах



10

Рисунок Б.10 – Слайд 10

Висновки

Розроблено функціональний прототип iOS-застосунку, який поєднує комп'ютерний зір, машинне навчання та сучасні архітектурні підходи для спрощення процесу пошуку кулінарних рецептів.

Практичне значення отриманих результатів полягає у можливості використання розробленого рішення як основи для створення комерційних продуктів у сфері кулінарних рекомендацій, персоналізованого харчування або фуд-аналітики.

11

Рисунок Б.11 – Слайд 11