МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ

ФАКУЛЬТЕТ МЕХАТРОНІКИ ТА КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

Machine Learning та способи його використання при розробці програмного забезпечення в екосистемі Apple

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
Освітня програма Комп'ютерні науки

Виконав: студент групи МгІТ-2-22

Нікіта ПОНОМАРЕНКО
Науковий керівник: к.т.н., доц. Тетяна ДЕМКІВСЬКА
Рецензент д.т.н., проф Віктор ЧУПРИНКА

Київ 2023

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ

Факультет мехатроніки та комп'ютерних технологій

Кафедра комп'ютерних наук

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки

Освітня програма Комп'ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри КН

_____ Володимир ЩЕРБАНЬ

«__» _____ 2023___ року

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТА

Пономаренку Нікіті Володимировичу

1. **Тема кваліфікаційної роботи** Machine Learning та способи його використання в екосистемі Apple ,

Науковий керівник роботи Демківська Тетяна Іванівна, к.т.н., доц.

затверджені наказом закладу вищої освіти від 12 . 09.2023 року , № 210-уч.

2. **Вихідні дані до кваліфікаційної роботи:** Розробки кафедри комп'ютерних наук; рекомендована література, додатки.

3. **Зміст кваліфікаційної роботи:** Вступ; РОЗДІЛ 1. Machine Learning в екосистемі Apple; РОЗДІЛ 2. Інуструменти для розробки програмного забезпечення Apple; РОЗДІЛ 3. Розробка мультиплатформенного додатку для взаємодії з OpenAI API із можливістю розпізнавання мовлення, зчитування тексту з картинки; Висновки; Список використаних джерел; ДОДАТОК А Програмний код; ДОДАТОК Б Публікація, ДОДАТОК В Презентація.

4. **Дата видачі завдання** 1 вересня 2023

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Термини виконання етапів	Примітка про виконання
1	Вступ	15.09.2023	
2	Розділ 1 Machine Learning в екосистемі Apple	20.09.2023	
3	Розділ 2. Інструменти для розробки програмного забезпечення Apple	30.09.2023	
4	Розділ 3. Розробка мультиплатформенного додатку для взаємодії з OpenAI API із можливістю розпізнавання мовлення, зчитування тексту з картинки	10.10.2023	
5	Висновки	25.10.2023	
6	Оформлення кваліфікаційної роботи (чистовий варіант)	1.11.2023	
7	Подача кваліфікаційної роботи (проєкту) науковому керівнику для відгуку (за 14 днів дозахисту)	4.11.2023	
8	Подача кваліфікаційної роботи (проєкту) для рецензування	6.11.2023	
9	Перевірка кваліфікаційної роботи на наявність ознак плагіату	8.11.2023	2%-9%
10	Подання кваліфікаційної роботи на затвердження завідувачу кафедри	10.11.2023	

З завданням ознайомлений:

Студент Нікіта ПОНОМАРЕНКО

Керівник Тетяна ДЕМКІВСЬКА

АНОТАЦІЯ

Пономаренко Н.В. Machine Learning та способи його використання при розробці програмного забезпечення в екосистемі Apple

Кваліфікаційна робота за спеціальністю 122 - «Комп’ютерні науки та технології» – Київський національний університет технологій та дизайну, Київ, 2023 рік.

Проведено ознайомлення з інструментами, необхідними для розробки програмного забезпечення для операційних систем iOS та macOS, включаючи наступні фреймворки SwiftUI, UIKit та AppKit. Досліджено різні парадигми програмування, характерні для сучасного розробницького середовища Apple, включаючи об'єктно-орієнтоване програмування, протоколо-орієнтоване програмування та елементи функціонального програмування.

Вивчено ключові особливості розробки на мові Swift, включаючи аспекти керування пам'яттю за допомогою Automatic Reference Counting (ARC), використання асинхронності за допомогою нововведених конструкцій async/await, а також механізми паралельної роботи, зокрема Grand Central Dispatch (GCD).

Як практичний результат, було розроблено мультиплатформенний додаток, що надає можливість користувачеві спілкуватись з чат-ботом зі штучним інтелектом, інтегручись з OpenAI API. Додаток також використовує Vision Framework для розпізнавання тексту з картинок і Speech Framework для перетворення мовлення в текст, та для мережевих запитів використовує URLSession.

Ключові слова: фреймворк, екосистема Apple, розробка програмного забезпечення, середовище розробки, машинне навчання.

ANNOTATION

Ponomarenko N.V. Machine Learning and its applications in software development within the apple ecosystem

Graduate Master's degree in specialty 122 - "Computer Science and Technologies" - Kyiv National University of Technology and Design, Kyiv, 2023.

An exploration of tools necessary for software development for the iOS and macOS operating systems has been carried out, including the following frameworks SwiftUI, UIKit and AppKit. Various programming paradigms characteristic of the modern Apple development environment were studied, encompassing object-oriented programming, protocol-oriented programming and elements of functional programming.

Key features of development in the Swift language were examined, including memory management aspects using Automatic Reference Counting (ARC), the utilization of asynchronicity through the newly introduced async/await constructs, and parallel processing mechanisms, notably through Grand Central Dispatch (GCD).

As a practical result, a cross-platform application was developed that integrates with the OpenAI API for communication with ChatGPT and DALL·E. The app also employs the Vision Framework for image-based text recognition and the Speech Framework for converting speech to text, and for network requests it utilizes URLSession.

Keywords: Framework, Apple ecosystem, Software development, Development Environment, Machine Learning systems.

ЗМІСТ

ВСТУП

Актуальність проблеми. В сучасному цифровому віці технології неперервно еволюціонують, призводячи до нових можливостей та викликів у світі програмування та дизайну. Однією з провідних компаній у цьому контексті є компанія Apple, яка пишається своєю здатністю привносити інновації та створювати продукти, які об'єднують людей. Її екосистема з продуктів — iPhone, iPad, Mac, Apple Watch — надає розробникам унікальний набір інструментів і ресурсів для створення додатків.

Ця екосистема, зокрема, стає особливо цікавою у контексті розробки програмного забезпечення. За роки свого існування, Apple надала розробникам серію потужних мов програмування, таких як Swift та Objective-C і набори інструментів, як SwiftUI, UIKit та AppKit, що спрощують розробку додатків.

На додачу до цього, компанія постійно вводить нові технологічні рішення, дозволяючи розробникам створювати все більш авангардні додатки. Поступово ця динаміка призвела до того, що межа між апаратним та програмним забезпеченням стає менш вираженою, а додатки починають використовувати все більше можливостей пристроїв, на яких вони працюють.

Мета дослідження. Метою даного дослідження є вивчення застосування Machine Learning технологій, а також самих технологій розробки програмного забезпечення в екосистемі Apple, із розробкою практичної реалізації додатку на базі SwiftUI, що інтегрується із сервісами OpenAI для взаємодії з ChatGPT та DALLE, Vision framework для розпізнавання тексту та Speech framework для перетворення голосу в текст.

Об’єктом дослідження цієї роботи є методи та інструменти розробки програмного забезпечення під різні платформи Apple, а також наявні Machine Learning технології, які дозволяють покращити досвід користувачеві.

Предметом дослідження є програмний продукт, що дозволяє користувачеві спілкуватись з чат-ботом зі штучним інтелектом, написаний з використанням SwiftUI, Vision та Speech фреймворків, та інтеграцією з OpenAI API.

Елементи наукової новизни. Розробка адаптивного інтерфейсу, оптимізованого для взаємодії з ML моделями, зокрема моделлю ChatGPT, а також реалізацією функції розпізнавання тексту та голосових команд для покращення користувацького досвіду.

Практична значущість роботи. Розроблений додаток може слугувати як інструмент для широкого спектру користувачів екосистеми Apple, дозволяючи їм ефективніше взаємодіяти з ML моделями. Сторінка сетингів забезпечує гнучкість налаштувань за запитами користувачів.

Апробація результатів роботи. Результати роботи були апробовані шляхом тестування додатку групою користувачів, які підтвердили його функціональність та зручність інтерфейсу. Додаток продемонстрував стабільність при розпізнаванні тексту та голосу, а також точність інтеграції з ChatGPT для здійснення інтерактивного діалогу.

РОЗДІЛ 1. MACHINE LEARNING В ЕКОСИСТЕМІ APPLE

1.1. Введення в концепції машинного навчання

Машинне навчання в останні роки стає все більш актуальним і невід'ємним елементом сучасних технологій. Це підгалузь штучного інтелекту, яка зосереджена на розробці алгоритмів, що дають комп'ютерам можливість вивчати дані без явного їх програмування. Замість того, щоб прямо навчати комп'ютери, як виконувати певну задачу, машинне навчання дозволяє системам самостійно навчатись на прикладах.

Основна ідея машинного навчання полягає в тому, щоб зрозуміти структуру даних та встановити зв'язки між різними елементами. Це може бути використано для розпізнавання образів, прогнозування подій, автоматизації процесів та інших завдань.

Однією з ключових характеристик машинного навчання є його гнучкість. В залежності від сценарію застосування, можливо вибрати підхід, який найкраще підходить для конкретної задачі. Так, існують різні типи машинного навчання (рис 1.1):

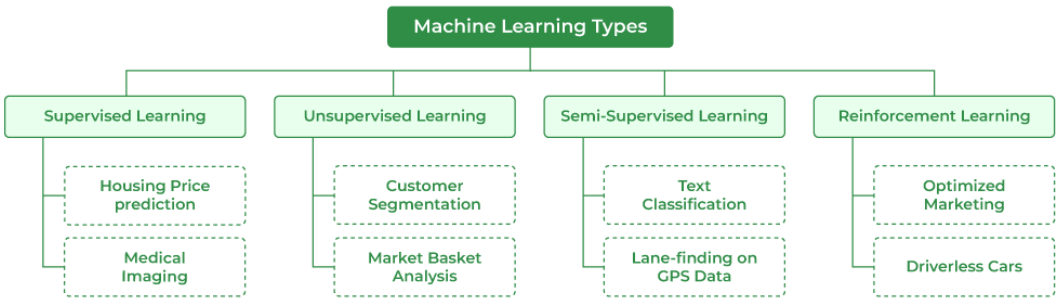


Рис 1.1 Типи машинного навчання

- кероване навчання: коли дані мають явно вказані мітки, та система намагається вивчити відношення між вхідними даними та виходом;
- некероване навчання: дані не мають міток, а система намагається знайти структуру або закономірності в даних;
- напівкероване навчання: частина даних має мітки, а інша частина – ні;

– навчання з підкріпленням**:** система намагається максимізувати певний показник, взаємодіючи з навколишнім середовищем.

Важливим аспектом є вибір правильного алгоритму для конкретної задачі. Вибір базується на природі даних, завданні, яке стоїть перед дослідником, та доступних ресурсах. Наприклад, для завдань класифікації часто використовуються логістична регресія, метод опорних векторів або нейронні мережі. Тоді як для завдань регресії можна використовувати лінійну регресію або методи на основі дерев.

Науковці та інженери постійно працюють над покращенням існуючих алгоритмів та створенням нових, що відкриває нові горизонти для досліджень у цій галузі.

З моменту появи машинного навчання та його широкого застосування в промисловості, багато компаній, включаючи Apple, активно інвестують у дослідження та розробку нових технологій. Це стосується не тільки алгоритмів та моделей, але й платформ для розробки, які роблять процес створення та реалізації моделей машинного навчання простішим і ефективнішим.

Та не дивлячись на всі ці переваги, важливо пам'ятати, що машинне навчання – це лише інструмент. Як і будь-який інструмент, він повинен використовуватися відповідно, з урахуванням етичних стандартів та обмежень.

В машинному навчанні особливу увагу слід приділяти даним. Дані – це основа, на якій побудована модель. Вони відіграють важливу роль у точності та ефективності алгоритмів. Важливо розуміти джерело даних, їх структуру та якість, адже це впливає на здатність моделі правильно прогнозувати або класифікувати інформацію.

Інший ключовий аспект машинного навчання – це валідація та тестування. Після того, як модель навчена, важливо переконатися, що вона працює коректно на нових, раніше невідомих даних. Це допомагає

визначити, наскільки модель є надійною та здатною до загального використання.

Машинне навчання має широкий спектр застосувань в різних галузях: від медицини до фінансів, від маркетингу до автоматизації виробництва. Проте, якщо говорити про екосистему Apple, вона включає в себе ряд інструментів та фреймворків, призначених специфічно для розробки та впровадження машинного навчання на їхніх пристроях. До таких інструментів відносяться Core ML, Create ML, Vision, Speech, Natural Language та Sound Analysis.

Та незалежно від конкретної платформи або застосування, основні принципи машинного навчання залишаються однаковими. Це комбінація математичних методів, статистики, алгоритмів та даних. Всі ці елементи разом дають можливість створювати потужні системи, які можуть вивчати з досвіду та адаптуватися до нових інформаційних потоків.

Завершуючи введення, слід підкреслити, що машинне навчання є одним із найбільш перспективних напрямків в сфері технологій. Його потенціал далеко не вичерпаний, і ми лише починаємо розуміти всі можливості, які воно може принести людству в майбутньому.

1.2. Core ML: основний інструмент Apple для інтеграції ML моделей у додатки

У світі машинного навчання існує безліч інструментів і платформ, але коли справа доходить до інтеграції моделей ML у додатки для iOS та macOS, Core ML стає ключовим інструментом у екосистемі Apple.

Core ML дозволяє розробникам вбудовувати натреновані моделі машинного навчання безпосередньо у додатки, оптимізуючи їх для використання на пристроях Apple. Це забезпечує високу продуктивність та ефективність, оскільки моделі оптимізуються для конкретних чіпів та архітектур, які використовуються в пристроях Apple.

Також Core ML тісно інтегрований із середою розробки Xcode, яка є необхідною для розробки програмного забезпечення під техніку Apple. Що легко дозволяє досліджувати поведінку та продуктивність моделі, перш ніж почати писати код. Легко інтегрувати моделі у програму за допомогою автоматично створених інтерфейсів у Swift і Objective-C, а також профілювати основні Core ML функції свого додатка за допомогою інструментів Core ML і Neural Engine (рис 1.2).

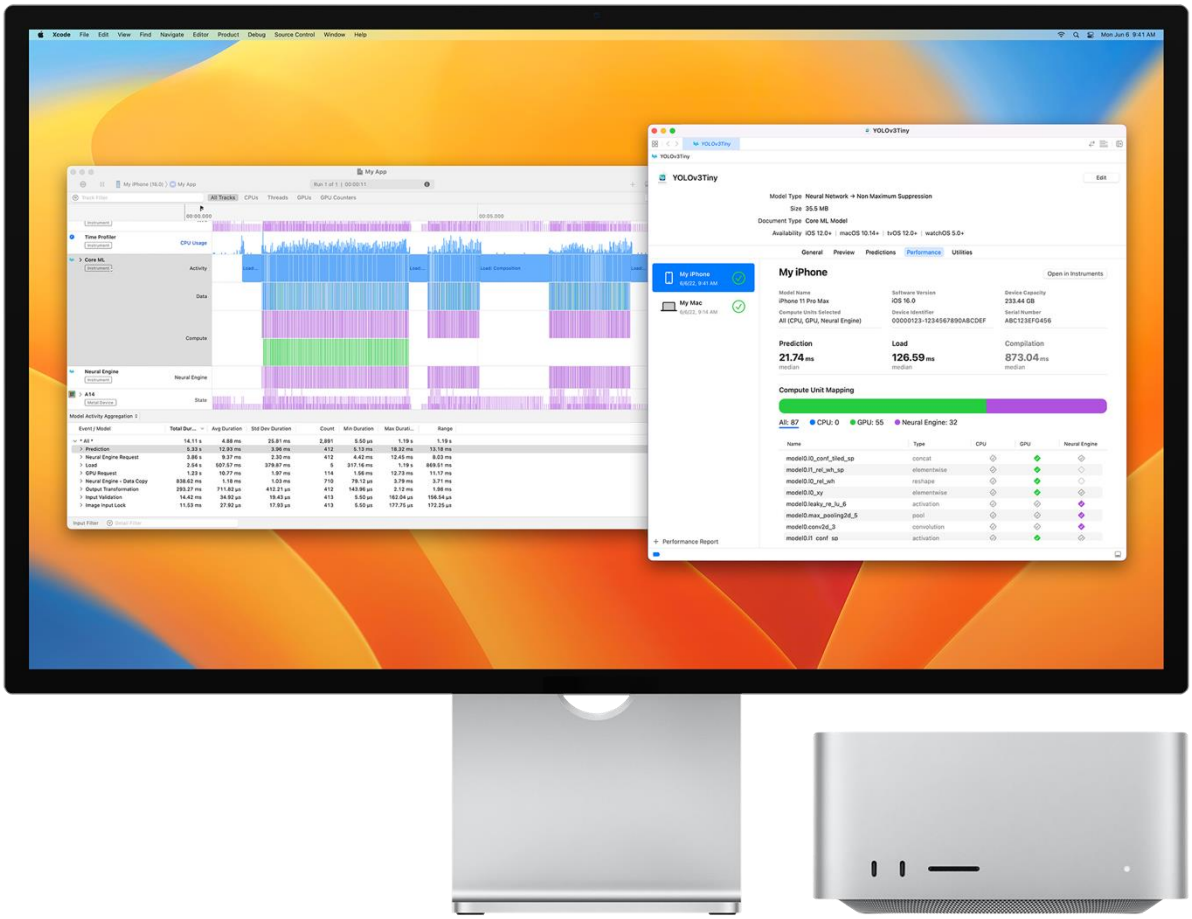


Рис 1.2 Інтеграція із Xcode

Одна з ключових переваг Core ML полягає в його гнучкості щодо імпорту моделей. Не всі розробники створюють свої моделі з нуля в екосистемі Apple. Багато команд уже мають готові моделі, розроблені та натреновані в інших середовищах, таких як TensorFlow, Keras, Scikit-learn

або PyTorch. Для цих команд існує питання: як використовувати ці моделі на пристроях Apple?

Саме тут на допомогу приходить інструмент конвертації Core ML Tools. Цей інструмент дозволяє розробникам конвертувати моделі з популярних бібліотек машинного навчання в формат .mlmodel, який можна безпосередньо інтегрувати в додатки на iOS, macOS, watchOS та tvOS.

Процес конвертації зазвичай є прямолінійним. Вам потрібно вказати вхідний файл моделі, визначити тип моделі та визначити, які додаткові параметри повинні бути використані під час конвертації. Якщо в моделі використовуються специфічні операції, які не підтримуються Core ML, інструмент надасть відповідне повідомлення про помилку, дозволяючи розробникам вирішити цю проблему.

Варто відзначити, що після конвертації модель буде оптимізована для виконання на пристроях Apple, забезпечуючи найкращу продуктивність та ефективність енергоспоживання. Така оптимізація забезпечує плавне виконання навіть на старіших пристроях.

Завдяки можливості конвертації, Core ML стає ще більш універсальним інструментом для розробників. Вони можуть користуватися перевагами потужних бібліотек машинного навчання на інших платформах, а потім легко переносити свої рішення в екосистему Apple, забезпечуючи користувачам найкращий досвід взаємодії.

Таким чином, інструмент конвертації Core ML Tools відкриває двері до глибокої інтеграції моделей машинного навчання в додатки для пристроїв Apple, роблячи цей процес максимально гладким і простим для розробників.

Також слід зазначити, що Core ML підтримує обчислення на CPU, GPU та, у випадку новіших пристроїв Apple, на Neural Engine. Це забезпечує максимальну продуктивність та ефективність виконання для різних типів завдань машинного навчання.

Завдяки такому широкому спектру можливостей та глибокої інтеграції із іншими фреймворками та технологіями Apple, Core ML стає вибором номер один для розробників, які хочуть інтегрувати машинне навчання в свої додатки на платформах Apple.

Фреймворк є основою для доменно-специфічних фреймворків і функцій. Він є фундаментом Vision для аналізу зображень, Natural Language для обробки тексту, Speech для перетворення аудіо в текст і Sound Analysis для визначення звуків у аудіо. А сам Core ML будується на основі низькорівневих примітивів, таких як Accelerate і BNNS, а також Metal Performance Shaders (рис 1.3).

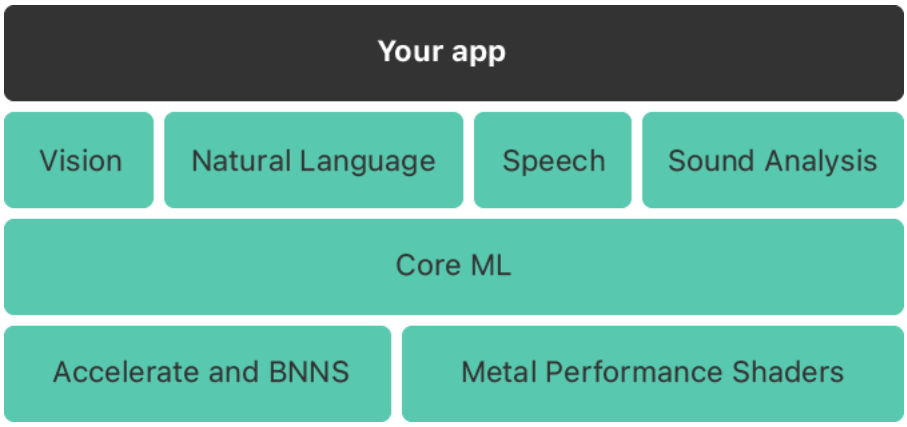


Рис 1.3. Ієрархічна структура ML фреймворків

Так, робота з Core ML не обмежується лише інтеграцією готових моделей. Цей фреймворк також надає можливості для удосконалення та адаптації моделей прямо на пристроях користувачів, завдяки чому можна досягти більшої точності та спеціальної адаптації до даних конкретного користувача.

Однією з важливих особливостей Core ML є його можливість працювати в "онлайн" та "офлайн" режимах. Це означає, що ваші додатки можуть використовувати машинне навчання без підключення до Інтернету, що забезпечує кращий досвід користувача, зокрема в умовах поганого інтернет-з'єднання або його відсутності.

Що стосується CreateML, це ще один цінний інструмент в екосистемі Apple, що дозволяє розробникам легко створювати власні моделі машинного навчання.

Після того, як модель натренована, ви можете легко конвертувати її в формат Core ML та інтегрувати в свій додаток. Це робить процес створення та впровадження моделей машинного навчання в застосунках iOS та macOS надзвичайно простим і доступним, навіть для розробників без глибоких знань у галузі машинного навчання.

Враховуючи постійний розвиток технологій та зростаючу популярність машинного навчання, інструменти, такі як Core ML та CreateML, стають необхідними для сучасних розробників додатків. Вони допомагають створювати потужні, адаптивні та інтуїтивно зрозумілі додатки, які можуть використовувати переваги машинного навчання, щоб забезпечити найкращий досвід користувача.

1.3. Create ML: інтуїтивний інструмент для створення ML моделей

Apple завжди славилася своїм підходом до розробки програмного забезпечення: вона намагається робити його якнайбільш доступним і зрозумілим для користувачів. Create ML – це яскравий приклад такої філософії в області машинного навчання. Це потужний інструмент, який було спроектовано з метою допомоги розробникам швидко і ефективно створювати моделі машинного навчання без глибоких знань в цій галузі.

Відразу варто звернути увагу на інтерфейс Create ML. Він максимально інтуїтивний і заснований на принципах драг-енд-дроп. Це означає, що розробник може просто перетягнути свої дані в інструмент, вибрати потрібний тип моделі та розпочати тренування. Наприклад, для створення класифікатора зображень, вам потрібно лише додати ваш набір даних із зображеннями до Create ML, визначити категорії та почати тренування моделі (рис 1.4).

Рис 1.4 Створення моделі класифікатора зображень

Це забезпечує низький поріг входу для розробників, які хочуть додати можливості машинного навчання до своїх застосунків на платформах Apple. Такий підхід відкриває двері до машинного навчання для широкого спектра фахівців, які можуть не мати формальної освіти в галузі data science.

Щодо швидкості, Create ML вражає. Використовуючи передові техніки оптимізації, інструмент дозволяє тренувати моделі на порядок швидше, ніж багато інших схожих інструментів. Це особливо корисно, коли розробнику потрібно провести низку експериментів з різними параметрами або наборами даних.

Однією з ключових особливостей Create ML є його тісна інтеграція з мовою програмування Swift. Для розробників це означає можливість використовувати знайомий для них інструментарій, такий як Swift Playground, щоб легко та ефективно тренувати та тестувати свої моделі прямо в середовищі Xcode.

Create ML також відрізняється підтримкою різноманітних типів даних. Він може працювати з зображеннями, текстом, аудіо і табличними даними. Це робить його ідеальним інструментом для багатьох різних задач машинного навчання, від класифікації зображень до обробки природньої мови.

Після тренування моделі, інструмент надає розробникам можливості для її детального аналізу. Зокрема, можна використовувати матрицю помилок, оцінити точність, вивести характеристики чутливості та специфічності.

Підсумовуючи, можна сказати, що Create ML – це неймовірно потужний інструмент, який зробив машинне навчання доступним для широкого кола розробників. Він демонструє, як можна зробити складну технологію простою та доступною для всіх.

1.4. Високорівневі фреймворки Apple для обробки даних: Vision, Speech, Sound Analysis та Natural Language

Apple завжди прагнула спростити інтеграцію технологій машинного навчання в свою екосистему, надаючи розробникам високорівневі інструменти. Ці інструменти дозволяють спеціалістам з різних галузей швидко та ефективно використовувати передові технології без глибокого занурення в складності машинного навчання.

Vision

Вступаючи в світ комп'ютерного зору, фреймворк Vision від Apple стає ключовим інструментом. Це потужний фреймворк для обробки зображень, який надає розробникам обширний ряд високорівневих можливостей.

Vision дозволяє не лише ефективно обробляти зображення, але й легко інтегрується з Core ML, надаючи можливість використовувати треновані моделі безпосередньо на пристроях Apple. Це забезпечує високу продуктивність та оптимізовану роботу для конкретного обладнання.

Основні можливості фреймворку Vision включають:

- Детекція облич та їх характеристик (наприклад, визначення позиції очей, рота тощо);
- Розпізнавання та відслідковування об'єктів на зображеннях;
- Визначення тексту на зображеннях;

– Аналіз сцен для визначення контекстної інформації.

Завдяки Vision розробники можуть легко створювати додатки, які взаємодіють і реагують на візуальний контент, без необхідності глибокого розуміння процесів обробки зображень.

Один з найпоширеніших завдань у галузі комп'ютерного зору – це виявлення облич. Фреймворк Vision надає інструменти для швидкого та ефективного розпізнавання облич на зображеннях.

Розглянемо короткий приклад на мові програмування Swift, який демонструє, як легко можна використовувати Vision для цієї задачі (рис 1.5)

```swift
import Vision

// Завантаження зображення
let image = UIImage(named: "sampleImage")!

// Створення запиту на визначення облич
let faceDetectionRequest = VNDetectFaceRectanglesRequest { request, error in
    guard let observations = request.results as? [VNFaceObservation] else { return }
    for face in observations {
        print("Обличчя знайдено на зображенні у прямокутнику: \(face.boundingBox)")
    }
}

// Виконання запиту
let handler = VNImageRequestHandler(cgImage: image.cgImage!, options: [:])
try? handler.perform([faceDetectionRequest])
```

Рис 1.5 Приклад застосування фреймворку Vision

У цьому прикладі ми імпортуємо фреймворк Vision, завантажуємо зображення з назвою "sampleImage". Після цього ми створюємо запит на виявлення облич VNDetectFaceRectanglesRequest, де обробляємо результати та виводимо інформацію про розташування облич на зображенні. На завершення, ми використовуємо VNImageRequestHandler для обробки завантаженого зображення та виконання запиту на виявлення облич.

Отримані результати можуть бути використані в ряді застосунків, таких як:

- фоторедактори: для автоматичного виправлення обличчя, налаштування експозиції або додавання ефектів;
- безпека і розпізнавання: виявлення осіб на камерах спостереження або для систем розпізнавання осіб;
- соціальні мережі: для автоматичного тегування осіб на фотографіях.

Крім базового виявлення облич, Vision також пропонує інструменти для визначення окремих особливостей обличчя, таких як очі, ніс і рот, а також для визначення нахилу, орієнтації і емоційної реакції особи. Ці функції можуть бути корисними для створення більш розширених застосунків, які вимагають глибокого розуміння контенту зображення.

Speech

Фреймворк Speech від Apple — це потужний інструмент для розпізнавання голосу та його конвертації в текстовий формат. Він використовує передові алгоритми машинного навчання для забезпечення високої точності розпізнавання навіть у складних аудіоумовах, пропонуючи широкий набір функцій.

Адаптивне розпізнавання. Speech може адаптуватися до особливостей мовлення конкретного користувача, враховуючи акценти, діалекти та інші нюанси.

Підтримка багатьох мов. Один із ключових аспектів цього фреймворку — багатомовність. Apple постійно розширює список підтримуваних мов, що робить його універсальним рішенням для глобальних додатків.

Робота в реальному часі. Інструмент дозволяє розпізнавати голос в реальному часі, що дозволяє створювати інтерактивні додатки, здатні миттєво відповідати на команди користувача.

Безпека та конфіденційність. Apple акцентує увагу на тому, що обробка даних відбувається локально на пристрої користувача, не вимагаючи передачі аудіоданих на зовнішні сервери.

Використання фреймворку Speech відкриває безліч можливостей для розробників. Це може бути все — від простих голосових помічників до складних систем голосового управління для додатків та ігор. Наприклад, додаток для навчання мов може використовувати цей інструмент для перевірки правильності вимови або для перекладу голосових записів на інші мови.

Sound Analysis

Фреймворк Sound Analysis від Apple відкриває широкі можливості для роботи з аудіоданими. Він надає розробникам інструменти для розпізнавання та класифікації звуків, що дозволяє створювати застосунки з більш високим рівнем інтеракції та адаптації до звукового оточення користувача, надаючи широкий набір функцій:

- розпізнавання звукових подій: завдяки передовим алгоритмам машинного навчання Sound Analysis може розпізнавати різноманітні звукові події, від шуму вентилятора до звуку дверного дзвінка. Це може бути використано, наприклад, в додатках безпеки;
- класифікація звуків: фреймворк дозволяє не тільки розпізнавати, але й класифікувати звуки за різними параметрами, що може бути корисним для музичних додатків або додатків для редагування аудіо;
- адаптація до зміни звукових умов: Sound Analysis може динамічно адаптуватися до змінних умов при яких звук був записаний, оптимізуючи роботу додатка в реальному часі;
- інтеграція з іншими технологіями Apple: фреймворк Sound Analysis ідеально поєднується з іншими технологіями Apple, такими як Core ML, що дозволяє розробникам створювати комплексні рішення на базі машинного навчання.

Застосування Sound Analysis відкриває безліч можливостей для розробки інноваційних додатків. Наприклад, додаток для медитації може адаптуватися під рівень шуму в кімнаті, підбираючи оптимальний звуковий

фон для користувача. Або система "розумного будинку" може визначати дії на основі розпізнавання звуків, таких як сигнал будильника або звук розбитого скла.

При роботі з фреймворком розробник в першу чергу повинен визначити специфіку завдання. Якщо мова йде про розпізнавання шуму вентилятора, процес може виглядати так:

- збір даних: необхідно зібрати аудіозаписи з різними варіантами звуку вентилятора — різні моделі, швидкості обертання тощо. Також потрібно мати датасет звуків, які не є шумом вентилятора, для тренування моделі розрізняти шум вентилятора від інших звуків;
- підготовка даних: це може включати в себе нормалізацію гучності, вирізання коротких фрагментів аудіо для тренування та валідації моделі та інше;
- створення моделі: за допомогою Create ML або іншого інструменту, який підтримує Sound Analysis, можна створити модель для розпізнавання звуків. Тренування моделі на зібраних даних дасть змогу їй вивчити особливості шуму вентилятора;
- тестування: після тренування моделі її потрібно протестувати на нових даних, щоб переконатися в її точності та надійності;
- інтеграція моделі: отримана модель може бути інтегрована в додаток за допомогою Core ML. Фреймворк Sound Analysis дасть змогу розробнику легко взаємодіяти з моделлю, реагуючи на розпізнані моделлю звуки в реальному часі;
- оптимізація: в залежності від результатів роботи додатка в реальних умовах, може виявитися необхідність додаткової оптимізації моделі або корекції даних.

Цей процес надає розробнику структурований підхід до розпізнавання звуків, враховуючи всі етапи розробки від підготовки даних до інтеграції моделі в додаток.

Фреймворк також надає вбудовані моделі для розпізнавання деяких загальних типів звукових подій. Це може бути, наприклад, розпізнавання різниці між мовленням та співом, визначення ритміки в музиці, або навіть розпізнавання певних типів звуків в побуті.

Якщо розробник хоче скористатися цими вбудованими можливостями, йому не потрібно проводити додаткове навчання моделі. Просто використовуючи API, ви можете отримувати результати розпізнавання звуків на льоту.

Отже, поки фреймворк може потребувати додаткового навчання для дуже специфічних задач, таких як розпізнавання шуму вентилятора, існують також кейси, де його можна використовувати одразу без додаткової підготовки.

Natural Language

Фреймворк Natural Language від Apple це високорівнева утиліта, що надає можливості для аналізу, обробки та інтерпретації природної мови. Він інтегрований в екосистему Apple і дозволяє розробникам ефективно виконувати ряд завдань з обробки мови без потреби в глибокому зануренні в деталі машинного навчання.

Можливості Natural Language:

- токенізація: фреймворк дозволяє розбивати текст на окремі "токени" - це може бути слова, речення, параграфи або інші логічні одиниці;
- визначення частин мови: Natural Language може аналізувати слова в контексті та визначати їх частини мови, такі як іменники, дієслова, прикметники тощо;
- виявлення іменованих сутностей: фреймворк вміє виявляти іменовані сутності, такі як імена людей, організацій, місць, дат та ін;

- лематизація: здатність перетворювати слово на його базову форму. Наприклад, з "бігаючи" до "бігати";
- виявлення мови: фреймворк може автоматично визначати мову, на якій написаний текст.

Однією з ключових особливостей Natural Language є його інтеграція з іншими високорівневими фреймворками Apple. Наприклад, є можливість поєднувати його з фреймворком Speech для створення потужних голосових асистентів, які можуть розуміти і відповідати на запитання користувача в природній мові.

Також важливо зазначити, що Natural Language має вбудовані моделі для декількох мов, що робить його зручним інструментом для глобальних застосунків. Втім, якщо потрібна підтримка дуже специфічних мов або діалектів, розробники можуть навчити свої моделі та інтегрувати їх з фреймворком.

У контексті реального світу, Natural Language може бути використаний для створення чат-ботів, систем аналізу відгуків користувачів, автоматичного перекладу, та інших застосунків, де ключовим є розуміння та обробка природної мови.

1.5. Приватність в контексті машинного навчання

Приватність даних у сучасному світі інформації здобула важливе значення. З появою машинного навчання, яке базується на величезних об'ємах даних, питання захисту особистої інформації стало особливо актуальним. Компанія Apple відома своїм підходом до захисту приватності своїх користувачів. Але як саме Apple захищає дані своїх користувачів в контексті машинного навчання?

Диференційована приватність – це підхід, який дозволяє отримувати загальні дані про групу користувачів без виявлення інформації про конкретних індивідів. Це здійснюється за допомогою додавання "шуму" до даних перед їхньою обробкою. Якщо Apple хоче дізнатися,

скільки користувачів використовують певний додаток, вони можуть зібрати цю інформацію, але так, що буде неможливо визначити, чи використовує конкретний користувач цей додаток.

Обробка даних на пристрої. Локальні обчислення: однією з ключових особливостей підходу Apple до машинного навчання є те, що велика частина обробки даних відбувається безпосередньо на пристрої користувача, а не на віддалених серверах. Це означає, що особисті дані користувача, як правило, залишаються на його пристрої. Цей підхід забезпечує додатковий рівень захисту, оскільки дані не передаються через мережу і не зберігаються на віддалених серверах, де вони можуть стати мішенню для хакерів.

Анонімізація даних. Apple намагається збирати якнайменше даних для надання своїх послуг. Це включає в себе використання технік, що дозволяють компанії отримувати необхідну інформацію без збору конкретних особистих даних. Якщо Apple хоче вдосконалювати Siri, вони можуть збирати анонімізовані голосові дані, такі, що не ідентифікують користувача.

Прозорість і контроль. Apple забезпечує користувачів докладною інформацією про те, які дані збираються і з якою метою. Це дає змогу користувачам розуміти, як їх дані використовуються і захищаються. Користувачі мають можливість контролювати збір та використання своїх даних. Це може бути відключення трекінгу локації, обмеження рекламного відстеження чи видалення історії Siri.

Стратегія Apple з приватності є поєднанням технічних рішень, користувацького контролю і прозорості. Компанія постійно шукає нові методи для забезпечення захисту особистої інформації своїх користувачів, при цьому не жертвуючи зручністю та функціональністю своїх продуктів.

Таким чином, приватність у контексті машинного навчання в Apple виходить на передову і стає ключовою частиною корпоративної культури.

1.6. Порівняльний аналіз із іншими екосистемами

Apple завжди відзначалася своєю унікальністю, створюючи екосистему, що інтегрує апаратне і програмне забезпечення. Це створило умови для розвитку власних інструментів машинного навчання. Однак розробники часто стикаються з вибором: використовувати інструменти Apple чи звертатися до інших. Давайте порівняємо основні переваги та недоліки Apple у контексті машинного навчання і порівняємо їх з іншими ключовими гравцями на ринку.

Apple. Перевагами Apple є висока інтеграція інструментів, акцент на приватності, висока продуктивність на пристроях Apple, наявність високорівневих фреймворків для швидкої інтеграції.

В свою чергу викликами такого підходу є обмежений екосистемою Apple, відсутність гнучкості для деяких специфічних завдань, вища вартість пристроїв.

Google (TensorFlow і Google Cloud ML). TensorFlow - це відкритий фреймворк машинного навчання від Google, який підтримує широкий спектр завдань: від базових операцій до нейронних мереж глибокого навчання. Google Cloud ML надає обчислювальні можливості та сервіси для навчання, розгортання та управління моделями TensorFlow. Його перевагами є легка інтеграція з іншими продуктами Google Cloud, такими як GCS, BigQuery тощо. Автоматичне масштабування для обробки великих об'ємів даних і навчання моделей. TensorFlow Board надає можливості візуалізації процесу навчання. А також сильна підтримка спільноти, багато навчальних матеріалів та документації.

Однак TensorFlow може бути складним для новачків у порівнянні з деякими іншими фреймворками. Хоча сам TensorFlow є безкоштовним, використання Google Cloud ML може вимагати суттєвих витрат, особливо на велику кількість обчислень. Для TensorFlow існують різні варіації, такі як TensorFlow Lite для мобільних пристроїв та TensorFlow.js для веб-

розробки. Це робить його універсальним рішенням для розробників, які працюють у різних областях.

Microsoft (Azure Machine Learning і ONNX). Azure Machine Learning - це хмарний сервіс для машинного навчання від Microsoft. Він дозволяє розробникам створювати, навчати та розгортати моделі в хмарі. ONNX (Open Neural Network Exchange) - це відкритий стандарт для обміну моделями машинного навчання між різними фреймворками. Перевагами якого є глибока інтеграція з іншими продуктами Microsoft та підтримка різних мов програмування та фреймворків. Також Azure Machine Learning Studio пропонує візуальний інтерфейс для створення моделей, що знижує поріг входу для новачків. Підтримка ONNX дозволяє розробникам легко переносити моделі між різними платформами та фреймворками.

Однак використання Azure Machine Learning може бути дорогим, особливо для великих об'ємів даних та обчислень. А також як і інші хмарні платформи, Azure Machine Learning може змусити розробників залежати від конкретної екосистеми.

Azure також пропонує рішення для Edge Computing, де моделі можуть бути навчені в хмарному середовищі і потім розгорнуті на пристроях для локального виконання.

Amazon (Amazon SageMaker і AWS ML сервіси). Amazon SageMaker - це багатофункціональна платформа, яка дозволяє розробникам легко будувати, навчати та розгортати моделі машинного навчання на Amazon Web Services (AWS). AWS також пропонує низку спеціалізованих ML сервісів для конкретних завдань, таких як розпізнавання мови, зображень та аналіз тексту.

SageMaker підтримує широкий вибір ML фреймворків, таких як TensorFlow, PyTorch, MXNet та інші. Масштабованість: Автоматичне масштабування, використовуючи потужність AWS, для обробки великих наборів даних. Оптимізація: Спрощений процес тюнінгу гіперпараметрів та

оптимізації моделей. Консолідація: Інтеграція з іншими AWS сервісами, такими як Lambda, S3, DynamoDB тощо.

Проте як і з іншими хмарними платформами, використання SageMaker може бути дорогим залежно від вимог до обчислень і зберігання даних. Складність: Деякі аспекти SageMaker можуть бути викликом для новачків, особливо якщо вони нові в екосистемі AWS.

Amazon зосереджений на створенні екосистеми, де розробники можуть легко інтегрувати машинне навчання в свої застосунки, використовуючи комбінацію SageMaker та інших AWS сервісів.

PyTorch — це відкрита бібліотека для машинного навчання, розроблена Facebook. Відома своєю гнучкістю та ефективністю, особливо під час дослідницької роботи.

Перевагами якого є динамічний обчислювальний граф: що забезпечує більш інтуїтивний спосіб будування та зміни моделей на льоту. Швидко зростаюча спільнота, яка надає підтримку та ресурси для дослідників та розробників. Легка інтеграція з науковими бібліотеками Python, такими як NumPy та SciPy.

Однак можливі проблеми із масштабуванням виробництва. Хоча PyTorch прославився серед дослідників, його перенесення до виробничого оточення може бути дещо викликом. А також незважаючи на велику спільноту, деякі аспекти PyTorch можуть бути недостатньо задокументовані.

Висновки до розділу 1

Досліджено набір інструментів для машинного навчання, які пропонує Apple, було зроблено фокус на їх схваленні до приватності даних користувачів, інтеграції та оптимізації.

Виконано аналіз інструментів інших великих платформ, таких як AWS, Google Cloud, Facebook та Microsoft Azure, що пропонують

інтегровані рішення, які масштабуються та оптимізуються для великих даних і розподіленого обчислення.

Також було детально проаналізовано переваги та виклики різних платформ. Apple відзначається своїм акцентом на приватності, інтеграції та оптимізації. Однак, коли мова йде про гнучкість, розширеність та широкий спектр застосувань, інші платформи та фреймворки можуть бути більш привабливими для розробників. Одним з ключових моментів є те, що вибір фреймворку або платформи часто визначається конкретними потребами проекту, доступними ресурсами та знаннями розробників. Незалежно від вибору, важливо розуміти можливості та обмеження кожного інструменту для досягнення найкращих результатів.

РОЗДІЛ 2. ІНСТРУМЕНТИ ДЛЯ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ APPLE

2.1. Xcode: Центральне середовище для розробки Apple

Xcode — це інтегроване середовище розробки (IDE) від Apple, яке призначене для створення додатків для macOS, iOS, watchOS та tvOS. Як ключовий інструмент розробки в екосистемі Apple, Xcode надає розробникам широкий спектр інструментів, які покривають весь цикл розробки продукту.

Основні можливості Xcode:

- редактор коду: Xcode оснащений потужним редактором коду, який підтримує підсвітку синтаксису, автодоповнення та інші зручні функції, які полегшують написання коду;
- візуальний редактор інтерфейсу (Interface Builder): дозволяє користувачам графічно проектувати інтерфейс додатку, використовуючи елементи керування та віджети, доступні в різних фреймворках Apple;
- підтримка Swift та Objective-C: хоча Swift є сучасною мовою програмування від Apple, Xcode також підтримує розробку на Objective-C, що дає можливість робити взаємодію між обома мовами в одному проекті;
- Simulator: інтегрований емулятор пристроїв, який дозволяє розробникам тестувати свої додатки на різних моделях пристроїв та версіях ОС без необхідності мати фізичний девайс;
- підтримка системи контролю версій: Xcode інтегрований з рядом систем контролю версій, включаючи Git, що полегшує співпрацю та ведення журналу змін;
- автоматизоване тестування: Xcode має вбудовану підтримку XCTest для створення юніт-тестів, що допомагає забезпечити якість коду;

– інтеграція App Store: Xcode надає інструменти для безпосередньої публікації додатків в App Store, що полегшує процес розповсюдження продукту.

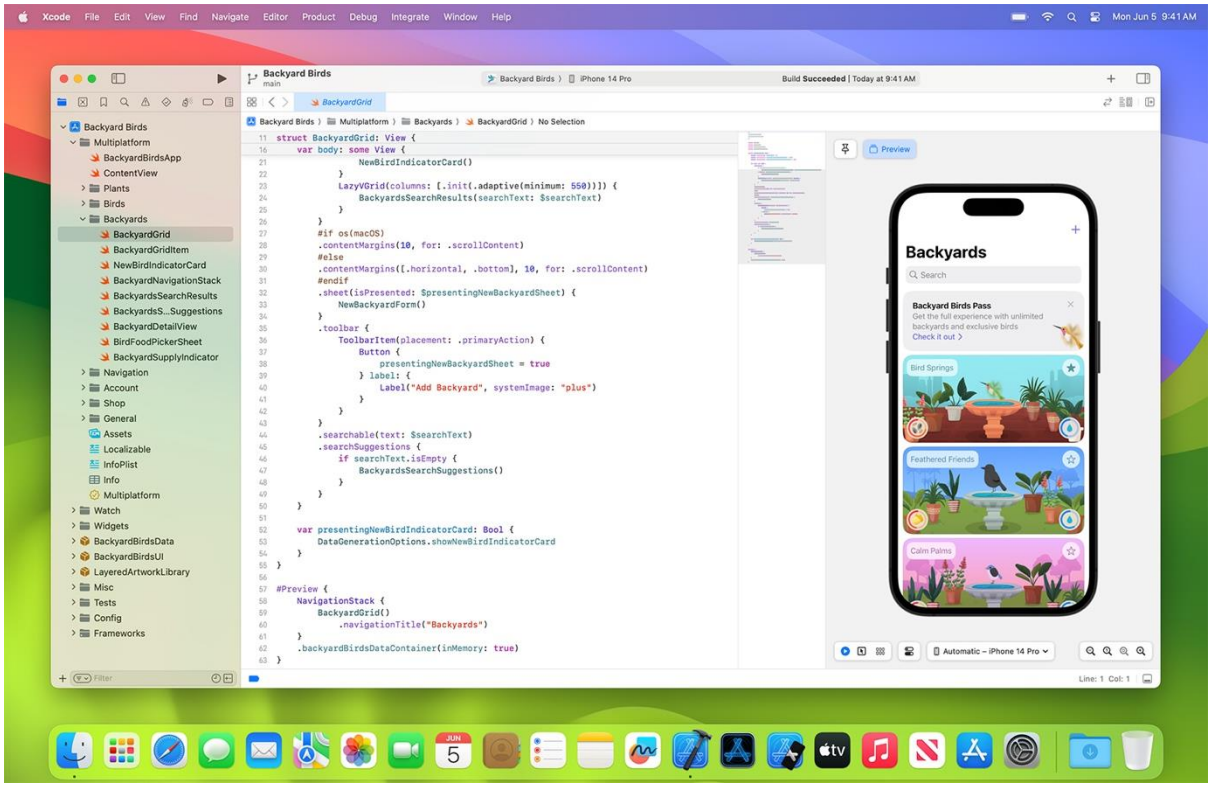


Рис 2.1 Інтерфейс Xcode

Xcode є серцем розробки додатків для екосистеми Apple. Він поєднує в собі ряд потужних інструментів, які допомагають розробникам ефективно створювати якісні додатки.

Instruments є набором інструментів профілювання, які входять у склад Xcode. Вони допомагають розробникам аналізувати, відлагоджувати та оптимізувати свої додатки під час процесу роботи на реальних пристроях або в Simulator. Профілювання — це процес збору даних про виконання програми з метою виявлення проблемних місць, втрат ресурсів та інших проблем.

Основні можливості Instruments:

– Time Profiler: цей інструмент допомагає визначити, де ваш додаток витрачає найбільше часу під час процесу роботи. Він відображає дерево

викликів та показує, які функції та методи викликаються найчастіше. Це дозволяє розробникам знайти і оптимізувати проблемні місця у коді;

- Memory Allocations: цей інструмент аналізує використання пам'яті вашим додатком. Він може допомогти виявити витоки пам'яті, зайві виділення пам'яті та інші проблеми, які можуть негативно впливати на продуктивність та стабільність додатку;
- Energy Diagnostics: цей інструмент зосереджений на визначенні того, як ваш додаток впливає на життєвий цикл батареї пристрою. Особливо цінний для мобільних додатків, які повинні бути ефективними з точки зору енергоспоживання;
- File Activity: цей інструмент моніторить файлові операції, що виконуються вашим додатком, допомагаючи визначити можливі проблеми із читанням/записом файлів;
- Network: інструмент, який допомагає аналізувати мережеву активність додатку. Це може бути корисним для виявлення проблем із з'єднанням, оптимізації використання даних та зменшення часу відгуку.

Завдяки цим інструментам розробники можуть глибоко аналізувати свої додатки, виявляти і виправляти проблеми на ранніх стадіях розробки. Instruments допомагає підтримувати високий стандарт якості програмного забезпечення Apple, забезпечуючи користувачам додатки відмінної якості.

Interface Builder — це графічний редактор в Xcode, який розробникам дозволяє створювати візуальний інтерфейс для своїх додатків з мінімальним написанням коду. За допомогою перетягування елементів керування з палітри на робочий лист, користувачі можуть легко і швидко розробляти інтерфейси.

Основні особливості Interface Builder:

- візуальне розташування компонентів: Розробники можуть використовувати мишку для перетягування різних UI елементів на екран;

– авто-розміщення: За допомогою системи обмежень можна визначити, як елементи інтерфейсу повинні розміщуватися та масштабуватися на різних розмірах екрану;
– Прив'язка даних: Легко прив'язати елементи інтерфейсу до конкретних даних або дій у вашому коді;
– Живий попередній перегляд: Можливість подивитись, як інтерфейс виглядає і веде себе в реальному часі;
– Інтеграція з кодом: Після того як візуальний інтерфейс створено, його можна легко інтегрувати з кодом за допомогою "outlets" та "actions".

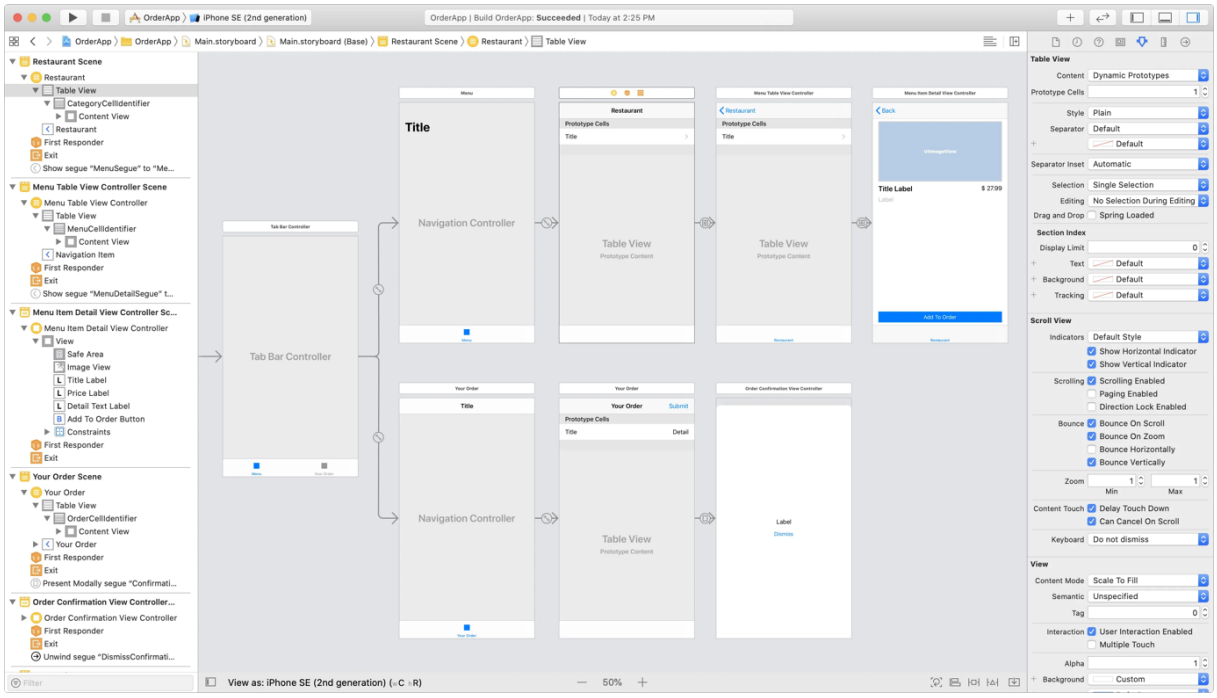


Рис 2.2 Приклад Storyboard файлу

Однак, важливо зазначити, що з ростом популярності декларативних UI фреймворків, таких як SwiftUI, Interface Builder і його підхід до імперативного дизайну інтерфейсу стає менш релевантним. SwiftUI пропонує більш сучасний і гнучкий підхід до створення інтерфейсів, дозволяючи розробникам створювати комплексні інтерфейси за допомогою декларативного коду. Це не тільки пришвидшує процес розробки, але й робить його більш зрозумілим та уніфікованим.

Таким чином, хоча Interface Builder ще досить потужний і корисний інструмент, розробники повинні розглядати сучасні підходи і технології, які Apple пропонує, особливо якщо вони хочуть залишатися на гребені хвилі технологічних нововведень.

Simulator у складі Xcode — це інструмент, який дозволяє розробникам тестувати додатки прямо на їхніх Mac комп'ютерах. Він відтворює інтерфейс користувача та поведінку реального пристрою, але працює як звичайний додаток на macOS.

Основні особливості Simulator:

- великий вибір пристроїв: Simulator має вбудовані конфігурації для всіх доступних моделей iPhone, iPad, Apple Watch, Apple TV та Mac. Це дозволяє розробникам легко перевірити, як їх додаток виглядає і функціонує на різних пристроях;
- тестування різних версій ОС: Можна інсталювати та запускати різні версії операційних систем, що дозволяє перевірити сумісність додатку з різними версіями ОС;
- імітація різних станів: Simulator може імітувати різні аспекти роботи реального пристрою, такі як обертання, переходи в режим «Не турбувати», з'єднання з мережею та інше;
- доступ до системних програм: Розробники можуть перевірити інтеграцію своєї програми з іншими системними додатками, такими як Контакти, Календар та Сповіщення;
- тестування функцій інтерфейсу: За допомогою Simulator можна перевірити, як ваш додаток реагує на різні події, такі як 3D Touch, жести масштабування або взаємодія з Touch Bar на MacBook Pro;
- імітація розташування: Simulator дозволяє імітувати різні GPS координати, що корисно для додатків, які використовують геолокацію. Розробники можуть вказувати конкретні координати, вибирати з

передвстановлених місць, або навіть симулювати рух за певним маршрутом;

- взаємодія з іншими додатками: Розробники можуть тестувати взаємодію своєї програми з іншими додатками в Simulator. Це особливо корисно для перевірки deep linking або інтеграції через системні служби, такі як Handoff;
- розширення та віджети: Simulator також підтримує тестування розширень додатків та виджетів, що дозволяє розробникам перевірити їх функціональність у різних контекстах і налаштуваннях.

На відміну від реальних пристроїв, Simulator має деякі обмеження. Продуктивність додатку в Simulator може відрізнятися від його продуктивності на реальному пристрої. Це означає, що якщо ваш додаток працює швидко в Simulator, це не завжди означає, що він буде так само швидко працювати на реальному пристрої.

Додатково, не всі апаратні можливості, доступні на реальних пристроях, можна симулювати в Simulator. Це включає такі можливості як Face ID, Touch ID, NFC та інші. Simulator також не може відтворити споживання батареї вашого додатку на реальному пристрої. Враховуючи ці обмеження, хоча Simulator є чудовим інструментом для початкового тестування, кінцеве тестування на реальних пристроях завжди залишається критично важливим кроком у процесі розробки додатків.

2.2. Новітній Swift та застарілий Objective-C

Історія Objective-C

Objective-C бере свої коріння від двох основних мов програмування: мови C і об'єктно-орієнтованої мови програмування Smalltalk.

Розроблена у 1970-х роках в Xerox PARC, Smalltalk була однією з перших об'єктно-орієнтованих мов програмування. Вона надавала унікальний підхід до програмування через використання об'єктів, які могли

приймати повідомлення та взаємодіяти один з одним. Цей підхід до програмування став фундаментом для Objective-C.

Мова C була стандартом промисловості для системного програмування завдяки своїй потужності, гнучкості та переносимості. Вона була широко використовувана для розробки операційних систем та вбудованого програмного забезпечення.

Brad Cox і Tom Love, у середині 1980-х роках, об'єднали особливості Smalltalk та C, щоб створити Objective-C. Їхня мета була простою: забезпечити потужність мови C з об'єктно-орієнтованими характеристиками Smalltalk. Вони хотіли створити мову, яка дозволила б програмістам легко розробляти масштабовані додатки, використовуючи об'єктно-орієнтований підхід.

Завдяки їхнім зусиллям, Objective-C об'єднала простоту та елегантність Smalltalk з низькорівневою потужністю C, дозволяючи розробникам створювати високоефективні програми з розширеними можливостями абстракції об'єктів.

Походження Objective-C відображає прагнення інженерів дати розробникам найкращі інструменти для створення програмного забезпечення. Об'єднуючи об'єктно-орієнтовані характеристики з потужністю мови системного програмування, Objective-C стала фундаментом для багатьох програмних продуктів, що змінили світ, особливо в екосистемі Apple.

Роль у ранніх версіях macOS

Після свого створення в 1980-х роках, Objective-C швидко знайшла своє застосування в сфері розробки програмного забезпечення, особливо в компанії NeXT, заснованій Стівом Джобсом після його відходу з Apple.

Ключовою частиною продуктової лінійки NeXT була операційна система NeXTSTEP. Ця ОС використовувала Objective-C як свою основну мову програмування, і вона стала підґрунтям для того, що ми тепер знаємо

як macOS (раніше відома як Mac OS X). NeXTSTEP була попередницею в багатьох технологічних інноваціях, зокрема в графічному користувацькому інтерфейсі, об'єктно-орієнтованому програмуванні та високорівневих абстракціях для розробки додатків.

Objective-C відіграла ключову роль у цьому успіху. Використовуючи динамічні можливості мови, розробники могли створювати потужні, гнучкі додатки, які легко адаптувалися до змінюваних вимог.

Коли Apple придбала NeXT в 1997 році, технології NeXT, включаючи Objective-C та NeXTSTEP, були інтегровані в основу нової версії операційної системи Apple - Mac OS X.

Objective-C стала основною мовою програмування для розробки додатків на Mac OS X. Ця мова дозволила Apple перевести багато ідей та парадигм з NeXTSTEP до нової ОС, включаючи такі компоненти як Cocoa (раніше відомий як OpenStep) - фреймворк для розробки додатків на Mac. В ранніх версіях macOS, Objective-C не лише дозволила розробникам створювати інноваційні додатки, але й служила мостом між піонерськими днями NeXT та сучасною ерою Apple. Її динамічний характер, об'єктно-орієнтовані можливості та відкриті стандарти зробили її ідеальним вибором для революційної операційної системи, яка лягла в основу сучасної екосистеми Apple.

Внесок у розвиток iOS

Objective-C стала фундаментальним стовпом для Apple не тільки в контексті macOS, але й у створенні та розвитку iOS, операційної системи для iPhone, iPad і iPod Touch.

Коли Apple представила свій перший iPhone у 2007 році, цей пристрій був еволюційним кроком у мобільному програмуванні. iOS, раніше відома як iPhone OS, була похідною від macOS, і, як результат, багато з основних технологій, що були частиною Mac OS X, також знаходили своє місце в iOS.

Objective-C, зокрема, стала основною мовою програмування для розробки додатків на iOS. Це забезпечило розробникам платформи інструменти, з якими вони вже були знайомі з розробки для Mac, дозволяючи їм створювати потужні, гнучкі додатки для нового мобільного середовища.

Компоненти, такі як Cocoa Touch, були введені для підтримки розробки мобільних додатків, які враховують специфіку сенсорного інтерфейсу та мобільного досвіду. Ці фреймворки, написані на Objective-C, надали розробникам набір засобів для створення інтуїтивних додатків, які стали відмінною особливістю екосистеми Apple.

Протягом років, як iOS розвивалася, Objective-C залишалася в центрі розробки, дозволяючи Apple інтегрувати передові технології, такі як ARKit для доповненої реальності та Core ML для машинного навчання, безпосередньо в середовище розробки.

Однак, незважаючи на великий внесок Objective-C в успіх iOS, потреба в сучасній, більш виразній та безпечній мові програмування призвела до створення Swift. Незважаючи на це, важливо визнати величезний внесок, який Objective-C зробила в основу того, як ми розуміємо сучасну мобільну розробку та як вона формувала підходи і практики, які продовжують використовуватися і на сьогоднішній день.

З урахуванням цього, можна сказати, що без Objective-C світ мобільних додатків, як ми його знаємо сьогодні, виглядав би зовсім по іншому.

Swift: За лаштунками революції

Objective-C служила Apple вірою та правдою протягом десятиліть, але як з будь-якою технологією, що має свої корені в глибокому минулому, із часом виникали певні обмеження та виклики. Хоча Objective-C постійно оновлювалася і адаптувалася до сучасних потреб розробників, було ясно, що потрібна мова, яка була б більш виразною, безпечною та швидкою.

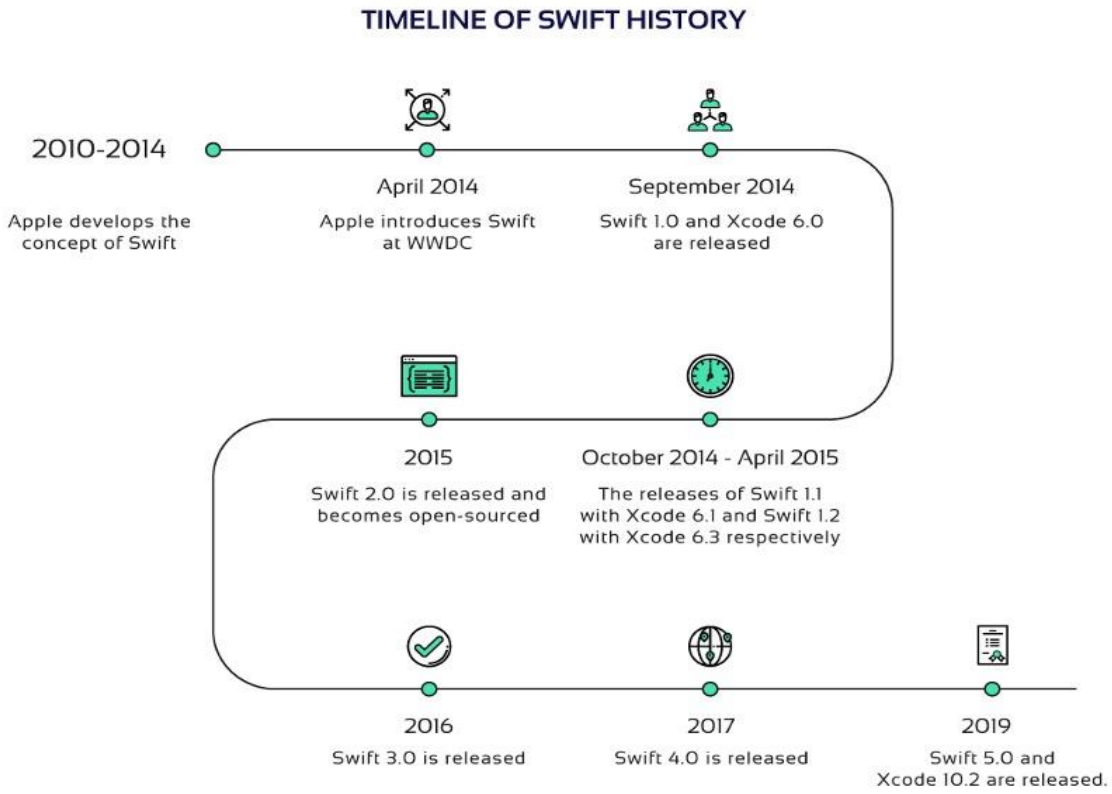


Рис 2.3 Історія мови Swift

Однією з ключових цілей було створити мову, яка б комбінувала в собі виразність сучасних мов програмування з простотою і доступністю. Розробники шукали способи зробити код більш читабельним, зменшуючи при цьому кількість помилок.

Найбільшою із проблем, які розробники намагалися вирішити, була безпека. Objective-C, як і багато старіших мов, могла бути схильною до певних типів помилок, таких як помилки вказівників. Swift було розроблено з метою мінімізувати ці джерела проблем, використовуючи сильну типізацію та інші сучасні парадигми.

Хоча Objective-C була дуже ефективною мовою, існують області, де було можливо досягти значних покращень. Swift було спроектовано так, щоб код виконувався швидше та ефективніше, зокрема завдяки оптимізаціям на рівні компілятора.

Apple розуміла, що нова мова могла б забезпечити більші можливості для розробників, незалежно від їхнього досвіду. Сучасний синтаксис, який використовує Swift, робить його легким для вивчення новачками, а також привабливим для досвідчених розробників, які шукають засоби для створення більш потужних та інноваційних додатків.

У цілому, розробка Swift була логічним кроком для Apple. Враховуючи швидкий розвиток технологій та зростаючі вимоги до програмного забезпечення, компанія почувала потребу в сучасній, гнучкій та безпечній мові програмування, яка могла б забезпечити майбутнє їхньої екосистеми. Swift був відповіддю на ці виклики.

Основні цінності:

- сучасний синтаксис: Swift використовує більш чистий та виразний синтаксис, який спроектовано так, щоб зробити код більш зрозумілим і читабельним. Це не тільки полегшує розуміння коду, але й сприяє швидкому вивченню мови новими розробниками;
- швидкість виконання: завдяки оптимізаціям на рівні компілятора Swift часто перевершує Objective-C в тестах продуктивності, особливо в складних обчисленнях;
- інтерактивна розробка: за допомогою Playgrounds в Xcode розробники можуть експериментувати з кодом Swift в реальному часі, без необхідності компіляції повного проекту. Це надає можливість швидко тестувати нові ідеї та алгоритми;
- сумісність з Objective-C: хоча Swift і має ряд переваг, важливим аспектом є його спроможність співпрацювати з кодом на Objective-C в одному проекті. Це забезпечує гладкий перехід для команд, які хочуть інтегрувати Swift у свої існуючі проекти;
- активна спільнота та підтримка: із моменту введення Swift, спільнота розробників активно прийняла мову, створюючи безліч бібліотек,

інструментів та ресурсів для навчання. Apple також активно підтримує Swift, регулярно випускаючи оновлення та розширення.

Враховуючи ці переваги, Swift не лише пропонує сучасні рішення для викликів розробки програмного забезпечення, але й позиціонує себе як мову, яка буде служити екосистемі Apple на довгі роки вперед. В історії програмування рідко з'являються інструменти, які корінним чином змінюють спосіб, яким ми навчаємося та експериментуємо з кодом. Swift Playgrounds є однією з таких інновацій, яка відкрила двері до світу програмування для безлічі новачків. Ця платформа дозволяє користувачам бачити результати свого коду в реальному часі, без необхідності запуску повного додатка, що значно сприяє процесу навчання.

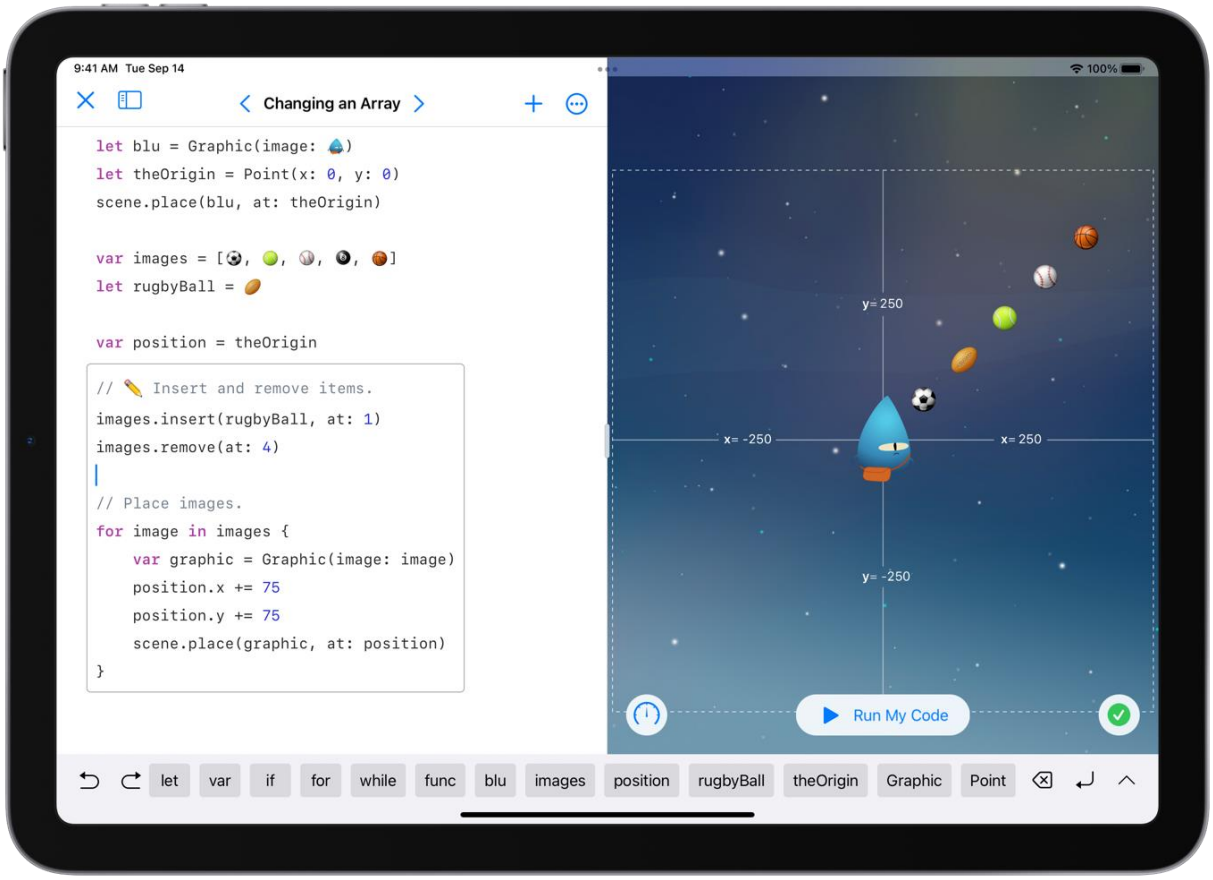


Рис 2.4 Середовище розробки Swift Playgrounds

Крім традиційного текстового введення, Playgrounds пропонує візуалізацію коду, дозволяючи користувачам спостерігати, як їх код

взаємодіє з графічними об'єктами на екрані (рис 2.4). Це робить інструмент особливо корисним для вивчення концепцій, які можуть бути складними для пояснень словами.

Apple також розробила цілий ряд готових навчальних модулів для Swift Playgrounds, спрямованих на вивчення основ програмування, алгоритмів та інших ключових тем, що робить Playgrounds ідеальним інструментом для використання у навчальних закладах і для самостійного навчання.

Особливу увагу варто звернути на доступність Swift Playgrounds. Цей інструмент наявний для використання на iPad, що робить його ідеально придатним для використання в різних освітніх контекстах.

Загалом, Swift Playgrounds революціонізує процес вивчення програмування. З його допомогою Apple не просто пропонує новий інструмент для розробки, а й розширює доступ до програмування для нового покоління студентів. Swift, завдяки своїм освітнім можливостям, стає мостом між традиційними методами програмування та новими підходами до освіти.

Технічні аспекти Swift

Swift було створено з акцентом на простоту та читабельність. Наприклад, не потрібно використовувати підкреслення для позначення властивостей об’єкту або крапку з комою для завершення інструкції. Це робить код менш завантаженим та легшим для читання.

В Swift наявна сильна типізація, при цьому мова пропонує можливість автоматичного визначення типу даних (type inference). Тим не менше, якщо потрібно, розробники можуть явно вказувати типи даних.

Однією з унікальних особливостей Swift є введення опціональних типів. Опціонали дозволяють представити відсутність значення, замість того, щоб використовувати такі значення як nil (null в інших мова програмування). Це забезпечує додатковий рівень безпеки, оскільки Swift

вимагає явного розпаковування опціоналів і попереджає багато поширених помилок, таких як спроби доступу до недійсних посилань.

Swift функції можна передавати в якості аргументів в інші функцій, повертати їх як значення або присвоювати змінним. Це робить мову надзвичайно гнучкою і дозволяє використовувати функціональні підходи до програмування.

Загалом, синтаксис Swift є відображенням бажання Apple створити мову, яка є не тільки потужною, але й інтуїтивно зрозумілою та легкою для навчання. Вона об'єднує найкращі аспекти багатьох сучасних мов програмування і вносить власні інноваційні рішення для поліпшення процесу розробки.

Управління пам'яттю є ключовим елементом будь-якої мови програмування, і Swift взяв багато найкращих підходів від своїх попередників, особливо від Objective-C. Автоматичний підрахунок посилань (ARC) було запроваджено в Objective-C перед Swift, але в Swift він став ще більш централізованим. Він автоматично вирішує задачі управління пам'яттю, звільняючи розробників від необхідності вручну керувати життєвим циклом об'єктів. Це зменшує кількість помилок, пов'язаних із управлінням пам'яттю, і робить код чистішим та безпечнішим.

Однак, незважаючи на переваги ARC, можливі ситуації, коли два об'єкти посилаються один на одного, створюючи цикл посилань. Swift вводить поняття "слабких" (weak) та "невласних" (unowned) посилань для розриву цих циклів та вирішень потенційних витоків пам'яті.

Swift пропонує типи даних за значенням (структури та перелічення), так і за посиланням (класи, закриття). Типи за значенням копіюються при передачі, тоді як типи за посиланням передаються за допомогою посилання. Ця відмінність важлива для розуміння того, як дані зберігаються та передаються в пам'яті.

Закриття в Swift можуть "захоплювати" та зберігати посилання на будь-які константи та змінні з контексту, в якому вони були створені. Це може призвести до неочікуваних витоків пам'яті, якщо не бути обережним. На допомогу вирішення цих проблем приходить вищезгаданий ARC.

Управління пам'яттю в Swift створено так, щоб забезпечити максимальну продуктивність та безпеку. Але, як і в будь-якій мові програмування, розробники повинні знати та розуміти ці механізми, щоб ефективно та безпечно працювати з пам'яттю.

Коли ми говоримо про Swift, одна з його найвидатніших особливостей – це поєднання об'єктно-орієнтованого та функціонального стилів, що робить Swift здатним якісно виконувати широкий спектр завдань.

Функціональне програмування (FP) є парадигмою програмування, що концентрується на обчисленні результатів за допомогою функцій, відсутність станів та змінних даних. В основі цього підходу лежить ідея, що вхідне значення функції завжди дасть одне й те ж вихідне значення, гарантуючи відсутність побічних ефектів.

Swift пропонує ряд функціональних можливостей, які спрощують процес розробки. Наприклад, функції вищого порядку, такі як map, filter та reduce, дозволяють розробникам легко маніпулювати колекціями даних без написання багатьох рядків коду. Ці функції приймають інші функції в якості параметрів, дозволяючи розробникам створювати потужні обчислювальні ланцюги з лаконічним синтаксисом.

Крім того, замикання в Swift пропонують способи визначення функцій на льоту, що можна передавати як аргументи або повертати з інших функцій. Замикання також можуть "захоплювати" значення зі свого зовнішнього контексту, надаючи потужні можливості для локального збереження стану і взаємодії з ним.

Закінчуючи, можна сказати, що функціональні можливості Swift не лише дозволяють розробникам писати лаконічний та чистий код, але й сприяють створенню більш безпечних та надійних додатків, завдяки структурованому підходу до обробки даних і відсутності побічних ефектів.

Інтеграція Objective-C та Swift

З появою Swift багато розробників стали задаватись питаннями про майбутнє Objective-C і як забезпечити гладкий перехід між двома мовами. Apple активно підтримує інтеграцію між Swift та Objective-C, що дозволяє розробникам використовувати код на обох мовах у одному проекті.

Основні особливості інтеграції двох мов

- міст між мовами, створений Apple, через який розробники можуть вільно використовувати класи та методи Objective-C в Swift і навпаки. Наприклад, при імпорті заголовка Objective-C у файл мосту Swift, весь код Objective-C стає доступний у Swift без будь-яких додаткових зусиль;
- сумісність з бібліотеками: більшість фреймворків та бібліотек, створених для iOS і macOS були написані на Objective-C. Swift може використовувати ці бібліотеки безпосередньо. Це означає, що нові проекти на Swift можуть легко інтегруватися з існуючими ресурсами;
- анотації для Objective-C: щоб забезпечити гладке спілкування між Swift і Objective-C, Apple ввела ряд анотацій для Objective-C. За допомогою цих анотацій розробники можуть вказати, як конкретний код Objective-C повинен взаємодіяти з Swift. Наприклад, анотації як nonnull та nullable допомагають вказати, чи може об'єкт бути nil, що покращує безпеку типів при використанні в Swift;
- система атрибутів: щоб допомогти в інтеграції між Swift та Objective-C, Apple ввела систему атрибутів, таких як @objc та @nonobjc, які дозволяють контролювати видимість та поведінку Swift коду в контексті Objective-C;

– еволюція, але не виключення: хоча Swift є майбутнім для розробки на платформах Apple, компанія не планує повністю відмовлятись від Objective-C. Замість цього акцент зроблено на гармонійне співіснування мов, дозволяючи розробникам вибирати найкраще для конкретного завдання.

Важливо згадати і такі інструменти як Swiftify, що дозволяє для автоматично конвертувати код із Objective-C в Swift. Хоча він не забезпечує 100% точної конвертації, це корисний інструмент для швидкої міграції основних частин коду.

Інтеграція Swift та Objective-C є ключовою особливістю екосистеми розробки Apple. Завдяки активному розвитку інструментів та ресурсів, розробники можуть легко комбінувати обидві мови в своїх проектах, використовуючи кожну з них там, де вона найкраще підходить.

Майбутнє Swift і його роль в екосистемі Apple

З моменту введення Swift в 2014 році його роль в екосистемі Apple значно посилилася. Якщо спочатку Swift був просто новим інструментом для розробників, то зараз він є домінуючим елементом стратегії Apple у сфері програмування.

Основні напрямки розвитку:

– Swift на сервері: один з напрямків розвитку Swift - це його використання на сервері. За допомогою таких ініціатив як SwiftNIO, Apple активно просуває ідею використання Swift для серверної розробки, намагаючись зробити його конкурентоспроможним у порівнянні з іншими мовами серверної сторони;
– SwiftUI: Це новий декларативний спосіб створення інтерфейсів користувача на всіх платформах Apple. Ця технологія показує, як Apple бачить майбутнє розробки програмного забезпечення, і яку роль у цьому буде відігравати Swift;

- підтримка спільноти: Apple активно підтримує спільноту Swift, відкривши код Swift і запустивши Swift.org. Це дозволяє розробникам брати активну участь у формуванні майбутнього мови;
- освіта: як ми вже згадували, за допомогою Swift Playgrounds Apple намагається зробити Swift доступним для нових генерацій розробників. Зосереджуючись на освіті, Apple прагне забезпечити довгостроковий інтерес і підтримку мови;
- перехід на ARM-архітектуру: З переходом Mac на ARM-архітектуру, Swift отримає нові можливості оптимізації, враховуючи особливості нової платформи. Це може призвести до ще більшої продуктивності та ефективності Swift-додатків.

Враховуючи всі ці фактори, можна стверджувати, що Swift не просто залишиться ключовою мовою для розробки на платформах Apple, а й продовжить зростання та розвиток, займаючи все більше сегментів у світі програмування. Apple безумовно вкладає багато зусиль у підтримку та розвиток цієї мови, роблячи її ще більш привабливою для розробників з усього світу.

2.3. UIKit та SwiftUI: Шлях до декларативних інтерфейсів

При розгляді SwiftUI та UIKit неможливо не звернути увагу на одну з найбільших різниць між ними: підхід до створення інтерфейсів. Де UIKit базується на імперативному підході, SwiftUI використовує декларативний стиль програмування. Давайте розберемося в основних відмінностях цих підходів.

Імперативний підхід (UIKit): Імперативне програмування зосереджено на тому, як досягти бажаного результату. Розробник вказує послідовність дій, які необхідно виконати, щоб досягти потрібного стану. В контексті інтерфейсу користувача, це може включати ручне створення елементів інтерфейсу, встановлення їх властивостей та відповідь на різні події, які відбуваються під час взаємодії користувача.

Декларативний підхід (SwiftUI): На відміну від імперативного, декларативний підхід фокусується на тому, що розробник хоче досягти, а не як. Ви просто описуєте бажаний стан інтерфейсу, і система вирішує, як досягти цього стану. Це веде до більш стислого та зрозумілого коду, який зосереджений на зовнішньому вигляді та функціональності, а не на деталях реалізації.

Переваги декларативного підходу:

- більш читабельний код: Оскільки декларативний код фокусується на кінцевому результаті, він стає більш читабельним і зрозумілим;
- спрощена реакція на зміни стану: Зміна стану компонентів інтерфейсу часто призводить до автоматичного оновлення відображення без необхідності явно це вказувати;
- консистентність інтерфейсу: Опис інтерфейсу як функції від його стану може допомогти уникнути помилок та несумісностей у відображенні.

Тим не менш, хоча SwiftUI і пропонує багато переваг, важливо розуміти, що UIKit лишається важливим інструментом у екосистемі Apple. У багатьох випадках вибір між UIKit та SwiftUI буде залежати від конкретних потреб проекту, досвіду команди та необхідності підтримки різних версій платформи.

SwiftUI, без сумніву, став великим кроком вперед у світі розробки інтерфейсів для продуктів Apple. Адаптація цього фреймворку приносить численні переваги для розробників і команд. Однією з ключових переваг є його декларативний синтаксис. Розробники тепер можуть описувати бажаний вигляд і поведінку інтерфейсу, не концентруючись на детальній послідовності дій для його створення. Це не тільки робить код коротшим і більш зрозумілим, але й зменшує шанси на помилки.

Окрім цього, функція Live Preview в Xcode дозволяє розробникам бачити, як їхній інтерфейс виглядає і взаємодіє з діями користувача в реальному часі, доки вони пишуть код, що суттєво прискорює розробку.

Однією з найбільших переваг SwiftUI є можливість використовувати одну базу коду для створення додатків для різних платформ Apple, включаючи iOS, macOS, watchOS та tvOS. Це спрощує процес розробки та обслуговування додатків на різних платформах.

SwiftUI також користується сучасними особливостями Swift, такими як property wrappers, що робить його дуже ще більш гнучким. Система розміщення SwiftUI також забезпечує, що додатки автоматично адаптуються до різних розмірів екранів і орієнтацій, гарантуючи послідовний досвід користувача.

Тим не менш, при вирішенні переходу на SwiftUI важливо також враховувати деякі обмеження, зокрема щодо підтримки старіших версій ОС. Проте, в цілому, переваги часто переважають можливі недоліки, особливо для нових або проектів, що активно розвиваються.

Технічні аспекти SwiftUI

З моменту своєї анонсації на WWDC 2019, SwiftUI зазнав бурхливого розвитку і прийняття серед розробників Apple. Давайте докладніше розглянемо деякі з технічних аспектів SwiftUI.

SwiftUI Views. Усі компоненти інтерфейсу в SwiftUI — це представлення (View). Вони являють собою легковісні структури, що можуть бути легко комбінованими або вкладеними один в одного. Завдяки цьому можливо легко створювати складні інтерфейси з простих базових блоків.

Взаємодія з даними. В SwiftUI використовуються специфічні об'єкти, такі як @State, @Binding, @ObservableObject і @EnvironmentObject, для управління та взаємодії з даними в додатку. Це дає можливість розробникам легко зв'язувати інтерфейс з даними, використовуючи систему реактивного програмування.

Анімація. SwiftUI надає простий спосіб анімувати дії користувача. Застосування анімацій до змін в інтерфейсі часто вимагає всього лише додавання одного модифікатора до представлення.

Інтеграція з UIKit та AppKit. Незважаючи на всі переваги SwiftUI, іноді розробникам може знадобитися інтеграція із традиційними UIKit або AppKit компонентами. І тут SwiftUI також не розчаровує, надаючи способи вбудовування цих компонентів безпосередньо в декларативні інтерфейси. Apple зробила можливим використання компонентів UIKit всередині SwiftUI за допомогою UIViewControllerRepresentable та UIViewRepresentable протоколів.

Коли мова заходить про розширення можливостей SwiftUI, важливо розуміти, що ця технологія є частиною більшої екосистеми Apple. Це означає, що інші фреймворки та бібліотеки, доступні в рамках iOS, macOS, watchOS та tvOS, часто можуть бути легко інтегровані з SwiftUI.

SpriteKit та SceneKit. Ці фреймворки дозволяють розробникам створювати 2D та 3D графіку відповідно. Вони можуть бути інтегровані в SwiftUI за допомогою спеціалізованих виглядів, дозволяючи комбінувати традиційні інтерфейси з графічними елементами.

Core ML та Vision. Для розробників, які хочуть інтегрувати можливості машинного навчання та комп'ютерного зору в свої додатки, SwiftUI пропонує швидкий доступ до результатів, отриманих з цих фреймворків, за допомогою декларативних виглядів.

Swift Package Manager. Цей інструмент дозволяє розробникам легко додавати сторонні бібліотеки та фреймворки до своїх проектів. Це особливо корисно для розширення можливостей SwiftUI за допомогою великої спільноти розробків.

Все це підкреслює гнучкість та адаптивність фреймворку SwiftUI як основного інструменту для сучасної розробки інтерфейсів в екосистемі Apple.

Висновки до розділу 2

Розглянуто основні інструменти, які використовуються для розробки програмного забезпечення на платформах Apple. Основне місце в інструментарії розробника займає середовище Xcode, яке є центральним місцем для створення, тестування та відлагодження програм. Xcode надає розробникам доступ до потужних засобів для ефективної роботи.

Розглянуто мови програмування Swift та Objective-C. Swift, як новітній та більш безпечний варіант, рекомендується для нових проектів, в той час як Objective-C може бути корисним для підтримки старих проектів.

Проаналізовано основні технології для створення користувацьких інтерфейсів - UIKit та SwiftUI. SwiftUI, як більш новий та декларативний фреймворк, який значно спрощує процес розробки інтерфейсів, особливо для нових проектів, в той час як UIKit може залишатися вибором для проектів, які вже використовують його.

Зроблено висновок про те, який інструментарій краще використовувати розробникам для створення програмного забезпечення для продуктів Apple, і показує, як вибір правильних інструментів може поліпшити ефективність та продуктивність процесу розробки.

РОЗДІЛ 3. РОЗРОБКА МУЛЬТИПЛАТФОРМЕННОГО ДОДАТКУ ДЛЯ ВЗАЄМОДІЇ З OPENAI API ІЗ МОЖЛИВІСТЮ РОЗПІЗНАВАННЯ МОВЛЕННЯ ТА ЗЧИТУВАННЯ ТЕКСТУ З КАРТИНКИ

Мета додатку: Створення чат-боту із можливістю зчитування тексту з картинок повідомлення та можливістю розпізнавання мовлення.

Розроблений додаток має отримати таку функціональність:

- При першому запуску додаток надає можливість користувачеві ввести OpenAI API ключ.
- Додаток надає інтерфейс чату для спілкування із ChatGPT.
- Є можливість обрати різні моделі для спілкування.
- Інтерефейс чату має текстове поле вводу, а також можливість записувати аудіо, що в живому часі трансформується в текст заповнючи текстове поле;
- Відповідь чат-боту оновлюється у живому часі, не чекаючи на кінцевий результат;
- Додаток надає можливість навести камеру на документ із текстом, зчитати текст і вставити його у повідомлення;
- Додаток має функціонал запиту на генерацію картинок із конфігурацією розміру і кількості екземплярів.
- Має підтримку світлої, темної або системної теми, локалізацію англійської і української мов.

Для нативної розробки додатків під платформи екосистеми Apple зазвичай використовується техніка Apple з встановленим середовищем Xcode, яке надає основний набір інструментів для створення, тестування та відладки додатків. Варто відзначити, що деякі розробники користуються альтернативними методами, такими як використання macOS на віртуальних машинах чи збірку Hackintosh (комп'ютер на базі PC, налаштований для

роботи з macOS). Однак такі підходи можуть бути складнішими, менш стабільними та мати проблеми юридичного характеру.

Нативна розробка (native development) - це процес створення програмного забезпечення або додатків для конкретної платформи або операційної системи, використовуючи офіційні мови програмування та інструменти розробки, які спеціально призначені для даної платформи. Наприклад, розробка мобільних додатків для iOS використовує мову програмування Swift або Objective-C і інструменти розробки від Apple, такі як Xcode. Розробка для платформи Android вимагає використання мов Java або Kotlin та інструментів, які надаються Google.

Нативна розробка дозволяє створювати додатки, які максимально оптимізовані для конкретної платформи, забезпечуючи високий рівень продуктивності та доступ до всіх функцій та можливостей даної платформи. Однак вона також може вимагати більше часу та зусиль для розробки окремих версій додатка для різних платформ (наприклад, iOS та Android).

Порівнюючи нативну розробку з іншими методами, такими як хресна розробка (cross-platform development) або веб-розробка, важливо вибрати підхід, який найкраще відповідає конкретним потребам та цілям проекту.

Перед тим як почати процес розробки додатка необхідно зареєструватися в Apple Developer Program, платна підписка коштує 99 USD на рік. Платна підписка надає доступ до додаткових інструментів, таких як TestFlight для бета-тестування додатків, налаштування пуш-нотіфікацій, а найважливіше — це можливість публікуватись в App Store. Якщо план розробки не передбачає публікацію в App Store, платна підписка не є обов'язковою, що задовольняє наші потреби.

Для розробки мультплатформенного додатка в межах екосистеми Apple ми оберемо фреймворк SwiftUI. Він дозволяє писати нативний код, який буде працювати на різних платформах техніки Apple, включаючи iOS, watchOS, macOS та tvOS. Якщо загальні риси інтерфейсу можливо

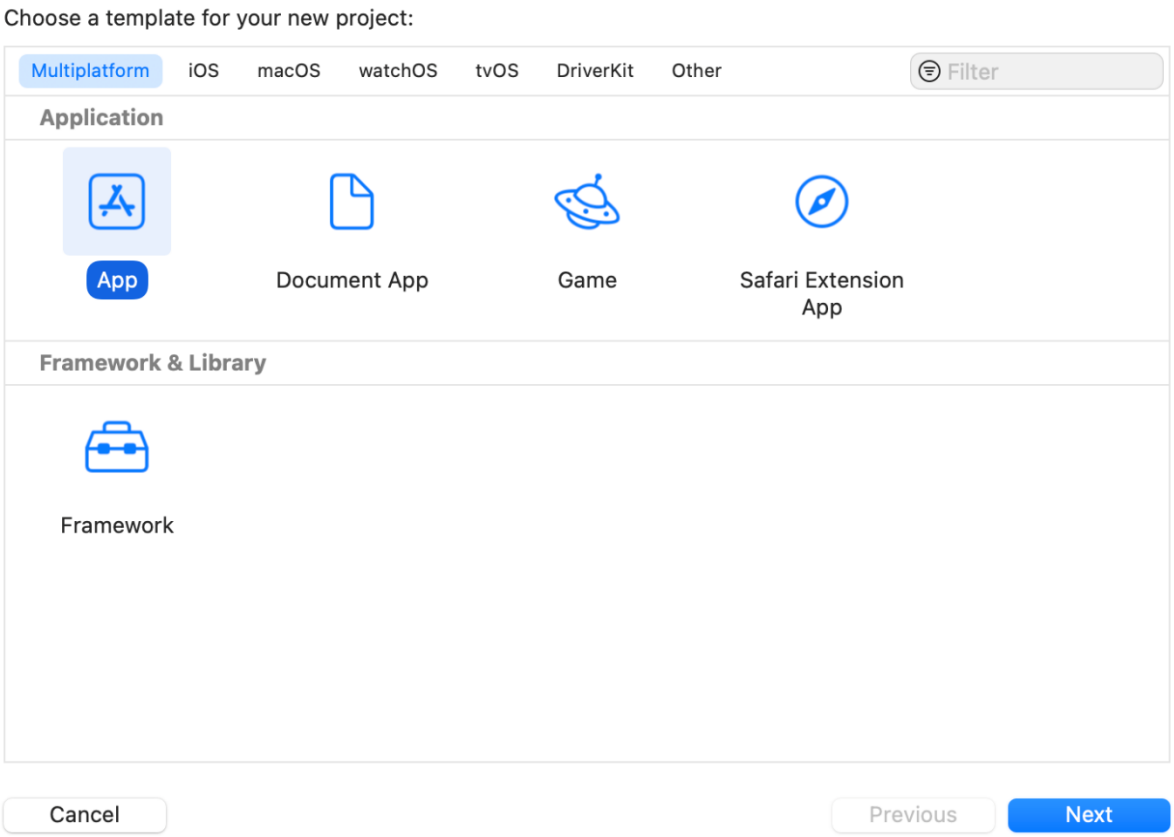


Рис 3.1 Темплейт проекту

реалізувати виключно на SwiftUI, то детальні аспекти потребуватимуть умовних уточнень. Ми зосередимось на платформах iOS (включно з iPadOS) та macOS, які підтримують iPhone, iPad та комп'ютери Mac.

Для використання останніх технологій оберемо 17 мінімальну версію iOS та 14 macOS в налаштуваннях проекту.

Для системи контролю версій використовуватимемо git та github для хостингу репозиторію. Тож для уникнення трекінгу небажаних файлів створимо gitignore файл і додамо до нього стандартний темплейт (рис 3.2)

```
## User settings
xcuserdata/

## compatibility with Xcode 8 and earlier (ignoring not required starting Xcode 9)
*.xcscmblueprint
*.xccheckout

## compatibility with Xcode 3 and earlier (ignoring not required starting Xcode 4)
build/
DerivedData/
*.moved-aside
*.pbxuser
!default.pbxuser
*.mode1v3
!default.mode1v3
*.mode2v3
!default.mode2v3
*.perspectivev3
!default.perspectivev3

## Obj-C/Swift specific
*.hmap

## App packaging
*.ipa
*.dSYM.zip
*.dSYM

timeline.xctimeline
playground.xcworkspace

.build/

Carthage/Build/

# Accio dependency management
Dependencies/
.accio/
```

Рис 3.2 Gitignore файл

Перед початком написання основних компонентів додатка ключовим етапом є вибір архітектури. Щоб уникнути надмірної складності, яка могла б перевантажити розробку, було прийнято рішення відмовитися від використання MVVM у поєднанні із Clean Architecture та іншого роду подібних архітектур. Замість цього було обрано підхід, де бізнес-логіка виноситься у окремі сервіси та менеджери, що дозволяє зручно використовувати фреймворки SwiftData або Core Data, для збереження даних. Цього підходу також дотримується Apple що можна бачити у їхніх туторіалах, де вони акцентують увагу на схожому способі організації компонентів програми.

3.1. Огляд ключових компонентів

Основною складовою частиною додатку є спілкування із OpenAI API, файлова структура Network компонентів виглядає наступним чином (рис 3.3)

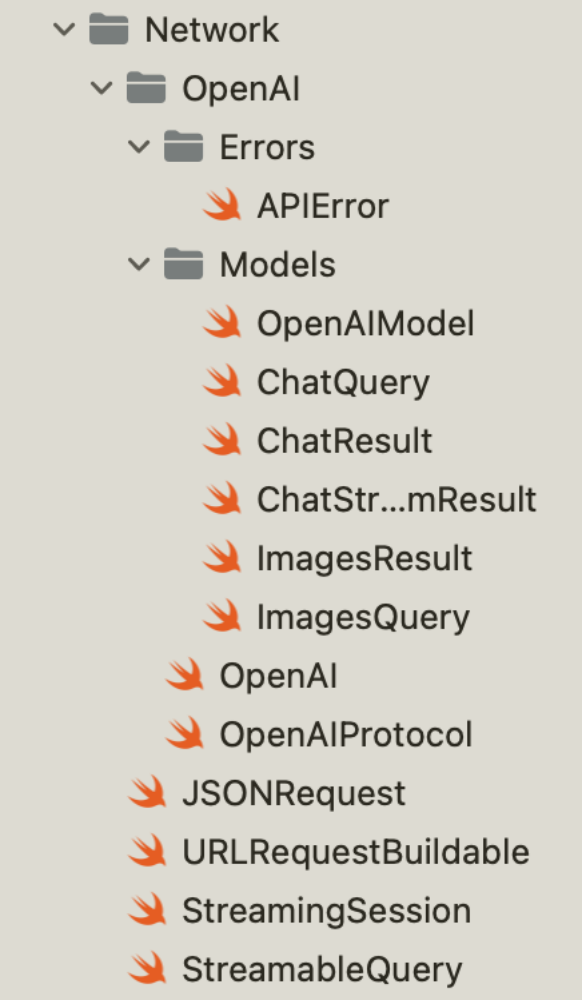


Рис 3.3 Структура Network шару

Інтерфейсом Network який надалі використовуватиметься із інших місць програми є OpenAIProtocol, протоколи у Swift за своєю суттю схожі на інтерфейси в інших мовах програмування. Таким чином виглядає OpenAI протокол з яким комунікують інші місця програми, енкапсулючи реалізацію від користувача.

Розширення протоколу створює міст для використання більш новітнього підходу до написання так званого паралельного коду async/await, що поєднує у собі концепції асинхронності і багатопоточності у контексті мови Swift.

Інтерфейс OpenAIProtocol:

```
protocol OpenAIProtocol {
   func images(query: ImagesQuery, completion: @escaping (Result<ImagesResult, Error>) -> Void)
   func chatsStream(query: ChatQuery, onResult: @escaping (Result<ChatStreamResult, Error>) -> Void, completion: ((Error?) -> Void)?)
}

extension OpenAIProtocol {
   func images(query: ImagesQuery) async throws -> ImagesResult {
      try await withCheckedThrowingContinuation { continuation in
         images(query: query) { result in
            switch result {
            case let .success(success):
               return continuation.resume(returning: success)
            case let .failure(failure):
               return continuation.resume(throwing: failure)
            }
         }
      }
   }

   func chatsStream(query: ChatQuery) -> AsyncThrowingStream<ChatStreamResult, Error> {
      return AsyncThrowingStream { continuation in
         return chatsStream(query: query)  { result in
            continuation.yield(with: result)
         } completion: { error in
            continuation.finish(throwing: error)
```

```
        }
      }
    }
}
```

Специфіку реалізації спадкоємця OpenAI протоколу можна переглянути у додатках. Використовується OpenAIProtocol у ChatManager класі в методі completeChat, що відповідає за логіку роботи чату та у ImageManager, який у свою чергу відповідає за генерацію картинок.

Сигнатура методу completeChat:

```
@MainActor func completeChat(convoID: ConvoID, model: OpenAIModel) async
```

Метод chatsStream, що за своєю суттю слідкує за оновленням контенту повідомлення та пропагує його або будь-які помилки, які трапились під час виконання блоку do до графічного інтерфейсу через оновлення @Published властивостей класу. @MainActor є важливим атрибутом, який дозволяє впевнитись, що оновлення графічного інтерфейсу відбуватиметься на головному потоці.

Властивості класу ChatManager:

```
@Published var convos: [Convo] = []
@Published var convoErrors: [ConvoID: Error] = [:]
@Published var selectedConvoID: ConvoID?
@Published var inProgress: Bool = false

var selectedConvo: Convo? {
    selectedConvoID.flatMap { id in
        convos.first { $0.id == id }
    }
}

var selectedConvoPublisher: AnyPublisher<Convo?, Never> {
    $selectedConvoID.receive(on: RunLoop.main).map { id in
        self.convos.first(where: { $0.id == id })
    }
    .eraseToAnyPublisher()
}
```

Також важливою частиною програми це процес зчитування тексту з картинки, для цього ми маємо структуру VisionService. Цей сервіс використовує VNImageRequestHandler з Vision фреймворку Apple для обробки зображень. Коли зображення передається до методу detectText, він ініціює текстовий запит VNRecognizeTextRequest. Цей запит сканує зображення на наявність тексту. Після виявлення тексту, він збирає розпізнані рядки тексту, об'єднує їх та повертає як результат. Якщо тексту не виявлено, сервіс викликає помилку. Також, для забезпечення більшої точності розпізнавання, рівень встановлено як .accurate і враховується локальний мовний код користувача.

Частина блоку VNRecognizeTextReques, де fullText є результатом скану:

```
guard let recognizedText = request.results as? [VNRecognizedTextObservation]
else { return }

var fullText = ""
let maximumCandidates = 1

for observation in recognizedText {
   guard let candidate = observation.topCandidates(maximumCandidates).first
else { continue }
   let recognizedLine = candidate.string.trimmingCharacters(in:
.whitespacesAndNewlines)

   if recognizedLine.count > 3 {
      fullText.append(recognizedLine + "\n")
   }
}
```

Розглянемо ще один із компонентів шару графічного інтерфейсу, а саме структуру ChatView. Частина ChatView, що відповідає за відображення чату, складається з основного компоненту NavigationStack. Всередині цього компоненту розташована прокручувана область, що користується ScrollViewReader для контролю над процесом.

Центральною частиною є список повідомлень, створених за допомогою List. Він використовує ForEach для ітерації кожного повідомлення з розмови та відображення його за допомогою MessageView. Додатково, анімації та стилі застосовуються до списку для забезпечення приємного досвіду користувача.

Приклад застосування структури List:

```
List {
   ForEach(convo.messages) { message in
      MessageView(message: message)
   }
   .listRowSeparator(.hidden)
}
.listStyle(.plain)
.animation(.default, value: convo.messages)
```

Також при зміні convo відбувається автоматичне прокручування до останнього повідомлення та оновлення введеного тексту.

Модифікатор спостереження за змінною convo:

```
.onChange(of: convo) { _, newValue in
   if let lastMessage = newValue.messages.last {
      scrollViewProxy.scrollTo(lastMessage.id, anchor: .bottom)
   }
}
```

Додатково, коли View вперше з'являється на екрані, ми запрошуємо авторизацію для розпізнавання мови. Під списком повідомлень, якщо існує

якась помилка з chatManager, вона відображається за допомогою компоненту Text.

В нижній частині відображення розміщено InputBarView, яка служить для введення тексту та відправлення повідомлень. Коли користувач подає команду відправити повідомлення або торкається мікрофону, виконуються відповідні функції.

Також додано атрибути для керування назвою навігаційної панелі та її режимом відображення для iOS платформи.

3.2 Огляд основних компонентів інтерфейсу

Екран введення ключа API (рис 3.4) є критичним компонентом для додатків, які вимагають взаємодії з зовнішніми веб-сервісами. Такий екран зазвичай розробляється з метою забезпечення безпеки та конфіденційності даних користувача, оскільки пряме зберігання ключів API в додатках може призвести до зловживань.

Основні компоненти екрана:

- Поле для введення: Центральний елемент інтерфейсу, де користувач може ввести свій особистий ключ API. Зазвичай це текстове поле, яке підтримує форматування тексту, щоб відображати введений ключ належним чином.
- Кнопка підтвердження: Після введення ключа користувач натискає на цю кнопку, щоб підтвердити його та надіслати на перевірку.

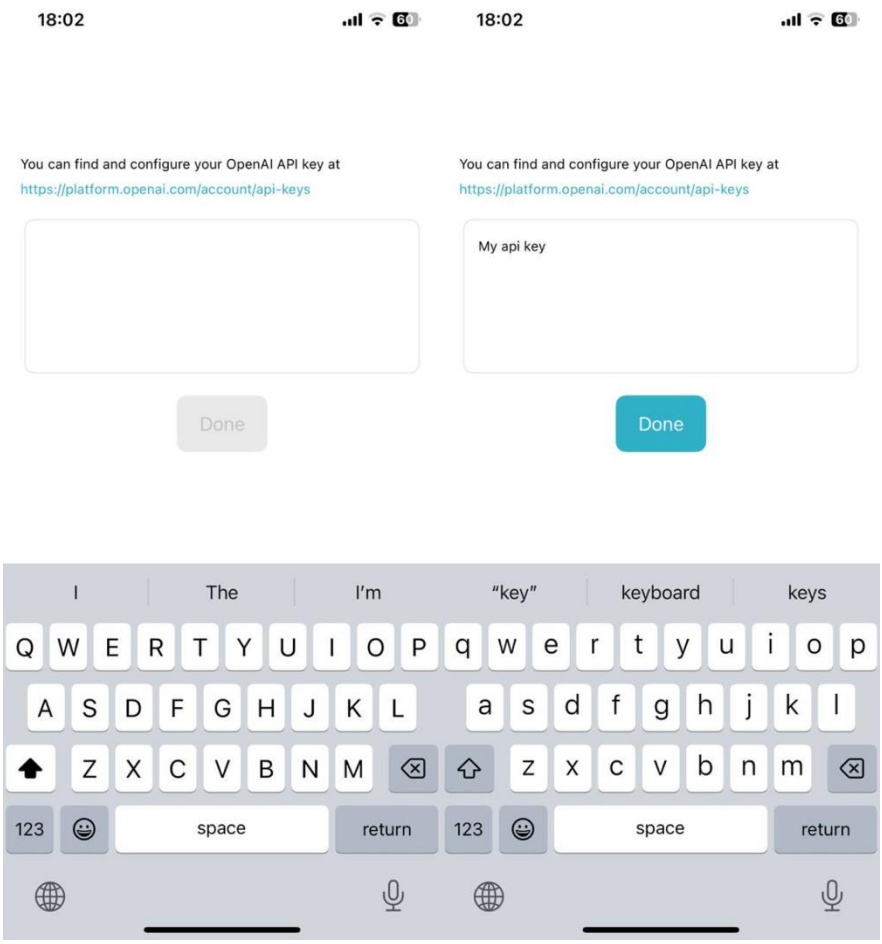


Рис 3.4 Екран введеня API ключу

Екран списку чатів (рис 3.5) представляє собою класичний список, де кожен елемент являє собою назву чату, зручно відображену для користувача. Верхня панель може містити заголовок "Діалоги" і кнопку дії для створення нового чату. Користувач може переглядати весь список доступних чатів, вибирати будь-який з них для переходу до індивідуального чату. Також передбачена можливість швидкого видалення чату зі списку. При виборі чату він виділяється, що підтверджує вибір користувача. Існує анімація переходу між списком чатів та індивідуальним чатом.

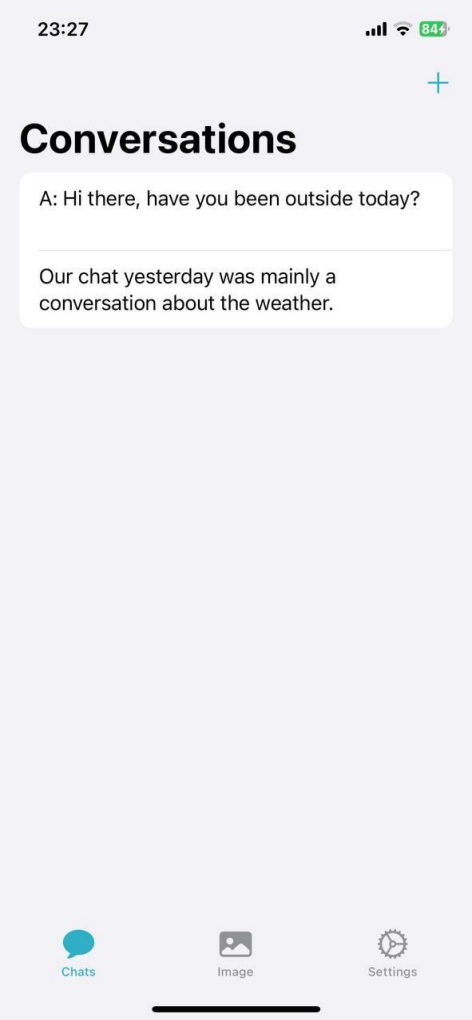


Рис 3.5 Екран зі списком чатів

Екран чату (рис 3.6) розроблено з урахуванням комфорту користувача: повідомлення відображаються зліва. Внизу розташоване поле введення тексту з кнопкою відправлення. Користувач може відправляти текстові повідомлення, при цьому для зручності передбачена можливість записувати голосові повідомлення, які транслюються у текст. Всі повідомлення миттєво відображаються в історії чату. При відправці нового повідомлення відбувається м'яка анімація його появи в історії чату. Якщо користувач отримує відповідь, воно також анімовано відображається в чаті, поетапно заповнюючи відповідь. У випадку виникнення помилок або проблем із з'єднанням, користувачеві відображається відповідне повідомлення.

Також при натисканні на іконку “інше” показується меню у якому є можливість обрати модель чату, а також просканувати камерою текст вставивши його у повідомлення.

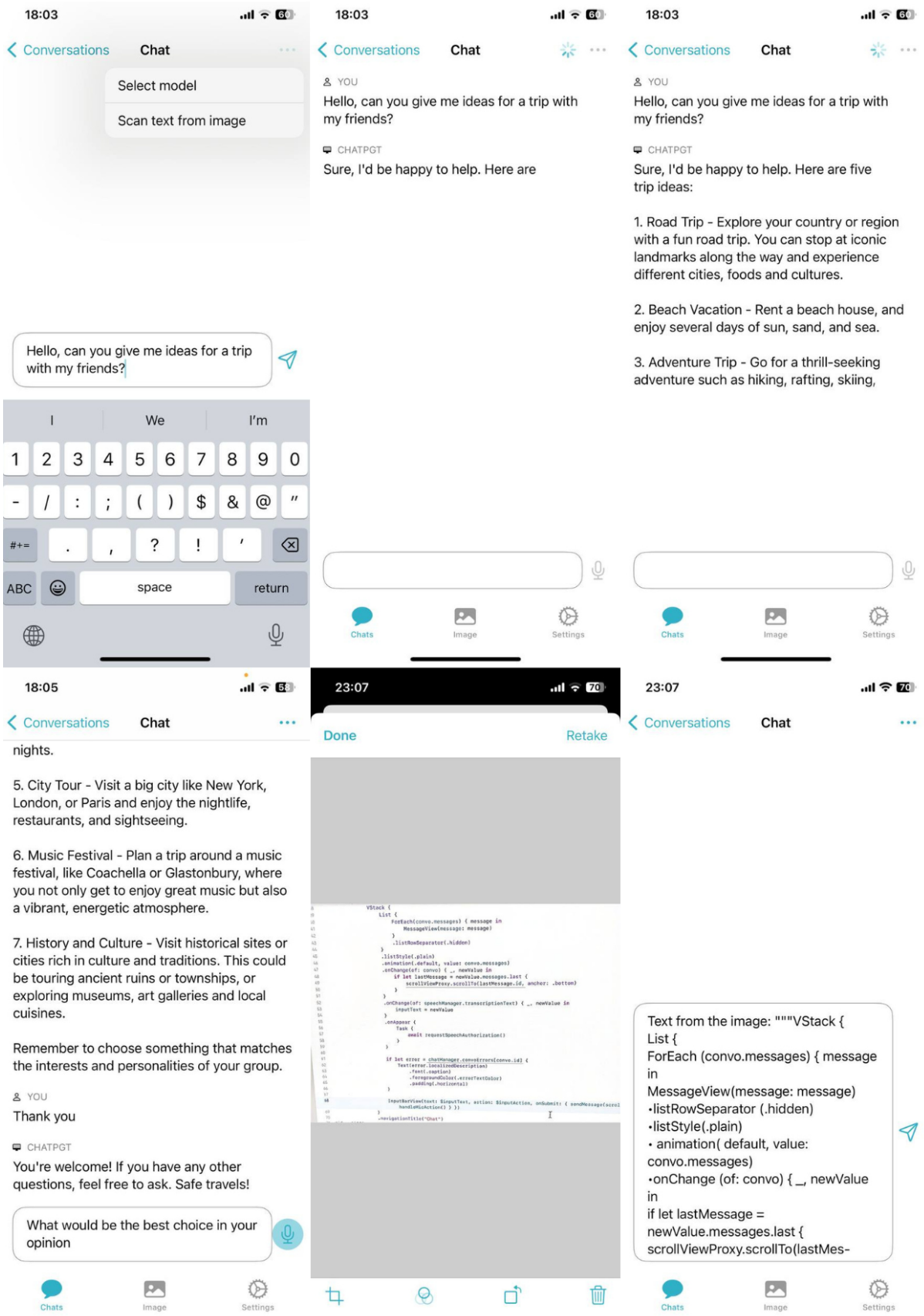


Рис 3.6 Екран чату у різних станах

На екрані генерації зображення (рис 3.7) в додатку користувач може

ввести запит, наприклад "Image of a cow", обрати необхідну кількість зображень та їх розмір. Після натискання на кнопку "Create Image" очікується, що додаток створює зображення відповідно до введеного запиту, цей процес також анімується.

Після того як користувач вводить свій запит і натискає "Create Image", додаток генерує відповідне зображення, яке відображається на екрані, під ним розташована назва файлу, тип та розмір. Зображення можно попередньо переглянути або отримати доступ для завантаження через ресурс на "core.windows.net".

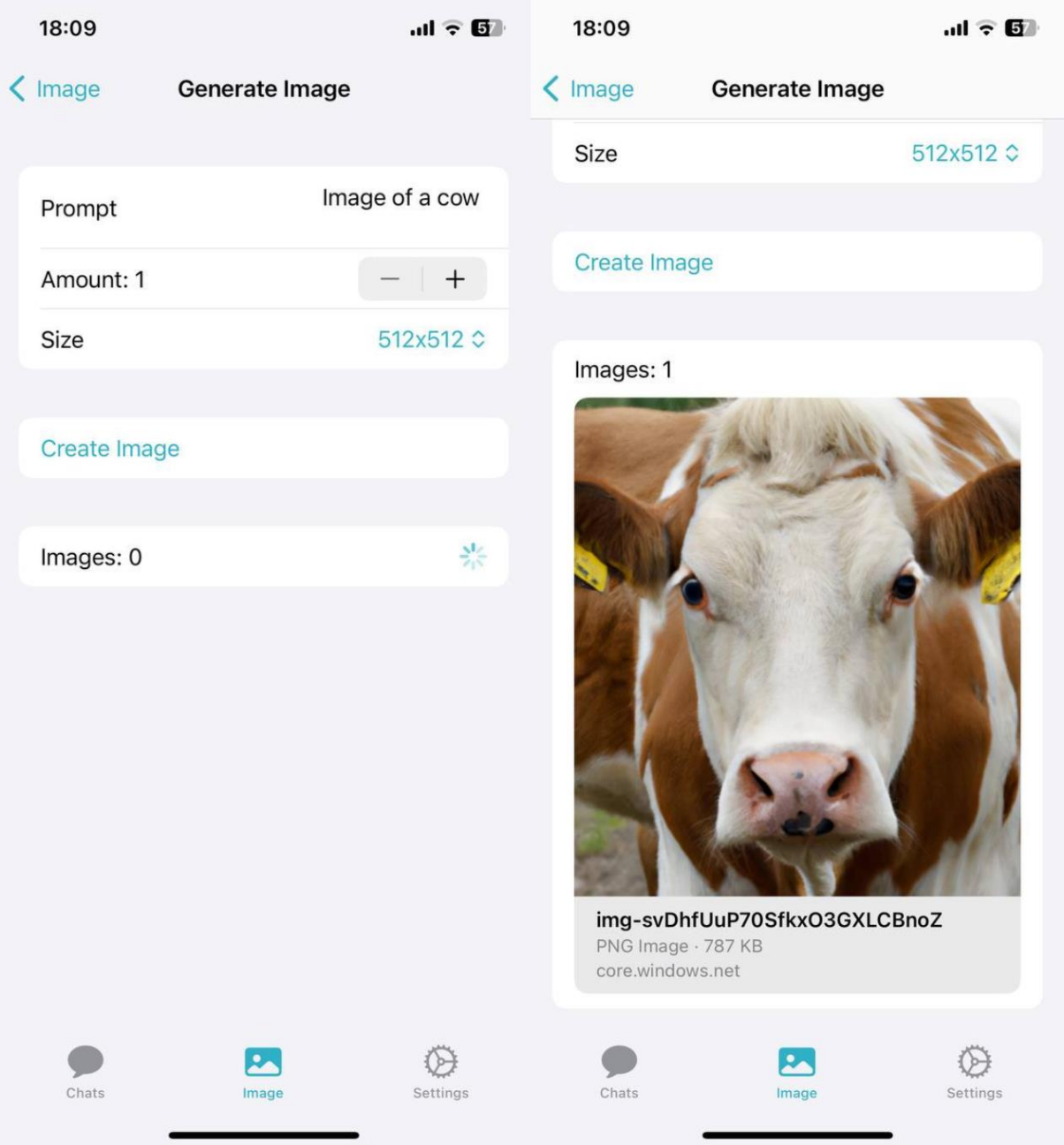


Рис 3.7 Екран генерації зображень

Екран налаштувань (рис 3.8) із українською локалізацією і обраною темною темою. Однією із опцій цього розділу є вибір кольорової схеми, де користувач може обрати одну з трьох доступних кольорових схем для інтерфейсу додатка: "Системна", "Світла" або "Темна". Системна кольорова схема означає, що додаток буде використовувати кольорову схему, встановлену на рівні операційної системи пристрою користувача. Другою опцією є можливість змінити API ключ.

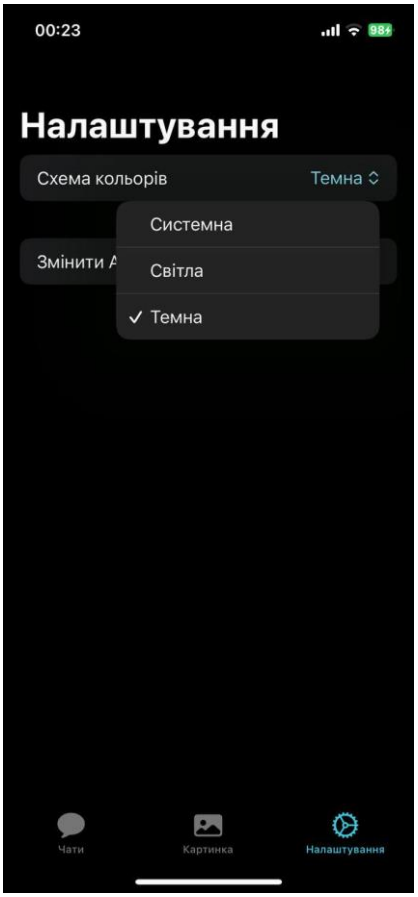


Рис 3.8 Екран налаштувань

Висновки до розділу 3

Розроблено додаток на основі SwiftUI, що надає можливість спілкуватись із моделлю OpenAI ChatGPT. Додаток дозволяє користувачам отримувати швидкі та змістовні відповіді в реальному часі.

Завдяки інтеграції із Vision framework, користувачі мають змогу розпізнавати текст з картинок і вставляти його безпосередньо в діалогове вікно для подальшого використання.

Додаткова функція перетворення голосу в текст за допомогою Speech framework дозволяє користувачам говорити, а не вводити текст вручну, що робить спілкування з моделлю ще простішим та зручнішим.

Генерація картинки: Функціонал запиту на генерацію картинки розширює можливості взаємодії з моделлю, додаючи візуальний аспект.

Сторінка налаштувань дозволяє користувачам кастомізувати досвід взаємодії з додатком, враховуючи їх індивідуальні потреби.

У майбутньому планується розширення функціоналу додатку, що зробить його ще більш адаптованим до потреб користувачів.

ВИСНОВКИ

В ході даного дослідження вивчено концепції розробки програмного забезпечення в екосистемі Apple, попутно розглядаючи, такі інструменти як Core ML і Create ML, а також високорівневі фреймворки для обробки даних. Особлива увага приділена питанням приватності в машинному навчанні та порівнянню інструментів Apple із засобами інших екосистем.

Було проведено аналіз основних інструментів та мов програмування, доступних розробникам для створення додатків під екосистему Apple, з акцентом на Xcode, SwiftUI та мову програмування Swift.

На прикладі мультиплатформенного додатку для взаємодії з OpenAI API було продемонстровано практичний застосунок знань та навичок, отриманих під час вивчення попередніх розділів. Цей додаток комбінує розпізнавання мовлення, зчитування тексту з картинки та інші можливості, що робить його відмінним прикладом сучасної розробки програмного забезпечення.

В якості підсумку, ця дипломна робота представляє цілісний погляд на можливості та інструменти, доступні розробникам в екосистемі Apple, а також демонструє практичний застосунок цих знань. Набуті знання та досвід можуть служити відмінною базою для подальшого розвитку та вдосконалення додатків під платформи Apple.

Результатом цієї роботи є програмний продукт, фукнціоналом якого є можливість спілкування з чат-ботом зі штучним інтелектом, робити запити на генерацію картинок за текстовим запитом. Додаток надає можливість користувачеві змінити базові налаштування, такі як вибір кольорової схеми, а також підтримує локалізацію української та англійської мов.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Mars Geldard, et al. “Practical Artificial Intelligence with Swift” O’Reilly, (2019), 493 pages.

2. Chris Eidhof, et al. “Advanced Swift” 5th editon, (2022), 362 pages.

3. Matt Neuburg “iOS 15 Programming Fundamentals with Swift”, (2021), 700

4. Apple Developer Documentation “Core ML”[Електронний ресурс]. – Посилання на ресурс: https://developer.apple.com/documentation/coreml

5. Apple Developer Documentation “Create ML” [Електронний ресурс]. – Посилання на ресурс: https://developer.apple.com/documentation/createml

6. Apple Developer Documentation “Vision” [Електронний ресурс]. – Посилання на ресурс: https://developer.apple.com/documentation/vision

7. Apple Developer Documentation “Natural Language” [Електронний ресурс]. – Посилання на ресурс: https://developer.apple.com/documentation/naturallanguage

8. Apple Developer Documentation “Sound Analysis” [Електронний ресурс]. – Посилання на ресурс: https://developer.apple.com/documentation/soundanalysis

9. Apple Developer Documentation “Speech” [Електронний ресурс]. – Посилання на ресурс: https://developer.apple.com/documentation/speech

10. Apple Developer Documentation “Xcode” [Електронний ресурс]. – Посилання на ресурс: https://developer.apple.com/documentation/xcode

11. Swift Language Documentation [Електронний ресурс]. – Посилання на ресурс: https://www.swift.org/documentation

12. Antonio Bello, et al. “SwiftUI by Tutorials”, (2021), 376 pages.

13. Artificial Intelligence at Apple – Two Current Applications [Електронний ресурс]. – Посилання на ресурс: https://emerj.com/ai-sector-overviews/ai-at-apple/

14. The Use of Artificial Intelligence Technology in Apple Devices [Електронний ресурс]. – Посилання на ресурс: https://roboticsandautomationnews.com/2021/06/14/the-use-of-artificial-intelligence-technology-in-apple-devices/43866

15. Deploying Transformers on the Apple Neural Engine [Електронний ресурс]. – Посилання на ресурс: https://machinelearning.apple.com/research/neural-engine-transformers

16. Bring Machine Learning to iOS apps using Apache MXNet and Apple Core ML [Електронний ресурс]. – Посилання на ресурс: https://aws.amazon.com/blogs/machine-learning/bring-machine-learning-to-ios-apps-using-apache-mxnet-and-apple-core-ml

17. Machine Learning on MacRumors [Електронний ресурс]. – Посилання на ресурс: https://www.macrumors.com/guide/machine-learning

18. Core ML Explained: Apple's Machine Learning Framework [Електронний ресурс]. – Посилання на ресурс: https://www.createwithswift.com/core-ml-explained-apples-machine-learning-framework

19. Apple Core ML: Easily Leverage the Power of Machine Learning [Електронний ресурс]. – Посилання на ресурс: https://www.iflexion.com/blog/coreml

20. Planning your macOS app [Електронний ресурс]. – Посилання на ресурс: https://developer.apple.com/macos/planning

21. How to Develop Apple Apps: Using Xcode & Swift to Program for iOS & macOS [Електронний ресурс]. – Посилання на ресурс: https://www.sitepoint.com/develop-apple-apps

22. iOS Development Trends for 2023 [Електронний ресурс]. – Посилання на ресурс: https://talent500.co/blog/ios-development-trends-for-2023

23. OpenAI API Reference [Електронний ресурс]. – Посилання на ресурс: https://platform.openai.com/docs/api-reference

24. AppKit Integration [Електронний ресурс]. – Посилання на ресурс: https://developer.apple.com/documentation/swiftui/appkit-integration

25. SwiftUI Tutorials [Електронний ресурс]. – Посилання на ресурс: https://developer.apple.com/tutorials/swiftui

26. Hacking with iOS: SwiftUI Edition [Електронний ресурс]. – Посилання на ресурс: https://www.hackingwithswift.com/books/ios-swiftui

27. Discover SwiftUI [Електронний ресурс]. – Посилання на ресурс: https://www.swiftbysundell.com/discover/swiftui

28. Apple Developer Documentation “UIKit” [Електронний ресурс]. – Посилання на ресурс: https://developer.apple.com/documentation/uikit

29. Advanced cross-platform apps using local Swift packages and UIKit/AppKit [Електронний ресурс]. – Посилання на ресурс: https://itnext.io/advanced-cross-platform-apps-using-local-swift-packages-and-uikit-appkit

30. AppKit for UIKit Developers [Електронний ресурс]. – Посилання на ресурс: https://www.objc.io/issues/14-mac/appkit-for-uikit-developers

Додаток А

Код програми

```
//  InferifyApp.swift
import SwiftUI

@main
struct InquirifyApp: App {
    @AppStorage(AppStorageKeys.apiKey) var apiKey = ""
    @AppStorage(AppStorageKeys.userColorScheme) var userColorScheme:
UserColorScheme = .system

    @State var isShowingAPIConfigModal: Bool = true

    let getID: () -> String
    let getDate: () -> Date

    init() {
        self.getID = {
            let id = UUID().uuidString
            return id
        }
        self.getDate = Date.init
    }

    var body: some Scene {
        WindowGroup {
            Group {
                ContentView(apiKey: $apiKey, getID: getID)
```

```
            }
            .preferredColorScheme(userColorScheme.colorScheme)
#if os(iOS)
            .fullScreenCover(isPresented: $isShowingAPIConfigModal) {
                EnterAPIKeyView(apiKey: $apiKey)
            }
#elseif os(macOS)
            .popover(isPresented: $isShowingAPIConfigModal, arrowEdge:
.bottom) {
                EnterAPIKeyView(apiKey: $apiKey)
                    .frame(width: 500, height: 500)
            }
#endif
        }
    }
}


//  UserColorScheme.swift

import SwiftUI

enum UserColorScheme: String, Codable {
    case system
    case light
    case dark

    var colorScheme: ColorScheme? {
        return switch self {
```

```
        case .system: .none
        case .light: .light
        case .dark: .dark
        }
    }

    var name: String {
        return switch self {
        case .system: "System".localized
        case .light: "Light".localized
        case .dark: "Dark".localized
        }
    }
}

//  SettingsView.swift

import SwiftUI

struct SettingsView: View {
    @AppStorage(AppStorageKeys.apiKey) private var apiKey = ""
    @AppStorage(AppStorageKeys.userColorScheme) private var
userColorScheme: UserColorScheme = .system

    var body: some View {
        NavigationView {
            List {
                Section {
```

```
                colorSchemeRow
            }

            Section {
                changeApiKeyRow
            }
        }
        .navigationTitle("Settings")
    }
}

@ViewBuilder private var colorSchemeRow: some View {
    Picker("Color Scheme", selection: $userColorScheme) {
        Text(UserColorScheme.system.name)
            .tag(UserColorScheme.system)

        Text(UserColorScheme.light.name)
            .tag(UserColorScheme.light)

        Text(UserColorScheme.dark.name)
            .tag(UserColorScheme.dark)
    }
    .pickerStyle(.menu)
}

@ViewBuilder private var changeApiKeyRow: some View {
    NavigationLink("Change API Key") {
```

```
            EnterAPIKeyView(apiKey: $apiKey)
        }
    }
}

//  ImageManager.swift

import Foundation

final class ImageManager: ObservableObject {
    var openAIClient: OpenAIProtocol

    @Published var images: [ImagesResult.URLResult] = []
    @Published var inProgress = false

    init(openAIClient: OpenAIProtocol) {
        self.openAIClient = openAIClient
    }

    @MainActor func images(query: ImagesQuery) async {
        inProgress = true
        images.removeAll()
        do {
            let response = try await openAIClient.images(query: query)
            images = response.data
        } catch {
            print(error.localizedDescription)
```

```
        }

        inProgress = false
    }
}

//  ChatManager.swift

import Foundation
import Combine

final class ChatManager: ObservableObject {
    var openAIClient: OpenAIProtocol
    let getID: () -> String

    @Published var convos: [Convo] = []
    @Published var convoErrors: [ConvoID: Error] = [:]
    @Published var selectedConvoID: ConvoID?
    @Published var inProgress: Bool = false

    var selectedConvo: Convo? {
        selectedConvoID.flatMap { id in
            convos.first { $0.id == id }
        }
    }

    var selectedConvoPublisher: AnyPublisher<Convo?, Never> {
```

```
        $selectedConvoID.receive(on: RunLoop.main).map { id in
            self.convos.first(where: { $0.id == id })
        }
        .eraseToAnyPublisher()
    }

    init(openAIClient: OpenAIProtocol, getID: @escaping () -> String) {
        self.openAIClient = openAIClient
        self.getID = getID
    }

    func createConvo() {
        let convo = Convo(id: getID(), messages: [])
        convos.append(convo)
    }

    func selectConvo(_ convoID: ConvoID?) {
        selectedConvoID = convoID
    }

    func deleteConvo(_ convoID: ConvoID) {
        convos.removeAll(where: { $0.id == convoID })
    }

    @MainActor func send(message: Message, convoID: ConvoID, model:
OpenAIModel) async {
        guard let index = convos.firstIndex(where: { $0.id == convoID }) else {
return }
```

```
        convos[index].messages.append(message)

        await completeChat(convoID: convoID, model: model)
    }

    @MainActor func completeChat(convoID: ConvoID, model: OpenAIModel)
async {
        guard let convo = convos.first(where: { $0.id == convoID }) else { return
}
        inProgress = true
        convoErrors[convoID] = nil

        do {
            guard let convoIndex = convos.firstIndex(where: { $0.id == convoID })
else { return }

            let chatsStream: AsyncThrowingStream<ChatStreamResult, Error> =
openAIClient.chatsStream(query: ChatQuery(model: model, messages:
convo.messages.map { Chat(role: $0.role, content: $0.content) }, functions:
nil))

            var functionCallName = ""
            var functionCallArguments = ""

            for try await partialChatResult in chatsStream {
                for choice in partialChatResult.choices {
                    let existingMessages = convos[convoIndex].messages
```

```
                if let functionCallDelta = choice.delta.functionCall {
                    if let nameDelta = functionCallDelta.name {
                        functionCallName += nameDelta
                    }
                    if let argumentsDelta = functionCallDelta.arguments {
                        functionCallArguments += argumentsDelta
                    }
                }

                var messageText = choice.delta.content ?? ""

                if let finishReason = choice.finishReason, finishReason ==
"function_call" {
                    messageText += "Function call: name=\(functionCallName)
arguments=\(functionCallArguments)"
                }

                let message = Message(id: partialChatResult.id, role:
choice.delta.role ?? .assistant, content: messageText, createdAt:
Date(timeIntervalSince1970: TimeInterval(partialChatResult.created)))

                if let existingMessageIndex = existingMessages.firstIndex(where: {
$0.id == partialChatResult.id }) {
                    let previousMessage = existingMessages[existingMessageIndex]
                    let combinedMessage = Message(id: message.id, role:
message.role, content: previousMessage.content + message.content, createdAt:
message.createdAt)

                    convos[convoIndex].messages[existingMessageIndex] =
combinedMessage
```

```
                } else {
                    convos[convoIndex].messages.append(message)
                }
            }
        }
    } catch {
        convoErrors[convoID] = error
    }

    inProgress = false
  }
}

//  SpeechManager.swift

import Speech
import AVFoundation
import Combine

enum SpeechManagerError: Error {
  case unableToCreateAudioBufferRecognitionRequest
}

final class SpeechManager: NSObject, ObservableObject {
  private let speechRecognizer = SFSpeechRecognizer(locale:
Locale(identifier: "en-US"))!
  private var recognitionRequest: SFSpeechAudioBufferRecognitionRequest?
  private var recognitionTask: SFSpeechRecognitionTask?
```

```
    private let audioEngine = AVAudioEngine()
    private var inputNode: AVAudioInputNode!

    @Published var isListening = false
    @Published var transcriptionText: String = ""

    override init() {
        super.init()
        inputNode = audioEngine.inputNode
    }

    func requestAuthorization() async throws ->
SFSpeechRecognizerAuthorizationStatus {
        speechRecognizer.delegate = self
        return try await withCheckedThrowingContinuation { continuation in
            SFSpeechRecognizer.requestAuthorization { authStatus in
                continuation.resume(returning: authStatus)
            }
        }
    }

    func startRecording() throws {
        isListening = true

        if let recognitionTask = recognitionTask {
            recognitionTask.cancel()
            self.recognitionTask = nil
        }
```

```
#if os(iOS)
    let audioSession = AVAudioSession.sharedInstance()
    try audioSession.setCategory(.record, mode: .measurement, options:
.duckOthers)
    try audioSession.setActive(true, options: .notifyOthersOnDeactivation)
#endif

    recognitionRequest = SFSpeechAudioBufferRecognitionRequest()

    guard let recognitionRequest = recognitionRequest else { throw
SpeechManagerError.unableToCreateAudioBufferRecognitionRequest }

    recognitionRequest.shouldReportPartialResults = true
    recognitionRequest.requiresOnDeviceRecognition = true

    recognitionTask = speechRecognizer.recognitionTask(with:
recognitionRequest) { [weak self] result, error in
        guard let self else { return }

        if let result = result {
            self.transcriptionText = result.bestTranscription.formattedString
        }

        if error != nil || result?.isFinal == true {
            self.audioEngine.stop()
            inputNode.removeTap(onBus: 0)
```

```
                self.recognitionRequest = nil
                self.recognitionTask = nil

                self.isListening = false
            }
        }

        let recordingFormat = inputNode.outputFormat(forBus: 0)

        inputNode.installTap(onBus: 0, bufferSize: 1024, format:
recordingFormat) { (buffer: AVAudioPCMBuffer, when: AVAudioTime) in
            self.recognitionRequest?.append(buffer)
        }

        audioEngine.prepare()
        try audioEngine.start()

        isListening = true
    }

    func stopRecording() {
        audioEngine.stop()
        recognitionRequest?.endAudio()
        isListening = false
    }
}

extension SpeechManager: SFSpeechRecognizerDelegate {
```

```
    func speechRecognizer(_ speechRecognizer: SFSpeechRecognizer,
availabilityDidChange available: Bool) {
    }
}

//  Image+Extensions.swift

import SwiftUI

extension Image {
    func convertToCGImage(completion: @escaping (CGImage?) -> Void) {
        DispatchQueue.main.async {
            completion(ImageRenderer(content: self).cgImage)
        }
    }
}

//  View+Extensions.swift

import SwiftUI

extension View {
    func eraseToAnyView() -> AnyView { AnyView(self) }

    func errorAlert(error: Binding<Error?>, buttonTitle: String = "OK") -> some
View {
        let localizedAlertError = LocalizedAlertError(error: error.wrappedValue)
```

```
        return alert(isPresented: .constant(localizedAlertError != nil), error:
localizedAlertError) { _ in
            Button(buttonTitle) {
                error.wrappedValue = nil
            }
        } message: { error in
            Text(error.recoverySuggestion ?? "")
        }
    }
}

//  String+Extensions.swift

import Foundation

extension String {
    var localized: String {
        return NSLocalizedString(self, comment: "")
    }
}

//  DateGetterKey.swift

import SwiftUI

private struct DateGetterKey: EnvironmentKey {
    static let defaultValue: () -> Date = Date.init
}
```

```
extension EnvironmentValues {
    public var dateGetterValue: () -> Date {
        get { self[DateGetterKey.self] }
        set { self[DateGetterKey.self] = newValue }
    }
}

extension View {
    public func dateGetterValue(_ dateGetterValue: @escaping () -> Date) -> some View {
        environment(\.dateGetterValue, dateGetterValue)
    }
}

//  IDGetterKey.swift

import SwiftUI

private struct IDGetterKey: EnvironmentKey {
    static var defaultValue: () -> String = {
        UUID().uuidString
    }
}

extension EnvironmentValues {
    public var idGetterValue: () -> String {
        get { self[IDGetterKey.self] }
```

```
        set { self[IDGetterKey.self] = newValue }
    }
}

extension View {
    public func idGetterValue(_ idGetterValue: @escaping () -> String) -> some View {
        environment(\.idGetterValue, idGetterValue)
    }
}

//  Colors.swift

import SwiftUI

extension Color {
    static let commonAccent = Self.teal

    static var inputFillColor: Self {
#if os(iOS)
        return Color(uiColor: UIColor.systemBackground)
#elseif os(macOS)
        return Color(nsColor: NSColor.textBackgroundColor)
#endif
    }

    static var strokeColor: Self {
#if os(iOS)
```

```
        return Color(uiColor: UIColor.systemGray5)
#elseif os(macOS)
        return Color(nsColor: NSColor.lightGray)
#endif
    }

    static var errorTextColor: Self {
#if os(iOS)
        return Color(uiColor: .systemRed)
#elseif os(macOS)
        return Color(.systemRed)
#endif
    }
}

//  CrossPlatformImage.swift

#if os(iOS)
import UIKit
#elseif os(macOS)
import Cocoa
#endif

struct CrossPlatformImage {
#if os(iOS)
    typealias Image = UIImage
#elseif os(macOS)
```

```
    typealias Image = NSImage
#endif
}

//  LocalizedAlertError.swift

import Foundation

struct LocalizedAlertError: LocalizedError {
    let underlyingError: LocalizedError

    var errorDescription: String? {
        underlyingError.errorDescription
    }

    var recoverySuggestion: String? {
        underlyingError.recoverySuggestion
    }

    init?(error: Error?) {
        guard let localizedError = error as? LocalizedError else { return nil }
        underlyingError = localizedError
    }
}

//  ContentView.swift
```

```
import SwiftUI
import SwiftData

struct ContentView: View {
    @Binding var apiKey: String
    @StateObject private var chatManager: ChatManager
    @StateObject private var imageManager: ImageManager

    @Environment(\.idGetterValue) private var getID
    @Environment(\.dateGetterValue) private var getDate

    init(apiKey: Binding<String>, getID: @escaping () -> String) {
        self._apiKey = apiKey
        self._chatManager = StateObject(wrappedValue:
ChatManager(openAIClient: OpenAI(apiToken: apiKey.wrappedValue), getID:
getID))
        self._imageManager = StateObject(wrappedValue:
ImageManager(openAIClient: OpenAI(apiToken: apiKey.wrappedValue)))
    }

    var body: some View {
        TabsView(chatManager: chatManager, imageManager: imageManager)
        .onChange(of: apiKey) { _, newApiKey in
            let client = OpenAI(apiToken: newApiKey)
            chatManager.openAIClient = client
            imageManager.openAIClient = client
        }
    }
```

```
}

//  EnterAPIKeyView.swift

import SwiftUI

struct EnterAPIKeyView: View {
    @Environment(\.dismiss) private var dismiss
    @Binding private var apiKey: String
    @State private var internalAPIKey: String

    private let isMandatory: Bool

    public init(apiKey: Binding<String>, isMandatory: Bool = true) {
        self._apiKey = apiKey
        self._internalAPIKey = State(initialValue: apiKey.wrappedValue)
        self.isMandatory = isMandatory
    }

    var body: some View {
        VStack(alignment: .leading, spacing: 16) {
            VStack(alignment: .leading, spacing: 8) {
                Text("You can find and configure your OpenAI API key at")
                    .font(.caption)

                Link("https://platform.openai.com/account/api-keys", destination:
URL(string: "https://platform.openai.com/account/api-keys")!)
                    .font(.caption)
```

```
}

TextEditor(text: $internalAPIKey)
    .frame(height: 120)
    .font(.caption)
    .padding(8)
    .background(
        RoundedRectangle(cornerRadius: 8)
            .stroke(Color.strokeColor, lineWidth: 1)
    )
    .padding(4)
    .clipShape(RoundedRectangle(cornerRadius: 8))

if isMandatory {
    HStack {
        Spacer()

        Button {
            apiKey = internalAPIKey
            dismiss()
        } label: {
            Text("Done")
                .padding(8)
        }
        .buttonStyle(.borderedProminent)
        .disabled(internalAPIKey.isEmpty)
```

```
                Spacer()
            }
        }
    }
    .padding()
    .navigationTitle("OpenAI API Key")
    .tint(.commonAccent)
    .toolbar {
        ToolbarItem(placement: .primaryAction) {
            if isMandatory {
                EmptyView()
            } else {
                Button("Close") {
                    apiKey = internalAPIKey
                    dismiss()
                }
            }
        }
    }
}

//  TabsView.swift

import SwiftUI

struct TabsView: View {
```

```
@ObservedObject private var chatManager: ChatManager
@ObservedObject private var imageManager: ImageManager
@Environment(\.idGetterValue) private var getID

init(chatManager: ChatManager, imageManager: ImageManager) {
   self.chatManager = chatManager
   self.imageManager = imageManager
}

var body: some View {
   TabView {
      ChatListView(chatManager: chatManager)
         .tabItem {
            Label("Chats", systemImage: "message")
         }

      ImageView(imageManager: imageManager)
         .tabItem {
            Label("Image", systemImage: "photo")
         }

      SettingsView()
         .tabItem {
            Label("Settings", systemImage: "gear")
         }
   }
   .tint(.commonAccent)
```

```
    }
}

//  ChatView.swift

import SwiftUI

struct ChatView: View {
    @ObservedObject private var chatManager: ChatManager
    @Environment(\.dateGetterValue) private var getDate
    @Environment(\.idGetterValue) private var getID

    @StateObject private var speechManager = SpeechManager()

    @State var inputText: String = ""
    @State var inputAction: InputAction = .micIsReadyToRecord
    @State private var selectedChatModel: OpenAIModel = .gpt4_0613
    @State private var error: Error?

    @State private var toShowModelSelectionSheet = false
    @State private var toShowDocumentScanner = false

    private let availableChatModels: [OpenAIModel] = [.gpt3_5Turbo0613,
.gpt4_0613]
    private let visionService = VisionService()

    private let convo: Convo
```

```
init(chatManager: ChatManager, convo: Convo) {
    self.chatManager = chatManager
    self.convo = convo
}

var body: some View {
    NavigationStack {
        ScrollViewReader { scrollViewProxy in
            VStack {
                List {
                    ForEach(convo.messages) { message in
                        MessageView(message: message)
                    }
                    .listRowSeparator(.hidden)
                }
                .listStyle(.plain)
                .animation(.default, value: convo.messages)
                .onChange(of: convo) { _, newValue in
                    if let lastMessage = newValue.messages.last {
                        scrollViewProxy.scrollTo(lastMessage.id, anchor: .bottom)
                    }
                }
                .onChange(of: speechManager.transcriptionText) { _, newValue in
                    inputText = newValue
                }
                .onAppear {
                    Task {
```

```
                    await requestSpeechAuthorization()
                }
            }

            if let error = chatManager.convoErrors[convo.id] {
                Text(error.localizedDescription)
                    .font(.caption)
                    .foregroundColor(.errorTextColor)
                    .padding(.horizontal)
            }

            InputBarView(text: $inputText, action: $inputAction, onSubmit: { sendMessage(scrollViewProxy: scrollViewProxy) }, onMicTapped: { Task { await handleMicAction() } })
        }
        .navigationTitle("Chat")
#if os(iOS)
        .navigationBarTitleDisplayMode(.inline)
#endif
        .toolbar {
            if chatManager.inProgress {
                ToolbarItem {
                    ProgressView()
                }
            }
            ToolbarItem {
                Menu(content: {
```

```
                    Button("Select model", action: {
toShowModelSelectionSheet.toggle() })


                    Button("Scan text from image", action: {
toShowDocumentScanner.toggle() })
                }, label: {
                    Image(systemName: "ellipsis")
                })
            }
        }
        .confirmationDialog(
            "Select model",
            isPresented: $toShowModelSelectionSheet,
            titleVisibility: .visible,
            actions: {
                ForEach(availableChatModels, id: \.self) { model in
                    Button(action: { selectedChatModel = model }, label: {
Text(model) })
                }


                Button("Cancel", role: .cancel) {
toShowModelSelectionSheet.toggle() }
            },
            message: {
                Text("View https://platform.openai.com/docs/models/overview
for details")
                    .font(.caption)
            }
        )
```

```
#if os(iOS)
            .sheet(isPresented: $toShowDocumentScanner, content: {
               ScannerView(
                  cancelAction: { toShowDocumentScanner.toggle() },
                  resultAction: { result in
                     Task {
                        await handleScan(result: result)
                     }
                  }
               )
            })
#endif
            .errorAlert(error: $error)
         }
         .disabled(chatManager.inProgress)
      }
   }


#if os(iOS)
   private func handleScan(result: ScannerView.CameraActionType) async {
      switch result {
      case .success(let scan):
         for pageNumber in 0..<scan.pageCount {
            guard let cgImage = scan.imageOfPage(at: pageNumber).cgImage
else {
               toShowDocumentScanner.toggle()
               let error = LocalizedAlertError(error:
AlertError.cgImageConversionFailed)
```

```
                self.error = error
                return
            }

            do {
                let text = try await visionService.detectText(from: cgImage)
                toShowDocumentScanner.toggle()
                self.inputText = #"Text from the image: """\#(text)""" "#
            } catch {
                toShowDocumentScanner.toggle()
                self.error = LocalizedAlertError(error: error)
            }
        }
    case .failure(let error):
        toShowDocumentScanner.toggle()
        self.error = LocalizedAlertError(error: error)
    }
  }
#endif

  private func sendMessage(scrollViewProxy: ScrollViewProxy) {
    let textToSend = inputText

    Task {
        await chatManager.send(message: Message(id: getID(), role: .user,
content: textToSend, createdAt: getDate()), convoID: convo.id, model:
selectedChatModel)
    }
```

```
#if os(iOS)

UIApplication.shared.sendAction(#selector(UIResponder.resignFirstResponder
), to: nil, from: nil, for: nil)
#endif


        inputText = ""


        if let lastMessage = convo.messages.last {
            scrollViewProxy.scrollTo(lastMessage.id, anchor: .bottom)
        }
    }
}

// MARK: - Mic handle
private extension ChatView {
    @MainActor func handleMicAction() async {
        switch inputAction {
        case .micIsReadyToRecord:
            withAnimation {
                inputAction = .micIsRecording
            }
            startSpeechRecording()

        case .micIsRecording:
            withAnimation {
                inputAction = .readyToSend
```

```
        }
        stopSpeechRecording()

    default: break
    }
}

@MainActor func requestSpeechAuthorization() async {
    do {
        let result = try await speechManager.requestAuthorization()

        switch result {
        case .authorized: Preferences.speechRecognizerAuthorized = true
        default: error = AlertError.unableToRecordVoice
        }
    } catch(let error) {
        self.error = LocalizedAlertError(error: error)
    }
}

func startSpeechRecording() {
    do {
        try speechManager.startRecording()
    } catch(let error) {
        self.error = LocalizedAlertError(error: error)
    }
}
```

```
    func stopSpeechRecording() {
        speechManager.stopRecording()
    }
}

// MARK: - Error
private extension ChatView {
    enum AlertError: LocalizedError {
        case cgImageConversionFailed
        case unableToRecordVoice
    }
}

//  InputBarView.swift

import SwiftUI

struct InputBarView: View {
    @Binding var text: String
    @Binding var action: InputAction
    var onSubmit: () -> Void
    var onMicTapped: () -> Void

    @State private var micRecordingEffect = false

    var body: some View {
```

```
HStack {
    TextEditor(text: $text)
        .padding(.vertical, -8)
        .padding(.horizontal, -4)
        .frame(minHeight: 22, maxHeight: 300)
        .foregroundColor(.primary)
        .padding(EdgeInsets(top: 12, leading: 16, bottom: 12, trailing: 16))
        .background(
            RoundedRectangle(cornerRadius: 16, style: .continuous)
                .fill(.background)
                .overlay(
                    RoundedRectangle(cornerRadius: 16, style: .continuous)
                        .stroke(.primary, lineWidth: 0.3)
                )
        )
        .fixedSize(horizontal: false, vertical: true)
        .onChange(
            of: text,
            { oldValue, newValue in
                withAnimation(.interactiveSpring) {
                    if action != .micIsRecording {
                        if !newValue.isEmpty {
                            action = .readyToSend
                        } else {
                            action = .micIsReadyToRecord
                        }
                    }
```

```
            }
        }
    )
    .onSubmit {
        withAnimation {
            onSubmit()
        }
    }
    .padding(.leading)

Button {
    if action == .micIsRecording || action == .micIsReadyToRecord {
        onMicTapped()
    } else if action == .readyToSend {
        onSubmit()
    }
} label: {
    Image(systemName: action.icon)
        .resizable()
        .aspectRatio(contentMode: .fit)
        .frame(width: 24, height: 24)
}
.tint(.commonAccent)
.background(
    action == .micIsRecording ?
    Circle()
        .frame(width: 40, height: 40)
```

```
                .foregroundColor(.commonAccent.opacity(0.5))
                .scaleEffect(micRecordingEffect ? 1 : 0.8)
                .onAppear() {
                    withAnimation(.linear.repeatForever(autoreverses:
true).speed(1)) {
                        micRecordingEffect.toggle()
                    }
                } : nil
            )
            .padding(.trailing)
        }
        .padding(.bottom)
    }
}

//  MessageView.swift

import SwiftUI

struct MessageView: View {
    private let message: Message

    init(message: Message) {
        self.message = message
    }

    var body: some View {
        VStack(alignment: .leading) {
```

```
HStack {
    Image(
        systemName: {
            return switch message.role {
            case .user: "person"
            default: "desktopcomputer"
            }
        }()
    )
    .resizable()
    .aspectRatio(contentMode: .fit)
    .frame(width: 10, height: 10)

    Text("\(message.role == .user ? "YOU" : "CHATPGT")")
    .font(.caption)
    .foregroundStyle(.secondary)

    Spacer()
}

switch message.role {
case .function:
    Text(message.content)
        .font(.footnote.monospaced())
        .foregroundStyle(.primary)
default:
    Text(message.content)
```

```
                .foregroundStyle(.primary)
            }
        }
    }
}

//  ChatListView.swift

import SwiftUI

struct ChatListView: View {
    @ObservedObject private var chatManager: ChatManager

    @Environment(\.dateGetterValue) private var getDate
    @Environment(\.idGetterValue) private var getID

    init(chatManager: ChatManager) {
        self.chatManager = chatManager
    }

    var body: some View {
        NavigationSplitView {
            ListView(
                convos: $chatManager.convos,
                selectedConvoID: Binding<ConvoID?>(
                    get: {
                        chatManager.selectedConvoID
```

```
                }, set: { newID in
                    chatManager.selectConvo(newID)
                })
            )
#if os(macOS)
            .navigationSplitViewColumnWidth(min: 180, ideal: 200)
#endif
            .toolbar {
                ToolbarItem(placement: .primaryAction) {
                    Button {
                        chatManager.createConvo()
                    } label: {
                        Image(systemName: "plus")
                    }
                }
            }
        } detail: {
            if let convo = chatManager.selectedConvo {
                ChatView(chatManager: chatManager, convo: convo)
            }
        }
    }
}

fileprivate struct ListView: View {
    @Binding var convos: [Convo]
    @Binding var selectedConvoID: ConvoID?
```

```
    var body: some View {
        List($convos, editActions: [.delete], selection: $selectedConvoID) {
$convo in
            Text(convo.messages.last?.content ?? "New Conversation".localized)
            .lineLimit(2)
        }
        .navigationTitle("Conversations")
    }
}

//  ScannerView.swift

import SwiftUI
import VisionKit

#if os(iOS)
struct ScannerView: UIViewControllerRepresentable {
    typealias CameraActionType = Result<VNDocumentCameraScan, Error>
    typealias CancelActionType = () -> Void
    typealias ResultActionType = (CameraActionType) -> Void

    private let cancelAction: CancelActionType
    private let resultAction: ResultActionType

    init(cancelAction: @escaping CancelActionType = {}, resultAction:
@escaping ResultActionType) {
        self.cancelAction = cancelAction
```

```
        self.resultAction = resultAction
    }

    func makeCoordinator() -> Coordinator {
        return Coordinator(cancelAction: cancelAction, resultAction:
resultAction)
    }

    func makeUIViewController(context: Context) ->
VNDocumentCameraViewController {
        let controller = VNDocumentCameraViewController()
        controller.delegate = context.coordinator
        return controller
    }

    func updateUIViewController(_ uiViewController:
VNDocumentCameraViewController, context: Context) {}
}

// MARK: - Coordinator
extension ScannerView {
    class Coordinator: NSObject, VNDocumentCameraViewControllerDelegate
{
        init(cancelAction: @escaping CancelActionType, resultAction:
@escaping ResultActionType) {
            self.cancelAction = cancelAction
            self.resultAction = resultAction
        }
```

```
        private let cancelAction: CancelActionType
        private let resultAction: ResultActionType


        func documentCameraViewControllerDidCancel(_ controller:
VNDocumentCameraViewController) {
            cancelAction()
        }


        func documentCameraViewController(_ controller:
VNDocumentCameraViewController, didFailWithError error: Error) {
            resultAction(.failure(error))
        }


        func documentCameraViewController(_ controller:
VNDocumentCameraViewController, didFinishWith scan:
VNDocumentCameraScan) {
            resultAction(.success(scan))
        }
    }
}
#endif
//  InputAction.swift


enum InputAction {
    case micIsRecording
    case micIsReadyToRecord
    case blocked
    case readyToSend
```

```
    var icon: String {
        return switch self {
        case .micIsRecording: "mic"
        case .micIsReadyToRecord: "mic"
        case .blocked: "square.slash"
        case .readyToSend: "paperplane"
        }
    }
}

//  ImageView.swift

import SwiftUI

struct ImageView: View {
    @ObservedObject var imageManager: ImageManager

    public init(imageManager: ImageManager) {
        self.imageManager = imageManager
    }

    var body: some View {
        NavigationStack {
            List {
                NavigationLink("Generate Image", destination:
GenerateImageView(imageManager: imageManager))
            }
            .navigationTitle("Image")
```

```
        }
    }
}

//  GenerateImageView.swift

import SwiftUI
#if os(iOS)
import UIKit
#endif

struct GenerateImageView: View {
    @ObservedObject private var imageManager: ImageManager

    @State private var prompt: String = ""
    @State private var n: Int = 1
    @State private var size: String

    private let sizes = ["256x256", "512x512", "1024x1024"]

    init(imageManager: ImageManager) {
        self.imageManager = imageManager
        size = sizes[0]
    }

    var body: some View {
        List {
```

```
        Section {
            HStack {
                Text("Prompt")
                Spacer()
                TextEditor(text: $prompt)
                    .multilineTextAlignment(.trailing)
            }

            HStack {
                Stepper("Amount: \(n)", value: $n, in: 1...10)
            }

            HStack {
                Picker("Size", selection: $size) {
                    ForEach(sizes, id: \.self) {
                        Text($0)
                    }
                }
            }
        }

        Section {
            HStack {
                Button(n == 1 ? "Create Image".localized : "Create
Images".localized) {
                    Task {
                        let query = ImagesQuery(prompt: prompt, n: n, size: size)
                        await imageManager.images(query: query)
```

```
                }
            }
            .foregroundColor(.accentColor)

            Spacer()
        }
    }

    ZStack {
        Section {
            VStack {
                HStack {
                    Text("Images: \($imageManager.images.count)")

                    Spacer()

                    if imageManager.inProgress {
                        ProgressView()
                    }
                }
                if !$imageManager.images.isEmpty {
                    ForEach($imageManager.images, id: \.self) { image in
                        let urlString = image.wrappedValue.url
                        if let imageURL = URL(string: urlString) {
#if os(iOS)
                            if UIApplication.shared.canOpenURL(imageURL) {
                                LinkPreview(previewURL: imageURL)
```

```
                            .aspectRatio(contentMode: .fit)
                        }
#elseif os(macOS)
                        if NSWorkspace.shared.open(imageURL) {
                            LinkPreview(previewURL: imageURL)
                                .aspectRatio(contentMode: .fit)
                        }
#endif
                    } else {
                        Text(urlString)
                            .foregroundStyle(.secondary)
                    }
                }
            }
        }
    }
    .scrollDismissesKeyboard(.interactively)
    #if os(iOS)
    .navigationBarTitleDisplayMode(.inline)
    #endif
    .navigationTitle("Generate Image")
  }
}

//  LinkPreview.swift
```

```
import SwiftUI
import LinkPresentation

#if os(iOS)
struct LinkPreview: UIViewRepresentable {
    typealias UIViewType = LPLinkView

    var previewURL: URL

    func makeUIView(context: Context) -> LPLinkView {
        LPLinkView(url: previewURL)
    }

    func updateUIView(_ uiView: UIViewType, context: Context) {
        LPMetadataProvider().startFetchingMetadata(for: previewURL) {
metadata, error in
            guard error == nil else { return }
            guard let metadata else { return }

            uiView.metadata = metadata
        }
    }
}
#elseif os(macOS)
struct LinkPreview: NSViewRepresentable {
    typealias NSViewType = LPLinkView
```

```
    var previewURL: URL

    func makeNSView(context: Context) -> LPLinkView {
        LPLinkView(url: previewURL)
    }

    func updateNSView(_ nsView: NSViewType, context: Context) {
        LPMetadataProvider().startFetchingMetadata(for: previewURL) {
metadata, error in
            guard error == nil else { return }
            guard let metadata else { return }

            nsView.metadata = metadata
        }
    }
}
#endif

//  APIError.swift

import Foundation

enum OpenAIError: Error {
    case emptyData
}

struct APIError: Error, Decodable, Equatable {
    let message: String
```

```
    let type: String
    let param: String?
    let code: String?

    init(message: String, type: String, param: String?, code: String?) {
        self.message = message
        self.type = type
        self.param = param
        self.code = code
    }

    enum CodingKeys: CodingKey {
        case message
        case type
        case param
        case code
    }

    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)

        if let string = try? container.decode(String.self, forKey: .message) {
            self.message = string
        } else if let array = try? container.decode([String].self, forKey: .message)
{
            self.message = array.joined(separator: "\n")
        } else {
```

```
            throw DecodingError.typeMismatch(String.self, .init(codingPath:
[CodingKeys.message], debugDescription: "message: expected String or
[String]"))
        }

        self.type = try container.decode(String.self, forKey: .type)
        self.param = try container.decodeIfPresent(String.self, forKey: .param)
        self.code = try container.decodeIfPresent(String.self, forKey: .code)
    }
}

extension APIError: LocalizedError {
    var errorDescription: String? {
        return message
    }
}

struct APIErrorResponse: Error, Decodable, Equatable {
    let error: APIError
}

extension APIErrorResponse: LocalizedError {
    var errorDescription: String? {
        return error.errorDescription
    }
}

//  OpenAIModel.swift
```

```
import Foundation

typealias OpenAIModel = String

extension OpenAIModel {
    static let gpt4_0613 = "gpt-4-0613"
    static let gpt3_5Turbo0613 = "gpt-3.5-turbo-0613"
}

//  ChatQuery.swift

import Foundation

struct Chat: Codable, Equatable {
    let role: Role
    let content: String?
    let name: String?
    let functionCall: ChatFunctionCall?

    enum Role: String, Codable, Equatable {
        case system
        case assistant
        case user
        case function
    }
```

```
    enum CodingKeys: String, CodingKey {
        case role
        case content
        case name
        case functionCall = "function_call"
    }

    init(role: Role, content: String? = nil, name: String? = nil, functionCall:
ChatFunctionCall? = nil) {
        self.role = role
        self.content = content
        self.name = name
        self.functionCall = functionCall
    }

    func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: CodingKeys.self)
        try container.encode(role, forKey: .role)

        if let name = name {
            try container.encode(name, forKey: .name)
        }

        if let functionCall = functionCall {
            try container.encode(functionCall, forKey: .functionCall)
        }

        if content != nil || (role == .assistant && functionCall != nil) {
```

```
            try container.encode(content, forKey: .content)
        }
    }
}

struct ChatFunctionCall: Codable, Equatable {
    let name: String?
    let arguments: String?

    init(name: String?, arguments: String?) {
        self.name = name
        self.arguments = arguments
    }
}

struct JSONSchema: Codable, Equatable {
    let type: JSONType
    let properties: [String: Property]?
    let required: [String]?
    let pattern: String?
    let const: String?
    let enumValues: [String]?
    let multipleOf: Int?
    let minimum: Int?
    let maximum: Int?

    private enum CodingKeys: String, CodingKey {
```

```
    case type, properties, required, pattern, const
    case enumValues = "enum"
    case multipleOf, minimum, maximum
}

struct Property: Codable, Equatable {
    let type: JSONType
    let description: String?
    let format: String?
    let items: Items?
    let required: [String]?
    let pattern: String?
    let const: String?
    let enumValues: [String]?
    let multipleOf: Int?
    let minimum: Double?
    let maximum: Double?
    let minItems: Int?
    let maxItems: Int?
    let uniqueItems: Bool?

    private enum CodingKeys: String, CodingKey {
        case type, description, format, items, required, pattern, const
        case enumValues = "enum"
        case multipleOf, minimum, maximum
        case minItems, maxItems, uniqueItems
    }
```

```
        init(type: JSONType, description: String? = nil, format: String? = nil,
items: Items? = nil, required: [String]? = nil, pattern: String? = nil, const:
String? = nil, enumValues: [String]? = nil, multipleOf: Int? = nil, minimum:
Double? = nil, maximum: Double? = nil, minItems: Int? = nil, maxItems: Int? =
nil, uniqueItems: Bool? = nil) {
            self.type = type
            self.description = description
            self.format = format
            self.items = items
            self.required = required
            self.pattern = pattern
            self.const = const
            self.enumValues = enumValues
            self.multipleOf = multipleOf
            self.minimum = minimum
            self.maximum = maximum
            self.minItems = minItems
            self.maxItems = maxItems
            self.uniqueItems = uniqueItems
        }
    }

    enum JSONType: String, Codable {
        case integer = "integer"
        case string = "string"
        case boolean = "boolean"
        case array = "array"
        case object = "object"
```

```
        case number = "number"
        case `null` = "null"
    }

    struct Items: Codable, Equatable {
        let type: JSONType
        let properties: [String: Property]?
        let pattern: String?
        let const: String?
        let enumValues: [String]?
        let multipleOf: Int?
        let minimum: Double?
        let maximum: Double?
        let minItems: Int?
        let maxItems: Int?
        let uniqueItems: Bool?

        private enum CodingKeys: String, CodingKey {
            case type, properties, pattern, const
            case enumValues = "enum"
            case multipleOf, minimum, maximum, minItems, maxItems,
uniqueItems
        }

        init(type: JSONType, properties: [String : Property]? = nil, pattern: String?
= nil, const: String? = nil, enumValues: [String]? = nil, multipleOf: Int? = nil,
minimum: Double? = nil, maximum: Double? = nil, minItems: Int? = nil,
maxItems: Int? = nil, uniqueItems: Bool? = nil) {
            self.type = type
```

```
            self.properties = properties
            self.pattern = pattern
            self.const = const
            self.enumValues = enumValues
            self.multipleOf = multipleOf
            self.minimum = minimum
            self.maximum = maximum
            self.minItems = minItems
            self.maxItems = maxItems
            self.uniqueItems = uniqueItems
        }
    }

    init(type: JSONType, properties: [String : Property]? = nil, required:
[String]? = nil, pattern: String? = nil, const: String? = nil, enumValues:
[String]? = nil, multipleOf: Int? = nil, minimum: Int? = nil, maximum: Int? =
nil) {
        self.type = type
        self.properties = properties
        self.required = required
        self.pattern = pattern
        self.const = const
        self.enumValues = enumValues
        self.multipleOf = multipleOf
        self.minimum = minimum
        self.maximum = maximum
    }
}
```

```
struct ChatFunctionDeclaration: Codable, Equatable {
    let name: String
    let description: String

    let parameters: JSONSchema

    init(name: String, description: String, parameters: JSONSchema) {
        self.name = name
        self.description = description
        self.parameters = parameters
    }
}

struct ChatQueryFunctionCall: Codable, Equatable {
    let name: String?
    let arguments: String?
}

struct ChatQuery: Equatable, Codable, Streamable {
    let model: OpenAIModel
    let messages: [Chat]
    let functions: [ChatFunctionDeclaration]?
    let functionCall: FunctionCall?
    let temperature: Double?
    let topP: Double?
    let n: Int?
```

```
let stop: [String]?
let maxTokens: Int?
let presencePenalty: Double?
let frequencyPenalty: Double?
let logitBias: [String:Int]?
let user: String?

var stream: Bool = false

enum FunctionCall: Codable, Equatable {
    case none
    case auto
    case function(String)

    enum CodingKeys: String, CodingKey {
        case none = "none"
        case auto = "auto"
        case function = "name"
    }

    func encode(to encoder: Encoder) throws {
        switch self {
        case .none:
            var container = encoder.singleValueContainer()
            try container.encode(CodingKeys.none.rawValue)
        case .auto:
            var container = encoder.singleValueContainer()
```

```
            try container.encode(CodingKeys.auto.rawValue)
        case .function(let name):
            var container = encoder.container(keyedBy: CodingKeys.self)
            try container.encode(name, forKey: .function)
        }
    }
}

enum CodingKeys: String, CodingKey {
    case model
    case messages
    case functions
    case functionCall = "function_call"
    case temperature
    case topP = "top_p"
    case n
    case stream
    case stop
    case maxTokens = "max_tokens"
    case presencePenalty = "presence_penalty"
    case frequencyPenalty = "frequency_penalty"
    case logitBias = "logit_bias"
    case user
}

init(model: OpenAIModel, messages: [Chat], functions:
[ChatFunctionDeclaration]? = nil, functionCall: FunctionCall? = nil,
temperature: Double? = nil, topP: Double? = nil, n: Int? = nil, stop: [String]? =
```

```
nil, maxTokens: Int? = nil, presencePenalty: Double? = nil, frequencyPenalty:
Double? = nil, logitBias: [String : Int]? = nil, user: String? = nil, stream: Bool =
false) {
        self.model = model
        self.messages = messages
        self.functions = functions
        self.functionCall = functionCall
        self.temperature = temperature
        self.topP = topP
        self.n = n
        self.stop = stop
        self.maxTokens = maxTokens
        self.presencePenalty = presencePenalty
        self.frequencyPenalty = frequencyPenalty
        self.logitBias = logitBias
        self.user = user
        self.stream = stream
    }
}

//  ChatResult.swift

import Foundation

struct ChatResult: Codable, Equatable {
    struct Choice: Codable, Equatable {
        let index: Int
        let message: Chat
```

```
    let finishReason: String?

    enum CodingKeys: String, CodingKey {
        case index
        case message
        case finishReason = "finish_reason"
    }
}

struct Usage: Codable, Equatable {
    let promptTokens: Int
    let completionTokens: Int
    let totalTokens: Int

    enum CodingKeys: String, CodingKey {
        case promptTokens = "prompt_tokens"
        case completionTokens = "completion_tokens"
        case totalTokens = "total_tokens"
    }
}

let id: String
let object: String
let created: TimeInterval
let model: OpenAIModel
let choices: [Choice]
let usage: Usage?
```

```
    enum CodingKeys: String, CodingKey {
        case id
        case object
        case created
        case model
        case choices
        case usage
    }

    init(id: String, object: String, created: TimeInterval, model: OpenAIModel, choices: [Choice], usage: Usage) {
        self.id = id
        self.object = object
        self.created = created
        self.model = model
        self.choices = choices
        self.usage = usage
    }
}
//  ChatStreamResult.swift

import Foundation

struct ChatStreamResult: Codable, Equatable {
    struct Choice: Codable, Equatable {
        struct Delta: Codable, Equatable {
            let content: String?
```

```
        let role: Chat.Role?
        let name: String?
        let functionCall: ChatFunctionCall?

        enum CodingKeys: String, CodingKey {
            case role
            case content
            case name
            case functionCall = "function_call"
        }
    }

    let index: Int
    let delta: Delta
    let finishReason: String?

    enum CodingKeys: String, CodingKey {
        case index
        case delta
        case finishReason = "finish_reason"
    }
}

let id: String
let object: String
let created: TimeInterval
let model: OpenAIModel
```

```
    let choices: [Choice]

    enum CodingKeys: String, CodingKey {
        case id
        case object
        case created
        case model
        case choices
    }

    init(id: String, object: String, created: TimeInterval, model: OpenAIModel,
choices: [Choice]) {
        self.id = id
        self.object = object
        self.created = created
        self.model = model
        self.choices = choices
    }
}

//  ImagesResult.swift

import Foundation

struct ImagesResult: Codable, Equatable {
    struct URLResult: Codable, Equatable {
        let url: String
    }
```

```
    let created: TimeInterval
    let data: [URLResult]
}

extension ImagesResult.URLResult: Hashable { }

//  ImagesQuery.swift

import Foundation

struct ImagesQuery: Codable {
    let prompt: String
    let n: Int?
    let size: String?

    init(prompt: String, n: Int?, size: String?) {
        self.prompt = prompt
        self.n = n
        self.size = size
    }
}

//  OpenAI.swift

import Foundation
```

```
final class OpenAI: OpenAIProtocol {
    struct Configuration {
        let token: String
        let organizationIdentifier: String?
        let host: String
        let timeoutInterval: TimeInterval
        init(token: String, organizationIdentifier: String? = nil, host: String = "api.openai.com", timeoutInterval: TimeInterval = 60.0) {
            self.token = token
            self.organizationIdentifier = organizationIdentifier
            self.host = host
            self.timeoutInterval = timeoutInterval
        }
    }

    private let session: URLSession
    private var streamingSessions: [NSObject] = []

    let configuration: Configuration

    convenience init(apiToken: String) {
        self.init(configuration: Configuration(token: apiToken), session: URLSession.shared)
    }

    convenience init(configuration: Configuration) {
        self.init(configuration: configuration, session: URLSession.shared)
    }
```

```
    init(configuration: Configuration, session: URLSession) {
        self.configuration = configuration
        self.session = session
    }

    func images(query: ImagesQuery, completion: @escaping
(Result<ImagesResult, Error>) -> Void) {
        performRequest(request: JSONRequest<ImagesResult>(body: query, url:
buildURL(path: .images)), completion: completion)
    }

    func chatsStream(query: ChatQuery, onResult: @escaping
(Result<ChatStreamResult, Error>) -> Void, completion: ((Error?) -> Void)?) {
        performSteamingRequest(request: JSONRequest<ChatResult>(body:
query.makeStreamable(), url: buildURL(path: .chats)), onResult: onResult,
completion: completion)
    }

    public func chats(query: ChatQuery, completion: @escaping
(Result<ChatResult, Error>) -> Void) {
        performRequest(request: JSONRequest<ChatResult>(body: query, url:
buildURL(path: .chats)), completion: completion)
    }
}

extension OpenAI {
    func performRequest<ResultType: Codable>(request: any
URLRequestBuildable, completion: @escaping (Result<ResultType, Error>) ->
Void) {
```

```
        do {
            let request = try request.build(token: configuration.token,
organizationIdentifier: configuration.organizationIdentifier, timeoutInterval:
configuration.timeoutInterval)
            let task = session.dataTask(with: request) { data, _, error in
                if let error {
                    completion(.failure(error))
                    return
                }
                guard let data else {
                    completion(.failure(OpenAIError.emptyData))
                    return
                }

                var apiError: Error? = nil
                do {
                    let decoded = try JSONDecoder().decode(ResultType.self, from:
data)
                    completion(.success(decoded))
                } catch {
                    apiError = error
                }

                if let apiError {
                    do {
                        let decoded = try
JSONDecoder().decode(APIErrorResponse.self, from: data)
                        completion(.failure(decoded))
                    } catch {
```

```
                completion(.failure(apiError))
            }
        }
    }
    task.resume()
} catch {
    completion(.failure(error))
}
}

func performSteamingRequest<ResultType: Codable>(request: any URLRequestBuildable, onResult: @escaping (Result<ResultType, Error>) -> Void, completion: ((Error?) -> Void)?) {
    do {
        let request = try request.build(token: configuration.token, organizationIdentifier: configuration.organizationIdentifier, timeoutInterval: configuration.timeoutInterval)
        let session = StreamingSession<ResultType>(urlRequest: request)
        session.onReceiveContent = { _, object in
            onResult(.success(object))
        }
        session.onProcessingError = { _, error in
            onResult(.failure(error))
        }
        session.onComplete = { [weak self] object, error in
            self?.streamingSessions.removeAll(where: { $0 == object })
            completion?(error)
        }
        session.perform()
```

```
            streamingSessions.append(session)
        } catch {
            completion?(error)
        }
    }
}

extension OpenAI {
    func buildURL(path: String) -> URL {
        var components = URLComponents()
        components.scheme = "https"
        components.host = configuration.host
        components.path = path
        return components.url!
    }
}

typealias APIPath = String

extension APIPath {
    static let completions = "/v1/completions"
    static let images = "/v1/images/generations"
    static let embeddings = "/v1/embeddings"
    static let chats = "/v1/chat/completions"
    static let edits = "/v1/edits"
    static let models = "/v1/models"
    static let moderations = "/v1/moderations"
```

```
    static let audioTranscriptions = "/v1/audio/transcriptions"
    static let audioTranslations = "/v1/audio/translations"

    func withPath(_ path: String) -> String {
        self + "/" + path
    }
}

//  OpenAIProtocol.swift

import Foundation

protocol OpenAIProtocol {
    func images(query: ImagesQuery, completion: @escaping
(Result<ImagesResult, Error>) -> Void)
    func chatsStream(query: ChatQuery, onResult: @escaping
(Result<ChatStreamResult, Error>) -> Void, completion: ((Error?) -> Void)?)
}

extension OpenAIProtocol {
    func images(query: ImagesQuery) async throws -> ImagesResult {
        try await withCheckedThrowingContinuation { continuation in
            images(query: query) { result in
                switch result {
                case let .success(success):
                    return continuation.resume(returning: success)
                case let .failure(failure):
```

```
                return continuation.resume(throwing: failure)
            }
        }
    }
}

func chatsStream(query: ChatQuery) -> AsyncThrowingStream<ChatStreamResult, Error> {
    return AsyncThrowingStream { continuation in
        return chatsStream(query: query)  { result in
            continuation.yield(with: result)
        } completion: { error in
            continuation.finish(throwing: error)
        }
    }
}
}

//  JSONRequest.swift

import Foundation

final class JSONRequest<ResultType> {
    let body: Codable?
    let url: URL
    let method: String

    init(body: Codable? = nil, url: URL, method: String = "POST") {
```

```
        self.body = body
        self.url = url
        self.method = method
    }
}

extension JSONRequest: URLRequestBuildable {
    func build(token: String, organizationIdentifier: String?, timeoutInterval: TimeInterval) throws -> URLRequest {
        var request = URLRequest(url: url, timeoutInterval: timeoutInterval)
        request.setValue("application/json", forHTTPHeaderField: "Content-Type")
        request.setValue("Bearer \(token)", forHTTPHeaderField: "Authorization")
        if let organizationIdentifier {
            request.setValue(organizationIdentifier, forHTTPHeaderField: "OpenAI-Organization")
        }
        request.httpMethod = method
        if let body {
            request.httpBody = try JSONEncoder().encode(body)
        }
        return request
    }
}

//  URLRequestBuildable.swift

import Foundation
```

```
protocol URLRequestBuildable {
    associatedtype ResultType

    func build(token: String, organizationIdentifier: String?, timeoutInterval:
TimeInterval) throws -> URLRequest
}

//  StreamingSession.swift

import Foundation

final class StreamingSession<ResultType: Codable>: NSObject, Identifiable,
URLSessionDelegate, URLSessionDataDelegate {

    enum StreamingError: Error {
        case unknownContent
        case emptyContent
    }

    var onReceiveContent: ((StreamingSession, ResultType) -> Void)?
    var onProcessingError: ((StreamingSession, Error) -> Void)?
    var onComplete: ((StreamingSession, Error?) -> Void)?

    private let streamingCompletionMarker = "[DONE]"
    private let urlRequest: URLRequest
    private lazy var urlSession: URLSession = {
```

```
        let session = URLSession(configuration: .default, delegate: self,
delegateQueue: nil)
        return session
    }()

    init(urlRequest: URLRequest) {
        self.urlRequest = urlRequest
    }

    func perform() {
        self.urlSession
            .dataTask(with: self.urlRequest)
            .resume()
    }

    func urlSession(_ session: URLSession, task: URLSessionTask,
didCompleteWithError error: Error?) {
        onComplete?(self, error)
    }

    func urlSession(_ session: URLSession, dataTask: URLSessionDataTask,
didReceive data: Data) {
        guard let stringContent = String(data: data, encoding: .utf8) else {
            onProcessingError?(self, StreamingError.unknownContent)
            return
        }
```

```
        let jsonObjects = stringContent.components(separatedBy: "data:").filter {
$0.isEmpty == false }.map { $0.trimmingCharacters(in:
.whitespacesAndNewlines) }


        guard jsonObjects.isEmpty == false, jsonObjects.first !=
streamingCompletionMarker else { return }


        jsonObjects.forEach { jsonContent  in
            guard jsonContent != streamingCompletionMarker else { return }


            guard let jsonData = jsonContent.data(using: .utf8) else {
                onProcessingError?(self, StreamingError.unknownContent)
                return
            }


            var apiError: Error? = nil


            do {
                let decoder = JSONDecoder()
                let object = try decoder.decode(ResultType.self, from: jsonData)
                onReceiveContent?(self, object)
            } catch {
                apiError = error
            }


            if let apiError {
                do {
                    let decoded = try JSONDecoder().decode(APIErrorResponse.self,
from: jsonData)
```

```
                onProcessingError?(self, decoded)
            } catch {
                onProcessingError?(self, apiError)
            }
        }
    }
}

//  StreamableQuery.swift

import Foundation

protocol Streamable {
    var stream: Bool { get set }
    func makeStreamable() -> Self
}

extension Streamable {
    func makeStreamable() -> Self {
        var copy = self
        copy.stream = true
        return copy
    }
}

//  UserDefaultsWrapper.swift
```

```
import Foundation

private protocol AnyOptional {
   var isNil: Bool { get }
}

extension Optional: AnyOptional {
   var isNil: Bool { self == nil }
}

@propertyWrapper struct UserDefaultsWrapper<Value> {
   var wrappedValue: Value {
        get {
           let value = self.storage.getAny(for: key) as? Value
           return value ?? self.defaultValue
        }
      set {
         if let optional = newValue as? AnyOptional, optional.isNil {
            self.storage.remove(for: key)
         } else {
            self.storage.save(newValue, for: key)
         }
      }
    }

    private let key: String
```

```
    private let defaultValue: Value
    private let storage: Storage

    init(wrappedValue defaultValue: Value,
         key: String,
         storage: Storage = UserDefaults.standard) {
        self.defaultValue = defaultValue
        self.key = key
        self.storage = storage
    }
}

extension UserDefaultsWrapper where Value: ExpressibleByNilLiteral {
    init(key: String, storage: UserDefaults = .standard) {
        self.init(wrappedValue: nil, key: key, storage: storage)
    }
}

//  UserDefaults+Storage.swift

import Foundation

extension UserDefaults: Storage {
    public func save(_ object: NSObject?, for key: String) {
        self.setValue(object, forKey: key)
    }
```

```
    public func getObject(for key: String) -> NSObject? {
        return self.object(forKey: key) as? NSObject
    }

    public func save<T>(_ object: T, for key: String) where T: Codable {
        let data = object.toJson()
        self.set(data, forKey: key)
    }

    public func get<T>(_ objectType: T.Type, for key: String) -> T? where T:
Codable {
        guard let data = self.object(forKey: key) as? Data else { return nil }

        return try? T.from(jsonData: data)
    }

    public func save(_ value: Any, for key: String) {
        self.setValue(value, forKey: key)
    }

    public func getAny(for key: String) -> Any? {
        return self.value(forKey: key)
    }

    public func save(_ string: String?, for key: String) {
        self.set(string, forKey: key)
    }
```

```
public func getString(for key: String) -> String? {
    return self.object(forKey: key) as? String
}

public func save(_ int: Int?, for key: String) {
    self.set(int, forKey: key)
}

public func getInt(for key: String) -> Int? {
    return self.object(forKey: key) as? Int
}

public func save(_ bool: Bool?, for key: String) {
    self.set(bool, forKey: key)
}

public func save(_ double: Double?, for key: String) {
    self.set(double, forKey: key)
}

public func getDouble(for key: String) -> Double? {
    self.double(forKey: key)
}

public func getBool(for key: String) -> Bool? {
    return self.object(forKey: key) as? Bool
}
```

```
    public func remove(for key: String) {
        self.removeObject(forKey: key)
    }
}

//  Storage.swift

import Foundation

public protocol Storage {
    func save(_ string: String?, for key: String)
    func getString(for key: String) -> String?

    func save(_ int: Int?, for key: String)
    func getInt(for key: String) -> Int?

    func save(_ bool: Bool?, for key: String)
    func getBool(for key: String) -> Bool?

    func save(_ object: NSObject?, for key: String)
    func getObject(for key: String) -> NSObject?

    func save(_ double: Double?, for key: String)
    func getDouble(for key: String) -> Double?

    func save<T: Codable>(_ object: T, for key: String)
```

```
    func get<T: Codable>(_ objectType: T.Type, for key: String) -> T?

    func save(_ value: Any, for key: String)
    func getAny(for key: String) -> Any?

    func remove(for key: String)
}

public extension Storage {
    func bool(for key: String) -> Bool {
        return (self.getBool(for: key) as Bool?) ?? false
    }

    func int(for key: String) -> Int {
        return (self.getInt(for: key) as Int?) ?? 0
    }

    func string(for key: String) -> String {
        return (self.getString(for: key) as String?) ?? ""
    }
}

//  Codable+Extensions.swift

import Foundation

extension Decodable {
```

```
    static func from(jsonData: Data) throws -> Self {
        return try JSONDecoder().decode(Self.self, from: jsonData)
    }
}

extension Encodable {
    func toJson() -> Data? {
        return try? JSONEncoder().encode(self)
    }
}

//  Convo.swift

import Foundation

typealias ConvoID = String

struct Convo {
    let id: ConvoID
    var messages: [Message] = []

    init(id: String, messages: [Message]) {
        self.id = id
        self.messages = messages
    }
}
```

```
extension Convo: Equatable, Identifiable {
}

//  Message.swift

import Foundation

struct Message {
    var id: String
    var role: Chat.Role
    var content: String
    var createdAt: Date

    init(id: String, role: Chat.Role, content: String, createdAt: Date) {
        self.id = id
        self.role = role
        self.content = content
        self.createdAt = createdAt
    }
}

extension Message: Equatable, Codable, Hashable, Identifiable {}

//  Preferences.swift

import Foundation
```

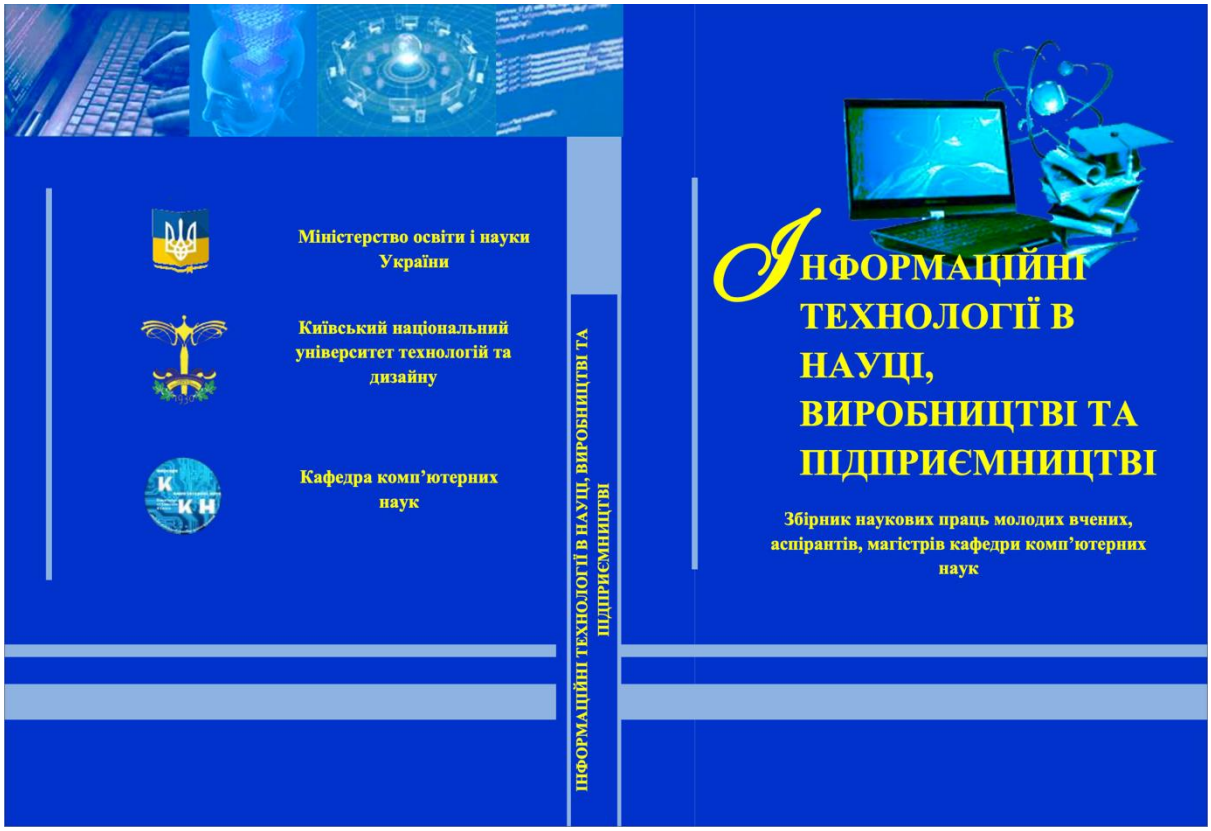
```
final class Preferences {
    enum Key: String {
        case speechRecognizerAuthorized
    }

    @UserDefaultsWrapper(wrappedValue: false, key:
Key.speechRecognizerAuthorized.rawValue)
    static var speechRecognizerAuthorized
}

//  AppStorageKeys.swift

import Foundation

enum AppStorageKeys {
    static let apiKey = "apiKey"
    static let userColorScheme = "userColorScheme"
}

//  VisionService.swift

import Vision

struct VisionService {
    func detectText(from cgImage: CGImage) async throws -> String {
        return try await withCheckedThrowingContinuation { continuation in
            let requestHandler = VNImageRequestHandler(cgImage: cgImage)
```

```
        let request = VNRecognizeTextRequest { request, error in
            if error != nil {
                continuation.resume(throwing:
VisionServiceError.unrecognizedText)
                return
            }

            guard let recognizedText = request.results as?
[VNRecognizedTextObservation] else { return }

            var fullText = ""
            let maximumCandidates = 1

            for observation in recognizedText {
                guard let candidate =
observation.topCandidates(maximumCandidates).first else { continue }
                let recognizedLine = candidate.string.trimmingCharacters(in:
.whitespacesAndNewlines)

                if recognizedLine.count > 3 {
                    fullText.append(recognizedLine + "\n")
                }
            }

            if !fullText.isEmpty {
                continuation.resume(returning: fullText)

            } else {
                continuation.resume(throwing: VisionServiceError.emptyText)
```

```
            }
        }

        request.recognitionLevel = .accurate
        if let languageCode = Locale.current.language.languageCode?.identifier
{
            request.recognitionLanguages = [languageCode]
        }

        do {
            try requestHandler.perform([request])
        } catch {
            continuation.resume(throwing:
VisionServiceError.failedToPerformRequest)
        }
    }
  }
}

//  VisionServiceError.swift

import Foundation

enum VisionServiceError: LocalizedError {
   case unrecognizedText
   case noTextFound
   case failedToPerformRequest
   case emptyText
```

```
    var errorDescription: String? {
        return switch self {
        case .unrecognizedText: "Unrecognized text"
        case .noTextFound: "No text found"
        case .failedToPerformRequest: "Unknow error"
        case .emptyText: "No text found"
        }
    }
}
```

Додаток Б


Міністерство освіти і науки України
Київський національний університет технологій та дизайну
Кафедра комп'ютерних наук
ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ В НАУЦІ, ВИРОБНИЦТВІ ТА ПІДПРИЄМНИЦТВІ
ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ В НАУЦІ, ВИРОБНИЦТВІ ТА ПІДПРИЄМНИЦТВІ
Збірник наукових праць молодих вчених, аспірантів, магістрів кафедри комп'ютерних наук

ПОНОМАРЕНКО Н.В., ДЕМКІВСЬКА Т.І.

ОГЛЯД ІНСТРУМЕНТІВ APPLE, ПОВ'ЯЗАНИХ З МАШИННИМ НАВЧАННЯМ

PONOMARENKO N.V., DEMKIVSKA T.I.

AN OVERVIEW OF APPLE'S MACHINE LEARNING TECHNOLOGIES

In the rapidly digitalizing world, the need for robust, responsive, and intelligent systems is paramount. Machine Learning (ML), a subset of artificial intelligence (AI), stands at the forefront of providing such capabilities. Within the Apple ecosystem, ML is harnessed through pre-trained frameworks like Vision, Natural Language, Sound Analysis and Speech or something like CoreML and CreateML that provide APIs to train your own models for specific tasks and integrate them into the OS, fostering a rich experience for both users and developers.

This article delves into some of these frameworks, discussing the challenges they address and their practical application in software development. To better illustrate these concepts, an application has been developed, showcasing these possibilities and shedding light on Apple's native UI development framework, SwiftUI.

Вступ

У сучасному світі розробка програмного забезпечення стає все більш складним завданням, яке вимагає від розробників не лише глибоких знань програмування, але й використання новітніх технологій. Однією з найбільш впливових екосистем у сфері розробки програмного забезпечення є екосистема Apple. Apple, як виробник популярних продуктів, таких як iPhone, iPad та Mac, надає розробникам широкі можливості для створення інноваційних додатків та сервісів.

В останні роки машинне навчання стало однією з ключових галузей в інформаційних технологіях. Воно відіграє важливу роль у багатьох сферах, включаючи обробку даних, розпізнавання образів, рекомендаційні системи та автоматичне управління. Машинне навчання дає змогу

програмному забезпеченню "навчитися" на основі великого обсягу даних та здійснювати складні аналітичні та прогностичні завдання.

Працюючи в екосистемі Apple, розробники отримують доступ до різноманітних інструментів та технологій, що дозволяють впроваджувати машинне навчання у свої проекти.

Основна частина

Екосистема Apple відрізняється цілісністю та взаємодією між різними продуктами. Це створює сприятливе середовище для розробників, які мають можливість створювати додатки, що працюють на всіх платформах Apple: iOS, macOS, watchOS та tvOS. Ця екосистема спирається на потужну мову програмування Swift.

Swift є сучасною, безпечною та ефективною мовою програмування, розробленою спеціально для платформ Apple. Вона пропонує високий рівень експресивності та простоти використання, що дозволяє розробникам швидко створювати надійне та ефективне програмне забезпечення. Swift також має потужну систему типів та безпеку, яка сприяє уникненню помилок та полегшує розробку складних додатків.

Основою розробки інтерфейсів користувача для додатків в екосистемі Apple є фреймворк SwiftUI. Він пропонує декларативний підхід до розробки інтерфейсів, де ви можете описувати, як ваш інтерфейс має виглядати та вести себе, замість написання великої кількості коду. Наразі такий підхід є трендом, що приходить на заміну імперативному програмуванню. Що підтверджує також випуск компанією Google свого фреймворку Jetpack Compose, який у свою чергу також слідує парадигмі декларативного програмування. Це робить розробку користувацького інтерфейсу більш простою, зрозумілою та ефективною. SwiftUI дозволяє створювати додатки, які працюють як на пристроях з екранами сенсорного керування, так і на пристроях з клавіатурою та мишею.

В області машинного навчання Apple пропонує ряд потужних фреймворків. Vision, наприклад, - це фреймворк для аналізу зображень та відео, який дозволяє створювати додатки з можливістю розпізнавання обличчя, слідкуванням за об'єктами на відео, визначенням тексту на зображеннях та багато іншого.

У свою чергу, Natural Language - це інструмент для роботи з текстом, який може проводити лемматизацію, морфологічний аналіз, визначення частин мови та багато іншого. З його допомогою можна створити додатки, які здатні аналізувати та обробляти текст на природній мові.

Sound Analysis - це фреймворк, який дозволяє додаткам визначати та розпізнавати звуки в аудіопотоці. Це може бути корисним для створення додатків, що реагують на звукові події в реальному світі.

Speech - це фреймворк, який дозволяє додаткам транскрибувати живу мову та перетворювати її на текст. Цей інструмент може бути

корисним у великому спектрі сценаріїв, від додатків для заміток до більш складних задач, як-от аналіз настрою користувача.

Для безпосередньої інтеграції моделей машинного навчання в додатки, Apple пропонує Core ML. Він надає розробникам можливість використовувати сучасні алгоритми машинного навчання для створення більш інтелектуальних додатків.

В свою чергу, Create ML надає інтерфейс для створення та тренування моделей машинного навчання. З його допомогою розробники можуть використовувати свої дані для створення власних моделей машинного навчання, які потім можуть бути інтегровані в додатки за допомогою Core ML.

Таким чином, розробники в екосистемі Apple мають у своєму розпорядженні широкий спектр інструментів для створення додатків з можливостями машинного навчання. Використання цих потужних фреймворків разом із SwiftUI дозволяє створити багатофункціональні, інтуїтивні та ефективні додатки, які виглядають та відчуваються як частина екосистеми Apple.

Висновки

У цій статті було досліджено та проаналізовано різноманітні технології машинного навчання, які надає Apple розробникам у своїй екосистемі. Ці технології, включаючи SwiftUI, Core ML, Create ML та спеціалізовані фреймворки, такі як Vision, Natural Language, Sound Analysis та Speech, дозволяють розробникам створювати програми з передовими алгоритмами машинного навчання для обробки зображень, розпізнавання тексту та аналізу звуку.

Переваги використання цих технологій у розробці програмного забезпечення для екосистеми Apple включають:

- висока продуктивність та швидкість обробки завдяки оптимізованому апаратному забезпеченню Apple;
- інтеграція з потужними функціями та сервісами, які пропонує Apple, забезпечує розробникам широкі можливості для створення інноваційних додатків;
- простота використання та зручність розробки завдяки SwiftUI та іншим інтуїтивним інструментам.

Аналіз показав, що використання технологій машинного навчання в програмному забезпеченні для екосистеми Apple може призвести до таких результатів:

Додаток В

MACHINE LEARNING ТА СПОСОБИ ЙОГО ВИКОРИСТАННЯ ПРИ РОЗРОБЦІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ В ЕКОСИСТЕМІ APPLE

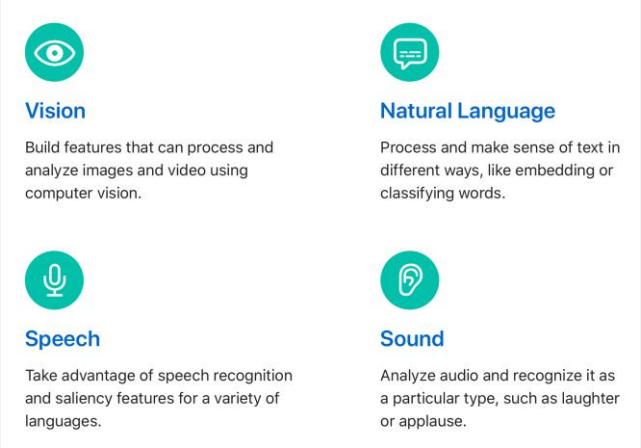
Виконавець – Пономаренко Н.В.

Науковий керівник – к.т.н., доц. Демківська Т.І.

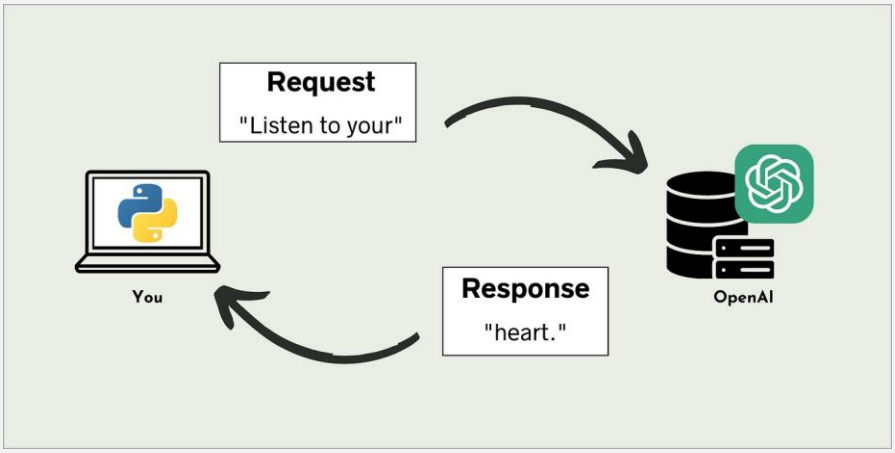
МЕТА РОБОТИ

- Розробка мультиплатформенного застосунку для екосистеми Apple, що інтегрується з OpenAI API та має можливості розпізнавання голосу та считування тексту із зображень.

ФРЕЙМВОРКИ МАШИННОГО НАВЧАННЯ ЯКІ НАДАЄ РОЗРОБНИКАМ APPLE



ПЕРЕНЕСЕННЯ ЗАВДАНЬ МАШИННОГО НАВЧАННЯ НА СЕРВЕРНУ ЧАСТИНУ



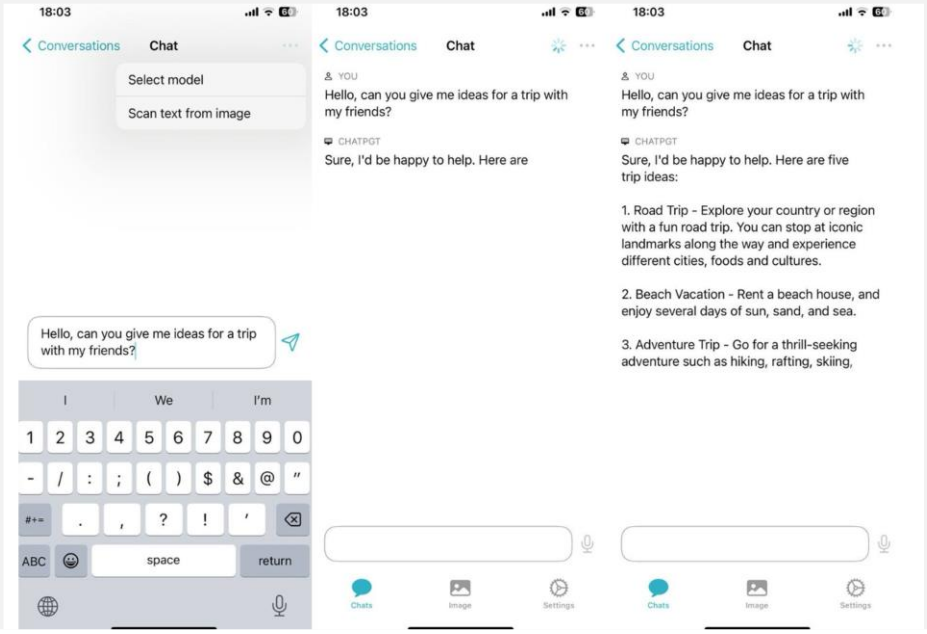
МОВА ПРОГРАМУВАННЯ SWIFT

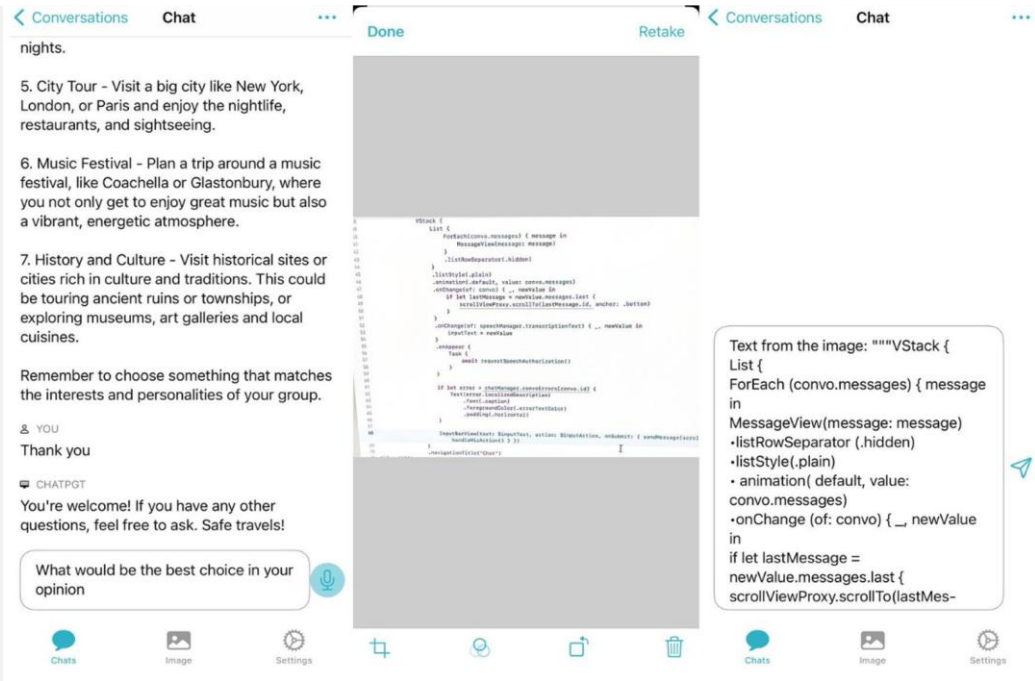
1. **Швидкодія**: Swift оптимізована для продуктивності. Вона швидше обробляє операції, ніж Objective-C, завдяки сучасному компілятору та оптимізованому коду.
2. **Безпека типів**: Строга система типізації та автоматична перевірка помилок допомагають уникнути помилок у коді, забезпечуючи більш безпечні додатки.
3. **Сучасний синтаксис**: Swift має чистий, лаконічний синтаксис, який робить код легшим для читання і написання, зменшуючи кількість коду, необхідного для вираження задуму.
4. **Легкість вивчення**: Swift розроблена з метою бути більш доступною для новачків, зробивши навчання мови програмування простішим і більш інтуїтивно зрозумілим.
5. **Сумісність з Objective-C**: Swift може співіснувати в одному проекті з кодом на Objective-C, що дозволяє розробникам поступово переходити на Swift без необхідності повної переписки існуючих проектів.

НОВІТНІЙ ФРЕЙМВОРК SWIFTUI

```swift
import SwiftUI
import Charts

struct BarChartView: View {
    let weekDays =
        Calendar.current.shortWeekdaySymbols
    let sales = [900,1500,500,200,1200,800,2500]

    var body: some View {
        Chart {
            ForEach(weekDays.indices, id: \.self){
                index in
                BarMark(
                    x: .value("Day",
                        weekDays[index]),
                    y: .value("Sales", sales[index])
                )
                .foregroundStyle(by: .value("Day",
                    weekDays[index]))
                .annotation{
                    Text("$\(sales[index])")
                }

            }
        }
    }
}

struct BarChartView_Previews: PreviewProvider {
    static var previews: some View {
        BarChartView()
    }
}
```

ПРЕЗЕНТАЦІЯ ДОДАТКУ





ВИСНОВКИ

Розроблено мультиплатформенний додаток, що надає можливість користувачеві спілкуватись з чат-ботом зі штучним інтелектом, інтегручись з OpenAI API. Додаток також використовує Vision Framework для розпізнавання тексту з картинок і Speech Framework для перетворення мовлення в текст, та для мережевих запитів використовує URLSession.