МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА ЗАСТОСУНКУ ДЛЯ ВИВЧЕННЯ ІНОЗЕМНИХ МОВ З АВТОМАТИЧНОЮ ГЕНЕРАЦІЄЮ ЗАВДАНЬ НА ОСНОВІ NLP»**

Виконав: студент 2 курсу, групи 8.1214-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
М.І. Носенко
(ініціали та прізвище)

Керівник професор кафедри програмної інженерії, доцент, к.ф.-м.н. Кудін О.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри фундаментальної та прикладної математики, доцент, к.ф.-м.н. Панасенко Є.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Носенку Миколі Івановичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка застосунку для вивчення іноземних мов з автоматичною генерацією завдань на основі NLP

керівник роботи Кудін Олексій Володимирович, к.ф.-м.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 28 » лютого 2025 року № 326-с

2. Строк подання студентом роботи 05.12.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз предметної області та обґрунтування гібридної методології.
2. Проєктування архітектури програмного комплексу.
3. Розробка та тестування програмного комплексу.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 07.03.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	10.04.2025	
2.	Збір вихідних даних.	20.06.2025	
3.	Обробка методичних та теоретичних	11.08.2025	
	джерел.		
4.	Розробка першого та другого розділу.	25.09.2025	
5.	Розробка третього розділу.	20.11.2025	
6.	Оформлення та нормоконтроль	01.12.2024	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	19.12.2025	

Студент _____ М.І. Носенко
(підпис) (ініціали та прізвище)

Керівник роботи _____ О.В. Кудін
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка застосунку для вивчення іноземних мов з автоматичною генерацією завдань на основі NLP»: 73 с., 48 рис., 6 табл., 21 джерело, 3 додатки.

АВТОМАТИЧНА ГЕНЕРАЦІЯ ПИТАНЬ, АРХІТЕКТУРА КЛІЄНТ-СЕРВЕР, ВЕЛИКІ МОВНІ МОДЕЛІ, ОБРОБКА ПРИРОДНОЇ МОВИ, FASTAPI, JETPACK COMPOSE, KOTLIN MULTIPLATFORM, NLTK, PYTHON, SPACY.

Об’єкт дослідження – процес автоматичної генерації навчальних завдань.

Мета роботи – створення додатку для вивчення мов, яка використовує методи обробки природної мови для автоматичної генерації навчальних завдань.

Методи дослідження – методи моделювання, проєктування, конструювання та тестування програмного забезпечення, методи NLP та методики оцінки якості згенерованих системою завдань.

Розроблено програмний комплекс, що складається з API на Python/FastAPI та мобільного додатку на Kotlin Multiplatform. Обґрунтовано гібридну архітектуру, що розмежовує завдання: класичний NLP використовується для надійних, але простих вправ, тоді як LLM залучаються вибірково для генерації складних, контекстних завдань. Новизна підходу полягає у балансуванні педагогічної якості, обчислювальних витрат та екологічного впливу.

В результаті роботи отримано програмний комплекс, що реалізує гібридну методологію та дозволяє генерувати адаптивні навчальні завдання з довільних текстів користувача.

SUMMARY

Master’s qualifying paper “Development of a Language Learning Application with Automated Task Generation Based on NLP”: 73 pages., 48 figures, 6 tables, 21 references, 3 supplements.

AUTOMATIC QUESTION GENERATION, CLIENT-SERVER ARCHITECTURE, FASTAPI, JETPACK COMPOSE, LARGE LANGUAGE MODELS, KOTLIN MULTIPLATFORM, NATURAL LANGUAGE PROCESSING, NLTK, PYTHON, SPACY.

The object of the study is the process of automatic generation of learning tasks.

The purpose of the work is the creation of a language learning application that uses natural language processing methods for automatic generation of learning tasks.

The methods of research are methods of modeling, design, implementation and testing of software, NLP methods and quality assessment methods for system-generated tasks.

A software system has been developed, consisting of a Python/FastAPI API and a Kotlin Multiplatform mobile application. A hybrid architecture is proposed to differentiate tasks: classical NLP is used for reliable but simple exercises, while LLM is selectively employed to generate complex, context-aware tasks. The novelty of the approach lies in balancing pedagogical quality, computational costs and environmental impact.

The work resulted in a software system that implements the hybrid methodology and allows for the generation of adaptive educational tasks based on arbitrary user texts.

ЗМІСТ

Завдання на кваліфікаційну роботу……...2
Реферат 4
Summary 5
Вступ..... 8
1 Аналіз предметної області та обґрунтування гібридної методології 9
1.1 Огляд методів автоматичної генерації завдань..... 9
1.2 Аналіз підходів на основі класичного NLP..... 10
1.3 Аналіз підходів на основі великих мовних моделей (LLM)..... 12
1.4 Обґрунтування гібридної методології 13
1.5 Стратегії генерації відволікаючих відповідей 15
1.6 Огляд та порівняння аналогічних рішень..... 16
1.7 Ккологічні та обчислювальні аспекти LLM як чинник гібридизації. 18
1.8 Методика оцінки якості згенерованих завдань..... 19
2 Проєктування архітектури програмного комплексу 21
2.1 Інструменти та технології 21
2.2 Архітектура API 22
2.3 Архітектура мобільного додатку..... 23
2.4 Прототипування та сценарії використання 25
3 Розробка та тестування програмного комплексу..... 30
3.1 Реалізація серверної частини 30
3.2 Тестування серверної частини..... 33
3.3 Реалізація мобільного додатку 35
3.4 Тестування мобільного додатку 42
3.5 Розгортання та конфігурація інфраструктури AWS..... 44
3.6 Оцінка згенерованих завдань..... 46
Висновки 49
Перелік посилань..... 50

Додаток А Посилання на Git 53
Додаток Б Прототип мобільного додатку 54
Додаток В Приклади роботи мобільного додатку 62

ВСТУП

Автоматична генерація питань (AQG) стає невід'ємною частиною сучасних освітніх технологій, особливо в умовах поширення масових відкритих онлайн-курсів та платформ електронного навчання. Цей процес передбачає використання обчислювальних методів для автоматичного створення оціночних матеріалів (вікторин, граматичних вправ) безпосередньо з текстів. Історично ця галузь спиралася на класичні методи обробки природної мови (NLP) для виконання структурованих завдань. Однак нещодавній розвиток великих мовних (LLM) моделей відкрив нові горизонти для генерації більш складних, контекстно-залежних та семантично насичених питань.

Актуальність роботи зумовлена необхідністю подолання обмежень, властивих кожному з підходів. З одного боку, класичні методи NLP є прозорими, обчислювально ефективними та надійними для створення простих вправ, як-от заповнення пропусків. Проте їм бракує гнучкості для створення завдань вищого когнітивного рівня. З іншого боку, LLM демонструють високу якість у генерації складних питань та правдоподібних дистракторів, але їхнє повномасштабне розгортання є дорогим, а результати можуть бути педагогічно нерелевантними без належного структурного контролю.

Метою даної роботи є створення додатку для вивчення мов, що використовує методи обробки природної мови для автоматичної генерації навчальних завдань.

На основі поставленої мети можливо виділити такі задачі цієї роботи:

– порівняльний аналіз класичних NLP та LLM для AQG;
– проєктування гібридної системної архітектури;
– розробка API для генерації простих завдань та складних завдань;
– розробка мобільного додатку та його інтеграція зі створеним API.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОБҐРУНТУВАННЯ ГІБРИДНОЇ МЕТОДОЛОГІЇ

1.1 Огляд методів автоматичної генерації завдань

Автоматична генерація питань стосується обчислювального процесу генерування питань з тексту або структурованих даних без втручання людини. В освітньому контексті системи AQG розроблені для підтримки навчання шляхом автоматичного створення оціночних матеріалів, таких як вікторини, перевірки розуміння та граматичні вправи. Метою AQG є не лише зменшення ручного навантаження на викладачів, але й підвищення залученості студентів та сприяння персоналізованому навчальному досвіду [1].

Актуальність AQG в освіті значно зросла за останні роки, особливо з розвитком масових відкритих онлайн-курсів, інших технологій електронного навчання, поширенням інструментів обробки природної мови та штучного інтелекту (ШІ) [2, 3]. У вивченні мови системи AQG служать цінними ресурсами для підкріплення словникового запасу, розпізнавання граматичних образів та розуміння прочитаного. Такі формати, як питання із заповненням пропусків та питання з множинним вибором, особливо ефективні для тих, хто вивчає іноземні мови, оскільки вони сприяють активному запам’ятовуванню, увазі до лінгвістичної структури та контекстуальному розумінню. Дослідження показують, що автоматизовані оцінювання можуть пришвидшити цикли зворотного зв’язку, підтримувати диференційоване навчання та покращувати мотивацію студентів за допомогою гейміфікованих взаємодій [4].

Варто виділити два підходи AQG: системи на основі правил та системи на основі великих мовних моделей. AQG на основі правил використовує класичні методи NLP, такі як токенізація, тегування частин мови (POS) та розпізнавання іменованих сутностей (NER), для вилучення структурованих

лінгвістичних ознак з вхідного тексту та створення синтаксично значущих питань. Ці методи часто прозорі, обчислювально ефективні та інтерпретовані, що робить їх придатними для економічно чутливих застосувань, таких як вивчення мови в умовах обмеженого ресурсопостачання [5].

На противагу цьому, AQG на основі LLM використовує попередньо навчені нейронні архітектури, такі як трансформери (наприклад, GPT, BERT), для створення питань безпосередньо з необробленого тексту, що часто вимагає значних обчислювальних ресурсів. Хоча ці моделі можуть генерувати складні та контекстно-залежні питання, вони створюють проблеми, пов’язані з вартістю, контролем та відтворюваністю. Гібридні методи: поєднання класичного NLP з вибірковим використанням LLM можуть досягти балансу між масштабованістю та якістю [6].

Інформаційні системи, які мають на меті створювати завдання, зосереджені на граматиці та ідіоматичному розумінні для тих, хто вивчає іноземні мови, класичні методи NLP залишаються не тільки життєздатними, але й стратегічно вигідними. Вони відповідають педагогічним потребам щодо інтерпретованості, точності та адаптації до специфічної предметної області, уникаючи при цьому фінансових та технічних витрат, повномасштабного розгортання LLM.

1.2 Аналіз підходів на основі класичного NLP

Класичні методи обробки природної мови можуть слугувати основою економічно ефективних та інтерпретованих систем автоматичної генерації питань [7]. На відміну від великих мовних моделей, які часто функціонують як чорні скриньки, класичні методи NLP забезпечують прозорі та керовані механізми для вилучення лінгвістичних структур та генерації педагогічно обґрунтованих питань. Ці методи (токенізація, позначення частин мови, розпізнавання іменованих сутностей та синтаксичне зіставлення шаблонів)

дозволяють точно контролювати вміст завдання, що робить їх ідеальними для застосування в системах навчання, орієнтованих на граматику [8, 9].

На базовому рівні токенізація використовується для сегментації вхідного тексту на слова або підсловні одиниці, що є першим кроком у лінгвістичному аналізі. Далі йде позначення частин мови (POS), яке призначає граматичні мітки (наприклад, іменник, дієслово, прикметник) кожному токену [9]. Ці анотації мають вирішальне значення для ідентифікації значущих одиниць у реченнях, які підходять для генерації питань.

Ключовим розширенням тегування POS є розпізнавання іменованих сутностей, яке ідентифікує певні категорії, такі як люди, місця, організації, дати тощо, у тексті. NER особливо корисний у створенні навчальних вікторин, оскільки він виділяє сутності, які є семантично щільними та контекстуально багатими, що є ідеальними кандидатами для видалення в питаннях із заповненням пропусків або як опорні точки для генерації дистракторів [9].

Ще одним важливим компонентом є синтаксичне зіставлення зі зразками на основі правил, яке передбачає застосування вручну або шаблонних правил для визначення структур речень, придатних для перетворення на питання [8]. Наприклад, правила можуть бути розроблені для виявлення оповідних речень з чітким порядком підмет-дієслово-додаток або для розпізнавання підрядних речень, де видалення ключового елемента зберігає граматику, водночас вводячи прогалину в знаннях. Ці вручну створені правила пропонують високий ступінь інтерпретованості та можуть бути узгоджені з лінгвістичними цілями, характерними для викладання іноземної мови, такими як практика дієвідмін або використання артиклів.

Таким чином, пріоритизація частин мови та вилучення іменованих сутностей служать не лише технічними стратегіями, але й педагогічними важелями. Вони дозволяють системам генерації завдань виділяти основний словниковий запас та граматичні структури, контролювати когнітивне навантаження та гарантувати, що згенеровані питання є лінгвістично значущими.

1.3 Аналіз підходів на основі великих мовних моделей (LLM)

Інтеграція штучного інтелекту в освітнє середовище дедалі частіше набуває форм, що безпосередньо впливають на процеси навчання, викладання та академічного управління. Одним із найбільш зрілих напрямів такого застосування є автоматичне створення персоналізованих завдань, включно з тестами. Ця функція отримала не лише теоретичне обґрунтування, а й емпіричну підтримку [10].

Методологічно, генерація завдань за допомогою LLM фундаментально відрізняється від підходів на основі правил. Замість явного синтаксичного розбору та шаблонів, LLM використовують вивчені статистичні шаблони для генерації тексту. Основних підходів два: інжиніринг промптів (prompt engineering) та доналаштування (fine-tuning). При інжинірингу промптів попередньо навчена модель (наприклад, GPT-4) отримує вхідний контекст (текст, з якого потрібно згенерувати питання) разом із чіткою інструкцією. Ця інструкція, або «промпт», може бути простою («zero-shot»), наприклад: «Згенеруй 5 питань із заповненням пропусків на основі цього тексту», або вона може містити кілька прикладів («few-shot»), щоб показати моделі бажаний формат. Цей метод гнучкий і не вимагає додаткового навчання, але результати можуть бути менш передбачуваними [11].

Для вищої якості та кращого контролю застосовується доналаштування. Цей процес бере базову «фундаментальну» модель (наприклад, T5 або BERT) і продовжує її навчання на спеціалізованому, меншому наборі даних. Цей набір даних може мати різні формати залежно від задачі. Одним із приладів є структура «Речення з пропущеними словами, Очікуване повне речення». Альтернативною структурою є «Питання, Відповідь». У випадку автоматичної генерації завдань доречною буде структура «Оригінальний текст, Питання» або «Оригінальний текст, Питання з відповідями». В останньому прикладі під час доналаштування модель вчиться саме завданню генерації питань, адаптуючись до специфічного стилю та складності.

Поширеною та ефективною технікою є ланцюжок міркувань (chain of thought). Замість того, щоб вказувати LLM створити будь-яке питання з уривка, система спочатку генерує міркування про те, як вирішити задачу та після цього надає фінальну відповідь. Цей підхід може підвищити якість відповідей, але вимагає використання великих моделей. Малі мовні моделі не покращують свої відповіді під час використання цього підходу [11].

Доцільно розглянути результати досліджень, що демонструють потенціал великих мовних моделей у складних когнітивних умовах. Одним із таких є дослідження, присвячене створенню тестових завдань для медичної освіти, де ChatGPT (на основі GPT-4) використовувався для генерації клінічних вінієт у форматі USMLE/COMLEX-USA [12].

У межах експерименту було створено 65 тестових запитань, з яких 49% мали виявлені фахівцями помилки змістового або структурного характеру. Водночас 91% завдань були визнані такими, що можуть бути використані після редагування, а 70% із них легко або дуже легко адаптуються до потрібного формату. Показово, що всі запитання відповідали базовим формальним критеріям: не містили граматичних помилок, мали правильну структуру варіантів відповідей, а основні недоліки стосувалися уточнення формулювань або релевантності відволікаючих відповідей (дистракторів).

1.4 Обґрунтування гібридної методології

Запропонована система автоматичної генерації навчальних завдань базується на гібридній архітектурі, що поєднує швидкість та точність класичних методів обробки природної мови з семантичною гнучкістю великих мовних моделей. Такий підхід дозволяє розмежувати завдання за складністю та типом, використовуючи для кожного з них найбільш ефективний інструмент. Загальна логіка системи полягає у використанні підсистеми, де класичні NLP-модулі створюють основу: структуровані та граматично прості

завдання, а LLM залучаються для виконання більш когнітивно складних операцій.

Ця методологія чітко розмежовує ролі класичного NLP та LLM, запроваджуючи модель, де залучення LLM відбувається вибірково і контролюється діями користувача. У пасивному режимі, система наповнює банк речень, що можливо використати для генерації простих питань. Логіка наповнення банку речень полягає у отриманні прогресу читання та виокремлення випадкових речень з тексту. Це дозволяє ефективно використовувати ресурси та забезпечує високу продуктивність.

На противагу цьому, активація LLM відбувається в активному режимі та слугує відповіддю на конкретний запит користувача. Початком для залучення моделі є свідома дія: виділення певного речення чи фрагмента тексту. Система розцінює цю дію як запит на поглиблений аналіз. Саме тоді LLM проводить семантичну та стилістичну інтерпретацію лише обраного уривка, шукаючи складніші лінгвістичні конструкції, які класичний NLP міг пропустити, зокрема ідіоми, умовні речення, пасивний стан чи фразові дієслова. В результаті система генерує персоналізовані завдання вищого порядку, що відповідають навчальним цілям учня.

Така стратегія, орієнтована на взаємодію, є оптимальною з кількох причин. По-перше, вона значно економить обчислювальні ресурси, оскільки дорогі запити до LLM виконуються вибірково, а не застосовуються до всього тексту. По-друге, це підвищує педагогічну релевантність завдань, оскільки вони стосуються саме того фрагмента, який є актуальним для учня в даний момент. Це дозволяє поєднати швидкість та стабільність класичних методів для загального огляду з гнучкістю та глибиною LLM для цільового, поглибленого навчання.

1.5 Стратегії генерації відволікаючих відповідей

Розробка відволікаючих відповідей, тобто неправильних, але правдоподібних альтернатив є центральною проблемою у створенні питань з множинним вибором та питань із заповненням пропусків, особливо в системах автоматизованої генерації питань. Ефективні дистрактори повинні відповідати подвійним критеріям синтаксичної доречності та семантичної правдоподібності, щоб гарантувати, що завдання оцінює лінгвістичну компетентність, а не стратегію складання тесту чи поверхневе розпізнавання.

Більшість систем AQG, спрямованих на вивчення мови, спираються на відносно прості типи питань, але якість дистракторів відіграє непропорційну роль у визначенні валідності цих завдань [13]. Коли дистрактори порушують граматичні або семантичні норми, учасник тестування часто може усунути їх, не взаємодіючи змістом змістовно, знижуючи діагностичну цінність питання.

Одна поширена стратегія створення синтаксично доречних дистракторів полягає у збереженні частин мови та морфологічних особливостей цільового слова. Наприклад, у завданнях на закриття, де пропущений елемент є дієсловом, дистрактори витягуються з тієї ж парадигми дієвідмін (наприклад, писати, писав, писати, написано), що дозволяє учневі вибрати правильну форму на основі часу, аспекту або синтаксичної ролі.

Семантична правдоподібність часто вирішується шляхом вилучення дистракторів із зовнішніх баз знань, таких як WordNet або Вікіпедія. Ці ресурси підтримують пошук лексичних елементів у тому ж семантичному полі, що й правильна відповідь, дозволяючи створювати варіанти, які є тематично або узгодженими. Ця стратегія гарантує, що всі варіанти не лише граматично правильно сформовані, але й взаємозамінні на поверхневому рівні, що вимагає від учня глибшого контекстуального розуміння.

Ще один принциповий метод передбачає генерацію контрастних наборів дистракторів, які відрізняються від правильної відповіді точно визначеними лінгвістичними або концептуальними способами. Ці майже пропущені

дистрактори особливо цінні в граматичних завданнях, де учень повинен розрізняти альтернативи, що відрізняються відмінком, числом або синтаксичною функцією. Наприклад, у питанні, що вимагає правильного прийменника, дистрактори можуть бути граматично можливими, але контекстуально неправильними, тим самим посилюючи чутливість учня до колокаційних норм та семантики прийменників. Таке створення дистракторів, хоча й вимогливе з точки зору обчислень, було показано, що підвищує як залученість, так і діагностичну точність [14, 15].

Незважаючи на ці методологічні досягнення, сучасні системи AQG все ще стикаються з постійними труднощами у балансуванні правдоподібності та складності. Багатьом системам бракує валідованих моделей складності дистракторів, і мало хто надає пояснювальний зворотний зв’язок, коли учні вибирають неправильні варіанти. Крім того, надмірне використання синтаксичних шаблонів без семантичних обмежень часто призводить до появи дистракторів, які можна тривіально відхилити [13].

З огляду на ці обмеження, варто звернути увагу на стратегії, що поєднують синтаксичний контроль на основі правил із семантичним збагаченням з великих баз знань або попередньо навчених мовних моделей. Ці підходи можуть гарантувати, що дистрактори не лише структурно узгоджуються зі основою мови, але й контекстуально інтегровані та когнітивно вимогливі.

1.6 Огляд та порівняння аналогічних рішень

Існує низка цифрових платформ, що пропонують різні аспекти підтримки при вивченні іноземної мови через читання. Наприклад, LingQ та ReadLang дозволяють працювати з автентичними текстами, перекладати слова та створювати персональні словники. Проте ці сервіси не надають автоматично згенерованих граматичних чи лексичних вправ на основі тексту

та майже не враховують індивідуальний рівень знань або потреби користувача. Порівняння цих сервісів наведено в таблиці 1.1.

Таблиця 1.1 – Порівняльна таблиця аналогів

Платформа	Читання власних текстів	Генерація вправ	Персоналізація контенту	Використання NLP / LLM	Адаптація до рівня
LingQ	Дозволяє імпорт текстів	Відсутня	Обмежена словникова робота	Немає	Часткова через контент
ReadLang	Повна підтримка	Відсутня	Словник і збереження слів	Немає	Немає
Duolingo Stories	Власні тексти не підтримуються	Присутні, але шаблонні	Фіксовані сценарії	Використовує LLM	Вбудована в структуру курсу
Quazel.ai	Орієнтовано на діалог	Так, через LLM	Тематичні або граматичні запити	Використовує LLM	Через діалог
Запропонований підхід	Підтримка будь-якого тексту	Автоматична на основі тексту	За виділеними фрагментами тексту	Поєднання NLP і LLM	Динамічна адаптація

Duolingo Stories частково реалізує ідею інтерактивного читання через адаптовані діалоги з контрольними запитаннями. Однак їхній контент є фіксованим, і користувач не може завантажити власний текст або адаптувати завдання під свої цілі.

Інструменти на базі великих мовних моделей, такі як ChatGPT або експериментальні агенти на кшталт Quazel.ai, демонструють високий потенціал у створенні персоналізованих мовних завдань на основі будь-якого тексту. Проте наразі вони не мають цілісного освітнього інтерфейсу, що поєднував би читання, автоматичну генерацію вправ, збереження прогресу та врахування афективних факторів навчання.

Таким чином, хоча окремі компоненти вже реалізовані у різних продуктах, відсутня інтегрована система, яка б поєднувала:

– роботу з довільним текстом користувача;
– автоматичну генерацію граматичних і лексичних завдань;
– адаптацію під рівень володіння мовою.

1.7 Екологічні та обчислювальні аспекти LLM як чинник гібридизації

На противагу класичним методам обробки природної мови, нейронні методи, особливо ті, що базуються на великих мовних моделях (наприклад, GPT або Llama), здатні генерувати плавні, контекстно багаті та різноманітні питання. Ці системи ефективно справляються з формулюванням відкритих запитань та інтерпретацією нюансованих семантичних зв'язків у великих текстах. Однак їх використання супроводжується значними компромісами, насамперед високими обчислювальними витратами та негативним впливом на довкілля.

Навчання великомасштабних мовних моделей є надзвичайно енергоємним процесом. Згідно з методологією обліку вуглецю, наведеною у звіті Meta щодо Llama 2, лише попереднє навчання цієї моделі потребувало понад 3,3 мільйона годин роботи GPU, що призвело до викидів приблизно 539 тонн CO_2-екв [16]. Загалом, обсяг обчислень для навчання передових моделей глибокого навчання зріс у 300 000 разів за період з 2015 по 2021 рік, значно випереджаючи темпи закону Мура [17]. Екологічний слід навчання варіюється залежно від архітектури моделі, ефективності обладнання (наприклад, перехід від GPU P100 до TPUv4 скорочує викиди у сотні разів) та енергоефективності дата-центрів [18].

Проте високі витрати не обмежуються лише фазою навчання. Хоча донавчання менших моделей (наприклад, BERT-small) споживає відносно небагато енергії (близько 3 кВт·год), використання сучасних LLM у режимі інференсу (генерації відповідей) створює постійне навантаження на інфраструктуру [19]. Аналіз екологічного впливу показує, що енерговитрати на окремий запит до потужної моделі рівня Llama-3-70B складають приблизно 0,0195 кВт·год, а з урахуванням повного життєвого циклу обладнання потребують витрати близько 140 мл води [20]. Хоча ці цифри можуть здаватися незначними в межах одного запиту, при масштабуванні системи на тисячі користувачів вони формують суттєвий екологічний слід.

Напруга між педагогічною корисністю та екологічною відповідальністю стає очевидною при порівнянні традиційних систем NLP та сучасних LLM. Якщо системи на основі правил можуть працювати на пристроях із низьким рівнем ресурсів майже без відчутних енергозатрат, то великі нейронні моделі вимагають потужних графічних процесорів, що робить їхнє використання дорогим як у фінансовому, так і в екологічному вимірі.

1.8 Методика оцінки якості згенерованих завдань

У системах, які алгоритмічно генерують освітній контент, зокрема питання з вибором однієї правильної відповіді або питання на заповнення пропусків, оцінка якості згенерованих елементів має вирішальне значення для забезпечення педагогічної ефективності, ясності та надійності. Структурована система оцінювання не лише підтримує внутрішнє тестування та оцінку якості, але й сприяє порівнянню різних стратегій генерації, включаючи підходи на основі правил, гібридні та нейронні.

Доречним буде використати методологію, яка запроваджує формалізовану систему оцінювання для оцінювання тестів. Система окреслює п’ять основних критеріїв, кожен з яких стосується певного виміру якості згенерованого питання [21]:

- граматична плинність (GF): вимірює синтаксичну та граматичну правильність питання;
- наявність відповіді (A): оцінює, чи є питання зрозумілим та чи має воно чітку, правильну відповідь;
- різноманітність (D): відображає лексичну та тематичну різноманітність у питаннях;
- складність (C): відображає когнітивне навантаження або міркування, необхідні для відповіді;
- релевантність (R): оцінює тематичну відповідність між питанням та його вихідним матеріалом.

Кожен із цих показників оцінюється за шкалою від 1 (погано) до 5 (відмінно) оцінювачами-людьми. Для заданого показника середній бал усіх оцінювачів розраховується за допомогою формули 1.1.

$$Avg.Score = \frac{\sum_{i=1}^{n} S_i}{n}, \quad (1.1)$$

де S_i – результат оцінки, n –загальна кількість оцінювачів.

Щоб підсумувати загальну ефективність завдання вікторини, визначається загальний бал ефективності (OPS) (формула 1.2). Цей нормалізований показник дозволяє порівнювати якість питань за різними методами генерації та підтримує оцінку якості експериментальних систем.

$$OPS = \frac{GF+A+D+C+R}{5}, \quad (1.2)$$

де GF – граматична плинність, A – наявність відповіді,

D – різноманітність, C – складність,

R – релевантність.

Ці метрики можуть бути вбудовані у внутрішні процедури тестування, цикли ітерацій на основі зворотного зв’язку або у порівняльні дослідження між різними версіями запропонованої системи.

2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ПРОГРАМНОГО КОМПЛЕКСУ

2.1 Інструменти та технології

Для реалізації серверної частини (API) було обрано стек технологій на базі мови Python. В якості основного вебфреймворку використано FastAPI завдяки його високій продуктивності, асинхронній природі та автоматичній валідації даних, що забезпечується інтеграцією з Pydantic. Взаємодія з реляційною базою даних PostgreSQL реалізована через ORM SQLAlchemy, використовуючи драйвер psycopg. Для розгортання додатку застосовується WSGI-сервер Gunicorn. Безпека системи забезпечується на кількох рівнях: автентифікація користувачів базується на JSON Web Tokens (PyJWT), а паролі безпечно хешуються за допомогою bcrypt та passlib. Інтеграція з інфраструктурою Amazon Web Services, зокрема для управління сховищами або конфігураціями, здійснюється через Boto3 SDK.

Функціональне ядро API зосереджено на обробці природної мови та генерації навчального контенту. Для виконання фундаментальних лінгвістичних завдань, таких як токенізація та морфологічний аналіз, використовуються бібліотеки NLTK, SpaCy та PatternLite. Важливою складовою є інтеграція з OpenAI API, що дозволяє залучати потужні великі мовні моделі для генерації та оцінки завдань.

Мобільний додаток розроблено з використанням Kotlin Multiplatform (KMP), що дозволяє спільно використовувати бізнес-логіку, мережеву взаємодію та логіку доступу до даних між цільовими платформами. Архітектура проєкту спирається на ключові бібліотеки екосистеми Kotlin: Kotlinx Coroutines для управління асинхронними операціями, Koin для реалізації шаблону впровадження залежностей (Dependency Injection) та Kotlinx Serialization для серіалізації/десеріалізації даних. Графічний інтерфейс користувача побудований на декларативному фреймворку Jetpack Compose.

Взаємодія KMP-додатку з API здійснюється через багатоплатформний HTTP-клієнт Ktor Client, налаштований для роботи з JSON. Для локального зберігання даних та кешування застосовується SQLDelight, що генерує типобезпечний Kotlin-код зі SQL-запитів, з використанням специфічних драйверів для Android, JVM та нативних платформ. Для реалізації ключової функціональності читання електронних книг використано бібліотеку Readium Toolkit. Управління спільними ресурсами у багатоплатформному середовищі реалізовано за допомогою Moko Resources.

2.2 Архітектура API

Архітектура серверної частини (API) спроєктована з використанням фреймворку FastAPI та дотримується багатошарового підходу для чіткого розподілу відповідальності (рис. 2.1). Така сегрегація компонентів дозволяє ізолювати логіку обробки даних від мережевих інтерфейсів, що значно спрощує тестування окремих модулів та подальше масштабування системи. Потік запитів ініціюється зовнішнім клієнтом, мобільним додатком KMP, і надходить до головного екземпляру FastAPI. Кожен запит спершу проходить через проміжне програмне забезпечення (Middleware) для автентифікації користувача. Успішно автентифіковані запити передаються на рівень маршрутизації (API Layer), де відповідні контролери (Routers), наприклад, роутер завдань (Quizzes Router), обробляють ендпоінти. Для виконання бізнес-логіки роутер завдань звертається до сервісного рівня (Service Layer).

Ключова функціональність генерації навчальних завдань реалізована через шаблон «Стратегія». Роутер делегує процес створення контенту одній із конкретних стратегій, що реалізують абстрактний базовий клас QuizGenerationStrategy. Залежно від типу необхідного завдання, обирається або ContextQuizStrategyLLM, яка звертається до зовнішнього LLM API (OpenAI) для генерації складних контекстних питань, або стратегії

SimpleQuizStrategy та SequenceQuizStrategy. Ці стратегії використовують інструментарій з рівня даних та утиліт (Data & Utility Layer), зокрема локальні NLP-бібліотеки (NLTK, SpaCy, Pattern.en) та токенізатор, для аналізу тексту. Після завершення всіх операцій роутер формує та повертає клієнту HTTP-відповідь.

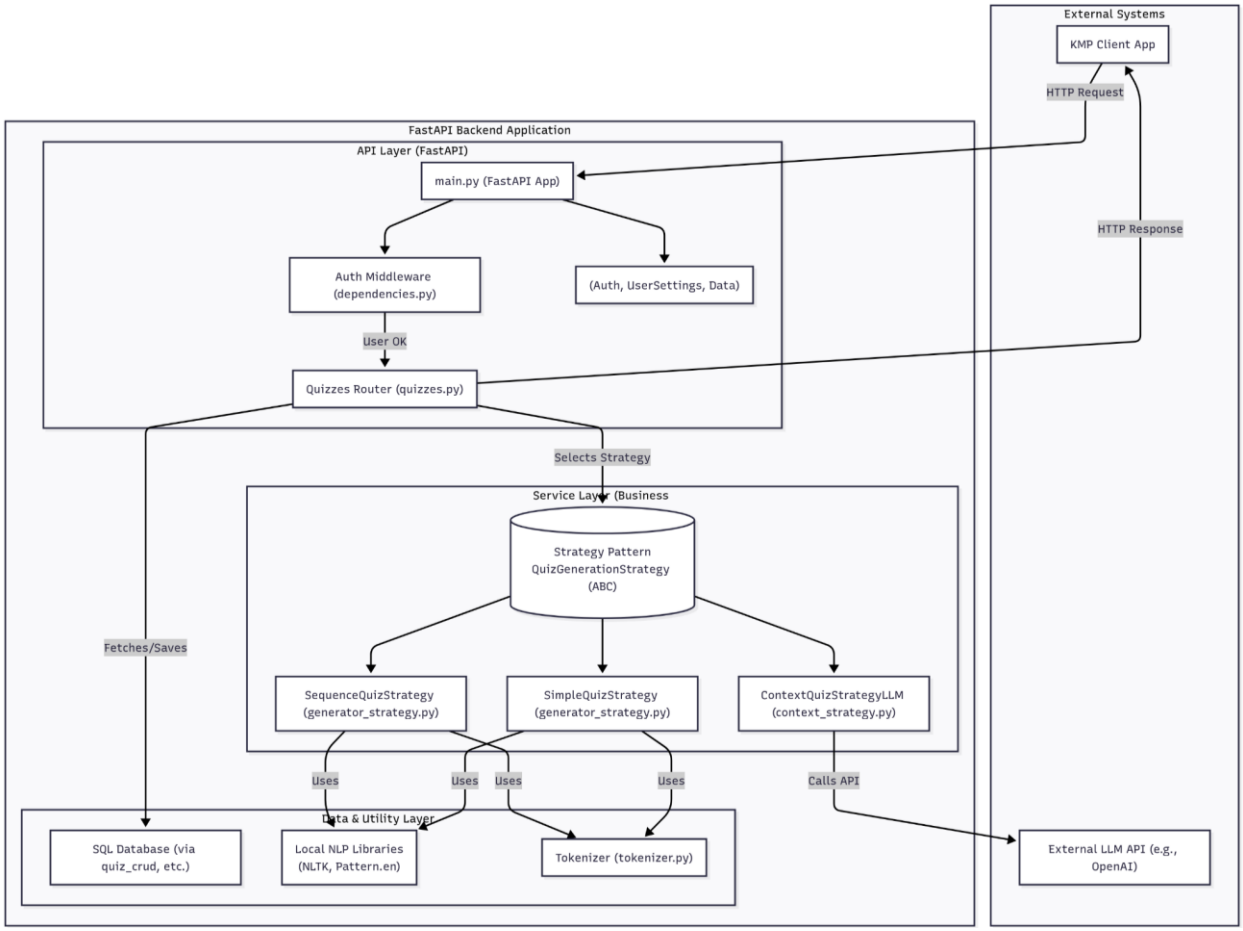


Рисунок 2.1 – Архітектура API

2.3 Архітектура мобільного додатку

Архітектура мобільного додатку побудована на засадах Kotlin Multiplatform (KMP), що забезпечує чіткий поділ між платформо-специфічними компонентами та спільним модулем commonMain. Платформний рівень, такий як Android або Desktop, містить виключно компоненти, відповідальні за інтерфейс користувача (наприклад, Jetpack

Compose) та управління його станом (ViewModel). Кожен ViewModel конкретної платформи взаємодіє зі спільним кодом, спостерігаючи за змінами стану та ініціюючи бізнес-операції (рис. 2.2).

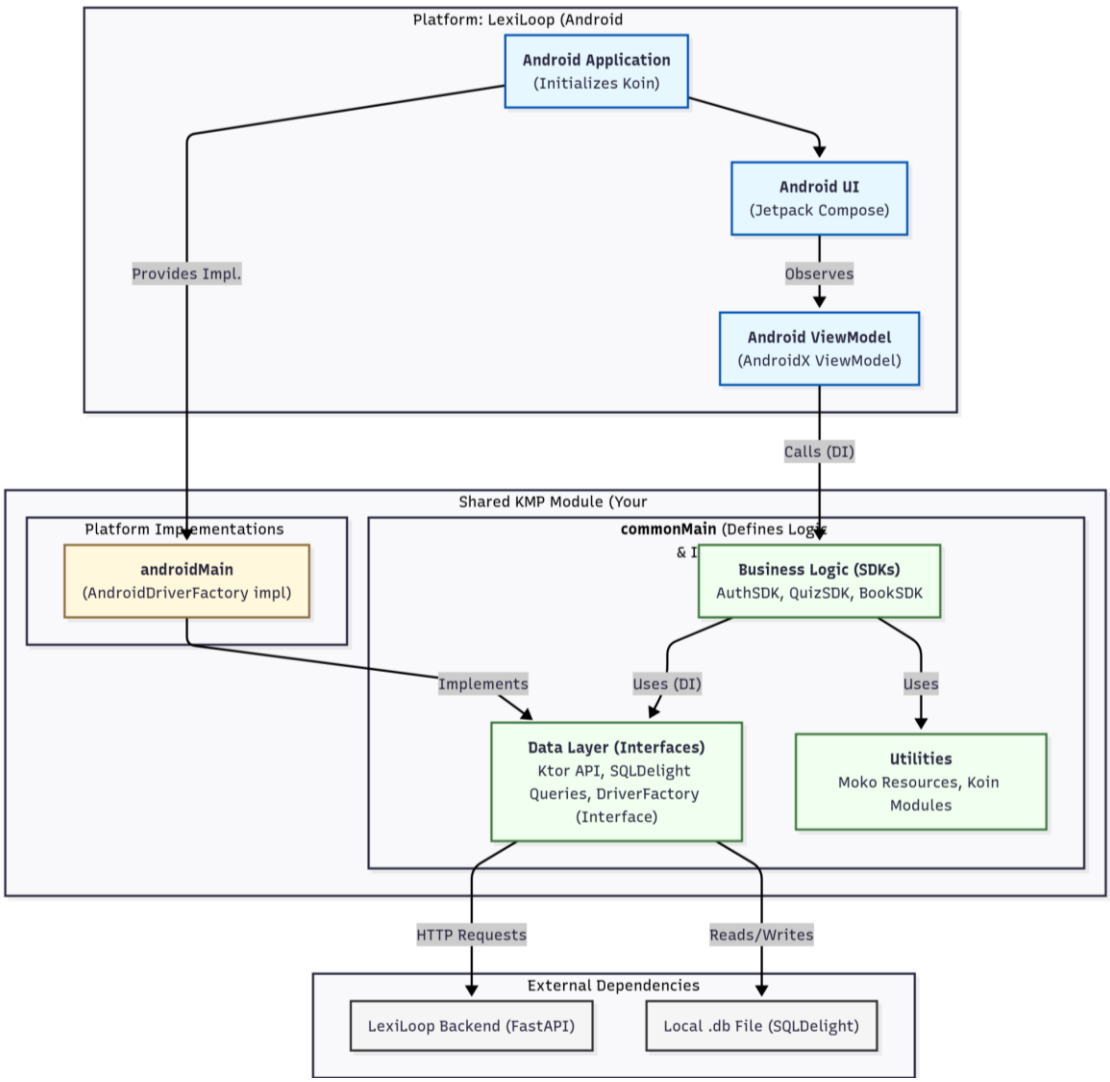


Рисунок 2.2 – Архітектура мобільного додатку

Ядро бізнес-логіки інкапсульовано у спільному модулі commonMain. Ця логіка структурована у вигляді окремих модулів (SDK), таких як AuthSDK, QuizSDK та BookSDK, що реалізують шаблон «Фасад». Ці фасади надають спрощений та єдиний інтерфейс для ViewModel, приховуючи складність реалізації. Нижче рівня SDK знаходиться шар даних (Data Layer), який відповідає за взаємодію з зовнішніми джерелами. Цей шар містить мережевий клієнт (Ktor Client) для комунікації з сервером API та локальну базу даних (SQLDelight) для кешування та зберігання даних на пристрої. Для

забезпечення роботи платформо-залежних компонентів, таких як драйвер бази даних SQLDelight, використовується механізм Впровадження залежностей через інтерфейси. Модуль androidMain надає конкретну реалізацію інтерфейсу DriverFactory (наприклад, AndroidDriverFactory).

Потік взаємодії у додатку починається з ViewModel, який надсилає команди до відповідного SDK у commonMain. SDK, отримавши команду (наприклад, запит на автентифікацію), делегує мережевий виклик спеціалізованому API-клієнту (наприклад, AuthApi). Після отримання відповіді від серверу, SDK обробляє її та може зберігати необхідні дані, такі як токен користувача, у локальну базу даних SQLDelight. Таким чином, ViewModel залишається ізольованим від деталей реалізації мережевих запитів та локального кешування, взаємодіючи лише з абстрактним фасадом SDK.

2.4 Прототипування та сценарії використання

Прототипування мобільного додатку було зосереджено на визначенні ключових сценаріїв взаємодії користувача (Use Cases) та проєктуванні потоків навігації між екранами. Прототип мобільного додатку наведено у додатку Б.

Основні сценарії охоплюють повний цикл використання додатку. Початковий сценарій – це автентифікація. З екрана LoginScreen користувач може ініціювати вхід, що веде до HomeScreen (рис. 2.3).

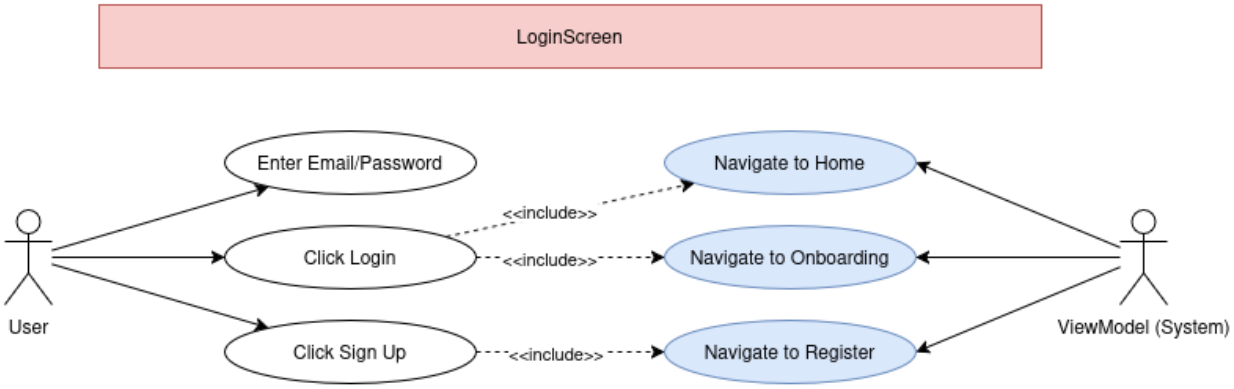


Рисунок 2.3 – Діаграма прецедентів для LoginScreen

Альтернативним сценарієм є перехід до RegisterScreen для створення нового акаунту (рис. 2.4).

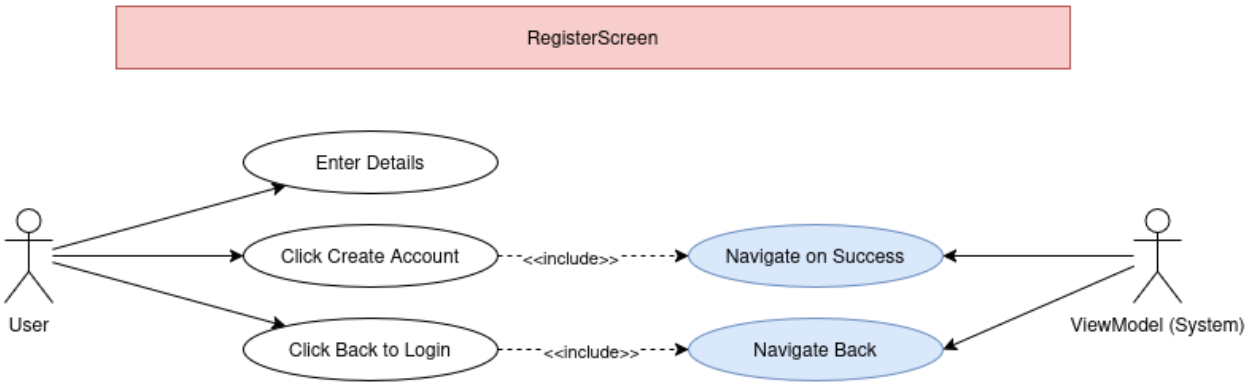


Рисунок 2.4 – Діаграма прецедентів для RegisterScreen

Після успішної реєстрації або першого входу користувач потрапляє на OnboardingScreen для початкового налаштування (рис. 2.5). Цей етап вимагає від користувача вибору рідної та цільової мов, після чого відбувається збереження налаштувань та перехід на HomeScreen. Важливість цього етапу полягає у формуванні базового профілю користувача, що є необхідною умовою для коректної роботи алгоритмів генерації завдань. Вибрані параметри передаються через ViewModel у шар даних, що дозволяє системі автоматично адаптувати контент під час усіх наступних сесій.

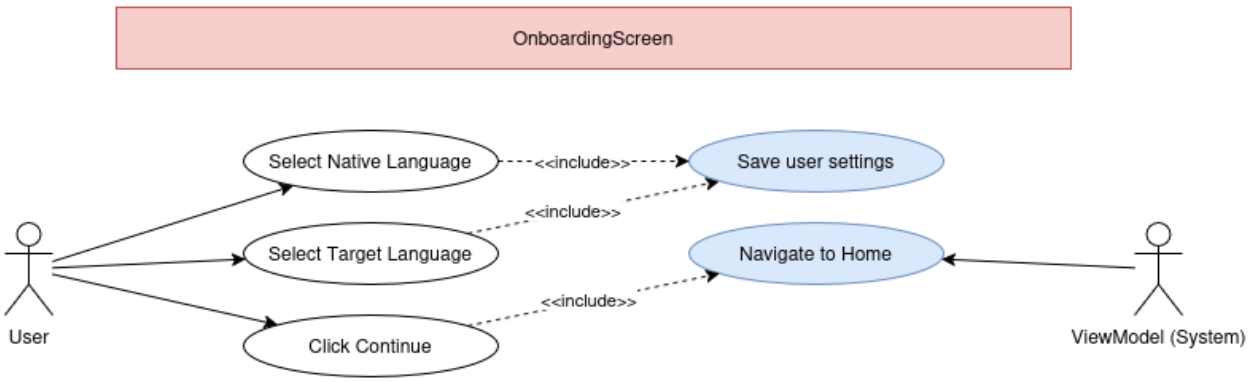


Рисунок 2.5 – Діаграма прецедентів для OnboardingScreen

HomeScreen є центральним навігаційним вузлом. Звідси користувач може ініціювати основні функції: натискання “Start Quiz Session” запускає алгоритми підбору завдань і переводить на QuizScreen для активної перевірки

знань, тоді як “Click Read” відкриває ReaderScreen для занурення у контекст. Такий дизайн мінімізує когнітивне навантаження, дозволяючи користувачеві миттєво перемикатися між вивченням нового матеріалу та його закріпленням (рис. 2.6).

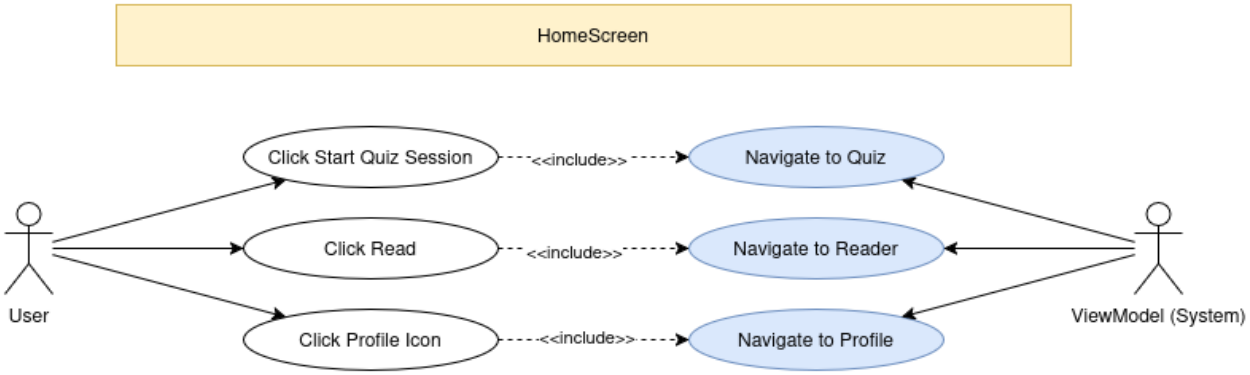


Рисунок 2.6 – Діаграма прецедентів для HomeScreen

Сценарій проходження тестування на QuizScreen реалізовано як поліморфний UI, що адаптується до структури отриманих даних. Він включає такі дії, як “Select Answer” для класичних тестів із множинним вибором або “Interact with Word Bank” для вправ на конструювання речень. Інтерфейс екрану динамічно перебудовується залежно від типу завдання (Simple, Context або Sequence), забезпечуючи інтуїтивну взаємодію для кожного механізму перевірки знань (рис. 2.7).

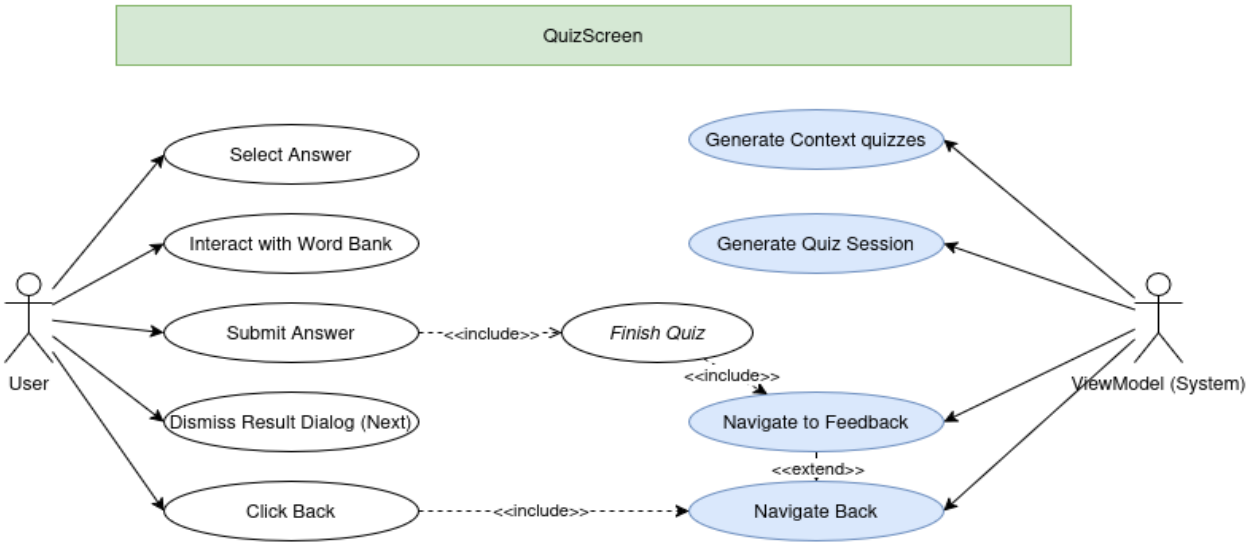


Рисунок 2.7 – Діаграма прецедентів для QuizScreen

Після завершення сесії (Finish Quiz) користувач перенаправляється на FeedbackScreen для перегляду результатів (рис. 2.8).

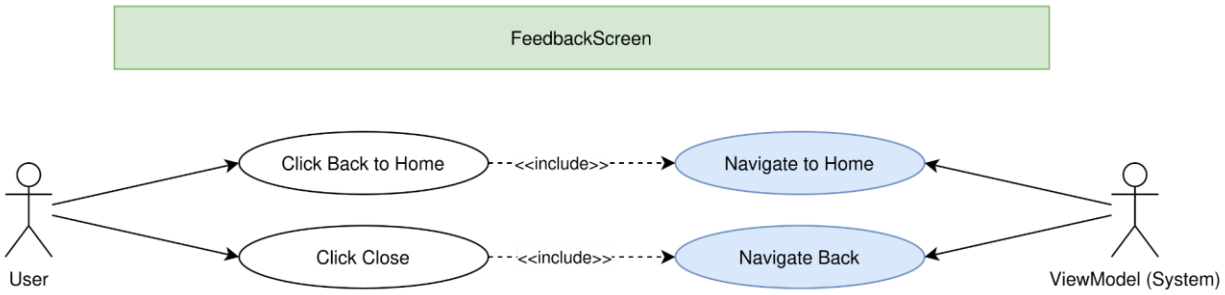


Рисунок 2.8 – Діаграма прецедентів для FeedbackScreen

Сценарій роботи з контентом починається на ReaderScreen. Окрім читання, користувач може ініціювати генерацію тесту на основі поточного тексту (Start Quiz from Text) або перейти до LibraryScreen через Click Library (рис. 2.9). Ця функція забезпечує безшовний перехід від пасивного сприйняття інформації до активної перевірки знань, автоматично використовуючи виділений фрагмент тексту як контекст для генерації персоналізованих вправ у реальному часі.

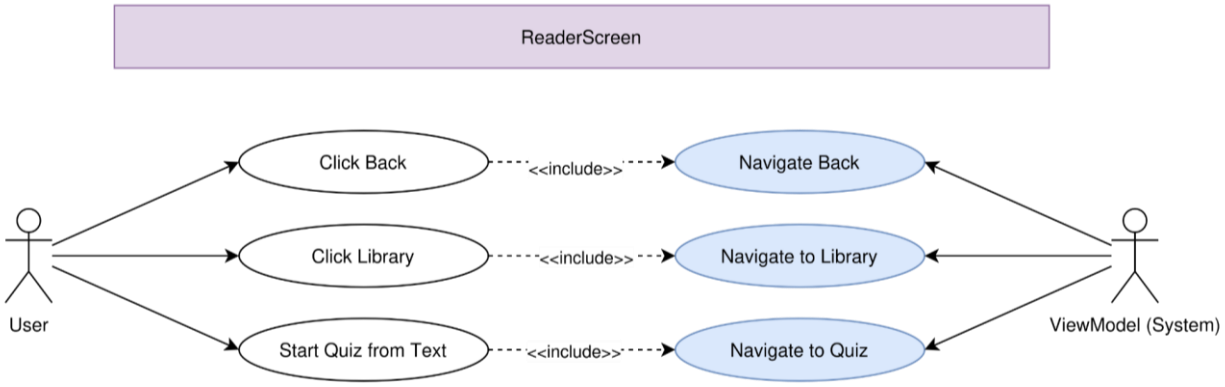


Рисунок 2.9 – Діаграма прецедентів для ReaderScreen

LibraryScreen дозволяє керувати книгами (рис. 2.10). Дія Pick File активує системну утиліту для роботи з файловим сховищем, дозволяючи користувачу обрати файл (наприклад, EPUB) для завантаження у ReaderScreen за допомогою Readium Toolkit.

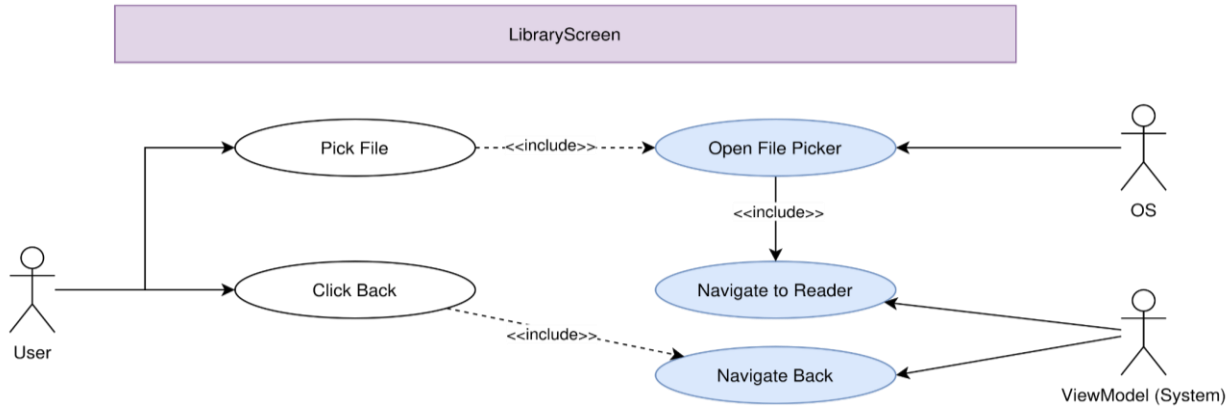


Рисунок 2.10 – Діаграма прецедентів для LibraryScreen

Усі можливі сценарії разом дозволяють побудувати загальну систему навігації додатку (рис. 2.11), яка забезпечує цілісний та інтуїтивно зрозумілий користувацький досвід. Пізніше, цей граф навігації буде використаний під час реалізації додатку. Ця схема стане основою для налаштування типізованої маршрутизації, та дозволить зменшити кількість помилок пов’язаних з налаштуванням маршрутів між екранами.

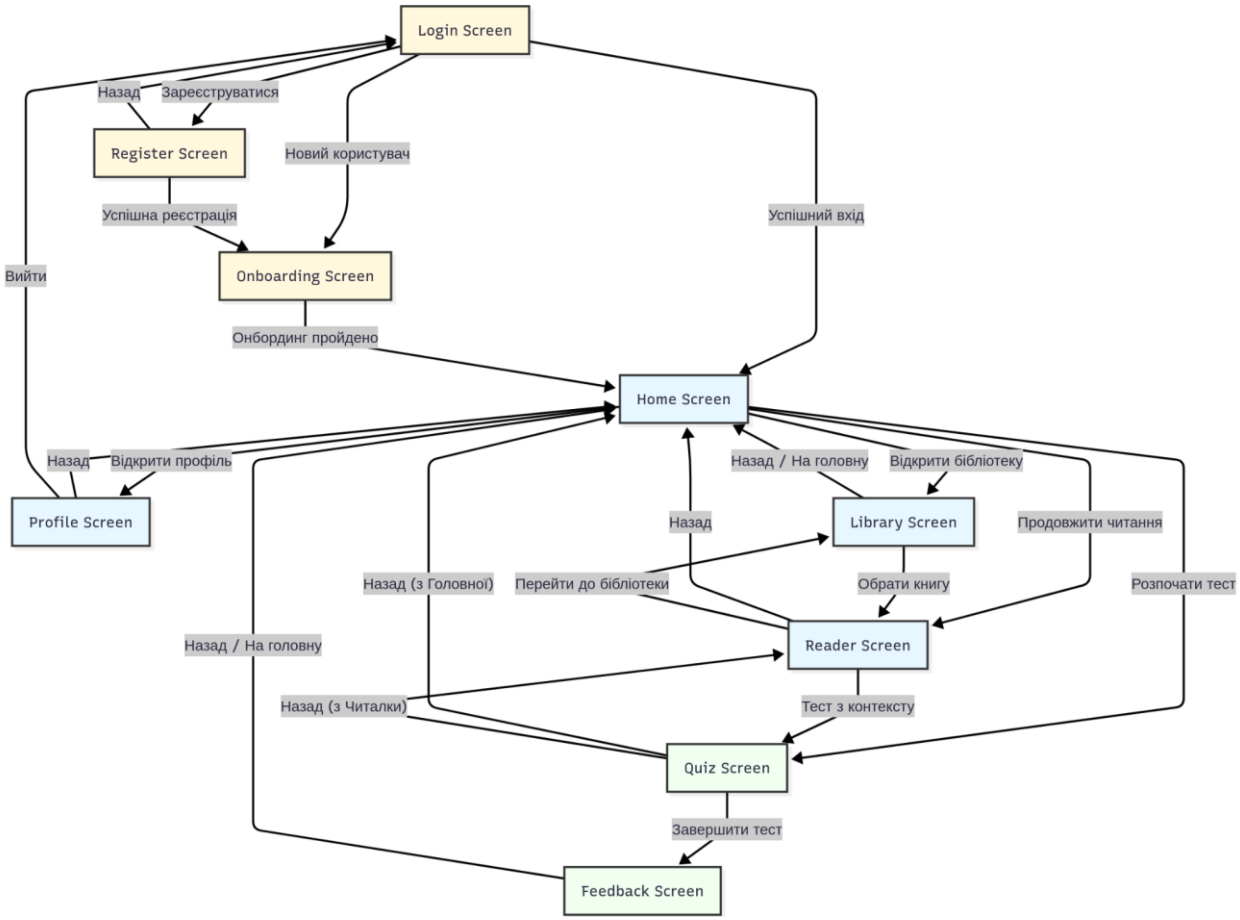


Рисунок 2.11 – Діаграма навігації мобільного додатку

3 РОЗРОБКА ТА ТЕСТУВАННЯ ПРОГРАМНОГО КОМПЛЕКСУ

3.1 Реалізація серверної частини

Реалізація серверної частини виконана з використанням фреймворку Python FastAPI. Архітектура проєкту дотримується чіткого модульного підходу, де головний додаток поєднує маршрути з окремих модулів (рис. 3.1). Ця декомпозиція включає auth.py для процесів автентифікації та реєстрації, quizzes.py для ендпоінтів, що відповідають за генерацію завдань та user_settings.py для управління налаштуваннями користувача.

```
from fastapi import Depends, FastAPI
from app.api.routers import data, quizzes, auth, user_settings
from app.db import models
from app.db.database import engine
models.Base.metadata.create_all(bind=engine)
app = FastAPI()
app.include_router(quizzes.router)
app.include_router(data.router)
app.include_router(auth.router)
app.include_router(user_settings.router)
```

Рисунок 3.1 – Точка входу головної програми API

Взаємодія з базою даних PostgreSQL реалізована за допомогою SQLAlchemy ORM (рис. 3.2). На рисунку 3.2 продемонстровано декларативний опис моделі UserSettings, де атрибути класу безпосередньо відображаються на колонки таблиці. Такий підхід дозволяє маніпулювати даними як звичайними об’єктами Python, автоматизуючи серіалізацію та роботу із зовнішніми ключами.

```
class UserSettings(Base):
        __tablename__ = "user_settings"
        id = Column(Integer, primary_key=True, index=True)
        native_language_code = Column(String(10), nullable=False)
        target_language_code = Column(String(10), nullable=False)
        user_id = Column(Integer, ForeignKey("users.id",
ondelete="CASCADE"), nullable=False, unique=True)
        @hybrid_property
        def user_email(self):
          if self.owner:
            return self.owner.email
          return None
        owner = relationship("User", back_populates="settings")
```

Рисунок 3.2 – Приклад використання SQLAlchemy

Для валідації даних, що надходять до API та повертаються з нього, використовуються DTO (Data Transfer Objects), що створені за допомогою Pydantic (рис. 3.3).

```
from pydantic import BaseModel, EmailStr
class UserSettingsDTO(BaseModel):
  user_id: int
  user_email: EmailStr
  native_language_code: str
  target_language_code: str
```

Рисунок 3.3 – Приклад використання Pydantic

Логіка виконання операцій CRUD (Create, Read, Update, Delete) виокремлена у спеціалізовані модулі (наприклад, user_settings_crud.py та text_crud.py), що взаємодіють з сесією бази даних, яка надається через залежності FastAPI (див. рис. 3.4).

```
    from sqlalchemy.orm import Session, joinedload
    from app.db.models import UserSettings
    from app.db.schemas import UserSettingsCreate

    def get_user_settings(db: Session, user_id: int) -> UserSettings | None:
        return
db.query(UserSettings).options(joinedload(UserSettings.owner)).filter(UserSettin
gs.user_id == user_id).first()
    …
```

Рисунок 3.4 – Приклад CRUD функції, що потребує ін’єкції залежностей

Безпека системи забезпечується на кількох рівнях. Для зберігання паролів користувачів використовується хешування bcrypt, реалізоване через passlib. Процес автентифікації базується на JSON Web Tokens (JWT). Система генерує access та refresh токени, що підписані секретним ключем, використовуючи функції create_access_token та create_refresh_token. Для захисту ендпоінтів FastAPI використовує механізм впровадження залежностей. Ключова залежність get_current_user_or_api_key вимагає наявності заголовка Authorization та валідує його або як Bearer токен (JWT), або як статичний API-ключ, забезпечуючи гнучкі методи доступу до API (рис. 3.5).

```
    from app.service.auth.dependencies import get_current_user_or_api_key
    …
    router = APIRouter(prefix="/api/settings", tags=["settings"],
dependencies=[Depends(get_current_user_or_api_key)])

    @router.get( "/", response_model=UserSettingsDTO, summary="Get
current user's settings")
    def read_user_settings(db: Session = Depends(get_db), current_user:
UserModel = Depends(get_current_user_or_api_key)):
        …
```

Рисунок 3.5 – Приклад використання залежності get_current_user_or_api_key

Цей механізм дозволяє абстрагувати логіку перевірки прав доступу від основної бізнес-логіки. Функція-обробник отримує вже ідентифікований об’єкт користувача, що спрощує код і запобігає дублюванню перевірок безпеки в кожному методі.

Додаток запускається за допомогою WSGI-сервера Gunicorn. Управління конфігураціями та чутливими даними, зокрема обліковими даними для підключення до бази даних, реалізовано через інтеграцію з AWS Secrets Manager. Модуль aws_secrets використовує бібліотеку Boto3 для безпечного отримання секретів під час запуску додатку, кешуючи результат для мінімізації запитів до AWS.

3.2 Тестування серверної частини

Тестування серверної частини проводилося з використанням фреймворку Pytest у поєднанні з TestClient FastAPI. Ключовим принципом тестування стала повна ізоляція компонентів для забезпечення швидких, відтворюваних та незалежних тестів. Для досягнення цієї ізоляції було застосовано механізм впровадження залежностей FastAPI для заміщення двох критичних компонентів: бази даних та автентифікації.

Для систематизації процесу верифікації всіх розроблених компонентів системи було сформовано план тестування, який наведено у таблиці 3.1.

Таблиця 3.1 – План тестування серверної частини

Категорія тестування	**Призначення**
Автентифікація та безпека	Перевірка, що захищені ендпоінти дійсно вимагають автентифікації та повертають помилку (401/403) у разі її відсутності.
CRUD-операції (/api/settings/)	Повний цикл роботи з налаштуваннями користувача: Створення (POST), Читання (GET), Оновлення (POST), Обробка помилки (404, якщо налаштувань немає).

Продовження табл. 3.1

Категорія тестування	**Призначення**
Локальні NLP-стратегії (/simple/, /sequence/)	Перевірка API: чи повертають ці ендпоінти коректну структуру JSON та дотримуються запитаного ліміту.
Комбінована NLP-стратегія (/session/)	Перевірка, що ендпоінт для сесії коректно обробляє вхідний список речень та повертає комбінований набір завдань.
LLM-стратегії (/context/)	Тестування логіки роутера (з використанням моків): чи коректно обробляється успішна відповідь від LLM, яи повертає API помилку 400, якщо LLM не зміг згенерувати завдання.

Для кожного тестового запуска створювалася ізольована база даних SQLite в оперативній пам’яті. Спеціальна фікстура pytest (db_session) відповідала за створення повної схеми SQLAlchemy перед кожним тестом та її повне видалення після завершення. Потім клієнт TestClient отримував цю сесію через заміщення залежності get_db, гарантуючи, що жоден тест не взаємодіяв з основною базою даних PostgreSQL.

Аналогічним чином була ізольована система автентифікації. Замість валідації реальних JWT-токенів, залежність get_current_user_or_api_key була заміщена фікстурою mock_auth, яка просто повертала попередньо створений у тестовій базі диних об’єкт test_user. Це дозволило тестувати логіку захищених ендпоінтів, не турбуючись про процес генерації токенів.

Тести для ендпоінтів CRUD (наприклад, /api/settings/) валідували повний цикл «створення-читання-оновлення» (Create-Read-Update), використовуючи db_session для початкового заповнення даних та client для виконання HTTP-запитів.

Окрему увагу було приділено тестуванню стратегій генерації завдань у test_quizzes.py. Локальні NLP-стратегії (simple, sequence) тестувалися безпосередньо, перевіряючи, що їхні ендпоінти повертають коректну структуру JSON та дотримуються вказаних лімітів. Стратегії, що залежать від зовнішніх LLM (ContextQuizStrategyLLM), реалізовані за допомогою моків

(mock) за допомогою unittest.mock.patch. Тести перевіряли не логіку LLM, а коректність обробки роутером доменних об’єктів (ContextQuiz), які повертала мок стратегія, а також валідацію обробки помилок, наприклад, повернення 400 Bad Request, якщо стратегія не змогла згенерувати завдання.

Тести з плану тестування було реалізовано та виконано (див. рис. 3.6). На результатах тестування бачимо, що усі тести були успішні. Зауваження, що були надані бібліотекою для тестування, зосереджені на реалізації коду використаних бібліотек, а не серверної частини.

```
============================= test session starts ==============================
platform linux -- Python 3.12.3, pytest-9.0.0, pluggy-1.6.0
rootdir: /mnt/Storage/dev/lexiloop-backend
plugins: anyio-4.9.0
collected 10 items

tests/routers/test_quizzes.py .....                                      [ 50%]
tests/routers/test_settings.py .....                                     [100%]

============================== warnings summary ===============================
.venv/lib/python3.12/site-packages/starlette/formparsers.py:12
  /mnt/Storage/dev/lexiloop-backend/.venv/lib/python3.12/site-packages/starlette/formparsers.py:12: PendingDeprecationWarning: Please use `import python_m
ultipart` instead.
    import multipart

.venv/lib/python3.12/site-packages/passlib/utils/__init__.py:854
  /mnt/Storage/dev/lexiloop-backend/.venv/lib/python3.12/site-packages/passlib/utils/__init__.py:854: DeprecationWarning: 'crypt' is deprecated and slatec
 for removal in Python 3.13
    from crypt import crypt as _crypt

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
======================== 10 passed, 2 warnings in 0.38s ========================
```

Рисунок 3.6 – Результат виконання тестування API

3.3 Реалізація мобільного додатку

Реалізація мобільного додатку зосереджена у двох ключових модулях: commonMain для спільної логіки та androidMain для специфічної реалізації платформи Android (рис. 3.7). Ядро бізнес-логіки інкапсульовано у фасадах (SDK), визначених у commonMain. Наприклад, QuizSDK отримує через впровадження залежностей екземпляри QuizApi та DatabaseDriverFactory. Його функції, такі як generateContextQuiz, делегують мережевий виклик до QuizApi. Отримана відповідь, обгорнута в Result<D, NetworkError>, обробляється: у випадку успіху (Result.Success), мережевий QuizDTO перетворюється на локальну сутність CompleteQuizEntity за допомогою логіки з QuizMapper.kt для подальшої обробки у ViewModel.

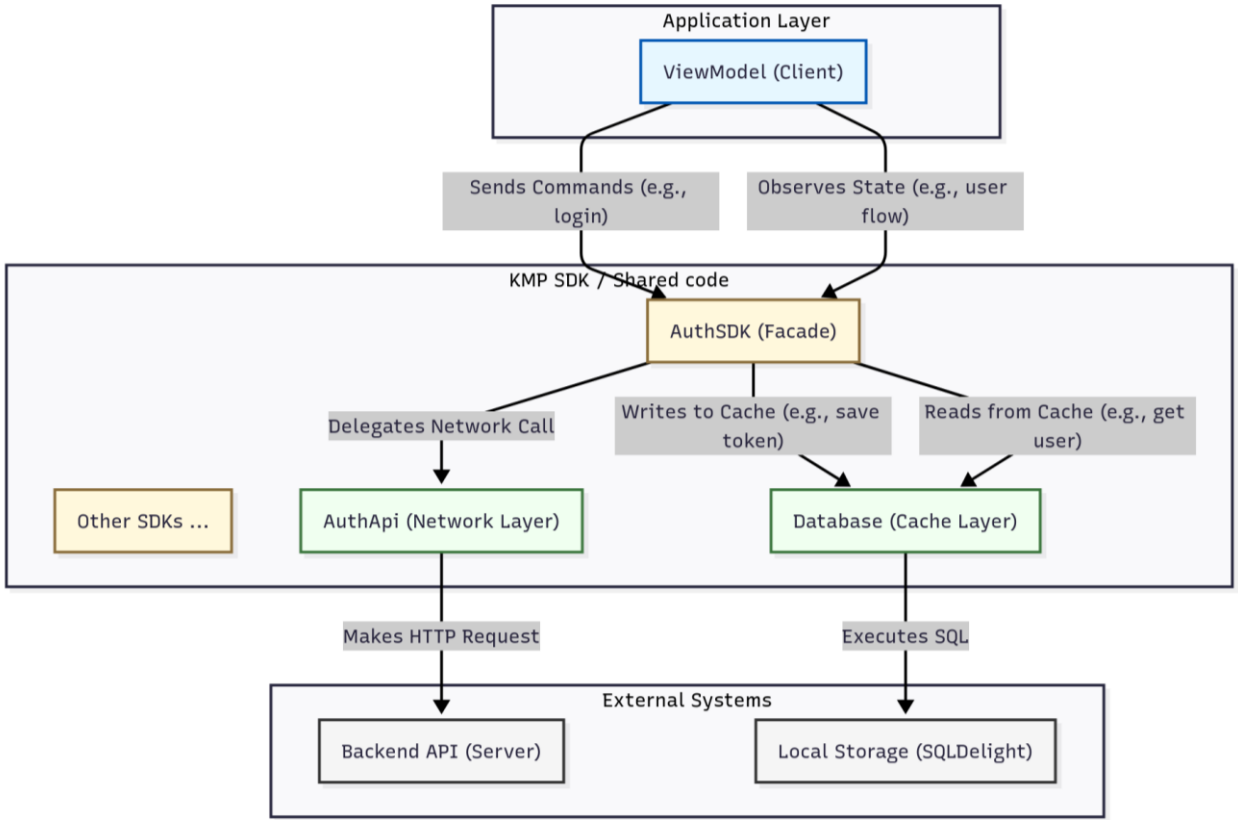


Рисунок 3.7 – Приклад загальної архітектури KMP-додатку

Result<D, NetworkError> – це запечатаний (sealed) інтерфейс, що використовується в commonMain для обробки результатів операцій, які можуть завершитися невдало, зокрема мережевих запитів (рис. 3.8). Цей підхід є альтернативою традиційній обробці винятків і дозволяє явно моделювати два можливі стани: успіх або помилку.

```
sealed interface Result<out D, out E: BaseError> {
    data class Success<out D>(val data: D): Result<D, Nothing>
    data class Error<out E: BaseError>(val error: E): Result<Nothing, E>
}
```

Рисунок 3.8 – Приклад інтерфейсу Result

Інтерфейс це загальний тип, що приймає два параметри: D (Data) для типу успішних даних та E (Error), який обмежений базовим інтерфейсом BaseError. Така архітектура дозволяє перетворити помилки на об’єкти, що є частиною сигнатури методу, забезпечуючи передбачувану поведінку коду на

всіх цільових платформах без ризику неочікуваного приринення роботи програми.

Мережевий шар реалізовано за допомогою Ktor Client. Ключовим елементом є createAuthenticatedHttpClient, який налаштовує Ktor Auth плагін (рис. 3.9).

```
refreshTokens {
        val oldRefreshToken = oldTokens?.refreshToken
        if (oldRefreshToken == null) {
            authSdk.logout()
            null
        } else {
            val response = unauthenticatedClient.post(...) { ... }
            ...
    }
}
```

Рисунок 3.9 – Конфігурація Ktor Auth плагіну

Цей плагін автоматично перехоплює 401 відповіді (Unauthorized) та оновлює токени, що запобігає рекурсивним помилкам автентифікації. У разі успішного оновлення нові облікові дані зберігаються у локальному сховищі через authSdk. Якщо ж оновлення неможливе, система ініціює процедуру виходу користувача, гарантуючи цілісність сесії.

Усі компоненти системи пов’язані між собою за допомогою Koin. Модуль впровадження залежностей надає два різні типи HttpClient, прив’язує платформенний AndroidDatabaseDriverFactory до спільного інтерфейсу та визначає single (для SDK) і viewModel (для ViewModel) залежності (рис. 3.10).

Локальне кешування реалізовано за допомогою SQLDelight. Файл QuizDatabase.sq у commonMain визначає схему та SQL-запити. Зв’язок з платформою забезпечується через інтерфейс DatabaseDriverFactory, який отримує конкретну реалізацію AndroidDatabaseDriverFactory у androidMain. Ця архітектура дозволяє писати SQL-запити, що перевіряються під час компіляції,

і автоматично генерувати з них типобезпечні Kotlin-інтерфейси для доступу до даних.

```
val appModule = module {
   single<HttpClient>(named("unauthenticatedClient")) {
      createHttpClient(OkHttp.create())
   }
   single<HttpClient>(named("authenticatedClient")) {
      createAuthenticatedHttpClient(..., unauthenticatedClient =
get(named("unauthenticatedClient")))
   }
   single<DatabaseDriverFactory> {
AndroidDatabaseDriverFactory(androidContext()) }
   single<AuthSDK> { AuthSDK(...) }
   viewModel { HomeScreenViewModel(...) }
   viewModel { (handle: SavedStateHandle) -> QuizScreenViewModel(...,
savedStateHandle = handle) }
}
```

Рисунок 3.10 – Приклад налаштування Koin-модуля

Рівень презентації (UI) побудовано на Jetpack Compose. Розглянемо реалізацію інтерфейсу користувача на прикладі одного з екранів додатку (рис. 3.11). На рисунку наведено приклад використання Jetpack Compose. Composable-функція QuizScreen отримує ViewModel через koinViewModel() та підписується на зміни стану за допомогою collectAsState(). QuizScreenViewModel управляє станом екрану (QuizScreenState) через MutableStateFlow.

```
@Composable
fun QuizScreen(viewModel: QuizScreenViewModel = koinViewModel(), ...) {
   val state by viewModel.state.collectAsState()
   ...
   Box(...) {
      if (state.isLoading) {
```

```
            CircularProgressIndicator()
        } else if (state.currentQuiz == null) {
            Text("Failed to load quiz")
        } else {
            when (state.currentQuiz!!.quiz.type) {
                "simple" -> SimpleQuizContent(...)
                "context" -> ContextQuizContent(...)
                "sequence" -> SequenceQuizComponent(...)
            }
        }
    }
}
```

Рисунок 3.11 – Декларативне відображення UI у Compose

Реалізація функціональності читання EPUB-книг та генерації контекстних завдань спирається на інтеграцію з бібліотекою Readium Toolkit (рис. 3.12).

```
private fun loadPublication(bookUriString: String) {
    viewModelScope.launch {
        _state.update { it.copy(isLoading = true) }
        val asset = assetRetriever.retrieve(absoluteAssetUrl).getOrElse { ... }
        val publication = publicationOpener.open(asset, allowUserInteraction =
true)
            .getOrElse { ... }
        ...
        _state.update {
            it.copy(isLoading = false, publication = publication)
        }
    }
}
```

Рисунок 3.12 – Завантаження EPUB у ViewModel

Readium Toolkit використовує AssetRetriever та PublicationOpener (надані через Koin) для відкриття файлу та отримання об’єкта Publication, який зберігається у стані ViewModel. Для відображення книги ReaderScreen

використовує AndroidView, щоб інтегрувати EpubNavigatorFragment у Compose-ієрархію. Такий підхід є необхідним, оскільки Readium базується на класичній системі View/Fragment, тоді як решта додатку побудована на сучасному декларативному UI.

Для відстеження прогресу читання ViewModel підписується на потік fragment.currentLocator. При кожній зміні локації викликається onLocationChanged (рис. 3.13), яка зберігає поточний Locator у базу даних SQLDelight.

```
fun onLocationChanged(locator: Locator) {
   this.lastKnownGoodLocator = locator
   viewModelScope.launch {
      bookSdk.saveProgress(bookId, currentUserEmail,
locator.toJSON().toString())
      processAndSaveReadText(locator)
   }
}

private fun parseSentences(xhtml: String): List<String> {
   val cleanText = xhtml
      .replace(Regex("<[^>]*>"), " ")
      .replace(Regex("\\s+"), " ")
      .trim()

   return cleanText.split(Regex("[.!?]+\\s*"))
      .filter { it.isNotBlank() && it.length > 5 && it.contains(" ") }
}
```

Рисунок 3.13 – Збереження прогресу та парсинг речень

Функція, що зображена на рисунку 3.13, також запускає processAndSaveReadText, що реалізує логіку «банку речень»: метод завантажує XHTML-контент поточного розділу, очищує його від HTML-тегів та розбиває на речення, зберігаючи нові у базу даних.

Ключовою функцією є генерація тестів з виділеного тексту. Це реалізовано через ReadiumSelectionCallback (рис. 3.14). Цей колбек додає пункт “Start Quiz” до стандартного меню виділення тексту. При його активації асинхронно отримується виділений фрагмент, який передається через лямбду onStartQuiz для ініціації переходу на QuizScreen із відповідним контекстом.

```
class ReadiumSelectionCallback(
    private val scope: CoroutineScope,
    private val onTextSelected: (String) -> Unit
) : ActionMode.Callback {
    private val quizMenuItemId = Menu.FIRST
    override fun onCreateActionMode(mode: ActionMode, menu: Menu):
Boolean {
        menu.add(Menu.NONE, quizMenuItemId, Menu.NONE, "Start Quiz")
        return true
    }
    override fun onActionItemClicked(mode: ActionMode, item: MenuItem):
Boolean {
        if (item.itemId == quizMenuItemId) {
            scope.launch {
                val text = navigator?.currentSelection()?.locator?.text?.highlight
                if (!text.isNullOrBlank()) {
                    onTextSelected(text)
                }
                mode.finish()
            }
            return true
        }
        return false
    }
    ...
}
```

Рисунок 3.14 – Обробка виділення тексту в Readium

Навігація в додатку реалізована за допомогою бібліотеки Navigation компоненту, що розроблений спеціально для Jetpack Compose. Для цього

використовуються та використовуються серіалізовані об’єкти (Routes.kt) для типізованих переходів між екранами. Візуальні приклади роботи реалізованого інтерфейсу користувача, що демонструють описані сценарії та логіку адаптації UI, наведено у Додатку В.

3.4 Тестування мобільного додатку

Тестування мобільного додатку зосереджено на валідації бізнес-логіки у спільному модулі commonMain. Для ізоляції тестів зовнішні залежності (API, база даних) заміщено моками за допомогою бібліотеки MockK. Асинхронний код перевірявся через runTest, а поведінка моків налаштовувалася за допомогою coEvery, що дозволило симулювати як успішні відповіді (Result.Success), так і помилки (Result.Error).

Для тестування асинхронного коду використовувалася runTest. Поведінка моків налаштовувалася через coEvery (для suspend-функцій), імітуючи як успішні відповіді (наприклад, Result.Success), так і помилки (наприклад, Result.Error) . Після виконання тестованої функції, verify та coVerify використовувалися для підтвердження, що очікувані методи (наприклад, databaseMock.insertTokenEntity() або databaseMock.removeAllUsers()) були викликані, або, у випадку помилки, не викликані взагалі.

Особливу увагу приділено мережевому шару. За допомогою MockEngine від Ktor було протестовано механізм автоматичного оновлення токенів у createAuthenticatedHttpClient, зокрема сценарії перехоплення 401 помилки, успішного оновлення токена та повторного виконання запиту.

Розроблений план тестування (табл. 3.2) охоплює перевірку всіх ключових фасадів системи: від автентифікації та роботи з локальним кешем до логіки «local-first» налаштувань.

Таблиця 3.2 – План тестування мобільного додатку

Категорія тестування	Призначення
Мережевий клієнт (Ktor)	Перевірка логіки автоматичного оновлення токенів на рівні HTTP-клієнта: автоматичне оновлення токену при 401 помилці та успішний повторний запит, автоматичний logout у разі, якщо оновлення токену також завершується помилкою.
Автентифікація (AuthSDK)	Тестування повного життєвого циклу користувача та стану бази даних: успішний вхід, невдалий вхід (API), виклик logout коректно очищує всі токени та дані користувача з БД.
«Local-first» логіка (UserSettingsSDK)	Валідація «local-first» стратегії для налаштувань користувача: якщо дані є в локальній БД, API не викликається, якщо в БД даних немає, SDK робить запит до API, а потім зберігає результат у БД, виклик saveSettings оновлює дані і через API, і в локальній БД.
Робота з БД (BookSDK)	Перевірка коректності роботи SDK з локальною базою даних (без викликів API), збереження та читання прогресу читання, робота «банку речень» (збереження та отримання речень для тестування), оновлення статусу речень (markSentencesAsQuizzed).
Проксі до API (QuizSDK)	Перевірка того, що SDK коректно передає запити до API та повертає результат: успішне отримання завдань, коректна обробка помилок, якщо API повертає Error.

Усі 24 unit-тести, що покривають модулі AuthSDK, BookSDK, QuizSDK та UserSettingsSDK, а також мережевий клієнт, були успішно виконані (рис. 3.15).

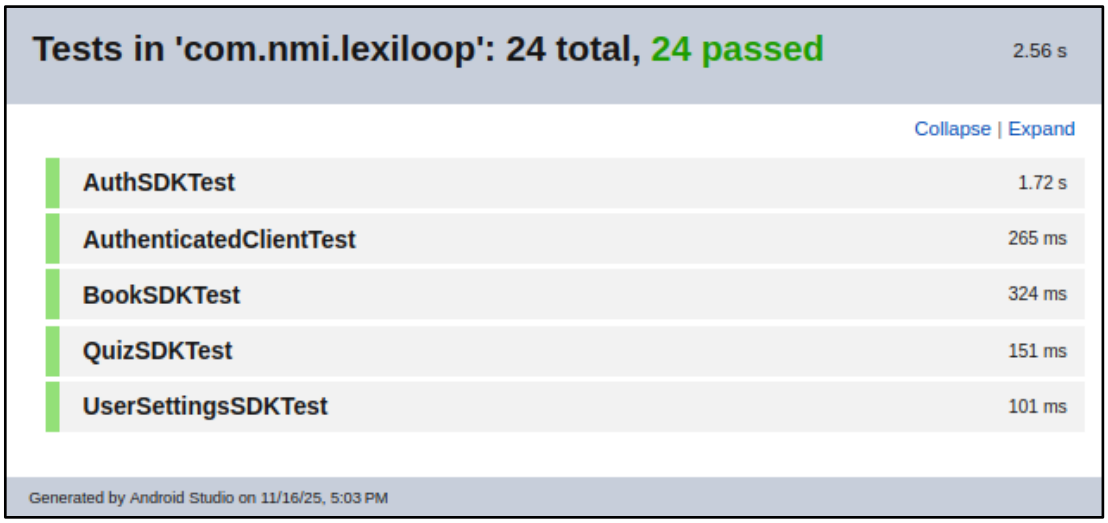


Рисунок 3.15 – Результат тестування мобільного додатку

3.5 Розгортання та конфігурація інфраструктури AWS

З метою гарантування високого рівня масштабованості, відмовостійкості та захисту даних, серверну частину системи розгорнуто на хмарній платформі Amazon Web Services (AWS). Архітектурне рішення базується на сучасних підходах контейнеризації та використанні безсерверних обчислювальних потужностей (рис. 3.16).

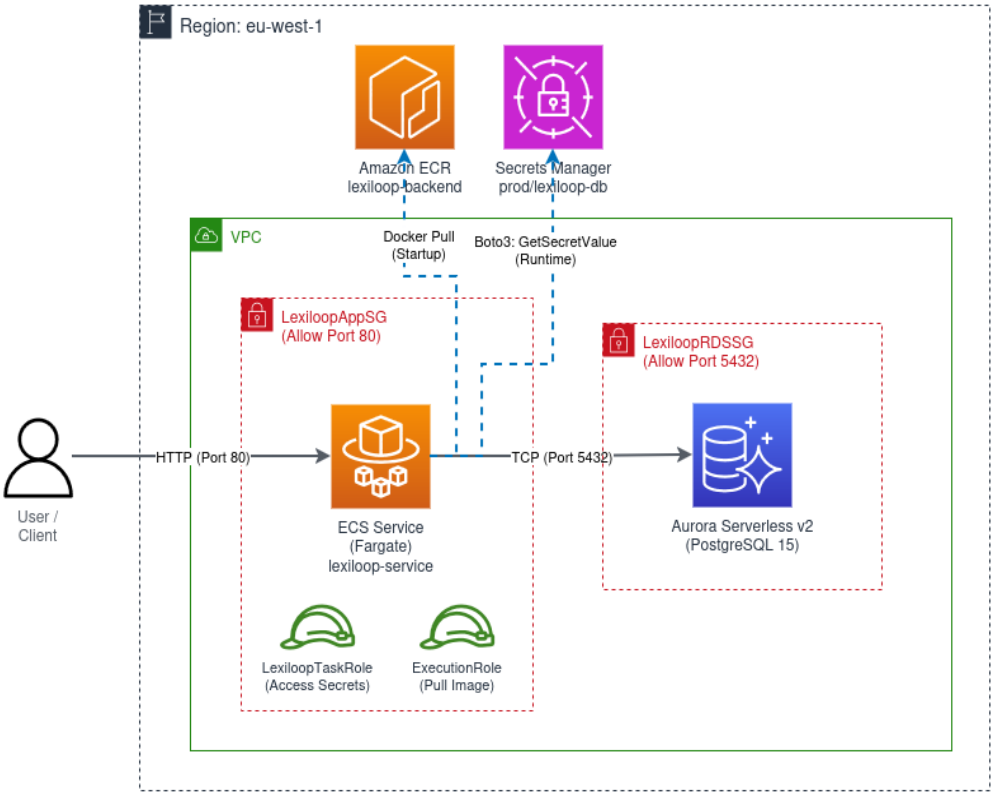


Рисунок 3.16 – Мінімальна AWS-архітектура

В якості середовища виконання обрано AWS Fargate. AWS Fargate – компонент сервісу оркестрації Amazon ECS. Ця технологія дозволяє абстрагуватися від управління фізичною інфраструктурою та віртуальними машинами, фокусуючись безпосередньо на логіці додатку. Використання Fargate усуває необхідність налаштування, патчингу та масштабування кластерів серверів, що значно знижує операційне. Процес розгортання включає збірку образу Docker-контейнера та його збереження у приватному реєстрі Amazon ECR.

База даних реалізована за допомогою Amazon Aurora Serverless. Цей сервіс автоматично адаптує обчислювальні ресурси під поточне навантаження. Мережева безпека забезпечується через конфігурацію груп безпеки: база даних ізольована у приватній мережі та приймає з'єднання виключно від авторизованих контейнерів застосунку.

Управління конфіденційними даними здійснюється через AWS Secrets Manager. Додаток отримує секрети динамічно за допомогою Boto3 SDK. Важливим аспектом безпеки є дотримання принципу найменших привілеїв (Least Privilege Principle), який є фундаментальним для екосистеми AWS. Реалізація цього принципу досягається через IAM-ролі, що надають контейнерам лише ті дозволи, які критично необхідні для їх функціонування.

Поточна реалізація забезпечує базову працездатність системи. Однак, відповідно до кращих практик (AWS Well-Architected Framework), доречним буде впровадження Application Load Balancer (рис. 3.17). Це дозволить створити єдину статичну точку входу, забезпечити балансування навантаження між кількома екземплярами контейнерів та реалізувати безшовне оновлення сервісу без переривання обслуговування користувачів.

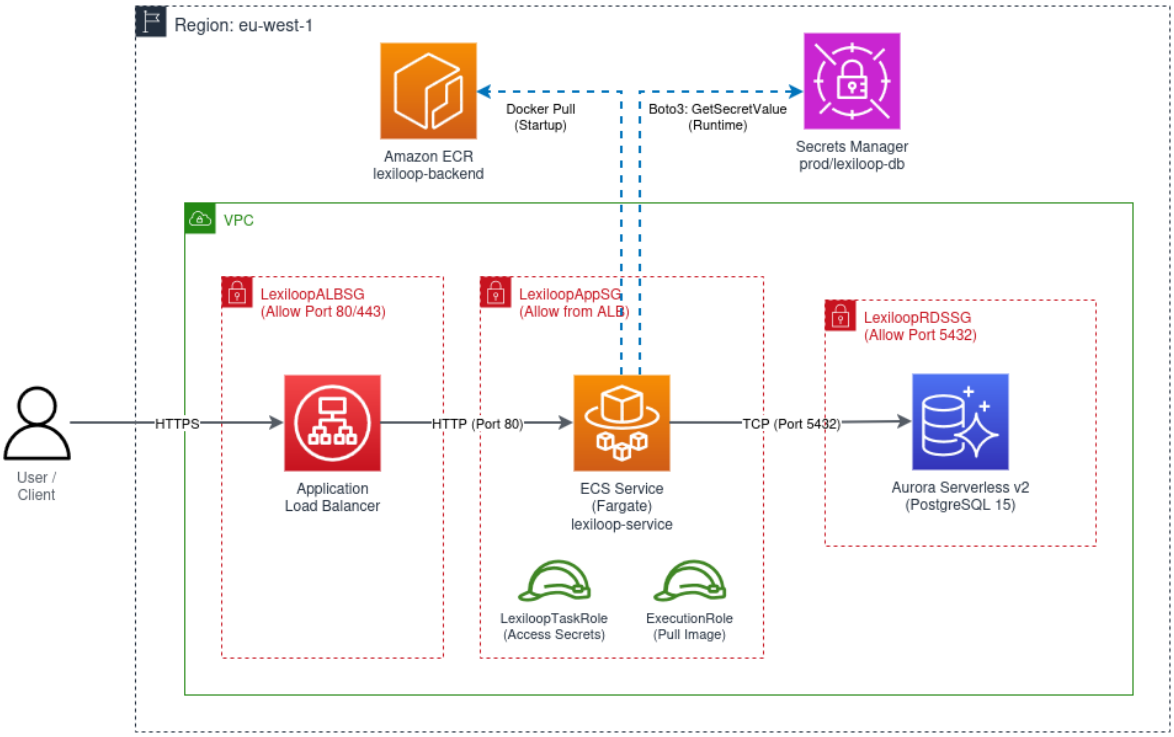


Рисунок 3.17 – Вдосконалена AWS-архітектура

3.6 Оцінка згенерованих завдань

Для оцінки якості згенерованих завдань було сформовано дві тестові вибірки: одна з використанням класичних NLP-підходів, інша за допомогою LLM. Посилання на приклади згенерованих завдань наведено у Додатку А.

Перша партія, згенерована за допомогою простого підходу NLP, досягла загального балу продуктивності 3,60 (див. табл. 3.3). Продуктивність цієї системи визначалася її надійністю. Вона набрала 5,00 балів за граматичну плинність, відповідальність та релевантність, що свідчить про те, що кожен згенерований тест був ідеально плавним, мав одну однозначну правильну відповідь та був релевантним до вихідного тексту. Однак на цьому її сильні сторони закінчуються. Система має значний недолік, що полягає у повній відсутності різноманітності. Це демонструє генератор, який створює лише низькорівневі, повторювані вправи на заповнення пропусків, витягуючи фрази безпосередньо з вихідного тексту.

Таблиця 3.3 – Результати оцінки якості згенерованих завдань (NLP)

GFavg	Avg	Davg	Cavg	Ravg	OPS
5.00	5.00	1.00	2.00	5.00	3.60

Друга партія використовувала більш просунутий підхід і досягла вищого OPS (див. табл. 3.4). Ця система була набагато амбітнішою, намагаючись перевірити складні граматичні правила, що принесло їй набагато кращі бали за різноманітність та складність.

На відміну від попереднього методу, цей підхід дозволив створювати завдання, які вимагають від користувача аналізу контексту та розуміння нюансів мови, а не простого запам’ятовування тексту. Генерація нових синтаксичних конструкцій забезпечила варіативність вправ, що значно підвищує їхню педагогічну цінність та стійкість до вгадування.

Таблиця 3.4 – Результати оцінки якості згенерованих завдань (LLM)

GFavg	Avg	Davg	Cavg	Ravg	OPS
4.59	3.76	3.00	3.71	5.00	4.01

Однак головним недоліком другої системи була надійність. Вона отримала найнижчий бал за відповідальність, оскільки надавала неправильні граматичні пояснення, які суперечили її власним відповідям. Вона також створила кілька неграматичних питань, що знизило її граматичну плинність. Цей феномен, відомий як «галюцинації» штучного інтелекту, є особливо критичним у контексті освітніх застосунків, де точність навчального матеріалу має бути пріоритетною. Відсутність суворого контролю призвела до того, що система генерувала лінгвістично некоректні конструкції, які могли б ввести студента в оману. Варто зосередити увагу на кількох згенерованих завданнях (див. табл. 3.5).

Таблиця 3.5 – Приклад невдалих завдань

Запитання	**Згенероване питання**	**Правильна відповідь**	**Додаткова інформація від системи**
1	How slowly the time _ here...	passes	“explanation”: “У цьому реченні використовується теперішній тривалий час для вираження дії, яка відбувається в даний момент.”
2	I _ a vessel for my journey.	have hired	“identified_grammar”: “Present Perfect Continuous”
7	There will be none to _ my joy.	participate	“explanation”: “Фразове дієслово 'participate in' означає 'брати участь у чомусь', тому правильний варіант – 'participate'.”

Розглянемо приклади з таблиці 3.5 детальніше. У поясненні системи для завдання 1 неправильно зазначено, що в реченні використовується час “Present Continuous”, але правильна відповідь “Present Simple”. У завданні 2 тег “identified_grammar” для тесту: “Present Perfect Continuous”, але правильна

відповідь (“have hired”), що відповідає часовій формі “Present Perfect”. У завданні 3 пояснення системи правильно ідентифікує сучасне фразове дієслово “participate in”, але сама основа питання пропускає необхідний прийменник “in”, що робить питання хибним. Хоча педагогічно ця система була більш досконалою, її недосконале виконання зробило її менш надійною.

ВИСНОВКИ

Таким чином, під час виконання кваліфікаційної роботи було досліджено та обґрунтовано гібридну методологію автоматичної генерації навчальних завдань, що поєднує класичні методи обробки природної мови (NLP) та великі мовні моделі (LLM). Було розроблено серверний API та мобільний додаток на базі Kotlin Multiplatform, що реалізують цей підхід.

У першому розділі було проведено порівняльний аналіз класичних підходів NLP та великих мовних моделей для автоматичної генерації питань. Досліджено переваги та недоліки кожного методу, зокрема високу вартість та екологічний вплив LLM. Також було проаналізовано існуючі аналогічні рішення та визначено метрики для подальшої оцінки якості згенерованих завдань.

У другому розділі було спроєктовано системну архітектуру. Для серверної частини обрано Python, FastAPI. Для мобільного додатку обрано технологію Kotlin Multiplatform з декларативним UI.

У третьому розділі реалізовано та протестовано (Pytest, MockK) обидві частини системи, включаючи розгортання API на AWS та інтеграцію Readium в KMP. Оцінка якості підтвердила компроміс: NLP-модель (OPS=3.60) є надійною, але простою (C=2.00), тоді як LLM-модель (OPS=4.01) має вищий педагогічний потенціал (C=3.71), але містить помилки у виконанні (A=3.76).

В результаті роботи отримано програмний комплекс, що реалізує поставлену мету.

ПЕРЕЛІК ПОСИЛАНЬ

1. Mariyono D., Nur Alif Hd A. AI’s role in transforming learning environments: a review of collaborative approaches and innovations. *Quality education for all*. 2025. Vol. 2, no. 1. P. 265–288. DOI:10.1108/QEA-08-2024-0071.
2. Qayyum A., Zawacki-Richter O. Distance education in Australia, Europe and the Americas. *Open and distance education in Australia, Europe and the Americas*. Singapore. 2018. P. 121–131. DOI: 10.1007/978-981-13-0298-5_14.
3. E-Learning in european higher education institutions / M. Gaebel et al. *European University Association (EUA)*. URL: https://www.eua.eu/publications/reports/e-learning-in-european-higher-education-institutions.html (дата звернення: 09.08.2025).
4. Rosqueta L. R. A. Perceptions and challenges: A qualitative inquiry of elementary teachers’ experiences with AI-powered tools in DepEd Dingras District I. *Asian journal of education and social studies*. 2025. Vol. 51, no. 6. P. 1078–1088. DOI: 10.9734/ajess/2025/v51i62059.
5. Laddha M., Jadhav P. An automatic gap filling questions generation using NLP. *International journal of computer science & engineering technology*. 2017. Vol. 8. P. 325.
   URL: https://www.researchgate.net/publication/359391691_An_Automatic_Gap_Filling_Questions_Generation_using_NLP (дата звернення: 09.08.2025).
6. Ghuge A. R. A comparative review for automatic question generation using NLP. *International journal of scientific research in engineering and management*. 2025. Vol. 09, no. 04. P. 1–9. DOI: 10.55041/ijsrem46129.
7. Mitkov R., An Ha L., Karamanis N. A computer-aided environment for generating multiple-choice test items. *Natural language engineering*. 2006. Vol. 12, no. 2. P. 177–194. DOI: 10.1017/s1351324906004177.
8. Heilman M., Smith N. A. Good question! Statistical ranking for question generation. *Human language technologies: the 2010 annual conference of the north american chapter of the association for computational linguistics* / ed. by

R. Kaplan et al. Los Angeles, California. 2010. P. 609–617. URL: https://aclanthology.org/N10-1086/ (дата звернення: 09.08.2025).

9. Jurafsky D., Martin J. H. Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition with language models. 2025. URL: https://web.stanford.edu/~jurafsky/slp3/ (дата звернення: 09.08.2025).
10. Nguyen N. D. Exploring the role of AI in education. *London journal of social sciences*. 2023. No. 6. P. 84–95. DOI: 10.31039/ljss.2023.6.108.
11. Labrak Y., Rouvier M., Dufour R. A zero-shot and few-shot study of instruction-finetuned large language models applied to clinical and biomedical tasks. DOI: 10.48550/arXiv.2307.12114.
12. LLM-Generated multiple choice practice quizzes for preclinical medical students / T. Camarata et al. *Advances in physiology education*. 2025. Vol. 49, no. 3. P. 758–763. DOI: 10.1152/advan.00106.2024.
13. A systematic review of automatic question generation for educational purposes / G. Kurdi et al. *International journal of artificial intelligence in education*. 2019. Vol. 30, no. 1. P. 121–204. DOI: 10.1007/s40593-019-00186-y.
14. Alsubait T. M., Parsia B., Sattler U. Generating multiple choice questions from ontologies: lessons learnt. *OWL: experiences and directions*. 2014. URL: https://api.semanticscholar.org/CorpusID:17193311 (дата звернення: 05.04.2025).
15. Ellis R. Measuring implicit and explicit knowledge of a second language: A Psychometric Study. *Studies in second language acquisition*. 2005. Vol. 27, no. 2. P. 141-172. DOI: 10.1017/s0272263105050096.
16. Llama 2: open foundation and fine-tuned chat models / H. Touvron et al. DOI: 10.48550/arXiv.2307.09288.
17. On the dangers of stochastic parrots: can language models be too big? / E. M. Bender et al. *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*. New York, NY, USA, 2021. P. 610–623. DOI: 10.1145/3442188.3445922.

18. The carbon footprint of machine learning training will plateau, then shrink / D. Patterson et al. DOI: 10.48550/arXiv.2204.05149.
19. Measuring the carbon intensity of AI in cloud instances / J. Dodge et al. *Proceedings of the 2022 ACM conference on fairness, accountability, and transparency*. New York, NY, USA. 2022. P. 1877–1894. DOI: 10.1145/3531146.3533234.
20. Reconciling the contrasting narratives on the environmental impact of large language models / S. Ren et al. *Scientific reports*. 2024. Vol. 14, no. 1. P. 26310. DOI: 10.1038/s41598-024-76682-6.
21. Personalized MCQ generator for adaptive learning by using leveraging RAG method in gen AI / T. Kumar et al. *International journal of experimental & theoretical artificial intelligence*. 2025. Vol. 4. P. 1–10. DOI: 10.34218/ijetai_04_01_001.

ДОДАТОК А

Посилання на Git

1. nosenko-mi/LexiLoop. *GitHub*. URL: https://github.com/nosenko-mi/LexiLoop.

2. nosenko-mi/lexiloop-backend. *GitHub*. URL: https://github.com/nosenko-mi/lexiloop-backend.

3. nosenko-mi/lexiloop-generated-quizzes. *GitHub*. URL: https://github.com/nosenko-mi/lexiloop-generated-quizzes.

ДОДАТОК Б

Прототип мобільного додатку



Рисунок Б.1 – Прототип екрану LoginScreen

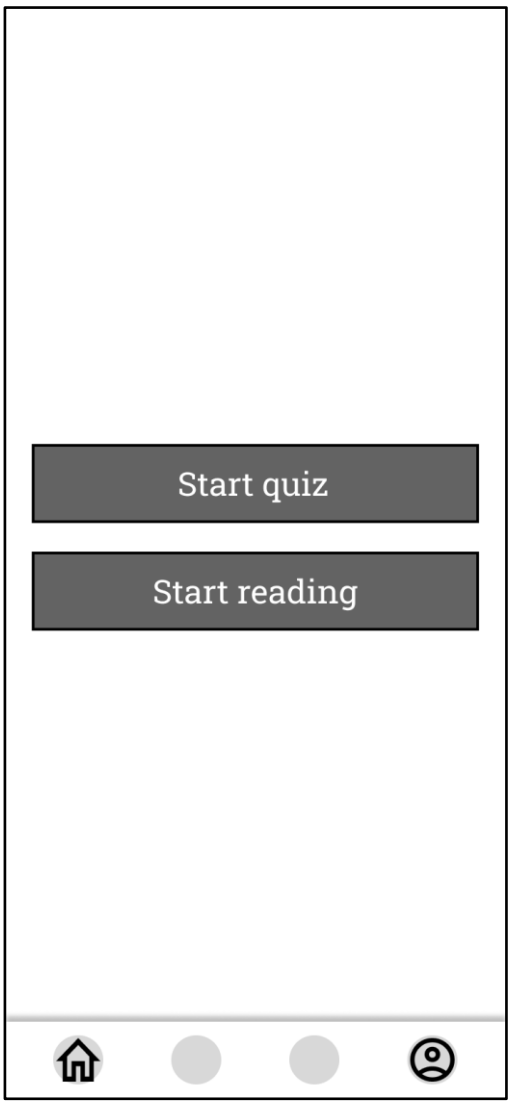


Рисунок Б.2 – Прототип екрану HomeScreen

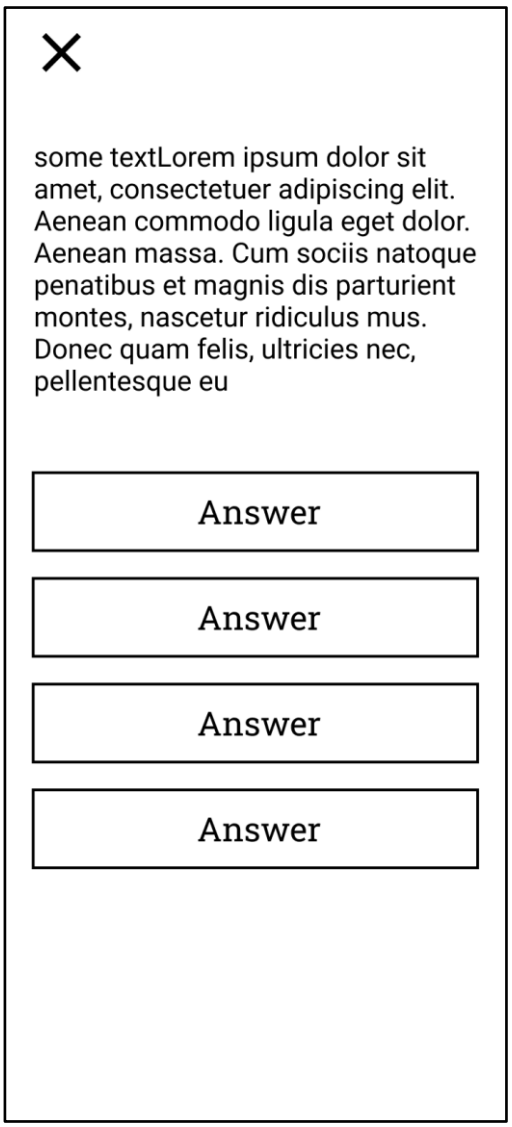


Рисунок Б.3 – Прототип екрану QuizScreen

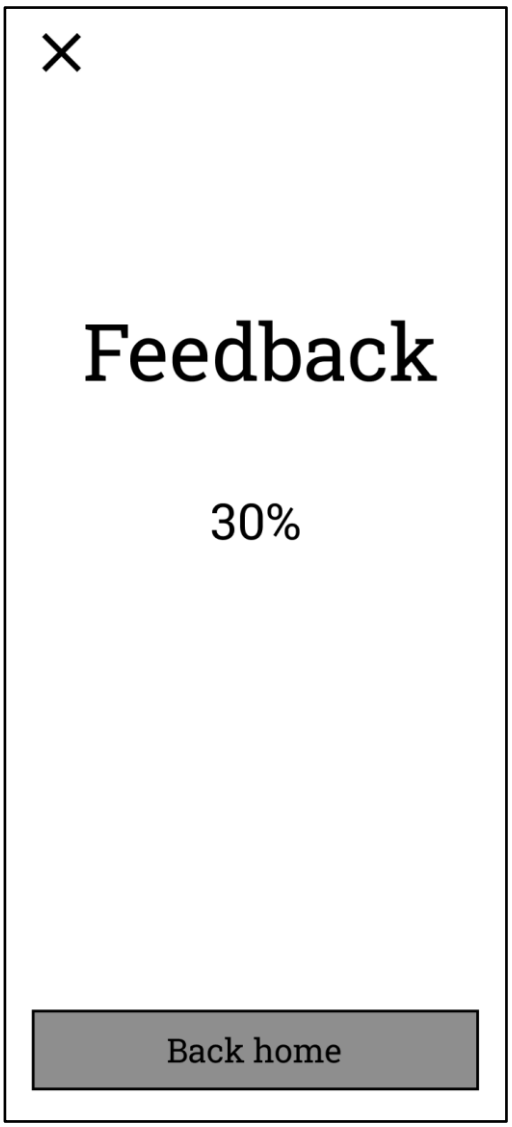


Рисунок Б.4 – Прототип екрану FeedbackScreen



Рисунок Б.5 – Прототип екрану RegisterScreen

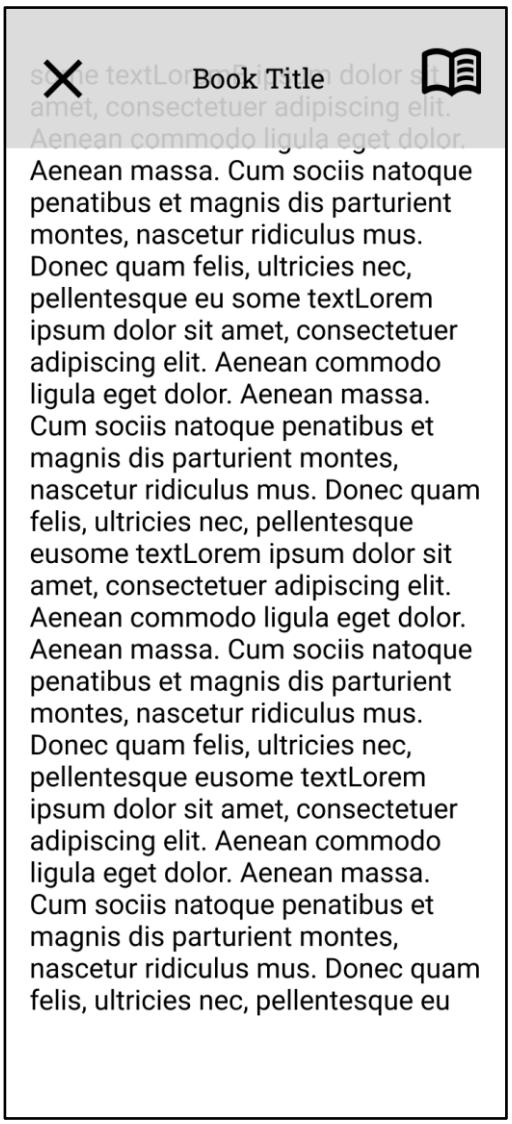


Рисунок Б.6 – Прототип екрану ReaderScreen

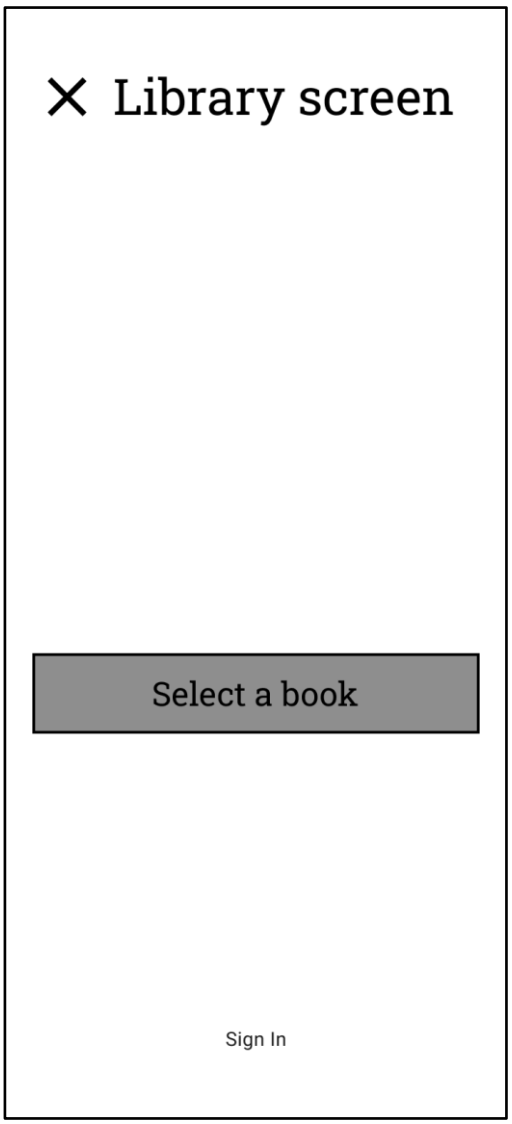


Рисунок Б.7 – Прототип екрану LibraryScreen

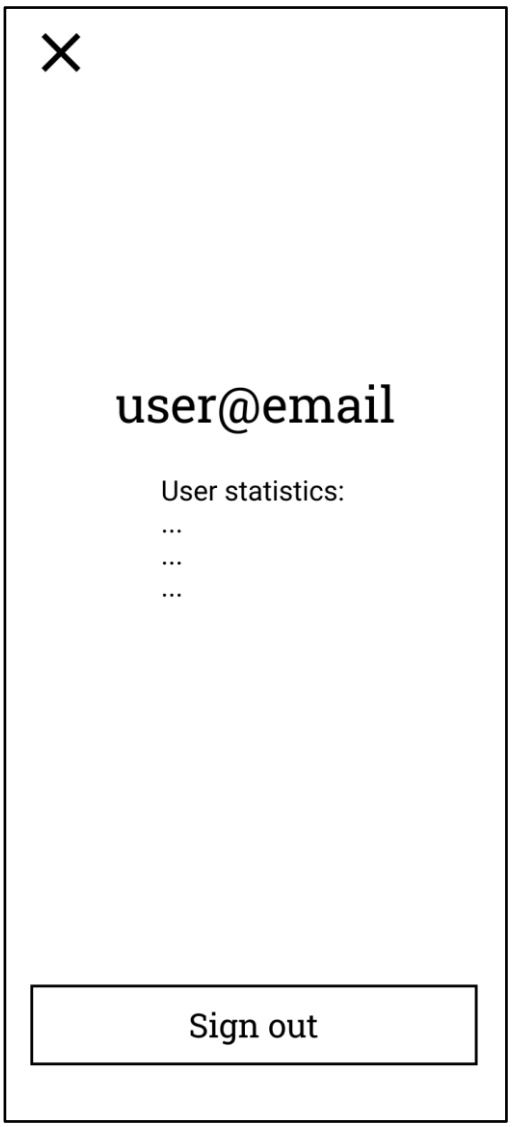


Рисунок Б.8 – Прототип екрану ProfileScreen

ДОДАТОК В

Приклади роботи мобільного додатку



Рисунок В.1 – Приклад створення завдання з виділеного тексту

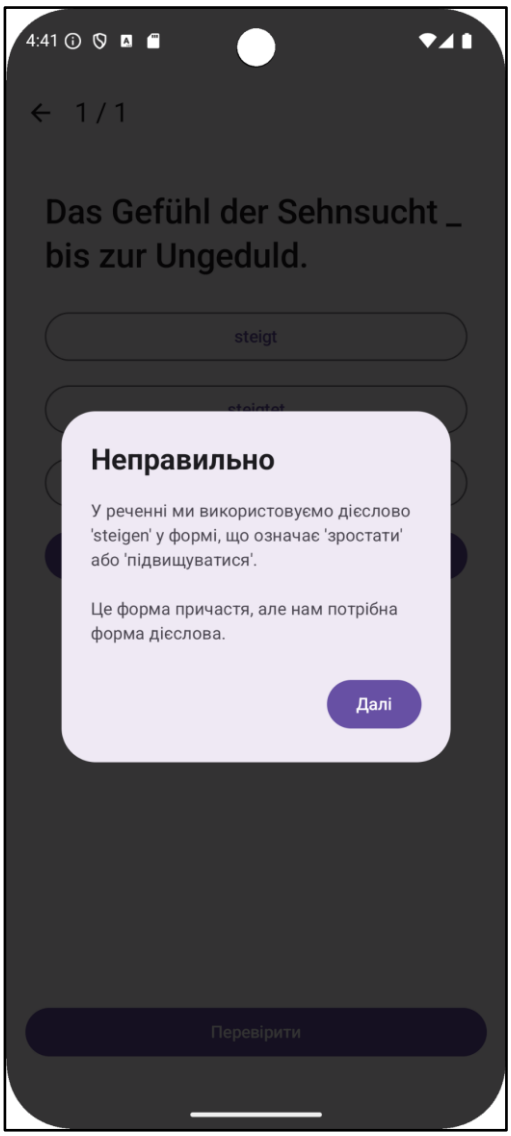


Рисунок В.2 – Приклад згенерованого пояснення до завдання

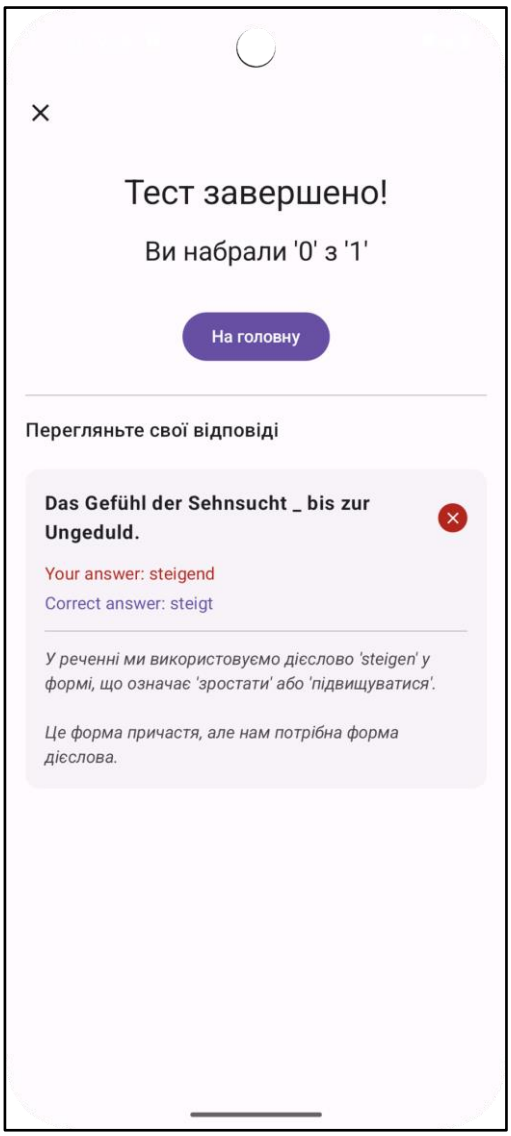


Рисунок В.3 – Приклад відгуку про проходження тесту

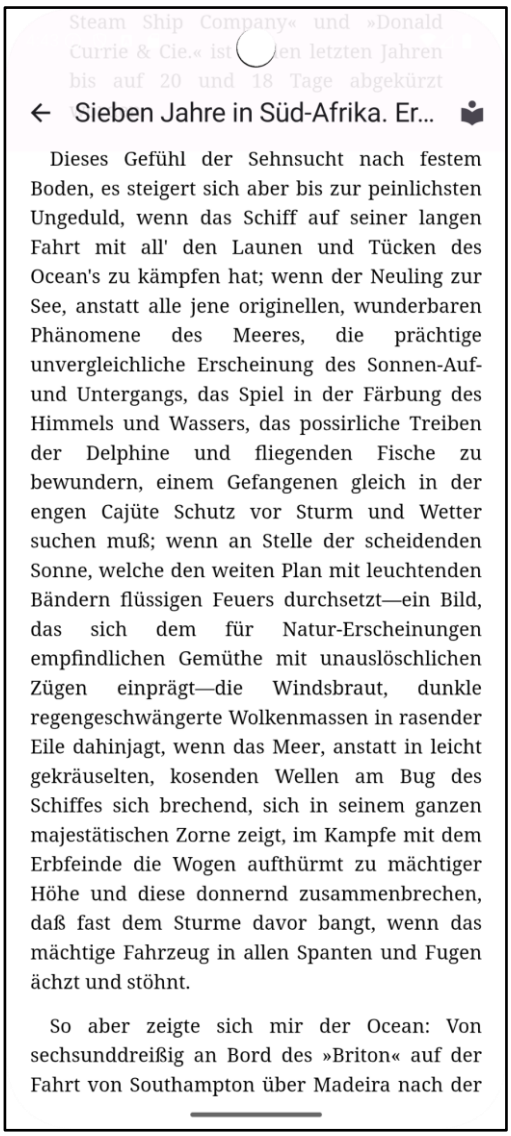


Рисунок В.4 – Приклад екрану читання

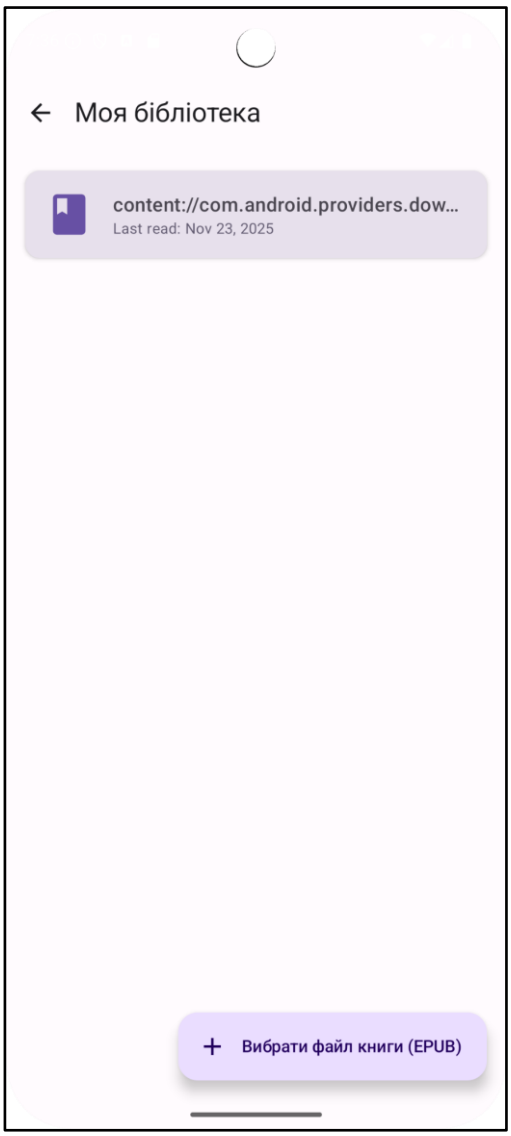


Рисунок В.5 – Приклад екрану бібліотеки

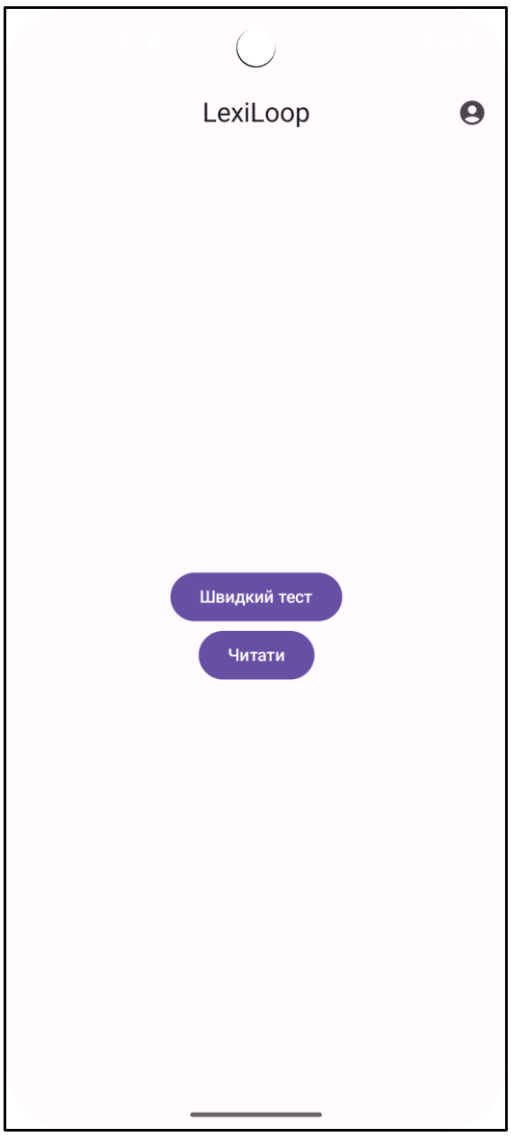


Рисунок В.6 – Приклад головного екрану

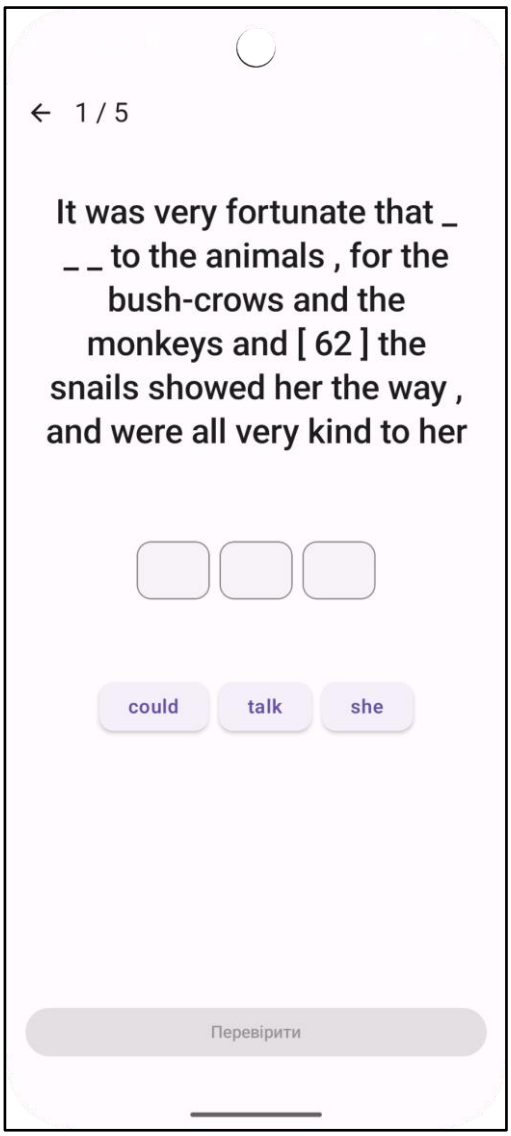


Рисунок В.7 – Приклад завдання на встановлення послідовності

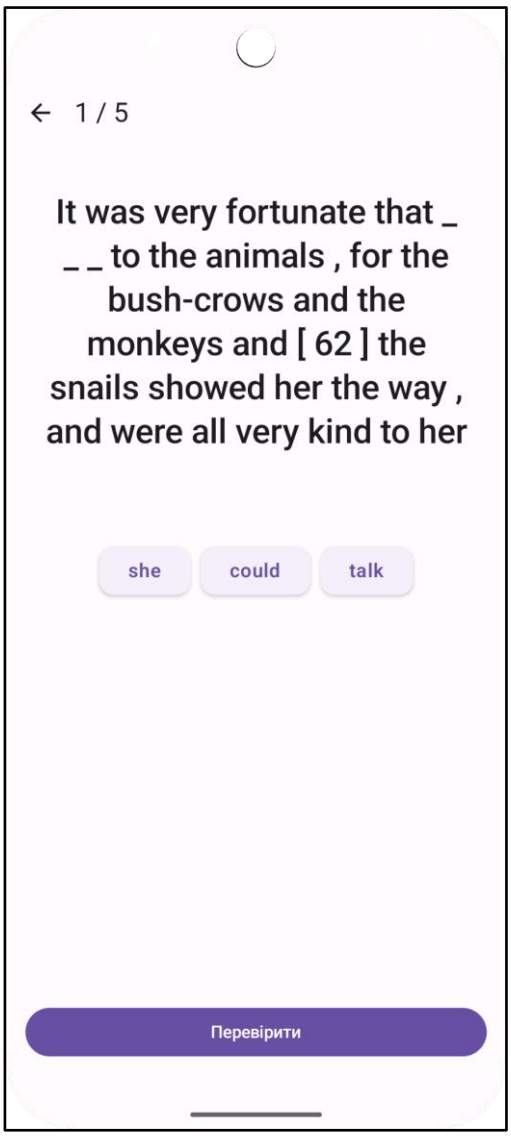


Рисунок В.8 – Приклад заповненого завдання на встановлення послідовності



Рисунок В.9 – Приклад статистики у профілі користувача

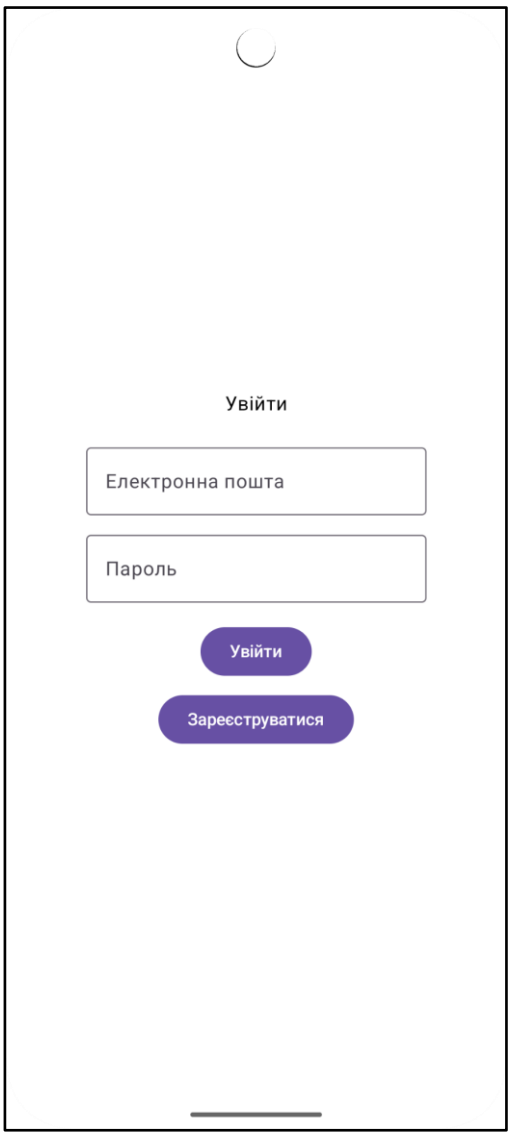


Рисунок В.10 – Приклад екрану входу в акаунт

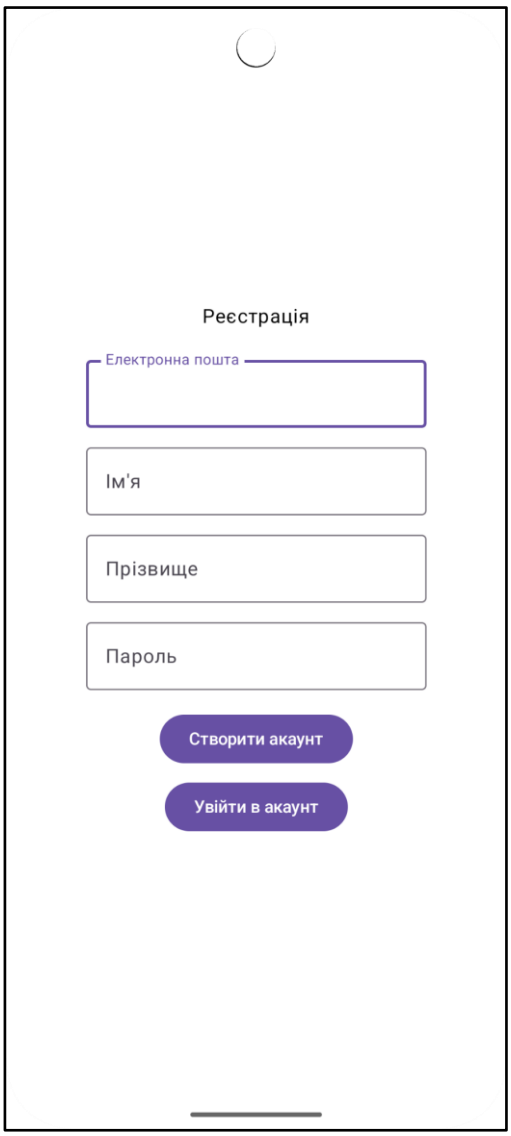


Рисунок В.11 – Приклад екрану реєстрації

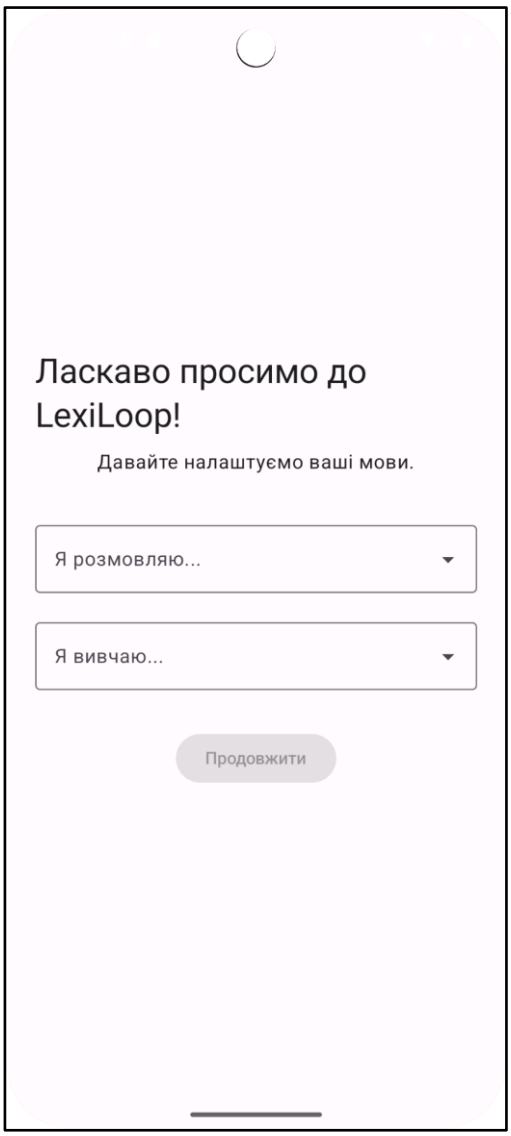


Рисунок В.12 – Приклад екрану Onboarding