Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Комп'ютерна система розпізнавання тексту на основі нейромережі»

08-54.БКР.012.00.000 ПЗ

Виконав: студент 4 курсу, групи 1КІ–21Б
спеціальності 123 — «Комп'ютерна інженерія»
освітня програма — «Системне програмування»
Левчук О.О.

Керівник к.т.н., доц. каф. ОТ
Городецька О.С.

Рецензент д.т.н., професор каф. програмного забезпечення
Ліщинська Л.Б.

Допущено до захисту
Завідувач кафедри ОТ
д.т.н., проф. Азаров О. Д.

«17» 06 2025 р.

Вінниця ВНТУ – 2025 рік

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки
Освітньо–кваліфікаційний рівень бакалавр
Галузь знань 12 — Інформаційні технології
Спеціальність — 123 Комп'ютерна інженерія
Освітня програма — Комп'ютерна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
д.т.н.проф._____Азаров О. Д.
«21» березня 2025 року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ

Левчуку Олегу Олеговичу

1 Тема роботи: «Комп'ютерна система розпізнавання тексту на основі нейромережі», керівник роботи Городецька Оксана Степанівнан к.т.н., доц., доцент кафедри ОТ, затверджено наказом вищого навчального закладу від «20» березня 2025 року №97.

2 Термін подання студентом роботи: 13.05.2025

3 Вихідні дані до роботи: текст, що був розпізнаний і перетворений у цифрову форму, з можливістю подальшої обробки чи аналізу; визначення мови тексту.

4 Зміст розрахунково-пояснювальної: вступ, аналіз методів розпізнавання тексту, обґрунтування вибору технологій, розробка комп'ютерної системи, перевірка працездатності та тестування, список використаних джерел, додатки.

5 Перелік графічного матеріалу: діаграма діяльності комп'ютерної системи розпізнавання тексту, алгоритм розпізнавання рукописного тексту; перелік ілюстративного матеріалу: рукописний контрастний шрифт із нахилом, рукописний шрифт із частковим розмиттям під нахилом, цифровий рукописний шрифт, результати опрацювання системою.

6 Консультанти розділів роботи наведені в таблиці 1.

Таблиця 1 — Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-3	к.т.н., доц. каф. ОТ Городецька О. С.	21.03.25	13.05.25
Нормконтроль	ас. каф. ОТ Швець С. І.	14.05.25	30.05.25

7 Дата видачі завдання: 21.03.2025 р.

8 Календарний план роботи наведено в таблиці 2.

Таблиця 2 — Календарний план

№ з/п	Назва етапів дипломної роботи	Термін виконання	Підпис
1	Постановка завдання роботи	21.03.25	вик
2	Аналіз існуючих методів розпізнавання тексту	24.03.25 – 26.03.25	вик
3	Вибір технологій для розробки системи	27.03.25 – 29.03.25	вик
4	Розробка архітектури системи та вибір інфраструктури	31.03.25 – 03.04.25	вик
5	Розробка бекенд частини системи	04.04.25 – 19.04.25	вик
6	Розробка фронтенд частини системи	21.04.25 – 26.04.25	вик
7	Використання засобів віртуалізації	28.04.25 – 29.04.25	вик
8	Тестування системи	30.04.25 – 03.05.25	вик
9	Оформлення пояснювальної записки та графічної частини	05.05.25 – 12.05.25	вик
10	Перевірка якості виконання бакалаврської кваліфікаційної роботи та усунення недоліків	13.05.25	вик

Студент Олег ЛЕВЧУК

Керівник роботи к.т.н., доц. каф. ОТ Оксана ГОРОДЕЦЬКА

АНОТАЦІЯ

УДК 004.8

Левчук О.О. Комп'ютерна система розпізнавання тексту на основі нейромережі. Бакалаврська кваліфікаційна робота зі спеціальності 123 — Комп’ютерна інженерія, освітня програма — Системне програмування. Вінниця: ВНТУ, 2025.

На укр. мова. Бібліогр.: 17 назв; рис.: 36; табл.: 4; 75 с.

У бакалаврській кваліфікаційній роботі розроблено комп’ютерну систему для розпізнавання тексту з графічних зображень, використовуючи нейромережеві технології. Проведено аналіз існуючих методів розпізнавання тексту, зокрема технологій OCR та ICR, а також принципів роботи алгоритмів машинного навчання для обробки рукописного тексту. Обґрунтовано вибір програмних інструментів і архітектури системи, реалізовано серверну частину застосунку для обробки запитів і клієнтський інтерфейс для завантаження зображень. Для навчання нейронних мереж застосовано Python та відповідні фреймворки, система розгорнута за допомогою Docker для забезпечення портативності та масштабованості. Проведено тестування системи, яке підтвердило її точність та стабільність при розпізнаванні тексту з різних типів зображень. Результатом є функціональний прототип системи, що дозволяє ефективно розпізнавати текст із графічних зображень і трансформувати його в текстовий формат для подальшої обробки.

Ключові слова: розпізнавання тексту, нейронні мережі, машинне навчання, OCR, ICR, Python, Docker, FastAPI, глибоке навчання.

ABSTRACT

Levchuk O.O. Text Recognition System Based on a Neural Network. Bachelor's thesis in specialty 123 — Computer Engineering, educational program — System Programming. Vinnytsia: VNTU, 2025.

Ukrainian language. Bibliography: 17 sources; figures: 36; tables: 4; 75 p.

The bachelor's thesis presents the development of a computer system for text recognition from graphical images using neural network technologies. The study includes an analysis of modern approaches to text processing, particularly OCR and ICR technologies, and the principles of handwritten text recognition. The selection of programming tools and system architecture is justified. The backend and frontend components of the application were implemented. Python and relevant AI frameworks were used for training neural networks, while Docker was employed for deployment to ensure portability and scalability. The system was tested to evaluate its performance, recognition accuracy, and stability. As a result, a functional prototype was developed to effectively recognize text from graphical images and convert it into editable text for further processing.

Keywords: text recognition, neural networks, machine learning, OCR, ICR, Python, Docker, FastAPI, deep learning, neural network.

ЗМІСТ

ВСТУП

Обробка текстів — основний вид діяльності багатьох спеціалістів, на який вони витрачають значний час та зусилля. Це робота, що вимагає великої концентрації і терпіння. В основному в текстах шукають, аналізують і порівнюють інформацію для подальшого використання. Прикладами таких професій є юристи, економісти, рекрутери та лікарі, для яких робота з текстами є важливою частиною діяльності.

У світі сучасних технологій з’явилось безліч інструментів, які допомагають працювати з великими об’ємами інформації у вигляді тексту: від простих пошукових систем, що спрощують таргетований пошук інформації за ключовими словами, до просунутих систем, які дозволяють аналізувати текст, створювати підсумки, перефразовувати та виконувати інші маніпуляції. Такі системи будуються на основі штучного інтелекту, який відповідає за методи обробки та змінення тексту. Останнім часом штучний інтелект активно застосовується для роботи з текстами, що сприяло стрімкому розвитку ШІ в цій сфері. Безліч ШІ-асистентів допомагають аналізувати, форматувати текст, а деякі навіть генерують текст за допомогою генеративних моделей, що значно спрощує роботу з ним.

Основним викликом для таких систем є те, що вони здатні працювати з текстом, переданим у вигляді цифрового представлення символів, але не так ефективно з рукописним текстом, написаним безпосередньо людиною.

Розпізнавання рукописного тексту є окремою галуззю в розробці ШІ.
Існує декілька підходів до обробки такого тексту.

ML (Machine Learning) — за допомогою якого зображення з текстом перетворюється на комп’ютерне представлення шляхом зчитування цифрового вигляду тексту та співставлення цифрової сигнатури з передбачуваною сигнатурою символу в моделі штучного інтелекту.

OCR-алгоритми, які працюють не з вмістом фотографії на рівні її цифрового коду, а розпізнають текст за допомогою візуальної сигнатури та співставлення її з передбачуваною.

Комбінований підхід, який передбачає використання алгоритму OCR для визначення візуальної сигнатури символу, яка потім передається в модель ШІ для більш точного співвідношення візуального вигляду символу з комп’ютерним представленням символу.

У процесі роботи буде створено систему, яка надаватиме мережевий інтерфейс для розпізнавання символів рукописного тексту. Це буде здійснюватися шляхом створення OCR-моделі та визначення символу за допомогою моделі штучного інтелекту. Окрім того, буде розроблено графічний інтерфейс, який забезпечить зручний спосіб взаємодії користувача з системою, що сприятиме покращенню користувацького досвіду.

Об’єктом роботи є процес розпізнавання символів рукописного створення за допомогою алгоритму розпізнавання OCR.

Предметом дослідження є використання нейронних мереж та алгоритмів OCR для розпізнавання символів рукописного створення.

Метою роботи є підвищення точності розпізнавання рукописного тексту шляхом використання комбінованого підходу, що поєднує технології OCR та нейромережеві моделі.

Задачі дослідження бакалаврської роботи:

— провести аналіз предметної області;
— обрати технологію, за допомогою якої буде здійснюватись обробка;
— розробити структурну схему системи та встановити перелік функціональних вимог до системи;
— підібрати комбінацію технологій, які будуть реалізовувати встановлений перелік функціоналу;
— протестувати систему на відповідність результатів.

1 АНАЛІЗ МЕТОДІВ РОЗПІЗНАВАННЯ ТЕКСТУ

1.1 Аналіз сфери застосування технологій штучного інтелекту

За останні десять років технології штучного інтелекту стали дуже популярними, і індустрія в цілому почала стрімко розвиватися. Великі компанії почали активно інвестувати в цю галузь, що дозволило технологіям перейти на новий етап розвитку. На сьогодні ця галузь є однією з найбільш швидко зростаючих і перспективних. Така популярність цих систем зумовлена їх здатністю суттєво оптимізувати великі процеси, покращувати результати виконання завдань, а в деяких сферах навіть заміщати людей, наприклад, у сфері аналізу великих даних.

Розвиток таких систем відбувається у різних сферах:

— соціологія — моделі, здатні працювати з великими об’ємами даних, використовуються для опрацювання результатів опитувань, оцінки соціальних категорій, отримання їх характеристик, класифікації тощо;

— аеро-космічна промисловість — моделі штучного інтелекту активно застосовуються для обробки даних аеродинамічного скринінгу та прорахунку силового скелета літаків, космічних шатлів, аеродинаміки та конфігурації технічних частин;

— автомобільна промисловість — ШІ активно використовуються для віртуальних тестувань передвиробничих екземплярів двигунів, прорахунку електросхем, проектування частин підвісок, кузовних елементів та частин трансмісії.

Окрім інженерно-технічних застосувань, штучний інтелект також використовується в економічному секторі. Існує велика кількість асистентів, інтегрованих у системи контролю продажів (CRM) та розподілу ресурсів (ERP), які допомагають визначати продукти з більшим попитом залежно від сезонності, свят, ринкових умов, а також прогнозувати маржинальність, доходи та витрати, і навіть розраховувати податкові ставки. Прикладом такої системи є польська система **V-Firma** від компанії **V-Firm Spz.o.o.** Це підтверджує, що різноманітні моделі штучного інтелекту, машинного навчання, глибокого навчання та комп’ютерного

зору (CV) мають широкий спектр застосувань і можуть принести практичну вигоду, якщо їх правильно використовувати, чітко визначаючи сферу відповідальності, яку покладають на ШІ.

У 2007 році в рамках експериментальної програми НАТО було впроваджено модель штучного інтелекту, основною задачею якої було провести аналіз фінансового сектору оборонної промисловості однієї з дочірніх компаній. Відділ з 20 працівників в експериментальному режимі був замінений на спеціальну закриту модель штучного інтелекту, яку було створено спеціально для цієї задачі. Розробка та тестування моделі, включно з процесами збору, структуризації та фільтрації даних, зайняли 7 місяців, після чого модель було впроваджено в життя. Класифікаційна модель повинна була провести аналіз річних фінансових звітів, вказати на місця, де є певні фінансові аномалії, наприклад, занадто високу відсоткову ставку фінансування для одного сектора, структуризувати цю інформацію та підготувати фінансовий звіт, який допоможе компанії дослідити свої витрати та оптимізувати їх, що дозволить збільшити прибуток компанії. Паралельно з моделлю штучного інтелекту працював цілий відділ фінансистів, що налічував 20 працівників. Обробка даних, створення звіту та інші процеси, які необхідно було провести над вхідними даними, зайняли у програми майже 4 дні, після чого був отриманий результат у вигляді фінансового звіту. Команда спеціалістів, яка готувала той самий звіт, витратила на аналогічну роботу 7 робочих днів, що в робочих годинах складало 1120 годин роботи 20 спеціалістів. Коли команда завершила свою роботу, звіти, отримані від моделі штучного інтелекту та команди фінансистів, були порівняні. В результаті модель штучного інтелекту надала точніший результат, ніж фінансові аналітики з досвідом у сфері та гарними навичками. Результатами експеременту стали реальні зміни в компанії - провелася реструктуризація структурного підрозділу. Компанія скоротила штат, за рахунок чого змогла дозволити собі витрачати більші кошти на підтримання працівників, які залишились в компанії, а також інвестувати у інші сфери роботи компанії.

Хоча, незважаючи на те, що різноманітні форми штучного інтелекту зарекомендували себе як дуже ефективні в певних сферах застосування, практика

показує, що штучний інтелект на теперішньому етапі його розвитку все ще не здатний повністю замістити людей навіть у суто машинальних сферах. Наприклад, на основі того ж експерименту підрядника-компанії НАТО: не було усунено підрозділ фінансистів як такий, його було суттєво оптимізовано. Певний ступінь скорочення значно вплинув на простоту керування підрозділом, а додатково покращив якість виконання роботи. Такого результату вдалося досягти завдяки тому, що штучний інтелект відповідає лише за машинальну роботу, яку виконували люди, але сам штучний інтелект не є самодостатнім, оскільки хтось повинен надавати йому дані для аналізу та валідувати результати, які він повертає. Окрім того, незважаючи на те, що штучний інтелект не має параметрів, притаманних людині (він не має втоми, не потребує відпусток, не хворіє, і ще ряд інших властивостей йому не притаманні через його природу як машини), він не може виконувати безпосередній процес мислення, хоча може його імітувати. Найкращою конфігурацією співпраці, яка продемонструвала найбільшу ефективність, є співпраця операторів штучного інтелекту, які керують роботою AI, надають йому вхідні дані, перевіряють результати виконання, налагоджують збої в роботі програм і самій програмі безпосередньо.

1.2 Штучний інтелект та його види

Поняття штучного інтелекту існує з середини 50-х років минулого століття. Після винайдення першого електронного комп’ютера, програмованого за допомогою команд і здатного виконувати математичні (переважно базові арифметичні операції) обчислення, було висунуто кілька теорій про те, що існує можливість створити машину, яка буде «думати» подібно до людського мозку. Ці теорії стали першим серйозним прецедентом, який сприяв початку дослідження цього аспекту інформаційних технологій. Насправді, теорії, які були відносно схожими, існували ще до того. Перші уявлення про ШІ були озвучені ще у 15-16 століттях філософами, які намагалися описати процес людського мислення як паттерн операційних послідовностей у символьному вигляді. Тобто, концепція сучасного розуміння поняття «Штучний інтелект» була висунута ще в той час.

Перші предметні дослідження почалися після конференції на тему штучного інтелекту, проведеної в Сполучених Штатах Америки. У 1956 році Алан Тюрінг висунув теорію (рис. 1.1) того, як запрограмувати комп'ютерний процесор для того, щоб він міг повторити поведінку мисленнєвого процесу людини.

Його концепція полягала в тому, що комп'ютерна система здатна змінювати свій поточний стан на основі певного набору вхідних параметрів. Фактично, з огляду на рівень розвитку технологій того часу, це було продовженням абстракції кінцевого автомата. Кінцевий автомат — це абстрактне поняття, яке означає, що стан дискретної системи повинен змінюватися залежно від переліку обставин, у яких знаходиться комп'ютерна система.

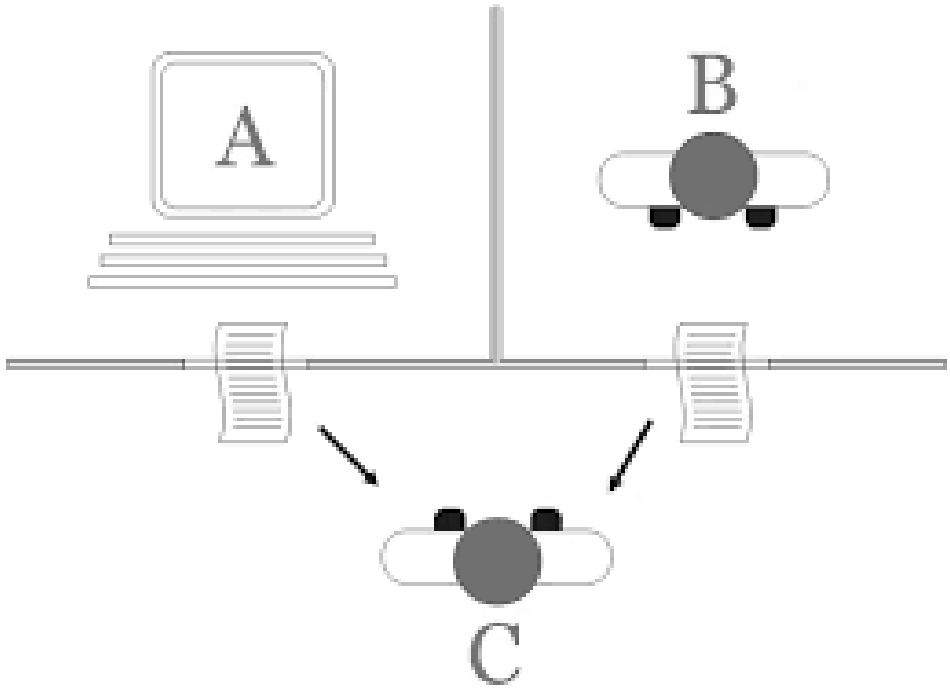

Рисунок 1.1 — Наочне представлення теорії штучного інтелекту Алана Тюрінга

Теорія Алана Тюрінга в основі своїй унаслідувала філософські праці на тему того, як працюють людські патерни мислення. Його теорія полягала в тому, що процес мислення є алгоритмічною послідовністю певних факторів, на основі яких приймається рішення. Відповідно до цього, він вважав, що таку саму поведінку можна відтворити за допомогою перемикання позитивних та негативних комп'ютерних сигналів, повторюючи тим самим роботу нейрону в мозку людини.

Його теорія була дуже загальною, і, незважаючи на те, що більшість свого життя він присвятив удосконаленню цієї теорії та створенню прототипу машини, здатної повторити поведінку людського мозку, йому не вдалося досягти результату, який би наблизив його теорію до реальності. Проте його праці були продовжені, і, хоча його ідеї залишаються актуальними й досі, тест Тюрінга сьогодні є однією з метрик для оцінки ефективності роботи штучного інтелекту та різноманітних систем машинного і глибокого навчання, теоретична та практична частина зробила великий крок уперед від того часу.

На сьогоднішній день існує чітке поняття того, що таке штучний інтелект і які інтелектуальні системи можуть підпадати під його визначення. Отже, штучний інтелект — це інтелектуальна система, яка повинна виконувати завдання, для яких зазвичай потрібно втручання людини. Такі інтелектуальні системи здатні до навчання, самонавчання та можуть приймати рішення, що базуються на основних когнітивних функціях. Такі системи повинні пройти процес навчання, щоб отримати досвід, і на основі цього досвіду система зможе видавати рішення в майбутньому. Сучасні технології дозволяють суттєво покращити процес навчання завдяки впровадженню епох навчання. Епоху навчання штучного інтелекту можна уявити як покоління людей. Кожне покоління, в певному сенсі, є більш розвиненим за попереднє. Наприклад, діти зараз краще орієнтуються в технологіях на базовому рівні, ніж два покоління тому. За тим самим принципом працюють епохи штучного інтелекту. Наприклад, коли в системі штучного інтелекту є п’ять епох, остання, п’ята, має весь досвід попередніх чотирьох, тому точність цієї епохи є вищою. Якщо система має правильні налаштування, з кожною епохою точність виконання операцій, наприклад, класифікації фотографій, буде збільшуватися. Таким чином, результати роботи можна покращити завдяки лінійному збільшенню кількості епох.

На основі концепції штучного інтелекту було розроблено більш змістовні та глибокі підходи до створення інтелектуальних машин, які можуть приймати рішення. Саме поняття штучного інтелекту, як можна було зрозуміти з визначення, є загальним уявленням концепції того, що машини можуть думати, приймати

рішення, навчатися, алгоритмічно будувати послідовність рішень. Це, так би мовити, оболонка для більш конкретних підходів та систем.

Наприклад, на основі теорії штучного інтелекту було створено поняття машинного навчання. Машинне навчання — це сфера комп'ютерної науки та інформаційних систем, що передбачає навчання на основі певного обсягу інформації, яка має назву «база знань штучного інтелекту». Тобто, це інформаційна система, яка пройшла етап навчання на певних вибіркових даних, підготовлених для неї, представлених у певному вигляді, структурованих та проведених через систему. Прикладом такої системи є чат-боти підтримки користувачів. Такі боти зазвичай реалізуються саме за допомогою такого підходу.

Протягом певного часу підготовлюється інформація із випадками, які ставались із користувачами

Коли зібрано обсяг інформації, що має певне покриття проблемних аспектів роботи технічного рішення, для якого створюється цей бот, інформація структурується. Наприклад, вона перетворюється на так званий лінгвістичний граф (рис. 1.2), де інформація має вигляд (назва випадку) — (опис випадку) — (потенційні проблеми, які могли призвести до цього випадку) — (дії, які потрібно виконати для вирішення цього випадку). Кожна така послідовність поєднується з наступною, що в результаті формує N-верховий граф, де вершиною є випадок, а зв'язком — пов'язаність між описами випадків. Кожному зі зв'язків присвоюється вага, яка також враховується при визначенні того, до якої саме відповіді звертатиметься чат-бот. 1

Коли такі графи побудовано, вони передаються до моделі штучного інтелекту як матеріал, на основі якого він має провести навчання. Навчання являє собою процес, в результаті якого модель машинного навчання буде розуміти кожен окремий випадок для відповіді, а також вартість зв'язків між кожним із випадків. Оцінюючи запит, модель порівнюватиме вміст запиту з описом випадку. Коли буде знайдено випадок, у якому опис максимально підходить до вхідних даних, які надав користувач, модель на основі сумарної вартості зв'язків видаватиме відповідь, що містить послідовність дій, які потрібно виконати для вирішення проблеми.

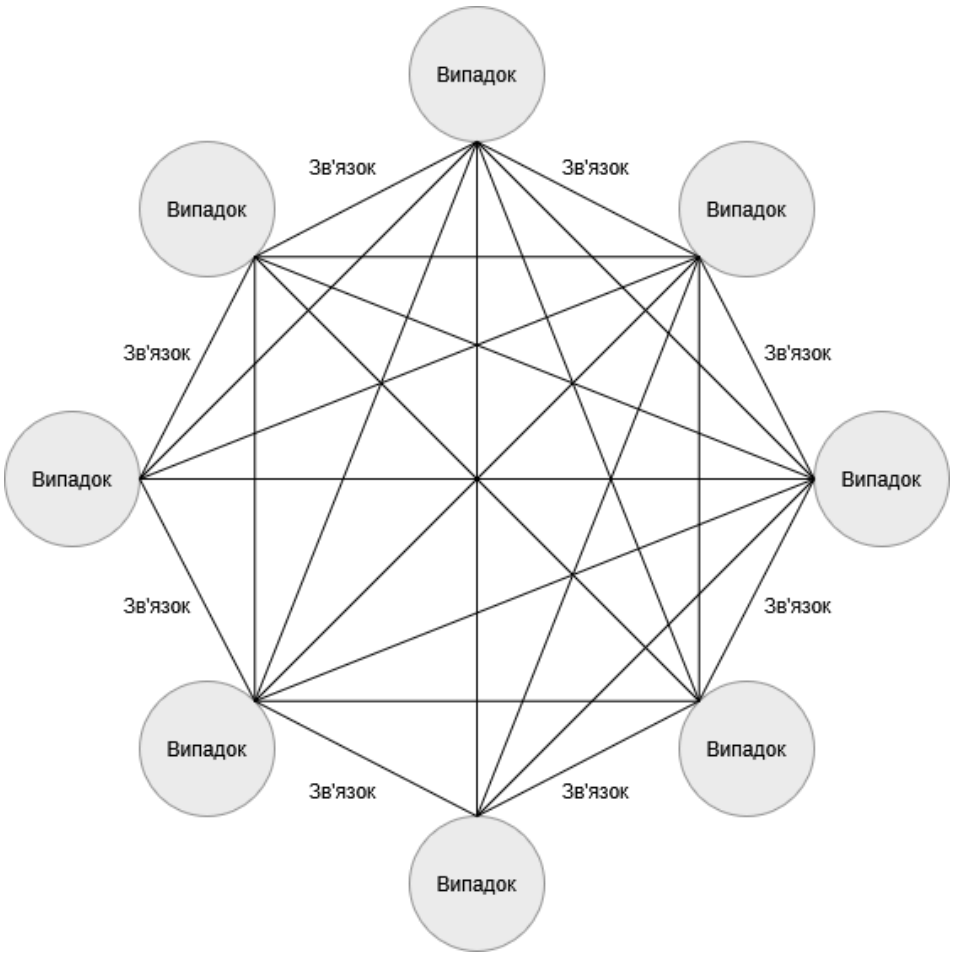

Рисунок 1.2 — N-порядковий граф прийняття рішення з вагою зв’язку

Після того, як модель була навчена, вона зберігається у файл зі спеціальним розширенням, щоб не проходити процес навчання при кожному запиті до моделі.

Коли файл збережено, його можна повторно використовувати для отримання відповіді на передбачені запити.

Ще більш точними є системи глибокого навчання. Згідно з визначенням, модель глибокого навчання — це модель, яка навчена на певному переліку даних, наданих розробниками системи, але модель навчається в глибину. Навчання в глибину передбачає наявність кількох ліній навчання та декількох нейронів, які приймають рішення (рис. 1.3).

Дивлячись на те, як виглядає процес навчання, можна знайти деякі схожості між процесом навчання моделі глибокого навчання та моделлю машинного навчання, але насправді між ними є дуже важлива різниця.

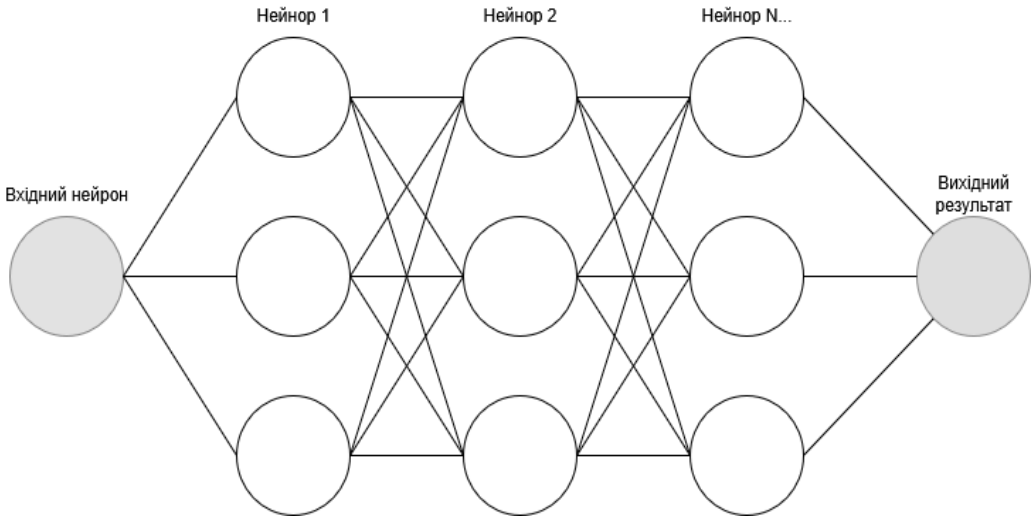

Рисунок 1.3 — Процес навчання N-нейронної моделі глибокого навчання

У моделі машинного навчання використовується 1 нейрон, який навчається, і відповідно тільки він приймає рішення. Модель глибокого навчання працює докорінно інакшим чином. Під час процесу навчання існують 3 поняття: вхідний нейрон, нейрон навчання, вихідний нейрон, кожен із яких має окреме призначення. Вхідний нейрон по суті є просто точкою входу в модель навчання, його задачею є отримати вхідну інформацію та розподілити її між кожною вершиною наступного нейрона. Тобто, він займається виключно створенням окремих підпакетів інформації, які формуються з цілого пакету інформації, підготовленої для навчання. Коли інформація опрацьована вхідним нейроном і рівномірно розподілена між кожною вершиною графа першого нейрона навчання, відбувається безпосередній процес навчання. Процес безпосереднього навчання передбачає той самий алгоритм — встановлення зв’язків між інформацією та встановлення ваги зв’язків. Кількість нейронів може бути довільною, але точність моделі пропорційно залежить від кількості нейронів, за умови, що нейрони не деградують. Деградація нейронів — це процес, при якому точність опрацювання інформації з кожною епохою (нейроном) зменшується. Таке явище відбувається через різні причини, але основним фактором є те, що інформація для навчання була підібрана неправильно.

Наприклад, в системі, яка призначена для розділення фотографій, на яких зображені коти та собаки, було надано фотографії для навчання, серед яких були зображення тварин, що візуально схожі на ті, які повинні класифікуватися.

Відповідно, кожна наступна епоха буде гірше класифікувати фотографії котів та собак, оскільки вона буде плутатися, адже дані, що класифікуються, не точно співпадають із даними, на яких навчалась модель.

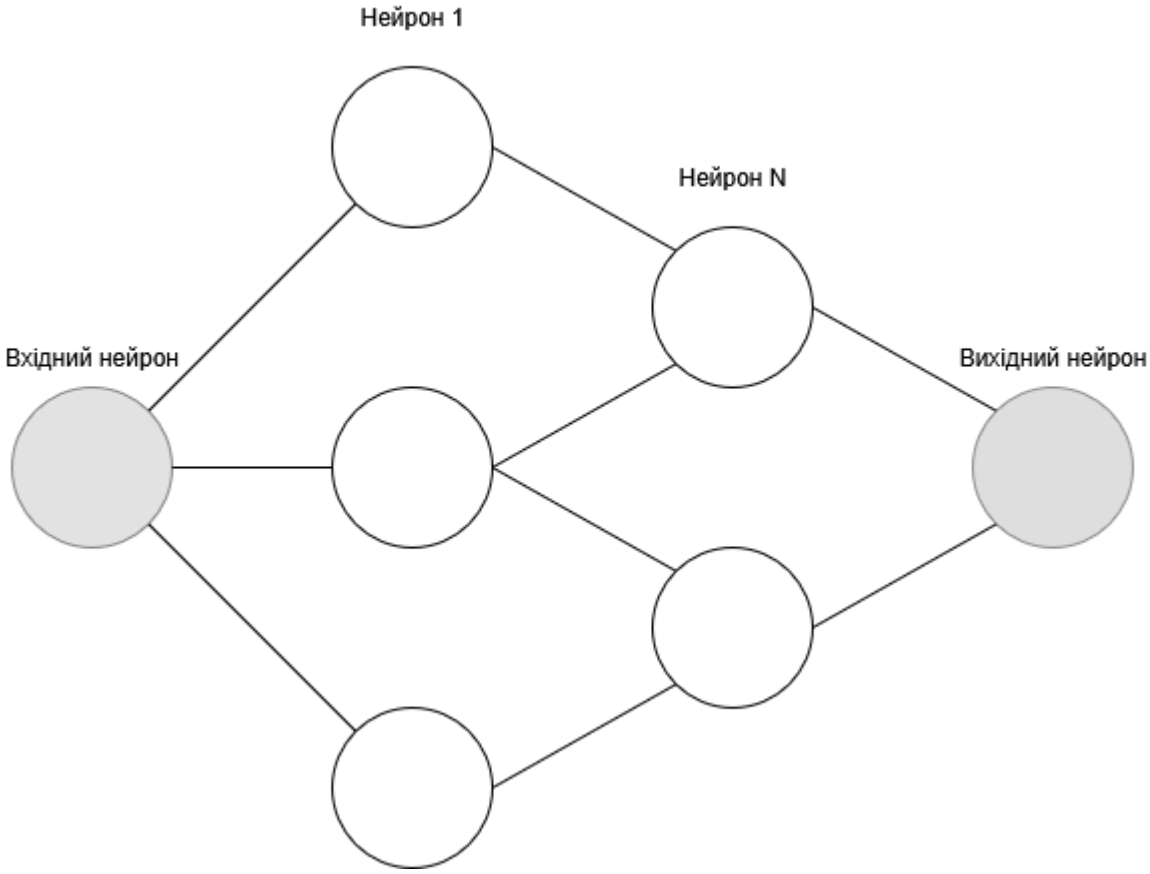

Рисунок 1.4 — Приклад прийняття рішення моделлю deep learning

Алгоритм прийняття рішень моделлю deep learning за допомогою N нейронів схожий на той, який використовується для навчання з N нейронами, але різниця полягає в тому, що алгоритм прийняття рішень працює таким чином: вхідний нейрон, подібно до алгоритму навчання, розподіляє усі вхідні дані між нейронами прийняття рішень. Надалі кожен нейрон приймає своє рішення, після чого результат передається на наступний шар нейронів. У наступному шарі нейронів кількість графів гарантовано стає меншою (як показано на рис. 1.4: після входу в нейрон трьох вершин, потім двох, і зрештою вихідний нейрон). Однак кількість нейронів може збільшуватись, відповідно кількість графів для порівняння також буде більша.

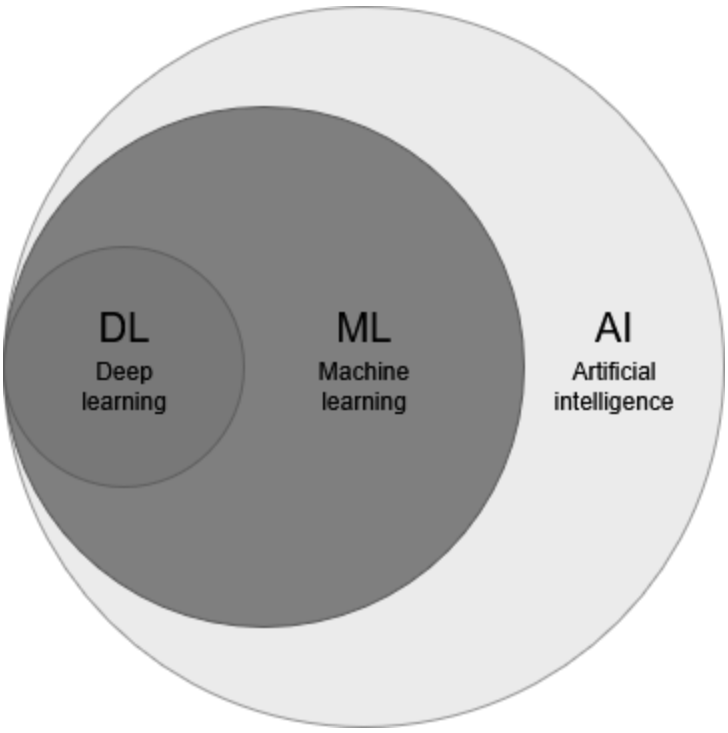

Рисунок 1.5 — Ієрархічне співставлення різних технологій ШІ

Проаналізувавши усі форми штучного інтелекту, було досліджено всі популярні методи застосування AI (рис. 1.5), такі як: Artificial Intelligence (Штучний інтелект) — загальне поняття, яке включає в себе опис підходу використання комп'ютерної системи як системи, здатної мислити та виконувати когнітивні функції; machine learning (машинне навчання) — більш конкретний підхід, який визначає певні правила застосування комп'ютерів для задач, для яких необхідний інтелект людини; deep learning — підхід до використання комп'ютера як багаторівневої системи мислення, здатної на основі багаторазового вивчення певної інформації опрацьовувати складні дані, які потребують врахування багатьох факторів, і є більш здатною до прийняття зважених рішень.

1.3 OCR та принцип роботи алгоритму

Обробка фото є окремою гілкою розвитку AI, яка профільно займається різного роду роботою з фото: автоматизованою обробкою, видаленням артефактів, класифікацією та навіть генерацією зображень. Складність обробки зображень пропорційно більша, ніж обробки текстових запитів, через те, що зображення не так просто сприймаються штучним інтелектом, оскільки мають більшу кількість

параметрів. Серед основних параметрів зображення є: кольоровість, розмір, чіткість, чистота. Наприклад, обробити монохромне (чорно-біле) зображення розміром 28 на 28 пікселів на одноманітному тлі з відцентрованим зображенням (рис. 1.6), виставленим по центру без нахилу, значно простіше, ніж опрацювати зображення у розмірі Full HD (1920 пікселів на 1080 пікселів) на кольоровому заповненому тлі з перспективою, повернутим під кутом, яке має візуальні погрішності та артефакти (рис. 1.7).

Рисунок 1.6 — Приклад якісного зображення для розпізнавання

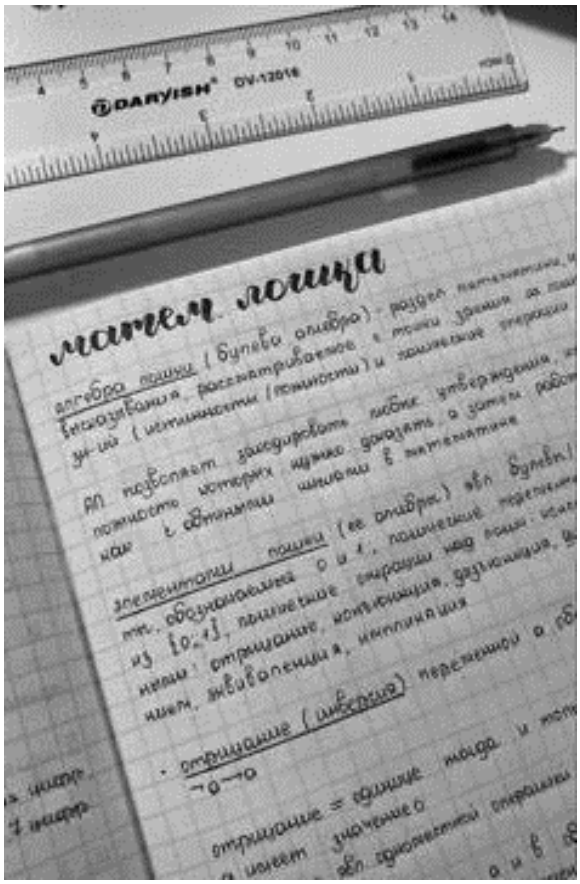

Рисунок 1.7 — Приклад поганого зображення для розрпізнавання

Найкращим підходом для навчання моделі розпізнавати символи є надання їй набору добре підготовленої інформації у відповідному вигляді для опрацювання. Існують спеціалізовані набори підготовлених даних, які вже заздалегідь підходять під ті параметри, які найкраще підходять для навчання. Такі набори даних мають назву датасети і можуть бути як у безкоштовному доступі, так і приватні, використання яких обкладається ліцензіями. Після завантаження цих датасетів у програму штучного інтелекту вони переводяться у вигляд, зручний для опрацювання та порівняння. Вони набувають вигляду абсолютних величин, які розміщуються у певній матриці (рис. 1.8). Через це рекомендується надавати моделі монохромний кольоровий спектр та певний уніфікований розмір.

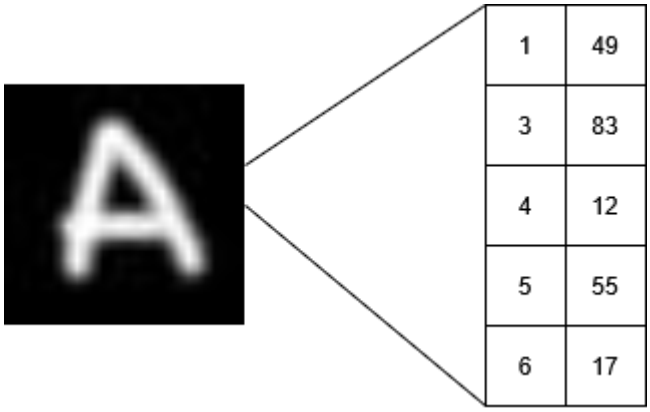

Рисунок 1.8 — Представлення зображення у вигляді матриць абсолютних величин

Таким чином, моделі проводять операції над зображеннями. Щоб гарантувати консистентність даних, системи переводять зображення у такий вигляд, в якому вони будуть зручно сприйматися — монохромне зображення розміром 28 пікселів на 28 пікселів. Потім AI отримує послідовно кожне значення з певної комірки та порівнює його з відповідним значенням тієї ж комірки в еталонних даних.

Що знаходиться в комірках? Для монохромного зображення використовується матриця 2 на N, де в першому стовпці знаходиться значення порядкового номера пікселя на зображенні, а в іншому — його кольорове значення в певній системі кодування (RGB або SMIK).

Найкращою якістю обробки зображень славляться системи, які працюють за принципом OCR. OCR — це моделі, які використовують глибоке навчання для визначення тексту на зображенні. Їх існує велика кількість, хоча більшість із них наслідує базові OCR-системи з певними доопрацюваннями або спеціалізацією. Серед них — ICR, який профільно заточений під розпізнавання рукописного тексту.

Такі системи часто використовуються для автоматизації роботи, пов’язаної з взаємодією з документами. Їх можна зустріти під час проходження KYC (Know Your Customer) на певних вебресурсах, а також у фінансових та державних установах. Класичним застосуванням цього алгоритму є його використання для отримання паспортних даних у різних системах перепусток, наприклад, у комерційних сканерах документів, які можна зустріти в багатьох бізнес-центрах.

Також такі системи використовуються для цифровізації та діджиталізації архівних документів. Процес їх оцифровки зайняв би дуже багато часу, якби доводилося переписувати їх вміст за допомогою мануального друкування цих документів. Натомість системи ICR можуть це робити цілодобово з належною якістю та майже автоматизовано.

1.4 Вибір алгоритму ICR

Існує велика кількість підходів, за допомогою яких можна створити систему розпізнавання рукописного тексту на основі фотографій або цифрових сканів. Серед доступних варіантів: створення системи з нуля, використання систем, які призначені для навчання, але вже мають готові інтерфейси для взаємодії з ними у такому ключі, або використання готової моделі, яка навчена та надає зручний інтерфейс для комунікації.

Порівняння доступних варіантів:

1) Створення системи з нуля

Створення такої системи з повного нуля ставить перед розробниками велику кількість викликів. Потрібно створити алгоритм нормалізації зображення, а саме трансформації зображення у потрібний розмір, перетворення його в монохромний

спектр, центрування зображення по всіх осях, керування кутом обертання зображення. Окрім цього, потрібно буде навчити модель, яка буде розпізнавати символи, а відповідно — створити алгоритм переведення зображень у матриці абсолютних величин та створити процесор, який буде зберігати та ефективно використовувати цю інформацію. Цей підхід буде доцільним у випадку дуже специфічних задач, якщо готового рішення просто не існує або воно не задовольняє всіх вимог, які стоять перед майбутньою системою.

2) Використання підготовленої системи для навчання

Цей підхід є більш розповсюдженим, бо має гарне співвідношення витрачених ресурсів — як часових, так і фінансових — до якості роботи. Такі моделі отримали розповсюдження не тільки у якості персональних AI агентів, а й корпоративних рішень, які обслуговують цілі корпорації. Перевагою такого підходу є те, що спеціалісту не потрібно занурюватися в усі тонкощі роботи із зображеннями, їх обробки, створення процесорів та налагодження всієї системи для роботи. Задачею спеціаліста є підготовка якісного набору інформації для навчання та побудова потоку даних — як вхідних, так і вихідних, а також створення інтерфейсу для взаємодії, який буде відповідати всім потребам продукту.

3) Використання навченої моделі

Найпопулярніший і найрелевантніший варіант використання OCR/ICR або інших технологій розпізнавання символів. Цей підхід не потребує власного навчання, оскільки модель уже натренована на великій кількості даних, що гарантує високу якість розпізнавання. Всі витрати на навчання та утримання моделі бере на себе вендор, який надає уніфікований інтерфейс для взаємодії з моделлю, що спрощує використання для всіх користувачів. Часто моделі мають алгоритми самонавчання, що дозволяє покращити їх роботу за рахунок збільшення обсягу даних. Такі рішення пропонуються великими IT-компаніями, є безкоштовними до певного ліміту використання та дуже стабільними й точними.

2 ОБҐРУНТУВАННЯ ВИБОРУ ТЕХНОЛОГІЙ

2.1 Вибір мови програмування

Початком розробки програмного забезпечення є вибір мови програмування, на якій буде проводитися розробка. Основуючись на аналітиці (рис. 2.1), яку збирає популярна платформа DOU, можна дослідити, які мови програмування зараз є популярними.

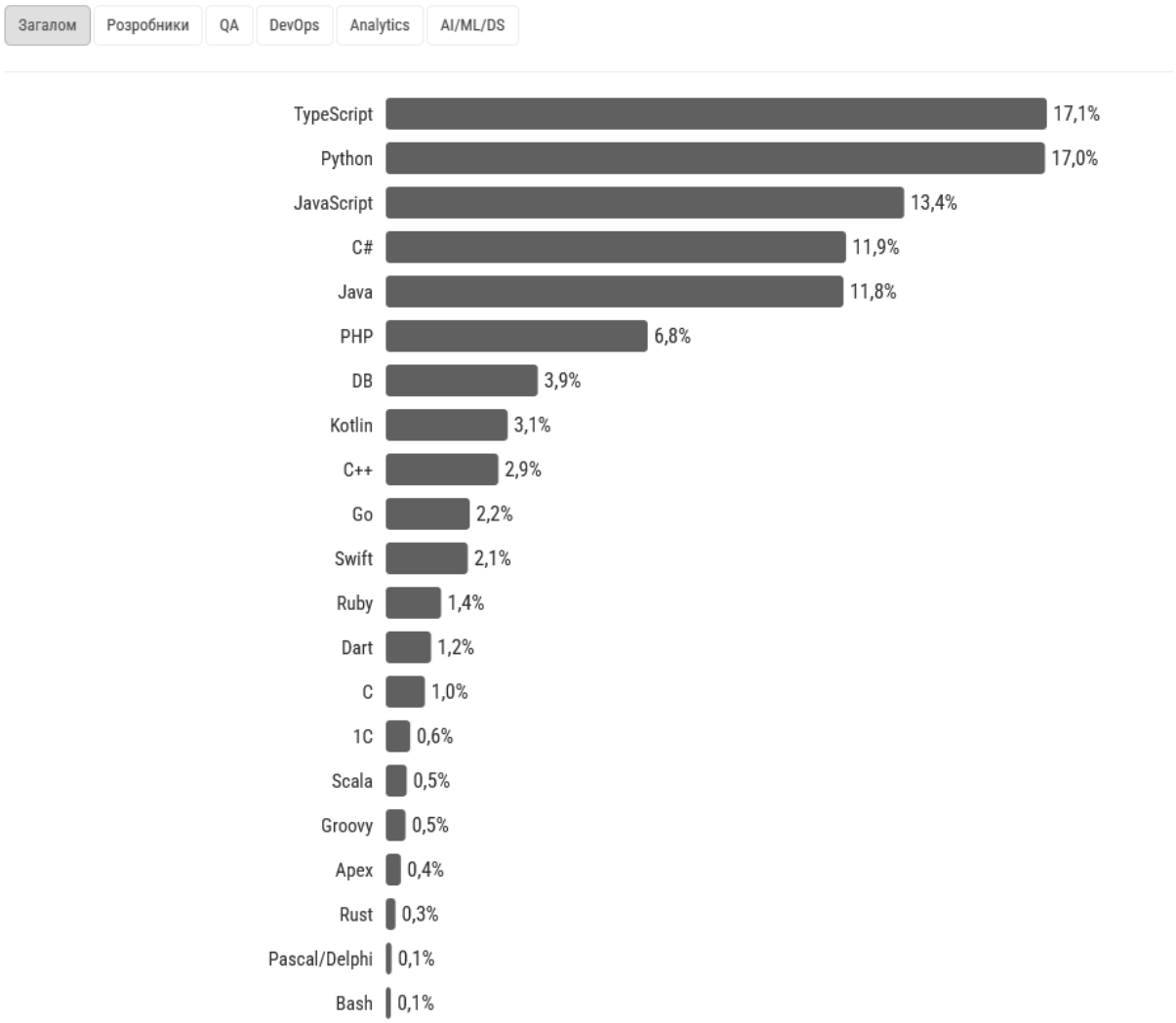

Рисунок 2.1 — Інфорграфіка мов програмування на платформі DOU

Проаналізувавши цей графік, який формується на основі популярності мов програмування в категорії «Загалом», найпопулярнішою мовою програмування є TypeScript, другою за популярністю є Python, на третьому місці — JavaScript. Категорія «Загалом» включає в себе усі сфери, які відносяться до програмування, наприклад, «Тестування», «Аналітика», «Операційна». Якщо звернутися до того ж

провайдера інформації, але, наприклад, у розділ «Розробники», то ми отримаємо вже іншу картину (рис. 2.2):

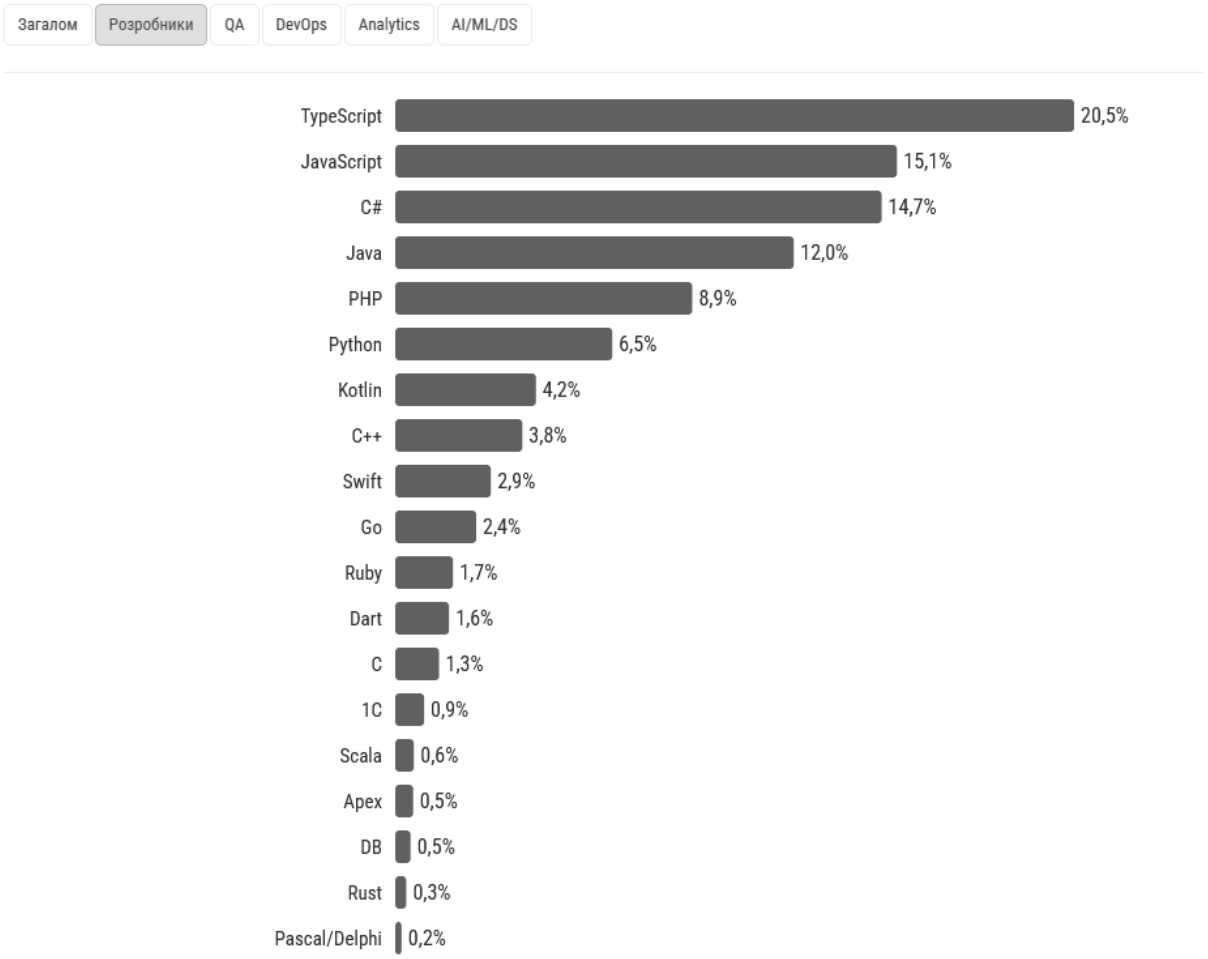

Рисунок 2.2 — Інфорграфіка мов програмування на платформі DOU, розділ «Розробники»

Тут видно, що перша трійка змінена. Перше місце досі займає мова програмування TypeScript, наступне після неї — JavaScript, а останньою в трійці є C#. Отже, можна зробити висновок, що, наприклад, мова програмування Python, яка займала друге місце у загальному топі популярності, не є настільки популярною серед розробників, але при цьому є у двійці лідерів у категорії “QA”. Водночас важливо уточнити, що така ситуація спостерігається на ринку надання інформаційних послуг локально в Україні. Якщо звернутися до міжнародної практики, то там ситуація істотно відрізняється від тієї, що була представлена

вище. Портал index.dev, який також є інформаційним ресурсом, що займається дослідженнями тенденцій у спільноті програмістів, показує наступну статистику:

Таблиця 1.1 — Список популярності мов програмування з порталу index.dev

Feb 2025 Ranking	Feb 2024 Ranking	Programming Language	Ratings	Ratings Change
1	1	Python	23.88%	+8.72%
2	3	C++	11.37%	+0.84%
3	4	Java	10.66%	+1.79%
4	2	C	9.84%	-1.14%
5	5	C#	4.12%	-3.41%
6	6	JavaScript	3.78%	+0.61%
7	7	SQL	2.87%	+1.04%
8	8	Go	2.26%	+0.53%
9	12	Delphi/Object Pascal	2.18%	+0.78%
10	9	Visual Basic	2.04%	+0.52%

У таблиці 1.1 чітко видно, що мова програмування Python є абсолютним лідером серед усіх мов програмування станом на лютий 2025 року.

Чому світова тенденція так сильно різниться від тієї, що є в Україні? Обґрунтуванням цього явища є те, що у різних регіонах і країнах пропагуються різні підходи до використання різних мов програмування. На такі тенденції впливає все — починаючи від розвитку спільнот мов і закінчуючи ментальністю людей у певній країні чи регіоні. Наприклад, рівень розвитку сфери інформаційних технологій у Сполучених Штатах Америки є дуже високим. Країна виділяє великі кошти на розвиток умов та інфраструктури для того, щоб ця сфера розвивалася, створюється багато просторів, в яких люди з такими інтересами можуть проводити час, розвиваються університети та осередки навчання, завдяки чому кількість спеціалістів постійно зростає. Відповідно, спільнота цього регіону є дуже різноманітною та всебічно розвинутою, за рахунок чого перелік технологій та мов, які там популярні, також є широким.

Ситуація у Західній Європі є кардинально протилежною. Ментальність людей, які живуть у країнах старої Європи, таких як Іспанія, Франція, Португалія, є доволі консервативною та має дещо ксенофобічний характер. Відповідно до цього нові технології, які з'являються у світі, там не так швидко набувають популярності. Про це свідчить наступна інфографіка (рис. 2.4). Згідно з інформацією, наданою порталом statista.com, у тій частині світу ситуація виглядає наступним чином:

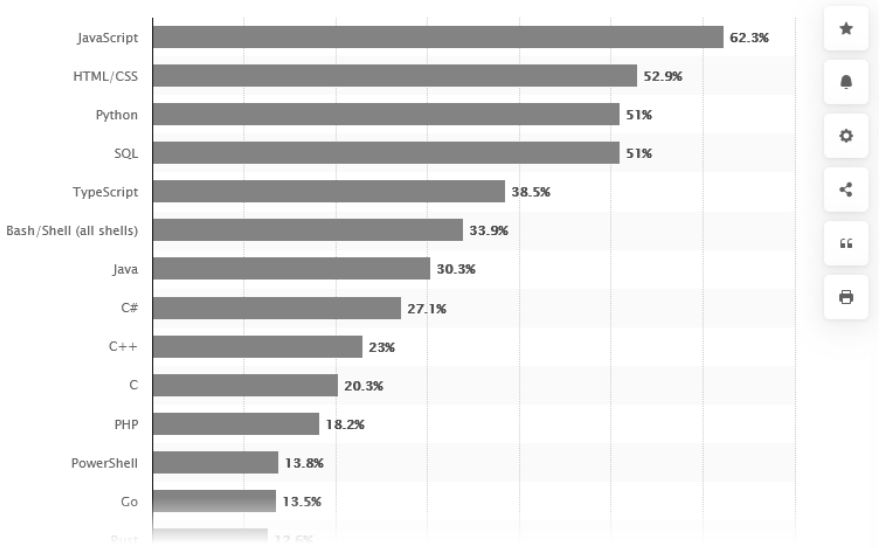

Рисунок 2.4 — Статистика популярності мов програмування в Європі

Найпопулярнішою мовою програмування там є JavaScript, який уже дуже давно зайняв свою частину ринку програмного забезпечення і сам по собі є доволі старою мовою програмування. Мова програмування Python посідає третє місце, хоча вона є більш новою та сучасною.

Окрім того, яка мова програмування популярна у певному локальному середовищі, не менш важливо мати впевненість у тому, що обрана мова програмування повністю забезпечує усі необхідні вимоги, які встановлюються на етапі проектування системи. Найбільш правильним підходом до вибору мови є інженерний. Інженерний підхід диктує те, що не потрібно обирати мову в залежності від своїх персональних преференцій, а з точки зору того, що найкраще підходить для реалізації усіх потрібних функціональностей.

Серед варіантів для виконання роботи є такі мови програмування:

1) TypeScript
2) JavaScript
3) C#
4) Python
5) Java

Кожна з цих мов програмування має як свої переваги, так і свої недоліки.

Для ефективного порівняння потрібно мати список того, що система повинна виконувати, і яким чином вона це повинна робити. Після цього ці списки можна співставити з відповідним списком кожної окремої мови програмування.

У контексті системи, що розробляється, є такі вимоги:

— ефективність роботи системи;

Система повинна ефективно опрацьовувати запити користувачів, щоб забезпечити найкращий користувацький досвід та швидко повертати результати обробки.

— надійність системи;

Система повинна мати алгоритми, за допомогою яких вона буде опрацьовувати непередбачувані ситуації, які можуть виникати в процесі виконання корисної роботи. Також це впливатиме на зручність роботи із системою, оскільки система зможе направляти користувача на те, що саме пішло не так у процесі виконання, та повідомляти про це користувача.

— елегантність виконання рішення;

Система повинна бути виконана таким чином, щоб мати повний набір необхідних методів, які їй потрібні для роботи, але при цьому не бути перенавантаженою синтаксичними конструкціями, які потрібні для виконання роботи. Це впливатиме також на простоту розробки та надійність системи, оскільки прямо пропорційно з тим, наскільки просто система виконана, буде зростати її прозорість, а відповідно — шанс допустити помилку також зменшиться.

— наявність готових рішень для виконання роботи;

Суттєвий вплив на те, наскільки добре буде працювати системний комплекс загалом, має те, чи потрібно програмісту створювати рішення з нуля або ж можна використати вже перевірені готові рішення. Це дозволяє скоротити час на

створення системи, а також покращує простоту системи, коли використовуються уніфіковані методи виконання роботи.

— здатність імплементувати нові рішення;

Якщо трапляється ситуація, в якій не вдається підібрати готове рішення через його відсутність або якщо таке рішення не може забезпечити увесь набір функціональності, чи реалізація не є достатньо якісною, повинна бути можливість створити своє кастомне рішення, за допомогою якого усі критерії будуть унаслідувані.

Співставивши усі вимоги з переліком аспектів кожної з представлених мов програмування, було обрано мову програмування Python. Ця мова програмування має велику спільноту програмістів в Україні та в світі в цілому, за рахунок чого існує велика кількість готових рішень, гайдів, документації та прикладів виконання того чи іншого програмного рішення. Таким чином, процес розробки програми буде оптимізовано, система буде реалізована якісно, з використанням готових рішень та найкращих підходів, які диктує спільнота. Це позитивно впливатиме на процес розробки, процес розширення та підтримання програми, а також, за потреби, сторонні спеціалісти зможуть унаслідувати готову систему для визначення власного вектору її розвитку.

Python — це мова програмування високого рівня, яка є мовою програмування з нестрогою типізацією. Це означає, що змінні, які широко використовуються в процесі виконання корисної роботи програмою, не мають конкретно означеного типу даних, які зберігаються всередині змінної. Тобто, одна й та сама змінна, яка є виділеною частиною оперативної пам’яті комп’ютера, на якому працює програма, може змінювати тип, наприклад, спочатку бути “str”, що відповідає рядковому значенню, а потім та сама змінна може бути переназначена з типом “int”, який відповідає цілочисловому значенню.

Також важливою особливістю мови програмування Python є те, що це мова програмування, яка є інтерпретованою, тобто екземпляр програми виконується поступово, перетворюючись у низькорівневий код у процесі виконання. Якщо, наприклад, програма містить 10 інструкцій, то інтерпретатор буде переводити

десять разів по одній інструкції в низькорівневий код. На практиці це означає, що ймовірність того, що інструкція не виконається прямо в ході виконання програми, є вищою, ніж у мовах програмування, які компілюються. Тому важливо приділяти належну увагу тому, яким чином обробляються помилки, та створити умови, при яких будуть виконуватись принципи консистентності програми. Наприклад, поганим тоном є надання одній змінній різних типів у процесі виконання. Не дивлячись на наявність такої можливості, це дуже не рекомендується робити, оскільки втрачається консистентність програми, що підвищує ймовірність виникнення помилок під час виконання програми.

Натомість, незважаючи на проблеми, які можуть виникати через неправильний режим взаємодії із змінними та їх типами даних, Python має велику кількість переваг, які відіграють важливу роль у тому, що було обрано саме цю мову програмування. Наприклад, простота мови програмування надає лаконічність рішення, коли мова програмування не вимагає створення великої кількості синтаксичного цукру для виконання програми. Завдяки цьому виконується вимога до простоти рішення і дозволяється використовувати лише ті синтаксичні конструкції, які потрібні у конкретному випадку.

Іншим фактором, який також грає важливу роль, є те, що Python є дуже легким для дистрибуції завдяки своїй природі. Для запуску програми, створеної на цій мові програмування, достатньо мати лише встановлений інтерпретатор на комп'ютері. Таким чином, розгортання такого рішення на сервері дозволяє легко переміщати його з одного сервера на інший, при цьому сервер, як віддалена машина, не потребує великої кількості потужностей для того, щоб виконуватися.

Python є найпопулярнішою мовою програмування у світі. Переважно свою популярність він отримав завдяки зручності для створення різноманітних рішень у домені AI та аналізу даних, але сфера веб-розробки також є дуже популярною в рамках цієї мови програмування. Завдяки цьому існує велика кількість інструментів, які дозволяють працювати з різними рішеннями та реалізовувати все на базі однієї платформи, що, у свою чергу, гарантує сумісність всього комплексу технічного рішення.

2.2 Вибір інфраструктурних елементів

Перш ніж починати розробку програмного коду безпосередньо, варто вирішити, як саме буде використовуватись рішення: розгортання локально, на віддаленому віртуальному сервері, у рамках віддаленого робочого столу або на виділеному комп'ютері. Розуміння цього аспекту системи дозволить підібрати найкращу сумісність апаратного та програмного забезпечення, що забезпечить належні властивості системи для розгортання в тому чи іншому середовищі, а від цього, у свою чергу, залежатиме вектор того, як буде розроблятись система.

Планується використовувати всю систему у розрізі розгортання її на базі віртуального виділеного серверу. Таке рішення обумовлене потребою мати постійний доступ до сервісу з будь-якої географічної точки світу. Таким чином, інфраструктура повинна забезпечити простоту рішення для розгортання. Відповідно, потрібно обрати такі рішення, які дозволять виконати цю вимогу.

Існує великий перелік можливостей для реалізації такої інфраструктури. Наприклад, віртуалізація дозволяє забезпечити виділене середовище, яке буде мати увесь набір інструментів для роботи програми. Серед опцій, які доступні для виконання цієї вимоги, є: WSL, VMware, AWS S3, Google Cloud Compute Engine. Деякі рішення є безкоштовними, інші потребують додаткових витрат. Умовно, усі перелічені технології можна поділити на хмарні та не хмарні.

До хмарних рішень входять AWS S3, Google Cloud та Cloud Compute Engine. Це сучасні системи, які дозволяють розгортати програми в хмарі, що дає велику кількість переваг. Наприклад, такі рішення є надзвичайно варіативними в плані фінансових вкладень на потужності. Обидва хмарні провайдери пропонують принцип “Pay-as-you-go”, що відповідає формату плати тільки за ті ресурси, які використовуються програмою. Система сама буде перераховувати еквівалент використання певних потужностей із тривалістю їх використання та на основі похвилинної або погодинної вартості стягуватиме плату за використання платформи. Це є надзвичайно ефективним з точки зору фінансів, бо таким чином буде оплачуватись тільки ті ресурси, які були використані.

Іншою перевагою є здатність таких систем адаптуватися до навантаження. Завдяки глобальності сервісів неможливо уявити ситуацію, в якій система відмовить через велике навантаження на серверну частину. Якщо порівняти два сценарії використання програми: перший передбачає використання виділеного сервера з обмеженими операційними комп'ютерними ресурсами, а інший варіант — використання ресурсів платного провайдера, то при раптовому збільшенні навантаження через різке збільшення кількості користувачів платформи система, розгорнута в хмарі, з більшою ймовірністю впорається з критичним піковим навантаженням. Вона зможе задіяти більше ресурсів, ніж було зазначено в обраному тарифі. Наприклад, за потреби, система хмарного розподілу навантаження може додати кількість оперативної пам'яті, потоків процесора або навіть кількість екземплярів програми, що будуть розподіляти навантаження. Завдяки цьому ймовірність відмови програм, розгорнутих у хмарних рішеннях, менша, ніж у випадку з чітко виділеними ресурсами.

Натомість, такі системи є складнішими в налаштуванні, і первинний вибір системи створює стійкий зв'язок між програмою та інфраструктурою. Тобто, обравши початковим хмарним провайдером Google Cloud, при бажанні перейти на інший хмарний провайдер, наприклад AWS, потрібно буде переналаштувати інфраструктурні елементи, рівні дозволів, зв'язки між елементами з самого початку, оскільки хмарні провайдери відрізняються методами взаємодії з ними, назвами ресурсів, підходами до керування цими ресурсами та іншими аспектами.

Спосіб розгортання системи на базі віртуального виділеного сервера, у свою чергу, переважно не має можливості швидко залучити надлишкові ресурси, не дозволяє так ефективно використовувати фінансові засоби, але має повну свободу у виборі набору технологій та елементів, які будуть використовуватись. Тобто, у порівнянні з хмарними рішеннями, де програміст або спеціаліст, який відповідає за розгортання системи, повинен працювати виключно в тому розрізі, в якому його позиціонують правила хмарної платформи, виділені віртуальні сервери дозволяють у повній мірі використовувати ті технології, які були задіяні в проекті, а також налаштовувати їх обсяг та властивості відповідно до конкретних вимог. Виділений

віртуальний сервер (VPS) дозволяє не обмежуватися в кастомізації, простіше адаптується під функціонал, який потрібен для роботи, та дозволяє встановлювати власний набір правил, заборон, дозволів і рівнів доступу, що часто є ключовим у виборі інфраструктури.

Розгортання системи на виділених серверах не вимагає конкретного програмного забезпечення, що робить розгортання менш залежним від стороннього ПЗ. Більшість віртуальних виділених серверів мають встановлену операційну систему Linux через її здатність до налаштувань та модифікацій залежно від потреб, а також ефективність використання ресурсів у порівнянні з Windows Server.

Не зважаючи на відсутність вимог до програмного забезпечення, розгортати систему з її безпосередньою інтеграцією в операційну систему вважається поганим досвідом, оскільки у такому випадку програма та операційна система стають взаємопов'язаними, що ускладнює відтворення розгортання та знижує мобільність системи. Для вирішення цього аспекту роботи існує віртуалізація. Програмне забезпечення для віртуалізації, наприклад WSL, VMware або Docker, це програмні застосунки, які дозволяють створити віртуальне середовище для роботи програми в межах поточної інформаційної системи. Тобто, такі рішення дозволяють створити окрему операційну систему всередині основної операційної системи, таким чином розділяючи основну ОС і ту, в рамках якої працює програма. Це покращує мобільність системи, не створює безпосередньої залежності програми від поточної ОС і дозволяє зменшити увагу до сумісності та конфліктів програмного забезпечення. Також це зручно для розробки, бо незалежно від того, чи працює програміст або команда програмістів на Linux, Windows, macOS чи будь-якій іншій ОС, програма скрізь буде повторювати одну й ту саму поведінку, і використовуватиметься одне й те саме супутнє програмне забезпечення.

Усі перераховані не хмарні рішення базово будуються на технології віртуалізації процесів. Наприклад, WSL (Windows Subsystem for Linux), який є ядром операційної системи Linux, вбудованим всередину Windows, що дозволяє запускати програми, призначені для Unix-систем, всередині операційної системи Windows за допомогою створення віртуальної машини всередині хост-машини.

На базі цього будуються різноманітні програми, цільовим призначенням яких є робота з віртуальними машинами, їх створення, управління та взаємодія. Серед найпопулярніших застосунків є Hyper-V та VMware, які дозволяють реалізовувати такі віртуальні машини. Вони надають простий, зрозумілий та надійний інтерфейс для роботи з ними, дозволяють використовувати різноманітні ISO-образи для ручного створення машин, що дає змогу обрати саме той дистрибутив або версію, яка потрібна в конкретному випадку, а також мають заздалегідь готові збірки із певними дистрибутивами, які підходять для виконання різноманітних задач та є оптимізованими для запуску, що суттєво спрощує їх налаштування. Таке програмне забезпечення є більш дружнім до користувача, оскільки має вбудований GUI (Graphic User Interface), який покращує спосіб взаємодії з віртуальними машинами, на відміну від Docker або WSL, де основним методом взаємодії з програмою є CLI (Command Line Interface). Проте, Hyper-V та VMware не є широко розповсюдженими для програмування, а більше орієнтовані на користувацькі потреби: тестування певних застосунків, можливість використання різних інтерфейсів без необхідності встановлювати дві операційні системи фізично на диску та надають можливість потестувати ОС перед переходом на неї. Малу популярність серед програмістів ці програми отримали через те, що фактично не мають можливості програмно взаємодіяти з ними. Для створення образів систем потрібно завантажувати ISO-файли, а для їх конфігурації необхідно вручну налаштовувати всі потрібні аспекти через графічний інтерфейс.

Але для використання можливостей віртуалізації був створений Docker компанією Google. Docker — це програмне забезпечення, яке реалізує концепцію контейнеризації із віртуалізацією, має можливість задавання шаблонів для створення копій середовищ та великі можливості для налаштування віртуального середовища в контейнері. Docker реалізує можливість налаштування підмереж всередині контейнера, між контейнерами, а також між контейнерами та системою, за рахунок чого є дуже зручним для роботи. Основним методом взаємодії з Docker є термінал, але також присутня можливість створювати конфігураційні файли, за допомогою яких можна гарантувати повторюваність конфігурації віртуального

середовища при кожному наступному його створенні. Також існує можливість об’єднувати окремі контейнери в цілісний каскад контейнерів, завдяки чому ефективність використання такого рішення стає ще вищою. Наприклад, якщо є потреба розгорнути систему на базі Docker, яка буде містити базу даних, брокер повідомлень та безпосередньо образ програми, що розробляється, то в Docker це робити дуже зручно. Створення конфігураційного файлу не займає багато часу, а його дистрибуція є легкою. Для описаного вище рішення можна використати технологію Docker Compose, яка дозволить через один інтерфейс керувати усіма програмними компонентами системи. Docker Compose описаної вище системи виглядатиме так:

Рисунок 2.5 — Приклад системи розгорнутої в Docker Compose

Тут можна побачити каскад контейнерів (рис. 2.5), один з яких має назву fastapi-app, всередині якого знаходиться екземпляр застосунку, побудованого на FastAPI, що репрезентує REST API. Для взаємодії з ним було використано port-forwarding (прокидання портів) між фізичним портом хост-системи та віртуальним портом контейнера (рис. 2.6), в якому знаходиться застосунок. Це працює таким чином: на фізичній машині існує перелік відкритих портів мережевого інтерфейсу, віртуальна машина також має такий набір. Однак, оскільки система розгортається всередині віртуальної машини та слухає певний мережевий порт віртуальної машини, звернутись до нього з хост-машини безпосередньо не вийде. Але можна створити зв’язок між мережевим портом віртуальної машини та фізичним портом хост-операційної системи. Встановивши такий зв’язок, з’являється можливість налаштувати взаємодію між REST API та клієнтом, що дозволяє створити зручний інтерфейс для роботи. По такому ж принципу використовується з’єднання з базою даних PostgreSQL, але там port-forwarding виконується на локальну мережу хост-

машини та на віртуальну підмережу для контейнера. Це дозволяє інкапсулювати дані, які передаються мережею, всередині неї, з меншою ймовірністю їх перехоплення, модифікації чи втрати через сторонні процеси.

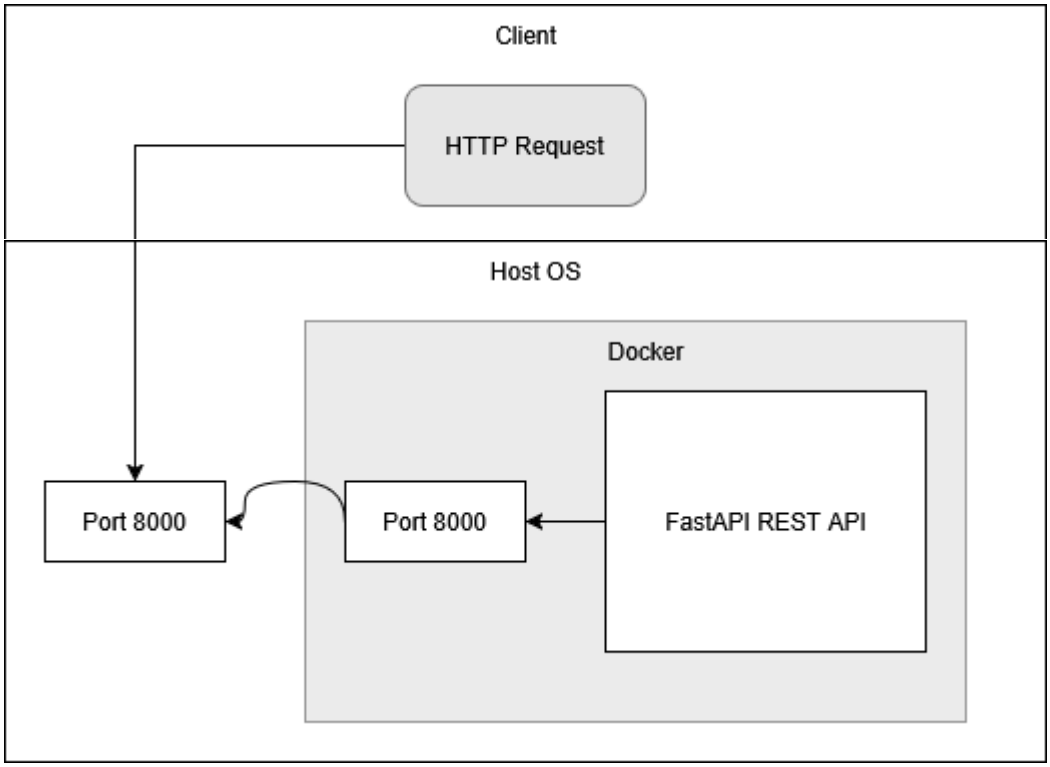

Рисунок 2.6 — Приклад port-forwarding

2.3 Вибір архітектури

Архітектурна організація додатку дозволяє правильно побудувати взаємодію між різними інфраструктурними елементами системи, а також вказує на те, в який спосіб і чим саме ці елементи повинні об’єднуватися.

Розробка програмного забезпечення є дуже багатогранною сферою, в якій для вирішення однієї задачі існує багато варіантів, і при цьому всі вони виконують задачу. Але якщо не використовувати різноманітних підходів до створення програмного забезпечення, на певній ітерації розробка стане дуже важкою через заплутаність коду, строгі залежності та відсутність різних семантичних і синтаксичних конструкцій, які допомогли б покращити розуміння та розширюваність коду.

Сфера веб-розробки є однією з найпопулярніших, тому вона є дуже розвиненою. Відповідно до цього архітектурні підходи, стилі коду, патерни також розвиваються. Вибір код-стилю та архітектури є дуже важливим, і обрати їх необхідно на початку розробки, оскільки перехід від одного до іншого може зайняти дуже багато часу, а іноді адаптація може бути настільки складною, що простіше переписати додаток з нуля, ніж впроваджувати зміни в уже існуючому.

Для застосунку була обрана архітектура “Client/Server” через її варіативність, розділення відповідальностей, простоту та надійність. Архітектура, яка будується за принципом “Client/Server”, передбачає розділення застосунку на дві частини, кожна з яких має свою сферу відповідальності і несе різну функцію відповідно. “Client” — це частина, яка відповідає за взаємодію з сервером. Іншими словами, це інтерфейс, за допомогою якого відбувається взаємодія з логікою застосунку. У розроблюваній системі клієнтом є фронтенд-застосунок, який відображається через браузер. “Server” — це частина застосунку, яка відповідає за прийняття запиту від клієнта, його обробку та повернення відповіді. Таким чином, кожна частина є незалежною від іншої і виконує чітко окреслену частину роботи.

Переваг у такого підходу дуже багато. Перша перевага, яка вже була обговорена раніше — розділення відповідальності. Кожен структурний елемент системи відповідає за свої обов’язки. Наприклад, клієнт майже не має логіки роботи з даними, окрім перевірки їх на наявність і перевірки цих даних на співпадіння з тими, які очікує сервер. Натомість він відповідає за те, щоб ефективно працювати з сервером, бути зручним та варіативним залежно від того, яку відповідь повертає сервер.

Іншою перевагою є те, що кожен застосунок є сепарованим від іншого, що, в свою чергу, покращує процес розробки і усуває строгий зв’язок між застосунками, а відповідно помилки на сервері не зможуть вплинути на клієнт і навпаки.

Також застосунки, побудовані за принципом «Клієнт/Сервер», як правило, є агностичними до клієнтів. На практиці це означає, що серверу не важливо, з яким клієнтом працювати — головне, щоб дані, які сервер отримує від клієнта, були повними. Уявімо ситуацію, в якій розроблювана система будувалася б, наприклад,

за принципом “MVC” (Model-View-Controller). Було б створено єдиний монолітний застосунок, в якому одночасно було б реалізовано графічний інтерфейс для роботи з користувачем та логіка обробки даних, що надаються користувачем. Якби виникла потреба додати не тільки спосіб інтеракції за допомогою браузерного клієнта, а й мобільний застосунок, увесь додаток довелося б переробляти, оскільки він не міг би працювати з обома клієнтами одночасно.

Підхід «Клієнт/Сервер» дозволяє реалізувати концепцію багатьох клієнтів для єдиної логіки. Як уже було сказано раніше, сервер не залежить від клієнта. Більше того, частіше за все сервер не знає, хто є його клієнтом, тому йому не потрібно враховувати особливості роботи з кожним із них. Натомість серверу важливо лише, чи отримав він усі необхідні дані для роботи чи ні. І якщо сервер отримав усі необхідні дані, він їх опрацює та поверне відповідь. Завдяки цьому можна почати зі створення одного клієнта, а за потреби додавати інші, що дозволить розширити цільову аудиторію та забезпечити можливість імплементації для кожного з них різної логіки роботи з користувачем, при цьому не змінюючи серверну логіку, яка буде працювати за однаковими правилами для всіх. Це, у свою чергу, потенційно може збільшити маржинальність, дозволити розширити охоплення потенційних користувачів та потребуватиме значно менших грошових вкладень порівняно з тими підходами, які встановлюють нерозривний зв’язок між двома компонентами системи, об’єднуючи їх в одне ціле. Навіть за умови, що на поточній ітерації немає вимоги мати кілька клієнтів, архітектура — це завжди про те, щоб закласти фундамент для розвитку застосунку наперед, забезпечивши при цьому максимальну простоту для змін та адаптації під потреби. Інтерфейсом на стороні клієнта можуть виступати різноманітні форми представлення. Наприклад, це може бути графічний інтерфейс (GUI), термінал (CLI) або навіть інша програма, яка використовує серверну логіку для створення корисного навантаження.

Коли клієнт збирає дані, які надав користувач, ці дані валідуються, тобто проходять процес перевірки на відповідність очікуваному формату, довжині, значенню тощо, після чого вони передаються на сервер (рис. 2.7).

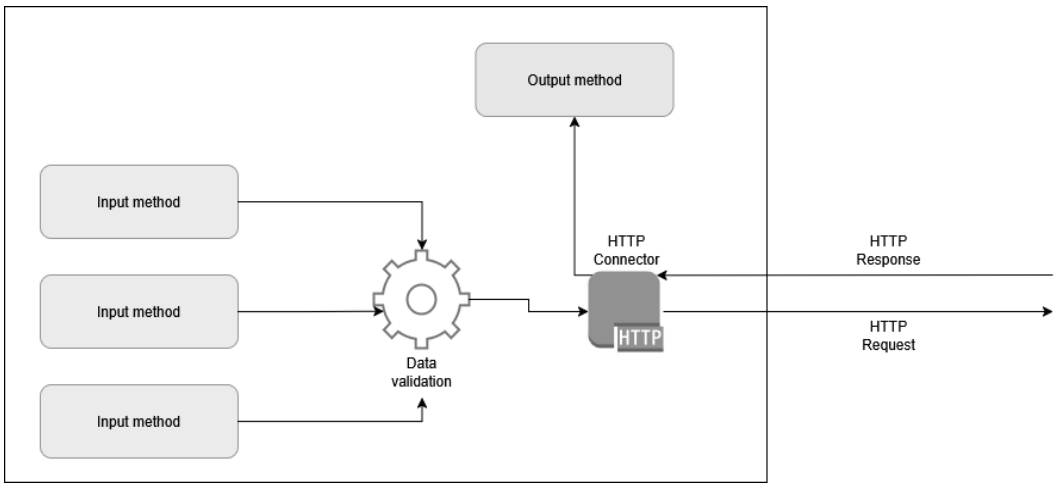

Рисунок 2.7 — Концепція абстаркції “Client”

“Server” — це інша абстракція, яка безпосередньо відповідає за обробку зібраних даних. Переважна більшість серверів — це програмний комплекс, який складається з серверного застосунку (server-side application), HTTP-сервера (HTTP server) та API (Application Programming Interface). Кожна з цих складових виконує таку функцію:

— server side application;

Серверний застосунок — це набір певної логіки, яка реалізується при виклику, реагує на певні умови, події та дані. Він отримує дані у певному форматі від API, після чого декомпозує їх (переводить з складної форми до більш простої та зрозумілої), перевіряє на відповідність очікуваному формату, значенню та іншим аспектам, а потім компонує їх у складнішу структуру і передає на HTTP-сервер.

— HTTP server;

HTTP-сервер — це програма, функцією якої є встановлення зв’язку між програмою, що реалізує логіку, та API. По суті, сервер займається тим, що керує цими зв’язками та налагоджує комунікацію між застосунком і API. Сервер встановлює, як дані приймаються, куди вони передаються після отримання, та як передаються оброблені дані до API. Його сфера відповідальності — це ефективний спосіб передачі даних і менеджмент мережевої взаємодії. Сервер також встановлює, які з’єднання опрацьовувати, а які ігнорувати, які статус-коди (100 Initial Connection, 200 Successfully Processed, 201 Resource Created, 300 Redirect, 301

Permanent Redirect, 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found) повертати у яких випадках, і який формат даних повертати через API.

— Application Programming Interface (API);

API — це мережевий інтерфейс, який реалізує доступ до даних за допомогою мережі. Переважно API використовує HTTP-протокол для зв'язку з сервером. Завданням API є передача даних від клієнта до HTTP-сервера в правильному напрямку, після чого API отримує відповідь і, у правильному форматі з правильним статус-кодом, повертає її тому, хто надіслав запит. Також API інформує клієнта про результат запиту або про помилки, які виникли під час обробки запиту.

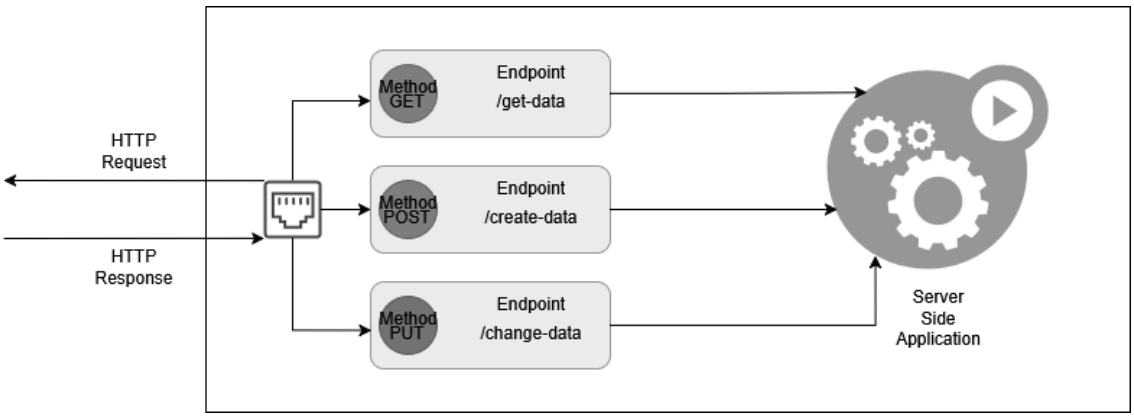

Рисунок 2.8 — Концепція сервер

Налаштувавши взаємодію одного елемента з іншим, формується програмний комплекс (рис. 2.8), який є дуже гнучким, варіативним, без прямих строгих залежностей і при цьому дуже функціональним, з великою кількістю переваг у порівнянні з монолітними системами, які менш придатні для використання в сценаріях, що передбачають активні зміни.

3 РОЗРОБКА КОМП'ЮТЕРНОЇ СИСТЕМИ РОЗПІЗНАВАННЯ ТЕКСТУ НА ОСНОВІ НЕЙРОМЕРЕЖІ

3.1 Структурна схема комп'ютерної системи та її функціональні вимоги

Розроблена система розпізнавання рукописного тексту є інтегрованим програмним комплексом, призначеним для автоматичного вилучення текстової інформації з графічних зображень. Архітектура системи побудована за принципом «клієнт-сервер» з використанням сучасних засобів контейнеризації та хмарних сервісів. Користувач взаємодіє із системою через графічний вебінтерфейс, де здійснює завантаження зображень, після чого виконується обробка зображення на сервері та повернення результату розпізнавання у зручному вигляді.

Опис функціонування системи.

Завантаження файлу. Користувач, використовуючи вебінтерфейс (вебсторінку), обирає зображення (формати JPG, JPEG або PNG), яке містить рукописний або надрукований текст. Після вибору файл передається на сервер за допомогою HTTP POST-запиту.

Обробка на сервері. Сервер, реалізований з використанням FastAPI, приймає файл, генерує унікальну назву для нього (UUID), і передає його у хмарне сховище Google Cloud Storage (GCS). Після успішного завантаження формується посилання на зображення, яке використовується для виклику сервісу Google Vision API.

Використання Google Vision API. Vision API виконує аналіз зображення і повертає результат у вигляді JSON-структури. У ній міститься:

— повний розпізнаний текст;

— мовна локалізація (наприклад, en, uk, pl);

— координати виявлених блоків тексту (не використовуються явно у проекті, але доступні в API).

Формування відповіді. На основі результатів Vision API сервер формує відповідь у спрощеному вигляді: виводиться локальна мова тексту та повний розпізнаний текст. Після цього тимчасовий файл із GCS видаляється.

Відображення результату. Інтерфейс користувача відображає розпізнаний текст у форматованому вигляді, з іконкою прапора країни для зручності. У разі виникнення помилки (наприклад, помилковий формат файлу) система повідомляє про проблему.

Функціональні вимоги до системи.

Функціональні можливості:

— прийом зображень формату PNG, JPG, JPEG;

— передача зображень у хмарне сховище;

— розпізнавання тексту з використанням зовнішнього API;

— визначення локальної мови розпізнаного тексту;

— надання результатів розпізнавання у веб-інтерфейсі.

Вимоги до інтерфейсу користувача:

— мінімалістичний, адаптивний дизайн;

— завантаження файлів через drag-and-drop або форму;

— індикація процесу обробки (завантаження, аналіз);

— виведення результату з можливістю очищення.

Вимоги до продуктивності:

— середній час відповіді не повинен перевищувати 3–5 секунд;

— підтримка одночасної обробки кількох запитів (через асинхронний API).

Безпека та обмеження:

— обмеження на розширення файлів (тільки зображення);

— видалення файлів одразу після обробки;

— валідація даних на клієнтському та серверному рівні.

Вимоги до середовища виконання:

— портативність системи забезпечується за рахунок контейнеризації;

— можливість запуску на локальному сервері або в хмарному середовищі.

Технічна реалізація з точки зору архітектури. Система спроектована з дотриманням принципів модульності, що дозволяє легко масштабувати її функціонал або змінювати окремі компоненти без необхідності повної переробки.

Контейнеризація усуває залежності між середовищами розробки й розгортання, а застосування FastAPI і React забезпечує високу продуктивність та зручність підтримки. Представлена комп'ютерна система розпізнавання тексту базується на сучасних технологіях, включаючи React, FastAPI та хмарні сервіси Google. Користувач завантажує зображення через веб-інтерфейс, після чого файл надсилається на сервер для подальшої обробки. Розроблена діаграма діяльності наведена на рис. В.1 (Додаток В).

3.2 Розробка серверної частини застосунку

При побудові рішень, які передбачають розділення усього програмного комплексу на клієнтську та серверну частини, обидві частини створюються незалежно одна від одної. Однак для налаштування взаємодії однієї частини з іншою потрібно вирішити, яка частина застосунку буде підлаштовуватися під яку. Існує два підходи: "Backend first" та "Frontend first". З назви кожного з підходів можна зрозуміти їх семантичну складову. Для "Backend first" використовується підхід, згідно з яким частина серверної логіки диктує набір правил, за якими з нею повинен працювати клієнт, а клієнт адаптується під вимоги сервера. Тобто умовно можна сказати, що спочатку реалізується серверний інтерфейс, який опрацьовує дані, після чого клієнт успадковує набір правил наданого мережевого інтерфейсу та підлаштовується під них. Натомість підхід "Frontend first" пропонує обернену стратегію: спочатку створюється клієнтська частина застосунку, яка визначає, як вона буде взаємодіяти з серверною логікою, після чого створюється сервер з мережевим інтерфейсом, який реалізує логіку роботи, адаптуючись під клієнта.

У системі, що розроблялася, був використаний підхід "Backend first", тобто спочатку створювався сервер. Для побудови сервера, як зазначалося раніше, була використана мова програмування Python, а саме фреймворк (набір бібліотек із зворотною сумісністю) FastAPI, який надає великий вибір сучасних інструментів для створення серверних застосунків з вбудованим HTTP-сервером та набором методів для створення API.

Вибір саме цього фреймворку для створення серверного застосунку пояснюється тим, що FastAPI — це сучасний інструмент, який дозволяє реалізовувати мережеві застосунки з вбудованим HTTP ASGI (Async Server Gateway Interface) сервером Uvicorn, який показує дуже гарні результати використання апаратних ресурсів, є дуже швидким, дозволяє використовувати підхід із декількома екземплярами застосунку як встановлену функцію, має зручний набір програмних методів для побудови серверної логіки та зручний асинхронний синтаксис, що дозволяє реалізувати концепцію реактивного програмування і суттєво покращує швидкість обробки даних.

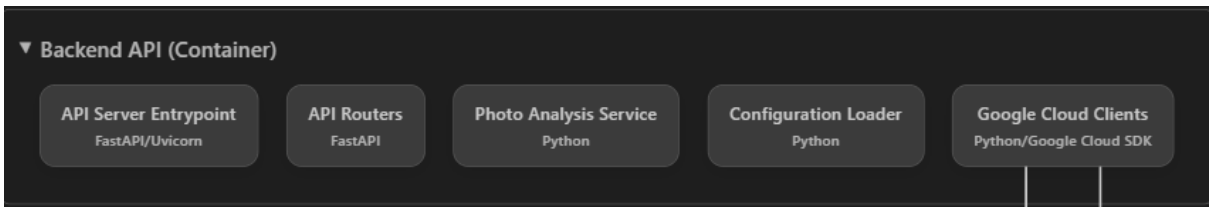

Рисунок 3.2 — Загальний вигляд бекенд частини застосунку

Серверний застосунок являє собою REST API (рис. 3.2) із набором ендпоінтів (кінцевих точок, які являють собою маршрути для взаємодії із застосунком), створений за правилами багато-рівневої архітектури. У рамках цієї архітектури кожен рівень має певну сферу відповідальності. Наприклад, класичною трирівневою архітектурою є застосунок, який має три рівні, кожен із яких є абстрактним поняттям, але при цьому має фізичну реалізацію в коді. У класичній трирівневій архітектурі є три рівні: PL (presentation layer), який є рівнем представлення даних; BLL (business logic layer), на якому розташовані усі елементи, які відповідають за реалізацію бізнес-логіки додатку; та DAL (Data access layer), який відповідає за рівень взаємодії з даними та місцями їх зберігання. У випадку додатку, що розробляється, на поточній ітерації проекту немає потреби використовувати базу даних або будь-яке інше місце постійного зберігання даних, через що в додатку на поточній ітерації є тільки два рівні: PL та BLL, кожен із яких реалізує відповідні сфери відповідальності. При цьому архітектура додатку

ідеологічно підходить під визначення багато-рівневої архітектури, за рахунок чого, при потребі, рівень взаємодії з даними буде легко імплементувати.

На кожному з рівнів розміщуються відповідні абстрактні елементи, які повинні використовуватися тільки в певному контексті задачі. Наприклад, на рівні презентації даних можуть бути розміщені DTO (Data Transfer Object, які умовно можна назвати шаблоном даних, що очікуються для виконання контрактів по ходу виконання роботи програми) та контролери, що по суті є абстрактним поняттям, але в контексті фактичного коду вони є місцем, куди приходять HTTP-запити, де дані з них обробляються та повертається відповідь. На рівні бізнес-логіки знаходяться сервіси, за допомогою яких виконується робота над даними. У фактичній кодовій реалізації це фрагменти коду, які містять набір методів, що обробляють дані та повертають набір простих даних залежно від бізнес-логіки застосунку.

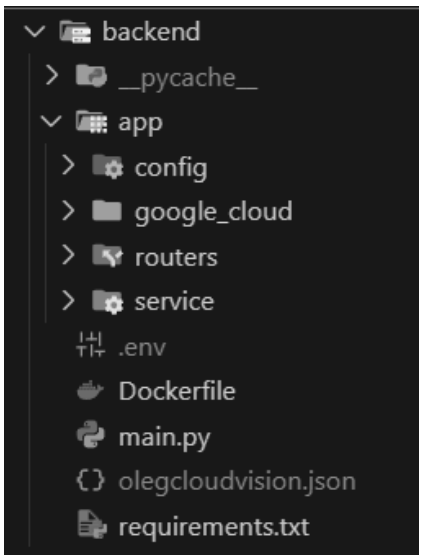

Рисунок 3.3 — Вигляд структури бекенд частини застосунку

На рис. 3.3 зображено структуру застосунку, яка реалізує багато-рівневу архітектуру. В директорії “app”, яка є основною виконуваною директорією всього бекенду, знаходяться папки “routers” та “service”, у яких містяться відповідні структурні елементи застосунку. Окрім цього, проаналізувавши структуру, можна побачити директорію з назвою “google_cloud”, яка реалізує концепцію Middleware.

Middleware — це абстрактне поняття, яке позначає елементи коду, що виконують корисну роботу в процесі виконання програми, але при цьому не є частиною будь-якої іншої структурної одиниці. Тобто це інкапсульовані фрагменти коду, які реалізують певний функціонал, мають свій інтерфейс для взаємодії і можуть бути задіяні незалежно від інших фрагментів абстракцій. У даному випадку middleware являють собою конектори для хмарних сервісів Google Cloud, за допомогою яких відбувається обробка зображень. Алгоритм розпізнавання рукописного тексту виглядає таким чином (рис. 3.4):

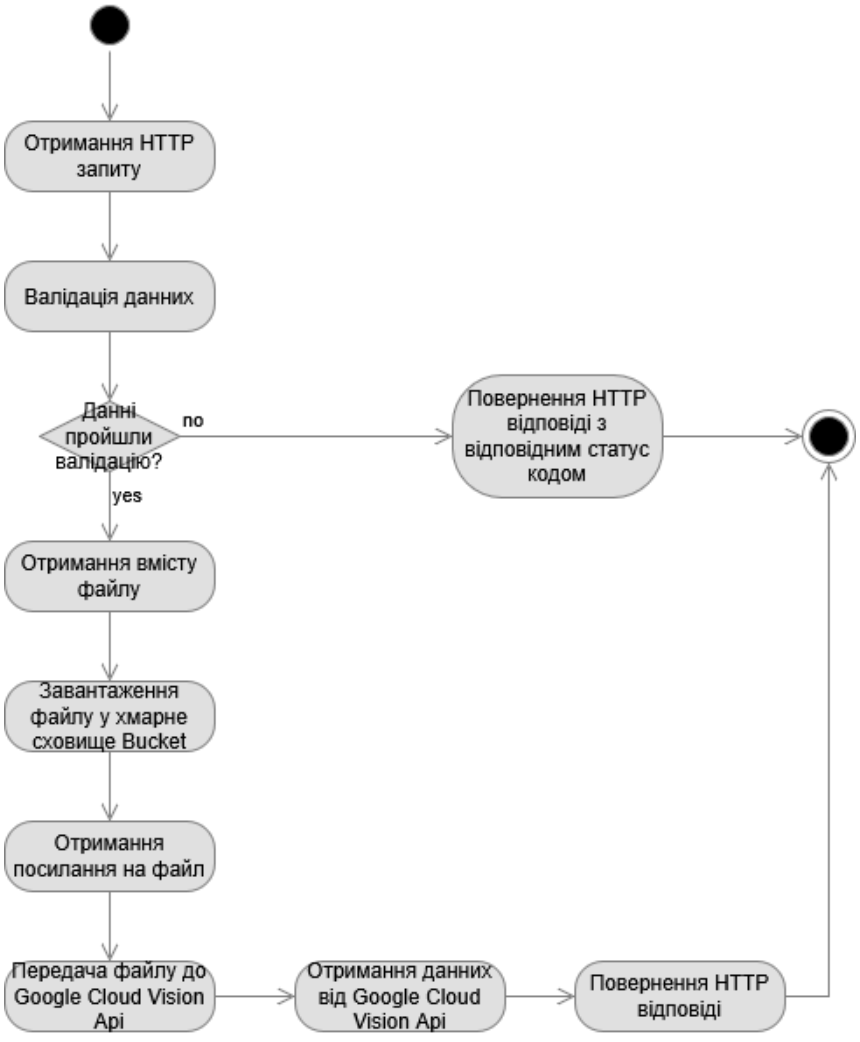

Рисунок 3.4 — Алгоритм розпізнавання рукописного тексту

Для обробки зображень використовується рішення від компанії Google, а саме Google Cloud Vision API. Це рішення було обрано через великий ряд переваг у порівнянні зі створенням власної моделі. По-перше, моделі Google навчаються на

дуже великих обсягах даних, які отримуються з пошукової мережі самого Google, відповідно, модель навчена на найрізноманітніших даних, що дозволяє їй досягати високої точності та добре визначати текст. По-друге, використання моделей штучного інтелекту потребує значних обсягів ресурсів, що може призвести або до втрати ефективності роботи, або до збільшення вартості утримання такої моделі. У свою чергу, Cloud Vision API має ліміт у вигляді 1000 запитів на місяць для безкоштовного використання, а всі обчислення відбуваються на стороні хмарного сервісу, що знімає потребу у великих ресурсах для розгортання цієї системи. Третім фактором є те, що існують різні моделі з широким спектром налаштувань, що дає можливість для розширення функціональності, наприклад, додавання розпізнавання облич або характеристик об'єктів. Четвертим фактором, який посприяв такому вибору, є наявність API, завдяки чому весь процес взаємодії з сервісом здійснюється через обмін даними за допомогою HTTP запитів.

Таким чином, використання Google Cloud Vision API є обґрунтованим рішенням, яке надає великий обсяг функціональності за мінімальних затрат і є легким для інтеграції в поточний проект.

3.3 Розробка клієнтської частини застосунку

Розробивши серверну логіку застосунку згідно з підходом "Backend first", була створена частина застосунку, що реалізує клієнтську частину архітектурного підходу "Client/Server" (рис. 3.5). У якості клієнта на поточній ітерації проекту використовується графічний інтерфейс (GUI), який є веб-застосунком, що працює в браузері і являє собою програму на базі React, за допомогою якої створюється графічний інтерфейс. React — це сучасний фреймворк, побудований на мові програмування JavaScript, який дозволяє створювати браузерні інтерфейси за допомогою єдиного синтаксису за принципом JS-everywhere. Основним операндом у React є модулі, кожен з яких відповідає за певну структурну одиницю інтерфейсу. Коли додаток запускається, код React-застосунку компілюється в стандартний набір HTML, CSS та JavaScript файлів, за допомогою чого створюється браузерна сторінка. Використання саме React було зумовлено тим, що це стандартне

поєднання технологій для розробки фронтенду та бекенду “Python/React Stack”, яке показало свою ефективність при парному використанні та має велику кількість документації, чудово поєднується та стабільно працює. React-застосунок працює на базі Node.js, завдяки чому він успадковує асинхронність і є досить швидким.

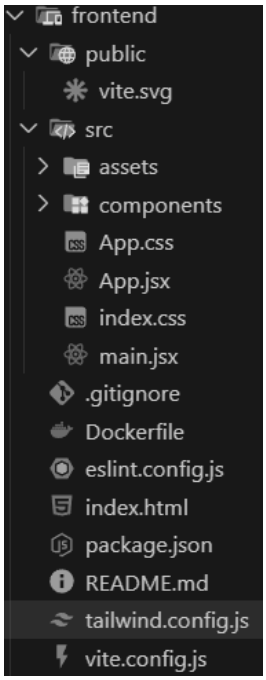

Рисунок 3.5 — Структура фронтенд частини застосунку

Фронтенд-частина реалізує форму для взаємодії з сервером та має певний набір методів вводу та виводу інформації. Наприклад, для завантаження користувачем фото, з якого він хоче отримати рукописний текст, використовується елемент стандартної форми завантаження (рис. 3.6), що дозволяє користувачеві обирати лише файли з розширенням “.png”, “.jpg”, “.jpeg”.

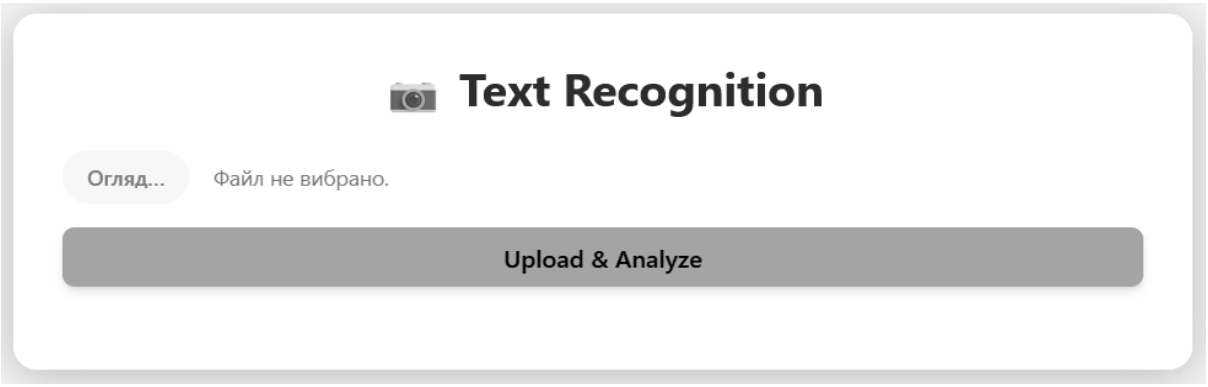

Рисунок 3.6 — Форма завантаження фотографій

Коли користувач натискає кнопку «Огляд», відкривається стандартне вікно файлового менеджера операційної системи. У файловому менеджері будуть відображатися лише файли потрібних розширень. До того часу, поки користувач не завантажить потрібний файл, кнопка “Upload & Analyze” буде недоступною. Якщо користувач завантажить файл, який неможливо обробити, тобто на ньому не буде тексту або програмі не вдасться його зчитати, користувач отримає повідомлення про помилку. У разі, якщо сталася помилка під час виконання запиту для аналізу тексту, користувач також отримає повідомлення про помилку (рис. 3.7).

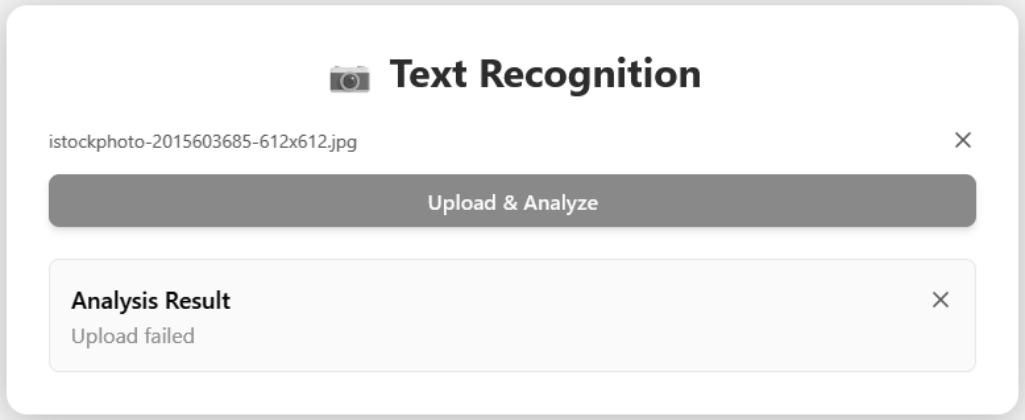

Рисунок 3.7 — Повідомлення про помилку

У разі, якщо користувач завантажив фото з текстом, запит був виконаний успішно, і вдалося отримати відповідь від сервера, користувач отримає вивід зчитаного тексту та код мови, на якій написано цей текст (рис. 3.8). Ці дані знаходяться у відповідній панелі, і кожен з них підписаний відповідно.

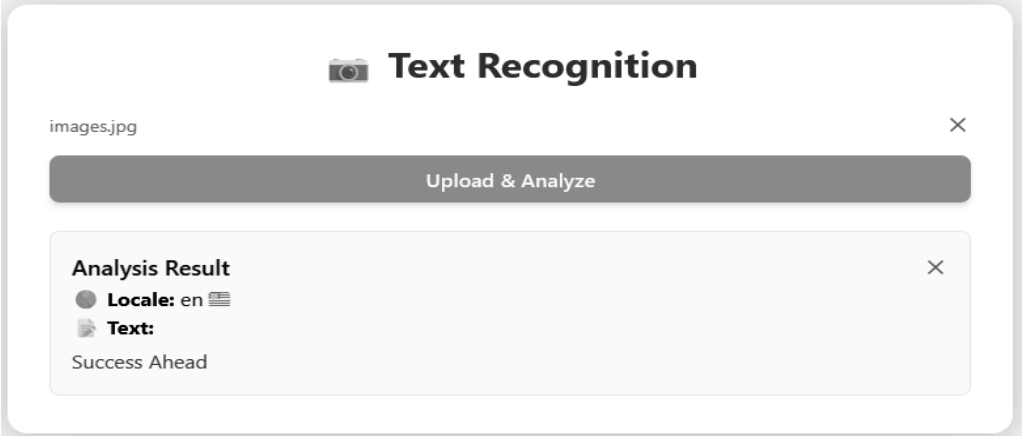

Рисунок 3.8 — Приклад коректного зчитування данних із фотографії

У відповідь на запит клієнта, який отримує сервер, повертається об’єкт у нотації JSON, що містить два ключі: “locale” та “text”. HTTP запит виконується за допомогою вбудованого менеджера запитів AJAX і відправляє form-data запит на адресу “/api/photo/analize”, де у формі знаходиться ключ “file”, у якому міститься файл безпосередньо (рис. 3.9).

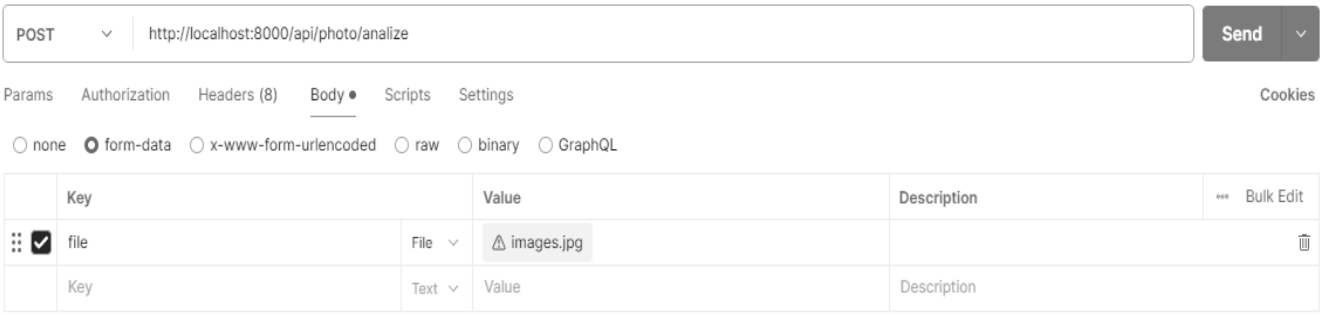

Рисунок 3.9 — Приклад запиту, який виконує клієнт до серверу

У відповідь клієнт отримує JSON об’єкт із текстом та локалізацією або відповідь із запитом (рис. 3.10).

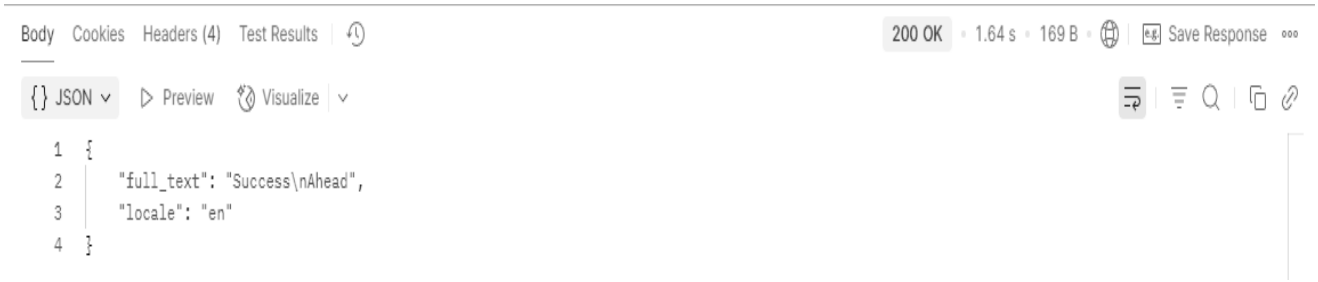

Рисунок 3.10 — Приклад відповіді, який отримує клієнт

3.4 Використання засобів віртуалізації

Для розгортання застосунку використовувалася технологія контейнеризації з використанням програмного забезпечення Docker та підходу Docker Compose, що дозволяє об’єднувати контейнери в каскад контейнерів (рис. 3.11). Каскад складається з двох контейнерів: “fastapi_app”, всередині якого знаходиться серверна частина застосунку, та “frontend_app”, в якому знаходиться клієнтська частина застосунку.

	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
☐	handwrittentextrecognitior	-	-	-	3.24%	24 minutes ago	
☐	fastapi_app	e17c671bf2c2	handwrittentextrecognitior	8000:8000	3.24%	24 minutes ago	
☐	frontend_app	5626a0fbb5c0	handwrittentextrecognitior	3000:80	0%	24 minutes ago	

Рисунок 3.11 — Docker Compose з системою

Файли, за допомогою яких конфігурується вся система, знаходяться всередині основної директорії проекту та мають назви “Dockerfile” та “docker-compose.yml”. Всередині цих файлів містяться конфігурації обох частин системи (рис. 3.12):

docker-compose.yml > {} services > {} frontend

```yaml
# docker-compose.yml - The Compose specification
version: '3.9'

# ▷ Run All Services
services:
  # ▷ Run Service
  fastapi:
    build:
      context: ./backend
    container_name: fastapi_app
    ports:
      - "8000:8000"
    volumes:
      - ./backend:/app
    env_file:
      - ./backend/.env
    environment:
      - PYTHONUNBUFFERED=1
    restart: unless-stopped

  # ▷ Run Service
  frontend:
    build:
      context: ./frontend
    container_name: frontend_app
    ports:
      - "3000:80"
    restart: unless-stopped

```

Рисунок 3.12 — Вміст docker-compose.yml

Головними аспектами цього файлу є такі компоненти:

— version — версія розмітки, за допомогою якої система розуміє, як саме інтерпретувати вміст файлу.

— services — набір контейнерів, які будуть існувати всередині каскаду.

— сервіс fastapi — сервіс, в якому знаходиться увесь бекенд застосунку:

— build — команда, в якій зазначається назва директорії, в яку буде встановлено програму в рамках контейнера.

— container_name — назва контейнера, яка буде відображатися в системі та за допомогою якої буде здійснюватися доступ до контейнера.

— ports — інструкції для port-forwarding. У поточній конфігурації порт 8000, стандартний порт запуску Uvicorn, з мережі контейнера передається на порт 8000 локальної машини.

— volumes — інструкція того, вміст якої директорії локального середовища потрібно передати в яку директорію віртуального середовища. У даному випадку вміст папки “/app” локального середовища передається в папку “/backend” віртуального середовища.

— env_file — вказівка для контейнера, звідки отримати файл “.env” для формування конфігурації.

— restart — вказівка того, коли варто перезавантажувати контейнер у разі збою. Наприклад, у поточній конфігурації встановлене значення “unless-stopped”, що означає, що контейнер буде перезавантажуватися, доки не буде зупинений вручну. Існують також інші значення, наприклад, “never”, що означає, що контейнер ніколи не буде перезавантажуватись, навіть якщо він відмовить.

Для го, щоб із конфігураційних файлів контейнерів та коду, який створено локально, створити каскад контейнерів, які будуть працювати з вмістом у терміналі, відкритому з тієї директорії, де знаходиться конфігураційний файл, потрібно запустити команду docker compose up --build, яка ініціює виконання конфігураційних файлів. Елементами цієї команди є такі складові:

— docker — ключове слово для запуску контейнера.

— compose — вказує, що створюється саме каскад контейнерів, а не окремий образ.

— up — команда запуску контейнера.

— --build — прапорець, який вказує, що контейнер потрібно побудувати. Встановивши цей прапорець, система автоматично перевірить наявність змін у коді чи конфігурації в локальному середовищі. Якщо зміни були виявлені, контейнер буде доповнено ними, але він не буде створюватися з нуля. Якщо це перший запуск контейнера — його буде створено.

Для доступу до сервісів застосунку використовуються такі адреси:

— localhost:3000 — інтерфейс користувача, що дозволяє взаємодіяти із системою через вебзастосунок;

— localhost:8000 — адреса для прямої взаємодії з API.

Крім того, за адресою localhost:8000/docs розміщена документація Swagger, яка містить опис усіх доступних API-методів, їх параметрів, типів відповідей і можливих статус-кодів.

4 ПЕРЕВІРКА ПРАЦЕЗДАТНОСТІ ТА ТЕСТУВАННЯ

4.1 Тестування серверної частини за допомогою use-cases

Для того, щоб провести тестування бекенд-частини застосунку без задіяння фронтенд-частини, потрібно мати можливість надсилання запитів на відповідні контрольні точки (рис. 4.1). У поточній конфігурації можна скористатися сторінкою документації для цього.

Рисунок 4.1 — Приклад виконання запитів зі сторінки документації

На рис. 4.2 видно, що запит виконався, відповідь від сервера прийшла зі статус-кодом 200, що відповідає успішному виконанню. При цьому видно, як формувався запит у секції curl, адресу, куди відбувався запит, та відповідь, яка знаходиться у секції Response.

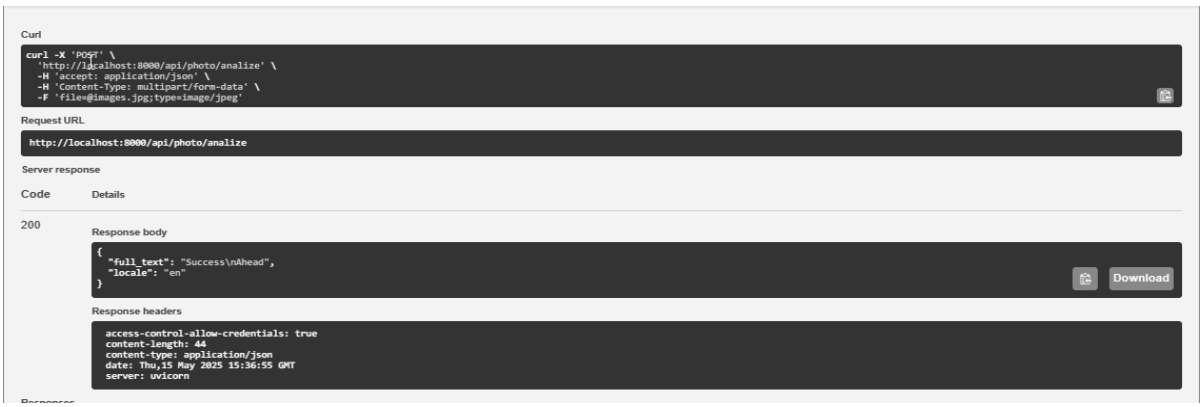

Рисунок 4.2 — Приклад відповіді від серверу

Таким чином, можна провести тестування застосунку, але є ще один варіант виконання запитів за допомогою спеціалізованого програмного забезпечення для веб-тестування — Postman (рис. 4.3).

Postman — це інший HTTP-клієнт, який реалізований на базі того ж curl і дозволяє здійснювати більш точне налаштування запитів, вказувати заголовки до запиту, методи авторизації, налаштовувати віртуальне оточення, з якого буде виконуватись запит, виконувати базові load-тести (тести на навантаження) та ще багато іншого функціоналу.

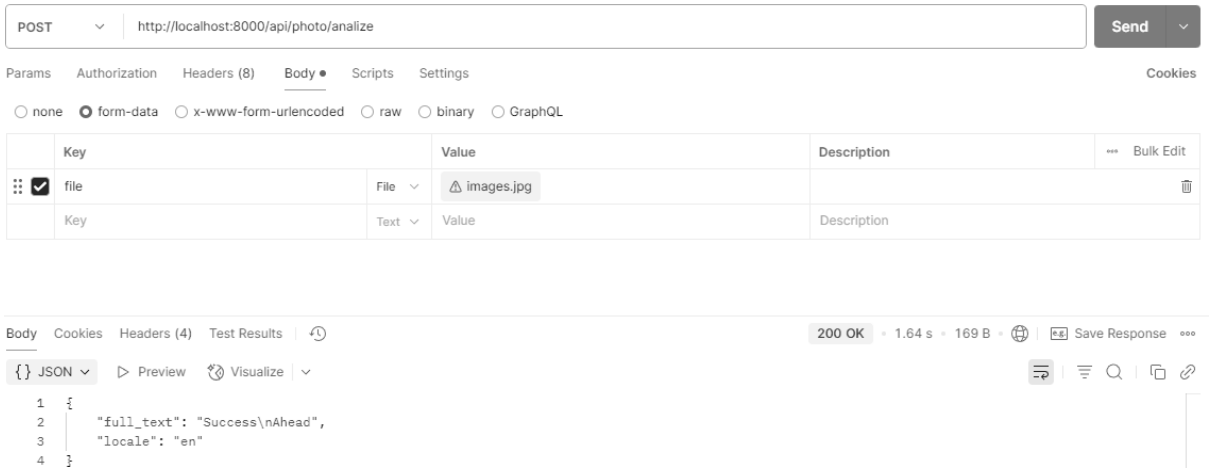

Рисунок 4.3 — Тестування застосунку за допомгою Postman

Виконавши ряд запитів до API за допомогою стороннього клієнта, було виявлено, що вони опрацьовуються у потрібному режимі та успішно повертають відповідь на запит. Тому можна переходити до наступного етапу тестування — “End to End”.

4.2 End to end тестування системи

Провівши тестування окремо серверної частини застосунку, потрібно провести комплексне тестування End-to-End. Такий вид тестування перевіряє роботу усієї системи, а не окремої її частини, завдяки чому можна повторити той процес, який буде виконувати користувач, щоб мати впевненість у результатах роботи системи. Такі тести проводяться на Use Case та обов'язково через той спосіб взаємодії, який передбачений для користувача.

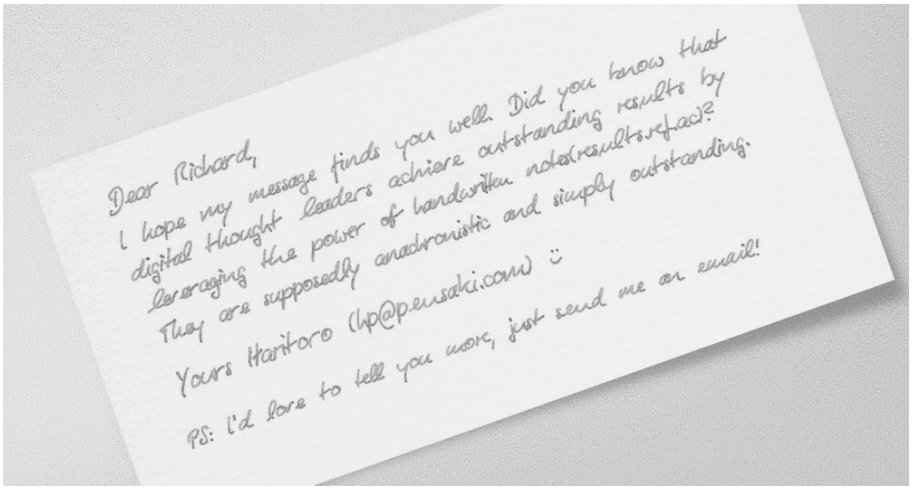

Рисунок 4.4 — Рукописний контрастний шрифт із нахилом

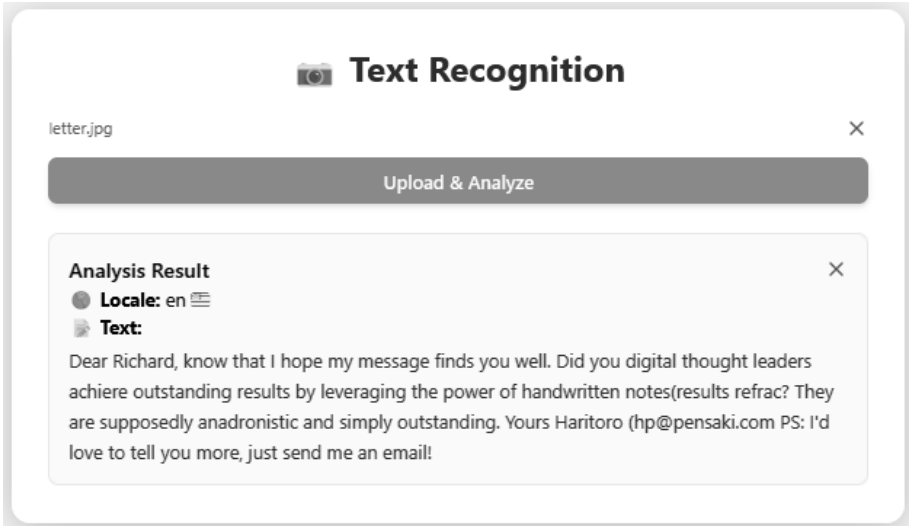

Рисунок 4.5 — Результати опрацювання системою

Рисунок 4.6 — Рукописний шрифт із частковим розмиттям під нахилом

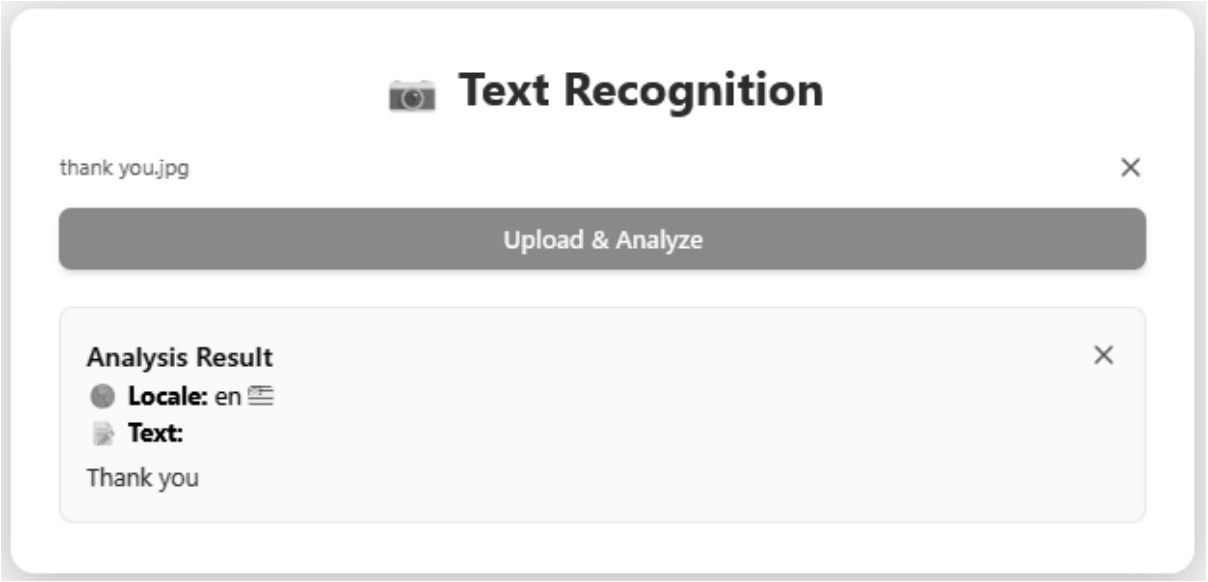

Рисунок 4.7 — Результати опрацювання системою

Рисунок 4.8 — Цифровий рукописний шрифт

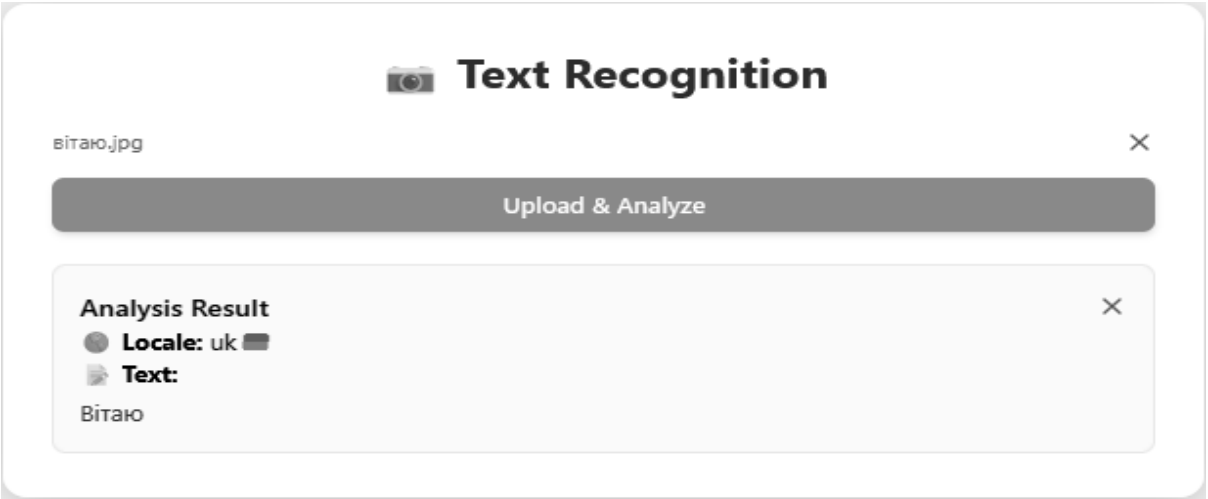

Рисунок 4.9 — Результати опрацювання системою

Провівши тестування системи через користувацький інтерфейс на прикладах рукописного тексту різного формату, контрастності та чіткість зображення (рис. 4.4 - 4.9)., було виявлено, що система чудово працює з різними локалізаціями, чітко визначаючи їх, різною якістю зображення та різного роду артефактами, які присутні на фотографіях. При цьому суттєвої ролі не грає те, чи створено цей текст за допомогою цифрових засобів, чи за допомогою будь-яких інших.

ВИСНОВКИ

У процесі виконання бакалаврської кваліфікаційної роботи було досягнуто поставлену мету — реалізовано програмне рішення, яке дозволяє підвищити точність розпізнавання рукописного тексту за рахунок використання комбінованого підходу, що поєднує технології OCR та нейромережеві моделі, а також розроблено інтуїтивно зрозумілий графічний інтерфейс для взаємодії з системою.

Система забезпечує прийом зображень, обробку їх у хмарному середовищі та повернення розпізнаного тексту користувачу у зручному вигляді. Такий підхід дозволив досягти високої точності навіть у випадках неідеальної якості зображень, що містять рукописні символи, завдяки послідовному використанню оптичного розпізнавання та інтелектуального аналізу тексту.

У першому розділі було здійснено аналіз предметної області, зокрема сучасних підходів до розпізнавання тексту, включаючи технології OCR (оптичне розпізнавання символів), ICR (інтелектуальне розпізнавання рукописного тексту) та глибинного навчання. Проведено порівняльний аналіз цих підходів і обґрунтовано доцільність використання їх комбінації для досягнення вищої точності.

У другому розділі було обґрунтовано вибір технологічного стеку. Для реалізації серверної частини застосовано FastAPI та мову програмування Python — як найбільш придатну для роботи з нейромережевими бібліотеками. Для створення клієнтської частини використано JavaScript-бібліотеку React. Застосування Docker забезпечило портативність і зручність розгортання системи.

У третьому розділі реалізовано безпосередньо архітектуру системи: серверну логіку, яка приймає запити, передає зображення в хмарне сховище Google Cloud та викликає Google Vision API, а також обробляє отримані результати для подальшого відображення. Клієнтський інтерфейс дозволяє користувачеві зручно завантажувати зображення, отримувати результат та взаємодіяти з системою в один клік. У структурі системи передбачено видалення тимчасових файлів після обробки, що підвищує її безпечність і ефективність.

У четвертому розділі проведено тестування системи для оцінки її функціональності, стабільності та точності розпізнавання. Виявлені результати показали високу ефективність системи при обробці зображень різної якості, контрастності та формату, що підтверджує її здатність до точного розпізнавання тексту з різних джерел. Модульне тестування та end-to-end перевірка дозволили виявити можливі слабкі місця й покращити точність роботи системи.

У результаті виконання бакалаврської роботи було:

— розроблено клієнт-серверну систему з можливістю розпізнавання тексту з графічних зображень;

— реалізовано комбінований підхід: OCR + нейромережі, що дозволяє значно підвищити точність розпізнавання рукописного тексту;

— побудовано графічний інтерфейс, який забезпечує зручну та швидку взаємодію користувача з системою;

— забезпечено портативність та простоту розгортання завдяки використанню технологій контейнеризації;

— проведено успішне тестування системи, яке підтвердило її працездатність, ефективність та точність.

Отже, створена система може бути використана як основа для подальшого розвитку програмних рішень у галузі розпізнавання рукописного тексту. Її практичне застосування можливе в таких сферах, як автоматизація документообігу, освітні та медичні системи, а також у мобільних та вебдодатках. У перспективі можливе вдосконалення точності системи через використання сучасніших архітектур глибинного навчання, розширення мовної підтримки та оптимізація для обробки в реальному часі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Goodfellow, I., Bengio, Y., Courville, A. Deep Learning. — MIT Press, 2016. — 800 p.
2. LeCun, Y., Bengio, Y., Hinton, G. Deep learning // Nature. — 2015. — Vol. 521, pp. 436–444.
3. Morgan D. Node.js Web Development. 6th ed. — Packt Publishing, 2023. — 486 p.
4. Russell, S., Norvig, P. Artificial Intelligence: A Modern Approach. — 4th ed. — Pearson, 2021. — 1136 p.
5. Singh K. Mastering Full Stack Development with FastAPI and React. — O'Reilly Media, 2024. — 384 p.
6. Кравченко О. В. Застосування мікросервісної архітектури у веброзробці // Інформаційні технології і засоби навчання. — 2022. — № 6(96). — С. 25–32.
7. Мартинюк Т. Б., Войцеховська О. В., Городецька О. С., Рижков А. К. Модуль інтеграції вебзастосунків із штучним інтелектом // Інформаційні технології та комп'ютерна інженерія. — 2024. — №1. — [Електронний ресурс] — Режим доступу: https://itce.com.ua/web/uploads/journals_pdf/IT_2024_1.pdf
8. Сергієнко І. В., Бондаренко С. А. Архітектура інформаційних систем. — Київ: КНЕУ, 2021. — 284 с.
9. Чалий А. І., Дьяків А. М. Інформаційні системи в інженерії. — Львів: Видавництво НУ «ЛП», 2020. — 204 с.
10. API design and management // TechTarget [Електронний ресурс]. — Режим доступу: https://techtarget.com/searchapparchitecture/resources/API-design-and-management
11. FastAPI. Офіційна документація [Електронний ресурс]. — Режим доступу: https://fastapi.tiangolo.com
12. Tailwind CSS Documentation [Електронний ресурс] — Режим доступу: https://tailwindcss.com/docs
13. Tailwind CSS. Документація [Електронний ресурс]. — Режим доступу: https://tailwindcss.com/docs

14. Документація Docker [Електронний ресурс] — Режим доступу: https://docs.docker.com/
15. Документація FastAPI [Електронний ресурс] — Режим доступу: https://fastapi.tiangolo.com/
16. Документація Python [Електронний ресурс] — Режим доступу: https://docs.python.org/3/
17. Документація React [Електронний ресурс] — Режим доступу: https://react.dev/

ДОДАТОК А

Технічне завдання

Міністерство освіти та науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп’ютерної інженерії
Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
д.т.н, проф. Азаров О.Д.

«21» березня 2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання бакалаврської кваліфікаційної роботи

«Комп'ютерна система розпізнавання тексту на основі нейромережі»

08-54.БКР.012.00.000 ТЗ

Науковий керівник к.т.н., доцент каф. ОТ

_____ Городецька О.С.
(підпис)

Студент групи 1КІ-21б

_____ Левчук О.О.
(підпис)

ВНТУ 2025

1 Підставою для виконання бакалаврської кваліфікаційної роботи (БКР)

1.1 Актуальність виконання бакалаврської кваліфікаційної роботи зумовлена потребою в ефективних інструментах для обробки рукописних текстів. Завдяки розвитку штучного інтелекту, виникає необхідність у створенні систем, здатних точно розпізнавати рукописні документи, що значно спрощує роботу в таких сферах, як юриспруденція, медицина та економіка. Розробка такої системи допоможе автоматизувати обробку текстів та підвищити ефективність роботи з інформацією.

1.2 Наказ про затвердження теми БКР.

2 Мета БКР і призначення розробки.

Мета проекту — підвищення точності розпізнавання рукописного тексту шляхом використання комбінованого підходу, що поєднує технології OCR та нейромережеві моделі, а також створення зручного графічного інтерфейсу для оперативної взаємодії з системою.

3 Вихідні дані для виконання БКР

3.1 Текст, що був розпізнаний і перетворений у цифрову форму, з можливістю подальшої обробки чи аналізу

3.2 Визначення мови тексту

4 Вимоги до виконання БКР

Головна вимога до виконання бакалаврської кваліфікаційної роботи полягає в розробці точної та ефективної системи розпізнавання рукописних символів з використанням алгоритмів OCR та нейронних мереж.

5 Етапи БКР та очікувані результати

Етапи роботи та очікувані результати приведено в таблиці А.1.

Таблиця А.1 — Етапи БКР

№ з/п	Назва етапів дипломної роботи	Термін виконання	Очікувані результати
1	Постановка завдання роботи	21.03.25	Розділ 1
2	Аналіз існуючих методів розпізнавання тексту	24.03.25 – 26.03.25	Розділ 1
3	Вибір технологій для розробки системи	27.03.25 – 29.03.25	Розділ 2
4	Розробка архітектури системи та вибір інфраструктури	31.03.25 – 03.04.25	Розділ 2
5	Розробка бекенд частини системи	04.04.25 – 19.04.25	Розділ 3
6	Розробка фронтенд частини системи	21.04.25 – 26.04.25	Розділ 3
7	Використання засобів віртуалізації	28.04.25 – 29.04.25	Розділ 3
8	Тестування системи	30.04.25 – 03.05.25	Розділ 4
9	Оформлення пояснювальної записки та графічної частини	05.05.25 – 12.05.25	ПЗ, презентація
10	Перевірка якості виконання бакалаврської кваліфікаційної роботи та усунення недоліків	13.05.25	Оформлені документи

6 Матеріали, що подаються до захисту БКР

До захисту подаються: пояснювальна записка БКР, ілюстративні матеріали, протокол попереднього захисту БКР на кафедрі, відгук наукового керівника, анотації до БКР українською та іноземною мовами, довідка про відповідність оформлення БКР діючим вимогам.

7 Порядок контролю виконання та захисту БКР

Виконання етапів графічної та розрахункової документації БКР контролюється науковим керівником згідно зі встановленими термінами. Захист БКР відбувається на засіданні Екзаменаційної комісії, затвердженої наказом ректора.

8 Вимоги до оформлювання та порядок виконання БКР

8.1 При оформлювання БКР використовуються:

— ДСТУ 3008: 2015 «Звіти в сфері науки і техніки. Структура та правила оформлювання»;

— ДСТУ 8302: 2015 «Бібліографічні посилання. Загальні положення та правила складання»;

— міждержавний ГОСТ 2.104-2006 «Єдина система конструкторської документації. Основні написи»;

— Методичні вказівки до виконання бакалаврських кваліфікаційних робіт зі спеціальності 123 «Комп'ютерна інженерія» (освітня програма «Системне програмування»). Кафедра обчислювальної техніки ВНТУ 2022;

— документами на які посилаються у вище вказаних.

8.2 Порядок виконання БКР викладено в «Положення про кваліфікаційні роботи на першому (бакалаврському) рівні вищої освіти СУЯ ВНТУ-03.02.02-П.001.01:21».

ДОДАТОК Б

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Комп'ютерна система розпізнавання тексту на основі нейромережі

Тип роботи: _____ бакалаврська кваліфікаційна робота _____
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ _____кафедра обчислювальної техніки_____
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism __1__%

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

✓ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту

☐ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

☐ У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

_____Азаров О.Д., зав. каф. ОТ _____ _____
(прізвище, ініціали, посада) (підпис)

____Крупельницький Л.В., доц. каф ОТ_____ _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ ___Захарченко С.М.____
(підпис) (прізвище, ініціали)

З висновком експертної комісії ознайомлений(-на)

Керівник _____ _____
(підпис) (прізвище, ініціали, посада)

Здобувач _____ _____
(підпис) (прізвище, ініціали)

ДОДАТОК В

Графічна частина

КОМП’ЮТЕРНА СИСТЕМА РОЗПІЗНАВАННЯ ТЕКСТУ НА ОСНОВІ НЕЙРОМЕРЕЖІ

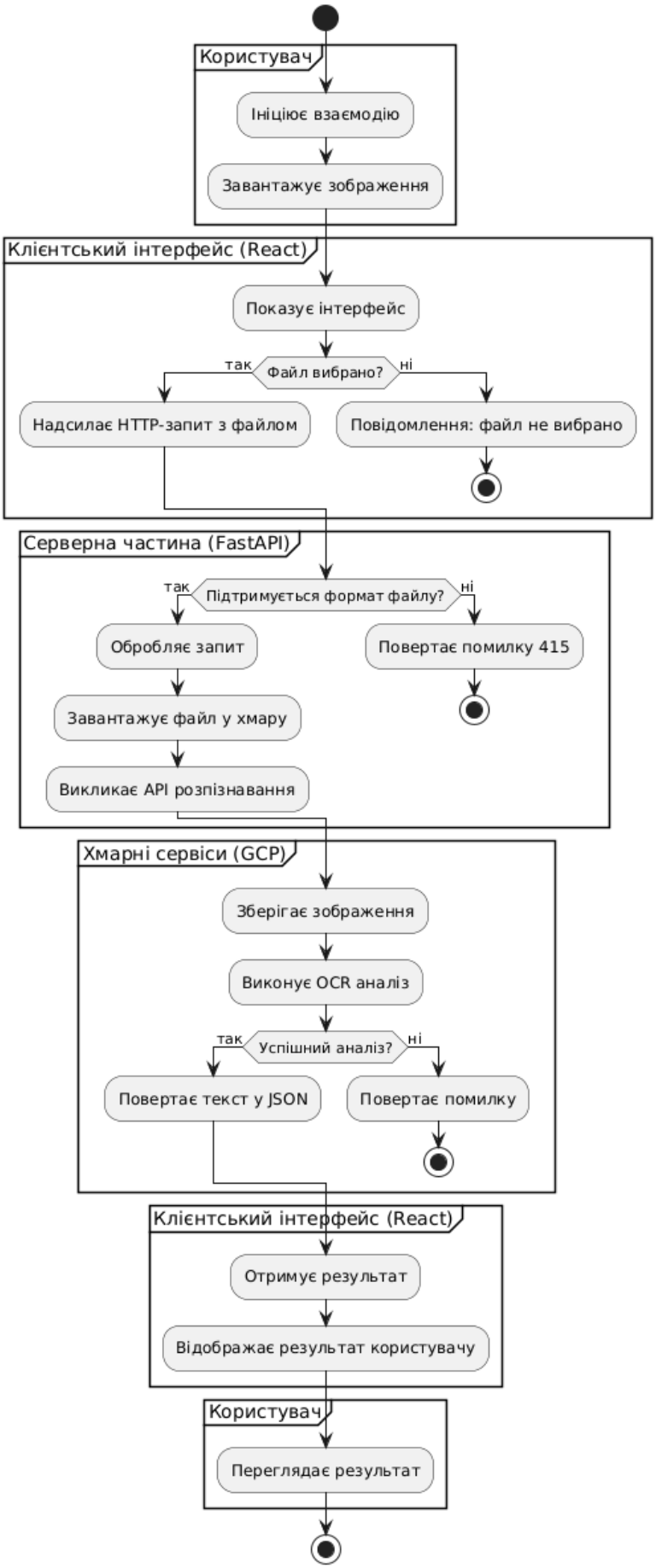

Рисунок В.1 — Діаграма діяльності комп’ютерної системи розпізнавання тексту

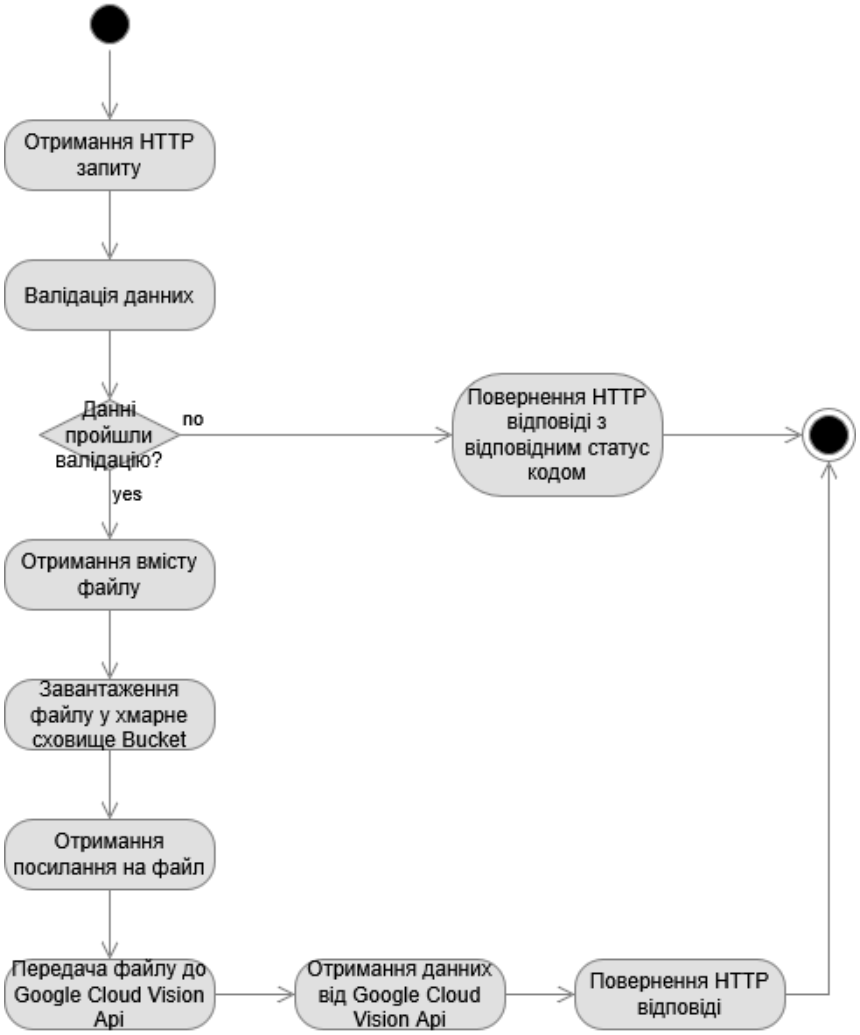

Рисунок В.2 — Алгоритм розпізнавання рукописного тексту

ДОДАТОК Г

Ілюстративна частина

КОМП’ЮТЕРНА СИСТЕМА РОЗПІЗНАВАННЯ ТЕКСТУ НА ОСНОВІ НЕЙРОМЕРЕЖІ

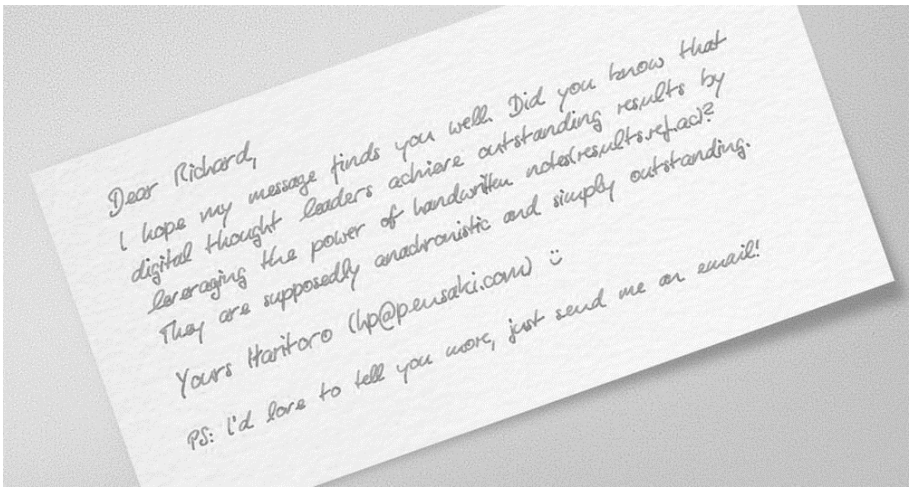

Рисунок Г.1 — Рукописний контрастний шрифт із нахилом

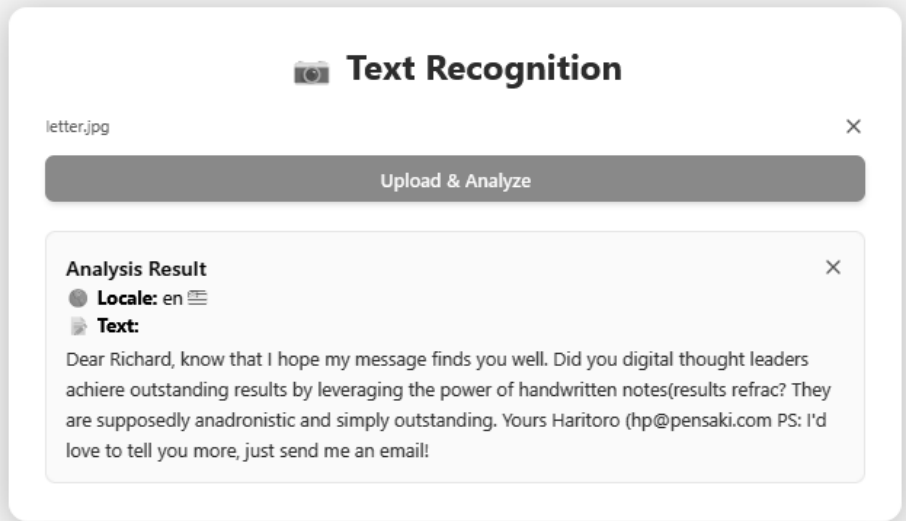

Рисунок Г.2 — Результати опрацювання системою

Рисунок Г.3 — Рукописний шрифт із частковим розмиттям під нахилом

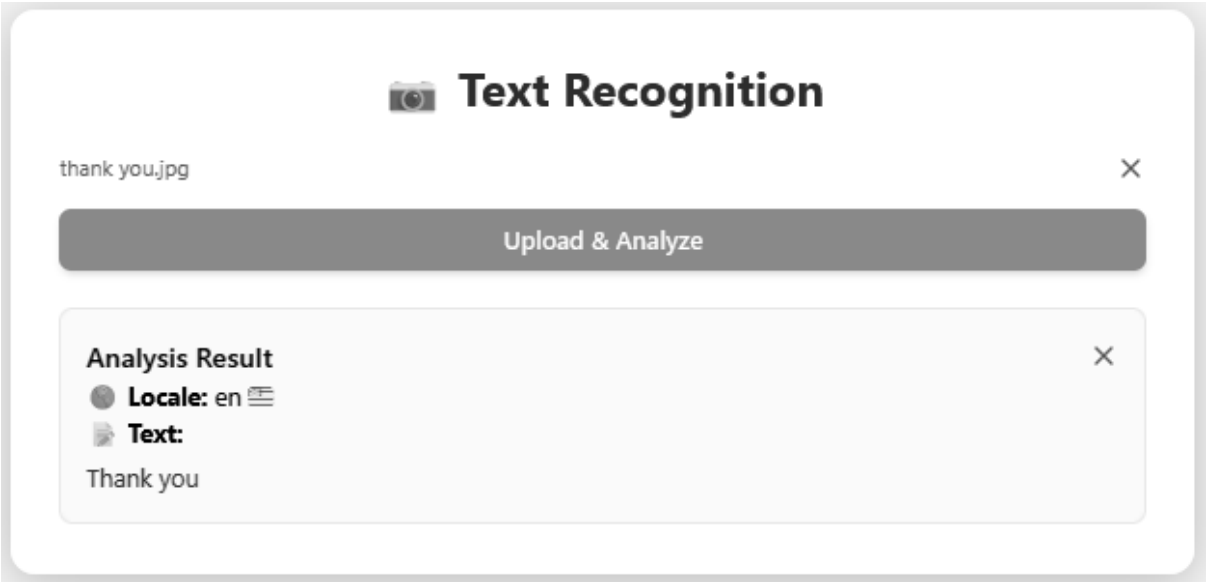

Рисунок Г.4 — Результати опрацювання системою

Рисунок Г.5 — Цифровий рукописний шрифт

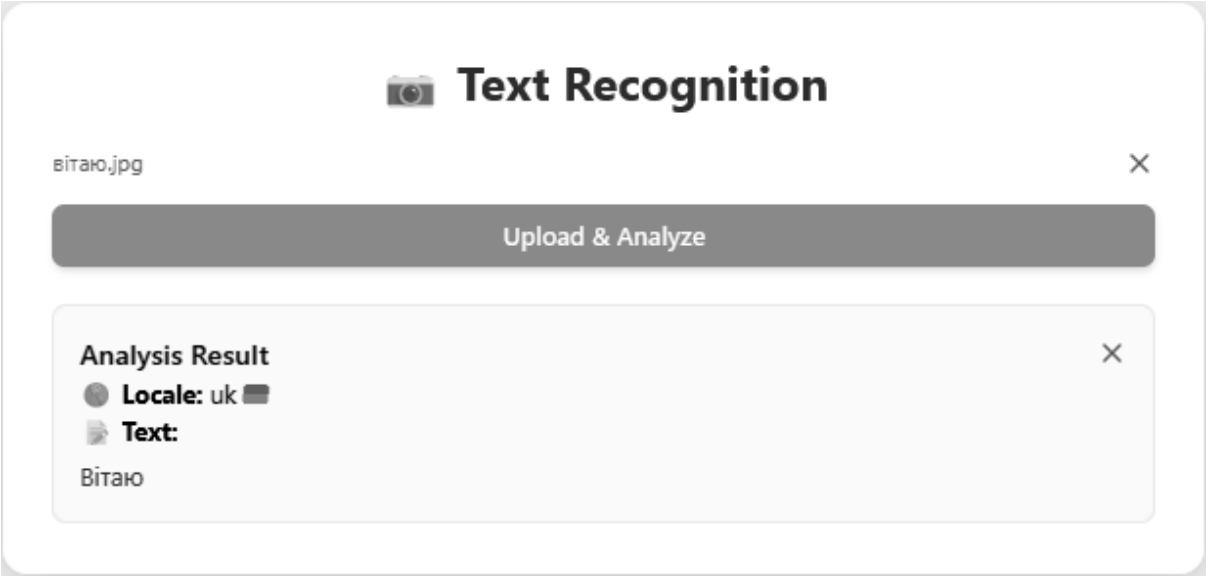

Рисунок Г.6 — Результати опрацювання системою

ДОДАТОК Д

Лістинг програмного забезпечення

Лістинг Д.1 — Головний компонент застосунку “frontend\src\App.jsx”

```
import React, { useState } from 'react'
import axios from 'axios'
import Header from './components/Header'
import FileUploader from './components/FileUploader'
import ResultCard from './components/ResultCard'
function App() {
  const [file, setFile] = useState(null)
  const [response, setResponse] = useState(null)
  const [loading, setLoading] = useState(false)
  const handleUpload = async () => {
    if (!file) return
    const formData = new FormData()
    formData.append('file', file)
    setLoading(true)
    setResponse(null)
    try {
      const res = await axios.post('http://localhost:8000/api/photo/analize', formData, {
        headers: { 'Content-Type': 'multipart/form-data' },
      })
      setResponse(res.data)
    } catch (err) {
      setResponse({ error: 'Upload failed' })
    } finally {
      setLoading(false)
    }
  }
  const resetResponse = () => {
    setResponse(null)
  }
  return (
    <div className="min-h-screen bg-gray-100 flex items-center justify-center p-4">
      <div className="bg-white rounded-2xl shadow-[0_4px_30px_rgba(0,0,0,0.25)] p-8 w-full max-
w-3xl">
        <Header />
        <FileUploader file={file} setFile={setFile} handleUpload={handleUpload} loading={loading} />
        <ResultCard response={response} resetResponse={resetResponse} />
      </div>
    </div>
  )
}
export default App
```

Лістинг Д.2 — Компонент завантаження файлів з попереднім переглядом “frontend\src\components\FileUploader.jsx”

```
import React from 'react'
import { Loader2, X } from 'lucide-react'
const FileUploader = ({ file, setFile, handleUpload, loading }) => {
  const handleResetFile = () => {
    setFile(null)
  }
  return (
    <div className="mb-6">
      {file ? (
        <div className="flex items-center justify-between mt-2">
          <span className="text-sm text-gray-600 truncate">{file.name}</span>
          <button
            onClick={handleResetFile}
            className="text-gray-600 hover:text-gray-800"
            aria-label="Remove file"
          >
            <X className="h-5 w-5" />
          </button>
        </div>
      ) : (
        <input
          type="file"
          accept="image/*"
          onChange={(e) => setFile(e.target.files[0])}
          className="block w-full text-sm text-gray-500 file:mr-4 file:py-2 file:px-4
                  file:rounded-full file:border-0 file:text-sm file:font-semibold
                  file:bg-blue-50 file:text-blue-700 hover:file:bg-blue-100"
        />
      )}      <button
        onClick={handleUpload}
        className={`mt-4 w-full py-2 px-4 font-semibold rounded-lg shadow-md transition
          ${!file || loading ? 'bg-gray-400 cursor-not-allowed' : 'bg-blue-600 hover:bg-blue-700 text-
white'}`}
        disabled={!file || loading}
      >
        {loading ? (
          <span className="flex items-center justify-center">
            <Loader2 className="animate-spin mr-2 h-5 w-5" />
            Processing...
          </span>
        ) : (
          'Upload & Analyze'
        )}
      </button>
    </div>
  )
}
export default FileUploader
```

Лістинг Д.3 — Dockerfile для налаштування середовища та запуску frontend-частини “frontend\Dockerfile”

```
# build stage
FROM node:18-alpine AS build
WORKDIR /app
COPY . .
RUN npm install && npm run build
# production stage
FROM nginx:stable-alpine
COPY --from=build /app/dist /usr/share/nginx/html
EXPOSE 80
```

Лістинг Д.4 — Сервіс аналізу фото за допомогою Google Cloud Vision “backend\app\service\photo_analize_service.py”

```
from uuid import uuid4
from fastapi import File
from app.google_cloud.bucket_client import BucketClient
from app.google_cloud.cloud_vision_client import CloudVisionClient
class PhotoAnalizeService:
    def __init__(self):
        self._bucket_client = BucketClient()
        self._vision_client = CloudVisionClient()
    async def analize_photo(self, file: File, filename: str):
        file_uuid_name: str = str(uuid4()) + filename
        g_bucker_uri: str = await self._bucket_client.upload_file(
            filename=file_uuid_name,
            file=file
        )
        photo_description: list[dict] = await self._vision_client.analize_photo(
            photo_uri=g_bucker_uri
        )
        await self._bucket_client.delete_file(
            filename=file_uuid_name
        )
        response: dict = await self.prepare_response(raw_gc_response=photo_description)
        return response

    async def prepare_response(self, raw_gc_response) -> dict:
        if raw_gc_response:
            full_text = raw_gc_response.text_annotations[0].description
            locale = raw_gc_response.text_annotations[0].locale
        response: dict = {
            'full_text': full_text,
            'locale': locale
        }
        return response
```

Код Д.5 — Маршрут для завантаження та аналізу фото

“backend\app\routers\file_analize_router.py”

```
from fastapi import APIRouter, UploadFile, File, HTTPException
from app.service.photo_analize_service import PhotoAnalizeService
photo_analize_router = APIRouter(prefix='/photo')
ph_analize_service = PhotoAnalizeService()

@photo_analize_router.post('/analize')
async def analize_photo(file: UploadFile = File(...)):
    filename: str = file.filename
    file_extension: str = filename.split('.')[-1]
    if file_extension not in ['png', 'jpg', 'jpeg']:
        return HTTPException(
            status_code=415,
            detail='Unsupported file extension')
    response: dict = await ph_analize_service.analize_photo(
        file=file.file,
        filename=filename
    )
    return response
```

Код Д.6 — Клієнт для аналізу фото через Google Cloud Vision

“backend\app\google_cloud\cloud_vision_client.py”

```
from google.cloud.vision import ImageAnnotatorClient, Feature
from google.cloud.vision_v1.types.image_annotator import AnnotateImageResponse
from google.protobuf.json_format import MessageToDict
from app.config.config_parser import ConfigParser
class CloudVisionClient:
    def __init__(self):
        self._config = ConfigParser()
        self._vision_client = ImageAnnotatorClient.from_service_account_json(
            self._config.GCLOUD_JSON_KEY
        )
    async def analize_photo(
            self,
            photo_uri: str,
            ) -> list:
        image: dict = {
            'source': {
                'image_uri': photo_uri
            }
        }
        features: list = [
            {'type_': Feature.Type.TEXT_DETECTION},
            {'type_': Feature.Type.IMAGE_PROPERTIES},
        ]
        request: dict = {
```

```
            'image': image,
            'features': features
        }
        response: AnnotateImageResponse = self._vision_client.annotate_image(request)
        return response
```

Код Д.7 — Клієнт для взаємодії з Google Cloud Storage

“backend\app\google_cloud\bucket_client.py”

```
from google.cloud.storage import Client
from datetime import datetime
from fastapi import File
from app.config.config_parser import ConfigParser
class BucketClient:
    def __init__(self):
        self._config = ConfigParser()
        self._bucket_name: str = self._config.GCLOUD_BUCKET_NAME
        self._storage_client = Client.from_service_account_json(
            self._config.GCLOUD_JSON_KEY
        )
        self._bucket = self._storage_client.bucket(
            self._bucket_name
        )
        print(f'|{datetime.now()} \t Connected to Bucket: {self._bucket_name}')
    async def upload_file(
            self,
            filename: str,
            file: File
            ) -> str:
        g_blob = self._bucket.blob(filename)
        g_blob.upload_from_file(file)
        file_path: str = f'gs://{self._bucket_name}/{filename}'
        print(f'|{datetime.now()} \t File uploaded with filename: {filename}')
        print(f'|{datetime.now()} \t Bucket file path: {file_path}')
        return file_path
    async def delete_file(
            self,
            filename: str):
        g_blob = self._bucket.blob(filename)
        g_blob.delete()
        print(f'|{datetime.now()} \t File: {filename} have been deleted successfully')
```

Код Д.8 — Dockerfile для налаштування середовища та запуску backend-частини

“backend\Dockerfile”

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --upgrade pip && pip install -r requirements.txt
COPY . .
```

Код Д.9 — Docker Compose конфігурація “docker-compose.yml”

```
version: '3.9'
services:
  fastapi:
    build:
      context: ./backend
    container_name: fastapi_app
    ports:
      - "8000:8000"
    volumes:
      - ./backend:/app
    env_file:
      - ./backend/.env
    environment:
      - PYTHONUNBUFFERED=1
    restart: unless-stopped

  frontend:
    build:
      context: ./frontend
    container_name: frontend_app
    ports:
      - "3000:80"
    restart: unless-stopped
```