Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра автоматизації та інтелектуальних інформаційних технологій

Бакалаврська кваліфікаційна робота на тему:

«РОЗРОБКА ДОДАТКУ ДЛЯ ПІДБОРУ ФІЛЬМІВ НА ОСНОВІ КАТЕГОРІЙ ІЗ ВИКОРИСТАННЯМ ШТУЧНОГО ІНТЕЛЕКТУ»

Виконав: студент IV курсу, групи 1ІСТ-21б
спеціальності 126 – Інформаційні системи
та технології
(шифр і назва спеціальності)
Медведєв М. К.
(прізвище та ініціали)

Керівник: доцент кафедри АІІТ
Гармаш. В. В.
(прізвище та ініціали)
«12» 06 2025 р

Рецензент: к.т.н, доцент кафедри САІТ
Жуков С. О.
(прізвище та ініціали)
«13» 06 2025 р

Допущено до захисту
завідувач кафедри АІІТ
Олег БІСІКАЛО
«16» 06 2025 р.

Вінниця ВНТУ – 2025 рік

Вінницький національний технічний університет
(повне найменування вищого навчального закладу)

Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра автоматизації та інтелектуальних інформаційних технологій
Рівень вищої освіти перший (бакалаврський)
Галузь знань 12 – Інформаційні технології
Спеціальність 126 – Інформаційні системи та технології
Освітньо-професійна програма Інтелектуальні інформаційні системи

ЗАТВЕРДЖУЮ
завідувач кафедри АІІТ
д.т.н., проф. Олег БІСІКАЛО
«16» 06 2025 року

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

Медведєву Максиму Костянтиновичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Розробка додатку для підбору фільмів на основі категорій із використанням штучного інтелекту»
керівник роботи Гармаш Володимир Володимирович, доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від «20» березня 2025 року №97
2. Термін подання студентом роботи 10.06.2025 р.
3. Вихідні дані до роботи:
   - розробка програмної системи для інтелектуального підбору фільмів;
   - забезпечення локального зберігання даних користувача;
   - реалізація механізмів обробки помилок та асинхронних операцій;
   - інтеграція із зовнішніми джерелами даних.
4. Зміст текстової частини (перелік питань, які потрібно розробити) Провести огляд існуючих систем рекомендації фільмів та проаналізувати сучасні підходи до їх реалізації із застосуванням штучного інтелекту; запропонувати та обгрунтувати архітектуру програмної системи; розробити алгоритми функціонування системи; провести тестування.
5. Перелік ілюстративного матеріалу (з точним зазначенням обов'язкових креслень) Схема архітектури; схема взаємодії компонентів; діаграма варіантів використання; схема роботи алгоритму рекомендації.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Гармаш В. В., доц. каф. АІІТ		

7. Дата видачі завдання 20.03.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Термін виконання		Примітки
		початок	закінчення	
1	Вибір, узгодження та затвердження теми БКР	03.10.2024	09.10.2024	
2	Аналіз предметної області та опрацювання літературних джерел.	12.03.2025	29.03.2025	
3	Постановка задач для вирішення в БКР	01.04.2025	26.04.2025	
4	Вирішення поставлених задач	29.04.2025	10.05.2025	
5	Підтвердження достовірності отриманих результатів	13.05.2025	24.05.2025	
7	Оформлення пояснювальної записки та графічної частини	27.05.2025	31.05.2025	
8	Перевірка ПЗ на відповідність вимогам	03.06.2025	07.06.2025	
9	Попередній захист, доопрацювання, та рецензування БКР	10.06.2025	15.06.2025	
10	Захист БКР	16.06.2025	23.06.2025	

Студент _____ Максим МЕДВЕДЄВ
(підпис) (ім'я та прізвище)

Керівник роботи _____ Володимир ГАРМАШ
(підпис) (ім'я та прізвище)

АНОТАЦІЯ

Бакалаврська кваліфікаційна робота складається з 87 сторінок формату А4, в тому числі і додатків, на яких є 12 рисунків, список використаних джерел містить 15 найменувань.

У роботі розглядаються основні принципи розробки програмної системи для інтелектуального підбору фільмів, включаючи методи та алгоритми персоналізованих, а також їхню інтеграцію із зовнішніми хмарними сервісами. Зручність системи полягає у гнучкості налаштування критеріїв підбору фільмів відповідно до вподобань користувача, а також у можливості розширення функціоналу в майбутньому без значних змін в основній архітектурі. Додаток реалізовано засобами мови програмування Python та сучасних бібліотек для взаємодії зі штучним інтелектом, що дозволяє виконувати обробку складних запитів швидко й ефективно, забезпечуючи безперебійну роботу системи та зручний користувацький інтерфейс.

Ключові слова: підбір фільмів, рекомендаційна система, штучний інтелект, Google Gemini, персоналізація, Python, користувацький інтерфейс.

ANNOTATION

The bachelor's qualification paper consists of 87 A4 pages, including appendices, and contains 12 images. The list of references includes 15 sources.

The paper explores the fundamental principles of developing a software system for intelligent movie recommendation, including methods and algorithms for personalized suggestions, as well as their integration with external cloud services. The convenience of the system lies in its flexibility to adjust movie selection criteria according to user preferences, as well as the possibility to expand its functionality in the future without significant changes to the core architecture. The application is implemented using the Python programming language and modern libraries for interacting with artificial intelligence, enabling efficient processing of complex queries, ensuring seamless system operation and a user-friendly interface.

Keywords: movie recommendation, recommendation system, artificial intelligence, Google Gemini, personalization, Python, user interface.

ЗМІСТ

ВСТУП

Сучасний світ перебуває на етапі глобальної цифрової трансформації, де технології відіграють ключову роль у зміні способу життя, веденні бізнесу, навчанні та дозвіллі. Одним із найбільш перспективних напрямів технічного розвитку є штучний інтелект, який дедалі активніше інтегрується в різні сфери людської діяльності. Особливого значення він набуває в індустрії розваг, де ефективність і зручність взаємодії з інформаційним контентом мають вирішальне значення.

У зв'язку з постійним зростанням обсягів медіаконтенту, користувачам дедалі складніше самостійно обирати фільми, серіали та інші розважальні матеріали, що відповідають їхнім індивідуальним смакам, адже цього контенту стає надто багато. Крім того, вибір фільму часто залежить не лише від жанрових уподобань, а й від ситуативного контексту – поточного настрою, наявності компанії для перегляду, часу доби чи навіть пори року. Традиційні рекомендаційні системи, які базуються переважно на історії переглядів, популярності або оцінках інших користувачів, не завжди здатні врахувати ці тонкі, але важливі нюанси. Це призводить до того, що користувачі нерідко отримують нерелевантні рекомендації, які не відповідають їхнім поточним потребам чи емоційному стану.

Зважаючи на вищезазначене, розробка інтелектуальних систем для персоналізованого підбору фільмів, які враховують не тільки явні переваги, а й динамічні фактори, стає актуальним напрямом досліджень та інновацій. Сучасні досягнення у галузі штучного інтелекту, зокрема розвиток великих мовних моделей, відкривають нові можливості для створення більш гнучких та інтуїтивних рекомендаційних систем, які здатні інтерпретувати складні запити та генерувати дійсно релевантні пропозиції.

Основним науково-практичним результатом роботи є розробка програмної системи для інтелектуального підбору фільмів. Ця система базується на таких критеріях, як жанр, поточний настрій користувача та рік випуску фільму, не

потребує додаткових атрибутів (крім комп'ютера з доступом до мережі), є простою у використанні та модифікації, і здатна інтегруватися з існуючими хмарними API. Цей додаток покращує взаємодію користувачів з кіноіндустрією, значно спрощуючи процес вибору фільму та підвищуючи задоволеність від перегляду.

Метою роботи розширення функціональних можливостей програмного забезпечення для підбору фільмів на основі заданих категорій із використанням штучного інтелекту.

Для досягнення мети потрібно **вирішити наступні задачі**:

1. Провести аналіз предметної області персоналізованих рекомендаційних систем.
2. Провести огляд та аналіз наявних рішень та технологій у сфері рекомендаційних систем із використанням штучного інтелекту.
3. Здійснити варіативний аналіз методів та моделей штучного інтелекту для цілей генерації кінорекомендацій.
4. Розробити архітектуру, алгоритми та структуру програмної системи для підбору фільмів.
5. Реалізувати програмне забезпечення та провести його тестування.

Об’єктом роботи є процес підбору фільмів.

Предметом роботи є засоби та методи розробки програмного забезпечення для інтелектуального підбору фільмів на основі заданих користувачем категорій.

Методи дослідження включають аналіз і синтез, методи проєктування та розробки програмного забезпечення.

Науково-практичний результат роботи полягає в розробці програмної системи для інтелектуального підбору фільмів на основі категорій із використанням штучного інтелекту, реалізованої засобами мови програмування Python та бібліотек для взаємодії з Google Gemini API та TMDb, що не потребує додаткових атрибутів, є простою у використанні та модифікації, і здатна надавати персоналізовані та релевантні рекомендації фільмів.

1 ЗАГАЛЬНА ХАРАКТЕРИСТИКА ДОСЛІДЖУВАНОЇ СИСТЕМИ

1.1 Постановка проблеми

Сучасний цифровий світ характеризується безпрецедентним обсягом доступного контенту, особливо у сфері розваг. Стрімінгові платформи, зокрема Netflix, є яскравим прикладом цього явища. Вони пропонують користувачам величезний обсяг фільмів та серіалів. За даними статистичного порталу Statista, загальна кількість фільмів та серіалів у бібліотеці Netflix перевищує сім тисяч назв, залежно від регіону, що свідчить про вражаючі масштаби пропозиції [1]. Це проілюстровано на рисунку 1.1

Однак, попри цю вражаючу кількість доступних матеріалів, саме перенасичення контентом створює суттєву проблему для користувачів — надзвичайно важко знайти саме той фільм або серіал, який відповідає їхнім особистим, індивідуальним вподобанням. Велика бібліотека не гарантує зручності вибору, а радше навпаки — призводить до інформаційного перевантаження, коли користувач відчуває себе розгубленим у безмежному потоці варіантів. Це викликає так званий "парадокс вибору", коли надмірна кількість варіантів призводить до зниження задоволення від вибору або навіть до його відмови.

Однією з ключових особливостей алгоритмів рекомендації більшості популярних платформ, включаючи Netflix, є фокус на найбільш популярних та трендових фільмах і серіалах. Такі рекомендації, безумовно, мають на меті швидко зацікавити максимально широку аудиторію та забезпечити високий рівень залученості, проте вони часто залишають поза увагою менш відомі, але не менш якісні та унікальні роботи [2]. В результаті, користувачі бачать обмежений набір варіантів, що часто повторюються, що значно звужує спектр вибору і знижує шанси на відкриття нових, унікальних продуктів, які могли б ідеально відповідати їхнім нішевим інтересам або настрою. Це призводить до своєрідної "інформаційної бульбашки", де користувач постійно бачить лише те, що вже є

популярним, і не може вийти за межі цього кола. Попри те, що алгоритми Netflix адаптуються до історії переглядів, їхня основна логіка все ще орієнтована на максимізацію показників перегляду, а не на глибоку персоналізацію, яка могла б запропонувати несподівані, але релевантні варіанти.

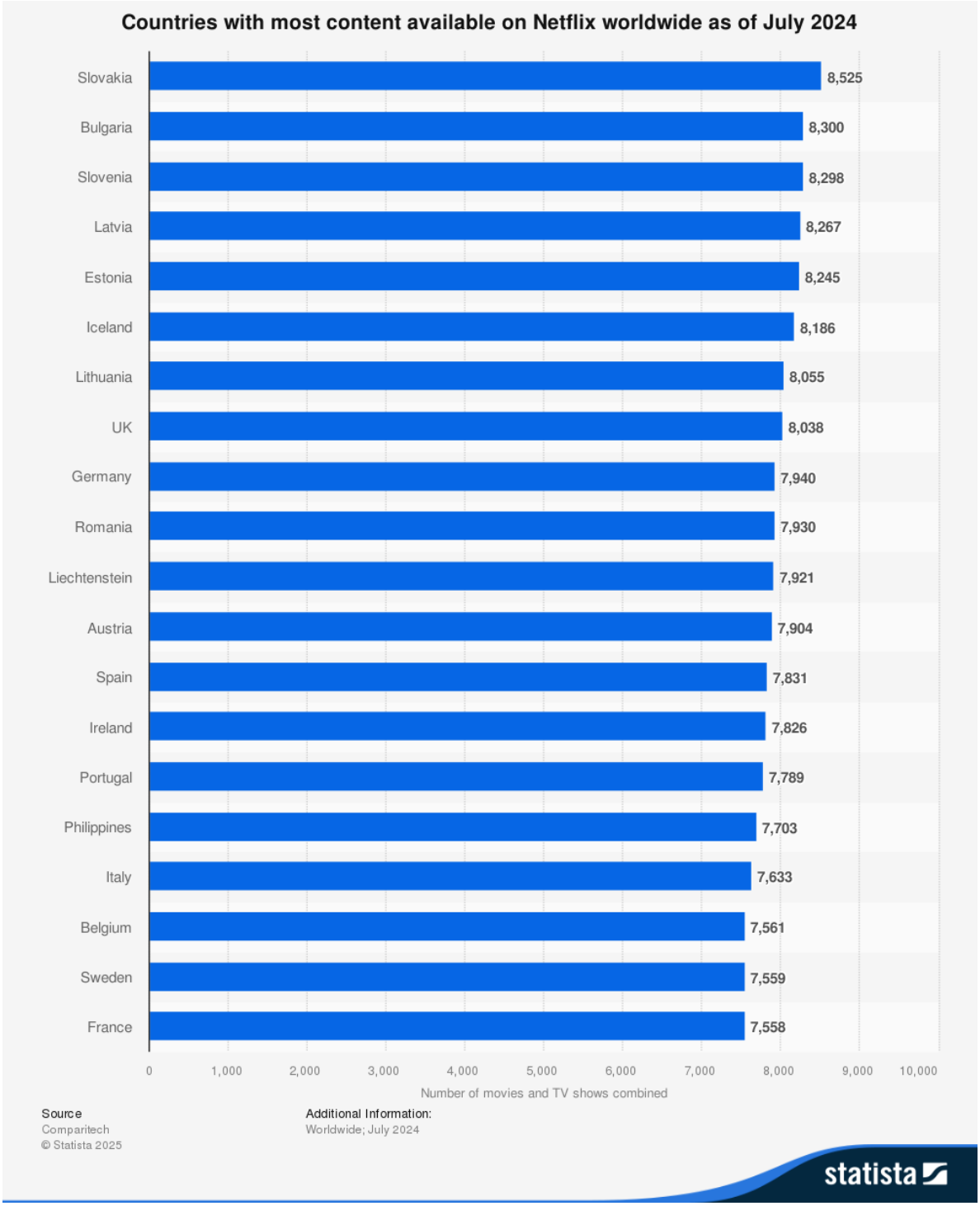

Рисунок 1.1 - Статистичні дані, що демонструють кількість фільмів та серіалів на платформі Netflix залежно від регіону

Хоча Netflix та подібні платформи безперечно забезпечують величезний каталог контенту, існуючі механізми рекомендацій не завжди ефективно допомагають у глибоко персоналізованому виборі. Це підкреслює нагальну необхідність розвитку більш адаптивних і контекстно-орієнтованих систем рекомендацій, які здатні враховувати не лише очевидні інтереси користувачів, а й їхній поточний настрій, ситуацію, часові обмеження та бажання відкривати нові, неочевидні жанри чи піджанри [2]. Нерелевантні або надмірно узагальнені рекомендації не тільки не вирішують проблему інформаційного перевантаження, а й посилюють її, змушуючи користувача витрачати значний час на пошук. Дослідження Nielsen показують, що в середньому глядач витрачає понад сім хвилин на пошук контенту, перш ніж розпочати перегляд, а це означає, що щороку користувач може витрачати до п'яти діб, гортаючи стрічку Netflix у пошуках чогось, що б йому сподобалося [3]. Ця суттєва втрата часу, яку можна було б ефективно оптимізувати за рахунок впровадження більш інтелектуальних та точних рекомендаційних систем.

1.2 Актуальність розробленої системи

Концепція рекомендацій не є новою, але її автоматизація та інтеграція у цифрові платформи почалася активно розвиватися з початком 2000-х років. На початкових етапах системи рекомендацій були відносно простими і базувалися на основних статистичних методах або вручну визначених правилах, пропонуючи переважно неперсоналізовані рекомендації, такі як "найпопулярніші товари" або "найбільш продавані". Ці ранні підходи не враховували індивідуальних вподобань кожного користувача і були трудомісткими та не масштабованими, якщо вимагали створення ручних профілів контенту чи користувачів. Зі зростанням обсягів даних та розвитком обчислювальних потужностей з'явилися перші ефективні алгоритми для персоналізованих рекомендацій. Ключовим проривом стало впровадження колаборативної фільтрації, яка дозволила автоматично виявляти приховані закономірності у поведінці користувачів без необхід-

ності явного опису контенту. Перші комерційні успіхи систем рекомендацій, зокрема на сайті Amazon, пов'язані саме з цим методом, який значно збільшив продажі. Одночасно з колаборативною фільтрацією розвивалися змістовно-орієнтовані системи. Вони пропонували елементи, схожі на ті, що користувач вже вподобав, на основі атрибутів самого контенту. З часом стало очевидно, що чисті колаборативні чи змістовно-орієнтовані підходи мають свої обмеження, що призвело до появи гібридних систем, які поєднують переваги обох методів. Паралельно, розвиток машинного навчання та, зокрема, глибокого навчання, відкрив нові можливості для створення більш складних та точних рекомендаційних моделей, дозволяючи обробляти неструктуровані дані та будувати більш тонкі профілі.

Актуальність розробленої системи визначається декількома ключовими факторами, що відображають сучасні тенденції та невирішені проблеми у сфері споживання медіа.

По-перше, незважаючи на колосальні каталоги таких корпорацій, як Netflix, Amazon Prime Video чи YouTube, навігація та пошук потрібного фільму залишаються значним викликом для користувачів. Існуючі платформи, хоча й використовують рекомендаційні алгоритми, часто покладаються на базові фільтри або примітивні механізми, які не завжди справляються зі завданням ефективного та швидкого вибору [4]. Ці системи часто пропонують надмірно широкий спектр варіантів або зосереджуються лише на найпопулярнішому контенті, що не звужує коло вибору достатньо, щоб користувач міг швидко прийняти рішення. Це створює відчуття "інформаційного перевантаження", коли велика кількість вибору парадоксальним чином призводить до складнощів у прийнятті рішення та зниження задоволення від процесу пошуку.

По-друге, сучасний користувач все більше потребує справжньої персоналізації, що виходить за межі простих узагальнень. Загальні пропозиції, що базуються виключно на популярності або узагальненій історії переглядів інших глядачів, часто не враховують унікальний поточний контекст конкретного користувача. Це може бути його поточний настрій, час доби, коли він шукає

фільм, або навіть склад компанії для перегляду (наприклад, перегляд наодинці, з родиною, з друзями). Створення системи, яка може адаптуватися до цих тонких, динамічних факторів, є критично важливим для надання дійсно релевантних рекомендацій, які збігаються з поточним бажанням користувача.

По-третє, останні досягнення у галузі машинного навчання та обробки природної мови відкривають принципово нові можливості для створення більш складних та інтелектуальних рекомендаційних систем. Ці технології дозволяють глибше розуміти контекст запиту користувача, аналізувати не лише явні оцінки, але й неявну поведінку, а також обробляти неструктуровані дані, такі як описи фільмів чи відгуки. Ефективні рекомендаційні системи мають очевидну економічну вигоду, що підкреслює їхню актуальність для бізнесу. Вони не тільки значно покращують користувацький досвід та підвищують задоволення від взаємодії з платформою, але й значно збільшують залученість користувачів. Це безпосередньо впливає на час, проведений на платформі, обсяг переглянутого контенту і, відповідно, на доходи компаній. Наприклад, Netflix, заявляє про понад 80% переглядів, ініційованих саме рекомендаціями. Це яскраво демонструє значну перевагу, яку можна отримати на ринку завдяки інтелектуальному підбору контенту. Ця вражаюча статистика найкраще демонструє важливість та актуальність теми розробки інтелектуальних рекомендаційних систем для сучасної медіаіндустрії.

1.3 Основне призначення системи

Основне призначення розробленої системи полягає у наданні користувачеві ефективної та швидкої допомоги у підборі фільму, який буде йому до вподоби, враховуючи ключові параметри, такі як жанр, поточний настрій та рік випуску. Система функціонує як інтелектуальний помічник, що аналізує вибрані користувачем параметри та пропонує найбільш відповідні варіанти. Важливою особливістю її роботи є використання виключно сучасних інструментів та алгоритмів штучного інтелекту, що дозволяє забезпечити високу точність та реле-

вантність рекомендацій, виходячи за межі простих фільтрів чи загальних рейтингів популярності.

Призначення системи зводиться до досягнення кількох ключових цілей, що вирішують проблеми, окреслені у попередньому розділі.

По-перше, вона забезпечує суттєву економію часу користувача, який зазвичай витрачається на тривалий та часто неефективний пошук фільму чи серіалу у великих каталогах. Система мінімізує "парадокс вибору", дозволяючи користувачеві швидко знайти оптимальний варіант без інформаційного перевантаження.

По-друге, розроблений додаток сприяє значному підвищенню задоволеності глядача. Надаючи точні та персоналізовані варіанти, які відповідають не лише його загальним вподобанням, а й поточним потребам та настрою. Система забезпечує більш приємний та цілеспрямований досвід перегляду. Це відрізняє її від існуючих платформ, які часто пропонують лише популярний контент, що може не відповідати індивідуальним запитам.

По-третє, система гарантує високу доступність інструменту для широкого кола користувачів. Вона створена як надзвичайно простий та інтуїтивно зрозумілий додаток, який не вимагає спеціальних технічних знань чи навичок для використання, що робить його доступним для будь-кого. Зрештою, система демонструє інноваційність у підході до вирішення повсякденної проблеми вибору контенту, застосовуючи передові методи штучного інтелекту для досягнення виняткової точності та персоналізації.

Важливою архітектурною особливістю цієї програми є її функціонування як десктопного додатку. Він працює локально на комп'ютері користувача, забезпечуючи високу продуктивність та швидкий відгук інтерфейсу. Проте, для реалізації функціоналу підбору фільмів та аналізу даних, додаток взаємодіє з хмарними сервісами штучного інтелекту. Такий гібридний підхід забезпечує підвищену конфіденційність для користувача, що є важливим аспектом у сучасному цифровому світі. Зокрема, історія переглядів користувача не зберігається локально і не аналізується для створення постійного профілю, що міг би

бути використаний для відстеження поведінки. Кожен запит користувача є унікальним, заснованим виключно на введених параметрах жанру, настрою та року випуску. Це гарантує, що система надає рекомендації на основі безпосереднього, свідомого запиту користувача, а не на основі прихованого збору даних про його поведінку, що значно підвищує рівень довіри та контролю за особистою інформацією.

1.4 Користувачі системи

Розроблена система підбору фільмів орієнтована на широке коло кінцевих користувачів, які щоденно стикаються з проблемою вибору контенту. Основна цільова аудиторія – це приватні особи, які прагнуть оптимізувати свій час на дозвілля та отримати персоналізовані рекомендації фільмів, що максимально відповідають їхнім поточним смакам та емоційному стану.

Серед цільових груп користувачів можна виділити кілька типових категорій.

По-перше, це ті, хто цінує свій час і не бажає витрачати його на тривалий пошук фільмів. Вони прагнуть ефективності та швидкості отримання рекомендацій, щоб одразу перейти до перегляду, не гортаючи безкінечні каталоги.

По-друге, це особи, чий вибір фільму сильно залежить від їхнього поточного емоційного стану. Їхня головна потреба – знайти фільм, який або гармонійно відповідає їхньому настрою, або допоможе його змінити, наприклад, підняти настрій.

По-третє, існують користувачі, що шукають різноманітність. Це ті, хто втомився від повторюваних рекомендацій стандартних стрімінгових платформ і прагне відкрити для себе нові менш очевидні варіанти. Їхня потреба полягає у виявленні нових фільмів або жанрів, які не були запропоновані іншими джерелами. Звісно, система також орієнтована на випадкових глядачів, що дивляться

фільми періодично, без глибокого занурення в кінематограф. Для них критично важливі простота та інтуїтивність, мінімум складних налаштувань.

Важливою перевагою розробленої системи є її висока конфіденційність. Цей принцип реалізується завдяки тому, що вся історія взаємодії користувача, включаючи перегляди, збережені фільми та параметри пошуку для формування профілю, зберігається виключно локально – безпосередньо на пристрої користувача. Жодні дані не передаються на віддалені сервери розробника або третіх сторін, а кожен запит до зовнішнього API генерується на основі введених локально параметрів, забезпечуючи максимальний контроль користувача над його інформацією та її надійний захист у межах власного середовища.

При розробці інтерфейсу та логіки взаємодії з користувачем були враховані ключові принципи юзабіліті. Це включає наочність, де елементи управління та результати чітко відображаються на екрані. Користувач повністю контролює параметри пошуку і може легко змінити їх. Забезпечується гнучкість та ефективність використання. Естетичний та мінімалістичний дизайн інтерфейсу, який не перевантажує зайвою інформацією, сприяє комфортному та приємному досвіду взаємодії.

2 АНАЛІЗ ТА ВИБІР ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

2.1 Аналіз життєвого циклу системи

Розробка такої інтелектуальної системи вимагає ретельного вибору відповідних інформаційних технологій, які забезпечать ефективність, масштабованість та зручність використання як під час реалізації, так і в процесі експлуатації програмного продукту.

Однією з ключових вимог до системи є здатність працювати із запитами користувача у природній мові, інтерпретувати їхній зміст, здійснювати семантичний аналіз та формувати точні відповіді на основі великої кількості даних. Для досягнення цього необхідно застосувати сучасні підходи до обробки текстової інформації, зокрема методи Natural Language Processing (NLP), а також машинного навчання [5].

На початковому етапі проєктування було здійснено аналіз життєвого циклу програмної системи з метою визначення ключових етапів її розробки — від формування вимог до тестування та розгортання. Було враховано обмеження, пов’язані з ресурсами, а також обрано оптимальні технологічні інструменти для реалізації кожного з етапів.

Життєвий цикл програмної системи описує повний шлях її існування — від ідеї до виведення з експлуатації. Було обрано класичну модель життєвого циклу програмного забезпечення, відому як каскадна модель (Waterfall model) [6]. Цей підхід є одним із найдавніших та найбільш усталених у розробці, і його вибір був повністю обґрунтований як специфікою самого проєкту, так і вимогами, що висуваються до академічних робіт. Він дозволяє чітко та послідовно проходити всі основні етапи розробки, що виявилося особливо зручним для дипломного проєкту з чітко фіксованими вимогами та обмеженими часовими рамками.

У рамках каскадної моделі процес розробки розбивається на кілька дискретних фаз, кожна з яких має бути повністю завершена, перш ніж почнеться наступна.

Однією з найбільш очевидних переваг є її простота та інтуїтивна зрозумілість. Ця модель легка для освоєння та застосування, що дозволяє розробнику чітко розуміти весь процес розробки від початку до кінця. Кожен етап має визначені входи та виходи, що створює чіткий шлях для виконання роботи. Це, у свою чергу, забезпечує високу структурованість та дисципліну в процесі розробки. Послідовний характер каскадної моделі вимагає повного завершення однієї фази перед початком наступної, що зменшує ймовірність хаотичних змін та необхідності повернення до попередніх етапів.

Однак, незважаючи на свої сильні сторони, каскадна модель має і суттєві недоліки, які варто врахувати. Одним з головних є її низька гнучкість до змін. Якщо вимоги до системи змінюються вже на пізніх етапах розробки, внесення цих змін стає надзвичайно складним, ресурсоємним та дороговартісним. Це пояснюється тим, що будь-яка зміна вимагає повернення до попередніх фаз, що порушує лінійний потік моделі та може призвести до значних затримок. Іншим істотним недоліком є відсутність раннього зворотного зв'язку від кінцевих користувачів.

Оскільки повноцінний, робочий прототип системи з'являється лише на етапі реалізації та тестування, користувачі не можуть надати свої зауваження та побажання на ранніх етапах. Це підвищує ризик того, що кінцевий продукт може не повністю відповідати їхнім очікуванням або фактичним потребам, оскільки їхній досвід взаємодії не враховується до самого кінця процесу розробки. Також, каскадна модель не є ідеальною для великих та складних проєктів з невизначеними вимогами, оскільки вона передбачає, що всі аспекти проєкту можуть бути чітко зазначені на самому початку. У таких випадках ризики накопичуються до кінця проєкту, і виявлення серйозних проблем на пізніх етапах може мати катастрофічні наслідки.

Важливо згадати, що сучасна інженерія програмного забезпечення активно використовує альтернативні підходи, які набувають все більшої популярності, особливо для проєктів, де вимоги можуть змінюватися, а швидкий зворотний зв'язок є критично важливим. Серед таких альтернатив особливо виділяється Scrum [7].

На відміну від лінійної та послідовної природи каскадної моделі, Scrum належить до категорії гнучких (Agile) методологій. Це ітеративний та інкрементальний підхід, що фокусується на швидкій доставці функціональних частин продукту. Розробка в Scrum організована у короткі, фіксовані за часом цикли, які називаються спринтами (зазвичай від одного до чотирьох тижнів). Кожен спринт має на меті створити невелику, але повністю функціональну частину системи, яку можна показати замовнику та отримати від нього зворотний зв'язок.

Основна ідея Scrum полягає в адаптивності до змін. Замість того, щоб намагатися визначити всі вимоги заздалегідь і жорстко їх дотримуватися, Scrum визнає, що вимоги можуть еволюціонувати протягом проєкту. Це досягається завдяки постійній взаємодії з замовником, регулярним оглядам результатів спринтів та гнучкому плануванню. Команда розробки в Scrum є самоорганізованою та міжфункціональною, що дозволяє їй оперативно реагувати на виклики. Схема роботи Scrum зображена на рисунку 2.1

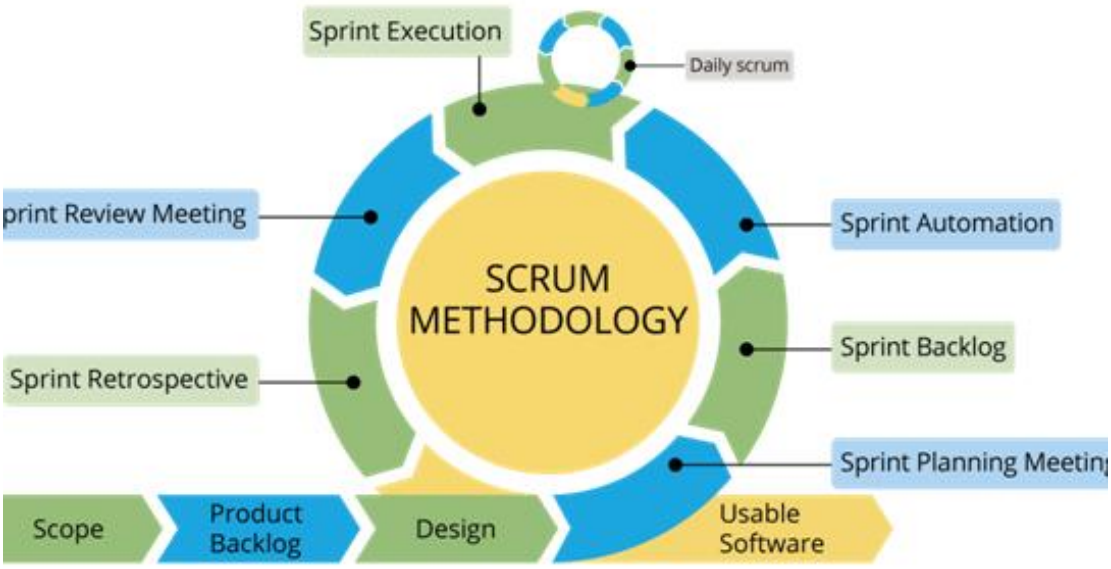

Рисунок 2.1 - Схема принципу роботи методології Scrum

Однак, було вирішено зупинитися на моделі Waterfall, адже вона краще відповідає умовам та особливостям проєкту. Життєвий цикл системи в даному випадку охоплює такі основні етапи:

1. Формулювання вимог — на цьому етапі було визначено цілі системи, її функціональні та нефункціональні вимоги, цільову аудиторію, а також обмеження, з якими доведеться працювати.
2. Аналіз предметної області та аналогів — вивчення вже існуючих рішень у сфері рекомендаційних систем, особливостей реалізації подібних систем, аналіз підходів до персоналізації.
3. Проєктування архітектури системи — визначення структури програми, вибір моделі збереження та обробки даних, побудова логіки взаємодії між модулями, визначення структури інтерфейсу.
4. Вибір технологій — обрання мови програмування, бібліотек та середовища розробки, що найбільше відповідають потребам системи, а також доступним ресурсам.

5. Реалізація — написання коду, створення моделей машинного навчання, розробка графічного інтерфейсу користувача та реалізація бізнес-логіки.
6. Тестування та налагодження — перевірка функціональності системи, її стабільності, правильності роботи моделей, виправлення помилок і оптимізація.
7. Документування та підготовка до впровадження — підготовка супровідної документації, інструкцій для користувачів та демонстраційних матеріалів.
8. Експлуатація — демонстрація роботи системи, аналіз можливих шляхів її подальшого розвитку.

Вибір каскадної моделі дозволив структуровано планувати розробку, а також поетапно формувати звітну документацію. У майбутньому, у разі масштабування або комерціалізації проєкту, можна перейти до гнучкіших моделей, таких як Scrum, що дозволяють швидше адаптуватися до змін вимог.

2.2 Архітектура системи

Архітектура розробленої системи є класичною розподіленою системою, побудованою за клієнт-серверною моделлю. У цій структурі клієнтською частиною є настільний додаток, який працює на комп'ютері користувача, тоді як серверною частиною виступає хмарний сервіс Google Gemini API [8].

Такий архітектурний вибір дозволяє чітко відокремити логіку взаємодії з користувачем та візуальне представлення від обчислювально інтенсивних операцій, що виконуються на потужній хмарній інфраструктурі Google. Це забезпечує не лише високу продуктивність системи, але й її масштабованість, усуваючи необхідність локального розгортання складних моделей штучного інтелекту.

Система складається з декількох ключових компонентів, що тісно взаємодіють між собою. Першим з них є графічний інтерфейс користувача (GUI). Цей компонент відповідає за безпосередню взаємодію з користувачем, збираю-

чи вхідні дані та відображаючи результати обробки. Він реалізований за допомогою стандартної Python-бібліотеки Tkinter [9], яка надає набір базових, але ефективних віджетів, таких як кнопки, випадаючі списки, повзунки та текстові поля, що формують візуальну складову програми. Tkinter обробляє всі дії користувача – натискання кнопок, зміну значень повзунків – і викликає відповідні функції у бізнес-логіці програми. Саме через GUI користувач обирає жанри, налаштовує параметр настрою та задає часові рамки, а результати пошуку – назви фільмів та їхні описи – відображаються безпосередньо в цьому інтерфейсі.

Другим важливим компонентом є модуль обробки запитів та даних, який можна назвати "мозком" клієнтської частини. Цей модуль, написаний на Python, відповідає за агрегацію даних, отриманих з GUI, формування запиту до моделі Gemini, подальшу обробку отриманої відповіді та передачу відформатованих результатів назад до інтерфейсу користувача.

Його ключові функції включають парсинг вхідних даних, тобто перетворення вибору користувача (наприклад, числових значень повзунка) у текстовий формат, який є зрозумілим для генеративної моделі. Далі відбувається формування промпта – структурованого текстового запиту до Gemini API, що містить усі обрані користувачем критерії (жанр, настрій, роки). Цей промпт формується таким чином, щоб максимально точно та однозначно поставити завдання моделі, наприклад, "Порекомендуй 5 фільмів жанру 'Комедія', знятих у 2000-х, що підходять для підняття настрою. Надай назву та короткий синопсис кожного фільму". Після отримання відповіді від Gemini, цей модуль здійснює обробку відповіді, парсинг отриманого тексту та виділення назв фільмів та їхніх коротких описів, які потім передаються до GUI для відображення.

Для забезпечення безперебійної роботи інтерфейсу користувача під час виконання тривалих операцій, таких як звернення до віддалених API, використовується Модуль асинхронної обробки, який базується на стандартних Python-бібліотеках threading та queue. Бібліотека threading [9] дозволяє запускати мережеві запити до Gemini API в окремому фоновому потоці. Це запобігає блоку-

ванню основного потоку виконання програми, який відповідає за оновлення інтерфейсу та взаємодію з користувачем.

Без багатопоточності, під час очікування відповіді від API, інтерфейс би "завис", що негативно позначилося б на користувацькому досвіді. Бібліотека queue [10], у свою чергу, забезпечує безпечну передачу даних між основним потоком GUI та фоновим потоком API-запитів. Фоновий потік поміщає результат (список рекомендованих фільмів) у чергу, а основний потік періодично перевіряє цю чергу і, при наявності даних, вилучає їх та оновлює інтерфейс. Цей механізм ефективно усуває типові проблеми багатопоточності, такі як "перегони даних" (race conditions) та "дедлоки".

Основою інтелектуальної частини системи є Google Gemini API, що виступає в ролі хмарного сервісу. Він надає доступ до потужної моделі генеративного штучного інтелекту Gemini 1.5 Flash. Ця модель є однією з найпродуктивніших та найефективніших для завдань генерації тексту, розуміння контексту та ведення діалогу. Вона оптимізована для швидкого виконання, що є критично важливим для інтерактивних систем, де користувач очікує миттєвої реакції. Клієнтська частина системи надсилає HTTP-запити до спеціального ендпойнта Gemini API, передаючи сформований промпт, а у відповідь отримує структурований текст з рекомендаціями.

Принцип роботи архітектури можна описати наступною послідовністю дій: Користувач взаємодіє з графічним інтерфейсом (Tkinter), обираючи бажані параметри для пошуку фільму. У момент натискання кнопки "Підібрати", Модуль обробки запитів збирає ці дані з елементів GUI. Далі цей модуль формує промпт і, використовуючи бібліотеку threading, запускає новий потік для виконання запиту до API. Цей новостворений потік, за допомогою бібліотеки google.generativeai, надсилає сформований запит до Google Gemini API. Хмарний сервіс Google Gemini API обробляє запит, використовуючи свої алгоритми штучного інтелекту, генерує рекомендації фільмів і повертає їх у відповідь. Потік, що отримав відповідь, поміщає її у спеціальну чергу (queue). Основний потік GUI постійно перевіряє цю чергу. Як тільки в черзі з'являються дані, він

безпечно вилучає їх. Модуль обробки запитів парсить отримані дані, виділяючи назви фільмів та їхні описи. Нарешті, графічний інтерфейс користувача (Tkinter) відображає отримані назви фільмів та їхні описи для користувача.

Така архітектура забезпечує високу швидкодію, оскільки основні обчислювальні операції перенесені на потужні сервери Google, знімаючи навантаження з комп'ютера користувача. Вона також є гнучкою та розширюваною: у майбутньому можна легко інтегрувати інші API, наприклад, для отримання обкладинок фільмів або трейлерів, або ж замінити поточну модель штучного інтелекту на іншу без суттєвих змін у логіці клієнтського застосунку. Схема взаємодії Python із Gemini API проілюстрована на рисунку 2.2

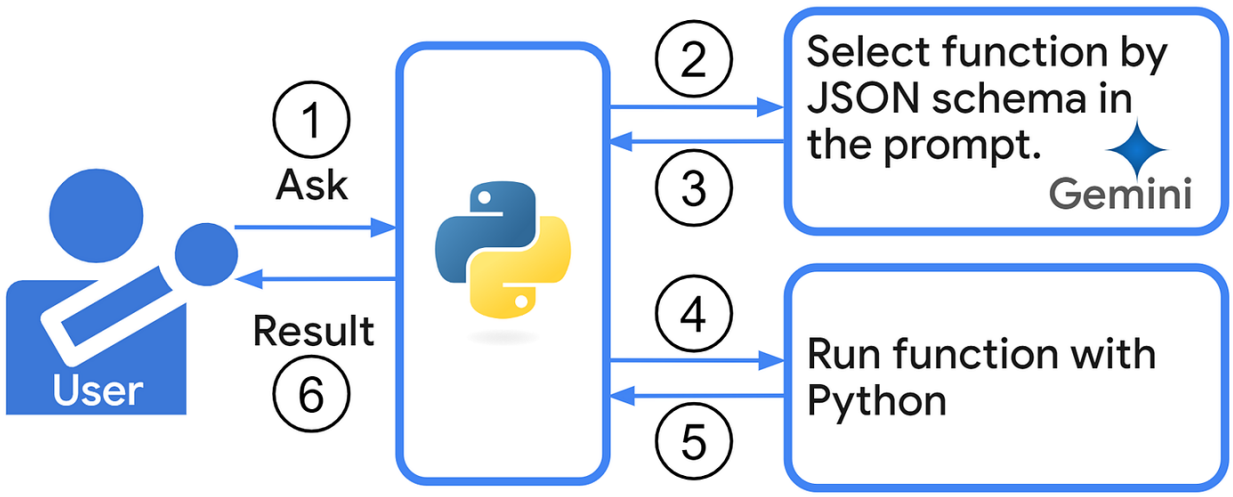

Рисунок 2.2 - Схема взаємодії Python із Gemini API

2.3 Технічні характеристики

Для забезпечення коректного функціонування, ефективної роботи та досягнення всіх поставлених цілей розробленої системи підбору фільмів на основі штучного інтелекту, було критично важливо дотриматися певних технічних вимог та використовувати конкретні, ретельно обґрунтовані технологічні рішення. Вибір цих характеристик ґрунтувався на всебічному аналізі принципів

оптимізації продуктивності, що забезпечує швидкий відгук системи, зручності розробки, що прискорює ітерації та тестування, та, що особливо важливо, простоти розгортання та використання для кінцевого користувача. Цей підхід дозволив мінімізувати бар'єри входу для широкої аудиторії, зробивши додаток доступним та легким у використанні без складних попередніх налаштувань.

З боку програмного забезпечення, основною мовою розробки був обраний Python версії 3.10. Цей вибір зумовлений його винятковою гнучкістю та універсальністю, що дозволяє застосовувати його для широкого спектру завдань, від розробки веб-додатків до наукових обчислень. Ключовою перевагою Python є його надзвичайно широка екосистема бібліотек, що є критично важливою для розробки додатків зі штучним інтелектом та машинним навчанням. Наявність готових, високооптимізованих бібліотек значно прискорює процес розробки, дозволяючи зосередитися на бізнес-логіці, а не на низькорівневих деталях. Крім того, простота синтаксису Python та його висока читабельність значно прискорюють процес розробки та налагодження коду, що є важливим фактором для ефективного управління проєктом.

Як інтерпретована мова, Python є кросплатформною, що дозволяє потенційно запускати розроблений додаток на різних операційних системах, таких як Microsoft Windows, macOS та різні дистрибутиви Linux, без значних змін у коді. Це не тільки розширює потенційну аудиторію користувачів, але й зменшує витрати на підтримку та розгортання.

Використання однієї з останніх стабільних версій Python, а саме 3.10, забезпечує доступ до новітніх мовних оптимізацій, покращеного функціоналу, такого як structural pattern matching, та підвищеної безпеки, що сприяє загальному покращенню продуктивності та стабільності програми, мінімізуючи потенційні вразливості.

Серед ключових бібліотек Python, які були використані для побудови функціоналу системи, варто особливо виділити google.generativeai. Ця бібліотека є офіційною клієнтською бібліотекою, спеціально розробленою та підтримуваною Google для зручної та ефективної взаємодії з Google Gemini API. Її викори-

стання значно спрощує та уніфікує процес формування складних текстових запитів до генеративної моделі штучного інтелекту, обробку отриманих відповідей, що можуть мати різну структуру, а також безпечне керування API-ключем, який необхідний для авторизації доступу до хмарного сервісу Gemini. Таким чином, google.generativeai є абсолютно ключовим елементом, що забезпечує безшовну та надійну інтеграцію клієнтського застосунку з потужною інтелектуальною частиною системи, розташованою у хмарних сервісах Google.

Ця інтеграція дозволяє використовувати передові можливості штучного інтелекту для генерації персоналізованих рекомендацій фільмів, які є основою функціоналу додатку. Для створення графічного інтерфейсу користувача (GUI) була обрана стандартна бібліотека **tkinter**. Цей вибір обґрунтований тим, що tkinter є вбудованою в Python, що означає відсутність потреби у встановленні додаткових зовнішніх залежностей або складних конфігурацій. Це значно спрощує процес розгортання додатку для кінцевого користувача, оскільки йому не потрібно виконувати додаткові кроки для підготовки середовища.

Переваги tkinter полягають у його простоті використання для розробки десктопних додатків, відсутності складних конфігурацій та достатній функціональності для створення інтуїтивно зрозумілого та легкого у використанні інтерфейсу.

Хоча tkinter може не пропонувати таку ж розширену візуальну естетику або повний набір віджетів, як деякі сучасні фреймворки (наприклад, PyQt або Kivy), його надійність, простота та легковажність роблять його ідеальним вибором для цього типу локального, орієнтованого на функціональність застосунку, де головний акцент робиться на зручності використання та швидкому доступі до рекомендацій.

Важливим аспектом, що забезпечує плавність та високу відгуковість інтерфейсу користувача, є ретельна реалізація багатопотоковості. Для цього був використаний стандартний модуль Python threading. Принцип його роботи полягає у можливості виконання декількох частин програми одночасно у так званих "потоках" (threads) – легковагових одиницях виконання в рамках одного

процесу. У контексті цієї системи, це дозволяє ефективно виконувати операції, які вимагають очікування (наприклад, тривалі мережеві запити до віддаленого Google Gemini API або TMDb API, які можуть займати декілька секунд або більше, залежно від швидкості мережі та завантаженості сервісу), в окремих, фонових потоках.

Таким чином, основний потік, що відповідає за графічний інтерфейс користувача (GUI), не блокується під час очікування відповіді від API. Це означає, що користувач може продовжувати взаємодіяти з елементами інтерфейсу – натискати кнопки, вводити текст, змінювати налаштування, прокручувати списки – навіть коли система виконує складні обчислення або чекає на відповідь від хмарного сервісу.

Це суттєво покращує загальний користувацький досвід, роблячи додаток більш відгуковим, приємним у використанні та створюючи враження високої продуктивності, оскільки користувач не бачить "зависань" програми. Реалізація міжпотокової комунікації за допомогою механізму черг або сигналів забезпечує безпечний обмін даними між фоновими та основним потоками, запобігаючи проблемам конкурентного доступу та забезпечуючи стабільність роботи. Принцип роботи зображено на рисунку 2.3

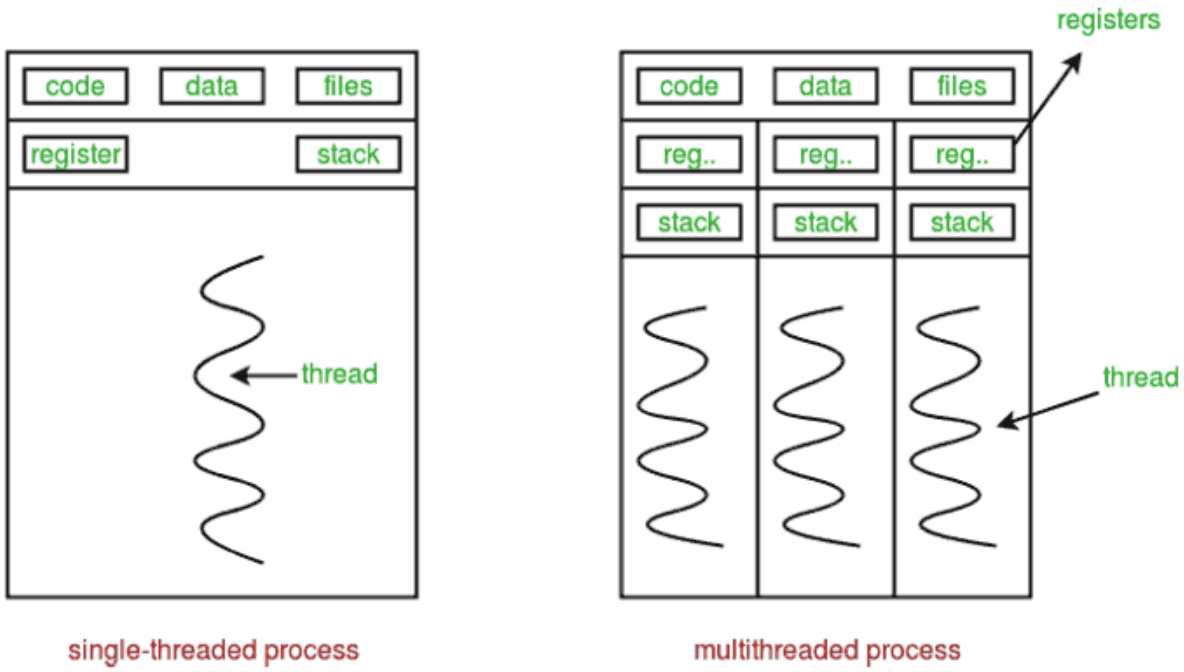

Рисунок 2.3 - Принцип роботи бібліотеки threading

Доповнює цю функціональність модуль queue, який надає класи для реалізації черг, що є необхідним для безпечного обміну даними між основним потоком GUI та фоновим потоком, що виконує запити до API, запобігаючи потенційним проблемам синхронізації та перегонів даних. Основним зовнішнім сервісом є Google Gemini API, що надає доступ до моделі генеративного штучного інтелекту Gemini 1.5 Flash. Це є центральним компонентом для обробки природної мови та генерації рекомендацій. Важливою умовою доступу до сервісу є наявність API-ключа Google Gemini, який користувач повинен отримати та налаштувати для авторизації та білінгу використання сервісу.

З точки зору апаратних вимог, оскільки основні обчислення виконуються на стороні Google Gemini API, локальні вимоги для клієнтського застосунку є відносно невисокими.

Мережеві вимоги включають наявність стабільного інтернет-з'єднання, яке є критично важливим для постійного обміну даними з Google Gemini API. Швидкість з'єднання має бути достатньою для передачі текстових запитів та отримання відповідей у реальному часі; рекомендовано мінімум 1 Мбіт/с. Важ-

ливо також, щоб фаєрволи або інші мережеві обмеження не блокували вихідні HTTP/HTTPS запити до доменів Google API.

Щодо вимог до операційної системи, завдяки використанню Python та Tkinter, програма потенційно є кросплатформною та сумісною з основними операційними системами: Microsoft Windows (починаючи з Windows 7), macOS (версії 10.12 Sierra або новіші) та будь-яким сучасним дистрибутивом Linux з встановленим Python 3.10+. Щодо продуктивності, час відгуку системи залежить від швидкості інтернет-з'єднання та поточного завантаження Google Gemini API, проте зазвичай він становить від 1 до 5 секунд.

Система також передбачає базову обробку помилок мережевого з'єднання або помилок API, інформуючи користувача про можливі проблеми. Вибір цих технічних характеристик дозволив створити функціональну та відносно легку систему, яка не вимагає значних ресурсів від кінцевого користувача, забезпечуючи при цьому доступ до потужних можливостей штучного інтелекту через хмарні сервіси.

3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Основні функціональні можливості

Запропонована система розроблена для реалізації інноваційного підходу до персоналізованого підбору фільмів та серіалів. Вона базується на трьох ключових критеріях, які роблять її унікальною та особливо зручною для користувача.

Першим, і водночас найбільш інноваційним критерієм, є настрій користувача. Система надає можливість вказати поточний емоційний стан або бажаний настрій, який користувач прагне викликати під час перегляду фільму. Після отримання цієї інформації, штучний інтелект системи, використовуючи свої розширені можливості аналізу, підбирає фільми, які не просто відповідають цьому емоційному стану, а й можуть допомогти його скоригувати або посилити, залежно від побажань. Цей підхід суттєво відрізняє розроблену систему від більшості існуючих платформ, які зазвичай ігнорують такий важливий аспект емоційного та психологічного контексту, обмежуючись лише жанровими або рейтинговими фільтрами. Можливість врахувати настрій дозволяє створювати набагато більш релевантні та задовільні рекомендації, оскільки перегляд фільму часто є емоційно мотивованим досвідом.

Другим ключовим критерієм є вибір жанру, який пропонується користувачеві залежно від попередньо обраного настрою, а не як самостійний, незалежний параметр. Це забезпечує більш контекстуалізований підбір, оскільки певні жанри краще асоціюються з конкретними емоціями. Користувач може обрати один або декілька жанрів, що уточнить запит для алгоритмів ШІ.

Третій критерій – це часові рамки випуску фільму. Ценадає користувачеві гнучку можливість шукати як перевірену часом класику, так і найновіші релізи, що відповідають його перевагам щодо давності контенту. Цей діапазон дат дозволяє відфільтрувати фільми, які можуть бути занадто старими або, навпаки,

занадто новими для поточного бажання користувача. Інтерфейс головної сторінки зображено на рисунку 3.1

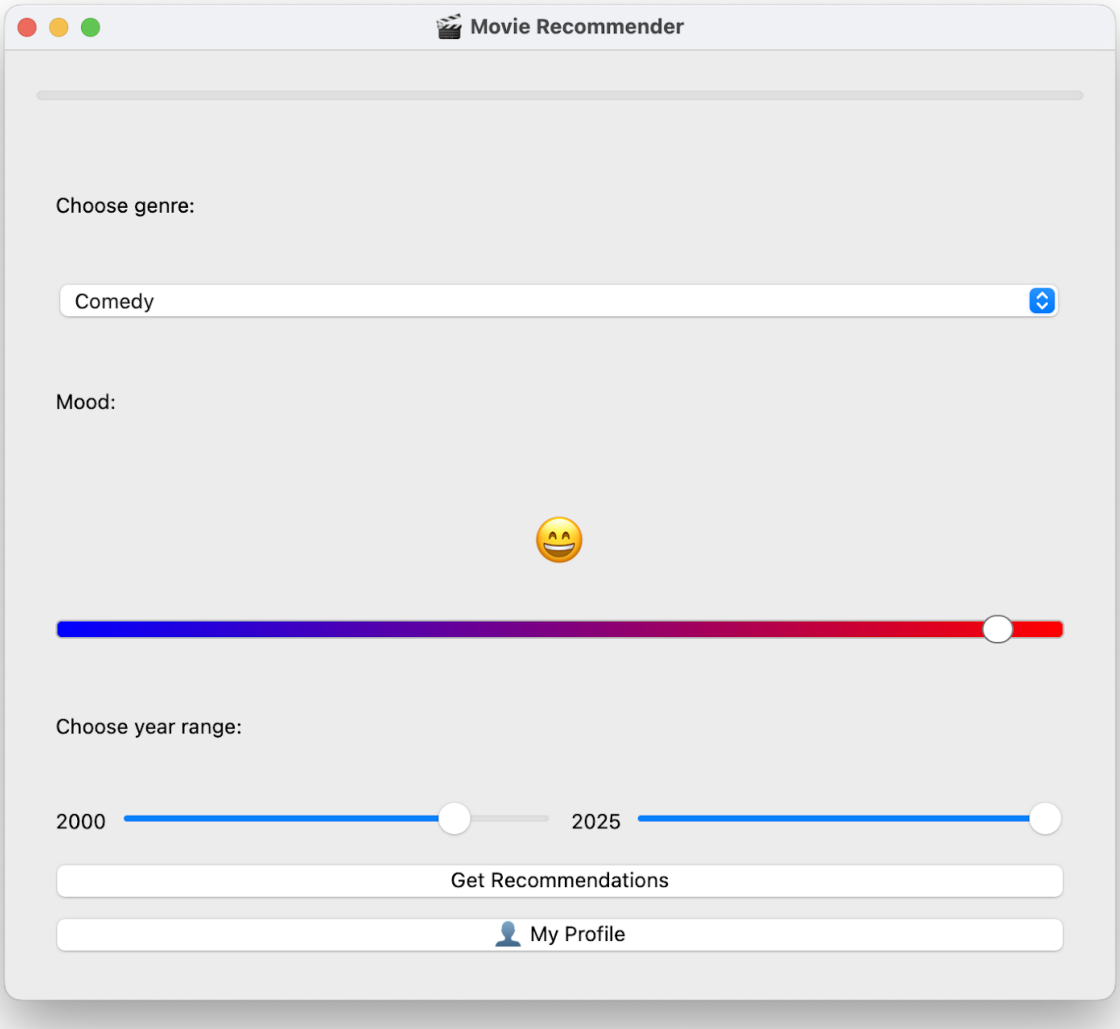

Рисунок 3.1 - Головна сторінка графічного інтерфейсу програми

Після введення цих параметрів на головній сторінці, програма формує запит, враховуючи всі нюанси, і надсилає його до хмарної мовної моделі штучного інтелекту. Ця модель аналізує отриманий запит, інтерпретує бажаний настрій, жанрові переваги та вікові рамки, і на основі свого обширного знання про кінематографічний контент підбирає найбільш відповідні фільми.

Після обробки, результати оперативно відображаються користувачеві у вигляді зручного та інтуїтивно зрозумілого списку на окремій сторінці.

Вигляд сторінки зі рекомендаціями зображено на рисунку 3.2

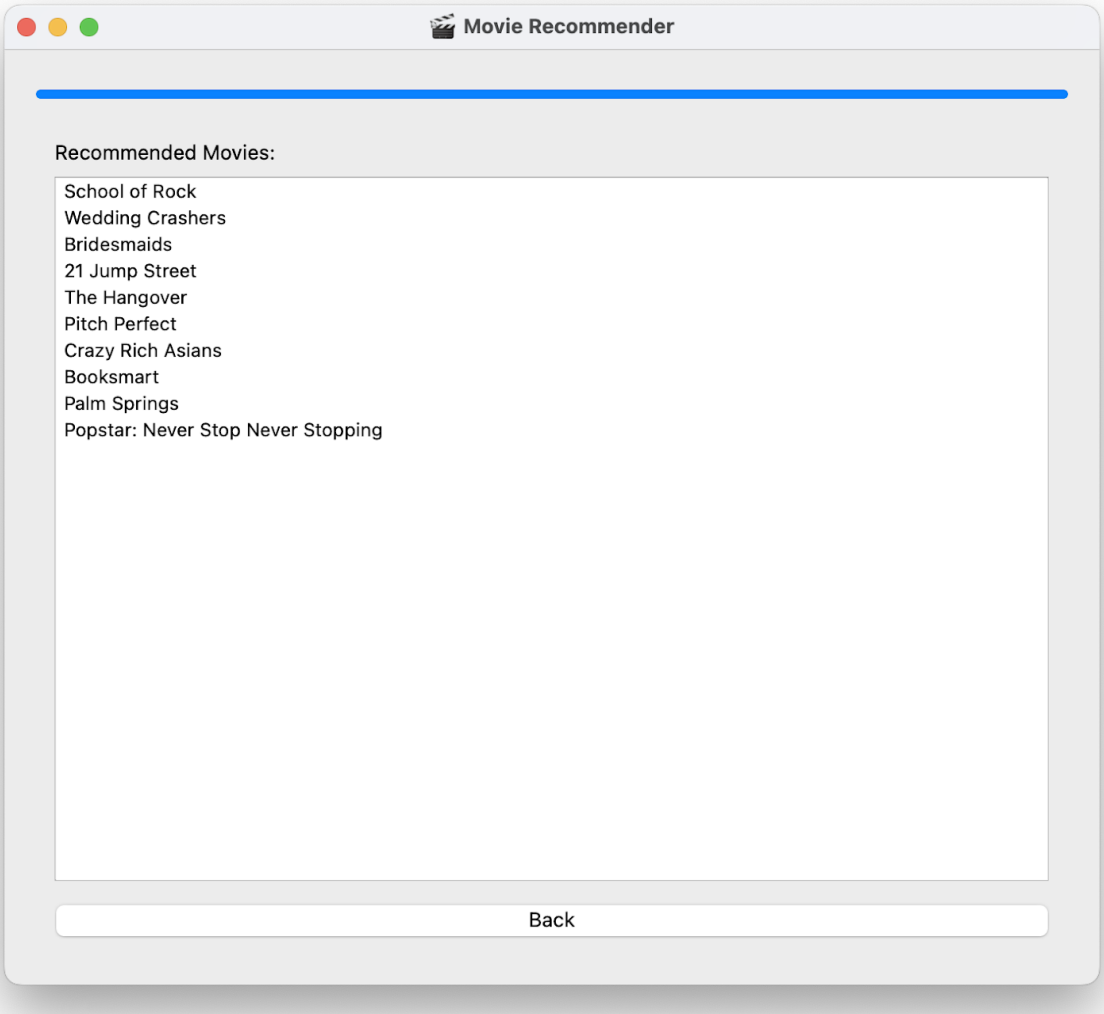

Рисунок 3.2 - Сторінка зі запропонованим списком фільмів

Користувач може переглянути короткий опис фільму, який дає загальне уявлення про його сюжет та основні ідеї, а також отримати доступ до трейлера. Це дозволяє швидко зрозуміти, про що йдеться у стрічці, оцінити її візуальний стиль та динаміку, і вирішити, чи варто її дивитись далі. Завдяки цій інтегрованій функціональності користувачу не доводиться відкривати додаткові сайти, такі як IMDb чи YouTube, чи вручну шукати інформацію про стрічки. Сторінка, на якій міститься опис та трейлер, зображена на рисунку 3.3

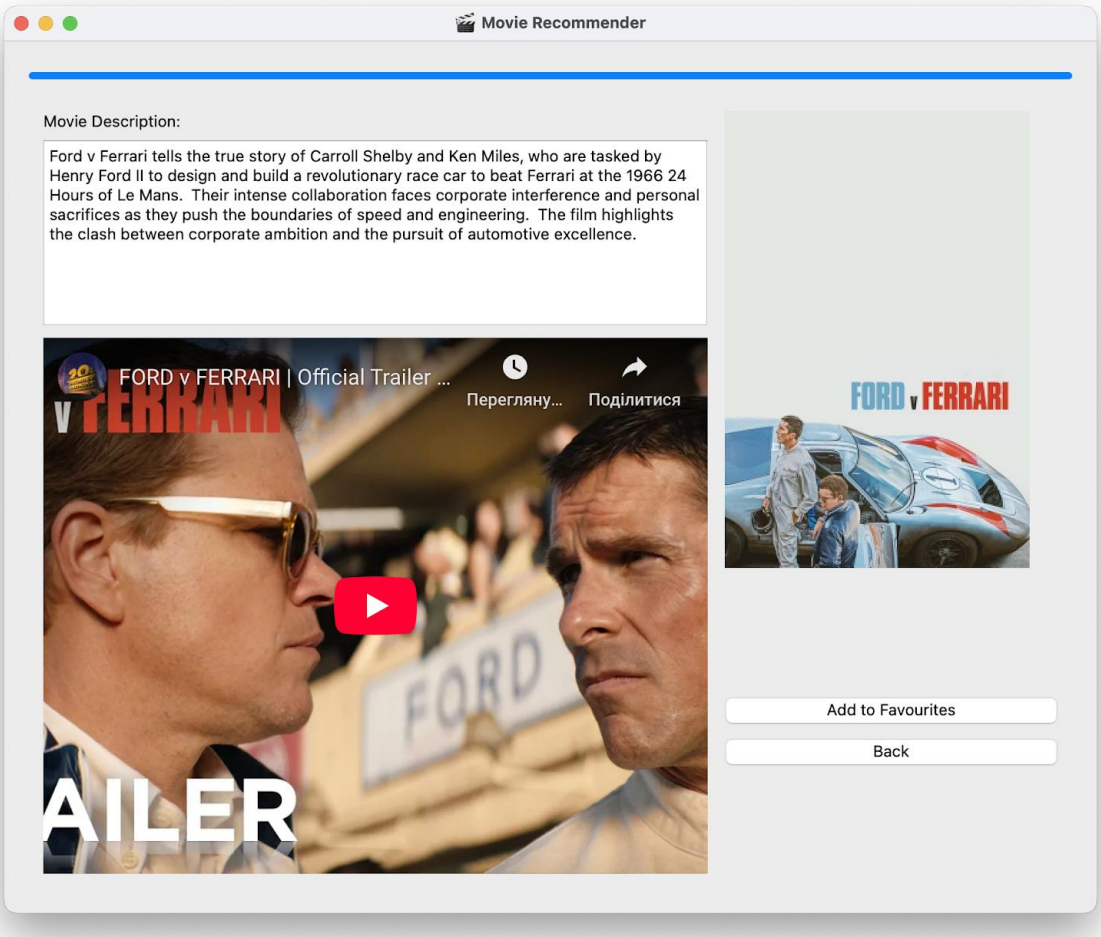

Рисунок 3.3 - Сторінка з інформацією про фільм

Варто відзначити, що користувач матиме можливість зберігати фільми, які йому особливо сподобалися або які він планує переглянути пізніше, в окремому, спеціально виділеному розділі. Цей функціонал діє як персональний список бажань.

Це дозволяє користувачеві легко повертатися до цих обраних стрічок у будь-який зручний момент, без необхідності повторного пошуку або запам'ятовування їхніх назв. Таке зберігання фільмів значно підвищує зручність використання системи, перетворюючи її з інструменту одноразового пошуку на особистий, динамічний каталог улюбленого кіно.

Це також може слугувати основою для подальшого, більш глибокого аналізу уподобань користувача, за умови, якщо така функціональність буде розширена

у майбутніх версіях програми. Розділ з улюбленими фільмами показано на рисунку 3.4

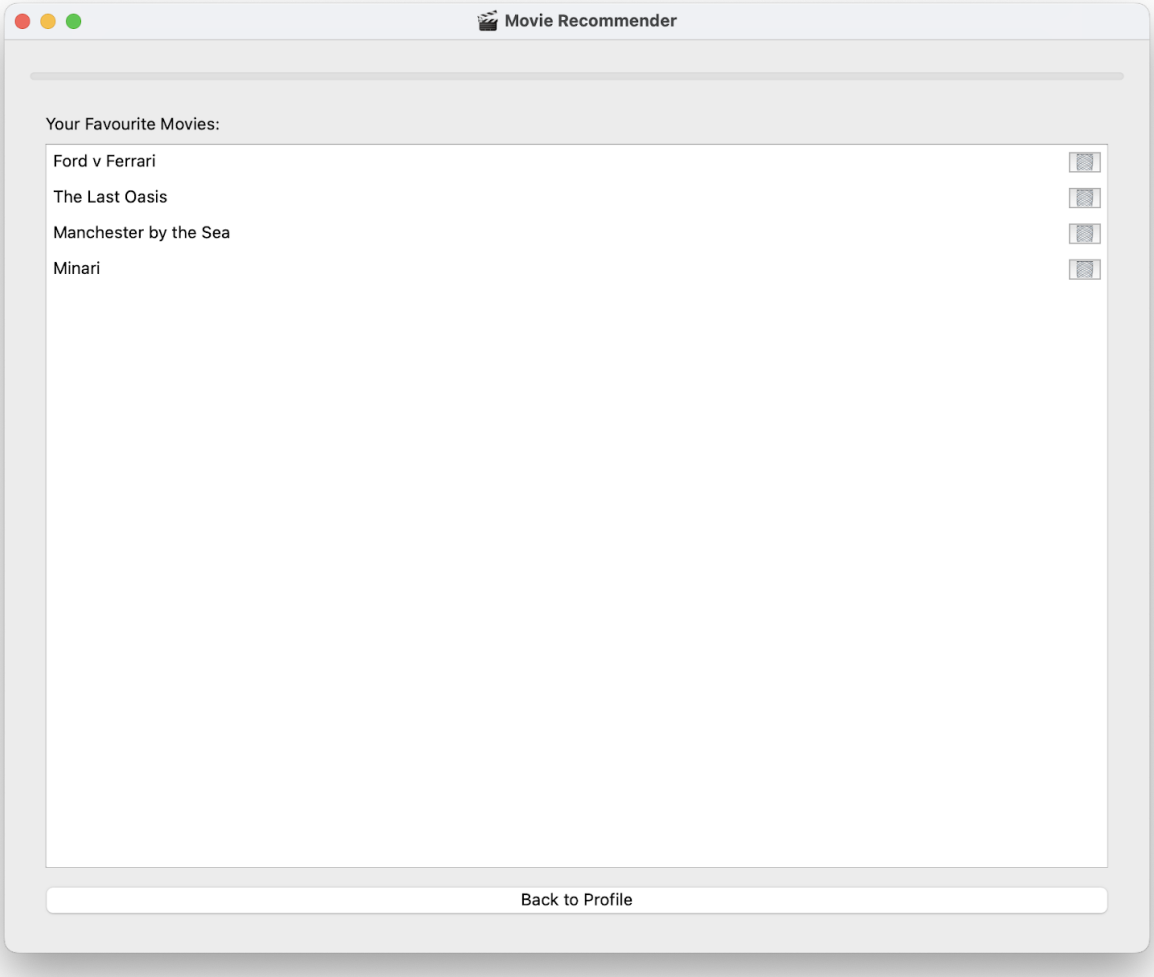

Рисунок 3.4 - Сторінка розділу з улюбленими фільмами

Додатково, для розширення можливостей персоналізації та аналізу, в системі передбачається окремий розділ особистого профілю користувача. У цьому профілі користувач зможе не лише переглядати та керувати фільмами, які йому сподобались або були збережені до "улюблених", а й отримувати цікаву статистику щодо своїх кінематографічних смаків та звичок. Зокрема, система зможе аналізувати агреговану історію його запитів та збережених фільмів, відображаючи, які жанри він найчастіше шукає або обирає, які роки випуску

фільмів йому до вподоби, та як його настрій впливає на вибір контенту. Ця функціональність надасть користувачеві унікальний інсайт у власні кінематографічні вподобання, допомагаючи йому краще зрозуміти себе як глядача, а також може бути використана для подальшого вдосконалення рекомендацій з часом. Профіль користувача зображено на рисунку 3.5

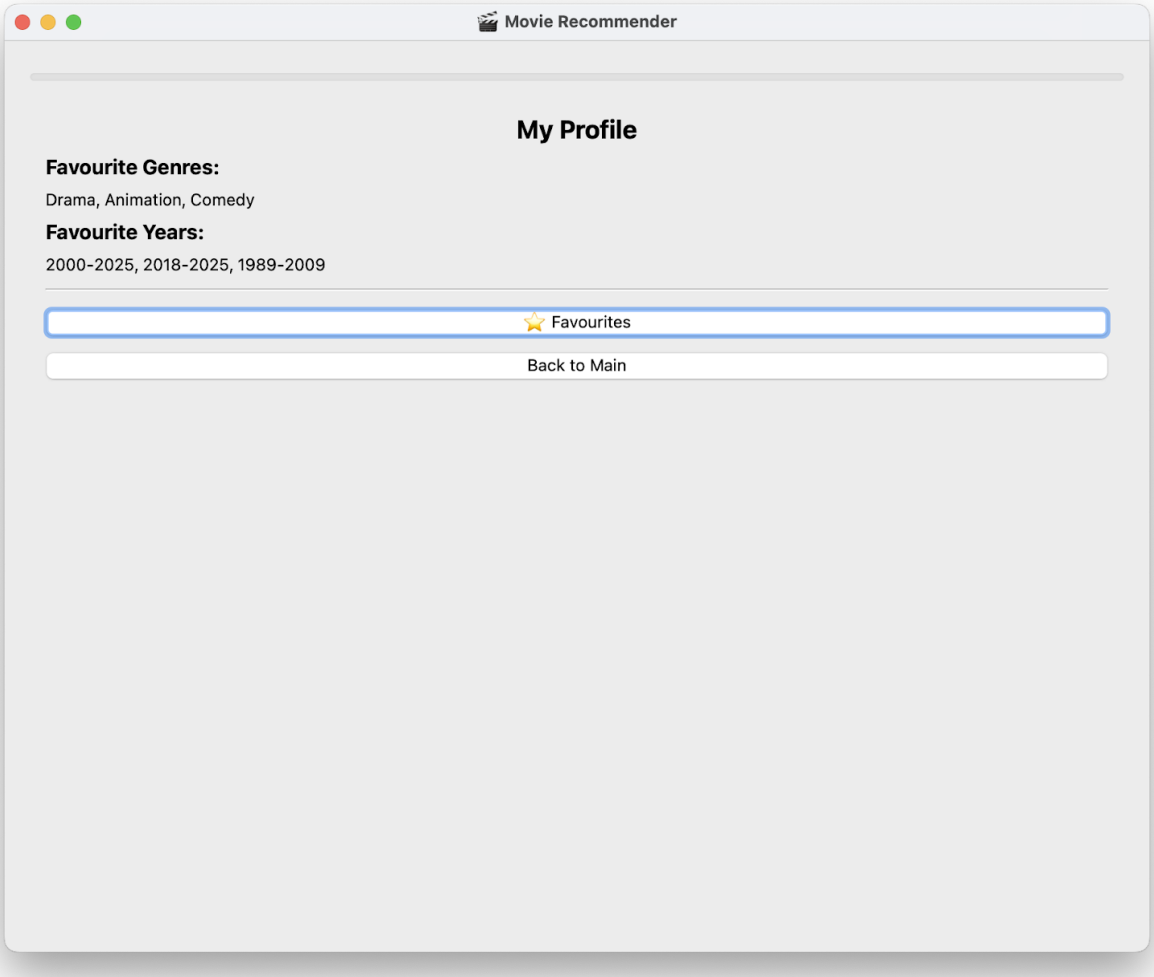

Рисунок 3.5 - Сторінка профілю користувача

Ще однією ключовою особливістю системи, що сприяє високому рівню користувацького досвіду, є швидкість її роботи. Сам процес підбору фільмів, від моменту відправки запиту до отримання та відображення результатів, займає мінімум часу – зазвичай це відбувається лише протягом кількох секунд після натискання відповідної кнопки. Така оперативність досягається завдяки

ефективній архітектурі, яка дозволяє обробляти складні запити до ШІ без затримок. Важливою перевагою є те, що усе функціонує у межах одного вікна програми. Це означає, що користувачеві не потрібно перенаправлятися у браузер або відкривати зовнішні веб-сторінки. Відсутність таких перенаправлень значно підвищує швидкість взаємодії та запобігає відволіканню, оскільки завантаження сторінок у браузері, як правило, є значно повільнішим і може створювати відчуття роз'єднаності в роботі додатку. Таким чином, система пропонує цілісне, швидке та інтуїтивно зрозуміле рішення для персоналізованого підбору фільмів.

Розроблена система, окрім надання рекомендацій та управління колекцією, також закладає фундамент для майбутнього розвитку та вдосконалення. Постійне збагачення бази знань мовної моделі та вдосконалення алгоритмів ШІ потенційно дозволить розширити діапазон емоційних станів для підбору фільмів, включити більш гранульовані жанрові категорії або навіть враховувати інші, непрямі фактори, такі як часові уподобання перегляду (день тижня, пора доби) або навіть соціальний контекст (перегляд наодинці, з друзями, з родиною). Можливість збереження улюблених фільмів та збору статистики переглядів створює унікальний локальний датасет, який, у майбутньому, може бути використаний для навчання персоналізованих рекомендаційних моделей безпосередньо на пристрої користувача (on-device learning), що ще більше підвищить конфіденційність та точність рекомендацій, адаптуючись до постійно змінюваних смаків. Ця архітектурна гнучкість та орієнтація на користувача роблять систему не просто інструментом, а особистим кінокомпаньйоном

3.2 Внутрішня взаємодія компонентів

Архітектура розробленої програмної системи побудована за гнучким та ефективним гібридним клієнт-серверним принципом. У цій моделі десктопний додаток, який працює безпосередньо на пристрої користувача, виконує роль

клієнта. Він відповідає за забезпечення інтерфейсу користувача, збір вхідних даних, обробку локальних операцій та відображення результатів. Натомість, зовнішні API-сервіси, такі як Google Gemini та TMDb (The Movie Database), виступають у ролі віддалених серверів. Вони надають основні обчислювальні потужності для складних операцій штучного інтелекту та доступ до обширних баз даних кінематографічного контенту, що дозволяє системі бути легкою для кінцевого користувача, не вимагаючи значних локальних ресурсів для обробки.

Центральним елементом клієнтської частини та основою для побудови всього графічного інтерфейсу є клас MainForm. Цей клас успадковується від базового класу QWidget бібліотеки PyQt6, що забезпечує міцну основу для створення віконних застосунків. Клас MainForm не лише відповідає за всю логіку управління графічним інтерфейсом, включаючи відображення елементів, обробку подій користувача та оновлення стану, але й відіграє ключову роль у координації взаємодії з усіма фоновими процесами, що виконують тривалі операції. Для забезпечення плавної та інтуїтивно зрозумілої навігації в інтерфейсі до класу MainForm інтегровано віджет QStackedWidget. Це потужне рішення дозволило створити безперебійні та естетично привабливі переходи між п'ятьма основними сторінками додатку, кожна з яких виконує свою унікальну функцію. Зокрема, selection_page призначена для початкового збору вхідних даних від користувача, включаючи його настрій, бажаний жанр та діапазон років. Після обробки запиту, results_page елегантно відображає отриманий список рекомендованих фільмів. Для глибшого ознайомлення з контентом, description_page показує розширений синопсис обраного фільму разом із відповідним постером та вбудованим трейлером. Favourites_page слугує персональним сховищем, де користувач може переглядати та керувати своїми збереженими улюбленими фільмами, а profile_page надає користувачеві цінну статистичну інформацію про його кінематографічні уподобання, відображаючи найчастіше обирані жанри та роки.

Ключовим аспектом архітектури, що забезпечує високу чуйність та стабільність додатку, є стратегія виконання всіх потенційно тривалих операцій у

фонових потоках виконання. Це стосується, перш за все, мережевих запитів до віддалених сервісів Google Gemini та TMDb API. Цей підхід запобігає блокуванню основного потоку графічного інтерфейсу (GUI), що є типовою проблемою при синхронному виконанні мережевих операцій у віконних додатках. Блокування GUI призводить до того, що програма "зависає" і не реагує на дії користувача, що значно погіршує користувацький досвід. Натомість, у цій архітектурі, коли користувач ініціює дію, що вимагає звернення до зовнішнього сервісу (наприклад, натискає кнопку "Отримати рекомендації"), відповідний метод у класі MainForm не виконує цей запит безпосередньо в основному потоці. Натомість, він створює та запускає новий об'єкт threading.Thread – легковажний потік виконання. Цьому новому потоку передаються всі необхідні параметри запиту, а також посилання на функцію, яка буде безпосередньо виконувати мережеву операцію (наприклад, google.generativeai.GenerativeModel.generate_content() або запит до TMDb). Таким чином, основний потік залишається вільним для обробки подій інтерфейсу, підтримуючи його безперервну та плавну взаємодію з користувачем.

Комунікація між цими фоновими робочими потоками та основним потоком GUI здійснюється за допомогою потужного та безпечного механізму сигналів та слотів, що є невід'ємною частиною бібліотеки PyQt6 [12]. Спеціально визначений клас WorkerSignals слугує ефективним мостом для міжпотокової комунікації. Цей клас містить об'єкти pyqtSignal – спеціальні сигнали, які можуть бути випромінювані з одного потоку та "прийняті" іншим. Наприклад, сигнали result (для передачі списку рекомендованих фільмів), desc_result (для передачі даних про опис, постер та трейлер обраного фільму) та error (для повідомлень про виникнення помилок) є ключовими для цієї взаємодії. Фоновий потік, після успішного виконання своєї задачі (наприклад, отримання рекомендацій від Gemini) або у випадку виникнення будь-якого винятку, випромінює відповідний сигнал разом з обробленими даними або деталізованим повідомленням про помилку. Методи класу MainForm (або інших класів, що відповідають за UI-логіку) "підключені" до цих сигналів як "слоти". Отже, коли фоновий по-

тік випромінює сигнал, відповідний слот автоматично викликається в контексті основного потоку GUI. Це дозволяє безпечно та безпосередньо оновити елементи графічного інтерфейсу (наприклад, оновити QListWidget зі списком фільмів або відобразити повідомлення про помилку) без ризику виникнення проблем синхронізації або перегонів даних, що є типовими для багатопотокових програм [13]. Цей механізм є надзвичайно важливим для підтримки стабільності та надійності системи. Додатково, для забезпечення візуального зворотного зв'язку про прогрес цих асинхронних операцій, користувачеві надається інформація через QProgressBar, який динамічно оновлюється під час виконання мережевих запитів, інформуючи про тривалість очікування та підтверджуючи, що система активно працює. Це значно підвищує користувацький комфорт, зменшуючи відчуття затримки та невідомості.

3.3 Проєктування та розробка алгоритмів обробки даних

Фундаментом ефективності та інноваційності розробленої системи є її ретельно спроєктовані алгоритми обробки даних. Ці алгоритмічні рішення охоплюють ключові процеси: від генерації рекомендацій фільмів, що є серцем функціоналу, до деталізації інформації про обрані стрічки та персоналізації досвіду користувача через розумне локальне зберігання та аналіз даних. Кожен алгоритм розроблений з урахуванням оптимізації продуктивності, точності та зручності взаємодії.

Ключовим алгоритмом системи, що забезпечує її основний функціонал, є процес генерації рекомендацій фільмів, який безпосередньо використовує хмарну модель Google Gemini. Цей алгоритм починається з отримання вхідних параметрів від користувача через графічний інтерфейс. Ці параметри включають обраний жанр, який є традиційним фільтром, а також два інноваційні для подібних систем критерії: поточний емоційний настрій та бажаний діапазон років випуску фільмів. Емоційний настрій, обраний за допомогою спеціального повзунка (mood_slider) на інтерфейсі користувача, є не просто числовим значен-

ням; він внутрішньо трансформується в одну з п'яти дискретних категорій емодзі, кожна з яких попередньо асоційована з певним набором жанрів, визначених у спеціально розробленому словнику mood_genres. Ця асоціація дозволяє системі не просто збирати дані, а й інтерпретувати емоційний контекст запиту. Далі всі зібрані параметри – жанри, перетворений емоційний стан та діапазон років – динамічно інтегруються у структурований текстовий запит, відомий як "промпт", який надсилається до хмарної моделі Google Gemini. Наприклад, для отримання комедійних фільмів, що відповідають гарному настрою користувача, та випущених у певному діапазоні, промпт буде сформований з особливою точністю, щоб максимізувати релевантність відповіді від генеративної моделі. Приклад такого промпту може виглядати приблизно так: "Надай мені список з 10 назв комедійних фільмів англійською мовою для людини в гарному настрої, випущених з 2000 по 2025 рік. Лише назви, без описів, без номерів, без маркерів, один фільм на рядок." Модель штучного інтелекту, використовуючи свої розширені можливості обробки природної мови та доступ до величезної бази знань, генерує список фільмів, що максимально відповідає заданим критеріям. Отримана текстова відповідь від API далі проходить ще один обов'язковий та критично важливий етап обробки. Алгоритм виконує ретельний парсинг тексту, очищаючи його від будь-яких зайвих символів, які можуть бути результатом генерації (таких як маркери списків, номери, зайві пробіли або інші елементи форматування), та розділяючи отриманий потік тексту на окремі, чисті назви фільмів. Цей очищений та структурований список назв фільмів потім передається для відображення користувачеві у вигляді інтерактивного списку. Паралельно з цим, обрані користувачем жанр та діапазон років оперативно фіксуються та передаються до окремого алгоритму збору статистики для подальшого оновлення профілю користувача, що забезпечує функцію персоналізації.

При виборі користувачем конкретного фільму зі списку рекомендацій, отриманого від Gemini, або ж з персонального розділу "Улюблені фільми", запускається багатоетапний алгоритм отримання детальної інформації, що забезпечує швидкий та повний доступ до необхідних даних. Першим кроком є ефек-

тивне використання механізму локального кешу описів. Система перевіряє, чи існує опис для обраного фільму у спеціальному файлі descriptions.txt. Це значно прискорює процес завантаження інформації, оскільки уникнення повторного мережевого запиту до віддаленого сервісу дозволяє миттєво відобразити опис, якщо він вже був отриманий раніше. Якщо опис вже існує в кеші, він одразу зчитується з локального файлу та відображається в інтерфейсі, суттєво зменшуючи затримки та знижуючи навантаження на API. Якщо ж опис відсутній у кеші, система оперативно формує новий промпт для Google Gemini API, запитуючи лише детальний опис для конкретної назви фільму. Одночасно або після отримання опису від Gemini, ініціюється послідовність запитів до TMDb API (The Movie Database) – всесвітньо відомого ресурсу, який надає обширну базу даних з метаданими про фільми, включаючи посилання на трейлери та постери. Спочатку виконується пошук фільму за його назвою за допомогою методу search/movie TMDb API. Алгоритм обробки результатів цього пошуку розроблений таким чином, щоб спочатку намагатися знайти максимально точний збіг назви фільму серед отриманих результатів, щоб забезпечити найвищу релевантність. Якщо точного збігу не знайдено (наприклад, через незначні відмінності у назвах або відсутність фільму), використовується перший найбільш релевантний результат, який повертає API. Після успішного отримання унікального ідентифікатора фільму (movie_id) від TMDb, виконується другий запит до того ж API, але вже за методом movie/{movie_id}/videos та movie/{movie_id}/images, для отримання списку пов'язаних відео (трейлерів, тизерів) та посилання на постер фільму. Алгоритм детально перевіряє отриманий список відео, щоб знайти офіційний трейлер, що належить до типу "Trailer" (або "Official Trailer") та розміщений на платформі YouTube, оскільки саме YouTube-трейлери можуть бути безпосередньо вбудовані в інтерфейс. Нарешті, формуються повні URL-адреси для зображення постера та для вбудованого плеєра YouTube, які потім передаються для відображення на сторінці детального опису (description_page), забезпечуючи повноцінну мультимедійну презентацію фільму.

Для забезпечення функціоналу "Улюблені фільми" та "Профіль користувача" були розроблені спеціалізовані алгоритми локального зберігання та обробки даних, що підкреслює пріоритет конфіденційності користувача. Список улюблених фільмів зберігається у простому текстовому файлі favourites.txt, де кожен фільм займає окремий рядок. Цей вибір формату забезпечує максимальну простоту та прозорість для користувача, не вимагаючи складних механізмів управління базою даних. Алгоритми додавання та видалення фільмів до цього списку обробляють дані безпосередньо у пам'яті програми, а потім перезаписують файл favourites.txt для збереження всіх змін. Це дозволяє користувачеві повністю та конфіденційно керувати своїм особистим списком улюбленого кіно без залучення зовнішніх серверів або збереження історії на віддалених ресурсах. Описи фільмів, які були успішно отримані від Google Gemini API, кешуються у файлі descriptions.txt. Метою цього кешування є не тільки прискорення завантаження інформації при повторному запиті до того ж фільму, але й значне зниження навантаження на зовнішні API та мінімізація мережевого трафіку.

Найбільш цікавим з точки зору аналізу локальних даних та персоналізації є алгоритм збору та відображення статистики користувача. Ця статистика зберігається у файлі user_stats.json у стандартному форматі JSON, який дозволяє легко структурувати дані. Файл містить два ключові списки: "genres" – для збереження обраних користувачем жанрів, та "years" – для збереження діапазонів років, які він задавав у своїх запитах. Кожного разу, коли користувач успішно робить запит на рекомендації, обраний жанр та діапазон років автоматично додаються до відповідних списків у структурі даних user_stats та зберігаються. При перегляді розділу профілю користувача (profile_page), алгоритм зчитує ці списки з файлу. Для визначення улюблених жанрів та найбільш часто обираних років застосовується ефективний алгоритм підрахунку частотності, реалізований за допомогою вбудованого класу collections.Counter з бібліотеки Python. Цей клас дозволяє швидко та ефективно підрахувати кількість виборів кожного унікального жанру та кожного унікального діапазону років. Після цього система обирає три найбільш поширені елементи з кожного списку (наприклад, три

найпопулярніші жанри та три найпопулярніші діапазони років) і відображає їх у візуально привабливому форматі у профілі користувача. Це надає користувачеві унікальний інсайт у його власні кінематографічні вподобання, показуючи, як його вибір формує його персональний смак.

3.4 Реалізація користувацького інтерфейсу

Для створення та реалізації графічного інтерфейсу користувача (GUI) розробленої системи була обрана бібліотека PyQt6. Цей вибір є цілком логічним та обґрунтованим, зумовленим низкою її виняткових переваг, що роблять її ідеальним інструментом для розробки складних, але водночас чуйних та візуально привабливих десктопних додатків. PyQt6 надає розробникам доступ до величезної кількості високоякісних графічних віджетів, які охоплюють весь спектр елементів керування, від простих кнопок та текстових полів до складних таблиць та мультимедійних плеєрів. Крім того, вона пропонує потужні інструменти для компонування елементів, що дозволяє створювати гнучкі та адаптивні макети для різних розмірів екранів. Архітектурна модель PyQt6, яка природно підтримує парадигму Model-View-Controller (MVC) або її варіації, дозволяє чітко розмежувати візуальне представлення інтерфейсу від базової логіки програми. Це сприяє підтримці чистоти коду, його модульності та значно полегшує подальше масштабування системи, оскільки зміни в UI не впливають на бізнес-логіку і навпаки. Однією з найзначніших переваг PyQt6 є її кросплатформність. Ця властивість дозволяє розробленому додатку без будь-яких модифікацій коду працювати у різних операційних системах, таких як Microsoft Windows, macOS та різні дистрибутиви Linux. Це суттєво розширює потенційну аудиторію користувачів та значно знижує витрати на підтримку та розгортання, оскільки немає потреби в розробці та підтримці окремих версій програми для кожної платформи.

Центральним архітектурним елементом інтерфейсу користувача є головна форма, інкапсульована у клас MainForm. Цей клас слугує єдиною точкою

входу для всього графічного функціоналу та координує взаємодію між різними частинами інтерфейсу. Всередині MainForm інтегровано віджет QStackedWidget – потужний динамічний контейнер, який дозволяє організувати декілька візуальних сторінок або "вікон" в єдиному вікні програми. Таке рішення значно покращує користувацький досвід, оскільки усуває необхідність відкриття численних окремих вікон, що може бути заплутаним і незручним. Завдяки QStackedWidget, переходи між різними функціональними розділами системи відбуваються плавно та безперервно, створюючи відчуття єдиного, інтегрованого додатку. Система функціонально поділена на п'ять основних сторінок, кожна з яких має унікальне призначення та ретельно підібраний набір віджетів, що відповідають її функціоналу, забезпечуючи оптимальну ергономіку.

Початкова точка взаємодії користувача з системою – це selection_page, або сторінка вибору параметрів. Ця сторінка відповідає за збір вхідних даних, необхідних для генерації рекомендацій. Компонування елементів на цій сторінці реалізовано за допомогою гнучких менеджерів компонування QVBoxLayout (вертикальний макет) та QHBoxLayout (горизонтальний макет), що забезпечує адаптивний та охайний вигляд інтерфейсу. Користувач взаємодіє з декількома ключовими елементами управління: віджети QLabel надають чіткі текстові підказки та інструкції, спрямовуючи користувача. QComboBox дозволяє обрати основний жанр фільмів зі спливаючого списку, пропонуючи широкий діапазон категорій. Для визначення емоційного настрою користувача інтегрований QSlider (повзунок настрою), який дозволяє точно вказати бажаний рівень настрою (від 0 до 100), а його візуальне представлення динамічно змінюється, можливо, відображаючи відповідні емодзі або текстові мітки, що поглиблює взаємодію. Діапазон років випуску фільмів задається за допомогою двох окремих горизонтальних QSlider – year_start_slider та year_end_slider. Вони дозволяють користувачеві визначити конкретні часові проміжки, які цікавлять глядача, відбираючи як класику, так і найновіші релізи. Після встановлення всіх необхідних параметрів, кнопка QPushButton з текстом "Get Recommendations" іні-

ціює асинхронний процес пошуку та обробки фільмів, а окрема кнопка "My Profile" надає швидкий доступ до персонального профілю користувача.

Після успішного отримання рекомендацій від хмарної моделі ШІ, система автоматично перемикається на results_page – сторінку рекомендованих фільмів. Її центральним та найбільш функціональним елементом є QListWidget з ідентифікатором movie_list. Цей віджет слугує для візуалізації списку рекомендованих назв фільмів у зрозумілому та легкому для сприйняття форматі. Кожен елемент у цьому списку є інтерактивним; простий клік на ньому ініціює фоновий процес завантаження детального опису та мультимедійних даних (постер, трейлер) для обраного фільму, що мінімізує очікування користувача. Кнопка QPushButton з текстом "Back" дозволяє користувачеві повернутися до сторінки вибору початкових параметрів, зберігаючи при цьому стан введених даних.

Коли користувач обирає фільм зі списку рекомендацій або з розділу "Улюблені фільми", додаток переходить на окрему сторінку детального опису фільму (description_page). Ця сторінка реалізована за допомогою QHBoxLayout, який ефективно організовує її вміст у дві чітко розділені візуальні колонки. Ліва колонка містить QTextEdit з ідентифікатором description, призначений для відображення детального синопсису фільму, що дозволяє користувачеві швидко ознайомитися з сюжетом та іншими подробицями. Безпосередньо під описом розташований QWebEngineView (trailer_view). Це потужний віджет, який здатний вбудовувати та відтворювати відео трейлери безпосередньо з платформи YouTube, забезпечуючи повноцінний мультимедійний досвід без необхідності відкривати зовнішні веб-браузери. Права колонка цілком присвячена візуалізації QLabel (poster_label), на якому відображається постер фільму, завантажений з бази даних TMDb. Зображення автоматично масштабується для оптимального відповідності розміру віджета, забезпечуючи приємний візуальний вигляд, а у випадку будь-яких помилок завантаження постера (наприклад, відсутності зображення в базі даних) користувач отримує відповідне текстове повідомлення, що покращує стійкість інтерфейсу. Нижче постера розміщені дві функціональні

кнопки QPushButton: "Add to Favourites" для збереження фільму до особистої колекції користувача та "Back" для повернення на попередню сторінку.

Для управління особистою колекцією фільмів розроблено favourites_page – окрему сторінку улюблених фільмів. Цей розділ дозволяє користувачеві переглядати список збережених ним фільмів, представлений у QListWidget з ідентифікатором fav_list. Кожен елемент у цьому списку є не просто текстовим рядком, а спеціально створеним кастомним віджетом (create_widget_with_button). Він містить назву фільму та компактну кнопку для його зручного видалення зі списку, що забезпечує високий рівень інтерактивності та повний контроль користувача над своєю особистою колекцією. Як і на сторінці рекомендацій, клік на назві фільму у списку улюблених також ініціює перехід до сторінки з його детальним описом, забезпечуючи безшовну навігацію. Кнопка QPushButton з текстом "Back to Profile" відповідає за повернення до персонального профілю користувача.

І, нарешті, profile_page – сторінка профілю користувача – надає користувачеві цінну персоналізовану статистику його кінематографічних уподобань. За допомогою віджетів QLabel (fav_genres_label та fav_years_label) відображаються три найбільш часто обрані жанри та діапазони років, що були запитані користувачем. Ці дані обчислюються на основі локально збереженої історії запитів, надаючи користувачеві інсайт у його власні смаки. Для візуального розділення секцій та покращення читабельності інтерфейсу на цій сторінці використовується QFrame з типом HLine, що створює елегантну горизонтальну розділювальну лінію. Загалом, ці ретельно спроєктовані та реалізовані елементи інтерфейсу PyQt6 створюють цілісну, інтуїтивно зрозумілу та функціональну систему для персоналізованого підбору фільмів.

4 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Тестування інтерфейсу користувача

Тестування користувацького інтерфейсу (UI) було спрямоване на всебічне підтвердження коректності відображення візуальних елементів, функціональності всіх інтерактивних компонентів та забезпечення безперебійної та інтуїтивно зрозумілої навігації між різними розділами додатку [14]. Мета цього етапу полягала у забезпеченні позитивного та ефективного користувацького досвіду, де кожен елемент інтерфейсу працює так, як очікується. Ретельно перевірялася функціональність кожного віджета, від кнопок до полів вводу та слайдерів, на всіх сторінках програми.

На selection_page, яка є початковою точкою взаємодії користувача із системою, було проведено детальне тестування. Перевірялася коректність вибору жанрів з елемента QComboBox (випадаючого списку), включаючи перевірку всіх доступних опцій, їх вибір та відображення.

Особливу увагу було приділено плавності та точності роботи QSlider для налаштування параметрів настрою, який є унікальним критерієм системи, а також діапазону років випуску фільму. Перевірялося, як змінюються значення при перетягуванні повзунка, чи коректно вони відображаються, та чи не виникає візуальних артефактів. Після вибору всіх необхідних параметрів, тестувалася активація кнопки "Get Recommendations" та коректність подальшого переходу на сторінку результатів (results_page). Це включало перевірку того, що перехід відбувається лише після валідного вводу даних.

На сторінці рекомендованих фільмів (results_page), яка відображає підібраний контент, ключову увагу було приділено коректному відображенню списку фільмів у QListWidget. Тестувалася його здатність обробляти різну кількість результатів – від одного фільму до великого списку, перевіряючи, чи коректно відображаються назви, і чи не виникає проблем з прокруткою.

Важливим аспектом була інтерактивність елементів списку: тестувалися кліки на назвах фільмів для отримання детального опису, перевіряючи, чи кожен клік викликає правильний фільм на сторінці опису. Функціональність кнопки "Back" на цій сторінці також була ретельно перевірена, щоб забезпечити коректне повернення до сторінки вибору параметрів без втрати даних або некоректних станів.

Особлива увага приділялася тестуванню description_page – сторінки з детальною інформацією про обраний фільм. Тут перевірялося коректне завантаження та відображення повного опису фільму у QTextEdit, включаючи форматування тексту та його відповідність вмісту. Важливим було підтвердження відповідності постера (poster_label) обраному фільму – чи завантажується правильне зображення обкладинки.

Ключовим елементом на цій сторінці є функціональність QWebEngineView для відтворення трейлерів з YouTube. Тестувалася не тільки можливість відтворення, але й коректне завантаження, контроль відтворення та паузи. Важливим аспектом цього етапу було тестування поведінки системи при граничних випадках та відхиленнях: наприклад, що відбувається, якщо для фільму відсутній постер або трейлер (чи відображається заглушка, чи система коректно обробляє помилку), а також коректне масштабування зображень та відео при зміні розмірів вікна. Функціональність кнопок "Add to Favourites" (додати до улюблених) та "Back" також була ретельно перевірена, включаючи перевірку додавання фільму до списку та його відображення у відповідному розділі.

На favourites_page, що відображає список збережених користувачем фільмів, тестувалася коректність відображення доданих фільмів у QListWidget. Перевірялася працездатність кнопок видалення для кожного елемента списку, а також коректність їх зникнення після видалення.

Важливо було переконатися, що видалення одного елемента не впливає на інші та що зміни зберігаються після перезапуску програми. Також перевірялася можливість переходу до детального опису фільму з цього списку, що забезпечує повний цикл взаємодії з улюбленим контентом.

На сторінці профілю (profile_page) тестувалася правильність відображення обчисленої статистики у fav_genres_label та fav_years_label. Перевірялося, чи коректно відображаються найбільш часто обирані жанри та роки випуску фільмів, а також чи оновлюється ця статистика після додавання нових фільмів до улюблених або після нових запитів.

Функціональність кнопок для переходу до улюблених фільмів та повернення на головну сторінку також була підтверджена.

Цей комплексний підхід до тестування UI гарантував, що користувацький інтерфейс є не тільки візуально привабливим, але й повністю функціональним, забезпечуючи безперебійний та приємний досвід використання системи.

4.2 Тестування API-інтеграцій

Тестування API-інтеграцій [15] є критично важливим етапом валідації системи, оскільки її основний функціонал повністю залежить від коректної та стабільної взаємодії із зовнішніми хмарними сервісами. Основна мета цього тестування полягала у всебічній перевірці надсилання запитів, правильності обробки отриманих відповідей та стійкості системи до можливих помилок або нестабільних умов мережі. Це забезпечує надійність роботи інтелектуальної складової та доступність рекомендацій.

Тестування взаємодії з Google Gemini API включало ретельну перевірку коректності формування промптів – текстових запитів, що надсилаються до моделі ШІ. Було підтверджено, що промпти формуються з урахуванням усіх вхідних параметрів користувача: обраного жанру, вказаного настрою та заданого діапазону років. Надсилалися запити з різними, найрізноманітнішими комбінаціями параметрів, включаючи вибір усіх можливих жанрів одночасно, а також широкий діапазон років, щоб перевірити, чи модель коректно інтерпретує складні запити.

Окремо тестувався алгоритм парсингу відповіді від Gemini. Це включало перевірку його здатності коректно витягувати назви фільмів та їхні короткі

описи з текстового формату, ігноруючи сторонні символи, небажані фрагменти тексту або непередбачуване форматування, яке може надходити від генеративної моделі. Це гарантувало, що користувач отримує чисті та зрозумілі результати.

Також, тестувалася обробка помилок у випадку, якщо Gemini не може надати рекомендації (наприклад, за дуже специфічним або некоректним запитом, який не має відповідних фільмів у базі знань моделі) або якщо виникають проблеми з мережевим з'єднанням (таймаут, відсутність інтернету). Перевірялося, чи система коректно повідомляє користувача про такі ситуації, чи не зависає, і чи дозволяє продовжити роботу. Запити на отримання детальних описів фільмів, які використовуються для сторінки description_page, також були предметом ретельного тестування.

Перевірялося, чи коректно генерується опис для заданої назви фільму, чи збігається він з очікуваними даними, і чи немає розбіжностей. Додатково, було проведено тестування взаємодії з TMDb (The Movie Database) API для отримання постерів та трейлерів. Перевірялася коректність надсилання запитів за назвою фільму, обробка відповідей, включаючи випадки, коли постер або трейлер відсутні у базі даних TMDb.

Усі ці інтеграції були протестовані не тільки на функціональність, але й на предмет їхньої стабільності при повторних запитах та під навантаженням, щоб переконатися, що система може витримувати інтенсивне використання без збоїв.

4.3 Тестування локального зберігання даних

Тестування локального зберігання даних було зосереджено на забезпеченні надійності та коректності операцій збереження, зчитування та маніпулювання даними, що зберігаються безпосередньо на пристрої користувача. Це є критично важливим для функціоналу улюблених фільмів, ефективного кешу-

вання описів та збору статистики, що підвищує продуктивність та персоналізацію системи.

Тестування збереження та завантаження улюблених фільмів включало повний цикл взаємодії з цією функціональністю. Було проведено сценарії додавання різних фільмів до списку улюблених, перевіряючи, чи коректно вони з'являються у списку на favourites_page.

Потім тестувалося їх видалення, забезпечуючи, що фільми зникають зі списку та з файлу зберігання. Найважливішим аспектом була перевірка збереження цих змін після перезапуску програми: чи залишаються додані фільми після закриття та повторного відкриття додатку, і чи дійсно видалені фільми не з'являються знову. Були проведені також тести на додавання великої кількості фільмів, щоб переконатися, що система коректно обробляє зростаючий обсяг даних без уповільнень або помилок.

Тестування кешування описів фільмів зосередилося на перевірці логіки роботи з файлом descriptions.txt, який зберігає кешовані описи. Сценарії тестування включали:

1. Первинне збереження: Перевірку, чи опис фільму коректно зберігається у файлі descriptions.txt після першого успішного запиту до Gemini API для цього фільму.
2. Повторне завантаження: Перевірку, чи коректно завантажується кешований опис при повторному зверненні до того ж фільму, уникаючи повторного дороговартісного запиту до API. Це підтвердило, що кешування дійсно підвищує швидкість завантаження інформації та зменшує залежність від мережевих запитів, забезпечуючи швидший відгук.
3. Граничні випадки: Перевірку поведінки системи при порожньому файлі кешу, неіснуючому файлі кешу, або пошкодженому файлі, щоб забезпечити її стійкість.

Найбільш комплексним було тестування збору та аналізу статистики профілю користувача, що зберігається у файлі user_stats.json. Перевірялася коректність додавання обраних жанрів та діапазонів років до відповідних внутрі-

шніх списків та їх подальшого збереження після кожного запиту на рекомендації.

Проводилися тести з різними послідовностями вибору жанрів та років, щоб забезпечити, що статистика накопичується правильно, враховуючи повтори та унікальні значення.

Особлива увага приділялася алгоритму підрахунку частотності, реалізованому, наприклад, за допомогою collections.Counter: перевірялася коректність визначення трьох найбільш популярних жанрів та років, навіть у складних випадках, коли декілька елементів мають однакову частоту. Також тестувалася поведінка системи при першому запуску (коли файл статистики порожній або відсутній) та при великій кількості накопичених даних, щоб переконатися, що обчислення виконуються швидко та без помилок, а дані коректно відображаються у профілі користувача, надаючи йому достовірний інсайт у власні вподобання.

4.4 Результати тестування

За результатами проведеного багатоетапного та всебічного тестування, розроблена система підбору фільмів продемонструвала високий рівень функціональності, стабільності та надійності, що підтверджує її готовність до використання.

Під час тестування були виявлені та успішно усунені незначні дефекти, переважно пов'язані з граничними випадками обробки даних, що надходили від зовнішніх API, та з оптимізацією асинхронного оновлення деяких елементів інтерфейсу у дуже специфічних умовах завантаження. Зокрема, було оперативно скориговано логіку парсингу відповідей від Google Gemini API для більш надійного та точного вилучення назв фільмів та їхніх описів з непередбачуваних форматів генеративної моделі. Також доопрацьовано механізми обробки ситуацій, коли постерів чи трейлерів не виявлялося у базі даних TMDb, забезпечуючи відображення коректних заглушок замість помилок.

Крім того, було виявлено і виправлено ситуації, коли тривале очікування відповіді від API могло тимчасово блокувати деякі елементи інтерфейсу, що було вирішено за допомогою додаткового контролю станів робочих потоків та покращеної синхронізації.

Система успішно пройшла всі функціональні тести, підтвердивши свою здатність коректно генерувати персоналізовані рекомендації фільмів на основі вхідних параметрів (настрій, жанр, рік випуску). Було підтверджено, що вона здатна надавати детальну інформацію про фільми, включаючи якісні описи, релевантні постери та функціональні трейлери.

Також було підтверджено ефективне керування локальними даними користувача, включаючи збереження улюблених фільмів, кешування описів та накопичення статистики вподобань. Тестування інтеграції з Google Gemini API та TMDb API підтвердило високу надійність цих зв'язків та коректність обробки всіх даних, що надходять від цих ключових зовнішніх сервісів, забезпечуючи безперебійний обмін інформацією.

Тестування користувацького інтерфейсу виявило його високу інтуїтивність та зручність у використанні, а всі візуальні та інтерактивні віджети демонстрували очікувану поведінку без збоїв. Загальний показник якості програмного забезпечення оцінюється як високий, з огляду на успішне виконання всіх заявлених функціональних вимог та стабільну роботу в усіх різноманітних тестових сценаріях, що симулювали як типове, так і граничне використання.

Незважаючи на успішні результати та високу функціональність, завжди існують напрямки для подальшого вдосконалення та підвищення якості системи. Рекомендації щодо подальшого покращення можуть включати:

1. Розширення автоматизованого тестування: Впровадження більш повного покриття коду за допомогою автоматизованих тестів (юніт-тестів, інтеграційних тестів, UI-тестів) для забезпечення швидшої регресії при майбутніх змінах та зменшення часу на ручне тестування.
2. Впровадження моніторингу продуктивності: Запровадження інструментів для постійного моніторингу продуктивності системи в реальних умовах,

що дозволить виявляти потенційні "вузькі місця" або вразливі місця при інтенсивному використанні та оперативно реагувати на них.

3. Проведення юзабіліті-тестування: Організація тестування за участю реальних кінцевих користувачів, які не є розробниками. Це дозволить зібрати цінний якісний зворотний зв'язок щодо зручності використання, інтуїтивності інтерфейсу та задоволеності загальним досвідом, що сприятиме подальшому вдосконаленню користувацького досвіду та загальної привабливості розробленої системи. Це дозволить ще більше підвищити надійність, ефективність та привабливість розробленої системи у довгостроковій перспективі.

ВИСНОВКИ

У процесі виконання даної кваліфікаційної роботи була успішно розроблена, імплементована та апробована програмна система, яка представляє собою інноваційний підхід до інтелектуального підбору фільмів.

Ключовим досягненням цієї розробки стало забезпечення високоперсоналізованої рекомендації кінематографічного контенту, що базується на унікальному для подібних систем наборі параметрів, визначених самим користувачем. Зокрема, система пропонує фільми з урахуванням таких важливих категорій, як бажаний жанр, поточний емоційний настрій користувача, що є її особливою інновацією, та часові рамки випуску фільму.

Цей підхід дозволив подолати обмеження традиційних рекомендаційних систем, які часто зосереджуються лише на узагальнених даних про популярність або історію переглядів, і створити рішення, що дійсно відповідає індивідуальним потребам та контексту перегляду користувача.

Для досягнення поставлених цілей було визначено та успішно впроваджено оптимальну двокомпонентну архітектуру програмного забезпечення. Перший компонент – це інтуїтивно зрозумілий графічний інтерфейс користувача (GUI), розроблений за допомогою бібліотеки Tkinter, що забезпечує легку та ефективну взаємодію користувача із системою. Цей інтерфейс дозволяє без зусиль вводити необхідні критерії для підбору фільмів та візуалізувати отримані рекомендації.

Другий, і центральний, компонент – це інтелектуальний нейромережевий модуль, відповідальний за обробку запитів та генерацію рекомендацій. Основу цього модуля склав потужний генеративний штучний інтелект – сервіс Gemini 1.5 Flash від Google. Цей вибір забезпечив доступ до передових можливостей обробки природної мови та складних алгоритмів машинного навчання, дозволивши системі не просто шукати за збігом ключових слів, а й інтерпретувати нюанси настрою та контексту запиту користувача.

В рамках архітектури було ретельно розроблено та реалізовано повний цикл функціональних можливостей системи, що охоплює всі етапи взаємодії: від ефективного збору вхідних даних від користувача через графічний інтерфейс, до формування складних, але оптимальних запитів до нейромережі Gemini 1.5 Flash. Після отримання відповіді від хмарного сервісу, система здійснює її оперативну обробку та подальше візуально привабливе та зручне для сприйняття відображення результатів у вигляді списку рекомендованих фільмів, кожен з яких супроводжується коротким описом та можливістю перегляду трейлера.

Оптимізована логіка взаємодії між основними компонентами системи, включаючи реалізацію асинхронних операцій та використання черг для безпечного обміну даними між потоками, дозволила досягти високої продуктивності та чуйності інтерфейсу, що є критично важливим для комфортного користувацького досвіду.

В процесі розробки були ретельно визначені та застосовані відповідні технічні характеристики та середовище реалізації, що гарантують ефективну та швидку роботу програми на різних платформах.

Вибір мови програмування Python 3.10 з її широким набором бібліотек, включаючи google.generativeai для взаємодії з API та tkinter для GUI, забезпечив гнучкість розробки та кросплатформність додатку. Особлива увага була приділена реалізації асинхронних операцій за допомогою модулів threading та queue, що дозволило виконувати ресурсомісткі мережеві запити до Gemini API у фоновому режимі, не блокуючи основний потік графічного інтерфейсу. Це забезпечило постійну чуйність програми, навіть під час очікування відповіді від хмарних сервісів, що значно покращило користувацький досвід та сприйняття швидкості роботи.

Апаратні вимоги до локального комп'ютера користувача були мінімізовані, оскільки основні обчислення виконуються на стороні хмарного ШІ, що робить систему доступною для більшості сучасних пристроїв. Були чітко ідентифіковані основна цільова аудиторія користувачів, які стикаються з проблемою

інформаційного перевантаження на стрімінгових платформах, та окреслено потенційні сценарії використання системи у реальному середовищі, що включають швидкий вибір фільму для вечора, пошук контенту під конкретний настрій або відкриття нових кінематографічних творів.

Для підвищення надійності та стійкості роботи системи були забезпечені базові механізми обробки помилок, що інформують користувача про проблеми з мережевим з'єднанням або з API, дозволяючи йому швидко реагувати на можливі збої.

Таким чином, розроблена система є повноцінним, функціональним та інноваційним рішенням, яке успішно вирішує проблему вибору контенту, надаючи користувачам потужний та інтуїтивно зрозумілий інструмент для персоналізованого пошуку фільмів. Її архітектура, заснована на синергії локального інтерфейсу та хмарного штучного інтелекту, а також увага до конфіденційності користувача, роблять її актуальним та перспективним продуктом на сучасному ринку стрімінгових послуг.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Countries with most content available on Netflix worldwide as of July 2024. URL: https://www.statista.com/statistics/1013571/netflix-library-size-worldwide/ (дата звернення 03.06.2025)
2. Netflix Tech Blog. Personalized Recommendations at Netflix. URL: https://netflixtechblog.com/foundation-model-for-personalized-recommendation-1a0bd8e02d39 (дата звернення 03.06.2025)
3. Streaming Overload? Nielsen Report Finds Average Viewer Takes 7 Minutes To Pick What To Watch; Just One-Third Bother To Check Menu. URL: https://deadline.com/2019/07/streaming-overload-netflix-nielsen-report-average-viewer-takes-7-minutes-to-pick-what-to-watch-1202640213/ (дата звернення 03.06.2025)
4. Aggarwal, C. C. Recommender Systems: The Textbook. – Springer, 2016 - C. 229. (дата звернення 03.06.2025)
5. Raschka S., Mirjalili V. Python Machine Learning. – Packt Publishing, 2022
6. Waterfall model. URL: https://en.wikipedia.org/wiki/Waterfall_model (дата звернення 03.06.2025)
7. Що таке методологія Scrum і коли варто її використовувати URL: https://career.softserveinc.com/uk-ua/stories/what-is-scrum-methodology (дата звернення 03.06.2025)
8. Tkinter GUI Application Development Cookbook. (2019). Packt Publishing.
9. Python threading — Python Docs. URL: https://docs.python.org/3/library/threading.html (дата звернення 03.06.2025)
10. Queue — A synchronized queue class. URL: https://docs.python.org/3/library/queue.html (дата звернення 03.06.2025)
11. Python Software Foundation. Python Documentation. URL: https://docs.python.org/3/ (дата звернення 03.06.2025)
12. PyQt6 Documentation. URL: https://doc.qt.io/qtforpython-6/ (дата звернення 03.06.2025)

13. Python Multithreading: Benefits, Use Cases, and Comparison. URL: https://pieces.app/blog/python-multithreading-benefits-use-cases-and-comparison (дата звернення 03.06.2025)
14. A complete guide to UI testing. URL: https://www.hotjar.com/ui-design/testing/ (дата звернення 03.06.2025)
15. How to run API integration tests. URL: https://www.merge.dev/blog/api-integration-testing (дата звернення 03.06.2025)

ДОДАТКИ

Вінницький національний технічний університет
Кафедра автоматизації та інтелектуальних інформаційних технологій

ЗАТВЕРДЖУЮ
завідувача кафедри АІІТ
д.т.н., професор Олег БІСІКАЛО

_____
(підпис)

«_23_» _березня_ 2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ
на бакалаврську кваліфікаційну роботу
«Розробка додатку для підбору фільмів на основі категорій із використанням штучного інтелекту»
08-31.БКР.011.02.000 ТЗ

Керівник бакалаврської кваліфікаційної роботи
доц. Володимир ГАРМАШ

_____
(підпис)

«_23_» _травня__ 2025 р.

Розробив студент гр. 1ІСТ-21б
_____Максим МЕДВЕДЄВ

_____
(підпис)

«_23_» _травня__ 2025 р.

Вінниця ВНТУ – 2025 рік

Додаток А
(обов'язковий)
Технічне завдання на бакалаврську кваліфікаційну роботу

1 Підстава для проведення робіт

Підставою для виконання бакалаврської кваліфікаційної роботи на тему: «Розробка додатку для підбору фільмів на основі категорій із використанням штучного інтелекту» є наказ № 97 від 20.03.2025 р.

Термін виконання робіт:

початок 03.09.2024 р.
кінець 10.06.2025 р.

2 Мета та вихідні дані для проведення робіт

Метою бакалаврської кваліфікаційної роботи є розробка системи для рекомендацій фільмів, яка працює на основі штучного інтелекту.

Вихідними даними для проведення робіт є індивідуальне завдання на бакалаврську кваліфікаційну роботу від 20.03.2025 р.

3 Етапи виконання робіт

Виконавцем всіх перерахованих в даному розділі етапів є: студент групи **1ІСТ-21б Медведєв Максим Костянтинович** факультету інтелектуальних інформаційних технологій та автоматизації Вінницького національного технічного університету, а замовником є кафедра автоматизації та інтелектуальних інформаційних технологій.

№ Етапу	Зміст етапу	Строки виконання
Е1	Аналіз предметної області та визначення основних етапів розробки	07.03 – 19.03
Е2	Розробка структури додатка	20.04 – 26.04
Е3	Вибір технологічних засобів	27.04 – 09.04
Е4	Розробка програмного забезпечення	10.05 – 01.06
Е5	Аналіз результатів роботи та підготовка роботи до захисту	02.06 – 20.06

4 Призначення і галузь застосування

Система для підбору фільмів, розроблена як настільний додаток, призначена для надання користувачам персоналізованих рекомендацій кінофільмів. Вона дозволяє автоматизувати процес вибору фільму, враховуючи індивідуальні вподобання користувача, його поточний настрій, бажаний жанр та часовий діапазон випуску фільму. Система застосовується у сфері індивідуального дозвілля та розваг для швидкого та ефективного пошуку релевантного контенту, а також для ведення персональної бібліотеки улюблених фільмів.

5 Технічні дані

5.1 основна програмна платформа розробки – Python 3;
5.2 бібліотека для розробки графічного інтерфейсу – PyQt6;
5.3 основний модуль штучного інтелекту – Google Gemini 1.5 Flash API;
5.4 метод зберігання локальних даних – текстові файли (.txt).

6 Джерела розробки

6.1 Tkinter GUI Application Development Cookbook. (2019). Packt Publishing.
6.2 Python Software Foundation. Python Documentation. URL https://docs.python.org/3/
6.3 Python Software Foundation. Python Documentation. URL: https://docs.python.org/3/
6.4 Методичні вказівки до виконання бакалаврських кваліфікаційних робіт (проєктів) для студентів спеціальностей: 126 «Інформаційні системи та технології», «Автоматизація та комп'ютерно-інтегровані технології») / Уклад. К. В. Овчинников, О. В. Бісікало – Вінниця : ВНТУ, 2022. – 50 с.

Здобувач ст. гр. 1ІСТ-21б _____ Максим МЕДВЕДЄВ

Додаток Б
(обов'язковий)

ІЛЮСТРАТИВНА ЧАСТИНА

РОЗРОБКА ДОДАТКУ ДЛЯ ПІДБОРУ ФІЛЬМІВ НА ОСНОВІ КАТЕГОРІЙ ІЗ ВИКОРИСТАННЯМ ШТУЧНОГО ІНТЕЛЕКТУ

Діаграма послідовності

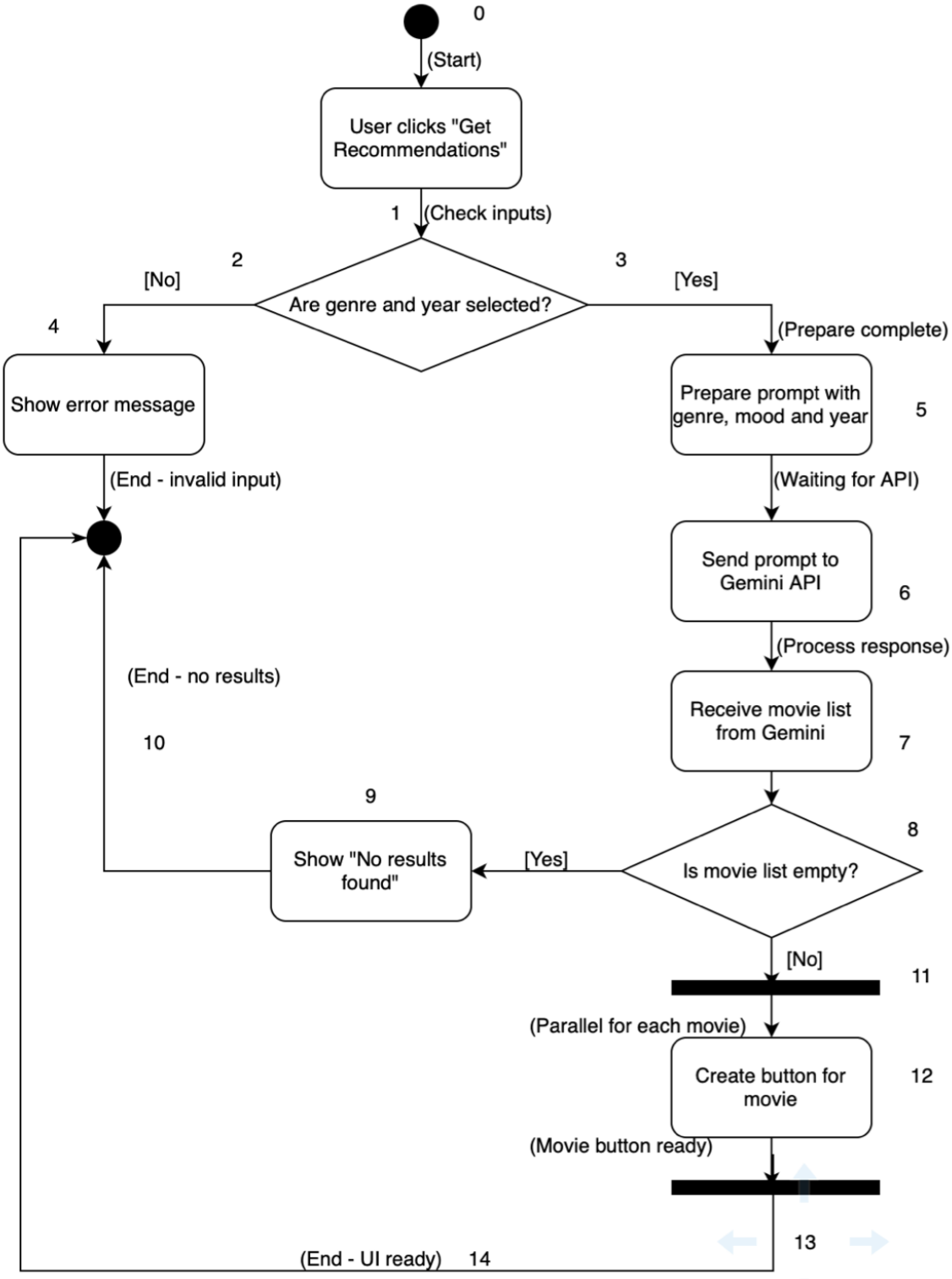

Рисунок Б.1 – Діаграма послідовності

Діаграма класів

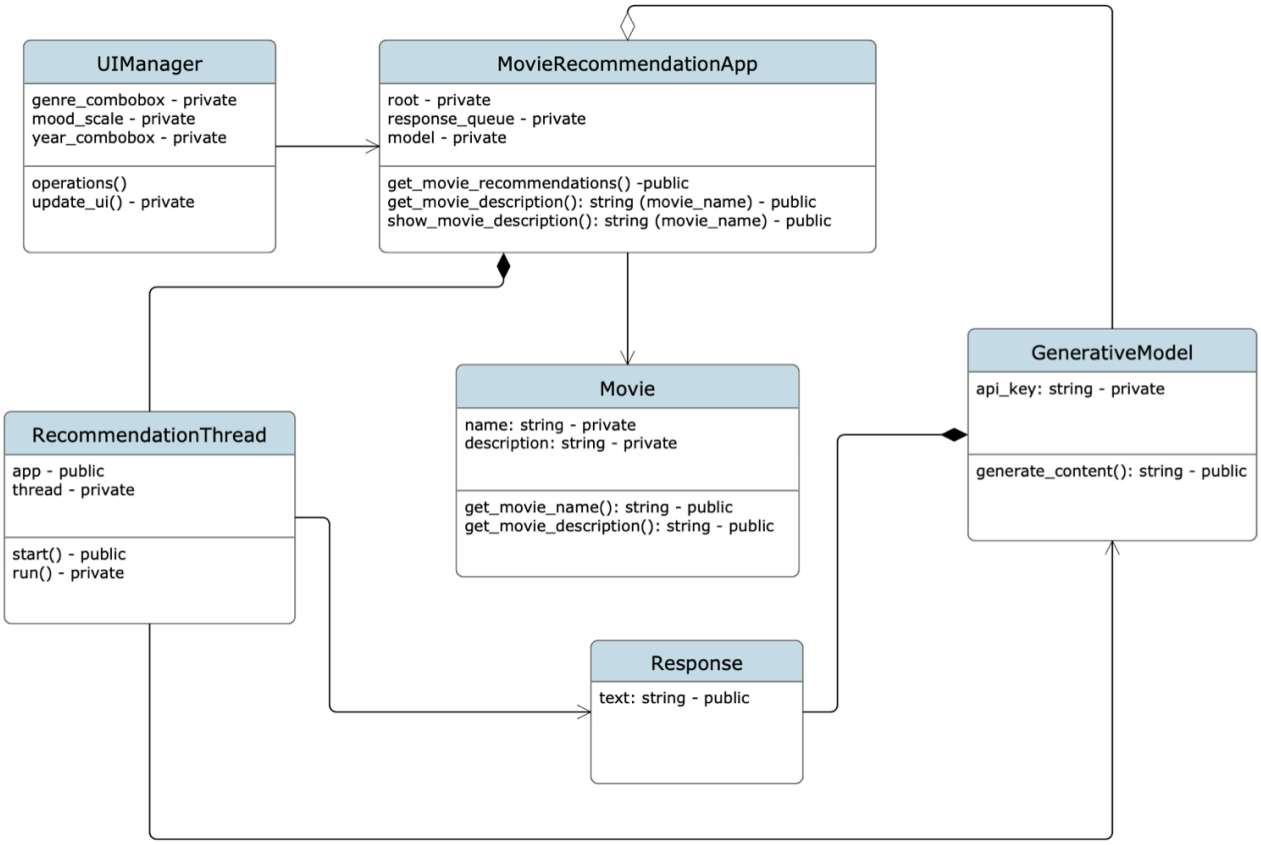

Рисунок Б.2 – Діаграма класів

Діаграми варіантів використання

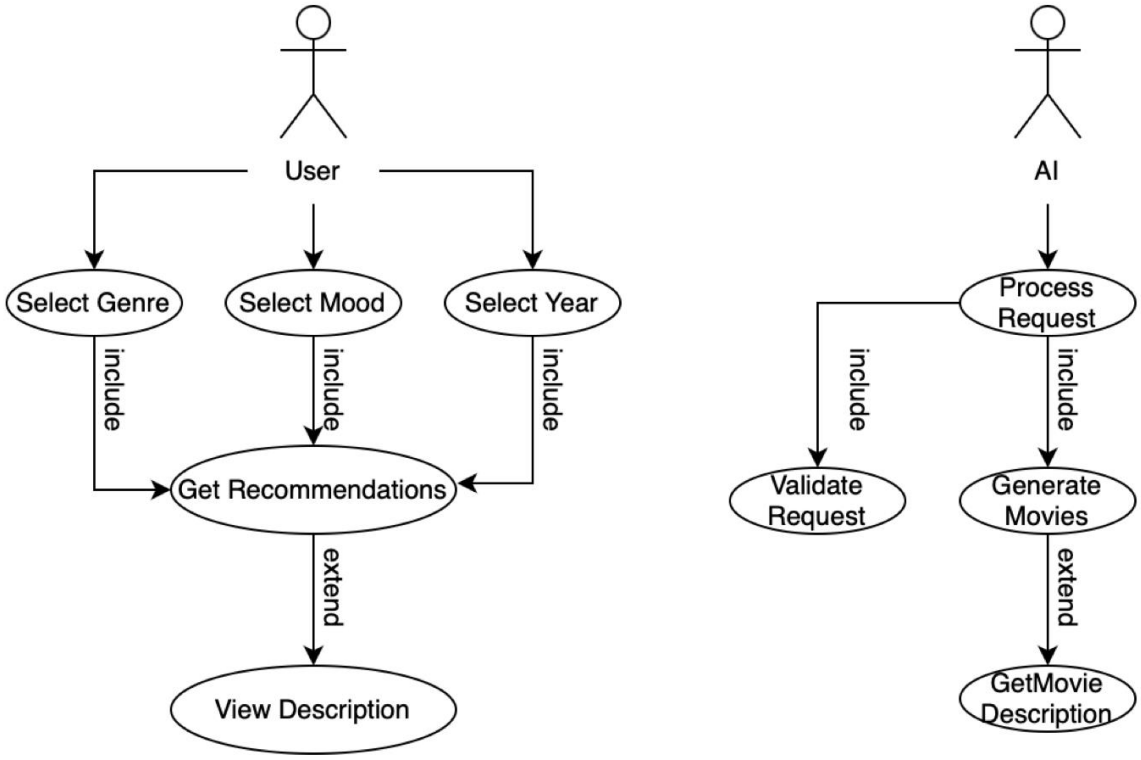

Рисунок Б.3 – Діаграма варіантів використання

Додаток В
(обов'язковий)
Лістинг коду програми

```
import sys
import threading
import os
import json
import requests
from collections import Counter
from PyQt6.QtWidgets import (
    QApplication, QWidget, QVBoxLayout, QLabel, QComboBox, QSlider,
    QPushButton, QListWidget, QListWidgetItem, QTextEdit, QProgressBar,
    QStackedWidget, QHBoxLayout, QMessageBox, QFrame
)
from PyQt6.QtCore import Qt, pyqtSignal, QObject, QUrl, QByteArray
from PyQt6.QtGui import QPixmap, QImage
from PyQt6.QtNetwork import QNetworkAccessManager, QNetworkRequest,
QNetworkReply
from PyQt6.QtWebEngineWidgets import QWebEngineView

import google.generativeai as genai

genai.configure(api_key="**********")
model = genai.GenerativeModel('gemini-1.5-flash')

TMDB_API_KEY = "**********"
TMDB_BASE_URL = "https://api.themoviedb.org/3"
TMDB_IMAGE_BASE_URL = "https://image.tmdb.org/t/p/w500"
TMDB_YOUTUBE_BASE_URL = "https://www.youtube.com/embed/"
```

```
FAVOURITES_FILE = "favourites.txt"
DESCRIPTIONS_FILE = "descriptions.txt"
STATS_FILE = "user_stats.json"

class WorkerSignals(QObject):
    result = pyqtSignal(list)
    desc_result = pyqtSignal(list)
    error = pyqtSignal(str)

class MainForm(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("□ Movie Recommender")
        self.setMinimumSize(700, 600)
        self.layout = QVBoxLayout()
        self.progress = QProgressBar()
        self.layout.addWidget(self.progress)
        self.stacked_widget = QStackedWidget()

        self.last_page_before_description = None

        self.mood_genres = {
            "□": ["Drama", "Romance", "Psychological"],
            "□": ["Drama", "Romance", "Documentary"],
            "□": ["Drama", "Thriller", "Crime", "Biography"],
            "□": ["Comedy", "Adventure", "Romantic Comedy"],
            "□": ["Comedy", "Family", "Musical", "Animation"]
        }

        self.selection_page = QWidget()
```

```
self.selection_layout = QVBoxLayout()
self.selection_layout.addWidget(QLabel("Choose genre:"))
self.genre_combo = QComboBox()
self.selection_layout.addWidget(self.genre_combo)

self.selection_layout.addWidget(QLabel("Mood:"))
self.emoji_label = QLabel("□")
self.emoji_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
self.emoji_label.setStyleSheet("font-size: 32px;")
self.selection_layout.addWidget(self.emoji_label)

self.mood_slider = QSlider(Qt.Orientation.Horizontal)
self.mood_slider.setRange(0, 100)
self.mood_slider.setValue(50)
self.mood_slider.valueChanged.connect(self.update_emoji)
self.selection_layout.addWidget(self.mood_slider)

self.mood_slider.setStyleSheet("""
   QSlider::groove:horizontal {
      border: 1px solid #bbb;
      background: qlineargradient(x1:0, y1:0, x2:1, y2:0,
                        stop:0 #0000FF, stop:1 #FF0000);
      height: 10px;
      border-radius: 4px;
   }
   QSlider::handle:horizontal {
      background: #fff;
      border: 1px solid #777;
      width: 18px;
      margin-top: -4px;
```

```
        margin-bottom: -4px;
        border-radius: 9px;
    }
""")

self.selection_layout.addWidget(QLabel("Choose year range:"))

year_range_layout = QHBoxLayout()
self.year_start_slider = QSlider(Qt.Orientation.Horizontal)
self.year_start_slider.setRange(1900, 2025)
self.year_start_slider.setValue(2000)
self.year_start_slider.valueChanged.connect(self.update_year_range)
self.year_start_label = QLabel("2000")
year_range_layout.addWidget(self.year_start_label)
year_range_layout.addWidget(self.year_start_slider)

self.year_end_slider = QSlider(Qt.Orientation.Horizontal)
self.year_end_slider.setRange(1900, 2025)
self.year_end_slider.setValue(2025)
self.year_end_slider.valueChanged.connect(self.update_year_range)
self.year_end_label = QLabel("2025")
year_range_layout.addWidget(self.year_end_label)
year_range_layout.addWidget(self.year_end_slider)

self.selection_layout.addLayout(year_range_layout)

self.submit_btn = QPushButton("Get Recommendations")
self.submit_btn.clicked.connect(self.get_recommendations)
self.selection_layout.addWidget(self.submit_btn)
```

```
        self.to_profile_btn = QPushButton("□ My Profile")
        self.to_profile_btn.clicked.connect(self.show_profile_page)
        self.selection_layout.addWidget(self.to_profile_btn)

        self.selection_page.setLayout(self.selection_layout)
        self.stacked_widget.addWidget(self.selection_page)

        self.results_page = QWidget()
        self.results_layout = QVBoxLayout()
        self.results_layout.addWidget(QLabel("Recommended Movies:"))
        self.movie_list = QListWidget()
        self.movie_list.itemClicked.connect(lambda item:
self.get_description(item.text(), from_favourites=False))
        self.results_layout.addWidget(self.movie_list)

        self.back_btn = QPushButton("Back")
        self.back_btn.clicked.connect(self.show_selection_page)
        self.results_layout.addWidget(self.back_btn)
        self.results_page.setLayout(self.results_layout)
        self.stacked_widget.addWidget(self.results_page)

        self.description_page = QWidget()
        self.description_layout = QHBoxLayout()

        left_column_layout = QVBoxLayout()
        left_column_layout.addWidget(QLabel("Movie Description:"))
        self.description = QTextEdit()
        self.description.setReadOnly(True)
        left_column_layout.addWidget(self.description)
```

```
self.trailer_view = QWebEngineView()
self.trailer_view.setMinimumHeight(250)
self.trailer_view.hide()
left_column_layout.addWidget(self.trailer_view)

right_column_layout = QVBoxLayout()
self.poster_label = QLabel()
self.poster_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
self.poster_label.setText("Poster not available.")
self.poster_label.setStyleSheet("font-style: italic; color: gray;")
self.poster_label.setFixedSize(250, 375)
right_column_layout.addWidget(self.poster_label)
right_column_layout.addStretch()

self.add_fav_btn = QPushButton("Add to Favourites")
self.add_fav_btn.clicked.connect(self.add_to_favourites)
right_column_layout.addWidget(self.add_fav_btn)

self.back_desc_btn = QPushButton("Back")
self.back_desc_btn.clicked.connect(self.return_from_description)
right_column_layout.addWidget(self.back_desc_btn)
right_column_layout.addStretch()

self.description_layout.addLayout(left_column_layout, 2)
self.description_layout.addLayout(right_column_layout, 1)

self.description_page.setLayout(self.description_layout)
self.stacked_widget.addWidget(self.description_page)

self.favourites_page = QWidget()
```

```
        self.fav_layout = QVBoxLayout()
        self.fav_layout.addWidget(QLabel("Your Favourite Movies:"))
        self.fav_list = QListWidget()
        self.fav_layout.addWidget(self.fav_list)
        self.back_from_fav = QPushButton("Back to Profile")
        self.back_from_fav.clicked.connect(self.show_profile_page)
        self.fav_layout.addWidget(self.back_from_fav)
        self.favourites_page.setLayout(self.fav_layout)
        self.stacked_widget.addWidget(self.favourites_page)

        self.profile_page = QWidget()
        self.profile_layout = QVBoxLayout()
        self.profile_layout.addWidget(QLabel("<h2>My                    Profile</h2>",
alignment=Qt.AlignmentFlag.AlignCenter))

        self.profile_layout.addWidget(QLabel("<h3>Favourite Genres:</h3>"))
        self.fav_genres_label = QLabel("No data yet.")
        self.profile_layout.addWidget(self.fav_genres_label)

        self.profile_layout.addWidget(QLabel("<h3>Favourite Years:</h3>"))
        self.fav_years_label = QLabel("No data yet.")
        self.profile_layout.addWidget(self.fav_years_label)

        line = QFrame()
        line.setFrameShape(QFrame.Shape.HLine)
        line.setFrameShadow(QFrame.Shadow.Sunken)
        self.profile_layout.addWidget(line)

        self.to_fav_from_profile_btn = QPushButton("□ Favourites")
        self.to_fav_from_profile_btn.clicked.connect(self.show_favourites)
```

```
self.profile_layout.addWidget(self.to_fav_from_profile_btn)

self.back_from_profile_btn = QPushButton("Back to Main")
self.back_from_profile_btn.clicked.connect(self.show_selection_page)
self.profile_layout.addWidget(self.back_from_profile_btn)

self.profile_layout.addStretch()
self.profile_page.setLayout(self.profile_layout)
self.stacked_widget.addWidget(self.profile_page)

self.layout.addWidget(self.stacked_widget)
self.setLayout(self.layout)

self.signals = WorkerSignals()
self.signals.result.connect(self.populate_movies)
self.signals.error.connect(self.show_error)

self.desc_signals = WorkerSignals()
self.desc_signals.desc_result.connect(self.show_description)
self.desc_signals.error.connect(self.show_desc_error)

self.image_manager = QNetworkAccessManager()
self.image_manager.finished.connect(self.handle_poster_response)

self.current_movie = ""
self.descriptions = self.load_descriptions()
self.favourites = self.load_favourites()
self.user_stats = self.load_user_stats()

self.update_emoji(50)
```

```
        self.update_year_range()

    def update_emoji(self, value):
        emoji = "□" if value < 20 else "□" if value < 40 else "□" if value < 60 else "□"
if value < 80 else "□"
        self.emoji_label.setText(emoji)
        self.genre_combo.clear()
        self.genre_combo.addItems(self.mood_genres.get(emoji, []))

    def update_year_range(self, value=None):
        start_year = self.year_start_slider.value()
        end_year = self.year_end_slider.value()

        if start_year > end_year:
            if self.sender() == self.year_start_slider:
                self.year_end_slider.setValue(start_year)
            elif self.sender() == self.year_end_slider:
                self.year_start_slider.setValue(end_year)

        self.year_start_label.setText(str(self.year_start_slider.value()))
        self.year_end_label.setText(str(self.year_end_slider.value()))

    def get_recommendations(self):
        genre = self.genre_combo.currentText()
        mood = "bad mood" if self.mood_slider.value() < 50 else "good mood"
        start_year = self.year_start_slider.value()
        end_year = self.year_end_slider.value()
        year_range = f"{start_year}-{end_year}"
```

```
        prompt = f"Give me a list of 10 {genre} movie titles in English for someone in a {mood}, from the years {year_range}. Just the titles, no descriptions, no numbers, no bullet points, one movie per line."
        self.movie_list.clear()
        self.progress.setValue(50)
        self.submit_btn.setEnabled(False)
        self.to_profile_btn.setEnabled(False)
        threading.Thread(target=self.fetch_movies, args=(prompt,)).start()

    def fetch_movies(self, prompt):
        try:
            response = model.generate_content(prompt)
            movies = [m.strip(" -•*") for m in response.text.strip().split("\n") if m.strip()]
            self.signals.result.emit(movies)
        except Exception as e:
            self.signals.error.emit(str(e))
        finally:
            self.submit_btn.setEnabled(True)
            self.to_profile_btn.setEnabled(True)

    def populate_movies(self, movies):
        self.movie_list.addItems(movies)
        self.progress.setValue(100)
        self.stacked_widget.setCurrentWidget(self.results_page)

    def show_error(self, msg):
        QMessageBox.critical(self, "Error", f"An error occurred: {msg}")
        self.progress.setValue(0)
        self.submit_btn.setEnabled(True)
        self.to_profile_btn.setEnabled(True)
```

```
    def get_description(self, movie_title, from_favourites=False):
        self.current_movie = movie_title
        self.last_page_before_description = self.favourites_page if from_favourites else self.results_page

        if not from_favourites:
            current_genre = self.genre_combo.currentText()
            start_year = self.year_start_slider.value()
            end_year = self.year_end_slider.value()
            current_year_range = f"{start_year}-{end_year}"
            self.update_user_stats(current_genre, current_year_range)

        self.progress.setValue(50)

        self.poster_label.clear()
        self.poster_label.setText("Loading details...")
        self.poster_label.setStyleSheet("font-style: italic; color: gray;")
        self.trailer_view.setUrl(QUrl("about:blank"))
        self.trailer_view.hide()

        if movie_title in self.descriptions:
            self.description.setText(self.descriptions[movie_title])
            threading.Thread(target=self.fetch_movie_details_only_media, args=(movie_title,)).start()
        else:
            self.description.setText(f"Loading description for {movie_title}...")
            threading.Thread(target=self.fetch_movie_details, args=(movie_title,)).start()

        self.stacked_widget.setCurrentWidget(self.description_page)
```

```
    def fetch_movie_details(self, movie_title):
        try:
            gemini_prompt = f"Briefly describe the movie '{movie_title}' in English, maximum 3-4 sentences."
            gemini_response = model.generate_content(gemini_prompt)
            description = gemini_response.text.strip()
            self.descriptions[movie_title] = description
            self.save_descriptions()

            search_url = f"{TMDB_BASE_URL}/search/movie?api_key={TMDB_API_KEY}&query={movie_title}&language=en-US"
            response = requests.get(search_url).json()

            movie_id = None
            poster_path = None
            if response and response.get('results'):
                found_exact = False
                for result in response['results']:
                    if result.get('title', '').lower() == movie_title.lower():
                        movie_id = result.get('id')
                        poster_path = result.get('poster_path')
                        found_exact = True
                        break
                if not found_exact:
                    first_result = response['results'][0]
                    movie_id = first_result.get('id')
                    poster_path = first_result.get('poster_path')
```

```
            trailer_url = None
            if movie_id:
                videos_url = f"{TMDB_BASE_URL}/movie/{movie_id}/videos?api_key={TMDB_API_KEY}&language=en-US"
                videos_response = requests.get(videos_url).json()
                if videos_response and videos_response.get('results'):
                    for video in videos_response['results']:
                        if video.get('site') == 'YouTube' and video.get('type') == 'Trailer':
                            trailer_url = f"{TMDB_YOUTUBE_BASE_URL}{video.get('key')}"
                            break

            self.desc_signals.desc_result.emit([description, poster_path, trailer_url])

        except Exception as e:
            self.desc_signals.error.emit(f"Error fetching movie details: {str(e)}")

    def fetch_movie_details_only_media(self, movie_title):
        try:
            description = self.descriptions.get(movie_title, "Description not available.")

            search_url = f"{TMDB_BASE_URL}/search/movie?api_key={TMDB_API_KEY}&query={movie_title}&language=en-US"
            response = requests.get(search_url).json()

            movie_id = None
            poster_path = None
            if response and response.get('results'):
```

```
            found_exact = False
            for result in response['results']:
                if result.get('title', '').lower() == movie_title.lower():
                    movie_id = result.get('id')
                    poster_path = result.get('poster_path')
                    found_exact = True
                    break
            if not found_exact:
                first_result = response['results'][0]
                movie_id = first_result.get('id')
                poster_path = first_result.get('poster_path')

        trailer_url = None
        if movie_id:
            videos_url = f"{TMDB_BASE_URL}/movie/{movie_id}/videos?api_key={TMDB_API_KEY}&language=en-US"
            videos_response = requests.get(videos_url).json()
            if videos_response and videos_response.get('results'):
                for video in videos_response['results']:
                    if video.get('site') == 'YouTube' and video.get('type') == 'Trailer':
                        trailer_url = f"{TMDB_YOUTUBE_BASE_URL}{video.get('key')}"
                        break

        self.desc_signals.desc_result.emit([description, poster_path, trailer_url])

    except Exception as e:
        self.desc_signals.error.emit(f"Error fetching media details: {str(e)}")
```

```
def show_description(self, data):
    description = data[0]
    poster_path = data[1]
    trailer_url = data[2]

    self.description.setText(description)

    if poster_path:
        poster_url = f"{TMDB_IMAGE_BASE_URL}{poster_path}"
        request = QNetworkRequest(QUrl(poster_url))
        self.image_manager.get(request)
        self.poster_label.setText("Loading poster...")
        self.poster_label.setStyleSheet("font-style: italic; color: gray;")
    else:
        self.poster_label.clear()
        self.poster_label.setText("Poster not available.")
        self.poster_label.setStyleSheet("font-style: italic; color: gray;")

    if trailer_url:
        self.trailer_view.show()
        self.trailer_view.setUrl(QUrl(trailer_url))
    else:
        self.trailer_view.hide()
        self.trailer_view.setUrl(QUrl("about:blank"))

    self.progress.setValue(100)

def handle_poster_response(self, reply: QNetworkReply):
    if reply.error() == QNetworkReply.NetworkError.NoError:
        image_data = reply.readAll()
```

```
            image = QImage()
            if image.loadFromData(image_data):
                pixmap = QPixmap.fromImage(image)
                self.poster_label.setPixmap(pixmap.scaled(self.poster_label.size(),
                                                Qt.AspectRatioMode.KeepAspectRatio,

Qt.TransformationMode.SmoothTransformation))
                self.poster_label.setText("")
            else:
                self.poster_label.setText("Failed to load poster image.")
                self.poster_label.setStyleSheet("font-style: italic; color: red;")
        else:
            self.poster_label.setText(f"Error fetching poster: {reply.errorString()}")
            self.poster_label.setStyleSheet("font-style: italic; color: red;")
        reply.deleteLater()

    def show_desc_error(self, message):
        self.description.setText(f"Error: {message}")
        self.poster_label.clear()
        self.poster_label.setText("Error loading details.")
        self.poster_label.setStyleSheet("font-style: italic; color: red;")
        self.trailer_view.setUrl(QUrl("about:blank"))
        self.trailer_view.hide()
        self.progress.setValue(100)

    def add_to_favourites(self):
        movie = self.current_movie
        if movie and movie not in self.favourites:
            self.favourites.append(movie)
            self.save_favourites()
```

```
            QMessageBox.information(self, "Favourites", f"Movie '{movie}' added to favourites successfully!")
        elif movie in self.favourites:
            QMessageBox.information(self, "Favourites", f"Movie '{movie}' is already in your favourites.")
        else:
            QMessageBox.warning(self, "Favourites", "No movie selected to add to favourites.")

    def show_favourites(self):
        self.fav_list.clear()
        if not self.favourites:
            self.fav_list.addItem("No favourite movies yet.")
            try:
                self.fav_list.itemClicked.disconnect()
            except TypeError:
                pass
            self.stacked_widget.setCurrentWidget(self.favourites_page)
            return

        try:
            self.fav_list.itemClicked.disconnect()
        except TypeError:
            pass

        for movie in self.favourites:
            item = QListWidgetItem(self.fav_list)
            item_widget = self.create_widget_with_button(movie)
            item.setSizeHint(item_widget.sizeHint())
            self.fav_list.addItem(item)
```

```
            self.fav_list.setItemWidget(item, item_widget)

        self.fav_list.itemClicked.connect(lambda item:
self.get_description(self.fav_list.itemWidget(item).movie_name,
from_favourites=True))

        self.stacked_widget.setCurrentWidget(self.favourites_page)

    def create_widget_with_button(self, movie):
        widget = QWidget()
        layout = QHBoxLayout(widget)

        widget.movie_name = movie

        label = QLabel(movie)
        label.setWordWrap(True)
        layout.addWidget(label)
        layout.addStretch()
        btn = QPushButton("□")
        btn.setFixedSize(25, 25)
        btn.clicked.connect(lambda: self.remove_favourite(movie))
        layout.addWidget(btn)
        layout.setContentsMargins(5, 0, 5, 0)
        return widget

    def remove_favourite(self, movie):
        reply = QMessageBox.question(self, 'Remove Favourite',
                                     f"Are you sure you want to remove '{movie}' from your
favourites?",
```

```
                QMessageBox.StandardButton.Yes |
QMessageBox.StandardButton.No,
                QMessageBox.StandardButton.No)
        if reply == QMessageBox.StandardButton.Yes:
            if movie in self.favourites:
                self.favourites.remove(movie)
                self.save_favourites()
                self.show_favourites()
                QMessageBox.information(self, "Favourites", f"Movie '{movie}' removed
from favourites.")

    def load_favourites(self):
        if os.path.exists(FAVOURITES_FILE):
            with open(FAVOURITES_FILE, "r", encoding="utf-8") as f:
                return [line.strip() for line in f if line.strip()]
        return []

    def save_favourites(self):
        with open(FAVOURITES_FILE, "w", encoding="utf-8") as f:
            f.writelines(movie + "\n" for movie in self.favourites)

    def load_descriptions(self):
        if os.path.exists(DESCRIPTIONS_FILE):
            with open(DESCRIPTIONS_FILE, "r", encoding="utf-8") as f:
                lines = f.readlines()
                return dict(line.strip().split("||", 1) for line in lines if "||" in line.strip())
        return {}

    def save_descriptions(self):
        with open(DESCRIPTIONS_FILE, "w", encoding="utf-8") as f:
```

```
        for movie, desc in self.descriptions.items():
            f.write(f"{movie}||{desc}\n")

def load_user_stats(self):
    if os.path.exists(STATS_FILE):
        try:
            with open(STATS_FILE, "r", encoding="utf-8") as f:
                return json.load(f)
        except json.JSONDecodeError:
            return {"genres": [], "years": []}
    return {"genres": [], "years": []}

def save_user_stats(self):
    with open(STATS_FILE, "w", encoding="utf-8") as f:
        json.dump(self.user_stats, f, ensure_ascii=False, indent=4)

def update_user_stats(self, genre, year_range):
    if genre and genre != "Choose genre:":
        self.user_stats["genres"].append(genre)
    if year_range:
        self.user_stats["years"].append(year_range)
    self.save_user_stats()

def show_profile_page(self):
    self.update_profile_stats()
    self.stacked_widget.setCurrentWidget(self.profile_page)

def update_profile_stats(self):
    if self.user_stats["genres"]:
        genre_counts = Counter(self.user_stats["genres"])
```

```
            top_genres = [genre for genre, count in genre_counts.most_common(3)]
            self.fav_genres_label.setText(", ".join(top_genres))
        else:
            self.fav_genres_label.setText("No data yet.")

        if self.user_stats["years"]:
            year_counts = Counter(self.user_stats["years"])
            top_years = [year for year, count in year_counts.most_common(3)]
            self.fav_years_label.setText(", ".join(top_years))
        else:
            self.fav_years_label.setText("No data yet.")

    def show_selection_page(self):
        self.progress.setValue(0)
        self.stacked_widget.setCurrentWidget(self.selection_page)

    def show_results_page(self):
        self.progress.setValue(0)
        self.stacked_widget.setCurrentWidget(self.results_page)

    def show_favourites_page(self):
        self.progress.setValue(0)
        self.stacked_widget.setCurrentWidget(self.favourites_page)

    def return_from_description(self):
        self.progress.setValue(0)
        if self.last_page_before_description:
            self.stacked_widget.setCurrentWidget(self.last_page_before_description)
        else:
            self.stacked_widget.setCurrentWidget(self.results_page)
```

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainForm()
    window.show()
    sys.exit(app.exec())
```

Додаток Г
(обов'язковий)

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Розробка додатку для підбору фільмів на основі категорій із використанням штучного інтелекту

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

Підрозділ кафедра АІІТ, ФІІТА, ІІСТ-21б
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism 0.06%

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:

Олег БІСІКАЛО, зав. кафедри АІІТ _____ (підпис)
(прізвище, ініціали, посада)

Любов ЯНЧУК, секретар кафедри АІІТ _____ (підпис)
(прізвище, ініціали, посада)

Особа, відповідальна за перевірку _____ Костянтин ОВЧИННИКОВ
(підпис) (прізвище, ініціали)

З висновком експертної комісії ознайомлений(-на)

Керівник _____ Гармаш В.В, доцент
(підпис) (прізвище, ініціали, посада)

Здобувач _____ Медведєв М. К
(підпис) (прізвище, ініціали)