МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В.Н. Каразіна

Факультет: **ННІ Каразінський банківський інститут**
Кафедра: **Інформаційних технологій та математичного моделювання**
Спеціальність: **122 Комп'ютерні науки**
Освітня програма: **Комп'ютерні науки**

Група: АК-21М **денна форма навчання**

КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

на тему:

Розробка та дослідження ефективності бота-асистента для автоматизації роботи QA спеціалістів

ЗА НАКАЗОМ № 4601-5_3262 ВІД 15 вересня 2025 РОКУ

Здобувача вищої освіти **Голубєва Павла Сергійовича**

Робота допущена до захисту в ЕК
протокол кафедри ІТММ №__ від _____2025 р.

Завідувач кафедри ІТММ
к.п.н., доцент
____________________ **Н.І. Стяглик**

Науковий керівник
к.т.н., доцент
____________________ **О.В. Соболєв**

м. Харків 2025 р.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В. Н. Каразіна

Факультет навчально-науковий інститут "Каразінський банківський інститут"
Кафедра інформаційних технологій та математичного моделювання
Рівень вищої освіти другий (магістерський)
Спеціальність 122 Комп'ютерні науки
Освітня програма Комп'ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри

____________ **Н. І. Стяглик**
Підпис ініціали, прізвище

"15" вересня 2025 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ (ПРОЄКТ)

Голубєву Павлу Сергійовичу
(прізвище, ім'я, по батькові студента)

1. Тема роботи: Розробка та дослідження ефективності бота-асистента для автоматизації роботи QA спеціалістів.

керівник роботи Соболєв Олександр Вікторович, к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від "15" вересня 2025року № 4601-5_3262

2. Строк подання студентом роботи 24 листопада 2025 року

3. Перелік питань, які потрібно розробити:

У розділі 1: Дослідити сутність ручного та автоматизованого тестування, а також їх видів.

У розділі 2: Проаналізувати сучасний стан підходів та рішень автоматизації роботи QA-спеціалістів.

У розділі 3: Описати принцип роботи розробленого бота-асистента та його компонентів.

У розділі 4: Проаналізувати ефективність розробленого рішення.

4.План роботи

№ з/п	Назви етапів роботи
1	Вибір здобувачем теми кваліфікаційної магістерської роботи
2	Затвердження плану і завдання кваліфікаційної магістерської роботи
3	Здача кваліфікаційної магістерської роботи керівнику
4	Підпис кваліфікаційної магістерської роботи керівника
5	Підпис кваліфікаційної магістерської роботи у нормоконтролера
6	Допуск завідувачем кафедри до захисту кваліфікаційної магістерської роботи
7	Захист кваліфікаційної магістерської роботи

5.Дата видачі завдання 15 вересня 2025 року

Студент _________________ Голубєв П.С.
підпис ініціали, прізвище

Керівник роботи _________________ Соболєв О.В.
підпис ініціали, прізвище

РЕФЕРАТ
НА КВАЛІФІКАЦІЙНУ МАГІСТЕРСЬКУ РОБОТУ
«РОЗРОБКА ТА ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ БОТА-АСИСТЕНТА ДЛЯ АВТОМАТИЗАЦІЇ РОБОТИ QA СПЕЦІАЛІСТІВ»

Голубєва Павла Сергійовича

Кваліфікаційна магістерська робота містить: 69 сторінки, 5 таблиць, 50 рисунків, список літератури із 35 найменувань.

Об’єкт дослідження: тестування програмних продуктів.

Предмет дослідження: сучасні методи та рішення автоматизації роботи QA спеціалістів.

Мета кваліфікаційної магістерської роботи: полягає у розробці бота-асистента для автоматизації деяких задач роботи тестувальників та аналізі ефективності його впливу.

Завдання кваліфікаційної магістерської роботи:

1. Дослідити сутність ручного та автоматизованого тестування, а також їх видів.
2. Проаналізувати сучасний стан і інструменти автоматизації тестування та вплив впровадження штучного інтелекту до тестування програмного забезпечення.
3. Розробити, описати принцип роботи та показати на практиці бота-асистента для спеціалістів забезпечення якості програмних продуктів.
4. Проаналізувати ефективність розробленого рішення.

Актуальність дослідження: зумовлена стрімким розвитком сфери розробки і тестування програмного забезпечення з пришвидшенням бізнес-процесів, пов’язаних з цим, що змушує шукати нові способи підвищення їх ефективності.

За результатами дослідження: розроблено та перевірено на практиці бота-помічника для автоматизації роботи QA спеціалістів.

Практичне значення роботи: полягає у розробці бота-помічника, якого можна розглядати як платформу для автоматизації рутинних завдань, не тільки QA-спеціалістів, а й усієї команди розробників програмного забезпечення.

Одержані результати можуть бути використані спеціалістами забезпечення якості для підвищення ефективності їх роботи.

КЛЮЧОВІ СЛОВА: QA, QUALITY ASSURANCE, QA-ASSISTANT, AI-ASSISTANT, BOT-ASSISTANT, PYTHON, РОЗРОБКА ВЛАСНОГО AI-ПОМІЧНИКА.

ABSTRACT

FOR QUALIFICATION MASTER'S THESIS

"DEVELOPMENT AND RESEARCH OF THE EFFICIENCY OF A BOT-ASSISTANT FOR AUTOMATION OF THE WORK OF QA SPECIALISTS"

Holubiev Pavlo

The qualification master's thesis contains: 69 pages, 5 tables, 50 figures, a list of references from 35 titles.

Object of research: testing of software products.

Subject of research: modern methods and solutions for automating the work of QA specialists.

The purpose of the qualification master's thesis: is to develop a bot-assistant to automate some tasks of testers' work and analyze the effectiveness of its impact.

Tasks of the qualification master's thesis:

1. To investigate the essence of manual and automated testing, as well as their types.
2. Analyze the current state and tools of test automation and the impact of implementing artificial intelligence in software testing.
3. Develop, describe the principle of operation and demonstrate in practice an assistant bot for software quality assurance specialists.
4. Analyze the effectiveness of the developed solution.

The relevance of the study: is due to the rapid development of the field of software development and testing with the acceleration of business processes related to this, which forces us to look for new ways to increase their efficiency.

According to the results of the study: an assistant bot was developed and tested in practice to automate the work of QA specialists.

The practical significance of the work: consists in developing an assistant bot, which can be considered as a platform for automating routine tasks, not only of QA specialists, but also of the entire team of software developers.

The results obtained can be used by quality assurance specialists to improve the efficiency of their work.

KEYWORDS: QA, QUALITY ASSURANCE, QA-ASSISTANT, AI-ASSISTANT, PYTHON, DEVELOPMENT OF YOUR OWN AI-ASSISTANT.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧОК, СИМВОЛІВ І ТЕРМІНІВ 9
ВСТУП 10
РОЗДІЛ 1. РУЧНЕ ТА АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 12
1.1. Ручне тестування, його сутність, переваги та недоліки. 12
1.2. Автоматизація процесів тестування 14
1.3. Види тестування програмного забезпечення 16
РОЗДІЛ 2. АНАЛІЗ СУЧАСНИХ ПІДХОДІВ ТА РІШЕНЬ АВТОМАТИЗАЦІЇ РОБОТИ QA-СПЕЦІАЛІСТІВ 21
2.1. Сучасний стан автоматизації в тестуванні програмних продуктів 21
2.2. Аналіз сучасних інструментів автоматизованого тестування 25
2.3. Впровадження інтелектуальних систем у роботу QA-спеціалістів 29
РОЗДІЛ 3. БОТ-АСИСТЕНТ ДЛЯ АВТОМАТИЗАЦІЇ РОБОТИ QA-СПЕЦІАЛІСТІВ 34
3.1. Ідея Проекту 34
3.2. Аналіз готових рішень 35
3.3. Опис технологій рішення, яке було розроблено 36
3.4. Реєстрація нового Telegram бота через BotFather 39
3.5. Підготовка до початку роботи 44
3.6. Опис роботи логіки компонентів розробленого рішення 47
3.7. Технічна реалізація компонентів програми 52
РОЗДІЛ 4. АНАЛІЗ ЕФЕКТИВНОСТІ РОЗРОБЛЕНОГО РІШЕННЯ 60
4.1. Аналіз ефективності роботи 60
4.2. Подальші можливості модифікації 64
ВИСНОВКИ 65
ПЕРЕЛІК ПОСИЛАНЬ 66

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧОК, СИМВОЛІВ І ТЕРМІНІВ

ПЗ – Програмне забезпечення

QA – Quality Assurance

ШІ – Штучний Інтелект

CI/CD – Continuous Integration / Continuous Deployment

NLP – Natural Language Processing

TTM – Time To Market

DLP – Data Leak Prevention

YAML – Ain't Markup Language

HTTP – HyperText Transfer Protocol

API – Application Programming Interface

ВСТУП

В сучасній IT-індустрії постійно зростає кількість та складність програмних продуктів різноманітної направленості. Також зростають і вимоги до них. Користувачі вимагають більшої якості та стабільності, а розробникам, для більшої конкурентоспроможності, необхідно додавати все більше функцій та модулів, зберігаючи при цьому необхідний рівень якості продукту. Для полегшення цієї задачі у команді розробників ПЗ є люди, що забезпечують якість програмного продукту (Quality Assurance, QA). Вони відповідають за стабільність, надійність та узгодженість ПЗ встановленим вимогам. Але з розвитком технологій та збільшенням тестових сценаріїв спеціалісти QA все більша часу приділяють одноманітним простим задачам замість більш важливих, що не може не відобразитися на ефективності роботи тестувальників.

Автоматизвція процесів тестування не нова річ. На сьогодні існує широкий спектр інструментів, що дають змогу автоматизувати виконання тестів, керування дефектами, створення звітів та інтеграцію з CI/CD (безперервна інтеграція та доставка) процесами. Проте навіть попри наявність потужних інструментів, значна частина робочих процесів QA спеціалістів залишається ручною: формування тестових сценаріїв, ведення документації, аналітика результатів, пошук помилок у логах, оцінка пріоритетів дефектів тощо. Такі завдання є рутинними, часто повторюваними та забирають значну частину робочого часу, який можна було б спрямувати на стратегічні аспекти забезпечення якості. У зв'язку з цим виникає потреба у створенні нових підходів до автоматизації роботи QA-спеціалістів

З сучасними тенденціями розвитку штучного інтелекту (ШІ), машинного навчання та інтелектуальних систем відкриваються нові можливості для автоматизації. Один з прикладів цього це створення ботів-асистентів, які не лише виконують окремі автоматизовані дії, але й підтримують спеціалістів у прийнятті рішень, аналізі результатів тестів,

генерації тестових сценаріїв та управлінні тестовим процесом.

Об’єктом дослідження є тетсування програмних продуктів.

Предметом дослідження є сучасні методи та рішення автоматизіції роботи QA спеціалістів.

Мета кваліфікаційної магістерської роботи полягає у розробці бота-асистента для автоматизації деяких задач роботи тестувальників та аналізі ефективності його впливу.

Для досягнення мети були вирішені такі завдання:

1. Дослідити сутність ручного та автоматизованого тестування, а також їх видів.

2. Проаналізувати сучасний стан і інструменти автоматизації тестування та вплив впровадження штучного інтелекту до тестування програмного забезпечення.

3. Розробити, описати принцип роботи та показати на практиці бота-асистента для спеціалістів забезпечення якості програмних продуктів.

4. Проаналізувати ефективність розробленого рішення.

Наукова новизна полягає у створені бота-помічника на базі штучного інтелекту та залученню його до реальних задач спеціалістів забезпечення якості програмного забезпечення для автоматизації їх виконання.

Написання роботи було виконано з дотриманням методичних рекомендацій та не менш встановленого обсягу. В квадратних дужках наприкінці вказано джерело.

РОЗДІЛ 1
РУЧНЕ ТА АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1. Ручне тестування, його сутність, переваги та недоліки

Історія тестування починається разом з формуванням програмування як окремої галузі комп'ютерних наук. Але поняття «тестування» у сучасному розумінні тоді ще не існувало: процес перевірки програм зводився до простого пошуку помилок у коді, який програмісти самі писали, виконували та виправляли.

Приблизно у 1970-х роках, завдяки розвитку великих інформаційних систем, з'явилась потреба у систематичному підході до перевірки якості програм. Тоді ж було сформовано перші методології, які розглядали тестування не лише як пошук помилок, а як процес підтвердження відповідності програми вимогам. У 1979 році Гленфорд Майєрс у своїй книзі "The Art of Software Testing"[1] вперше чітко сформулював принципи тестування як самостійної дисципліни, відокремленої від процесу розробки.

Протягом 1980–1990-х років розвиток тестування став практичним напрямком. З появою персональних комп'ютерів і складних графічних інтерфейсів тестування почали розглядати як окрему професійну діяльність. Також в цей період було сформовано класифікацію тестування на ручне та автоматизоване. Однак основна частина перевірок усе ще виконувалась вручну, оскільки програмні продукти часто змінювалися, а автоматизація вимагала великих затрат часу та ресурсів.

На початку 2000-х років почали набувати популярність гнучкі методології розробки програмних продуктів (такі як Agile чи Scrum), а з ними роль тестування значно зросла. Замість ізольованого етапу наприкінці розробки тестування стало безперервним процесом, який супроводжує кожну частину створення продукту. Приблизно тоді виник термін QA – забезпечення

якості, який охоплює не лише пошук дефектів, а й превентивні дії для їх уникнення.

На сьогодні тестування програмного забезпечення є невід'ємною частиною життєвого циклу розробки програмних продуктів. Його головною метою є виявлення недоліків, тобто різниць між очікуваним результатом, який відповідає вимогам замовника, та реальною поведінкою компонентів або всієї системи при виконанні конкретних дій. Дві основні форми тестування – це ручне та автоматизоване. Незважаючи на значний розвиток інструментів автоматизації, ручне тестування залишається важливою частиною контролю якості програмного забезпечення.

Ручне тестування (або manual testing англ.) передбачає перевірку якості програмного продукту тестувальником без використання автоматичних інструментів та засобів. Людина вручну виконує поставлені тести та перевіряє роботу програного продукту, оцінює поведінку системи та компонентів і аналізує отримані результати, порівнюючи їх з очікуваними.

Перевагами такого підходу є гнучкість та адаптивність. Тестувальник, користуючись своїм досвідом, може помітити аномальну поведінку системи чи логічні невідповідності, що автоматизовані засоби, націлені на конкретні результати, пропустять. Ще людина може швидко адаптувати тестування до змін у програмному продукті, або розглянути та оцінити ПЗ з точки зору користувача. Також для впровадження такого тестування не потрібні спеціалізовані інструменти та навички роботи з ними, достатньо тестувальника, програмного продукту та тестової документації до нього. Один важливий плюс такого тестування, що є наслідком легкого впровадження, – його можна застосовувати на дуже ранніх етапах розробки програного продукту або для тестування експериментальних чи нестандартних функцій.

Недоліками ручного тестування є велика трудомісткість, людський фактор та складність масштабування. Виконання тестів вручну вимагає значних часових ресурсів, особливо коли йдеться про повторювані перевірки після кожної зміни у програмі, а якщо проект великий та довготривалий сюди

додається ще й висока вартість такого тестування, тому що стає необхідність залучення все більшої кількості тестувальників. Людський фактор також відіграє значну роль, він може як допомогти креативним підходом та досвідом людини так і нашкодити її втомою неуважністю чи недостатньою кваліфікацією, що може привезти до пропуску дефектів різної важливості. Складність масштабування зумовлена багатою кількістю тестових сценаріїв у великих проектах, де вони можуть обчислюватися тисячами, що робить неможливим їх виконання людиною за прийнятний термін. [2, 3]

1.2. Автоматизація процесів тестування

Як видно з попереднього підрозділу, одним ручним тестуванням не подолати всі виклики забезпечення якості ПЗ. Вирішення цих недоліків стало основним стимулом для розвитку автоматизованого тестування (або automated testing англ.). Ця форма тестування передбачає створення тестових сценаріїв, по яким спеціальні програмні засоби виконують заздалегідь визначені дії без безпосередньої участі людини. Основна ідея автоматизації тестування полягає у зменшенні витрат часу та ресурсів на виконання повторюваних тестів з підвищенням їх точності та стабільності.

На відміну від ручного тестування, де всі кроки виконує тестувальник, в автоматизованому людина лише пише план дій, що реалізує логіку роботи з системою, по якому спеціальні утиліти вже самі роблять необхідні перевірки. Вони можуть автоматично запускатися, аналізувати результати, створювати звіти та навіть реагувати на виявлені помилки.

Основною перевагою такої автоматизації є можливість виконувати дуже велику кількість схожих тестів, зберігаючи стабільність і повторюваність, адже на відміну від людини, вона не залежить від зовнішніх факторів. Також можна виділити швидкість та масштабованість виконання, тому що тести, які людина виконувала б годинами, програма виконає за лічені хвилини із збереженням однакового рівня якості. Ще автоматичне тестування набагато

вигідніше за ручне у довгостроковому використанні, тому що, хоч початкові витрати на створення тестів можуть бути і значними, потім їх можна використовувати скільки завгодно для подібних перевірок. Ці переваги роблять автоматизацію незамінною при безперервному тестуванні, коли необхідно після кожної ітерації продукту повторно виконувати велику кількість тестів. Але це все доцільно лише при гарному плануванні проекту та за наявності досвідченого персоналу. Розробка скриптів для автоматизованих тестів вимагає значних затрат ресурсів та часу, що може затримати початок тестування. Також слід пам’ятати, що з кожним оновленням функціоналу або змінами в інтерфейсі автоматизовані тести необхідно постійно оновлювати та доробляти для їх правильної роботи.

Незважаючи на переваги, автоматизацію доцільно застосовувати не до всіх випадків тестування. Наприклад задачі, що вимагають оцінки зручності, інтуїтивності чи логічності та всього пов’язаного з поведінкою, діями або емоціями користувача, загалом тестування з його точки зору складно описати тестовими сценаріями. Ще дослідницьке тестування неможливо автоматизувати, тому що при ньому не має жорстко прописаних сценаріїв і тестувальник, опираючись на свій досвід та розуміння, одночасно вивчає програмний продукт, проектує до нього тести та виконує їх. Також не слід автоматизувати тестування нестабільних програмних продуктів. Якщо ПЗ знаходиться на стадії активної розробки та постійно змінюється, автоматизація його тестування буде дуже недоречною. І на кінець, сценарії тестування, які будуть виконані лише обмежену невелику кількість разів не має сенсу автоматизувати – написання скрипту для цього займе майже стільки, а може й більше, часу та ресурсів скільки його ручне виконання. [2-4]

Загалом, поширеною практикою сьогодні є поєднання ручного та автоматичного тестування, в залежності від етапу розробки або типу тестування, користуючись їхніми сильними сторонами задля покращення якості програмних продуктів.

1.3. Види тестування програмного забезпечення

Хоча тестування загально можна поділити та ручне та автоматизоване, така класифікація ускладнює вибір підходу до конкретної задачі тестування, тому що вона не відображає мету тестування. Класифікувати можна по різним критеріям: за рівнем, методами, направленням чи об'єктом тестування, варіантів багато. Найбільш поширеним сьогодні є розділення на функціональне та нефункціональне тестування, також до них можна додати структурне та тестування змін. Така класифікація є універсальною та забезпечує повноту перевірки програмних продуктів.

Функціональне тестування спрямоване на перевірку того, наскільки програмний продукт відповідає визначеним вимогам і виконує свої основні функції. Метою цього виду тестування є підтвердження правильності реалізації логіки роботи системи та її можливостей. Найбільш популярні підвиди функціонального тестування згруповані у таблицю 1.1. [5-7]

Таблиця 1.1

Популярні підвиди функціонального тестування

Назва	Опис
Димове тестування	поверхнева перевірка базової працездатності системи після збірки або розгортання нового релізу, щоб переконатися що основний функціонал програми працює, і система загалом готова до подальших, глибших тестів.
Модульне тестування	перевірка окремих функцій або модулів програми в ізоляції від інших компонентів.
Інтеграційне тестування	тестування взаємодії між окремими модулями системи без зовнішнього впливу, що дозволяє виявити помилки у процесі їх взаємозв'язку.
Системне тестування	перевірка всієї системи в цілому, щоб визначити, чи відповідає вона функціональним вимогам замовника.
Приймальне тестування	здійснюється з метою підтвердження готовності програмного забезпечення до передачі кінцевому користувачу або впровадження у виробниче середовище.

Нефункціональне тестування спрямоване на оцінку характеристик якості програмного забезпечення, які не пов’язані безпосередньо з його функціональністю. Воно дозволяє визначити, наскільки ефективно система виконує свої функції за певних умов експлуатації. Тобто не ставиться питання «Що робить система?», як у попередньому виді, а питається «Як само вона це робить?». Цей вид тестування значно ширший за функціональне – в нього входить перевірка відповідності всім вимогам замовника, які не стосуються виконання функцій програмним продуктом. Підвиди нефункціонального тестування, якими користуються найчастіше представлено у таблиці 1.2. [5-7]

Таблиця 1.2

Популярні підвиди нефункціонального тестування

Назва	Опис
Тестування навантаження	починаючи з мінімального навантаження, перевіряє роботу системи при поступовому збільшенні кількості користувачів або обсягу даних до максимуму.
Стрес-тестування	перевіряє, як система реагує на граничні або позаштатні навантаження.
Тестування стабільності	перевіряється стабільність системи при довготривалому використанні з нормальним навантаженням.
Тестування безпеки	виявляє потенційні вразливості та перевіряє захищеність даних. В залежності від типу програмного продукту, що розроблюється, може бути як функціональним так і нефункціональним видом тестування.
Тестування юзабіліті	оцінює зручність та інтуїтивність інтерфейсу користувача.
Тестування інтерфейсу	перевіряє відповідність інтерфейсу вимогам замовника (наприклад колір якогось елементу, його форма, зовнішній вигляд та подібне).
Тестування сумісності	перевіряє роботу системи на різних платформах, пристроях і браузерах.
Тестування документації	перевіряє вимоги замовника (їх адекватність, актуальність, зрозумілість і подібне), специфікації та супутню документацію (наприклад правильність посібника користувача).

Структурне тестування, або тестування білої скриньки, можна розглядати як частину функціонального, але більш вужчу його версію, тому що в ньому не перевіряється зовнішня поведінка системи. Тестування базується на аналізі внутрішньої структури системи чи коду програми, де оцінюється логіка реалізації, покриття коду та ефективність алгоритмів. Цей тип тестування зазвичай виконується розробниками або автоматизованими засобами, що дозволяє підвищити якість програмного коду та зменшити кількість помилок ще на ранніх етапах розробки.

Тестування змін – це вид тестування, який проводиться після внесення змін у програмний код або виправлення помилок. Його мета полягає у підтвердженні того, що внесені зміни не призвели до появи нових дефектів і не порушили роботу вже перевірених функцій. Виділяють два основні підвиди: Повторне та Регресійне тестування.

Повторне тестування проводиться після виникнення помилки та її виправлення для перевірки, чи була вона дійсно виправлена. Виконується після виходу нового білда продукту, але з такими ж даними та оточенням що і перше тестування, яке показало помилку. Не може бути автоматизовано.

Регресійне тестування, при якому, з виходом нової версії продукту та зміни деякого функціоналу або доданням нового, повторно проводяться усі попередньо вдалі тести для впевненості, що нові зміни не вплинули негативно на існуючий функціонал. Цілком піддається автоматизації.

Кожен із розглянутих видів тестування має своє призначення та застосовується на певному етапі життєвого циклу програмного забезпечення. Їх комплексне використання забезпечує повноцінну перевірку якості ПЗ та зменшує ризики помилок у фінальній версії продукту. [5-7]

У таблиці 1.3 наведено декілька прикладів сфер інформаційних технологій та форм тестування програмних засобів, які в них використовуються.

Таблиця 1.3

Сфери ІТ та форми тестування ПЗ в них

Сфера застосування	Форма тестування	Пояснення
Веб-додатки	Автоматизоване + ручне	Для перевірки функціональності, стабільності та кросбраузерності застосовують автоматизацію, а для UX-оцінки – ручне тестування
Мобільні додатки	Автоматизоване + ручне	Тестуються поведінка на різних пристроях і ОС; автоматизація використовується для регресії, ручне — для юзабіліті.
Ігрова індустрія	Переважно ручне	Через високу креативність і змінність контенту ручне тестування ефективніше для перевірки ігрових сценаріїв та балансу.
Банківські системи	Автоматизоване + ручне	Вимагають стабільності й точності, тому для роботи з великою кількістю даних та сценаріїв використовують автоматизацію, але тестування критично важливих функцій виконується вручну.
Вбудовані системи (IoT, автомобільна електроніка)	Ручне + часткова автоматизація	Перевірка реальної взаємодії пристроїв, автоматизація можлива лише частково через залежність від фізичних компонентів.
Освітні платформи	Ручне + автоматизоване	Ручне тестування важливе для оцінки інтерфейсу та навчального контенту, автоматизація використовується для тестів стабільності.
Хмарні сервіси	Переважно автоматизоване	Перевіряється масштабованість, безпека та доступність сервісів під час змін навантаження; активно використовуються скрипти для безперервного тестування. Юзабіліті перевіряється вручну.

Зростаюча залежність різних галузей від програмних рішень активно підсилює попит на забезпечення якості цього програмного забезпечення. У такому процесі цифрової трансформації бізнес стикається з необхідністю мати стабільні та високопродуктивні програмні системи. Будь-які помилки, вразливості чи проблеми з якістю можуть спричинити значні фінансові збитки, погіршення репутації або навіть юридичні наслідки. Тому недивно, що компанії надають дедалі більшої ваги процесам контролю якості, щоб їхні програмні продукти відповідали жорстким вимогам та стандартам. Натомість засобам забезпечення якості необхідно змінюватися, розвиватися та підстроюватися під нові тренди та виклики, задля можливості відповідати новим вимогам.

РОЗДІЛ 2
АНАЛІЗ СУЧАСНИХ ПІДХОДІВ ТА РІШЕНЬ АВТОМАТИЗАЦІЇ РОБОТИ QA-СПЕЦІАЛІСТІВ

2.1. Сучасний стан автоматизації в тестуванні програмних продуктів

Згідно звіту Software Quality Assurance Market, опублікованого у 2024 році компанією Research and Markets [8], на момент 2023 року розмір глобального ринку забезпечення якості програмних продуктів складав приблизно 13.5 мільярдів доларів з передбаченням росту до 22.4 мільярдів доларів до 2029 року із середньорічним темпом зростання 8.66%. Це свідчить про велику важливість та актуальність забезпечення якості, як компонента життєвого циклу розробки програмного забезпечення. У сучасних умовах цифрової трансформації, коли розробка програмних продуктів стає дедалі складнішою, швидшою та більш орієнтованою на безперервну інтеграцію й доставку (CI/CD), питання автоматизації процесів тестування виходить на перший план. Якщо ще десятиліття тому більшість компаній покладалася переважно на ручне тестування, то сьогодні автоматизація охоплює практично всі етапи QA процесу – від перевірки коду до комплексного регресійного аналізу. За результатами звіту World Quality Report 2024-25 компанії Capgemini [9], понад 80% IT-компаній світу використовують автоматизоване тестування, а близько 60% планують збільшувати рівень автоматизації у найближчі роки. Сучасна автоматизація тестування вже не обмежується лише створенням скриптів для перевірки функцій. Це комплексна інфраструктура, що охоплює керування тестовими середовищами, збір даних, аналітику результатів, моніторинг, прогнозування дефектів та навіть самонавчання системи тестування.

Серед глобальних тенденцій розвитку автоматизованого тестування можна виділити інтеграцію у гнучкі методології розробки та CI/CD процеси. Ці підходи орієнтовані на безперервну розробку, інтеграцію та доставку

продукту, що значно прискорює вихід нових версій програмних систем. За таких умов якісне та стабільне тестування стає складнішим завданням, адже такі методології розробки, як наприклад Agile і DevOps, передбачають постійні ітерації та швидкі цикли релізів. Це стимулює зростання попиту на сучасні інструменти та методики QA, що здатні підтримувати високий рівень якості в умовах динамічних змін. Agile-підхід робить акцент на тісній взаємодії між розробниками, тестувальниками та бізнес-замовниками, забезпечуючи регулярні релізи й оперативний обмін зворотним зв’язком. Такий формат роботи потребує широкого застосування автоматизованих засобів тестування, інструментів реального моніторингу та безперервної інтеграції для того, щоб кожне оновлення ПЗ залишалося стабільним, відповідало вимогам та правильно виконувало свої функції. Методологія DevOps, в свою чергу, поєднує процеси розробки й експлуатації, спрямовуючи зусилля на автоматизацію та оптимізацію всього життєвого циклу програмного забезпечення. Для її підтримки практики QA повинні адаптуватися, запроваджуючи інструменти безперервного тестування, автоматизованого регресійного аналізу та оцінки продуктивності, що дає можливість швидко впроваджувати зміни та гарантовано контролювати якість кожної ітерації. Такий розвиток ринку відповідає загальній тенденції до швидкого, ефективного та якісного постачання програмного забезпечення. Внаслідок цього гнучкі методології відіграють одну з ключових ролей у формуванні попиту на новітні рішення для забезпечення якості програмних продуктів. [8-11]

Хмарні технології також стають одним із ключових факторів, що впливають на розвиток ринку забезпечення якості програмного забезпечення. У міру того як компанії дедалі активніше переводять свої сервіси та дані у хмарні середовища, зростає потреба у платформах для тестування, які також працюють у хмарі. Такі рішення мають низку переваг порівняно з традиційними локальними підходами: вони забезпечують високу масштабованість, гнучкість у конфігурації середовищ і значне зниження

витрат. Завдяки хмарній інфраструктурі організації можуть використовувати широкий спектр інструментів тестування тоді, коли це потрібно, не вкладаючи кошти у власні сервери чи спеціальне обладнання. Це особливо важливо при роботі з великими або складними програмними продуктами, де обсяг тестування може суттєво змінюватися. Хмарні сервіси тестування також сприяють ефективній співпраці між членами команди, що знаходяться в різних локаціях. Вони дозволяють спільно використовувати тестові середовища, забезпечують оперативний обмін результатами перевірок і швидке виправлення помилок. Такі платформи ідеально відповідають потребам організацій, які працюють за принципами Agile та DevOps, адже забезпечують безперервність тестування та його інтеграцію у цикл розробки. Крім того, хмарні інструменти дозволяють проводити тестування на різних пристроях чи конфігураціях систем що підвищує якість кінцевого продукту та гарантує коректну роботу ПЗ на різних ОС, та браузерах. Окремою перевагою є можливість проводити у хмарі тестування продуктивності – відтворювати високі навантаження, моделювати реальні сценарії використання та оцінювати поведінку системи під тиском. Хоча питання безпеки даних у хмарі все ще викликає занепокоєння у деяких компаній, стрімкий розвиток технологій захисту та сучасних стандартів шифрування істотно знижує ці ризики. У майбутньому роль хмарних рішень у забезпеченні якості програмного забезпечення буде лише зростати, адже вони надають бізнесу потужні, масштабовані та економічні інструменти для підвищення якості продуктів. Крім того, такі платформи відкривають шлях до ширшої автоматизації тестових процесів, скорочуючи обсяг ручної роботи та підвищуючи як швидкість, так і охоплення тестування. [8-11]

Ще один, напевно найвпливовіший, тренд останніх років в IT і зокрема в автоматизації тестування – це залучення інструментів на основі штучного інтелекту. З розвитком інформаційних технологій об’єм даних та швидкість їх обробки тільки збільшується. І сучасні компанії, для збереження конкурентоспроможності, оптимізують свої процеси розробки програмних

продуктів. Стрімкий розвиток штучного інтелекту вже вносить великий вклад в це: ШІ-асистенти, що оброблюють природну мову та допомагають у аналізі вимог, впровадження та подальшої підтримці; моделі, які досліджують схожі програмні продукти та допомагають з проектуванням; генератори програмного коду з подальшим його аналізом та оптимізацією. Все це вже реально робочі інструменти, які використовуються у сучасних компаніях при виконанні поставлених завдань. В тестуванні програмного забезпечення рішення на основі штучного інтелекту також стають невід'ємною частиною забезпечення якості. Але чому традиційної автоматизації більше не достатньо? Звичайні інструменти для автоматизації тестування зазвичай працюють на основі фіксованих сценаріїв і заздалегідь визначених тест-кейсів. Такі фреймворки часто стикаються з труднощами під час взаємодії з динамічними інтерфейсами, потребують регулярного оновлення тестових сценаріїв і не здатні автоматично реагувати на непередбачені зміни у програмному продукті. У середовищах по типу Agile чи DevOps подібна негнучкість призводить до зростання витрат на підтримку тестів, уповільнення циклів релізів, появи прогалин в охопленні тестуванням та утворення обмежень у масштабуванні процесів забезпечення якості. Саме в таких умовах штучний інтелект дає змогу еволюціонувати сфері забезпечення якості. Як колись автоматизація процесів тестування стала відповіддю на неможливість перевірки дуже великих об'ємів даних вручну, інструменти на основі ШІ стають логічним наступним кроком – переходом від механічної автоматизації до інтелектуальної, де системи здатні самостійно навчатися, аналізувати дані, робити висновки та приймати рішення. Такі інструменти допомагають в генерації тестових сценаріїв та аналізі їх покриття, що підвищує якість і ефективність тестування. Також ШІ інструменти використовуються для автоматичного оновлення та повторного виконання тестів у випадку регресійного тестування з виходом нового релізу. [8-11]

Дослідивши сучасний стан автоматизованого тестування можна дійти висновку, що попит на інструменти автоматизації тестування тільки збільшиться в найближчі роки.

2.2. Аналіз сучасних інструментів автоматизованого тестування

Рівень ефективності процесу автоматизації значною мірою залежить не лише від якості створених тестів, але й від правильного вибору інструментів. З огляду на різноманіття фреймворків та платформ, важливо керуватися низкою критеріїв, що визначають доцільність використання конкретного рішення:

1. Тип програмного продукту та технологічний стек. Вибір інструменту має відповідати типу застосунку: веб, мобільний, десктопний, API, тощо. Наприклад, для мобільних застосунків є логічним використання Appium, а для навантажувальних тестів – JMeter.

2. Підтримка мов програмування. У команді може бути встановлений технологічний стек, який визначає вибір інструменту. Наприклад Selenium, підтримує багато мов, тоді як Cypress – лише JavaScript/TypeScript.

3. Сумісність із CI/CD та DevOps-процесами. Якщо в компанії використовують гнучкі методології розробки то автоматизовані тести повинні мати можливість інтегруватися до них.

4. Швидкість виконання тестів та стабільність роботи. Якщо тестування необхідно провести у дуже стислий час, то з’являються серйозні вимоги до ефективності та стабільності роботи автоматизованих програм.

5. Поріг входу та потреба у програмуванні. Low-code середовища скорочують час впровадження, тоді як класичні фреймворки вимагають глибших технічних навичок.

6. Вартість інструменту. Вона може бути критичною для компаній із обмеженим бюджетом. Selenium, Cypress та JMeter є повністю безкоштовними, тоді як TestComplete і Applitools – комерційні рішення.

7. Підтримка інструменту, спільнота та оновлення. Популярні фреймворки мають активну спільноту та швидко адаптуються до змін у браузерах чи операційних системах.

8. Інші критерії, специфічні для місця роботи чи об'єкту тестування.

Це лише базові критерії, які не охоплюють всі тонкощі робочого процесу, але важно зазначити, що використовуючи лише їх вже є можливість підібрати декілька програмних засобів тестування з якими вже можна працювати. Далі йде попереднє тестування інструменту, так званий Proof of Concept, пишуться пробні типові тести та оцінюється продуктивність та зручність роботи. Після вдалої перевірки інструменту, він інтегрується у процес розробки разом із визначенням стратегії автоматизації та підходів до побудови автоматизованих тестів. [12, 13, 14]

Представлені на ринку інструменти демонструють надзвичайне різноманіття як за архітектурою, так і за сферою призначення. Це обумовлено тим, що автоматизація не є універсальним процесом: кожен тип застосунку, кожний технологічний стек і кожний підхід до розробки вимагає специфічних засобів тестування. У веб-додатках необхідні браузерні фреймворки; у мобільних це крос-платформні інструменти, здатні взаємодіяти як з Android, так і з iOS; для тестування продуктивності потрібні спеціалізовані рішення, орієнтовані на симуляцію навантажень; для API – інструменти, що дозволяють структуровано і масштабовано перевіряти різні типи інтеграцій. У таблиці 2.1 представлені декілька популярних інструментів автоматизованого тестування з коротким описом, перевагами та недоліками.

Таблиця 2.1

Характеристика популярних інструментів автоматизації

Назва інструменту	Основне призначення	Переваги та недоліки
Selenium	Фреймворк для автоматизації UI-тестування веб-додатків у різних браузерах. Підтримує Java, Python, C#, JS та інші мови.	Переваги: Відкрите ПЗ, підтримка багатьох мов програмування (Java, Python, C#), інтеграція з більшістю CI/CD систем, велика гнучкість та масштабованість завдяки різноманітним плагінам. Недоліки: Потребує знань мов програмування, нестабільність при зміні DOM, відсутність власної системи звітів.
Cypress	Інструмент для тестування фронтенду, працює всередині браузера, аналог Selenium аде з дещо меншим функціоналом	Переваги: швидке виконання тестів, зручний дебагінг, проста конфігурація та більша швидкість освоєння порівняно з Selenium. Недоліки: обмежена підтримка браузерів та специфічних тестових сценаріїв.
Appium	Інструмент для автоматизації тестування мобільних застосунків на різних платформах (Android та iOS).	Переваги: підтримка різних платформ, адаптивність, інтеграція з Selenium. Недоліки: повільна робота, складне налаштування середовищ і емуляторів.
Katalon Studio	Платформа для UI, API, мобільного і десктопного тестування з підтримкою low-code.	Переваги: доступність, зручний інтерфейс, інтеграція з CI/CD. Недоліки: обмежена кастомізація, деякі функції лише у платній версії.
Postman	Інструмент для автоматизації тестування API, підтримує REST, GraphQL, SOAP	Переваги: простий інтерфейс, колекції, змінні, середовища, інтеграція з CI/CD. Недоліки: не підходить для складних сценаріїв автоматизації, частково ручні процеси.

Продовження таблиці 2.1

Назва інструменту	Основне призначення	Переваги та недоліки
TestComplete	Комерційна платформа для UI-тестування веб, десктопних і мобільних застосунків.	Переваги: low-code, візуальне створення тестів, широкий набір інтеграцій. Недоліки: висока вартість, залежність від Windows-середовища.
Playwright	Сучасний крос-браузерний фреймворк для UI-тестування, підтримує браузери на Chromium, Gecko та WebKit.	Переваги: паралельні тести, висока стабільність, автозапис сценаріїв. Недоліки: новий інструмент, менша спільнота, потребує знань програмування.
Apache JMeter	Засіб для навантажувального та стрес-тестування різних систем і сервісів.	Переваги: безкоштовний, гнучкий, підтримує розподілені навантаження. Недоліки: може бути складним у налаштуванні через застарілий інтерфейс.

Порівнюючи наведені інструменти, можна зробити висновок, що не має універсального рішення, яке б підходило до всіх типів проектів. Інструменти різняться за архітектурою, продуктивністю, вимогами до технічної експертизи та сферами застосування. Але незважаючи на різноманіття інструментів, їх використання пов’язане з низкою обмежень. У багатьох випадках автоматизація хоч і підвищує швидкість виконання тестів, однак водночас збільшує навантаження на фахівців, оскільки потребує постійного оновлення сценаріїв, аналізу результатів та адаптації тестів до змін у програмному продукті. Саме ці виклики стали передумовою появи нових підходів у забезпеченні якості – зокрема впровадження інтелектуальних систем, здатних аналізувати поведінку застосунку, автоматично оптимізувати тести та підтримувати роботу QA-спеціалістів.[14-16]

2.3. Впровадження інтелектуальних систем у роботу QA-спеціалістів

Використання інтелектуальних технологій у тестуванні програмного забезпечення дає низку суттєвих переваг, які істотно впливають на загальну ефективність QA-процесів. Насамперед інтеграція ШІ дозволяє значно підвищити продуктивність роботи тестувальників, оскільки більшість рутинних завдань може виконуватися автоматично. Це скорочує час на підготовчі етапи та дає можливість спеціалістам зосередитися на аналітичних, дослідницьких або завданнях високого рівня, які потребують людського досвіду та критичного мислення. Крім того, інтелектуальні системи здатні зменшувати кількість помилок, що виникають через людський фактор. Алгоритми машинного навчання працюють послідовно та об'єктивно, покладаючись на аналіз структурованої інформації, а не на суб'єктивне трактування вимог. Це сприяє більш точному розумінню критичних сценаріїв, покращує якість сформованих тестів та сприяє підвищенню рівня тестового покриття.

Варто зазначити позитивний результат впровадження ШІ до тестування та розробки програмних продуктів в цілому. Як показує дослідження компанії Copilot [17] команди розробників, які використовували інструменти ШІ при розробці, виконали на 26% більше завдань ніж ті, які їх не використовували. Проте дослідження компанії Metr [18] показує цікаві результати. Виявилось, що більш досвідчені розробники (рівня senior) виконуючи тестові завдання з використанням ШІ інструментів витратили на 19% більше часу на виконання ніж ті які не користувалися ШІ. Додаткові витрати часу пояснюються необхідністю перевірки та редакції згенерованого ШІ коду розробниками. Але у Metr зазначають – при виконані інших завдань, сценаріїв або коли у інженера не має навичок роботи з конкретною кодовою базою, використання інструментів на базі штучного інтелекту може бути дуже корисним. Також на думку більшості розробників, які приймали участь у дослідженні, інструменти ШІ полегшують розробку – з їх слів редагувати код легше ніж писати його

повністю з початку.

Попри всі переваги, які глобальне впровадження ШІ надасть у процес розробки ПЗ, слід також не забувати про серйозні виклики цього. Одним із ключових є висока вартість розробки та інтеграції таких рішень у наявні робочі процеси. Не всі команди мають ресурси для адаптації внутрішніх інструментів до роботи з ШІ або для налаштування складних моделей машинного навчання. Окрім цього, якість роботи інтелектуальних систем напряму залежить від якості даних, на яких вони навчаються: неповні, нерелевантні або застарілі дані можуть призвести до неправильних висновків і некоректної генерації тестових сценаріїв. Складність також полягає в тому, що впровадження ШІ потребує від працівників нових компетенцій. Вони повинні розуміти принципи роботи моделей, оцінювати якість їхніх відповідей, контролювати коректність висновків та вміти поєднувати результати роботи алгоритмів зі своїм досвідом. Це підвищує вимоги до кваліфікації персоналу та створює потребу в додатковому навчанні. Не менш важливим викликом є ризик надмірної автоматизації, коли частина рішень приймається системою без належного контролю людини. У критичних проектах це може спричинити пропуск важливих дефектів або хибну інтерпретацію результатів тестування. Додатково, використання інтелектуальних систем піднімає питання інформаційної безпеки та захисту даних, оскільки для роботи моделей часто використовуються великі масиви внутрішньої інформації, яка може бути конфіденційною. [19-22]

Усе це свідчить про те, що хоча штучний інтелект і пропонує потужні інструменти для підвищення ефективності тестування, його застосування потребує зваженого підходу та не може розглядатися як повна заміна людської експертизи. Оптимальним напрямом розвитку є інтеграція ШІ як допоміжного механізму, здатного розширити можливості QA-команди, а не замінити її.

На сьогодні вплив ШІ на тестування можна виділити в декілька напрямів, що мають практичну цінність. Хоча не всі з них можливо реалізувати на достойному рівні зараз, на мою думку їх включення до процесів

QA в майбутньому значно підвищить якість та ефективність тестування програмних засобів.

Перший напрям – це генерація тестових сценаріїв штучним інтелектом на основі існуючої інформації про продукт. Системи, що використовують машинне навчання, можуть аналізувати специфікації, історію змін коду, лог-файли або поведінку користувачів для формування найбільш релевантних тестів. Наприклад, моделі на основі аналізу природньої мови (Natural Language Processing, NLP) можуть інтерпретувати вимоги, що написані людиною зовсім незнайомою з процесом розробки ПЗ, і перетворювати їх у тестові сценарії. Це значно скоротить час підготовку тестової бази й мінімізує людські помилки при розумінні вимог.

Далі, системи-тестувальники з штучним інтелектом можуть автоматично аналізувати результати виконання тестів, розпізнаючи закономірності в поведінці програми. Наприклад, вони зможуть виявляти типові причини збоїв, групувати помилки за схожими характеристиками або визначати, які дефекти мають найвищий пріоритет для виправлення. Ще таким рішенням можна зробити здатність навчатися на історії минулих тестів, прогнозуючи, які модулі найбільш схильні до збоїв, і вчасно попереджати команду про потенційні ризики.

Ще один з важливих напрямів – це надання тестам можливості самим адаптуватися до змін у системі чи коді. Традиційна автоматизація стикається з проблемою «крихкості» тестів, де навіть незначні зміни в інтерфейсі або структурі коду можуть призвести до збоїв при їх виконанні. ШІ дозволяє створювати тести, які здатні «підлаштовуватись» під зміни системи. Наприклад, якщо змінюється елемент інтерфейсу (назва кнопки або її положення), система з ШІ може самостійно знайти новий відповідник, спираючись на контекст, а не на жорстко задані селектори. Такий підхід значно зменшить обсяг ручної підтримки автоматизованих сценаріїв.

Аналіз реальної поведінки користувачів теж можна відстежувати за допомогою інструментів з штучним інтелектом. Це дозволить визначити

найбільш популярні сценарії використання й сформувати на їх основі тестові кейси. Такий підхід забезпечить тісний зв’язок між тестуванням і реальним користувацьким досвідом, роблячи тести більш практичними та релевантними.

І на кінець, за допомогою ШІ вже зараз можна створити ботів-помічників, які будуть інтегруватися у робочі середовища і допомагати аналізувати дані, формувати звіти, пропонувати оптимальні сценарії тестування чи навіть створювати автоматизовані тести за запитом користувача. Такі системи фактично будуть виконувати роль “цифрових асистентів” у QA-процесах, що підвищить продуктивність команди та мінімізує кількість рутинних завдань.

Сьогодні на ринку вже існує низка інструментів, що реалізує деякою мірою вищезазначені можливості. Приклади найбільш відомих з них наведені у таблиці 2.2.

Таблиця 2.2

Приклади сучасних рішень на основі штучного інтелекту

Назва	Опис
Testim.io	Платформа, що використовує ШІ для автоматичного створення та підтримки UI-тестів, здатних адаптуватися до змін у DOM-структурі.
Applitools Eyes	Інструмент для візуального регресійного тестування та порівняння UI за допомогою штучного інтелекту.
Mabl	Інтелектуальна платформа для безперервного тестування, що аналізує поведінку програми й автоматично оновлює тести при змінах у коді.
Test.ai	Рішення для автоматизації мобільного тестування, яке використовує нейронні мережі для розпізнавання елементів інтерфейсу.

Усі ці інструменти демонструють, що ШІ у тестуванні вже не є теоретичною концепцією – це практичний інструмент, який реально змінює підхід до забезпечення якості. [19-22]

Отже резюмуючи проведений аналіз, можна сказати, що зараз на ринку інструментів автоматизації тестування є попит на невеликих інтелектуальних помічників, які можна легко вбудувати у процеси безперервної розробки програмних продуктів.

РОЗДІЛ 3
БОТ-АСИСТЕНТ ДЛЯ АВТОМАТИЗАЦІЇ РОБОТИ QA-СПЕЦІАЛІСТІВ

3.1. Ідея Проекту

Ціллю розробки бота-асистента є підвищення ефективності роботи працівників відділу забезпечення якості та всієї команди розробки в цілому шляхом автоматизації деяких процесів. Перше, що повинно надати впровадження цього рішення – це зменшити показник Time To Market (TTM), тобто час, необхідний команді на розробку проекту від ідеї до виходу на ринок. Досягається це зменшенням часу на виконання рутинних завдань, а саме створення та написання тест-кейсів (або тестових сценаріїв – детального опису послідовності дій для перевірки конкретної функції ПЗ). Також передбачається, що впровадження рішення підвищить якість розроблюваних продуктів та задоволеність співробітників.

Вимоги до проекту такі:

1. Вміння генерувати тестові кейси
2. Можливість кастомізації
3. Збереження інформаційної безпеки
4. Інтеграція з системою керування проектами (Jira)
5. Невеликий розмір
6. Можливість інтеграції у CI/CD та гнучкі методології

Обрано Jira, тому що це одна з найпоширеніших у світі систем для управління проектами, завданнями та процесами розробки програмного забезпечення, створена компанією Atlassian. Первинно орієнтована на відстеження помилок, Jira згодом еволюціонувала у комплексну платформу для організації командної роботи, яка активно використовується в IT-компаніях, зокрема в командах QA, DevOps, аналітиків та менеджерів. Jira є веб-орієнтованою системою, яка дозволяє централізовано керувати життєвим циклом завдань у проектах. Вона має вбудовані модулі для широкої підтримки

методології Agile (Scrum, Kanban), інтеграції з CI/CD процесами, інструментами тестування та використовується для планування, координації і моніторингу роботи всієї команди. Загалом Jira повністю задовольняє вимоги проекту.

Збереження інформаційної безпеки у крупних компаніях забезпечується DLP-системами. Це комплекс програмних та організаційних рішень, призначених для запобігання витоку чутливої, конфіденційної або корпоративної інформації за межі компанії. Для IT-компаній, які працюють із вихідним кодом, клієнтськими даними, внутрішніми документаціями та інтелектуальною власністю, DLP є критично важливим елементом кібербезпеки. Така система контролює, які дані і через які канали працівники можуть передавати чи отримувати, реєструє несанкціоновані дії персоналу та забезпечує політики, що регламентують, хто і які саме дані може переглядати, копіювати, пересилати чи завантажувати. Можна сказати що це серйозна, велика та дорога система, впровадження якої займає багато часу. В мене не має доступу до реальної системи, але для цього проекту вона і не потрібна – при розробці було реалізовано модуль зв’язку, який можна використати з реальною системою, а логіка її роботи реалізована простою функцією.

3.2. Аналіз готових рішень

Рішенням з найбільш задовольняючим функціоналом є Applitools. Це платформа для автоматичного візуального тестування програмного забезпечення, що використовує штучний інтелект для виявлення помилок у інтерфейсі користувача. Замість простого порівняння пікселів, технологія "AI Vision" від Applitools аналізує семантичний зміст екрану, щоб визначити, чи не змістилася кнопка або чи не перекривається текст, і інтегрується з популярними фреймворками на кшталт Selenium, Cypress та Playwright. З плюсів можна виділити те, що це повністю хмарна платформа, а значить не потрібно витрачати ресурси та її легко інтегрувати у робочі процеси. Вона

підтримує багато середовищ та має сумісність з багатою кількістю фреймворків. Має свою вбудовану систему керування проектами. Мінуси – це комерційний продукт за який треба платити, а значить й кастомізувати його ніяк не вийде. Це хмарне рішення, яке виконує аналіз на серверах власника продукту, що створює потенційну загрозу витоку важливої інформації та додаткові напрями можливих атак зловмисників. Загалом це рішення не підходить під вимоги до проекту.[23]

Наступна ідея вирішення поставленої задачі – це надати працівникам можливість самостійно використовувати великі лінгвістичні моделі (наприклад ChatGPT). Перевагою такого підходу є те, що не потрібно нічого створювати, розробляти чи впроваджувати. Основним недоліком є відсутність можливості стежити за відправленими даними, що надає великий шанс витоку внутрішньої інформації. Також ефективність такого рішення, тобто значущість наданих їм даних, буде цілком залежати від працівника: як він збере необхідні дані для генерації та як напише запит до моделі. Тобто працівнику необхідно бути не лише досвідченим, а ще й вміти правильно писати промпт для запиту, що все в купі суперечить основній ідеї розробки проекту.

3.3. Опис технологій рішення, яке було розроблено

Розроблене рішення представляє собою текстового помічника з frontend-ом, реалізованим у вигляді телеграм-бота та backend-ом на Python 3.10 [24]. Бібліотеки Python, які використовувалися:

1. Telebot, також відома як PyTelegramBotAPI, – це популярна Python бібліотека для створення ботів у месенджері Telegram. Вона надає простий та інтуїтивний інтерфейс для взаємодії з Telegram Bot API.
   Основні можливості:
   a. обробка текстових повідомлень, команд та callback-подій;
   b. робота із кнопками (reply та inline keyboards);

c. отримання та надсилання медіа файлів, документів, геолокації;
d. підтримка веб-хуків та long polling;
e. організація діалогів і станів користувача.

Завдяки цій бібліотеці є можливість реалізувати роботу телеграм чат-бота [25].

2. Взаємодія з файлами yaml у Python реалізується через бібліотеку PyYAML, яка дозволяє зчитувати, обробляти та записувати дані у форматі YAML. Це формат структурованих даних, який часто використовують для:
   a. конфігурацій сервісів;
   b. налаштувань застосунків;
   c. зберігання словників, параметрів, списків.
   d. Можливості PyYAML:
   e. завантаження YAML-файлів у Python-структури (dict, list);
   f. серіалізація Python-об'єктів у YAML;
   g. безпечне читання;
   h. підтримка вкладених структур.

Ця бібліотека дає можливість створити єдиний файл з усіма параметрами, які користувач повинен передати перед використанням застосунку [26].

3. Requests – одна з найпопулярніших Python-бібліотек для виконання HTTP-запитів. Вона значно спрощує роботу з веб-API та сервісами [27]. Можливості:
   a. виконання GET, POST, PUT, DELETE запитів;
   b. робота з JSON-даними;
   c. авторизація (Basic, Bearer, OAuth);
   d. налаштування заголовків, параметрів та cookies;
   e. робота з сесіями;
   f. обробка помилок та кодів відповіді.
4. Logging – стандартна бібліотека Python для ведення журналів роботи

застосунку. Вона забезпечує збирання інформації про помилки, попередження, діагностичні повідомлення, що особливо важливо для застосунків у розробці та впровадженні. Можливості:

a. різні рівні логів: DEBUG, INFO, WARNING, ERROR, CRITICAL;
b. запис логів у консоль, файл, ротацію файлів;
c. форматування повідомлень;
d. кастомні логери та хендлери;
e. централізований контроль ведення жкрналів у складних застосунках.

Забезпечує основне логування подій та помилок, що не раз допомагало при розробці [28].

5. JIRA (jira-python) – офіційна Python-бібліотека для взаємодії з Atlassian JIRA через REST API. Автоматизація бізнес-процесів у JIRA, таких як створення, оновлення та аналіз задач. Можливості:
   a. створення нових issue (bugs, tasks, stories);
   b. зміна статусів задач;
   c. додавання коментарів, файлів, описів;
   d. виконання JQL-запитів;
   e. отримання даних про дошки та спринти користувачів;
   f. інтеграція з ботами та CI/CD.

   Через неї реалізована взаємодія з Atlassian JIRA [29].
6. OpenAI (openai-python) – офіційна Python-бібліотека для інтеграції з API моделей OpenAI, таких як GPT, Embeddings, Whisper тощо. Забезпечує взаємодії з мовними моделями ШІ, створення інтелектуальних сервісів, чат-ботів та автоматизаційних систем. Можливості:
   a. надсилання запитів до моделей GPT;
   b. генерування тексту, коду, пояснень, структур даних;
   c. робота з embedding-векторними представленнями;
   d. запуск класифікації, резюмування, переписування тексту;
   e. обробка аудіо та зображень (якщо дозволено API-моделлю);

f. створення діалогових ботів та асистентів.

Відправка та отримання повідомлень від ШІ ChatGPT реалізовано через неї [30].

3.4. Реєстрація нового Telegram бота через BotFather

Перед використанням розробленого рішення, необхідно створити власного Telegram бота, який буде інтерфейсом користувача для нього. Свого Telegram бота можна легко зареєструвати за допомогою офіційного чат-боту BotFather від Telegram. Для цього необхідно зайти в телеграм, в пошуку написати BotFather і з списку знайдених обрати того, що з синьою галочкою. (Рис. 3.1.)

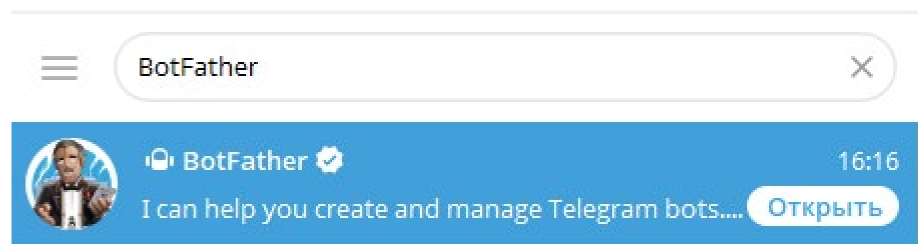

Рис. 3.1. Пошук BotFather

Далі необхідно написати /start та з’явиться список зі всіма командами, які цей бот може зробити. (Рис. 3.2.)

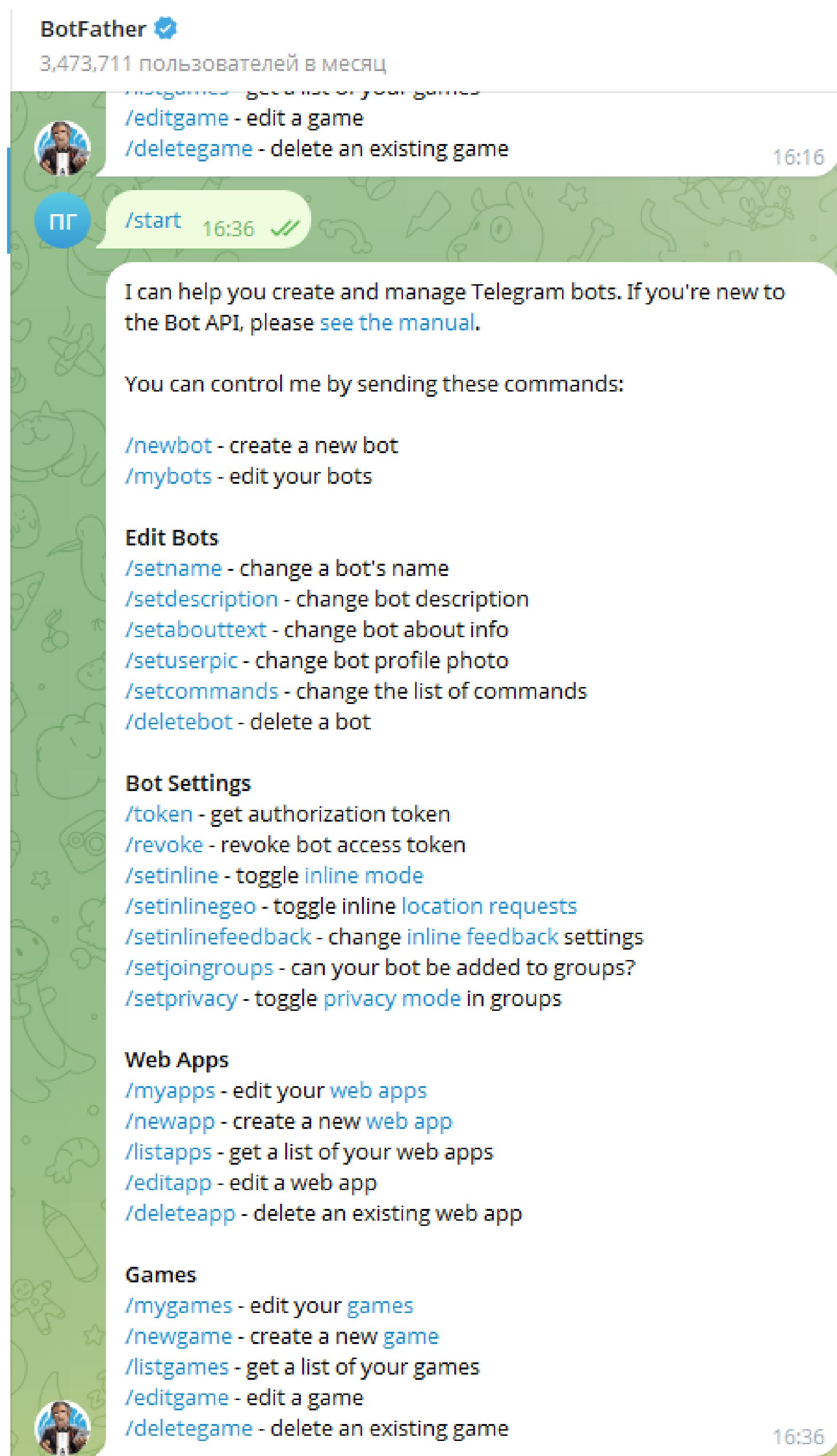

Рис. 3.2. Всі команди BotFather

Для створення власного бота необхідно ввести /newbot. Далі BotFather попросить придумати та ввести ім'я нового бота, а після його унікальний @username, який обов'язково повинен закінчуватися на bot. (Рис. 3.3.)

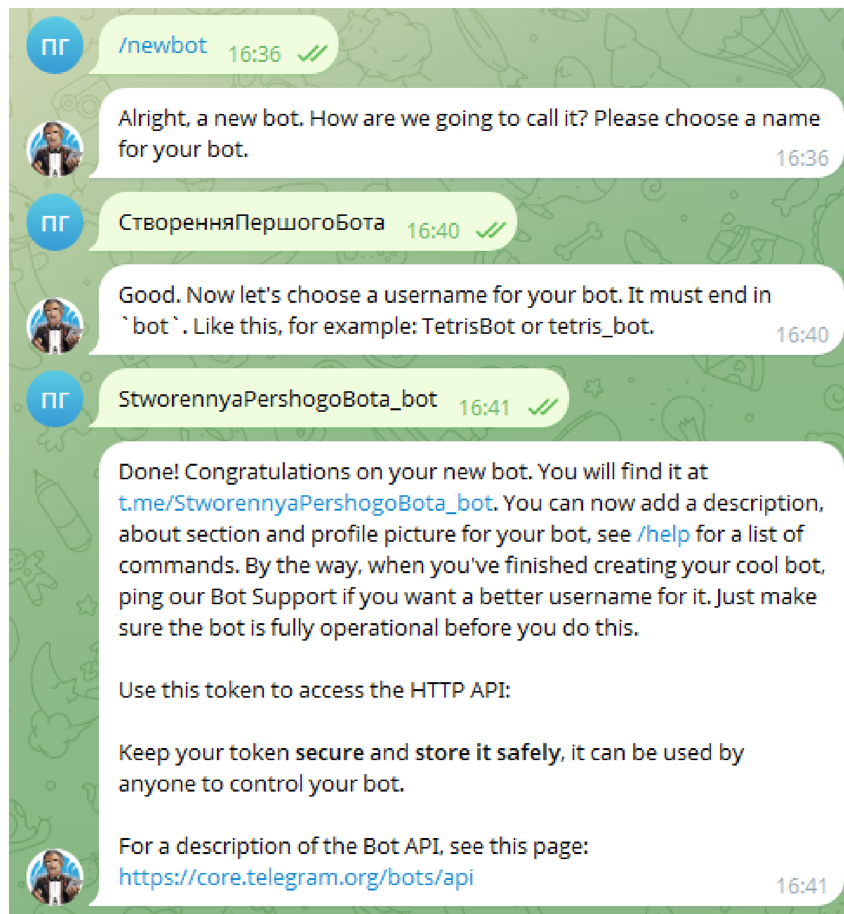

Рис. 3.3. Створення нового телеграм-бота

Результатом цих дій повинно стати повідомлення від BotFather, у якому буде потрібний нам токен створеного бота. Далі всі налаштування будуть робитися через Python, але тут, в BotFather, також можна зробити багато чого. Для огляду наявних дій з своїм ботом необхідно набрати /help для повторного показу повідомлення зі всіма командами та написати /mybots, після чого буде показано всіх створених ботів. (Рис. 3.4.)

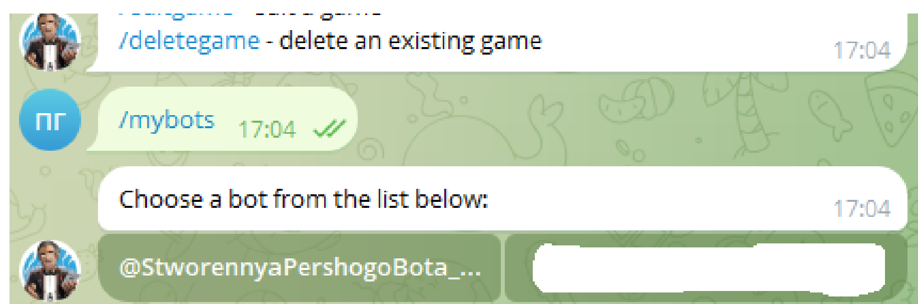

Рис. 3.4. Налаштування створеного бота через BotFather

Далі обираємо необхідного бота та бачимо головне меню управління ним. (Рис.3.5.)

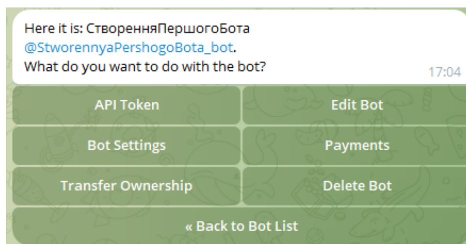

Рис. 3.5. Головне меню управління створеним ботом

Тут ми бачимо, що можна перейти до підменю, у яких зробити наступне:

Вибрати Api Token та побачити токен бота з можливість його зміни (Рис.3.6.)

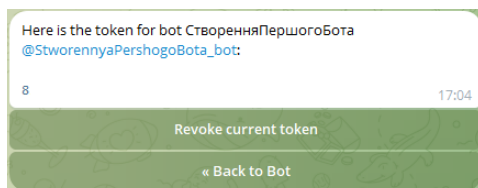

Рис. 3.6. Управління токеном бота

Вибрати Bot Settings, в яких налаштувати поведінку бота (Рис.3.7.)

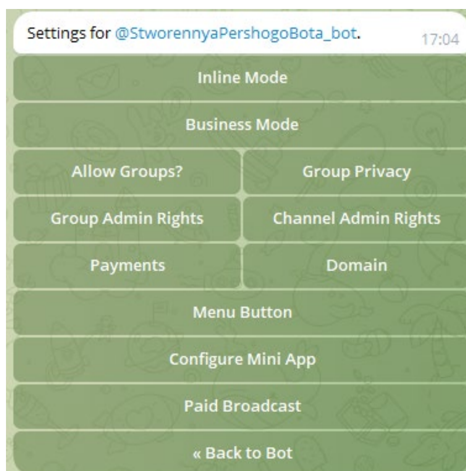

Рис. 3.7. Налаштування поведінки бота

Вибрати Transfer Ownership та передати володіння ботом іншому користувачу телеграма (Рис.3.8.)

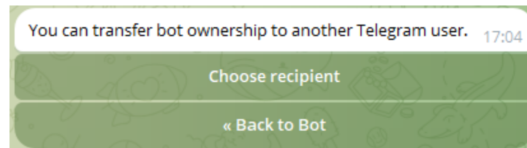

Рис. 3.8. Передача бота іншому користувачеві телеграм

Edit Bot, де можна змінити ім'я бота, опис, аватар, політику приватності та команди. (Рис.3.9.)

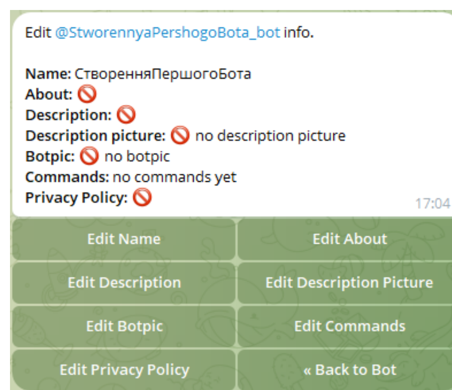

Рис. 3.9. Налаштування функціоналу бота через BotFather

І на кінець Delete Bot для видалення бота. (Рис.3.10.)

Рис. 3.10. Видалення бота

При реєстрації нового Telegram-бота було використано офіційну документацію[31].

3.5. Підготовка до початку роботи

Перед початком роботи з програмою необхідно заповнити файл config.yaml (Рис.3.11.), а саме такі пункти:

bot_id – токен Telegram бота, який видав BotFther;

log_path – шлях до log, у який буде записуватися журнал подій;

jira_link – URL-посилання на робочий простір у Jira;

jira_email – електрону пошту для входу до Jira;

jira_token – Jira Api Token для аутентифікації;

ChatGPT_API – OpenAI Api Key для інтеграції ChatGPT;

dlp_url – посилання на DLP-систему;

dlp_token – токен аутентифікації в DLP-системі.

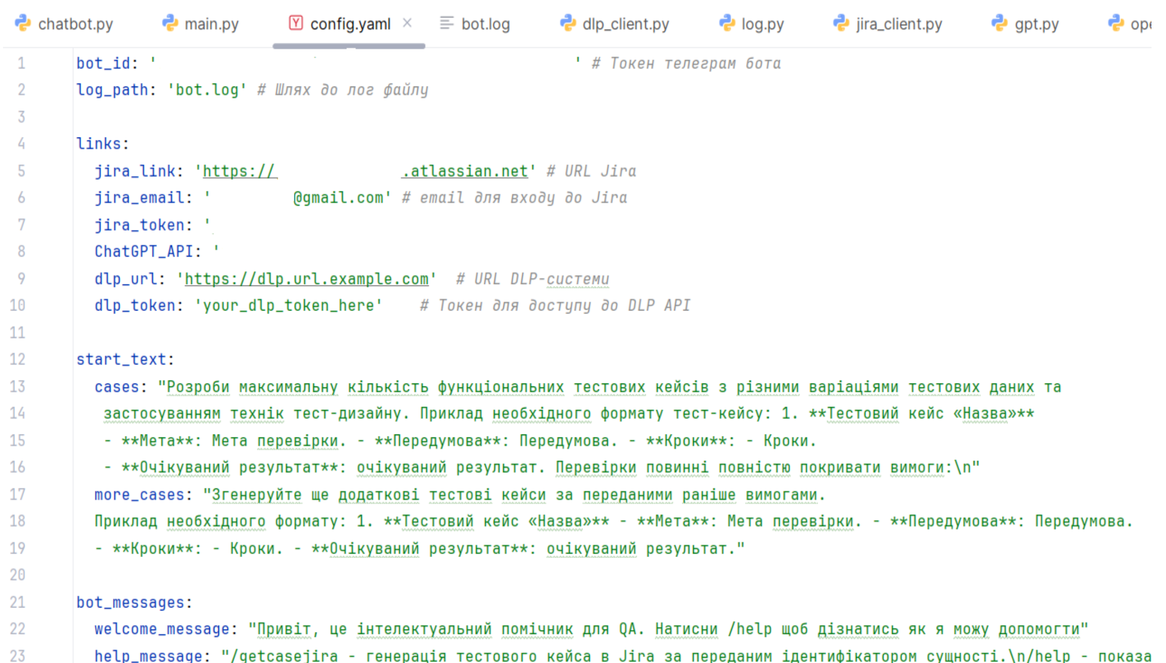

Рис. 3.11. Заповнений config.yaml

Jira Api Token можна отримати виконуючи наступні кроки:

Увійти до свого робочого простору у Jira, натиснути у правому верхньому куті на свій профіль і вибрати Account Settings (Рис.3.12.)

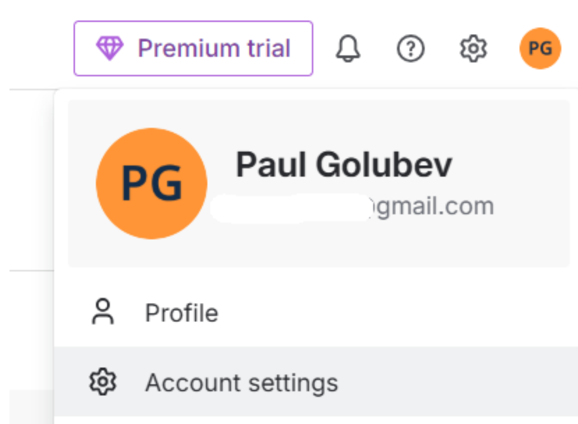

Рис. 3.12. Відкриття налаштувань акаунта у Jira

Далі у верхній строчці вибрати Security, де потім Create and manage API tokens (Рис.3.13.). Відкриється меню керування, де можна подивитися наявні, створити нові чи видалити токени Jira Api (Рис.3.14.) [32].

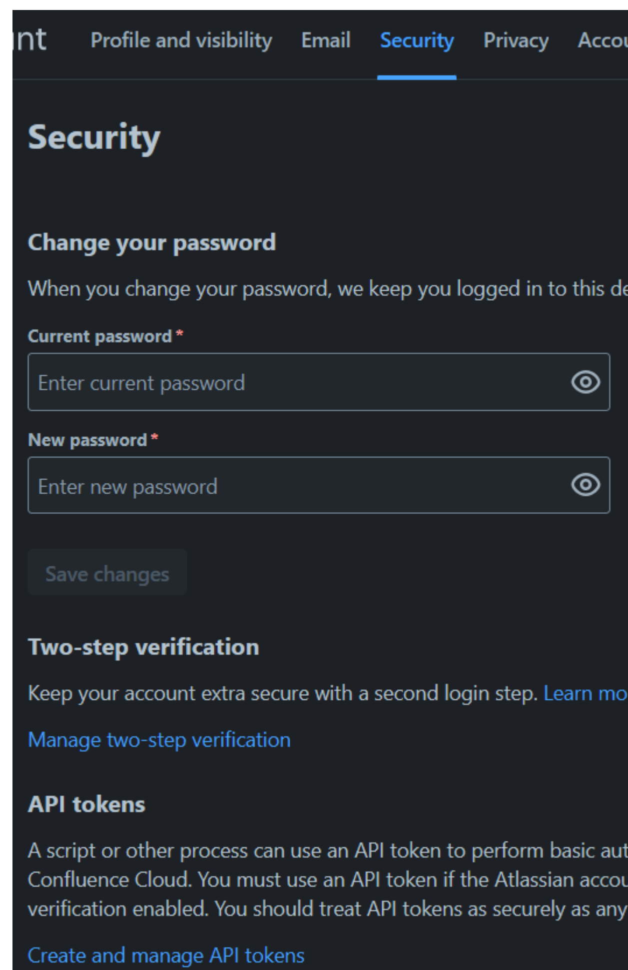

Рис. 3.13. Налаштування безпеки у акаунті Jira

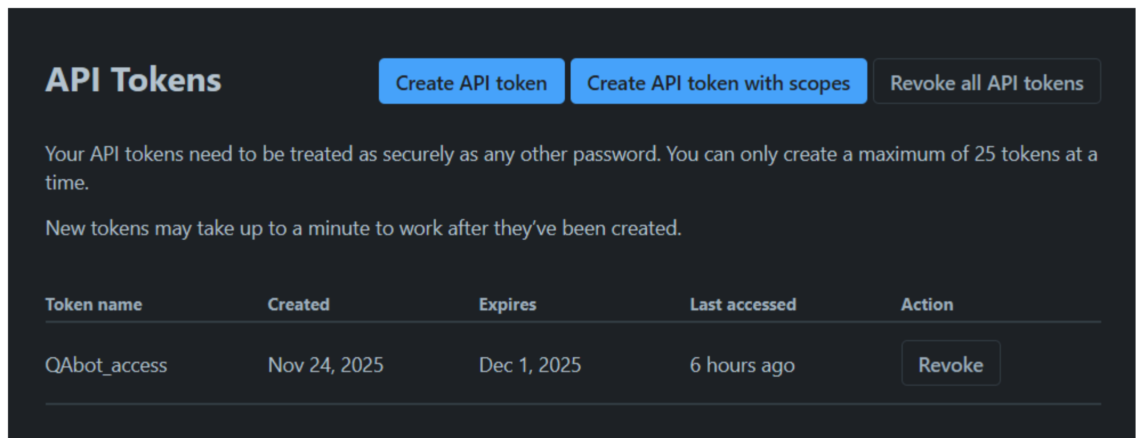

Рис. 3.14. Меню керування токенами Jira Api

Для отримання OpenAI API Key необхідно спочатку зареєструватися чи увійти до OpenAI акаунта на сайті [33] та перейти до меню керування ключами за посиланням [34] і вибрати Create new secret key (Рис.3.15.). Далі необхідно обрати ім'я для ключа та натиснути Create secret key, після чого буде показано створений ключ, який можна використовувати. [35]

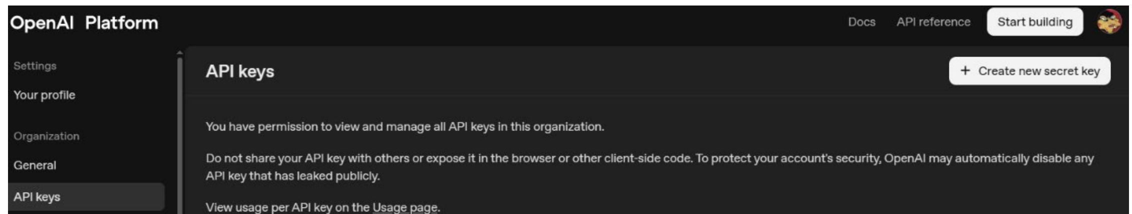

Рис. 3.15. Управління API ключами на сайті OpenAI

Для отримання посилання на DLP-систему та токену її аутентифікації необхідно звернутися до документації конкретної системи, яка в вас встановлена, а також в процесі дотримуватися всі' політик інформаційної безпеки у вашій компанії.

3.6. Опис роботи логіки компонентів розробленого рішення

Рішення працює за наступними схемами: (Рис.3.16.), (Рис.3.17.)

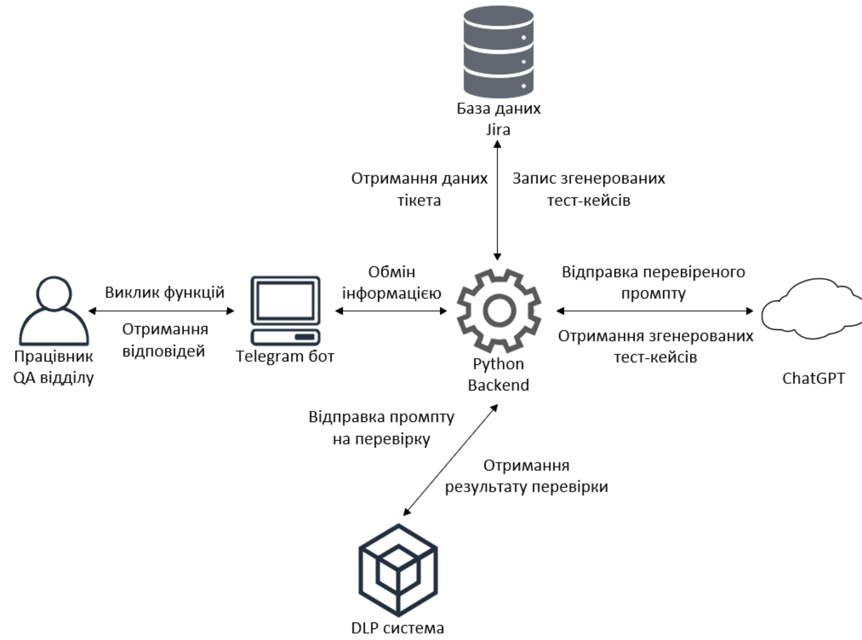

Рис. 3.16. Схема роботи розробленого рішення

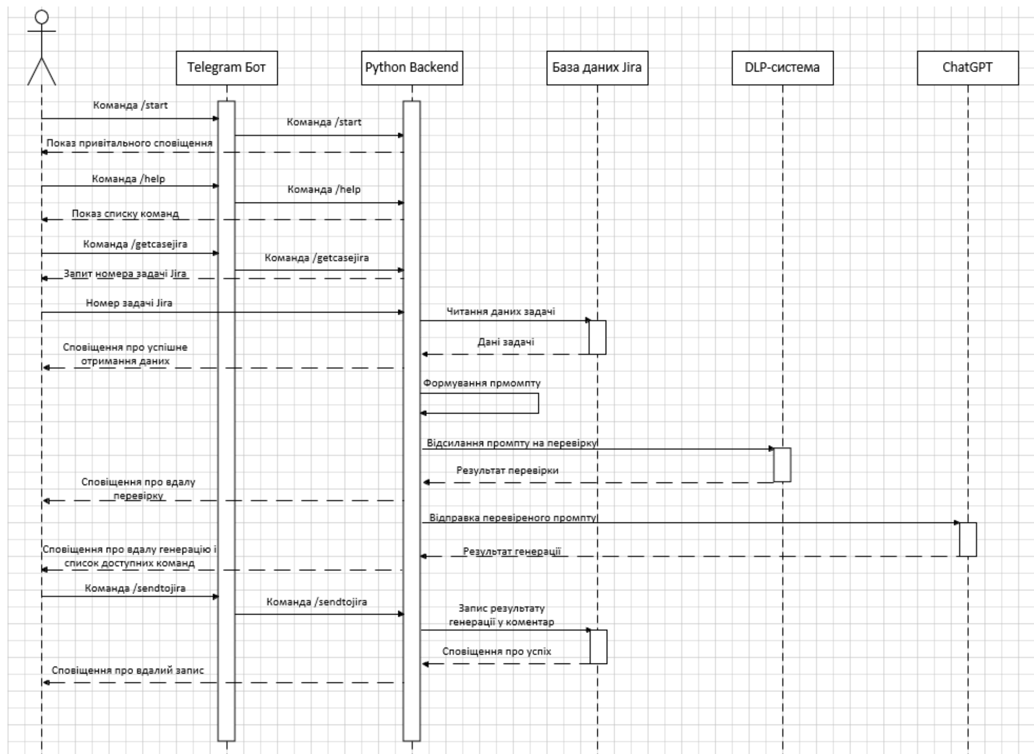

Рис. 3.17. Схема послідовності

Після заповнення config.yaml та запуску файлу main.py уся подальша взаємодія з рішенням буде проходити через Telegram бота. Тепер необхідно написати /start чат-боту, і він покаже привітальне повідомлення (Рис.3.18.), з пропозицією виклику команди /help для перегляду всіх доступних команд (Рис.3.19.).

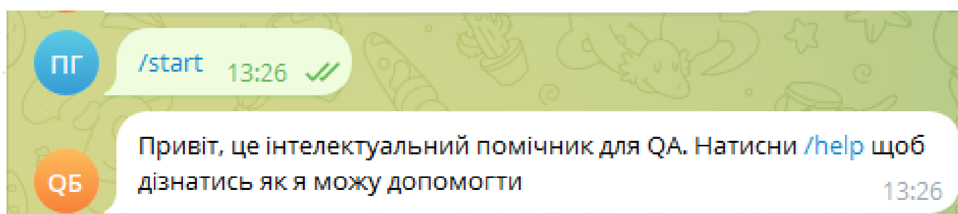

Рис. 3.18. Привітальне повідомлення чат-боту

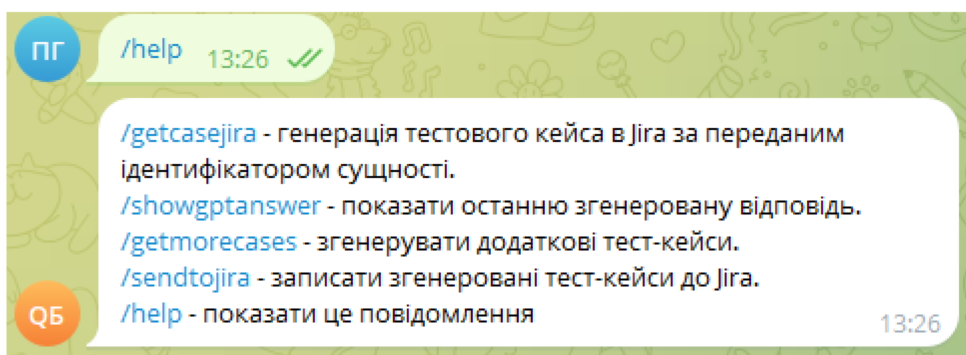

Рис. 3.19. Виклик списку всіх команд

Зараз доступна лише /getcasejira, тому що для інших спочатку треба виконати генерацію. Викликаємо /getcasejira і бот запитує нас ідентифікатор задачі у Jira, опис якої і буде використовуватися для генерації тестових кейсів. (Рис.3.20.)

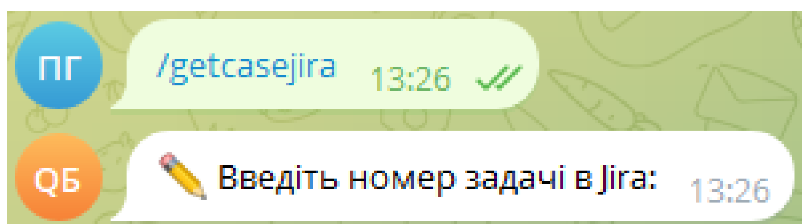

Рис. 3.20. Виклик функції /getcasejira

Для прикладу використання я написав задачу у Jira, в якій треба створити тестові кейси за поданими вимогами, її номер – KAN-5 (Рис.3.21.).

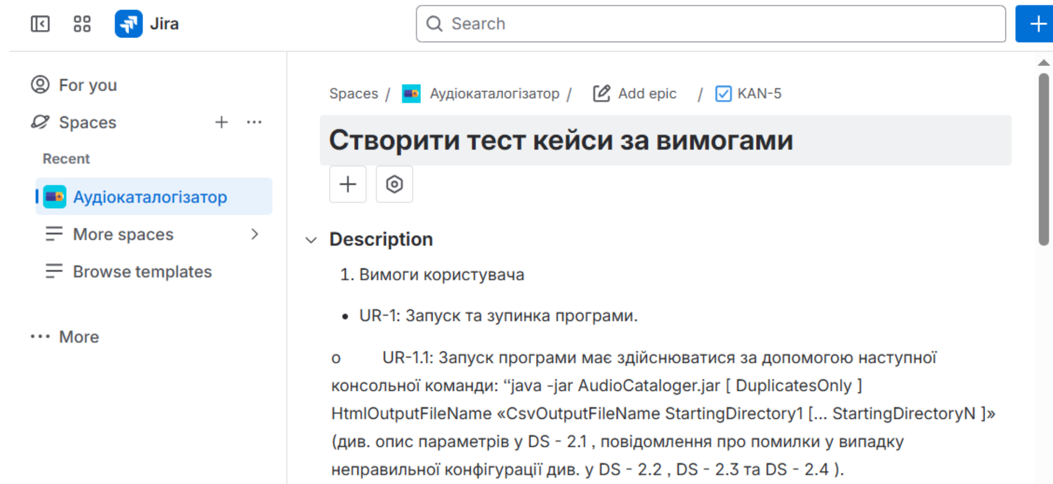

Рис. 3.21. Задача KAN-5 у Jira

Далі я відправляю номер необхідної задачі чат-боту. Програма зчитує дані з задачі у Jira, створює з них промпт, відправляє його на перевірку до DLP-системи, якщо вона вдала, то відсилає промпт до ChatGPT і зчитує відповідь з нього. При успішному виконанні цих пунктів буде про це сповіщення (Рис.3.22.).

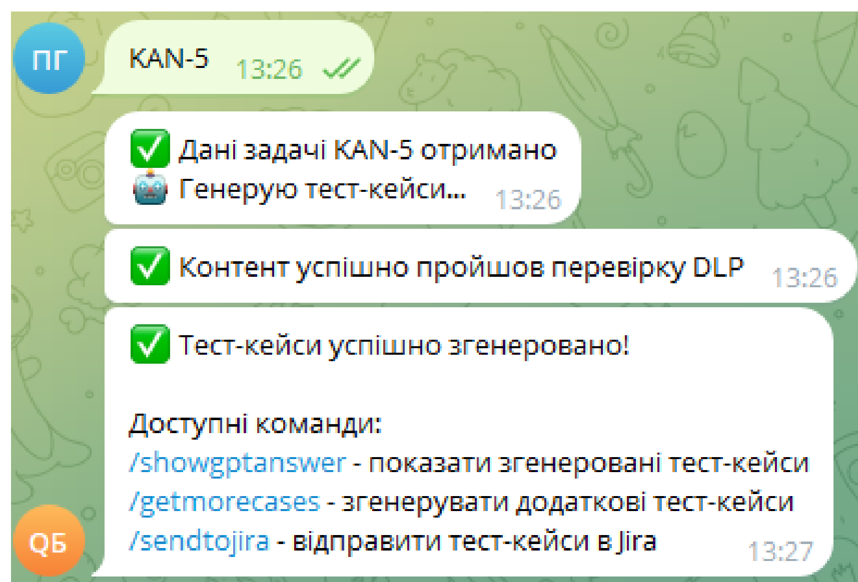

Рис. 3.22. Отримання згенерованих тест-кейсів

Бот виводить сповіщення про успішну генерацію та пропонує наступні дії: показати результат генерації, згенерувати додаткові тест-кейси та записати усі згенеровані тест-кейси у коментарі до задачі Jira. Спочатку використаємо генерацію додаткових кейсів, а потім відправимо їх у Jira (Рис.3.23.).

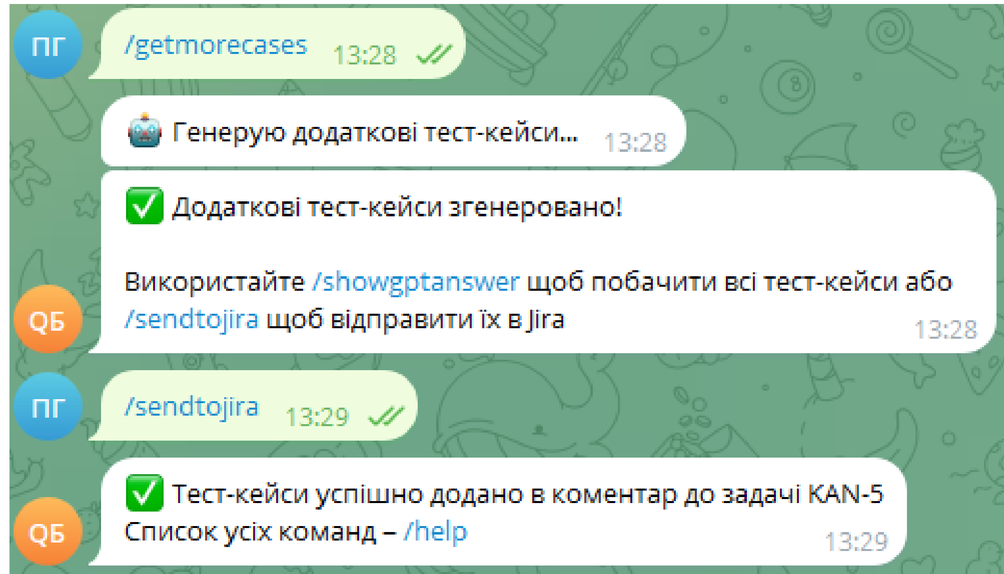

Рис. 3.23. Генерація додаткових тест-кейсів та запис їх у Jira

Після цих дій в коментарях к задачі KAN-5 у Jira з’явиться велика кількість згенерованих тестових кейсів, які відповідають поставленим вимогам (Рис.3.24.).

Add a comment...

Suggest a reply... | Who is working on this...? | Status update...

Pro tip: press M to comment

Paul Golubev
11 hours ago

Ось набір функціональних тестових кейсів, що максимально покривають вимоги, із застосуванням різних технік тест-дизайну (еквівалентне розбиття, граничні значення, позитивні/негативні сценарії, комбінації параметрів, перевірка стійкості, тощо):

—

1. a. i. 1. *Тестовий кейс «Запуск програми з усіма обов'язковими параметрами»*

- *Мета*: Перевірити, що програма успішно запускається з усіма обов'язковими параметрами.
- *Передумова*: Всі вказані файли та каталоги існують і доступні.
- *Кроки*:
  1. Відкрити консоль.
  2. Виконати команду:
  `java -jar AudioCataloger.jar output.html output.csv C:\Music`
- *Очікуваний результат*: Програма запускається, сканує каталог, виводить журнал у консоль, створює файли output.html та output.csv.

—

1. a. i. 2. *Тестовий кейс «Запуск програми з параметром DuplicatesOnly»*

- *Мета*: Перевірити, що програма працює у режимі відображення лише дублікатів.
- *Передумова*: У каталозі є дублікати аудіофайлів.
- *Кроки*:
  1. Виконати команду:
  `java -jar AudioCataloger.jar DuplicatesOnly output.html output.csv C:\Music`
- *Очікуваний результат*: У звіті HTML відображаються лише дублікати, фон дублікатів білий.

—

1. a. i. 3. *Тестовий кейс «Запуск програми без обов'язкового параметра HtmlOutputFileName»*

- *Мета*: Перевірити реакцію програми на відсутність обов'язкового параметра.
- *Передумова*: -
- *Кроки*:
  1. Виконати команду:
  `java -jar AudioCataloger.jar output.csv C:\Music`
- *Очікуваний результат*: Програма завершує роботу, виводить повідомлення про використання (DS-3.1).

—

1. a. i. 4. *Тестовий кейс «Запуск програми з неіснуючим каталогом»*

- *Мета*: Перевірити обробку неіснуючого каталогу.
- *Передумова*: Каталог C:\NoSuchDir не існує.
- *Кроки*:
  1. Виконати команду:
  `java -jar AudioCataloger.jar output.html output.csv C:\NoSuchDir`
- *Очікуваний результат*: Програма завершує роботу, виводить повідомлення:

Рис. 3.24. Результат роботи програми

3.7. Технічна реалізація компонентів програми

Головним модулем програми є main.py (Рис.3.25.), він запускає модуль Telegram-бота chatbot.py (Рис.3.26.-Рис.3.34.). В цьому файлі реалізована логіка роботи бота. Спочатку підключаються необхідні бібліотеки, потім починається обробка з відкриття та зчитування вмісту файлу config.yaml, потім додаються модулі запису журналу, клієнту Jira, зв'язку з ChatGPT та клієнту DLP. Основна взаємодія користувача з telegram ботом реалізована за допомогою регулярних виразів, тобто коли в сповіщені від користувача знаходиться певний вираз активується пов'язана з ним функція. Всього присутні такі функції: start_message – обробка команди /start та показ привітального повідомлення; help_message – обробка команди /help з показом списку команд; get_case_jira – обробка команди /getcasejira, суть якої в отриманні від користувача номеру задачі та передачі його наступній функції; process_jira_ticket – обробка номеру задачі Jira, зчитування вмісту задачі, формування з нього та з сповіщення-вирівнювача промпту, відсилання промпту на перевірку та отримання відповіді, відсилання перевіреного промпту до ChatGPT та збереження його відповіді; show_gpt_answer – обробка команди /showgptanswer та показ користувачеві згенерованої відповіді; get_more_cases – обробка команди /getmorecases, генерація додаткових кейсів та допис їх до вже згенерованих; send_to_jira – обробка команди /sentojira і додання згенерованих тест-кейсів в коментарі до задачі в Jira.

Взаємодія з Jira виконується модулем jira_client.py (Рис.3.35.-Рис.3.37.). В цьому файлі спочатку йде підключення необхідних бібліотек і модулю записів журналу, потім відкриття та зчитування config.yaml, а саме параметрів jira_link, jira_email та jira_token. Далі йде ініціалізація клієнту Jira за зчитаними параметрами. Тут є три функції: get_issue_description, в якій береться номер задачі Jira та повертається вміст опису; get_issue_summary, де по номеру задачі повертається її заголовок; send_comment, береться номер задачі і згенеровані тест-кейси та записуються у коментар до задачі у Jira,

повертає True коли операція успішна і False коли ні.

Запис журналу у всіх модулях реалізовано завдяки модулю log.py (Рис.3.38.), у ньому спочатку завантажуються необхідні бібліотеки, а саме yaml – для відкриття файлу config.yaml і зчитування шляху до файлу журналу та logging – для запису подій до нього.

Зв’язок з ChatGPT здійснюється завдяки файлу gpt.py (Рис.3.39., Рис.3.40.). Завантажуються необхідні бібліотеки та модуль запису журналу. Зчитується OpenAI Api Key з config.yaml та створюється точка доступу до ChatGPT. Присутня одна функція – ask_gpt, яка отримує на вхід готовий промпт, відсилає його до ChatGPT, і повертає відповідь від нього.

Відправка промпту на перевірку виконується завдяки файлу dlp_client.py (Рис.3.41.-Рис.3.43.). Перше, що в ньому виконується – це завантаження необхідних бібліотек: requests для відправки http-запитів; yaml для зчитування dlp_url та dlp_token з config.yaml; модуль запису журналу. Ініціалізується клієнт для взаємодії з DLP-системою за зчитаними параметрами. Має одну функцію – check_content, яка приймає на вхід скомпонований промпт, передає його до DLP-системи та в залежності від її відповіді повертає True, якщо перевірка відала та False, якщо ні.

```
1  from chatbot import bot
2
3  def main():
4      bot.infinity_polling()
5
6  if __name__ == '__main__':
7      main()
```

Рис. 3.25. Файл main.py

```
import telebot
import yaml
from jira_client import JiraClient
#from dlp_client import DLPClient
from gpt import ask_gpt
import log

logger = log.logger()
#dlp_client = DLPClient()

with open("config.yaml", "r", encoding="utf8") as f:
    config = yaml.load(f, Loader=yaml.FullLoader)

bot = telebot.TeleBot(config['bot_id'])
jira_client = JiraClient()

# Словник для зберігання згенерованих відповідей
generated_responses = {}

@bot.message_handler(commands=['start'])
def start_message(message):
    """Обробка команди start"""
    bot.send_message(message.chat.id, config['bot_messages']['welcome_message'])

```

Рис. 3.26. Файл chatbot.py частина 1

```

@bot.message_handler(commands=['help'])
def help_message(message):
    """Обробка команди help"""
    help_text = (
        "/getcasejira - генерація тестового кейса в Jira за переданим ідентифікатором сущності.\n"
        "/showgptanswer - показати останню згенеровану відповідь.\n"
        "/getmorecases - згенерувати додаткові тест-кейси.\n"
        "/sendtojira - записати згенеровані тест-кейси до Jira.\n"
        "/help - показати це повідомлення"
    )
    bot.send_message(message.chat.id, help_text)

@bot.message_handler(commands=['getcasejira'])
def get_case_jira(message):
    """Обробка команди getcasejira"""
    msg = bot.send_message(message.chat.id,  text: "✏️ Введіть номер задачі в Jira:")
    bot.register_next_step_handler(msg, process_jira_ticket)

def process_jira_ticket(message):  1 usage
    """Обробка номера тікета та генерація тест-кейсів"""
    ticket = message.text.strip().upper()
    processing_msg = bot.send_message(message.chat.id,  text: f"🔍 Отримую дані задачі {ticket}...")

```

Рис. 3.27. Файл chatbot.py частина 2

```

    try:
        # Отримання даних з Jira
        issue_description = jira_client.get_issue_description(ticket)
        if not issue_description:
            bot.edit_message_text(
                 text: "❌ Не вдалося отримати дані задачі. Перевірте номер задачі та спробуйте ще раз.",
                message.chat.id,
                processing_msg.message_id
            )
            return

        issue_summary = jira_client.get_issue_summary(ticket)
        if not issue_summary:
            bot.edit_message_text(
                 text: "❌ Не вдалося отримати ім'я задачі. Перевірте номер задачі та спробуйте ще раз.",
                message.chat.id,
                processing_msg.message_id
            )
            return

        bot.edit_message_text(
             text: f"✅ Дані задачі {ticket} отримано\n🤖 Генерую тест-кейси...",
            message.chat.id,
            processing_msg.message_id
        )

```

Рис. 3.28. Файл chatbot.py частина 3

```
73
74          # Формування промпта для GPT
75          prompt = (
76              f"{config['start_text']['cases']}\n"
77              f"Назва задачі: {issue_summary}\n"
78              f"Опис задачі: {issue_description}"
79          )
80
81          # Перевірка через DLP
82          #is_allowed, reason = dlp_client.check_content(prompt)
83          is_allowed, reason = True, ''
84
85          if not is_allowed:
86              bot.edit_message_text(
87                  text: f"❌ Контент не пройшов перевірку DLP:\n{reason}",
88                  message.chat.id,
89                  processing_msg.message_id
90              )
91              logger.warning(f"DLP check failed for ticket {ticket}: {reason}")
92              return
93
94          # Сповіщення про успішну перевірку DLP
95          bot.send_message(
96              message.chat.id,
97              text: "✅ Контент успішно пройшов перевірку DLP")
98          logger.info(f"DLP check succeeded for ticket {ticket}")
99
100         # Генерація тест-кейсів
101         test_cases = ask_gpt(prompt)
102
```

Рис. 3.29. Файл chatbot.py частина 4

```
102
103         if not test_cases or test_cases.startswith('На жаль'):
104             bot.send_message(
105                 message.chat.id,
106                 text: "❌ Помилка при генерації тест-кейсів. Спробуйте пізніше."
107             )
108             return
109
110         # Зберігаємо згенеровану відповідь та тікет
111         generated_responses[message.chat.id] = {
112             'test_cases': test_cases,
113             'ticket': ticket,
114             'description': issue_description,
115             'summary': issue_summary
116         }
117
```

Рис. 3.30. Файл chatbot.py частина 5

```
117
118             bot.send_message(
119                 message.chat.id,
120                 text: "✅ Тест-кейси успішно згенеровано!\n\n"
121                 "Доступні команди:\n"
122                 "/showgptanswer - показати згенеровані тест-кейси\n"
123                 "/getmorecases - згенерувати додаткові тест-кейси\n"
124                 "/sendtojira - відправити тест-кейси в Jira"
125             )
126
127         except Exception as e:
128             logger.error(f"Error processing ticket {ticket}: {e}")
129             bot.send_message(
130                 message.chat.id,
131                 text: "❌ Виникла помилка при обробці запиту. Спробуйте пізніше."
132             )
133
134 @bot.message_handler(commands=['showgptanswer'])
135 def show_gpt_answer(message):
136     """Показує останню згенеровану відповідь"""
137     if message.chat.id in generated_responses and generated_responses[message.chat.id].get('test_cases'):
138         bot.send_message(message.chat.id, generated_responses[message.chat.id]['test_cases'])
139     else:
140         bot.send_message(message.chat.id,
141                          text: "❌ Немає збережених згенерованих тест-кейсів. Спочатку виконайте /getcasejira")
```

Рис. 3.31. Файл chatbot.py частина 6

```
142
143 @bot.message_handler(commands=['getmorecases'])
144 def get_more_cases(message):
145     """Генерація додаткових тест-кейсів"""
146     if message.chat.id not in generated_responses:
147         bot.send_message(message.chat.id,
148                          text: "❌ Спочатку виконайте /getcasejira для генерації початкових тест-кейсів")
149         return
150
151     saved_data = generated_responses[message.chat.id]
152     prompt = (
153         f"{config['start_text']['more_cases']}\n"
154         f"Назва задачі: {saved_data['summary']}\n"
155         f"Опис задачі: {saved_data['description']}"
156     )
157
158     processing_msg = bot.send_message(message.chat.id,  text: "🐱 Генерую додаткові тест-кейси...")
159
160     try:
161         additional_cases = ask_gpt(prompt)
162         if not additional_cases or additional_cases.startswith('На жаль'):
163             bot.send_message(
164                 message.chat.id,
165                 text: "❌ Помилка при генерації додаткових тест-кейсів. Спробуйте пізніше."
166             )
167             return
168
```

Рис. 3.32. Файл chatbot.py частина 7

```
168
169         # Додаємо нові тест-кейси до існуючих
170         saved_data['test_cases'] += "\n\nДодаткові тест-кейси:\n" + additional_cases
171         bot.send_message(
172             message.chat.id,
173             text: "✅ Додаткові тест-кейси згенеровано!\n\n"
174             "Використайте /showgptanswer щоб побачити всі тест-кейси або /sendtojira щоб відправити їх в Jira"
175         )
176
177     except Exception as e:
178         logger.error(f"Error generating additional cases: {e}")
179         bot.send_message(
180             message.chat.id,
181             text: "❌ Виникла помилка при генерації додаткових тест-кейсів. Спробуйте пізніше."
182         )
183
```

Рис. 3.33. Файл chatbot.py частина 8

```
183
184 @bot.message_handler(commands=['sendtojira'])
185 def send_to_jira(message):
186     """Відправка згенерованих тест-кейсів в Jira"""
187     if message.chat.id not in generated_responses:
188         bot.send_message(message.chat.id,  text: "❌ Немає збережених тест-кейсів. Спочатку виконайте /getcasejira")
189         return
190
191     saved_data = generated_responses[message.chat.id]
192     try:
193         if jira_client.send_comment(saved_data['ticket'], saved_data['test_cases']):
194             bot.send_message(
195                 message.chat.id,
196                 text: f"✅ Тест-кейси успішно додано в коментар до задачі {saved_data['ticket']}"
197                 f"\nСписок усіх команд - /help"
198             )
199         else:
200             bot.send_message(
201                 message.chat.id,
202                 text: f"⚠ Виникла помилка при додаванні коментаря до задачі {saved_data['ticket']}"
203             )
204     except Exception as e:
205         logger.error(f"Error sending to Jira: {e}")
206         bot.send_message(
207             message.chat.id,
208             text: "❌ Виникла помилка при відправці в Jira. Спробуйте пізніше."
209         )
210
```

Рис. 3.34. Файл chatbot.py частина 9

```python
from jira import JIRA
import yaml
import log

logger = log.logger()

with open("config.yaml", "r", encoding="utf8") as f:
    config = yaml.load(f, Loader=yaml.FullLoader)

class JiraClient:  2 usages
    def __init__(self):
        """
        Ініціалізація клієнта Jira з конфігураційного файлу
        """
        try:
            jira_options = {"server": config['links']['jira_link']}
            self.jira = JIRA(
                options=jira_options,
                basic_auth=(config['links']['jira_email'], config['links']['jira_token'])
            )
            logger.info("Jira client initialized successfully")
        except Exception as e:
            logger.error(f"Failed to initialize Jira client: {e}")
            raise

```

Рис. 3.35. Файл jira_client.py частина 1

```python

    def get_issue_description(self, ticket: str) -> str:  1 usage
        """
        Отримання опису задачі
        :param ticket: номер задачі
        :return: опис задачі
        """
        try:
            issue = self.jira.issue(ticket)
            issue_description = issue.fields.description
            logger.info(f"Successfully retrieved description for {ticket}")
            return issue_description or ''
        except Exception as e:
            logger.error(f"Failed to get description for {ticket}: {e}")
            return ''

    def get_issue_summary(self, ticket: str) -> str:  1 usage
        """
        Отримання заголовку задачі
        :param ticket: номер задачі
        :return: заголовок задачі
        """
        try:
            issue = self.jira.issue(ticket)
            issue_summary = issue.fields.summary
            logger.info(f"Successfully retrieved summary for {ticket}")
            return issue_summary or ''
        except Exception as e:
            logger.error(f"Failed to get summary for {ticket}: {e}")
            return ''

```

Рис. 3.36. Файл jira_client.py частина 2

```python

    def send_comment(self, ticket: str, test_cases: str) -> bool:  1 usage
        """
        Додавання коментаря з тест-кейсами
        :param ticket: номер задачі
        :param test_cases: текст тест-кейсів
        :return: True якщо успішно, False якщо помилка
        """
        try:
            issue = self.jira.issue(ticket)
            self.jira.add_comment(issue, test_cases)
            issue.update(fields={"labels": ["ctt_gpt"]})
            logger.info(f"Successfully added comment to {ticket}")
            return True
        except Exception as e:
            logger.error(f"Failed to add comment to {ticket}: {e}")
            return False

```

Рис. 3.37. Файл jira_client.py частина 3

```python
import logging
import yaml

with open("config.yaml", "r", encoding="utf8") as f:
    config = yaml.load(f, Loader=yaml.FullLoader)

def logger():
    """Налаштування логування"""
    logging.basicConfig(filename=config['log_path'], level=logging.INFO)
    return logging.getLogger()
```

Рис. 3.38. Файл log.py

```python
from openai import OpenAI
import yaml
import log

logger = log.logger()

# Завантаження конфігурації
with open("config.yaml", "r", encoding="utf8") as f:
    config = yaml.load(f, Loader=yaml.FullLoader)

# Ініціалізація клієнта OpenAI
client = OpenAI(api_key=config['links']['ChatGPT_API'])

```

Рис. 3.39. Файл gpt.py частина 1

```python

def ask_gpt(prompt: str) -> str:
    """
    Генерація відповіді через OpenAI ChatGPT
    :param prompt: текст запиту
    :return: згенерована відповідь
    """
    try:
        response = client.chat.completions.create(
            model="gpt-4-1106-preview",
            messages=[
                {"role": "system",
                 "content": "Ви - досвідчений QA інженер, який спеціалізується на створенні тестових кейсів."},
                {"role": "user", "content": prompt}
            ],
            temperature=0.7,
            max_tokens=4000,
            top_p=1.0,
            frequency_penalty=0.0,
            presence_penalty=0.0
        )
        logger.info("Successfully generated response from GPT")
        return response.choices[0].message.content

    except Exception as e:
        error_msg = f"Error in GPT request: {str(e)}"
        logger.error(error_msg)
        return 'На жаль виникли проблеми з GPT, спробуйте пізніше'
```

Рис. 3.40. Файл gpt.py частина 2

```python
import requests
import yaml
import log

logger = log.logger()

with open("config.yaml", "r", encoding="utf8") as f:
    config = yaml.load(f, Loader=yaml.FullLoader)

class DLPClient:
    def __init__(self):
        """Ініціалізація клієнта DLP"""
        self.dlp_url = config['links']['dlp_url']
        self.dlp_token = config['links']['dlp_token']

```

Рис. 3.41. Файл dlp_client.py частина 1

```
15
16  def check_content(self, content: str) -> tuple[bool, str]:
17      """
18      Перевірка контенту через DLP-систему
19
20      :param content: текст для перевірки
21      :return: tuple(дозволено/заборонено, причина_заборони)
22      """
23      try:
24          headers = {
25              'Authorization': f'Bearer {self.dlp_token}',
26              'Content-Type': 'application/json'
27          }
28
29          data = {
30              'content': content,
31              'source': 'qa_bot',
32              'user_id': 'qa_bot_system'
33          }
34
35          response = requests.post(
36              url: f"{self.dlp_url}/api/v1/check",
37              headers=headers,
38              json=data
39          )
40
```

Рис. 3.42. Файл dlp_client.py частина 2

```
40
41          if response.status_code == 200:
42              result = response.json()
43              return result.get('allowed', False), result.get('reason', '')
44
45          logger.error(f"DLP check failed with status {response.status_code}")
46          return False, f"Помилка перевірки DLP: {response.status_code}"
47
48      except Exception as e:
49          logger.error(f"Error during DLP check: {e}")
50          return False, f"Помилка перевірки DLP: {str(e)}"
```

Рис. 3.43. Файл dlp_client.py частина 3

РОЗДІЛ 4
АНАЛІЗ ЕФЕКТИВНОСТІ РОЗРОБЛЕНОГО РІШЕННЯ

4.1. Аналіз ефективності роботи бота

Для перевірки розробленого рішення у Jira був створений проект «Аудіокаталогізатор», з задачею під номером KAN-5 «Створити тест кейси за вимогами» і списком вимог для тестування у описі. Повний список вимог тестування представлені на Рис.4.1. та Рис.4.2.

1. Вимоги користувача
   - a. UR-1: Запуск та зупинка програми.
     - i. UR-1.1: Запуск програми має здійснюватися за допомогою наступної консольної команди: “java -jar AudioCataloger.jar [DuplicatesOnly] HtmlOutputFileName «CsvOutputFileName StartingDirectory1 [... StartingDirectoryN]» (див. опис параметрів у DS - 2.1 , повідомлення про помилки у випадку неправильної конфігурації див. у DS - 2.2 , DS - 2.3 та DS - 2.4).
     - ii. UR-1.2: Зупинку (вимкнення) програми слід виконувати шляхом застосування Ctrl+C до вікна консолі, в якому знаходиться запущена програма.
   - b. UR- 2 : Конфігурація застосунку.
     - i. UR-2.1: Єдине налаштування доступне через параметри командного рядка (див. DS- 2).
     - ii. UR-2.2: Цільове кодування для вихідних текстових повідомлень – UTF8.
   - c. UR-3: Журнал програми.
     - i. UR-3.1: Програма повинна виводити свій журнал на консоль (див. DS –4).
     - ii. UR-3.2: Вміст та формат журналу описано в DS - 4.2 та DS- 4.3.
2. Бізнес-правила
   - a. BR-1: Підтримувані формати : mp3, flac , wav, ogg, wma .
   - b. BR-2: Вихідні формати – HTML та CSV.
   - c. BR-3: Тільки в дублікатах У цьому режимі колір фону для дублікатів у HTML-виводі має бути білим. У звичайному режимі фон для дублікатів має бути червоним.
   - d. BR-4: Будь-яке ім'я каталогу або файлу у виводі консолі має бути повністю кваліфікованим нормалізованим ім'ям.
3. Атрибути якості
   - a. QA-1: Стійкість до вхідних даних
     - i. QA-1.1: Див. BR-1 щодо вимог до форматів вхідних файлів.
     - ii. QA-1.2: Див. DS - 5.2 щодо вимог до розміру вхідного файлу.
     - iii. QA-1.3: Див. DS - 5.3 для отримання детальної інформації про реакцію програми на неправильний формат вхідного файлу.
   - b. QA-2: Обробка винятків: за жодних обставин програма не повинна аварійно завершувати роботу з необробленим винятком. Незалежно від того, наскільки пошкоджений аудіофайл, програма повинна або витягти необхідні дані, або замінити їх на попередньо визначені заглушки у виводі.
   - c. QA-3: Якщо вказано кілька початкових каталогів , програма повинна проаналізувати набір на наявність вкладеності та/або дублювання, щоб просканувати кожен реальний каталог лише один раз.
4. Детальні специфікації
   - a. DS- 2: Параметри командного рядка
     - i. DS-2.1: Під час запуску програма отримує такі параметри командного рядка:
       1. [DuplicatesOnly] – необов’язковий параметр, який вказує, що у виводі мають відображатися лише дублікати аудіофайлів;
       2. HtmlOutputFileName – обов’язковий параметр, вказує на файл для виводу HTML;
       3. CsvOutputFileName – обов’язковий параметр, вказує на файл для виводу CSV;
       4. StartingDirectory1 – обов’язковий параметр, вказує на каталог для сканування;
       5. [... StartingDirectoryN] – необов’язкові параметри, кожен з яких вказує на інший каталог для сканування (див. також QA-3).

Рис. 4.1. Вимоги тестування частина 1

ii. DS-2.2: Якщо якийсь обов'язковий параметр командного рядка пропущено , програма має завершити роботу, відображаючи стандартне повідомлення про використання (див. DS- 3.1).

iii. DS-2.3: Будь-яку кількість параметрів командного рядка після StartingDirectory1 слід інтерпретувати як набір каталогів для сканування (див. також QA-3) .

iv. DS-2.4: Якщо значення будь-якого параметра командного рядка неправильне, програма повинна завершити роботу, відображаючи стандартне повідомлення про використання (див. DS- 3.1) та неправильну назву параметра, значення та відповідне повідомлення про помилку (див. DS- 3.2).

b. DS-3: Повідомлення

i. DS-3.1: Повідомлення про використання: «Використання: java -jar AudioCataloger.jar [DuplicatesOnly] HtmlOutputFileName Ім'яВихідногоФайлаCsvПочатковийКаталог1 [... ПочатковийКаталогN]”.

ii. DS-3.2: Повідомлення про помилки:

1. «Наступний каталог не знайдено або недоступний: {повний шлях}»;
2. «Наступний файл не доступний для запису: {повний шлях}».
3. «Немає даних аудіозаголовка або аудіотегу в: {повний шлях}».

c. DS-4: журнал

i. DS-4.1: Програма повинна відображати свою поточну активність у консолі. Журнали не потрібні.

ii. DS-4.2: Формат журналу консолі залежить від розробників.

iii. DS-4.3: [Необов'язково] Застосунок повинен перераховувати та описувати список заданих параметрів командного рядка в журналі.

d. DS-5: Формат і розмір файлу

i. DS-5.1: Програма повинна обробляти файли в таких форматах: див. BR - 1 .

ii. DS-5.2: Програма повинна обробляти файли розміром до 2 ГБ (включно).

iii. DS-5.3: Якщо виявлено пошкоджений файл або файл з непідтримуваною внутрішньою структурою, програма повинна відобразити повідомлення журналу «Немає даних аудіозаголовка або аудіотегу в: {повний шлях}».

Рис. 4.2. Вимоги тестування частина 2

Далі був запит на створення тест-кейсів до бота з подальшою відправкою номера задачі. Від бота було отримано повідомлення про успішне виконання і потім була дана команда на запис згенерованої відповіді. Як видно на Рис.4.3. між передачею номера задачі та відповіддю про успішну генерацію пройшла приблизно хвилина часу.

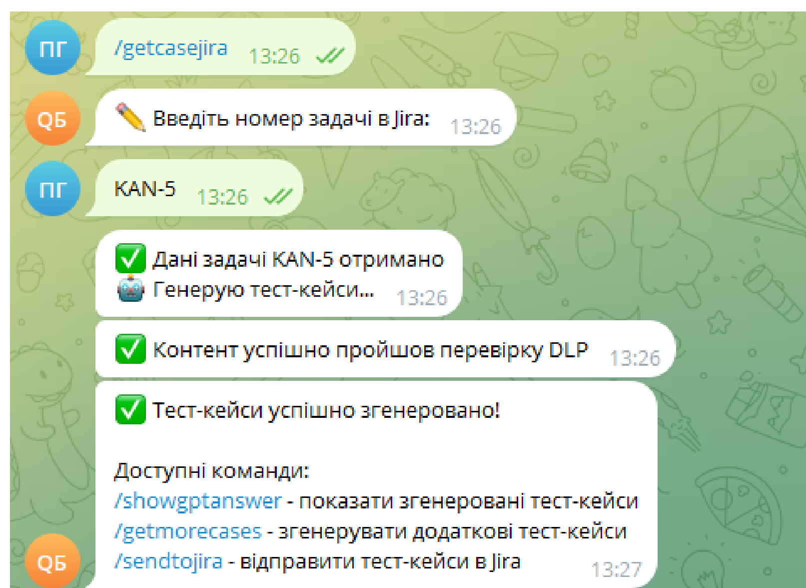

Рис. 4.3. Запит на генерацію та успішна відповідь

За цей проміжок часу було згенеровано 40 тестових кейсів, які потім були записані у коментар до задачі у Jira. Приклади згенерованих тест-кейсів наведено у Рис.4.4., Рис.4.5. та Рис.4.6.

i. 1. *Тестовий кейс «Запуск програми з усіма обов'язковими параметрами»*

- *Мета*: Перевірити, що програма успішно запускається з усіма обов'язковими параметрами.
- *Передумова*: Всі вказані файли та каталоги існують і доступні.
- *Кроки*:
  1. Відкрити консоль.
  2. Виконати команду:
  `java -jar AudioCataloger.jar output.html output.csv C:\Music`
- *Очікуваний результат*: Програма запускається, сканує каталог, виводить журнал у консоль, створює файли output.html та output.csv.

1.

a.

i. 2. *Тестовий кейс «Запуск програми з параметром DuplicatesOnly»*

- *Мета*: Перевірити, що програма працює у режимі відображення лише дублікатів.
- *Передумова*: У каталозі є дублікати аудіофайлів.
- *Кроки*:
  1. Виконати команду:
  `java -jar AudioCataloger.jar DuplicatesOnly output.html output.csv C:\Music`
- *Очікуваний результат*: У звіті HTML відображаються лише дублікати, фон дублікаті білий.

1.

a.

i. 3. *Тестовий кейс «Запуск програми без обов'язкового параметра HtmlOutputFileName»*

- *Мета*: Перевірити реакцію програми на відсутність обов'язкового параметра.
- *Передумова*: -

Рис. 4.4. Приклад 1 згенерованих тест-кейсів

i. 20. *Тестовий кейс «Обробка файлів з відсутніми аудіотегами»*

- *Мета*: Перевірити, що для файлів без аудіотегів виводиться відповідне повідомлення.
- *Передумова*: У каталозі є файл без аудіотегу.
- *Кроки*:
  1. Запустити програму.
- *Очікуваний результат*: У журналі повідомлення:
  «Немає даних аудіозаголовка або аудіотегу в: {повний шлях}»

—

1.

a.

i. 21. *Тестовий кейс «Обробка великої кількості файлів»*

- *Мета*: Перевірити стійкість програми при обробці великої кількості файлів (наприклад, 10 000).
- *Передумова*: У каталозі 10 000 аудіофайлів.
- *Кроки*:
  1. Запустити програму.
- *Очікуваний результат*: Програма обробляє всі файли, не аварійно завершується.

—

1.

a.

i. 22. *Тестовий кейс «Обробка файлів з різними іменами (спецсимволи, пробіли, юнікод)»*

- *Мета*: Перевірити коректність обробки файлів з різними іменами.
- *Передумова*: У каталозі є файли з пробілами, спецсимволами, юнікодом.
- *Кроки*:
  1. Запустити програму.
- *Очікуваний результат*: Всі файли оброблені, імена коректно відображаються у виводі.

Рис. 4.5. Приклад 2 згенерованих тетс-кекйсів

i. 38. *Тестовий кейс «Обробка файлів з іменами, що містять емодзі»*

- *Мета*: Перевірити, що програма коректно обробляє імена з емодзі.
- *Передумова*: У каталозі є файл ".mp3".
- *Кроки*:
  1. Запустити програму.
- *Очікуваний результат*: Файл оброблено, ім'я коректно відображається у виводі.

1.

a.

i. 39. *Тестовий кейс «Обробка файлів з іменами, що містять крапки»*

- *Мета*: Перевірити, що програма коректно обробляє імена з крапками (наприклад, "my.song.mp3").
- *Передумова*: У каталозі є файл "my.song.mp3".
- *Кроки*:
  1. Запустити програму.
- *Очікуваний результат*: Файл оброблено, ім'я коректно відображається у виводі.

1.

a.

i. 40. *Тестовий кейс «Обробка файлів з іменами, що містять декілька розширень»*

- *Мета*: Перевірити, що програма коректно обробляє імена типу "song.mp3.bak".
- *Передумова*: У каталозі є файл "song.mp3.bak".
- *Кроки*:
  1. Запустити програму.
- *Очікуваний результат*: Файл ігнорується як непідтримуваний.

Рис. 4.6. Приклад 3 згенерованих тест-кейсів

Завдяки сповіщенню-вирівнювачу (Рис.4.7.), яке додається до промпту, структура згенерованих тестових сценаріїв залишається правильною та легкою для розуміння.

```
cases: "Розроби максимальну кількість функціональних тестових кейсів з різними варіаціями тестових даних та
 застосуванням технік тест-дизайну. Приклад необхідного формату тест-кейсу: 1. **Тестовий кейс «Назва»**
 - **Мета**: Мета перевірки. - **Передумова**: Передумова. - **Кроки**: - Кроки.
 - **Очікуваний результат**: очікуваний результат. Перевірки повинні повністю покривати вимоги:\n"
```

Рис. 4.7. Сповіщення-вирівнювач для промпту

Також варто зазначити, що ChatGPT чудово розуміє сутність представлених вимог і, як видно в даному випадку, генерує окремий тест для відповідного випадку. Людина напише тестові сценарії, які покривають подібну кількість вимог, приблизно за 30 хвилин, а розроблене рішення за – хвилину-дві. До того ж воно має невеликий розмір та дуже зрозумілий інтерфейс, що в купі дає змогу легко вбудуватися у процеси CI/CD та гнучкі методології розробки.

4.2. Подальші можливості модифікації

Можливостей до подальшого покращення у розробленого рішення багато. Перше, що можна зробити це скомпонувати його у контейнер Docker для більш швидшого і легшого розгортання. Далі можна додати зв’язок з базою даних (наприклад PostgreSQL) для більш об’ємного запису подій та сесій користувачів. Також є можливість розширити місце, звідки програма бере дані і додати, наприклад Confluence, у якого чудова інтеграція з Jira. Чи піти зовсім далеко і замінити Telegram бота на повноцінний Frontend, додати можливість асинхронної роботи з великою кількість сповіщень та зробити повноцінні компоненти ведення журналів і аутентифікації.

Суть та структуру згенерованих сценаріїв також легко можна модифікувати, якщо вписати зміни до сповіщення-вирівнювача, наприклад щоб в одному тест-кейсі перевірялось декілька вимог чи взагалі змінити його сутність та попросити генерувати скорочений опис завдання або малювати ілюстрації до нього. Загалом можна сказати, що розроблене рішення є чудовою платформою для інтеграції помічника на базі штучного інтелекту у величезну кількість процесів розробки.

ВИСНОВКИ

В рамках виконання роботи було досліджено сутність ручного та автоматизованого типів тестування, розібрано популярні види тестування, що найчастіше використовуються. Проаналізовано сучасний стан та інструменти автоматизації в тестуванні програмного забезпечення, а також вплив впровадження штучного інтелекту до тестування. За результатом аналізу зроблено висновок про попит на ботів-помічників на ринку автоматизації тестування, що сильніше доказує актуальність розробки обраного рішення. Розроблено, проаналізовано та детально описано роботу бота-асистента і його компонентів. Досліджено ефективність виконання реальних робочих задач ботом-помічником, з чого робиться висновок, що реалізація обраної ідеї вийшла вдалою та розроблене рішення можна інтегрувати до справжніх робочих процесів. Розробка має потенціал до подальшої модифікації та покращення у вигляді не просто бота-асистента, а платформи для впровадження штучного інтелекту до виконання рутинних процесів усієї команди розробки програмних продуктів.

ПЕРЕЛІК ПОСИЛАНЬ

1. «The Art of Software Testing» by Myers, Glenford J., 1979
2. Стаття про порівняння ручного та автоматизованого тестування [Електронний ресурс] – Режим доступу: https://university.sigma.software/manual-testing-vs-automation-testing/
3. Стаття про порівняння ручного та автоматизованого тестування [Електронний ресурс] – Режим доступу: https://training.qatestlab.com/blog/technical-articles/manual-testing-vs-automation-testing/
4. Стаття про переваги та недоліки автоматизованого тестування [Електронний ресурс] – Режим доступу: https://redstone.agency/blog/perevagi-ta-nedoliki-avtomatyzovanogo-testuvannya/
5. Стаття про типи тестування - https://training.qatestlab.com/blog/technical-articles/review-the-types-of-testing/
6. Стаття про види функціонального та нефункціонального тестування [Електронний ресурс] – Режим доступу: https://dan-it.com.ua/uk/blog/vidy-funkcionalnogo-i-nefunkcionalnogo-testirovanija/
7. Стаття про відмінності між функціональним та нефункціональним видами тестування [Електронний ресурс] – Режим доступу: https://training.qatestlab.com/blog/technical-articles/difference-between-functional-and-non-functional-testing/
8. Звіт про розміри глобального ринку забезпечення якості програмних продуктів компанії Research and Markets на 2024 рік [Електронний ресурс] – Режим доступу: https://www.researchandmarkets.com/report/software-quality-assurance-market#:~:text=The%20Software%20Quality%20Assurance%20Market,future%20of%20the%20SQA%20market.
9. Шістнадцяте видання World Quality Report на 2024-25 роки від Capgemini [Електронний ресурс] – Режим доступу:

https://www.capgemini.com/wp-content/uploads/2024/10/WQR-24-MAIN-REPORT-CG.pdf

10. Стаття про сучасні тенденції QA сфери [Електронний ресурс] – Режим доступу: https://training.qatestlab.com/blog/helpful-materials/current-trends-in-the-qa-field/

11. Стаття про тренди в автоматизованому тестуванні [Електронний ресурс] – Режим доступу: https://campus.epam.ua/ua/blog/615

12. Стаття про вибір інструменту автоматизації [Електронний ресурс] – Режим доступу: https://qalight.ua/baza-znaniy/yak-obrati-instrument-avtomatizaczii/

13. Стаття про те як, обрати інструмент для автоматизованого тестування [Електронний ресурс] – Режим доступу: https://www.practitest.com/resource-center/blog/how-to-choose-the-right-automation-tool/

14. Стаття про вибір та порівняння інструменту автоматизованого тестування [Електронний ресурс] – Режим доступу: https://robotdreams.cc/uk/blog/630-10-best-tools-for-qa-automation-in-2025

15. Стаття про порівняння інструментів автоматизації тестування [Електронний ресурс] – Режим доступу: https://a4.com.ua/najkrashhi-instrumenti-avtomatizacii-testuvannja/

16. Стаття про вибір та порівняння інструментів автоматизації тестування [Електронний ресурс] – Режим доступу: https://www.leapwork.com/blog/top-20-test-automation-tools

17. Дослідження компанії Copilot впливу використання ШІ при розробці ПЗ [Електронний ресурс] – Режим доступу: https://economics.mit.edu/sites/default/files/inline-files/draft_copilot_experiments.pdf

18. Дослідження компанії Metr впливу використання ШІ на роботу досвідчених розробників [Електронний ресурс] – Режим доступу: https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/

19. Стаття про те, як штучний інтелект трансформує тестування

[Електронний ресурс] – Режим доступу: https://proit.com.ua/news/yak-shtuchnyj-intelekt-transformuye-testuvannya/

20. Стаття про те, як штучний інтелект впливає на розробку ПЗ [Електронний ресурс] – Режим доступу: https://wezom.com.ua/ua/blog/yak-shi-vplivaje-na-rozrobku-programnih-produktiv

21. Стаття про штучний інтелект в тестуванні ПЗ [Електронний ресурс] – Режим доступу: https://visuresolutions.com/alm-guide/ai-in-software-testing/

22. Стаття про вплив впровадження штучного інтелекту до тестування ПЗ [Електронний ресурс] – Режим доступу: https://www.nayka.com.ua/index.php/ee/article/view/4247/4282

23. Документація та опис продукту Applitools з офіційного сайту [Електронний ресурс] – Режим доступу: https://applitools.com/docs/

24. Офіційна документація Python 3.10 [Електронний ресурс] – Режим доступу: https://docs.python.org/3.10/

25. Офіційна документація PyTelegramBotAPI [Електронний ресурс] – Режим доступу: https://pytba.readthedocs.io/en/latest/

26. Офіційна документація PyYAML [Електронний ресурс] – Режим доступу: https://pyyaml.org/wiki/PyYAMLDocumentation

27. Офіційна документація Requests [Електронний ресурс] – Режим доступу: https://requests.readthedocs.io/en/latest/

28. Офіційна документація Logging [Електронний ресурс] – Режим доступу: https://docs.python.org/3.10/library/logging.html

29. Офіційна документація Python Jira [Електронний ресурс] – Режим доступу: https://jira.readthedocs.io/

30. Офіційна документація OpenaAI Python [Електронний ресурс] – Режим доступу: https://platform.openai.com/docs/api-reference/introduction

31. Офіційна документація для BotFather від Telegram [Електронний ресурс] – Режим доступу: https://core.telegram.org/bots

32. Офіційна документація для Jira [Електронний ресурс] – Режим доступу: https://www.atlassian.com/software/jira/guides/getting-started/introduction#what-

is-jira-software

33. Сайт OpenAI [Електронний ресурс] – Режим доступу: https://platform.openai.com

34. Управління API ключами на сайті OpenAI [Електронний ресурс] – Режим доступу: https://platform.openai.com/settings/organization/api-keys

35. Офіційна документація OpenAI [Електронний ресурс] – Режим доступу: https://platform.openai.com/docs/overview