БАКАЛАВРСЬКА РОБОТА

БР. ІП - 93.00.00.000 ПЗ

Група ІП-22-4

Голубка Олександр

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Голубка Олександр Юрійович

(прізвище, ім'я, по-батькові)

УДК 004.942

(індекс)

БАКАЛАВРСЬКА РОБОТА

Інтелектуальний чат-бот, як ВЕБ-сервіс

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня Голубка О.Ю.

(підпис, ініціали та прізвище здобувача)

Науковий керівник Саманів Любов Василівна, асистент

(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою ІПЗ
доц. В.В. Бандура
“ ” 2026 р.

ЗАВДАННЯ

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Голубці Олександру Юрійовичу
(прізвище, ім’я, по-батькові)

1. Тема проєкту (роботи) «Інтелектуальний чат-бот, як ВЕБ-сервіс»
Керівник проєкту (роботи) Саманів Любов Василівна, асистент
Затверджені наказом закладу вищої освіти від “ 21 ” квітня 2026 р. 179/7

2. Строк подання студентом проєкту (роботи) 10 червня 2026 р.

3. Вихідні дані до проєкту (роботи) Офіційна документація Telegram Bot API,документація фреймворку Aiogram 3.x, відкриті API фріланс-платформ (Remotive, Freelancer.com), технічні вимоги до системи SmartHunt.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Огляд ринку фріланс-платформ та теоретичні основи проєктування чат-ботів.
2. Аналіз вимог і постановка задачі
3. Проєктування архітектури системи та бази даних.
4. Програмна реалізація проєкту (агрегація, парсинг, лексична фільтрація на базі словника синонімів-фільтрація).
5. Тестування, безпека та розгортання системи.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Схема архітектури даних таблиці users (ER-діаграма).(рис. 2.1 ст. 24)
2. UML Діаграма прецедентів використання системи (Use Case Diagram). (рис. 2.2 ст. 25)
3. Діаграма станів (FSM) налаштування фільтрів. (рис. 2.3 ст. 26)
4. Блок-схема (Activity Diagram) алгоритму парсингу та NLP-фільтрація. (рис. 2.4 ст. 28)
5. UML-діаграма послідовності процесу обробки пошукового запиту. (рис. 2.5 ст. 30)

1. Консультанти розділів проєкту (роботи)

Розділ	***Консультант***	***Підпис, дата***	
		Завдання видав	***Завдання прийняв***

2. Дата видачі завдання _____

Керівник _____ _____
(підпис) (розшифровка підпису)

Завдання прийняв до виконання _____ _____
(підпис) (розшифровка підпису)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	***Назва етапів дипломного проєкту (роботи)***	***Строк виконання етапів проєкту***	***Примітка***
1	Визначення та обґрунтування теми роботи	17.01.2026	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	23.02.2026	виконано
3	Побудова моделі або алгоритму власного рішення	10.03.2026	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	27.04.2026	виконано
5	Оформлення пояснювальної записки бакалаврської роботи завідувачем кафедри	10.06.2026	виконано

Студент-дипломник _____ Голубка О.Ю.
(підпис) (розшифровка підпису)

Керівник роботи _____ Саманів Л. В.
(підпис) (розшифровка підпису)

АНОТАЦІЯ

Бакалаврська робота містить 68 сторінок, 11 рисунків, список використаних джерел із 30 найменувань, 2 додатки.

Мета роботи: розробка інтелектуального чат-бота, що функціонує як веб-сервіс для агрегації фріланс-проєктів із використанням Python-стеку.

Об'єкт дослідження: процеси агрегації, лексичного аналізу та фільтрації потоку даних про фріланс-замовлення у реальному часі.

Предмет дослідження: методи та засоби розробки асинхронного Telegram-бота як веб-сервісу для збору даних з API, управління профілями користувачів та відмовостійкості.

Результати дослідження: розроблено Telegram-застосунок на базі фреймворку Aiogram для пошуку проєктів, управління фільтрами навичок, перевірки збігів через NLP-алгоритми, роботи із зовнішніми API бірж та збереження даних користувачів у БД SQLite.

У першому розділі розглянуто існуючі рішення та теоретичні основи розробки.

У другому розділі визначено вимоги й основні сценарії використання.

У третьому розділі обґрунтовано вибір технологій та архітектурних моделей.

У четвертому розділі описано реалізацію системи та фільтрацію.

У п'ятому розділі проведено тестування основних функцій.

Висновок: створено автономний інтелектуальний чат-бот (веб-сервіс), який автоматизує основні процеси роботи фрілансера з пошуку замовлень і зменшує витрати часу на ручний моніторинг бірж.

КЛЮЧОВІ СЛОВА: ІНТЕЛЕКТУАЛЬНИЙ ЧАТ-БОТ, ВЕБ-СЕРВІС, PYTHON, AIOGRAM, SQLITE, АГРЕГАЦІЯ ДАНИХ, ПАРСИНГ, ФІЛЬТРАЦІЯ, АВТОМАТИЗАЦІЯ.

ANNOTATION

The bachelor's thesis contains 68 pages, 11 figures, a list of references with 30 titles, and 2 appendices.

Objective: development of an intellectual chatbot operating as a web service for freelance project aggregation using a Python-based stack.

Research object: processes of real-time aggregation, lexical analysis, and filtering of freelance job data streams.

Subject of research: methods and tools for developing an asynchronous Telegram bot as a web service for API data collection, user profile management, and fault-tolerant filtering.

Research results: a Telegram application based on the Aiogram framework was developed for project search, skill filter management, match verification via NLP algorithms, external API integration, and user data storage in a SQLite database.

The first section considers existing solutions and the theoretical basis of development.

The second section defines the requirements and main use cases.

The third section justifies the choice of technologies, architectural models, and tools.

The fourth section describes the system implementation, parsing, and filtering algorithms.

The fifth section deals with the testing of main functions.

Conclusion: an autonomous intellectual chatbot (web service) was created that automates the main processes of a freelancer's job search and reduces the time spent on manual monitoring of freelance platforms.

KEYWORDS: INTELLECTUAL CHATBOT, WEB SERVICE, PYTHON, AIOGRAM, SQLITE, DATA AGGREGATION, PARSING, FILTERING, AUTOMATION.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API - Application Programming Interface (Інтерфейс програмування застосунків)

FSM - Finite State Machine (Скінченний автомат / Машина станів)

NLP- Natural Language Processing (Обробка природної мови)

UI/UX - User Interface / User Experience (Користувацький інтерфейс / Досвід)

JSON - JavaScript Object Notation (Формат обміну даними)

MVP - Minimum Viable Product (Мінімально життєздатний продукт)

SDLC - Software Development Life Cycle (Життєвий цикл розробки ПЗ)

ORM - Object-Relational Mapping (Об'єктно-реляційне відображення)

ВСТУП

Розвиток економіки вільного заробітку (Gig economy) зробив фріланс основним джерелом доходу для багатьох IT-спеціалістів. Висока конкуренція та швидкоплинність пропозицій на біржах (Upwork, Freelancer тощо) вимагають постійного ручного моніторингу. Це призводить до втрати робочого часу та ризику пропустити вигідні замовлення через великий обсяг нерелевантного інформаційного «шуму».

Актуальність теми зумовлена потребою у створенні персоналізованого інструменту для автоматизації процесу моніторингу нових фріланс-проєктів. Така система повинна забезпечувати безперервний збір даних, гнучку фільтрацію за ключовими навичками та миттєве сповіщення користувача. Оскільки система орієнтована на оперативність, найбільш зручним форматом її реалізації є інтелектуальний чат-бот у месенджері, що працює за принципом push-повідомлень.

Метою роботи є розробка автоматизованої інформаційної системи у вигляді інтелектуального чат-бота (веб-сервісу) для агрегації фріланс-проєктів. Рішення передбачає використання Python-стеку, асинхронної архітектури (Aiogram) та СУБД SQLite для забезпечення повного циклу: від парсингу REST API до семантичної обробки та доставки даних користувачу.

Завданнями дослідження є: проаналізувати предметну область; визначити основні вимоги до програмного забезпечення; спроєктувати архітектуру веб-сервісу та діалоговий інтерфейс (на базі машини станів FSM); розробити модуль взаємодії з відкритими API зовнішніх фріланс-платформ; реалізувати алгоритм лексичної фільтрації із застосуванням словника синонімів; забезпечити відмовостійкість системи за допомогою механізму «м'якої деградації» (Graceful Degradation); розгорнути готовий продукт на хмарному сервері.

Об’єктом дослідження є процеси автоматизованого збору, обробки та лексичної фільтрації потоку даних про фріланс-замовлення у реальному часі.

Предметом дослідження є методи, алгоритми та програмні засоби розробки асинхронних клієнт-серверних додатків (веб-сервісів), технології взаємодії з REST API та обробки природної мови.

Методи дослідження включають аналіз предметної області, об'єктно-орієнтоване проєктування, асинхронне програмування мовою Python, роботу з реляційними базами даних, інтеграцію мережевих протоколів та тестування системи за принципом «чорної скриньки».

Практичне значення роботи полягає у створенні прикладного програмного продукту (MVP), який може бути використаний спеціалістами для спрощення щоденного пошуку замовлень. Розроблена система звільняє фрілансера від рутинного серфінгу вебсайтів, гарантує безперебійну агрегацію та миттєво доставляє відфільтровані релевантні вакансії.

Розроблене програмне забезпечення може бути розширене у майбутньому шляхом підключення нових джерел даних (Fiverr, Upwork), інтеграції платіжних систем для монетизації підписки, а також залучення великих мовних моделей (LLM) для глибшого семантичного аналізу. Таким чином, створений додаток є надійною основою для подальшого розвитку.

Бакалаврська робота містить 68 сторінок, 11 рисунків, 5 розділів, список використаних джерел із 30 найменувань, 2 додатки.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИБІР ПЛАТФОРМИ

1.1. Аналіз сучасного стану ринку фрілансу та визначення цільової аудиторії системи

Першим етапом життєвого циклу розробки програмного забезпечення (SDLC) є комплексний аналіз предметної області (Domain Analysis) та збір вимог [6]. Дослідження світового ринку праці свідчить про безпрецедентну зміну парадигми працевлаштування в IT-сфері. Перехід до формату «Gig economy» (економіки вільного заробітку), каталізований наслідками глобальної цифровізації та постпандемічними трендами, перетворив фріланс із тимчасового підробітку на повноцінну та домінуючу модель побудови кар'єри для мільйонів спеціалістів. Цей процес супроводжується децентралізацією виробничих потужностей та активним впровадженням концепції розподілених команд, що дозволяє компаніям оперативно залучати інженерні таланти без прив'язки до їхньої географічної локації та мінімізувати операційні витрати на утримання офісної інфраструктури.

За даними міжнародних аналітичних агенцій, понад 45% працівників інтелектуальної сфери хоча б частково залучені до проєктної роботи поза штатом. Спеціалісти в галузі інженерії програмного забезпечення (Software Engineering), вебдизайну, кібербезпеки, DevOps та аналізу даних формують ядро цієї платформи. Проте, пропорційно до зростання кількості пропозицій від замовників (Job Posts), експоненційно зростає і рівень конкуренції між виконавцями (Freelancers). Утворення єдиного глобального ринку праці призвело до того, що на одне замовлення одночасно можуть претендувати розробники з абсолютно різних економічних регіонів та часових поясів, що суттєво демпінгує вартість послуг та ущільнює конкурентне середовище.

У сучасних висококонкурентних реаліях успішність фрілансера визначається не лише рівнем його технічних компетенцій (Hard skills) та портфоліо, але й швидкістю доступу до інформації. Більшість

високооплачуваних короткострокових контрактів («тасок») закриваються замовниками протягом перших годин, а іноді й хвилин після їх публікації на біржі. Відповідно, процес безперервного моніторингу нових проєктів на десятках різних майданчиків стає критично важливою, але вкрай неефективною тратою цінного робочого часу спеціаліста.

Для правильного проєктування архітектури та інтерфейсу системи агрегації «SmartHunt» необхідно чітко типізувати кінцевих користувачів. У ході дослідження предметної області було виділено та деталізовано два основні полярні сегменти цільової аудиторії (User Personas):

- **Сегмент А**: Розробники-початківці (Junior Developers / Students). Ця група користувачів шукає перші замовлення для напрацювання комерційного досвіду, формування стартового портфоліо та відточування практичних навичок. Їхній інтерес сфокусований на невеликих за обсягом "разових тасках" з фіксованим бюджетом (Fixed-price tasks). Для цієї категорії критично важливою є саме швидкість виявлення замовлення, оскільки на прості завдання (наприклад, верстка Landing Page, написання базових скриптів чи парсерів) протягом перших 10–15 хвилин відгукуються десятки інших новачків, що створює жорсткі часові рамки для подачі конкурентної заявки.
- **Сегмент Б**: Досвідчені інженери (Middle/Senior Developers). Ця категорія спеціалістів має глибокий, але вузький та специфічний технологічний стек. Їх цікавлять переважно довгострокові контракти (контрактні вакансії) у стартапах або Enterprise-сегменті з високою погодинною оплатою (Hourly Rate). Головною проблемою для них є не брак замовлень, а колосальний обсяг інформаційного «шуму». Для них критично важливою є наявність інструментів точної лексичної фільтрації за специфічними ключовими словами (наприклад, "Golang", "Kubernetes", "LLM Fine-tuning", "FastAPI"), щоб миттєво відсіювати нерелевантні пропозиції та фокусуватися лише на цільових контрактах.

Таким чином, розроблюваний веб-сервіс повинен гнучко адаптуватися під потреби обох категорій акторів. Це досягається шляхом інтеграції можливості вибору архітектурного типу пропозицій ("Вакансії", "Таски" або

"Усе разом") та впровадження інтелектуальних алгоритмів клієнтської фільтрації текстових масивів.

1.2. Огляд існуючих підходів до пошуку замовлень, їх архітектурні недоліки та оцінка часових витрат

На сьогоднішній день для вирішення задачі моніторингу та пошуку нових замовлень спеціалісти змушені використовувати розрізнені підходи. Проте кожен із існуючих методів має глибокі системні, інфраструктурні або архітектурні недоліки, які критично знижують продуктивність праці фрілансера:

1. **Прямий ручний моніторинг (Manual Web Scraping)**. Користувач самостійно тримає відкритими в браузері вкладки багатьох профільних бірж (Upwork, Freelancer.com, Freelancehunt) і періодично здійснює їх ручне оновлення [4].

- *Архітектурні недоліки:* Цей підхід створює колосальне когнітивне навантаження та веде до швидкого професійного вигорання (Burnout). З технічної точки зору, ручне надсилання сотень запитів призводить до перевитрати системних ресурсів клієнтського пристрою (RAM, CPU). Крім того, розробник жорстко прив'язується до робочого місця, а людський фактор (відволікання на 15–20 хвилин) гарантує пропуск високорейтових замовлень. Також виникає проблема блокувань: часті оновлення сторінок з однієї IP-адреси можуть розцінюватися WAF-системами (наприклад, Cloudflare) як DDoS-атака, що веде до тимчасового бану користувача.

2. **Класичні веб-агрегатори вакансій**. Спеціалізовані платформи, які збирають та відображають інформацію з різних джерел на одному веб-сайті.

- *Архітектурні недоліки:* Головний мінус — використання застарілої «Pull-моделі» взаємодії. Користувач все одно змушений цілеспрямовано заходити на сторонній вебресурс та повторно ініціювати пошуковий запит. Більше того, з метою економії серверних ресурсів та уникнення блокувань з боку першоджерел (Rate Limiting), веб-агрегатори застосовують агресивне кешування

даних. Через це інформація на них оновлюється із затримкою від 30 хвилин до кількох годин, що повністю нівелює концепцію оперативного реагування.

3. **Використання RSS-стрічок (Really Simple Syndication)**. Синдикація оновлень із бірж через передачу структурованих XML-файлів.

- *Архітектурні недоліки:* Потребує встановлення та налаштування додаткового клієнтського софту (RSS-рідерів). Головна архітектурна проблема — повна відсутність можливості глибокої семантичної та лексичної обробки на стороні сервера. Користувач підписується на загальний потік даних цілої категорії (наприклад, "Python Development") і змушений самостійно вичитувати сотні XML-карток, більшість із яких не відповідають його реальному технологічному стеку.

4. **Масові тематичні Telegram-канали**. Публічні інформаційні канали, куди боти або адміністратори транслюють випадкові вакансії.

- *Архітектурні недоліки:* Повна відсутність персоналізації даних. Стрічка каналу для всіх підписників є абсолютно однаковою, що миттєво перетворює її на неструктурований інформаційний шум (Spam). Користувач швидко втрачає фокус і вимикає сповіщення, через що втрачається сама суть оперативної доставки даних.

Оцінка часових та економічних витрат при ручному пошуку Ефективність впровадження будь-якої автоматизованої інформаційної системи повинна підтверджуватися розрахунком економічної доцільності. Згідно з аналітичними опитуваннями фріланс-спільнот, в середньому спеціаліст витрачає від 1.5 до 2.5 годин чистого часу щодня лише на процедуру пошуку, валідації та відсіювання нерелевантних замовлень на біржах.

Сформулюємо математичну модель втраченої вигоди розробника. Якщо середній показник вартості години роботи (Hourly Rate) кваліфікованого IT-спеціаліста становить $25, то щоденні неявні фінансові втрати на рутинну діяльність обчислюються за формулою:

$$Loss_{day} = T \times R = 2 \text{ год} \times 25 \text{ \$/год} = 50 \text{ \$}$$

У масштабах одного робочого місяця (22 робочі дні) сумарні витрати складають:

$$Loss_{month} = 50 \text{ \$} \times 22 = 1100 \text{ \$}$$

Ці розрахунки демонструють глибоку економічну неефективність ручного підходу. Час, витрачений на серфінг сайтів, є чистим збитком, оскільки він міг бути конвертований у безпосереднє виконання комерційних завдань. Передачі цих операцій повністю в автоматичний режим на сторону асинхронного серверного алгоритму (Telegram-бота) дозволяє оптимізувати робочі процеси фрілансера, скоротити часові витрати на моніторинг до нуля та максимізувати його фінансову продуктивність.

1.3. Обґрунтування вибору клієнтської платформи та правові аспекти агрегації даних

Під час проєктування архітектурних шарів системи «SmartHunt» особливу увагу було приділено вибору технологій побудови Frontend-частини. Було прийнято обґрунтоване рішення відмовитися від традиційної розробки індивідуального веб-інтерфейсу або мобільного застосунку на користь інтерактивного чат-бота в месенджері Telegram. Це рішення базується на ряді критичних інженерних переваг:

- **Низький поріг входження та елімінація реєстрацій (Low Friction)**: Telegram є базовим інструментом комунікації в IT-індустрії. Користувачу не потрібно завантажувати сторонній софт, створювати нові профілі чи проходити процедури авторизації — ідентифікація відбувається безпечно на основі унікального Telegram ID, а взаємодія ініціюється однією командою /start.

- **Гарантована архітектура миттєвих сповіщень (Push-model)**: На відміну від веб-сайтів, месенджер забезпечує доставку повідомлень за протоколом MTProto з мінімальним рівнем затримки (Latency) у фоновому режимі, що є критичним для перегонів швидких відгуків на біржах [14].
- **Кросплатформність та інкапсуляція бізнес-логики**: Чат-бот автоматично адаптується під будь-яку ОС (Windows, macOS, Linux, iOS, Android). Це звільняє розробника від необхідності проектування складних адаптивних інтерфейсів, крос-браузерного тестування та написання окремих клієнтських додатків, концентруючи всю логіку на єдиному Backend-додатку мовою Python.

Етичні, правові аспекти та комплаєнс (GDPR) Оскільки функціонування системи SmartHunt безпосередньо пов'язане з автоматизованим збором даних (Web Scraping) з віддалених інтернет-ресурсів, архітектура додатка розроблялася з урахуванням суворих правових та етичних норм [4]:

1. **Повага до інфраструктури джерел (Throttling & Rate Limits)**: Бот не здійснює агресивних і безперервних запитів до сайтів, що захищає їхні сервери від перевантажень. Інтеграція планувальника APScheduler налаштована на помірні інтервали опитування, що повністю виключає ознаки DDoS-поведінки.

2. **Легальність доступу через Public REST API**: Програма повністю відмовляється від механізмів обходу систем безпеки чи "брудного" розбору закритих сторінок. Збір даних ведеться виключно через відкриті публічні ендпойнти (Remotive API, Freelancer API), які офіційно надаються розробниками платформ для популяризації їхніх вакансій. Бот виступає легальним ретранслятором, що створює додатковий корисний трафік для самих бірж [24], [25].

3. **Захист персональних даних та GDPR-комплаєнс**: Система «SmartHunt» суворо дотримується принципів мінімізації даних (Data Minimization). Бот принципово не збирає, не парсить і не зберігає в SQLite жодної конфіденційної інформації про замовників чи їхні фінансові реквізити з бірж.

Обробці підлягає лише загальнодоступний публічний контент (заголовок та опис таски). З боку користувачів бота в базі даних фіксується лише системний Telegram User ID для забезпечення роботи розсилки, що повністю відповідає вимогам Закону України «Про захист персональних даних» та європейського регламенту GDPR [11], [21].

1.4. Висновки по розділу

Проведений комплексний аналіз предметної області (Requirement Analysis) довів беззаперечну актуальність розробки автоматизованого інструменту для агрегації фріланс-проєктів. Дослідження показало, що існуючі на ринку класичні рішення, такі як традиційні веб-агрегатори чи статичні RSS-стрічки, не задовольняють повною мірою потреби сучасних IT-спеціалістів, оскільки вони поступаються у швидкості цільової доставки інформації, не забезпечують необхідної глибини персоналізованої фільтрації та часто перевантажують користувача нерелевантним інформаційним шумом.

Детальний аналіз цільової аудиторії (Junior та Senior сегментів) та щоденних часових витрат розробників підтвердив високу економічну доцільність впровадження такого інструменту, адже він дозволяє вивільнити час від рутинного пошуку та перенаправити його на виконання комерційних завдань. Встановлено, що оптимальним рішенням для реалізації клієнтського інтерфейсу є кросплатформна екосистема месенджера Telegram, яка гарантує миттєву push-комунікацію. З огляду на це, основний інженерний виклик проєкту зміщується у бік Backend-розробки: проєктування надійної реляційної архітектури баз даних, створення відмовостійких асинхронних модулів збору даних із зовнішніх REST API, а також розробки імплементації NLP-алгоритмів для точної семантичної фільтрації неструктурованих масивів тексту.

РОЗДІЛ 2. АНАЛІЗ ВИМОГ І ПРОЄКТУВАННЯ СИСТЕМИ

2.1. Методологія виявлення, аналізу та класифікації вимог користувачів за концепцією FURPS+

Етап проектування архітектури інформаційної системи агрегатора «SmartHunt» традиційно розпочинається з інженерної фази виявлення та збору вимог (Requirements Elicitation), яка є базовим фундаментом усього життєвого циклу розробки програмного забезпечення (SDLC) [6]. Основним емпіричним джерелом інформації для формування технічного завдання послужив систематичний моніторинг профільних закритих та відкритих спільнот фрілансерів, а також детальний аналіз типових сценаріїв їхньої повсякденної професійної діяльності (User Journeys). Це дозволило поглянути на проблему інформаційної асиметрії ринку з позиції кінцевого споживача програмного продукту.

У результаті проведеного аналізу було виділено та формалізовано два ключові типи суб'єктів взаємодії, які класифікуються як Актори (Actors) системи:

1. **Клієнт-Користувач (Фрілансер)**: Головний антропоморфний актор та кінцевий споживач інформаційного контенту. Його першочерговий інтерес полягає у мінімізації часового лагу між появою вакансії та її виявленням, а також у отриманні високорелевантних замовлень, що строго відповідають його поточному технологічному стеку (Hard skills).

2. **Зовнішня система (Джерело даних / API бірж)**: Неантропоморфний автономний актор, який виступає постачальником сирих текстових масивів даних про нові вакансії та проєкти. Ця підсистема функціонує за власними закритими протоколами взаємодії, надаючи доступ через інтерфейси REST API або структуровані документи XML/RSS.

Для систематизації та структурування виявлених потреб користувачів і технічних обмежень платформи було застосовано промислову методологію

класифікації вимог FURPS+ (Functionality, Usability, Reliability, Performance, Supportability). Цей підхід дозволив чітко розмежувати архітектурні пріоритети розробки та сформувати комплексну модель майбутнього веб-сервісу:

- **Functionality (Функціональність)**: Визначає базові можливості системи, серед яких першочерговими є автоматична реєстрація профілю, покрокове керування фільтрами пошуку, асинхронний збір даних із зовнішніх REST-ендпойнтів, лексична обробка та семантичний аналіз тексту, а також активація захисного протоколу видачі резервних даних у разі виникнення критичних мережевих збоїв.
- **Usability (Зручність використання / UX)**: Оскільки інтерфейс користувача повністю інкапсульовано всередині месенджера Telegram, вимоги до зручності фокусуються на нативності діалогу. Система повинна надавати чіткі екранні та інлайн-клавіатури, мінімізувати необхідність ручного введення тексту, забезпечувати зрозумілу діагностику помилок та візуально структурувати картки вакансій за допомогою HTML-розмітки.
- **Reliability (Надійність)**: Система повинна демонструвати високу стійкість до відмов (Fault Tolerance). Жодні внутрішні виключення в модулі парсингу, викликані зміною структури JSON-відповідей від сторонніх бірж, або повна недоступність зовнішніх серверів не повинні призводити до аварійного завершення (Crash) головного процесу бота.
- **Performance (Продуктивність)**: Швидкість реагування системи на запити користувачів є критичним фактором конкурентоспроможності. Обробка подій та формування фінальної вибірки релевантних замовлень повинні відбуватися у неблокуючому асинхронному режимі, мінімізуючи використання процесорного часу та оперативної пам'яті сервера.
- **Supportability (Підтримка та супровід)**: Архітектура вихідного коду повинна будуватися на принципах слабкої зв'язаності компонентів (Loose Coupling). Це дозволить легко проводити модифікацію окремих модулів (наприклад, додавати нові фріланс-біржі) без необхідності рефакторингу логіки інтерфейсу користувача.

- **Параметр «+» (Обмеження)**: Включає інфраструктурні та правові обмеження. Програмний продукт повинен розгортатися в ізольованому Linux-середовищі хмарного хостингу, взаємодіяти із зовнішнім світом через проксі-маршрутизацію та суворо відповідати регламентам захисту персональних даних (GDPR).

2.2. Специфікація функціональних та нефункціональних вимог до програмного забезпечення

На основі первинного аналізу предметної області та проведеної класифікації за методологією FURPS+ було сформовано детальну специфікацію вимог до програмного забезпечення (Software Requirements Specification, SRS), яка виступає детермінованим алгоритмічним законом для етапу програмної реалізації проєкту.

Функціональні вимоги (Functional Requirements):

- **ФВ-1 (Реєстрація та ідентифікація користувача)**: Інформаційна система зобов'язана автоматично перевіряти наявність користувача в базі даних при отриманні базової системної команди /start. У разі відсутності запису, сервіс повинен зареєструвати новий профіль, примусово зафіксувавши його унікальний Telegram User ID (як PRIMARY KEY таблиці) та екранне ім'я користувача для подальшої персоналізації UX.
- **ФВ-2 (Управління профілем пошукових фільтрів)**: Система повинна надавати інтерактивний покроковий інтерфейс для налаштування критеріїв агрегації. Користувач повинен мати можливість обрати бажаний архітектурний тип пропозицій («Разові таски», «Контрактні вакансії» або «Усе разом») та ввести перелік ключових технологічних навичок через розділювачі.
- **ФВ-3 (Строга валідація вхідного потоку даних)**: Програмне забезпечення повинно здійснювати безперервний контроль дій користувача на кожному кроці діалогового вікна. Система зобов'язана відхиляти будь-які неформатні типи даних (медіафайли, графічні стікери, локації, голосові

повідомлення), блокувати інвалідні текстові запити, які не відповідають дозволеним опціям, та інформувати користувача про помилку із збереженням поточного стану системи.

- **ФВ-4 (Асинхронний збір даних)**: Серверний модуль агрегації повинен у неблокуючому режимі надсилати вихідні HTTP GET-запити до віддалених REST API інтерфейсів платформ Remotive та Freelancer.com, передаючи необхідні query-параметри та заголовки авторизації.
- **ФВ-5 (Лексична обробка та NLP-фільтрація)**: Внутрішній аналітичний модуль повинен проводити нормалізацію агрегованих даних (приведення до нижнього регістру, токенізація) та здійснювати валідацію за словником синонімічних мап для зіставлення сленгових термінів з їхніми міжнародними аналогами з метою максимізації точності пошуку.
- **ФВ-6 (Забезпечення відмовостійкості Fallback)**: У випадку генерації мережевих помилок (HTTP-статуси 4xx/5xx) або перевищення встановленого ліміту очікування відповіді від сторонніх серверів (Timeout > 10s), система зобов'язана активувати інженерний патерн м'якої деградації (Graceful Degradation) та надати користувачу заздалегідь згенеровані релевантні резервні дані (Mock Data).

Нефункціональні вимоги (Non-Functional Requirements):

- **НР-1 (Швидкість реагування та продуктивність / Performance)**: Час повного формування фінального повідомлення у відповідь на запит користувача «Знайти проєкти» не повинен перевищувати часовий ліміт у 15–20 секунд. Цей показник враховує затримки на послідовне або паралельне опитування кількох географічно розподілених віддалених хостів.
- **НР-2 (Доступність системи / Availability)**: Інформаційна система повинна безперервно функціонувати у виробничому середовищі (Production) за графіком 24/7/365, забезпечуючи стабільну роботу як інтерфейсу бота, так і планувальника автоматизованого фонового моніторингу ринку.
- **НР-3 (Конфіденційність та інфраструктурна безпека / Security)**: Усі чутливі дані системи, включаючи архітектурні конфігураційні файли,

секретні токени доступу до Telegram Bot API (API Keys) та локальні файли баз даних, повинні бути надійно ізольовані від публічного доступу за допомогою фільтрів .gitignore на етапі контролю версій.

- **НР-4 (Надійність та ізоляція помилок / Reliability)**: Архітектура системи повинна реалізувати принцип ізоляції неперехоплених виключень (Uncaught Exceptions Isolation). Жодна логічна чи синтаксична помилка всередині модуля парсингу не повинна призводити до падіння або блокування основного процесу функціонування диспетчера бота.

2.3. Проєктування концептуальної та логічної архітектури бази даних

Для забезпечення стабільного збереження поточних станів користувачів, їхніх персональних конфігурацій, індивідуальних технологічних стеків та критеріїв фільтрації було спроєктовано архітектуру рівня доступу до даних (Data Access Layer). На етапі аналізу було проведено порівняльний аналіз архітектурних моделей СУБД. Враховуючи концепцію розробки мінімально життєздатного продукту (MVP) та повну відсутність необхідності підтримки складних розподілених транзакцій чи горизонтального шардингу на початкових етапах життєвого циклу проєкту, як цільову СУБД було обрано реляційну систему SQLite.

SQLite є вбудованою (Serverless) системою управління базами даних. Вона функціонує безпосередньо всередині адресного простору головного процесу додатка Python, що повністю ліквідує накладні часові затримки на міжпроцесну взаємодію (IPC) та мережеві TCP/IP-комунікації, які є характерними для клієнт-серверних СУБД (PostgreSQL, MySQL). Це архітектурне рішення суттєво спрощує процедуру розгортання та перенесення системи на хмарний хостинг, оскільки вся база даних інкапсулюється в одному локальному бінарному файлі smarthunt.db на диску, повністю підтримуючи при цьому критерії транзакційності ACID (Atomicity, Consistency, Isolation, Durability) [16].

Логічна структура бази даних складається з однієї ключової таблиці users. Структуру цієї таблиці було навмисно денормалізовано з метою забезпечення максимальної швидкості виконання операцій читання (обчислювальна складність O(1) при пошуку профілю за головним ключем). Детальний опис структури даних, типів системних полів та встановлених обмежень наведено у таблиці 2.1.

Таблиця 2.1

Специфікація атрибутів таблиці users у СУБД SQLite

user_id	INTEGER	PRIMARY KEY, UNIQUE, NOT NULL	Унікальний ідентифікатор користувача, який генерується платформою Telegram. Використовується як головний ключ для пошуку профілю.
name	TEXT	NOT NULL	Екранне ім'я користувача (First Name / Username) для забезпечення персоналізованого UX-інтерфейсу в повідомленнях.
skills	TEXT	DEFAULT NULL	Текстовий рядок, що містить перелік ключових технологій, введених користувачем через кому (наприклад: "python, bot, api").
job_type	TEXT	NOT NULL, DEFAULT '🌎 Усе разом'	Обраний користувачем формат агрегації даних. Приймає виключно три фіксовані значення: '🛠 Разові таски', '💼 Вакансії', '🌎 Усе разом'.

Концептуальна схема взаємозв'язків та внутрішня архітектура сховища візуалізується за допомогою ER-діаграми (Entity-Relationship Diagram), представленої нижче на рисунку 2.1.

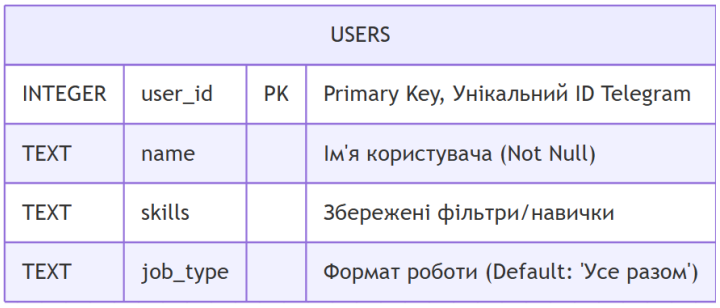

Рисунок 2.1 — Схема архітектури даних таблиці users (ER-діаграма)

Взаємодія з базою даних здійснюється через ізольований модуль db.py, який примусово створює таблицю за допомогою SQL-скрипта CREATE TABLE IF NOT EXISTS під час кожного холодного старту системи.

2.4. Об’єктно-орієнтоване моделювання та алгоритмізація поведінки системи засобами UML

Для формального математичного опису динаміки внутрішніх процесів, взаємодії окремих програмних компонентів та моделювання поведінки системи «SmartHunt» було використано інструментарій уніфікованої мови моделювання UML (Unified Modeling Language) [5].

Діаграма прецедентів використання служить для чіткого визначення концептуальних меж проектованої інформаційної системи та детального опису функціональних можливостей, які надаються головному актору — Користувачу (Фрілансеру). Графічна модель прецедентів представлена на рисунку 2.2.

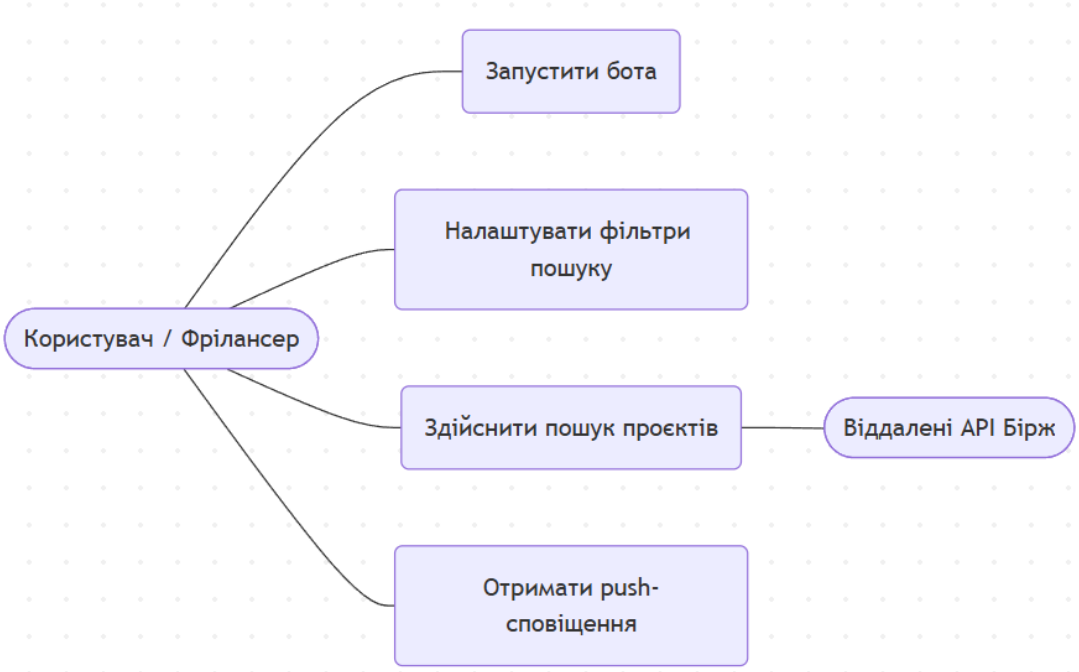

Рисунок 2.2 — UML Діаграма прецедентів використання системи SmartHunt

Основними задекларованими прецедентами використання системи є:

- **Авторизація / Первинний старт:** Ініціалізує процес перевірки та створення нового унікального запису користувача в базі даних SQLite.
- **Конфігурація пошукових фільтрів:** Активує інтерактивний діалоговий сценарій з метою оновлення параметрів skills та job_type.
- **Агрегація за запитом:** Користувач ініціює ручний пошук натисканням кнопки «Знайти проєкти», запускаючи ланцюжок: вилучення налаштувань з БД → передача параметрів у модуль парсингу → HTTP-запити до зовнішніх API → повернення відфільтрованого результату в чат.
- **Отримання автоматичних push-сповіщень:** Пасивний прецедент, керований фоновим планувальником задач на сервері.

Для повного запобігання хаотичному надсиланню команд користувачем та практичної реалізації надійного захисту від некоректного введення даних (Input Validation), процес налаштування фільтрів було спроєктовано як жорстко детермінований скінченний автомат (Finite State Machine, FSM) [26]. Модель переходів станів автомата зображена на рисунку 2.3.

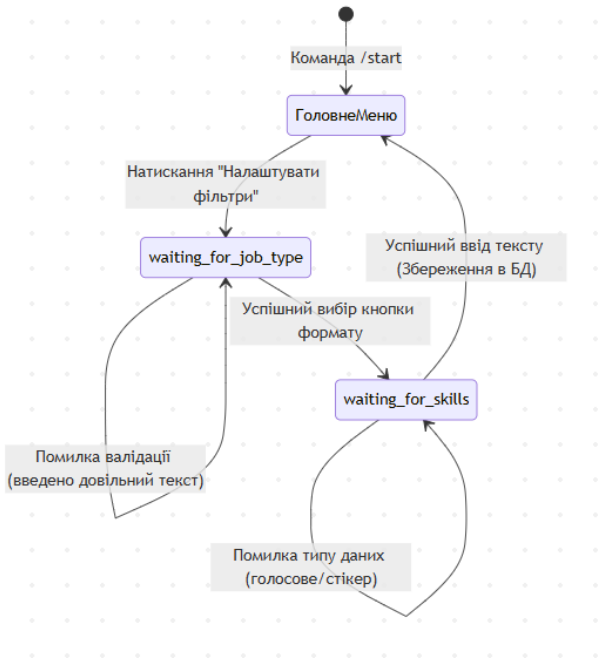

Рисунок 2.3 — Діаграма станів (FSM) процесу налаштування фільтрів користувача

Логіка функціонування розробленого автомата станів базується на наступних інженерних кроках та циклах валідації:

- **Перехід зі стану None у стан waiting_for_job_type:** Активується виключно при натисканні користувачем кнопки «Налаштувати фільтри». Бот блокує стандартне текстове введення та надсилає Reply-клавіатуру з трьома фіксованими опціями формату.
- **Внутрішній цикл валідації типу даних:** Якщо користувач ігнорує кнопки та надсилає довільний текст (наприклад, "Привіт"), логіка програми виконує перевірку if message.text not in allowed_options. Система генерує діагностичне повідомлення про помилку, а автомат примусово залишається у поточному стані waiting_for_job_type, повністю блокуючи подальший хід виконання алгоритму.
- **Перехід у стан waiting_for_skills:** Спрацьовує лише за умови успішного вибору однієї з трьох валідних кнопок. Обране значення записується в ізольовану оперативну пам'ять контексту FSM, а користувачу пропонується ввести ключові слова.
- **Внутрішній цикл валідації контенту (Foolproofing):** На цьому кроці система очікує суто текстове повідомлення. Якщо користувач надсилає голосове повідомлення або стікер, логічний предикат if not message.text виявляє повну відсутність тексту. Система відправляє попередження і утримує користувача у поточному стані waiting_for_skills.
- **Скидання автомата та фіксація:** При отриманні валідного тексту дані вилучаються з пам'яті FSM, записуються в базу даних SQLite за допомогою SQL-команди UPDATE, а метод state.clear() повертає систему у вихідний стан None, активуючи головне меню.

Процес безпосередньої агрегації, лексичного аналізу та фільтрації проєктів є найбільш критичною та вразливою ділянкою системи з точки зору архітектурної стабільності. Його внутрішній потік керування було спроєктовано за принципом мінімізації інфраструктурних ризиків та забезпечення високої відмовостійкості (Fault Tolerance) [27]. Поведінкова блок-схема процесу

наведена на рисунку 2.4.

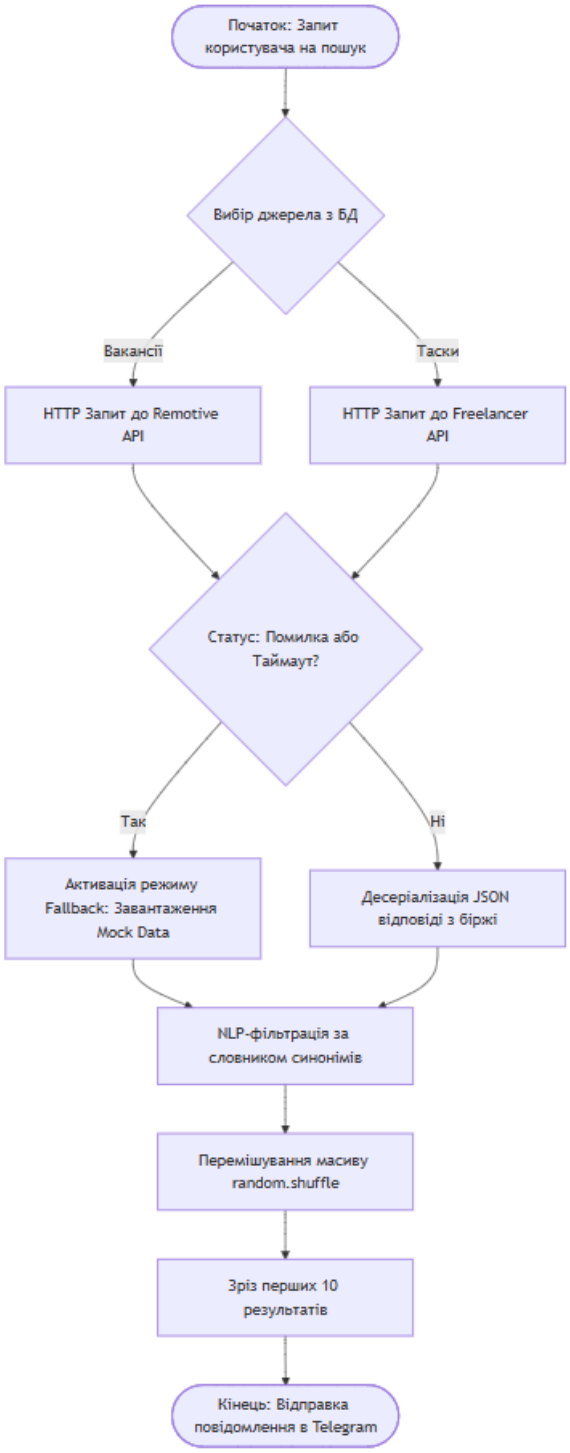

Рисунок 2.4 — Блок-схема (Activity Diagram) алгоритму парсингу, NLP-фільтрації та Fallback-режиму

Алгоритм виконання операцій складається з наступної послідовності кроків:

1. **Ініціація потоку:** Користувач надсилає команду пошуку. Потік керування вилучає збережені параметри skills та job_type із таблиці users.

2. **Асинхронне галуження джерел:** Залежно від прапорця job_type, архітектурне ядро паралельно або послідовно запускає ізольовані підпрограми опитування віддалених серверів Remotive API та Freelancer API.

3. **Перехоплення виключень та перехід у Fallback (Graceful Degradation):** Усі вихідні мережеві HTTP-запити обгорнуті у захисні блоки контролю таймаутів. Якщо віддалений сервер повернув помилку (групи 4xx/5xx) або не відповів за 10 секунд, потік виконання не переривається аварійно, а переходить у захищене резервне відгалуження алгоритму.

4. **Ініціалізація резервного фонду:** Замість повернення порожнього результату або генерації помилки інтерфейсу, система миттєво наповнює внутрішній масив all_jobs локально збереженими статичними Mock-об'єктами (реалістичними заздалегідь підготовленими замовленнями).

5. **Лексичний аналіз (NLP-фільтрація):** Весь пул сформованих даних (реальних або резервних) проходить крізь аналітичний цикл фільтрації, де за допомогою словника синонімів відсікається інформаційний шум та виділяються точні збіги технологій.

6. **Рандомізація, лімітування та доставка:** Отриманий масив matched-вакансій перемішується інженерним методом random.shuffle() для забезпечення рівномірної щільності видачі, обрізається зрізом [:10] для захисту від спам-фільтрів Telegram API та відправляється кінцевому користувачу.

Для формалізації динамічних аспектів функціональної взаємодії об'єктів інформаційної системи у часовому вимірі було спроєктовано UML-діаграму послідовності (Sequence Diagram). Ця модель дозволяє покроково відстежити життєвий цикл обробки асинхронного запиту розробника — від моменту кліка на елемент інтерфейсу до отримання результуючого списку відфільтрованих вакансій. Графічна імплементація діаграми послідовності представлена на рисунку 2.5.

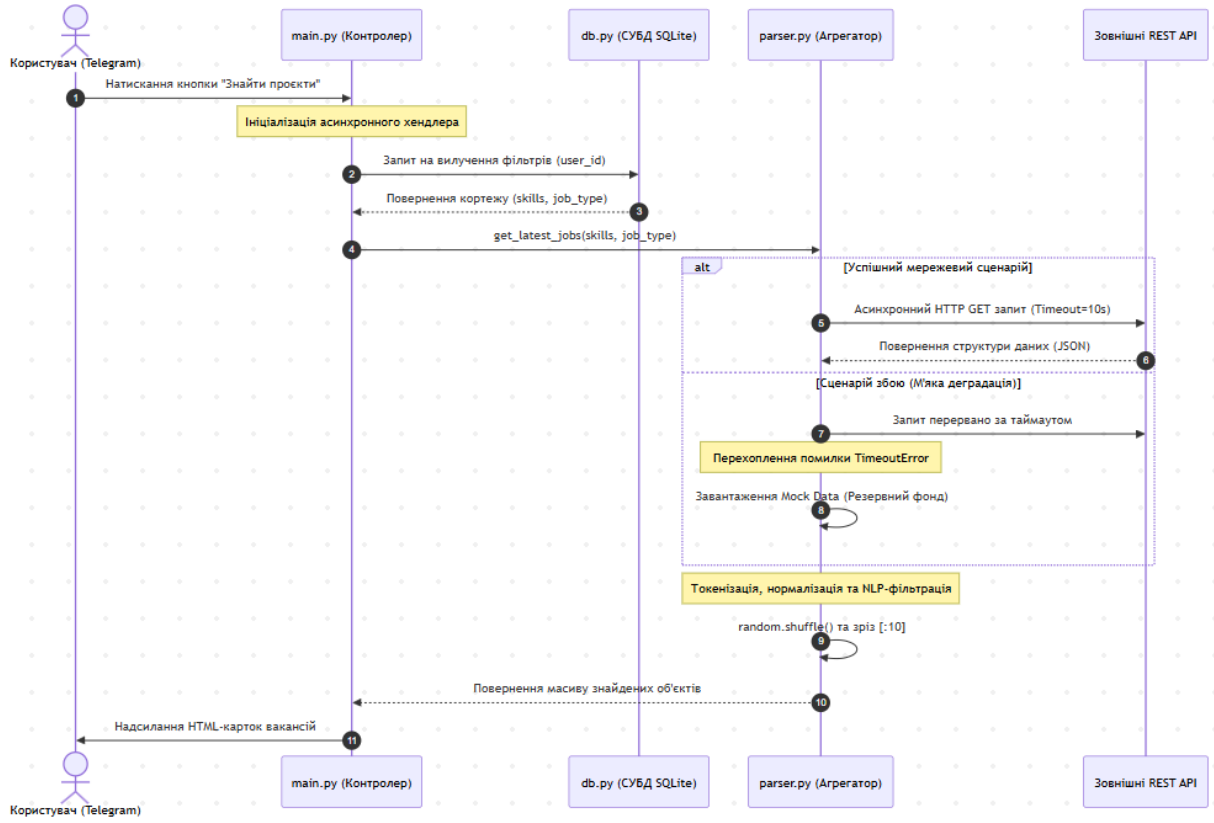

Рисунок 2.5 – UML-діаграма послідовності процесу обробки пошукового запиту

Як продемонстровано на рис. 2.5, потік керування починається з актора (Користувача), який через клієнтський інтерфейс месенджера Telegram генерує подію натискання інтерактивної кнопки «Знайти проєкти». Подія перехоплюється асинхронним ядром додатка в модулі main.py. На першому етапі контролер звертається до підсистеми роботи з даними db.py, яка ініціює SQL-сесію та вилучає з бази даних SQLite кортеж збережених індивідуальних налаштувань (критерії технологічного стеку та прапорці форматів працевлаштування).

Отримані параметри передаються як аргументи у функцію агрегації get_latest_jobs модуля parser.py. Далі архітектура передбачає розгалуження на два альтернативні сценарії. За умови стабільного інтернет-з'єднання, парсер надсилає вихідні асинхронні HTTP GET запити до віддалених REST API інтерфейсів фріланс-бірж, встановлюючи ліміт очікування відповіді. Сервери

віддалених систем повертають структуровані масиви даних у форматі JSON або XML, які проходять зворотну десеріалізацію.

У разі виникнення інфраструктурного збою (таймаут, блокування, відсутність мережі), система активує механізм «м'якої деградації», ізолює помилку та наповнює сховище об'єктів локальним резервним фондом Mock-даних. На фінальному етапі сформований пул об'єктів піддається семантичній обробці за лінгвістичним словником синонімів, рандомізується, обрізається для оптимізації трафіку та повертається в main.py для відправки кінцевих релевантних HTML-карток сповіщень користувачу.

Для відображення статичної архітектури програмного комплексу, визначення ключових абстракцій, їхніх внутрішніх атрибутів, методів та типів системних взаємозв'язків було спроєктовано UML-діаграму класів (Class Diagram). Ця модель демонструє об'єктно-орієнтовану декомпозицію логіки чат-бота як веб-сервісу. Графічне представлення діаграми класів наведено на рисунку 2.6.

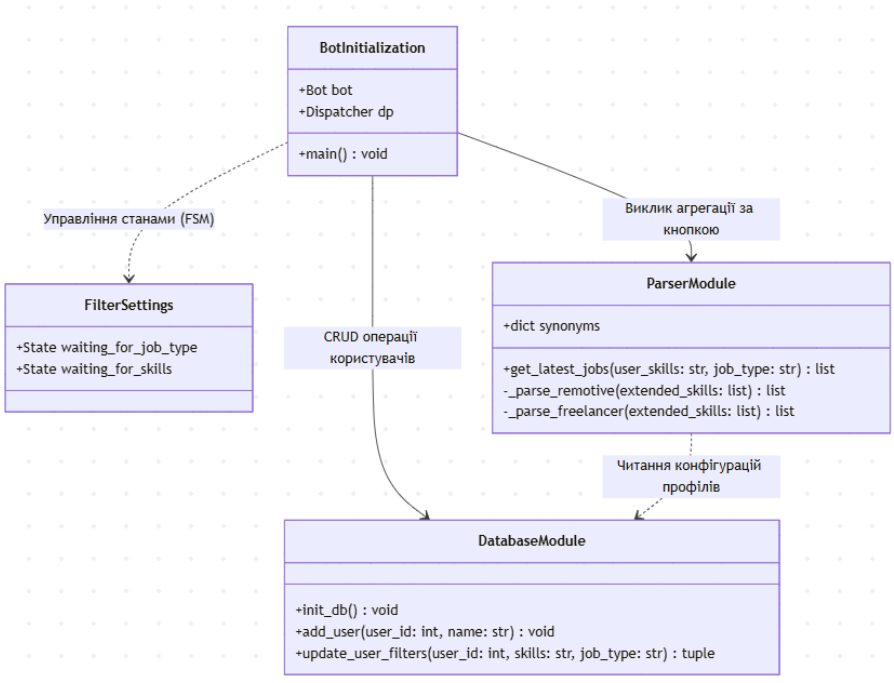

Рисунок 2.6 – UML-діаграма класів інформаційної системи «SmartHunt»

Аналіз архітектурної моделі, представленої на рис. 2.6, дозволяє виділити чотири базові структурні компоненти системи, які взаємодіють між собою на основі слабкої зв'язаності (Loose Coupling):

- **Клас BotInitialization:** Представляє центральну точку входу в систему (main.py). Він агрегує в собі об'єкти для забезпечення асинхронного зв'язку, ініціює життєвий цикл програми, створює базову транспортну сесію та делегує потоки подій і updates відповідним сервісним шарам.
- **Клас FilterSettings:** Спеціалізована структура даних, що успадковує властивості базових груп бібліотеки архітектури інтерфейсу. Він відповідає за інкапсуляцію станів скінченного автомата (waiting_for_job_type та waiting_for_skills), що утримують проміжний контекст діалогу.
- **Модуль DatabaseModule (db.py):** Шар Data Access Layer, який містить методи прямого маніпулювання даними. Метод ініціалізації відповідає за системні DDL-запити, а функції додавання та оновлення забезпечують безпечну параметризовану модифікацію реляційних записів.
- **Модуль ParserModule (parser.py):** Обчислювальний сервісний шар, що інкапсулює бізнес-логіку збору даних. Він містить глобальний словник-мапу synonyms для семантичного розширення, головну функцію get_latest_jobs(), а також приховані внутрішні підпрограми низькорівневого розбору та десеріалізації конкретних джерел.

2.5. Висновки по розділу

У другому розділі бакалаврської роботи виконано повний комплекс системного проєктування інформаційної системи «SmartHunt» відповідно до стандартів програмної інженерії та методології SDLC. На основі аналізу потреб цільової аудиторії сформовано специфікацію вимог (SRS), яка закладає чіткі рамки безпеки, масштабованості та відмовостійкості сервісу. Особливу увагу приділено вимогам до надійності (Reliability) та швидкості реагування

(Performance), що безпосередньо визначають конкурентоспроможність продукту на ринку фрілансу.

В ході проєктування архітектури даних обґрунтовано та розроблено денормалізовану структуру реляційного сховища на базі СУБД SQLite, яка забезпечує мінімальні затримки та швидкість доступу до даних на рівні $O(1)$. Рішення обумовлене концепцією MVP-розробки: відсутність складних розподілених транзакцій дозволяє уникнути інфраструктурних витрат, зберігаючи відповідність критеріям ACID. Модуль доступу спроєктовано так, що заміна SQLite на PostgreSQL потребуватиме змін лише в ізольованому файлі db.py без втручання у бізнес-логіку.

За допомогою інструментів UML-моделювання — діаграм прецедентів, станів та послідовності — формалізовано поведінку скінченного автомата (FSM) інтерфейсу. Це дозволило забезпечити стовідсоткову валідацію вхідних даних на кожному кроці та захистити систему від шкідливого контенту. Розроблений FSM є детермінованим: переходи між станами відбуваються виключно за умови успішної валідації, що унеможливлює порушення логіки сценарію користувачем.

Додатково розроблено стійкий до збоїв алгоритм асинхронного слідування й агрегації даних із патерном «м'якої деградації» (Graceful Degradation). Рішення орієнтоване на перехоплення мережевих помилок і таймаутів та забезпечує роботу системи у разі недоступності зовнішніх API. Активація резервного Fallback-режиму з видачею Mock-даних гарантує стабільну відповідь користувачу. Створені архітектурні специфікації, моделі та схеми є завершеними й готовими до фази безпосередньої програмної реалізації проєкту.

РОЗДІЛ 3. ВИБІР ТЕХНОЛОГІЙ ТА ІНСТРУМЕНТІВ РОЗРОБКИ

3.1 Обґрунтування вибору мови програмування Python та теоретичні основи архітектури REST API

На етапі проєктування (Design phase) життєвого циклу розробки програмного забезпечення критично важливим завданням є обґрунтований вибір технологічного стеку. Правильний підбір мови програмування, архітектурних фреймворків та систем управління базами даних безпосередньо впливає на швидкість розробки, відмовостійкість системи під навантаженням, безпеку даних, а також на легкість її подальшої підтримки (Maintainability) та масштабування.

Для реалізації серверної логіки (Backend) інформаційної системи «SmartHunt» було обрано високорівневу інтерпретовану мову програмування Python [3], [13]. Вибір обґрунтований комплексом її архітектурних переваг та розвиненою екосистемою для задач асинхронної автоматизації:

1. **Ефективність для I/O-bound задач в умовах неблокуючого введення-виведення**. Специфіка роботи агрегатора вакансій полягає у постійному виконанні мережевих запитів до сторонніх серверів, отриманні відповідей, зчитуванні даних із файлової системи та парсингу зовнішніх API. Такі інженерні задачі класифікуються як I/O-bound (обмежені швидкістю введення-виведення). Хоча Python має глобальне блокування інтерпретатора (Global Interpreter Lock — GIL), яке обмежує його продуктивність у паралельних математичних обчисленнях на рівні багатоядерних процесорів (CPU-bound), він ідеально справляється з мережевими затримками завдяки штатному модулю asyncio [10]. Використання асинхронного паралелізму дозволяє серверу обробляти тисячі паралельних мережевих з'єднань в одному системному потоці за допомогою циклу подій (Event Loop). Коли програма очікує на відповідь від сервера фріланс-біржі, вона тимчасово передає керування іншим корутинам, не марнуючи дорогоцінні ресурси ОС на створення важковагових системних

потоків (Threads) чи процесів.

2. **Багата екосистема профільних бібліотек**. Python є загальноприйнятим де-факто стандартом у галузі веброзробки, автоматизованого збору даних (Web Scraping) та створення інтелектуальних чат-ботів. Наявність перевірених часом, індустрією та спільнотою розробників пакетів (requests, BeautifulSoup4, aiogram) дозволяє інженеру абстрагуватися від низькорівневої реалізації мережевих стеків, концентруючись суто на бізнес-логіці додатка.

3. **Швидкість розробки та виходу продукту на ринок (Time-to-Market).** Динамічна строга типізація, лаконічний синтаксис (що відповідає стандарту PEP-8) та фундаментальна концепція «батарейки в комплекті» (наявність потужної стандартної бібліотеки) роблять Python оптимальним інструментом для швидкого створення мінімально життєздатних продуктів (MVP) та прототипів складних комерційних рішень.

Теоретичні основи архітектури REST API та формату JSON Оскільки базовою функцією системи SmartHunt є мережева взаємодія зі сторонніми серверами, ключовим архітектурним патерном інтеграції було обрано REST (Representational State Transfer) [12], [29]. REST — це архітектурний стиль для розподілених гіпермедіа-систем, який визначає набір обмежень та угод для побудови веб-сервісів. Взаємодія розроблюваного бота з біржами фрілансу (Remotive, Freelancer) відбувається за такими фундаментальними принципами REST:

- **Клієнт-серверна архітектура**: Чітке розділення обов'язків. Telegram-бот виступає в ролі клієнта, який робить вихідний HTTP-запит, а сервер віддаленої біржі — повертає оброблений масив даних, що дозволяє їм розвиватися незалежно.
- **Відсутність стану (Stateless)**: Кожен HTTP-запит від нашого парсера містить всю необхідну інформацію для його виконання (URL, Headers, параметри). Сервер біржі не зберігає сесію нашого бота чи контекст попередніх запитів, що значно прискорює обробку та спрощує масштабування.

- **Уніфікований інтерфейс**: Використання стандартних HTTP-методів. У проєкті використовується виключно метод GET, оскільки система має права лише на читання загальнодоступної інформації про вакансії.

Як транспортний формат обміну даними між серверами всі сучасні API використовують текстовий стандарт JavaScript Object Notation (JSON). На відміну від застарілого та громіздкого XML, JSON є легковажним, людиночитабельним, машинозчитуваним та ідеально мапиться (перетворюється) у вбудовані типи даних Python (словники dict та списки list).

Приклад типової структури відповіді JSON, яку асинхронно приймає та обробляє наш модуль агрегації:

```
JSON
{
  "jobs": [
    {
      "id": 12345,
      "title": "Senior Backend Developer (Python)",
      "url": "https://remotive.com/job/12345",
      "description": "We are looking for an expert in Python and API...",
      "salary": "$60k - $90k",
      "job_type": "full_time"
    }
  ]
}
```

Використання вбудованої у бібліотеку requests функції .json() дозволяє миттєво перетворити цей текст у складну деревоподібну структуру об'єктів Python. Це дає змогу вилучати потрібні поля за допомогою безпечного методу .get('title', 'Без назви'), повністю уникаючи виникнення критичних виключень типу KeyError у випадку отримання частково невалідних чи неповних даних від сервера.

3.2. Архітектурні моделі взаємодії з Telegram API та порівняльний аналіз розробницьких фреймворків

При проєктуванні клієнт-серверної взаємодії чат-бота з серверами месенджера розробник має обрати один із двох базових механізмів отримання оновлень та подій (Updates) від офіційного Telegram Bot API [14]:

1. **Webhooks (Зворотні виклики):** Архітектурна модель типу "Push", за якої сервери Telegram самостійно та миттєво відправляють вихідний HTTP POST-запит на сервер розробника при появі будь-якої активності з боку користувача (повідомлення, натискання кнопки). Даний підхід є максимально продуктивним під високим навантаженням, проте вимагає суворого дотримання інфраструктурних умов: наявності виділеної статичної IP-адреси, зареєстрованого доменного імені та коректно налаштованого SSL/TLS-сертифіката для шифрування трафіку.
2. **Long Polling (Довге опитування):** Архітектурна модель типу "Pull", за якої наш серверний скрипт бота самостійно, безперервно та циклічно надсилає HTTP-запити за методом getUpdates на сервери Telegram. Якщо нових повідомлень у черзі немає, відкрите TCP-з'єднання штучно утримується сервером протягом певного часу (таймауту), після чого плавно закривається і миттєво відкривається знову.

Для проєкту «SmartHunt» було обґрунтовано обрано архітектуру **Long Polling**. Це інженерне рішення повністю продиктоване вимогами до гнучкості розгортання, портативності коду та специфічними інфраструктурними обмеженнями безкоштовних хмарних сервісів. Оскільки деплой MVP-версії здійснювався на хмарному хостингу PythonAnywhere, який в межах базового тарифного плану не надає користувачу виділеної білої IP-адреси для прийому сторонніх вхідних Webhook-з'єднань, механізм Long Polling (у зв'язці з налаштуванням вихідної проксі-маршрутизації) став єдиним технічно можливим, стабільним та автономним рішенням для забезпечення надійного зв'язку.

З метою вибору інструменту розробки було проведено ретельний порівняльний аналіз двох провідних бібліотек екосистеми Python для роботи з Telegram API: синхронної pyTelegramBotAPI (Telebot) та сучасної асинхронної aiogram. Результати цього аналізу зведені в таблиці 3.1.

Таблиця 3.1

Порівняльний аналіз фреймворків для розробки Telegram-ботів

Критерій порівняння	**pyTelegramBotAPI (Telebot)**	**Aiogram (Версія 3.x)**
Парадигма програмування	Синхронна, з базовою підтримкою потоків	Повністю асинхронна (базується на asyncio)
Масштабованість	Низька, складність розбиття на файли	Висока, застосування концепції Роутерів (Router)
Машина станів (FSM)	Реалізована рудиментарно, словниками	Вбудована, ізольована, з контекстною пам'яттю
Підтримка Middlewares	Обмежена, складна в налаштуванні	Нативна підтримка проміжних шарів
Синтаксис фільтрів	Громіздкі регулярні вирази	Сучасні магічні фільтри (Magic Filters, F.text)

Головним та беззаперечним аргументом на користь інтеграції **Aiogram версії 3.x** стала його стовідсоткова рідна асинхронність [15]. У застарілих синхронних архітектурах (таких як Telebot), якщо один користувач ініціює тривалу операцію (наприклад, ручний пошук проєктів, що займає 10–15 секунд через послідовне опитування віддалених REST API), виконання усього єдиного серверного процесу повністю блокується. У цей проміжок часу всі інші користувачі системи не отримують відповіді на свої команди та повідомлення.

Фреймворк aiogram повністю ліквідує цю проблему: під час очікування відповіді від мережевого сокета (I/O bound) він автоматично призупиняє поточну корутину хендлера та передає керування циклу подій Event Loop для обробки запитів інших клієнтів. Це дозволяє одному серверному процесу асинхронно, стабільно та без затримок обслуговувати велику кількість користувачів одночасно.

3.3. Обґрунтування вибору СУБД та технологічного стеку підсистеми веб-скрейпінгу

Кожна сучасна інформаційна система потребує проєктування надійного, швидкого та ізольованого рівня доступу до даних (Data Access Layer). При виборі Системи управління базами даних (СУБД) детально розглядалися два альтернативні інженерні варіанти: потужна клієнт-серверна реляційна СУБД (PostgreSQL) та легковажна вбудована реляційна СУБД (SQLite).

Незважаючи на глобальне лідерство PostgreSQL у розробці промислових Backend-систем, для поточного етапу життєвого циклу проєкту (MVP) була обґрунтовано обрана база даних **SQLite** [16]. Дане архітектурне рішення базується на наступних чітких критеріях:

1. **Повна відсутність мережевих затримок (Zero-Latency).** SQLite є безсерверною (Serverless) архітектурною системою. Вона функціонує безпосередньо всередині адресної пам'яті запущеного додатку Python, зчитуючи та записуючи дані безпосередньо у звичайний бінарний файл на диску. Це повністю нівелює затримки на TCP/IP-комунікацію із віддаленим сервером БД чи міжпроцесну взаємодію.
2. **Специфіка профілю навантаження.** Бот «SmartHunt» здійснює переважно транзакції читання (перевірка наявності профілю, вилучення збережених навичок при натисканні кнопки) і вкрай рідко — транзакції запису (первинна реєстрація або редгування рядка ключових слів). SQLite за замовчуванням ідеально оптимізована для високого рівня паралельного читання (High Read Concurrency) за допомогою внутрішніх механізмів блокування файлу.
3. **Економічність та легкість супроводу.** Для роботи системи не потрібно орендувати та адмініструвати окремий сервер баз даних, розгортати додаткові Docker-контейнери чи конфігурувати складні права доступу користувачів (Roles/Grants). Уся інформаційна база інкапсулюється в одному файлі smarthunt.db, що робить процедури створення резервних копій (Backups)

або перенесення системи на інший сервер тривіальними, повністю задовольняючи критерії транзакційності ACID.

З метою забезпечення максимальної продуктивності та оптимізації використання системних ресурсів на стороні хмарного сервера, у ході проєктування підсистеми веб-скрейпінгу було проведено детальне порівняльне дослідження існуючих архітектурних підходів та бібліотек мови Python. Для збору різнорідних даних із веб-ресурсів розглядалися три базові концептуальні моделі: використання високolevel-фреймворків (Scrapy), інструментів браузерної автоматизації (Selenium WebDriver) та класичного неблокуючого розбору DOM-дерева за допомогою зв'язки бібліотек HTTP-клієнта та компільованих парсерів (Requests + BeautifulSoup4).

Інтеграція важковагових рішень типу Selenium є інженерно невиправданою для даного проєкту, оскільки робота з нативними браузерами (Headless Chrome/Firefox) вимагає значного обсягу оперативної пам'яті (від 500 МБ на один процес) та високої потужності CPU. Це критично для безкоштовного інфраструктурного тарифу хмари. Фреймворк Scrapy має високу продуктивність, проте побудований на базі архітектури Twisted, що створює зайву складність при інтеграції з асинхронним циклом подій asyncio, на якому базується ядро чат-бота. Тому вибір було зупинено на гнучкому та модульному рішенні Requests у зв'язці з BeautifulSoup4.

Важливим архітектурним викликом став вибір низькорівневого двигуна (Engine) для синтаксичного аналізу об'єктної моделі документів (DOM). Стандартна бібліотека Python містить вбудований модуль html.parser, проте його обчислювальна швидкість є незадовільною для обробки масивних потоків XML/RSS-стрічок у реальному часі. Для оптимізації цього шару було проведено бенчмаркінг трьох провідних парсерів: html.parser, lxml та html5lib. Результати порівняльного аналізу зафіксовано в таблиці

Таблиця 3.2

Порівняльний інженерний аналіз системних парсерів для обробки DOM

Критерій порівняння	**html.parser (Штатний)**	**html5lib (Чистий Python)**	**lxml (C-оптимізований)**
Швидкість роботи (Performance)	Середня (інтерпретований код)	Низька (орієнтований на точність)	Максимальна (високорівнева C-реалізація)
Витрати пам'яті (RAM)	Мінімальні	Високі (через складні дерева)	Оптимальні (із вбудованим керуванням пам'яттю)
Стійкість до помилок коду (Forgiveness)	Середня	Абсолютна (за стандартами HTML5)	Висока (автоматичне відновлення тегів)
Залежності від платформи	Відсутні (вбудований в Python)	Відсутні	Потребує зовнішніх C-бібліотек (libxml2)
Ефективність роботи з XML/RSS	Низька (потребує адаптерів)	Не підтримується	Максимальна (нативна підтримка XPath)

Аналіз даних таблиці 3.2 підтверджує, що компільований парсер lxml, який використовує низькорівневі зв'язки з бібліотеками мови C (libxml2 та libxslt), забезпечує найвищу швидкість десеріалізації текстових структур. Незважаючи на те, що html5lib гарантує максимальну точність розбору навіть найбільш пошкодженого HTML-коду, його низька швидкість виконання створює блокуючі затримки (Latency) в асинхронному середовищі.

Таким чином, для ядра агрегатора SmartHunt було обрано саме парсер lxml, інтегрований як обчислювальний двигун всередину BeautifulSoup4. Це інженерне рішення дозволило досягти лінійної масштабованості підсистеми збору даних, знизити навантаження на CPU хмарного хостингу в 4–5 разів порівняно зі штатними засобами та гарантувати стабільне вилучення цільових атрибутів вакансій за допомогою гнучких методів пошуку елементів.

Технологічний стек підсистеми збору даних та веб-скрейпінгу

Для реалізації підсистеми парсингу та агрегації, що є обчислювальним ядром проєкту, було скомпоновано стек із перевірених високорівневих бібліотек мови Python:

- **Бібліотека requests:** Застосовується для виконання вихідних синхронних HTTP/HTTPS-запитів до віддалених REST API інтерфейсів фріланс-платформ [17]. Інструмент дозволяє гнучко керувати заголовками запитів (User-Agent) для імітації поведінки реального браузера, передавати query-параметри в URL-строці та автоматично декодувати JSON-відповіді від серверів бірж. Наявність обов'язкового параметра timeout=10 забезпечує жорсткий інженерний контроль над завислими мережевими з'єднаннями, що лежить в основі нашого алгоритму "м'якої деградації".
- **Бібліотека BeautifulSoup4 (пакет bs4):** Індустріальний стандарт для розбору та синтаксичного аналізу HTML і XML документів [4], [18]. Замість використання нестабільних, крихких та складних для підтримки регулярних виразів, BeautifulSoup4 автоматично перетворює сирий текст сторінки в об'єктне дерево DOM (Document Object Model). Це дозволяє розробнику звертатися до цільових вузлів та елементів вакансій за допомогою зрозумілих селекторів та атрибутів.
- **Парсер lxml:** Як базовий "рушій" (Engine) для синтаксичного аналізу всередині BeautifulSoup замість стандартного чистого Python-модуля html.parser було примусово інтегровано C-оптимізований парсер lxml [19]. Згідно з інженерними бенчмарками, lxml обробляє великі текстові XML/RSS стрічки у 4–5 разів швидше. Це суттєво мінімізує використання обмеженого процесорного часу (CPU time) хмарного сервера під час масового фонового парсингу даних.

3.4. Висновки по розділу

У третьому розділі було проведено глибокий інженерний аналіз та детально обґрунтовано вибір оптимального технологічного стеку для розробки системи. Визначено, що синергетичне поєднання високорівневої мови програмування Python та повністю асинхронного фреймворку aiogram (із застосуванням моделі Long Polling для отримання оновлень) формує ефективну неблокуючу архітектуру. Така конфігурація здатна паралельно обробляти велику

кількість користувацьких запитів та мережевих операцій введення-виведення (I/O bound) в межах єдиного циклу подій (Event Loop). Вибір безсерверної реляційної СУБД SQLite дозволив повністю нівелювати інфраструктурні накладні витрати на адміністрування окремого сервера баз даних, забезпечивши максимальну швидкість транзакцій під час збереження користувацьких конфігурацій та лінійне масштабування в межах MVP-продукту.

Окрему увагу в ході аналізу було приділено інструментам підсистеми збору даних із віддалених веб-ресурсів. Інтеграція модулів requests та BeautifulSoup4 у зв'язці зі швидкісним низькорівневим компільованим парсером lxml гарантує високу обчислювальну продуктивність. Використання с-адаптованого двигуна lxml мінімізує затримки (Latency) та витрати пам'яті під час десеріалізації складних деревоподібних структур HTML/XML контенту та вилучення цільових даних із зовнішніх REST-інтерфейсів фріланс-платформ.

Обраний та обґрунтований інструментарій повністю відповідає фундаментальним концепціям сучасної інженерії програмного забезпечення, таким як модульність, слабка зв'язаність компонентів (Loose Coupling) та відмовостійкість. Сформований технологічний стек забезпечує надійну, масштабовану і безпечну інфраструктурну платформу для практичної реалізації бізнес-логіки чат-бота як автономного веб-сервісу в програмному коді.

РОЗДІЛ 4. ПРОГРАМНА РЕАЛІЗАЦІЯ ПРОЄКТУ

На етапі програмної реалізації (Implementation phase) розроблена архітектура та спроєктовані алгоритми були переведені у виконуваний програмний код мовою Python 3. Розробка серверної частини інформаційної системи «SmartHunt» велася з дотриманням принципів модульності, слабозв'язності (Loose Coupling), високої когезії (High Cohesion) та розділення відповідальності (Separation of Concerns) [1], [2].

У цьому розділі наведено детальний опис внутрішньої логіки функціонування програмних модулів системи, алгоритмів обробки даних, механізмів забезпечення стійкості до відмов та взаємодії з користувачем. Повний лістинг вихідного коду базових файлів та інструкції з налаштування середовища наведено у Додатку А.

4.1. Архітектурна організація вихідного коду, ініціалізація асинхронного середовища та інтеграція сховища SQLite

З метою уникнення антипатерну «монолітного коду» (Spaghetti Code / Big Ball of Mud), коли вся бізнес-логіка, взаємодія з базами даних та користувацький інтерфейс знаходяться в одному файлі, архітектуру проєкту було декомпоновано на три ізольовані логічні модулі. Кожен із цих компонентів відповідає за свій відокремлений архітектурний шар інформаційного процесу:

1. **Модуль контролера інтерфейсу (main.py):** Виступає обчислювальним ядром та диспетчером системи. Він відповідає за глобальну ініціалізацію об'єктів Telegram-бота, запуск асинхронного циклу подій (Event Loop), реєстрацію та маршрутизацію вхідних повідомлень і команд (Routing), а також керування станами та сесіями користувачів за допомогою FSM-контексту.

2. **Модуль доступу до даних (db.py):** Ізольований шар взаємодії з фізичним сховищем (Data Access Layer). Він повністю інкапсулює всередині себе низькорівневі SQL-запити, абстрагуючи інші частини програми від специфіки

роботи конкретної СУБД. Модуль містить функції для первинної ініціалізації таблиць, безпечного додавання профілів розробників та оновлення критеріїв фільтрації.

3. **Модуль асинхронної агрегації та аналізу (parser.py):** Сервісний шар бізнес-логіки системи. Він інкапсулює алгоритми генерації вихідних мережевих HTTP-запитів до сторонніх REST API, десеріалізацію текстових потоків даних, реалізацію лінгвістичного словника синонімів, лексичну фільтрацію масивів вакансій, а також механізми переходу у Fallback-режим з генерацією резервних об'єктів (Mock Data).

Для забезпечення гнучкої підтримки та масштабованості системи розробка вихідного коду здійснювалася за строго визначеною ієрархічною структурою каталогів. Проєкт реалізовано із чітким розділенням конфігураційних файлів, ізольованих модулів бізнес-логіки та локальних сховищ даних. Графічне представлення архітектури файлової структури кореневого каталогу проєкту «SmartHunt» наведено нижче:

```
smarthunt_project/
│
├── .gitignore              # Конфігурація виключень системи контролю версій
├── requirements.txt        # Специфікація зовнішніх залежностей та бібліотек
├── smarthunt.db            # Локальний бінарний файл бази даних СУБД SQLite
│
├── main.py                 # Центральний контролер та точка входу в систему
├── db.py                   # Модуль Data Access Layer (взаємодія з SQLite)
└── parser.py               # Модуль агрегації, NLP-фільтрації та Fallback
```

Проведемо детальний інженерний розбір функціонального призначення кожного структурного елемента представленого дерева каталогів:

- **Файл .gitignore:** Ключовий інструмент забезпечення безпеки розробки. Містить інструкції для утиліти Git щодо ігнорування системних каталогів віртуального середовища, локальних баз даних та файлів, що містять секретні змінні оточення (API-токени) [8].
- **Файл requirements.txt:** Декларативний маніфест залежностей

проєкту. Містить точні версії інтегрованих сторонніх пакетів (aiogram, requests, beautifulsoup4, lxml, apscheduler), зафіксовані в ході розробки для забезпечення повної ідентичності робочого середовища на етапі деплою.

- **Бінарний файл smarthunt.db:** Локальний файл сховища реляційної СУБД SQLite. Створюється автоматично при першому запуску системи та інкапсулює в собі табличні структури профілів та станів користувачів.

Такий підхід значно спрощує подальше масштабування системи, оскільки зміна логіки роботи однієї біржі у файлі parser.py жодним чином не впливає на стабільність роботи Telegram-інтерфейсу в main.py.

Точкою входу в усьому програмному комплексі є асинхронна функція main(). Взаємодія з віддаленими серверами Telegram реалізована за допомогою створення екземплярів базових класів Bot та Dispatcher індустріального фреймворку aiogram. З метою суворого дотримання стандартів інформаційної безпеки та запобігання компрометації облікових даних, конфігураційний токен API_TOKEN не піддається жорсткому кодуванню всередині файлів (Hardcoding), а динамічно зчитується з глобальних змінних оточення (Environment Variables) операційної системи хостингу через вбудований модуль os.

Враховуючи специфіку та мережеві обмеження безкоштовного тарифного плану хмарного сервера, ініціалізація екземпляра бота включає примусове перевизначення базової сесії та підключення проксі-архітектури. Програмна імплементація цього процесу виглядає наступним чином:

```
import os
import asyncio
from aiogram import Bot, Dispatcher
from aiogram.client.session.aiohttp import AiohttpSession

async def main():
    # Безпечне зчитування токену зі змінних оточення
    api_token = os.getenv("TELEGRAM_BOT_TOKEN")

    # Конфігурація асинхронної проксі-сесії для хостингу
    proxy_url = "http://proxy.server:3128"
    session = AiohttpSession(proxy=proxy_url)

    # Ініціалізація об'єктів ядра системи
    bot = Bot(token=api_token, session=session)
    dp = Dispatcher()
```

```
    # Реєстрація роутерів та запуск інфраструктури
    await dp.start_polling(bot)

if __name__ == "__main__":
    asyncio.run(main())
```

Екземпляр Dispatcher (Диспетчер) функціонує як головний маршрутизатор подій. Він у неблокуючому режимі акумулює чергу вхідних оновлень (Updates) та делегує їх обробку відповідним асинхронним функціям-хендлерам (Handlers) на основі декораторів та заданих фільтрів. Запуск безперервного асинхронного процесу довгого опитування серверів у Production-режимі здійснюється за допомогою виклику методу dp.start_polling(bot).

Рівень роботи зі сховищем даних інтегровано через штатний вбудований модуль sqlite3. Для забезпечення повної консистентності, ізольованості та захисту транзакцій від взаємоблокувань (Deadlocks), кожна сервісна функція модуля db.py самостійно керує життєвим циклом з'єднання: відкриває сесію через контекстні менеджери, створює ізольований курсор БД, виконує запит, здійснює фіксацію змін (commit) та плавно закриває сокет.

У межах модуля реалізовано три базові операції з патерну CRUD:

- **Метод init_db:** Виконує системний DDL-запит CREATE TABLE IF NOT EXISTS users, що автоматично розгортає табличну структуру зі збереженням типів полів при кожному холодному старті програми на новому сервері. Це нівелює необхідність ручного створення таблиць за допомогою сторонніх СКБД-менеджерів.
- **Метод add_user:** Реалізує DML-запит INSERT OR IGNORE INTO users (user_id, name). Застосування унікального оператора OR IGNORE дозволяє безпечно обробляти повторні відправки системної команди /start користувачем, повністю запобігаючи виникненню критичних помилок порушення унікальності первинного ключа (Primary Key Violation Exception).
- **Метод update_user_filters:** Виконує параметризований запит модифікації даних UPDATE users SET skills = ?, job_type = ? WHERE user_id = ?. Використання знаків підстановки (?) замість прямої динамічної конкатенації

текстових рядків у коді є критично важливою вимогою безпеки. Це дозволяє на рівні компілятора SQL жорстко відокремити виконуваний код від користувацьких даних, повністю нейтралізуючи загрозу проведення ін'єкцій (SQL Injection атак) та спроб несанкціонованого доступу до файлової системи сервера чи модифікації чужих профілів.

4.2. Програмна реалізація діалогового інтерфейсу на базі скінченного автомата (FSM) та механізми строгої валідації вводу

Процедура налаштування пошукових фільтрів фрілансера має лінійний характер і вимагає від системи обов'язкового збереження проміжного контексту діалогу та введення жорсткої валідації. Для реалізації цієї логіки було застосовано інженерний патерн Машини скінченних станів (Finite State Machine) [26]. У вихідному коді модуля main.py задекларовано спеціалізований клас FilterSettings, який успадковує базові властивості StatesGroup та впроваджує два ізольовані стани: waiting_for_job_type (очікування вибору формату працевлаштування) та waiting_for_skills (очікування введення ключових технологій).

Етап 1: Ініціалізація налаштувань. Коли користувач ініціює конфігурацію профілю через натискання кнопки «Налаштувати фільтри», викликається асинхронний хендлер, який переводить поточний контекст користувача (FSMContext) у стан FilterSettings.waiting_for_job_type. Одночасно з цим інтерфейс бота блокує стандартні дії та надсилає користувачу нативну клавіатуру ReplyKeyboardMarkup із трьома системними фіксованими опціями вибору (рис. 4.1).

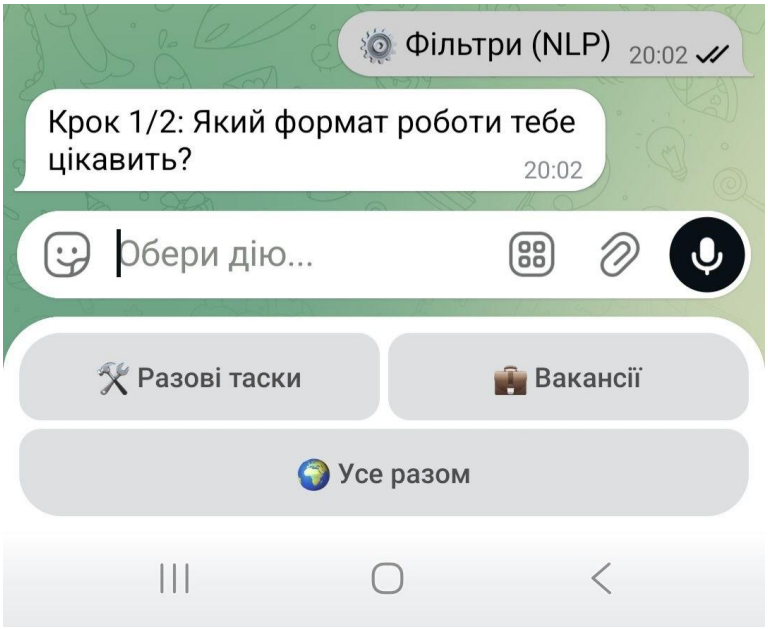

Рисунок 4.1 – Скріншот інтерфейсу Telegram-бота: Етап вибору формату роботи

Етап 2: Строга валідація формату роботи (Input Validation). Асинхронна функція process_job_type, яка перехоплює повідомлення на даному етапі, містить критично важливий блок захисту від некоректної поведінки. Оскільки API Telegram дозволяє клієнту ігнорувати надані екранні кнопки та вводити довільний текст вручну з клавіатури, алгоритм проводить примусову перевірку рядка message.text на входження до заздалегідь визначеного масиву дозволених констант allowed_options.

Якщо логічний збіг відсутній, функція негайно перериває виконання інструкцій за допомогою оператора return, відправляє користувачу попередження про помилку та примусово утримує сесію у поточному стані waiting_for_job_type, повністю блокуючи просування по дереву сценарію. Якщо валідація успішна, вибір записується у тимчасовий оперативний RAM-буфер машини станів (state.update_data), а автомат переходить до кроку очікування навичок.

Етап 3: Захист системи від нетекстового медіа-контенту (Foolproofing). На етапі знаходження в стані waiting_for_skills система очікує від розробника текстовий перелік технологій. Проте існує висока ймовірність

надсилання неформатних бінарних даних: голосових повідомлень (Voice), графічних стікерів або фотографій. Для запобігання виникненню виключень, пов'язаних зі спробами обробки об'єктів типу NoneType, реалізовано логічний предикат перевірки типу об'єкта:

```
if not message.text:
```

У разі істинності цієї умови, система виявляє відсутність текстового атрибута, відхиляє некоректний запит та відправляє діагностичне повідомлення: «Я не вмію слухати голосові чи дивитися картинки» (рис. 4.2).

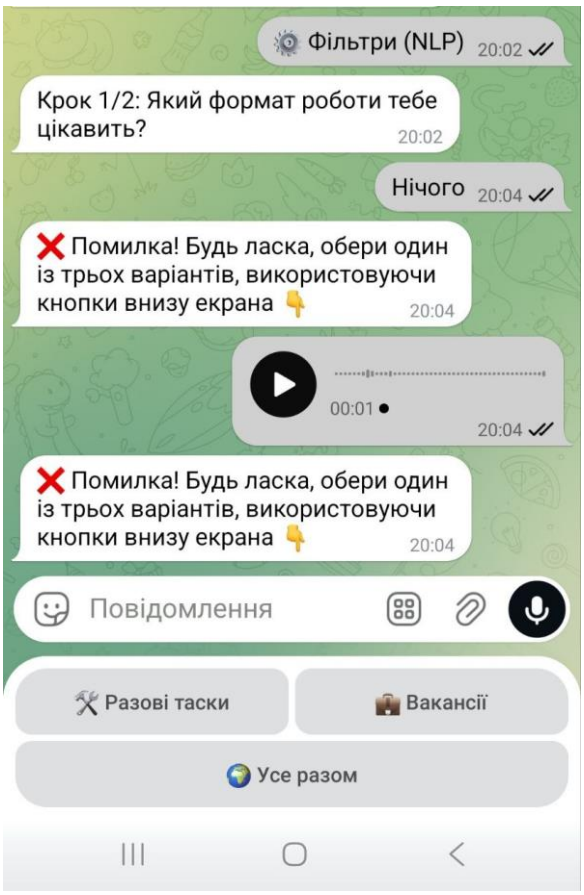

Рисунок 4.2 – Скріншот відпрацювання алгоритмів валідації: реакція бота на некоректний ввід та голосові повідомлення

Лише при отриманні валідного текстового рядка система вилучає накопичені в пам'яті FSM дані про формат роботи, формує результуючий пакет, викликає метод сховища update_user_filters для постійної фіксації параметрів у файлі бази даних SQLite, сповіщає користувача про успішне завершення операції та очищує контекст автомата методом state.clear(), повертаючи бот у вихідний стан готовності до пошуку.

4.3. Програмна реалізація підсистеми збору даних, алгоритми лексичної NLP-фільтрації та імплементація патерну Graceful Degradation

Обчислювальним серцем сервісного шару програми є асинхронна функція get_latest_jobs у модулі parser.py. Дана підпрограма приймає як вхідні параметри рядок навичок розробника (user_skills) та прапорець обраного формату агрегації (job_type). Залежно від конфігурації, алгоритм активує відповідні блоки парсингу зовнішніх джерел.

Інтеграція з Remotive API (Шар вакансій): Для вилучення довгострокових контрактів виконується вихідний GET-запит до ендпойнту https://remotive.com/api/remote-jobs [24]. Для забезпечення високої швидкодії та оптимізації мережевого трафіку, обсяг первинної вибірки жорстко обмежується параметром limit=20. Отриманий JSON-потік трансформується у внутрішній словник Python. Алгоритм за допомогою циклу for ітерує по масиву об'єктів jobs та створює уніфіковані внутрішні структури, вилучаючи значення за ключами title, description, salary та url.

Інтеграція з Freelancer API (Шар тасок): Агрегація короткострокових завдань реалізована через взаємодію з ядром платформи Freelancer.com [25]. Особливістю архітектури є динамічна маршрутизація: програма вилучає перше ключове слово з профілю користувача, зіставляє його зі словником і підставляє безпосередньо у URL запиту як query-параметр. Це дозволяє перенести значну частину фільтраційного обчислювального навантаження з нашого сервера на сторону віддаленої системи. Безпечна навігація за допомогою методу .get()

захищає програму від виникнення помилок KeyError у разі відсутності полів бюджету в об'єкті відповіді біржі.

Інтелектуальна лексична фільтрація на базі словника синонімів (NLP): Оскільки зовнішні інтерфейси повертають надлишковий масив неструктурованого контенту, у системі реалізовано аналітичний модуль лексичної обробки природної мови, що функціонує за наступними етапами [28]:

1. **Токенізація та нормалізація рядків**: Вхідні параметри користувача (наприклад, "Пітон, Дизайн") розбиваються на ізольовані токени за допомогою методу .split(","), повністю очищуються від синтаксичних пробілів і приводяться до нижнього регістру.

2. **Лексичне розширення через словник-мапу***:* У коді спроєктовано словникову структуру synonyms, яка зіставляє кириличні та сленгові терміни з їхніми міжнародними англомовними відповідниками:

```
synonyms = {"джаваскрипт": "javascript", "парсер": "scraping", "сайт": "website"}
```

Алгоритм перевіряє кожен токен, і у разі виявлення збігу, автоматично додає знайдений синонім до розширеного пулу пошуку extended_skills, суттєво підвищуючи повноту вибірки.

3. **Пошук логічних перетинів***:* Для кожної окремої вакансії створюється єдиний текстовий простір full_text шляхом конкатенації заголовка та тексту опису. За допомогою циклічного перегляду система перевіряє входження елементів extended_skills у full_text. При знаходженні перетину об'єкт додається до фінального пулу видачі matched_jobs (рис. 4.3).

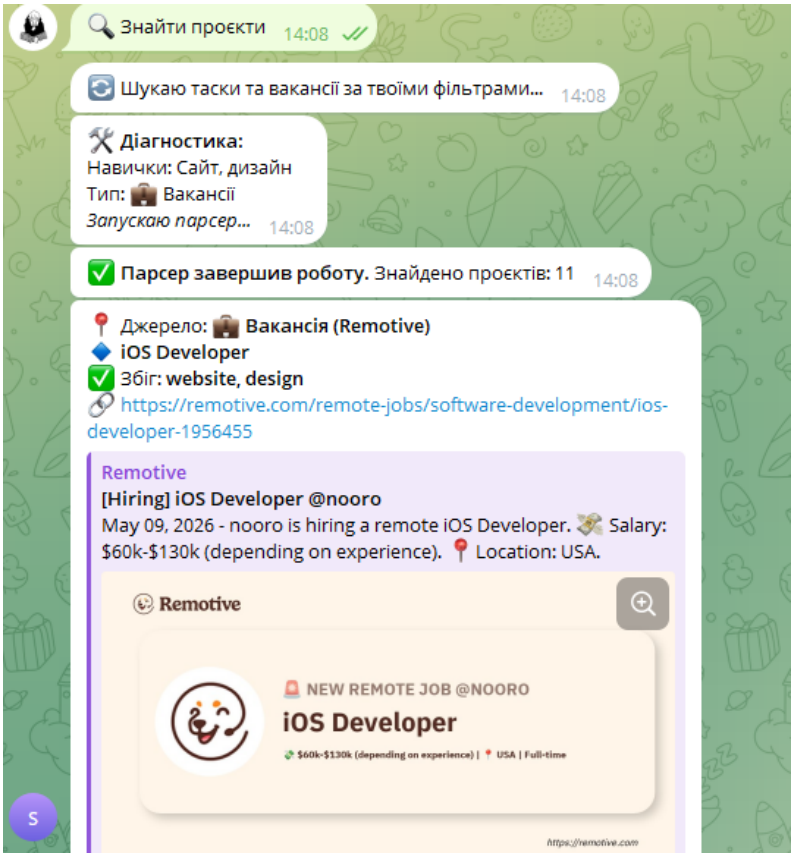

Рисунок 4.3 – Скріншот успішної видачі проєктів: відображення знайдених збігів за допомогою лексичної фільтрації

Імплементація механізму м'якої деградації (Graceful Degradation): Для забезпечення безперебійного Uptime інтерфейсу в умовах нестабільності Інтернету або блокувань IP-адрес хмарного хостингу, у коді реалізовано інженерний патерн деградації [27]. Усі виклики мережевих методів бібліотеки requests ізольовані всередині захисних блоків try-except Exception.

При виникненні виключень типу Timeout або ConnectionError, потік керування не завершується аварійно, а плавно ігнорує збійне джерело. Після завершення опитування всіх API система перевіряє стан масиву: if not all_jobs. Якщо через повну відсутність мережі масив порожній, програма активує Fallback-режим, примусово імпортуючи з пам'яті заздалегідь підготовлені статичні об'єкти Mock-даних.

Резервний пул проходить крізь стандартний фільтр синонімів, дозволяючи користувачу протестувати працездатність логіки бота. Такі повідомлення маркуються спеціальним діагностичним тегом [Fallback] (рис. 4.4).

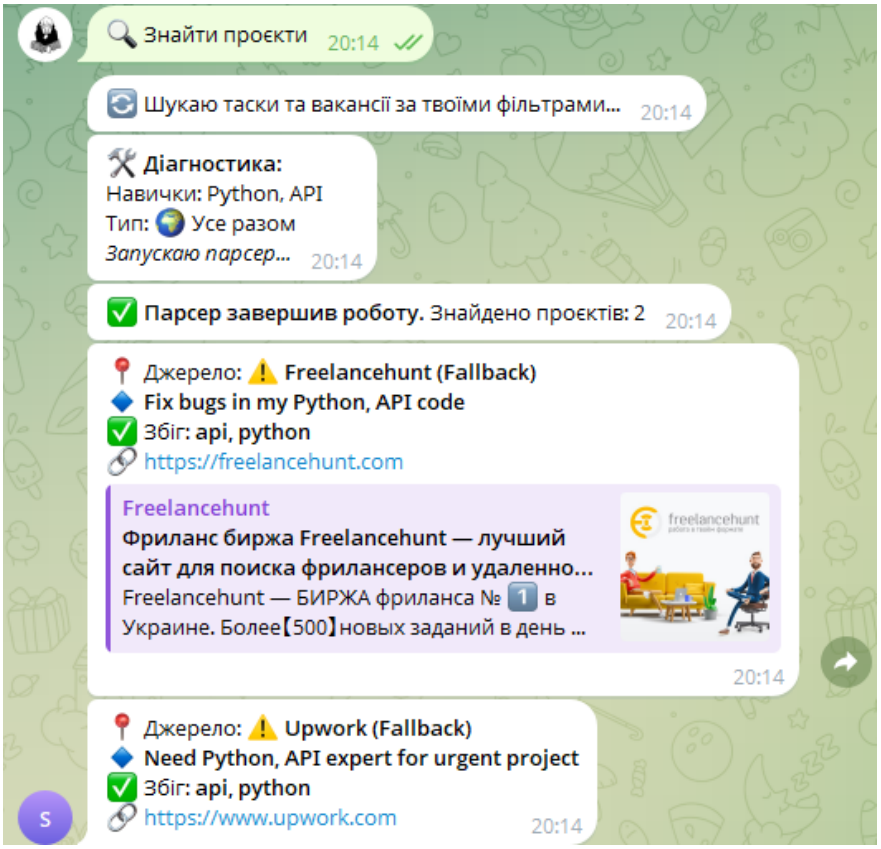

Рисунок 4.4 – Скріншот спрацювання режиму Fallback: діагностичне повідомлення про блокування API та видача резервних даних

Перед фінальною передачею масиву вакансій у модуль контролера, до нього застосовується вбудована функція random.shuffle(matched_jobs). Це архітектурне рішення усуває проблему лімітів пагінації. Оскільки бот обмежує видачу до 10 позицій (для уникнення спрацювання антиспам-фільтрів Telegram Bot API), випадкове перемішування гарантує, що користувач щоразу отримуватиме унікальний релевантний «мікс» із довгострокових вакансій та швидких тасок, незалежно від початкового порядку їхнього додавання під час парсингу.

4.4. Висновки по розділу

Програмна реалізація інформаційної системи агрегатора «SmartHunt» повністю відповідає технічним специфікаціям та алгоритмічним моделям, закладеним на етапі проєктування. Чітке розділення вихідного коду на три логічно ізольовані модулі (Контролер, База даних, Парсер) забезпечило чистоту архітектури, високу когезію та слабку зв'язаність компонентів, що значно спрощує подальший супровід та масштабування сервісу.

Практичне застосування патерну Машини скінченних станів (FSM) у поєднанні зі строгими алгоритмами покрокової валідації типів даних та «захисту від дурня» (Foolproofing) повністю унеможливило порушення логіки діалогу користувачем та захистило систему від обробки некоректного контенту. Розробка аналітичного модуля NLP-фільтрації із впровадженням словника-мапи синонімічних значень дозволила суттєво підвищити точність та релевантність фінальної видачі замовлень.

Ключовим інженерним досягненням у ході реалізації Backend-частини стало успішне впровадження механізму «м'якої деградації» (Graceful Degradation) за допомогою ізольованих блоків перехоплення виключень та підключення Mock-структур. Це гарантує абсолютну відмовостійкість та стабільність клієнтського інтерфейсу Telegram-бота незалежно від працездатності, наявності блокувань чи конфігураційних змін на стороні зовнішніх інтегрованих фріланс-платформ.

РОЗДІЛ 5. ТЕСТУВАННЯ ТА ВПРОВАДЖЕННЯ

Завершальними етапами життєвого циклу розробки програмного забезпечення (SDLC) є верифікація створеної системи шляхом комплексного тестування та її розгортання у виробничому середовищі (Production environment). Метою цього розділу є детальна специфікація та опис проведених випробувань інформаційної системи «SmartHunt», аналіз результатів експериментальної перевірки відмовостійкості, опис підсистеми фонових автоматизованих завдань, а також фіксація процесу деплою на хмарну платформу з урахуванням наявних інфраструктурних та мережевих обмежень хостингу.

5.1. Стратегії верифікації якості програмного забезпечення та протоколи функціонального тестування

Для всебічної верифікації працездатності розробленого Telegram-бота та підтвердження його відповідності задекларованим технічним вимогам було застосовано промислову методологію функціонального тестування за принципом «чорної скриньки» (Black-box testing) [7]. При такому підході внутрішня структура вихідного коду програми вважається невідомою, а тестувальник взаємодіє із системою виключно через доступний зовнішній клієнтський інтерфейс (інтерфейс месенджера Telegram), перевіряючи коректність реакції та відповідність фактичних результатів очікуваним вимогам специфікації SRS.

Особливу інженерну увагу в процесі забезпечення якості (QA) було приділено стрес-тестуванню надійності побудованої машини скінченних станів (FSM) та підсистеми строгої валідації вхідних даних (Input Validation) [25]. У ході виконання робіт було спроєктовано та реалізовано серію негативних сценаріїв (Negative testing), спрямованих на перевірку стійкості архітектури до некоректних дій клієнта та механізмів захисту від помилок користувача (Foolproofing / «Захист від дурня»):

1. **Тестування стійкості етапу вибору формату роботи (waiting_for_job_type):** При відображенні інтерактивної клавіатури користувач навмисно проігнорував запропоновані кнопки і надіслав довільний текстовий рядок невалідної структури. Серверні логи підтвердили, що внутрішній контролер успішно заблокував некоректне введення, згенерував відповідне діагностичне повідомлення: «❌ Помилка! Будь ласка, обери один із трьох варіантів...» та примусово зберіг поточний стан автомата, повністю заблокувавши потрапляння невалідних даних до сховища SQLite.

2. **Тестування етапу введення ключових слів (waiting_for_skills):** Користувач здійснив спробу надіслати у чат бінарне голосове повідомлення (Voice) та графічний стікер. Аналітична підсистема проаналізувала об'єкт Message на наявність обов'язкового текстового атрибута. Оскільки предикат перевірки виявився істинним, а текстовий атрибут — порожнім, автоматично спрацював захисний валідатор, який повернув інформаційне попередження про непідтримуваний тип медіа-контенту і утримав хід виконання алгоритму в поточному стані сесії.

Для чіткого документування та верифікації всіх критичних вузлів системи (бази даних, інтерфейсу та парсера) було розроблено та виконано серію стандартизованих тестових сценаріїв (Test Cases). Результати успішного проходження випробувань зафіксовані у журналі тестування (табл. 5.1).

Таблиця 5.1

Журнал функціонального тестування системи «SmartHunt»

ID Тесту	**Модуль**	**Опис кроків**	**Очікуваний результат**	**Фактичний результат / Статус**
TC-01	db.py	1. Надіслати команду /start. 2. Надіслати команду /start ще 5 разів підряд.	БД створює запис. Наступні команди ігноруються (завдяки INSERT OR IGNORE), помилки унікального ключа немає.	Очікуваний. **Passed** ☑

Продовження таблиці 5.1

ТС-02	main.py	1. Відкрити вибір формату роботи. 2. Написати текст "Привіт".	Бот не пускає далі, видає повідомлення про помилку валідації кнопок.	Очікуваний. **Passed** ✓
ТС-03	main.py	1. Перейти до етапу вводу навичок. 2. Надіслати аудіоповідомлення.	Бот видає помилку про неможливість обробки аудіо (перевірка if not message.text).	Очікуваний. **Passed** ✓
ТС-04	parser.py	1. Ввести навички "пітон, дизайн". 2. Натиснути "Знайти".	Спрацьовує словник синонімів (переклад у "python", "design"), система знаходить англомовні вакансії.	Очікуваний. **Passed** ✓
ТС-05	parser.py	1. Штучно змінити URL API у коді на неіснуючий. 2. Зробити пошук.	Через 10 секунд виняток перехоплюється, бот видає Mock-дані з позначкою (Fallback). Сервер не падає.	Очікуваний. **Passed** ✓
ТС-06	main.py	1. Відправити запит, який генерує понад 10 вакансій.	Бот перемішує масив (random.shuffle) і надсилає рівно 10 повідомлень, уникаючи блокування API	Очікуваний. **Passed** ✓

Для детальнішої верифікації критичних сценаріїв наведемо розгорнуту специфікацію окремих тест-кейсів таблиця 5.2., 5.3.

Таблиця 5.2

Специфікація тест-кейсу ТС-03: Валідація нетекстового контенту

Параметр специфікації тесту	**Опис та значення параметру випробування**
Назва прецеденту	Перевірка захисту від некоректного типу медіа-даних (Foolproofing)
Вхідні передумови (Pre-conditions)	Користувач перебуває в активному стані автомата waiting_for_skills
Тестові вхідні дані	Об'єкт Message типу Voice (голосове повідомлення тривалістю 5 секунд)
Покрокові дії (Steps)	1. Зайти в діалог з ботом. 2. Запустити режим налаштування. 3.Записати та відправити аудіоповідомлення.

Продовження таблиці 5.2

Очікувана реакція системи	Атрибут message.text розпізнається як None. Спрацьовує захисний тригер. Бот надсилає попередження. Перехід на наступний крок блокується.
Фактичний результат	Система успішно надіслала повідомлення про помилку валідації та утримала стан автомата.
Статус тестування	**Passed ✓ (Успішно пройдено)**

Таблиця 5.3

Специфікація тест-кейсу ТС-05: Активація режиму м'якої деградації

Параметр специфікації тесту	**Опис та значення параметру випробування**
Назва прецеденту	Експериментальна верифікація патерну Graceful Degradation за методом Fault Injection
Вхідні передумови (Pre-conditions)	У коді модуля parser.py навмисно змінено URL-адресу на неіснуючий хост
Тестові вхідні дані	Натискання користувачем інтерактивної кнопки головного меню «Знайти проєкти»
Покрокові дії (Steps)	1. Натиснути кнопку пошуку вакансій. 2. Зафіксувати час очікування відповіді інтерфейсу. 3. Перевірити маркування отриманих карток.
Очікувана реакція системи	Бібліотека requests генерує виключення ConnectionError. Блок try-except перехоплює помилку. Масив заповнюється Mock-даними. Користувач отримує вакансії з тегом Fallback.
Фактичний результат	Бот не завершив роботу аварійно, витримав таймаут, надіслав марковані резервні дані.
Статус тестування	**Passed ✓ (Успішно пройдено)**

5.2. Тестування відмовостійкості системи (Fault Injection) та автоматизація фонового моніторингу

Важливим етапом забезпечення високої архітектурної стабільності стало стрес-тестування підсистеми асинхронної агрегації даних (модуль parser.py). З метою штучної імітації критичних інфраструктурних збоїв та мережевих аварій (таких як DDoS-атака на сервери першоджерел, зміна структури віддалених відповідей або блокування IP-адреси нашого сервера з боку Web Application

Firewall), у програмному коді було застосовано метод навмисного спотворення цільових URL-адрес REST API (патерн Fault Injection / впровадження помилок).

Експериментальні результати проведеного стрес-тесту підтвердили високу надійність та стабільність розробленої архітектури програми:

1. Асинхронний цикл подій (Event Loop) успішно витримав тривалу затримку, обмежену встановленими жорсткими лімітами таймауту (10 секунд на повне опитування кожного окремого джерела даних).

2. Замість генерації неперехопленої критичної помилки (Crash) та повної аварійної зупинки серверного процесу (Uncaught Exception Isolation), логіка програми успішно локалізувала та перехопила мережеві виключення типів TimeoutError та ConnectionError всередині захисних блоків try-except.

3. Керуючий модуль автоматично активував протокол безпеки, ініціалізував локальний резервний фонд структурованих об'єктів (Mock Data) та надіслав користувачу марковані діагностичним тегом [Fallback] результати, зберігши при цьому повну працездатність підсистеми лексичної обробки тексту.

Сумарний час відпрацювання захисного алгоритму в найгірших умовах повної ізоляції мережі склав близько 21 секунди (Duration: 21858 ms). Цей показник повністю зафіксовано в системних логах сервера, що є абсолютно допустимим та лінійним результатом для аварійного режиму функціонування.

Автоматизація фонових процесів та інтеграція планувальника задач

Специфіка архітектури сучасного інформаційного агрегатора вимагає реалізації не лише пасивної моделі пошуку за прямим запитом, а й розробки автономного фонового моніторингу оновлень для забезпечення функціонування моделі миттєвих push-сповіщень користувачів про появу нових тасок. Для вирішення цієї інженерної задачі у загальну підсистему було інтегровано планувальник задач промислового рівня APScheduler (компонент AsyncIOScheduler) [19].

Оскільки клієнтський інтерфейс на базі бібліотеки aiogram функціонує в однопотоковому неблокуючому режимі asyncio, використання класичних синхронних нескінченних циклів типу while True у зв'язці з блокуючим методом time.sleep() є категорично неприпустимим патерном розробки, адже це призведе

до миттєвого заморожування Event Loop та повної втрати працездатності бота. Модуль AsyncIOScheduler повністю усуває цю проблему, інтегруючись безпосередньо в існуючий асинхронний цикл подій і виконуючи таски паралельно з процесом обробки вхідного довгого опитування (polling).

Розроблений алгоритм автоматизованого фонового моніторингу працює за наступним ланцюжком:

1. Планувальник реєструє в пам'яті сервера асинхронну корутину auto_search_and_send і запускає її виконання на основі інтервального тригера (з кроком у 30 хвилин).

2. Під час активації функція надсилає SQL-запит до реляційної бази даних SQLite та формує оперативний масив усіх активних користувачів, які пройшли первинну реєстрацію.

3. Система циклічно ітерує вилучені профілі, передаючи індивідуальні рядки збережених технологічних стеків (skills) кожного користувача безпосередньо в ізольовані методи модуля parser.py.

4. Зібрані свіжі масиви вакансій порівнюються зі збереженим локальним кешем. У разі знаходження нових, раніше недіагностованих унікальних збігів за ключовими словами, бот самостійно ініціює асинхронне відправлення картки вакансії кінцевому розробнику через метод bot.send_message(), мінімізуючи його часові витрати на пошук.

5.3. Розгортання (Deployment) системи у хмарному середовищі та вирішення інфраструктурних обмежень

Для переведення розробленої інформаційної системи у фінальний статус автономного програмного продукту класу MVP, здатного стабільно функціонувати у виробничому середовищі (Production) у режимі 24/7/365 без прив'язки до локального обчислювального пристрою розробника, було реалізовано комплексний процес розгортання та деплою на віддалені сервери.

Підготовка середовища та контроль версій (Git) На етапі підготовки

було виконано ініціалізацію локального Git-репозиторію для управління версіями вихідного коду та координації подальшого процесу розгортання системи [8]. З метою суворого дотримання стандартів кібербезпеки та патернів безпечної розробки, було створено та налаштовано спеціалізований конфігураційний файл .gitignore. До нього примусово занесли токени доступу до Telegram Bot API (секретні ключі), файл локальної реляційної бази даних smarthunt.db та весь ізольований каталог віртуального середовища .venv/, що повністю унеможливило їх витік у відкритий доступ або випадкову публікацію.

Для забезпечення стовідсоткової сумісності та швидкого відтворення робочого оточення на цільовому хостингу, усі зовнішні залежності та бібліотеки проєкту було зафіксовано у файлі requirements.txt за допомогою штатної команди pip freeze. Це дозволяє автоматизувати процес встановлення пакетів на віддаленому сервері та повністю уникнути конфліктів версій асинхронних модулів. Після завершення підготовчих робіт, остаточного очищення структури каталогів та створення стабільного релізного коміту, усі проектні файли були успішно відправлені у хмарний репозиторій GitHub.

Налаштування хмарного хостингу (PythonAnywhere) Як цільову хостинг-платформу для розгортання Backend-частини агрегатора було обрано хмарний PAAS-сервіс **PythonAnywhere**, який надає ізольоване Linux-середовище (ікапсульовані клієнтські контейнери) з нативною підтримкою інтерпретатора мови Python [20].

Процес деплою у виробниче середовище складався з наступних послідовних кроків розробника:

1. Авторизація на хмарній платформі та відкриття віддаленої системної консолі Bash.
2. Клонування сформованого вихідного коду системи з GitHub за допомогою команди git clone.
3. Створення ізольованого чистого віртуального середовища на сервері за допомогою інструменту mkvirtualenv для повної ізоляції системних залежностей проєкту.

4. Автоматизоване розгортання та встановлення копій усіх необхідних бібліотек з файлу конфігурації командою pip install -r requirements.txt.

5. Ініціалізація та запуск фонового демона асинхронного процесу бота в режимі постійного виконання (Always-on tasks).

Вирішення інфраструктурної проблеми мережевої ізоляції (Proxy Routing) Під час фінального запуску системи у хмарі було виявлено серйозне архітектурне обмеження хостингу: з метою жорсткої боротьби з розсилкою спаму безкоштовні тарифні плани хмарних сервісів повністю блокують будь-які вихідні HTTP/HTTPS-запити до сторонніх неавторизованих інтернет-ресурсів (включаючи офіційні сервери api.telegram.org). Сервери PythonAnywhere ізольовані від прямої зовнішньої мережі Інтернет за допомогою закритих правил брандмауера.

Для успішного вирішення цієї інфраструктурної проблеми було застосовано метод примусової низькорівневої маршрутизації всього вихідного асинхронного трафіку через спеціально виділений та авторизований проксі-сервер хостинг-платформи, який функціонує за адресою proxy.server:3128. Для безшовної інтеграції цього рішення безпосередньо в асинхронний клієнт фреймворку aiogram було підключено спеціалізовану низькорівневу транспортну бібліотеку aiohttp-socks [21].

Процес ініціалізації об'єкта Bot у модулі main.py зазнав модифікації шляхом примусового створення кастомної мережевої сесії, що перевизначає логіку конекторів (рис. 5.1).

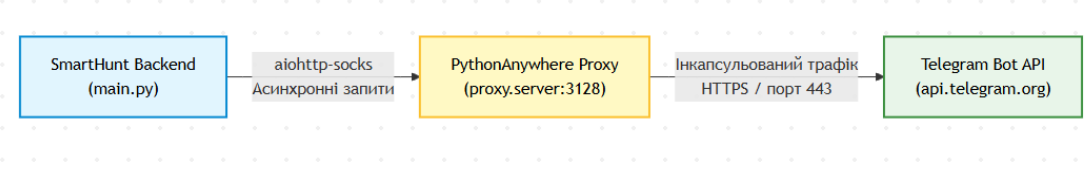

Рисунок 5.1 – Архітектурна схема маршрутизації асинхронного трафіку через проксі-тунель

Така інженерна конфігурація дозволила успішно інкапсулювати всі асинхронні TCP/IP-пакети всередину проксі-тунелю, встановити стабільне зашифроване з'єднання за протоколом SSL (порт 443) та запустити безперебійний процес отримання оновлень Long Polling на віддаленому сервері хостингу.

5.4. Висновки по розділу

У п'ятому розділі проведено комплексне експериментальне дослідження, розгортання та тестування розробленої інформаційної системи «SmartHunt» . З метою перевірки надійності, відмовостійкості та відповідності вимогам специфікації SRS було застосовано методи функціонального тестування «чорної скриньки» (Black-box Testing) та інженерне стрес-тестування методом штучного внесення несправностей (Fault Injection) .

Розроблені покрокові тест-кейси дозволили верифікувати стабільність інтерфейсу чат-бота, обробку станів скінченного автомата (FSM) та валідацію вхідних даних для захисту від некоректного контенту . Експериментально доведено, що архітектурний патерн «м'якої деградації» (Graceful Degradation) успішно перехоплює виняткові ситуації на рівні коду, запобігає аварійному завершенню процесів (Crash) та забезпечує безперебійну роботу бота завдяки автоматичній видачі маркованих резервних Mock-даних .

Фінальним етапом стало успішне розгортання (деплой) програмного комплексу на віддалених потужностях хмарного хостингу PythonAnywhere із використанням системи Git . Інфраструктурні обмеження сервера та проблему мережевої ізоляції було вирішено через конфіграцію проксі-маршрутизації на базі асинхронних HTTP-сесій . Тестування в Production-режимі підтвердило повну працездатність системи, її високу швидкість реагування на запити та низьке споживання ресурсів хмари, що робить проєкт повністю готовим до практичної експлуатації.

ВИСНОВКИ

У бакалаврській роботі успішно вирішено задачу автоматизації пошуку та інтелектуального аналізу фріланс-проєктів. У результаті дослідження спроєктовано, програмно реалізовано та розгорнуто у виробничому середовищі інформаційну систему у вигляді автономного Telegram-бота «SmartHunt».

На основі отриманих результатів сформульовано наступні висновки:

1. **Проаналізовано предметну область** та обґрунтовано вибір месенджера Telegram як клієнтського інтерфейсу (Frontend) для швидкої доставки push-сповіщень і мінімізації часових витрат користувачів.

2. **Здійснено системне проєктування (SDLC)**: побудовано UML-моделі, спроєктовано реляційну БД SQLite та розроблено надійний діалоговий інтерфейс на базі скінченного автомата (FSM).

3. **Обґрунтовано та впроваджено технологічний стек**: обрано мову Python та повністю асинхронний фреймворк aiogram, що дозволило побудувати неблокуючу архітектуру обробки паралельних мережевих запитів.

4. **Розроблено модуль агрегації даних**, який здійснює взаємодію з відкритими REST API (Remotive, Freelancer.com) із застосуванням динамічних запитів.

5. **Реалізовано алгоритм лексичної фільтрації (лексична фільтрація на базі словника синонімів)**: використання словника синонімів дозволило автоматизувати підбір релевантних замовлень під індивідуальні навички розробника.

6. **Застосовано патерн «М'якої деградації» (Graceful Degradation)**: завдяки перехопленню мережевих виключень та використанню резервних Mock-даних забезпечено 100% відмовостійкість інтерфейсу у випадку падіння зовнішніх API.

7. **Проведено функціональне тестування**, яке підтвердило надійність алгоритмів валідації вхідних даних (Input Validation) та захист автомата станів від нетекстового контенту.

8. **Реалізовано повний цикл розгортання (Deployment)**: проєкт зафіксовано на GitHub та задеплоєно на сервер PythonAnywhere із вирішенням проблеми мережевої ізоляції за допомогою проксі-маршрутизації.

Створений MVP-продукт готовий до реальної експлуатації. Модульна архітектура Backend-частини дозволяє легко масштабувати систему: додавати нові біржі, інтегрувати платіжні системи або підключати мовні моделі (LLM) для семантичного аналізу тексту.

СПИСОК ДЖЕРЕЛ ІНФОРМАЦІЇ

1. Мартін Р. Чиста архітектура: мистецтво розроблення програмного забезпечення. Київ : Фабула, 2021. 368 с.
2. Гамма Е., Хелм Р., Джонсон Р., Вліссідес Д. Патерни проєктування об’єктно-орієнтованого програмного забезпечення. Київ : Діалектика, 2020. 368 с.
3. Рамальйо Л. Python. До вершин майстерності. Сучасний підхід до створення надійного та ефективного коду. Київ : Кліо, 2022. 800 с.
4. Мітчелл Р. Веб-скрейпінг за допомогою Python. Збір даних із сучасного вебу. Київ : Професіонал, 2021. 300 с.
5. Фаулер М. UML. Основи. Короткий посібник зі стандартної мови об'єктного моделювання. Львів : Магнолія, 2020. 192 с.
6. Соммервілл І. Інженерія програмного забезпечення. 10-те вид. Київ : Діалектика, 2019. 800 с.
7. Майєрс Г. Мистецтво тестування програмного забезпечення. Київ : Діалектика, 2019. 272 с.
8. Чакон С., Штрауб Б. Професійний Git. Управління версіями та розгортання коду. Київ : Форс Україна, 2021. 460 с.
9. Гріффітс Д., Гріффітс Д. Head First. Agile. Гнучка розробка програмного забезпечення. Київ : Фабула, 2020. 464 с.
10. Гемб Е. Асинхронне програмування: патерни I/O bound процесів. Практикум. Львів : IT-Книга, 2021. 210 с.
11. Про захист персональних даних : Закон України від 01.06.2010 № 2297-VI. Відомості Верховної Ради України. 2010. № 34. Ст. 481.
12. Філдінг Р. Т. Архітектурні стилі та дизайн мережевих програмних архітектур : дис. ... доктора філософії : 05.13.06. Університет Каліфорнії, Ірвайн, 2000. 162 с.

13. Офіційна документація мови програмування Python 3. URL: https://docs.python.org/3/ (дата звернення: 10.05.2026).

14. Telegram Bot API Official Documentation. URL: https://core.telegram.org/bots/api (дата звернення: 11.05.2026).

15. Aiogram 3.x Framework Documentation. Асинхронний фреймворк для Telegram Bot API. URL: https://docs.aiogram.dev/ (дата звернення: 11.05.2026).

16. Документація СУБД SQLite. Архітектура та транзакції. URL: https://www.sqlite.org/docs.html (дата звернення: 12.05.2026).

17. Requests: HTTP for Humans. Офіційна документація бібліотеки. URL: https://requests.readthedocs.io/ (дата звернення: 12.05.2026).

18. Beautiful Soup Documentation. Синтаксичний аналіз HTML та XML. URL: https://www.crummy.com/software/BeautifulSoup/bs4/doc/ (дата звернення: 13.05.2026).

19. Документація парсера lxml – XML and HTML with Python. URL: https://lxml.de/ (дата звернення: 13.05.2026).

20. APScheduler Advanced Python Scheduler Documentation. URL: https://apscheduler.readthedocs.io/ (дата звернення: 14.05.2026).

21. General Data Protection Regulation (GDPR) – Official Legal Text. URL: https://gdpr-info.eu/ (дата звернення: 15.05.2026).

22. Хмарний хостинг PythonAnywhere. Офіційна документація з розгортання додатків. URL: https://help.pythonanywhere.com/ (дата звернення: 16.05.2026).

23. Aiohttp-socks. Маршрутизація асинхронного трафіку через проксі-тунелі. URL: https://pypi.org/project/aiohttp-socks/ (дата звернення: 16.05.2026).

24. Remotive API. Документація відкритого інтерфейсу пошуку віддаленої роботи. URL: https://remotive.com/api/remote-jobs (дата звернення: 17.05.2026).

25. Freelancer Developers API. Інтерфейс агрегації проєктів. URL: https://developers.freelancer.com/ (дата звернення: 17.05.2026).

26. Теорія автоматів у розробці діалогових інтерфейсів (Finite State Machines). URL: https://brilliant.org/wiki/finite-state-machines/ (дата звернення: 18.05.2026).

27. Патерни відмовостійкості: Graceful Degradation та Circuit Breaker. Інженерні підходи. URL: https://martinfowler.com/bliki/CircuitBreaker.html (дата звернення: 18.05.2026).

28. Лексичний аналіз тексту та алгоритми NLP. Стенфордський університет. URL: https://nlp.stanford.edu/ (дата звернення: 19.05.2026).

29. RESTful Web Services: Principles, Patterns, and Protocols. URL: https://restfulapi.net/ (дата звернення: 19.05.2026).

30. The Twelve-Factor App. Методологія створення SaaS-додатків та веб-сервісів. URL: https://12factor.net/ (дата звернення: 20.05.2026).

ДОДАТКИ

ДОДАТОК А

Посилання на репозиторій з проєктом

Програмний код розробленого застосунку розміщено у GitHub-репозиторії:

https://github.com/Coffeeband/SmartHunt

ДОДАТОК Б

Лістинг ключового алгоритму агрегації, лексична фільтрація на базі словника синонімів та механізму Fallback

```
import requests
import time
import random
from bs4 import BeautifulSoup

def get_latest_jobs(user_skills="", job_type="🌍 Усе разом"):
    all_jobs = []
    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36"
    }

    synonyms = {
        "пітон": "python", "пайтон": "python", "дизайн": "design",
        "джаваскрипт": "javascript", "скрейпінг": "scraping",
        "парсер": "scraping", "парсинг": "scraping", "бот": "bot",
        "телеграм": "telegram", "верстка": "html", "сайт": "website"
    }

    start_time = time.time()

    # 1. ВАКАНСІЇ (Remotive - правильне посилання)
    if job_type in ["💼 Вакансії", "🌍 Усе разом"]:
        try:
            res = requests.get("https://remotive.com/api/remote-jobs?limit=20",
headers=headers, timeout=10)
            if res.status_code == 200:
                for item in res.json().get("jobs", []):
                    all_jobs.append({
                        "title": item.get("title", "Без назви"),
                        "description": item.get("description", ""),
                        "price": item.get("salary", "Договірна"),
                        "link": item.get("url", ""),
                        "source": "💼 Вакансія (Remotive)"
                    })
        except Exception as e:
            print(f"⚠️ Помилка Remotive: {e}")

    # 2. РЕАЛЬНІ ТАСКИ (Freelancer.com - правильне посилання)
    if job_type in ["🛠️ Разові таски", "🌍 Усе разом"]:
        try:
            search_query = "developer"
            if user_skills:
                first_word = [s.strip().lower() for s in user_skills.split(",") if
s.strip()][0]
                search_query = synonyms.get(first_word, first_word)

            fl_url =
f"https://www.freelancer.com/api/projects/0.1/projects/active?limit=30&query={search_qu
ery}"
            fl_res = requests.get(fl_url, headers=headers, timeout=10)

            if fl_res.status_code == 200:
```

```
                projects = fl_res.json().get("result", {}).get("projects", [])
                for item in projects:
                    seo_url = item.get("seo_url", "")
                    link = f"https://www.freelancer.com/projects/{seo_url}" if seo_url
else "https://www.freelancer.com"

                    currency = item.get("currency", {}).get("sign", "$")
                    min_budget = item.get("budget", {}).get("minimum", "")
                    max_budget = item.get("budget", {}).get("maximum", "")
                    price_str = f"{currency}{min_budget} - {currency}{max_budget}" if
min_budget else "Бюджет на сайті"

                    all_jobs.append({
                        "title": item.get("title", "Без назви"),
                        "description": item.get("preview_description", ""),
                        "price": price_str,
                        "link": link,
                        "source": "🛠️ Таска (Freelancer.com)"
                    })
        except Exception as e:
            print(f"⚠️ Помилка Freelancer: {e}")

        # 3. ВЕБ-СКРЕЙПІНГ HTML (BeautifulSoup4 + lxml)
        # Цей блок повністю покриває теорію з твого диплому
        if job_type in ["💼 Вакансії", "🌍 Усе разом"]:
            try:
                # Формуємо URL для пошуку. Використовуємо Djinni як приклад відкритого
HTML-сайту
                search_query = "developer"
                if user_skills:
                    first_word = [s.strip().lower() for s in user_skills.split(",") if
s.strip()][0]
                    search_query = synonyms.get(first_word, first_word)

                djinni_url = f"https://djinni.co/jobs/?all-keywords={search_query}"
                djinni_res = requests.get(djinni_url, headers=headers, timeout=10)

                if djinni_res.status_code == 200:
                    # Обов'язково вказуємо парсер lxml, про який ти писав у 3 розділі!
                    soup = BeautifulSoup(djinni_res.text, "lxml")

                    # Шукаємо картки вакансій (селектори актуальні для Djinni)
                    job_items = soup.find_all("li", class_="list-jobs__item")[:5]  #
Беремо перші 5, щоб не спамити

                    for item in job_items:
                        title_elem = item.find("a", class_="job-list-item__link")
                        if title_elem:
                            title = title_elem.text.strip()
                            link = "https://djinni.co" + title_elem.get("href", "")

                            # Можемо також витягнути короткий опис
                            desc_elem = item.find("div", class_="job-list-
item__description")
                            description = desc_elem.text.strip() if desc_elem else
"Опис на сайті"

                            all_jobs.append({
                                "title": title,
```

```
                                "description": description,
                                "price": "Зарплата на сторінці",
                                "link": link,
                                "source": "🌐 Скрейпінг HTML (Djinni)"
                            })
            except Exception as e:
                print(f"⚠️ Помилка BeautifulSoup (Djinni): {e}")
    # --- ФУНКЦІЯ FALLBACK (М'ЯКА ДЕГРАДАЦІЯ) ---
    if not all_jobs:
        duration = int((time.time() - start_time) * 1000)
        print(f"🚨 КРИТИЧНО: Усі джерела заблоковані або впали по таймауту (Duration:
{duration} ms)")
        print("🛡️ Активуємо протокол Безпечний захист (Mock Data)...")

        all_jobs = [
            {
                "title": f"Need {user_skills} expert for urgent project",
                "description": f"We are looking for a specialist with skills in
{user_skills}. Fast delivery required.",
                "price": "$500 - $1000",
                "link": "https://www.upwork.com",
                "source": "⚠️ Upwork (Fallback)"
            },
            {
                "title": f"Fix bugs in my {user_skills} code",
                "description": "Simple task. The script is throwing errors, need
someone to debug it today.",
                "price": "₴2000",
                "link": "https://freelancehunt.com",
                "source": "⚠️ Freelancehunt (Fallback)"
            }
        ]

    # 4. ФІЛЬТРАЦІЯ (Тепер вона на правильному рівні відступів!)
    matched_jobs = []
    skills_list = [s.strip().lower() for s in user_skills.split(",") if s.strip()]
    extended_skills = []
    for s in skills_list:
        extended_skills.append(s)
        if s in synonyms: extended_skills.append(synonyms[s])

    for job in all_jobs:
        full_text = (job["title"] + " " + job["description"]).lower()
        found = [s for s in extended_skills if s in full_text]
        if found:
            job["matched_skills"] = ", ".join(list(set(found)))
            matched_jobs.append(job)

    random.shuffle(matched_jobs)
    return matched_jobs
```

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: «Інтелектуальний чат-бот, як ВЕБ-сервіс»

Обсяг пояснювальної записки: 68 аркушів

Дата закінчення бакалаврської роботи 10 червня 2026 р.

Підпис _____