МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри інтелектуальних інформаційних систем

_____ Євген СІДЕНКО

«____»_____ 20__ р.

КВАЛІФІКАЦІЙНА РОБОТА

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА

ІНФОРМАЦІЙНА СИСТЕМА ВЕБСКРЕПІНГУ НА ОСНОВІ ВИКОРИСТАННЯ MCP-ПРОТОКОЛУ

Спеціальність 122 Комп'ютерні науки

Освітня програма «Комп'ютерні науки»

Здобувач _____ Денис АДАМЕНКО

«___»_____ 2026 р.

Керівник канд. техн. наук, ст.викладач _____ Віктор ГОЖИЙ

«___»_____ 2026 р.

Миколаїв – 2026

Чорноморський національний університет імені Петра Могили

(повне найменування закладу вищої освіти)

Факультет	Комп’ютерних наук
Кафедра	Інтелектуальних інформаційних систем
Рівень вищої освіти	Перший (бакалаврський)
Освітній ступень	Бакалавр
Спеціальність	122 Комп’ютерні науки
Освітня програма	Комп’ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри інтелектуальних інформаційних систем

_____ Євген СІДЕНКО

«____» _____ 20__ р.

ЗАВДАННЯ
на кваліфікаційну роботу здобувача

Адаменка Дениса Олеговича

(прізвище, ім’я, по батькові здобувача)

1. Тема кваліфікаційної роботи: «Інформаційна система вебскрепінгу на основі використання MCP-протоколу».

Керівник роботи: Гожий Віктор Олександрович, старший викладач кафедри інтелектуальних інформаційних систем, канд. техн. наук.

Затверджена наказом ЧНУ ім. Петра Могили від «25» грудня 2025 р. № 353.

2. Строк представлення кваліфікаційної роботи «__» _____ 20__ р.

3. Очікуваним результатом роботи є розробка інформаційної системи вебскрапінгу на основі MCP-протоколу, яка забезпечує автоматизований збір, обробку та структуризацію даних із веб-ресурсів, зокрема вакансій із сайту work.ua та djinni.co, а також передавання результатів зовнішньому агенту через стандартизований MCP-інтерфейс.

4. Перелік питань, що підлягають розробці: аналіз сучасних підходів до вебскрапінгу; дослідження теоретичних засад MCP-протоколу та взаємодії LLM із зовнішніми інструментами; визначення вимог до інформаційної системи; проєктування архітектури MCP-інструменту; розробка програмних модулів для збору, парсингу, фільтрації та структурування даних із Work.ua; налаштування взаємодії агента з MCP-сервером; тестування працездатності, стабільності та продуктивності розробленої системи.

5. Перелік графічних матеріалів: презентація.

Керівник роботи _____ Віктор ГОЖИЙ

(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Денис АДАМЕНКО

(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата видачі завдання «21» грудня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

кваліфікаційної роботи

Тема: Інформаційна система вебскрепінгу на основі використання MCP-протоколу

№	Найменування роботи	Початок	Закінчення	Примітки
1	Отримання завдання на виконання КР	21.12.2025	24.12.2025	Виконано
2	Аналіз предметної області та постановка задачі	25.12.2025	30.01.2026	Виконано
3	Огляд літературних джерел за темою кваліфікаційної роботи, зокрема сучасних підходів до вебскрапінгу та збору даних із веб-ресурсів	31.01.2026	08.03.2026	Виконано
4	Дослідження теоретичних засад MCP-протоколу та підходів до інтеграції LLM із зовнішніми інструментами	09.03.2026	04.04.2026	Виконано
5	Проєктування архітектури системи та реалізація програмних модулів для збору, парсингу, фільтрації й структурування даних	05.04.2026	24.05.2026	Виконано
6	Перший попередній захист КР на засіданні комісії кафедри	25.05.2026	25.05.2026	Виконано
7	Корегування роботи за результатами попереднього захисту	26.05.2026	04.06.2026	Виконано
8	Другий попередній захист КР на засіданні комісії кафедри	05.06.2026	05.06.2026	Виконано
9	Доробка та остаточне оформлення КР	06.06.2026	14.06.2026	Виконано
10	Подання КР, її електронної копії та інших документів (відгуку, рецензії) до захисту	15.06.2026	19.06.2026	Виконано

Керівник роботи _____ Віктор ГОЖИЙ

(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Денис АДАМЕНКО

(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата складання календарного плану

«29» січня 2026 р.

АНОТАЦІЯ

до кваліфікаційної роботи
здобувача групи 401 ЧНУ ім. Петра Могили

Адаменка Дениса Олеговича

на тему: **“ІНФОРМАЦІЙНА СИСТЕМА ВЕБСКРЕПІНГУ НА ОСНОВІ ВИКОРИСТАННЯ MCP-ПРОТОКОЛУ”**

Актуальність роботи полягає у необхідності автоматизації збору даних із веб-ресурсів та інтеграції мовних моделей із зовнішніми інструментами за допомогою MCP-протоколу. Це дозволяє створювати гнучкі інформаційні системи для отримання, обробки та передавання структурованих даних агенту.

Об’єктом роботи є процес автоматизованого збору та структуризації даних із веб-ресурсів.

Предметом роботи є методи та програмні засоби реалізації інформаційної системи вебскрапінгу на основі MCP-протоколу.

Метою роботи є розробка інформаційної системи вебскрапінгу на основі MCP-протоколу та дослідження можливостей його використання для автоматизованого збору даних із веб-ресурсів.

У результаті виконання роботи було досліджено теоретичні основи Model Context Protocol, сучасні методи вебскрапінгу та засоби інтеграції Python із MCP. Розроблено інформаційну систему для автоматизованого збору вакансій із сайтів Work.ua та Djinni, яка забезпечує виконання HTTP-запитів, парсинг HTML-структур, фільтрацію даних і передавання результатів агенту через стандарт MCP.

Дана робота складається з чотирьох розділів. У першому розділі розглянуто предметну область, підходи до вебскрапінгу та MCP-протокол. У другому розділі визначено вимоги до системи й описано її архітектуру. У третьому розділі наведено програмну реалізацію MCP-сервера та модулів збору вакансій. У четвертому розділі проведено тестування, аналіз результатів і визначено обмеження системи.

Загальний обсяг роботи – 100 сторінок. Кваліфікаційна робота містить 6 додатків, 15 рисунків, 23 таблиці і 41 джерело у списку використаних джерел.

Ключові слова: інформаційна система, MCP-протокол, Python, вебскрапінг, Work.ua, Djinni, LLM, автоматизація, обробка даних.

ABSTRACT

to the qualification work by the student of the group 401 of Petro Mohyla Black Sea National University

Adamenko Denys

“ INFORMATION SYSTEM FOR WEB SCRAPING BASED ON THE USE OF THE MCP PROTOCOL”

The relevance of this work lies in the need to automate data collection from web resources and integrate language models with external tools using the MCP protocol.

The object of the work is the process of automated collection and structuring of data from web resources.

The subject of the work is the methods and software tools for implementing a web scraping information system based on the MCP protocol.

The purpose of the work is to develop a web scraping information system based on the MCP protocol and to study the possibilities of its use for automated data collection from web resources.

As a result of the work, the theoretical foundations of Model Context Protocol, modern web scraping methods, and Python integration tools with MCP were studied. An information system for automated collection of vacancies from Work.ua and Djinni was developed. The system provides HTTP request processing, HTML structure parsing, data filtering, and transfer of results to the agent through the MCP standard.

This work consists of four sections. The first section analyzes the subject area, web scraping approaches, and the MCP protocol. The second section defines the system requirements and describes its architecture. The third section presents the implementation of the MCP server and vacancy collection modules. The fourth section presents testing, analysis of results, and system limitations.

The overall scope of the work is 100 pages. The thesis contains 6 appendices, 15 figures, 23 tables, and 41 references.

Key words: information system, MCP protocol, Python, web scraping, Work.ua, Djinni, LLM, automation, data processing.

ЗМІСТ

ВСТУП

У сучасному цифровому середовищі веб-ресурси є одним із наймасовіших джерел актуальної інформації для бізнесу, науки та власного користування. Значна частина даних, що відображають реальні процеси та тенденції (ринок праці, ціни, новини, відкриті реєстри, статистика, каталоги товарів і послуг), публікується саме в інтернеті у вигляді веб-сторінок. Проте ці дані зазвичай представлені у форматі, орієнтованому на людину (HTML-інтерфейс, інтерактивні блоки, візуальні компоненти, тощо), а не в структурованому вигляді, зручному для автоматизованої обробки. Саме тому задачі автоматизованого отримання та перетворення даних із веб-сторінок (задачі вебскрапінгу) залишаються надзвичайно актуальними.

Паралельно з розвитком інструментів збору та аналізу даних активно еволюціонують великі мовні моделі (LLM), які здатні виконувати інтелектуальні задачі: узагальнення, класифікацію, виділення сутностей, формування звітів, пошук закономірностей. Проте для практичного застосування LLM у прикладних інформаційних системах часто виникає критична потреба в інтеграції моделі з реальними даними та зовнішніми сервісами: базами даних, API, веб-ресурсами, файловими сховищами тощо. У цьому контексті ключовою проблемою стає стандартизований, надійний та контрольований механізм взаємодії мовної моделі із зовнішніми інструментами.

Одним із перспективних підходів, що вирішує зазначену проблему, є Model Context Protocol (MCP) – протокол, який забезпечує єдині принципи інтеграції LLM з інструментами та сервісами через стандартизований інтерфейс. Використання MCP дозволяє розмежувати логіку «мислення» (агент/LLM) і логіку виконання дій (інструмент/сервер), що підвищує керованість системи, полегшує тестування, повторне використання компонентів та масштабування.

Актуальність теми бакалаврської кваліфікаційної роботи обумовлена кількома факторами. По-перше, веб-дані є цінним джерелом інформації для аналітики, прийняття рішень і прогнозування. По-друге, сучасні системи на основі

LLM потребують стандартизованого «мосту» між моделлю та зовнішнім світом – таким мостом може виступати MCP. Отже, розробка інформаційної системи для вебскрапінгу є практично значущою і актуальною задачею.

Метою роботи є розробка інформаційної системи вебскрапінгу на основі MCP-протоколу та дослідження можливостей його використання для автоматизованого збору даних із веб-ресурсів.

Для досягнення поставленої мети необхідно виконати такі завдання:

1) проаналізувати особливості збору даних із веб-ресурсів, визначити типи даних і типові сценарії використання результатів вебскрапінгу;
2) розглянути сучасні підходи до вебскрапінгу та обробки веб-даних, визначити їх сильні та слабкі сторони;
3) дослідити теоретичні засади MCP-протоколу, його структуру, концепцію інструментів і принципи взаємодії агента з інструментом;
4) визначити вимоги до інформаційної системи вебскрапінгу: функціональні та нефункціональні (надійність, продуктивність, розширюваність, стійкість до змін HTML-структури);
5) спроєктувати архітектуру системи взаємодії зовнішнього агента з інформаційною системою, визначити формат вхідних параметрів, вихідних даних і сценарії помилок;
6) реалізувати програмні модулі інформаційної системи для збору вакансій із Work.ua: виконання HTTP-запитів, парсинг HTML, нормалізація й фільтрація даних, формування структурованого результату;
7) провести тестування працездатності програмного забезпечення, оцінити коректність збору даних, стабільність та продуктивність.

Об’єктом дослідження є процес автоматизованого збору та структуризації даних із веб-ресурсів.

Предметом дослідження є методи, моделі та програмні засоби реалізації інформаційної системи для вебскрапінгу, а також механізми взаємодії зовнішнього агента (LLM) з ІС через MCP-протокол.

У роботі застосовано такі методи та підходи: аналіз предметної області, порівняльний огляд технологій, проєктування архітектури інформаційних систем, реалізація модульного коду, тестування (функціональне та частково навантажувальне), аналіз результатів.

Практична цінність роботи полягає у створенні інформаційної системи, яка може бути використана як підсистема у складі інтелектуальних систем збору даних, аналітичних платформ або агентних застосунків на основі LLM. ПЗ може бути розширене на інші веб-ресурси або адаптоване під інші типи даних, зберігаючи стандартизовану взаємодію через MCP.

Структура роботи: бакалаврська кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків. У першому розділі наведено аналіз предметної області, сучасних підходів до вебскрапінгу, а також теоретичні засади MCP-протоколу. У другому розділі описано постановку задачі, вимоги до системи, її архітектуру та механізми взаємодії MCP-компонентів. У третьому розділі розглянуто програмну реалізацію MCP-сервера, модулів збору й обробки вакансій, а також інтеграцію з агентом. У четвертому розділі проведено тестування розробленої системи, перевірено коректність взаємодії агента з MCP-сервером, оцінено точність, повноту та продуктивність збору даних, а також проаналізовано отримані результати й обмеження системи.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ТЕОРЕТИЧНІ ОСНОВИ ДЛЯ СТВОРЕННЯ MCP-ПРОТОКОЛУ ДЛЯ ЗБОРУ ДАНИХ ІЗ ВЕБ-РЕСУРСІВ

1.1 Аналіз підходів до отримання даних із веб-ресурсів

Предметна область даної роботи охоплює питання **автоматизованого отримання даних із веб-ресурсів**, їх перетворення у структурований вигляд, а також інтеграцію результатів збору з інтелектуальними компонентами, зокрема з агентами на базі LLM. Актуальність таких підходів підтверджується сучасними дослідженнями у сфері цифрових систем і сталого розвитку, де підкреслюється важливість ефективного використання даних для прийняття рішень. Для коректної постановки задачі важливо визначити ключові поняття, класифікувати типи веб-даних, описати типові сценарії використання та окреслити обмеження й виклики, характерні для вебскрапінгу.

Веб-ресурси подають інформацію у вигляді веб-сторінок, які зазвичай створюються з використанням HTML, а також доповнюються стилями (CSS) і сценаріями (JavaScript) для покращення вигляду та взаємодії з користувачем.

Розглядаючи веб-дані з точки зору автоматизованої обробки [1], можна виділити такі їх типи залежно від ступеня впорядкованості:

– структуровані – таблиці, чітко визначені поля, елементи з повторюваними шаблонами;

– напівструктуровані – HTML-розмітка, де дані представлені блоками, картками, списками, тобто структура існує, але не є формальною схемою даних;

– неструктуровані – вільний текст, описи, коментарі, які потребують додаткової обробки для виділення сутностей.

На практиці більшість сайтів, що публікують оголошення, вакансії або каталоги, **містять напівструктуровані дані**, які повторюються за одним шаблоном на різних сторінках. Саме такі дані найчастіше і стають об’єктом вебскрапінгу.

Вебскрапінг (web scraping) – це процес автоматизованого отримання даних із веб-сторінок із подальшим виділенням потрібних фрагментів та перетворенням їх у структурований формат (наприклад JSON, CSV, таблиці бази даних) [4]. Типовий цикл вебскрапінгу включає:

– надсилання HTTP-запиту до веб-ресурсу та отримання HTML-відповіді;

– аналіз отриманого HTML, а саме парсинг DOM-структури;

– вилучення потрібних даних за селекторами/шаблонами;

– очистка та нормалізація тексту, валют, формату дат, усунення шуму;

– структуризація та збереження/передача результатів.

Суміжними поняттями є:

– **парсинг** (*parsing*) – перетворення HTML-тексту у дерево DOM, що дозволяє звертатися до елементів сторінки як до структурованих вузлів;

– **краулінг** (*web crawling*) – автоматизований обхід набору сторінок за посиланнями для пошуку й збору даних у масштабі сайту або багатьох сайтів [1];

– **ETL-процес** (*Extract–Transform–Load*) – підхід, де дані витягуються (*Extract*), перетворюються (*Transform*) і завантажуються (*Load*) у сховище. Вебскрапінг часто є частиною етапу Extract, а подальша обробка – Transform;

– **конвеєр даних** (*Data pipelin*) – організований процес збору, обробки й передачі даних між компонентами системи.

У межах даної роботи вебскрапінг розглядається не як ізольований скрипт, а як *інструмент*, який може викликатися агентом та надавати результати у стандартизованому вигляді.

Вакансії на сайтах з пошуку роботи – типовий приклад даних, що мають високу практичну цінність для аналітики: оцінка попиту на спеціалістів, аналіз зарплатних вилок, визначення популярних технологій, трендів у вимогах до кандидатів. Оголошення про вакансію зазвичай містить такі ключові поля:

– назва посади;

– назва компанії/роботодавця;

– місто/регіон або формат роботи (офіс/гібрид/віддалено);
– заробітна плата (якщо вказана);
– опис вакансії та вимоги;
– дата публікації;
– посилання на сторінку вакансії;
– додаткові атрибути (досвід, зайнятість, графік, категорія).

З погляду автоматизації важливо, що ці поля можуть бути представлені на сторінках у різних форматах (числа з пробілами, діапазони, «договірна», різні способи зазначення міста тощо). Тому система збору повинна передбачати *нормалізацію* та *обробку відсутніх значень*, щоб результати були придатними для подальшого аналізу [1].

Аналізуючи процес вебскрапінгу, попри уявну простоту ідеї *«завантажити сторінку та витягти дані»*, у реальних умовах доводиться стикатися з низкою проблем:

1) **зміни HTML-структури**: сайт може змінити верстку, назви класів, розміщення елементів. Надто «крихкі» селектори призводять до поломок;
2) **динамічний контент**: частина даних може завантажуватися через JavaScript після відкриття сторінки. У таких випадках класичний HTTP-запит не завжди достатній;
3) **обмеження доступу та антибот-захист**: rate limits, CAPTCHA, перевірки User-Agent, блокування підозрілої активності;
4) **якість і чистота даних**: дублікати, неповні записи, форматні відмінності, «шум» у тексті;
5) **етичні та правові аспекти**: потрібно дотримуватися правил використання сайту, уникати надмірного навантаження та некоректних сценаріїв збору.

Враховуючи зазначені виклики, можна сформулювати основні вимоги до інструменту, серед яких повторні спроби (retry), обмеження частоти запитів, логування, обробка винятків та гнучка логіка парсингу.

Розглядаючи інтеграцію таких інструментів у сучасні інтелектуальні системи [2], доцільно уточнити ключові поняття, пов'язані з агентами на базі LLM:

– **LLM** (*Large Language Model*) – модель, що оперує природною мовою й може виконувати завдання аналізу та генерації тексту;

– **агент** – програмний компонент, який використовує LLM для планування дій, прийняття рішень та взаємодії із зовнішніми ресурсами. Агент може «вирішити», коли потрібно звернутися до інструменту для отримання реальних даних;

– **інструмент** (*tool*) – функціональний модуль, який виконує конкретну дію у зовнішньому світі: отримує веб-дані, звертається до API, працює з файлами або базою даних, виконує обчислення тощо.

Проблема інтеграції полягає в тому, що без стандартизації кожен інструмент реалізується *«по-своєму»*, що породжує різні формати запитів/відповідей, складність тестування та підтримки. Саме тому виникає необхідність використання **уніфікованого протоколу взаємодії**, яким і виступає MCP.

Model Context Protocol (MCP) [3] у межах даної роботи розглядається як спосіб формалізувати взаємодію між агентом та інструментом вебскрапінгу. Ключові концепції, важливі для подальших розділів:

– **MCP-сервер** – компонент, що, забезпечуючи доступ до інструментів, надає їх у вигляді набору методів та приймає стандартизовані запити від агента;

– **MCP-клієнт** – частина системи на боці агента, яка, ініціюючи виклики інструментів, передає необхідні параметри та отримує результати їх виконання;

– **опис інструменту** – набір метаданих, що, визначаючи можливості інструменту, містить інформацію про параметри, типи даних, формат відповіді та можливі помилки;

– **контекст** – це сукупність вхідних даних і умов, яка, передаючись від агента до інструменту, визначає умови виконання конкретної задачі.

У задачі вебскрапінгу MCP дозволяє зробити інструмент **модульним і повторно придатним**: агент може викликати його за потреби, передаючи

параметри (наприклад, ключове слово пошуку, місто, сторінку, ліміт вакансій), а інструмент повертає **структуровані дані** в узгодженому форматі.

Отже, предметна область роботи поєднує два взаємопов'язані напрями: **автоматизований збір веб-даних** та **стандартизовану інтеграцію інструментів із LLM-агентами**. Вебскрапінг вакансій є практичним кейсом, який демонструє цінність систем збору даних і одночасно виявляє типові виклики реальних веб-ресурсів. Використання MCP у цьому контексті дозволяє формалізувати взаємодію, забезпечити стабільність, розширюваність та зручність використання інструменту в агентних застосунках.

1.2 Огляд та аналіз сучасних технологій вебскрапінгу та інструментів обробки даних

Сучасні веб-ресурси є важливим джерелом даних для багатьох сфер діяльності людини і для цього доцільним є застосування вебскрапінгу, який розглядається як автоматизований збір даних із веб-сторінок із подальшим перетворенням отриманої інформації у структурований вигляд, придатний для аналізу та використання в інформаційних системах.

Kahlon і Singh [5] провели систематичний огляд методів вебскрапінгу, оглянувши не лише технічні підходи, а й їх ефективність, застосування великих мовних моделей, а також юридичні та етичні аспекти. Робота формує цілісну картину сучасного стану галузі й чітко показує, що вебскрапінг сьогодні – це не лише про збір, а й про очищення, нормалізацію, верифікацію та збереження даних.

Kazmali і Sayar [6] зосередилися на прикладному рівні, адже автори розглядають вебскрапінг як інструмент збору даних у фінансах, виробництві, штучному інтелекті та академічних дослідженнях. Вони класифікують основні техніки – HTML-парсинг, API scraping, XPath – і окремо розглядають правові та етичні обмеження, зокрема персональні дані, правила сайтів і технічні механізми захисту: Rate Limiting, IP Blocking, CAPTCHA.

Іванов і Гаркуша [7] розробили методологію парсингу для автоматизованого збору різноструктурованих даних. Автори описують загальну архітектуру процесу, яка охоплює аналіз джерела, вибір інструментів, формування правил парсингу, обробку помилок і збереження результатів. Серед конкретних інструментів прямо названо BeautifulSoup, Scrapy, Selenium і Puppeteer. Окремо розглядаються robots.txt, правові аспекти та складність обробки мультимодального контенту.

Отже, можна побачити, що усі три роботи [5-7] охоплюють подібне коло технік та те що спільними рішеннями є HTML-парсинг за допомогою CSS-селекторів і XPath, звернення до API та структурованих відповідей, а також модульна організація конвеєра збору. Перевагами даного підходу є простота реалізації HTML-парсингу та низькі вимоги до ресурсів; API scraping забезпечує зручний структурований формат. Однак усі три джерела фіксують однакові недоліки: залежність rule-based підходів від конкретної структури сторінки, ризик блокування та необхідність регулярного оновлення правил при зміні розмітки.

Perkins і Akorede [9] описують свою систему щотижневого вебскрапінгу вакансій у сфері K-12 та вищої освіти. Дані регулярно збиралися, очищалися та архівувалися, після чого об'єднувалися у датасет для аналізу попиту на педагогічні кадри. Завдяки цьому система дозволяла щомісячно відстежувати тренди, порівнювати предметні напрями та географічні відмінності.

Molina, Morales і Keith [10] використали BeautifulSoup і Selenium для збору матеріалів із 15 чилійських новинних медіа за 2019–2023 роки. Початкова вибірка налічувала 1254 статті, після фільтрації дублікатів і нерелевантного контенту залишилося 931 запис. Важливо зазначити те, що різні медіа потребували різних підходів та правил для вилучення даних, а це все через відмінності HTML-структур, а частина ресурсів була недоступна через paywall або anti-scraping механізми.

Отже, можна побачити, що обидві роботи [9, 10] демонструють процес вебскрапінгу не як одноразову операцію, а як регулярний процес моніторингу. Також важливим є те, що обидва рішення використовують поєднання BeautifulSoup і Selenium, обов'язкову фільтрацію дублікатів, стандартизацію та збереження у

структурованому форматі (JSON, CSV), що буде доцільно використати в роботі. Перевагою постійного збору є можливість аналізувати зміни в часі та будувати тривалі датасети. Недоліком, який фіксують обидва дослідження, є неможливість застосування єдиних правил до різних сайтів навіть однієї тематичної категорії, адже будь-яке оновлення вимагає повний перегляд підходу та кожне джерело вимагає окремої конфігурації парсера.

Abd Rahman, Ahmed і Md Ali [11] провели систематичний огляд AI-орієнтованих методів скрапінгу з акцентом на transformer-based моделях. Автори показують, що rule-based підходи, які спираються на XPath, CSS-селектори та регулярні вирази, є нестійкими при зміні структури сторінок або при JavaScript-рендерингу контенту. Натомість моделі на кшталт BERT і GPT краще враховують контекст і є стійкішими до змін макета.

Lazebnyi та співавтори [8] запропонували метод, який забезпечує агрегування неструктурованих даних із веб-джерел із використанням LLM. В їхньому рішенні Goose3 обробляє статичні сторінки, Selenium WebDriver – динамічні, MongoDB зберігає результати, а LLM згодом нормалізує зібрані дані до JSON-схеми. Для зниження ризику похибки, автори впровадили двоетапну перевірку результатів.

Lacher і Rohs [12] застосували AI-assisted web scraping для аналізу 7305 німецькомовних веб-сторінок про підготовку до надзвичайних ситуацій. Методика поєднує expert-guided sampling, схему кодування, масштабний скрапінг та AI-assisted класифікацію за типом автора, темою та медіаформатом.

Отже, можна побачити, що усі три роботи [8, 11, 12] вказують на одну закономірність: LLM найефективніше використовувати не як самостійний інструмент для збору даних, а як модуль для нормалізації та класифікації вже отриманої інформації. Також спільним є гібридна архітектура (класичний скрапінг + LLM-постобробка), двоетапна верифікація та визначена JSON-схема як цільовий формат. Перевагою є здатність моделей обробляти неструктурований текст, виділяти ключові сутності та приводити різноформатні дані до єдиного вигляду.

Недоліком, на який вказують усі три джерела, є ризик неточних або вигаданих результатів, що вимагає обов'язкової верифікації виводу моделі.

Liu та співавтори [13] представили систему SCRIBES, вона збирає напівструктуровані дані з HTML-таблиць, списків та інформаційних блоків. Головна відмінність від підходів, де LLM запускається окремо для кожної сторінки, це те, що SCRIBES формує сценарії, що повторно використовуються для груп однотипних сторінок. Тобто, система вже буде навчена з підкріпленням для автоматичного покращення сценаріїв.

Fahrudin та співавтори [14] розробили систему збору наукових статей, де поєднано кілька API, багатопоточну обробку, BeautifulSoup, регулярні вирази, PyPDF2 та Sentence Transformers для семантичного порівняння. За результатами їх експериментів кількість релевантних повнотекстових PDF збільшилася в середньому на 64,38%, а час відповіді скоротився на 59,78%.

Отже, можна побачити, що обидві роботи [13, 14] спрямовані на підвищення масштабованості та швидкості, але досягають цього різними шляхами. Спільним рішенням, яке можна виділити є відокремлення логіки вилучення даних від загальної архітектури системи, що дозволяє повторно використовувати правила та підключати нові джерела без переписування всієї системи. Перевагами є зниження ручної роботи та суттєве прискорення збору. Недоліками є вузька застосовність сценаріїв: підхід SCRIBES добре працює лише для однорідних за структурою сторінок, а багатопоточність вимагає ретельного контролю навантаження на веб-ресурс – надмірна кількість запитів може призвести до блокування або порушення умов використання сайту.

Аналіз джерел дозволяє систематизувати наявні підходи табл. 1.1, з цього виплиавє, що жоден із розглянутих підходів не є універсальним. На практиці доцільно використовувати комбіновану архітектуру: для статичних сторінок – прямий HTML-парсинг, для динамічних – браузерну автоматизацію або аналіз внутрішніх запитів сайту, для обробки результатів – нормалізацію та перевірку даних, а для масштабування – повторювані сценарії та асинхронну обробку.

Таблиця 1.1 – Порівняльна характеристика підходів до вебскрапінгу

Підхід	Переваги	Недоліки
HTML-парсинг / CSS / XPath [4]	Простота, швидкість, низькі вимоги до ресурсів	Залежність від структури сторінки; потребує оновлення при зміні розмітки
API scraping [5]	Структурований формат відповіді, висока швидкість	Не завжди доступний; може змінюватися без попередження
Браузерна автоматизація (Selenium) [5]	Підтримка JS-рендерингу, робота з динамічними сторінками	Повільніша робота; вищі вимоги до ресурсів
LLM-нормалізація та класифікація [8]	Обробка неструктурованого тексту; стійкість до змін макета	Ризик похибки; потребує верифікації
Повторювані сценарії (SCRIBES) [13]	Зменшення ручної роботи; масштабованість	Ефективний лише для однотипних сторінок
Багатопоточність і асинхронна обробка [14]	Суттєве прискорення збору	Ризик блокування; складніший контроль навантаження
Регулярний моніторинг з очищенням [9]	Поздовжні датасети; аналіз трендів у часі	Потреба в окремих конфігураціях для кожного джерела

1.3 MCP-протокол: призначення, структура та принципи взаємодії.

Model Context Protocol (MCP) є відкритим стандартом, призначеним для організації взаємодії між AI-застосунками, що використовують великі мовні моделі, та зовнішніми сервісами. Його основна ідея полягає в тому, щоб надати єдиний механізм підключення зовнішніх інструментів, джерел даних і шаблонів взаємодії без необхідності створювати окрему інтеграцію для кожного сервісу.

Архітектурно MCP реалізує клієнт-серверну модель [3]. Центральним компонентом є MCP Host, тобто AI-застосунок, який взаємодіє з мовною моделлю та керує використанням зовнішніх можливостей. Для кожного підключеного MCP-сервера Host створює окремий MCP Client, через який здійснюється обмін повідомленнями. MCP Server, у свою чергу, є окремим процесом або сервісом, який реалізує конкретну функціональність: наприклад, збір даних із веб-ресурсів, обробку файлів, звернення до API або роботу з базою даних.

Такий розподіл відповідальності дозволяє відокремити логіку роботи з LLM від виконання прикладних задач. У результаті система стає більш гнучкою, а розроблені інструменти можна повторно використовувати в різних AI-застосунках. Загальну схему взаємодії MCP Host, клієнтів і серверів наведено на рис. 1.1.

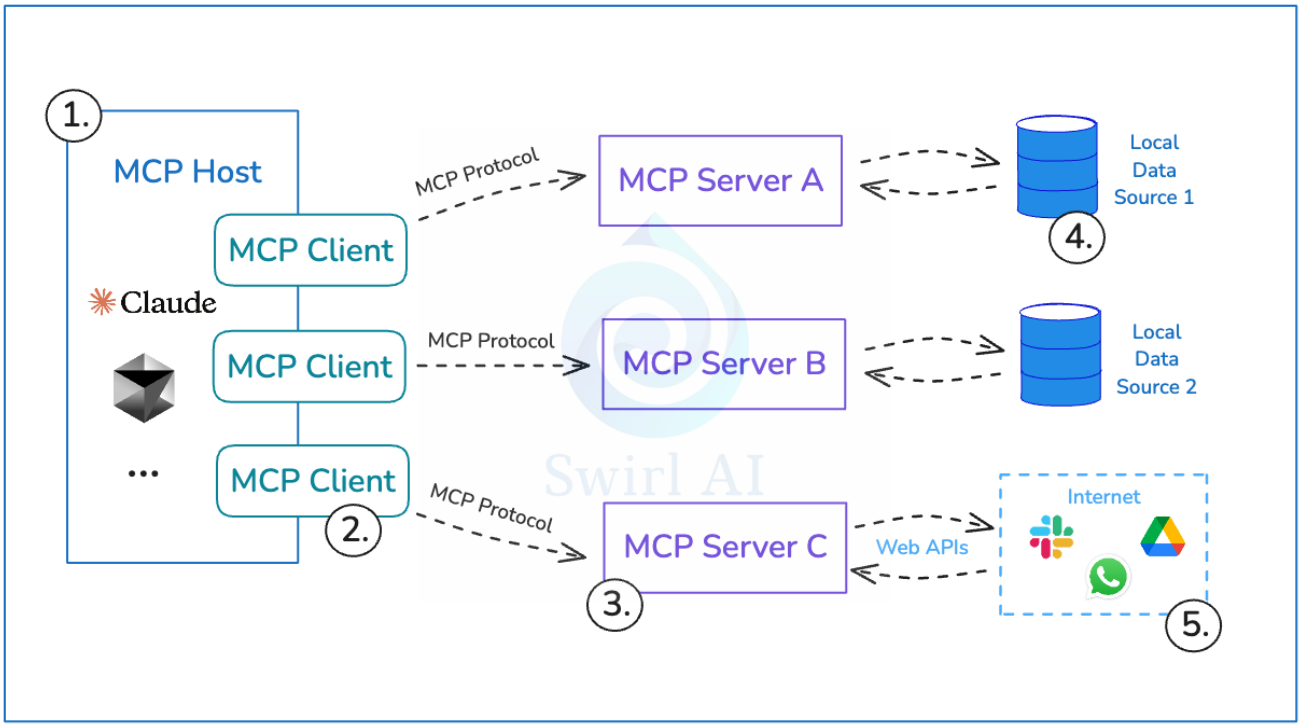

Рисунок 1.1 – Архітектура взаємодії MCP Host, клієнтів та серверів [15]

MCP складається з двох основних рівнів: рівня даних і транспортного рівня. Рівень даних базується на JSON-RPC 2.0, тому всі повідомлення між клієнтом і сервером передаються у стандартизованому JSON-форматі та кодуються в UTF-8. На цьому рівні визначаються життєвий цикл з'єднання, узгодження можливостей сторін, а також базові примітиви MCP: **tools**, **resources** і **prompts** [3].

Tools – це виконувані інструменти, які AI-застосунок може викликати для отримання даних або виконання певної дії. Клієнт спочатку отримує список доступних інструментів через метод tools/list, а потім викликає потрібний інструмент через tools/call, передаючи його назву та аргументи відповідно до визначеної схеми параметрів. Прикладом можуть бути інструменти Chrome DevTools MCP, які дозволяють взаємодіяти з веб-сторінкою, аналізувати мережеві запити, виконувати скрипти та отримувати технічну інформацію про сторінку.

Рисунок 1.2 – Chrome Devtools MCP інструменти

Resources – це джерела даних, які сервер може надати клієнту як контекст. Такими ресурсами можуть бути файли, записи бази даних, результати API або збережені результати вебскрапінгу. Наприклад, MCP-сервер може надавати файл із вакансіями у форматі JSON. Клієнт отримує перелік доступних ресурсів через resources/list, після чого читає потрібний ресурс за його URI через resources/read.

Prompts – це повторно використовувані шаблони повідомлень або інструкцій для роботи з мовною моделлю. Вони дозволяють стандартизувати взаємодію з LLM у типових сценаріях. Наприклад, prompt може містити інструкцію для аналізу опису вакансії та виділення з нього назви посади, рівня досвіду, ключових навичок і зарплатного діапазону. Клієнт отримує список промптів через prompts/list, а конкретний шаблон – через prompts/get.

Життєвий цикл MCP-з’єднання починається з ініціалізації. Спочатку клієнт надсилає серверу запит initialize, у якому передає версію протоколу, власні можливості та інформацію про реалізацію. Сервер повертає відповідь із погодженою версією протоколу, власними можливостями та службовою інформацією про себе. Після цього клієнт надсилає нотифікацію notifications/initialized, яка підтверджує готовність до подальшої роботи.

Під час робочої фази клієнт і сервер обмінюються JSON-RPC-повідомленнями відповідно до узгоджених можливостей. Завершення з’єднання виконується на рівні транспортного механізму, оскільки окремого повідомлення для завершення роботи MCP-протокол не передбачає.

Транспортний рівень визначає спосіб доставки JSON-RPC-повідомлень. MCP підтримує два основні варіанти транспорту: stdio, тобто обмін через стандартні потоки вводу/виводу, і Streamable HTTP, тобто обмін через HTTP із можливістю потокової передачі. Незалежно від обраного транспорту, формат повідомлень залишається однаковим – JSON-RPC 2.0 [16].

Для кращого розуміння принципу роботи MCP можна розглянути два базові приклади повідомлень.

Перший приклад демонструє ініціалізацію з’єднання. Запит initialize містить версію протоколу, можливості клієнта та інформацію про клієнтський застосунок.

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-06-18",
    "capabilities": {
      "roots": { "listChanged": true },
      "sampling": {},
      "elicitation": {}
    },
    "clientInfo": {
      "name": "ExampleClient",
      "title": "Example Client Display Name",
      "version": "1.0.0"
    }
  }
}
```

У цьому повідомленні поле jsonrpc вказує на використання JSON-RPC 2.0, id є ідентифікатором запиту, method визначає тип операції, а params містить параметри ініціалізації. Після відповіді сервера клієнт надсилає notifications/initialized, завершуючи етап встановлення з’єднання.

Другий приклад демонструє виклик інструмента через tools/call [17]. Розглядаючи наступне повідомлення, воно демонструє типовий сценарій використання інструментів. Поле "method": "tools/call" визначає виклик інструмента, а "id": 2 дозволяє співставити запит із відповіддю. У "params" передаються:

– "name" – назва інструмента, який доступний на сервері;

– "arguments" – набір параметрів для його виконання (у прикладі – "location": "New York").

Приклад коду наведено нижче.

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/call",
  "params": {
    "name": "get_weather",
    "arguments": {
      "location": "New York"
    }
  }
}
```

Після виконання запиту сервер повертає результат у полі "result", який, як правило, містить структуровані дані або текстовий вміст. У випадку помилки також може повертатися відповідна інформація про некоректність запиту або неможливість його виконання.

Отже, MCP забезпечує стандартизований спосіб взаємодії між AI-застосунками та зовнішніми сервісами. Завдяки поділу на Host, Client і Server, а також використанню tools, resources і prompts, протокол дозволяє будувати

модульні системи, у яких мовна модель може безпечно й уніфіковано працювати з зовнішніми інструментами. У межах цієї дипломної роботи MCP доцільно використовувати як основу для створення інструменту вебскрапінгу, який надаватиме LLM-клієнту доступ до функцій збору, обробки та повернення структурованих веб-даних.

1.4 Аналіз існуючих підходів до інтеграції LLM з зовнішніми інструментами

У міру того, як розвиваються систем які працюють на базі великих мовних моделей важливим напрямом стала інтеграція LLM із зовнішніми інструментами та джерелами даних. Завдяки цьому частково можна подолати обмеження мовних моделей, зокрема відсутність доступу до актуальної інформації, неможливість самостійно виконувати дії у зовнішніх системах та залежність від даних, на яких модель була навчена. Також, завдяки інтеграції із зовнішніми інструментами модель може звертатися до API, працювати з файлами, базами даних, пошуковими сервісами або спеціалізованими програмними модулями.

Основні підходи до інтеграції LLM із зовнішніми інструментами подано в табл. 1.2.

Таблиця 1.2 – Основні підходи до інтеграції LLM із зовнішніми інструментами

Підхід	Суть підходу	Переваги	Недоліки
Prompt-орієнтована інтеграція	Модель отримує інструкцію сформувати відповідь у потрібному форматі, наприклад JSON або HTTP-запит	Простота реалізації, не потребує складної архітектури	Модель може порушити формат, передати неповні або помилкові параметри
Function / tool calling	Інструмент описується через назву, призначення та схему параметрів, а модель формує структурований виклик	Контроль параметрів, можливість валідації, зручне логування	Потребує попереднього опису інструментів і логіки їх виконання

Кінець таблиці 1.2

Підхід	Суть підходу	Переваги	Недоліки
Reasoning + Acting	Модель поєднує міркування з послідовним викликом інструментів для розв'язання задачі	Підходить для складних багатокрокових сценаріїв	Ускладнює контроль, потребує оркестрації та обробки помилок
Прикладні фреймворки	Використання готових бібліотек і платформ для поєднання LLM з інструментами	Швидка розробка, готові інтеграції	Залежність від конкретного фреймворку та його внутрішніх форматів
Retrieval-Augmented Generation	Пошук потрібних документів у зовнішньому джерелі та додавання їх до контексту моделі	Підвищує актуальність відповіді, зменшує ризик галюцинацій	Не виконує дії напряму, а переважно надає інформаційний контекст
Model Context Protocol	Інтеграція через єдиний стандартизований протокол між AI-застосунком і зовнішніми сервісами	Модульність, повторне використання інструментів, єдиний формат взаємодії	Потребує реалізації MCP-сервера та дотримання специфікації протоколу

Першим і найпростішим підходом є prompt-орієнтована інтеграція, схему якої наведено на рис. 1.3. У цьому випадку розробник задає моделі інструкцію повернути відповідь у певному форматі, наприклад сформувати JSON, HTTP-запит або набір параметрів для подальшої обробки. Такий підхід легко реалізувати, однак він має суттєві недоліки: модель може порушити формат відповіді, пропустити потрібні поля або згенерувати параметри, які не були передбачені розробником. Саме ці обмеження стали причиною переходу до більш формалізованих механізмів виклику інструментів.

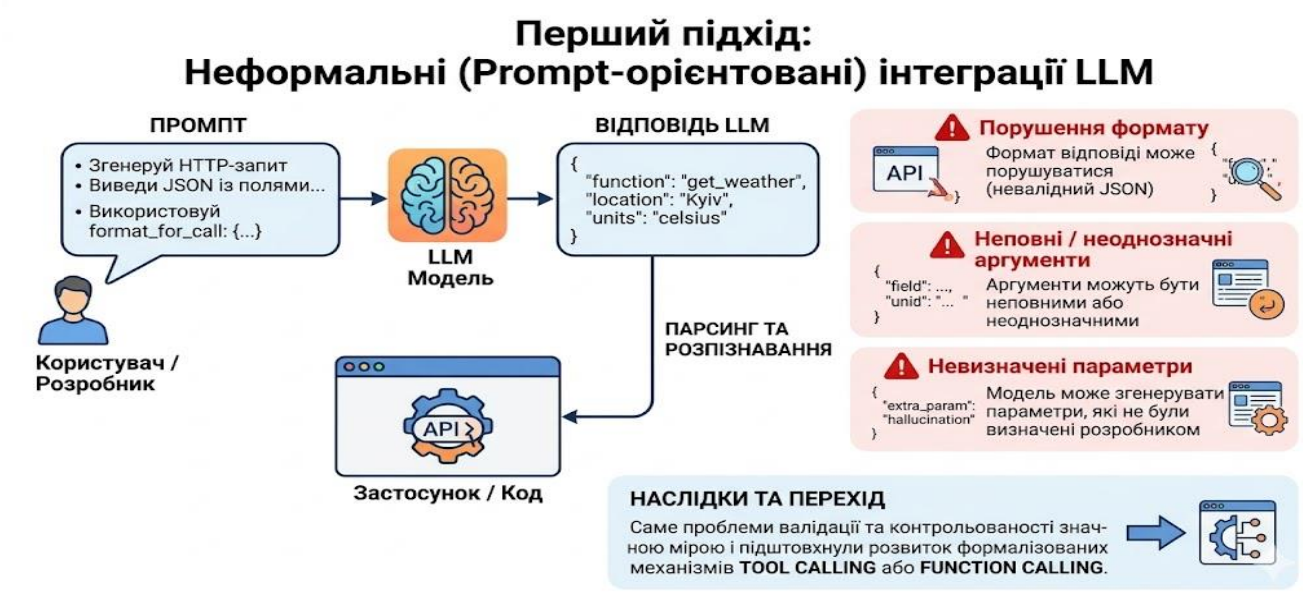

Рисунок 1.3 – Неформальні або prompt-орієнтовані

Другий підхід – це структуроване викликання інструментів, або function / tool calling, приклад якого подано на рис. 1.4. У цьому випадку інструмент описується явно: зазначається його назва, призначення та схема вхідних параметрів. Модель повертає не довільний текст, а структурований виклик, який застосунок може перевірити й виконати. Наприклад, для інструмента get_weather(location) модель може сформувати виклик із параметром location, після чого застосунок звертається до погодного сервісу та повертає результат моделі. Перевагою такого підходу є можливість перевіряти параметри, логувати виконання інструментів і працювати з результатами в уніфікованому форматі [18].

Рисунок 1.4 – Function calling

Третій підхід пов'язаний з агентними схемами типу Reasoning + Acting [19], загальну логіку яких показано на **рис. 1.5.** У таких системах модель не просто викликає один інструмент, а виконує задачу у кілька етапів: аналізує ситуацію, обирає дію, звертається до зовнішнього джерела, отримує результат і продовжує розв'язання задачі. До цього напряму належать підходи ReAct і MRKL. ReAct поєднує кроки міркування та дії, а MRKL передбачає використання зовнішніх модулів, наприклад пошуку, баз знань або калькуляторів. Такі підходи добре підходять для складних задач, але потребують додаткового контролю.

Рисунок 1.5 – Reasoning + Acting

Четвертий підхід полягає у використанні прикладних фреймворків. Приклад такої інтеграції наведено на **рис. 1.6.** Такі засоби, як LangChain або Semantic Kernel, уже містять готові механізми для підключення мовної моделі до інструментів і організації їх виклику. Наприклад, LangChain дозволяє будувати ланцюжки взаємодії між моделлю та інструментами, а Semantic Kernel використовує plugins як набір функцій, які можуть бути доступні моделі [20]. Перевагою цього підходу є швидкість розробки та наявність готових інтеграцій. Недоліком є залежність від конкретного фреймворку, його архітектури та внутрішніх форматів.

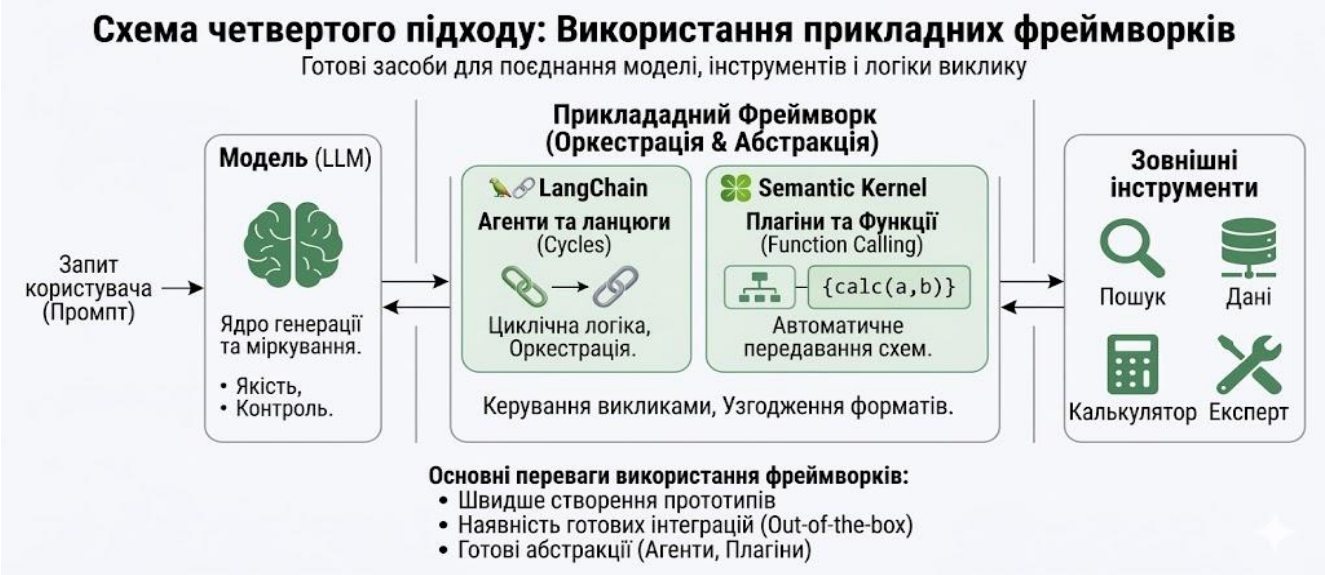

Рисунок 1.6 – Використанням прикладних фреймворків

П’ятий підхід – Retrieval-Augmented Generation, або RAG [21], схему якого показано на рис. 1.7. Його основна ідея полягає не у виконанні дій, а в пошуку потрібної інформації у зовнішніх джерелах і додаванні її до контексту моделі. Система може звертатися до бази знань, набору документів або векторного сховища, знаходити релевантні фрагменти й передавати їх моделі для формування відповіді. Це підвищує актуальність і обґрунтованість результату, а також зменшує ризик генерації неправдивої інформації. Водночас RAG більше

підходить для роботи зі знаннями, ніж для виконання активних дій у зовнішніх системах.

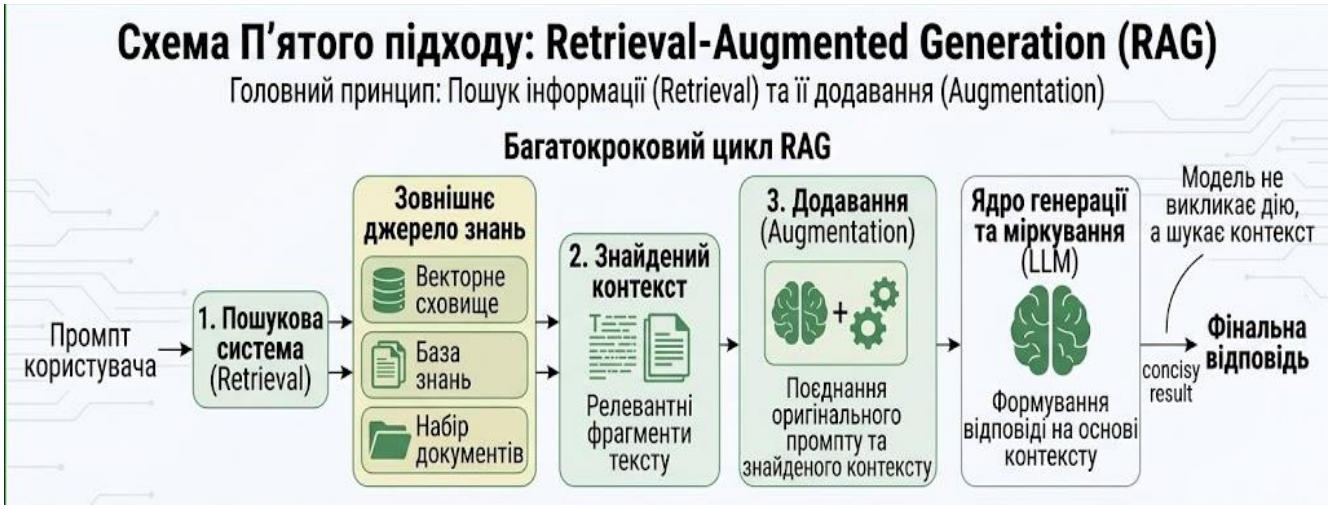

Рисунок 1.7 – Retrieval-Augmented Generation

Окремим напрямом є протокольна стандартизація інтеграції. До цього підходу належить Model Context Protocol. MCP пропонує не окрему інтеграцію для кожного сервісу, а єдиний стандарт взаємодії між AI-застосунком і зовнішніми системами. Відповідно до документації MCP, протокол використовує клієнт-хост-серверну архітектуру та обмін повідомленнями у форматі JSON-RPC 2.0. Завдяки цьому достатньо реалізувати MCP-сумісний сервер, щоб надалі підключати його до різних AI-застосунків без створення окремої логіки для кожного з них.

Отже, існує кілька основних підходів до інтеграції LLM із зовнішніми інструментами: prompt-орієнтовані схеми, structured function / tool calling, агентні підходи Reasoning + Acting, прикладні фреймворки, RAG та протокольна стандартизація через MCP. Найбільш надійними для практичних систем є формалізовані й стандартизовані підходи, оскільки вони забезпечують кращий контроль, валідацію параметрів, повторне використання інструментів і простіше масштабування. У межах цієї дипломної роботи саме MCP є найбільш доцільним підходом, оскільки він дозволяє створити окремий інструмент вебскрапінгу та підключати його до LLM-клієнта через уніфікований протокол.

Висновки до першого розділу

У першому розділі було сформовано теоретичну основу бакалаврської роботи та визначено основні поняття, підходи й технологічні засоби, необхідні для подальшого проєктування та реалізації MCP-інструменту для вебскрапінгу.

У підрозділі 1.1, аналізуючи предметну область автоматизованого збору даних із веб-ресурсів, було визначено ключові особливості веб-даних як джерела інформації. Встановлено, що значна частина практично цінної інформації на сайтах подається у напівструктурованому вигляді, а тому потребує парсингу, нормалізації та перевірки якості. Окремо було окреслено типові труднощі вебскрапінгу, серед яких залежність від HTML-структури сторінок, наявність динамічного контенту, неповнота або варіативність полів, а також потреба у стабільності й повторюваності процесу збору.

У підрозділі 1.2, розглядаючи сучасні технології вебскрапінгу, було показано, що вибір інструментів безпосередньо залежить від типу веб-ресурсу. Для статичних сторінок доцільно використовувати HTTP-клієнти та HTML-парсери, тоді як для динамічних сайтів ефективнішими є аналіз запитів фронтенду або застосування headless-браузерів. Також було встановлено, що для масштабного збору даних важливими є фреймворки краулінгу та асинхронні підходи, а для подальшого використання результатів – засоби очищення, нормалізації та структурування даних.

У підрозділі 1.3, досліджуючи Model Context Protocol, було визначено його роль як стандартизованого механізму взаємодії між LLM-агентом і зовнішніми інструментами. Встановлено, що MCP забезпечує формалізований спосіб підключення інструментів, передачі контексту та отримання результатів через узгоджені примітиви й єдиний контракт взаємодії. Важливою перевагою такого підходу є чітке розмежування відповідальності між агентом, який відповідає за логіку роботи й прийняття рішень, та інструментом, який виконує конкретну операцію і повертає структурований результат.

У підрозділі 1.4, аналізуючи наявні підходи до інтеграції LLM із зовнішніми інструментами, було встановлено, що сучасні системи поступово переходять від неформальних prompt-орієнтованих інтеграцій до більш надійних і структурованих механізмів виклику інструментів. Було розглянуто tool/function calling, агентні цикли з послідовним виконанням дій, фреймворки з готовими абстракціями для підключення інструментів, а також RAG як підхід до роботи із зовнішніми знаннями. На цьому тлі MCP було визначено як перспективний напрям протокольної стандартизації, що зменшує залежність від конкретних платформ і спрощує повторне використання інструментів.

Таким чином, узагальнюючи результати першого розділу, можна зробити висновок, що було сформовано достатню теоретичну й технологічну основу для переходу до наступних етапів роботи – постановки вимог, проєктування архітектури та практичної реалізації інформаційної системи вебскрапінгу. Отримані результати підтверджують доцільність використання MCP як стандартизованого механізму взаємодії між агентом та інструментом збору даних, а також дозволяють визначити коло технологій, на яких доцільно будувати розроблюване рішення.

2 ПРОЄКТУВАННЯ ТА АРХІТЕКТУРА MCP-ІНСТРУМЕНТУ ДЛЯ ЗБОРУ ДАНИХ

2.1 Постановка задачі та вимоги до інформаційної системи на основі MCP-інструменту

Задачею роботи є розроблення інформаційної системи, яка автоматично збирає вакансії з сайтів Work.ua та Djinni. Система має працювати з LLM-клієнтом через протокол Model Context Protocol (stdio), приймати структуровані запити, обробляти HTML-сторінки, парсити дані та формувати результати у форматі JSON або CSV.

Система повинна виконувати такі основні завдання: шукати вакансії на Work.ua за ключовими словами, отримувати детальну інформацію про окремі вакансії Work.ua, шукати вакансії на Djinni з фільтрами, отримувати повний опис вакансій Djinni та зберігати отримані дані у зовнішній файл. Для цього сервер реалізує близько п’яти MCP-інструментів: два для Work.ua, два для Djinni та один для збереження даних.

До функціональних вимог належить підтримка пошуку вакансій за параметрами користувача. Для Work.ua основним параметром є текстовий запит, додатково можуть задаватися номер сторінки, максимальна кількість результатів і прапорець отримання розширеної інформації. Для Djinni передбачено ширший набір фільтрів, зокрема тип пошуку, зарплата, рівень досвіду, тип зайнятості, тип компанії, рівень англійської, домен, регіон, компанія та додаткові параметри запиту.

Важливою вимогою є отримання детальної інформації про окремі вакансії. Інструмент має працювати як зі сторінками пошуку, так і зі сторінками конкретних вакансій. У результаті повинні повертатися назва вакансії, компанія, посилання, опис, а також інші доступні атрибути, наприклад зарплата для Work.ua або канонічне посилання для Djinni.

Окрему увагу необхідно приділити уніфікації вихідних даних. Незалежно від джерела результат має формуватися як JSON-об’єкт із метаданими запиту та масивом знайдених записів. Для Work.ua використовується масив vacancies, для Djinni - масив jobs. Таке подання спрощує подальшу обробку даних у LLM-клієнті або зовнішніх аналітичних компонентах.

До функціональних вимог також належить валідація вхідних параметрів. Перевірка має виконуватися до початку мережевої взаємодії. Для цього використовуються моделі Pydantic, які контролюють типи, обов’язкові поля та допустимі значення[22]. У разі некоректних аргументів сервер повинен повертати повідомлення про помилку.

Система має підтримувати експорт результатів. Збереження може виконуватися автоматично через параметр export_file або за допомогою окремого інструменту s_save_payload, який дозволяє зберігати дані у форматах JSON або CSV. Шлях до файлу повинен перевірятися на безпечність і бути обмеженим каталогом експорту.

До нефункціональних вимог належать сумісність із MCP-клієнтами, що працюють через stdio, підтримка обміну повідомленнями JSON-RPC, запуск як локального процесу без окремого HTTP-сервера[23], а також модульність архітектури. Розділення логіки MCP-взаємодії, HTTP-запитів, парсингу та збереження даних спрощує супровід і подальше розширення системи.

Оскільки збір даних здійснюється шляхом аналізу HTML без використання офіційного API, система повинна бути адаптивною до змін структури сторінок Work.ua та Djinni. Для цього правила парсингу винесено в окремі модулі, що дозволяє оновлювати їх без зміни загальної архітектури.

Отже, задача проєктування полягає у створенні інформаційної системи MCP-інструменту, який забезпечує стандартизований пошук вакансій, отримання їх деталей, парсинг HTML-сторінок, формування уніфікованого JSON-результату, перевірку параметрів і збереження даних у файл. Ці вимоги визначають подальшу архітектуру та програмну реалізацію системи. Окрему увагу під час проєктування

слід приділити надійності роботи інструменту, обробці помилок і можливості подальшого розширення функціональності.

2.2 Архітектура системи збору даних і взаємодія MCP-компонентів

Архітектура інформаційної системи збору даних буде побудована у вигляді модульного MCP-сервера, який забезпечуватиме взаємодію з LLM-клієнтом, приймання запитів, доступ до зовнішніх веб-ресурсів, парсинг HTML-вмісту та формування результатів у структурованому JSON-форматі. Подібний підхід до організації взаємодії між LLM та зовнішніми джерелами даних розглядається в сучасних дослідженнях, зокрема у роботі [24], де MCP використовується як проміжний шар для доступу мовних моделей до даних IoT-просторів.

На концептуальному рівні інформаційна система складатиметься з трьох основних рівнів: клієнтського, серверного та рівня зовнішніх джерел даних. Клієнтський рівень буде представлений LLM-клієнтом або середовищем розробки з підтримкою MCP. Серверний рівень буде реалізований у вигляді локального MCP-сервера, який оброблятиме виклики інструментів. Рівень зовнішніх джерел даних включатиме веб-ресурси Work.ua та Djinni, з яких здійснюватиметься отримання інформації про вакансії.

Взаємодія між клієнтом і сервером здійснюватиметься через транспорт stdio у форматі JSON-RPC. Такий підхід дозволить запускати сервер як окремий локальний процес без необхідності створення окремого HTTP-сервера, а також забезпечує можливість роботи в автономному режимі без доступу до зовнішньої мережі. Саме такий принцип застосовано в роботі [25], де MCP-сервер використовується для офлайн-взаємодії малих мовних моделей із промисловими системами керування даними через локальний транспорт.

Серверний рівень інформаційної системи передбачається побудувати за принципом модульності та розподілу відповідальності. Подібні архітектурні підходи також використовуються у прикладних MCP-рішеннях, зокрема в системах індустріального моніторингу, де MCP виступає як інтеграційний рівень між LLM

та джерелами даних [26]. Центральний модуль відповідатиме за MCP-взаємодію, реєстрацію доступних інструментів, приймання аргументів виклику, їх перевірку, маршрутизацію запитів до відповідних обробників і формування підсумкової відповіді. Також у цьому модулі планується реалізувати централізовану обробку помилок, зокрема помилок валідації, некоректних параметрів і помилок мережевої взаємодії. Основні компоненти майбутньої інформаційної системи подано в таблиці 2.1.

Таблиця 2.1 – Основні компоненти інформаційної системи

Компонент	Призначення
LLM-клієнт	Ініціювання запитів до MCP-сервера та отримання структурованих результатів
MCP-сервер	Приймання запитів, маршрутизація викликів, взаємодія з модулями системи
Модуль Work.ua	Формування запитів, завантаження сторінок і парсинг вакансій з Work.ua
Модуль Djinni	Обробка параметрів пошуку, завантаження сторінок і парсинг вакансій з Djinni
Модуль валідації	Перевірка коректності вхідних параметрів перед виконанням запиту
Модуль збереження	Експорт отриманих результатів у файли JSON або CSV
Веб-ресурси	Зовнішні джерела даних про вакансії

Функції безпосереднього збору даних будуть винесені в окремі модулі, кожен з яких відповідатиме за конкретне джерело. Для Work.ua передбачається реалізувати модуль, який виконуватиме побудову URL-адреси пошуку, надсилання HTTP-запиту, обробку HTML-сторінки та виділення основних даних про вакансії. Для Djinni буде створено окремий модуль із подібною логікою, але з урахуванням специфіки параметрів пошуку, структури сторінок і можливих редіректів цього ресурсу.

Таке розділення дозволить ізолювати логіку роботи з кожним веб-ресурсом. У разі зміни структури HTML-сторінок Work.ua або Djinni потрібно буде оновити лише відповідний модуль парсингу, не змінюючи загальну архітектуру MCP-

сервера. Це підвищить гнучкість інформаційної системи та спростить її подальший супровід.

Окремим компонентом архітектури буде модуль збереження результатів. Він забезпечуватиме експорт отриманих даних у зовнішні файли. У межах цього компонента планується реалізувати перевірку безпечності шляху до файлу, обмеження збереження визначеним каталогом експорту та підтримку форматів JSON і CSV. Завдяки цьому інформаційна система зможе використовуватися не лише для інтерактивного отримання відповідей у LLM-клієнті, а й для накопичення даних у межах подальшого аналізу.

Взаємодію MCP-компонентів доцільно розглядати як послідовний процес обробки запиту, що відповідає підходам описаним у сучасних дослідженнях застосування MCP [27]. Спочатку LLM-клієнт надсилатиме виклик інструмента із зазначенням його назви та набору параметрів. Далі MCP-сервер перевірятиме отримані аргументи, визначатиме потрібний модуль обробки, формуватиме URL-адресу запиту до відповідного веб-ресурсу та виконуватиме завантаження HTML-сторінки. Після цього модуль виділяє необхідні дані, а сервер форматує відповідь у форматі JSON. За потреби результат додатково зберігатиметься у файл.

Такий поділ етапів робить роботу системи більш зрозумілою та контрольованою, адже кожен компонент відповідає лише за свою частину процесу. Крім того, у разі помилки легше визначити, на якому саме етапі вона виникла. Основні етапи обробки запиту наведено в **таблиці 2.2.**

Таблиця 2.2 – Етапи обробки запиту в інформаційній системі

Етап	Опис
Отримання запиту	LLM-клієнт надсилає виклик MCP-інструмента
Валідація параметрів	Система перевіряє структуру та коректність вхідних даних
Формування URL	Створюється адреса запиту до Work.ua або Djinni
HTTP-запит	Виконується завантаження HTML-сторінки з веб-ресурсу
Парсинг HTML	Із вмісту сторінки виділяються потрібні дані про вакансії

Кінець таблиці 2.2

Етап	Опис
Формування відповіді	Результат перетворюється у структурований JSON-об’єкт
Експорт даних	За потреби результат зберігається у файл

Отже, архітектура інформаційної системи збору даних передбачатиме модульну побудову з чітким розподілом функцій між компонентами. MCP-сервер відповідатиме за взаємодію з клієнтом і маршрутизацію викликів, окремі модулі забезпечуватимуть збір і парсинг даних з Work.ua та Djinni, механізм валідації контролюватиме коректність параметрів, а модуль збереження відповідатиме за експорт результатів. Така структура забезпечить керованість розроблення, можливість подальшого розширення та сумісність інформаційної системи з MCP-клієнтами.

2.3 Проєктування механізму збереження вакансій

Механізм збереження вакансій в інформаційній системі буде потрібний для того, щоб після збору даних результат можна було записати у файл і використати пізніше. Цей компонент не відповідатиме за пошук вакансій або парсинг сторінок, а працюватиме лише з уже підготовленими даними.

Логіку збереження планується винести в окремий модуль storage.py. Це дозволить розділити обов’язки між частинами системи: модулі Work.ua та Djinni відповідатимуть за отримання й обробку даних, а модуль збереження за запис результатів у файл.

В інформаційній системі передбачається два способи збереження результатів. Перший спосіб – це автоматичне збереження через параметр export_file. У цьому випадку користувач зможе вказати назву файлу під час виклику інструменту, і результат буде збережено одразу після виконання запиту. Другий спосіб – це використання окремого інструменту s_save_payload, який дозволятиме зберігати вже підготовлені дані у форматі JSON або CSV.

Основним форматом збереження буде JSON, оскільки він добре підходить для структурованих даних і часто використовується для обміну інформацією між програмними компонентами[28]. У цьому форматі зручно зберігати як список вакансій, так і додаткову службову інформацію про запит, наприклад джерело даних, параметри пошуку або кількість знайдених записів.

Додатково передбачається підтримка формату CSV. Він буде корисним тоді, коли зібрані вакансії потрібно переглянути у вигляді таблиці або використати в табличних редакторах. Формат CSV є поширеним способом подання табличних даних у текстовому вигляді[29].

Після успішного збереження інформаційна система повинна повертати шлях до створеного файлу. Для цього до результату можна додавати поле exported_to, у якому буде вказано місце збереження. Завдяки цьому користувач одразу бачитиме, де знаходиться файл із результатами.

Також механізм збереження має бути пов’язаний із валідацією параметрів. Якщо користувач передасть неправильний шлях, невідомий формат або файл із невідповідним розширенням, система повинна повернути повідомлення про помилку. Для такої перевірки доцільно використовувати механізми валідації, які дозволяють контролювати типи даних, обов’язкові поля та допустимі значення ще до виконання основної логіки.

Отже, механізм збереження вакансій буде окремим сервісним компонентом інформаційної системи. Він забезпечуватиме запис результатів у JSON або CSV, перевірку безпечності шляху, збереження файлів у визначеному каталозі та повернення інформації про створений файл. Такий підхід зробить систему зручнішою для подальшого аналізу та повторного використання зібраних даних.

2.4 Модель виклику MCP-інструменту агентом та логіка обробки запитів

Модель виклику MCP-інструменту в інформаційній системі буде побудована на взаємодії між агентом, який підтримує протокол Model Context Protocol, та

локальним MCP-сервером. Сервер працюватиме через транспорт stdio, прийматиме запити у форматі JSON-RPC і повертатиме результати у структурованому вигляді.

Робота агента з MCP-сервером складатиметься з двох основних етапів. Спочатку агент отримуватиме список доступних інструментів через запит tools/list. На цьому етапі він дізнаватиметься, які дії може виконувати сервер: пошук вакансій, отримання деталей вакансії або збереження результатів. Після цього агент обиратиме потрібний інструмент і викликатиме його через tools/call, передаючи параметри у вигляді JSON-об’єкта. акий механізм відповідає стандартній моделі інструментальної взаємодії, визначеній у специфікації MCP [30]. Після отримання запиту сервер перевірятиме передані аргументи. Для цього планується використовувати моделі Pydantic[31], які дозволять контролювати типи даних, обов’язкові поля та правильність значень. Якщо параметри будуть некоректними, сервер повертатиме повідомлення про помилку без виконання запиту до зовнішнього сайту.

Якщо перевірка пройде успішно, сервер визначатиме, який саме модуль потрібно використати. Для роботи з Work.ua запит передаватиметься до відповідного модуля скрапінгу, для Djinni до окремого модуля, а для збереження даних до модуля експорту. Такий підхід дозволить розділити логіку обробки запитів і зробити інформаційну систему більш зрозумілою та зручною для підтримки.

Для інструментів пошуку або отримання деталей вакансії сервер формуватиме URL-адресу запиту до Work.ua або Djinni. Після цього виконуватиметься HTTP-запит, завантажуватиметься HTML-сторінка, а потрібні дані виділятимуться за допомогою парсингу. У результаті система зможе отримати назву вакансії, компанію, посилання, зарплату, опис та інші доступні поля. Результат роботи інструменту повертатиметься агенту у форматі JSON.

Таблиця 2.3 – Загальна логіка обробки запиту

Етап	Опис
Отримання списку інструментів	Агент надсилає запит tools/list
Вибір інструменту	Агент визначає, яку дію потрібно виконати
Виклик інструменту	Агент надсилає tools/call з параметрами
Валідація параметрів	Сервер перевіряє коректність аргументів
Обробка запиту	Виконується пошук, парсинг або збереження даних
Формування відповіді	Результат перетворюється у JSON
Повернення результату	Сервер надсилає відповідь агенту

Це важливо, оскільки агент отримуватиме не звичайний текст, а структуровані дані, які можна далі аналізувати, узагальнювати або використовувати для наступних викликів.

Отже, модель виклику MCP-інструменту буде побудована як зрозумілий послідовний процес: агент обирає потрібний інструмент, передає параметри, сервер перевіряє їх, виконує потрібну дію та повертає результат у форматі JSON. Такий підхід дозволить зробити інформаційну систему зручною для взаємодії з LLM-клієнтом і придатною для подальшого розширення.

Висновки до другого розділу

У другому розділі було визначено основні проєктні рішення для створення інформаційної системи автоматизованого збору даних про вакансії з веб-ресурсів Work.ua та Djinni.

Було сформульовано задачу розроблення системи, яка повинна взаємодіяти з LLM-клієнтом через протокол Model Context Protocol, приймати структуровані запити, отримувати HTML-сторінки з веб-ресурсів, виконувати парсинг потрібних даних і повертати результат у форматі JSON.

Також було описано архітектуру інформаційної системи. Вона передбачає поділ на кілька основних компонентів: MCP-сервер, модулі роботи з Work.ua та

Djinni, механізм валідації параметрів і модуль збереження результатів. Такий підхід робить систему зрозумілою, модульною та зручною для подальшого розширення.

Окрему увагу було приділено механізму збереження вакансій. Передбачено можливість експорту результатів у файли JSON або CSV, перевірку безпечності шляху до файлу та повернення інформації про місце збереження результату. Це дозволяє використовувати зібрані дані не лише в межах одного запиту, а й для подальшого аналізу.

Було також розглянуто модель виклику MCP-інструменту агентом. Визначено, що взаємодія відбуватиметься через послідовність дій: отримання списку інструментів, вибір потрібного інструменту, передача параметрів, валідація, виконання запиту, формування JSON-відповіді та, за потреби, збереження результатів.

Отже, у другому розділі було сформовано загальну структуру майбутньої інформаційної системи, визначено її основні компоненти, логіку їх взаємодії та вимоги до обробки й збереження даних. Запропоновані проєктні рішення створюють основу для подальшої програмної реалізації MCP-інструменту.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ НА ОСНОВІ MCP-ІНСТРУМЕНТУ ДЛЯ ВЕБСКРАПІНГУ

3.1 Вибір середовища розробки та бібліотек

Для реалізації програмної частини дипломного проєкту обрано мову програмування Python версії 3.10 і вище. Такий вибір був зумовлений тим, що ця мова має розвинену екосистему бібліотек для мережевої взаємодії, аналізу HTML-документів, валідації структур даних і побудови прикладних серверів. Для цієї дипломної роботи це є особливо важливим, оскільки розроблений застосунок являє собою MCP-сервер, який приймає виклики від LLM-клієнта, виконує HTTP-запити до сайтів Work.ua та Djinni, здійснює парсинг сторінок вакансій і повертає структуровані результати.

Під час вибору середовища розробки та програмних засобів було враховано такі основні вимоги:

- підтримка асинхронної мережевої взаємодії;
- наявність засобів для парсингу HTML-сторінок;
- можливість побудови MCP-сумісного серверного застосунку;
- формалізований опис і валідація вхідних параметрів;
- зручність тестування, пакування та подальшої підтримки проєкту.

Середовище розробки проєкту побудовано за сучасною для Python-систем схемою. Конфігурація залежностей, метаданих пакета та параметрів збирання зосереджена у файлі *pyproject.toml*, а відтворюваність набору встановлених пакетів забезпечується файлом фіксації залежностей. Структура вихідного коду організована за підходом *src/*, де основна логіка винесена до пакета **mcp_server_work_ua**. Такий спосіб організації є доцільним для серверного застосунку, оскільки він спрощує підтримку модулів, розділяє транспортний рівень, парсинг та підсистему збереження результатів, а також полегшує тестування й пакування проєкту. Основні обрані програмні засоби наведено у таблиці 3.1.

Таблиця 3.1 – Основні програмні засоби

Компонент	Призначення у проєкті	Обґрунтування вибору
Python 3.10+	Базова мова реалізації MCP-сервера, модулів парсингу та тестів	Дає змогу поєднати серверну логіку, асинхронні HTTP-виклики, парсинг HTML і зручне тестування в межах однієї платформи
httpx [32]	Отримання сторінок пошуку та сторінок вакансій через HTTP	Бібліотека підтримує синхронну й асинхронну модель роботи, керування заголовками, перенаправленнями та тайм-аутами, що добре узгоджується з архітектурою MCP-сервера
Beautiful Soup [33]	Аналіз HTML-вмісту сторінок Work.ua та Djinni	Дає змогу зручно шукати елементи за тегами, класами, атрибутами та CSS-селекторами, чого достатньо для вилучення назв вакансій, компаній, зарплат, локацій і описів
mcp [34]	Реалізація Model Context Protocol на серверній стороні	Забезпечує побудову Server, опис інструментів, обробку tools/call і повернення результатів у стандартизованому форматі
Pydantic [31]	Моделі аргументів інструментів і валідація вхідних даних	Дозволяє формалізувати параметри запитів, перевіряти їх до виконання основної логіки та генерувати JSON Schema для MCP-клієнта
pytest [35]	Модульне тестування функцій парсингу, збереження та валідації	Забезпечує просту організацію тестів і зручну перевірку окремих частин системи
pytest-asyncio [36]	Перевірка асинхронних сценаріїв роботи	Дає змогу коректно тестувати функції, що використовують asyncio та асинхронні HTTP-виклики
Hatchling [37]	Пакування та збирання застосунку	Підтримує сучасний спосіб опису збірки через pyproject.toml і спрощує формування дистрибутива Python-пакета

Використання бібліотеки **httpx** [32] у цьому проєкті є технічно виправданим, оскільки MCP-сервер повинен звертатися до зовнішніх вебресурсів і отримувати HTML-вміст сторінок. У поточній реалізації це дозволяє завантажувати як сторінки пошуку, так і детальні сторінки вакансій через єдиний механізм мережевого доступу. Для серверного застосунку це особливо зручно, бо асинхронне виконання HTTP-запитів добре узгоджується із загальною архітектурою системи.

Бібліотека Beautiful Soup [33] обрана як основний інструмент для розбору HTML-коду. У межах цього проєкту її можливостей достатньо для роботи з типовою структурою сторінок вакансій, де основні дані представлені у вигляді HTML-блоків. Як можливий напрям оптимізації в подальшому можна розглядати підключення lxml [40], якщо виникне потреба пришвидшити розбір HTML або розширити функціональність обробки XML-подібних документів.

Принципово важливим для цієї роботи є використання бібліотеки mcp [34], оскільки розроблений застосунок повинен функціонувати не як окремий локальний скрапер, а як інструмент у складі MCP-сумісної інтелектуальної системи. Саме ця бібліотека використовується для побудови *Server*, опису списку доступних інструментів, обробки викликів *tools/call* і повернення результатів у стандартизованому вигляді. Таким чином, вибір спеціалізованого SDK для протоколу MCP безпосередньо відповідає темі дипломної роботи.

Для опису аргументів інструментів і перевірки коректності вхідних даних використано Pydantic [31]. У межах проєкту це дозволяє формувати моделі параметрів для пошуку вакансій, отримання детальної інформації та збереження результатів у JSON або CSV. Перевага такого підходу полягає в тому, що помилки у вхідних даних виявляються ще до запуску основної логіки парсингу, а структура аргументів може бути автоматично представлена у вигляді JSON Schema, що є зручним для MCP-клієнтів.

Окрему роль у реалізації проєкту відіграють засоби стандартної бібліотеки Python. Їх використання узагальнено в такому списку:

– модуль *json* застосовується для серіалізації результатів і формування текстового вмісту відповідей;

– модуль *csv* використовується для експорту зібраних даних у табличному форматі;

– модулі *pathlib* та *os* забезпечують роботу з каталогами, файлами й нормалізацією шляхів;

– модуль *io* використовується під час формування CSV-вмісту в пам’яті;

– модуль *re* застосовується для перевірки допустимості відносних шляхів і службової обробки рядків.

Таке рішення є доцільним, оскільки функції збереження в цьому проєкті мають сервісний характер, а стандартних засобів Python достатньо для безпечної реалізації експорту в JSON і CSV без потреби у підключенні додаткових зовнішніх бібліотек.

Для перевірки коректності реалізації використано **pytest** [35] як основний інструмент модульного тестування. Через нього перевіряються:

– функції парсингу сторінок Work.ua;

– функції парсингу сторінок Djinni;

– валідація вхідних параметрів інструментів;

– робота підсистеми збереження результатів.

Оскільки серверна логіка містить асинхронні виклики, додатково використано pytest-asyncio [36], який дає змогу коректно запускати та перевіряти асинхронні сценарії. У результаті забезпечується роздільне тестування окремих частин системи без необхідності щоразу виконувати повний цикл взаємодії з реальним MCP-клієнтом.

Для пакування та збирання застосунку використано Hatchling [37] як backend збирання, визначений у *pyproject.toml*. Такий вибір є доцільним для проєкту, який має поширюватися як окремий Python-пакет із точкою входу **mcp-server-work**. Використання *hatchling* спрощує формування дистрибутива, підтримує сучасний

спосіб опису збірки через *pyproject.toml* і робить інсталяцію сервера більш передбачуваною як у локальному середовищі, так і при інтеграції з MCP-клієнтами.

Крім основного стеку, для подальшого розвитку системи доцільно враховувати й перспективні засоби розширення. Вони наведені в таблиці 3.2.

Таблиця 3.2 – Засоби використання

Засіб	Потенційне використання	Доцільність застосування
logging [38]	Фіксація службових подій, помилок HTTP-запитів і результатів обробки	Корисний для налагодження, супроводу та аналізу збоїв під час розвитку системи
Playwright [39]	Робота зі сторінками, що формуються динамічно на стороні браузера	Доцільний у разі, якщо звичайного HTTP-запиту буде недостатньо для отримання повного вмісту сторінки
lxml [40]	Прискорений розбір HTML або XML	Може бути застосований як продуктивніша альтернатива для окремих сценаріїв обробки документів
Scrapy [41]	Масштабні crawler-сценарії та обхід великої кількості сторінок	Доцільний при переході від компактного MCP-інструмента до повноцінної системи масового збирання даних

Таким чином, для програмної реалізації проєкту обрано поєднання Python, бібліотек httpx [32], Beautiful Soup [33], mcp [34], Pydantic [31], pytest [35], pytest-asyncio [36] і Hatchling [37], а також стандартних модулів Python для серіалізації, файлових операцій і службової логіки. Такий набір інструментів відповідає реальним потребам створеного MCP-сервера, забезпечує модульність реалізації, підтримує тестованість і робить систему придатною як для практичного використання, так і для подальшого розширення в межах тематики дипломної роб.оти

3.2 Реалізація MCP-сервера та набору інструментів

Основне призначення сервера полягає в тому, щоб надати LLM-клієнту стандартизований доступ до функцій пошуку вакансій, отримання детальної інформації про окремі вакансії, а також збереження результатів у форматах JSON і CSV. На відміну від звичайного локального скрапера, розроблений застосунок функціонує як інструмент у межах Model Context Protocol, тому архітектура серверної частини орієнтована не лише на парсинг HTML, а й на коректний опис інструментів, валідацію параметрів, формування уніфікованих відповідей і обробку помилок відповідно до вимог MCP [34].

Для того, щоб відокремлювати транспортний рівень, парсинг різних сторінок сайтів, валідацію аргументів, і окремо підсистему експорту результатів, ми використовували модульний принцип. І саме такий підхід є доцільним, оскільки:

- розширення набору інструментів буде можливе без зміни базової архітектури;
- наявне індивідуальне тестування компонентів;
- в разі змін структури HTML-сторінок або параметрів інструментів, його підтримка не стає важкою дією. Основні модулі системи наведено в таблиці 3.3.

Реалізація сервера була почата з функції *main(),* Вона розташована в модулі *__init__.py* та відповідає за початковий запуск програми. Вона зчитує аргументи командного рядка, за потреби обробляє параметр *--user-agent*, а після чого запускає основну асинхронну функцію *serve()* за допомогою *asyncio.run(...)*. Що дозволяє використовувати main() як стартову точку для початку роботи MCP-сервера. Саме такий підхід дозволить використовувати власний User-Agent для HTTP-запитів, що є гарним рішенням під час інтеграції з різними клієнтами або для налагодження мережевої взаємодії [32].

MCP-сервер створюється в модулі **server.py** через об’єкт *Server("mcp-work-ua")*. Взаємодія з клієнтом відбувається через *stdio_server*, тобто сервер працює

через стандартні потоки введення та виведення. Такий режим є типовим для локального MCP-застосунку [34] і має такі переваги:

– для нього не потрібен окремий HTTP-сервер або відкриття мережевого порту;

– легко інтегрується з локальними IDE та агентами;

– дозволяє стандартизовано обмінюватися JSON-RPC-подібними повідомленнями між клієнтом і сервером.

Таблиця 3.3 – Загальна логіка обробки запиту

Модуль	Основне призначення	Реалізована функціональність
__init__.py	Точка входу до пакета	Містить функцію main(), яка обробляє аргумент --user-agent і запускає сервер через asyncio.run(serve(...))
__main__.py	Підтримка запуску як модуля	Дозволяє запускати сервер командою python -m mcp_server_work_ua
server.py	Центральний модуль MCP-сервера	Створення об’єкта Server, опис інструментів, валідація аргументів, маршрутизація викликів, обробка винятків, формування відповідей
scraper.py	Робота з Work.ua	Формування URL пошуку, завантаження HTML, парсинг списку вакансій, парсинг детальної сторінки, нормалізація URL
djinni.py	Робота з Djinni	Побудова URL з фільтрами, парсинг списку вакансій, парсинг детальної сторінки, формування знімка параметрів пошуку
storage.py	Збереження результатів	Безпечна нормалізація шляхів, запис JSON, формування CSV, експорт результатів у файл
tests/	Перевірка коректності реалізації	Містить модульні тести для парсерів, валідації параметрів і підсистеми експорту [35], [36]

Загальна послідовність ініціалізації сервера є такою:

– користувач або MCP-клієнт запускає команду *mcp-server-work*;
– викликається функція *main();*
– через *asyncio.run(...)* запускається функція *serve(custom_user_agent=...)*;
– створюється екземпляр Server;
– реєструються обробники *list_tools() і call_tool();*
– відкривається *stdio_server();*
– сервер переходить у режим очікування запитів від клієнта.

Такий життєвий цикл є простим, але достатній для реалізації прикладного MCP-сервера в межах дипломного проєкту. Невеликий фрагмент коду, що демонструє запуск сервера, наведено нижче.

```
def main() -> None:
    parser = argparse.ArgumentParser(...)
    parser.add_argument("--user-agent", type=str, default=None)
    args = parser.parse_args()
    asyncio.run(serve(custom_user_agent=args.user_agent))
```

У свою чергу, транспортний рівень запускається через stdio_server(), Повний лістинг MCP-сервера доцільно винести в **ДОДАТОК А**:

```
options = server.create_initialization_options()
async with stdio_server() as (read_stream, write_stream):
    await server.run(read_stream, write_stream, options, raise_exceptions=False)
```

Однією з ключових частин реалізації є опис параметрів інструментів через моделі Pydantic [31]. Це дозволяє задати структуру та типи вхідних даних; встановити обмеження (наприклад, для номера сторінки або кількості результатів); автоматично генерувати JSON Schema для MCP-клієнта через model_json_schema(); відхиляти некоректні або неповні виклики ще до виконання основної логіки. Моделі аргументів, реалізовані в server.py, наведено в таблиці 3.4.

Таблиця 3.4 – Моделі аргументів

Модель	Призначення	Основні поля
SearchVacancies	Пошук вакансій на Work.ua	query, page, max_results, export_file, include_job_details
VacancyDetails	Отримання повного опису вакансії Work.ua	vacancy, export_file
DjinniSearch	Пошук вакансій на Djinni	keywords, page, max_results, search_type, salary, exp_level, employment, company_type, english_level, domain, region, editorial, hide_contacted, show_stale, company, primary_keyword, extra_query, export_file, include_job_details
DjinniJobRef	Отримання повного опису вакансії Djinni	job, export_file
SavePayload	Явне збереження переданих даних у файл	format, export_file, data

Для моделі SavePayload важливим є додавання валідатора, який перевіряє відповідність розширення файла обраному формату. Для прикладу, якщо користувач указує format="json", то назва файлу повинна завершуватися на .json. Якщо ж указано format="csv", файл повинен мати розширення .csv. Завдяки цьому некоректний запит відхиляється ще на етапі перевірки параметрів. Фрагмент валідатора наведено нижче.

```
@model_validator(mode="after")
def extension_matches_format(self) -> SavePayload:
    suf = Path(self.export_file.strip()).suffix.lower()
    if self.format == "json" and suf != ".json":
        raise ValueError("Для format=json поле export_file має закінчуватися на .json")
    if self.format == "csv" and suf != ".csv":
        raise ValueError("Для format=csv поле export_file має закінчуватися на .csv")
    return self
```

Такий підхід підвищує надійність роботи інструмента, оскільки не дозволяє зберігати дані у файл із невідповідним розширенням.

Було розроблено п'ять інструментів для роботи з вакансіями та збереженням даних для MCP-сервер. Їх реєстрація виконується у функції list_tools(). У цій функції для кожного інструмента задається унікальна назва, короткий опис його призначення та схема вхідних параметрів inputSchema, яка автоматично формується на основі відповідної моделі Pydantic. Цей підхід є пріоритетним, адже

MCP-клієнт може заздалегідь отримати список доступних інструментів, зрозуміти, для чого вони призначені, та які самі параметри потрібно передати для виклику кожного з них. Набір реалізованих інструментів подано в таблиці 3.5.

Таблиця 3.5 – Інструменти MCP-сервера

Ім’я інструмента	Призначення
s_work_ua_search_vacancies	Пошук вакансій на Work.ua зі списковой сторінки
s_work_ua_vacancy_details	Отримання повного опису окремої вакансії Work.ua
s_djinni_search_jobs	Пошук вакансій на Djinni з підтримкою розширених фільтрів
s_djinni_job_details	Отримання повного опису окремої вакансії Djinni
s_save_payload	Збереження довільного payload у форматі JSON або CSV

Перші чотири інструменти реалізують два основні сценарії: отримання списків вакансій за параметрами пошуку та отримання детальної інформації про конкретну вакансію. П’ятий інструмент (s_save_payload) виконує сервісну функцію – дозволяє явно зберегти будь-які структуровані дані. Більш детальна реалізація інструментів пошуку та парсингу, а саме робота з **scraper.py та djinni.py** буде наведена в підрозділі 3.3. Також, механізми експорту результатів (робота з **storage.py**, автоматичне та явне збереження) розкрито в підрозділі 3.4.

Важливо зазначити, що у функції **call_tool()** реалізовано єдиний механізм перехоплення винятків. Завдяки цьому клієнт отримує не внутрішні помилки Python, а стандартизовані MCP-помилки:

– INVALID_PARAMS – при помилках валідації Pydantic, некоректних шляхах або виклику невідомого інструмента;

– INTERNAL_ERROR – при збоях мережевої взаємодії (наприклад, помилка HTTP).

Приклад перетворення нижче.

```
except ValidationError as e:
    raise McpError(ErrorData(code=INVALID_PARAMS, message=str(e))) from e


except httpx.HTTPError as e:
    raise McpError(
        ErrorData(code=INTERNAL_ERROR, message=f"Помилка мережі: {e!s}")
```

```
) from e
```

Такий підхід забезпечить передбачуваний формат відповіді, навіть у разі, коли у нас виникають помилки, і спрощує інтеграцію з клієнтом.

3.3 Реалізація модуля збору та структуризації вакансій

У передньому підрозділі було вже розглянуто реалізацію MCP-сервера та механізм прийняття викликів інструментів, перевірку вхідних параметрів і формування відповідей для клієнта. Також центральною прикладною частиною розробленої інформаційної системи є модуль для збору та структуризації вакансій, адже саме він забезпечує отримання даних із зовнішніх вебресурсів, вилучення корисної інформації з HTML-сторінок, її нормалізацію та перетворення у формат, придатний для подальшої обробки засобами MCP.

Функціональність збору вакансії реалізована з допомогою двох основних модулів **scraper.py та djinni.py**. Перший модуль відповідає за взаємодію із сайтом Work.ua, другий – за роботу з платформою Djinni. Таке розділення дозволяє ізолювати логіку обробки різних джерел даних і спростити подальший супровід системи. Таке розділення дозволяє ізолювати логіку обробки різних джерел, спростити супровід та модульне тестування. Основними завданнями модуля є:

- формування URL для пошуку та сторінок окремих вакансій;
- завантаження HTML через HTTP-запити [32];
- парсинг HTML-структури сторінок за допомогою BeautifulSoup [33];
- вилучення ключових полів вакансії;
- нормалізація посилань, назв компаній, зарплат і метаданих;
- збагачення коротких результатів повною інформацією з детальних сторінок;
- підготовка структурованого результату для повернення через MCP-сервер.

Загальна логіка модуля збору в загальній структурі така: server.py приймає виклик інструмента, моделі Pydantic перевіряють параметри [31], після чого сервер

звертається до функцій scraper.py або djinni.py. Далі відповідний модуль формує URL, завантажує HTML-сторінку, виконує парсинг і повертає структуровані дані назад до server.py, де результат серіалізується у JSON. Відповідність між джерелами вакансій та модулями обробки наведено в таблиці 3.6.

Таблиця 3.6 – Відповідність між джерелами вакансій та модулями обробки

Джерело вакансій	Основний модуль	Основні функції
Work.ua	scraper.py	build_search_url, resolve_vacancy_url, parse_vacancy_list, parse_vacancy_detail, search_vacancies, fetch_vacancy_detail
Djinni	djinni.py	build_djinni_search_url, resolve_djinni_job_ref, parse_djinni_list, parse_djinni_job_page, search_djinni_jobs, fetch_djinni_job_detail

Такий підхід є доцільним, оскільки кожне джерело має власну HTML-структуру та правила фільтрації. Ізоляція логіки в окремих модулях зменшує взаємозалежність компонентів, дає змогу змінювати парсер одного сайту без впливу на інший і спрощує модульне тестування [35, 36].

Попри існуючі відмінності, між Work.ua та Djinni загальний сценарій роботи для обох джерел є однаковим:

- формування URL сторінки пошуку або детальної сторінки вакансії;
- виконання асинхронного HTTP GET-запиту через httpx.AsyncClient;
- передача отриманого HTML у парсер на основі BeautifulSoup;
- вилучення ключових полів вакансії;
- (опціонально) збагачення результатів даними з детальної сторінки;
- повернення структурованого об'єкта (список вакансій + блок meta).

Такий підхід дозволить нам розділити мережну взаємодію та аналіз HTML, , що дозволяє змінювати правила формування URL, параметри HTTP-клієнта або CSS-селектори парсингу незалежно.

Перейдемо до реалізації парцера для Work.ua. Модуль **scraper.py** містить усі необхідні компоненти для роботи з сайтом Work.ua. Основні елементи наведено в таблиці 3.7. Повний текст модуля винесено в **ДОДАТОК Б**.

Таблиця 3.7 – Основні елементи модуля scraper.py

Елемент	Призначення
WORK_UA_ORIGIN	Базова адреса сайту https://work.ua
DEFAULT_USER_AGENT	Стандартний User-Agent для HTTP-запитів
_JOB_PATH	Регулярний вираз для перевірки шляху вакансії /jobs/<id>/
_looks_like_salary_text()	Евристика для визначення, чи є рядок зарплатним полем
_extract_company_from_card()	Виділення назви компанії з картки вакансії
_extract_salary_from_card()	Виділення зарплати зі спискової картки
VacancySummary	Структура короткого представлення вакансії
build_search_url()	Формування URL сторінки пошуку
resolve_vacancy_url()	Нормалізація URL окремої вакансії
parse_vacancy_list()	Парсинг спискової сторінки
parse_vacancy_detail()	Парсинг детальної сторінки вакансії
merge_work_list_with_detail()	Об’єднання скорочених і повних даних
search_vacancies()	Повний сценарій пошуку та структуризації списку
fetch_vacancy_detail()	Отримання структурованої інформації про одну вакансію

Далі доцільним буде розглянути формування та нормалізацію URL для даного вебсайту. Для цього ми використовуємо функцію **build_search_url(query, page**) яка формує адресу сторінки пошуку Work.ua на основі ключового слова та номера сторінки. Пошуковий рядок очищується від зайвих пробілів, кодується через *quote_plus*, а номер сторінки обмежується мінімальним значенням 1.

```
def build_search_url(query: str, page: int) -> str:
q = quote_plus(query.strip())
p = max(1, page)
return f"{WORK_UA_ORIGIN}/jobs/?ss=1&search={q}&page={p}"
```

Для окремих вакансій використовується функція **resolve_vacancy_url(url_or_id).** Вона підтримує числовий ідентифікатор, відносний шлях виду */jobs/<id>/* або повний URL. У всіх випадках результатом є

канонічне посилання https://work.ua/jobs/<id>/, що забезпечує однакову подальшу обробку вакансій.

Окремо варто приділити увагу розпізнаванню зарплати та парсинг на Work.ua, бо під час парсингу спискової сторінки важливо відокремити зарплатне поле від назви компанії або інших елементів картки. Для цього використовується функція _looks_like_salary_text(text), яка перевіряє наявність маркерів грн, ₴, валютних позначень або числового діапазону.

```
def _looks_like_salary_text(text: str) -> bool:
t = text.strip().lower()
if "грн" in t or "₴" in t:
return True
if re.search(r"(?:^|\\s)(?:usd|eur|uah|\\$|€)(?:\\s|$)", t, re.I):
return True
return bool(re.search(r"\\d{1,3}(?:[\\s\\u202f\\u00a0]\\d{3})+\\s*[-—−]\\s*\\d", t))
```

Також існують дві додаткові функції, які теж варто дорого розглянути. Функція **parse_vacancy_list(html)** перетворює HTML-код у дерево елементів, знаходить посилання, що відповідають шаблону */jobs/<id>/,* відкидає дублікати та формує об’єкти VacancySummary. Для кожної вакансії визначаються назва, компанія, локація, зарплата та URL. Функція **parse_vacancy_detail(html, page_url)** виконує глибший аналіз сторінки конкретної вакансії. Вона вилучає назву вакансії, повний опис, компанію, зарплату та остаточне посилання. Детальна сторінка використовується тоді, коли потрібно отримати не лише короткий результат, а й повний зміст вакансії.

Для поєднання короткої інформації зі списку та повного опису використовується функція **merge_work_list_with_detail(list_row, detail)**. Вона оновлює назву, компанію, зарплату, додає поле description і зберігає канонічний URL. Такий механізм застосовується в режимі *include_job_details=True*. Коротка інформація про вакансію зберігається у структурі **VacancySummary.**

Таблиця 3.8 – Поля структури VacancySummary

Поле	Зміст
job_id	Числовий ідентифікатор вакансії
title	Назва вакансії
company	Назва компанії
location	Місце роботи
salary	Текстове представлення зарплати
url	Канонічне повне посилання на вакансію

Окрім списку вакансій, функція **search_vacancies(...)** формує блок meta, який містить службову інформацію про пошук.

Таблиця 3.9 – Структура блоку meta для Work.ua

Поле meta	Зміст
search_url	Повний URL сторінки пошуку
filters	Об’єкт із параметрами query та page
page	Номер сторінки пошуку
total_parsed_on_page	Загальна кількість вакансій, знайдених на сторінці
returned	Кількість вакансій, що потрапили до підсумкового результату
include_job_details	Ознака, чи виконувалося збагачення деталями
source	Джерело даних, тобто work.ua

Наявність метаданих дозволяє відтворити пошуковий запит, проаналізувати кількість результатів і використати дані для подальшого експорту.

Перейдемо до розгляду реалізації збору вакансій з сайту Djinni. Модуль **djinni.py** виконує подібні функції, однак адаптований до структури сайту Djinni та його розширеної системи фільтрів. Він відповідає за побудову URL пошуку, нормалізацію посилань, парсинг спискових і детальних сторінок, а також

формування метаданих пошуку. Повний текст модуля обробки Djinni доцільно винести в **ДОДАТОК В** (djinni.py).

Таблиця 3.10 – Основні елементи модуля djinni.py

Елемент	Призначення
DJINNI_ORIGIN	Базова адреса сайту https://djinni.co
_DJINNI_JOB_HREF	Регулярний вираз для посилань вакансій Djinni
DjinniJobSummary	Коротка структура представлення вакансії
DjinniListFilters	Формалізований набір фільтрів пошуку
djinni_filters_snapshot()	Побудова повного знімка параметрів пошуку
build_djinni_search_url()	Формування URL спискової сторінки з фільтрами
resolve_djinni_job_ref()	Нормалізація посилань на окрему вакансію
parse_djinni_list()	Парсинг спискової сторінки вакансій
parse_djinni_job_page()	Парсинг повної сторінки вакансії
merge_djinni_list_with_detail()	Збагачення спискових даних повним описом
search_djinni_jobs()	Повний сценарій пошуку вакансій
fetch_djinni_job_detail()	Отримання детальної структурованої інформації

Для опису параметрів пошуку використано *dataclass DjinniListFilters*, який містить поля *search_type, salary, exp_level, employment, company_type, english_level, domain, region, editorial, hide_contacted, show_stale, company ma primary_keyword.* Ця структура відокремлює внутрішню логіку пошуку від моделей валідації в server.py і використовується як для побудови URL, так і для формування метаданих.

Функція **build_djinni_search_url(...)** формує URL сторінки вакансій Djinni. Ключові слова передаються в параметрі *all_keywords*, номер сторінки – у *page*, тип пошуку – у *search_type*, а додаткові фільтри додаються лише за наявності значення. Словник *extra_query* дозволяє передавати додаткові параметри й за потреби перевизначати стандартні.

```
params: dict[str, str] = {
"all_keywords": keywords.strip(),
"search_type": f.search_type,
"page": str(max(1, page)),
}
if f.salary is not None:
params["salary"] = str(int(f.salary))
if extra_query:
for k, v in extra_query.items():
if v is not None and str(v).strip() != "":
params[str(k).strip()] = str(v).strip()
```

Фінальне кодування параметрів виконується через *urlencode*, що забезпечує коректне представлення спеціальних символів у URL.

Окремо треба розглянути нормалізацію та парсинг в цьому модулі, і для цього використовуються три окремі функції, які будуть описані нижче.

Функція **resolve_djinni_job_ref(ref)** перетворює числовий ідентифікатор, відносний шлях */jobs/<id>-<slug>/* або повний URL у канонічне посилання на вакансію. Якщо формат некоректний, генерується **ValueError**, який обробляється на рівні MCP-сервера.

Функція **parse_djinni_list(html)** аналізує сторінку *djinni.co/jobs/:* знаходить блоки *div.job-item*, посилання **a.job_item__header-link[href^='/jobs/'],** перевіряє їх за регулярним виразом і вилучає поля *job_id, title, company, salary, meta_line, path, url*. Поле *meta_line* зберігає короткий контекст вакансії, наприклад формат роботи, регіон або досвід.

Функція **parse_djinni_job_page(html, page_url)** вилучає з детальної сторінки заголовок, компанію, повний опис і остаточний URL. На Djinni зарплатне поле не завжди подане в стабільному окремому блоці, тому основний акцент зроблено на отриманні повного опису вакансії.

Також невід'ємною частиною є структура даних Djinni. Для короткого представлення вакансії використовується **dataclass DjinniJobSummary.**

Таблиця 3.11 – Поля структури DjinniJobSummary

Поле	Зміст
job_id	Ідентифікатор вакансії
title	Назва вакансії
company	Назва компанії
Salary	Текстове представлення зарплати
meta_line	Додаткова коротка інформація з картки
path	Відносний шлях вакансії
url	Повний URL вакансії

Додатково тут відбувається формування розширеного блоку meta.filters, який буде містити в собі ключові слова, номер сторінки, тип пошуку, застосовані фільтри та параметри extra_query. Завдяки цьому результат можна відтворити або використати для подальшого експорту.

Отже, для обох джерел використовується HTTP-клієнт httpx [32]. Він виконує GET-запити, передає заголовки *User-Agent і Accept-Language*, підтримує перенаправлення, використовує тайм-аут і перевіряє HTTP-статус через raise_for_status(). Важливими характеристиками реалізації є асинхронне виконання запитів, повторне використання клієнта в межах одного сценарію пошуку, обробка редиректів і можливість передавання власного *User-Agent* через точку входу MCP-сервера. Після завантаження, парсингу та нормалізації функції **search_vacancies(...)** і **search_djinni_jobs(...)** формують підсумковий результат. Список знайдених вакансій обмежується значенням *max_results*, кожний *dataclass* перетворюється на словник через **asdict(...)**, за потреби додаються детальні дані, після чого формується блок *meta.*

Таблиця 3.12 – Порівняння структури підсумкового результату для Work.ua та Djinni

Характеристика	Work.ua
Основний список результатів	vacancies

Кінець таблиці 3.12

Характеристика	Work.ua
Ідентифікатор елемента	job_id
Повне посилання	url
Додатковий короткий контекст	location, salary
Метадані пошуку	meta
Знімок фільтрів	query, page
Можливість збагачення деталями	так

Попри різницю в HTML-структурі та фільтрах, обидва модулі формують узгоджений формат результатів. Це спрощує подальшу обробку даних у MCP-сервері, експорт результатів і використання інформації LLM-клієнтом.

Отже, модуль збору та структуризації вакансії у проєкті реалізовує повний цикл прикладної обробки даних, від початку форматування URL і виконання HTTP запитів, до вилучення ключових полів, нормалізації результатів, збагачення їх даними з деталі сторінок і підготовки структурованої відповіді для MCP-сервера.

Розділення логіки між модулями scraper.py і djinni.py, використання httpx [32] для мережевої взаємодії, BeautifulSoup [33] для аналізу HTML та перевірка через pytest [35] і pytest-asyncio [36] дозволили створити керований і розширюваний механізм отримання вакансій, придатний для інтеграції з інтелектуальною системою на основі MCP.

3.4 Реалізація інтеграції з агентом і механізму експорту результатів

Після того, як інформаційна система отримала та структурувала вакансії, вона має забезпечити взаємодії з агентом і можливість збереження цих результатів. У даній роботі ця функція реалізована через модулі **server.py, __init__.py та storage.py.**

Інтеграція побудована відповідно до підходу Model Context Protocol [34], де сервер працює не як класичний REST API, а як локальний MCP-процес із набором

формально описаних інструментів. Агент отримує перелік доступних інструментів, їхні схеми параметрів, виконує виклики та одержує структуровані відповіді. Основні завдання інтеграційного шару:

- запуск MCP-сервера як локального процесу;
- оголошення інструментів і схем їхніх параметрів;
- маршрутизація викликів агента до прикладних функцій;
- повернення результатів у форматі TextContent;
- автоматичне або явне збереження результатів у JSON і CSV;
- перевірка безпечності файлових шляхів.

Інтеграційний шар буде проміжною ланкою між модулем збору та агентом. Він буде приймати протокольні запити, перевіряти аргументи, викликати функції **scraper.py, djinni.py або storage.py**, після чого формувати відповідь у форматі, сумісному з MCP-клієнтом.

Таблиця 3.13 – Основні компоненти інтеграційного шару

Компонент	Розміщення	Призначення
main()	__init__.py	Точка входу, обробка параметра --user-agent
serve()	server.py	Ініціалізація MCP-сервера
Server("mcp-work-ua")	server.py	Центральний об’єкт сервера [34]
list_tools()	server.py	Опис доступних агенту інструментів
call_tool()	server.py	Валідація, маршрутизація та виконання виклику
storage.py	storage.py	Збереження результатів у JSON і CSV

Запуск сервера відбувається через функцію **main()**, вона зчитує аргументи командного рядка та передає необов’язковий параметр *--user-agent* до функції **serve(...)**. Сервер працює через *stdio*, тому не потребує відкриття мережевого порту.

Такий спосіб запуску є зручним для локальних MCP-клієнтів та агентних середовищ.

У функції **serve()** створюється об’єкт **Server("mcp-work-ua"),** реєструються обробники **list_tools() і call_tool()**, після чого сервер запускається в контексті **stdio_server().** Параметр *raise_exceptions=False* дозволяє не завершувати процес аварійно в разі помилки, а повертати її агенту у стандартизованому вигляді. **Логіка взаємодії агента із сервером** складається з наступних кроків:

1) агент запускає MCP-сервер як дочірній процес через stdio;

2) агент викликає **list_tools()** для отримання описів інструментів;

3) на основі отриманих JSON-Schema агент формує виклик потрібного інструмента з аргументами;

4) виклик передається в **call_tool()**, де відбувається валідація аргументів за допомогою Pydantic [31];

5) валідовані параметри спрямовуються до відповідної функції scraper.py, djinni.py або storage.py;

6) результат (структуровані дані) серіалізується в JSON і повертається агенту як TextContent.

Також важливим компонентом є оголошення та виконання інструментів, які реалізує функція **list_tools(),** вона повертає список об’єктів **Tool**, які описують доступні агенту дії. Кожен інструмент має унікальне ім’я, опис і формальну JSON-схему параметрів, сформовану через **model_json_schema()** на основі моделей Pydantic [31]. Інструменти вже були наведені в **табл 3.5.** Після вибору інструмента агент передає аргументи до **call_tool(name, arguments)**. Для кожного інструмента використовується відповідна Pydantic-модель: *SearchVacancies, VacancyDetails, DjinniSearch, DjinniJobRef або SavePayload*. Валідація перевіряє типи, обов’язкові поля, числові межі та відповідність формату файлу його розширенню. Приклад оголошення інструмента:

```
Tool(
   name="s_work_ua_search_vacancies",
   description="Шукає вакансії на Work.ua...",
   inputSchema=SearchVacancies.model_json_schema(),
)
```

Після валідації call_tool() перенаправляє виклик виклик до відповідної функції. Де він оброблюється і повертається агенту як TextContent. Це забезпечує Експорт реалізовано в модулі storage.py. Він відповідає за вибір базового каталогу, перевірку файлових шляхів і запис результатів у форматах JSON та CSV. Повний текст підсистеми експорту винесено в **ДОДАТОК Г**. У системі передбачено **два режими експорту**: автоматичний та явний.

Таблиця 3.14 – Режими експорту результатів

Ознака	Автоматичний експорт	Явний експорт
Запуск	Через export_file	Через s_save_payload
Момент виконання	Під час пошуку або отримання деталей	Окремий сервісний крок
Формати	JSON або CSV за розширенням	JSON або CSV за параметром format
Результат	Основний payload + exported_to	ok, format, exported_to

Базовий каталог експорту визначається функцією **get_export_base():**

– якщо задано змінну середовища *mcp_jobs_export_dir*, використовується її значення;

– якщо змінна відсутня, результати зберігаються в каталозі mcp_scrape_exports у поточній робочій директорії;

– каталог створюється автоматично за потреби;

– функція **_resolved_export_path(relative)** виконує нормалізацію та перевірку;

– забороняє порожні значення, сегменти .. та символи, що не входять до дозволеного набору (латиниця, цифри, _, -, ., /);

– переконується, що результуючий шлях знаходиться в межах базового каталогу експорту.

Важливою частиною експорту є перевірка шляху за допомогою функції **_resolved_export_path(relative)** виконує нормалізацію та перевірку:

– забороняє порожні значення, сегменти .. та символи, що не входять до дозволеного набору (латиниця, цифри, _, -, ., /);

– переконується, що результуючий шлях знаходиться в межах базового каталогу експорту.

Для збереження у форматі JSON використовується функція **write_payload_json(export_file, payload)**, яка серіалізує дані з параметрами *ensure_ascii=False ma indent=2*. У разі автоматичного експорту застосовується функція **attach_export_path(...)**: вона додає до відповіді поле *exported_to*, записує файл на диск і повертає оновлений payload агенту.

Для формату CSV додатково виконується нормалізація через **normalize_to_csv_rows(data).** Якщо *payload* містить *vacancies* або *jobs*, кожна вакансія перетворюється на окремий рядок, а частина метаданих дублюється у службових колонках. Якщо передано одиничний detail-об’єкт, формується один рядок CSV.

Такий підхід дозволяє зберігати як спискові результати, так і детальні об’єкти у форматі, придатному для відкриття в табличних редакторах. Вкладені структури не втрачаються, а записуються у комірки як JSON-рядки.

Отже, інтеграція з агентом реалізована як MCP-шар, що забезпечує запуск сервера у stdio-режимі, публікацію інструментів, валідацію параметрів через Pydantic [31], маршрутизацію викликів до прикладних функцій і повернення результатів у вигляді TextContent відповідно до моделі MCP [34]. Механізм експорту доповнює цю взаємодію можливістю безпечного збереження результатів у JSON і CSV. Перевірка шляхів, підтримка змінної MCP_JOBS_EXPORT_DIR, автоматичне додавання поля *exported_to* та окремий інструмент *s_save_payload*

роблять сервер не лише засобом отримання вакансій, а й інструментом накопичення та подальшого використання результаті

Висновки до третього розділу

У третьому розділі було розглянуто програмну реалізацію інформаційної системи на основі MCP-інструменту для вебскрапінгу вакансій. У межах розділу обґрунтовано вибір середовища розробки, основних бібліотек і технологій, а також описано реалізацію MCP-сервера, модуля збору та структуризації вакансій, інтеграції з агентом і механізму експорту результатів.

На етапі вибору технологічної бази було визначено, що для реалізації системи доцільно використовувати мову програмування Python. Такий вибір пояснюється наявністю розвиненої екосистеми бібліотек для мережевої взаємодії, обробки HTML-документів, валідації даних, асинхронного виконання запитів і тестування програмних компонентів. Використання бібліотек httpx, BeautifulSoup, Pydantic, mcp, pytest, pytest-asyncio та інших засобів дозволило сформувати надійну технологічну основу для реалізації MCP-сумісного серверного застосунку.

У процесі реалізації серверної частини було створено MCP-сервер, який забезпечує стандартизовану взаємодію між LLM-клієнтом і прикладною логікою системи. Сервер відповідає за оголошення доступних інструментів, приймання викликів, перевірку вхідних параметрів, маршрутизацію запитів і формування уніфікованих відповідей. Застосування моделей Pydantic дало змогу формалізувати параметри кожного інструмента та відхиляти некоректні запити ще до виконання основної логіки обробки.

Окремо було реалізовано модуль збору та структуризації вакансій. Для цього створено два спеціалізовані модулі: scraper.py для роботи з Work.ua та djinni.py для роботи з Djinni. Такий поділ дозволив врахувати особливості HTML-структури кожного джерела та ізолювати логіку їх обробки. Реалізовані функції забезпечують формування пошукових URL, асинхронне завантаження сторінок, парсинг списків

вакансій і детальних сторінок, нормалізацію отриманих даних та збагачення результатів повним описом вакансій.

У результаті роботи модуля збору система формує структуровані об’єкти, що містять основні поля вакансій: назву посади, компанію, зарплату, локацію, опис і посилання на сторінку. Додатково формується блок метаданих, який зберігає інформацію про джерело даних, параметри пошуку, кількість знайдених результатів і режим отримання деталей. Це підвищує прозорість роботи системи та забезпечує можливість відтворення виконаних запитів.

У підрозділі, присвяченому інтеграції з агентом, було показано, що MCP-сервер працює як локальний процес через stdio-транспорт і надає агенту набір формально описаних інструментів. Результати виконання повертаються у вигляді TextContent із JSON-вмістом, що забезпечує зручність подальшої обробки як для агента, так і для користувача. Такий підхід дозволяє використовувати розроблену систему не як окремий скрипт, а як повноцінний інструмент у складі MCP-сумісного середовища.

Також у межах розділу було описано механізм експорту результатів. Реалізовано два режими збереження: автоматичний експорт через параметр *export_file* та явний експорт через інструмент *s_save_payload*. Підсистема експорту підтримує формати JSON і CSV, виконує перевірку файлових шляхів, використовує змінну середовища MCP_JOBS_EXPORT_DIR для визначення каталогу збереження та додає до результату службове поле *exported_to*. Це робить систему придатною не лише для оперативного отримання вакансій, а й для подальшого аналізу, архівування та використання результатів в інших програмних засобах.

Отже, у третьому розділі було послідовно розглянуто практичну реалізацію всіх основних компонентів інформаційної системи: від вибору технологічної бази до створення MCP-сервера, scraper-модулів, інтеграції з агентом і механізму експорту. Запропонована реалізація є модульною, розширюваною та придатною для подальшого розвитку.

4 ТЕСТУВАННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ РОБОТИ MCP-ІНСТРУМЕНТУ

4.1 Методика тестування працездатності системи

Метою тестування розробленої інформаційної системи є перевірка коректності роботи MCP-сервера, стабільності окремих програмних модулів і правильності формування результатів, що повертаються агенту або зберігаються у файл. Так як система поєднує декілька функціональних частин – валідацію параметрів, HTTP-завантаження сторінок, HTML-парсинг, маршрутизацію викликів MCP-інструментів та експорт даних, – її працездатність доцільно перевіряти на кількох рівнях.

В межах роботи було застосовано два основні підходи для тестування, це-модульне та функціональне. Модульне тестування було спрямоване на перевірку окремих функцій та компонентів системи, таких, як парсери, моделей валідації та підсистеми експорту. Функціональне тестування в свою чергу дає нам змогу перевірити повні сценарії роботи MCP-інструментів: від отримання запиту від агента до формування структурованої відповіді або збереження результатів у файл.

На модульному рівні основна увага була приділена функціям, від яких залежить правильність обробки вакансій. У проєкті автоматизоване тестування зосереджене у таких файлах, як **test_scraper.py, test_djinni.py, test_storage.py і test_save_payload.py**, що виконуються засобами pytest [35]. Такий підхід дозволяє перевіряти логіку системи без постійної залежності від зовнішнього мережевого середовища. Для цього в тестах можуть використовуватися підготовлені HTML-фрагменти, а для перевірки експорту – тимчасові каталоги.

Перейдемо до модуля тестування для Work.ua, він призначений для того щоб тестувати функції побудови URL пошуку, нормалізації посилань на вакансії, визначення зарплатного рядка, розбору спискової сторінки та отримання даних із детальної сторінки вакансії. Зокрема, перевіряються функції *build_search_url, resolve_vacancy_url, _looks_like_salary_text, parse_vacancy_list,*

parse_vacancy_detail і merge_work_list_with_detail. Очікуваним результатом є формування структурованого об'єкта з коректно заповненими полями *job_id, title, company, location, salary, url ma description.*

Аналогічно перевіряється модуль роботи з Djinni. У цьому випадку тестування охоплює побудову URL з фільтрами, формування знімка параметрів пошуку, нормалізацію посилань на вакансії, обробку спискової сторінки та сторінки окремої вакансії. Для цього перевіряються функції *build_djinni_search_url, djinni_filters_snapshot, resolve_djinni_job_ref, parse_djinni_list, parse_djinni_job_page і merge_djinni_list_with_detail*. Успішним результатом є отримання структурованих даних про вакансію, зокрема ідентифікатора, назви посади, компанії, зарплати, службового контексту, метаданих пошуку та фінального URL.

Окремо варто виділити такий напрям тестування, як перевірка валідації параметрів і роботи підсистеми експорту. Модель SavePayload перевіряється на наявність обов'язкових полів *format, export_file і data*, а також на відповідність розширення файлу обраному формату. У модулі **storage.py** перевіряються функції *get_export_base, write_payload_json, write_payload_csv, normalize_to_csv_rows і attach_export_path*. Це дозволяє переконатися, що система коректно зберігає результати у форматах JSON і CSV.

Функціональне тестування має перевіряти повний сценарій роботи MCP-інструменту. У такому сценарії агент викликає інструмент, сервер виконує валідацію параметрів через Pydantic [31], передає керування відповідному модулю парсингу, формує підсумковий payload, за потреби виконує експорт і повертає відповідь у форматі TextContent відповідно до MCP [34]. Для сценаріїв пошуку очікується JSON-відповідь із блоком meta та масивом vacancies або jobs. Для сценаріїв деталізації очікується повернення структурованого опису однієї вакансії, а для сценаріїв з параметром export_file – додаткове створення файлу на диску.

Важливою частиною методики є перевірка обробки помилок. Оскільки система працює із зовнішніми вебресурсами та приймає параметри від MCP-

клієнта, вона повинна коректно реагувати як на помилки користувацького введення, так і на мережеві збої. Некоректні або неповні аргументи мають призводити до помилки INVALID_PARAMS, тоді як помилки HTTP-запитів або інші мережеві винятки повинні перетворюватися на INTERNAL_ERROR. Такий підхід забезпечує передбачувану реакцію системи для агента та спрощує виявлення причин збоїв під час експлуатації [32], [34].

Основні об’єкти тестування та очікувані результати перевірки наведено в **табл. 4.1**.

Таблиця 4.1 – Об’єкти тестування та очікувані результати

Об’єкт тестування	Що перевіряється	Очікуваний результат
Парсер Work.ua	Побудова URL, розбір списку вакансій і детальної сторінки	Отримано коректні структуровані поля вакансії
Парсер Djinni	Формування URL з фільтрами, розбір списку та сторінки вакансії	Отримано коректні поля вакансії та знімок параметрів пошуку
Валідація параметрів	Обов’язковість полів і відповідність розширення файлу формату	Невалідні аргументи відхиляються до виконання основної логіки
Експорт JSON/CSV	Запис файлів, нормалізація CSV-рядків.	Результат збережено у потрібному форматі в межах каталогу експорту
Обробка помилок	Реакція на помилки валідації, файлові та мережеві винятки	Повертається стандартизована MCP-помилка з відповідним кодом

Як видно з **табл. 4.1**, тестування охоплює як окремі програмні компоненти, так і сценарії їхньої взаємодії. Це дозволяє перевірити не лише правильність роботи парсерів, а й стабільність усієї системи під час приймання запитів, обробки даних і формування результатів.

Отже, методика тестування працездатності системи передбачає поєднання модульної перевірки окремих функцій із функціональною перевіркою повного сценарію роботи MCP-інструменту. Такий підхід є доцільним для розробленої системи, оскільки дозволяє окремо підтвердити коректність парсингу, валідації, експорту та обробки помилок, а також оцінити цілісну роботу MCP-сервера як інструменту для агента.

4.2 Перевірка коректності взаємодії агента з MCP-сервером

Перевірка коректності взаємодії агента з MCP-сервером необхідна для того, щоб підтвердити, що розроблена система може працювати як повноцінний інструмент у середовищі Model Context Protocol. На відміну від модульного тестування, яке перевіряє окремі функції парсингу, валідації та експорту, цей підрозділ оснований на увазі, яка приділяється перевірці повного інтеграційного сценарію. Від запуску серверного процесу, отриманню агентом списку доступних інструментів, передаванню параметрів, виконанню прикладної логіки та поверненню структурованої відповіді [34].

У розробленому проєкті взаємодія агента із сервером реалізована через локальний процес **mcp-server-work**, який працює як stdio-MCP-сервер. Точка входу визначена у файлі **pyproject.toml**, а запуск основної логіки виконується через функцію *main()* пакета *mcp_server_work*. Далі функція *main()* передає керування асинхронній функції *serve(...)*, яка створює сервер, реєструє інструменти та переводить процес у режим очікування MCP-повідомлень.

Першим етапом перевірки буде підтвердження коректного запуску MCP-сервера. На цьому етапі необхідно переконатися, що агент або MCP-клієнт може запустити локальний процес без аварійного завершення, а сервер залишається активним і готовим приймати запити. Також необхідно тут перевірити, що у *server.py* сервер реєструє п’ять MCP-інструментів: *s_work_ua_search_vacancies, s_work_ua_vacancy_details, s_djinni_search_jobs, s_djinni_job_details і s_save_payload*. Отже, очікуваний результат має показати, що сервер запускається

без аварійного перезапуску або відключення, та агент отримує повний перелік інструментів разом із їхніми назвами, описами та схемами параметрів *inputSchema*, сформованими на основі моделей Pydantic [31]. Завдяки цьому агент зможе визначити, які саме параметри потрібно передати для виконання конкретного сценарію. На **рис 4.1** можна побачити запущений MCP-сервер. Також, для прикладу, було наведено на окремий інструмент **s_djini_job_details** для виводу його опису.

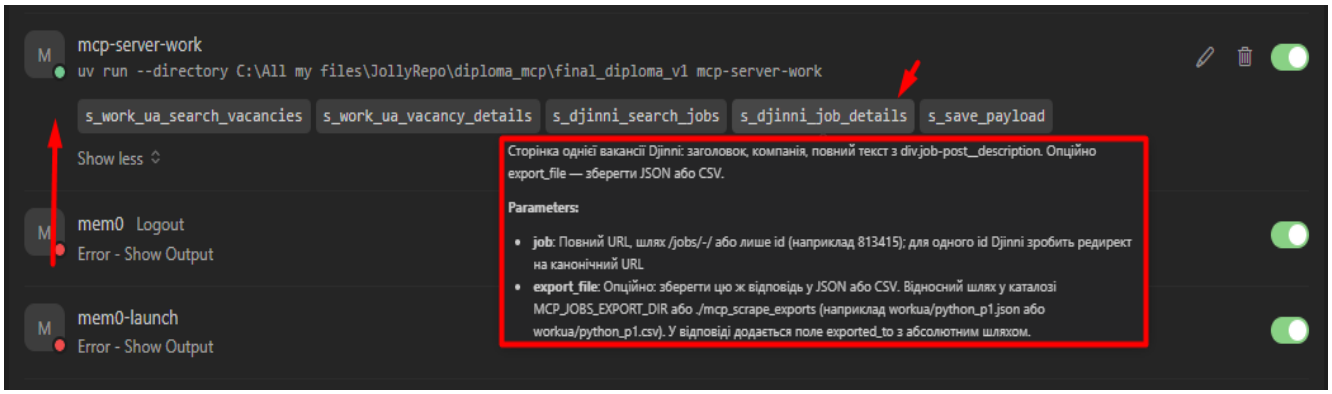

Рисунок 4.1 – Підключення MCP-сервера **mcp-server-work** до агента

Другим важливим етапом є перевірка виклику інструментів через **call_tool().** Під час цієї перевірки агент має передавати серверу ім'я інструменту та об'єкт *arguments*, після чого сервер виконує валідацію параметрів, маршрутизує запит до відповідного модуля та формує результат. Для інструментів пошуку очікується відповідь із блоком *meta* та списком вакансій *vacancies або jobs*. Для інструментів деталізації очікується структурований опис однієї вакансії, а для інструмента **s_save_payload** – службова відповідь із даними про формат і шлях збереження.

Для прикладу, під час виклику інструмента **s_work_ua_search_vacancies** агент передає ключове слово, наприклад, номер сторінки або максимальну кількість результатів, і сервер повинен перевірити ці параметри через модель *SearchVacancs*, сформувати URL пошуку, отримати HTML-сторінку Work.ua, виконати парсинг і повернути результат у форматі JSON. Такий сценарій підтверджує, що агент може не лише бачити інструмент, а й успішно використовувати його для отримання даних. Для прикладу, ми візьмемо такий

запит: *“Використай інстурмент s_work_ua_search_vacancies з MCP mcp-server-work та збери інформацію про перші 3 вакансії для UI/UX в форматі JSON”*. Таким чином, ми зможемо протестувати взаємодію агента з сервером та чи правильно він використає і поверне саме 3 вакансії, приклад використання на **рис. 4.2.**

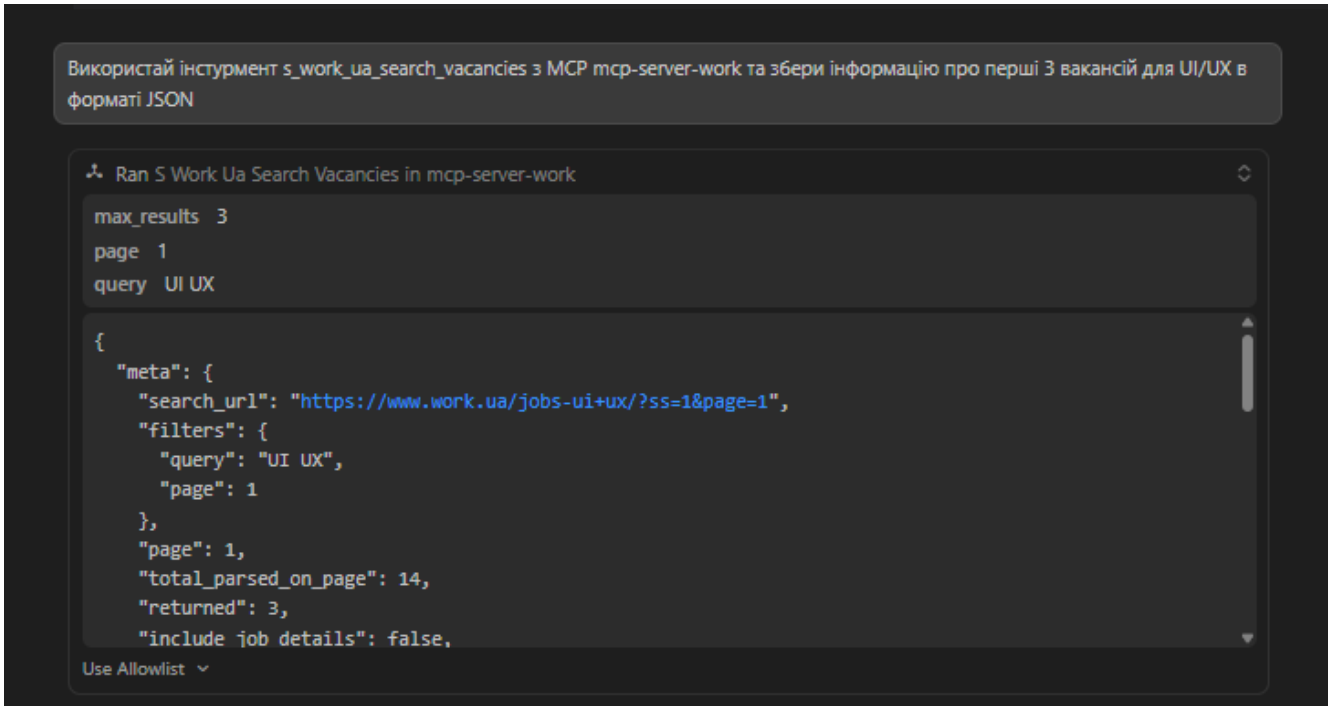

Рисунок 4.2 – Використання інстурменту s_work_ua_search_vacancies

Як видно з **рис. 4.2**, агент передав серверу три основні параметри: *page, max_results та query*. У результаті сервер сформував пошуковий запит до Work.ua, отримав HTML-вміст сторінки, виконав парсинг результатів і повернув відповідь у форматі JSON **ДОДАТОК Д**. Як видно, з отриманої відповіді усі параметри були оброблені коректно: у блоці meta зазначено пошуковий URL *https://www.work.ua/jobs-ui+ux/?ss=1&page=1*, номер сторінки *page: 1*, загальну кількість розпізнаних вакансій на сторінці *total_parsed_on_page: 14* та кількість повернутих результатів *returned: 3*. Це відповідає заданому обмеженню *max_results = 3*, тобто система повернула саме три перші вакансії з результатів пошуку.

Для перевірки правильності отриманих результатів було перевірено зі сторінкою пошуку Work.ua, наведеною на **рис. 4.3.** Порівняння показує, що назви вакансій, порядок їх відображення, локації та зарплатні поля відповідають даним,

які відображаються на сайті. Зокрема, у JSON-відповіді першою повернуто вакансію UX/UI Designer з локацією Київ, другою – Game UI/UX Designer з локацією Дистанційно та зарплатою 65 000 – 71 000 грн, третьою – UX/UI Designer, військовослужбовець з локацією Вся Україна та зарплатою 25 000 – 125 000 грн.

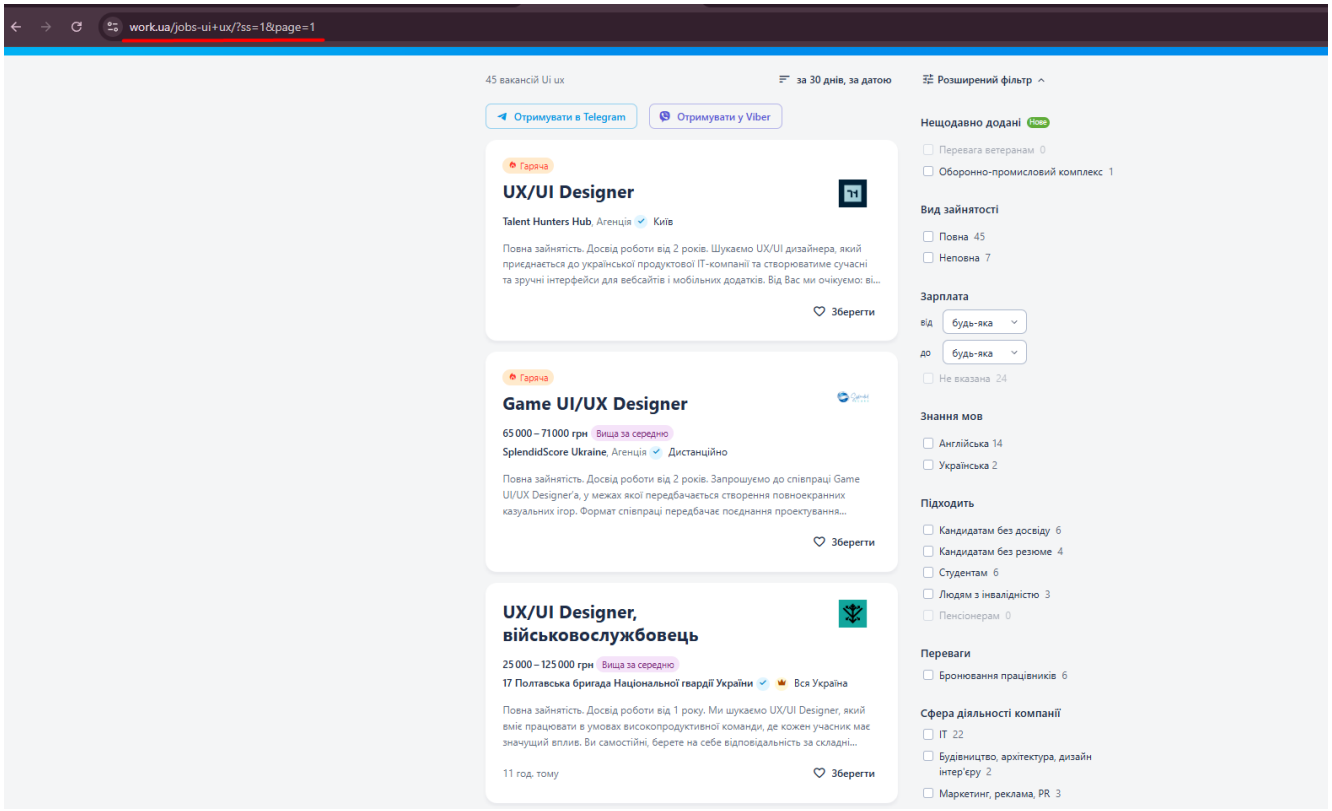

Рисунок 4.3 – Сторінка пошуку на сайті Work.ua

Окремо варто зазначити, що для вакансії, у якій на сторінці не вказану зарплатню, відповідне поле залишається порожнім, що свідчить про коректну роботу парсера, і те, що він не підставляє некоректні або неперевірені дані. Таким чином, приклад на **рис. 4.2** і порівняння зі сторінкою Work.ua на **рис. 4.3** підтверджують працездатність інструмента **s_work_ua_search_vacancies**. Система коректно приймає параметри від агента, формує пошуковий URL, обмежує кількість результатів, повертає службові метадані та формує структурований список вакансій, придатний для подальшого аналізу або експорту.

Аналогічно необхідно перевірити роботу інструмента **s_djinni_search_jobs**, і оскільки для нього агент може передати ключові слова та додаткові фільтри пошуку, цей сценарій є важливим для перевірки правильності передавання

складніших параметрів між агентом і MCP-сервером. Для прикладу, ми візьмемо такий запит: *“Використай інстурмент s_djinni_search_jobs з MCP mcp-server-work та збери інформацію про перші 3 вакансії для UI/UX в форматі JSON з такими фільтрами як регіон роботи Україна та досвід від 0-2 років”*. Таким чином, ми зможемо протестувати коректну роботу програми та фільтрів, приклад роботи на **рис.4.4.**

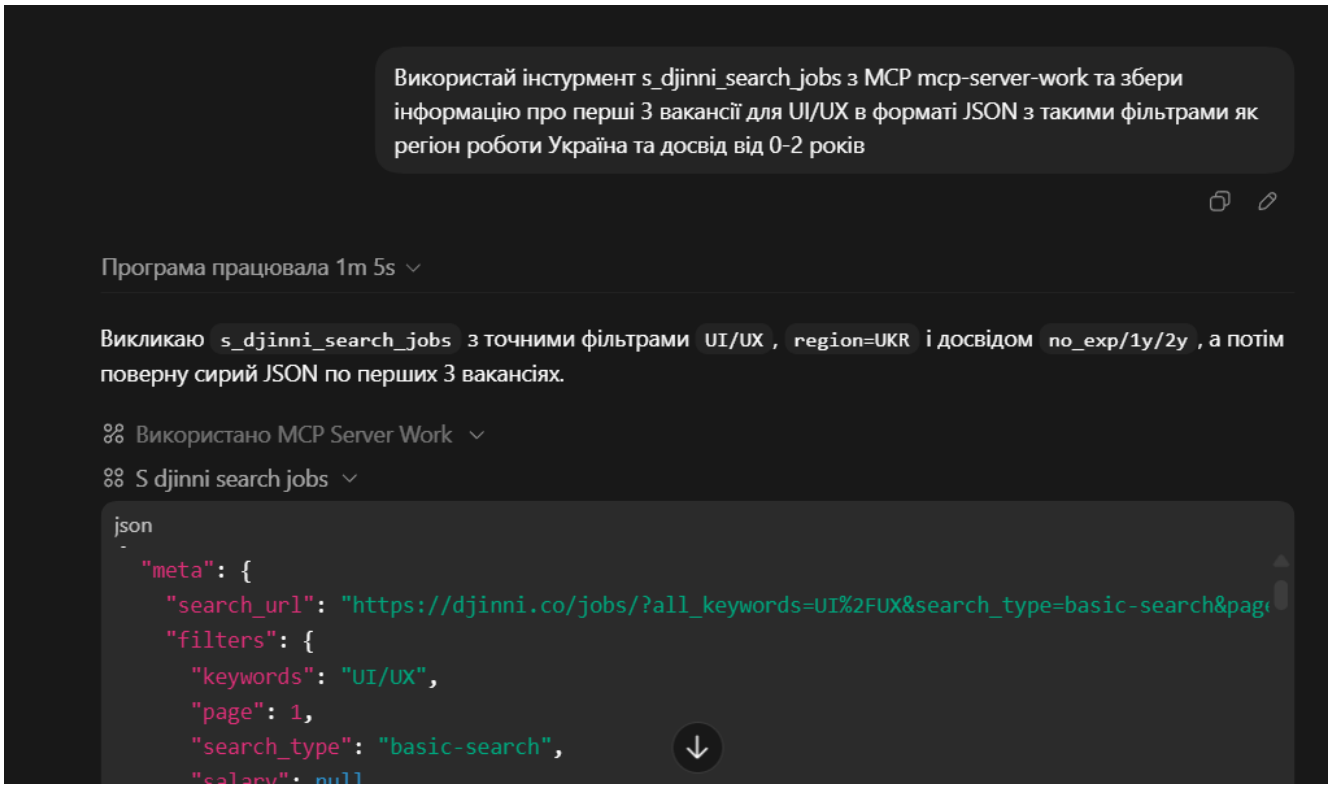

Рисунок 4.4 – Підключення MCP-сервера **mcp-server-work** до агента

Коректність обробки параметрів підтверджується блоком *meta*, наведеним на **рис. 4.5**. У ньому збережено сформований пошуковий URL, номер сторінки *page: 1*, кількість розпізнаних вакансій на сторінці *total_parsed_on_page: 15* та кількість повернутих результатів *returned: 3*. Значення *returned: 3* відповідає обмеженню на кількість результатів, заданому під час виклику інструмента. Окремо важливо зазначити, що в блоці *filters* коректно збережено всі застосовані параметри пошуку. Зокрема, поле *keywords* має значення UI/UX, поле *region* містить значення *ukr*, а поле *exp_level* містить список *no_exp, 1y, 2y*. Це підтверджує, що MCP-сервер

правильно передав складені фільтри до функції побудови URL і зберіг їх у метаданих відповіді


```
"meta": {
    "search_url": "https://djinni.co/jobs/?all_keywords=UI%2FUX&search_type=basic-search&page=1&primary_keyword
    "filters": {
      "keywords": "UI/UX",
      "page": 1,
      "search_type": "basic-search",
      "salary": null,
      "exp_level": [
        "no_exp",
        "1y",
        "2y"
      ],
      "employment": null,
      "company_type": null,
      "english_level": null,
      "domain": null,
      "region": "UKR",
      "editorial": null,
      "hide_contacted": null,
      "show_stale": null,
      "company": null,
      "primary_keyword": "ui_ux",
      "extra_query": {}
    },
    "page": 1,
    "total_parsed_on_page": 15,
    "returned": 3,
    "include_job_details": false,
    "source": "djinni.co"
  },
```


Рисунок 4.5 – Мета дані пошуку

Для перевірки коректності отриманих даних, їх було порівняно зі сторінкою пошуку Djinni наведеною на **рис. 4.6**. На сторінці видно, що застосовано фільтри UI/UX, Ukraine, No experience, 1 year і 2 years, тобто вони відповідають параметрам, які збережені у JSON-відповіді. Також на сторінці відображаються вакансії UI/UX Designer AI operator, Game UI/UX Designer та Middle UX/UI designer, які збігаються з першими трьома елементами масиву jobs у відповіді сервера.

Порівняння показує, що система коректно зчитала основні поля вакансій: ідентифікатор *job_id*, назву посади *title*, компанію *company*, службовий рядок *meta_line*, відносний шлях *path* і повне посилання url. Наприклад, для вакансії UI/UX Designer AI operator у відповіді зазначено компанію Join.To.IT, а в полі *meta_line* збережено додаткову інформацію про формат роботи, регіон, досвід,

рівень англійської та домен. Аналогічно для вакансії Game UI/UX Designer коректно визначено компанію 24hit.games, а для Middle UX/UI designer – компанію OnFire.

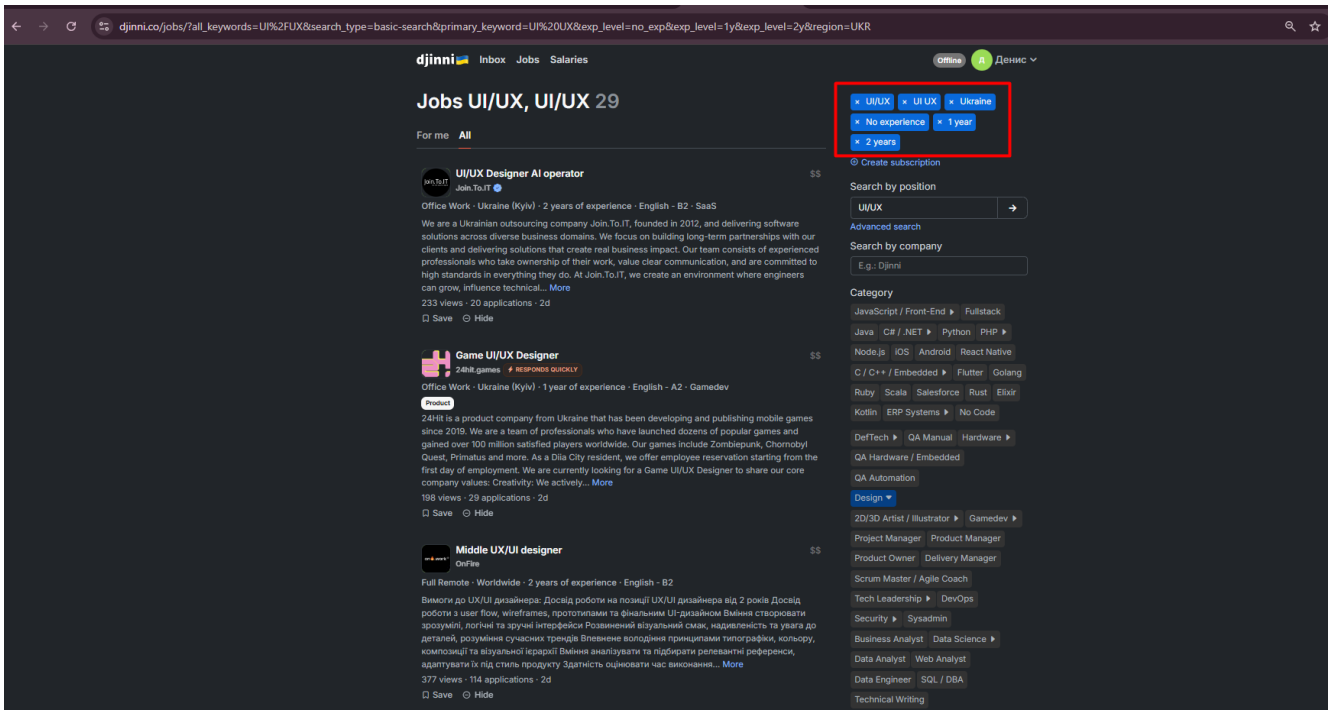

Рисунок 4.6 – Сторінка пошуку на сайті Djinni

Таким чином, приклад використання інструмента s_djinni_search_jobs підтверджує правильність взаємодії агента з MCP-сервером під час роботи зі складнішими фільтрами пошуку. Система коректно приймає параметри від агента, формує URL з кількома значеннями фільтра досвіду, повертає службові метадані, обмежує кількість результатів відповідно до запиту та формує структурований список вакансій, який збігається з даними, представленими на сторінці Djinni.

Окремий виклик інструментів для отримання детальної інформації про вакансію в межах цієї перевірки не є обов'язковим. Інструменти **s_work_ua_vacancy_details і s_djinni_job_details** уже були враховані в загальній логіці роботи системи, оскільки під час виклику основних інструментів пошуку може використовуватися параметр **include_job_details**. Якщо цей параметр увімкнено, сервер автоматично завантажує сторінки окремих вакансій і доповнює результати повним описом. Тому для перевірки взаємодії агента з MCP-сервером

достатньо показати, що основні інструменти пошуку коректно приймають параметри, виконують запит, повертають структуровані результати та за потреби можуть бути доповнені детальною інформацією.

Не менш важливою частиною перевірки є контроль коректності параметрів, які агент передає серверу. Моделі *SearchVacancies*, *VacancyDetails*, *DjinniSearch*, *DjinniJobRef і SavePayload* задають обов'язкові поля, типи даних і допустимі значення. Якщо агент передає неповні або некоректні аргументи, сервер не переходить до виконання основної логіки, а повертає помилку *INVALID_PARAMS*. Це підтверджує, що валідація параметрів виконується до запуску парсингу або експорту. Для демонстрації цього сценарію можна розглянути **рис. 4.7** із прикладом невалідного запиту, наприклад із неправильним форматом файлу або відсутнім обов'язковим параметром.

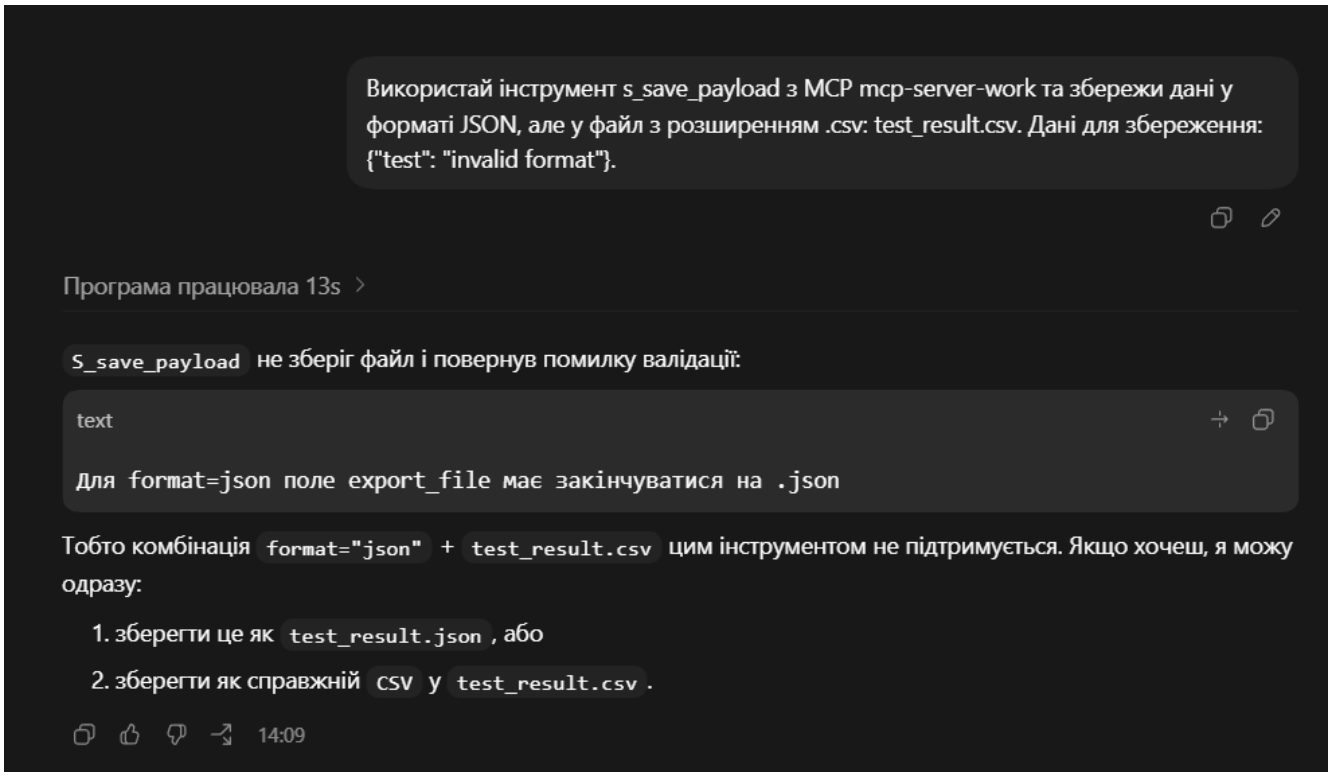

Рисунок 4.7 – Приклад обробки некоректних параметрів MCP-інструмента

Для демонстрації коректного прикладу був використаний той самий запит, але з узгодженими параметрами: формат збереження було встановлено як json, а назву файлу вказано з розширенням .json. У цьому випадку параметри відповідають

вимогам моделі SavePayload, тому сервер успішно проходить етап валідації та переходить до виконання логіки збереження даних **рис. 4.8.**

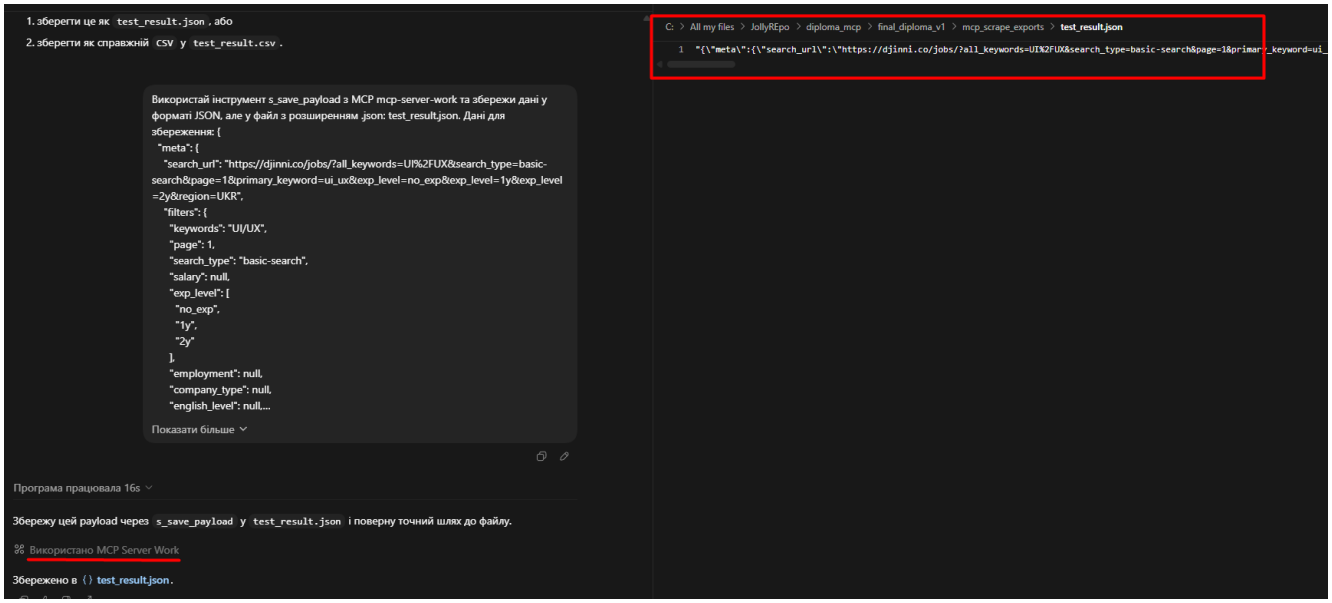

Рисунок 4.8 – Приклад обробки коректних параметрів MCP-інструмента

Отже, перевірка коректності взаємодії агента з MCP-сервером підтверджує, що розроблена інформаційна система здатна стабільно працювати в MCP-сумісному середовищі. Ключовими ознаками успішної інтеграції є коректний запуск stdio-процесу, доступність інструментів для агента, правильна валідація аргументів, формування структурованих JSON-відповідей, підтримка експорту результатів і передбачувана обробка помилок.

4.3 Тестування точності, повноти та продуктивності збору даних

Після перевірки взаємодії агента з MCP-сервером доцільно оцінити якість безпосереднього збору даних із зовнішніх джерел. У межах цього підрозділу перевіряються точність, повнота та продуктивність роботи інструментів, що відповідають за отримання вакансій із Work.ua та Djinni. Для оцінювання коректності результати, отримані через MCP-сервер, порівнювалися з фактичними даними на сторінках пошуку Work.ua та Djinni, наведеними відповідно на **рис. 4.3** і **рис. 4.6**.

Під точністю збору даних мається відповідність JSON-відповіді фактичній інформації представлені на сторінці. Для Work.ua аналізувалися назва вакансії, локація, зарплата за наявності, URL та службові метадані. Для Djinni додатково перевірялися назва компанії, службовий рядок *meta_line*, відносний шлях *path* і застосовані фільтри пошуку. Отримані JSON-результати для Work.ua винесено в ДОДАТОК Д, а результати для Djinni – у ДОДАТОК Е.

Під час перевірки інструмента **s_work_ua_search_vacancies** було виконано пошук за запитом UI UX із параметрами *page = 1 ma max_results = 3.* Система повернула три вакансії, що відповідає заданому обмеженню. Порівняння з фактичною сторінкою Work.ua на **рис. 4.3** показало, що назви вакансій, порядок їх відображення, локації та зарплатні поля були зчитані коректно. Якщо зарплата на сторінці не була вказана, у JSON-відповіді поле salary залишалося порожнім, що свідчить про відсутність підстановки неперевірених значень.

Аналогічну перевірку було виконано для інструмента **s_djinni_search_jobs**. Для тестування використовувався запит UI/UX із фільтрами регіону Україна та рівнів досвіду *no_exp, 1y, 2y*. У відповіді сервера було сформовано правильний пошуковий URL, а в блоці *filters* збережено всі застосовані параметри. Порівняння з результатами на сторінці Djinni, наведеній на **рис. 4.6**, показало, що перші три вакансії у JSON-відповіді збігаються з результатами на сайті. Також коректно були визначені назви вакансій, компанії, службові рядки *meta_line*, відносні шляхи *path* та повні посилання url.

Повнота збору даних оцінювалася за кількістю вакансій, які система змогла розпізнати на сторінці, та за наявністю ключових полів у сформованих об’єктах. Для Work.ua у блоці meta було вказано *total_parsed_on_page = 14*, а для Djinni – *total_parsed_on_page = 15*. Це означає, що система спочатку розпізнала всі доступні елементи на сторінці, а вже потім обмежила кількість повернутих результатів відповідно до параметра max_results. Основні критерії перевірки якості збору даних наведено в **табл. 4.2.**

Таблиця 4.2 – Критерії оцінювання якості збору даних

Критерій	Спосіб перевірки	Очікуваний результат
Точність назв вакансій	Порівняння поля title з даними на сайті	Назви у JSON відповідають назвам на сторінці
Коректність локації або опису	Перевірка location для Work.ua та meta_line для Djinni	Контекст вакансії передано без зміни змісту
Коректність зарплати	Аналіз поля salary	Зарплата заповнюється лише за її наявності на сайті
Повнота результатів	Аналіз total_parsed_on_page і returned	Система розпізнає вакансії та повертає задану кількість
Коректність посилань	Перевірка url і path	Посилання ведуть на відповідні сторінки вакансій
Відтворюваність пошуку	Аналіз блоку meta.filters	У відповіді збережено параметри виконаного запиту

Як видно з **табл. 4.2**, перевірка охоплює не лише факт отримання результатів, а й відповідність основних полів фактичним даним на сайтах. Це важливо для подальшого аналізу вакансій, оскільки помилки в назві, локації, зарплаті або посиланні можуть вплинути на достовірність результатів.

Окремо було оцінено продуктивність збору даних для двох режимів роботи: без деталізації вакансій та з отриманням повного опису. У режимі **include_job_details = false** система виконує один основний HTTP-запит до сторінки пошуку, після чого здійснює парсинг отриманого HTML-вмісту та формує структуровану JSON-відповідь. У цьому режимі середній час виконання запиту для Djinni становив приблизно **32 с**, а для Work.ua – близько **13 с**.

У режимі **include_job_details = true** час виконання очікувано збільшується, оскільки для кожної вакансії система додатково завантажує окрему сторінку та виконує її парсинг. У цьому випадку середній час виконання запиту для Djinni становив приблизно **1 хв 15 с**, а для Work.ua – близько **57 с**. Отримані результати показують, що режим без деталізації є доцільним для швидкого отримання списку вакансій, тоді як режим із деталізацією варто використовувати тоді, коли необхідно отримати повний опис кожної позиції. Чинники впливу в **табл. 4.3.**

Таблиця 4.3 – Чинники впливу на продуктивність збору даних

Чинник	Вплив на роботу системи
max_results	Більша кількість результатів збільшує час обробки
include_job_details	Увімкнення деталізації збільшує кількість HTTP-запитів
Швидкість відповіді сайту	Зовнішній ресурс може уповільнювати виконання запиту
Експорт у файл	Додає незначний час на запис результатів
Обсяг HTML-сторінки	Більший обсяг сторінки потребує більше часу на парсинг

Отже, тестування точності, повноти та продуктивності збору даних показало, що система коректно отримує вакансії з Work.ua та Djinni, зберігає параметри пошуку в метаданих, повертає задану кількість результатів і не підставляє неперевірені значення у випадках, коли певне поле відсутнє на сторінці. Порівняння з фактичними сторінками пошуку, наведеними на **рис. 4.3** і **рис. 4.6**, а також результати у **ДОДАТКУ Д** і **ДОДАТКУ Е** підтверджують придатність розробленого MCP-інструменту для практичного збору та первинної структуризації вакансій.

4.4 Аналіз отриманих результатів та обмежень системи

Отримані результати показали, що розроблена інформаційна система загалом коректно виконує функції MCP-інструмента для збору вакансій із Work.ua та Djinni. Було також підтверджено працездатність основних компонентів, таких як модели валідації параметрів, моделів парсингу, механізм експорту результатів і серверного шару, який забезпечує взаємодії агента з інструментами через Model Context Protocol [31], [34]. Додатково, під час перевірки, було встановлено, що інформаційна система коректно формує пошукові URL, отримує HTML-вміст сторінок, виділяє основні поля вакансії і повертає результат у структуризованому форматі. Для Work.ua система обробляє назву вакансії, локацію, зарплату за наявності та посилання на сторінку. Для Djinni додатково зберігаються компанія, службовий рядок *meta_line*, шлях вакансії та набір застосованих фільтрів. Це

підтверджує, що реалізований механізм парсингу забезпечує достатню точність для первинної структуризації даних [33].

До позитивних результатів можна віднести коректну інтеграцію функціональних модулів у межах MCP-сервера. Агент отримує перелік доступних інструментів, передає параметри, а сервер виконує їх валідацію через Pydantic, маршрутизує запит до відповідного модуля та повертає відповідь у форматі TextContent. Завдяки централізованій обробці помилок некоректні параметри повертаються як *INVALID_PARAMS*, а мережеві або внутрішні збої – як *INTERNAL_ERROR*. Окремо слід відзначити реалізацію експорту результатів у формати JSON і CSV. Це дозволяє використовувати систему не лише для перегляду результатів в агенті, а й для подальшого аналізу, збереження або перенесення даних в інші програмні засоби.

Разом із цим. Система, звичайно, має певні обмеження, і основне обмеження-це залежність від HTML-структури зовнішніх сайтів. Оскільки Work.ua та Djinni не обробляються через офіційні API, зміна верстки, CSS-класів або структури карток вакансій може потребувати оновлення селекторів і правил парсингу [32], [33]. Ще одним обмеженням є залежність від мережевого середовища та політики доступу зовнішніх сайтів. Адже, на швидкість і стабільність роботи системи впливають доступність вебресурсів, час відповіді серверів, редиректи, тимчасові збої або зміни правил доступу до сторінок. Окремо варто зазначити, що Work.ua може використовувати додаткові механізми захисту доступу, зокрема перевірки на рівні Cloudflare. У такому випадку, якщо VPN вимкнено або система визначає, що запит виконується не з території України, доступ до сторінок Work.ua може бути обмежений. Це може впливати на можливість отримання HTML-вмісту та, відповідно, на результат роботи парсера. Крім того, режим *include_job_details = true* збільшує час виконання, оскільки для кожної вакансії потрібно окремо завантажити детальну сторінку. Також система потребує періодичної перевірки коректності отриманих результатів, оскільки навіть незначні зміни у структурі сторінок можуть призводити до неповного або некоректного збору даних.

Таблиця 4.4 – Результати аналізу системи та виявлені обмеження

Напрям аналізу	Отриманий результат	Виявлене обмеження
Парсинг вакансій	Система коректно виділяє основні поля вакансій із Work.ua та Djinni: назву, локацію, компанію, зарплату за наявності, опис і посилання	Якість парсингу залежить від актуальної HTML-структури сайтів
MCP-взаємодія	Агент отримує доступ до інструментів, передає параметри та отримує структуровані JSON-відповіді	Коректність роботи залежить від правильності переданих аргументів і стабільності MCP-клієнта
Валідація параметрів	Некоректні або неповні запити відхиляються до виконання основної логіки	Потрібно заздалегідь дотримуватися вимог до типів, форматів і обов’язкових полів
Експорт результатів	Дані можуть зберігатися у форматах JSON і CSV.	Експорт орієнтований переважно на локальне збереження файлів
Продуктивність	У режимі без деталізації система швидше повертає список вакансій	Режим include_job_details = true збільшує час виконання через додаткові HTTP-запити
Мережевий доступ	Система може отримувати дані з відкритих сторінок сайтів	Доступ до сайтів може залежати від мережі, VPN, Cloudflare або регіональних обмежень
Тестування	Перевірено основні сценарії парсингу, валідації, експорту та взаємодії з агентом	Автоматизовані тести переважно охоплюють модульний рівень

Як видно з таблиці 4.5, розроблена система успішно виконує основні функції, необхідні для збору, структуризації та експорту вакансій. Водночас її робота залежить від зовнішніх чинників, зокрема стабільності сайтів, доступності HTML-сторінок, мережевого середовища та актуальності структури вебсторінок. Виявлені обмеження не заперечують працездатність системи, але визначають напрями її

подальшого вдосконалення, зокрема підвищення стійкості парсерів, оптимізацію режиму деталізації та розширення тестового покриття.

Висновок до четвертого розділу

Четвертий розділ був присвячений тестуванню та аналізу результатів роботи інформаційної системи. Основну увагу приділено перевірці працездатності, коректності взаємодії агента з MCP-сервером, точності та повноти зібраних даних, а також виявленню основних обмежень розробленого рішення.

Перший етап був присвячений методиці тестування системи. Вона передбачала поєднання модульної перевірки окремих функцій із функціональною перевіркою повного сценарію роботи MCP-інструменту. Такий підхід дозволив окремо оцінити коректність парсингу, валідації параметрів, експорту результатів і обробки помилок, а також перевірити цілісну роботу системи під час взаємодії з агентом.

Під час перевірки взаємодії агента з MCP-сервером було підтверджено, що сервер коректно запускається як локальний stdio-процес, надає агенту доступ до визначеного набору інструментів, приймає параметри запитів і повертає структуровані відповіді у форматі TextContent із JSON-вмістом. Також було перевірено обробку некоректних параметрів, що підтвердило правильну роботу механізму валідації та повернення стандартизованих помилок.

Окремо було проаналізовано точність і повноту збору даних із Work.ua та Djinni. Порівняння отриманих JSON-відповідей із фактичними сторінками пошуку показало, що система коректно визначає основні поля вакансій, зокрема назву, локацію, компанію, зарплату за наявності, службові метадані та посилання. Якщо певне поле відсутнє на сторінці, система не підставляє довільні значення, а залишає відповідне поле порожнім, що підвищує достовірність результатів.

Було також оцінено продуктивність роботи системи в режимах із деталізацією та без неї. Встановлено, що режим без деталізації працює швидше, оскільки передбачає обробку лише сторінки пошуку. У режимі *include_job_details*

= true час виконання збільшується через необхідність додаткового завантаження сторінок окремих вакансій. Це дозволяє зробити висновок, що вибір режиму роботи має залежати від потреб користувача: швидке отримання списку вакансій або повний опис кожної позиції.

У межах аналізу результатів було визначено основні обмеження системи. До них належать залежність від HTML-структури зовнішніх сайтів, вплив мережевого середовища, можливі регіональні обмеження доступу, зокрема через Cloudflare або VPN, а також збільшення часу виконання в режимі деталізації. Виявлені обмеження є характерними для систем, що працюють із відкритими вебсторінками без використання офіційних API.

Отже, результати тестування підтвердили, що розроблена система є працездатною, модульною та придатною для практичного використання як MCP-інструмент для збору вакансій. Вона забезпечує коректну взаємодію з агентом, структуроване подання результатів, підтримку експорту у JSON і CSV, а також передбачувану обробку помилок. Подальший розвиток системи може бути спрямований на підвищення стійкості парсерів до змін вебсторінок, оптимізацію продуктивності та розширення кількості підтримуваних джерел вакансій.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розроблено інформаційну систему на основі MCP-інструменту для вебскрапінгу вакансій із ресурсів Work.ua та Djinni. Система забезпечує пошук вакансій, отримання структурованих результатів, взаємодію з агентом через MCP-сервер, а також експорт отриманих даних у форматах JSON і CSV.

В процесі виконання було проаналізовано предметну область вебскрапінгу, розглянуто особливості використання Model Context Protocol та обґрунтовано вибір технологічного стеку. Для реалізації системи використано Python, бібліотеки httpx, BeautifulSoup, Pydantic, mcp та засоби тестування, що дозволило побудувати модульну й розширювану архітектуру.

Практична частина роботи включала створення MCP-сервера, реалізацію інструментів для пошуку вакансій, окремих модулів для Work.ua і Djinni, а також механізму експорту результатів. Проведене тестування підтвердило коректність взаємодії агента з сервером, правильність валідації параметрів, точність збору основних полів вакансій і працездатність експорту.

Отже, мету роботи було досягнуто: створено працездатний прототип MCP-інструменту для збору та структуризації вакансій. Подальший розвиток системи може бути спрямований на додавання нових джерел даних, підвищення стійкості парсерів до змін HTML-структури сайтів і оптимізацію швидкодії.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Liu B. Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data : підручник. 2-ге вид. Berlin : Springer, 2011. С. 311–315, 363–390.

2. Russell S. J., Norvig P. Artificial Intelligence: A Modern Approach : підручник. 3-тє вид. Upper Saddle River : Prentice Hall, 2010. С. 34–41, 50–61.

3. Model Context Protocol. Architecture : вебсайт. URL: https://modelcontextprotocol.io/docs/learn/architecture (дата звернення: 05.04.2026).

4. Mitchell R. Web Scraping with Python : збір даних із сучасного вебу. 2-ге вид. Sebastopol : O'Reilly Media, 2018.

5. Kahlon N., Singh W. A Systematic Review of Web Scraping: Techniques, LLM-Enhanced Approaches, Performance Metrics, and Legal-Ethical Issues. SSRN. 2025. URL: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5429131

6. Kazmali A. S., Sayar A. Web Scraping: Legal and Ethical Considerations in General and Local Context – A Review. Procedia Computer Science. 2025. Vol. 259. P. 1563–1572. DOI: 10.1016/j.procs.2025.04.111. URL: https://www.sciencedirect.com/science/article/pii/S1877050925012128 (дата звернення: 26.04.2026).

7. Іванов Д., Гаркуша І. Методологія парсингу сторінок веб-сайтів для автоматизації збору різноструктурованих даних. Вісник Хмельницького національного університету. Технічні науки. 2025. DOI: 10.31891/2307-5732-2025-359-128. URL: https://heraldts.khmnu.edu.ua/index.php/heraldts/article/view/2013 (дата звернення: 26.04.2026).

8. Lazebnyi V., Tereshkina N., Shabarina M., Fedorov D. Method for Aggregating Unstructured Data Using Large Language Models. arXiv. 2026. DOI: 10.48550/arXiv.2604.16425. URL: https://browse-export.arxiv.org/abs/2604.16425 (дата звернення: 26.04.2026).

9. Perkins M. A., Akorede B. A. Tracking K-12 and Higher Education Job Postings Through Web-Scraped Longitudinal Data. Data. 2026. Vol. 11, № 3. Article 52.

DOI: 10.3390/data11030052. URL: https://www.mdpi.com/2306-5729/11/3/52 (дата звернення: 26.04.2026).

10. Molina I., Morales J., Keith B. Web Scraping Chilean News Media: A Dataset for Analyzing Social Unrest Coverage (2019–2023). Data. 2025. Vol. 10, № 11. Article 174. DOI: 10.3390/data10110174. URL: https://www.mdpi.com/2306-5729/10/11/174 (дата звернення: 26.04.2026).

11. Abd Rahman K., Ahmed S. A., Md Ali S. AI-Driven Web Content Scraping Focusing on Transformer-Based: A Systematic Literature Review. e-Jurnal Penyelidikan dan Inovasi. 2025. Vol. 12, № 4. P. 200–212. DOI: 10.53840/ejpi.v12i4.291. URL: https://ejpi.uis.edu.my/index.php/ejpi/article/view/291 (дата звернення: 26.04.2026).

12. Lacher S., Rohs M. Preparedness Without Pedagogy? An AI-Assisted Web Scraping Analysis of Informal Online Disaster Preparedness Resources for the Public. Education Sciences. 2026. Vol. 16, № 1. Article 146. DOI: 10.3390/educsci16010146. URL:
https://www.researchgate.net/publication/399889563_Preparedness_Without_Pedagogy_An_AI-Assisted_Web_Scraping_Analysis_of_Informal_Online_Disaster_Preparedness_Resources_for_the_Public (дата звернення: 26.04.2026).

13. Liu S. та ін. SCRIBES: Web-Scale Script-Based Semi-Structured Data Extraction with Reinforcement Learning. arXiv. 2025. DOI: 10.48550/arXiv.2510.01832. URL: https://browse-export.arxiv.org/abs/2510.01832 (дата звернення: 26.04.2026).

14. Fahrudin T. M. та ін. An Improved Reference Paper Collection System Using Web Scraping with Three Enhancements. Future Internet. 2025. Vol. 17, № 5. Article 195. DOI: 10.3390/fi17050195. URL: https://www.mdpi.com/1999-5903/17/5/195 (дата звернення: 26.04.2026).

15. SwirlAI Newsletter. Everything you need to know about MCP : вебсайт. URL: https://www.newsletter.swirlai.com/p/everything-you-need-to-know-about?r=15fduh&triedRedirect=true (дата звернення: 26.04.2026).

16. JSON-RPC 2.0 Specification : вебсайт. URL: https://www.jsonrpc.org/specification (дата звернення: 05.04.2026).

17. Schick T. та ін. Toolformer: Language Models Can Teach Themselves to Use Tools. 2023. URL: https://arxiv.org/abs/2302.04761 (дата звернення: 05.04.2026).

18. Function calling : документація OpenAI API. URL: https://developers.openai.com/api/docs/guides/function-calling (дата звернення: 07.04.2026).

19. ReAct: The Power of Reasoning and Acting in LLM Agents : вебсайт. URL: https://sangeethasaravanan.medium.com/react-the-power-of-reasoning-and-acting-in-llm-agents-3332c95b1e25 (дата звернення: 07.04.2026).

20. Plugins in Semantic Kernel : документація Microsoft Learn. URL: https://learn.microsoft.com/en-us/semantic-kernel/concepts/plugins/?pivots=programming-language-csharp (дата звернення: 07.04.2026).

21. Lewis P. та ін. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks : електронний ресурс. URL: https://arxiv.org/abs/2005.11401 (дата звернення: 07.04.2026).

22. Pydantic. Strict Mode : вебсайт. URL: https://pydantic.com.cn/en/concepts/strict_mode/ (дата звернення: 26.04.2026).

23. Model Context Protocol. Specification : вебсайт. URL: https://modelcontextprotocol.io/specification/2025-11-25 (дата звернення: 26.04.2026).

24. Athanasopoulou A., Fotiou N., Chatzopoulos A. Interacting with IoT Data Spaces Using LLMs and the Model Context Protocol. Sensors. 2026. Vol. 26, № 4. Article 1193. DOI: 10.3390/s26041193. URL: https://www.mdpi.com/1424-8220/26/4/1193 (дата звернення: 26.04.2026).

25. Hu N.-Z., Lin Y.-X., Huang H.-L., Lu P.-H., Lin Ch.-Ch., Hung Yu.-T., Jhang S.-C., Chou P.-Y. Applying Model Context Protocol for Offline Small Language Models in Industrial Data Management. Engineering Proceedings. 2026. Vol. 134, № 1. Article

31. DOI: 10.3390/engproc2026134031. URL: https://doi.org/10.3390/engproc2026134031 (дата звернення: 26.04.2026).

26. Di Maggio L. G. Predictive Maintenance MCP: An Open-Source Framework for Bridging Large Language Models and Industrial Condition Monitoring via the Model Context Protocol. Applied Sciences. 2026. Vol. 16, № 6. Article 2812. DOI: 10.3390/app16062812. URL: https://www.mdpi.com/2076-3417/16/6/2812 (дата звернення: 26.04.2026).

27. Fard B. J., Hasan S. A., Bell J. E. Facilitating AI-Driven Sustainability: A Service-Oriented Architecture for Interoperable Environmental Data Access. Sustainability. 2026. Vol. 18, № 5. Article 2445. DOI: 10.3390/su18052445. URL: https://doi.org/10.3390/su18052445 (дата звернення: 26.04.2026).

28. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259. URL: https://www.rfc-editor.org/rfc/rfc8259 (дата звернення: 26.04.2026).

29. Common Format and MIME Type for Comma-Separated Values (CSV) Files. RFC 4180. URL: https://www.rfc-editor.org/rfc/rfc4180 (дата звернення: 26.04.2026).

30. Model Context Protocol. Tools : документація. URL: https://modelcontextprotocol.io/specification/2025-11-25/server/tools

31. Pydantic documentation. Models : документація. URL: https://docs.pydantic.dev/latest/concepts/models/ (дата звернення: 07.04.2026).

32. HTTPX : документація. URL: https://www.python-httpx.org/ (дата звернення: 07.04.2026).

33. Beautiful Soup 4 documentation : документація. URL: https://beautiful-soup-4.readthedocs.io/en/latest/ (дата звернення: 07.04.2026).

34. SDKs : документація Model Context Protocol. URL: https://modelcontextprotocol.io/docs/sdk (дата звернення: 07.04.2026).

35. pytest documentation : документація. URL: https://docs.pytest.org/ (дата звернення: 07.04.2026).

36. pytest-asyncio. Configuration : документація. URL: https://pytest-asyncio.readthedocs.io/en/stable/reference/configuration.html (дата звернення: 07.04.2026).

37. Hatchling : документація. URL: https://hatch.pypa.io/latest/config/build/ (дата звернення: 07.04.2026).

38. logging – Logging facility for Python : документація Python. URL: https://docs.python.org/3/library/logging.html (дата звернення: 07.04.2026).

39. Playwright Python : документація. URL: https://playwright.dev/python/docs/intro (дата звернення: 07.04.2026).

40. lxml : документація. URL: https://lxml.de/ (дата звернення: 07.04.2026).

41. Scrapy documentation : документація. URL: https://docs.scrapy.org/ (дата звернення: 07.04.2026).

ДОДАТОК А

Лістинг коду server.py

```
"""MCP-сервер (stdio), стиль як у mcp-server-fetch: Server + list_tools + call_tool."""

from __future__ import annotations

import json
import os
from pathlib import Path
from typing import Annotated, Any, Literal

import httpx
from curl_cffi.requests.exceptions import HTTPError as CurlHTTPError
from mcp.server import Server
from mcp.server.stdio import stdio_server
from mcp.shared.exceptions import McpError
from mcp.types import ErrorData, TextContent, Tool, INTERNAL_ERROR, INVALID_PARAMS
from pydantic import BaseModel, BeforeValidator, Field, ValidationError, model_validator

from mcp_server_work.djinni import (
    DjinniListFilters,
    fetch_djinni_job_detail,
    search_djinni_jobs,
)
from mcp_server_work.scraper import (
    DEFAULT_USER_AGENT,
    fetch_vacancy_detail,
    search_vacancies,
)
from mcp_server_work.storage import (
    attach_export_path,
    normalize_to_csv_rows,
    write_payload_csv,
    write_payload_json,
)

_EXPORT_JSON_DESC = (
    "Опційно: зберегти цю ж відповідь у JSON або CSV. Відносний шлях у каталозі "
    "MCP_JOBS_EXPORT_DIR або ./mcp_scrape_exports (наприклад workua/python_p1.json "
    "або workua/python_p1.csv). У відповіді додається поле exported_to з абсолютним шляхом."
)
…….
```

ДОДАТОК Б

Лістинг коду scraper.py

```
"""HTTP-клієнт та парсинг HTML сторінок Work.ua."""
from __future__ import annotations
import asyncio
import re
import sys
from dataclasses import asdict, dataclass
from typing import Any
from urllib.parse import quote_plus, urlparse
from bs4 import BeautifulSoup
from curl_cffi.requests import AsyncSession
WORK_UA_ORIGIN = "https://www.work.ua"
# curl_cffi підбирає TLS/JA3 як у Chromium – без цього Work.ua часто відповідає 403 на httpx.
WORK_UA_IMPERSONATE = "chrome"
# Реалістичний UA + заголовки браузера: явний «ботівський» UA дає 403 на Work.ua (Cloudflare).
DEFAULT_USER_AGENT = (
    "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 "
    "(KHTML, like Gecko) Chrome/131.0.0.0 Safari/537.36"
)
def work_ua_browser_headers(
    user_agent: str,
    *,
    referer: str | None = None,
    accept_language: str = "uk-UA,uk;q=0.9,en-US;q=0.8,en;q=0.7",
    sec_fetch_site: str = "none",
) -> dict[str, str]:
    """Заголовки, близькі до Chromium – разом із Referer знижують 403 на Work.ua."""
    headers: dict[str, str] = {
        "User-Agent": user_agent,
        "Accept": (
            "text/html,application/xhtml+xml,application/xml;q=0.9,"
            "image/avif,image/webp,image/apng,*/*;q=0.8"
        ),
        "Accept-Language": accept_language,
        "Accept-Encoding": "gzip, deflate, br",
        "DNT": "1",
        "Connection": "keep-alive",
        "Upgrade-Insecure-Requests": "1",
        "Sec-Fetch-Dest": "document",
        "Sec-Fetch-Mode": "navigate",
        "Sec-Fetch-Site": sec_fetch_site,
        "Sec-Fetch-User": "?1",
        "Cache-Control": "max-age=0",
    }
    if referer:
        headers["Referer"] = referer
    return headers
```

ДОДАТОК В

Лістинг коду djinni.py

```
"""Парсинг вакансій Djinni (djinni.co/jobs/): список і сторінка вакансії."""
from __future__ import annotations
import re
from dataclasses import asdict, dataclass
from typing import Any
from urllib.parse import urlparse, urlencode
import httpx
from bs4 import BeautifulSoup
DJINNI_ORIGIN = "https://djinni.co"
# /jobs/813418-designer/
_DJINNI_JOB_HREF = re.compile(r"^/jobs/(\d+)-[^/]+/$")

async def _djinni_http_get_text(
    client: httpx.AsyncClient,
    url: str,
    user_agent: str,
    *,
    accept_language: str = "en-US,en;q=0.9,uk;q=0.8",
) -> str:
    """GET HTML з djinni.co лише через httpx – без curl_cffi / Playwright / заголовків Work.ua."""
    r = await client.get(
        url,
        headers={
            "User-Agent": user_agent,
            "Accept-Language": accept_language,
            "Accept": (
                "text/html,application/xhtml+xml,application/xml;q=0.9,"
                "image/avif,image/webp,*/*;q=0.8"
            ),
            "Upgrade-Insecure-Requests": "1",
        },
        timeout=30.0,
    )
    r.raise_for_status()
    return r.text

@dataclass
class DjinniJobSummary:
    job_id: str
    title: str
    company: str
    salary: str
    meta_line: str
    path: str
    url: str
```

ДОДАТОК Г

Лістинг коду storage.py

```
"""Збереження результатів у JSON і CSV (лише в межах каталогу експорту)."""
from __future__ import annotations
import csv
import io
import json
import os
import re
from pathlib import Path
from typing import Any
_REL_SAFE = re.compile(r"^[a-zA-Z0-9_.\-/]+$")
def get_export_base() -> Path:
    """Базовий каталог: MCP_JOBS_EXPORT_DIR або <cwd>/mcp_scrape_exports."""
    raw = os.environ.get("MCP_JOBS_EXPORT_DIR", "").strip()
    if raw:
        base = Path(raw).expanduser().resolve()
    else:
        base = (Path.cwd() / "mcp_scrape_exports").resolve()
    base.mkdir(parents=True, exist_ok=True)
    return base

def _resolved_export_path(relative: str) -> Path:
    rel = relative.strip().replace("\\", "/").lstrip("/")
    if not rel or ".." in rel.split("/"):
        raise ValueError(
            "export_file: потрібен відносний шлях без '..' (наприклад workua/search.json)"
        )
    if not _REL_SAFE.match(rel):
        raise ValueError(
            "export_file: дозволені лише a-z A-Z 0-9 _ - . /"
        )
    base = get_export_base()
    full = (base / rel).resolve()
    try:
        full.relative_to(base)
    except ValueError as e:
        raise ValueError("export_file виходить за межі каталогу експорту") from e
    full.parent.mkdir(parents=True, exist_ok=True)
    return full
```

ДОДАТОК Д

Лістинг коду work.ua JSON


```
{
  "meta": {
    "search_url": "https://www.work.ua/jobs-ui+ux/?ss=1&page=1",
    "filters": {
      "query": "UI UX",
      "page": 1
    },
    "page": 1,
    "total_parsed_on_page": 14,
    "returned": 3,
    "include_job_details": false,
    "source": "work.ua",
    "fetch_fallback": "curl_cffi",
    "playwright_navigation": null
  },
  "vacancies": [
    {
      "job_id": "8011782",
      "title": "UX/UI Designer",
      "company": "",
      "location": "Київ",
      "salary": "",
      "url": "https://www.work.ua/jobs/8011782/"
    },
    {
      "job_id": "8017458",
      "title": "Game UI/UX Designer",
      "company": "",
      "location": "Дистанційно",
      "salary": "65 000 – 71 000 грн",
      "url": "https://www.work.ua/jobs/8017458/"
    },
    {
      "job_id": "7284454",
      "title": "UX/UI Designer, військовослужбовець",
      "company": "",
      "location": "Вся Україна",
      "salary": "25 000 – 125 000 грн",
      "url": "https://www.work.ua/jobs/7284454/"
    }
  ]
}
```

ДОДАТОК Е

Лістинг коду djinni JSON


```
{
  "meta": {
    "search_url":"https://djinni.co/jobs/?all_keywords=UI%2FUX&search_type=basic-
search&page=1&primary_keyword=ui_ux&exp_level=no_exp&exp_level=1y&exp_level=2y&region=ukraine",
    "filters": {
      "keywords": "UI/UX",
      "page": 1,
      "search_type": "basic-search",
      "salary": null,
      "exp_level": [
        "no_exp",
        "1y",
        "2y"
      ],
      "employment": null,
      "company_type": null,
      "english_level": null,
      "domain": null,
      "region": "ukraine",
      "editorial": null,
      "hide_contacted": null,
      "show_stale": null,
      "company": null,
      "primary_keyword": "ui_ux",
      "extra_query": {}
    },
    "page": 1,
    "total_parsed_on_page": 15,
    "returned": 3,
    "include_job_details": false,
    "source": "djinni.co"
  },
  "jobs": [
    {
      "job_id": "816462",
      "title": "UI/UX Designer AI operator",
      "company": "Join.To.IT",
      "salary": "",
      "meta_line": "Тільки офіс · Україна (Київ) · 2 роки досвіду · Англійська - B2 · SaaS",
      "path": "/jobs/816462-ui-ux-designer-ai-operator/",
      "url": "https://djinni.co/jobs/816462-ui-ux-designer-ai-operator/"
    },
    {
      "job_id": "822096",
      "title": "Game UI/UX Designer",
      "company": "24hit.games",
      "salary": "",
```

```
    "meta_line": "Тільки офіс · Україна (Київ) · 1 рік досвіду · Англійська - A2 · Gamedev",
    "path": "/jobs/822096-game-ui-ux-designer/",
    "url": "https://djinni.co/jobs/822096-game-ui-ux-designer/"
   },
   {
    "job_id": "809438",
    "title": "Middle UX/UI designer",
    "company": "OnFire",
    "salary": "",
    "meta_line": "Тільки віддалено · Весь світ · 2 роки досвіду · Англійська - B2",
    "path": "/jobs/809438-middle-ux-ui-designer/",
    "url": "https://djinni.co/jobs/809438-middle-ux-ui-designer/"
   }
  ]
}
```