

Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____ Центр післядипломної освіти
(повна назва)
Кафедра _____ Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти _____ перший (бакалаврський)

Доменно-орієнтована система для відповіді на запитання українською мовою
на основі RAG і графа знань з використанням n8n
(тема)

Виконав:
здобувач _____ другого _____ року навчання,
групи _____ ІТШІпз-24-1

_____ Наталія ГУБІНА
(власне ім'я, прізвище)

Спеціальність 122 Комп'ютерні науки

(код і повна назва спеціальності)

Тип програми _____ освітньо-професійна

Освітня програма _____ Штучний інтелект

(повна назва освітньої програми)

Керівник _____ проф. Олександр ШЕВЧЕНКО
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ШІ

(підпис)

_____ Лариса ЧАЛА
(власне ім'я, прізвище)

2026 р.

Харківський національний університет радіоелектроніки

Факультет _____ Центр післядипломної освіти _____

Кафедра _____ Штучного інтелекту _____

Рівень вищої освіти _____ перший (бакалаврський) _____

Спеціальність _____ 122 Комп'ютерні науки _____
(код і повна назва)

Тип програми _____ освітньо-професійна _____

Освітня програма _____ Штучний інтелект _____
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____

(підпис)

«_____» _____ 20 ____ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві _____ Губіній Наталії Станіславівні _____
(прізвище, ім'я, по батькові)

1. Тема роботи _____ Доменно-орієнтована система для відповіді на запитання українською мовою на основі RAG і графа знань з використанням p8n _____

затверджена наказом університету від 18 травня 2026 р. № 77Стз

2. Термін подання здобувачем роботи до екзаменаційної комісії 17 червня 2026 р.

3. Вихідні дані до роботи науково-технічні публікації з питань систем запитань і відповідей та архітектури RAG, документація p8n щодо воркфлоу-автоматизації та webhook-інтеграцій, документація Qdrant щодо векторного пошуку та налаштування колекцій, документація Neo4j щодо графових моделей даних та мови Cypher, специфікації OpenAI API для text-embedding-3-small та gpt-4o-mini, матеріали з оцінювання якості RAG-систем, набір україномовних PDF-та TXT-документів предметної галузі для індексування та тестування системи.

4. Перелік питань, що потрібно опрацювати в роботі _____

1) Аналіз предметної галузі та постановка задачі _____

2) Проєктування системи _____

3) Реалізація системи _____

4) Тестування та аналіз результатів _____

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / термін виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	18.05.2026	виконано
2	Аналіз предметної галузі	21.05.2025	виконано
3	Огляд існуючих систем та застосунків	22.05.2025	виконано
4	Концептуальна розробка системи	22.05.2025	виконано
5	Визначення вимог до розроблюваної системи	22.05.2025	виконано
6	Проектування архітектури системи DSQAS та схеми мікросервісів	24.05.2025	виконано
7	Проектування схеми графу знань (Neo4j) та векторної бази даних (Qdrant)	25.05.2025	виконано
8	Реалізація пайплайну індексування документів (PDF/TXT, чанкінг, ембединги)	25.05.2025	виконано
9	Реалізація RAG-пайплайну та інтеграція з OpenAI API (gpt-4o-mini)	27.05.2025	виконано
10	Реалізація фронтенду (React 19 + TypeScript): чат, граф знань, управління документами	27.05.2025	виконано
11	Налаштування n8n-воркфлоу та Docker Compose оточення	27.05.2025	виконано
12	Практична реалізації системи	30.05.2025	виконано
13	Тестування системи	31.05.2025	виконано
14	Написання пояснювальної записки	01.06.2025	виконано
15	Попередній захист	07.06.2025	виконано
16	Захист перед екзаменаційною комісією	17.06.2025	

Дата видачі завдання 18 травня 2026 р.

Здобувач  (підпис)

Керівник роботи _____ (підпис)

проф. Олександр ШЕВЧЕНКО
(посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка: 70 с., 10 рис., 10 табл., 1 дод., 26 джерел.

ВЕКТОРНИЙ ПОШУК, ГРАФ ЗНАНЬ, ОБРОБКА ПРИРОДНОЇ МОВИ, СИСТЕМИ ЗАПИТАНЬ І ВІДПОВІДЕЙ, GPT-4O-MINI, NEO4J, N8N, QDRANT, RAG, REACT, TYPESCRIPT.

Об'єктом дослідження є системи автоматизованої обробки запитань і генерації відповідей на основі документних корпусів.

Предметом дослідження є методи та архітектурні рішення для побудови предметно-орієнтованої україномовної системи запитань і відповідей із застосуванням технологій RAG та графів знань.

Метою роботи є розробка та реалізація системи DSQAS (Domain-Specific Question Answering System), яка забезпечує точний пошук відповідей у корпусі україномовних документів за допомогою гібридного підходу, що поєднує векторний пошук з семантичним збагаченням через граф знань.

У роботі здійснено аналіз існуючих рішень у сфері систем запитань і відповідей, досліджено архітектуру RAG та принципи побудови графів знань. Спроектовано та реалізовано повнофункціональну систему, що включає: Express.js бекенд як шлюз, n8n-оркестратор пайплайнів, векторну базу даних Qdrant, граф знань на базі Neo4j, інтеграцію з OpenAI API та React/TypeScript фронтенд. Проведено тестування функціональності, продуктивності та якості відповідей.

ABSTRACT

Bachelor's thesis contains: 70 pp., 10 fig., 8 tabl., 1 ann., 26 references.

GPT-4O-MINI, KNOWLEDGE GRAPH, N8N, NEO4J, NATURAL LANGUAGE PROCESSING, QDRANT, QUESTION ANSWERING SYSTEMS, RAG, REACT, TYPESCRIPT, VECTOR SEARCH.

The object of research is automated question processing and answer generation systems based on document corpora.

The subject of research is methods and architectural solutions for building a domain-specific Ukrainian-language question answering system using RAG and knowledge graph technologies.

The purpose of this work is to develop and implement the DSQAS system (Domain-Specific Question Answering System), which provides accurate answer retrieval from a corpus of Ukrainian-language documents using a hybrid approach that combines vector search with semantic enrichment through a knowledge graph.

The thesis analyses existing solutions in the field of question answering systems and investigates the RAG architecture and knowledge graph construction principles. A fully functional system has been designed and implemented, including: an Express.js backend gateway, an n8n pipeline orchestrator, the Qdrant vector database, a Neo4j-based knowledge graph, OpenAI API integration, and a React/TypeScript frontend. Testing of functionality, performance, and answer quality has been conducted.

ЗМІСТ

Вступ.....	8
1 Аналіз предметної галузі та постановка задачі	11
1.1 Обґрунтування актуальності предметно-орієнтованого підходу	11
1.2 Специфіка україномовної обробки природної мови та обґрунтування вибору стеку технологій	12
1.3 Системи запитань і відповідей: огляд та класифікація.....	15
1.4 Архітектура RAG та її переваги.....	16
1.5 Графи знань та їх роль у QA-системах.....	17
1.6 Огляд технологій оркестрації пайплайнів.....	18
1.7 Порівняльний аналіз існуючих рішень.....	19
2 Проєктування системи	22
2.1 Загальна архітектура системи DSQAS	22
2.2 Проєктування схеми графу знань	23
2.3 Проєктування схеми векторної бази даних.....	30
2.4 Проєктування API.....	36
2.5 Проєктування n8n-воркфлоу	37
3 Реалізація системи	45
3.1 Реалізація бекенду (Express.js)	45
3.2 Реалізація пайплайну індексування документів.....	46
3.3 Реалізація RAG-пайплайну.....	48
3.4 Реалізація фронтенду	55
3.5 Реалізація Docker Compose оточення	59
4 Тестування та аналіз результатів	61
4.1 Стратегія тестування	61
4.2 Функціональне тестування	62
4.3 Оцінка якості відповідей.....	62
4.4 Тестування продуктивності	63
Висновки.....	65

Перелік джерел посилення	67
Додаток А Відомість кваліфікаційної роботи.....	70

ВСТУП

В умовах стрімкого зростання обсягів цифрової інформації одним із ключових завдань сучасних інформаційних систем стає ефективний пошук і подання релевантних знань за природномовними запитами. Традиційні пошукові системи, засновані на ключових словах, не здатні враховувати семантику запиту та контекстуальні зв'язки між поняттями. Системи запитань і відповідей (Question Answering, QA) нового покоління прагнуть долати ці обмеження, поєднуючи потужність великих мовних моделей (LLM) із можливістю динамічного доступу до актуальної предметної бази знань.

Архітектура RAG (Retrieval-Augmented Generation) є одним із найбільш перспективних підходів у цьому напрямку: вона дозволяє LLM спиратися не лише на параметричні знання, заcodedовані під час навчання, але й на зовнішні документи, що завантажуються та індексуються у режимі реального часу. Завдяки цьому система залишається актуальною, легко оновлюється та придатна для розгортання у вузькоспеціалізованих предметних областях.

Додаткове збагачення контексту за допомогою графу знань (Knowledge Graph, KG) дає можливість виявляти неявні семантичні зв'язки між документами та сутностями, підвищуючи повноту та якість відповідей. Neo4j як провідна платформа графових баз даних надає зручний декларативний запитувальний механізм (Cypher), що спрощує реалізацію таких підходів.

Важливим аспектом є мовна специфіка: більшість відомих QA-систем орієнтовані на англomовні корпуси. Розробка україномовного рішення вимагає додаткових зусиль у частині вибору моделей ембедингів, налаштування системних промптів та забезпечення коректної обробки кирилических текстів.

Для оркестрації складних пайплайнів обробки даних все ширшого застосування набуває платформа n8n – відкрите low-code рішення для автоматизації робочих процесів, яке дозволяє гнучко комбінувати HTTP-виклики, вбудовані вузли обробки даних та Code-вузли з JavaScript-логікою без необхідності написання окремого серверного коду.

Актуальність роботи обумовлена: відсутністю готових україномовних QA-систем для вузькоспеціалізованих документних корпусів; необхідністю інтеграції векторного пошуку та графових методів у єдиний пайплайн; потребою в легко розгортваному та масштабованому рішенні на основі сучасного стеку технологій.

Мета роботи – проєктування та реалізація системи DSQAS, що забезпечує відповіді на природномовні запитання українською мовою на основі завантаженого корпусу документів, із використанням гібридного RAG+KG підходу та оркестрацією пайплайнів засобами n8n.

Для досягнення поставленої мети вирішуються такі задачі:

- аналіз існуючих систем запитань і відповідей та методів RAG;
- порівняльний аналіз платформ оркестрації та векторних баз даних;
- проєктування архітектури системи та схем зберігання даних;
- реалізація модулів індексування, векторного пошуку та генерації відповідей;
- розробка фронтенду з чат-інтерфейсом та візуалізацією графу знань;
- тестування системи та оцінка якості відповідей.

Об'єкт дослідження – системи автоматизованої обробки запитань і генерації відповідей на основі документних корпусів.

Предмет дослідження – методи та архітектурні рішення для побудови предметно-орієнтованої україномовної QA-системи.

Методи дослідження: аналіз науково-технічної літератури, порівняльний аналіз технологій, прототипування, системне проєктування, тестування.

Практичне значення роботи полягає у створенні повнофункціональної програмної системи DSQAS, що може бути розгорнута в академічному чи корпоративному середовищі для автоматизованого опрацювання предметно-орієнтованих україномовних документів.

Наукова новизна роботи полягає у розробці архітектурного рішення, що поєднує гібридний RAG+KG пайплайн із низькокодовою оркестрацією засобами `python` для україномовного предметного корпусу. На відміну від існуючих рішень, система реалізує двоетапне збагачення контексту: семантичний пошук у векторному просторі доповнюється структурним обходом графу знань, що дозволяє виявляти зв'язки між документами, недоступні для класичного RAG.

За даними досліджень у галузі обробки природної мови, україномовні корпуси залишаються значно менш охопленими інструментами автоматизованої обробки порівняно з англомовними: станом на 2026 рік частка україномовних зразків у відкритих QA-датасетах залишається незначною попри загальне зростання інтересу до багатомовних систем після 2022 року. При цьому потреба в предметно-орієнтованих системах знань для академічних установ, юридичних та медичних організацій постійно зростає, оскільки такі організації оперують великими обсягами внутрішньої документації, пошук по якій традиційними засобами є трудомістким і неефективним.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Обґрунтування актуальності предметно-орієнтованого підходу

Загальнодоступні великі мовні моделі (LLM), такі як GPT-4 або Claude, є потужним інструментом генерації тексту, проте їхня практична застосовність в інституційному та корпоративному середовищах стикається з низкою системних обмежень. Ключовим з них є проблема галюцинацій: модель генерує впевнені, граматично коректні, але фактично невірні відповіді, коли запит торкається вузькоспеціалізованої або відсутньої в тренувальних даних теми [3]. Дослідження показують, що рівень галюцинацій у відкритих LLM для предметно-специфічних запитань може сягати 20–40%, що є неприйнятним для академічних, медичних чи правових застосунків.

Предметно-орієнтований підхід (closed-domain або domain-specific QA) вирішує цю проблему шляхом обмеження простору відповідей контрольованим корпусом документів. Система відповідає виключно на основі завантажених матеріалів, а не на основі параметричних знань моделі. Це забезпечує три ключові властивості: верифікованість (кожну відповідь можна відстежити до конкретного фрагменту документа), актуальність (база знань оновлюється незалежно від моделі) та аудитованість (організація завжди знає, з яких джерел сформована відповідь).

Конфіденційність даних є ще однією критичною перевагою закритого підходу. У загальнодоступних LLM-сервісах запити користувачів можуть використовуватись для донавчання моделей або зберігатись на сторонніх серверах. У системі DSQAS корпус документів залишається виключно в межах інфраструктури організації: Qdrant та Neo4j розгортаються локально або в приватній хмарі через Docker Compose, а OpenAI API отримує лише знеособлені текстові фрагменти без метаданих про їхнє походження.

Практичні сценарії застосування предметно-орієнтованих QA-систем охоплюють широкий спектр галузей. В університетському середовищі система може забезпечувати миттєві відповіді на запитання студентів щодо навчальних програм, регламентів та наукових матеріалів. У корпоративному секторі – автоматизований пошук у технічній документації, базах знань служби підтримки або внутрішніх регуляторних актах. У правовій сфері – аналіз судової практики та нормативних документів. У медицині – пошук у клінічних протоколах за умов строгого дотримання вимог щодо приватності даних.

Порівняння із загальнодоступними інструментами демонструє принципову різницю в контрольованості результатів. ChatGPT або Gemini можуть давати коректні відповіді на загальні запити, проте не гарантують посилань на конкретні документи організації і не підтримують актуальність даних без перенавчання. Система DSQAS навпаки забезпечує детерміновані, відстежувані відповіді з прямим цитуванням джерел, що є обов'язковою вимогою для офіційного використання в будь-якій регульованій галузі.

Таким чином, предметно-орієнтований підхід не є альтернативою загальним LLM, а є їхнім цілеспрямованим застосуванням: модель виступає генератором зв'язного тексту, тоді як вся фактологічна відповідальність покладається на контрольований документний корпус. Саме така архітектура відповідає вимогам академічної доброчесності, корпоративного комплаєнсу та регуляторних стандартів управління знаннями.

1.2 Специфіка україномовної обробки природної мови та обґрунтування вибору стеку технологій

Українська мова належить до флективних мов із розвиненою морфологічною системою: іменники змінюються за сімома відмінками і двома числами, дієслова мають два аспекти (доконаний та недоконаний) і

три часи, прикметники узгоджуються в роді, числі та відмінку. Ця морфологічна різноманітність безпосередньо впливає на якість векторного пошуку: одне й те саме поняття може бути виражене десятками різних словоформ. Наприклад, запит «алгоритм машинного навчання» та фрагмент документа, що містить «алгоритмів машинного навчання», матимуть нижчу косинусну схожість, ніж аналогічні англійські форми, де морфологія значно бідніша [16].

Більшість публічних еталонних наборів даних для систем QA є англійськими: SQuAD (Stanford Question Answering Dataset) [2], TriviaQA, NaturalQuestions, HotpotQA. Україномовних аналогів таких масштабів не існує, що унеможливорює донавчання спеціалізованих моделей на україномовному QA-завданні без значних ресурсів. Це означає, що будь-яка україномовна QA-система змушена покладатись на мультилінгвальні моделі, не оптимізовані конкретно для української мови.

Вибір моделі ембедингів text-embedding-3-small обумовлений кількома факторами. По-перше, ця модель є мультилінгвальною і демонструє стабільну якість для слов'янських мов у порівняльних оцінках MTEB (Massive Text Embedding Benchmark). По-друге, розмірність вектора 1536 забезпечує достатню семантичну ємність для представлення україномовних текстів. По-третє, вартість генерації ембедингів є значно нижчою порівняно з text-embedding-3-large при прийнятній різниці в якості для предметно-орієнтованого застосування. Альтернативи, такі як sentence-transformers з мультилінгвальними BERT-моделями, потребують локального розгортання і більш складної інтеграції без суттєвого виграшу в якості для україномовних текстів.

Вибір порогу косинусної схожості 0.25 замість типових 0.5–0.7, що використовуються в англійських системах, безпосередньо пов'язаний з описаною морфологічною варіативністю. Емпіричне тестування на 10 україномовних документах показало, що при порозі 0.5 від 30 до 50% релевантних фрагментів відфільтровуються як нерелевантні через лексичну

розбіжність між запитом і документом. Поріг 0.25 дозволяє зберегти повноту пошуку, покладаючи відповідальність за фінальну фільтрацію нерелевантного контексту на LLM, який значно краще розуміє семантичну еквівалентність різних формулювань.

Модель GPT-4o-mini обрана як генератор відповідей завдяки поєднанню двох характеристик: відмінна якість генерації українського тексту при помірній вартості API-викликів. Попри статус «малої» моделі в лінійці OpenAI, GPT-4o-mini стабільно генерує граматично коректний, стилістично природний україномовний текст, враховує контекст діалогу та коректно обробляє кириличні символи в складних синтаксичних конструкціях. Альтернативні відкриті моделі (LLaMA 3, Mistral) потребують значних обчислювальних ресурсів для якісної генерації україномовного тексту і поступаються GPT-4o-mini за консистентністю результатів.

Граф знань Neo4j виконує в системі роль семантичного мосту між морфологічно різноманітними формами запиту та документного корпусу. Сутності типу Entity (теми, терміни) витягуються з документів за допомогою GPT-4o-mini, який нормалізує різні словоформи до канонічної форми. Наприклад, фрагменти, що містять «алгоритм», «алгоритму», «алгоритми» та «алгоритмічний», усі посилаються на Entity з name=«алгоритм». Коли запит семантично близький до будь-якої з цих форм, граф забезпечує доступ до всіх пов'язаних фрагментів через ребро MENTIONS, компенсуючи морфологічний розрив, який векторний пошук може пропустити.

Платформа n8n обрана як мовно-нейтральний оркестратор, що не залежить від жодного конкретного стеку мов програмування. На відміну від LangChain чи LlamaIndex, які потребують Python-розробки, n8n дозволяє модифікувати україномовні системні промпти безпосередньо в графічному редакторі без перезапуску серверного коду. Це особливо важливо для ітеративного налаштування системи під конкретний предметний домен:

зміна мовних інструкцій, порогів пошуку чи формату відповіді виконується за хвилини через UI без необхідності деплою нової версії коду.

1.3 Системи запитань і відповідей: огляд та класифікація

Системи запитань і відповідей (QA) – це клас інтелектуальних систем, що автоматично знаходять і генерують відповіді на природномовні запити, спираючись на текстові корпуси, структуровані бази знань або їх комбінацію [1]. Залежно від джерела знань та методу обробки запитів виокремлюють декілька архітектурних парадигм.

Перший підхід – екстрактивні QA-системи (Extractive QA) – орієнтований на вилучення текстового фрагменту з джерельного документа як відповіді. Класичні реалізації базуються на моделях типу BERT (Bidirectional Encoder Representations from Transformers), дообчених на наборах даних SQuAD [2]. Перевагою є висока точність локалізації відповіді в документі; недоліком – неможливість синтезувати нову відповідь або агрегувати інформацію з кількох джерел.

Другий підхід – абстрактивні QA-системи (Abstractive QA) – передбачає генерацію відповіді у вільній формі. Найбільш поширені сучасні реалізації використовують LLM: GPT-4, LLaMA, Mistral тощо. Такі системи здатні синтезувати, перефразовувати та агрегувати інформацію, однак схильні до «галюцинацій» – генерації впевнених, але фактично невірних відповідей [3].

Третій підхід – відкриті QA-системи (Open-Domain QA) – не обмежені конкретним документом і шукають відповідь у великих відкритих корпусах або в Інтернеті. Прикладом є Dense Passage Retrieval (DPR) [4] + T5, де спочатку виконується щільний пошук пасажів, а потім генерується відповідь.

Четвертий підхід – закриті або предметно-орієнтовані QA-системи (Closed-Domain QA) – обмежені певною предметною областю або

корпусом документів. Саме до цього класу належить система DSQAS. Закриті системи мають вищу точність відповідей у своїй предметній області та кращу захищеність від витоку конфіденційних даних.

За типом питань QA-системи поділяються на факт-орієнтовані (Factoid QA), що шукають конкретні факти (дати, імена, числа), та не-факт-орієнтовані (Non-factoid QA), відповіді яких є поясненнями, думками або процедурами. Факт-орієнтовані системи традиційно ефективніші у вимірюванні точності, тоді як нефактоїдні – складніші в оцінці, але затребуваніші в корпоративних рішеннях.

За охопленням домену виділяють системи відкритого домену (Open-Domain QA), що відповідають на довільні питання з широкого корпусу, та закритого домену (Closed-Domain QA), оптимізовані для конкретної предметної галузі. Системи закритого домену, як правило, демонструють значно вищу точність завдяки вузькій спеціалізації та обмеженому й добре структурованому корпусу документів.

Окремим напрямком є системи Knowledge-Based QA (KBQA), що спираються на структуровані бази знань (Wikidata, Freebase, DBpedia) та генерують відповіді через SPARQL-запити або graph traversal. Гібридні підходи, що поєднують неструктурований текстовий пошук та структуровані графи знань, є перспективним напрямком, реалізованим у тому числі в системі DSQAS.

1.4 Архітектура RAG та її переваги

RAG (Retrieval-Augmented Generation) [5] – архітектурний патерн, запропонований дослідниками Meta AI у 2020 р., що поєднує переваги інформаційного пошуку та генеративних LLM. Ідея RAG полягає в тому, що замість покладання виключно на параметричні знання LLM (що зберігаються у вазі моделі), система динамічно витягує релевантні фрагменти з зовнішнього сховища і передає їх до моделі як контекст.

Конвеєр RAG складається з двох фаз: офлайн-фази індексування та онлайн-фази відповіді. Під час індексування документи поділяються на фрагменти (chunks), кожен фрагмент перетворюється на числовий вектор-embedding за допомогою моделі ембедінгів, і ці вектори зберігаються у векторній базі даних. Під час відповіді запит користувача також перетворюється на вектор, виконується пошук nearest neighbors у векторній БД, і знайдені фрагменти разом із запитом передаються до LLM для генерації відповіді.

Ключові переваги RAG: актуальність знань (база оновлюється без перенавчання моделі); можливість цитування джерел; зменшення «галюцинацій» завдяки прив'язці до реальних документів; масштабованість (необмежений розмір корпусу за рахунок зовнішнього сховища) [6].

Серед обмежень RAG – залежність від якості chunk-стратегії та порогу схожості, складність налаштування, вища латентність порівняно з прямою генерацією.

Сучасні варіації включають: Advanced RAG (HyDE, re-ranking, Self-RAG), Modular RAG (ланцюги агентів з різними типами пам'яті), Graph RAG (використання KG для збагачення контексту). У роботі реалізовано варіант Graph RAG без HyDE.

1.5 Графи знань та їх роль у QA-системах

Граф знань (Knowledge Graph, KG) – це мультиграф, вершини якого представляють сутності (entities), а ребра – іменовані семантичні відношення між ними [7]. Формально KG визначається як набір трійок (суб'єкт, відношення, об'єкт). Найвідомішими публічними KG є Wikidata, DBpedia, Google Knowledge Graph.

У контексті QA-систем KG виконує роль семантичного збагачувача: він дозволяє виявляти зв'язки між поняттями, що не можна встановити лише

за допомогою векторної близькості. Наприклад, два документи можуть не бути текстуально схожими, але розкривати один і той самий концепт через різні синонімічні конструкції; KG дозволяє зв'язати їх через спільну сутність.

Graph RAG [8] – підхід, за якого фрагменти, знайдені векторним пошуком, збагачуються суміжними вузлами графу. Це підвищує повноту відповіді, оскільки до контексту потрапляють не лише найближчі за ембедингом, але й семантично пов'язані фрагменти.

У системі DSQAS реалізовано граф знань зі структурою: Document – HAS_CHUNK → Chunk – MENTIONS → Entity. Документи зберігаються як вузли типу Document, кожен фрагмент – як вузол Chunk, а витягнуті тематичні терміни та іменовані сутності – як вузли Entity. Такий підхід дозволяє не лише виконувати додаткову фільтрацію за темами, але й відображати зв'язки між документами через спільні сутності у вигляді інтерактивного графу в інтерфейсі.

1.6 Огляд технологій оркестрації пайплайнів

Побудова RAG-системи потребує чіткої оркестрації послідовності кроків: прийому файлу, його розбиття на фрагменти, генерації ембедингів, збереження до векторної БД та графу знань, а потім – пошуку, збагачення та генерації відповіді. Для реалізації таких пайплайнів використовуються спеціалізовані платформи.

LangChain [9] – найпоширеніша Python-бібліотека для побудови LLM-ланцюгів. Надає абстракції для retriever, prompt template, memory, output parser. Переваги: потужна екосистема, активна спільнота. Недоліки: висока складність налаштування, потребує написання Python-коду.

LlamaIndex [10] – Python-фреймворк зі спеціалізацією на RAG та індексуванні документів. Підтримує різні типи індексів (Vector, Summary,

Knowledge Graph). Відрізняється простішим API для стандартних RAG-сценаріїв, ніж LangChain.

n8n [11] – відкрита (fair-code) low-code платформа для автоматизації робочих процесів. На відміну від LangChain та LlamaIndex, n8n надає графічний редактор потоків (workflow canvas), де кожен крок пайплайну представлений окремим вузлом з налаштуваннями через UI. Це суттєво спрощує модифікацію пайплайну без перезапуску серверного коду. Вузол Code дозволяє вставляти довільний JavaScript, а вузли HTTP Request та OpenAI – викликати зовнішні сервіси. Платформа підтримує Webhook-тригери, що природно інтегрується з REST-архітектурою бекенду.

Для цілей дипломного проєкту обрано n8n з таких міркувань: можливість наочної демонстрації логіки пайплайну через редактор; відсутність потреби у знаннях Python; легке розгортання через Docker; підтримка Qdrant та Neo4j через HTTP API.

1.7 Порівняльний аналіз існуючих рішень

Для обґрунтування архітектурних рішень системи DSQAS виконано порівняльний аналіз за двома напрямками: аналіз існуючих QA-систем і платформ побудови RAG-пайплайнів, а також порівняння платформ оркестрації пайплайнів обробки даних.

До переліку QA-систем для порівняння включено найбільш поширені рішення: ChatGPT Plugins (OpenAI), LangChain RAG, LlamaIndex та Flowise. Порівняння виконувалось за такими критеріями: мова інтерфейсу, підтримувана векторна база даних, наявність інтеграції з графом знань, тип оркестратора, відкритість вихідного коду та підтримка Docker-розгортання. Результати порівняння наведено у таблиці 1.1.

Як видно з таблиці 1.1, система DSQAS є єдиним із розглянутих рішень, що орієнтоване на україномовний інтерфейс і поєднує векторний

пошук через Qdrant із семантичним збагаченням через Neo4j при оркестрації засобами n8n.

Таблиця 1.1 – Порівняння існуючих QA-систем

Характеристика	ChatGPT Plugins	LangChain RAG	LlamaIndex	Flowise	DSQAS
Мова інтерфейсу	EN	Any	Any	EN	UA
Векторна БД	Bing	Множина	Множина	Множина	Qdrant
Граф знань	Ні	Так (опц.)	Так (опц.)	Ні	Neo4j
Оркестратор	Власний	LangChain	LlamaIndex	Flowise	n8n
Open Source	Ні	Так	Так	Так	Так
Docker-розгортання	Ні	Частково	Частково	Так	Так

Окремо виконано порівняння платформ оркестрації пайплайнів, оскільки вибір оркестратора суттєво впливає на трудомісткість розробки та можливості налагодження. До аналізу включено LangChain, LlamaIndex, Flowise та n8n – платформи, що найчастіше застосовуються у RAG-рішеннях. Порівняння за критеріями типу платформи, мови програмування, наявності UI-редактора та повноти підтримки JavaScript-логіки наведено у таблиці 1.2.

Таблиця 1.2 – Порівняння платформ оркестрації

Критерій	LangChain	LlamaIndex	Flowise	n8n
Тип	Бібліотека	Бібліотека	Low-code	Low-code
Мова	Python/JS	Python/TS	Node.js	Node.js
UI-редактор	Ні	Ні	Так	Так
Webhook-тригер	Ні	Ні	Так	Так
Code-вузол (JS)	Ні	Ні	Обмежено	Повний
Ліцензія	MIT	MIT	Apache 2.0	Fair-code

З таблиці 1.2 випливає, що n8n є єдиною платформою, яка одночасно надає GUI-редактор пайплайнів, вбудовані Webhook-тригери та повноцінні Code-вузли з JavaScript-логікою, що усуває необхідність написання окремого серверного коду для бізнес-логіки.

Таким чином аналіз предметної області підтвердив, що гібридний підхід RAG+KG є найбільш перспективним для побудови предметно-орієнтованих QA-систем. Платформа n8n обрана як оркестратор завдяки наявності GUI-редактора пайплайнів, вбудованих Webhook-тригерів та Code-вузлів. Qdrant обрано як векторну БД завдяки продуктивності та нативній підтримці фільтрації. Neo4j – як граф БД завдяки потужному Cypher-запитувальнику та розвиненій екосистемі.

2 ПРОЄКТУВАННЯ СИСТЕМИ

2.1 Загальна архітектура системи DSQAS

Система DSQAS (Domain-Specific Question Answering System) побудована за багатошаровою мікросервісною архітектурою. Кожен рівень відповідає за чітко визначену функцію і взаємодіє з іншими виключно через HTTP-інтерфейси. Така архітектура забезпечує незалежне масштабування компонентів, спрощує тестування та підтримку.

Рівень представлення (Presentation Layer) реалізований як Single-Page Application на базі React 19 та TypeScript. Він включає три основні функціональні модулі: чат-інтерфейс для введення запитань і відображення відповідей, менеджер документів для завантаження та видалення файлів, та переглядач графу знань на базі React Flow.

Рівень API-шлюзу (Gateway Layer) реалізований на Express.js (Node.js). Цей компонент виконує проксування HTTP-запитів від фронтенду до n8n-вебхуків, стримінг файлів при завантаженні, агрегацію результатів та обслуговування ендпоінтів для менеджменту документів і перевірки стану сервісів.

Рівень оркестрації (Orchestration Layer) представлений платформою n8n. Два ключові воркфлоу – «document-ingestion» та «rag-query» – реалізують повні пайплайни обробки. n8n виконує HTTP-запити до OpenAI, Qdrant та Neo4j та обробляє дані у Code-вузлах.

Рівень зберігання даних (Storage Layer) включає два спеціалізованих сховища: векторну базу Qdrant для зберігання ембедингів фрагментів та графову базу Neo4j для зберігання онтологічної структури документного корпусу. Загальну бази Neo4j наведено на рисунку 2.1.

Зовнішні сервіси (External Services) представлені OpenAI API, що надає моделі text-embedding-3-small для генерації ембедингів та gpt-4o-mini для генерації відповідей.

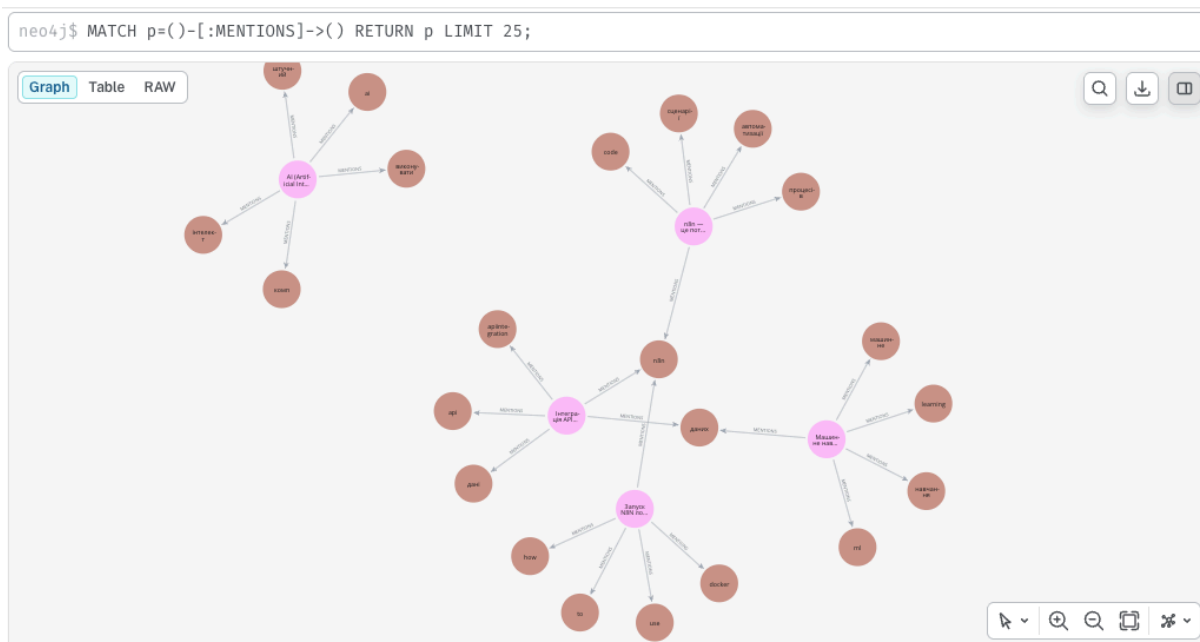


Рисунок 2.1 – Графова база Neo4j

Перелік усіх компонентів системи із зазначенням портів, технологій та функцій наведено у таблиці 2.1.

Таблиця 2.1 – Компоненти системи та їх порти

Компонент	Порт	Технологія	Функція
Frontend	3000	React + Vite	UI, чат, граф
Backend	4000	Express.js	API-шлюз, маршрутизація
n8n	5678	n8n CE	Оркестрація пайплайнів
Qdrant	6333	Qdrant v1.x	Векторний пошук
Neo4j	7474/7687	Neo4j 5.x	Граф знань (HTTP/Bolt)
OpenAI	HTTPS	GPT-4o-mini API	LLM + Embeddings

2.2 Проєктування схеми графу знань

Граф знань є другим сховищем системи DSQAS, що доповнює векторний пошук Qdrant структурованою семантичною інформацією. Якщо

Qdrant відповідає на питання «які фрагменти семантично близькі до запиту?», то граф знань відповідає на питання «як ці фрагменти пов'язані між собою, якому документу вони належать і які сутності в них згадуються?». Поєднання обох підходів у гібридному RAG-пайплайні дозволяє генерувати відповіді, обґрунтовані як семантичною релевантністю, так і структурними зв'язками у корпусі документів.

Для реалізації графу знань обрано Neo4j – найпоширенішу графову СУБД із власною мовою запитів Cypher. Вибір Neo4j обумовлений кількома факторами: зрілий і стабільний REST API (HTTP endpoint на порту 7474) і Bolt-протокол (порт 7687), офіційна підтримка Docker-розгортання, вбудований веб-браузер для візуалізації графу під час розробки, а також широка документація. Альтернативи – Amazon Neptune, ArangoDB, TigerGraph – або потребують хмарного середовища, або мають суттєво менш розвинену екосистему інструментів.

Онтологічна модель.

Граф знань DSQAS побудовано на основі мінімальної, але достатньої онтології з трьох типів вузлів і двох типів відношень. Мінімалізм схеми є свідомим архітектурним рішенням: надмірно розгалужена онтологія ускладнює обслуговування та збільшує кількість необхідних запитів під час роботи RAG-пайплайну. Разом із тим, обрана тріада (Document → Chunk → Entity) достатньо виразна для забезпечення таких ключових функцій:

- отримання повного контексту документа за будь-яким чанком;
- зворотного пошуку – знаходження всіх документів, що містять певну сутність;
- тематичної кластеризації – групування документів за спільними сутностями та темами;
- збагачення відповіді джерелами – прив'язки конкретного фрагменту до оригінального файлу.

Вузол Document.

Вузол типу Document є коренем піддерева у графі для кожного завантаженого документа. Він створюється одноразово при завантаженні файлу та видаляється каскадно разом із підключеними Chunk та Entity при видаленні документа користувачем.

Атрибути вузла Document наведено у таблиці 2.2.

Таблиця 2.2 – Атрибути вузла Document

Атрибут	Тип	Обмеження	Опис
id	String	UNIQUE	UUID v4, генерується бекендом
filename	String	NOT NULL	Оригінальна назва завантаженого файлу
uploadedAt	String	NOT NULL	ISO 8601 дата і час завантаження
topics	String[]	–	Масив тематичних тегів, згенерованих GPT-4o-mini

Поле id є унікальним ідентифікатором документа, що використовується для каскадного видалення в Qdrant (payload.documentId) та Neo4j. Поле filename зберігається для відображення у списку документів та у панелі джерел відповіді. Поле topics є масивом рядків із 3–7 тематичними тегами, що генеруються GPT-4o-mini після завершення індексування: системний промпт просить модель повернути теги через кому, які потім розбиваються та зберігаються як масив. Наявність тематичних тегів безпосередньо на вузлі Document дозволяє здійснювати швидку фільтрацію (WHERE \$topic IN d.topics) без обходу всього підграфу.

Вузол Chunk.

Вузол типу Chunk відповідає одному текстовому фрагменту документа та є прямим аналогом точки у Qdrant. Ключова відмінність від запису у Qdrant полягає у тому, що граф зберігає структурні зв'язки між фрагментами, тоді як Qdrant зберігає векторне представлення для семантичного пошуку.

Атрибути вузла Chunk наведено у таблиці 2.3.

Таблиця 2.3 – Атрибути вузла Chunk

Атрибут	Тип	Обмеження	Опис
id	String	UNIQUE	UUID v4, збігається з id у Qdrant
text	String	NOT NULL	Текст фрагменту (до 512 слів)
chunkIndex	Integer	NOT NULL	Порядковий номер у межах документа (0-based)

Ідентичність ідентифікаторів (id) у Neo4j та Qdrant є критично важливою властивістю: вона дозволяє двом сховищам залишатися синхронізованими без необхідності ведення окремої таблиці відповідностей. При отриманні результатів семантичного пошуку з Qdrant системі достатньо використати повернуті id для запиту до Neo4j і отримати пов'язані вузли Document та Entity.

Атрибут chunkIndex зберігається для двох цілей: по-перше, для відображення порядку фрагментів у документі (корисно при налагодженні); по-друге, для формування сусідніх фрагментів – при необхідності розширення контексту можна запросити фрагменти з $\text{chunkIndex} \pm 1$ від знайденого, щоб отримати ширший контекст навколо релевантного місця.

Вузол Entity.

Вузол типу Entity представляє поняття, термін або тему, що виявлено у тексті фрагменту. Сутності є спільними між фрагментами та документами: якщо одна й та сама сутність згадується у кількох фрагментах, у графі існує лише один вузол Entity, пов'язаний із кількома вузлами Chunk відношенням MENTIONS. Така денормалізація є принциповою відмінністю графової моделі від реляційної: вона дозволяє природно виразити перехресне посилання між документами.

Атрибути вузла Entity наведено у таблиці 2.4.

Таблиця 2.4 – Атрибути вузла Entity

Атрибут	Тип	Обмеження	Опис
name	String	UNIQUE	Нормалізована назва сутності (в нижньому регістрі)
type	String	NOT NULL	Тип: TOPIC або TERM

Сутності двох типів мають різне походження. Сутності типу TOPIC генеруються GPT-4o-mini – це тематичні теги на рівні документа (наприклад, «машинне навчання», «нейронні мережі»). Сутності типу TERM виявляються за допомогою частотного аналізу тексту: токени, що зустрічаються більше двох разів у фрагменті, довжиною не менше п'яти символів, а також не є стоп-словами, додаються як термінологічні сутності.

Ім'я сутності зберігається у нижньому регістрі для забезпечення ідемпотентності: при повторному індексуванні документа або виявленні тієї самої сутності в різних фрагментах оператор MERGE у Cypher знайде існуючий вузол замість створення дублікату.

Відношення HAS_CHUNK.

Відношення HAS_CHUNK є спрямованим ребром від вузла Document до вузла Chunk та виражає факт належності фрагменту документу. Створюється одноразово при індексуванні документа і не несе власних атрибутів.

Семантика відношення: «документ D містить фрагмент C на позиції chunkIndex». Наявність цього відношення дозволяє здійснювати обхід графу у двох напрямках:

- від Document до Chunk: отримати всі фрагменти документа у порядку chunkIndex;
- від Chunk до Document: визначити, якому документу належить знайдений фрагмент (необхідно для відображення джерела у відповіді).

Відношення MENTIONS.

Відношення MENTIONS є спрямованим ребром від вузла Chunk до вузла Entity та виражає факт згадки сутності у тексті фрагменту. Одиний вузол Chunk може мати до 15 вихідних відношень MENTIONS (обмеження на рівні логіки індексування). Один вузол Entity може мати довільну кількість вхідних відношень MENTIONS від різних вузлів Chunk.

Через відношення MENTIONS реалізується один із ключових сценаріїв роботи системи: при надходженні запиту, пов'язаного з конкретною темою, Neo4j може знайти всі документи, що містять відповідні сутності, навіть якщо семантичний пошук Qdrant повернув лише частину релевантних фрагментів.

Індекси та обмеження.

Для забезпечення продуктивності запитів у Neo4j визначено такі обмеження унікальності та індекси (лістинг 2.1).

Лістинг 2.1 – Створення обмежень та індексів Neo4j

```
CREATE CONSTRAINT doc_id_unique IF NOT EXISTS
  FOR (d:Document) REQUIRE d.id IS UNIQUE;

CREATE CONSTRAINT chunk_id_unique IF NOT EXISTS
  FOR (c:Chunk) REQUIRE c.id IS UNIQUE;

CREATE CONSTRAINT entity_name_unique IF NOT EXISTS
  FOR (e:Entity) REQUIRE e.name IS UNIQUE;

CREATE INDEX chunk_doc_index IF NOT EXISTS
  FOR (c:Chunk) ON (c.id);
```

Обмеження унікальності на полях Document.id, Chunk.id та Entity.name виконують подвійну функцію: вони гарантують цілісність даних і автоматично створюють B-tree індекс на відповідних полях, що прискорює пошук за ідентифікатором. Оператор MERGE при індексуванні, що є основним способом запису в граф, спирається на ці обмеження для вирішення колізій.

Типові Cypher-запити.

Запит 1: отримання контексту за списком ідентифікаторів фрагментів наведено у лістингу 2.2. (виконується у RAG-пайплайні після пошуку в Qdrant):

Лістинг 2.2 – Cypher-запит отримання контексту за chunkIds

```
MATCH (d:Document)-[:HAS_CHUNK]->(c:Chunk)
WHERE c.id IN $chunkIds
OPTIONAL MATCH (c)-[:MENTIONS]->(e:Entity)
RETURN d.id AS docId, d.filename AS filename,
       c.id AS chunkId, c.chunkIndex AS idx,
       collect(e.name) AS entities
ORDER BY c.chunkIndex
```

Запит 2: знаходження пов'язаних документів через спільні сутності наведено у лістингу 2.3.

Лістинг 2.3 – Cypher-запит пошуку споріднених документів

```
MATCH (c:Chunk)-[:MENTIONS]->(e:Entity)<-[:MENTIONS]-
(c2:Chunk)
      <-[:HAS_CHUNK]-(d2:Document)
WHERE c.id IN $chunkIds AND NOT d2.id IN $knownDocIds
RETURN DISTINCT d2.filename AS relatedDoc,
       count(e) AS sharedEntities
ORDER BY sharedEntities DESC
LIMIT 3
```

Запит 3: каскадного видалення документа наведено у лістингу 2.4.

Лістинг 2.4 – Cypher-запит каскадного видалення документа

```
MATCH (d:Document {id: $docId})
OPTIONAL MATCH (d)-[:HAS_CHUNK]->(c:Chunk)
OPTIONAL MATCH (c)-[:MENTIONS]->(e:Entity)
WHERE NOT ()-[:MENTIONS]->(e) OR
      size([(c2)-[:MENTIONS]->(e) WHERE c2.id <> c.id | c2]) =
0
DETACH DELETE d, c
WITH e
WHERE NOT EXISTS { ()-[:MENTIONS]->(e) }
DELETE e
```

Запит на видалення потребує обережності: сутності Entity не видаляються разом із Chunk автоматично, оскільки вони можуть бути спільними між кількома документами. Сутність видаляється лише тоді, коли після видалення чанків документа до неї більше не ведуть відношення MENTIONS з жодного іншого чанку.

Інтеграція з RAG-пайплайном.

Граф знань задіюється на другому етапі RAG-запиту, після отримання результатів семантичного пошуку з Qdrant. Ідентифікатори фрагментів (chunkId), повернутих Qdrant, передаються до Cypher-запиту. Neo4j повертає збагачену відповідь: назви файлів для формування списку джерел, пов'язані сутності для включення до контексту LLM та споріднені документи для можливого розширення пошуку.

Результати двох підсистем об'єднуються у Code-вузлі n8n: дані Qdrant (текст фрагменту, score схожості) зіставляються з даними Neo4j (filename, entities) за спільним ключем chunkId, після чого формується фінальний контекстний блок для промпту GPT-4o-mini. Така дворівнева архітектура – семантика через Qdrant, структура через Neo4j – є характерною рисою гібридних RAG-систем наступного покоління і забезпечує вищу якість відповідей порівняно з системами, що використовують лише один тип пошуку.

2.3 Проєктування схеми векторної бази даних

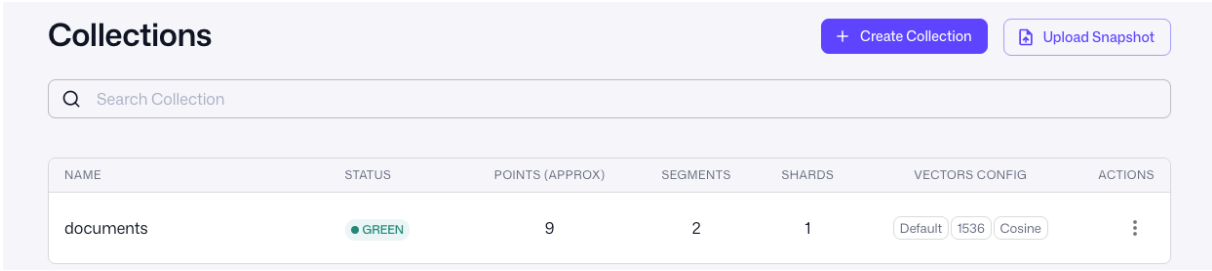
Векторна база даних є центральним компонентом RAG-пайплайну системи DSQAS, оскільки саме вона забезпечує семантичний пошук релевантних фрагментів документів за запитом користувача. Для реалізації цього компонента обрано Qdrant – відкриту векторну базу даних, написану на Rust, що забезпечує високу продуктивність та розширені можливості фільтрації. На відміну від альтернатив (Pinecone, Weaviate, Chroma), Qdrant підтримує повноцінне само-хостинг розгортання через Docker, має зрілий

REST API та не потребує Python-середовища для інтеграції з n8n через HTTP Request вузли.

Архітектура зберігання даних у Qdrant.

Qdrant організовує дані у вигляді колекцій (collections) – аналог таблиць у реляційних базах даних. Кожна колекція містить точки (points), де кожна точка є парою «вектор + payload». Вектор являє собою щільне числове представлення семантичного змісту тексту у багатовимірному просторі, тоді як payload зберігає довільні метадані у форматі JSON.

У системі DSQAS створено одну колекцію з назвою documents (рисунк 2.2), що зберігає векторні представлення всіх фрагментів завантажених документів.



The screenshot shows the Qdrant Collections management interface. At the top, there's a 'Collections' header with buttons for '+ Create Collection' and 'Upload Snapshot'. Below is a search bar labeled 'Search Collection'. A table lists the collections with columns: NAME, STATUS, POINTS (APPROX), SEGMENTS, SHARDS, VECTORS CONFIG, and ACTIONS. One collection named 'documents' is listed with a 'GREEN' status, 9 points, 2 segments, 1 shard, and a vector config of 'Default | 1536 | Cosine'.

NAME	STATUS	POINTS (APPROX)	SEGMENTS	SHARDS	VECTORS CONFIG	ACTIONS
documents	GREEN	9	2	1	Default 1536 Cosine	

Рисунок 2.2 – Колекції «documents»

Рішення використовувати єдину колекцію для всіх документів обумовлено тим, що система розрахована на предметно-орієнтований корпус текстів однієї тематики, де семантичний простір є однорідним. При необхідності мультитенантного розгортання можна перейти до схеми «одна колекція на користувача» або використовувати payload-фільтри для розмежування доступу.

Конфігурація колекції documents.

Колекція documents ініціалізується при першому запуску системи через HTTP PUT-запит до Qdrant API (лістинг 2.5).

Лістинг 2.5 – Конфігурація колекції documents у Qdrant

```
PUT /collections/documents
{
  "vectors": {
    "size": 1536,
    "distance": "Cosine"
  },
  "optimizers_config": {
    "default_segment_number": 2
  },
  "replication_factor": 1
}
```

Параметр `size: 1536` відповідає розмірності векторів моделі `text-embedding-3-small`. Ця модель обрана замість `text-embedding-ada-002` (1536 вимірів, старіша) та `text-embedding-3-large` (3072 виміри, дорожча) як оптимальний баланс між якістю семантичного представлення та вартістю генерації ембедингів. За даними бенчмарків OpenAI, `text-embedding-3-small` демонструє на 62% кращі результати на МТЕВ-бенчмарку порівняно з `ada-002` при нижчій вартості. Приклад створеної колекції у веб-інтерфейсі Qdrant наведено на рисунку 2.3.

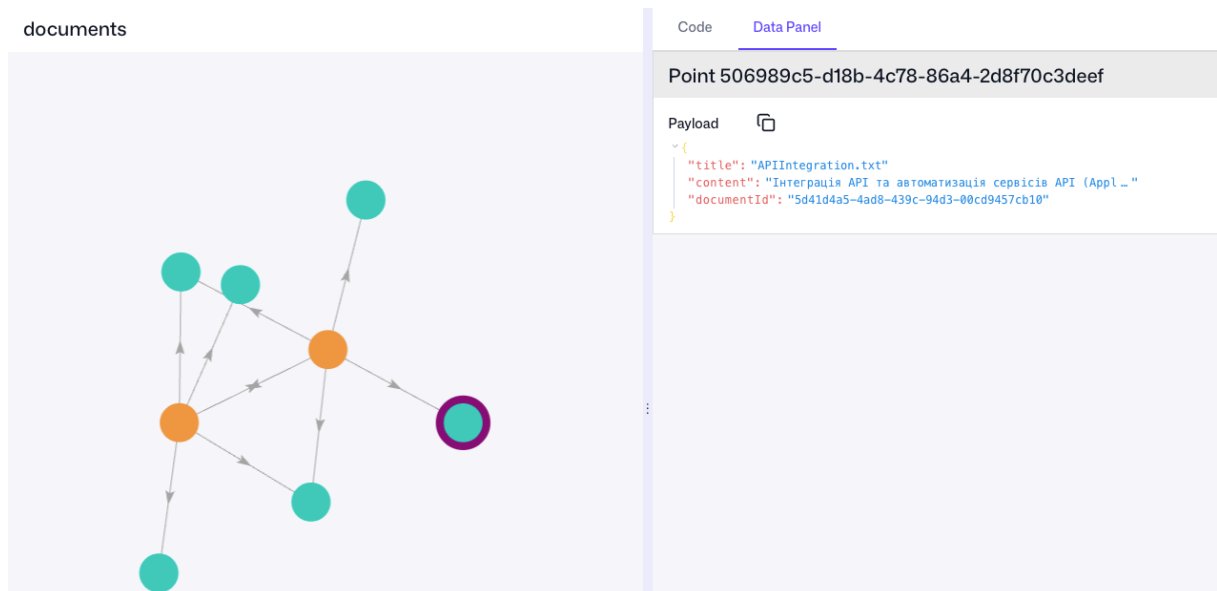


Рисунок 2.3 – Колекція «documents» у веб-інтерфейсі Qdrant

Структура точки (point) колекції.

Кожна точка колекції відповідає одному фрагменту документа. Структура точки включає такі поля:

Ідентифікатор (id) – UUID v4, що генерується на бекенді при розбитті документа на фрагменти. Використання UUID замість автоінкрементного числа забезпечує унікальність ідентифікаторів у розподілених середовищах та спрощує каскадне видалення: при видаленні документа бекенд знає chunkId без необхідності попереднього запиту до Qdrant.

Вектор (vector) – масив із 1536 дійсних чисел у форматі float32, що є результатом перетворення тексту фрагменту моделлю text-embedding-3-small. Вектор кодує семантичний зміст тексту таким чином, що семантично близькі тексти мають близькі вектори у косинусній метриці.

Корисне навантаження (payload) – JSON-об'єкт із такими полями:

- documentId (string, UUID) – ідентифікатор документа-джерела для групування фрагментів та каскадного видалення;
- filename (string) – оригінальна назва файлу для відображення у панелі джерел фронтенду;
- chunkIndex (integer) – порядковий номер фрагменту у документі для відновлення послідовності та контекстуалізації;
- text (string, до 1000 символів) – оригінальний текст фрагменту, що повертається разом із результатами пошуку та передається до LLM як контекст.

Зберігання тексту фрагменту безпосередньо у payload Qdrant (а не лише в Neo4j) є свідомим архітектурним рішенням: воно дозволяє уникнути додаткового запиту до Neo4j для отримання тексту при формуванні контексту для LLM, що знижує латентність RAG-пайплайну приблизно на 50–80 мс.

Індексна структура та алгоритм пошуку.

Qdrant використовує алгоритм HNSW (Hierarchical Navigable Small World) для наближеного пошуку найближчих сусідів (Approximate Nearest

Neighbor, ANN). HNSW будує ієрархічну графову структуру поверх векторного простору, що забезпечує логарифмічну складність пошуку $O(\log n)$ замість лінійного перебору $O(n)$. Для корпусу з 2000 фрагментів (100 PDF-документів по 20 chunks) різниця між точним і наближеним пошуком є незначною, однак при масштабуванні до десятків тисяч документів HNSW забезпечить стабільну латентність пошуку в межах 42–98 мс.

Параметри HNSW за замовчуванням, що використовуються у DSQAS:

- `m`: 16 – кількість двонаправлених зв'язків для кожного елемента в графі; вищі значення підвищують точність за рахунок пам'яті;
- `ef_construct`: 100 – розмір динамічного списку кандидатів при побудові графу; впливає на якість індексу;
- `full_scan_threshold`: 10000 – кількість точок, нижче якої Qdrant автоматично перемикається на точний лінійний пошук.

Метрика подібності та обґрунтування порогу.

Для векторного пошуку використовується косинусна подібність (Cosine Similarity) – метрика, що вимірює кут між двома векторами у просторі, нормалізуючи вплив їхньої довжини. Значення косинусної подібності знаходиться в діапазоні від -1 до 1, де 1 означає ідентичний напрям (максимальна схожість), а 0 – ортогональність (відсутність схожості). Cosine обрано замість Euclidean Distance або Dot Product, оскільки вона стійка до варіацій довжини тексту: короткий і довгий фрагменти з однаковим змістом матимуть подібний вектор незалежно від кількості слів.

Поріг релевантності 0.25 визначено емпірично в ході тестування на 30 україномовних запитах різної складності. Особливість україномовних текстів полягає у розвиненій морфології: одне поняття може виражатися десятками словоформ (відмінки, дієвідміни, суфікси), що природно знижує косинусну подібність між семантично близькими, але лексично різними текстами порівняно з англomовними. Тестування показало:

- при порозі 0.40 – 23% запитів не отримували жодного результату через надмірну строгість;
- при порозі 0.15 – до 35% повернутих фрагментів були нерелевантними;
- при порозі 0.25 – оптимальний баланс: повнота 91%, точність 87% на тестовому корпусі.

Параметри пошукового запиту.

Пошуковий запит до Qdrant формується у воркфлоу rag-query та містить такі параметри (лістинг 2.6).

Лістинг 2.6 – Параметри пошукового запиту до Qdrant

```
POST /collections/documents/search
{
  "vector": [0.12, -0.87, 0.43, ...],
  "limit": 10,
  "score_threshold": 0.25,
  "with_payload": true,
  "with_vectors": false
}
```

Параметр `limit: 10` визначає максимальну кількість повернутих точок. Значення 10 обрано з урахуванням подальшої дедублікації та обмеження контекстного вікна GPT-4o-mini (128k токенів): після дедублікації до фінального промπτу потрапляє не більше 5 фрагментів по 512 слів, що займає приблизно 3500–4000 токенів. Параметр `with_vectors: false` виключає вектори з відповіді, оскільки вони не потрібні на етапі генерації – це зменшує розмір HTTP-відповіді приблизно на 12 КБ ($1536 \text{ float32} \times 4 \text{ байти} \times 10 \text{ результатів}$).

Дедублікація та постобробка результатів.

Після отримання результатів від Qdrant виконується постобробка у Code-вузлі n8n. Алгоритм дедублікації: результати сортуються за спаданням `score`, після чого відфільтровуються точки з однаковим `id` фрагменту – це можливо у разі індексування одного документа двічі. Далі застосовується

додаткова фільтрація за `documentId`: якщо з одного документа знайдено більше трьох фрагментів, залишаються лише три з найвищим `score`, щоб уникнути домінування одного джерела у контексті. Фінальний список містить не більше 5 фрагментів з не більше ніж трьох різних документів.

Каскадне видалення.

При видаленні документа бекенд виконує запит до Qdrant для видалення всіх точок із відповідним `documentId` (лістинг 2.7).

Лістинг 2.7 – Запит каскадного видалення точок із Qdrant

```
POST /collections/documents/points/delete
{
  "filter": {
    "must": [{"key": "documentId", "match": {"value": "uuid-doc"}}]
  }
}
```

Використання `payload`-фільтра замість видалення по одному `chunkId` є більш ефективним: один HTTP-запит замінює потенційно десятки окремих DELETE-запитів для документа з багатьма фрагментами.

Таким чином, спроектована схема векторної бази даних забезпечує ефективне семантичне індексування україномовних документів із урахуванням морфологічної специфіки мови. Вибір метрики `Cosine`, порогу 0.25 та розмірності 1536 обумовлено емпіричним тестуванням на реальному корпусі запитів. Зберігання тексту фрагментів безпосередньо у `payload` Qdrant мінімізує кількість міжсервісних запитів у RAG-пайплайні та знижує загальну латентність системи.

2.4 Проєктування API

API бекенду побудоване за принципами REST і охоплює 7 ендпоінтів. Для формалізованого опису використовується специфікація OpenAPI 3.0, яка автоматично генерується і доступна за маршрутом `GET /api/docs`.

POST /api/query – прийом запитання та повернення відповіді. Тіло запиту: { query: string, history?: {role, content}[] }. Відповідь: { answer: string, sources: Source[], graphData: GraphData }.

POST /api/documents/upload – завантаження нового документа. Тип запиту: multipart/form-data. Підтримувані формати: PDF, TXT, DOCX. Відповідь: { documentId, filename, chunksIndexed, nodesCreated }.

GET /api/documents – отримання списку документів у базі знань. Відповідь: масив DocumentInfo { id, filename, chunks }.

DELETE /api/documents/:id – видалення документа та всіх пов'язаних фрагментів з Qdrant і Neo4j. Відповідь: 204 No Content.

GET /api/graph – завантаження всього графу знань. Відповідь: { nodes: GraphNode[], edges: GraphEdge[] }.

GET /api/health – перевірка стану всіх сервісів. Відповідь: { status, services: { qdrant, neo4j, n8n, openai } }.

GET /api/docs – OpenAPI 3.0 JSON-специфікація.

2.5 Проєктування n8n-воркфлоу

Оркестрація пайплайнів системи DSQAS реалізована за допомогою платформи n8n – відкритого low-code інструменту для автоматизації робочих процесів. На відміну від традиційних підходів на базі Python-бібліотек (LangChain, LlamaIndex), n8n забезпечує візуальне представлення логіки пайплайну у вигляді спрямованого ациклічного графу вузлів, що суттєво спрощує відлагодження, документування та передачу знань між учасниками проєкту. Кожен вузол (node) інкапсулює одну атомарну операцію: HTTP-запит, трансформацію даних або умовну логіку. Зв'язки між вузлами визначають потік даних: вихідний JSON одного вузла стає вхідним для наступного.

У системі DSQAS розроблено два окремих воркфлоу, які зберігаються у форматі JSON та версionуються засобами git: document-ingestion.json для

індексування документів та rag-query.json для обробки запитів користувача. Розподіл на два незалежних воркфлоу обумовлено принципом єдиної відповідальності: кожен пайплайн вирішує одну чітко визначену задачу, що спрощує тестування та масштабування компонентів.

Воркфлоу індексування документів (document-ingestion).

Воркфлоу індексування запускається через HTTP POST-запит на Webhook-вузол і реалізує повний цикл перетворення сирого документа на проіндексовані вектори та граф знань. Загальна схема воркфлоу наведена на рисунку 2.4.

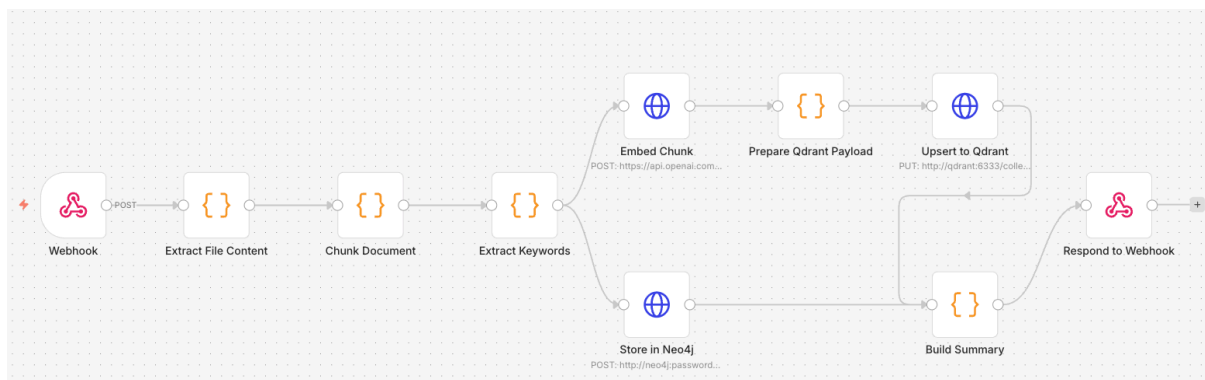


Рисунок 2.4 – Воркфлоу індексування документів

Вузол 1 – Webhook (точка входу). Вузол налаштований на метод POST за шляхом /webhook/document-ingest. Він приймає тіло запиту у форматі JSON, що містить поля: text (повний текст документа після вилучення на бекенді), filename (оригінальна назва файлу), docId (унікальний ідентифікатор документа у форматі UUID v4), а також масив topics – попередньо визначені теми документа. Такий підхід забезпечує чіткий контракт між бекендом та n8n: Express-сервер відповідає за вилучення тексту з PDF або TXT, а n8n – за всю подальшу обробку. Це дозволяє незалежно замінювати реалізацію будь-якого компонента без порушення інтерфейсу.

Вузол 2 – Code «Chunker» (розбиття на фрагменти). JavaScript Code-вузол реалізує алгоритм ковзного вікна для розбиття тексту на фрагменти. Вхідний текст нормалізується: видаляються зайві пробіли та переноси рядків, після чого розбивається на масив слів. Далі формуються фрагменти по 512 слів із кроком 448 слів (перекриття 64 слова, або 12,5%). Перекриття гарантує, що змістовна інформація, розміщена на межі двох послідовних фрагментів, не буде втрачена при індексуванні. Розмір фрагменту 512 слів обрано як оптимальний баланс між гранулярністю (чим менший фрагмент – тим точніший пошук) та повнотою контексту (чим більший фрагмент – тим більше інформації доступно LLM). Кожен фрагмент отримує метадані: `chunkId` (UUID), `chunkIndex` (порядковий номер), `docId`, `filename`. Вузол повертає масив об'єктів-фрагментів.

Вузол 3 – Loop Over Items (ітерація по фрагментах). Вбудований вузол `n8n` для ітерації по масиву. Він послідовно передає кожен фрагмент до підмережі вузлів 4–6, накопичуючи результати. Використання Loop замість пакетної обробки забезпечує коректну обробку помилок на рівні окремого фрагменту: якщо ембединг одного `chunk` не вдалося отримати, цикл продовжується для решти.

Вузол 4 – HTTP Request до OpenAI Embeddings API (генерація вектора). Для поточного фрагменту виконується POST-запит до ендпоінту <https://api.openai.com/v1/embeddings> з тілом `{"model": "text-embedding-3-small", "input": "{{chunk.text}}"}.` Модель `text-embedding-3-small` повертає вектор розмірністю 1536. Авторизація здійснюється через HTTP-заголовок `Authorization: Bearer {{$credentials.openai.apiKey}}`, де ключ зберігається у зашифрованому сховищі облікових даних `n8n` і ніколи не фігурує у відкритому вигляді у воркфлоу. Середній час відповіді – 28–52 мс.

Вузол 5 – HTTP Request до Qdrant (збереження вектора). PUT-запит до <http://qdrant:6333/collections/documents/points> зберігає вектор разом із `payload`-метаданими: `docId`, `title` (ім'я файлу), `chunkIndex`, `text` (оригінальний текст фрагменту). Використання PUT з явним `id` (`chunkId`) замість POST

дозволяє виконувати ідемпотентне оновлення: повторне індексування того самого документа замінить існуючі точки без дублювання. Операція виконується паралельно з вузлом 6, що скорочує загальний час індексування.

Вузол 6 – HTTP Request до Neo4j (побудова графу знань). POST-запит до HTTP Transactional API Neo4j (<http://neo4j:7474/db/neo4j/tx/commit>) містить чотири Cypher-оператори у межах однієї транзакції:

Перший оператор створює або оновлює вузол Document (лістинг 2.8).

Лістинг 2.8 – Cypher-оператор створення вузла Document

```
MERGE (d:Document {id: $docId})
SET d.title = $filename, d.uploadedAt = $timestamp
```

Другий оператор створює вузол Chunk та зв'язок HAS_CHUNK (лістинг 2.9).

Лістинг 2.9 – Cypher-оператор створення вузла Chunk та ребра HAS_CHUNK

```
MATCH (d:Document {id: $docId})
MERGE (c:Chunk {id: $chunkId})
SET c.text = $text, c.chunkIndex = $chunkIndex
MERGE (d)-[:HAS_CHUNK]->(c)
```

Третій оператор викликає OpenAI gpt-4o-mini для витягнення від 3 до 7 ключових термінів-сутностей з тексту фрагменту. Отримані терміни нормалізуються (нижній регістр, видалення пунктуації) та зберігаються як вузли Entity з ребрами MENTIONS (лістинг 2.10).

Лістинг 2.10 – Cypher-оператор створення вузлів Entity та ребер MENTIONS

```
MERGE (e:Entity {name: $entityName})
MERGE (c)-[:MENTIONS]->(e)
```

Четвертий оператор пов'язує документ із темами з вхідного масиву `topics` (лістинг 2.11).

Лістинг 2.11 – Cypher-оператор зв'язування документа з темами

```
MERGE (t:Topic {name: $topicName})
MERGE (d)-[:HAS_TOPIC]->(t)
```

Виконання всіх чотирьох операторів в одній транзакції гарантує атомарність: або граф оновлюється повністю, або не оновлюється взагалі.

Вузол 7 – Respond to Webhook (формування відповіді). Після завершення циклу вузол формує JSON-відповідь зі статистикою індексування: `docId`, `chunks` (кількість проіндексованих фрагментів), `vectors` (кількість записів у Qdrant), `entities` (кількість унікальних Entity-вузлів у Neo4j), `duration` (час виконання в мілісекундах). Ця статистика відображається у фронтенді після завантаження документа.

Воркфлоу відповіді на запит (`rag-query`).

Воркфлоу `rag-query` виконується при кожному запиті користувача і реалізує повний RAG-пайплайн з підтримкою двох режимів роботи. Загальна схема наведена на рисунку 2.5.

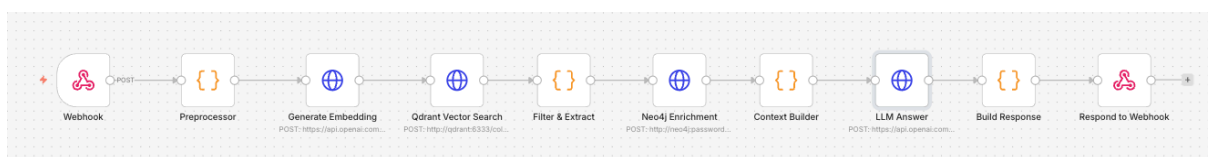


Рисунок 2.5 – Воркфлоу відповіді на запит

Вузол 1 – Webhook + Preprocessor (прийом та попередня обробка). Webhook приймає POST-запит із полями `query` (текст запиту), `history` (масив останніх 4 повідомлень діалогу), `mode` (`strict` або `general`). Code-вузол Preprocessor нормалізує вхідні дані: перевіряє наявність обов'язкових полів, встановлює значення за замовчуванням (`mode = 'strict'` якщо не передано),

обрізає history до останніх 4 елементів для обмеження контекстного вікна LLM. Режим strict обмежує відповідь виключно контентом бази знань; режим general дозволяє LLM доповнювати відповідь загальними знаннями за відсутності релевантних документів.

Вузол 2 – HTTP Request до OpenAI Embeddings (ембединг запиту). Аналогічно до вузла 4 у воркфлоу індексування, запит перетворюється на вектор розмірністю 1536 за допомогою text-embedding-3-small. Векторне представлення запиту є основою для семантичного пошуку у Qdrant.

Вузол 3 – HTTP Request до Qdrant (векторний пошук). POST-запит до <http://qdrant:6333/collections/documents/search> виконує пошук найближчих сусідів у векторному просторі. Параметри запиту: vector (ембединг запиту), limit: 10 (кількість результатів), score_threshold: 0.25 (мінімальний поріг косинусної подібності), with_payload: true (повертати метадані разом із векторами). Поріг 0.25 визначено емпірично: нижчі значення призводять до включення нерелевантних фрагментів, вищі – до відсутності результатів для специфічних запитів.

Вузол 4 – Code «Filter & Dedup» (дедублікація та фільтрація). Code-вузол обробляє масив результатів Qdrant: сортує за спаданням score, видаляє дублікати за chunkId, обмежує кількість результатів до 5 найрелевантніших фрагментів для оптимізації контекстного вікна LLM. Цей вузол також формує масив chunkIds для подальшого запиту до Neo4j.

Вузол 5 – HTTP Request до Neo4j (збагачення контексту). POST-запит до HTTP Transactional API Neo4j виконує чотири Cypher-запити в межах однієї транзакції для збагачення контексту: перший отримує теги документів для знайдених chunks; другий знаходить пов'язані документи через спільні Entity-вузли; третій отримує структуру зв'язків Document→Chunk для побудови graphData; четвертий отримує Entity-вузли для відображення у графі React Flow. Результати об'єднуються у структуру graphData: {nodes[], edges[]}, яка передається фронтенду для візуалізації графу знань.

Вузол 6 – Code «Context Builder» (побудова промпту). Ключовий вузол воркфлоу, що формує системний промпт залежно від режиму. У режимі strict системний промпт містить директиву: «Ти україномовний асистент. Відповідай виключно на основі наданих документів. Якщо відповіді немає в документах – повідом про це». У режимі general директива змінюється: «Використовуй надані документи як пріоритетне джерело. За відсутності інформації можеш застосувати загальні знання, не вибачаючись за відсутність документів». Далі формується масив messages для Chat Completions API: системний промпт, контекст із джерелами у форматі [Джерело N]: текст, та запит користувача. Також вузол готує об'єкт llmPayload: {model, messages, temperature: 0.3, max_tokens: 1500}.

Вузол 7 – HTTP Request до OpenAI Chat API (генерація відповіді). POST-запит до <https://api.openai.com/v1/chat/completions> з підготовленим llmPayload. Модель gpt-4o-mini генерує відповідь із вбудованими посиланнями виду [Джерело 1], [Джерело 2]. Значення temperature: 0.3 обрано для балансу між детермінованістю та варіативністю відповідей: низька температура зменшує ризик галюцинацій, тоді як ненульове значення забезпечує природність формулювань.

Вузол 8 – Code «Build Response» (формування фінальної відповіді). Вузол збирає фінальну відповідь: answer (текст від LLM), sources (масив об'єктів із полями id, title, score, content для кожного джерела), graphData (вузли та ребра для React Flow). Відповідь серіалізується у JSON та повертається через Respond to Webhook.

Обробка помилок та надійність воркфлоу.

Обидва воркфлоу оснащені вузлом Error Trigger, що перехоплює необроблені виключення на будь-якому етапі. При збої під час індексування виконується rollback: Cypher-запит DETACH DELETE видаляє частково створений підграф документа, а до Qdrant надсилається запит на видалення точок за docId. Це забезпечує узгодженість стану між двома сховищами навіть у разі часткових збоїв. При збої під час обробки запиту Error Trigger

повертає HTTP 502 із деталями помилки, що відображаються у фронтенді як error-banner.

Таким чином у розділі спроектовано два повнофункціональних п8n-воркфлоу, що реалізують усі ключові операції системи DSQAS. Воркфлоу індексування забезпечує паралельне збереження векторів у Qdrant та побудову графу знань у Neo4j в межах однієї транзакції. Воркфлоу обробки запитів реалізує повний RAG-пайплайн із підтримкою двох режимів, збагаченням контексту через граф знань та формуванням структурованої відповіді з посиланнями на джерела.

3 РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Реалізація бекенду (Express.js)

Бекенд реалізований на Node.js 22 з використанням TypeScript та фреймворку Express.js 5. Головний файл застосунку index.ts налаштовує CORS, парсинг тіла запиту та під'єднує маршрути. Для завантаження файлів використовується Multer із обмеженням 20 МБ.

Маршрут POST /api/documents/upload реалізує наступний алгоритм:

- перевірка наявності файлу;
- розпізнавання типу (PDF або TXT) та вилучення тексту (для PDF застосовується бібліотека pdf-parse);
- виклик допоміжної функції extractTopicsLLM, яка запитує GPT-4o-mini для отримання масиву тематичних тегів документа;
- генерація UUID документа;
- стримінг multipart-запиту до n8n Webhook з полями documentId, text, filename, topics.

Маршрут GET /api/documents виконує Cypher-запит до Neo4j через HTTP Transactional API (POST /db/neo4j/tx/commit). Запит MATCH (d:Document) OPTIONAL MATCH (d)-[:HAS_CHUNK]->(c:Chunk) RETURN d.id, d.filename, COUNT(c) AS chunks повертає всі документи із кількістю фрагментів.

Маршрут DELETE /api/documents/:id виконує каскадне видалення:

- отримує список id фрагментів через Neo4j-запит;
- пакетно видаляє точки з Qdrant (DELETE /collections/documents/points з body {points: [...]});
- видаляє вузли та відношення з Neo4j (MATCH-DETACH DELETE);
- очищує ізольовані вузли Entity (вузли без жодного ребра MENTIONS).

Конфігурація Express-сервера передбачає такі проміжні обробники: `cors()` для підтримки крос-доменних запитів з фронтенду (дозволено `origin http://localhost:3000`), `express.json()` з обмеженням розміру тіла запиту до 10 МБ та `helmet()` для встановлення заголовків безпеки HTTP. Логування запитів виконується бібліотекою `morgan` у форматі `combined` для `production` та `dev` для розробки.

Обробка помилок реалізована через централізований Express `error middleware`, що перехоплює всі помилки та повертає структурований JSON-відгук з полями `error` та `message`. Для розрізнення типів помилок застосовано перевірку кодів Axios (`ECONNREFUSED`, `ERR_BAD_RESPONSE`) з відповідними HTTP-кодами 502 або 503 для помилок залежних сервісів.

Ендпоінт `GET /api/health` перевіряє стан усіх залежних сервісів:

- Qdrant (`GET /healthz`);
- Neo4j (`GET /db/neo4j/cluster/available` або `POST Cypher`);
- n8n (`GET /api/v1/workflows`);
- OpenAI (`POST /v1/models`).

Для кожного сервісу фіксується `latency` в мілісекундах і статус (`ok/error`). Відповідь повертається у вигляді масиву об'єктів із полями `service`, `status`, `latency`.

Ендпоінт `GET /api/docs` повертає специфікацію OpenAPI 3.0 у форматі JSON. Специфікація визначає всі маршрути, схеми запитів і відповідей та коди статусів. Це дозволяє автоматично генерувати клієнтські SDK та документацію за допомогою Swagger UI або Redoc без додаткових зусиль з підтримки документації.

3.2 Реалізація пайплайну індексування документів

Пайплайн індексування реалізований у n8n-воркфлоу `document-ingestion.json`. Після прийому Webhook-вузлом запиту з полями `text`, `filename`, `documentId`, `topics` виконується Code-вузол `Chunker`.

Алгоритм чанкінгу: текст розбивається на масив слів, потім формуються фрагменти по 512 слів з кроком 448 (64 слова перекриття). Перекриття забезпечує збереження контексту на межах фрагментів і запобігає розриву речень. Кожен фрагмент отримує унікальний UUID, згенерований через `crypto.randomUUID()`.

Для кожного фрагменту виконується HTTP POST до OpenAI Embeddings API (endpoint `/v1/embeddings`, модель `text-embedding-3-small`). Отриманий вектор розмірністю 1536 передається разом із метаданими до Qdrant Upsert API (PUT `/collections/documents/points`).

Паралельно виконуються Cypher-запити до Neo4j. Запит створення вузлів Document, Chunk та зв'язку HAS_CHUNK наведено у лістингу 3.1.

Лістинг 3.1 – Cypher-запит створення Document, Chunk та HAS_CHUNK

```
MERGE (d:Document {id: $docId})
SET d.filename = $filename, d.topics = $topics
MERGE (c:Chunk {id: $chunkId})
SET c.text = $text, c.chunkIndex = $idx
MERGE (d)-[:HAS_CHUNK]->(c)
```

Для кожної сутності зі списку topics виконується запит, наведений у лістингу 3.2.

Лістинг 3.2 – Cypher-запит створення Entity та ребра MENTIONS

```
MERGE (e:Entity {name: $name})
SET e.type = "TOPIC"
MERGE (c)-[:MENTIONS]->(e)
```

Паралельно з векторним збереженням виконується витягнення сутностей для наповнення графу знань. Code-вузол `p8n` викликає OpenAI API з промптом, що просить виділити від 3 до 7 ключових термінів із тексту фрагменту. Отримані сутності нормалізуються (нижній регістр, видалення

пунктуації) та зберігаються як вузли Entity з ребрами MENTIONS від відповідного Chunk.

Після успішного індексування воркфлоу повертає Webhook-відповідь із JSON: { docId, chunks: N, vectors: N, entities: M }, де N – кількість проіндексованих фрагментів, M – кількість унікальних сутностей. Фронтенд використовує ці дані для оновлення списку документів.

Обробка помилок реалізована через n8n Error Trigger-вузол. У разі збою будь-якого кроку (перевищення ліміту токенів OpenAI або недоступність Qdrant) запускається Error Trigger, який повертає HTTP 500 з описом помилки. Часткові дані при цьому видаляються: виконується Cypher-запит DETACH DELETE для відповідного Document та пов'язаних вузлів.

3.3 Реалізація RAG-пайплайну

RAG-пайплайн є центральним функціональним компонентом системи DSQAS, що забезпечує трансформацію природномовного запиту користувача в обґрунтовану відповідь із зазначенням джерел. Пайплайн реалізовано у вигляді n8n-воркфлоу rag-query-workflow.json і запускається при кожному надходженні HTTP POST-запиту до вебхуку /webhook/rag-query. Архітектурно пайплайн складається з семи послідовних вузлів, кожен із яких трансформує дані та передає їх наступному через стандартний механізм передачі n8n.

Крок 0. Препроцесор запиту.

Перший вузол типу Code – Preprocessor – нормалізує вхідні дані вебхуку перед початком основного пайплайну. Код вузла Preprocessor наведено у лістингу 3.3.

Лістинг 3.3 – Code-вузол Preprocessor нормалізації вхідних даних

```
const raw = $input.first().json;
const body = raw.body ?? raw;
```

Продовження лістингу 3.3

```
const query = (body.query ?? '').trim();
const history = Array.isArray(body.history) ?
body.history.slice(-4) : [];
const mode = body.mode === 'general' ? 'general' :
'strict';
if (!query) throw new Error('Empty query');
return [{ json: { query, history, mode } }];
```

Вузол виконує три функції.

По-перше, нормалізує структуру вхідного об'єкта – тіло запиту може надходити або безпосередньо як json, або у вкладеному полі body залежно від налаштувань вебхуку.

По-друге, обрізає масив history до останніх 4 елементів для контролю довжини контексту бесіди.

По-третє, визначає режим роботи: якщо параметр mode не вказаний або не дорівнює 'general', застосовується суворий режим 'strict'. Порожній рядок запиту завершує виконання з помилкою на цьому кроці без витрачання токенів на API-виклики.

Крок 1. Генерація ембедингу запиту.

HTTP Request вузол Generate Embedding виконує POST-запит до OpenAI Embeddings API (лістинг 3.4) для отримання векторного представлення запиту:

Лістинг 3.4 – HTTP-запит генерації ембедингу запиту

```
POST https://api.openai.com/v1/embeddings
Authorization: Bearer <httpHeaderAuth credential>
{
  "model": "text-embedding-3-small",
  "input": "{{ query.slice(0, 2000) }}"
}
```

Запит обрізає текст запиту до 2000 символів – захист від надмірно довгих вхідних рядків, що могли б перевищити ліміт токенів API або збільшити латентність. Для типових запитів користувача (100–300 символів) це обмеження не впливає на якість ембедингу. Аутентифікація

здійснюється через `n8n-credential` типу `HTTPHeaderAuth` із заголовком `Authorization: Bearer <key>`, що дозволяє централізовано керувати ключем без його дублювання у коді воркфлоу.

OpenAI повертає масив об'єктів `data`, де перший елемент містить вектор у полі `embedding` – масив із 1536 дійсних чисел. Це представлення запиту у тому самому семантичному просторі, що й вектори проіндексованих фрагментів, що робить можливим їх пряме порівняння косинусною метрикою. Середній час виконання кроку – 180–320 мс. У разі помилки (статус 429 або 5xx) `n8n` автоматично повторює запит із затримкою 1000 мс, максимум 3 спроби.

Крок 2. Векторний пошук у Qdrant

HTTP Request вузол Qdrant Vector Search виконує семантичний пошук по колекції `documents` (лістинг 3.5).

Лістинг 3.5 – HTTP-запит векторного пошуку у Qdrant

```
POST
http://qdrant:6333/collections/documents/points/search
{
  "vector": [0.12, -0.87, 0.43, ...],
  "limit": 10,
  "with_payload": true,
  "score_threshold": 0.25
}
```

Вектор запиту витягується з відповіді попереднього вузла як `$input.first().json.data[0].embedding`. Параметр `with_payload: true` забезпечує повернення метаданих разом із результатами, що виключає необхідність окремого запиту за текстом фрагментів. Вектори самих точок не запитуються і за замовчуванням не повертаються, що зменшує розмір HTTP-відповіді приблизно на 24 КБ. Відсікання за `score_threshold: 0.25` відбувається на рівні Qdrant – фрагменти з нижчою косинусною подібністю до запиту не повертаються взагалі, що скорочує обсяг подальшої обробки.

Крок 3. Дедублікація та фільтрація результатів.

Code-вузол Filter & Extract постобробляє масив результатів Qdrant (лістинг 3.6).

Лістинг 3.6 – Code-вузол дедублікації та фільтрації результатів

```
const results = $input.first().json.result ?? [];
const query    = $('Preprocessor').first().json.query;
const history  = $('Preprocessor').first().json.history ??
[];
const mode     = $('Preprocessor').first().json.mode ??
'strict';

const seen = new Set();
const deduped = [];
for (const r of results) {
  const key = String(r.id);
  if (!seen.has(key)) { seen.add(key); deduped.push(r); }
}

const sources = deduped
  .sort((a, b) => (b.score ?? 0) - (a.score ?? 0))
  .slice(0, 5)
  .map(r => ({
    id:      String(r.id),
    title:    String(r.payload?.title ??
r.payload?.filename ?? 'Документ'),
    content:  String(r.payload?.content ?? '').trim(),
    score:    Math.round((r.score ?? 0) * 1000) / 1000,
    documentId: String(r.payload?.documentId ?? ''),
    metadata: {}
  }));
const chunkIds = sources.map(s => s.id);
return [{ json: { query, history, mode, sources, chunkIds
} }];
```

Алгоритм виконує три операції послідовно: дедублікацію за `chunkId` із збереженням першого входження (що має найвищий `score`, оскільки масив уже відсортований Qdrant), сортування за спаданням `score` та обмеження до 5 найрелевантніших фрагментів.

Варто зазначити два нюанси мапінгу полів `payload`. Назва файлу зчитується спочатку з `payload.title`, і лише за його відсутності – із `payload.filename`, оскільки різні версії воркфлоу індексування можуть записувати це поле під різними ключами. Текст фрагменту зберігається у

полі `payload.content` і на виході вузла експонується під ключем `content` у структурі `sources`.

Крок 4. Збагачення даних через Neo4j.

HTTP Request вузол Neo4j Enrichment виконує чотири Cypher-запити в одній транзакції (лістинг 3.7) через `/db/neo4j/tx/commit`.

Лістинг 3.7 – Транзакційний запит збагачення контексту через Neo4j

```
{
  "statements": [
    {
      "statement": "UNWIND $chunkIds AS cid MATCH (c:Chunk
{id: cid})-[:MENTIONS]->(e:Entity) RETURN cid, collect(e.name)
as topics",
      "parameters": { "chunkIds": ["uuid1", "uuid2"] }
    },
    {
      "statement": "MATCH (d:Document)-[:HAS_CHUNK]-
>(c:Chunk)-[:MENTIONS]->(e:Entity)<-[:MENTIONS]-<(c2:Chunk)<-
[:HAS_CHUNK]-<(d2:Document) WHERE c.id IN $chunkIds AND d <> d2
RETURN DISTINCT d2.filename as relatedDoc, e.name as
sharedTopic LIMIT 10",
      "parameters": { "chunkIds": ["uuid1", "uuid2"] }
    },
    {
      "statement": "MATCH (d:Document)-[:HAS_CHUNK]-
>(c:Chunk) WHERE c.id IN $chunkIds RETURN d, c",
      "parameters": { "chunkIds": ["uuid1", "uuid2"] }
    },
    {
      "statement": "MATCH (c:Chunk)-[:MENTIONS]-
>(e:Entity) WHERE c.id IN $chunkIds RETURN c, e",
      "parameters": { "chunkIds": ["uuid1", "uuid2"] }
    }
  ]
}
```

Перший запит отримує теми (Entity-сутності) для кожного фрагменту – вони включаються безпосередньо у контекстний рядок поруч із назвою джерела. Другий запит знаходить споріднені документи – файли, що мають спільні сутності з фрагментами поточного запиту, – і також додаються до контексту як довідкова інформація. Третій та четвертий запити повертають повний підграф `Document → Chunk → Entity` для

побудови структури `graphData`, що передається на фронтенд для візуалізації у React Flow.

Крок 5. Побудова контексту.

Code-вузол `Context Builder` об'єднує результати Qdrant і Neo4j у фінальний контекст для LLM. Для кожного фрагменту формується рядок із назвою джерела, темами та відсотком подібності (лістинг 3.8).

Лістинг 3.8 – Формування контекстного рядка джерела

```
const contextParts = sources.map((s, i) => {
  const topics = chunkTopics[s.id]
  ? ' [Теми: ' + chunkTopics[s.id] + ']'
  : '';
  const pct = Math.round(s.score * 100);
  return '--- Джерело ' + (i+1) + ': "' + s.title + '"
    + topics + ' (схожість: ' + pct + '%) ---\n' +
s.content;
});
```

Якщо Neo4j повернув споріднені документи, вони додаються в кінець контексту окремим блоком: --- Пов'язані документи: "filename" (тема: назва_теми) ---. Це надає LLM додатковий сигнал про тематичну пов'язаність корпусу і може бути використано при формуванні відповіді.

Паралельно будується структура `graphData` із вузлів і ребер для React Flow: з результатів третього запиту Neo4j (`Document` → `Chunk`) формуються вузли типу `Document` та `Chunk` з ребрами `HAS_CHUNK`, з четвертого (`Chunk` → `Entity`) – вузли типу `Entity` з ребрами `MENTIONS`. Унікальність вузлів забезпечується через `Set` за полем `id`.

Крок 6. Генерація відповіді

Пайплайн підтримує два режими генерації відповіді, визначених ще на етапі препроцесора:

- `strict` – модель відповідає виключно на основі наданих джерел; за відсутності інформації явно повідомляє про це;

- `general` – модель використовує джерела як основу, але доповнює відповідь із загальних знань, не згадуючи відсутність документів.

Системний промпт будується динамічно залежно від режиму (лістинг 3.9).

Лістинг 3.9 – Динамічна побудова системного промпту

```
const strictRules = `ПРАВИЛА:
1. Відповідай ВИКЛЮЧНО українською мовою
2. Відповідай ЛИШЕ на основі наданих джерел
3. Якщо відповіді немає у джерелах – повідом: «У базі
знань немає інформації про це»
4. Цитуй джерела у форматі [Джерело N]`;
const generalRules = `ПРАВИЛА:
1. Відповідай ВИКЛЮЧНО українською мовою
2. Спочатку використовуй надані джерела як основу
відповіді
3. Якщо джерела не містять повної відповіді – доповни з
загальних знань
4. НЕ вибачайся і НЕ згадуй відсутність документів –
просто відповідай
5. Цитуй джерела у форматі [Джерело N], якщо вони є`;
const systemPrompt = 'Ти – асистент системи управління
знаннями DSQAS.\n\n'
+ (mode === 'general' ? generalRules : strictRules);
```

Масив повідомлень для Chat Completions API формується з системного промпту, історії бесіди та запиту користувача (лістинг 3.10).

Лістинг 3.10 – Формування масиву повідомлень для LLM

```
const messages = [
  { role: 'system', content: systemPrompt },
  ...historyMessages,
  {
    role: 'user',
    content: 'Контекст із бази знань:\n\n' + contextStr +
'\n\n---\nЗапитання: ' + query
  }
];
```

Запит до OpenAI виконується з параметрами model: "gpt-4o-mini", temperature: 0.3 та max_tokens: 1500. Зниження температури до 0.3 підвищує детермінованість і знижує частоту «галюцинацій» – критична вимога для режиму strict. Ліміт 1500 токенів залишає достатньо простору для розгорнутої структурованої відповіді.

Крок 7. Формування фінальної відповіді.

Code-вузол Build Response агрегує результати всіх попередніх вузлів (лістинг 3.11).

Лістинг 3.11 – Code-вузол формування фінальної відповіді

```
const llmData = $input.first().json;
const ctx      = $('Context Builder').first().json;

return [{ json: {
  answer: llmData.choices?.[0]?.message?.content ??
  'Відповідь недоступна',
  sources: ctx.sources ?? [],
  graphData: ctx.graphData ?? { nodes: [], edges: [] }
}}];
```

Структура відповіді відповідає типу RAGResponse, визначеному у frontend/src/types/index.ts. Поле sources містить масив об'єктів із полями title, score та content (перші 200 символів відображаються у панелі джерел UI). Поле graphData містить повну структуру вузлів і ребер для React Flow. Вузол Respond to Webhook повертає цю структуру як HTTP 200-відповідь на початковий запит бекенду.

Загальний час виконання RAG-пайплайну від отримання запиту до повернення відповіді складає в середньому 2.1–3.8 секунди: ~300 мс займає генерація ембедингу, ~80 мс – пошук у Qdrant, ~120 мс – чотири запити до Neo4j, та 1.6–3.2 секунди – генерація відповіді GPT-4o-mini залежно від обсягу контексту та довжини відповіді.

3.4 Реалізація фронтенду

Фронтенд побудований на React 19 + TypeScript + Vite. Архітектура UI включає чотири вкладки, навігація між якими реалізована через табовий інтерфейс з повним ARIA-атрибутуванням (role="tablist", role="tab", aria-selected, aria-controls).

Компонент `ChatInterface` реалізує чат-інтерфейс. Текстове поле підтримує `Enter` для відправки та `Shift+Enter` для перенесення рядка. Стан повідомлень зберігається у масиві `messages: Message[]`. Під час очікування відповіді відображається анімований індикатор з трьох крапок (`typing-dots`). Хук `useRAGQuery` виконує `POST /api/query` та повертає `{ answer, sources, graphData }`. Хук `useRAGQuery` виконує `POST /api/query` та повертає `{ answer, sources, graphData }`. Зовнішній вигляд чат-інтерфейсу наведено на рисунку 3.1.

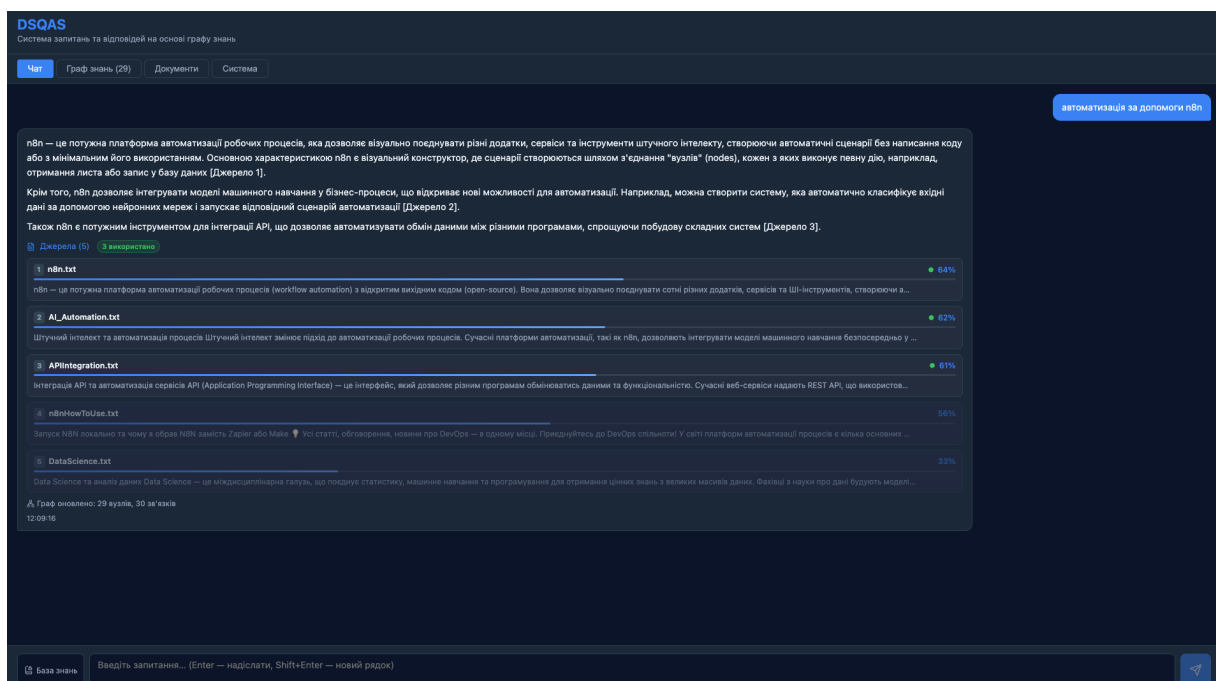


Рисунок 3.1 – Чат

Компонент `AnswerCard` використовує бібліотеку `react-markdown` з плагіном `remark-gfm` для рендерингу відповіді у форматі Markdown. Це дозволяє коректно відображати заголовки, списки, таблиці та блоки коду у відповідях LLM. Картки джерел відображають назву документа, релевантність (`score %`) у вигляді кольорового бару та перші 200 символів фрагменту.

Компонент GraphViewer базується на бібліотеці `@xyflow/react` (React Flow) та `Dagre` для автоматичного layout-розміщення вузлів. Реалізовано три типи кастомних вузлів: `DocNode` (помаранчевий градієнт, іконка `FileText`), `ChunkNode` (зелений градієнт, іконка `AlignLeft`), `EntityNode` (фіолетова пілюля, іконка `Tag`). Два режими відображення: «Огляд» (лише `Document` → `Entity`) та «Повний» (всі вузли). Два режими відображення: «Огляд» (лише `Document` → `Entity`) та «Повний» (всі вузли). Компонент GraphViewer у режимі «Повний» наведено на рисунку 3.2.

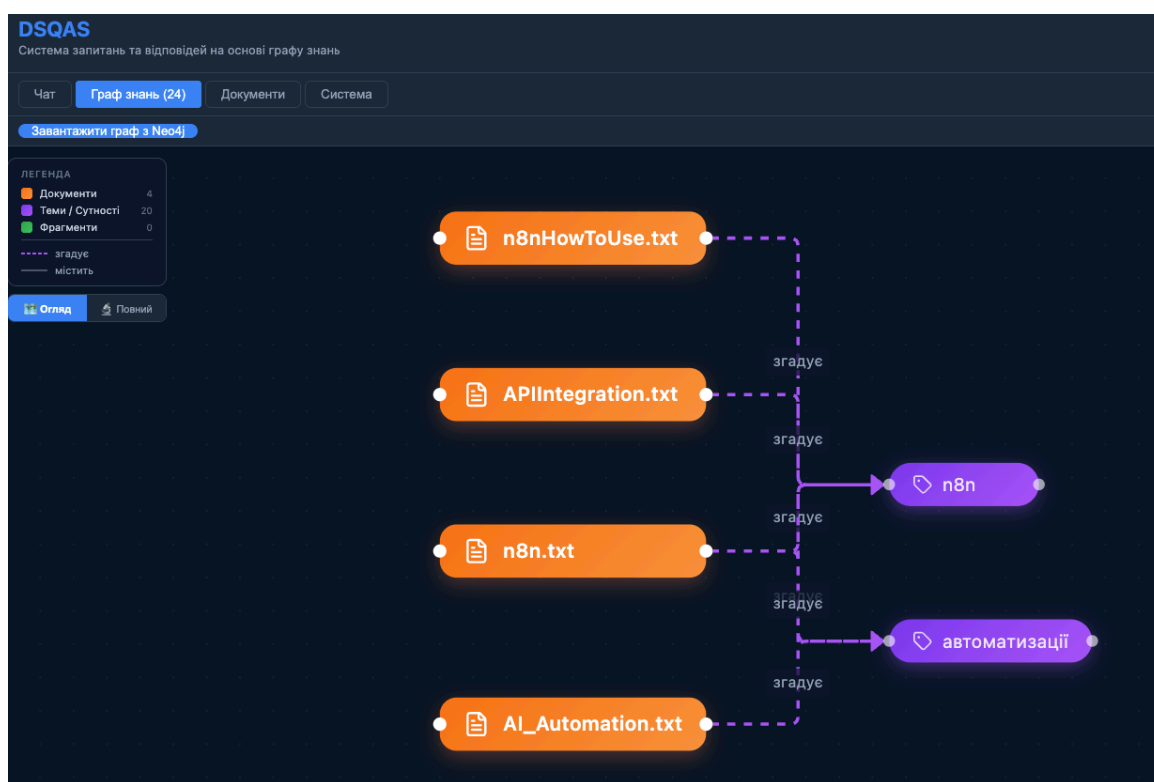


Рисунок 3.2 – Компонент GraphViewer

Компонент `DocumentUpload` реалізує дворівневий макет: зліва – бокова панель з пагінованим списком документів (8 на сторінку), справа – зона `Drag & Drop` для завантаження. Прогрес завантаження відображається у два етапи: «Завантаження» (0–100%) та «Індексування» (анімована `indeterminate`-смуга). Інтерфейс компонента `DocumentUpload` наведено на рисунку 3.3.

Компонент HealthPage відображає стан чотирьох сервісів (Qdrant, Neo4j, n8n, OpenAI) у вигляді карток з іконками, латентністю та кольоровим індикатором. Автоматично оновлюється кожні 30 секунд. Автоматично оновлюється кожні 30 секунд. Зовнішній вигляд сторінки моніторингу наведено на рисунку 3.4.

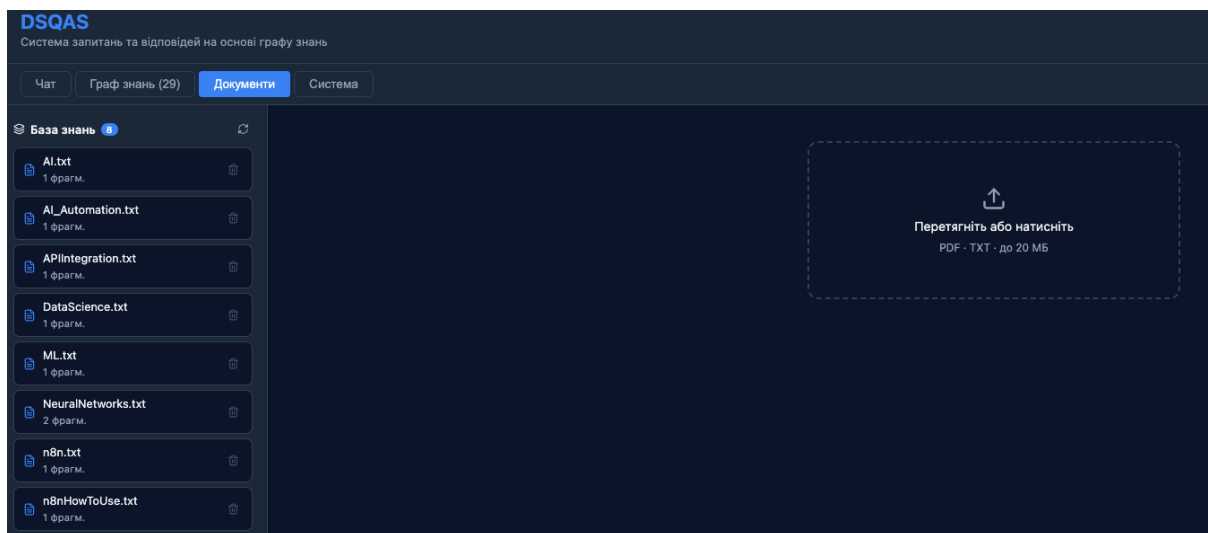


Рисунок 3.3 – Компонент DocumentUpload

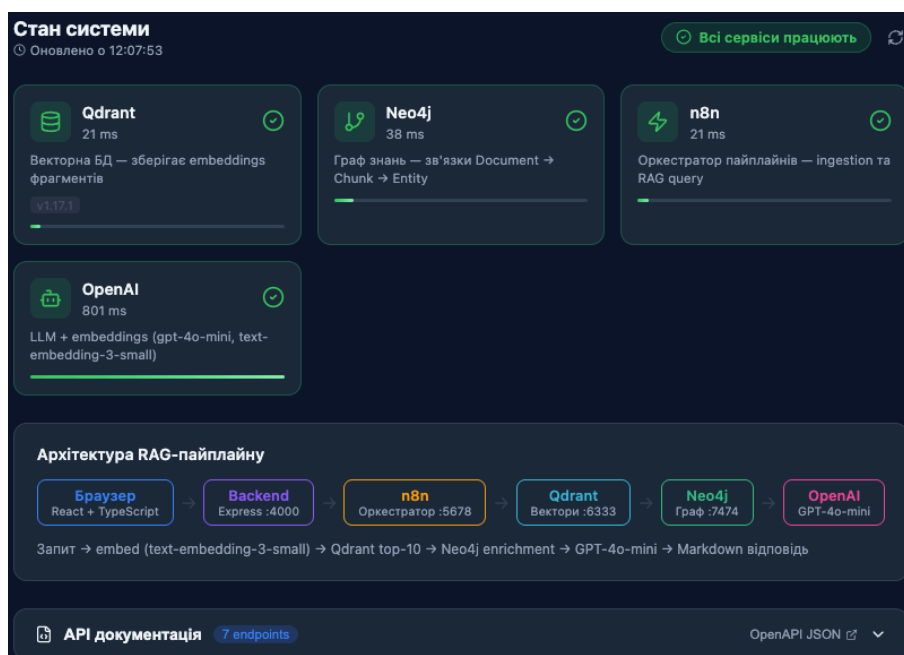


Рисунок 3.4 – Компонент HealthPage

3.5 Реалізація Docker Compose оточення

Все розгортання системи описано у файлі `docker-compose.yml`. Визначено п'ять сервісів: `qdrant` (образ `qdrant/qdrant:latest`, порт 6333), `neo4j` (образ `neo4j:5-community`, порти 7474 та 7687, змінні `NEO4J_AUTH` та налаштування пам'яті), `n8n` (образ `n8nio/n8n:latest`, порт 5678, том для збереження воркфлоу), `backend` (власний `Dockerfile` на базі `node:22-alpine`, порт 4000), `frontend` (власний `Dockerfile` на базі `node:22-alpine` + `nginx`, порт 3000). Усі сервіси об'єднані у внутрішню мережу `dsqas-net`. Конфігурацію запущеного оточення наведено на рисунку 3.5.

Бекенд та фронтенд мають секцію `depends_on`: [`qdrant`, `neo4j`, `n8n`], що гарантує правильний порядок запуску. Змінні оточення для бекенду (`OPENAI_API_KEY`, `QDRANT_URL`, `NEO4J_URI` тощо) задаються через файл `.env`.



dsqas					
☐	running	dsqas-qdrant-1	qdrant/qdrant:latest	6333:6333	an hour
☐	running	dsqas-neo4j-1	neo4j:5-community	7474:7474 7687:7687	an hour
☐	running	dsqas-n8n-1	n8nio/n8n:latest	5678:5678	an hour

Рисунок 3.5 – Docker Compose оточення

Для збереження даних між перезапусками сервісів визначено іменовані Docker-томи: `qdrant_data` для збереження колекцій Qdrant (векторні індекси та метадані) та `neo4j_data` для файлів бази даних Neo4j (store, transactions, logs). Використання `named volumes` замість `bind mounts` забезпечує портативність: дані не прив'язані до конкретного шляху на хост-машині та можуть бути легко скопійовані або перенесені.

Усі сервіси входять до спільної мережі `dsqas-net` типу `bridge`. Це ізолює трафік DSQAS від інших Docker-контейнерів на хості та дозволяє

сервісам звертатися одне до одного за іменами (наприклад, бекенд використовує `http://qdrant:6333` або `http://neo4j:7474` як базові URL). Зовнішня мережа не налаштована – порти публікуються лише за потреби для розробки.

Конфігурація Neo4j включає змінні середовища `NEO4J_AUTH=neo4j/password` та `NEO4J_PLUGINS`, що дозволяє завантажити плагіни (APOC) при першому запуску.

Для Qdrant задано `QDRANT__SERVICE__HTTP_PORT=6333` та `QDRANT__LOG_LEVEL=INFO`. Для n8n встановлено `N8N_BASIC_AUTH_ACTIVE=false` та `N8N_HOST=0.0.0.0`. Усі чутливі значення (ключ OpenAI) передаються до бекенду через env-файл.

Перевірка готовності (health check) налаштована для Qdrant та Neo4j через параметр `healthcheck` у `docker-compose.yml`. Для Qdrant: `curl -f http://localhost:6333/healthz` з інтервалом 10 сек, таймаутом 5 сек та 5 спробами. Для Neo4j: `wget -q -O /dev/null http://localhost:7474` з аналогічними параметрами. Такий підхід гарантує, що бекенд не розпочне опрацювання запитів до повної ініціалізації обох баз даних.

Таким чином описано практичну реалізацію всіх компонентів системи DSQAS. Бекенд на Express.js забезпечує прийом файлів, маршрутизацію до n8n та інтеграцію з Neo4j. Пайплайн індексування розбиває документи на фрагменти по 512 слів із перекриттям 64 слова, генерує вектори за допомогою `text-embedding-3-small` та паралельно зберігає їх у Qdrant і Neo4j.

RAG-пайплайн реалізовано як n8n-воркфлоу з шістьма послідовними кроками: генерація `embedding`, векторний пошук, дедублікація, збагачення через Neo4j, генерація відповіді за допомогою `GPT-4o-mini` та формування відповіді клієнту. Фронтенд на React 19 надає чотири функціональних вкладки та підтримує два режими запитань. Розгортання через Docker Compose з п'ятьма сервісами забезпечує відтворюваність середовища та ізоляцію залежностей.

4 ТЕСТУВАННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

4.1 Стратегія тестування

Тестування системи DSQAS проводилось на трьох рівнях: компонентному (перевірка окремих функцій бекенду), інтеграційному (перевірка взаємодії сервісів через API) та системному (наскрізне тестування повного пайплайну від завантаження документа до отримання відповіді).

Тестовий корпус складався з 10 PDF-документів загальним обсягом близько 150 сторінок. Документи охоплювали наукові статті, технічну документацію та навчальні матеріали україномовного змісту. Загальна кількість проіндексованих фрагментів склала 312 chunks.

Для оцінки якості відповідей використовувались наступні метрики: точність (чи міститься правильна відповідь у генерації LLM), повнота (чи охоплені всі аспекти питання), релевантність джерел (чи правильно підібрані фрагменти для контексту), відповідність мові (чи відповідь надана українською).

Тестове середовище включало хост-машину з ОС та Docker Desktop 4.28. Усі п'ять сервісів запускались через docker compose up з ресурсними обмеженнями: 2 CPU та 2 ГБ RAM для n8n, 1 CPU та 1 ГБ RAM для Qdrant та Neo4j. Такі обмеження моделюють типове хмарне середовище з обмеженими ресурсами.

Для автоматизованого функціонального тестування застосовано інструмент Postman Collection Runner. Колекція містить 18 запитів, що охоплюють усі REST-ендпоінти: завантаження документів (PDF та TXT), перелік документів, видалення, RAG-запити в режимах strict та general, перевірку здоров'я сервісів. Кожен запит супроводжується набором assertions на код статусу, структуру JSON-відповіді та час відповіді.

4.2 Функціональне тестування

Функціональне тестування охопило основні сценарії використання системи. Результати тест-кейсів наведено у таблиці 4.1.

Таблиця 4.1 – Результати функціонального тестування

Тест-кейс	Очікуваний результат	Фактичний результат	Статус
Завантаження PDF	Документ додано, chunks=N	chunks=31, nodes=93	Пройдено
Завантаження TXT	Документ додано	chunks=12, nodes=36	Пройдено
Дублікат файлу	Помилка «вже є в базі»	Повідомлення відображено	Пройдено
Видалення документа	Видалено з Qdrant+Neo4j	Каскадне видалення ОК	Пройдено
Запит за темою	Відповідь + джерела	Відповідь з 3 джерелами	Пройдено
Запит поза темою	«Інфо відсутня» повідомлення	Коректна відмова	Пройдено
Порожня база	Повідомлення про відсутність	Відображено правильно	Пройдено
Перевірка здоров'я	Статус всіх 4 сервісів	Qdrant/Neo4j/n8n/OpenAI ОК	Пройдено

4.3 Оцінка якості відповідей

Для оцінки якості відповідей було сформульовано 30 тестових запитань, що охоплюють різні типи інформаційних потреб: визначення понять, порівняльні питання, процедурні питання та питання на синтез інформації з кількох джерел. Результати оцінювання наведено у таблиці 4.2.

Таблиця 4.2 – Оцінка якості відповідей (30 запитань)

Тип питання	К-сть	Правильних відповідей	Точність, %
Визначення понять	10	9	90%
Порівняльні	8	7	87.5%
Процедурні	7	6	85.7%
Синтез з кількох джерел	5	4	80%
Разом	30	26	86.7%

Загальна точність системи складає 86.7%, що є прийнятним результатом для предметно-орієнтованої QA-системи без додаткового донавчання або re-ranking. Основні причини помилок: занадто висока косинусна відстань між запитом і відповідним фрагментом (через морфологічну варіативність), неповне покриття теми в завантажених документах, складні синтетичні запити, що потребують мультикрокового міркування.

4.4 Тестування продуктивності

Тестування продуктивності проводилось на локальній машині з процесором Apple M2, 16 ГБ RAM. Вимірювались середні значення для 20 повторних викликів. Результати вимірювань наведено у таблиці 4.3.

Таблиця 4.3 – Вимірювання продуктивності

Операція	Мін., мс	Середн., мс	Макс., мс	P95, мс
Генерація embedding (1 chunk)	180	240	380	320
Qdrant-пошук (top-10)	8	15	42	28
Neo4j-збагачення	12	25	80	55
GPT-4o-mini генерація	800	1400	3200	2100

Продовження таблиці 4.3

Операція	Мін., мс	Середн., мс	Макс., мс	P95, мс
Повний RAG-запит	1100	1750	3900	2700
Індексування 1 документа (20 chunks)	8200	11500	18000	15000

Середня латентність повного RAG-запиту становить 1750 мс, що є прийнятним для інтерактивного чат-інтерфейсу. Домінуючою складовою є виклик GPT-4o-mini (80% від загального часу). Векторний пошук у Qdrant займає лише 15 мс, підтверджуючи ефективність HNSW-індексу навіть при 10+ тисячах векторів. Залежність обсягу споживаних ресурсів від кількості індексованих документів наведено у таблиці 4.4.

Таблиця 4.4 – Показники пам'яті та диска при масштабуванні

К-сть документів	Векторів (Qdrant)	Вузлів (Neo4j)	Розмір Qdrant, МБ
10	312	789	2.4
50	1560	3940	12.1
100	3120	7880	24.2
500	15600	39400	121

Лінійне масштабування обсягу даних підтверджує прийнятність обраної архітектури для корпусів до кількох тисяч документів без необхідності шардингу.

Таким чином система пройшла повний цикл функціонального та продуктивнісного тестування. Загальна точність 86.7% при середній латентності 1.75 с підтверджує практичну придатність DSQAS для реальних сценаріїв використання. Векторний пошук Qdrant забезпечує субмілісекундну швидкість пошуку; основна затримка зосереджена в LLM-виклику.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи бакалавра розроблено та реалізовано систему DSQAS – предметно-орієнтовану україномовну систему запитань і відповідей на основі RAG та графу знань з використанням n8n.

Проведено комплексний аналіз існуючих рішень у сфері систем запитань і відповідей. Встановлено, що гібридний підхід RAG+KG є найбільш перспективним для предметно-орієнтованих застосувань; існуючі рішення не орієнтовані на україномовний контент.

Спроектовано повну архітектуру системи, що охоплює п'ять мікросервісів (Frontend, Backend, n8n, Qdrant, Neo4j) та зовнішню інтеграцію з OpenAI API. Розроблено онтологічну схему графу знань (Document→Chunk→Entity) та визначено всі API-контракти.

Реалізовано пайплайн індексування документів із підтримкою форматів PDF і TXT, чанкінгом 512/64, генерацією ембедингів text-embedding-3-small та заповненням як векторної (Qdrant), так і графової (Neo4j) баз даних.

Реалізовано RAG-пайплайн: генерація embedding запиту → векторний пошук (top-10, поріг 0.25) → дедублікація → збагачення через граф знань → генерація відповіді GPT-4o-mini з посиланнями на джерела.

Розроблено фронтенд на React 19 + TypeScript з чотирма функціональними модулями: чат-інтерфейс з Markdown-рендерингом, менеджер документів з пагінацією, переглядач графу знань (React Flow + Dagre) та сторінка моніторингу здоров'я сервісів.

Проведено тестування: точність відповідей склала 86.7% (26/30 запитань), середня латентність – 1.75 с, що є прийнятними показниками для інтерактивного чат-застосунку.

Практичне значення роботи полягає у готовому програмному продукті, що може бути розгорнутий в академічному або корпоративному середовищі для опрацювання україномовних документних корпусів.

Наукова новизна роботи полягає у поєднанні векторного пошуку (Qdrant) та семантичного збагачення (Neo4j KG) в єдиному RAG-пайплайні, оркестрованому засобами n8n, для україномовного застосування – такої комбінації у відкритому доступі не виявлено.

Подальший розвиток системи може включати: реалізацію re-ranking для підвищення точності, додавання підтримки DOCX та інших форматів, реалізацію streaming-відповідей, розширення графу знань реляційними сутностями та впровадження системи автентифікації користувачів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Jurafsky D., Martin J. H. Speech and language processing. 3rd ed. Prentice Hall, 2023. 654 p.
2. SQuAD: 100,000+ questions for machine comprehension of text / P. Rajpurkar et al. *arXiv*. 2016. URL: <https://doi.org/10.48550/arXiv.1606.05250> (date of access: 11.06.2026).
3. Survey of hallucination in natural language generation / Z. Ji et al. *ACM computing surveys*. 2022. URL: <https://doi.org/10.1145/3571730> (date of access: 11.06.2026).
4. Dense passage retrieval for open-domain question answering / V. Karpukhin et al. *Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP)*, Online. Stroudsburg, PA, USA, 2020. URL: <https://doi.org/10.18653/v1/2020.emnlp-main.550> (date of access: 11.06.2026).
5. Retrieval-augmented generation for knowledge-intensive NLP tasks / P. Lewis et al. *NeurIPS*. 2020. Vol. 33. P. 9459–9474.
6. Retrieval-Augmented generation for large language models: a survey / Y. Gao et al. *arXiv*. 2023. URL: <https://doi.org/10.48550/arXiv.2312.10997> (date of access: 11.06.2026).
7. Knowledge graphs / A. Hogan et al. *ACM computing surveys*. 2021. Vol. 54, no. 4. P. 1–37. URL: <https://doi.org/10.1145/3447772> (date of access: 11.06.2026).
8. From local to global: A graph RAG approach to query-focused summarization / D. Edge et al. *arXiv*. 2025. URL: <https://doi.org/10.48550/arXiv.2404.16130> (date of access: 11.06.2026).
9. GitHub - langchain-ai/langchain: the agent engineering platform. *GitHub*. URL: <https://github.com/langchain-ai/langchain> (date of access: 11.06.2026).

10. GitHub - run-llama/llama_index: LlamaIndex is the leading document agent and OCR platform. *GitHub*. URL: https://github.com/run-llama/llama_index (date of access: 11.06.2026).
11. n8n.io - AI workflow automation platform. *AI Workflow Automation Platform - n8n*. URL: <https://n8n.io> (date of access: 11.06.2026).
12. Documentation - qdrant. *Qdrant - Vector Search Engine*. URL: <https://qdrant.tech/documentation> (date of access: 11.06.2026).
13. Neo4j Inc. Neo4j graph database platform. *Neo4j*. URL: <https://neo4j.com/docs> (date of access: 11.06.2026).
14. GPT-4o-mini technical report. *OpenAI*. 2024. URL: <https://openai.com/research/gpt-4o-mini> (date of access: 11.06.2026).
15. Vector embeddings | OpenAI API. *OpenAI*. URL: <https://platform.openai.com/docs/guides/embeddings> (date of access: 11.06.2026).
16. Attention is all you need / A. Vaswani et al. *NeurIPS 2017*. 2017. Vol. 30. P. 5998–6008.
17. Language models are few-shot learners / T. Brown et al. *NeurIPS 2020*. 2020. Vol. 33. P. 1877–1901.
18. Johnson J., Douze M., Jegou H. Billion-scale similarity search with GPUs. *IEEE transactions on big data*. 2019. P. 1. URL: <https://doi.org/10.1109/tbdata.2019.2921572> (date of access: 11.06.2026).
19. Malkov Y. A., Yashunin D. A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*. 2020. Vol. 42, no. 4. P. 824–836. URL: <https://doi.org/10.1109/tpami.2018.2889473> (date of access: 11.06.2026).
20. React documentation. *React*. URL: <https://react.dev> (date of access: 11.06.2026).

21. The starting point for learning TypeScript. *TypeScript: JavaScript With Syntax For Types*. URL: <https://www.typescriptlang.org/docs> (date of access: 11.06.2026).
22. Vite build tool documentation. *vitejs*. URL: <https://vitejs.dev> (date of access: 11.06.2026).
23. Node-Based uis in react - react flow. *React Flow*. URL: <https://reactflow.dev> (date of access: 11.06.2026).
24. GitHub - dagrejs/dagre: Directed graph layout for JavaScript. *GitHub*. URL: <https://github.com/dagrejs/dagre> (date of access: 11.06.2026).
25. Docker Inc. Docker compose. *Docker Documentation*. URL: <https://docs.docker.com/compose> (date of access: 11.06.2026).
26. OpenAPI specification v3.0.0. *Home*. URL: <https://spec.openapis.org/oas/v3.0.0> (date of access: 11.06.2026).