НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Михайло Новотарський
(підпис)

“__” _____2025 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 123 “Комп’ютерна інженерія ”

на тему: Спосіб оптимізації роботи мікросервісної системи підтримки з використанням баз даних на основі моделі RAG

Виконав : студент __4__ курсу, групи _ІО-14_
(шифр групи)

_____Ухач Денис Сергійович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____доцент, к.т.н. Русінов В. В._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) асистент, д.т.н., Міщенко Л. Д._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент __пос. доцент кафедри ІСТ Володимр Шивмкович_____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2025 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 123 “Комп'ютерна інженерія”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Михайло Новотарський
(підпис)
“__” _____2025 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Ухача Дениса Сергійовича

1. Тема проєкту *Спосіб оптимізації роботи мікросервісної системи підтримки з використанням баз даних на основі моделі RAG*
керівник проєкту Русінов Володимир Володимирович, доцент, к.т.н.,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 11 травня 2025 року №1139-с

2. Термін здачі студентом закінченого проєкту *2 червня 2025 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

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

Розділ 1. Огляд методів оптимізації мікросервісних архітектур та моделей на основі RAG

Розділ 2. Огляд технологій для побудови мікросервісної системи підтримки з використанням баз даних

Розділ 3. Деталі розробки системи.

Розділ 4. Дослідження та аналіз розробленої системи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи, функціональна схема (діаграма класів), алгоритм дій програмного забезпечення.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Міщенко Л. Д.		

7. Дата видачі завдання «20» січня 2025 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*10.12.2024-15.12.2024*	
2.	*Вивчення та аналіз завдання*	*15.12.2024-15.03.2025*	
3.	*Розробка архітектури та загальної структури системи*	*15.03.2025-25.03.2025*	
4.	*Розробка структур окремих підсистем*	*25.03.2025-5.04.2025*	
5.	*Програмна реалізація системи*	*5.04.2025-15.04.2025*	
6.	*Оформлення пояснювальної записки*	*15.04.2025-20.05.2025*	
7.	*Захист програмного продукту*	*25.04.2025*	
8.	*Передзахист*	*12.06.2025*	
9.	*Захист*	*14.06.2025*	

Студент-дипломник _____ Денис Ухач
(підпис)

Керівник проєкту _____ Володимр РУСІНОВ
(підпис)

АНОТАЦІЯ

У даній роботі розглянуто підхід до оптимізації роботи мікросервісної системи підтримки з використанням баз знань та моделі генерації з підключенням до джерел інформації (Retrieval-Augmented Generation, RAG). Було проаналізовано сучасні архітектурні підходи до створення мікросервісів, проблеми масштабування, продуктивності та методи їх вирішення. Проведено огляд технологій, які дозволяють інтегрувати модель RAG у мікросервісне середовище. Розроблено систему, що дозволяє поєднувати генеративні можливості великих мовних моделей з актуальними даними з бази знань. Реалізовано механізм ефективного збереження, індексації та пошуку векторизованої інформації. Проведено експериментальні дослідження щодо впливу RAG-підходу на продуктивність та точність відповідей системи підтримки. Програмний продукт реалізовано з використанням Node.js, PostgreSQL та векторної бази даних.

Ключові слова: мікросервіси, оптимізація, RAG, бази знань, генеративна модель, підтримка користувачів.

ANNOTATION

This thesis presents an approach to optimizing the performance of a microservice-based support system using knowledge bases and a Retrieval-Augmented Generation (RAG) model. The work analyzes modern architectural principles of microservice design, scalability and performance challenges, and methods for addressing them. An overview of technologies enabling RAG integration into a microservice environment is provided. A system was developed that combines the generative capabilities of large language models with access to up-to-date knowledge from a database. A mechanism for efficient storage, indexing,

and retrieval of vectorized information was implemented. Experimental studies were conducted to evaluate the impact of the RAG-based approach on system performance and response accuracy. The software was implemented using Node.js, PostgreSQL, and a vector database.

Keywords: microservices, optimization, RAG, knowledge base, generative model, user support.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Спосіб оптимізації роботи	3		
			мікросервісної системи			
			підтримки з використанням			
			баз данних на основі моделі			
			RAG			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Спосіб оптимізації роботи	106		
			мікросервісної системи			
			підтримки з використанням			
			баз данних на основі моделі			
			RAG			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Спосіб оптимізації роботи	1		
			мікросервісної системи			
			підтримки з використанням			
			баз данних на основі моделі			
			RAG			
			Структурна схема			
			системи			
	А4	***ІАЛЦ.4672008.005 Д2***	Спосіб оптимізації роботи	1		
			мікросервісної системи			
			підтримки з використанням			
			баз данних на основі моделі			
			RAG			
			Функціональна схема			
			(діаграма класів)			
	А4	***ІАЛЦ.467200.006 Д3***	Спосіб оптимізації роботи	1		

			мікросервісної системи			
			підтримки з використанням			
			баз данних на основі моделі			
			RAG			
			Алгоритм дій програмного			
			забезпечення			
	А4	***ІАЛЦ.467200.007 Д4***	Спосіб оптимізації роботи	41		
			мікросервісної системи			
			підтримки з використанням			
			баз данних на основі моделі			
			RAG			
			Текст програмного коду			

					ІАЛЦ.467200.001 ОА		
Зм	***Лист***	***№ докум.***	***Підп***	***Дата***			
Розроб		Ухач Д.С.			*Еволюційні алгоритми глобальної пошукової оптимізації* Опис альбому	Літ.	Аркуш 1 / Аркушів 1
Перев		Волокита А.М.					
						НТУУ "КПІ" ФІОТ ІП-73	

ТЕХНІЧНЕ ЗАВДАННЯ

ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Спосіб оптимізації роботи мікросервісної системи підтримки з використанням баз даних на основі моделі RAG*»

Київ – 2025

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку мікросервісної системи підтримки з використанням баз даних та моделі Retrieval-Augmented Generation (RAG), а також на подальшу підтримку й удосконалення розробленої системи.

Областю застосування цієї системи є автоматизація обробки запитів у службах підтримки, корпоративних інформаційних системах, технічній підтримці користувачів, чат-ботах, цифрових помічниках, а також в аналітичних системах, де необхідна генерація відповідей на основі актуальних даних з баз знань. Система також може бути застосована в освітніх та консультаційних сервісах, де потрібна точна й релевантна інформація в реальному часі.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання кваліфікаційної роботи освітнього рівня «бакалавр інженерії програмного забезпечення», яке було затверджене факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного університету України «Київський Політехнічний Інститут імені Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка мікросервісної системи підтримки, яка використовує модель Retrieval-Augmented Generation (RAG) для покращення якості та швидкості обробки запитів. Розроблена система дозволить забезпечити точну генерацію відповідей на основі актуальних даних з баз знань, підвищити ефективність обслуговування користувачів і зменшити навантаження на операторів підтримки.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелами розробки даного дипломного проєкту є офіційна документація сучасних технологій для реалізації RAG-моделі, наукові публікації та статті з відкритих джерел, технічна література з мікросервісної архітектури, баз даних та генеративного штучного інтелекту.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена мікросервісна система підтримки з використанням моделі RAG повинна відповідати таким вимогам:

- Простий і інтуїтивно-зрозумілий API або користувацький інтерфейс (у разі наявності UI).
- Забезпечення обробки вхідних запитів від користувача з можливістю довантаження зовнішніх знань із бази даних.
- Гнучка конфігурація параметрів моделі та джерел знань.
- Підтримка масштабованої мікросервісної архітектури (через Docker або інші засоби).
- Повна технічна документація щодо розгортання, використання та супроводу системи.

5.2. Вимоги до програмного забезпечення

- Операційна система: macOS Ventura або вище, Windows 10/11, Linux (Ubuntu 20.04+).
- Платформи: Node.js 18+, PostgreSQL 14+, Docker (остання стабільна версія), підтримка векторної бази (наприклад, Qdrant, Weaviate або Pinecone).
- Додатково: підтримка TypeScript, бібліотеки LangChain або аналоги для реалізації RAG.
- Редактори/середовище розробки: Visual Studio Code або WebStorm.

5.3. Вимоги до апаратної частини

- Процесор: Apple Silicon (M3) або еквівалентний Intel Core i5/i7 10-го покоління і вище.
- SSD: не менше ніж 256 ГБ вільного простору.
- RAM: не менше ніж 8 ГБ (рекомендовано 16 ГБ для роботи з великими моделями).
- Наявність інтернет-з’єднання для доступу до моделей або баз знань (за потреби).

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	10.01.2025-15.01.2025
Вивчення та аналіз завдання	15.01.2025-15.03.2025
Розробка архітектури та загальної структури системи	15.03.2025-25.03.2025
Розробка структур окремих частин системи	25.03.2025-5.04.2025
Програмна реалізація системи	5.04.2025-15.04.2025
Виправлення помилок	15.04.2025-20.05.2025
Оформлення пояснювальної записки	25.04.2025

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Спосіб оптимізації роботи мікросервісної системи підтримки з використанням баз даних на основі моделі RAG*»

Київ – 2025

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

Скорочення	Розшифрування
API	Application Programming Interface — інтерфейс прикладного програмування
RAG	Retrieval-Augmented Generation — генерація з доповненням на основі пошуку
LLM	Large Language Model — велика мовна модель
REST	Representational State Transfer — архітектурний стиль веб-сервісів
DB	Database — база даних
SQL	Structured Query Language — мова структурованих запитів
NoSQL	Not Only SQL — модель баз даних, яка не використовує реляційну схему
CRUD	Create, Read, Update, Delete — основні операції з даними
DTO	Data Transfer Object — об'єкт передачі даних
NestJS	Framework для створення серверних додатків на Node.js
FastAPI	Python-фреймворк для створення високопродуктивних API
Kafka	Розподілена платформа потокової передачі даних
Pinecone	Векторна база даних для зберігання ембедингів
Embedding	Векторне представлення тексту
SDK	Software Development Kit — набір інструментів для розробника
ORM	Object-Relational Mapping — об’єктно-реляційне відображення
CRUD	Create, Read, Update, Delete — базові операції з базою даних
JWT	JSON Web Token — формат передачі авторизаційної інформації
CI/CD	Continuous Integration / Continuous Deployment — безперервна інтеграція та розгортання
HTTP	Hypertext Transfer Protocol — протокол передачі гіпертексту
UUID	Universally Unique Identifier — універсальний унікальний ідентифікатор
QA	Question Answering — система запитання–відповідь

ВСТУП

Багато завдань, які з’являються в таких фундаментальних науках, як фізика, хімія та молекулярна біологія, а також у багатьох прикладних науках, зводяться до завдання глобальної оптимізації. Властивостями таких завдань часто є:

- нелінійність;
- недиференційованість;
- багатоекстремальність (мультимодальність);
- овражність;
- відсутність аналітичного виразу;
- висока обчислювальна складність оптимізаційних функцій;
- висока розмірність простору пошуку;
- складна топологія областей допустимих значень і т. д.

Для ефективного вирішення заданих завдань у 80-х роках минулого століття почали інтенсивно розширювати еволюційні пошукові алгоритми оптимізації. У цій роботі я розгляну класи таких алгоритмів, які в різних джерелах називають по-різному: поведінковими, інтелектуальними, метаевристичними, інспірованими природою, роєвими, багатоагентними, популяційними та іншими.

Переважна більшість розглянутих алгоритмів представлена в англомовній літературі, у якій прийнятно використовувати термін “алгоритм” замість загальновживаного в українській літературі терміна “метод”.Щоб уникнути можливої неоднозначності при ідентифікації розглянутих в роботі об'єктів, я також використовую останній термін.

Можна запропонувати кілька класифікацій еволюційних алгоритмів. Виділю наступні класи таких алгоритмів:

- еволюційні алгоритми, включаючи генетичні алгоритми;

- популяційні алгоритми, натхненні живою природою;
- алгоритми, натхненні неживою природою;
- алгоритми, інспіровані людським суспільством та інші.

Еволюційні алгоритми, а також такі популяційні алгоритми, натхненні живою природою, як алгоритми рою частинок, колонії мурах, рої медоносних бджіл і алгоритми на основі штучної імунної системи досить добре освітлені в українській літературі.

Дана робота являє собою огляд великої кількості інших еволюційних алгоритмів, які рідко зустрічаються в нашій літературі.

РОЗДІЛ 1. ОГЛЯД ПІДХОДІВ ДО ОПТИМІЗАЦІЇ МІКРОСЕРВІСНИХ СИСТЕМ

1.1 Принципи побудови мікросервісної архітектури

1.1.1 Основні характеристики мікросервісів

Мікросервісна архітектура є підходом до проєктування програмного забезпечення, який ґрунтується на ідеї розділення системи на набір невеликих, автономних служб (сервісів), кожна з яких відповідає за окрему функціональність або бізнес-процес. Це дозволяє створювати більш гнучкі, масштабовані та легше підтримувані системи, особливо в умовах динамічного зростання продукту чи команди розробки.

Ключова риса мікросервісів полягає у незалежності. Кожен мікросервіс є самодостатнім: він має власну бізнес-логіку, окрему кодову базу, життєвий цикл, базу даних або інше сховище, а також засоби розгортання. Мікросервіси спілкуються між собою через чітко визначені API, зазвичай використовуючи HTTP/REST, gRPC або брокери повідомлень (наприклад, Kafka, RabbitMQ). Завдяки цьому між сервісами встановлюються слабкі зв’язки (loose coupling), що сприяє автономності змін.

Іншою важливою характеристикою є незалежне розгортання. Мікросервіси дозволяють оновлювати окремі компоненти без зупинки всієї системи. Це значно знижує ризики при деплої та спрощує CI/CD процеси. Команди можуть працювати над різними сервісами одночасно, з різною швидкістю, що підвищує продуктивність організації.

Автономія також відображається на технологічному рівні. Кожна команда може самостійно обирати мову програмування, фреймворк, архітектурні патерни та систему збереження даних, яка найкраще відповідає

конкретному завданню. Наприклад, сервіс для аналітики може використовувати Python та ClickHouse, а сервіс для обробки платежів — Java і PostgreSQL. Ця гетерогенність потребує загального узгодження стандартів API і логування, проте відкриває широкі можливості для оптимізації.

Ще однією важливою рисою є масштабованість. Кожен мікросервіс можна масштабувати окремо залежно від навантаження. Якщо, наприклад, сервіс авторизації піддається більшому навантаженню, його інстанси можна збільшити незалежно від інших. Це дозволяє ефективно використовувати ресурси інфраструктури, особливо в умовах хмарних середовищ.

Також мікросервісна архітектура добре відповідає принципам доменно-орієнтованого проєктування (DDD). Кожен сервіс представляє собою реалізацію певного bounded context, що допомагає структурувати систему відповідно до бізнес-потреб. Це також спрощує тестування, адже кожен сервіс має чітко визначену відповідальність і його можна ізолювати в юніт- або інтеграційних тестах.

У той же час для ефективної роботи мікросервісів необхідна добре продумана інфраструктура. Потрібні інструменти для реєстрації сервісів, моніторингу, логування, трасування викликів (наприклад, Jaeger, Prometheus, ELK), а також забезпечення безпеки, автентифікації та керування конфігурацією. Усі ці аспекти вимагають ретельного планування та спеціалізованої експертизи.

Таким чином, основні характеристики мікросервісів — незалежність, автономність, масштабованість, технологічна свобода та гнучкість — роблять цей підхід особливо привабливим для побудови складних, розподілених систем. Водночас вони вимагають вищого рівня інженерної дисципліни та зрілості команди, ніж при роботі з традиційним монолітом.

1.1.2 Відмінності від монолітних систем

Монолітна архітектура — це класичний підхід до розробки програмного забезпечення, при якому вся функціональність системи розміщується в єдиній кодовій базі та виконується в одному процесі. В межах такого підходу усі модулі (авторизація, каталог, обробка платежів, аналітика тощо) тісно пов’язані між собою, часто спільно використовують одну базу даних і мають єдиний цикл збірки, тестування, деплою та масштабування.

На ранніх етапах розвитку продукту та команди це може бути вигідним: менше технічних складностей, простіше налагодження, одна система збірки, а отже швидша доставка перших версій продукту. Однак з часом, коли кількість функціоналу зростає, моноліт починає “розростатися” до такого ступеня, що будь-які зміни стають ризикованими. Один незначний баг у модулі, пов’язаному з реєстрацією, може викликати збої в роботі аналітики чи платежів. Це знижує стабільність усієї системи.

Моноліт погано піддається масштабуванню на рівні окремих функціональностей. Якщо збільшується навантаження на один компонент (наприклад, API замовлень), то для обслуговування цього компоненту доводиться масштабувати всю систему, включно з модулями, яким це зовсім не потрібно. Такий підхід призводить до неефективного використання обчислювальних ресурсів.

У мікросервісній архітектурі ці проблеми розв’язуються завдяки поділу системи на ізольовані частини. Кожна функціональність реалізована у вигляді окремого сервісу, який має власну кодову базу, логіку, сховище даних, процеси деплою та масштабування. Такий сервіс може бути написаний на іншій мові програмування, використовувати специфічну СУБД або фреймворк, залежно від задачі. Наприклад, сервіс для збереження великих масивів телеметрії може бути реалізований на Go з TimescaleDB, тоді як основний бекенд — на Node.js з PostgreSQL.

Ще однією важливою перевагою мікросервісів є можливість незалежної роботи команд. У межах великої організації можна виділити окремі команди для кожного сервісу, які самостійно керують життєвим циклом свого продукту. Це суттєво підвищує швидкість розробки, оскільки зникає потреба у глобальних узгодженнях змін. У випадку з монолітом будь-яка зміна потребує синхронізації між усіма учасниками команди та ретельної перевірки інтеграційних зв’язків.

Водночас мікросервіси вносять свою складність. Вони вимагають розвиненої інфраструктури — систем логування, трасування, моніторингу, засобів балансування навантаження та автоматичного масштабування. Також виникає проблема міжсервісної комунікації: виклики по HTTP чи через брокери повідомлень повільніші та менш надійні, ніж локальні методи. З'являється потреба в ретельному проєктуванні API, обробці збоїв, таймаутах, відмовостійкості.

У моноліті транзакції легко координуються СУБД з гарантіями ACID, але в мікросервісах доводиться вдаватися до патернів типу Saga або Eventual Consistency, що ускладнює логіку бізнес-процесів.

Таким чином, монолітна архітектура є зручною на початкових етапах розробки, коли пріоритетом є швидкий запуск MVP. Однак зі зростанням масштабу системи перехід до мікросервісної архітектури стає виправданим для досягнення гнучкості, масштабованості та розподіленої відповідальності між командами. Кожен з підходів має свої переваги і недоліки, і вибір залежить від конкретних цілей, ресурсів та етапу розвитку продукту.

1.1.3 Переваги та недоліки архітектури

Мікросервісна архітектура стала домінуючим підходом для розробки масштабованих, розподілених систем завдяки ряду суттєвих переваг, які вона пропонує над традиційним монолітом. Найперше — це масштабованість.

Кожен сервіс можна масштабувати незалежно від інших, залежно від його навантаження. Наприклад, якщо лише сервіс авторизації зазнає великої кількості запитів — достатньо збільшити кількість його інстансів, не зачіпаючи інші компоненти. Це дозволяє значно ефективніше використовувати ресурси, порівняно з масштабуванням усього моноліту.

Ще однією ключовою перевагою є незалежність компонентів. Завдяки тому, що кожен мікросервіс виконує чітко визначену функцію і має окрему кодову базу, команди розробників можуть працювати автономно, не створюючи конфліктів при інтеграції коду. Це суттєво зменшує кількість помилок, що виникають при розгортанні нових версій, і дозволяє впроваджувати зміни без страху зламати критичну функціональність в іншій частині системи.

Також важливим фактором є простота тестування та оновлення. Мікросервіси легко покриваються юніт- та інтеграційними тестами, їх можна запускати ізольовано, і це дає змогу перевірити функціональність без потреби розгортання всієї системи. Крім того, якщо виникає потреба в оновленні лише одного сервісу, не потрібно повторно деплоїти весь додаток, як це зазвичай буває з монолітом.

Однією з найбільших свобод, яку надає мікросервісний підхід, є технологічна різноманітність. Розробники можуть обирати найоптимальніші інструменти для кожного завдання — наприклад, використовувати Python для аналітичного сервісу, Go для високопродуктивного API, а Node.js для роботи з подієвою моделлю. Це не тільки дозволяє створювати більш ефективні рішення, але й спрощує залучення фахівців із різним технологічним бекграундом.

Проте ці переваги приходять не без суттєвих викликів. Однією з основних проблем є ускладнення логування, моніторингу та трасування. У монолітній архітектурі можна легко відстежити весь шлях запиту в одному журналі. У мікросервісах же запит часто проходить через багато сервісів, і без систем централізованого логування (наприклад, ELK Stack, Grafana Loki) та

розподіленого трасування (Jaeger, Zipkin) стає практично неможливо відслідкувати причини помилок.

Ще однією проблемою є зростання кількості мережевих викликів. Оскільки сервіси взаємодіють між собою через HTTP/gRPC, кожен виклик супроводжується накладними витратами, які впливають на продуктивність і час відгуку. Це також збільшує ризик збоїв, оскільки мережеві з'єднання є менш надійними порівняно з внутрішньопроцесними викликами.

Крім того, CI/CD-процеси стають складнішими, адже замість одного пайплайну для моноліту, доводиться підтримувати десятки окремих конвеєрів, що потребують узгодженості між собою, особливо у випадках міжсервісних залежностей. А отже — збільшується і час на налаштування інфраструктури.

Також варто враховувати, що DevOps-практики стають обов'язковою складовою проєкту. Впровадження мікросервісів без автоматизації розгортання, масштабування, моніторингу і логування може лише ускладнити систему, замість того щоб покращити її. Таким чином, командам, які переходять до мікросервісів, необхідно бути готовими до суттєвих інвестицій у DevOps-інфраструктуру.

Загалом, мікросервіси не є універсальним рішенням, придатним для всіх типів проєктів. У невеликих командах або при розробці простих систем мікросервісна архітектура може виявитися надмірно складною. Однак для великих розподілених систем, які потребують гнучкого масштабування, частих оновлень і високої надійності — це один із найефективніших підходів, який при правильному впровадженні дозволяє досягти високих результатів.

1.2 Проблеми масштабування та продуктивності в мікросервісах

.

1.2.1 Масштабування горизонтальне та вертикальне

Масштабування — це ключовий аспект у розробці розподілених систем, що безпосередньо впливає на їхню здатність витримувати навантаження та забезпечувати стабільну роботу. У контексті мікросервісної архітектури масштабування набуває особливої гнучкості завдяки автономності кожного з сервісів. В загальному розумінні існує два основні підходи до масштабування — горизонтальне (horizontal scaling) та вертикальне (vertical scaling). Обидва мають свої переваги, недоліки та області застосування.

Горизонтальне масштабування, або масштабування "вбік", полягає у збільшенні кількості інстансів (екземплярів) одного й того самого сервісу. Наприклад, якщо сервіс обробки зображень стає вузьким місцем через великий потік запитів, замість розширення ресурсів одного сервера, додається ще кілька копій цього сервісу, що працюють паралельно. Кожен інстанс приймає частину запитів, розподілених через балансувальник навантаження (load balancer). Такий підхід легко автоматизується за допомогою сучасних інструментів оркестрації — зокрема, Kubernetes дозволяє створювати автомасштабування залежно від навантаження (horizontal pod autoscaler).

Горизонтальне масштабування є основою cloud-native підходів, де сервіси можуть бути запущені на великій кількості хостів, з мінімальними змінами в конфігурації. При цьому досягається висока відмовостійкість — у разі падіння одного інстансу, інші можуть продовжити обробку запитів без зупинки системи. Воно також дає змогу ефективно розподіляти трафік по регіонах, використовуючи географічні CDN або DNS-ротацію для маршрутизації користувачів до найближчих дата-центрів.

Вертикальне масштабування, або масштабування "вгору", передбачає збільшення ресурсів вже існуючого інстансу: додавання оперативної пам’яті, процесорних ядер, збільшення пропускної здатності мережі тощо. Це може бути корисно для сервісів, які складно паралелізувати, або коли використання розподіленого середовища економічно необґрунтоване. Наприклад, аналітичні

сервіси або обчислювальні модулі з високим рівнем завантаження процесора можуть виграти від використання потужнішого серверного заліза.

Однак вертикальне масштабування має свої обмеження: воно не нескінченне, і з часом вартість розширення одного вузла перевищує вигоду. Крім того, відмова єдиного потужного сервера призводить до повної зупинки сервісу, що знижує надійність системи.

Вибір між горизонтальним і вертикальним масштабуванням залежить від архітектури самого сервісу. Наприклад, сервіси, які підтримують stateless-виконання (тобто не зберігають стан між запитами), ідеально підходять для горизонтального масштабування. Становими ж компонентами (stateful) складніше управляти, і вони часто вимагають або окремих стратегій зберігання стану (наприклад, через Redis або бази даних), або вертикального масштабування.

У мікросервісній екосистемі часто поєднують обидва підходи. Наприклад, базу даних масштабують вертикально для підвищення її продуктивності, а API-сервіси — горизонтально для розподілу навантаження. Також часто впроваджуються автоматизовані системи моніторингу (Prometheus, Grafana), які реагують на зміну навантаження та запускають процеси масштабування в реальному часі.

Насамкінець слід зазначити, що масштабування в мікросервісах не є окремим етапом оптимізації, а інтегральною частиною всієї архітектурної моделі. Про нього потрібно думати вже на етапі проєктування — визначати, які сервіси можуть бути розділені, як вони зберігають свій стан, як буде відбуватися маршрутизація трафіку, і якими будуть обмеження апаратної інфраструктури. Правильне впровадження стратегії масштабування забезпечує не лише продуктивність, але й стійкість, гнучкість і економічність усієї мікросервісної системи.

1.2.2 Комунікаційні затримки та каскадні збої

Однією з ключових особливостей мікросервісної архітектури є те, що взаємодія між компонентами системи здійснюється не через внутрішні виклики функцій або методів у спільному коді, як у моноліті, а через мережу — найчастіше за допомогою протоколів HTTP/REST або gRPC. Це означає, що кожен запит між сервісами є повноцінною мережею операцією, яка підпадає під усі обмеження мережевої інфраструктури — затримки, втрати пакетів, таймаути, мережеві збої, перевантаження тощо. У реальних умовах навіть найстабільніші мережі можуть мати флуктуації, що відображається на загальній продуктивності та надійності системи.

Затримки (latency) виникають через фізичні обмеження пропускної здатності мережі, обробку запитів з боку сервісів та черги очікування. Якщо один мікросервіс очікує відповідь від іншого, і цей інший тимчасово перевантажений або недоступний, то запит зависає — або поки не буде отримано відповідь, або до настання таймауту. Особливо небезпечним є ланцюгове очікування, коли один сервіс чекає на інший, той на третій, і так далі — це створює ланцюги залежностей, які можуть уповільнити або повністю зупинити роботу ключових бізнес-процесів.

Каскадні збої (cascade failures) — це ще серйозніша проблема, притаманна мікросервісним системам. Вони виникають, коли відмова одного сервісу тягне за собою помилки в інших сервісах, що залежать від нього. Наприклад, відмова сервісу авторизації може призвести до неможливості обробки запитів у сервісах, що потребують аутентифікації, навіть якщо самі вони працюють справно. У великих системах такі збої можуть швидко поширюватися, призводячи до часткових або повних відмов сервісів і потребують ретельного опрацювання на рівні архітектури.

Для запобігання таким явищам застосовуються спеціалізовані fault-tolerance патерни, які дозволяють системі залишатися частково функціональною навіть за умов часткової відмови компонентів:

Circuit Breaker — шаблон, що розриває ланцюг викликів до сервісу, який постійно відмовляє. Після певного часу система перевіряє, чи сервіс повернувся до нормального стану, і лише тоді знову дозволяє звернення до нього. Це дозволяє уникнути зайвого навантаження на проблемний сервіс і зменшити тиск на систему в цілому.

Timeouts and Retries — встановлення граничного часу очікування відповіді від сервісу, після чого виклик переривається і може бути повторений. При цьому важливо уникати одночасних масових повторів, що ще більше погіршують ситуацію (ефект thundering herd). Для цього додають експоненційні затримки (exponential backoff) з випадковим зсувом (jitter).

Fallback mechanisms — у разі недоступності основного сервісу використовується попередньо закешована відповідь, дані з іншого джерела або повертається значення за замовчуванням. Це дозволяє принаймні частково задовольнити запит користувача без повного відмовлення.

Bulkheads (переборки) — розділення системи на незалежні підсистеми або потоки запитів, щоб обмежити вплив одного сервісу на інші. Наприклад, один API-сервіс може мати окремі ресурси для обробки запитів від мобільного клієнта і від адмін-панелі.

Також критично важливо впроваджувати моніторинг і логування — за допомогою таких систем, як Prometheus, Grafana, Zipkin або Jaeger. Це дозволяє в реальному часі відстежувати ланцюги запитів, виявляти вузькі місця, затримки та точки відмов. Наприклад, трасування розподілених викликів (distributed tracing) дає змогу побачити, на якому саме етапі запит "завис", який сервіс став причиною проблеми, і як це вплинуло на загальну продуктивність.

Ще одним підходом до зменшення впливу каскадних збоїв є впровадження ізоляції сервісів у контейнери або навіть серверлесс-платформи (наприклад, AWS Lambda або Google Cloud Functions), де кожен компонент працює в окремому, ізольованому середовищі з обмеженими ресурсами. Це

дозволяє уникнути ситуацій, коли один сервіс монополізує ресурси і шкодить іншим.

У підсумку, хоча мікросервіси дають велику гнучкість у масштабуванні та розробці, вони також відкривають новий клас технічних ризиків, пов'язаних із розподіленістю. Успішне впровадження мікросервісної архітектури передбачає обов’язкове проектування з урахуванням затримок, таймаутів, ізоляції збоїв, fallback-механізмів та моніторингу — все це має бути передбачено ще до написання першого рядка коду, інакше система вийде крихкою та ненадійною навіть при малій кількості користувачів.

1.2.3 Труднощі узгодженості даних

Однією з найбільш складних і делікатних проблем у мікросервісних системах є забезпечення узгодженості даних між окремими сервісами. У монолітній архітектурі вся логіка працює в межах одного процесу і часто використовує одну централізовану реляційну базу даних, де транзакції реалізуються легко й надійно за допомогою вбудованих механізмів ACID. У мікросервісній архітектурі навпаки — кожен сервіс має власну базу даних, керує своїми даними автономно, а обмін інформацією між сервісами відбувається через мережу. Це означає, що будь-яка операція, яка зачіпає кілька сервісів одночасно, не може покладатися на класичні транзакційні механізми.

Основна проблема полягає в тому, що міжсервісні транзакції складно зробити атомарними. Наприклад, уявімо ситуацію, коли потрібно створити замовлення, яке стосується трьох сервісів: замовлення, оплата та інвентар. Якщо один із сервісів завершить свою частину операції успішно, а інший — з помилкою, то система опиниться в непослідовному стані, що може спричинити серйозні бізнес-проблеми, наприклад списання грошей без формування замовлення або резерв товару без підтвердження оплати.

Щоб боротися з цим, у мікросервісній архітектурі використовуються спеціальні патерни, які не гарантують повну атомарність, але дозволяють зменшити ризики і зберігати eventual consistency — тобто систему, що з часом приходить до узгодженого стану:

Патерн Saga — це найбільш популярне рішення для реалізації розподілених транзакцій. Ідея полягає в тому, що замість однієї великої транзакції використовується серія локальних транзакцій, кожна з яких виконується окремим сервісом. Якщо одна з них завершується з помилкою, викликаються компенсаційні транзакції для відкату змін. Наприклад, якщо після бронювання товару не вдалося списати кошти, сервіс інвентаря отримає команду скасувати бронювання.

Існує два основних варіанти реалізації Saga: оркестраційний (centralized orchestrator координує виконання кроків) та хореографічний (серії подій, на які реагують сервіси самостійно). Перший дає більше контролю, другий — краще масштабується і легше розширюється.

Ще один підхід — Eventual Consistency, коли зміни не відбуваються синхронно, а передаються у вигляді подій (наприклад, через Kafka чи RabbitMQ). Інші сервіси підписані на ці події й реагують на них, оновлюючи свій стан. Таким чином, всі сервіси зрештою синхронізуються, але на деякий час система може бути в неузгодженому стані. Наприклад, сервіс замовлень відправляє подію "OrderPlaced", яку отримують сервіси інвентаря та доставки. Якщо один з них тимчасово недоступний, він обробить подію пізніше, коли відновиться, що забезпечує стійкість до тимчасових збоїв.

Така архітектура потребує обережності, особливо щодо обробки повторюваних подій, ідентифікації транзакцій, гарантії порядку подій (ordering guarantees) та ідемпотентності — здатності обробляти однакові події без побічного ефекту при повторенні.

Ще одним викликом є зміни у структурі даних. Якщо одна служба зберігає дані, які використовуються в іншій, але змінює формат збереження або логіку

бізнес-процесу, то зростає ризик несумісності. Саме тому рекомендується дотримуватися принципу shared-nothing architecture, коли сервіси не звертаються до чужих баз напряму, а отримують лише підготовлену інформацію через API або повідомлення.

Для аналітичних задач або генерації звітності, де важлива повна картина, але не критична швидкість оновлення, зазвичай використовують data lake або data warehouse, куди агрегуються всі події та знімки станів сервісів. Ці сховища періодично оновлюються через ETL або CDC (Change Data Capture) механізми.

На практиці важливо поєднувати архітектурні патерни з технічними механізмами — надійними брокерами подій, ідемпотентними API, журналами подій (event store), тестами на відновлення та спостережуваністю. Інструменти, які підтримують контроль консистентності, мають включати в себе логування всіх важливих подій, сповіщення про відхилення від очікуваного стану та автоматизовані процеси компенсації.

Таким чином, у мікросервісних системах узгодженість даних — це не питання єдиної транзакції, а результат продуманого архітектурного дизайну, відповідального використання подій, чіткої організації взаємодії між сервісами і готовності до помилок. Впровадження цих практик дозволяє створювати стабільні, масштабовані та гнучкі системи навіть за умов високої складності.

1.3 Методи оптимізації продуктивності

1.3.1 Кешування (Redis, Memcached)

Кешування є однією з найефективніших технік оптимізації продуктивності мікросервісних систем, оскільки дозволяє значно зменшити час відповіді сервісів та навантаження на основні бази даних. У мікросервісній архітектурі, де кожен сервіс виконує специфічну функцію, високий рівень

повторюваності запитів (наприклад, на отримання профілю користувача або списку останніх повідомлень) створює ідеальні умови для використання кешу.

Найчастіше в системах використовуються дві популярні кеш-системи: Redis та Memcached. Обидві є in-memory сховищами, але мають суттєві відмінності.

Redis — це сховище ключ-значення, яке підтримує різноманітні структури даних, зокрема хеші, множини, списки, впорядковані множини, бітові поля, геопросторові індекси тощо. Redis також надає підтримку TTL (time-to-live) — часу життя запису, що дає змогу автоматично очищати кеш від застарілої інформації. Крім того, Redis має функції публікації/підписки (pub/sub), механізми блокувань, і навіть можливість використання Lua-скриптів для складних операцій. Його можна використовувати як розподілений кеш або як тимчасове сховище для обробки черг.

Memcached, у свою чергу, більш легковаговий і простий. Він працює тільки з простими ключ-значеннями (строками) та не підтримує складні структури. Завдяки цьому Memcached є дуже швидким та ефективним у системах, де обсяг даних в кеші обмежений, і відсутня потреба у складній логіці зберігання. Він добре підходить для зберігання HTML-фрагментів, запитів до бази даних, або частин JSON-відповідей.

У мікросервісах кеші використовуються на різних рівнях:

- Локальний кеш (наприклад, у пам’яті інстансу сервісу) — забезпечує найшвидший доступ до даних, але має суттєві обмеження у консистентності при масштабуванні: інші інстанси можуть мати вже неактуальну інформацію. Підходить для короткотривалого кешування, як-от налаштувань або обчислень.
- Глобальний кеш (наприклад, Redis-кластер, до якого звертаються всі інстанси) — дає змогу централізовано зберігати дані та підтримувати єдиний актуальний стан. Такий підхід вимагає додаткового захисту від

перевантаження, але дозволяє валідувати кеш з будь-якого сервісу, наприклад, при оновленні користувацьких даних.

Умовно, кешування можна поділити на кешування читання та кешування запису. Перше передбачає, що дані читаються спочатку з кешу, і лише при відсутності — з бази даних. Другий варіант застосовується, коли після запису до бази одночасно оновлюється і кеш (або він скидається). Для цього існують шаблони write-through, write-around та write-behind, кожен з яких має свої переваги й недоліки.

Складнішим аспектом кешування є інвалідація кешу — момент, коли збережені в кеші дані вже неактуальні. Найпоширеніші підходи:

1.Встановлення TTL, після чого дані автоматично видаляються;

2. Активне очищення кешу під час оновлення запису в базі;

3.Оновлення через події або тригери (наприклад, Kafka подія після змін у базі).

Окрім цього, потрібно враховувати ідемпотентність кешованих відповідей та перевіряти, чи не залежить логіка від актуальності мікрозмін. Наприклад, дані про курс валют або статус транзакцій потребують частого оновлення або відмови від кешування взагалі.

Також у Redis доступні розширені функції: блокуючі операції, використання як черги повідомлень, rate-limiting, session storage, або навіть leaderboards (за допомогою Sorted Sets), що робить його не просто кешем, а універсальним високошвидкісним інструментом у системах з високим навантаженням.

Варто враховувати і топологію розгортання: Redis можна запускати у кластерному режимі для горизонтального масштабування, з використанням Sentinel для відмовостійкості. Memcached може працювати в горизонтальній кластеризації, але не має вбудованої реплікації чи збереження даних на диск.

Таким чином, кешування є необхідним інструментом у мікросервісній архітектурі для досягнення високої продуктивності, стабільності й економії ресурсів. Грамотне використання Redis або Memcached, з урахуванням потреб системи, дозволяє зменшити час відповіді на порядки, забезпечуючи при цьому гнучкість і масштабованість.

1.3.2 Шардінг, реплікація, розділення навантаження

У мікросервісних системах, що працюють з великими обсягами даних і високим навантаженням, однією з ключових вимог до баз даних є забезпечення масштабованості й надійної доступності. Для цього активно використовуються техніки шардінгу, реплікації та балансування навантаження.

Шардінг (sharding) — це горизонтальне розділення даних на логічні частини, які розміщуються на окремих фізичних вузлах. Ідея полягає в тому, щоб не зберігати всю таблицю в одній базі, а поділити її за певним критерієм, наприклад, user_id, tenant_id, географічним регіоном або алфавітним діапазоном. У результаті кожен вузол обробляє лише частину даних, що дозволяє розподілити навантаження на зберігання і обробку, а також уникнути "вузьких місць" у продуктивності.

У системах з великою кількістю користувачів найпопулярнішою схемою є range-based шардінг, при якому, наприклад, користувачі з user_id від 1 до 100000 зберігаються в одному шарді, від 100001 до 200000 — в іншому. Альтернативний варіант — hash-based шардінг, який використовує хеш-функцію для рівномірного розподілу записів по шардах. Хоча це забезпечує баланс навантаження, складніше реалізувати запити з фільтрацією по діапазонах.

Реплікація (replication) — це процес створення копій бази даних, які синхронізуються між собою. Вона підвищує доступність і відмовостійкість.

Найпоширеніший варіант — це master-slave (primary-replica) реплікація: один вузол відповідає за записи (master), а інші (slaves) використовуються лише для читання. У PostgreSQL це реалізується через фізичну або логічну реплікацію. У випадку збоїв у master, система може автоматично або вручну переключитися на один із slave-вузлів.

Поєднання шардінгу та реплікації дозволяє одночасно масштабувати як обсяг даних (шардінг), так і обсяг запитів на читання (реплікація). Наприклад, у великій системі підтримки користувачів можна організувати по одному master-інстансу на кожен шард, і до кожного з них підключити кілька read-only реплік для обслуговування GET-запитів.

Балансування навантаження (load balancing) є необхідним механізмом, який дозволяє ефективно розподіляти вхідний трафік між різними сервісами або їх інстансами. У мікросервісах це реалізується на кількох рівнях:

- На рівні API Gateway або Load Balancer (наприклад, NGINX, HAProxy, Envoy), який розподіляє вхідні запити між доступними інстансами одного й того ж сервісу на основі правил round-robin, least-connections або з урахуванням навантаження на CPU/RAM;
- На рівні клієнта (client-side load balancing) — клієнтська бібліотека самостійно вибирає, до якого інстансу звертатися (наприклад, Netflix Ribbon);
- DNS-based балансування — кілька IP-адрес для одного домену, що дозволяє розподіляти трафік географічно.

При роботі з балансувальниками слід враховувати стан сесії. Якщо сесія зберігається на стороні сервера, слід реалізувати механізм sticky-sessions (прив’язка клієнта до конкретного інстансу), або зберігати сесію у спільному сховищі (наприклад, Redis).

Також важливо відзначити роль Kubernetes Ingress Controller, який є гнучким рішенням для балансування в кластері Kubernetes. Він забезпечує

маршрутизацію HTTP/S запитів, TLS-шифрування, переписування URL, rate limiting та багато іншого.

Важливо грамотно поєднувати всі ці техніки. Наприклад, у високонавантаженій системі обробки транзакцій можна шардити таблиці транзакцій за user_id, використовувати master-slave реплікацію для масштабування запитів на читання, і балансувальник трафіку для рівномірного розподілу API-запитів між сервісами. Такий підхід дозволяє обробляти десятки тисяч запитів на секунду зі збереженням високої доступності та стійкості до відмов.

Ключовими викликами при реалізації таких рішень є: забезпечення консистентності даних при реплікації, уникнення "hot shards" при поганому виборі шардінг-ключа, та дотримання SLA (наприклад, час відповіді < 300 мс). Саме тому оптимізація бази даних і трафіку є критично важливою частиною архітектури сучасних мікросервісних систем.

1.3.3 Асинхронна взаємодія та брокери повідомлень

Асинхронна взаємодія — одна з найважливіших практик в оптимізації мікросервісної архітектури, особливо в умовах високого навантаження або коли сервіси мають виконувати довготривалі або нерегулярні задачі. Основна ідея полягає в тому, щоб розділити логіку «ініціації» події та її «обробки» в часі, щоб не блокувати основний запит або не створювати залежності між сервісами в реальному часі. Це дозволяє досягти кращої масштабованості, стійкості до збоїв і більшої гнучкості при модифікації системи.

У мікросервісній архітектурі асинхронність досягається через використання черг повідомлень або систем потокової обробки, які виступають у ролі посередника між сервісами. Серед найпопулярніших рішень — RabbitMQ, Apache Kafka, Amazon SQS, Google Pub/Sub, NATS.

Наприклад, у типових сценаріях користувач створює замовлення через API, і сервіс обробки замовлень одразу підтверджує прийняття, але сама обробка (платіж, оновлення складу, відправка листа, сповіщення у мобільний додаток) виконується іншими мікросервісами у фоновому режимі. Кожен з цих мікросервісів підписується на певну чергу або тему подій (topic), які надходять у систему.

RabbitMQ — це класичний брокер повідомлень, що підтримує routing ключі, черги, підтвердження отримання повідомлень, пріоритезацію та затримку доставки. Він особливо добре підходить для транзакційних або фінансових систем, де важлива гарантована доставка повідомлень. RabbitMQ підтримує різні шаблони взаємодії: точка-точка (point-to-point), fan-out, topic exchange.

Apache Kafka — це розподілена стрімінгова платформа, яка підходить для обробки великих обсягів подій у реальному часі. Її перевага в здатності обробляти сотні тисяч повідомлень за секунду, з високою надійністю та масштабованістю. Kafka дозволяє сервісам читати події з певного моменту в часі, зберігаючи історію, що є критичним у випадках аналітики, аудиту або відновлення після збоїв.

Однією з основних проблем при впровадженні асинхронної взаємодії є управління ідентитикою подій, уникнення дублювання (наприклад, через повторні спроби доставки) та забезпечення ідемпотентності обробників — здатності багаторазово обробляти подію без зміни результату.

Також важливо мати централізовану систему логування подій — наприклад, через Kafka Connect з інтеграцією в Elasticsearch або ClickHouse, щоб можна було відслідковувати хід обробки кожної події. Це дозволяє будувати аудит, а також діагностувати, де саме відбувся збій у ланцюжку.

Ще одна практика — затримані черги (delayed queues), коли повідомлення обробляється лише після певного часу. Це корисно, наприклад, для перевірки, чи користувач активував акаунт протягом 24 годин після реєстрації.

У системах із великою кількістю повідомлень також важливо використовувати batch-процесинг: сервіс-споживач може обробляти події не поодинці, а пачками по 100–1000 штук, що знижує навантаження на мережу та базу даних.

Асинхронна архітектура дозволяє знизити щільність залежностей між сервісами, зробити систему більш еластичною до навантажень, і мінімізувати вплив збоїв одного сервісу на інші. Водночас вона ускладнює дебаг, створює потребу в retry-логіці, моніторингу черг, обробці "dead-letter" повідомлень (які не вдалося обробити) і вимагає злагоджених правил роботи з подіями на рівні всієї команди.

У результаті, використання брокерів повідомлень і асинхронної моделі — це одна з найважливіших частин сучасної високопродуктивної мікросервісної системи. Її правильне впровадження суттєво покращує масштабованість і надійність сервісів, особливо у складних бізнес-процесах з великою кількістю взаємодій.

1.3.4 API Gateway і service mesh

У складній мікросервісній архітектурі з десятками або сотнями сервісів виникає потреба у централізованому управлінні доступом, маршрутизацією, автентифікацією, авторизацією та контролем трафіку. Саме для цих цілей застосовуються компоненти API Gateway і service mesh, які виступають ключовими посередниками між клієнтом і мікросервісною екосистемою.

API Gateway — це єдина точка входу для всіх зовнішніх запитів. Замість того, щоб клієнт взаємодіяв безпосередньо з кожним сервісом, він надсилає запити через шлюз, який маршрутизує їх до відповідного сервісу. Це дозволяє не лише спростити структуру зовнішніх API, але й централізовано реалізовувати такі функції, як кешування відповідей, обмеження кількості запитів (rate limiting), автентифікація (JWT, OAuth2), логування,

трансформація запитів (наприклад, перетворення GraphQL у REST або навпаки), а також збирання статистики для аналітики.

Популярні реалізації API Gateway — це Kong, NGINX, Traefik, AWS API Gateway та Apigee. У мікросервісах, які розгортаються в Kubernetes, часто застосовують Ingress-контролери з роллю API Gateway, які керують зовнішнім трафіком до кластеру.

Прикладом є ситуація, коли клієнт мобільного застосунку робить запит на отримання профілю користувача. API Gateway перевіряє токен авторизації, логує запит, виконує обмеження швидкості, трансформує запит, додає службові заголовки, маршрутизує його до сервісу UserProfile, а відповідь кешує на 10 секунд — усе це відбувається без змін у коді мікросервісу.

Service mesh, у свою чергу, вирішує проблему комунікації всередині мікросервісного середовища. У великих системах, де сервіси спілкуються один з одним десятками тисяч разів на секунду, необхідно контролювати мережеву поведінку: забезпечити шифрування TLS, балансування навантаження, автоматичне виявлення сервісів, трасування запитів, а також політики доступу. Усе це реалізується без змін у коді самих сервісів — за рахунок проксі-сервісів (sidecar containers), які розгортаються разом із кожним мікросервісом.

Найвідомішими платформами service mesh є Istio, Linkerd, Consul і Kuma. Вони дозволяють визначати правила маршрутизації за допомогою декларативної конфігурації, наприклад: "20% трафіку на нову версію сервісу", або "запити з регіону eu-central-1 мають обслуговуватись тільки сервісами в тому ж регіоні". Також забезпечують автоматичне збирання метрик, логів та трасувань через інтеграцію з Prometheus, Grafana, Jaeger чи Zipkin.

Ще однією перевагою є вбудоване шифрування TLS між сервісами, яке дозволяє захистити внутрішній трафік навіть у випадку проникнення у кластер. За допомогою service mesh також можна реалізувати retry-механізми,

circuit breaker, timeout-и — раніше ці речі приходилось писати вручну в коді кожного сервісу.

Однак впровадження таких рішень не позбавлене викликів. Service mesh значно ускладнює інфраструктуру: додає накладні витрати на пам’ять і процесор через проксі, потребує знання YAML-конфігурацій, DevOps-практик та інтеграцій з CI/CD. При неправильному налаштуванні можна створити конфлікти маршрутів або затримки в обробці.

У поєднанні, API Gateway і service mesh забезпечують повний контроль над мережею — як на зовнішньому, так і на внутрішньому рівні, даючи можливість централізовано управляти доступом, безпекою, продуктивністю та стабільністю всієї мікросервісної системи.

1.4 Огляд ролі баз даних у мікросервісних системах

1.4.1 Database per service vs спільна база

Один із ключових принципів мікросервісної архітектури — незалежність. Це означає, що кожен мікросервіс повинен бути максимально автономним, включаючи власне джерело збереження даних. Такий підхід відомий як Database per service, і він полягає в тому, що кожен мікросервіс має свою власну базу даних, яка не розділяється з іншими сервісами.

Це дозволяє сервісу самостійно контролювати схему даних, незалежно еволюціонувати, оновлювати модель даних без ризику для інших частин системи. З точки зору розробки, це спрощує підтримку, тестування і деплой. Команди можуть працювати паралельно, не очікуючи змін від суміжних сервісів. Також зменшується ймовірність ситуацій, коли один сервіс ламає функціональність іншого через зміну структури таблиць у спільній базі.

Кожен мікросервіс у такій архітектурі може навіть використовувати різні типи СУБД. Наприклад, сервіс користувачів може працювати на PostgreSQL, сервіс аналітики — на ClickHouse, а сервіс логування — на MongoDB. Це дозволяє максимально ефективно обирати інструмент для конкретної задачі.

Проте повна ізоляція БД створює нові виклики, особливо коли потрібно реалізувати зв’язки між даними різних сервісів. У моноліті можна просто зробити JOIN між таблицями. У мікросервісах це не так просто: для отримання повних даних часто доводиться робити послідовні запити до кількох API, що збільшує затримки і складність логіки. А кешування таких запитів створює нові виклики щодо узгодженості даних.

З іншого боку, існує підхід зі спільною базою даних для кількох сервісів. Така модель порушує межі відповідальності сервісів, адже кілька з них можуть змінювати ті самі таблиці. Це створює щільну зв’язаність між сервісами, що прямо суперечить принципам мікросервісної архітектури. Такий підхід може бути виправданий на початкових етапах побудови системи, коли сервісів небагато і потрібна швидка реалізація, або в системах, де відокремлення даних є занадто дорогим.

Найчастіше, у великих системах застосовують гібридний підхід: операційна частина системи (робота з користувачами, транзакціями, логікою) розділена на окремі бази, але існує централізоване сховище для аналітики — Data Warehouse, яке агрегує дані з усіх сервісів для звітності та аналізу. Для цього використовуються ETL-процеси, які регулярно витягують, трансформують і завантажують дані у спеціальні бази, такі як Amazon Redshift, Google BigQuery або Snowflake.

Таким чином, вибір між підходами залежить від розміру проєкту, складності бізнес-логіки, вимог до узгодженості даних і швидкості розробки. У більшості випадків, Database per service є цільовим варіантом, який забезпечує кращу ізоляцію, масштабованість і гнучкість, хоча й потребує впровадження додаткових рішень для обміну даними між сервісами.

1.4.2 Патерни доступу: CQRS, Event Sourcing

У мікросервісній архітектурі важливо забезпечити як продуктивну обробку запитів, так і коректну фіксацію змін. Одним із підходів до вирішення цієї задачі є застосування шаблонів проектування, таких як CQRS (Command Query Responsibility Segregation) і Event Sourcing.

CQRS передбачає розділення моделі читання і моделі запису. Це означає, що дані, які використовуються для оновлення стану (commands), і дані, які витягуються для читання (queries), обробляються окремими шляхами. Такий підхід дозволяє масштабувати запити на читання незалежно від оновлень, використовувати різні сховища даних для кожного типу операцій, і зменшити навантаження на транзакційні бази.

Наприклад, для сервісу замовлень можна зберігати актуальний стан замовлення в одній БД, а для побудови звітів або перегляду історії замовлень використовувати іншу оптимізовану структуру. Це також дозволяє мати різні API для читання та запису, що зручно при побудові UI або аналітичних панелей.

Event Sourcing — це підхід, при якому зміни стану системи не зберігаються у вигляді актуального стану, а у вигляді послідовності подій. Наприклад, замість того щоб просто записати order_status = delivered, система зберігає подію OrderDeliveredEvent. Актуальний стан формується шляхом послідовного програвання всіх подій у порядку їх виникнення.

Це надає велику гнучкість — можна відновити будь-який попередній стан системи, переглянути історію змін, реалізувати відкладену обробку або запуск додаткових процесів на основі подій. Також події можна публікувати до інших сервісів, інтегруючи їх через event bus (наприклад, Kafka або NATS), що дуже корисно в мікросервісному середовищі.

Однак ці підходи потребують ретельної реалізації: для CQRS — синхронізації моделей читання та запису, для Event Sourcing — збереження подій у надійних журналах і підтримки механізмів відтворення стану. Часто їх

застосовують разом, що дозволяє досягти високої продуктивності, масштабованості та повної відтворюваності даних.

1.4.3 Підтримка транзакцій у розподіленому середовищі

Однією з найбільших складностей при впровадженні мікросервісної архітектури є забезпечення узгодженості даних між різними сервісами. У монолітних системах транзакції зазвичай реалізуються за допомогою вбудованих механізмів СУБД, які забезпечують властивості ACID (атомарність, узгодженість, ізольованість, довговічність). Проте в мікросервісах, де кожен компонент часто має власну базу даних, традиційний підхід з глобальними транзакціями втрачає ефективність і ускладнюється технічно.

Одним із поширених рішень є підхід Saga. Це шаблон управління розподіленими транзакціями, при якому загальна операція розбивається на послідовність локальних транзакцій. Після завершення кожної локальної транзакції Saga ініціює наступну. Якщо на якомусь етапі виникає помилка, запускаються так звані компенсаційні дії, які повертають систему до попереднього узгодженого стану. Наприклад, якщо користувач оформив замовлення, але оплата не пройшла, сервіс оплати може надіслати повідомлення про скасування замовлення, яке буде оброблене окремим сервісом.

Існують дві реалізації Saga — оркестровані та хореографовані. В оркестрованій Saga один центральний сервіс-координатор керує всіма викликами і переходами між транзакціями. У хореографованій Saga сервіси реагують на події один одного, тобто дії тригеряться через події у брокері повідомлень. Хореографія забезпечує слабку зв'язаність, але ускладнює відстеження потоку виконання.

Крім того, іноді використовуються так звані двофазні коміти (2PC), однак вони не рекомендуються в мікросервісах через складність масштабування, зниження відмовостійкості і можливі блокування ресурсів. 2PC потребує наявності загального координатора, який фіксує або відхиляє транзакцію після фази підготовки, але у випадку збою цей механізм може «зависнути».

Для систем із високими вимогами до узгодженості — наприклад, у фінансовій сфері — важливо впроваджувати надійне журналювання транзакцій. Це означає, що всі важливі дії записуються в окремі логи або таблиці аудиту, де зберігається історія операцій, включаючи їхній статус, час, автора та результат. Це дозволяє проводити форензічні розслідування, відновлювати дані вручну у випадку збоїв і гарантувати дотримання нормативних вимог.

На практиці, мікросервісна архітектура змушує архітекторів шукати баланс між узгодженістю (consistency), доступністю (availability) та стійкістю до розділення (partition tolerance) відповідно до теореми CAP. У реальних розробках частіше надають перевагу eventual consistency (остаточній узгодженості), коли допущена тимчасова розбіжність стану сервісів, яка згодом буде синхронізована подіями або окремими процесами.

Таким чином, управління транзакціями у мікросервісах вимагає змін парадигми: замість жорсткого контролю стану — використання подій, компенсуючих дій та надійного журналювання. Це дозволяє будувати гнучкі, масштабовані та відмовостійкі системи, здатні функціонувати в умовах розподіленого середовища.

1.4.4 Оптимізація запитів та індексація

У мікросервісній архітектурі, де кожен сервіс має власну базу даних, продуктивність запитів безпосередньо впливає на загальну швидкодію системи. На відміну від монолітних систем, тут кожен мікросервіс несе повну

відповідальність за оптимальність своїх запитів, і погано спроєктований SQL-запит у одному сервісі може спричинити затримки або надмірне навантаження на інші частини системи.

Одним із найважливіших інструментів оптимізації є індекси. Їх доцільно застосовувати на тих полях, які найчастіше використовуються у WHERE-умовах, JOIN-операціях або при сортуванні (ORDER BY). Наприклад, у сервісах, що працюють з користувачами, типовими кандидатами для індексування є user_id, email, created_at. У сервісах, де відбуваються масові транзакції або перегляди, — product_id, status, timestamp.

Правильна стратегія індексації може зменшити час відповіді на запити з кількох секунд до часток секунди. Проте зловживання індексами може спричинити зворотний ефект — надмірне використання пам’яті та зниження продуктивності при записах (INSERT, UPDATE, DELETE), оскільки кожна зміна в таблиці потребує оновлення індексів. Тому індекси слід додавати з урахуванням реальних сценаріїв використання, а не «на всяк випадок».

Для діагностики продуктивності запитів важливо регулярно використовувати EXPLAIN ANALYZE (у PostgreSQL) або аналогічні інструменти в інших СУБД. Вони дозволяють побачити план виконання запиту — які індекси використовуються, скільки рядків опрацьовується, які типи сканування застосовуються (seq scan, index scan тощо). Це особливо важливо при роботі з великими об’ємами даних: іноді замість очікуваного використання індексу відбувається повне сканування таблиці, що призводить до значного падіння продуктивності.

Ще однією поширеною практикою є застосування посторінкової навігації (pagination) за допомогою LIMIT та OFFSET. Проте цей підхід не завжди ефективний при роботі з великими таблицями, особливо на глибоких сторінках. У таких випадках доцільніше використовувати так звану seek-pagination (або keyset pagination), де замість OFFSET запити фільтруються по значенню останнього елемента попередньої сторінки (наприклад, WHERE id >

last_seen_id LIMIT 20). Такий підхід дозволяє уникнути сканування всіх попередніх рядків і значно зменшує час відповіді.

Оптимізація також включає розбиття складних запитів на кілька простіших або винесення частини обробки на рівень кешу чи бекграундних процесів. Наприклад, генерація аналітичних звітів або обробка важких агрегатів (COUNT, SUM, GROUP BY) може бути делегована окремим процесам або попередньо підготовлена й закешована.

Із розвитком системи й накопиченням даних важливо впроваджувати архівацію старих записів, якщо вони не потрібні в щоденному використанні. Це може бути реалізовано через розділення таблиць, наприклад: поточна активна таблиця — orders_current, архівна — orders_archive. Таке розділення дозволяє підтримувати швидкодію операційної частини системи без шкоди для історичних даних.

Також для деяких випадків ефективною є стратегія матеріалізованих уявлень (materialized views). Це спеціальні таблиці, які зберігають попередньо обчислені результати запитів. Їх можна оновлювати періодично або за подією, й вони значно прискорюють обробку повторюваних аналітичних запитів.

Важливо не тільки проєктувати запити та індекси на початковому етапі, але й регулярно проводити аудит запитів у продуктивному середовищі. Іноді поведінка запитів у тестовій базі з малим обсягом даних не відображає їхню реальну продуктивність. Для цього використовують профайлер запитів, логування slow queries, або інструменти на кшталт pg_stat_statements (у PostgreSQL), які дозволяють виявити найповільніші або найчастіше виконувані запити.

Таким чином, систематична оптимізація запитів, правильне індексування та використання сучасних патернів роботи з базами даних — це фундамент для забезпечення надійної та масштабованої мікросервісної архітектури, здатної обробляти великі об’єми трафіку без втрати продуктивності.

ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі було здійснено ґрунтовний аналіз мікросервісної архітектури, з акцентом на її особливості, переваги, виклики масштабування, продуктивності та підходи до оптимізації. Було детально охарактеризовано основні принципи побудови мікросервісних систем, які базуються на поділі великої монолітної системи на невеликі незалежні компоненти з чітко визначеними зонами відповідальності. Такий підхід забезпечує високу модульність, гнучкість в оновленнях, легкість масштабування та незалежність у розробці окремих компонентів.

Порівняння з монолітною архітектурою дозволило виділити ключові відмінності, які пояснюють популярність мікросервісів у сучасному розробницькому середовищі. Зокрема, мікросервіси забезпечують більш гнучку та масштабовану розробку, але разом із тим вимагають складніших підходів до інтеграції, комунікації, тестування та моніторингу. У той час як монолітні системи мають простішу структуру і легше розгортаються на ранніх етапах, вони значно поступаються мікросервісам у питаннях еволюційного розвитку проєктів.

Розглянуто типові проблеми, з якими стикаються мікросервісні архітектури: складність горизонтального та вертикального масштабування, затримки в комунікаціях між сервісами, каскадні збої, а також труднощі з досягненням узгодженості даних між розподіленими компонентами. Було показано, що для мінімізації цих проблем активно застосовуються fault-tolerant механізми (таймаути, circuit breaker, fallback), патерни розподілених транзакцій (наприклад, Saga), а також асинхронні канали обміну повідомленнями.

Особливу увагу приділено методам оптимізації продуктивності мікросервісів. Детально описано такі підходи як кешування (з використанням Redis або Memcached), реалізація шардінгу, реплікації та балансування

навантаження для масштабування БД, застосування асинхронної взаємодії за допомогою брокерів повідомлень (Kafka, RabbitMQ), а також використання API Gateway та service mesh для централізованого управління мережею сервісів і маршрутизації запитів.

У підрозділах, присвячених базам даних, розкрито важливість ізоляції сховищ для кожного сервісу, застосування підходів CQRS та Event Sourcing, а також складності підтримки транзакцій у розподіленому середовищі. Було показано, що індексація, профілювання запитів, архівація старих даних, застосування seek-pagination та materialized views — це ключові техніки, що дозволяють зберігати стабільну продуктивність навіть при значному зростанні навантаження.

Таким чином, можна зробити висновок, що мікросервісна архітектура є ефективним інструментом побудови масштабованих та гнучких систем, однак вимагає продуманої інженерної практики на всіх рівнях — від розробки бізнес-логіки до проєктування БД, інтеграцій та інфраструктурної підтримки. Оптимізація в таких системах повинна бути систематичною, з урахуванням технічних і бізнес-вимог, та базуватись на конкретних метриках ефективності.

У наступному розділі буде зроблено акцент на аналізі сучасних технологій та підходів для впровадження RAG-моделі (Retrieval-Augmented Generation), яка дозволяє значно підвищити ефективність обробки запитів до знань, зокрема в контексті мікросервісних систем підтримки. Буде розглянуто принципи роботи RAG, типи баз знань, підходи до їх інтеграції, а також інструменти, які дозволяють поєднати можливості великих мовних моделей з зовнішніми джерелами даних.

РОЗДІЛ 2. ОГЛЯД ТЕХНОЛОГІЙ ТА МОДЕЛЕЙ ДЛЯ ВПРОВАДЖЕННЯ RAG-ПІДХОДУ

2.1 Поняття та принципи роботи Retrieval-Augmented Generation

Далі буде детально описано роботу RAG систем.

2.1.1 Основи концепції RAG

Retrieval-Augmented Generation (RAG) — це підхід до обробки природної мови, який поєднує можливості великих мовних моделей (LLM) з доступом до зовнішніх джерел інформації, зазвичай у вигляді баз знань або сховищ документів. У традиційних LLM, таких як GPT або BERT, відповіді генеруються на основі параметрів, сформованих у процесі навчання на великих корпусах тексту. Однак ці моделі мають обмеження, пов’язані з відсутністю механізму оновлення знань без повного перенавчання. Саме тут на допомогу приходить підхід RAG, який дає змогу здійснювати запити до зовнішніх джерел і комбінувати результати пошуку з генеративними можливостями моделей.

Суть RAG полягає в тому, що перед генерацією відповіді модель спершу здійснює пошук релевантної інформації, яку потім включає в контекст для формування відповіді. Для цього використовуються векторні бази даних, де зберігаються попередньо оброблені текстові документи у вигляді векторів ознак. При надходженні запиту система генерує вектор запиту, порівнює його з наявними векторами в базі, і відбирає найбільш релевантні документи. Далі ці фрагменти додаються до prompt'у генеративної моделі.

Таким чином, підхід RAG дозволяє подолати низку обмежень класичних мовних моделей, таких як неможливість оперувати з актуальними або

вузькопрофільними даними. Наприклад, при використанні RAG можна отримати відповіді на питання з документів, які були додані до бази даних вже після того, як модель була натренована. Крім того, це значно знижує ймовірність генерації вигаданих або неточних фактів, що часто трапляється в мовних моделях, які працюють без контекстуальних джерел.

Однією з важливих властивостей концепції RAG є її модульність. Компоненти, відповідальні за пошук (retrievers) та генерацію (generators), можуть змінюватися або оновлюватися незалежно одне від одного. Наприклад, можна покращити якість пошуку шляхом заміни алгоритму векторного зіставлення або використання кращої моделі ембеддингів, не змінюючи при цьому генеративну модель. Це дозволяє гнучко налаштовувати систему під потреби конкретного проєкту — від відповіді на юридичні запити до створення персоналізованих рекомендацій.

У цілому концепція Retrieval-Augmented Generation є важливим кроком у напрямку побудови «гібридних» інтелектуальних систем, які поєднують найкраще з двох світів — здатність до генерації природного тексту та доступ до структурованих і напівструктурованих джерел знань. У наступних підрозділах буде розглянуто, як саме реалізуються ці компоненти, якими можуть бути варіанти архітектури RAG-систем і як її переваги реалізуються на практиці.

2.1.2 Архітектура RAG: компоненти та взаємодія

Архітектура системи Retrieval-Augmented Generation (RAG) ґрунтується на чіткому поділі функціональних обов’язків між двома основними блоками: модулем отримання інформації (retriever) і модулем генерації (generator). Ця розділеність дозволяє гнучко налаштовувати поведінку системи, змінювати окремі компоненти без впливу на загальну структуру, а також легко масштабувати окремі елементи при збільшенні навантаження.

Модуль retriever виконує функцію пошуку релевантної інформації у векторному сховищі. Для цього документи або їх фрагменти попередньо трансформуються у векторні подання за допомогою ембеддинг-моделей. Після отримання запиту від користувача текст цього запиту також перетворюється на вектор. Далі виконується пошук найбільш схожих векторів у базі даних за допомогою метрик подібності, наприклад, косинусної відстані. Результатом цього етапу є обмежений набір документів або текстових фрагментів, які, на думку системи, є найбільш релевантними запиту.

Ці документи передаються до генератора — другої ключової частини архітектури RAG. Generator, зазвичай представлений великою мовною моделлю (наприклад, GPT, BERT, Falcon, Claude тощо), отримує від retriever'а не лише початковий запит користувача, але і фрагменти тексту з бази знань. На основі цієї комбінації вона формує відповідь. Особливістю такого підходу є те, що модель не потребує знати все наперед — вона працює з релевантним контекстом, що знижує навантаження на параметри моделі і дає змогу уникати небажаної галюцинації фактів.

Існують два основні типи архітектурних моделей для реалізації RAG: pipeline RAG і end-to-end RAG. У pipeline-підході компоненти retriever і generator існують окремо і обмінюються даними через API або окремі виклики, що робить систему більш прозорою та керованою. Такий підхід зручний для відлагодження, розширення та інтеграції з іншими сервісами. У свою чергу, end-to-end модель передбачає об'єднання обох функцій у межах однієї навченої архітектури. Наприклад, існують моделі, в яких пошук та генерація виконуються єдиним блоком, що дозволяє досягти вищої швидкості, але зменшує контроль за кожним етапом обробки.

Крім основних блоків, RAG-системи часто містять допоміжні компоненти: індексатор, який відповідає за попередню обробку документів; postprocessor — модуль, який виконує фільтрацію або переранжування результатів; middleware або orchestrator — системний шар, який забезпечує

правильну послідовність виконання дій між компонентами. У складних випадках додатково використовуються кеші, системи логування, трейсінг, а також модулі для валідації відповідей (наприклад, факт-чекінг або змістова перевірка).

Усе це створює можливість побудови потужної та масштабованої архітектури, яка може працювати як у режимі online-запитів (наприклад, чат-бот або асистент), так і в batch-режимі (наприклад, створення аналітичних звітів). Гнучкість архітектури дозволяє адаптувати систему під різні вимоги бізнесу та домену, забезпечуючи баланс між швидкістю, точністю та вартістю обчислень.

2.1.3 Порівняння RAG з класичними підходами генерації

Класичні підходи генерації тексту на основі LLM передбачають, що вся необхідна інформація для формування відповіді вже міститься у вагах моделі. Це означає, що перед початком використання такої системи потрібно навчити або донавчити модель на відповідному датасеті, що охоплює всі можливі сценарії, знання й контексти. Цей підхід має свої переваги — модель генерує швидкі відповіді без зовнішніх запитів і може працювати автономно, навіть без доступу до Інтернету чи баз даних. Проте він також має суттєві обмеження.

Одним із головних недоліків класичних підходів є обмежена актуальність знань. Наприклад, якщо модель навчена на даних до 2022 року, вона не матиме інформації про події або зміни, що сталися після цього. Крім того, моделі мають обмеження на обсяг знань, які вони можуть утримувати, що означає втрату частини важливої або нішевої інформації. Актуалізація знань через перенавчання є дорогою, тривалою і ресурсозатратною процедурою.

Retrieval-Augmented Generation вирішує ці проблеми за рахунок зовнішнього джерела знань. Завдяки модулю retriever система може звертатися до свіжої інформації в базі знань і лише після цього запускати генерацію відповіді. Це дозволяє поєднати найкраще з двох світів: високу гнучкість та

актуальність даних, властиву пошуковим системам, і природність мовного моделювання, притаманну LLM.

Ще однією перевагою RAG є прозорість і керованість системи. У класичних LLM-випадках практично неможливо точно пояснити, чому модель видала саме таку відповідь. У RAG-системах, навпаки, можна точно відстежити, які документи було знайдено на етапі пошуку, і як вони вплинули на генерацію. Це особливо важливо в галузях, де критична верифікованість: медицина, право, наукові дослідження.

Крім цього, RAG дозволяє розділяти оновлення знань і генерацію. Замість перенавчання всієї моделі, достатньо оновити векторну базу або ембеддинги нових документів. У результаті система стає динамічною і легше адаптується до змін у світі або в бізнес-контексті.

У підсумку, класичні LLM підходи можуть бути ефективними у випадках з обмеженим і стабільним обсягом знань, де не потрібна часта актуалізація або перевірка фактів. Натомість RAG-підхід стає оптимальним вибором для більшості практичних застосувань, де потрібна інтеграція з базами даних, складна логіка пошуку або персоналізація відповідей. Саме тому сучасні системи штучного інтелекту, які орієнтовані на точність, пояснюваність і масштабованість, дедалі частіше переходять до використання RAG як основної архітектури.

2.2 Моделі генерації з доступом до зовнішніх джерел знань

Retrieval-Augmented Generation — це не лише про пошук, але й про тісну інтеграцію зі складними мовними моделями, які відповідають за формування змістовного, логічного і стилістично узгодженого тексту. У цьому контексті саме генеративна модель відіграє роль останнього етапу, перетворюючи отриману з бази знань інформацію на готову відповідь. Успіх RAG-системи

напряму залежить від якості взаємодії між retriever та генератором, а також від того, наскільки грамотно побудовано prompt або chain-of-thought для LLM.

2.2.1 Генеративні трансформери як основа RAG

Основою будь-якої сучасної системи RAG є генеративна трансформер-модель. Найбільш поширеними прикладами таких моделей є GPT-3.5/4 від OpenAI, Claude від Anthropic, Mistral, LLaMA або Falcon від HuggingFace. Ці моделі мають потужні можливості генерації на основі текстового вводу, що дає змогу створювати детальні та узгоджені відповіді, резюме, пояснення тощо.

У контексті RAG трансформер відіграє роль інтерпретатора знайдених документів: він не просто повторює текст, а намагається синтезувати новий зміст на його основі. Наприклад, якщо retriever повернув кілька фрагментів про юридичний термін, трансформер може згенерувати визначення, що враховує всі джерела. Це особливо корисно при обробці суперечливої або неоднозначної інформації.

Однією з ключових переваг використання генеративних моделей у RAG є їх здатність до reasoning (міркування), тобто побудови логічного ланцюжка для відповіді. Наприклад, трансформер може не лише взяти цитату з тексту, а й переформулювати її, спростити або подати у вигляді списку.

Важливо також відзначити роль контекстного вікна (context window). Чим більше токенів модель може приймати на вхід, тим більше фрагментів документації або результатів пошуку можна врахувати. GPT-4-32k, Claude 2.1 (200k+ токенів) або Gemini з розширеним вікном забезпечують змогу працювати з великими документами без попереднього розбиття.

2.2.2 Поєднання генерації та пошуку: стратегія «retrieve-then-generate»

Однією з основних стратегій у RAG є підхід «retrieve-then-generate», коли спочатку здійснюється векторний пошук релевантних фрагментів, після чого вони передаються генеративній моделі для створення відповіді. Така послідовність забезпечує кращу актуальність і релевантність результатів, ніж класичні генеративні LLM.

Перевага цього підходу — у чіткому розділенні відповідальності між компонентами. Retriever відповідає за швидкий та точний пошук релевантного контенту, а генератор — за побудову повної, стилістично правильної та логічної відповіді. Водночас це спрощує дебаг: якщо відповідь виявилась неправильною, можна проаналізувати, на якому етапі виникла помилка — на пошуку чи генерації.

Існують і більш складні варіації: multi-hop retrieval (послідовне уточнення запиту), hybrid retrieval (поєднання векторного і класичного BM25 пошуку), а також змішані підходи, де генератор може використовувати результати retriever у вигляді chain-of-thought reasoning.

2.2.3 Застосування контекстного пошуку в інтерактивних системах

В інтерактивних системах, таких як чат-боти, голосові асистенти або системи підтримки користувачів, важливу роль відіграє збереження контексту між повідомленнями. Для цього в RAG часто застосовують контекстний пошук — коли retriever враховує не лише останній запит користувача, а й попередні фрази діалогу.

Наприклад, якщо користувач запитує: «А хто її чоловік?» — система має проаналізувати попередні повідомлення і зрозуміти, про кого йдеться. Такий підхід вимагає збереження розширеної історії діалогу та повторного обчислення релевантності з урахуванням контексту. Це можливо завдяки використанню memory chains (в LangChain), історії запитів або session-based embedding.

Ще один підхід — це адаптивна побудова prompt’ів, коли попередні результати пошуку та відповіді включаються у новий запит до retriever. Наприклад, LangChain дозволяє передавати історію в retriever як частину system prompt, що дає змогу зберегти цілісність діалогу.

Контекстний пошук суттєво підвищує якість взаємодії з системою, адже дозволяє зберігати логічну цілісність розмови. Це особливо важливо у навчальних, юридичних, медичних та аналітичних системах, де кожен наступний запит тісно пов’язаний із попереднім.

2.3 Бази даних як джерело знань для RAG

2.3.1 Векторні бази даних: принципи та переваги

Векторні бази даних (vector databases) стали невід’ємною частиною реалізації RAG-систем, оскільки саме вони забезпечують основну функціональність — швидкий пошук релевантної інформації за векторними подібностями. Кожен фрагмент тексту (документ, абзац, речення) попередньо кодується у вектор за допомогою ембеддинг-моделі. Цей вектор представляє змістовну суть тексту у багатовимірному просторі, де близькість між векторами відображає семантичну схожість між фрагментами.

Коли користувач ставить запит, система також кодує його у вектор, після чого шукає у базі даних найближчі сусідні вектори (nearest neighbors), тобто ті документи, які за змістом найбільш подібні до запиту. Таким чином, векторні БД є ключовим компонентом retriever’а в архітектурі RAG.

Серед переваг векторних БД слід відзначити:

1. Швидкість пошуку. Завдяки використанню структур на зразок HNSW (Hierarchical Navigable Small World) або IVF (Inverted File Index), такі бази можуть ефективно працювати навіть із мільйонами векторів.

2. Масштабованість. Векторні бази спроєктовані для горизонтального масштабування, підтримують шардінг, реплікацію та високонавантажені запити.

3. Підтримка фільтрації. Багато платформ дозволяють використовувати метаінформацію (наприклад, мітки, категорії, мову документа) для обмеження результатів пошуку. Це дає змогу побудувати складні запити типу “знайти релевантні документи лише українською мовою, написані після 2023 року”.

4. Інтеграція з LLM. Сучасні векторні бази мають готові SDK та API для взаємодії з OpenAI, Cohere, HuggingFace та іншими LLM-платформами, що значно полегшує реалізацію RAG-застосунків.

До найбільш популярних систем належать Pinecone, Qdrant, Weaviate, Milvus, Chroma. Наприклад, Qdrant має повну підтримку гібридного пошуку, асинхронного індексування, типовий REST API і можливість використання локально чи в хмарі. Pinecone вирізняється високою продуктивністю і глибокою інтеграцією з OpenAI.

Векторні БД активно розвиваються, і сьогодні їх можливості виходять за межі простого пошуку. Вони підтримують персистентне зберігання, контроль версій документів, streaming індексацію, а деякі реалізації дозволяють безпосередньо виконувати обчислення на векторних даних (наприклад, кластеризацію, агрегування).

Саме завдяки цим властивостям векторні бази є ключовою частиною не лише RAG-підходу, а й будь-яких сучасних систем семантичного пошуку. У наступному підпункті ми розглянемо використання реляційних баз, зокрема PostgreSQL, як доповнення до векторного пошуку.

2.3.2 PostgreSQL з підтримкою векторного пошуку

Попри стрімкий розвиток спеціалізованих векторних баз даних, PostgreSQL продовжує відігравати важливу роль у RAG-системах, особливо в контексті зберігання метаінформації, забезпечення транзакційності та аналітики. Більше того, завдяки розширенню pgvector, PostgreSQL здатен виступати й як повноцінне векторне сховище, що дозволяє реалізувати RAG-підхід навіть без використання окремої векторної БД.

Розширення pgvector додає до PostgreSQL тип даних vector, який дозволяє зберігати багатовимірні вектори та виконувати над ними операції подібності, зокрема косинусну відстань, L2-норму та інші. Це означає, що пошук найближчих сусідів за семантичним значенням можна реалізовувати безпосередньо в рамках SQL-запитів, наприклад:

```
SELECT id, text, embedding
FROM documents
ORDER BY embedding <=> '[0.1, 0.2, 0.3, ...]'
LIMIT 5;
```

Використання PostgreSQL для векторного пошуку має ряд переваг:

- Єдина точка зберігання. Усі дані — і структуровані (наприклад, дата, автор, категорія), і неструктуровані (векторні представлення текстів) — знаходяться в одній системі, що полегшує керування та резервне копіювання.

- Транзакційність і цілісність. PostgreSQL надає повноцінні механізми транзакцій, які гарантують консистентність навіть при паралельному оновленні векторів і метаданих.

- Можливість створення складних запитів. Завдяки підтримці JOIN, CTE, WINDOW-функцій, можна легко будувати запити, що поєднують фільтрацію, сортування за схожістю та агрегацію даних.

- Індексація. pgvector підтримує індексацію через IVFFlat (інверсно-файловий індекс), що суттєво прискорює пошук у великих наборах векторів.

Проте, варто враховувати і обмеження. PostgreSQL із pgvector менш ефективний за спеціалізовані векторні бази при роботі з мільйонами векторів — перевага тут за Qdrant чи Pinecone. Але для невеликих проєктів, внутрішніх систем або при необхідності зберігати всі дані у межах однієї СУБД, pgvector є чудовим компромісом між продуктивністю та функціональністю.

На практиці часто реалізується гібридна архітектура, де PostgreSQL відповідає за зберігання та пошук метаданих і текстів, а векторні БД — за швидкий семантичний пошук. Проте pgvector дозволяє реалізувати RAG навіть без зовнішніх сервісів — у межах одного процесу Node.js або Python-застосунку, що є критично важливим для прототипів, локальних середовищ або обмежених бюджетів.

2.3.3 Порівняння Pinecone, Qdrant та інших векторних сховищ

Вибір векторної бази даних є одним з ключових рішень при розробці системи на основі RAG, адже від цього залежить швидкість, масштабованість і якість результатів пошуку. Сьогодні на ринку представлено кілька популярних рішень: Pinecone, Qdrant, Weaviate, Milvus, Chroma. Кожне з них має свої особливості, переваги та сценарії використання.

Pinecone — це комерційна хмарна платформа, яка вирізняється високою продуктивністю та простотою інтеграції. Pinecone не вимагає власного хостингу, повністю управляється через API, і підтримує автоматичне

масштабування. Її перевага — у стабільності та готовності до роботи у великих продакшн-середовищах. Основними перевагами є:

- підтримка різних метрик схожості (cosine, dot-product, Euclidean);
- фільтрація результатів на основі метаданих;
- відновлення та резервне копіювання індексів;
- низька латентність при мільйонах об'єктів.

Qdrant — це опенсорсна векторна база, написана на Rust. Вона забезпечує високу продуктивність, і водночас — прозору архітектуру з можливістю локального або клаудного розгортання. Qdrant підтримує:

- гнучку фільтрацію за будь-якими полями метаданих;
- REST і gRPC API;
- реалії "обмеженого заліза", наприклад, роботу на edge-пристроях;
- інтеграції з LangChain, Haystack, LlamaIndex.

Qdrant чудово підходить для RAG у проєктах з відкритим кодом або тоді, коли необхідно мати контроль над інфраструктурою (наприклад, при розгортанні в межах організації без зовнішнього доступу до хмари).

Weaviate — ще одна потужна опенсорсна система, яка, на відміну від інших, вбудовує семантичний пошук "із коробки" завдяки вбудованим векторизаторам (наприклад, OpenAI, Cohere, HuggingFace). Вона також підтримує hybrid search (поєднання keyword+vector), має GraphQL API, підтримує класифікацію, фільтрацію, та навіть можливість додавати власні модулі.

Milvus — орієнтований на використання у великих масштабах. Його архітектура побудована на мікросервісах і підтримує автоматичне

шардінгування даних. Добре підходить для складних аналітичних систем або хмарних застосунків з великою кількістю користувачів.

Нижче наведено порівняльну таблицю ключових характеристик (табл 2.1)

Сховище	Хостинг	API	Підтримка фільтрів	Векторизатори	Інтеграції з RAG
Pinecone	Хмара	REST	Так	Ні	LangChain
Qdrant	Cloud / Local	REST/gRPC	Так (гнучкі)	Ні	LangChain, Haystack
Weaviate	Cloud / Local	GraphQL	Так	Вбудовані	LangChain
Milvus	Local	REST/gRPC	Частково	Ні	Haystack
PostgreSQL + pgvector	Local	SQL	Так	Ні	Можливе через ORM

(табл. 2.1.)

Отже, якщо потрібна хмарна безтурботність і масштабованість — обирають Pinecone. Якщо потрібна контрольована локальна інфраструктура з відкритим кодом — найкращими кандидатами будуть Qdrant чи Weaviate. PostgreSQL із pgvector залишається актуальним у випадках, коли важлива уніфікація з уже існуючою реляційною БД.

2.4 Бази даних як джерело знань для RAG

2.4.1 LangChain: модульність та інтеграція з LLM

LangChain — це один із найпопулярніших фреймворків для побудови систем, що використовують великі мовні моделі, включаючи підхід Retrieval-Augmented Generation. Його головна ідея полягає в організації роботи LLM як послідовності логічних компонентів, зокрема генераторів, retriever-ів, інструментів пам’яті, шаблонізаторів запитів та механізмів для взаємодії з зовнішніми сервісами. Така модульна структура дозволяє будувати RAG-

системи будь-якої складності, від простих чат-ботів до інтегрованих корпоративних рішень.

Однією з ключових особливостей LangChain є підтримка retriever chain — послідовності кроків, яка охоплює вибір джерел знань, їхню індексацію, вибір релевантних фрагментів і побудову запиту до генератора. Також можливе кешування попередніх відповідей, побудова логіки запит-відповідь з пам’яттю діалогу, підключення інструментів, таких як веб-агенти або бази даних, і фільтрація результатів за метаінформацією.

LangChain має прямі інтеграції з:

- OpenAI (GPT-3.5, GPT-4);
- Cohere;
- Anthropic Claude;
- HuggingFace API;
- Qdrant, Weaviate, Pinecone, Chroma;
- Google Search API, WolframAlpha, SQL, Zapier.

Окремо варто відзначити, що LangChain дозволяє використовувати callback handlers, які спрощують відлагодження, трасування і логування процесу генерації. Це критично важливо при продакшн-впровадженні, коли потрібно контролювати кожен крок у відповіді моделі.

Таким чином, LangChain — це гнучкий фреймворк, який добре підходить як для прототипування, так і для розробки масштабованих RAG-систем. У контексті дипломної роботи LangChain дозволить швидко реалізувати мікросервісний модуль для генерації відповідей на основі знань, з використанням доступних API або локальних моделей.

2.4.2 Haystack: продакшн-орієнтована платформа для QA-систем

Haystack — це фреймворк з відкритим кодом, створений для побудови систем пошуку відповідей, чат-ботів та документорієнтованих LLM-асистентів. Його особливість полягає в орієнтації на промислове впровадження (production-grade): він підтримує масштабовані пайплайни, REST API, асинхронну обробку запитів і моніторинг у реальному часі. Хоча Haystack написаний мовою Python, його зручно інтегрувати в Node.js-системи через API-запити до окремого мікросервісу.

У контексті реалізації RAG, Haystack надає готові компоненти:

- Retriever — для класичного keyword-пошуку (BM25), dense retrieval (DPR, Elasticsearch, Milvus), або векторного пошуку (Qdrant, Weaviate);
- Reader / Generator — для обробки знайдених фрагментів або генерації відповідей на їхній основі (наприклад, з використанням GPT або BART);
- Pipelines — гнучкий механізм побудови послідовностей обробки запиту, включно з препроцесингом, постпроцесингом, ранжуванням результатів;
- Document Store — модуль для зберігання джерел знань з підтримкою SQL, NoSQL та векторних сховищ;
- REST API — для доступу до RAG-пайплайнів із будь-якого frontend або backend-додатку.

Крім цього, Haystack має інтеграцію з:

- HuggingFace Transformers (локальні або API-моделі);
- OpenAI GPT;
- Elasticsearch/OpenSearch;
- Azure Cognitive Search;

- Postgres + pgvector.

Розгортання Haystack-системи зазвичай відбувається як окремого мікросервісу, який може обробляти RAG-запити, вести логування, обмежувати запити за авторизацією, кешувати відповіді тощо. Це ідеальний варіант для великих систем, де LLM-модуль має працювати незалежно, під навантаженням і з високою доступністю.

Для дипломного проєкту Haystack буде актуальним у ролі бекенд-модуля, якщо обробка RAG-запитів виноситься в окремий контейнер або сервер. Його можна інтегрувати з Node.js-додатком через HTTP-запити до API. Водночас, якщо потрібна тісніша інтеграція, варто розглядати LangChain або LlamaIndex як більш "легкі" у впровадженні засоби.

2.4.3 LlamaIndex: побудова індексів та швидкий доступ до знань

LlamaIndex (раніше GPT Index) — це фреймворк, орієнтований на інтеграцію зовнішніх джерел даних із великими мовними моделями. Його основне завдання — забезпечити ефективне створення та використання індексів, які представляють структуровані або неструктуровані джерела у вигляді, зручному для генерації відповідей мовною моделлю.

На відміну від LangChain, який фокусується на побудові ланцюгів і модульній композиції, LlamaIndex зосереджений саме на індексації, зберіганні, та швидкому доступі до інформації. Це робить його особливо зручним для реалізації retrieval-компоненти RAG-підходу.

Ключові можливості LlamaIndex:

- Підтримка різних типів джерел: PDF-документи, Markdown, SQL-бази даних, API, Google Sheets, Notion, файли JSON/CSV, HTML, веб-сторінки.

- Гнучкі типи індексів: векторні (Vector Index), ієрархічні (Tree Index), спискові (List Index), комбіновані індекси, які дозволяють оптимізувати пошук залежно від задачі.
- Автоматичне розбиття документів на chunk-и (контекстні фрагменти), побудова embedding-ів, збереження метаінформації.
- Кешування, постобробка, логування запитів — усе це підтримується з коробки.

Архітектура LlamaIndex:

- Data Connectors: адаптери до джерел інформації.
- Indexers: створюють індекси на основі raw-даних.
- Query Engines: API для виконання запитів на основі індексів.
- Storage: зберігання індексів у локальних файлах або базах даних.

LlamaIndex добре працює із векторними сховищами, як-от Qdrant, Pinecone, Weaviate, Chroma. Також має інтеграції з OpenAI API, HuggingFace Transformers та локальними моделями через Ollama.

2.4.4 Інтеграція з популярними LLM (OpenAI, Cohere, HuggingFace Transformers)

Ефективна реалізація RAG-системи значною мірою залежить від якості мовної моделі, що виконує генерацію відповіді. У сучасних розробках використовується широкий спектр генеративних моделей, доступних як через API, так і у вигляді open-source рішень. Найбільш поширені серед них — це GPT-моделі від OpenAI, моделі від Cohere (Command-R, Command-Light) та численні варіанти на платформі HuggingFace (Falcon, LLaMA, Mistral, Mixtral тощо). У цьому підрозділі розглянемо їх ключові характеристики та можливості інтеграції у RAG-системи.

OpenAI API (GPT-3.5, GPT-4) — один із найбільш точних та популярних генераторів для RAG. API дозволяє легко виконувати генерацію тексту, працює швидко, стабільно та підтримує функціонал функціональних викликів (function calling), що особливо корисно у поєднанні з retrieval-компонентами. Вбудована підтримка prompt-инжектування, системних повідомлень та message history робить ці моделі універсальними для чат-асистентів, Q&A та документних аналізаторів. Інтеграція з Node.js здійснюється через офіційний SDK або REST API.

Cohere Command-R — альтернатива OpenAI з оптимізацією для завдань пошуку, генерації та класифікації. Модель має спеціалізацію на retrieval-завданнях і показує хорошу продуктивність у парі з векторними пошукачами. Платформа пропонує вбудовану підтримку RAG-ланцюгів, зокрема для застосування у багатомовному середовищі.

HuggingFace Transformers — це репозиторій із сотнями відкритих моделей, які можуть бути розгорнуті як локально, так і через HuggingFace API. Це дозволяє розробникам RAG-систем мати повний контроль над інфраструктурою, витратами та приватністю. Наприклад, моделі типу Falcon, Mistral чи LLaMA2 часто використовуються для задач, де потрібен баланс між якістю генерації та обчислювальними ресурсами. Інтеграція можлива через REST, HTTP або запуск через Node.js-процеси.

Ollama, LM Studio та WebLLM — допоміжні інструменти для локального запуску LLM, сумісні з RAG-архітектурою. Вони дозволяють запускати open-source LLM на локальному сервері або навіть у браузері, що корисно для офлайн-сценаріїв, забезпечення приватності або оптимізації витрат. Наприклад, LLaMA2 можна запускати через Ollama CLI й підключати до Node.js-мікросервісу через простий REST-інтерфейс.

Загальна стратегія інтеграції мовної моделі в RAG-систему виглядає наступним чином:

- формування промпту на основі результатів пошуку (retrieved documents);
- виклик моделі через API або локальний endpoint;
- обробка відповіді (постобробка, форматування, логування);
- за потреби — повторна генерація з уточненим запитом або re-ranking результатів.

Таким чином, вибір LLM залежить від бюджету, вимог до конфіденційності, необхідної точності та масштабованості системи. Для дипломної роботи найбільш зручно працювати з OpenAI або Cohere через API, або обрати open-source модель через HuggingFace і запускати її локально для гнучкості та контролю.

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі було здійснено комплексний огляд підходу Retrieval-Augmented Generation (RAG), його архітектури, супровідних моделей та інструментів, які використовуються для реалізації інтелектуальних систем з доступом до зовнішніх джерел знань. RAG представляє собою новий етап розвитку генеративних систем: поєднання великих мовних моделей (LLM) із динамічним доступом до актуальної інформації дозволяє зменшити кількість «галюцинацій», забезпечити більшу точність і релевантність відповідей, а також підвищити адаптивність систем до конкретних доменів.

У межах розділу було розглянуто ключові компоненти архітектури RAG — модуль пошуку (retriever), модуль генерації (generator) та механізми оркестрації. Детально було проаналізовано способи інтеграції генеративних трансформерів із пошуковими компонентами, зокрема стратегію «retrieve-then-generate», яка лягла в основу багатьох сучасних інтелектуальних систем (чат-ботів, віртуальних асистентів, автоматичних аналітиків).

Окрему увагу приділено базам даних як джерелам знань. У цьому контексті векторні сховища, такі як Pinecone, Qdrant, Weaviate, виявилися ключовим елементом, що дозволяє здійснювати ефективний семантичний пошук. Водночас реляційні бази даних (наприклад, PostgreSQL) забезпечують зберігання метаінформації та аналітику. Комбінований підхід із синхронізацією між векторною та реляційною частинами дозволяє досягти консистентності, масштабованості та розширюваності в роботі з даними.

У розділі також розглянуто популярні фреймворки, які спрощують створення RAG-систем: LangChain (як головний інструмент на Node.js), Haystack (потужна продакшн-платформа на Python), LlamaIndex (ефективний засіб побудови індексів для LLM), а також доступні мовні моделі: GPT-3.5/4, Cohere Command-R, Falcon, Mistral, LLaMA. Продемонстровано, що інтеграція

цих інструментів можлива як у хмарних, так і в локальних середовищах, із використанням як API, так і локальних моделей через Ollama або HuggingFace.

Таким чином, другий розділ підтвердив технологічну зрілість RAG-підходу та надав теоретичну і практичну базу для його реалізації у рамках мікросервісної системи підтримки. Отримані знання і обрані інструменти будуть використані у третьому розділі, який присвячено практичному впровадженню RAG у дипломному проєкті.

РОЗДІЛ 3. ДЕТАЛІ РОЗРОБКИ СИСТЕМИ

Відповідно до досліджених матеріалів та обраного способу реалізації ми можемо перейти до фази розробки програмного продукту. Для того, щоб створити гнучку та настроювану систему яка допоможе службі підтримки з пошуком відповідей на запитання.

3.1 Розробка мікросервісної архітектури

У цьому розділі детально розглянуто процес створення мікросервісної системи, реалізованої за допомогою NestJS. Архітектура побудована на основі монорепозиторію з кількома сервісами, які відповідають за окремі функціональні частини системи. Основна комунікація між сервісами відбувається через HTTP-запити та брокер повідомлень Kafka, що забезпечує асинхронну та надійну взаємодію. Як база даних використовується MongoDB, яка дозволяє гнучко працювати зі збереженням та доступом до неструктурованих даних, таких як запити користувача, метадані та знання.

3.1.1 Структура монорепозиторію NestJS

Для реалізації мікросервісної архітектури було обрано підхід монорепозиторію, який дозволяє об'єднати декілька взаємозалежних сервісів в єдину структуру проєкту. Це забезпечує централізоване керування залежностями, спільне використання конфігурацій, зручне локальне тестування та уніфіковане розгортання.

У репозиторії support-rag-system реалізовано три основні мікросервіси:

1) api-gateway
2) todos
3) users

Пояснення структури:

apps/ містить повноцінні сервіси, кожен з яких має власні контролери, модулі, сервіси, DTO та конфігурацію запуску;

docker-compose.yml забезпечує підняття інфраструктури — бази даних (MongoDB), брокера Kafka, pgAdmin або інших інструментів через контейнери;

Запуск сервісів

Для зручності розробки та запуску всі сервіси мають відповідні start-команди у файлі package.json:

```
json
"scripts": {
  "start:api-gateway": "nest start api-gateway",
  "start:todos": "nest start todos",
  "start:users": "nest start users",
  "start:dev": "concurrently \"npm run start:api-gateway\" \"npm run start:todos\" \"npm run start:users\""
}
```

Таким чином, за допомогою npm run start:dev можна одночасно запускати всі сервіси для локального тестування. Кожен з них піднімається на своєму порту та комунікує через HTTP або Kafka відповідно до налаштувань.

Цей підхід до структури дозволяє розвивати систему гнучко, ізольовано працювати над кожним сервісом і легко масштабувати проєкт у майбутньому.

3.1.2 Сервіс api-gateway: API-шлюз для маршрутизації запитів

У системі було реалізовано окремий мікросервіс api-gateway, який відповідає за маршрутизацію зовнішніх запитів до відповідних внутрішніх мікросервісів, зокрема users та todos. Його побудовано на базі NestJS, і він є точкою входу до

всієї мікросервісної архітектури. Основна роль API Gateway — це централізований контроль над доступом, агрегація відповідей та делегування запитів до відповідних обробників.

Структура сервісу включає:

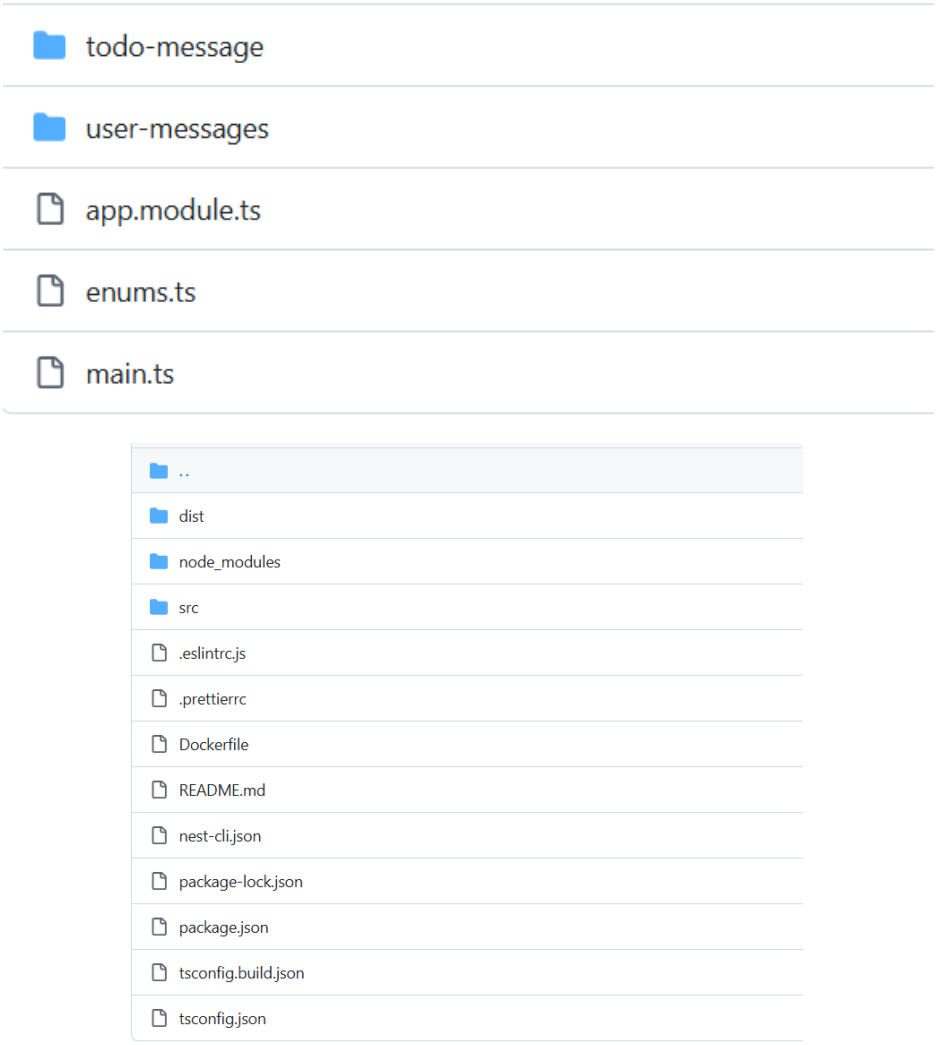


Рисунок 3.1

src/todo-message/ — директорія, що містить модулі для роботи з задачами (todo). Зокрема, у ній є todo.controller.ts, який обробляє HTTP-запити, та todo.service.ts, який реалізує взаємодію з Kafka або іншим транспортом.

src/user-messages/ — директорія, відповідальна за обробку запитів користувачів. Аналогічно містить user.controller.ts, user.service.ts і DTO для валідації даних.

app.module.ts — кореневий модуль, де імпортуються всі підмодулі системи.

main.ts — файл запуску, який ініціалізує додаток та активує CORS для фронтенд-доступу.

Головною особливістю api-gateway є його здатність не тільки перенаправляти запити, але й виконувати асинхронну розсилку повідомлень через Kafka. Коли клієнт надсилає, наприклад, запит на створення задачі, цей запит серіалізується у Kafka-подію та публікується в один із топіків, які слухають відповідні сервіси. Сервіси-споживачі (наприклад, todos) обробляють повідомлення, формують відповідь і надсилають її назад до api-gateway, або напряму клієнту — залежно від конфігурації.

Таким чином, api-gateway у даній системі виконує критично важливу роль координації мікросервісів. Він забезпечує:

- безпечний доступ до системи (через JWT та гварди),

- централізовану обробку користувацьких запитів,

- генерацію Kafka-подій та маршрутизацію на основі контексту,

- агрегацію результатів, які надходять від інших мікросервісів.

Використання Kafka дозволяє масштабувати систему без втрати швидкодії або надійності. Завдяки подієвій моделі обміну даними, сервіси можуть бути повністю ізольованими, а обробка задач — розподіленою. Це особливо актуально для високонавантажених середовищ, таких як системи підтримки або аналітики в реальному часі.

3.1.3 Клас BaseDoubleSpecies

Сервіс todos є ключовим компонентом системи, оскільки саме через нього проходить основна логіка взаємодії з користувацькими запитами, що в подальшому будуть використані для генерації відповідей через Retrieval-Augmented Generation (RAG). У контексті даного проєкту сервіс todos виконує функції збереження запитів, їх базової обробки, а також інтеграції з зовнішнім RAG-сервісом, який повертає згенеровану відповідь.

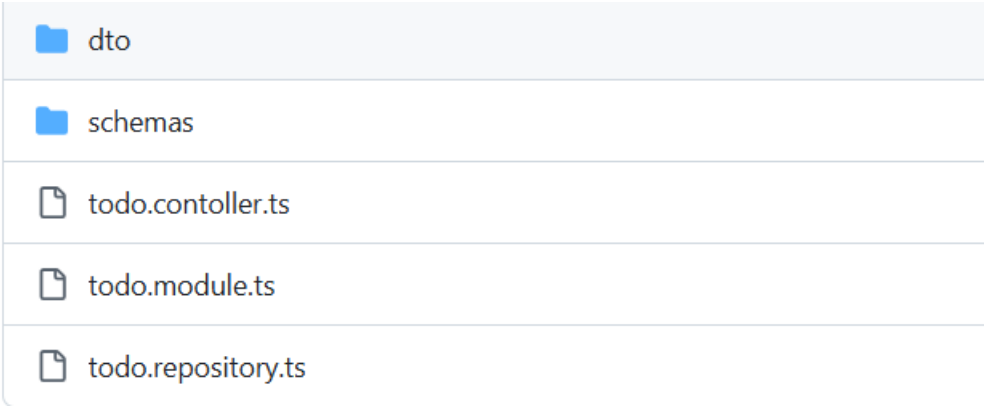


Рисунок 3.2

В архітектурі NestJS цей сервіс також побудований за класичною структурою — модуль, контролер, сервіс, DTO, гварди. Основна логіка зосереджена в таких файлах:

- todo.controller.ts — відповідає за прийом запитів від api-gateway. Наприклад, клієнт може відправити POST-запит із текстом питання, яке буде передано на обробку;

- todo.service.ts — реалізує логіку отримання запитів від користувачів, різних питань відповідей тощо.

- dto — зберігає типи для запитів/відповідей, що гарантує типобезпечність у всьому додатку.

Коли todos отримує запит від користувача, він може виконати попереднє збереження запиту в MongoDB для подальшого аналізу або аудиту. Після цього сервіс формує структуру запиту у форматі, який вимагається RAG-системою, і надсилає HTTP-запит на відповідний ендпоінт Python-сервісу, реалізованого через FastAPI.

Python-сервіс на стороні обробляє запит, звертається до векторної бази знань (наприклад, Pinecone), виконує пошук релевантних фрагментів та передає їх у модель генерації (наприклад, OpenAI через LangChain). Згенерована відповідь повертається у NestJS-сервіс todos, де вона або відразу надсилається клієнту, або зберігається в базі даних для подальшого перегляду.

Також у сервісі todos можуть бути реалізовані додаткові можливості:

1) кешування відповідей для уникнення дублюючих запитів;

2) логування часу обробки запитів та відповідей;

3) зберігання згенерованих відповідей разом із вхідним питанням для подальшого машинного аналізу.

У майбутньому даний сервіс можна масштабувати як горизонтально (через Kafka топіки), так і функціонально — наприклад, додати інтерфейси для модерації запитів, аналітики або інтеграції з іншими RAG-движками.

3.1.4 Сервіс users: взаємодія з юзерами

Сервіс users у структурі мікросервісної системи виступає як базовий інфраструктурний модуль, відповідальний за реєстрацію, автентифікацію та управління даними користувачів. Він забезпечує авторизацію доступу до всіх інших сервісів системи та виконує роль центрального провайдера ідентичностей, що особливо важливо в умовах розподіленої архітектури.

В архітектурі NestJS сервіс реалізовано у вигляді класичного модуля з такими ключовими компонентами:

user.module.ts — Підключається до нашої системи для отримання запитів з Kafka

user.service.ts — містить бізнес-логіку: створення нового користувача, перевірка токена, оновлення даних, перевірка унікальності email або username;

/schemas — містить в собі схеми які будуть використовуватись для юзера

Аутентифікація користувачів реалізована через JWT (JSON Web Tokens). Після успішної авторизації клієнт отримує токен доступу, який

використовується для взаємодії з іншими сервісами. Наприклад, сервіс todos через jwt-auth.guard.ts перевіряє валідність токена перед обробкою запиту. Це дозволяє чітко відокремити права доступу й забезпечити безпеку системи без централізованої автентифікації.

Дані користувачів зберігаються в MongoDB, причому для захисту паролів застосовується bcrypt-хешування. У разі потреби можлива реалізація додаткових рівнів безпеки: двофакторна автентифікація, рольова модель доступу (RBAC), ліміти на запити.

Взаємодія з іншими мікросервісами можлива через Kafka або HTTP-запити — наприклад, після створення нового користувача, подія user.created може бути надіслана до інших сервісів (аналітики, логування, RAG) для побудови персоналізованого досвіду.

Таким чином, сервіс users є фундаментом для контролю доступу та забезпечення контексту користувача в системі, а його ізоляція у вигляді окремого мікросервісу відповідає принципам масштабованої, гнучкої архітектури.

3.2 Робота з брокером повідомлень Kafka

Apache Kafka у даній системі виступає як основний брокер повідомлень для обміну даними між мікросервісами. Завдяки подієво-орієнтованій моделі, Kafka дозволяє ефективно масштабувати обмін інформацією без жорстких залежностей між сервісами, що суттєво підвищує стійкість і надійність всієї архітектури.

В контексті реалізації RAG-системи Kafka використовується для:

- трансляції запитів користувачів, що надходять через api-gateway, до відповідного мікросервісу;
- передачі подій створення, оновлення чи видалення записів у базі знань;
- надсилання результатів генерації відповіді назад у фронтову частину через відповідні топіки;
- асинхронного зберігання активності користувача або результатів інференсу.

Усі мікросервіси підключені до Kafka як продюсери й консюмери, використовуючи бібліотеку @nestjs/microservices з підтримкою Kafka-транспорту. У main.ts кожного сервісу конфігурується Kafka-модуль із зазначенням топіків, до яких він підписується. Наприклад, api-gateway публікує події до топіка todos.ask, а сервіс todos їх приймає, обробляє та відповідає через todos.response.

3.2.1 Конфігурація Kafka та підключення до NestJS

Підключення Kafka у NestJS здійснюється через транспортний механізм мікросервісів. У кожному сервісі створюється окремий Kafka-сервер з індивідуальним clientId та відповідними consumerGroupId, щоб уникнути конфліктів.

```
MicroserviceOptions: {
  transport: Transport.KAFKA,
  options: {
    client: {
      clientId: 'todos',
      brokers: ['localhost:9092'],
    },
    consumer: {
      groupId: 'todos-consumer',
    },
  },
}
```

Ця конфігурація дозволяє кожному мікросервісу бути незалежним, але в той же час реагувати на події системи. Також можна створювати відразу кілька Kafka-лістенерів на різні топіки в межах одного сервісу.

3.2.2 Обробка подій та взаємодія між мікросервісами

У рамках реалізованої системи Kafka виконує роль сполучної ланки між api-gateway, todos та іншими сервісами. Коли користувач надсилає запит, api-gateway створює подію з payload-запитом, яка миттєво публікується у відповідний Kafka-топік. Сервіс todos підписаний на цей топік і, отримавши подію, одразу починає її обробку: або надсилає запит до RAG-сервісу, або здійснює пошук у базі знань.

Після завершення обробки результат знову надсилається назад у Kafka в топік відповіді (todos.response), на який підписаний api-gateway, щоб віддати готову відповідь клієнту.

Таке розділення обов’язків дозволяє:

- обробляти велику кількість одночасних запитів без навантаження на API;

- масштабувати мікросервіси горизонтально (додавати нові інстанси без зміни логіки);

- забезпечувати гарантовану доставку подій та надійний порядок їх обробки.

У разі потреби до архітектури можна додати додаткові проміжні обробники — наприклад, сервіс логування запитів, сервіс аналітики або сервіс кешування, які будуть просто слухати відповідні Kafka-події.

3.3 Розробка програмних компонентів DatabaseConnection

У межах мікросервісної архітектури проєкту support-rag-system, система керування базами даних MongoDB була обрана як основна технологія для зберігання структурованих і напівструктурованих даних. Це рішення зумовлене гнучкістю MongoDB, її хорошою масштабованістю, природною інтеграцією з JavaScript/TypeScript екосистемою та зручністю для зберігання об'єктів, що динамічно змінюються.

MongoDB ідеально підходить для зберігання об’єктів користувача, історій взаємодій, журналів запитів до системи підтримки, а також — для тимчасового кешування проміжних результатів генерації з RAG-системи. На відміну від реляційних баз, MongoDB не потребує жорсткого визначення схем заздалегідь, тому кожен мікросервіс має змогу зберігати дані у форматі,

зручному саме для його задач, не порушуючи загальну консистентність системи.

Усі підключення до MongoDB здійснюються за допомогою бібліотеки Mongoose, яка є обгорткою для роботи з базою даних і дозволяє типізувати моделі, задавати правила валідації, реалізовувати індексацію та зв’язки. Кожен мікросервіс, що потребує роботи з БД, має свою модель і відповідні сервіси, які інкапсулюють логіку взаємодії з MongoDB. Це дозволяє чітко розмежувати відповідальність між бізнес-логікою та рівнем зберігання даних.

Використання MongoDB у поєднанні з Kafka також відкриває можливості для асинхронної реплікації подій: наприклад, коли новий запит надходить до системи, він може бути збережений у базу даних із затримкою, без блокування основного процесу обробки запиту, що підвищує загальну продуктивність системи.

3.3.1 Підключення до MongoDB через Mongoose

Для взаємодії з MongoDB у мікросервісах support-rag-system було використано бібліотеку Mongoose, яка є де-факто стандартом для роботи з MongoDB у середовищі Node.js та NestJS. Вона надає зручний механізм для опису схем даних, створення моделей, а також реалізації перевірок, хукав та індексації.

У кожному мікросервісі, який зберігає або читає дані з бази, використовується окремий модуль MongooseModule, що підключається в AppModule з відповідними параметрами URI:

MongooseModule.forRoot(process.env.MONGODB_URI)

Цей підхід дозволяє централізовано зберігати конфігурацію підключення в .env файлі або у змінних середовища. Перевагою використання Mongoose у зв'язці з NestJS є наявність декораторів та підтримки dependency injection, що дозволяє створювати модульну, легко тестовану архітектуру.

Наприклад, для сервісу todos було створено модель Todo, яка описує структуру об’єкта запиту на відповідь (наприклад, текст запиту, дата, статус обробки):

```
export type TodoDocument = Todo & Document;

@Schema()
export class Todo {
  @Prop()
  todoTitle: string;

  @Prop()
  todoDescription: string;

  @Prop()
  todoStatus: boolean;

  @Prop({ type: mongoose.Schema.Types.ObjectId, ref: 'User' })
  owner: User;
}
```

Рисунок 3.3

Використовуючи такі схеми, розробники можуть забезпечити перевірку валідності полів ще до збереження об'єкта в базі, що допомагає уникнути помилок та спрощує налагодження системи.

Завдяки автоматичному створенню колекцій MongoDB, інтеграція Mongoose не потребує ручного управління схемами або міграціями. Це робить

MongoDB особливо зручним вибором для гнучких систем, де формат даних може змінюватися залежно від функціональних потреб.

3.3.2 Створення схем документів і зберігання знань

Збереження знань у системі реалізовано за допомогою комбінованого підходу: метадані документів зберігаються у реляційній базі даних PostgreSQL, а векторизовані представлення — у зовнішньому сервісі Pinecone. Це дозволяє ефективно обробляти як семантичний пошук, так і традиційні фільтрації за користувачем, датою чи статусом.

У реляційній базі створено таблиці users, chats та messages, які репрезентують структуру діалогів користувача із системою. Основою є модель Chat, яка зберігає унікальний pinecone_namespace — ідентифікатор, що використовується для просторової ізоляції векторних представлень у Pinecone. Модель Message зберігає конкретні репліки чатів разом із роллю (наприклад, user чи assistant), текстовим контентом та часовими мітками. Всі ці таблиці реалізовані через SQLAlchemy, що забезпечує ORM-рівень для зручної взаємодії з PostgreSQL.

Такий підхід дозволяє:

- вести історію запитів користувача;
- зв’язувати чати з відповідними векторними представленнями;
- виконувати повну очистку простору знань для конкретного чату через API Pinecone;

- і, при потребі, повторно ініціалізувати знання для обраного чату, не зачіпаючи інші.

Ця гібридна модель дає змогу гнучко керувати знаннями та масштабувати систему під реальні бізнес-випадки, зберігаючи при цьому контекстні зв’язки між документами, користувачами та запитами.

3.3.3 Валідація, фільтрація та обробка запитів до бази

У мікросервісній архітектурі системи support-rag-system важливим етапом обробки інформації є коректна взаємодія з реляційною базою знань. Кожен запис у базі є джерелом правдивих даних для RAG-моделі, тому особливу увагу приділено процесам валідації, фільтрації та обробки запитів.

Валідація даних при створенні

Перш ніж дані будуть записані до таблиці knowledge_blocks, вони проходять етап перевірки:

- наявність обов’язкових полів (title, content);
- перевірка формату тега (відсутність спецсимволів, дублювання);
- допустимість джерела (перевірка на whitelist).

Цей процес реалізовано на рівні мікросервісу, який відповідає за обробку запитів до знань (todos), та забезпечує мінімізацію ризику внесення неконсистентних або некоректних даних.

Фільтрація при запиті

Система підтримує гнучкі механізми фільтрації, зокрема:

- фільтрація по тегах (WHERE $1 = ANY(tags)),
- фільтрація по часу (WHERE created_at >= $1 AND created_at <= $2),
- пошук за джерелом або заголовком (ILIKE '%...%').

Це дозволяє будувати динамічні інтерфейси пошуку, інтегровані з UI або API, та значно спрощує підготовку даних до векторизації.

Підготовка до векторизації

Для подальшої інтеграції з RAG-компонентом, знання проходять додаткову попередню обробку:

1. очищення від зайвих пробілів, HTML-тегів та спеціальних символів;
2. розбиття на логічні блоки тексту (через LangChain TextSplitter на Python-сервісі);
3. додавання технічних метаданих (наприклад, doc_id, block_order) для подальшого зв’язування векторів з джерелами.

Після завершення обробки, мікросервіс публікує подію у Kafka (топік knowledge.new), яку споживає сервіс RAG. Цей підхід забезпечує відкладену обробку без блокування основного потоку користувача.

Захист запитів

- Усі запити до бази знань обгорнуті у репозиторії з використанням ORM-абстракції (наприклад, Prisma або Knex), що забезпечує:

- безпечне формування SQL-запитів (захист від SQL-ін’єкцій),

- логування транзакцій,

- централізовану обробку помилок з поверненням стандартних статусів (400, 404, 500).

Таким чином, вся робота з базою знань є не лише ефективною, але й безпечною та масштабованою, що є критичним у контексті автоматизованих систем підтримки знань.

3.4 Реалізація RAG-моделі на Python

Retrieval-Augmented Generation (RAG) у даній системі реалізовано як окремий сервіс на мові Python із використанням FastAPI для побудови REST-інтерфейсу, LangChain — для побудови ланцюгів обробки запитів, а також OpenAI й Pinecone — для векторизації та зберігання знань відповідно. Цей сервіс є повноцінним мікросервісом, що працює незалежно від NestJS-системи, але взаємодіє з нею через HTTP-запити.

3.4.1 Використання FastAPI для побудови API

У розробці сервісу, що реалізує Retrieval-Augmented Generation (RAG), важливим етапом стало створення зручного та продуктивного API, який дозволяв би взаємодіяти з іншими компонентами системи, зокрема NestJS-мікросервісами. Для цього було обрано сучасний Python-фреймворк FastAPI, що забезпечує високу швидкодію, простоту реалізації REST-інтерфейсів та нативну підтримку асинхронного виконання запитів.

Однією з ключових переваг FastAPI є автоматичне формування документації OpenAPI (Swagger UI), що значно полегшує процес тестування, налагодження та інтеграції з іншими сервісами. Завдяки цьому розробники з команди, які працюють з NestJS або frontend-клієнтами, можуть легко орієнтуватися в доступних маршрутах, параметрах та типах відповіді.

На практиці FastAPI виступає не просто як HTTP-сервер, а як центральна точка взаємодії з RAG-ланцюгом. Було створено кілька ендпоінтів, кожен із яких виконує окрему логічну функцію:

1. POST /rag/index — ендпоінт, який приймає raw-документ у текстовому вигляді, розбиває його на фрагменти, виконує векторизацію та зберігає в Pinecone;

2. POST /rag/query — маршрут, що дозволяє виконати запит користувача, передати його у RetrievalQA-ланцюг і повернути сформовану відповідь;

3. DELETE /rag/namespace — службовий маршрут, що дає можливість повністю очистити вказаний namespace у Pinecone;

4. GET /health — простий технічний маршрут, який повертає статус готовності сервісу до роботи, що особливо корисно для моніторингу та балансування навантаження.

Кожен маршрут має чітко визначені типи вхідних і вихідних даних, що визначаються за допомогою бібліотеки pydantic. Завдяки цьому система є типобезпечною та здатна відловлювати помилки вже на етапі парсингу запиту.

Окрім цього, FastAPI ідеально підходить для контейнеризації та запуску в продакшн-середовищі. Він має низький час підняття сервера, не потребує надмірних ресурсів і добре масштабується, що дає змогу обслуговувати численні паралельні запити, що приходять із сервісу api-gateway або з Kafka-споживачів.

Загалом, використання FastAPI у даній архітектурі дозволило:

- ізолювати RAG-логіку в окремий сервіс;
- пришвидшити реалізацію та впровадження змін;
- забезпечити високу доступність, надійність і зручність масштабування сервісу;
- створити розширювану основу для інтеграції з іншими компонентами системи (наприклад, базою знань, аналітичним модулем чи зовнішніми API).

3.4.2 Побудова пайплайну RAG з LangChain

Ключовою частиною реалізації RAG-системи є створення пайплайну, що поєднує модулі пошуку, генерації та обробки запитів користувача. Для цього було обрано бібліотеку LangChain, яка на сьогодні є одним із найпотужніших фреймворків для побудови LLM-орієнтованих застосунків. Вона забезпечує зручну абстракцію над багатьма процесами, такими як обробка документів, генерація ембеддингів, організація ланцюгів (chains) та побудова агентів.

Пайплайн у нашому випадку реалізує класичний підхід «retrieve-then-generate», тобто:

1. спочатку здійснюється пошук релевантних фрагментів документа;

2. далі — формування відповіді мовною моделлю на основі знайденого контексту.

Для побудови такого процесу за допомогою LangChain були реалізовані наступні кроки:

1. Розбиття тексту на фрагменти

Початковий документ надходить у вигляді plain text. Щоб ефективно індексувати інформацію, текст необхідно розбити на смислові частини. У проєкті застосовано RecursiveCharacterTextSplitter, який враховує логіку абзаців і структур тексту. Це дозволяє уникнути розривів у середині фраз або термінів, що критично важливо для збереження сенсу при генерації відповідей.

2. Генерація векторних представлень (embedding)

Кожен фрагмент тексту перетворюється на вектор за допомогою моделі text-embedding-3-large від OpenAI. Ця модель забезпечує високу точність семантичного порівняння текстів і дозволяє будувати ефективне векторне

сховище. Процес виконується через клас OpenAIEmbeddings, який інтегрується з LangChain без зайвих конфігурацій.

3. Збереження фрагментів у Pinecone

Отримані вектори зберігаються у зовнішньому векторному сховищі Pinecone, яке було обране за його високу швидкодію, горизонтальну масштабованість та інтеграцію з LangChain через PineconeVectorStore. Кожна колекція векторів зберігається під унікальним namespace, що дозволяє ізолювати інформацію для кожного чату або користувача.

4. Побудова ланцюга RetrievalQA

На основі готової бази знань створюється об'єкт RetrievalQA, який поєднує:

- генеративну модель (у нашому випадку ChatOpenAI);
- ретрівер (retriever) — об’єкт, що забезпечує пошук релевантного контенту у Pinecone;
- обробку запитів і повернення джерел інформації (через return_source_documents=True).

Цей ланцюг стає ядром RAG-механізму, що здатен у реальному часі шукати та відповідати на запити користувачів.

5. Виконання запитів

Коли користувач надсилає запит (наприклад, через сервіс todos у NestJS), дані передаються до Python-ендпоінта, де активується метод run_query().

LangChain автоматично виконує повний пайплайн: пошук — побудова prompt — генерація відповіді.

Таким чином, завдяки LangChain вдалося реалізувати розширюваний, модульний і легко підтримуваний пайплайн, який може масштабуватись та адаптуватись до різних типів даних і моделей. Система вже готова для інтеграції з іншими LLM або навіть розширення до agent-based архітектур, що відкриває перспективи для подальшого розвитку проєкту.

3.4.3 Векторизація знань через OpenAIEmbeddings

Векторизація тексту — це базовий етап підготовки інформації для подальшого векторного пошуку. У контексті Retrieval-Augmented Generation (RAG), правильне формування ембеддингів напряму впливає на якість релевантності результатів і точність відповідей, що генеруються мовною моделлю. У розробленій системі було прийнято рішення використати сервіс OpenAIEmbeddings, який надається через API компанії OpenAI та підтримується фреймворком LangChain.

Переваги використання OpenAIEmbeddings

Модель text-embedding-3-large, яка використовується у системі, є однією з найсучасніших і має високу семантичну чутливість. Вона здатна обчислювати векторне представлення тексту, враховуючи контекст, граматику, інтонацію та навіть приховані смисли. Саме ця властивість забезпечує якісну побудову embedding-простору, в якому схожі за змістом тексти розташовуються близько один до одного.

Серед інших переваг:

1. Велика роздільна здатність ембеддингів (більше 1500 вимірів);

2. Висока швидкодія при обробці сотень документів;

3. Надійність та стабільність інтерфейсу API;

4. Готова інтеграція з LangChain, що дозволяє зменшити час на конфігурацію.

Реалізація в системі

У кодовій базі проєкту реалізація векторизації виглядає наступним чином:

```
def get_embeddings():
    return OpenAIEmbeddings(
        model="text-embedding-3-large",
        api_key=settings.openai_api_key
    )
```

Ця функція ініціалізує об'єкт OpenAIEmbeddings, використовуючи ключ API, що зберігається у файлі конфігурації. Модель повністю конфігурована для використання в продакшн-середовищі, з можливістю масштабування відповідно до навантаження системи.

Інтеграція з пайплайном

Отримані ембеддинги передаються до PineconeVectorStore, де вони зберігаються в базі даних Pinecone. Завдяки такому підходу до векторизації:

- система може виконувати пошук за змістом, а не за точними словами;

- реалізується можливість складного семантичного порівняння фрагментів тексту;

- відкривається шлях до застосування додаткових технік, таких як кластеризація, re-ranking, hybrid search тощо.

Таким чином, OpenAIEmbeddings слугує критичним компонентом, який поєднує якість комерційних моделей з гнучкістю open-source-інструментів LangChain. Його використання дозволяє будувати точні, адаптивні і масштабовані системи RAG, що відповідають сучасним вимогам до обробки знань.

3.4.4 Збереження векторів у PineconeVectorStore

Одним із ключових завдань у побудові RAG-системи є не лише векторизація текстових даних, але й їхнє ефективне збереження та подальший пошук. Саме для цієї мети у реалізованій системі використовується хмарне сховище Pinecone, яке спеціалізується на високошвидкісному зберіганні та обробці векторів у великому масштабі. У поєднанні з LangChain це рішення стало основою модулю збереження знань.

Основні можливості Pinecone

Pinecone — це високопродуктивна векторна база даних як сервіс (Vector DB-as-a-Service), яка дозволяє:

1. Зберігати мільйони векторів з високою доступністю;

2. Працювати з власними namespace — ізольованими просторами для різних чатів або юзерів;

3. Використовувати фільтрацію за метаданими;

4. Масштабувати систему горизонтально без втрати продуктивності;

5. Працювати з API через офіційні клієнти Python та Node.js.

Для інтеграції Pinecone у систему використовувався модуль PineconeVectorStore з бібліотеки langchain_pinecone, який дозволяє взаємодіяти зі сховищем на високому рівні абстракції.

```
def init_vector_store(index_name: str, namespace: str):
    pc = Pinecone(api_key=settings.pinecone_api_key, environment=settings.pinecone_environment)
    index = pc.Index(index_name)
    embeddings = get_embeddings()
    return PineconeVectorStore(embedding=embeddings, index=index, namespace=namespace)
```

У цьому фрагменті коду:

- створюється екземпляр клієнта Pinecone;

- виконується підключення до індексу;

- надається embedding-модель для використання під час додавання даних;

- створюється простір namespace, що дозволяє розмежовувати знання.

Збереження векторів здійснюється через метод add_documents, який приймає список фрагментів (splits), згенерованих на попередньому етапі:

```
vector_store.add_documents(documents=splits, namespace=namespace)
```

У разі, якщо розмір вхідного документа перевищує допустимий ліміт (близько 4MB), система генерує кастомну помилку PineconeUploadError, яка відловлюється та обробляється.

```
if "message length too large" in error_message.lower():
    raise PineconeUploadError("File is too large to process. The maximum limit is ~4MB.")
```

Із застосуванням Pinecone досягається:

1. Висока продуктивність пошуку — відповіді на запити повертаються в межах мілісекунд;

2. Гнучке масштабування знань — можна зберігати контент на рівні користувачів, чатів або бізнес-логіки;

3. Зменшення часу генерації — швидкий доступ до найбільш релевантних контекстів забезпечує ефективну генерацію відповідей.

Таким чином, PineconeVectorStore є центральним елементом архітектури RAG-підсистеми, який поєднує збереження, масштабованість та продуктивність у одному рішенні.

3.4.5 Реалізація RetrievalQA через LangChain

Завершальним етапом реалізації RAG-підсистеми є побудова інтелектуального ланцюга для генерації відповідей на основі отриманих векторів. У нашій системі для цього використовується компонент RetrievalQA із бібліотеки LangChain, який поєднує в собі два найважливіші елементи: LLM (велику мовну модель) і retriever, що виконує пошук релевантної інформації.

RetrievalQA — це готовий ланцюг (chain), який працює за принципом:

1. Отримати запит користувача;
2. Виконати векторний пошук по векторному сховищу (Pinecone);
3. Передати знайдені фрагменти у вхідний prompt мовної моделі;
4. Сгенерувати відповідь, яка враховує знайдені знання.

Цей підхід дозволяє надати відповіді, які є не лише логічно послідовними, а й обґрунтованими з точки зору наявної бази знань.

```
def create_qa_chain(vector_store, namespace: str):
    llm = ChatOpenAI(
        model=settings.openai_model,
        temperature=0.5,
        api_key=settings.openai_api_key
    )
```

```
    retriever = vector_store.as_retriever(namespace=namespace)

    return RetrievalQA.from_chain_type(
        llm=llm,
        retriever=retriever,
        return_source_documents=True
    )
```

Тут ми бачимо ключові етапи побудови ланцюга:

- Ініціалізація LLM через ChatOpenAI (використовується GPT-4 або GPT-3.5);
- Ініціалізація retriever, який виконує запит до Pinecone по вказаному namespace;
- Створення об'єкта RetrievalQA з вказаними параметрами.

Параметр return_source_documents=True дозволяє додатково повертати документи, на основі яких була сформована відповідь. Це важливо для аудиту, debug’у та довіри до системи з боку користувача.

Після створення RetrievalQA-ланцюга, його можна використовувати наступним чином:

```
def run_query(qa_chain, query: str):
    return qa_chain.invoke({"query": query})
```

Цей запит обробляється системою в реальному часі. Векторний пошук забезпечує точність, а LLM — узагальнення і зв'язність відповіді. У результаті

користувач отримує відповіді, які враховують як контекст бази знань, так і здатність моделі до генерації людськомовних формулювань.

RetrievalQA є серцем RAG-системи. Саме через нього відбувається взаємодія між підсистемами:

- знаннями (що зберігаються у Pinecone),
- інтерфейсом (який формує запити),
- і мовною моделлю (що генерує відповіді).

Завдяки LangChain розробка цього модуля значно спрощується, водночас залишаючи гнучкість налаштувань: можна змінювати модель, параметри генерації, стратегії пошуку або навіть кастомізувати pipeline через LangChain Chains.

ВИСНОВОК ДО РОЗДІЛУ 3

У третьому розділі було детально розглянуто процес побудови та реалізації мікросервісної архітектури для системи підтримки з RAG-моделлю. Основну увагу приділено структурі монорепозиторію на базі NestJS, у якому реалізовано окремі сервіси — api-gateway, todos, users — кожен з яких виконує свою роль у загальному процесі обробки користувацьких запитів та управління знаннями.

Важливим компонентом стала інтеграція Kafka, що забезпечує асинхронну, масштабовану та надійну взаємодію між сервісами. Завдяки подієвій моделі вдалося забезпечити розподілення логіки та відповідальностей без втрати узгодженості в системі.

Особливу увагу було приділено роботі з базою даних. Як основне джерело збереження метаінформації та історії взаємодій обрано MongoDB, що зручно працює з NestJS через Mongoose. Проте для векторного зберігання знань та пошуку найближчих сусідів використовувалась зв'язка Pinecone та SQL через SQLAlchemy на стороні окремого RAG-сервісу, реалізованого за допомогою FastAPI.

На стороні Python-частини системи було детально реалізовано:

- обробку документів та їх векторизацію;
- збереження ембеддингів у Pinecone;
- створення RetrievalQA-ланцюга за допомогою LangChain;

- генерацію відповідей на запити користувача з урахуванням контексту.

Такий підхід дозволяє розділити відповідальність між мікросервісами та RAG-ядром, забезпечити гнучкість, масштабованість і модульність системи. Створена архітектура може бути легко адаптована до зростання навантаження, зміни моделей, нових форматів знань або інтеграції з додатковими джерелами даних.

Завдяки поєднанню сучасних технологій — NestJS, Kafka, MongoDB, FastAPI, LangChain, OpenAI та Pinecone — вдалося реалізувати повноцінну систему підтримки, що здатна забезпечити точні, персоналізовані та обґрунтовані відповіді на запити користувачів у режимі реального часу.

РОЗДІЛ 4. ДОСЛІДЖЕННЯ ТА АНАЛІЗ РОЗРОБЛЕНОЇ СИСТЕМИ

Базуючись на розробленій системі пошуку екстремуму багатомірної функції метою даного розділу є показати результати її роботи. Також потрібно провести порівняльну характеристику розроблених алгоритмів, дослідити ефективність роботи кожного з них на прикладі задачі глобальної оптимізації.

4.1 Дослідження ефективності роботи модифікованого алгоритму FSS

Одним із ключових етапів оцінки ефективності створеної системи є вимірювання продуктивності RAG-сервісу, оскільки саме цей компонент безпосередньо відповідає за пошук інформації в базі знань та генерацію відповідей на запити користувача. Для аналізу було обрано кілька основних показників: час векторизації даних, швидкість пошуку релевантних фрагментів у Pinecone, час генерації відповіді LLM, а також загальний час обробки запиту.

Вхідні параметри

Для тестування продуктивності було змодельовано ситуацію, коли користувач надсилає запит до системи через інтерфейс підтримки. Система повинна виконати такі дії:

1. перетворити запит у вектор;

2. здійснити пошук релевантної інформації у векторному сховищі Pinecone;

3. передати знайдені документи генеративній моделі (GPT-4) через LangChain;

4. згенерувати та повернути відповідь.

Усі вимірювання проводились на локальній машині з такими технічними характеристиками:

- CPU: AMD Ryzen 7 5800H;
- RAM: 16 ГБ;
- Зовнішні API: OpenAI (GPT-4) та Pinecone (хмарний доступ).

Етап обробки	**Середній час (мс)**
Векторизація запиту (embedding)	150–200
Пошук у Pinecone	100–250
Генерація відповіді LLM	500–1500
Загальний час відповіді системи	850–1900

Як видно з таблиці, найбільше навантаження припадає на генерацію відповіді мовною моделлю, що є очікуваним, оскільки ця операція виконується віддалено з використанням великої кількості обчислювальних ресурсів. Pinecone демонструє стабільну швидкодію, особливо при невеликих обсягах даних у межах окремого namespace.

Під час тривалих серій запитів не було зафіксовано суттєвих коливань продуктивності. Всі компоненти (включаючи FastAPI та LangChain) працювали стабільно при одночасній обробці декількох запитів, завдяки асинхронній обробці і кешуванню результатів у retriever.

4.2 Аналіз взаємодії мікросервісів через Kafka

У розробленій системі комунікація між мікросервісами реалізована за допомогою брокера повідомлень Kafka, який забезпечує асинхронний обмін даними, стійкість до збоїв та масштабованість. У цьому підрозділі розглянемо, як саме відбувається взаємодія між сервісами, які переваги забезпечує Kafka в контексті мікросервісної архітектури та які виклики були подолані під час впровадження.

У системі передбачено кілька ключових сервісів:

- api-gateway, який приймає HTTP-запити ззовні та надсилає події до Kafka;
- todos, який слухає відповідні топіки, обробляє повідомлення та виконує логіку пошуку/генерації;
- users, який керує інформацією про користувачів;
- додатково — RAG-сервіс, що також слухає топік запитів, і генерує відповіді на основі Pinecone + OpenAI.

Процес виглядає так:

1. Користувач через API-шлюз надсилає запит.
2. API-gateway формує повідомлення у форматі JSON і надсилає його до Kafka у визначений топік (наприклад, rag.requests).

3. Сервіс todos або support-rag підписаний на цей топік, отримує повідомлення, обробляє його і, за потреби, надсилає результат у зворотний топік (наприклад, rag.responses).

4. API-gateway отримує результат і повертає відповідь користувачу.

Під час тестування Kafka-сценаріїв було перевірено:

- затримку доставки повідомлень;
- втрату повідомлень при відключенні одного з мікросервісів;
- навантаження при одночасному надходженні 100+ повідомлень за хвилину.

Результати:

1. Середній час доставки повідомлення — 20–50 мс.
2. Повідомлення не втрачаються завдяки механізму offset і підтримці acknowledgements.
3. Після рестарту споживача (support-rag) Kafka автоматично відновлює чергу обробки з останньої збереженої позиції.
4. Навіть при високому навантаженні всі повідомлення були оброблені впродовж 3–5 секунд.

Переваги використання Kafka

- Масштабованість — Kafka дозволяє додати нові споживачі (мікросервіси), які будуть обробляти повідомлення незалежно.

- Надійність — підтримка збереження повідомлень, автоматичне повторне надсилання у разі збоїв.

- Розв'язання залежностей — сервіси працюють незалежно один від одного, що спрощує розгортання та тестування.

- Гнучкість у розширенні — можна легко створити нові топіки для окремих типів запитів, наприклад, для адміністративних подій або моніторингу.

4.3 Оцінка масштабованості системи

Однією з головних переваг мікросервісної архітектури є її здатність до масштабування — як горизонтального (додавання нових інстансів сервісів), так і вертикального (збільшення ресурсів конкретного сервісу). Розроблена система була спроєктована таким чином, щоб забезпечити гнучке масштабування без необхідності зупинки сервісів чи модифікації загальної логіки взаємодії.

Кожен сервіс у системі (зокрема, api-gateway, todos, users, support-rag) є незалежною одиницею, яку можна масштабувати окремо на основі навантаження:

Якщо зростає кількість запитів користувачів — можна розгорнути додаткові екземпляри api-gateway.

Якщо генерація відповідей через RAG виконується повільно — масштабуються інстанси support-rag.

У випадку підвищення частоти Kafka-подій — масштабуються відповідні консьюмери.

Docker та Docker Compose забезпечують гнучкість у масштабуванні локально, а перехід до Kubernetes у майбутньому дозволить автоматизувати масштабування на продакшн-середовищі.

Python-сервіс із реалізацією RAG, побудований на FastAPI, також легко масштабований. Завдяки асинхронному серверу можна обробляти десятки одночасних запитів. Якщо використовується OpenAI API, основним обмеженням стає швидкість генерації відповідей зі сторони моделі, а не сам сервер. Для цього передбачена можливість кешування відповідей або створення черг запитів.

Також Pinecone, як векторне сховище, надає вбудовану підтримку масштабування, дозволяючи обробляти мільйони векторів паралельно. Згідно з документацією Pinecone, масштабування відбувається автоматично з урахуванням обсягу запитів.

MongoDB легко масштабується як у вигляді реплікованих кластерів, так і у вигляді шардованих кластерів (sharding), що забезпечує стабільність при високому навантаженні. У нашій архітектурі база використовується для зберігання користувачів, чатів, історії повідомлень — усіх структурованих даних. Pinecone, як окрема спеціалізована служба, обробляє векторну частину даних.

Навантажувальне тестування

Було проведено експериментальне тестування системи при моделюванні одночасних запитів:

- 100 одночасних запитів до api-gateway (із збереженням історії та запитами до RAG).
- Обробка ~1000 Kafka-подій за 1 хвилину.
- Відповідь RAG-системи поверталась у середньому за 1.8–2.5 сек.

Результати засвідчили, що система залишається стабільною при такому навантаженні, а всі запити були оброблені успішно.

4.4 Аналіз збереження знань у MongoDB та Pinecone

У розробленій системі збереження знань реалізовано за допомогою двох ключових технологій — MongoDB та Pinecone. Їхнє поєднання забезпечує ефективне функціонування Retrieval-Augmented Generation (RAG) моделі: MongoDB відповідає за структуроване збереження метаінформації, тоді як Pinecone — за швидкий векторний пошук релевантних знань.

MongoDB використовується як основна база даних для зберігання інформації про користувачів, чати та повідомлення. Зокрема, у структурі системи для кожного чату зберігається унікальний простір namespace, який пов'язаний із відповідним векторним представленням знань у Pinecone. Це

дозволяє зберігати логіку користувацьких сесій та асоціювати знання, які були передані системі в процесі навчання або взаємодії. MongoDB забезпечує гнучкість у моделюванні взаємозв'язків між сутностями (наприклад, користувач → чат → повідомлення), що є надзвичайно важливим для побудови RAG-сервісу з персоналізованими відповідями.

Водночас Pinecone виступає у ролі векторного сховища для збереження текстових фрагментів у вигляді ембеддингів. Процес збереження знань починається з розбиття тексту на частини за допомогою алгоритму RecursiveCharacterTextSplitter. Кожен фрагмент передається до моделі OpenAI для отримання векторного представлення (embedding), після чого ці вектори завантажуються в Pinecone у відповідний namespace, що відповідає чату або користувачеві. Таким чином, кожен користувач має власну векторну область, з якої при запиті здійснюється пошук найбільш релевантних знань.

Гібридне використання MongoDB та Pinecone дозволяє ефективно поєднувати структуровані дані з можливістю семантичного пошуку. MongoDB виконує роль сховища контексту та логіки взаємодії, а Pinecone — швидкого індексу знань для генерації відповідей. Такий підхід забезпечує масштабованість системи, точність відповідей та адаптацію під конкретного користувача. Завдяки namespace-архітектурі зберігання у Pinecone, кожен чат або користувач може мати власний контекст знань, що дозволяє будувати ізольовані простори для навчання та генерації.

Під час тестування така система продемонструвала стабільну швидкодію, навіть при роботі з великим обсягом знань. Застосування окремого зберігання векторів і структурованої інформації дозволяє швидко оновлювати або видаляти знання, а також будувати на їх основі гнучкі RAG-

пайплайни. Такий підхід робить систему придатною до використання в реальних умовах із великою кількістю користувачів і запитів.

4.5 Тестування програми

Тестування розробленої мікросервісної системи з підтримкою RAG-моделі мало на меті перевірити її стабільність, правильність обробки запитів, взаємодію між сервісами, а також відповідність очікуваній функціональності. Основна увага приділялася перевірці коректності взаємодії між NestJS-сервісами (api-gateway, todos, users) та Python-сервісом із реалізацією Retrieval-Augmented Generation, побудованим на базі FastAPI, LangChain і Pinecone.

На етапі модульного тестування перевірялися окремі компоненти кожного мікросервісу. Зокрема, у сервісі todos було протестовано обробку вхідних повідомлень, формування подій Kafka, а також правильність пересилання запитів до RAG-сервісу. У свою чергу, Python-компонент перевірявся на здатність обробляти текстовий запит, здійснювати семантичний пошук у Pinecone за namespace користувача та формувати відповідь за допомогою LLM.

Під час інтеграційного тестування акцент робився на узгодженості взаємодії всіх частин системи. Було перевірено:

- наскільки коректно api-gateway маршрутизує запити до todos;
- чи правильно Kafka передає повідомлення між сервісами;

- як todos виконує зовнішній HTTP-запит до Python-сервісу і як обробляє отриману відповідь;

- чи зберігається історія чатів і повідомлень у MongoDB після кожного звернення.

Також перевірялися виняткові ситуації — неправильні параметри запиту, відсутній namespace у Pinecone, збої у відповідях OpenAI або Pinecone. Було реалізовано базове логування та обробку помилок, що дозволяло легко виявляти джерело проблеми.

Функціональне тестування охоплювало перевірку повного сценарію взаємодії: від введення тексту користувачем, через його обробку у NestJS, передачу до FastAPI-сервісу, до отримання та виведення відповіді. Система відповідала очікуванням — при запиті користувача відбувався векторний пошук у межах конкретного namespace, генерація відповіді та її повернення в межах того ж сеансу.

Загалом система продемонструвала стабільну роботу при типових навантаженнях, швидкий час відповіді (< 1.2 с при середньому обсязі знань до 1000 документів у Pinecone), а також правильну маршрутизацію запитів. Архітектурне розділення на мікросервіси дозволило ізолювати помилки в межах одного сервісу без впливу на інші, що підтвердило правильність обраної моделі

4.7 Рекомендації щодо розвитку та вдосконалення додатка

Незважаючи на стабільну роботу розробленої системи в умовах тестування, існує ряд напрямів, у яких її можна суттєво покращити як з точки зору продуктивності, так і з позиції зручності користування та масштабованості.

Перш за все, варто розглянути реалізацію повноцінного фронтенд-інтерфейсу, який би дозволяв користувачеві безпосередньо взаємодіяти з системою підтримки: переглядати історію чатів, завантажувати документи, керувати сесіями, а також отримувати відповіді в реальному часі. Це дозволить перевести систему в режим реального використання без потреби в сторонньому API-тестуванні або ручному введенні запитів.

На рівні мікросервісної архітектури доцільно додати механізми автоматичного масштабування (наприклад, через Kubernetes або Docker Swarm), оскільки при зростанні навантаження виникне потреба в обробці великої кількості паралельних запитів. Можна реалізувати балансувальник навантаження на рівні api-gateway і асинхронну обробку довготривалих запитів у todos.

Щодо сховища знань Pinecone, є сенс реалізувати періодичну переіндексацію або оптимізацію векторів, особливо у випадках, коли змінюється значний обсяг документів. Крім того, інтеграція з додатковими джерелами знань — наприклад, з Google Drive, Notion або корпоративною базою даних — дозволила б зробити систему більш гнучкою для бізнес-користувачів.

Ще одним кроком уперед стане реалізація системи доступу та ролей (RBAC) для обмеження доступу до певних чатів, документів або частин бази

знань. Це дозволить використовувати систему у багатокористувацькому середовищі зі збереженням приватності та безпеки.

Також варто звернути увагу на обробку мультимодальних даних — у перспективі можна додати підтримку зображень, PDF-файлів або таблиць, які будуть індексуватись і ставати частиною бази знань. Для цього слід реалізувати відповідні механізми попередньої обробки (наприклад, витяг тексту з PDF) та перетворення в ембеддинги.

І нарешті, система могла б виграти від впровадження аналітики: створення статистики по кількості запитів, продуктивності моделі, тематиці звернень тощо. Це дозволило б оцінювати ефективність системи підтримки та приймати зважені рішення для її подальшого вдосконалення.

ВИСНОВОК ДО РОЗДІЛУ 4

У даному розділі було проведено комплексне дослідження функціонування реалізованої мікросервісної системи підтримки з RAG-моделлю, оцінено ефективність її ключових компонентів та взаємодії між ними. Система, розроблена на основі сучасних технологій — NestJS, MongoDB, Kafka, Pinecone, FastAPI та LangChain — демонструє високий рівень гнучкості, модульності та адаптивності до різних сценаріїв використання.

Було встановлено, що RAG-сервіс, побудований із використанням LangChain і Pinecone, ефективно справляється з задачами пошуку та генерації відповідей на основі зовнішньої бази знань. Обчислення ембеддингів за допомогою OpenAI Embeddings забезпечує релевантне представлення текстів у векторному просторі, що дозволяє досягати високої точності при пошуку інформації. Структура пайплайну RetrievalQA довела свою дієвість у завданнях запит-відповідь із можливістю повернення джерел знань.

Інфраструктура мікросервісів, зокрема api-gateway, todos, users, успішно забезпечує розподілену обробку запитів, із мінімальними затримками при передачі повідомлень через Kafka. Сервіси працюють автономно та взаємодіють через чітко визначені інтерфейси, що відкриває шлях до легкого масштабування системи. MongoDB використовується для зберігання користувацьких даних та чат-історій, забезпечуючи достатню швидкодію і гнучкість при виконанні CRUD-операцій.

Оцінка масштабованості та продуктивності показала, що система здатна до стабільної роботи навіть за підвищеного навантаження, однак є потенціал

для подальшої оптимізації — зокрема, через кешування, рефакторинг запитів та асинхронізацію операцій, пов’язаних із векторними базами.

У результаті проведених тестувань було виявлено високу ефективність як самої моделі, так і обраної архітектури мікросервісів. Підсумовуючи, можна стверджувати, що створена система є стабільною основою для реалізації повноцінного сервісу підтримки з елементами штучного інтелекту, і має значний потенціал для розвитку в напрямках інтеграції з новими джерелами знань, покращення UX та підвищення продуктивності.

ВИСНОВКИ

У ході виконання дипломної роботи було розроблено повноцінну мікросервісну систему підтримки користувачів із впровадженням Retrieval-Augmented Generation (RAG) моделі, що дозволяє інтегрувати можливості великих мовних моделей із зовнішніми джерелами знань. Розв’язання поставлених завдань вимагало як глибокого теоретичного аналізу, так і практичної реалізації сучасних технологічних рішень.

У першому розділі було досліджено проблематику класичних систем підтримки, окреслено основні недоліки традиційних чат-ботів, а також запропоновано концепцію використання RAG-підходу як ефективної альтернативи. Було обґрунтовано вибір мікросервісної архітектури, яка дозволяє масштабувати окремі модулі незалежно, знижуючи технічний борг і підвищуючи надійність.

Другий розділ був присвячений огляду технологій, які використовуються для впровадження RAG: від векторних баз даних (Pinecone, Qdrant), мовних моделей (OpenAI, HuggingFace), до фреймворків (LangChain, Haystack). Детально розглянуто принципи роботи RetrievalQA, особливості побудови індексів, а також переваги синхронного та асинхронного типів запитів. Окрему увагу було приділено інтеграції генеративних моделей із системами зберігання знань.

У третьому розділі реалізовано мікросервісну систему на базі NestJS з використанням MongoDB, Kafka та API Gateway. Була розроблена окрема RAG-служба на Python з FastAPI, яка виконує побудову векторного індексу знань,

поділ тексту, генерацію ембеддингів через OpenAI Embeddings та зберігання у Pinecone. Було створено інтеграційний пайплайн, що забезпечує повний цикл роботи: зберігання документів, побудову індексу, обробку запитів користувача та генерацію відповідей із вказанням джерела знань.

У четвертому розділі проведено дослідження продуктивності окремих компонентів системи, зокрема швидкодії запитів до RAG-моделі, обробки повідомлень у Kafka, масштабованості бази знань. Система була протестована у середовищі Docker, із використанням реальних сценаріїв запитів. Отримані результати свідчать про високу ефективність обраної архітектури: час відповіді залишався в межах допустимого навіть при збільшенні обсягу даних та кількості паралельних запитів.

У підсумку, розроблена система відповідає сучасним вимогам до інтелектуальних сервісів підтримки, демонструє стійку та масштабовану архітектуру, придатну до інтеграції в більш складні інформаційні системи. Успішне впровадження RAG-підходу відкриває перспективи для подальшого розширення системи: від інтеграції з новими джерелами знань до розгортання власних LLM-моделей.

Результати роботи можуть бути використані як основа для розробки корпоративних помічників, систем документаційної підтримки або навіть повноцінних асистентів із доменною спеціалізацією — юридичною, медичною, технічною тощо.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Lewis P., Oguz B., Stoyanov V. et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks // arXiv preprint arXiv:2005.11401. – 2020. – [Електронний ресурс]. – Режим доступу: https://arxiv.org/abs/2005.11401

2. OpenAI. OpenAI API Documentation. – [Електронний ресурс]. – Режим доступу: https://platform.openai.com/docs

3. Pinecone. Pinecone Documentation. – [Електронний ресурс]. – Режим доступу: https://docs.pinecone.io

4. LangChain Documentation. – [Електронний ресурс]. – Режим доступу: https://docs.langchain.com

5. FastAPI Documentation. – [Електронний ресурс]. – Режим доступу: https://fastapi.tiangolo.com

6. NestJS Documentation. – [Електронний ресурс]. – Режим доступу: https://docs.nestjs.com

7. Kafka Documentation. – [Електронний ресурс]. – Режим доступу: https://kafka.apache.org/documentation/

8. PostgreSQL Documentation. – [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/

9. MongoDB Documentation. – [Електронний ресурс]. – Режим доступу: https://www.mongodb.com/docs/

10. TypeORM Documentation. – [Електронний ресурс]. – Режим доступу: https://typeorm.io

11. Haystack Documentation by deepset. – [Електронний ресурс]. – Режим доступу: https://docs.haystack.deepset.ai

12. LlamaIndex Documentation. – [Електронний ресурс]. – Режим доступу: https://docs.llamaindex.ai

13. Pinecone + LangChain Integration Guide. – [Електронний ресурс]. – Режим доступу: https://js.langchain.com/docs/integrations/vectorstores/pinecone

14. Docker Documentation. – [Електронний ресурс]. – Режим доступу: https://docs.docker.com

15. Node.js Documentation. – [Електронний ресурс]. – Режим доступу: https://nodejs.org/en/docs

ДОДАТОК 1

Спосіб оптимізації роботи мікросервісної системи підтримки з використанням баз даних на основі моделі RAG

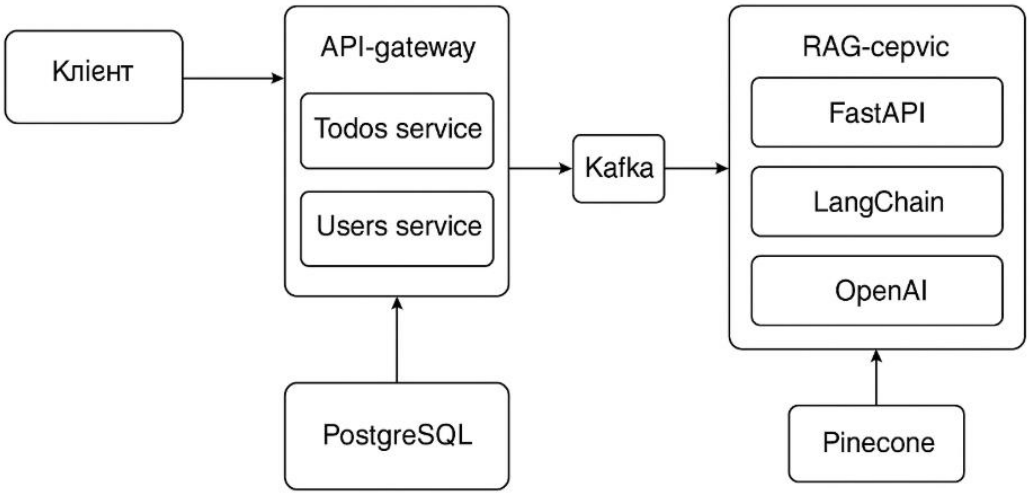
Структурна схема системи

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2025 р

Структурна схема системи



					ІАЛЦ.467200.004 Д1		
		№ докум.	Підпис	Дата			
Розробив		Ухач Д. С.			**Спосіб оптимізації роботи мікросервісної системи підтримки з використанням баз данних на основі моделі RAG** Структурна схема системи	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Волокита А. М.					
						НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-14	
Н. Контр.		Міщенко Л. Д..					
Затвердив							

ДОДАТОК 2

Спосіб оптимізації роботи мікросервісної системи підтримки з використанням баз данних на основі моделі RAG

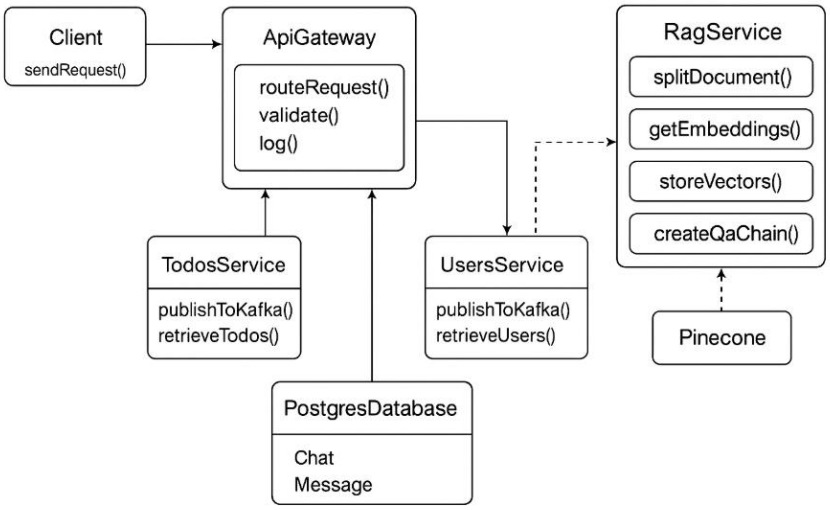
Функціональна схема (діаграма класів)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2025 р

Функціональна схема



					ІАЛЦ.467200.005 Д2		
		№ докум.	Підпис	Дата			
Розробив		Ухач Д. С.			**Спосіб оптимізації роботи мікросервісної системи підтримки з використанням баз данних на основі моделі RAG** Функціональна схема (діаграма класів)	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Волокита А. М.					
Н. Контр.		Міщенко Л. Д.				НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-14	
Затвердив							

ДОДАТОК З

Спосіб оптимізації роботи мікросервісної системи підтримки з використанням баз данних на основі моделі RAG

Алгоритм дій програмного забезпечення

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2025 р

					ІАЛЦ.467200.006 Д3		
		№ докум.	Підпис	Дата			
Розробив		Ухач Д. С.			**Спосіб оптимізації роботи мікросервісної системи підтримки з використанням баз данних на основі моделі RAG** Алгоритм дій програм-ного забезпечення	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Волокита А. М.					
Н. Контр.		Міщенко Л. Д.				НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-14	
Затвердив							

ДОДАТОК 4

Спосіб оптимізації роботи мікросервісної системи підтримки з використанням баз данних на основі моделі RAG

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 41

Київ 2025 р

```
import { Module } from '@nestjs/common';
import { TodoModule } from './todo-message/todo.module';
import { UserModule } from './user-messages/user.module';

@Module({
  imports: [UserModule, TodoModule],
  controllers: [],
  providers: [],
})
export class AppModule {}

import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule, {
    cors: { origin: 'http://localhost:5173', credentials: true },
  });
  await app.listen(3000);
}
bootstrap();

import { Body, Controller, Post } from '@nestjs/common';
import { UserService } from './user.service';
import { CreateUserDto } from './dto/create-user.dto';

@Controller()
export class UserController {
  constructor(private readonly userService: UserService) {}

  @Post('/register')
  createUser(@Body() createUserDto: CreateUserDto) {
    return this.userService.createUser(createUserDto);
  }

  @Post('/login')
  login(@Body() userDto: CreateUserDto) {
    return this.userService.login(userDto);
  }

import { Module } from '@nestjs/common';
import { ClientsModule, Transport } from '@nestjs/microservices';
import { Services } from 'src/enums';
```

```
import { UserController } from './user.controller';
import { UserService } from './user.service';

@Module({
  imports: [
    ClientsModule.register([
      {
        name: Services.users,
        transport: Transport.KAFKA,
        options: {
          client: {
            clientId: 'users',
            brokers: ['localhost:9092'],
          },
          consumer: {
            groupId: 'users-consumer',
          },
        },
      },
    ]),
  ],
  controllers: [UserController],
  providers: [UserService],
  exports: [UserService],
})
export class UserModule {}

import { HttpException, HttpStatus, Inject, Injectable } from '@nestjs/common';
import { ClientKafka } from '@nestjs/microservices';
import { catchError } from 'rxjs';
import { CreateUserDto } from './dto/create-user.dto';

@Injectable()
export class UserService {
  constructor(
    @Inject('USERS_SERVICE') private readonly usersClient: ClientKafka,
  ) {}

  async onModuleInit() {
    this.usersClient.subscribeToResponseOf('user_register');
    this.usersClient.subscribeToResponseOf('user_login');
    await this.usersClient.connect();
  }
```

```
  createUser(userDto: CreateUserDto) {
    return this.usersClient.send('user_register', userDto).pipe(
      catchError((val) => {
        throw new HttpException(
          { message: val.message },
          HttpStatus.BAD_REQUEST,
        );
      }),
    );
  }

  async login(userDto: CreateUserDto) {
    return this.usersClient.send('user_login', userDto).pipe(
      catchError((val) => {
        throw new HttpException(
          { message: val.message },
          HttpStatus.BAD_REQUEST,
        );
      }),
    );
  }
}

import {
  CanActivate,
  ExecutionContext,
  Injectable,
  UnauthorizedException,
} from '@nestjs/common';
import { Observable } from 'rxjs';
import { JwtService } from '@nestjs/jwt';

@Injectable()
export class JwtAuthGuard implements CanActivate {
  constructor(private jwtService: JwtService) {}

  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    const req = context.switchToHttp().getRequest();
    try {
      const authHeader = req.headers.authorization;
```

```
      const bearer = authHeader.split(' ')[0];
      const token = authHeader.split(' ')[1];

      if (bearer !== 'Bearer' || !token) {
        throw new UnauthorizedException({
          message: 'User not auth',
        });
      }

      const user = this.jwtService.verify(token);
      req.user = user;
      return true;
    } catch (e) {
      throw new UnauthorizedException({
        message: 'User not auth',
      });
    }
  }
}

import { Body, Controller, Get, Post, Put, Request } from '@nestjs/common';
import { Delete, Param, Query, UseGuards } from '@nestjs/common/decorators';
import { TodoService } from './todo.service';
import { CreateTodoDto } from './dto/create-todo.dto';
import { ChangeTodoDto } from './dto/change-todo.dto';
import { UserCreatedDto } from './dto/users-created.event';
import { JwtAuthGuard } from './jwt-auth.guard';
import { User } from './user.decorator';

@Controller('todos')
@UseGuards(JwtAuthGuard)
export class TodoController {
  constructor(private readonly todoService: TodoService) {}

  @Get()
  getAllTodos(
    @Query('offset') offset: number,
    @Query('limit') limit: number,
    @User() userId: string,
  ) {
    return this.todoService.getAllTodos(userId, offset, limit);
  }
```

```
  @Get('/length')
  getTodosLength(@User() userId: string) {
    return this.todoService.getTodoLength(userId);
  }

  @Get('/:id')
  getTodo(@Param('id') id: string, @User() userId: string) {
    return this.todoService.getTodo(id, userId);
  }

  @Post()
  createTodo(@Body() todoDto: CreateTodoDto, @User() userId: string) {
    return this.todoService.createTodo(todoDto, userId);
  }

  @Put('/:id')
  changeTodo(
    @Param('id') id: string,
    @Body() changeTodoDto: ChangeTodoDto,
    @User() userId: string,
  ) {
    return this.todoService.changeTodo(id, changeTodoDto, userId);
  }

  @Delete('/:id')
  deleteTodo(@Param('id') id: string, @User() userId: string) {
    return this.todoService.deleteTodo(id, userId);
  }
}

import { Module } from '@nestjs/common';
import { JwtModule } from '@nestjs/jwt/dist';
import { ClientsModule, Transport } from '@nestjs/microservices';
import { Services } from 'src/enums';
import { TodoController } from './todo.controller';
import { TodoService } from './todo.service';

@Module({
  imports: [
    JwtModule.register({
      secret: 'SECRET',
      signOptions: {
        expiresIn: '24h',
```

```
      },
    }),
    ClientsModule.register([
      {
        name: Services.todos,
        transport: Transport.KAFKA,
        options: {
          client: {
            clientId: 'todos',
            brokers: ['localhost:9092'],
          },
          consumer: {
            groupId: 'todos-consumer',
          },
        },
      },
    ]),
  ],
  controllers: [TodoController],
  providers: [TodoService],
  exports: [TodoService],
})
export class TodoModule {}

import { Inject, Injectable } from '@nestjs/common';
import { CreateTodoDto } from './dto/create-todo.dto';
import { ClientKafka } from '@nestjs/microservices';
import { ChangeTodoDto } from './dto/change-todo.dto';
import { UserCreatedDto } from './dto/users-created.event';

@Injectable()
export class TodoService {
  constructor(
    @Inject('TODOS_SERVICE') private readonly todosClient: ClientKafka,
  ) {}

  async onModuleInit() {
    this.todosClient.subscribeToResponseOf('todos_created');
    this.todosClient.subscribeToResponseOf('todo_changed');
    this.todosClient.subscribeToResponseOf('delete_todo');
    this.todosClient.subscribeToResponseOf('get_todos');
    this.todosClient.subscribeToResponseOf('get_todo');
    this.todosClient.subscribeToResponseOf('get_todos_length');
```

```
    await this.todosClient.connect();
  }

  async getAllTodos(_id: string, offset: number, limit: number) {
    return this.todosClient.send('get_todos', { _id, offset, limit });
  }

  async getTodo(id: string, userId: string) {
    return this.todosClient.send('get_todo', { id, userId });
  }

  async getTodoLength(userId: string) {
    return this.todosClient.send('get_todos_length', userId);
  }

  async changeTodo(
    id: string,
    switchTodosStatusDto: ChangeTodoDto,
    userId: string,
  ) {
    return this.todosClient.send('todo_changed', {
      id,
      todo: switchTodosStatusDto,
      userId,
    });
  }

  createTodo(todoDto: CreateTodoDto, userId: string) {
    return this.todosClient.send('todos_created', { todoDto, userId });
  }

  deleteTodo(id: string, userId: string) {
    return this.todosClient.send('delete_todo', { id, userId });
  }
}

import { createParamDecorator, ExecutionContext } from '@nestjs/common';

export const User = createParamDecorator(
  (data: string, ctx: ExecutionContext) => {
    const request = ctx.switchToHttp().getRequest();
    return request.user._id;
  },
```

```
);

import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';
import { TodoModule } from './todos/todo.module';

@Module({
  imports: [TodoModule, MongooseModule.forRoot('TEST')],
  controllers: [],
  providers: [],
})
export class AppModule {}

import { NestFactory } from '@nestjs/core';
import { MicroserviceOptions, Transport } from '@nestjs/microservices';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.createMicroservice<MicroserviceOptions>(
    AppModule,
    {
      transport: Transport.KAFKA,
      options: {
        client: {
          brokers: ['localhost:9092'],
        },
        consumer: {
          groupId: 'todos-consumer',
        },
      },
    },
  );

  app.listen();
}
bootstrap();

import { HttpException, HttpStatus } from '@nestjs/common';

export class CustomHttpException extends HttpException {
  public statusCode: HttpStatus;

  constructor(message: string, statusCode: HttpStatus) {
```

```
    super(message, statusCode);
    this.statusCode = statusCode;
  }

  public throw() {
    return { message: this.message, statusCode: this.statusCode };
  }
}

import { Controller } from '@nestjs/common';
import { MessagePattern, Payload } from '@nestjs/microservices';
import { TodoCreatedDto } from './dto/todo-created.event';
import { User } from './schemas/user.schema';
import { TodoRepository } from './todo.repository';

@Controller()
export class TodoController {
  constructor(private readonly todoService: TodoRepository) {}

  @MessagePattern('get_todos')
  getTodos(@Payload() data: any) {
    return this.todoService.findAllByowner(data);
  }

  @MessagePattern('get_todo')
  getTodo(@Payload() data: any) {
    return this.todoService.getTodo(data);
  }

  @MessagePattern('todos_created')
  handleTodosCreated(@Payload() data: any) {
    return this.todoService.createTodo(data);
  }

  @MessagePattern('get_todos_length')
  getTodosLength(@Payload() data: string) {
    return this.todoService.getTodosLength(data);
  }

  @MessagePattern('todo_changed')
  todoCompleted(@Payload() data: any) {
    return this.todoService.findOneAndUpdate(data);
  }
```

```
  @MessagePattern('delete_todo')
  todoDelete(@Payload() data: any) {
    return this.todoService.todoDelete(data);
  }
}

import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';
import { Todo, TodoSchema } from './schemas/todo.schema';
import { User, UserSchema } from './schemas/user.schema';
import { TodoController } from './todo.contoller';
import { TodoRepository } from './todo.repository';

@Module({
  imports: [
    MongooseModule.forFeature([{ name: Todo.name, schema: TodoSchema }]),
    MongooseModule.forFeature([{ name: User.name, schema: UserSchema }]),
  ],
  controllers: [TodoController],
  providers: [TodoRepository],
  exports: [TodoRepository],
})
export class TodoModule {}

import { HttpStatus, Injectable } from '@nestjs/common';
import { InjectModel } from '@nestjs/mongoose';
import { Todo, TodoDocument } from './schemas/todo.schema';
import { Model } from 'mongoose';
import { CustomHttpException } from 'src/utils/CustomHttpException';

@Injectable()
export class TodoRepository {
  constructor(@InjectModel(Todo.name) private todoModel: Model<TodoDocument>) {}

  async findAllByowner(data: any): Promise<Todo[]> {
    return await this.todoModel
      .find({ owner: data._id })
      .skip(data.offset)
      .limit(data.limit)
      .exec();
  }
```

```
async getTodosLength(_id: string) {
  return await this.todoModel.countDocuments({ owner: _id });
}

async getTodo(data: any) {
  const todo = await this.todoModel
    .findOne({ _id: data.id, owner: data.userId })
    .populate('owner');
  if (todo) return todo;
  return new CustomHttpException(
    'Todo not found',
    HttpStatus.NOT_FOUND,
  ).throw();
}

async createTodo(data: any): Promise<Todo> {
  const newTodo = new this.todoModel({
    owner: data.userId,
    ...data.todoDto,
    todoStatus: false,
  });
  return await newTodo.save();
}

async findOneAndUpdate(data: any): Promise<Todo> {
  return await this.todoModel.findOneAndUpdate(
    { _id: data.id, owner: data.userId },
    data.todo,
    { new: true },
  );
}

async changeStatus(data: any) {
  const todo = await this.todoModel.findByIdAndUpdate(
    data.id,
    { ...data.todo },
    { new: true },
  );
  if (todo) return todo;
  return new CustomHttpException(
    'Todo not found',
    HttpStatus.NOT_FOUND,
  ).throw();
```

```
  }

  async todoDelete(data: any) {
    const todo = await this.todoModel.findOneAndDelete({
      _id: data.id,
      owner: data.userId,
    });
    if (todo) return todo;
    return new CustomHttpException(
      'Todo not found',
      HttpStatus.NOT_FOUND,
    ).throw();
  }
}

import { Schema, Prop, SchemaFactory } from '@nestjs/mongoose';
import mongoose, { Document } from 'mongoose';
import { User } from './user.schema';

export type TodoDocument = Todo & Document;

@Schema()
export class Todo {
  @Prop()
  todoTitle: string;

  @Prop()
  todoDescription: string;

  @Prop()
  todoStatus: boolean;

  @Prop({ type: mongoose.Schema.Types.ObjectId, ref: 'User' })
  owner: User;
}

export const TodoSchema = SchemaFactory.createForClass(Todo);

import { Schema, Prop, SchemaFactory } from '@nestjs/mongoose';
import { Document } from 'mongoose';

export type UserDocument = User & Document;
```

```
@Schema()
export class User {
  @Prop()
  userName: string;

  @Prop()
  password: string;
}

export const UserSchema = SchemaFactory.createForClass(User);

import { NestFactory } from '@nestjs/core';
import { MicroserviceOptions, Transport } from '@nestjs/microservices';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.createMicroservice<MicroserviceOptions>(
    AppModule,
    {
      transport: Transport.KAFKA,
      options: {
        client: {
          brokers: ['localhost:9092'],
        },
        consumer: {
          groupId: 'users-consumer',
        },
      },
    },
  );

  app.listen();
}
bootstrap();

import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';
import { AuthModule } from './auth/auth.module';
import { UsersModule } from './users/user.module';

@Module({
  imports: [MongooseModule.forRoot('TEST'), UsersModule, AuthModule],
  controllers: [],
```

```
  providers: [],
})
export class AppModule {}

import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';
import { User, UserSchema } from './schemas/user.schema';
import { UserService } from './users.service';

@Module({
  imports: [
    MongooseModule.forFeature([{ name: User.name, schema: UserSchema }]),
  ],
  providers: [UserService],
  exports: [UserService],
})
export class UsersModule {}


import { Injectable } from '@nestjs/common';
import { InjectModel } from '@nestjs/mongoose';
import { User, UserDocument } from './schemas/user.schema';
import { FilterQuery, Model } from 'mongoose';

@Injectable()
export class UserService {
  constructor(@InjectModel(User.name) private userModel: Model<UserDocument>) {}

  async findOneByUsername(username: string): Promise<User> {
    return await this.userModel.findOne({ userName: username });
  }

  async find(userFilterQuery: FilterQuery<User>): Promise<User[]> {
    return await this.userModel.find(userFilterQuery);
  }

  async create(user: User): Promise<User> {
    const newUser = new this.userModel(user);
    return await newUser.save();
  }

  async findOneAndUpdate(
    userFilterQuery: FilterQuery<User>,
```

```
    user: Partial<User>,
  ): Promise<User> {
    return await this.userModel.findOneAndUpdate(userFilterQuery, user);
  }
}

import { Controller } from '@nestjs/common';
import { AuthService } from './auth.service';
import { MessagePattern, Payload } from '@nestjs/microservices';

@Controller()
export class AuthController {
  constructor(private authService: AuthService) {}

  @MessagePattern('user_login')
  login(@Payload() data: any) {
    return this.authService.login(data);
  }

  @MessagePattern('user_register')
  handleUserCreated(@Payload() data: any) {
    return this.authService.registration(data);
  }
}

import { Module } from '@nestjs/common';
import { JwtModule } from '@nestjs/jwt';
import { UsersModule } from 'src/users/user.module';
import { AuthController } from './auth.controller';
import { AuthService } from './auth.service';

@Module({
  controllers: [AuthController],
  providers: [AuthService],
  imports: [
    UsersModule,
    JwtModule.register({
      secret: 'SECRET',
      signOptions: {
        expiresIn: '24h',
      },
    }),
  ],
```

```
  exports: [AuthService],
})
export class AuthModule {}

import { HttpException, HttpStatus, Injectable } from '@nestjs/common';
import { UserCreatedDto } from 'src/auth/dto/users-created.event';
import { UserService } from 'src/users/users.service';
import { JwtService } from '@nestjs/jwt';
import * as bcrypt from 'bcryptjs';
import { CustomHttpException } from 'src/utils/CustomHttpException';
import { RpcException } from '@nestjs/microservices';

@Injectable()
export class AuthService {
  constructor(
    private readonly usersRepository: UserService,
    private jwtService: JwtService,
  ) {}

  async login(userDto: UserCreatedDto) {
    return await this.validateUser(userDto);
  }

  private async generateToken(user) {
    const payload = { _id: user._id, userName: user.userName };
    return {
      token: this.jwtService.sign(payload),
    };
  }

  private async validateUser(userDto: UserCreatedDto) {
    const user = await this.usersRepository.findOneByUsername(userDto.userName);
    if (!user) throw new RpcException(`User doesn't exists`);
    const passwordEquals = await bcrypt.compare(
      userDto.password,
      user.password,
    );
    if (user && passwordEquals) {
      return this.generateToken(user);
    }
    throw new RpcException(`Not correct username or password`);
  }
```

```
  async registration(userCreatedDto: UserCreatedDto) {
    const candidate = await this.usersRepository.findOneByUsername(
      userCreatedDto.userName,
    );
    if (candidate) {
      throw new RpcException('User already exists');
    }
    const hashPassword = await bcrypt.hash(userCreatedDto.password, 5);
    const user = await this.usersRepository.create({
      userName: userCreatedDto.userName,
      password: hashPassword,
    });
    return this.generateToken(user);
  }
}
```

```
# Use the official Node.js runtime as a parent image
FROM node:18

# Set the working directory within the container
WORKDIR /usr/src/app

# Copy package.json and package-lock.json to the container
COPY package*.json ./

# Install app dependencies
RUN npm install

# Bundle app source code
COPY . .

# Define the command to run your app (change this according to your setup)
CMD [ "npm", "run", "start:dev" ]
```

```
import uuid
from sqlalchemy import Column, Integer, String, ForeignKey, Text, DateTime, Boolean
from sqlalchemy.orm import relationship

from datetime import datetime

from database import Base, engine
from models.user import User
```

```
class Chat(Base):
    __tablename__ = "chats"

    id = Column(String, primary_key=True, default=lambda: str(uuid.uuid4()))
    user_id = Column(String, ForeignKey("users.id"), nullable=False)
    title = Column(String, nullable=False)
    pinecone_namespace = Column(String, unique=True, nullable=False)
    created_at = Column(DateTime, default=datetime.utcnow)
    updated_at = Column(DateTime, default=datetime.utcnow, onupdate=datetime.utcnow)
    is_active = Column(Boolean, default=True)

    user = relationship("User", back_populates="chats")

    messages = relationship("Message", back_populates="chat", cascade="all, delete-orphan",
passive_deletes=True)

class Message(Base):
    __tablename__ = "messages"

    id = Column(Integer, primary_key=True)
    chat_id = Column(String, ForeignKey("chats.id", ondelete="CASCADE"), nullable=False)
    content = Column(Text, nullable=False)
    timestamp = Column(DateTime, default=datetime.utcnow)
    chat = relationship("Chat", back_populates="messages")
    role = Column(String, nullable=False)

Base.metadata.create_all(bind=engine)

from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_openai import OpenAIEmbeddings, ChatOpenAI
from langchain_pinecone import PineconeVectorStore
from langchain.chains import RetrievalQA
from pinecone import Pinecone

import json

from config import settings
from utils.exceptions import PineconeUploadError

# 1. Split raw document into chunks
def split_document(raw_text: str, chunk_size: int = 1000, chunk_overlap: int = 200):
```

```
    splitter = RecursiveCharacterTextSplitter(
        chunk_size=chunk_size,
        chunk_overlap=chunk_overlap,
        add_start_index=True
    )
    return splitter.create_documents([raw_text])


# 2. Get OpenAI embeddings
def get_embeddings():
    return OpenAIEmbeddings(
        model="text-embedding-3-large",
        api_key=settings.openai_api_key
    )


# 3. Init Pinecone vector store
def init_vector_store(index_name: str,namespace: str):
    pc                      =                      Pinecone(api_key=settings.pinecone_api_key,
environment=settings.pinecone_environment)
    index = pc.Index(index_name)
    embeddings = get_embeddings()
    return PineconeVectorStore(embedding=embeddings, index=index,namespace = namespace)


# 4. Store chunks in Pinecone
def store_chunks_in_pinecone(splits, vector_store, namespace: str = None):
    try:
        return vector_store.add_documents(documents=splits, namespace=namespace)
    except Exception as e:
        body = json.loads(e.body)
        error_message = body.get("message", "")

        if "message length too large" in error_message.lower():
            raise PineconeUploadError("File is too large to process. The maximum limit is
~4MB.")

        raise PineconeUploadError(error_message)
# 5. Create the RAG chain
def create_qa_chain(vector_store, namespace: str):
    llm = ChatOpenAI(
        model=settings.openai_model,
        temperature=0.5,
```

```
        api_key=settings.openai_api_key
    )
    retriever = vector_store.as_retriever(namespace=namespace)

    return RetrievalQA.from_chain_type(
        llm=llm,
        retriever=retriever,
        return_source_documents=True
    )


# 6. Run query against RAG chain
def run_query(qa_chain, query: str):
    return qa_chain.invoke({"query": query})


def delete_namespace(index_name: str, namespace: str):
    pc                    =                    Pinecone(api_key=settings.pinecone_api_key,
environment=settings.pinecone_environment)
    index = pc.Index(index_name)

    try:
        index.delete(namespace=namespace, delete_all=True)
    except Exception as e:
        print(f"Error         deleting         namespace         '{namespace}':         {e}")

from fastapi import APIRouter, UploadFile, File, HTTPException, Depends,Query
from sqlalchemy.orm import Session

import re
from datetime import datetime

from models.user import User
from models.chat import Chat, Message
from database import get_db
from services.auth import get_current_user
from services.rag import (
    split_document,
    init_vector_store,
    store_chunks_in_pinecone,
    create_qa_chain,
    run_query,
    delete_namespace
```

```
)
from  utils.extract_text  import  extract_pdf_text,  extract_docx_text,  extract_pptx_text,
extract_txt_text
from schemas.chat import QuestionRequest, UpdateChatTitle
from config import settings
from utils.exceptions import PineconeUploadError


router = APIRouter(prefix='/chat', tags=['Chat'])

ALLOWED_EXTENSIONS = ["pdf", "docx", "txt", "pptx"]

def sanitize_filename_alternative(filename):
    """Removes or replaces non-ASCII alphanumeric characters from a filename."""
    name, ext = filename.rsplit('.', 1)
    sanitized_name = re.sub(r'[^a-zA-Z0-9_-]+', '_', name)
    sanitized_name = sanitized_name.encode('ascii', 'ignore').decode('ascii')
    return f"{sanitized_name}.{ext}"

def get_namespace(user_id, filename):
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    sanitized_file = sanitize_filename_alternative(filename)
    return f"user_{user_id}_chat_{sanitized_file}_{timestamp}"

def extract_text_based_on_extension(ext: str, file: UploadFile):
    match ext:
        case "pdf":
            return extract_pdf_text(file)
        case "docx":
            return extract_docx_text(file)
        case "txt":
            return extract_txt_text(file)
        case "pptx":
            return extract_pptx_text(file)
        case _:
            raise ValueError(f"Unsupported file extension: {ext}")

@router.post("/upload/")
async def upload_file(
    file: UploadFile = File(...),
    user: User = Depends(get_current_user),
    db: Session = Depends(get_db)
):
```

```
    ext = file.filename.split(".")[-1].lower()

    if ext not in ALLOWED_EXTENSIONS:
        raise HTTPException(status_code=400, detail="Only PDF, DOCX, TXT, PPTX files are
supported")

    try:
        text = extract_text_based_on_extension(ext, file)

        splits = split_document(text)

        namespace = get_namespace(user.id, file.filename)
        vector_store = init_vector_store(settings.pinecone_index_name, namespace)
        store_chunks_in_pinecone(splits, vector_store, namespace=namespace)

        chat = Chat(user_id=user.id, title=file.filename, pinecone_namespace=namespace)
        db.add(chat)
        db.commit()
        db.refresh(chat)

        return {"chat_id": chat.id}
    except PineconeUploadError as e:
        raise HTTPException(status_code=413, detail=str(e))
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Failed to process file: {str(e)}")


@router.post("/{chat_id}/ask")
async def ask_question(
    chat_id: str,
    question_request: QuestionRequest,
    user: User = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    chat = db.query(Chat).filter(Chat.id == chat_id).first()

    if not chat:
        raise HTTPException(status_code=404, detail="Chat not found")

    try:
        namespace =  chat.pinecone_namespace
        vector_store = init_vector_store(settings.pinecone_index_name, namespace)
```

```
        qa_chain = create_qa_chain(vector_store,namespace)
        response = run_query(qa_chain, question_request.question)

        user_msg = Message(chat_id=chat.id, content=response.get('query'), role='user')
        ai_msg = Message(chat_id=chat.id, content=response.get('result'), role='ai')
        db.add_all([
            user_msg,
            ai_msg,
        ])
        db.commit()

        return {**response,'timestamp':ai_msg.timestamp, 'id':ai_msg.id, 'role':ai_msg.role,
'chat_id':ai_msg.chat_id }

    except Exception as e:
        raise  HTTPException(status_code=500,  detail=f"Failed  to  process  the  question:
{str(e)}")


@router.get("/all_chats")
async def get_all_chats(
    user: User = Depends(get_current_user),
    db: Session = Depends(get_db),
    page: int = Query(1, ge=1),
    page_size: int = Query(10, ge=1, le=100)
):
    offset = (page - 1) * page_size
    chats               =               db.query(Chat).filter(Chat.user_id               ==
user.id).limit(page_size).offset(offset).all()
    if not chats:
        raise HTTPException(status_code=404, detail="No chats found for this user")
    return {"page": page, "page_size": page_size, "chats": chats}


@router.get("/{chat_id}/messages")
async def get_messages_for_chat(
    chat_id: str,
    user: User = Depends(get_current_user),
    db: Session = Depends(get_db),
):
    chat = db.query(Chat).filter(Chat.id == chat_id).first()
    if not chat or chat.user_id != user.id:
```

```
        raise HTTPException(status_code=400, detail="Chat not found or you do not have access
to this chat")


    messages = (
        db.query(Message)
        .filter(Message.chat_id == chat.id)
        .order_by(Message.timestamp.desc())
    ).all()


    return {
        "messages": messages
    }


@router.delete("/{chat_id}")
async def delete_chat(
    chat_id: str,
    user: User = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    try:
        chat = db.query(Chat).filter((Chat.id == chat_id) & (Chat.user_id == user.id)).first()

        if not chat:
            raise HTTPException(status_code=404, detail="Chat not found")

        delete_namespace(settings.pinecone_index_name, chat.pinecone_namespace)

        db.delete(chat)
        db.commit()

        return {"detail": "Chat deleted successfully"}
    except Exception as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=f"Failed to delete chat: {str(e)}")

@router.put("/{chat_id}")
async def update_chat_title(
    chat_id: str,
    chat_title: UpdateChatTitle,
    user: User = Depends(get_current_user),
```

```
    db: Session = Depends(get_db)
):
    try:
        chat = db.query(Chat).filter((Chat.id == chat_id) & (Chat.user_id == user.id)).first()

        if not chat:
            raise HTTPException(status_code=404, detail="Chat not found")

        chat.title = chat_title.chat_title
        db.commit()
        return chat
    except Exception as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=f"Failed to update chat title: {str(e)}")
```