Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Навчально-науковий
інститут електроенергетики
(навчально-науковий інститут)
Факультет інформаційних технологій
(факультет)
Кафедра інформаційних технологій та комп'ютерної інженерії
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня магістра

Здобувача вищої освіти Яковлєва Ярослава Юрійовича
(ПІБ)
академічної групи 123м-24-1
(шифр)
спеціальності 123 Комп'ютерна інженерія
(код і назва спеціальності)
за освітньо-професійною програмою «Комп'ютерна інженерія»
(офіційна назва)

на тему «Обґрунтування структури системи автоматизованого обслуговування клієнтів компанії постачальника будівельних інструментів та обладнання з інтеграцією бота з ШІ та базою даних MongoDB»
(назва за наказом ректора)

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
кваліфікаційної роботи	проф. Цвіркун Л.І.			
розділів:				
синтез системи	проф. Цвіркун Л.І.			
розроблення програмного забезпечення	доц. Бешта Л.В.			
Рецензент				
Нормоконтролер	проф. Цвіркун Л.І.			

Дніпро
2025

ЗАТВЕРДЖЕНО:
завідувач кафедри
інформаційних технологій
та комп'ютерної інженерії
(повна назва)
_____ В.В. Гнатушенко
(підпис) (ініціали, прізвище)

«__ ___» _____ 2025 року

ЗАВДАННЯ
на кваліфікаційну роботу
ступеня магістра
(бакалавра, магістра)

здобувача вищої освіти Яковлєва Я.Ю. **академічної групи** 123м-24-1
(прізвище та ініціали) (шифр)

спеціальності 123 Комп'ютерна інженерія

за освітньою-професійною програмою «Комп'ютерна інженерія»
(офіційна назва)

на тему «Обґрунтування структури системи автоматизованого обслуговування клієнтів компанії постачальника будівельних інструментів та обладнання з інтеграцією бота з ШІ та базою даних MongoDB»,
затверджену наказом ректора НТУ «Дніпровська політехніка» від 13 жовтня 2025 р. №1165-с

Розділ	Зміст	Термін виконання
Стан питання та постановка завдання	На основі матеріалів практик, інших науково-технічних джерел сформулювати наукове завдання, конкретизувати предмет та мету досліджень	10.10.2025
Теоретичний	Обґрунтувати теоретичну базу системи автоматизованого обслуговування клієнтів	24.10.2025
Синтез системи	Розробка комп'ютерної системи автоматизованого обслуговування клієнтів	14.11.2025
Розроблення програмного забезпечення	Розробка програмного забезпечення модулів системи автоматизованого обслуговування клієнтів	28.11.2025
Експериментальний розділ	Проведення і обробка результатів експериментів з системою автоматизованого обслуговування клієнтів	05.12.2025

Завдання видано _____ проф. Л. І. Цвіркун
(підпис керівника) (ініціали, прізвище)

Дата видачі 05 вересня 2025 р.

Дата подання до екзаменаційної комісії 10.12.2025 р.

Прийнято до виконання _____ Яковлєв Я.Ю.
(підпис здобувача вищої освіти) (ініціали, прізвище)

РЕФЕРАТ

Пояснювальна записка: 110 с., 23 рис., 1 дод., 15 табл., 50 джерел

GEMINI, MONGODB, CHROMADB, TELEGRAM-БОТ, БАЗА ДАНИХ, КОМП'ЮТЕРНА СИСТЕМА, ШТУЧНИЙ ІНТЕЛЕКТ, RAG

Об'єкт дослідження – система автоматизованого обслуговування клієнтів компанії-постачальника будівельних інструментів та обладнання.

Предметом дослідження є методи та засоби побудови системи автоматизованого обслуговування клієнтів, що поєднує Telegram-бот, штучний інтелект та документоорієнтовану базу даних MongoDB в єдиній архітектурі.

Метою роботи є розробка й дослідження комп'ютерної системи та програмного комлексу автоматизованої підтримки клієнтів на основі ШІ-бота та інтеграції з базою даних MongoDB, а також експериментальна перевірка її працездатності, продуктивності та якості відповідей.

У роботі проведено аналіз сучасних підходів до автоматизації клієнтської підтримки, використання Telegram-ботів, досліджено технологічні основи великих мовних моделей і документоорієнтованих баз даних, розглянуто підхід до інтеграції мовної моделі з корпоративною базою знань на основі MongoDB. Запропоновано багаторівневу розподілену архітектуру системи.

Методи дослідження базуються на апараті теорії масового обслуговування, векторних метрик оцінювання та експериментальних показників роботи системи.

Наукова новизна роботи полягає у формалізації архітектури автоматизованої системи клієнтського сервісу з інтеграцією ШІ великої мовної моделі та документоорієнтованої бази даних MongoDB, у побудові мережевої моделі на основі теорії масового обслуговування та у розробці комплексної методики експериментальної оцінки її показників.

Галузь застосування отриманих результатів – корпоративні контакт-центри та служби підтримки.

ЗМІСТ

С.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

AI, Artificial Intelligence, ШІ – штучний інтелект

API, Application Programming Interface – інтерфейс прикладного програмування

БД – база даних

FSM, Finite State Machine –автомат кінцевих станів

JSON, JavaScript Object Notation – формат обміну даними у вигляді об'єктів

MongoDB – документоорієнтована нереляційна база даних

RAG, Retrieval-Augmented Generation – генерація з доповненням через пошук

LLM, Large Language Model – велика мовна модель

Telegram API – інтерфейс взаємодії з Telegram Bot Platform

СМО – система масового обслуговування

ВСТУП

У сучасних умовах цифрової трансформації бізнесу, автоматизація клієнтського сервісу на базі інтелектуальних інформаційних технологій стає одним із ключових чинників конкурентоспроможності підприємств. Провідні світові компанії впроваджують системи віртуальних асистентів і чат-ботів, які інтегровані з корпоративними знаннями і здатні обробляти великі потоки звернень у режимі 24/7. Разом з тим, значна частина існуючих рішень залишається фрагментарною: вони погано узгоджуються з внутрішніми обліковими системами, не підтримують гнучке масштабування та демонструють обмежену якість відповідей при складних запитах клієнтів.

Світові тенденції розвитку інтелектуальних систем підтримки користувачів пов’язані з використанням великих мовних моделей, контекстуалізованого пошуку в корпоративних документах, гібридних підходах до обчислення, а також інтеграції чат-ботів із популярними месенджерами, зокрема Telegram. Це дає змогу поєднувати природну взаємодію з користувачем, швидкий доступ до даних та гнучке масштабування обчислювальних ресурсів. Однак на практиці залишається низка нерозв’язаних задач, пов’язаних із вибором архітектури системи, забезпеченням необхідного рівня продуктивності, надійності й захисту даних у корпоративному середовищі.

Актуальність роботи зумовлена потребою підприємств-постачальників будівельного обладнання в інтелектуальних програмно-апаратних комплексах, які забезпечують автоматизоване обслуговування клієнтів на основі сучасних методів обробки природної мови та інтеграції з наявними інформаційними ресурсами. Наукове завдання, що вирішується в роботі, полягає у розробленні та дослідженні гібридної архітектури інтелектуальної системи клієнтського сервісу з Telegram-ботом, великою мовною моделлю та спеціалізованим сховищем даних на основі MongoDB, за умов обмеженості обчислювальних і організаційних ресурсів.

Метою роботи є розроблення та наукове обґрунтування методів і інформаційної технології побудови комп’ютерної системи автоматизованої підтримки клієнтів на основі Telegram-бота з використанням великої мовної моделі та інтеграцією з документоорієнтованою базою даних.

Для досягнення поставленої мети необхідно вирішити такі основні завдання дослідження:

– узагальнити існуючі положення щодо застосування чат-ботів та інтелектуальних систем у клієнтському сервісі;

– виконати аналіз сучасного стану методів побудови аналогічних архітектур і засобів інтеграції з корпоративними базами даних;

– розробити модель системи масового обслуговування запитів клієнтів;

– розробити концептуальну та програмну архітектуру інтелектуальної системи з Telegram-інтерфейсом;

– дослідити залежності показників продуктивності та якості відповідей від параметрів системи;

– оптимізувати окремі параметри процесу обробки запитів для забезпечення заданих експлуатаційних характеристик.

Ідея роботи полягає у створенні клієнтського сервісу, що інтегрується у корпоративне середовище та поєднує сучасні методи обробки штучним інтелектом і корпоративні бази знань, основа такого підходу є принципи RAG. Отримані результати повинні підтверджувати сучасні наукові тенденції та сприяюти розвитку високопродуктивних клієнтських сервісів на базі штучного інтелекту.

1 СТАН ПИТАННЯ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1 Сучасні підходи до автоматизації клієнтського обслуговування

Автоматизація процесів клієнтського обслуговування є ключовим напрямом цифрової трансформації підприємств, що дозволяє забезпечити безперервність сервісу, знизити витрати на персонал і підвищити якість взаємодії з користувачами. Дослідження та розробки останніх років демонструють, що розвиток таких систем відбувається у напрямку інтеграції декількох технологічних рішень, серед яких провідне місце займають інтелектуальні діалогові агенти, корпоративні інформаційні платформи, інструменти аналітики та машинне навчання. За даними глобального опитування IBM 2023 року, автоматизація процесів клієнтського обслуговування на основі штучного інтелекту вже зайняла суттєву частку в корпоративних моделях цифрової взаємодії – 33% глобальних світових організацій вже використовують ці механізми. Окремий індикатор, безпосередньо пов’язаний із самообслуговуванням клієнтів, відображає, що 23% IT-підприємств уже впровадили системи автоматизованих відповідей і дій у режимі self-service [1].

Одним із провідних підходів є використання чат-ботів і віртуальних асистентів, які виконують роль першої лінії підтримки. Вони здатні автоматично обробляти стандартні запити, реєструвати звернення та забезпечувати базові консультаційні послуги. Інший напрям розвитку стосується інтеграції систем управління електронними сховищами з багатоканальними комунікаційними платформами. Це дозволяє централізовано накопичувати інформацію про клієнтів, взаємодії та корпоративні матеріали, застосовувати алгоритми персоналізації та підтримувати аналітичні моделі прогнозування поведінки. Окремі кейси трансформації підтверджують збільшення рівня задоволеності користувачів, разом із збільшенням використання цифрових каналів самообслуговування у 2-3 рази, скорочення кількості сервісних взаємодій на 40-50% та зменшення вартості обслуговування більш ніж на 20% [2].

Вагомим чинником у сучасних підходах останніх років є застосування штучного інтелекту, включаючи великі мовні моделі (LLM). Їх інтеграція у бізнес-процеси дозволяє формувати природну комунікацію, підтримувати довготривалий контекст діалогу і створювати адаптивні відповіді відповідно до запитів клієнтів. Аналітичні оцінки економічного потенціалу аналітичного та генеративного ШІ свідчать, що в поєднанні з іншими технологіями нинішнє покоління таких моделей може автоматизувати робочі активності, які сьогодні поглинають до 60-70% робочого часу співробітників у ряді задач [3].

Таким чином, аспекти автоматизації клієнтського обслуговування базуються на інтеграції віртуальних асистентів у вигляді діалогових систем, що включають використання чат-ботів, інформаційних платформ та баз знань, великих мовних моделей та аналітичних інструментів. Їх комплексне застосування забезпечує підприємствам не лише скорочення витрат, але й стратегічну перевагу за рахунок підвищення рівня сервісу, швидкості взаємодії та персоналізації пропозицій.

У контексті українського та міжнародного бізнес-середовища, аналогічні тенденції спостерігаються і у компаніях, що спеціалізуються на виробництві та дистрибуції інструментів. З огляду на значний обсяг клієнтських звернень та необхідність підтримувати стабільну якість сервісу, підприємства цього сектору зацікавлені у впровадженні інтелектуальних систем автоматизації. Використання чат-ботів і технологій штучного інтелекту розглядається як інструмент, здатний не лише зменшити навантаження на операторів, але й забезпечити цілодобову взаємодію з клієнтами, підвищуючи рівень задоволеності та лояльності споживачів.

1.2 Використання Telegram-ботів у бізнес-процесах

Досвід провідних світових лідерів у електронній комерції підтверджує, що месенджери та соціальні мережі є універсальною інфраструктурою для побудови бізнес-комунікацій. Telegram, завдяки відкритому програмному інтерфейсу, широкій користувацькій базі та підтримці різних форматів даних, перетворився

на платформу для розгортання автоматизованих сервісів, які інтегруються у корпоративні інформаційні системи.

Аналітичні огляди та практичні кейси демонструють, що використання Telegram-ботів дозволяє зменшити навантаження на операторів, прискорити реагування на стандартні запити клієнтів і забезпечити цілодобову доступність сервісів. Зокрема, розглядаючи омніканальні комунікації месенджеру, відзначається підвищення утримання користувачів та поліпшення клієнтського досвіду завдяки інтеграції Telegram-ботів у бізнес-процеси [4].

У зовнішніх комунікаціях Telegram-боти найчастіше застосовуються для клієнтської підтримки та інформаційного супроводу. Вони здатні автоматично відповідати на типові команди, надавати консультаційні послуги, реєструвати звернення чи обробляти замовлення. Водночас їх інтеграція з корпоративними системами дозволяє враховувати індивідуальні параметри клієнтів, що підвищує персоналізацію сервісу.

Окремим напрямом застосування є автоматизація маркетингових кампаній. За допомогою Telegram-ботів реалізуються сценарії розсилок, інтерактивних опитувань, зворотного зв’язку та аналітики ефективності кампаній у режимі реального часу. Це створює умови для більш гнучкого управління комунікаційними стратегіями та точнішого визначення потреб аудиторії.

У прикладних кейсах інтеграції Telegram-ботів з AI-CRM-системами для електронної комерції фіксується, зокрема, зростання продажів приблизно на 30% та поліпшення показників відновлення покинутих кошиків на 20% завдяки автоматизованим повідомленням і персоналізованим рекомендаціям, що демонструє ефективність такого підходу для підвищення конверсії та операційної результативності комунікацій з клієнтами [5].

Ключовою перевагою Telegram-ботів залишається відкритий API, що дає можливість створювати кастомізовані рішення та інтегрувати їх з корпоративними базами даних, платформами клієнтських взаємодій, платіжними системами й зовнішніми інтелектуальними модулями. Таке

поєднання формує основу для розвитку комплексних рішень, які виходять за межі простих чат-інтерфейсів і стають частиною ширших бізнес-процесів.

Таким чином Telegram-боти можна розглядати як універсальний інструмент цифрової трансформації бізнесу, що забезпечує поєднання простоти використання, високої інтеграційної гнучкості та можливості масштабування. Їхня роль у сучасних бізнес-процесах полягає не лише в автоматизації рутинних завдань, а й у створенні умов для подальшої інтеграції з технологіями штучного інтелекту, аналітики даних і корпоративних баз знань, що визначає перспективи розвитку комунікацій.

1.3 Технологічні основи інтелектуальних систем клієнтського сервісу

Великі мовні моделі та документоорієнтовані бази даних відносяться до ключових технологічних засобів побудови сучасних систем обслуговування клієнтів. Класифікація напрямів у даній сфері демонструє два вектори – автоматизація когнітивних процесів та організація керування даними у масштабованих середовищах з високою інтенсивністю транзакцій.

Великі мовні моделі (Large Language Models, LLM) становлять окремий клас систем штучного інтелекту, побудованих на архітектурі трансформерів і призначених для статистичного моделювання природної мови. Їх навчання здійснюється на масштабних корпусах текстів з метою формування параметризованого представлення мовних залежностей, що дозволяє моделі прогнозувати послідовності слів з урахуванням контексту. Основними характеристиками LLM є здатність до узагальнення, збереження контексту в межах великої кількості токенів – базова одиниця тексту, і гнучке донавчання на доменних та вузькоспеціалізованих даних. Завдяки цим властивостям вони забезпечують високий рівень точності при генерації текстів, автоматичному реферуванні, класифікації запитів і діалоговій взаємодії, що робить їх придатними для широкого спектра бізнес-застосувань.

У сфері е-комерції LLM розглядаються як когнітивний компонент корпоративних інформаційних систем, який забезпечує автоматизацію обробки

запитів, підтримку операцій з даними та інтерактивну взаємодію між користувачем і системою. Сучасні тенденції описують декілька напрямів їх використання: персоналізоване консультування клієнтів у сфері електронної комерції, динамічне управління базами знань, інтелектуальний пошук і семантичну аналітику великих обсягів текстової інформації. Застосування LLM у бізнес-процесах сприяє переходу від реактивного до адаптивного сервісу, коли система не лише відповідає на запити, а й прогнозує потреби користувача, використовуючи збережену історію взаємодії – контекст діалогу [6].

У свою чергу, напрям досліджень, пов’язаний із NoSQL-базами даних, відображає загальну еволюцію підходів до зберігання та обробки інформації в корпоративних середовищах. Традиційні реляційні системи управління базами даних (SQL) орієнтовані на суворо структуровані таблиці та жорсткі схеми даних, що обмежує їхню гнучкість при роботі з великими, змінними або неструктурованими обсягами інформації. На відміну від них, NoSQL-підходи базуються на принципі гнучкої схеми та горизонтального масштабування, що дозволяє ефективно обробляти великі потоки неоднорідних даних у режимі реального часу.

Документоорієнтовані бази даних, такі як MongoDB, займають центральне місце серед NoSQL-технологій завдяки універсальності моделі зберігання та відкритому коду. У таких БД дані подаються у вигляді різноорганізованих документів, що дає змогу поєднувати структуровані й напівструктуровані записи в одній колекції. Такий підхід спрощує роботу з даними різної природи – наприклад, об’єднання клієнтських профілів, транзакцій, звернень до служби підтримки та історії покупок в одному інформаційному просторі без необхідності складних зв’язків між таблицями.

Критичний аналіз впроваджень свідчить, що NoSQL-БД використовуються не лише як сховище даних, а й як аналітична платформа, здатна виконувати агрегаційні операції, пошук і попередню обробку даних. Її застосування у сфері електронної комерції, маркетингу та клієнтського сервісу забезпечує можливість

швидкого отримання інформації про поведінку користувачів, динаміку продажів і зміни попиту.

Разом з тим, відзначаються і певні обмеження NoSQL-підходів. Відсутність жорстких схем і транзакційних обмежень у деяких сценаріях може ускладнювати забезпечення цілісності даних, особливо у критичних фінансових або логістичних системах. Головним викликом залишається необхідність правильної архітектурної побудови – визначення меж колекцій, індексації та взаємодії з іншими сервісами у складі корпоративної інфраструктури [7].

Особливе місце займають підходи інтеграції LLM із системами управління базами даними та корпоративних знань. Метою таких підходів є формування когнітивного шару в бізнес-інфраструктурі, який дозволяє моделі не лише генерувати відповіді, але й виконувати операції аналітичного запиту, пошуку чи фільтрації даних. Поєднання генеративної моделі з базою даних створює нову архітектурну парадигму – Retrieval-Augmented Generation (RAG), у межах якої мовна модель звертається до зовнішніх джерел знань. Це забезпечує підвищення точності, релевантності та достовірності отриманих результатів, що особливо важливо у системах клієнтського обслуговування [8]. На рисунку 1.1 зображено типову RAG-архітектуру.

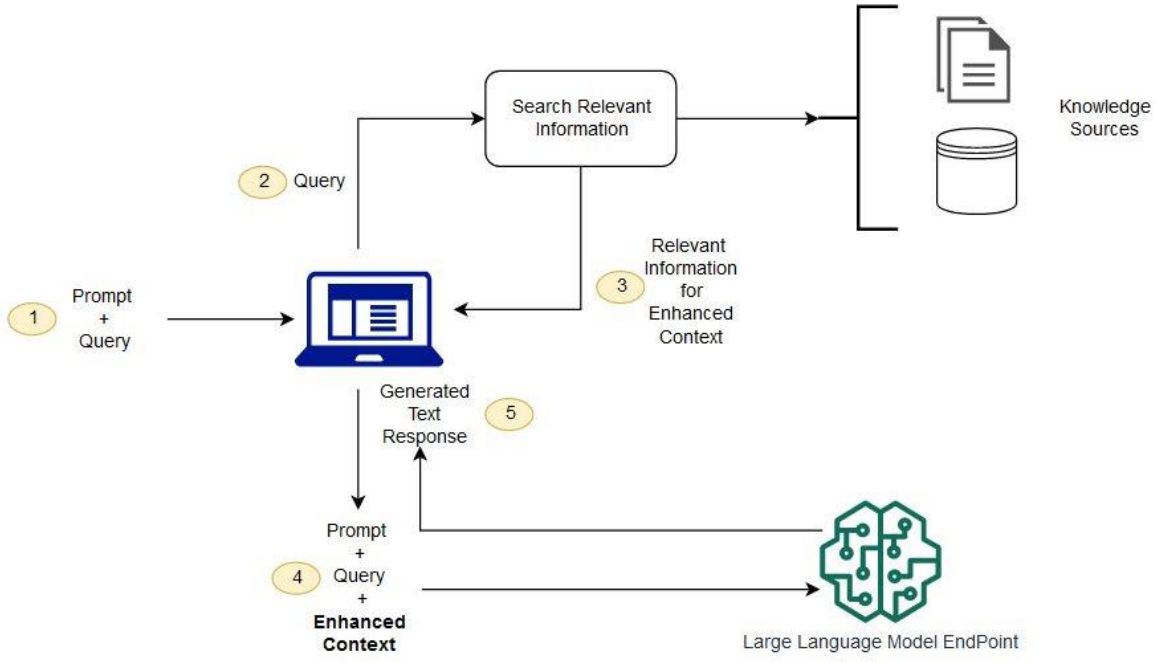


Рисунок 1.1 – Принцип роботи RAG-архітектури [9]

Для комерційних підприємств, інтеграція RAG-архітектури LLM та баз даних із чат-ботами як інтерфейсом взаємодії, визначає широку можливість скорочення витрат на підтримку клієнтів, підвищення масштабованості операцій та забезпечення персоналізованого підходу до користувача. Використання такого поєднання у вигляді інтелектуальної системи підтримки клієнтів, що інтегрується з каталогами та електронними сховища підприємства, обробляє інформацію про замовлення, надає консультації у реальному часі та зберігає історію взаємодій – це підвищення якості обслуговування, скорочення часу реагування на запити та формування персоналізованого клієнтського досвіду.

1.4 Аналіз існуючих корпоративних систем інтеграції та їхні обмеження

Інтеграція систем обслуговування клієнтів у корпоративні інформаційні середовища перейшла від експериментальних реалізацій до компоненту стратегічної цифрової трансформації організацій. У прикладних впровадженнях бот-асистент виступає як елемент інформаційної інфраструктури, що взаємодіє з сховищами даних, підсистемами CRM, HR-платформами та аналітичними інструментами, забезпечуючи автоматизацію рутинних процесів, прискорення обробки звернень і збір операційної телеметрії для подальшого аналізу. Підтвердженням цього є успішні комерційні та промислові кейси, які відзначають перехід від ізольованих чат-рішень до інтегрованих архітектур із підтримкою аналітики і машинного навчання.

Платформи прикладного рівня демонструють різні підходи до архітектури та інтеграції. Комерційне рішення Haptik реалізує повноцінні конвеєри обробки звернень, що поєднують модулі попередньої класифікації, маршрутизації запитів, контекстного менеджеру діалогів та аналітичних панелей. Архітектура таких систем передбачає інтерфейси для взаємодії з CRM, механізми персоналізації на основі історії взаємодій і інструменти оптимізації сценаріїв. Практичні кейси Haptik для міжнародних брендів свідчать про здатність розроблених рішень платформи обробляти великі обсяги звернень при

одночасному застосуванні персоналізації та бізнес-логіки, що робить її прикладом успішного гібридного індустріального рішення для зовнішньої клієнтської підтримки.

У сегменті внутрішніх корпоративних сервісів Leena AI демонструє архітектурну спеціалізацію на автоматизації HR-функцій і служби внутрішньої підтримки. Рішення інтегрує модулі обробки запитів співробітників, пошукові індекси корпоративних документів, механізми управління запитами щодо кадрових процесів і аналітичні інструменти для оцінки навантаження підтримки. Комерційні кейси Leena AI показують ефективність підходу у зниженні обсягу «тривіальних» запитів до HR-відділів і підвищенні показників самообслуговування. Архітектурно це досягається шляхом тісної інтеграції з кадровими системами (HRIS) та внутрішніми сховищами документів, а також використанням ШІ-модулів для класифікації запитів.

Telegram-орієнтовані рішення виступають як окрема категорія завдяки відкритому API та масовій поширеності платформи серед кінцевих користувачів. Приклади успішних брендових інтеграцій підтверджують практичну ефективність підходу та охоплюють широкий сектор. Бот компанії «Нова пошта» реалізує функції відстеження відправлень, інформування про статуси і взаємодію з платіжними шлюзами. Рішення авіакомпаній KLM інтегруються із системами бронювання для надання посадкових талонів, нагадувань і консультацій безпосередньо в месенджері. Ці кейси ілюструють можливість одночасного вирішення як операційних завдань клієнтської підтримки, так і зниження навантаження на контакт-центри шляхом делегування типових операцій ботам.

Досвід інтеграції показує, що поєднання ботів із системами класифікації звернень і зовнішніми аналітичними платформами дозволяє досягати високих показників автоматизації рутинних операцій; прикладом є рішення Enterprise Bot, яке демонструє можливість автоматизувати значну частину повторюваних операцій у транспортній та фінансовій галузях шляхом інтеграції обробки звернень із CRM та аналітикою. Практичні імплементації таких систем

підкреслюють важливість проектування наскрізної обробки даних, від реєстрації звернення до оновлення релевантних записів у корпоративних базах.

Поряд з успішними прикладами, аналіз практик і наукових джерел вказує на сукупність технічних, організаційних і концептуальних обмежень, що ускладнюють універсальне застосування сучасних платформ у корпоративному середовищі. До архітектурних викликів належить недостатня здатність ряду реалізацій підтримувати стійке тривале збереження історії діалогу, що істотно ускладнює обробку складних і багатоступеневих звернень. Аналіз сучасних праць в розмовних чат-системах свідчить про потребу в інтеграції гібридних архітектур, що поєднують детерміновані компоненти і генеративні модулі, для забезпечення одночасної передбачуваності та семантичної гнучкості.

Сумісність із гетерогенними корпоративними системами є наступним системним обмеженням. Корпоративні ландшафти часто містять змішані за структурою сховища даних, застарілі внутрішні системи та зовнішні API, а також строго регламентовану та сегментовану інформаційну інфраструктуру, що вимагає проміжних шарів інтеграції або розробки адаптерів. Відсутність уніфікованих інтерфейсів призводить до збільшення витрат на розробку та підтримку, ускладнює синхронізацію даних і знижує стабільність бізнес-процесів при розгортанні ботів у масштабі підприємства.

Проблематика інформаційної безпеки та конфіденційності набуває критичного значення при інтеграції чат-ботів у внутрішні корпоративні мережі. Багато комерційних рішень використовують хмарні сервіси, що обмежує прямий контроль організації над потоками даних та процесами обробки. Це створює ризики невідповідності вимогам стандартів інформаційної безпеки та нормативів щодо захисту персональних даних. Необхідні архітектурні рішення із розмежуванням зон обробки даних, криптографічними механізмами захисту та аудит-логами для критичних операцій.

Рівень персоналізації та адаптивності систем також виявляє істотні обмеження. У типовому комерційному впровадженні персоналізація обмежена використанням простих правил або неглибоких моделей, що не беруть до уваги

поведінкові особливості користувача. Наявність таких обмежень призводить до обмеженої релевантності відповідей і недостатнього задоволення потреб складніших бізнес-сценаріїв. Додатково, розгортання власних LLM-рішень та їх адаптація під вузькоспеціалізовані домени пов’язана з високими обчислювальними витратами і потребою у фахових кадрах, що робить повну локалізацію LLM недоцільною для багатьох середніх та малих бізнесів.

Наведені приклади промислових рішень і відповідні дослідження демонструють, що успішна інтеграція системи клієнтської взаємодії, у вигляді чат-бота, у корпоративну мережу вимагає комплексного підходу, що поєднує архітектурну модульність, надійні засоби інтеграції. Для більшості компаній існуючі готові платформи можуть забезпечити швидке впровадження базової функціональності, проте для досягнення вимог щодо контролю даних, глибокої інтеграції з внутрішніми базами даних і забезпечення високої якості обслуговування за пікових навантажень доцільно розробляти архітектуру з елементами локального контролю, інтеграцією готових компонентів для обробки природної мови та проміжними шарами для забезпечення сумісності.

1.5 Постановка завдання роботи

Метою проєкту є дослідження застосування інтелектуальних технологій для автоматизованого обслуговування клієнтів компанії-постачальника будівельного обладнання та розробка на цій основі концепції та моделі системи у вигляді Telegram-бота з інтеграцією LLM і бази даних MongoDB, подальшим проєктуванням програмної реалізації та експериментальною перевіркою її працездатності й ефективності.

Для досягнення цієї мети необхідно вирішити такі завдання:

– проаналізувати предметну область та сучасні підходи до автоматизації клієнтського обслуговування, включно з використанням баз даних, LLM і RAG у корпоративних системах;

– дослідити теоретичні основи побудови системи масового обслуговування, механізмів оркестрації діалогових систем та інтеграції мовних моделей із зовнішніми сховищами знань;

– сформулювати вимоги до функціональних, інформаційних та експлуатаційних характеристик системи підтримки клієнтів;

– обґрунтувати вибір загальної архітектури системи, що включає Telegram-бота як клієнтського інтерфейсу, великої мовної моделі та документно-орієнтованої бази даних MongoDB як кандидата для зберігання корпоративних даних;

– розробити функціональну структуру системи та загальну модель обробки клієнтських запитів;

– спроєктувати програмну реалізацію Telegram-бота на основі Python з використанням Telegram API та інтеграцією з обраними зовнішніми сервісами;

– розробити методику експериментальних досліджень і провести тестування системи щодо швидкодії, якості відповідей і безпеки обробки даних;

– узагальнити результати досліджень і оформити їх у вигляді дипломного проєкту.

Впровадження інтелектуальної системи дозволить:

– знизити навантаження на операторів контакт-центру та забезпечити цілодобову підтримку клієнтів;

– підвищити швидкість обробки запитів і рівень задоволеності користувачів;

– забезпечити персоналізовані відповіді на запити клієнтів завдяки інтеграції LLM та корпоративної бази даних;

– створити ефективний цифровий канал взаємодії, що відповідає сучасним стандартам корпоративних систем.

2 ТЕОРЕТИЧНИЙ РОЗДІЛ

2.1 Загальна характеристика об’єкта впровадження системи автоматизованого обслуговування клієнтів компанії-постачальника будівельного обладнання

Об’єктом дослідження є організація програмно-апаратного комплексу автоматизованого обслуговування клієнтів компанії-постачальника будівельного обладнання. Розроблювана система повинна поєднати інтерактивний користувацький інтерфейс, інтелектуальне ядро з обробки природної мови, сховища корпоративних даних і сервіси аналітики.

Організаційна структура компанії-постачальника будівельного обладнання відзначається ієрархічною системою управління. До складу компанії входять п’ять основних підрозділів: виконавчий, фінансовий, комерційний, IT-відділ та служба підтримки клієнтів. Структуру взаємозв’язків між цими підрозділами наведено на рисунку 2.1, який демонструє ієрархічну організацію бізнес-процесів компанії.

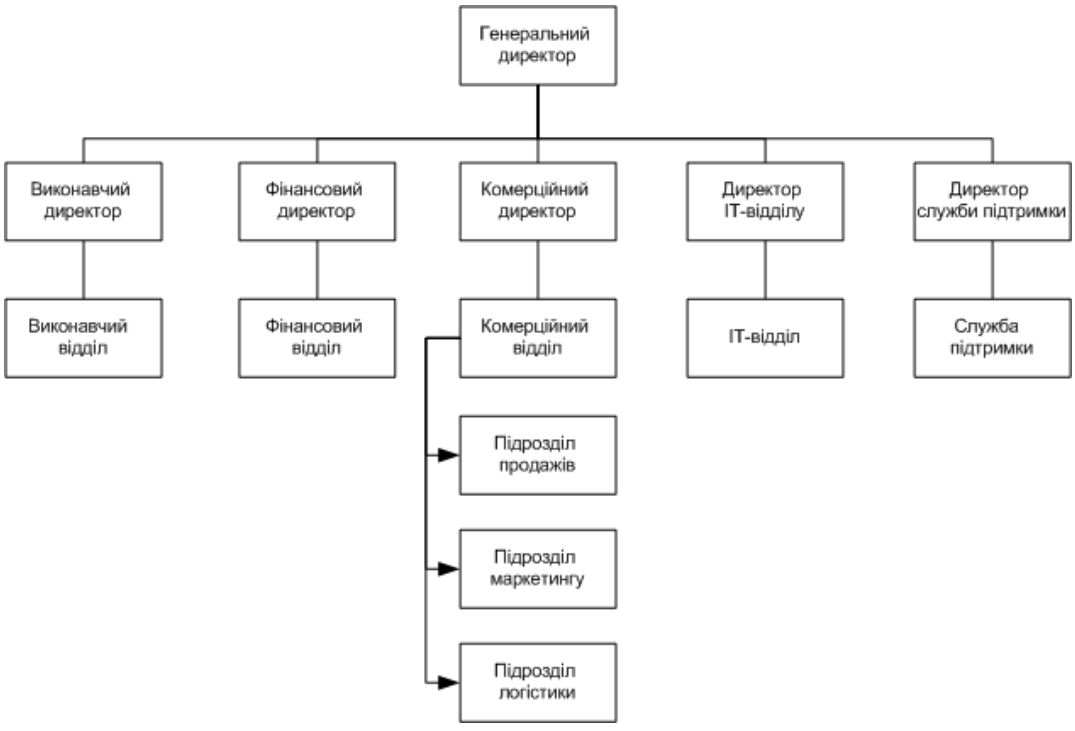


Рисунок 2.1 – Схема організаційної структури компанії

Підприємство є масштабною організацією з центральним офісом, виділеним офісом клієнтського обслуговування, а також веб-сервіси електронної комерції; усі ці елементи утворюють розподілену інформаційну екосистему з різними вимогами до доступності даних, конфіденційності й швидкості реакції.

IT-інфраструктура підприємства працює на основі централізованої комп'ютерної системи за ієрархічною структурою Cisco, що об'єднує локальні й хмарні ресурси в єдину цифрову структуру та підтримує обмін інформацією між елементами організації. Вона включає корпоративну комп'ютерну мережу, що побудована за гібридною схемою – сегменти функціонують у межах підрозділів офісів, тоді як віддалені взаємодії забезпечується через зовнішні канали. Взаємодія між ними реалізується через централізовані контролери доступу, маршрутизатори та комутатори, а також корпоративні шлюзи. У межах мережі функціонують сервери баз даних, ERP-система управління товарообігом, CRM для взаємодії з клієнтами, маркетингові сервіси тощо. Усе це формує основу, на якій має функціонувати нова система автоматизації клієнтського обслуговування. На рисунку 2.2 представлена логічна топологія мережі.

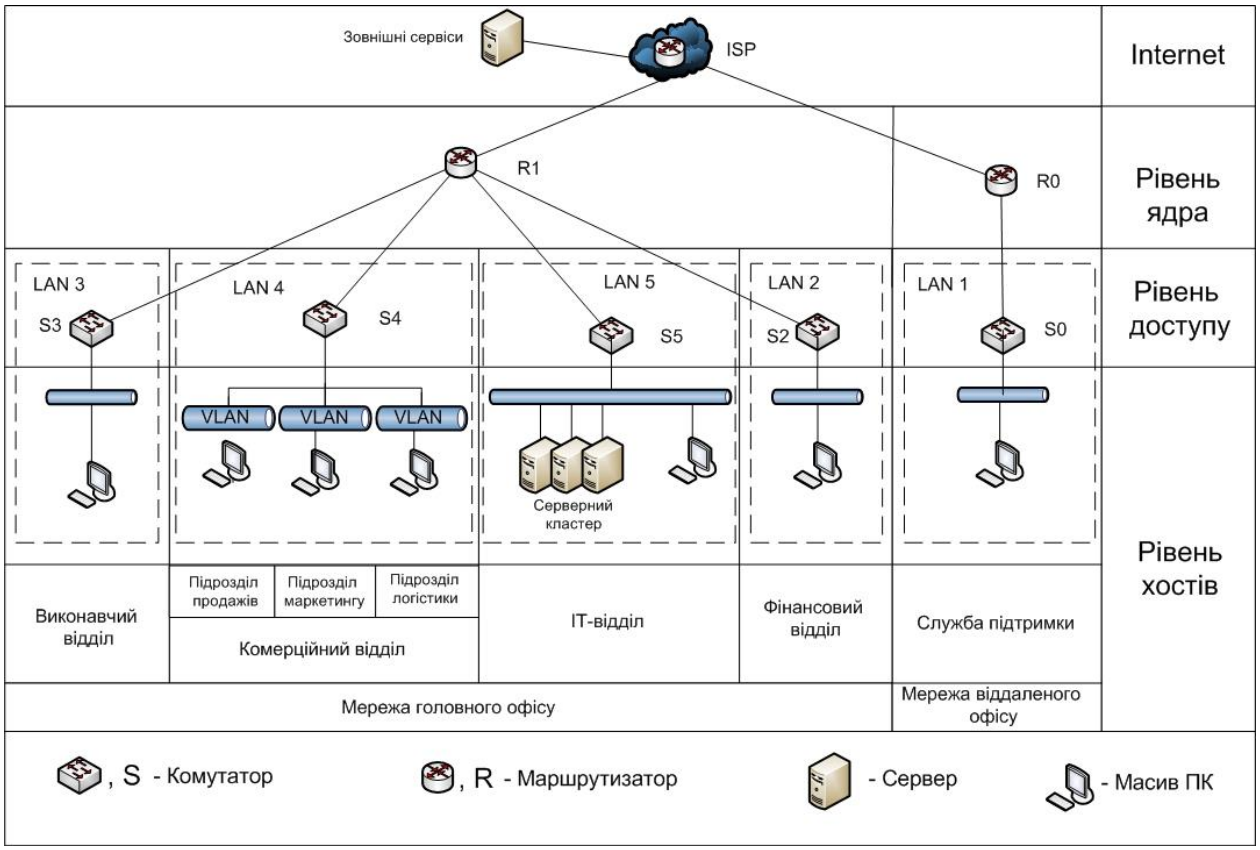


Рисунок 2.2 – Логічна топологія корпоративної мережі підприємства

Завданням є на базі розглянутої ІТ-інфраструктури реалізувати гібридний програмно-апаратний комплекс автоматизованого клієнтського обслуговування. Комплекс включатиме серверну частину, інтегровану у внутрішню корпоративну мережу, та хмарний компонент. Критично важливі компоненти локалізуються у внутрішній корпоративній мережі, що забезпечує дотримання політик безпеки та SLA (Service-Level Agreement). Обчислювальні модулі, які потребують масштабованості – розміщуються у хмарній інфраструктурі. Такий підхід дозволяє поєднати контроль над даними з гнучкістю масштабування, що є ключовим у пікові періоди клієнтських звернень.

У контексті бізнес-цілей компанії, головними вимогами до системи є висока достовірність відповідей, мінімізація часу очікування клієнта, підтримка транзакційних запитів (перевірка замовлення, наявність товару, обробка сервісних заявок), інтеграція з базами знань та забезпечення якісного користувацького досвіду (UX).

2.2 Структура програмно-апаратного комплексу об’єкта

Структура програмно-апаратного комплексу автоматизованого клієнтського обслуговування компанії-постачальника будівельного обладнання, у загальному вигляді, розглядається як багаторівнева сукупність взаємопов’язаних компонентів. Взаємодія між цими рівнями відбувається через уніфіковані API-шлюзи та внутрішні протоколи обміну даними, що забезпечує синхронізацію та узгодженість усіх компонентів.

Перший рівень утворює інтерфейс взаємодії з користувачем – зокрема, Telegram-бот, який є основною точкою доступу до системи. Він приймає запити користувачів, передає їх до ядра обробки, а потім повертає згенеровані відповіді.

Другий рівень формує модуль обробки, в якому використовується як логічний компонент генеративна мовна модель. Цей модуль здійснює маршрутизацію та аналіз запиту, визначає його семантичну структуру та формує відповідь. Проте компонент генеративної моделі сам по собі не завжди здатний забезпечити точність і актуальність інформації, тому пропонується ввести

сполучення між рівнями системи – модуль Retrieval-Augmented Generation (RAG), що виконує роль посередника між моделлю другого рівня і внутрішніми джерелами знань компанії, що розташовані на третьому рівні. Саме цей модуль відповідає за пошук відповідних даних у базі та їх підключення до контексту відповіді.

На третьому рівні розташовується база даних, що зберігає структуровану та неструктуровану інформацію про клієнтів, товари, історію замовлень, попередні звернення тощо. Для реалізації цього рівня слід розглянути документо-орієнтовану базу даних MongoDB із можливим гібридним підходом побудови сховища. Взаємодія з базою здійснюється за допомогою відповідних драйверів і API, інтегрованих у програмне середовище системи.

2.3. Аналіз предметної області та обґрунтування використаних моделей та компонентів системи

2.3.1 Обґрунтування вибору великої мовної моделі за метриками та параметрами

Для програмно-апаратного комплексу, що реалізує інтелектуальну підтримку клієнтів через Telegram-бота, генеративна модель виконує роль «ядра» всієї системи: саме вона формує відповіді, веде діалог, узагальнює інформацію з бази знань та вписується в обмеження черг обслуговування зі сторони провайдера послуг та його політик – латентність, пропускна здатність, вартість одного запиту тощо. Тому вибір конкретної LLM має спиратися не лише на загальні міркування «точності», а й на результати апробації в бізнес-сценаріях, а також на поведінку моделей в RAG-архітектурах.

Сучасні огляди застосування LLM у бізнесі показують, що автоматизоване обслуговування клієнтів, чат-боти й віртуальні асистенти є однією з базових сфер їх упровадження. Для великих компаній саме канал підтримки виділяють як приклад, де LLM одночасно зменшує навантаження на операторів, підвищує якість сервісу, дає можливість працювати 24/7 і забезпечує глибоку

персоналізацію. Економічна доцільність використання LLM у бізнесі при цьому визначається балансом між якістю відповідей, швидкодією й вартістю [10].

З погляду кількісних показників доцільно порівнювати останні актуальні моделі класу GPT-5, Gemini 2.5 та LLaMA 4 за типовими бенчмарками узагальнених знань, математичного міркування, програмування і мультимодального розуміння, а також за характеристиками контексту, вартості та надійності.

На наборі MMLU (Measuring Massive Multitask Language Understanding) модель GPT-5 у режимі поглибленого міркування перевищує 88,7 % точності, на GSM8K (Grade School Math 8K) демонструє понад 95 %, на іспиті AIME (American Invitational Mathematics Exam) 2025 досягає 94,6 %, на GPQA (Graduate-Level Google-Proof Q&A) – близько 87,3 %, на MMMU (Massive Multi-discipline Multimodal Understanding) – приблизно 84,2 %. На SWE-bench Verified для задач промислового програмування GPT-5 у «thinking»-конфігурації досягає 74,9 %. Модель підтримує контекстне вікно – максимальна кількість текстових даних, яку модель може обробити за один раз – до ≈272 тис. токенів і знижує частоту галюцинацій (оманлива інформація) до 4,5% у поєднанні з доступом до зовнішніх джерел [11, 12].

Gemini 2.5 Pro на тих самих класах задач забезпечує порівнюваний рівень якості. На MMLU фіксується близько 88,6% точності, на AIME 2025 – ≈83%, на GPQA – ≈83%. У кодових бенчмарках LiveCodeBench v5 Gemini 2.5 Pro досягає 75,6% pass@1, на SWE-bench Verified – 63,2%. Для складних математичних і логічних задач модель підтримує результати рівня 80-83 % на відповідних наборах із низькими показниками галюцинацій, забезпечуючи при цьому контекстне вікно порядку 10^6 токенів [13, 14].

LLaMA 4 у тестах демонструє дещо нижчі, але стабільні показники. Для варіантів Maverick/Behemoth надаються значення MMLU на рівні ≈82-83 %, на спеціалізованому наборі MATH-500 – ≈95 %, на GPQA Diamond – близько 73-74%. Для цієї лінійки заявляються розширені контекстні вікна (до 10^6-10^7 токенів

у старших конфігураціях) та відкриті ваги з обмеженнями ліцензування (можливість донавчання) [15, 16].

З урахуванням даних відкритих бенчмарків 2024-2025 рр. подано порівняльну таблицю 2.1 для типових представників трьох сімейств.

Таблиця 2.1 – Порівняльна таблиця для трьох великих мовних моделей

Сімейство / модель	MMLU, %	AIME 2025 / MATH, %	GPQA / MMMU, %	SWE-bench Verified, %	Контекстне вікно	Орієнтовні витрати на 1 млн. токенів (Вхід / Вихід)
GPT-5	>88,7	94,6 / >76,6	≈87,3 / ≈84,2	74,9	≈272 тис. токенів	$1.25 / $10.00
Gemini 2.5	≈88,6	83,0 / високі	≈83 / високі	63,2	≈1 млн токенів	Pro: $1.25/ $10.00 Flash: $0.30/ $2.50
LLaMA 4 (Maverick/ Behem.)	≈82-83	95,0 (MATH-500)	≈ 73-74	конкурента, нижча за комерційні	до 10^6-10^7 токенів	відкрите розгортання, вартість визначається власною інфраструктурою

Ще один важливий аспект – поєднання моделі з архітектурою Retrieval-Augmented Generation. У тестуваннях, де порівнюються старіші версії GPT, Gemini-Pro та Llama у RAG-сценарії з використанням фреймворку RAGAS, зафіксовано, що саме GPT демонструє найвищу якість відповідей, тоді як Gemini-Pro та Llama показують нижчі, але близькі значення метрик [17].

З огляду на вимоги проєкту – інтеграція з Telegram-ботом, робота в режимі масового обслуговування із змінною інтенсивністю черг, використання RAG над корпоративною базою знань та обмеження за вартістю – доцільно обрати як основне генеративне ядро один із представників родини Gemini 2.5. Така конфігурація забезпечує достатню якість міркування, великий контекст для

роботи з технічною документацією та прайс-листами, гнучке керування (через вибір між Pro і Flash-варіантами) та прийнятну собівартість запиту.

2.3.2 Обґрунтування вибору СУБД та архітектури комбінованого сховища

Для підсистеми підтримки клієнтів компанії-постачальника будівельного обладнання характерні напівструктуровані дані: картки товарів з різними атрибутами, історії звернень, гарантійні умови, результати діалогів із ботом, службові метадані. Такі об’єкти природно зберігати у вигляді вкладених різнорідних структур, а не жорстко нормалізованих таблиць. У сучасних оглядах SQL-NoSQL-технологій підкреслюється, що NoSQL-СУБД оптимізовані саме під напівструктуровані записи й часто дають простіший дизайн схеми та кращу масштабованість [18].

У низці порівняльних робіт продуктивність СУБД оцінюється за стандартними метриками: пропускна здатність (operations per second) та латентність операцій (середня, 95-й/99-й перцентиль). Для NoSQL-сховищ це зазвичай робиться за допомогою YCSB-бенчмарку на змішаних навантаженнях. Так порівняння MongoDB, Cassandra та Redis показало, що при навантаженнях типу Workload A-F MongoDB демонструє вищу пропускну здатність і меншу або співставну затримку читання й оновлення [19, 20]. Аналогічні висновки щодо MongoDB як сховища з хорошим балансом між продуктивністю та гнучкістю підтверджуються в огляді документних NoSQL СУБД, де MongoDB показує найкращий час виконання для більшості YCSB-навантажень [21].

Для реляційної альтернативи – PostgreSQL – характерна повна ACID-транзакційність і класична таблична модель, однак з версії 9.4 доступний тип JSONB, який дозволяє зберігати напівструктуровані дані, індексувати поля всередині JSON. У технічних оглядах підкреслюється, що JSONB є «компромісом» між жорстко структурованою схемою й повністю схематично-вільним документним підходом [22, 23].

Apache Cassandra, своєю чергою, реалізує розподілену колонкову модель з акцентом на горизонтальне масштабування і високу доступність. СУБД описується як лінійно масштабовану AP-систему (CAP), оптимізовану для важких та розподілених навантажень. Схема проєктується від запитів, а не від сутностей предметної області, що робить Cassandra дуже ефективною за заздалегідь відомих патернах доступу, але менш зручною для складних запитів на «ходу» і часто змінних структур [24, 25].

Узагальнення офіційної документації та незалежних порівняльних досліджень представлені у порівняльній таблиці 2.2, яка фокусується на тих властивостях, що суттєві для розроблюваної системи.

Таблиця 2.2 – Порівняльна таблиця характеристик популярних СУБД

Критерій	MongoDB	PostgreSQL	Apache Cassandra
Клас СУБД	Документно-орієнтована NoSQL	Реляційна СУБД	NoSQL wide-column (partitioned row store)
Модель даних	BSON-документи з вкладеними структурами	Таблиці з рядками/стовпцями; JSON/JSONB як додатковий тип	Партиції з широкими рядками, стовпці задаються на рядок
Гнучкість схеми	Висока, схематично-вільна з опційною валідацією схем	Середня/висока: фіксована схема + гнучкі JSONB-поля	Висока для рядкових стовпців, але схема прив’язана до запитів
Транзакційність	Повна ACID-підтримка з версії 4.0	Повна ACID-семантика	Повноцінні багаторядкові транзакції обмежені
Масштабування	Горизонтальне (sharding, replica set) на рівні колекцій	Переважно вертикальне; горизонтальність через розподілені розширення	Лінійне горизонтальне масштабування кластерів, мульти-DC реплікація
Підтримка складних запитів	Агрегаційний framework, pipeline-оператори; joins обмежені ($lookup)	Повноцінний SQL, складні JOIN/CTE, віконні функції	Запити в межах однієї партиції; JOIN відсутні, складна аналітика вимагає зовнішніх інструментів

Кінець таблиці 2.2

Пропускна здатність на змішаних навантаженнях	Висока; зазвичай має найбільшу пропускну здатність на змішаних сценаріях	Як правило, перевершує MongoDB (при подібних SQL-запитах), але гірше масштабується горизонтально при зростанні обсягу напівструктурованих даних [26, 27, 28]	Висока на важких запитах, стійка масштабованість при збільшенні операцій; latency читання часто вища, ніж у документних / ключ-значення сховищ
Затримка	Низька затримка читання й оновлення на більшості YCSB-навантажень (окрім окремих сценаріїв)	Низька затримка на OLTP-запитах; JSONB-операції можуть бути важчими без ретельної індексації	Оптимізована при великих обсягах і розподілених записах, але складні read/scan-операції можуть бути повільнішими [29]
Спеціалізація RAG / векторного пошуку	Atlas Vector Search – керована хмарна функція з підтримкою HNSW	Векторний пошук реалізується через розширення типу pgvector, інтегровані у загальну екосистему PostgreSQL	Використовується SAI-індекси, функції схожості для AI-сценаріїв

Згідно таблиці, PostgreSQL є цілком життєздатною альтернативою. Однак у випадку розглянутої системи, документаційна модель MongoDB дозволяє безболісно еволюціонувати схему (додавати/змінювати поля), не проводячи дорогих міграцій таблиць. Cassandra, навпаки, найкраще показує себе в системах, де запити стабільні й спроєктовані заздалегідь, тому для моделі динамічних клієнтських звернень з різними фільтрами й історією діалогів це означало б складне кодування патернів, що погано узгоджується з вимогами до гнучкості.

Окреме питання – інтеграція з RAG-підсистемою. З одного боку, MongoDB Atlas Vector Search – хмарні обчислення на кластерах – вже надає векторні індекси поверх документів з підтримкою HNSW, квантування й оптимізації затримки та пам’яті. З іншого боку, офіційні матеріали та практичні кейси

наголошують, що ці можливості доступні лише в керованих Atlas-кластерах певного тиру (не нижче M10) і можуть вимагати виділених Search Nodes, що суттєво впливає на вартість при масштабі великого сховища.

У галузевих оглядах сховищ для AI-архітектур відзначається, що спеціалізовані векторні бази даних забезпечують вищу ефективність операцій векторного пошуку порівняно з гібридними СУБД, які поєднують транзакційні та векторні дані в одному сховищі. На цій підставі для ресурсообмежених RAG-навантажень у системі обґрунтовується доцільність рознесення транзакційних та векторних операцій [30].

ChromaDB є відкритою векторною СУБД, орієнтованою на локальні/контейнеризовані RAG-сценарії: описується як легковагове сховище ембеддингів із підтримкою ANN-пошуку, простим API та можливістю інтеграції з LLM-фреймворками без жорсткої прив’язки до конкретного хмарного провайдера [31].

Тому для системи запропоновано двошарову архітектуру розподіленої бази знань:

– первинний шар – MongoDB як канонічне корпоративне сховище: каталоги товарів, гарантії, історії звернень, телеметрія роботи, журнали сесій;

– другий шар – ChromaDB, що містить лише векторні подання (ембеддинги) фрагментів документів та діалогів, пов’язані з MongoDB через унікальні ідентифікатори.

Для уникнення неоднозначності у подальшому викладі фіксується пріоритетність ролей компонентів: MongoDB розглядається як основна база даних системи та єдине джерело корпоративних відомостей. ChromaDB використовується лише як допоміжний векторний індекс для семантичного пошуку; первинні бізнес-дані та всі операції зміни виконуються в MongoDB, після чого індекс синхронізується. За такого підходу векторний шар є замінним і за потреби може бути перенесений у кероване середовище MongoDB Atlas із використанням Atlas Vector Search без зміни доменної моделі та бізнес-логіки.

2.3.3 Принципи Retrieval-Augmented Generation та обґрунтування застосування на об’єкті впровадження

Retrieval-Augmented Generation (RAG) сформувалася як одна з базових парадигм побудови інтелектуальних систем, що працюють з великомасштабними корпоративними знаннями. Поєднання механізмів пошуку з генеративними мовними моделями розглядається як спосіб пом’якшити ключові обмеження «чистих» LLM – статичність навчальних даних, схильність до галюцинацій, відсутність доступу до внутрішніх джерел знань і високу вартість постійного перенавчання. У оглядах RAG підкреслюється, що введення етапу ретривалу – пошук і витяг релевантної інформації із зовнішнього сховища перед генерацією відповіді – дає змогу підвищити якість і надійність результатів, оскільки модель отримує доступ до актуального контексту [32, 33].

У загальному вигляді RAG-конвеєр описується як послідовність таких етапів: формування запиту на основі повідомлення, перетворення цього запиту у представлення, придатне для пошуку, вилучення релевантних фрагментів з бази знань та генерація відповіді з урахуванням отриманого контексту. RAG розглядається як форма «зовнішньої пам’яті», що доповнює LLM актуальними фактами й доменною інформацією, яка зберігається в документних або гібридних сховищах [34]. На рисунку 2.3 зображено типовий RAG-конвеєр.

Рисунок 2.3 – Типовий приклад RAG-конвеєр у протиставлені «наївної» відповіді LLM [34]

Ключову роль у класичних реалізаціях RAG-систем відіграє векторний пошук. Векторно-орієнтовані індекси дозволяють будувати семантичний простір, у якому документи й запити подаються у вигляді ембеддингів (векторні вкладення), а пошук релевантного контексту зводиться до задачі пошуку найближчих сусідів – за методами kNN або ANN [35].

Разом із тим векторний пошук не покриває всіх сценаріїв взаємодії з даними. У бізнес-системах значна частина запитів є структурованими: отримати перелік замовлень за період, знайти всі звернення за певним типом гарантії, підрахувати кількість відкритих заявок тощо. Для таких випадків застосовується підхід text-to-SQL, який безпосередньо перетворює природномовний запит на формальний запит до структурованої БД. У роботах з розробки text-to-SQL систем описується, що перетворення запиту на SQL створює явний формальний шар доступу до даних, оскільки сформований запит можна переглянути, перевірити й виконати в наявній СУБД, а сам підхід орієнтований на інтеграцію природномовних звернень у існуючу інфраструктуру баз даних [36, 37].

Комбінування цих двох підходів приводить до гібридних архітектур роботи з даними. У роботах, що уніфікують text-to-SQL і векторний пошук, промпт розглядається як вхід до задачі text-to-VectorSQL: він трансформується у запит, який одночасно містить SQL-частину для роботи зі структурованими полями та векторну частину для пошуку за подібністю в просторі ембеддингів. Такі фреймворки демонструють, що в одному запиті можна поєднати семантичний пошук по неструктурованому контенту з класичними SQL-операціями над структурованими даними, забезпечуючи єдиний інтерфейс до обох типів джерел [38].

В окремих оглядах інтеграції RAG з SQL-сховищами підкреслюється, що поєднання векторного пошуку із запитами до реляційних даних дозволяє будувати єдину точку входу до корпоративних даних, де LLM або інші інтерфейси природної мови можуть працювати поверх як неструктурованих, так і структурованих джерел [39]. У практичних реалізаціях RAG-систем підкреслюється, що ефективність гібридних рішень визначається не лише

поєднанням векторного пошуку та SQL-запитів, а й чітко організованими етапами індексації й маршрутизації: оптимізацією розбиття документів на смислові фрагменти, багатопредставленим та ієрархічним індексуванням, використанням спеціалізованих ембеддингів, а також логічним чи семантичним роутером, який вирішує, чи слід спрямувати запит до векторного сховища, реляційної або іншого компоненту бази знань [40]. На рисунку 2.4 представлено розширену гібридну RAG-модель.

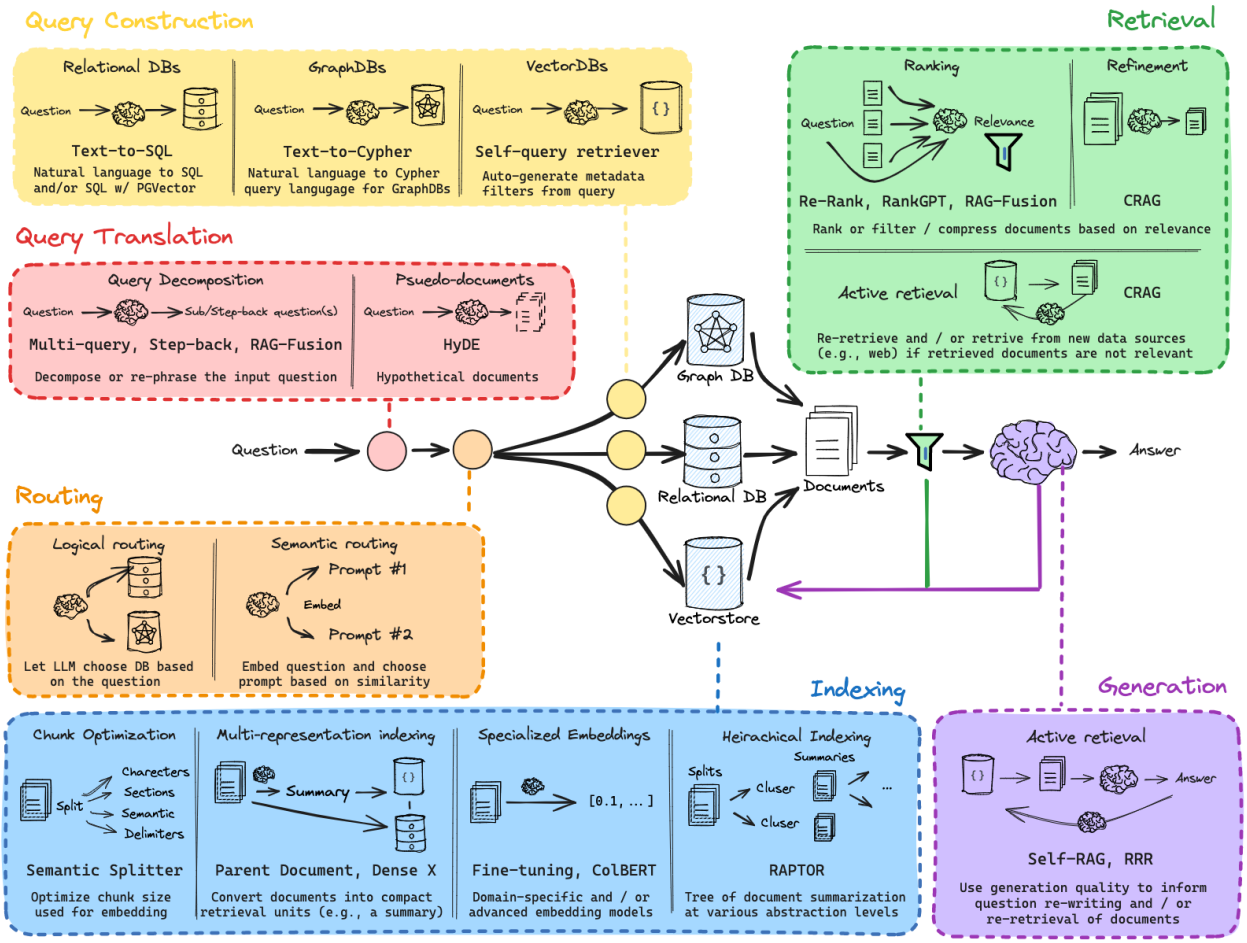


Рисунок 2.4 – Гібридна модель RAG-системи [40]

У межах проєкту системи автоматизованого обслуговування клієнтів компанії-постачальника будівельного обладнання необхідно реалізувати саме гібридний підхід: класичний RAG на основі векторного пошуку по базі знань (векторна індексація MongoDB) доповнюється гілкою prompt-RAG типу «text-to-NoSQL», де LLM генерує запити до структурованих даних. У RAG-ланцюжку маршрутизатор розв’язуватиме запит через комбінацію задач пошуку: модель

формує запит до структурованого шару даних, а паралельно виконується семантичний пошук по векторному індексу бази знань. Якщо структурована гілка не дає релевантного результату, роутер переводить запит у режим «чистого» векторного пошуку, концентруючись на документах, інструкціях і попередніх діалогах.

У роботах, де RAG застосовується для генерації та оцінки знань, запропоновано трикомпонентну схему оцінки – так звану RAG triad, що включає релевантність контексту, обґрунтованість (groundedness/faithfulness) відповіді за контекстом і релевантність відповіді до очікуваної; така схема дає змогу окремо вимірювати внесок ретриверу й генератора та налаштовувати гіперпараметри системи [41]. У якості формалізованої реалізації таких багатовимірних схем оцінювання використовується фреймворк RAGAS (Retrieval Augmented Generation Assessment), орієнтований на еталонно-незалежне оцінювання RAG-систем, із залученням великої мовної моделі, як аналог людського судження (когнітивного процесу оцінювання за суб'єктивними мірками). Фреймворк фокусується на трьох базових аспектах якості: узгодженість відповіді з наданим контекстом і відсутність галюцинацій; відповідність згенерованої відповіді початковому запиту; сфокусованість витягнутого контексту й мінімізацію зайвої інформації [42].

2.3.4 Обґрунтування вибору моделі системи масового обслуговування

Архітектура програмно-апаратного комплексу системи, яка поєднує Telegram-бота, генеративну модель LLM і базу даних, з погляду теорії масового обслуговування розглядається як система розподілу інформації, що обслуговує потоки вимог у випадкові моменти часу. У комп'ютерних мережах і сервісах подібні структури природно інтерпретуються як мережі систем масового обслуговування (СМО).

Кожне повідомлення до Telegram-бота формує вимогу на обслуговування. Потік цих вимог на робочих часових інтервалах доцільно апроксимувати стаціонарним пуассонівським потоком інтенсивності λ.

Подальша обробка утворює послідовність переходів вимог між вузлами мережі СМО. Після завершення обслуговування на Telegram-боті вимога з певними ймовірностями або завершується (простий сценарій відповіді), або передається до LLM-модуля, а за потреби – до підсистем доступу до бази даних.

Реалізація Telegram-бота ґрунтується на асинхронній обробці подій (Event Loop), де обробники конкурентно ведуть діалоги, тобто фізично вузол є одноканальною системою. Однак, Telegram API є невід'ємною функціональною частиною сервісу бота, який використовується додатком для надсилання та отримання даних. API функціонує як розподілена мережева служба, здатна до одночасної обробки багатьох запитів, забезпечуючи паралелізм обслуговування на своїй стороні. Завдяки неблокуючому характеру взаємодії з API, Event Loop усуває простій та блокування системи, виконуючи перемикання контексту. Це дозволяє боту відправляти одночасно декілька запитів, що імітує обслуговування логічними каналами.

З огляду на це логічне розпаралелення, вузол Telegram-бота пропонується розглядати як багатоканальну систему M/M/m. Параметр кількості каналів встановлюється виходячи з обмежень Telegram API, проте надалі буде вважатися рівним кількості користувачів, що працюють із системою. Таким чином, вхідний пуассонівський потік вимог розподіляється між m каналами, кожен з яких має експоненціальний розподіл часу обслуговування із параметром μ, а заявки формують спільну чергу з дисципліною FIFO. Структура моделі типу M/M/m на рисунку 2.5.

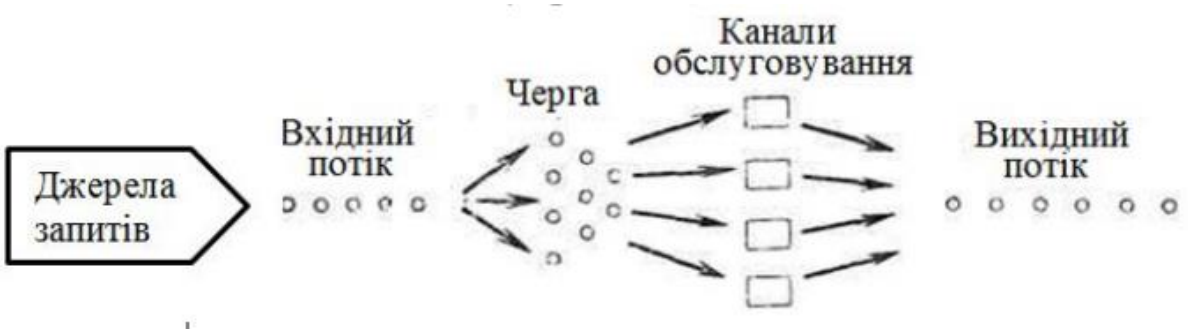


Рисунок 2.5 – Структура багатоканальної системи масового обслуговування [44]

Аналогічна інтерпретація застосовується до LLM-модуля та бази знань. Для генеративної моделі роль каналів відіграє максимально допустима кількість паралельних звернень, визначена квотами постачальника, а величина μ відображає середню швидкість формування відповіді однією сесією. Для документної бази даних MongoDB і векторного індексу кількість каналів визначається розміром пулу з'єднань і апаратними ресурсами вузла, а μ – середнім часом виконання типових операцій читання та запису.

Реальні обмеження на розмір буфера, максимальний час очікування та відмови в обслуговуванні при перевищенні квот фактично формують системи типу M/M/m/r, де r – розмір черги. Проте в рамках подальших теоретичних оцінок пропускної здатності та середнього часу відповіді ці ефекти враховуються узагальнено, а вузли Telegram-бота, LLM та бази знань розглядаються як елементи відкритої мережі СМО з відповідним типом M/M/m у найпростішому випадку з необмеженими чергами [43, 44].

2.3.5 Аналіз та формалізація інформаційних потоків і типів клієнтських запитів у системі обслуговування

Класична лінійна схема обробки запитів не завжди є оптимальною для системи, яка має справу з різнорідними запитами користувачів. У реальних умовах Customer Support частина звернень має чітко регламентовану структуру (наприклад, оформлення замовлення, уточнення реквізитів, перевірка статусу доставки), тоді як інша частина є відкритими інформаційними питаннями на кшталт «який інструмент краще вибрати», «як користуватися гарантією», «які умови сервісного обслуговування». Для таких сценаріїв доцільно використовувати різні типи обробки даних.

У зв'язку з цим було висунуто концепцію інтеграції стохастичної генерації та детермінованої логіки в межах єдиної інформаційної архітектури. Вона забезпечує узгодження «жорстких» бізнес-процесів (транзакційні сценарії) та «гнучких» інформаційних діалогів через єдину модель даних. У якості базових механізмів обрано:

– Finite State Machines (FSM) – для регламентованих процесів із фіксованою послідовністю кроків;

– Retrieval-Augmented Generation (RAG) – для інформаційно-довідкових запитів, де відповідь формується на основі релевантних документів.

Таким чином, логіку обробки запитів клієнтів можна подати як послідовний потік даних, що проходить через конвеєр із розгалуженням на дві гілки згідно концепції.

У межах формалізації інформаційних потоків позначимо через X вхідне повідомлення клієнта. На основі аналізу повідомлення X визначається маршрут, для запуску одного із сценаріїв:

$$\mathrm{F(X)} = f_{\mathrm{route}}(\mathrm{X}) \circ \begin{cases} \mathrm{F_{FSM}(X)}, & \text{якщо } \mathrm{C_{FSM}(X)} = 1, \\ \mathrm{F_{RAG}(X)}, & \text{якщо } \mathrm{C_{RAG}(X)} = 1, \end{cases}$$

де:

– $f_{\mathrm{route}}(X)$ – оператор маршрутизації верхнього рівня, який розділяє потоки запитів між FSM- та RAG-гілками – виконує «грубу» класифікацію звернень на основі простих детермінованих правил – команди меню, службові префікси, формат повідомлення тощо;

– $C_{\mathrm{FSM}}(X)$, $C_{\mathrm{RAG}}(X)$– бінарні умови належності запиту відповідно до регламентованого чи довільного сценарію;

– $F_{\mathrm{FSM}}(X)$, $F_{\mathrm{RAG}}(X)$ – відповідно детермінована та генеративна гілки обробки.

У межах виконання конвеєру-RAG поток даних деталізується послідовністю функціональних операторів:

– попередню обробку (нормалізацію) тексту – $f1(X)$;

– класифікацію запиту користувача – $f2$;

– пошук релевантної інформації у базі знань – $f3$;

– формування контекстного запиту до генеративної моделі – $f4$;

– побудову узгодженої відповіді – $f5$.

Оператор $f2$, згідно методики проектування гібридних RAG-систем, визначає як саме має працювати ретривер – тип пошуку, цільові колекції, релевантні поля, глибина контексту, можлива структура агрегувальних запитів до бази знань. На рисунку 2.6 спрямований ациклічний граф процесу.

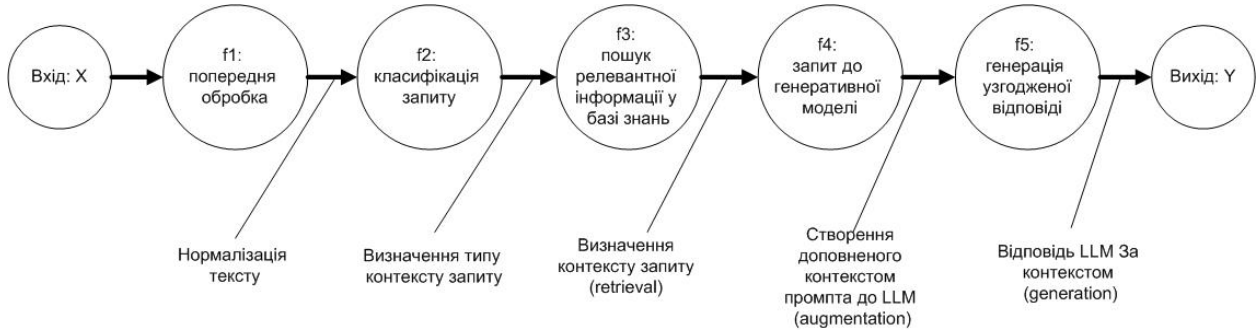


Рисунок 2.6 – Спрямований ациклічний граф потоку даних RAG-конвеєру

Механізм FSM виступає моделлю зі скінченною кількістю станів, у якій кожен стан відповідає певному регламентованому етапу діалогу з користувачем, а переходи між станами регламентуються чіткими умовами. Така модель гарантує детерміновану поведінку системи та повну передбачуваність результату.

FSM є скінченним автоматом [45]:

$$FSM = \langle S, A, \delta, s_0, s_n \rangle,$$

де:

– S – множина станів, що відповідають етапам діалогу;

– A – множина дій/подій, які ініціюють переходи;

– $\delta: S \times A \rightarrow S$ – функція переходів, що задає допустимі переходи між станами;

– s_0 – початковий стан;

– s_n – фінальний стан сценарію.

Для системи пропонується запровадити гілку FSM на процес оформлення замовлення процес. Послідовність станів для такого сценарію описується:

$$\begin{matrix} a1 & a2 & a3 & a4 & a5 \end{matrix}$$

$$s0 \leftrightarrow s1 \leftrightarrow s2 \leftrightarrow s3 \leftrightarrow s4 \leftrightarrow s5,$$

де $a1$ – вибір категорії товару, $a2$ – вибір товару, $a3$ – підтвердження ціни, $a4$ – вибір кількості товару, $a5$ – завершення замовлення. Також слід передбачити можливість відмовитись від замовлення.

Отже, запропонована концепція інформаційних потоків у системі забезпечує контроль послідовності кроків для критичних бізнес-процесів та надає адаптивність у роботі з відкритими інформаційними запитами.

2.4 Синтез загальної моделі системи компанії-постачальника будівельного обладнання

На основі проведеного аналізу та обґрунтування компонентів виконується синтез загальної моделі системи автоматизованого обслуговування клієнтів компанії-постачальника будівельного обладнання.

Для оцінювання часу реакції та пропускної здатності системи доцільно описувати її як мережу СМО, що реалізує розподіл інформаційних потоків між основними програмно-апаратними компонентами.

Перетворення вхідного повідомлення користувача X у вихідну відповідь Y подається як композиція операторів:

$$Y = fv_{\text{bot}} \circ fv_{\text{LLM}} \circ fv_{\text{DB}}(X, K),$$

де K – стан бази знань (набір документів, векторних подань і структурованих записів), оператор fv_{DB} реалізує функціонал доступу до гібридного сховища, fv_{LLM} – виконує генерацію відповіді за LLM, а fv_{bot} відповідає за маршрутизацію, FSM-логіку діалогу, валідацію параметрів і форматування результату для користувача. Комбінація та параметри операторів визначають структуру інформаційних потоків, що в свою чергу відповідають за якою ймовірністю заявка переходить у той чи інший вузол обслуговування.

Структуру моделі системи подано у вигляді графа $G = (V, E)$, зображеного на рисунку 2.7.

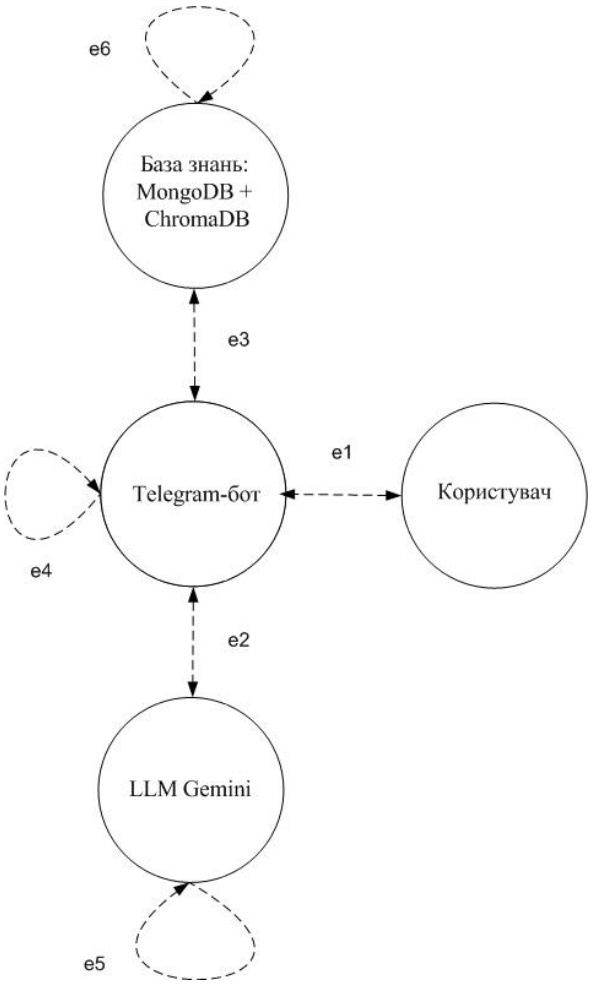


Рисунок 2.7 – Граф загальної моделі системи

Множина вузлів відповідає чотирьом ключовим компонентам:

– v_{user} – користувач системи (зовнішнє джерело та приймач інформації);

– v_{bot} – Telegram-бот, що реалізує інтерфейс взаємодії та прикладну логіку;

– v_{LLM} – хмарний модуль мовної моделі Gemini;

– v_{DB} – база знань, що поєднує документну БД MongoDB та векторне сховище.

Ребра графа задають канали взаємодії між вузлами разом з відповідними імовірностями переходів після завершення обслуговування. Імовірності переходів визначено теоретично як апроксимацію оптимального сценарію руху вимоги. Фактичні значення можуть мати інші значення, пропонується використати ці значення як еталонні. Ребра моделі:

– e1 – канал «Користувач ↔ Telegram-бот»: $P(v_{user} \rightarrow v_{bot}) = 1.0$ – усі нові запити користувача надходять у систему тільки через бота; $P(v_{bot} \rightarrow v_{user}) = 0.4$ – частка запитів, для яких бот формує відповідь самостійно та відразу відправляє користувачу;

– e2 – канал «Telegram-бот ↔ LLM Gemini»: $P(v_{bot} \rightarrow v_{LLM}) = 0.2$ – складні запити, що потребують генерації відповіді LLM; $P(v_{LLM} \rightarrow v_{bot}) = 0.9$ – випадки, коли результат LLM повертається до бота для форматування, постобробки чи продовження діалогу;

– e3 – канал «Telegram-бот ↔ База знань»: $P(v_{bot} \rightarrow v_{DB}) = 0.2$ – запити до MongoDB та векторного індексу: кешовані відповіді, прості операції читання/запису; $P(v_{DB} \rightarrow v_{bot}) = 0.85$ – повернення результатів БД до бота, який включає їх у відповідь або використовує у FSM-логіці;

– e4 – петля «Telegram-бот → Telegram-бот»: $P(v_{bot} \rightarrow v_{bot}) = 0.2$ – внутрішні переходи в FSM, уточнення даних, проміжні стани діалогу, службові повідомлення;

– e5 – петля «LLM Gemini → LLM Gemini»: $P(v_{LLM} \rightarrow v_{LLM}) = 0.1$ – повторні генерації в межах одного запиту: доопрацювання, переформатування або виправлення відповіді;

– e6 – петля «База знань → База знань»: $P(v_{DB} \rightarrow v_{DB}) = 0.15$ – послідовні та додаткові операції з даними – серії читань/записів, комбіновані запити.

Для формалізації маршрутизації вводиться матриця переходів R, що описує переходи між станами:

$$\mathrm{R} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0.4 & 0.2 & 0.2 & 0.2 \\ 0 & 0.9 & 0.1 & 0 \\ 0 & 0.85 & 0 & 0.15 \end{pmatrix}$$

Для подальшого розрахунку проводиться перехід від замкненої мережі до відкритої – розглядається вузол користувача як зовнішнє середовище – виділяється підматриця внутрішньої маршрутизації між трьома вузлами обслуговування v_{bot}, v_{LLM}, v_{DB}:

$$\mathrm{R}s = \begin{pmatrix} 0.2 & 0.2 & 0.2 \\ 0.9 & 0.1 & 0 \\ 0.85 & 0 & 0.15 \end{pmatrix}$$

Інтенсивність зовнішнього потоку вимог (звернення користувачів до бота) позначено через λ_0, а через λ_1, λ_2, λ_3 – стаціонарні інтенсивності потоків у вузлах v_{bot}, v_{LLM}, v_{DB}. Для відкритої мережі виконується система балансових рівнянь:

$$\begin{cases} \lambda_1 = \lambda_0 + 0.2\lambda_1 + 0.9\lambda_2 + 0.85\lambda_3 \\ \lambda_2 = 0.2\lambda_1 + 0.1\lambda_2 \\ \lambda_3 = 0.2\lambda_1 + 0.15\lambda_3 \end{cases} \quad (2.1)$$

Приведені навантаження вузла (коефіцієнти відвідувань) у моделі визначаються за:

$$\alpha_i = \lambda_i/\lambda_0,\ i = 1, 2, 3$$

Величина α_i показує, скільки разів у середньому одна зовнішня заявка відвідує i-й вузол мережі. Згідно розрахунків:

$$\alpha_1 = \frac{5}{2} = 2.5, \qquad \alpha_2 = \frac{5}{9} \approx 0.56, \qquad \alpha_3 = \frac{10}{17} \approx 0.59,$$

Кожен із вузлів v_{bot}, v_{LLM}, v_{DB} моделюється як багатоканальна система масового обслуговування типу M/M/m з параметрами λ_i, μ_i, m_i, де μ_i – інтенсивність обслуговування в одному каналі, а m_i – кількість каналів обслуговування. Коефіцієнт завантаження i-го вузла має вигляд:

$$\rho_i = \frac{\lambda_i}{m_i\mu_i} = \frac{\alpha_i\lambda_0}{m_i\mu_i}, \quad (2.2)$$

для стійкої роботи мережі повинен задовольняти умову $\rho_i < 1$ для всіх i. Середній час перебування заявки в і-му вузлі визначається як [44, 46]:

$$T_i = W_{q,i} + \frac{1}{\mu_i}, \quad (2.3)$$

де $W_{q,i}$ – середній час очікування в черзі – для M/M/m черга обчислюється згідно формули імовірності очікування Ерланга С (P_w) та розрахованою за нею середньою довжиною черги (L_q):

$$Lq_i = Pw_i\frac{\rho_i}{1-\rho_i}, \qquad Wq_{,i} = \frac{Lq_i}{\lambda_i},$$

З урахуванням коефіцієнтів відвідувань інтегральний показник якості сервісу – середній час реакції системи – задається співвідношенням:

$$T_{resp} = \alpha_1 T_{bot} + \alpha_2 T_{LLM} + \alpha_3 T_{DB} = 2.5T_{bot} + 0.56T_{LLM} + 0.59T_{DB} \quad (2.4)$$

Нормативний час реакції $T_{\text{допуст}}$ накладає обмеження

$$T_{resp} \leq T_{\text{допуст}}.$$

Конкретні числові значення λ_i, μ_i, m_i та розрахований за ними T_{resp} для проєктованої системи визначаються в експериментальному розділі на основі виміряної статистики звернень до Telegram-бота, LLM і бази знань та порівнюються з нормативом, після чого робиться висновок щодо продуктивності та якості обслуговування обраної архітектури.

2.5 Обґрунтування і вибір методів експериментальних досліджень

Для перевірки працездатності, ефективності та достовірності теоретичних положень моделі передбачено комплекс експериментальних досліджень, спрямованих на валідацію архітектури системи автоматизованого обслуговування клієнтів компанії-постачальника будівельного обладнання, що інтегрує Telegram-бота, гібридний механізм обробки запитів (FSM/RAG), генеративну мовну модель та сховище знань на базі MongoDB з векторним представленням.

Наукове завдання полягає у підтвердженні доцільності й ефективності запропонованого гібридного підходу архітектури до виконання вимог продуктивності та якості сервісу за змінних показників навантаження.

Для досягнення цієї мети слід використовувати комплекс методів, що включає:

– імітаційне моделювання процесів на основі апарату теорії масового обслуговування;

– навантажувальне тестування підсистеми зберігання даних;

– векторні методи оцінювання якості відповідей LLM та ефективності RAG-конвеєра із застосуванням спеціалізованих метрик RAGAS;

– мануальні тестування окремих компонентів.

Імітаційне моделювання мережі СМО використовується для валідації синтезованої відкритої мережевої моделі. У ході експерименту для фіксованих

сценаріїв навантаження формується масив тестових звернень до Telegram-бота, за яким оцінюється інтенсивність зовнішнього потоку λ_0. Стаціонарні інтенсивності потоків у вузлах приймаються відповідно до аналітичної моделі.

Параметри обслуговування μ_i визначаються окремими вимірюваннями на кожному вузлі: для вузла сховища – за результатами навантажувального тестування описаному нижче; для вузла LLM – за вибірками часу відгуку на типові запити; для вузла бота – за затримками внутрішньої обробки запитів без урахування зовнішніх викликів. На основі вибірок оцінюється середній час обслуговування в кожному вузлі, а інтенсивність обслуговування μ_i – як обернена величина.

Отримані значення використовуються: для перевірки виконання умов стійкості $\rho_i < 1$ для кожного вузла; розрахунку середнього часу очікування в черзі $W_{q,i}$, повного часу перебування у вузлі T_i та агрегованого часу реакції системи T_{resp}. Далі експериментальний час реакції $T_{resp}{}^{exp}$, обчислений на рівні всієї системи, порівнюється з теоретичним значенням T_{resp} і нормативним обмеженням $T_{\text{допуст}}$.

Підсистема бази знань у проєктованій системі включає документно-орієнтовану СУБД MongoDB та векторне сховище, які разом утворюють відповідний сервісний вузол мережі СМО. Вона розглядається як критичний компонент бізнес-процесів усього контуру обробки клієнтських запитів, оскільки визначає затримки доступу до каталогу товарів, регламенти зберігання корпоративних даних, а також впливає на якість відповіді в RAG-сценаріях. Тому метою тестування є перевірка того, що підсистема забезпечує такі значення продуктивності, за яких внесок роботи зі сховищем у загальний час відповіді системи є прийнятним, а виконання операцій не суперечить політикам інформаційної безпеки.

Експериментальне оцінювання продуктивності підсистеми бази знань ґрунтується на багаторазових незалежних вимірюваннях латентності та пропускної спроможності в двох групах сценаріїв при багатокористувацькому навантаженні:

– операції MongoDB – читання, запис, агрегувальні конвеєри – з фіксацією часу виконання та статусу кожної операції;

– векторний пошук (з локальним ONNX-енкодером).

На основі отриманих вибірок обчислюються типові значення затримки та «хвости» розподілу та дозволяють оцінити поведінку системи в найгірших 5%-1%. Ефективна інтенсивність обслуговування вузла бази знань у моделі СМО визначається на основі агрегованих результатів обох типів тестів для репрезентативних сценаріїв навантаження.

Методи формальної перевірки безпечності агрегувальних конвеєрів MongoDB ґрунтуються на аналізі структури запитів і виключенні потенційно небезпечних операторів (виконання довільного коду, неконтрольований вивід результатів у зовнішні колекції тощо). Цей метод забезпечує перевірку відповідності системи вимогам до захищеності обробки корпоративних даних.

Якість роботи генеративної мовної моделі та RAG-конвеєра оцінюється автоматизованими методами, які не вимагають ручного ранжування відповідей. Для цього формується набір тестових сценаріїв, у яких кожному запиту ставиться у відповідність еталонна відповідь; для RAG-сценаріїв додатково задається очікуваний набір релевантних документів із бази знань.

Перший рівень оцінювання базується на векторному представленні текстів. Запити, еталонні відповіді, фактичні відповіді та текстові фрагменти з бази знань відображаються у спільний векторний простір. На цій основі обчислюються косинусні коефіцієнти подібності, які використовуються для визначення: якості пошуку документів, ступеня опори на документи, галюцинації, якості відповіді відносно еталону.

Другий рівень оцінювання реалізується через використання спеціалізованого фреймворку RAGAS, який обчислює інтегральні метрики релевантності відповіді та її фактологічної коректності відносно еталонних відповідей і наданого контексту (документи).

У сукупності ці методи утворюють методику якісної оцінки роботи LLM і RAG-конвеєра, яка базується на числових показниках і не потребує суб’єктивного ручного оцінювання.

Окремо виділяється смок-тест гілки детермінованої FSM-логіки. Для нього формується еталонний сценарій оформлення замовлення користувачем, після чого виконується перевірка прохідності послідовності станів. Тестування повинно довести, що критичні бізнес-процеси не містить логічних розривів та суперечностей із заданими правилами обробки.

Фактори, які впливають на досліджувані показники, доцільно класифікувати на такі групи:

– архітектурно-структурні фактори – обмеження кількості паралельних каналів постачальниками сервісів, конфігурація серверних ресурсів, схема розподілу навантаження, параметри організації сховища знань;

– навантажувальні фактори – інтенсивність вхідного потоку λ, співвідношення між «легкими» сценаріями та запитами, що потребують повного RAG-ланцюга, характер навантаження (стаціонарний режим, пікові інтервали);

– інформаційні фактори – обсяг та структура бази знань, повнота каталогу товарів, репрезентативність набору тестових запитів і еталонних відповідей, довжина та складність запитів до LLM;

– алгоритмічні фактори – налаштування параметрів генерації, вибір і параметризація метрик.

Результати експериментальної перевірки мають підтвердити, що всі структурні елементи моделі – контур Telegram-бота, сервісні канали СМО, підсистема бази знань, контур RAG і детермінована FSM-логіка – у сукупності забезпечують виконання заданих функціональних вимог: час відповіді не більше 10 секунд при не менше ніж 10 активних заявок, стійкість до пікових навантажень і захищеність каналів обміну даними, а також зменшення частки некоректних відповідей мовної моделі після застосування гібридної стратегії обробки запитів із використанням RAG мінімум на 50%.

3 СИНТЕЗ КОМП'ЮТЕРНОЇ СИСТЕМИ

3.1 Цілі впровадження системи

Основною метою впровадження системи є автоматизація обробки звернень користувачів в інформаційній інфраструктурі компанії-постачальника будівельного обладнання на основі чат-бота з підтримкою взаємодії природною мовою, з використанням зовнішньої мовної моделі та доступом до внутрішніх корпоративних даних.

До основних цілей належать:

– забезпечення часу відповіді на типові запити до значень, що відповідають нормативам;

– забезпечення можливості одночасного обслуговування значної кількості користувачів без суттєвої деградації продуктивності;

– зменшення навантаження на персонал служби підтримки за рахунок автоматизації стандартних операцій та уніфікації сценаріїв відповіді;

– накопичення структурованих даних про звернення користувачів для подальшого аналізу та вдосконалення сервісу;

– забезпечення конфіденційності, цілісності та доступності даних під час обробки й зберігання інформації в системі.

3.2 Формулювання технічних вимог до розроблюваної системи

3.2.1 Вимоги до реалізації системи

Вимоги до реалізації інтегрованої комп'ютерної системи автоматизованого обслуговування клієнтів визначають архітектурні, продуктивні та експлуатаційні характеристики, необхідні для забезпечення стабільної роботи чат-бота.

Система повинна бути розгорнутою та інтегрованою в наявне корпоративне середовище компанії-постачальника будівельного обладнання, з урахуванням існуючої мережевої інфраструктури, засобів захисту інформації та регламентів адміністрування.

Система повинна бути реалізована як багаторівнева розподілена архітектура, що включає окремі логічні компоненти:

– сервер застосунку чат-бота, який взаємодіє з Telegram API та реалізує бізнес-логіку обробки діалогів;

– модуль інтеграції з зовнішньою мовною моделлю (LLM) через захищений API;

– RAG-ядро, як логічно виділена частина серверу застосунку, що виконує семантичний пошук по документній базі знань та формує контекст для LLM;

– підсистему роботи зі структурованими даними на основі СУБД MongoDB та векторного індексування;

– модулі журналювання, моніторингу та збору технічних метрик.

Реалізація повинна забезпечувати логічну і фізичну ізоляцію компонентів з можливістю їх незалежного оновлення та масштабування. Серверні компоненти мають бути розгорнуті у виділених сегментах корпоративної мережі з використанням демілітаризованої зони для зовнішніх сервісів.

Система повинна забезпечувати виконання таких вимог до продуктивності:

– обробку не менше 10 одночасних сесій користувачів без суттєвої деградації якості сервісу;

– час формування відповіді на типові запити не більше 10 с при вищезазначеному навантаженні та не більше 1 с для не менше ніж 95% простих регламентованих звернень (без залучення LLM- та RAG-модуля);

– час семантичного пошуку в RAG-ядрі (отримання релевантних фрагментів із бази знань) не більше заданого порогу, що забезпечує виконання загального часу відповіді в межах встановлених вище;

Реалізація повинна підтримувати асинхронну обробку запитів і використання механізмів черг/буферів для згладжування пікових навантажень, а також застосування кешування проміжних результатів для повторюваних типових звернень.

Система повинна забезпечувати виконання наступних вимоги до надійності:

– доступність сервісу не нижче 99,5% календарного часу;

– збереження працездатності при відмові окремих компонентів;

– централізоване журналювання технічних подій та інцидентів з можливістю оперативного аналізу причин збоїв;

– можливість керованого оновлення компонентів без тривалого простою сервісу.

3.2.2 Вимоги до функцій виконуваних системою

Функціональні вимоги визначають перелік операцій, які інтегрована комп’ютерна система повинна забезпечувати для користувачів, операторів та адміністратора, а також загальні обмеження щодо сценаріїв взаємодії.

Система повинна забезпечувати через інтерфейс чат-бота Telegram наступні функції рівня користувача:

– старт взаємодії, отримання довідкової інформації про можливості сервісу та доступних команд;

– пошук товарів за назвою, артикулом, категорією або ключовими словами з відображенням основних характеристик, орієнтовної вартості та наявності;

– оформлення замовлення з детальною інформації щодо умов придбання, доставки та оплати будівельного обладнання;

– перевірка статусу замовлення за номером або іншими ідентифікаторами, з відображенням поточного етапу обробки;

– перевірка гарантійних зобов’язань за серійним номером або документом придбання, отримання інформації про найближчі сервісні точки;

– передача звернень, що потребують участі персоналу, з можливістю наступного діалогу з оператором служби підтримки;

Забезпечення функцій рівня операторів та адміністраторів до системи наступні:

– забезпечувати фіксацію результатів обробки етапів звернення: класифікація, проміжні результати, відповідь системи;

– надавати базові засоби перегляду історії звернень користувачів для подальшого аналізу та коригування сценаріїв обслуговування;

– підтримувати накопичення структурованих даних у базі знань, які надалі використовуються для аналітики та удосконалення сервісу.

3.2.3 Вимоги до видів забезпечення

Функціонування інтелектуальної системи автоматизованого обслуговування клієнтів компанії-постачальника будівельного обладнання потребує визначення вимог до основних видів забезпечення: програмного, інформаційного, технічного, організаційного та методичного.

Програмне забезпечення повинно базуватися на використанні офіційного Telegram Bot API та засобів розробки на мові Python. Для реалізації чат-бота застосовується фреймворк aiogram. Для доступу до БД використовується офіційний драйвер MongoDB – motor. Зовнішньою мовною моделлю повинна виступати LLM сімейства Gemini 2.5 та інтегруватися через офіційний SDK Google GenAI.

Інформаційне забезпечення охоплює структуру та спосіб зберігання даних у базі знань. У системі, у компоненті MongoDB, мають бути визначені колекції для користувачів, замовлень, сесій діалогів, товарів, довідкових даних та журналів подій. Векторне подання бази знань повинно вміщувати документацію товарів, регламенти та метадані, для кожного з яких обчислюється векторні вкладення (ембеддинг) фіксованої розмірності за допомогою моделі класу all-MiniLM-L6-v2 з подальшим збереженням у ChromaDB.

Технічне забезпечення включає серверні ресурси та робочі станції достатні для роботи сервісу згідно вимог продуктивності та надійності. Мінімальні вимоги до серверів, на яких розгортаються чат-бот, база даних та векторне сховище, передбачають використання:

– багатоядерних процесорів – не менше 6 ядер із тактовою частотою від 2,6 ГГц;

– оперативної пам’яті обсягом від 16 ГБ;

– SSD-сховища обсягом не менше 60ГБ.

Мережева інфраструктура повинна забезпечувати пропускну здатність не нижче 1 Гбіт/с для серверних сегментів, підтримку повнодуплексного режиму та резервування ключових каналів зв’язку. До технічного забезпечення також належать джерела безперебійного живлення.

Методичне забезпечення передбачає наявність комплекту експлуатаційної документації та регламентів, що включають: інструкції для кінцевих користувачів чат-бота, керівництва для адміністраторів щодо розгортання, резервного копіювання та відновлення, а також протоколи тестування.

3.2.4 Вимоги до захисту інформації

Захист інформації в інтегрованій системі автоматизованого обслуговування клієнтів має забезпечувати конфіденційність, цілісність і доступність персональних даних користувачів та внутрішньої інформації компанії-постачальника будівельного обладнання.

До основних вимог захисту інформації належать:

– усі дані, що передаються між компонентами системи та клієнтськими пристроями, повинні передаватися виключно захищеними каналами з використанням сучасних криптографічних протоколів: TLS для серверних з’єднань, вбудований MTProto у середовищі Telegram;

– доступ до зовнішніх сервісів Telegram Bot API, Google-API LLM Gemini має здійснюватися за допомогою API-токенів (ключів), що зберігаються у захищеному серверному середовищі з обмеженим доступом; термін дії токенів та ключів доступу повинен бути обмеженим, а їх ротація – виконуватися з регламентованою періодичністю не рідше ніж один раз на 90 днів;

– доступ до бази даних та векторного сховища повинен здійснюватися лише з вузлів серверної інфраструктури із застосуванням механізмів

автентифікації та авторизації; сервісні облікові записи мають отримувати лише мінімально необхідні права для роботи застосунку;

– для серверних компонентів системи мають застосовуватися засоби міжмережевого екранування та фільтрації трафіку, обмежувати доступ до відкритих сервісів лише необхідними портами та протоколами;

– програмні компоненти повинні бути захищені від типових атак на веб- та API-рівні – NoSQL-ін’єкції, несанкціоновані виклики API та DDoS, з використанням засобів обмеження частоти запитів і контролю коректності вхідних даних;

– система повинна підтримувати регулярне резервне копіювання баз даних і конфігураційних файлів із цільовими часом відновлення не більше 15 хвилин;

– усі події, пов’язані з доступом до персональних даних, змінами прав користувачів, роботою з критичними конфігураціями та інцидентами безпеки, мають підлягати централізованому журналюванню; журнали повинні зберігатися не менше 1 року з обмеженим доступом;

– користувачам має надаватися інформація щодо правил безпечного використання сервісу.

3.2.5 Вимоги до ергономіки системи

Вимоги до ергономіки системи визначають зручність взаємодії користувачів, операторів та адміністраторів із сервісом, а також організацію робочого середовища так, щоб забезпечити швидке засвоєння, мінімальне когнітивне навантаження та стабільну якість обслуговування.

Система повинна забезпечувати такі властивості інтерфейсу для користувачів:

– взаємодія з сервісом здійснюється через Telegram-чат у форматі коротких, однозначно сформульованих повідомлень українською мовою без спеціалізованої термінології, що не є необхідною для виконання задачі;

– основні регламентовані сценарії повинні виконуватися не більше ніж за 4-5 кроків діалогу з боку користувача;

– повідомлення бота мають мати уніфіковану структуру та не перевищувати довжину, зручну для сприйняття на екранах мобільних пристроїв; у межах одного повідомлення не рекомендується відображати більше 5-7 позицій у списку;

– для регламентованих дій, а також для навігації в довгих відповідях (перегляд наступних/попередніх фрагментів тексту) мають використовуватися стандартні елементи Telegram – кнопки, інлайн-меню;

– повідомлення про помилки та виняткові ситуації повинні формулюватися у зрозумілій формі з пропозицією конкретних подальших дій: повторити запит потім, змінити параметри тощо.

Вимоги до ергономіки інтерфейсу та робочого місця оператора й адміністратора передбачають:

– підтримувати базові засоби фільтрації й агрегування для моніторингу навантаження й типових проблем, помилок;

– забезпечувати раціональне розташування монітора, клавіатури й маніпуляторів, належне освітлення робочого місця, відсутність надмірного шуму, шкідливих умов й відволікаючих факторів;

– забезпечувати доступ до актуальної експлуатаційної документації безпосередньо з робочого місця.

3.2.6 Розробка схеми функціональної структури

Функціональна схема інтегрованої системи автоматизованого обслуговування клієнтів відображає взаємодію користувачів з чат-ботом у середовищі Telegram, монолітним серверним застосунком Telegram Bot Application, зовнішньою мовною моделлю та гібридним сховищем даних MongoDB. У межах схеми умовно виділяються три рівні:

– інтерфейсу;

– серверного ядра;

– даних.

На рівні інтерфейсу користувачі працюють із стандартними клієнтами Telegram. Telegram API виконує функції транспортного та візуального фронтенда: приймає повідомлення від користувача, доставляє оновлення до вузла бота та повертає сформовані відповіді й кнопки інтерфейсу.

На рівні серверного ядра функціонує монолітний застосунок Telegram Bot Application. Він приймає оновлення від Telegram API, виконує розбір запитів і керування діалогами, розрізняючи детерміновані сценарії з наперед визначеними відповідями та запити, що потребують обробки мовною моделлю. Для першої групи запитів застосунок використовує шаблонні відповіді та прямий доступ до даних у сховищі. Для другої групи задіюється RAG-компонент: дані з MongoDB та векторного індексу, передаються до мовної моделі через Gemini API.

На рівні даних розташоване гібридне сховище MongoDB. Компонент MongoDB забезпечує зберігання структурованої інформації, ChromaDB – містить векторний індекс бази даних для семантичного пошуку релевантних фрагментів. Для детермінованих сценаріїв серверний застосунок звертається переважно до структурованих колекцій MongoDB, тоді як інтелектуальний модуль використовує обидві складові гібридного сховища.

Схематично функціональна структура подана у вигляді монолітної сервісної архітектури із зображення послідовності потоків взаємодій між компонентами системи представлених на рисунку 3.1.

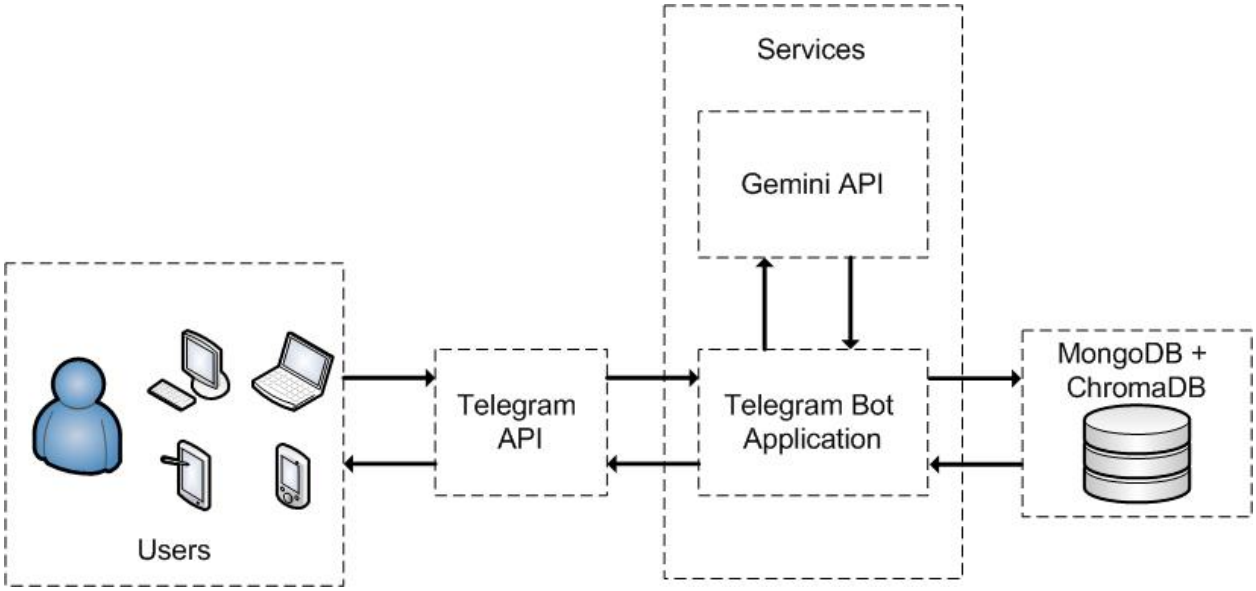


Рисунок 3.1 – Схема функціональної структури системи

3.3 Вибір апаратних засобів і обґрунтування бази розгортання

Вибір апаратних засобів для системи автоматизованого обслуговування клієнтів здійснюється з урахуванням вимог до продуктивності, відмовостійкості та доступності сервісу, сформульованих у попередніх підрозділах.

Система автоматизованого обслуговування клієнтів функціонує в межах уже розгорнутої корпоративної інфраструктури компанії-постачальника будівельного обладнання, мережева топологія підприємства використовується як готова платформа для розміщення серверних компонентів системи та в рамках даного проєкту не підлягає зміні. Передбачається, що наявне мережеве обладнання забезпечує достатню пропускну здатність, резервування й рівень надійності для роботи розроблюваної системи та не потребує масштабування.

Спеціальні вимоги до апаратних засобів кінцевих користувачів не висуваються, достатньо:

– персональний комп’ютер, смартфон чи планшет з установленим клієнтом Telegram актуальної версії;

– доступ до мережі Інтернет.

Серверна частина системи розгортається на виділених хостах у корпоративному ізольованому дата-центрі компанії-постачальника будівельного обладнання. Для розміщення монолітного застосунку Telegram-бота та гібридного сховища даних передбачається використання двох серверних вузлів із резервуванням живлення. Специфікацію обладнання наведено в таблиці 3.1.

Таблиця 3.1 – Специфікація серверного обладнання

Позиція	Найменування і технічна характеристика	Тип, марка, позначення	Одиниця виміру	Кількість	Примітка
1	Сервер для розгортання застосунку Telegram-бот: процесор 1×6 ядер, ≥2,6 ГГц; ОЗП 16 ГБ DDR4; SSD 1×60 ГБ; 2×RJ-45 1 Гбіт/с; ОС Linux/Windows	Сервер загального призначення	од.	1	Основний сервер для виконання монолітного застосунку чат-бота

Кінець таблиці 3.1

2	Сервер бази даних: процесор 1×6 ядер, ≥2,6 ГГц; ОЗП 16 ГБ DDR4; SSD 1×60 ГБ; 2×RJ-45 1 Гбіт/с	Сервер зберігання даних	од.	1	Централізоване сховище структурованих і векторних даних
3	Джерело безперебійного живлення потужністю не менше 3000 VA	UPS	од.	2	

Робочі станції адміністраторів та операторів системи необхідно реалізувати на базі стандартних офісних ПК, наявних у доступі компанії, рекомендовано з чотири- або більше ядерними процесорами, інтегрованим або десктопним графічним процесором, 8-16 ГБ оперативної пам’яті та моніторами із достатньою роздільною здатністю для відображення інтерфейсів моніторингу.

Мережеве обладнання надається компанією та експлуатується відповідно до існуючої топології, регламенту та політик. Розроблювана система використовує наявні мережеві ресурси, не змінюючи їх конфігурацію, окрім необхідних налаштувань доступу до серверів. Основні мережеві пристрої, що задіяні у функціонуванні системи, узагальнено в таблиці 3.2.

Таблиця 3.2 – Специфікація мережевого обладнання

Позиція	Найменування і технічна характеристика	Тип, марка, позначення	Один. вим.	Кіль-кість	Примітка
1	Граничний маршрутизатор з підтримкою NAT, VPN, ACL; пропускна здатність не менше 1 Гбіт/с; 2×WAN, 4×LAN RJ-45	Маршрутизатор корпоративного класу	од.	1	Організація виходу до Інтернету та розмежування зовнішнього/ внутрішнього трафіку

Кінець таблиці 3.2

2	Керований L3-комутатор ядра з 24×1 Гбіт/с портами, підтримкою VLAN, STP, QoS, SNMP	Комутатор ядра мережі	од.	1	Об’єднання серверів та основних сегментів мережі, маршрутизація між VLAN
3	Керований комутатор доступу 24×FastEthernet, 2×Gigabit uplink, підтримка VLAN, SNMP	Комутатор доступу	од.	2-4	Підключення робочих станцій операторів, адміністративних ПК тощо

3.4 Синтез структурної схеми інтегрованої системи

Синтез структурної схеми інтелектуальної системи автоматизованого обслуговування клієнтів проводиться на основах теоретичної моделі, технічних вимог, обраних апаратних засобах та узгоджується з функціональною схемою.

Основою побудови є розмежування зовнішнього середовища (користувачі, хмарні API) та захищеного внутрішнього периметра корпоративної мережі. Взаємодія між цими областями здійснюється через демілітаризовану зону, де розташовано граничний маршрутизатор і міжмережевий екран. Граничний маршрутизатор виконує функції NAT, первинної маршрутизації та повинен містити статичні маршрути до серверів, а міжмережевий екран реалізує політики фільтрації трафіку, контроль доступу та моніторинг мережевих потоків.

До зовнішніх компонентів належать:

– користувачі – клієнти компанії, які працюють із системою через застосунок Telegram на персональних комп’ютерах чи мобільних пристроях, передаючи запити мережею Інтернет;

– шлюз Telegram API – інфраструктура платформи Telegram, що приймає повідомлення від користувачів і за протоколом HTTPS доставляє їх до

серверного застосунку бота через налаштований Long polling, а також передає сформовані відповіді у зворотному напрямку;

– шлюз Gemini LLM – хмарний сервіс, що надає доступ до генеративної мовної моделі через захищений програмний інтерфейс. Серверний застосунок надсилає запити до цього сервісу лише з внутрішнього периметра.

Корпоративна мережа компанії логічно сегментована на локальні мережі підрозділів компанії. Сегменти, що не беруть участь у роботі серверної частини системи (робочі станції офісних підрозділів, допоміжні служби), ізольовано від критичної зони, де розташовано сервери системи, що обмежує можливість поширення загроз та несанкціонованого доступу.

Ключові серверні компоненти системи розгортаються в окремій віртуальній локальній мережі Servers VLAN у складі локальної мережі IT-відділу (IT LAN). У цьому сегменті розміщуються:

– сервер монолітного застосунку Telegram Bot Application, який приймає запити від Telegram API, реалізує керування діалогами й ініціює звернення до сховища даних та мовної моделі Gemini;

– сервер сховища даних, що забезпечує зберігання структурованої інформації та векторного індексу. Доступ до цього сервера дозволений лише з боку серверного застосунку та адміністративних станцій IT-відділу.

Проходження запиту у межах структурної схеми відбувається таким чином: повідомлення користувача надходить до шлюзу Telegram API, звідти передається на граничний маршрутизатор корпоративної мережі та після фільтрації спрямовується на сервер Telegram Bot Application; застосунок обробляє запит відповідно до вибраного сценарію – звернення до сховища та, за потреби, до мовної моделі Gemini; сформована відповідь повертається на шлюз Telegram API і доставляється користувачеві зворотним шляхом.

Структурна схема інтегрованої систем зображена на рисунку 3.2.

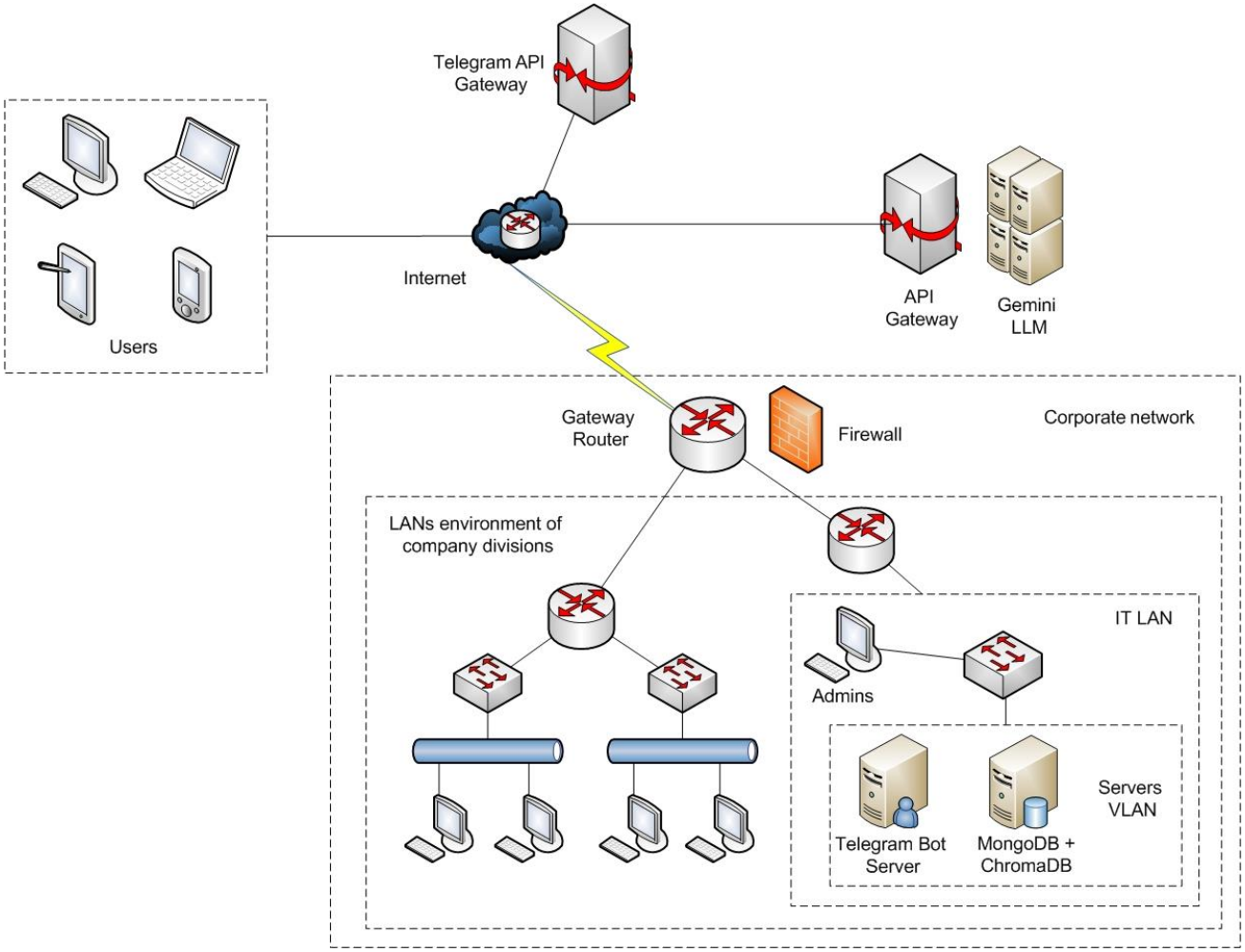


Рисунок 3.2 – Структурна схема інтегрованої системи

4 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Призначення й область застосування програми

Програмний засіб призначений для автоматизованої обробки текстових звернень користувачів у середовищі Telegram та забезпечує доступ до інформаційно-довідкових і сервісних функцій компанії-постачальника будівельних інструментів та обладнання. Програма виконує функції приймання повідомлень, семантичної інтерпретації запитів, формування відповідей та ініціювання процедур, пов’язаних із пошуком товарів і замовлень, отриманням відомостей про торгові точки та оформленням сервісних запитів.

Сфера застосування охоплює цифрові канали взаємодії з клієнтами, де необхідна безперервна обробка типових звернень у режимі реального часу без участі оператора.

4.2 Обґрунтування технічних характеристик

4.2.1 Постановка задачі та опис застосовуваних математичних методів

Завданням розроблення програмного засобу є створення компонента серверної частини, що здійснює приймання текстових повідомлень користувачів через Telegram Bot API, визначення типу запиту, обробку даних отриманих із документно-орієнтованої та векторної бази даних, та формування стандартизованих текстових відповідей. Програмний засіб повинен забезпечувати взаємодію з інформаційними ресурсами підприємства та підтримувати багатокрокові діалогові сценарії.

Для виконання зазначених функцій використовуються наступні математичні методи:

– інтерпретація вхідних текстових повідомлень здійснюється великою мовною моделлю, яка реалізує статистичну апроксимацію відображення тексту у простір ознак і використовується для визначення структури запиту та параметрів подальших операцій;

– семантичне зіставлення у векторний простір, у якому попередньо підготовлені текстові фрагменти зберігаються у вигляді числових векторів для оцінювання відстані між вектором запиту та векторами документів;

– організації послідовності дій у багатокрокових сценаріях застосовується скінчений автомат станів (FSM).

Застосовувані математичні методи передбачають низку обмежень:

– інтерпретація повідомлень залежить від властивостей та навченості великої мовної моделі;

– векторне зіставлення забезпечує лише наближену оцінку подібності текстових фрагментів;

– скінченний автомат передбачає лише фіксований набір станів та переходів.

4.2.2 Опис алгоритму функціонування програми з обґрунтуванням вибору схеми рішення задачі

Алгоритм функціонування програмного засобу побудовано за принципом подієво-орієнтованої обробки вхідних даних. Програма отримує текстові повідомлення через Telegram Bot API і передає їх на обробку серверному компоненту.

Подальша обробка реалізується у два етапи. На першому етапі відбувається розділення вхідних звернень на дві гілки: обробку явно заданих команд та обробку довільних текстових повідомлень мовною моделлю.

На другому етапі виконується власне обробка обраної гілки. Для командних запитів запускаються відповідні довідникові або функціональні сценарії. Для довільних запитів відбувається додаткова класифікація на дію.

Класи дій, що потребують доступу до корпоративних даних, визначають гібридну схему обробки за принципом RAG – формуються запити до БД та векторного сховища згідно запиту користувача. Отримані дані подаються мовній моделі для формування фінальної відповіді користувачу.

Для інформаційних запитів, що не потребують доступу до даних, відповідь генерується без звернення до корпоративних сховищ.

Для підвищення швидкодії застосовується кешування відповідей мовної моделі. Перед виконанням операцій перевіряється наявність відповідного результату у внутрішньому сховищі, що дозволяє скоротити затримку при повторних зверненнях з подібними параметрами.

Багатокрокові сценарії, зокрема процес формування замовлення, викликаються як явними командами, так і автоматично на основі класифікації запиту. Їх реалізовано за допомогою станової моделі: кожний крок визначається як окремий стан із фіксованим набором допустимих переходів. Вибір такої схеми забезпечує контрольованість виконання, виключає неоднозначність і гарантує завершеність процесу критичного бізнес-процесу.

4.2.3 Опис і обґрунтування вибору методу організації вхідних і вихідних даних

Вхідні дані з клієнтської сторони виступають текстові повідомлення та callback-ідентифікатори Telegram, що передаються у вигляді рядкових структур фіксованого формату. Використання таких форматів обумовлене вимогами Telegram Bot API та дозволяє гарантувати однозначну інтерпретацію дій користувача. Текстові повідомлення виступають джерелом неструктурованих даних, які інтерпретуються мовною моделлю, callback-дані та команди – містять параметри для обробки конкретних дій операторів та сценаріїв автоматів.

Структуровані вхідні дані, необхідні для формування відповідей, отримуються з документно-орієнтованої бази даних, де інформація організована у вигляді колекцій з фіксованими полями. Векторне сховище реалізовано на базі окремої колекції, у якій елементи містять числові вектори ознак та метадані для зв’язку з відповідними документами.

Вихідні дані організовані у вигляді текстових повідомлень Telegram із можливістю включення інтерактивних елементів, таких як inline-кнопки, а також записи до БД після взаємодії з користувачем.

4.2.4 Опис і обґрунтування вибору складу технічних і програмних засобів

Вибір складу технічних та програмних засобів повинен відповідати вимогам до функціонування програмного засобу, забезпечувати сумісність із застосовуваними інтерфейсами взаємодії, підтримувати використані обчислювальні моделі та дозволяти реалізувати необхідний алгоритмічний та інформаційний обмін у рамках розроблюваної системи.

Склад технічних засобів повинен включати серверне середовище на базі операційних систем Linux або Windows Server з підтримкою виконання Python-застосунків. Рекомендована апаратна конфігурація повинна включати: процесор з кількістю ядер не менше 6 та тактовою частотою від 2.6 ГГц; оперативну пам’ять обсягом не менше 16 ГБ; твердотілий накопичувач обсягом від 60 ГБ; канал підключення до мережі не менше ніж 1 Гб/с.

Склад програмних засобів повинен включати мову програмування Python версії 3.11 або новішої, що підтримує та включає: асинхронний фреймворк для взаємодії з Telegram Bot API, клієнт для доступу до MongoDB, векторне сховище ChromaDB, SDK інтеграції із зовнішньою мовною моделлю Google Gemini.

4.3 Опис програми

4.3.1 Загальні відомості

4.3.1.1 Позначення та назва програми

Програмний засіб має внутрішній ідентифікатор «TG.BudStore.Bot.1». Офіційна назва програми – «Телеграм-бот для автоматизованої взаємодії з клієнтами компанії-постачальника будівельного обладнання».

4.3.1.2 Необхідне програмне забезпечення

Для функціонування програмного засобу необхідне наступне програмне забезпечення для серверної частини:

– операційна система Ubuntu Server 22.04 LTS і новіша, або Windows Server 2019 і новіша;

– інтерпретатор мови програмування Python версії 3.11 або новішої із засоби керування пакетами та віртуальними середовищами (pip, Python-env);

– система керування базами даних MongoDB Community Server версії 8.0 або новішої;

– векторне сховище ChromaDB версії 1.3.4 або новішої;

Набір бібліотек Python:

– aiogram 3.13.0 або новіше;

– motor 3.7.0 та pymongo 4.11.3 або новіше;

– chromadb 1.3.4 або новіше;

– google-generativeai 0.8.5 або новішої.

Клієнтська частина:

– операційні системи: Android 10 або новіша, iOS 15 або новіша, Windows 10 або новіша, macOS 12 або новіша;

– встановлений офіційний клієнт Telegram для відповідної платформи або веб-версія Telegram Web.

4.3.1.3 Мови програмування

Текст програми реалізовано мовою програмування Python версії 3.11.

4.3.2. Функціональне призначення

4.3.2.1 Класи розв’язуваних задач

Програмний засіб орієнтований на розв’язання таких класів задач:

– інформаційно-довідкові запити щодо асортименту будівельних інструментів, умов продажу, акційних пропозицій та базових правил користування сервісом;

– пошук та фільтрація товарів за назвою, категорією, характеристиками, описом тощо;

– пошук та моніторинг замовлень за ідентифікаторами й атрибутами, прив’язаними до конкретного користувача, з відображенням статусу та основних параметрів замовлення;

– навігаційні запити щодо торгових точок і сервісних центрів (адреси, графік роботи, контактні дані);

– оформлення замовлень;

– реєстрація сервісних звернень і технічних запитів користувачів.

Зазначені класи задач реалізуються в єдиному діалоговому інтерфейсі Telegram.

4.3.2.2 Призначення програми

Програма слугує інтелектуальним шлюзом між користувачем у Telegram та корпоративною інформаційною інфраструктурою. Вона реалізує RAG-архітектуру, де внутрішні облікові системи виступають джерелом достовірних даних.

Основне призначення – автоматична обробка текстових звернень, їх перетворення у внутрішні операції доступу до документно-орієнтованої бази даних та векторного сховища, а також формування релевантних відповідей без участі оператора. Сервіс працює у цілодобовому режимі, інкапсулюючи складність структури даних та протоколів доступу.

4.3.2.3 Функціональні обмеження

Функціональність програми обмежується наступними чинниками:

– робота програми передбачає наявність постійного з’єднання з мережею Інтернет;

– обмеження квот мовної моделі (кількість запитів за хвилину/добу, максимальна довжина запиту й контексту); при перевищенні квоти відповіді моделі недоступні або повертаються з помилкою/затримкою;

– залежність від доступності зовнішніх сервісів: Telegram Bot API, MongoDB, сервіс моделі Gemini, векторне сховище; відмова будь-якого з них блокує відповідні функції бота;

– система не виконує фінансових операцій, фактичного резервування товару, зміни статусів у зовнішніх ERP/CRM;

– відсутній користувацький адміністративний інтерфейс: зміна конфігурації, ключів доступу, схеми БД, параметрів моделей і RAG-правил можлива тільки адміністратором на сервері;

– обмеження формату взаємодії: у поточній реалізації опрацьовуються текстові повідомлення й callback-кнопки; інші типи вмісту (медіа, файли) не використовуються в цільових сценаріях.

4.3.3 Опис логічної структури

4.3.3.1 Алгоритм роботи програми

Алгоритм роботи програми деталізує реалізацію загальної схеми, описаної в підрозділі 4.2.2, у вигляді взаємопов’язаних процесів обробки.

Першим запускається головний диспетчер Telegram-бота, що ініціалізує об’єкт бота, підключення до БД і векторного сховища та диспетчер подій і реєструє маршрутизатори для обробки команд, довільних текстових повідомлень і callback-запитів. Команди користувача обробляються окремим командним обробником. У межах цього обробника реалізовано стандартні сценарії: запуск бота з вітальним повідомленням, ініціалізація сценарію оформлення замовлення, очищення історії запитів користувача, перегляд правил користування ботом та службових повідомлень.

Обробка довільних текстових повідомлень починається з передавання повідомлення до відповідного обробника, який визначає тип дії для поточного запиту. На основі цього результату диспетчер активує відповідний сценарій обробки, що передбачає або доступ до корпоративних даних, або формування інформаційної відповіді.

Багатокроковий сценарій оформлення замовлення викликається окремим процесом за відповідною командою чи визначеною дією. У логіці сценарію послідовно виконуються такі кроки: очікування вибору категорії товару та показ інтерактивної клавіатури з доступними категоріями з колекції товарів; очікування вибору конкретного товару з динамічної клавіатури для обраної категорії; підтвердження ціни товару з урахуванням можливої знижки; вибір

кількості товару в межах допустимого залишку на складі; остаточне підтвердження замовлення або скасування.

У разі підтвердження замовлення дані про нього зберігаються в колекції замовлень, а залишки відповідного товару оновлюються в колекції товарів. При скасуванні або помилковому введенні стани сценарію очищуються, клавіатура деактивується і користувач повертається до звичайного режиму взаємодії.

Графічно алгоритм роботи програми представлений на рисунках 4.1-4.3.

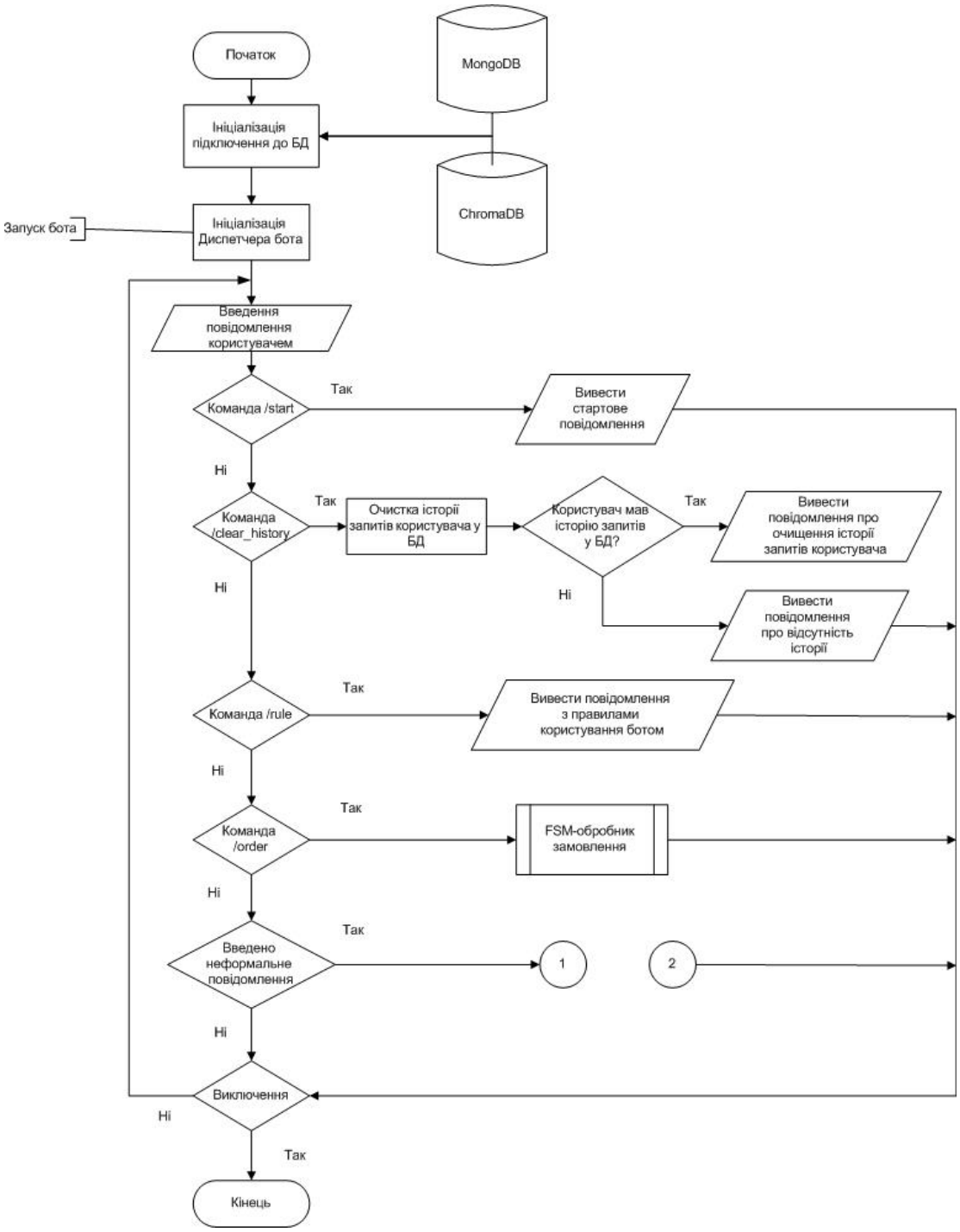


Рисунок 4.1 – Алгоритм роботи програми. Частина 1. Головний диспетчер

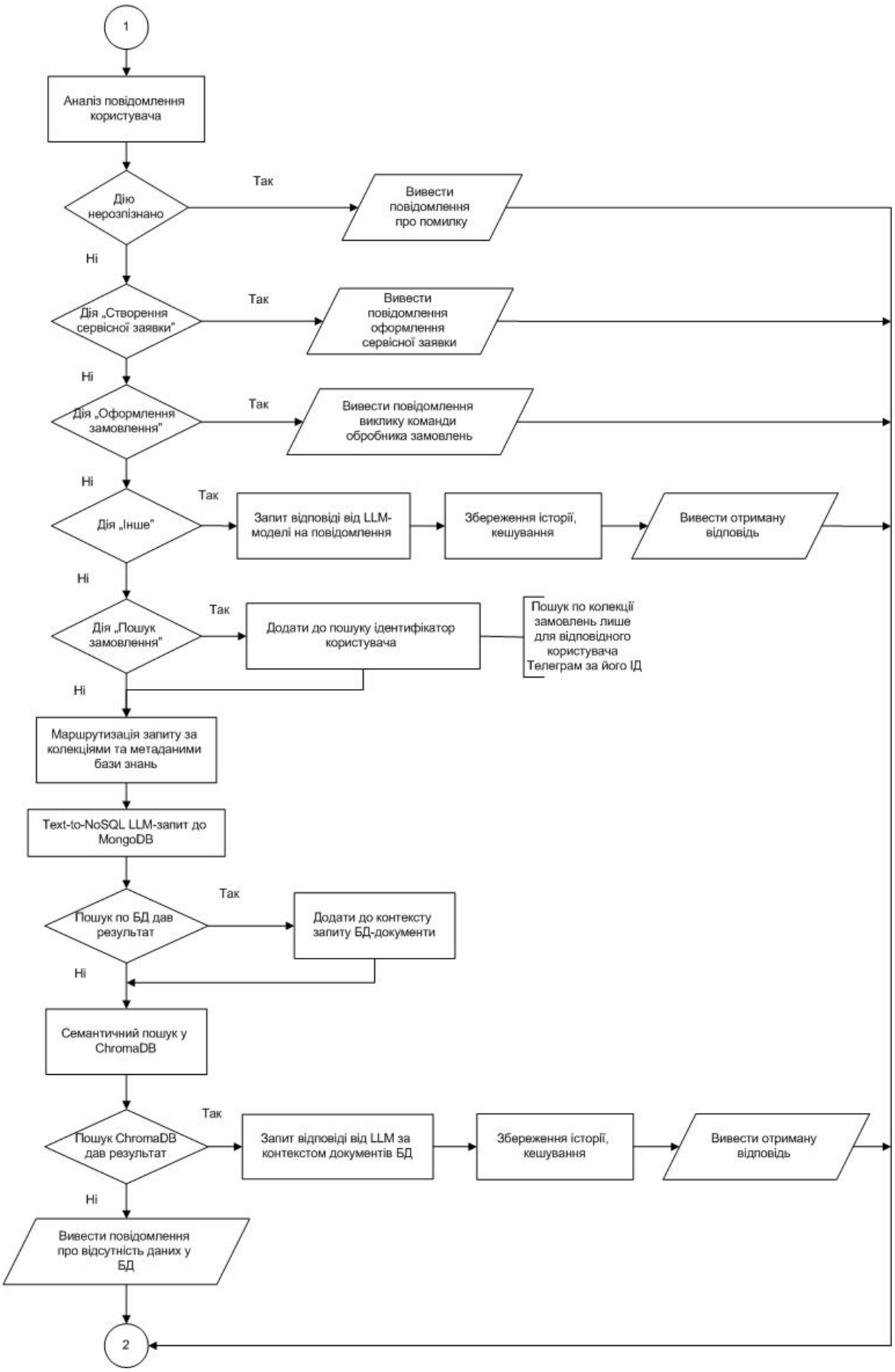


Рисунок 4.2 – Алгоритм роботи програми. Частина 2. Обробник повідомлень за AI-асистентом

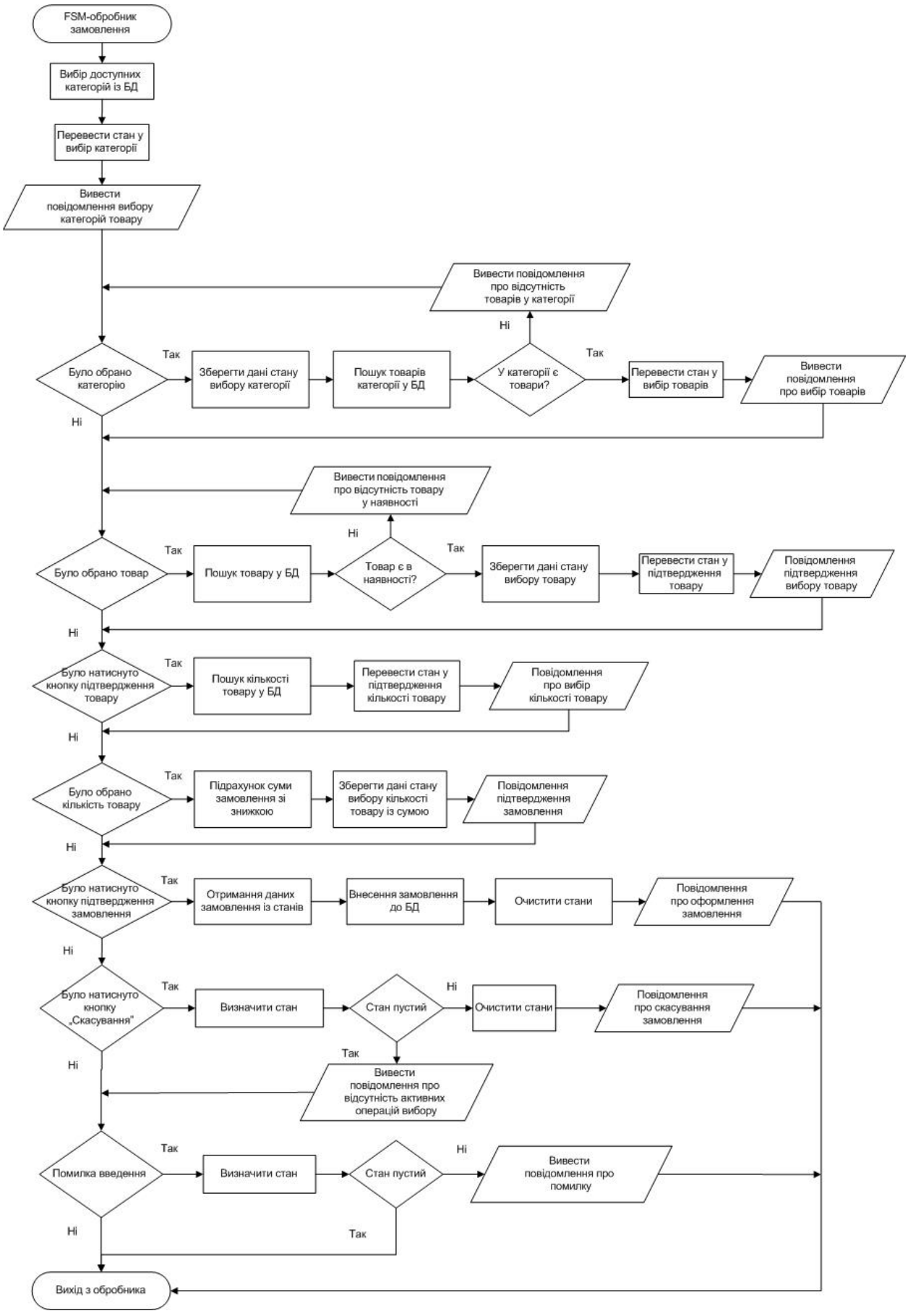


Рисунок 4.3 – Алгоритм роботи програми. Частина 3. Обробник замовлення

4.3.3.2 Використані методи

Програма використовує механізми асинхронного виконання, що дозволяє обробляти множину паралельних запитів від користувачів без блокування основного потоку та з мінімізацією затримок при зверненні до Telegram Bot API, MongoDB, ChromaDB та зовнішньої мовної моделі.

Центральним аналітичним компонентом системи довільних повідомлень є модуль на базі моделі Gemini. Він реалізує методи семантичної інтерпретації неструктурованих повідомлень, класифікації намірів користувача, побудови формалізованих запитів до бази даних та генерації контекстних відповідей. Модель використовується як універсальний інтерфейс «text-to-NoSQL», що трансформує текстові запити у формальні структури даних (JSON-об’єкти, MongoDB-пайплайни), придатні для безпечного виконання у програмному середовищі.

На вхід мовна модель отримує текст повідомлення та ідентифікатор користувача і формує у відповідь JSON-структуру, яка містить класифіковану дію та, за потреби, службові параметри для подальшої обробки. Попередньо визначені категорії дій включають:

– find_order: пошук замовлення (активація доступу до колекції замовлень);

– find_product: пошук товару (активація доступу до колекції товарів);

– product_order: оформлення замовлення (перехід до FSM-сценарію замовлення);

– create_service_request: оформлення сервісної заявки;

– find_store: пошук магазину (активація доступу до колекції торгових точок);

– other_action: інші інформаційні запити, що не потребують безпосереднього звернення до структури даних.

Для сценаріїв, які не потребують доступу до даних (інформативні, уточнювальні або загальні питання), використовується окремий метод генерації відповідей без звернення до бази даних. У цьому випадку модель працює в режимі «чистого» діалогового агента: формується промпт із текстом запиту та

коротким доменним контекстом, а відповідь генерується без залучення структурованих або векторних джерел.

Для сценаріїв доступу до бази знань застосовується на початку метод побудови MongoDB-pipeline: мовна модель на основі опису доступних полів та обмежень для конкретної колекції генерує запит у форматі масиву стадій Aggregation Framework. Для забезпечення коректності та безпеки виконання реалізовано окрему процедуру «м'якої» валідації запиту, яка фільтрує недопустимі поля, некоректні вирази та потенційно небезпечні конструкції, не змінюючи при цьому логіки дозволених стадій. У випадку, якщо очищення приводить до втрати всіх стадій, система зберігає вихідний запит як fallback-варіант, що унеможливлює аварійне завершення роботи через надмірну фільтрацію.

Формування відповіді реалізовано за принципами гібридного Retrieval-Augmented Generation. Після виконання згенерованого MongoDB-pipeline отримані структуровані дані перетворюються у скорочене JSON-подання з лімітованим обсягом. Паралельно окремий метод здійснює семантичний пошук у векторному індексі сховища. У формуванні відповіді ці два джерела – результати запиту до MongoDB та семантично близькі фрагменти з векторного індексу – об'єднуються в єдиний контекст, що подається на вхід моделі Gemini. Якщо структурований запит повертає порожній або нерелевантний результат, логіка обробки фактично зводиться до векторного пошуку та генерації відповіді на основі знайдених документів і текстового опису предметної області.

Для діалогових сценаріїв із фіксованою послідовністю кроків застосовано методи керування станами на основі скінченного автомата (FSM). Сценарій моделюється як послідовність станів класу OrderStates, між якими користувач переходить за допомогою інтерактивних inline-кнопок. Перехід між станами здійснюється через обробку callback-подій, що надходять від клавіатури, а FSM-контекст зберігається в асинхронному сховищі станів.

Методи побудови інтерфейсу взаємодії з користувачем базуються на динамічній генерації inline-клавіатур і пагінації результатів. При відображенні

списків товарів або варіантів відповіді формуються службові callback-дані, які дозволяють відстежувати обрані користувачем елементи та, за потреби, перемикатися між «сторінками» результатів. Це забезпечує зручну навігацію без перевантаження чат-інтерфейсу довгими текстовими повідомленнями.

Підтримка контекстної цілісності діалогу реалізується через збереження історії повідомлень у відповідній колекції бази даних. Історія використовується як додаткове джерело контексту для мовної моделі, а також як основа для кешування окремих відповідей, що знижує затримку повторних звернень. Додатково застосовуються методи валідації введених даних та обробки виняткових ситуацій, що забезпечує коректність виконання операцій і формування зрозумілих повідомлень про помилки для користувача.

4.3.3.3 Структура програми та зв’язок між компонентами

Програмне забезпечення реалізоване як монолітний серверний застосунок на Python з модульною структурою, у якій окремі компоненти виконують спеціалізовані функції в єдиному процесі обробки запитів Telegram-бота. Зв’язок між компонентами має ієрархічний характер.

Головний модуль BudStore.py ініціалізує об’єкт бота та диспетчер подій, підключає конфігурацію, реєструє маршрутизатори обробників і запускає отримання оновлень у режимі polling. Через нього всі Telegram-оновлення потрапляють у систему та передаються до відповідних обробників.

Модуль handlers.py відповідає за обробку текстових повідомлень і частини командної логіки. Він приймає вхідні повідомлення, викликає модуль аналізу запиту, готує службові параметри для звернення до бази даних і формує кінцеві відповіді та повідомлення з inline-кнопками для користувача.

Модуль ai_assistant.py інкапсулює взаємодію з мовною моделлю та базою знань. У межах цього модуля реалізовано формування запитів до моделі, обробку відповідей та побудову службових структур даних, які потім використовують обробники для звернення до MongoDB та векторного індексу.

Модуль db.py забезпечує з’єднання з MongoDB і ChromaDB та доступ до основних колекцій. Інші компоненти працюють із даними через цей модуль, що відокремлює прикладну логіку від конкретних механізмів взаємодії з БД.

Модуль order_fsm.py містить опис станів сценарію оформлення замовлення та відповідні callback-обробники. Він зареєстрований як окремий маршрутизатор диспетчера та отримує лише ті оновлення, які стосуються активних FSM-сценаріїв. Усі переходи між станами, перевірки введених даних та формування проміжних повідомлень реалізовано в цьому модулі.

Модуль config.py зберігає параметри конфігурації – токен бота, налаштування підключення до MongoDB, параметри мовної моделі тощо – які імпортуються іншими модулями, що дає змогу централізовано змінювати зовнішні налаштування без модифікації логіки.

4.3.3.4 Взаємодія з іншими програмами

У програмному засобі передбачено зв’язок із зовнішнім сервісом мовної моделі Google Gemini, доступ до якої здійснюється через офіційний REST-API з використанням API-token для авторизації.

Програмний засіб не взаємодіє з іншими зовнішніми програмними компонентами та сервісами, окрім вищезазначеного та тих що описані у 4.3.1.2.

4.3.4 Використовувані технічні засоби

Для коректної роботи програми необхідні мінімально рекомендовані характеристики технічних засобів, що представляють собою сервер, або засоби віртуалізації (віртуальні машини) з наступними підтримуваними характеристиками (або аналогічними обчислювальними можливостями):

– процесор архітектури x64 з не менше ніж 6 ядер з тактовою частотою 2.6 ГГц або вище;

– оперативна пам'ять мінімум 16 Гб RAM;

– дисковий простір мінімум 60 Гб вільного місця на диску для розміщення необхідних файлів і даних (включаючи ОС та середовища виконання);

– стабільне підключення до Інтернету з пропускною здатністю не менше 1 Гбіт/с.

4.3.5 Виклик та завантаження

Програмний засіб запускається через інтерпретатор Python з командного рядка, або середовища розробки (IDLE) за умови використання попередньо створеного віртуального середовища та встановлення необхідних бібліотек, вхідна точка – модуль BudStore.py. Програма не потребує специфічних параметрів під час запуску.

Перед запуском мають бути запущені служби MongoDB та ChromaDB.

Зупинка роботи здійснюється шляхом завершення процесу інтерпретатора або зупинки запуску в середовищі розробки.

4.3.6 Вхідні дані

4.3.6.1 Характеристика вхідних даних

Вхідними даними є користувацькі повідомлення, що надходять через Telegram Bot API у вигляді JSON-об’єктів (update, message, callback_query) і містять текстові поля та службові параметри. На стороні програми ці дані обробляються як текстові рядки та структури.

Повідомлення поділяються на:

– структуровані – команди, що починаються з символу «/»;

– неструктурований довільний текст;

– callback-запити – дані від натискання кнопок inline-клавіатури, що передаються як рядкові значення усередині JSON-поля callback_query.data.

До вхідних даних, пов’язаних із MongoDB, належать документи колекцій orders, products, stores, service_requests, sessions, для ChromaDB – колекція векторів product_rag.

4.3.6.2 Організація даних

Програма не передбачає введення користувачем даних із файлів або автоматизоване імпортування інформації з локальних джерел.

Внутрішні службові дані, необхідні для роботи програми, зберігаються у базі даних MongoDB та векторному сховищі, доступ до якої здійснюється у режимі виконання логіки програми без прямої участі користувача.

Логічна структура бази даних представлена на рисунку 4.4.

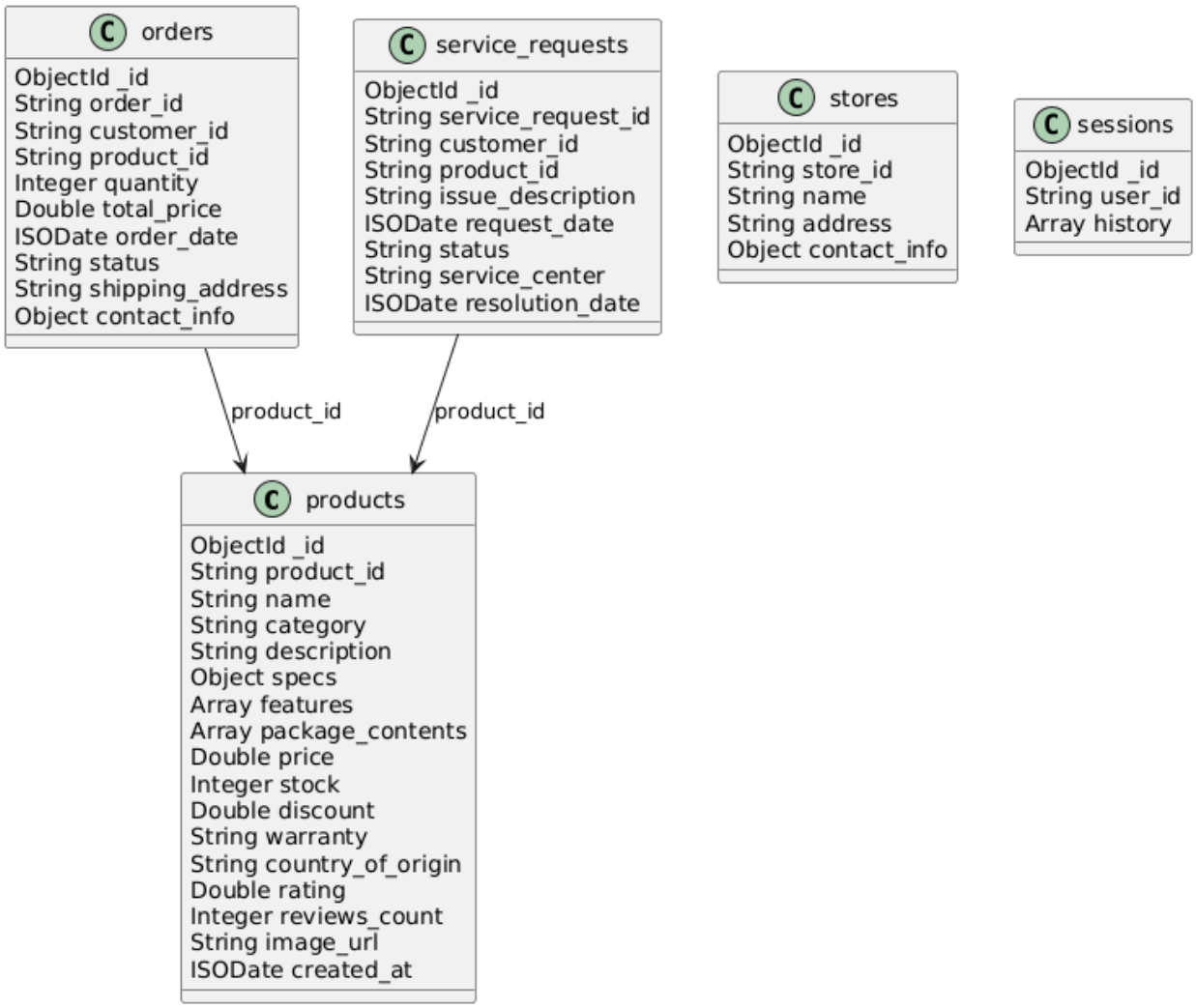


Рисунок 4.4 – Структура та взаємозв’язок колекцій MongoDB

Логічна структура векторного сховища представлена на рисунку 4.5.

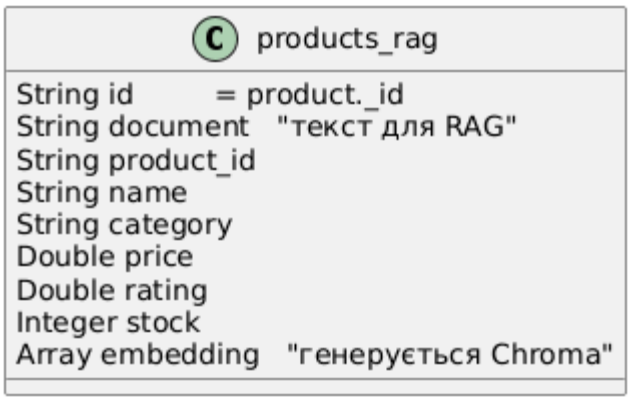


Рисунок 4.5 – Структура колекції product_rag

4.3.6.3 Попередня підготовка

Текстові повідомлення обробляються із застосуванням внутрішніх механізмів нормалізації, таких як видалення зайвих пробілів, приведення до нижнього регістру, фільтрація службових символів тощо. Програма самостійно здійснює валідацію повідомлення та, у разі необхідності, генерує уточнюючі запити для отримання повної інформації. Усі перетворення виконуються автоматично в межах логіки обробників.

Для векторного сховища підготовка даних виконується окремою службовою процедурою синхронізації: документ товару формується у вигляді цілісного текстового опису, після чого передається до колекції products_rag, де автоматично створюється векторне представлення локальною моделлю all-MiniLM-L6-v2 та зв’язується з документом БД за ідентифікатором та метаданими.

4.3.6.4 Формат вхідних даних

Основний формат користувацьких текстових даних – рядок, який передається в полі message.text JSON-об’єкта Telegram Bot API.

Структуровані запити надходять як команди Telegram і обробляються окремими обробниками. У програмі використано такі команди:

– /start – вітальне повідомлення та короткий опис можливостей бота;

– /order – активація FSM-сценарію оформлення замовлення;

– /clear_history – очищення історії повідомлень користувача;

– /rule – інструкція щодо формулювання запитів.

Callback-дані передаються всередині JSON-поля callback_query та мають рядковий формат з префіксами для позначення типу дії для обраних сценаріїв замовлення та пагінації:

– category:… – вибір категорії;

– product:… – вибір товару;

– confirm_price:… – підтвердження ціни;

– quantity:… – вибір кількості;

– place_order:… – підтвердження замовлення;

– page:…. – вибір сторінки.

Документи MongoDB мають формат BSON, що є бінарним представленням структур даних «ключ-значення» з підтримкою скалярних типів (рядки, числа) та вкладених структур (масиви, об’єкти). Для часових міток застосовується тип ISODate, числові значення зберігаються як Integer або Double залежно від параметра.

4.3.6.5 Спосіб кодування

Передача даних між користувачем і серверами Telegram здійснюється за протоколом шифрування MTProto. Взаємодія між програмним засобом (ботом) і Telegram Bot API виконується у вигляді JSON-об’єктів, що передаються через HTTPS, всі текстові поля кодуються у форматі UTF-8.

Програма не застосовує додаткового кодування, шифрування або форматування повідомлень, за винятком тих випадків, коли це прямо передбачено протоколами та у процесі виконання зовнішніх API.

4.3.7 Вихідні дані

При формуванні вихідних даних застосовуються ті самі формати та кодування, що й для вхідних.

Основними вихідними даними для користувача є текстові повідомлення, що надсилаються в чат Telegram. Вони можуть містити:

– інформаційні відповіді на довільні запити;

– списки товарів, торгових точок, клієнтської інформації, сформовані за результатами запитів до бази даних;

– проміжні та фінальні повідомлення FSM-сценарію оформлення замовлення.

За потреби разом із текстом передаються inline-клавіатури, описані у відповідних полях відповіді Telegram.

Вихідними даними на рівні MongoDB є документи, що створюються або змінюються в результаті обробки запитів:

– у колекції orders – нові записи про оформлені замовлення або оновлені статуси існуючих замовлень;

– у колекції service_requests – записи про сервісні звернення;

– у колекції products – оновлені залишки товарів;

– у колекції sessions – додані елементи історії діалогу користувача (до 20 останніх пар «запит-відповідь» у контексті одного користувача).

5 ЕКСПЕРИМЕНТАЛЬНИЙ РОЗДІЛ

5.1 Постановка мети та завдання експериментів для перевірки

Метою експериментальних досліджень є перевірка працездатності, ефективності та коректності роботи запропонованої архітектури системи автоматизованого обслуговування клієнтів компанії-постачальника будівельного обладнання в умовах змінного навантаження. Експеримент має підтвердити, що розроблена архітектура забезпечує прийнятний час реакції, стабільність роботи та якість відповідей на запити користувачів згідно з вимогами, сформульованими у попередніх розділах.

Для досягнення поставленої мети розв’язуються такі основні завдання:

– експериментально оцінити параметри обслуговування для всіх ключових компонентів системи – Telegram-бот, сервіс великої мовної моделі, гібридне сховище знань, та отримати середні часи обслуговування, які використовуються в теоретичній моделі;

– перевірити, що при заданих інтенсивностях вхідного потоку та параметрах обслуговування, фактичні коефіцієнти завантаження вузлів не призводять до втрати стійкості та відповідають обмеженням, закладеним у моделі;

– експериментально підтвердити, що агрегований час реакції системи на запит користувача не перевищує нормативного значення у 10 секунд та що залежність цього часу від інтенсивності навантаження до 10 заявок на секунда узгоджується з розрахунковими оцінками;

– оцінити якість відповідей генеративного модуля та RAG-конвеєра на репрезентативному наборі запитів за векторними метриками та метриками RAGAS, зіставивши отримані значення з очікуваними порогами – повинно бути покращення відповіді на 50% після застосування RAG у порівнянні з «наївною» відповіддю LLM;

– перевірити коректність виконання типового FSM-сценарію оформлення замовлення та інших критичних бізнес-процесів, зафіксувати частку успішних

проходжень і підтвердити відповідність логіки сценарію вимогам постановки задачі.

5.2 Вимоги до експериментального стенду та методика тестування

Перед початком експерименту слід задати вимоги до середовища виконання. Усі експерименти проводяться в однаковому апаратному та програмному середовищі з фіксованими типом і частотою процесора, обсягом оперативної пам’яті, типом і конфігурація дискової підсистеми, мережевим адаптером, а також версіями операційної системи, MongoDB, ChromaDB, сервісу LLM, Python та використаних бібліотек. Ці параметри зведені в таблицю 5.1 експериментального стенду і не підлягають змінам на час проведення всього експерименту та його частин.

Таблиця 5.1 – Апаратне забезпечення експериментального стенду

Тип	Характеристика
Процесор	6-ядерний CPU із тактовою частотою 3.5 ГГц
Оперативна пам’ять	16 ГБ
Накопичувач	SSD 512 ГБ
Операційна система	Windows 10
MongoDB	Версія 8.0.4
ChromaDB	Версія 1.3.4
LLM	Gemini 2.5 Pro/Flash Free Tier
Telegram-клієнт	Веб-клієнт Telegram, Pyrogram (імітаційна)
Мережеве підключення	1 Гб/с
Python	Версія 3.11

Облік результатів проводиться за допомогою журналів у форматі JSON. Для кожного окремого експерименту чи вибірки фіксуються за потреби:

– унікальний ідентифікатор;

– для часових вимірів: мітки часу надходження й завершення обробки, можливі проміжні часові відмітки (початок звернення до БД, початок виклику LLM тощо);

– коди успішності операцій;

– опис сценарію;

– мітки завершення сценарію з кінцевими результатами окремого виміру;

– службова інформація: номер клієнта, номер повтору, параметри навантаження тощо.

Журнали обліку виступають єдиним джерелом даних для подальших обчислень середніх значень, інтенсивностей, перцентилів, метрик якості тощо.

Для гібридного сховища знань застосовуються дві групи тестів. По-перше, виконується навантаження у програмному середовищі асинхронними операціями для фіксованої кількості паралельності (асинхронних воркерів підключення) векторного пошуку у визначеній колекції та типових доменних запитів, при цьому для кожного запиту вимірюється затримка відповіді як різниця між початком програмного таймера перед запуском операції та кінцем виконання таймера за отриманою успішною відповіддю. Сукупність N успішних вимірений значень затримки Si з JSON-журналів агрегується у показники середнього часу обслуговування:

$$\bar{\mathrm{S}} = \frac{1}{\mathrm{N}} \sum_{i=1}^{N} \mathrm{Si}$$

Додатково на вимірах Si сукупності N обчислюються перцентилі $p50$, $p95$, $p99$ які інтерпретуються як медіанна латентність, «майже найгірший» і «екстремальний» час відповіді.

По-друге, аналогічно формується змішане навантаження на MongoDB, де кожен асинхронний воркер паралельно виконує читання, запис і агрегаційні конвеєри в рівних частках 1 до 3 для репрезентативного режиму доступу до БД. Для кожного типу операцій окремо вимірюються успішні вибірки затримок програмними таймерами та розраховуються середні значення та перцентилі.

Інтегральний показник часу обслуговування компонента БД сховища розраховується як середнє арифметичне часу обслуговування читань, записів і агрегацій, та вважається як показник часу типового «візиту» до компоненту БД:

$$\bar{S}_{\text{Mongo}} = \frac{\bar{S}_{\text{read}} + \bar{S}_{\text{write}} + \bar{S}_{\text{agg}}}{3}$$

Оскільки у реальному сценарії одна логічна «заявка до бази знань» вимагає послідовного доступу як до MongoDB, так і векторного пошуку, вузол бази знань в моделі СМО описується агрегованим часом обслуговування, який обчислюється як сума послідовних операцій із середніми арифметичними часу обслуговування:

$$\bar{S}_{\text{KB}} = \bar{S}_{\text{Mongo}} + \bar{S}_{\text{Chroma}},$$

а інтенсивність обслуговування вузла визначається:

$$\mu = 1 / \bar{S}_{\text{KB}}.$$

Вузол LLM тестується програмним скриптом, який генерує серію паралельних одночасних запитів до сервісу великої мовної моделі з фіксованими параметрами (тип моделі, параметри генерації) при різних режимах навантаження. Для кожного запиту фіксується програмними таймерами час надсилання до LLM-сервісу та час отримання відповіді. Різниця між цими двома мітками розглядається як латентність LLM-вузла. За вибіркою та кількістю тестувань розраховується середній час відповіді S_{LLM}, а інтенсивність – як обернена величина до середнього часу.

Для вимірювання «чистого» часу обслуговування Телеграм-бота використовується бібліотека Pyrogram. Створюється задана кількість клієнтів, де кожен клієнт відповідає одному логічному користувачу та окремому каналу спілкування з ботом. Скрипт навантаження відправляє команди до бота в Telegram, причому модулі сховища знань та LLM спеціально відключаються, щоб зафіксувати лише внутрішній час роботи самого бота (парсинг команд, FSM-логіка, формування відповіді). З журналів для кожної команди фіксується час надсилання, час отримання відповіді користувачем, за якими вираховується час перебування заявки на обслуговувані, а також оцінка мережної затримки. На

основі отриманої вибірки «очищених» часів обслуговування розраховується середній час обслуговування Telegram-вузла та відповідна інтенсивність обслуговування.

Після отримання експериментальних оцінок інтенсивностей обслуговування для трьох основних вузлів, вони використовуються як вхідні параметри теоретичної моделі мережі СМО. Для підтвердження теоретичної моделі, застосовується методика зазначена вище – навантаження створюється віртуальними користувачами Pyrogram, при цьому вся система функціонує у штатному режимі з усіма включеними модулями.

Зовнішня інтенсивність λ_0 задається сценаріями навантаження і додатково перевіряється за логами – оцінюється за журналами як середнє число запитів за одиницю часу. Внутрішні інтенсивності потоків λ_1, λ_2, λ_3 у вузлах бота, LLM та бази знань обчислюються за балансовими рівняннями. Далі для кожного вузла i із параметрами λi, μi, mi розраховуються коефіцієнти завантаження та перевіряється стійкість $\rho i < 1$. Середній час очікування в черзі та час перебування заявки у вузлі обчислюються за класичними формулами моделі M/M/mi, а загальний теоретичний час реакції системи T_{resp} згідно визначеної суми за коефіцієнтами відвідувань.

Паралельно з теоретичними розрахунками, для кожного сценарію навантаження на основі логів Pyrogram для всієї системи обчислюється експериментальний середній час реакції: з різниці між мітками «вхідний запит» та «відповідь бота» формується вибірка повних часів у системі, по якій обчислюється середнє, та за середнім – робиться висновок по системі.

Для оцінки якості відповідей як «чистої» LLM (без доступу до сховища), так і системи з RAG-конвеєром використовується єдина експериментальна схема, що поєднує ембедінг-базовані метрики та метрики бібліотеки RAGAS. Різниця між двома варіантами полягає: у LLM-only-сценарії відсутній явний набір добутих документів, тому використовуються тільки ті показники, які не потребують документного контексту, тоді як для RAG-конвеєра застосовується повний набір векторних і RAGAS-метрик.

Для кожного тестового кейсу задається запит користувача, еталонна відповідь y (зафіксована для цього кейсу) та фактична відповідь моделі $\hat{y}$. Для еталонної та фактичної відповідей обчислюються ембедінги e_{gold} та e_{resp}, після чого як базову метричну оцінку використовують косинусну схожість:

$$sim(\hat{y}, y) = \frac{e_{resp} \cdot e_{gold}}{||e_{resp}|| \cdot ||e_{gold}|| + \varepsilon},$$

де ε – мала константа для запобігання діленню на нуль. Для кожного сценарію ця схожість обчислюється для всіх тестових прикладів.

До тих самих пар «запит – фактична відповідь – еталон» додатково застосовується RAGAS. У випадку «чистої» LLM обмежуються метриками, які не потребують явного переліку документів, а саме ResponseRelevancy (answer_relevancy), що характеризує відповідність відповіді формулюванню запиту, та FactualCorrectness (factual_correctness), що оцінює повноту й точність покриття еталону у F1-подібній шкалі від 0 до 1.

У випадку RAG-конвеєра, окрім наведеного вище, для кожного тестового прикладу додатково фіксується множина «очікуваних» документів, фактично витягнуті документи, відповідь RAG-системи та еталонна відповідь. На цій основі обчислюється кілька ембедінг-базованих показників за косинусною схожістю (всі дані при цьому переведені у векторне представлення): якість ретривера (retrieval_quality) – частка очікуваних документів, які потрапили до результатів), опертя відповіді на контекст (groundedness, середня схожість відповіді з витягнутими документами), узагальнений штраф за галюцинації (hallucination_penalty) та скалярна оцінка (answer_quality) відповідності до еталона. Для RAG-сценарію RAGAS використовується в повній конфігурації: додатково до answer_relevancy та factual_correctness обчислюються ContextRecall (context_recall), що характеризує повноту використання релевантного контексту, та Faithfulness (faithfulness), що оцінює узгодженість відповіді з контекстом і відсутність галюцинацій.

Перевірка безпеки агрегації до MongoDB виконується на основі тестового набору запитів, для яких система генерує агрегувальні пайплайни, з яких

витягуються всі операції та оператори агрегації. Далі виконується перевірка на наявність небезпечних або заборонених операторів типу $where, $function, $accumulator, $out, $merge та некоректних конструкцій, які можуть спричинити ін’єкції, неконтрольований вивід даних або обхід механізмів безпеки. Для кожного пайплайна у журналі фіксується статус «безпечний/небезпечний», а також причина відхилення (якщо така є). За всією вибіркою тестових кейсів обчислюється кількість безпечних запитів.

5.3 Сценарії та параметри експериментальної перевірки

Для бази знань виділено окремі сценарії навантаження для операцій MongoDB та для векторного пошуку:

– для MongoDB використовується багатопотоковий сценарій з асинхронними воркерами (користувачі/підключення); основна серія містить п’ять рівнів навантаження з кількістю воркерів 10, 25, 50, 75 і 100; кожен воркер виконує по 1000 операцій (читання, запис, агрегації у рівних частках); кожен рівень навантаження повторюється 5 разів для усереднення; перед тестуванням виконується один запуск для "прогрівки" – стабілізація кешів, підключень тощо;

– для векторного пошуку навантаження моделюється аналогічними рівнями паралелізму для векторного пошуку: 10, 25, 50, 75 і 100 одночасних клієнтів; кожен клієнт виконує по 1000 запитів; кожен рівень навантаження повторюється 5 разів.

Для вузла LLM застосовується сценарій вимірювання часу обслуговування при постійному навантаженні. Одна серія вимірювань складається з 10 послідовних запитів. Виконується 10 таких серій, загальний обсяг вибірки – 100 запитів.

Для вузла «чистого» Telegram використовується Pyrogram згідно методик тестування. Задається рівень паралелізму, що відповідає 10 клієнтів, кожен із яких надсилає по 10 команд. Для обмеження миттєвого навантаження використовується обмежувач запитів зі швидкістю до 20 запитів за секунду. Тест повторюється 10 разів.

Сценарій інтегрального навантаження всієї мережі СМО проводиться за допомогою Pyrogram. Зовнішнє навантаження задається у вигляді трьох рівнів інтенсивності потоку запитів за секунду: низький – 3, середній – 6, високий – 10. Для кожного з трьох рівнів система навантажується протягом 1 хвилини. Кожен рівень повторюється 3 рази, між прогонами робиться пауза.

Для оцінки LLM-only та RAG-конвеєра використовуються симетричні набори шаблонів запитів, кожен обсягом 30 кейсів:

– LLM-only: оцінюється 30 шаблонів запитів, охоплюючи класи доменних задач (запитання-відповіді по каталогам та асортименту, рекомендації, узагальнення тощо); вимірюється косинусна схожість та RAGAS-метрики, що не вимагають документального контексту –answer_relevancy, factual_correctness; на основі вибірок розраховується середні значення;

– RAG-конвеєр: оцінюється 30 шаблонів, прив'язаних до конкретних документів; обчислюється повний набір ембедінг- та RAGAS-метрик, включаючи ContextRecall та Faithfulness; на основі вибірок розраховується середні значення, порівнюються з результатами «чистої» LLM ;

Для перевірки коректності FSM-сценарію, організовується 30 незалежних прогонів логіки автомату. У кожному прогоні тестовий клієнт послідовно виконує всі визначені дії, а система логування фіксує, в який момент активується кожен стан, які переходи спрацьовують, і чи досягнуто фінальний стан без помилок. На основі вибірки оцінюється загальна частка успішно завершених сценаріїв та частота відмов на кожному з ключових станів або переходів.

5.4 Проведення тестування та аналіз результатів

5.4.1 Продуктивність і швидкодія системи

У цьому підрозділі наведені експериментальні показники продуктивності вузлів системи – гібридного сховища знань, вузла LLM, вузла Telegram-бота, та всієї мережі СМО, а також результати розрахунку теоретичного часу реакції з урахуванням багатоканальної моделі M/M/n для кожного вузла. Експеримент полягав у підтвердженні адекватності моделі в умовах реальної експлуатації та

оцінюванні її здатності прогнозувати час реакції системи за різних рівнів навантаження.

Для MongoDB проводились тести, згідно методик і сценарію, при 10, 25, 50, 75 та 100 паралельних користувачів у режимах читання, запису та агрегації. Для векторного пошуку використано ті самі рівні паралельності. Нижче у таблиці 5.2 та рисунках 5.1, 5.2 наведені відповідно: середні затримки окремих типів операцій та усереднені показники по сховищу; перцентилі затримки векторного пошуку; перцентилі затримки типових операцій для MongoDB.

Таблиця 5.2 – Усереднені показники сховищ за різної кількості клієнтів

Навантаження	Кількість операцій на 1 клієнта	MongoDB read avg, мс	MongoDB write avg, мс	MongoDB agg avg, мс	Chroma vector avg, мс
10 клієнтів	1000	1.1459	1.0291	66.4514	115.8817
25 клієнтів		17.5209	18.5565	109.9765	288.7602
50 клієнтів		45.7495	39.1139	198.3879	695.0678
75 клієнтів		72.3477	71.0592	304.8359	1078.8226
100 клієнтів		133.9255	129.5130	384.3132	1543.8215

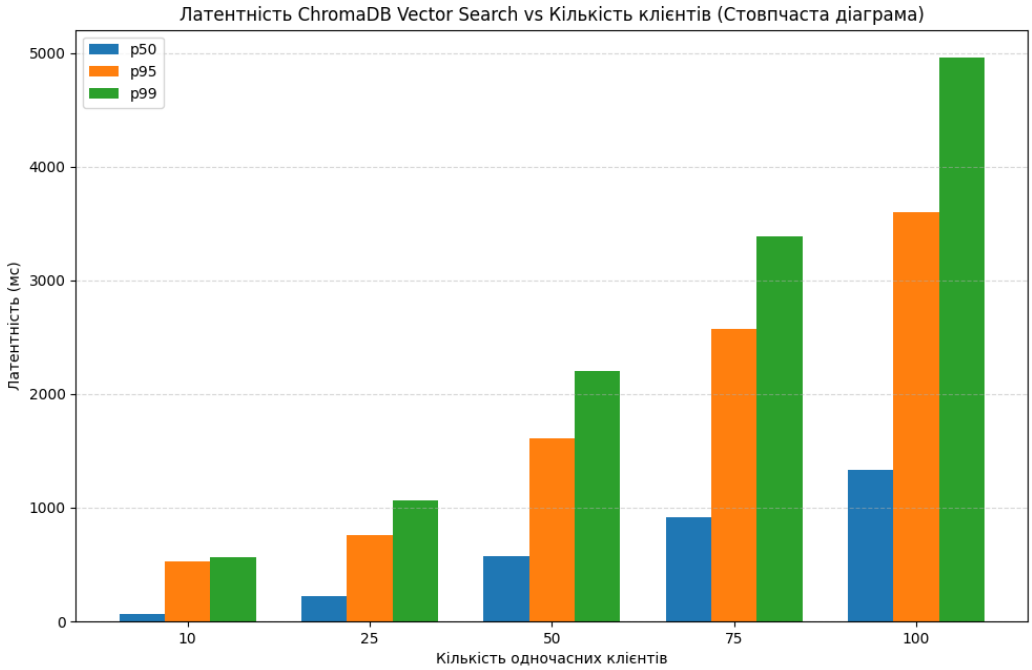


Рисунок 5.1 – Залежність затримки векторного пошуку від кількості клієнтів

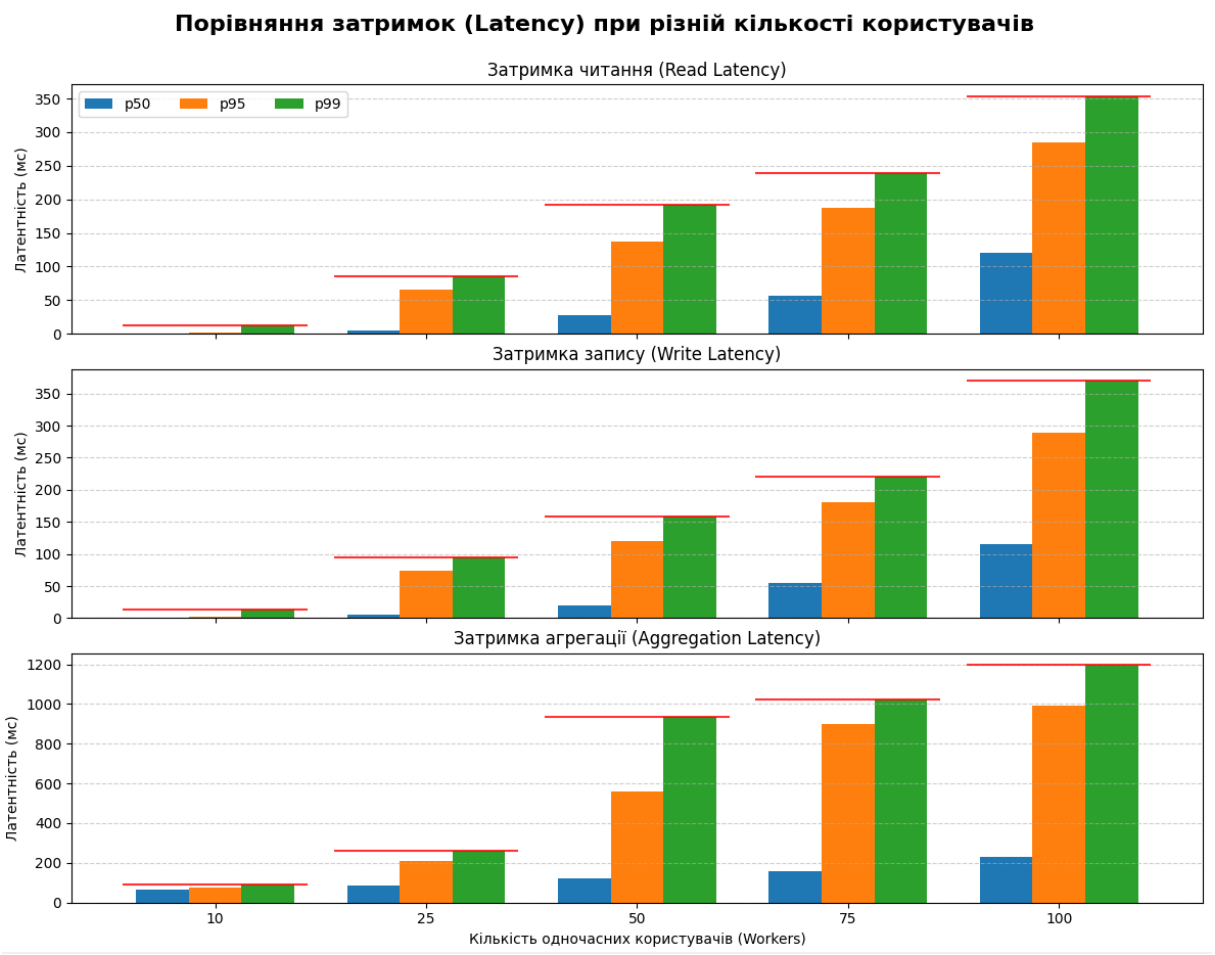


Рисунок 5.2 – Залежність затримки типових операцій БД від клієнтів

Для теоретичної моделі береться режим із 10 воркерами/клієнтами як репрезентативний режим роботи без насичення. Один «візит» до вузла бази знань розглядається як:

$$\bar{S}_{\text{Mongo}} = \frac{\bar{S}_{\text{read}} + \bar{S}_{\text{write}} + \bar{S}_{\text{agg}}}{3} = 22.88 \text{ мс},$$

$$\bar{S}_{\text{КВ}} = \frac{\bar{S}_{\text{Mongo}} + \bar{S}_{\text{Chroma}}}{1000} = 0.1388 \text{ с}$$

Інтенсивність обслуговування при такому режимі роботи складає:

$$\mu_{\text{КВ}} = \frac{1}{0.1388} = 7.2 \quad 1/\text{с}$$

Тестування вузла LLM відбувалось у межах заданого тестового сценарію без додаткових затримок RAG-конвеєра. Середній час відповіді за 100 запитів та інтенсивність обслуговування становили:

$$S_{LLM} = 9.07 \text{ с}, \qquad \mu_{LLM} = 0.1103 \ 1/\text{с}$$

На час тестування загальна квота безкоштовного рівня для моделей класу Gemini 2.5 становила 50 запити на хвилину, візьмемо це значення як теоретично можливу межу паралельності обробки, тобто кількість каналів обробки становить, відповідно, 50. Варто зазначити, що безкоштовний тариф задає сильні обмеження на використання мовної моделі, платні тарифи надають кращі можливості та час обробки, крім того дозволяють до 1000 або 5000 паралельних каналів обчислення (у залежності від витраченої суми). Тому репрезентативність результатів надають лише мінімізовану апроксимацію повних можливостей сервісу.

Тестування вузла «чистого» Telegram-бота за визначеним сценарієм дало наступні результати:

$$S_{bot} \approx 118{,}8 \text{ мс} = 0{,}1188 \text{ с}, \qquad \mu_{bot} = 8{,}42 \ 1/\text{с}$$

Збірна таблиця 5.3 результатів тестування та параметрів кожного окремого вузла розміщена нижче. За канали обслуговування n_i СМО враховувались припущення щодо обраних сценаріїв паралелізму. Коефіцієнти відвідування вузлів задані відповідно балансовим рівнянням для обраної моделі.

Таблиця 5.3 – Параметри кожного вузла мережі СМО

Вузол	α_i	S_i, с	μ_i, 1/с	n_i
Telegram-бот	2,5	0.1188	8.42	10
LLM	0.56	9.07	0.1103	50
База знань	0.59	0.1388	7.21	10

Під час проведення експерименту навантаження мережі СМО клієнтами Pyrogram, згідно логів, було отримано наступні параметри інтенсивності зовнішнього навантаження та середній час перебування заявки у системі:

– низький рівень – $\lambda_0 = 3.26$ з/с, $T_{resp}^{exp} = 2.028$ с;

– середній рівень – $\lambda_0 = 6.67$ з/с, $T_{resp}^{exp} = 4.761$ с;

– високий рівень – $\lambda_0 = 9.54$ з/с, $T_{resp}^{exp} = 7.953$ с.

Після отримання експериментальних значень зовнішньої інтенсивності, можна розрахувати стаціонарні інтенсивності λ_i кожного вузла мережі, розраховані згідно балансових рівнянь (2.1), далі – провести перевірку коефіцієнтів завантаження ρ_i (2.2) для визначення стійкості стану системи. У таблиці 5.4 зображено залежність завантаженості кожного вузла.

Таблиця 5.4 – Залежність завантаженості вузла від інтенсивності навантаження

		Telegram-бот		LLM		База знань	
Рівень навантаження	λ_0	λ_1	ρ_1	λ_2	ρ_2	λ_3	ρ_3
Низьке	3.26	8.15	0.10	1.83	0.33	1.92	0.027
Середнє	6.67	16.68	0.20	3.73	0.68	3.94	0.055
Високе	9.54	23.85	0.28	5.34	0.97	5.63	0.078

Згідно таблиці можна побачити що вузли бота і бази знань залишаються у «комфортній» зоні $\rho < 0.3$, тоді як вузол LLM працює майже на межі при високому навантаженню, що є очікуваним результатом з огляду на обмеження безкоштовного рівня API-клієнта мовної моделі.

Далі, згідно отриманих значень завантаженості вузлів, проведено розрахунок середнього часу перебування заявки у заданому вузлі мережі T_i з урахуванням часу очікування черги за змінної інтенсивності навантаження за формулою (2.3), та загальний час перебування заявки у мережі СМО за коефіцієнтами відвідувань (2.4), результати у таблиці 5.5.

Таблиця 5.5 – Час перебування заявки у вузлах та мережі загалом

		Telegram-бот	LLM	База знань	Мережа
Рівень навантаження	λ_0	T_{bot}	T_{LLM}	T_{KB}	T_{resp}
Низьке	3.26	0.1187	9.066	0.1386	5.46
Середнє	6.67	0.1187	9.069	0.1386	5.46
Високе	9.54	0.1187	13.362	0.1386	7.86

Як можна побачити при піковому навантажені фактичне і теоретичне значення майже співпадають і відповідають вимогам нормативу: за інтенсивності 10 одночасних заявок від користувачів надавати відповідь за час до 10 секунд.

Різниця між теоретичними та експериментальними показниками у низькому й середньому режимах пояснюється тим, як саме будувалась модель і сценарій навантаження. По-перше, ефективна інтенсивність обслуговування LLM та коефіцієнти відвідувань α_i фактично калібровані під піковий режим, де майже кожна заявка проходить повний ланцюжок, а вузол LLM працює майже на межі завантаження. У такій конфігурації модель дає коректний час реакції для високого навантаження. По-друге, при низькому та середньому навантаженні реальні діалоги в системі використовують LLM і БЗ рідше, ніж закладено, а сам LLM працює далеко від насичення, тому його реальна ефективна швидкодія вища, ніж та, що взята з пікового режиму. У результаті модель дає завищений (консервативний) теоретичний T_{resp}, тоді як експериментальні значення показують фактичний запас продуктивності системи.

5.4.2 Коректність роботи RAG-конвеєра та FSM-сценарію

Було проведено оцінку якості відповідей «наївної» LLM (без доступу до сховища) та системи з RAG-конвеєром на задачах доступу до каталогу товарів.

Згідно методології було використано для кожного кейсу:

– векторні метрики;

– метрики типу RAGAS;

Для «наївної» відповіді LLM вимірювалась косинусна схожість між embedding-представленнями відповіді моделі та еталонної відповіді з каталогу згідно сценарію. Для RAG-конвеєра використовувався набір embedding-показників для відповідних частин перевірки конвеєру: retrieval quality, groundedness, hallucination penalty, answer quality.

У таблиці 5.6 представлені векторні оцінки для «наївної» відповіді LLM для двох запитів із загальної вибірки та усереднене значення із усієї вибірки.

Таблиця 5.6 – Векторні метрики LLM (косинусна схожість)

Запит (LLM)	s
«Покажи всі доступні в каталозі товари бренду Bosch.»	0.4631
«Перелічіть перфоратори з каталогу разом з цінами.»	0.3032
Агреговано по вибірці тестів:	0.4331

У порівняння у таблиці 5.7 представлена аналогічна вибірка для двох запитів із загалу та усереднене значення із усієї вибірки для тестування RAG-конвеєра. Порівняння відбувається за полем answer_quality, що є аналогічною метрикою оцінювання еталону та фактичної відповіді, як для LLM.

Таблиця 5.7 – Векторні метрики RAG (для кожного етапу)

Запит (RAG)	retrieval_quality	groundedness	hallucination_penalty	answer_quality
«Покажи всі доступні в каталозі товари бренду Bosch.»	1.0	0.7221	0.2565	0.96
«Перелічіть перфоратори з каталогу разом з цінами.»	1.0	0.6306	0.1917	0.7501
Агреговано по вибірці тестів:	1.0	0.6763	0.2241	0.8751

Для головного скалярного показника embedding-якості різниця та коефіцієнт покращення становить:

$$\Delta_{\text{emb}} = \bar{q}_{\text{RAG}} - \bar{s}_{\text{LLM}} \approx 0.8751 - 0.4331 = 0.4419$$

$$K_{\text{emb}} = \frac{\bar{q}_{\text{RAG}}}{\bar{s}_{\text{LLM}}} \approx 2.02$$

Тобто за векторними метриками RAG-відповіді в середньому приблизно вдвічі ближчі до еталонних, ніж відповіді «чистої» LLM. Крім того для RAG, ретривер витягує всі необхідні документи з 100% точністю, контекст відповіді базується на 67% від отриманих документів, а штраф галюцинації складає

близько 23%, що свідчить про помірну частку тексту, не прямо підтвердженого документами.

Далі було порівняно результати тестування вибірок за RAGAS-метриками. Для «чистої»-відповіді LLM вимірювалися лише за методикою:

– answer_relevancy – наскільки відповідь відповідає формулюванню запиту;

– factual_correctness – повнота відповіді у F1-формулюванні.

Для RAG-конвеєра використано ті ж метрики та додатково оцінювалися:

– context_recall – повнота використання релевантного контексту;

– faithfulness – відсутність галюцинацій (відповідність контексту).

Сумарна статистика за агрегованими середніми значення по всій вибірці тестування згідно методики представлена у таблиці 5.8.

Таблиця 5.8 – Сумарні RAGAS-метрики для LLM та RAG

Метрика	LLM	RAG
answer_relevancy	0.00	0.5947
factual_correctness (F1)	0.03	0.785
context_recall	–	1.0
faithfulness	–	1.0

Чисельний приріст за фактологічною коректністю становить:

$$\Delta_{F1} = \overline{factual\_correctness}_{RAG} - \overline{factual\_correctness}_{LLM} = 0.755$$

$$K_{F1} = \frac{\overline{factual\_correctness}_{RAG}}{\overline{factual\_correctness}_{LLM}} = \frac{0.785}{0.03} \approx 26.17$$

Отже, за F1-метрикою RAG-конвеєр дає приблизно у 26 разів кращу фактологічну відповідність еталонним відповідям каталогу, ніж «чиста» LLM. Для answer_relevancy маємо перехід з 0.00 до ≈0.59, формальний коефіцієнт покращення не визначається через нуль у знаменнику, але приріст показує суттєву зміну класу якості відповіді. Для елементу ретривера та відповідності контексту, метрики показують відмінний результат з 100% достовірністю, що вище ніж аналогічні векторні метрики, але слід враховувати, що у тест RAGAS

залучено велику мовну модель, як суддю результатів, що опираються не на фактичну відповідність еталону, а на суб’єктивну оцінку LLM, що може бути більш точним, так як імітується розуміння контексту запиту і інших складових на людському рівні, а не простий розрахунок.

Для наглядної роботи та відображення фактичного покращення відповіді RAG-конвеєром, нижче наведено порівняльну таблицю 5.9 для запиту про товари Bosch, де є повний набір: еталон, відповідь LLM та відповідь RAG.

Таблиця 5.9 – Порівняння відповіді для типового запиту

Елемент	Зміст (скорочено)
Запит	«Покажи всі доступні в каталозі товари бренду Bosch.»
Еталон	"Ось доступні товари Bosch з наявного каталогу: Bosch Маска респіратор X 202 (Засоби безпеки та захисту, 1010.46₴), Bosch Викрутка S 929 (Ручний інструмент, 1273.42₴) Bosch Болгарка компакт 149 (Електроінструменти для дому, 10435.6₴), Bosch Секатор X 115 (Садові інструменти, 5439.1₴), Bosch Ліхтар акумуляторний X 223 (Освітлювальне обладнання, 5589.37₴) ..."
Відповідь LLM (без RAG)	"Звісно, я можу допомогти вам з інформацією про товари бренду Bosch. Однак, як штучний інтелект, я не маю доступу до **актуального каталогу конкретного магазину чи онлайн-платформи** в реальному часі, щоб показати вам наявність та ціни. Мої знання базуються на загальнодоступній інформації про продукцію Bosch .... Щоб знайти конкретні товари в наявності та актуальні ціни, рекомендую відвідати:**\n\n* **Офіційний сайт Bosch** у вашому регіоні ... **Сайти великих роздрібних мереж електроніки та побутової техніки** ... Якщо у вас є конкретна категорія товарів Bosch, яка вас цікавить (наприклад, \"холодильники Bosch\" ... я можу надати більш деталізовану інформацію..."

Кінець таблиці 5.9

Відповідь RAG	"Звичайно! Пропоную Вам ознайомитись з наступними товарами нашого магазину: - Bosch Маска респіратор X 202 - Засоби безпеки та захисту, 1010.46₴ - Bosch Викрутка S 929 - Ручний інструмент, 1273.42₴ - Bosch Болгарка компакт 149 - Електроінструменти для дому, 10435.6₴ - Bosch Секатор X 115 - Садові інструменти, 5439.1₴ - Bosch Ліхтар акумуляторний X 223 - Освітлювальне обладнання, 5589.37₴"

Для цього запиту наступні метрики на порівняння:

– для LLM: косинусна схожість до еталону ≈ 0.463, answer_relevancy = 0.0, factual_correctness = 0.06;

– для RAG: косинусна схожість до еталону ≈ 0.96, retrieval_quality = 1.0, groundedness ≈ 0.722, answer_quality = 1.0, answer_relevancy ≈ 0.655, factual_correctness = 1.0.

Суб’єктивно оцінюючи, можна побачити, що «чиста»-відповідь LLM (без підкріплення зовнішніми знаннями) генерує лише тематично схожий, довідковий текст без жодного каталожного факту, що підтверджує, що вона не виконує задачі у високоспеціалізованих доменах. Натомість RAG-конвеєр на тому самому запиті відтворює повну та майже ідентичну до еталонної інформацію із внутрішнього сховища, досягаючи майже відмінної точності по більшості метрик, що були описані вище. Окрім того, усі потрібні документи зі сховища витягуються, а відповіді не містять галюцинацій щодо каталогу (за винятком загальних фраз та позаконтекстних текстів, сформованих мовною моделлю). На рисунках 5.3-5.5 представлений типовий діалог з RAG-«агентом».

Рисунок 5.3 – Загальний запит до RAG-«агента»

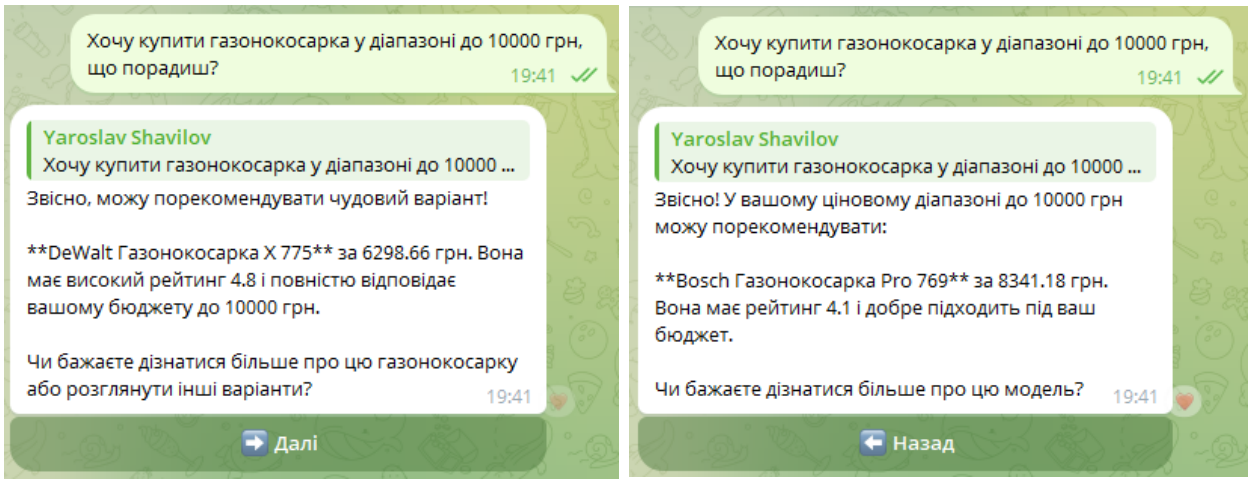


Рисунок 5.4 – Вузькоспеціалізований запит до RAG-«агента»

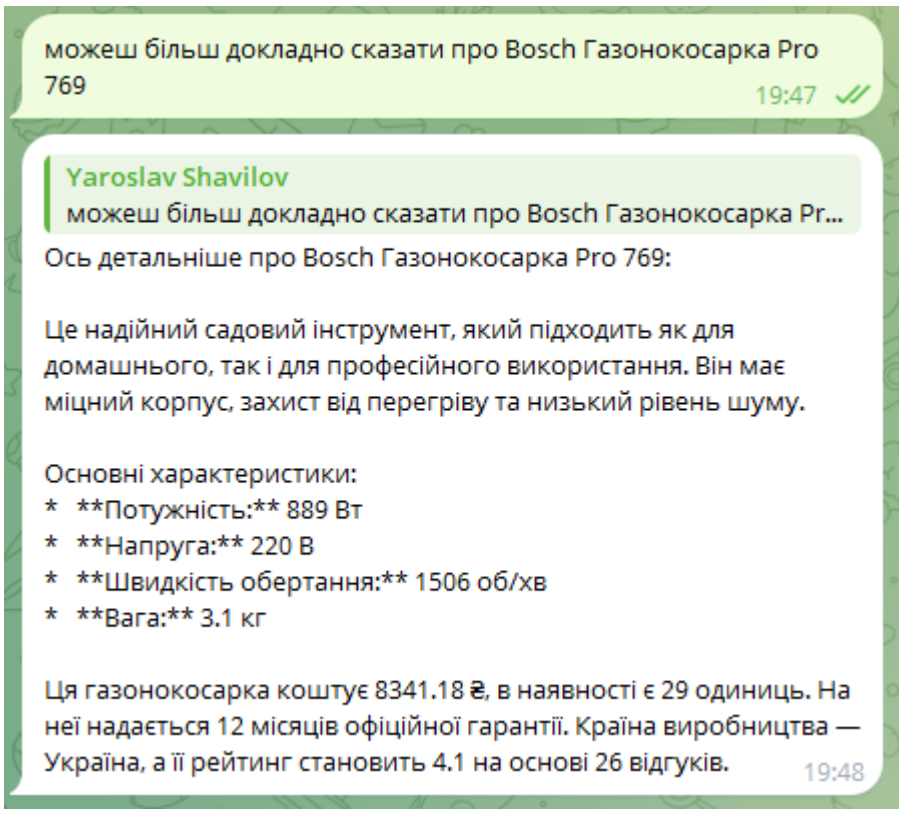


Рисунок 5.5 – Запит конкретної товарної позиції до RAG-«агента»

Тестування MongoDB-конвеєра виконувалося відповідно до методики, заданої у підпункті 5.2; самі запити та pipeline-и для тестування формувалися під час LLM-RAGAS-тесту як частини витягу конструкцій ретривера, тому наведені нижче результати у таблиці 5.10 кількісно відповідають сценарію RAG тестування. У таблиці 5.10 задано типові запити, згідно яких ретривер формує агрегаційні pipeline-и та узагальнені результати по всій вибірці. Загалом тестування було успішно, всі конструкції пройшли перевірку безпеки.

Таблиця 5.10 – Результати перевірки безпечності MongoDB-pipeline

№	Користувацький запит	Характеристика pipeline	Оцінка сценарію	Загальних перевірок (із 30)	Успішні (із 30)	Неуспішні (із 30)
1	«Покажи перфоратори»	$match по категорії + regex перфоратор у name, далі $project релевантних полів	безпечний	–	–	–

Кінець таблиці 10

2	«Знайди акумуляторні інструменти»	Один $match: категорії з білого списку + regex «акумуляторн» у description	безпечний	–	–	–
	Узагальнення по всіх сценаріях	Усі перевірені pipeline-и сформували коректні запити до колекції згідно методики	30/30: безпечний	30	30	0

Останньою перевіркою системи є тестування FSM-сценарію оформлення замовлення: перевіряється, що бот коректно проходить усі етапи процесу від старту оформлення до фіксації замовлення або відміни, без «зависань» у станах та з правильними переходами між ними. У таблиці 5.11 представлені результати тестового сценарію з усіма станами та фіксацією успіху обробки.

Таблиця 5.11 – Результати перевірки переходів FSM-сценарію замовлення

№	Перехід	Кількість успішних переходів	Кількість неуспішних переходів
1	Вибір категорії	30	0
2	Вибір товару	30	0
3	Підтвердження деталей товару	30	0
4	Введення кількості одиниць товару	30	0
5	Підтвердження замовлення	30	0
6	Дострокова відміна замовлення користувачем	30	0
	Загалом пройдених сценаріїв замовлення:	30/30	0

За результатами, тестування показало що всі етапи та сценарії FSM-замовлення є успішними. На рисунках 5.6-5.8 зображено виконання різних етапів з підтвердженням результату.

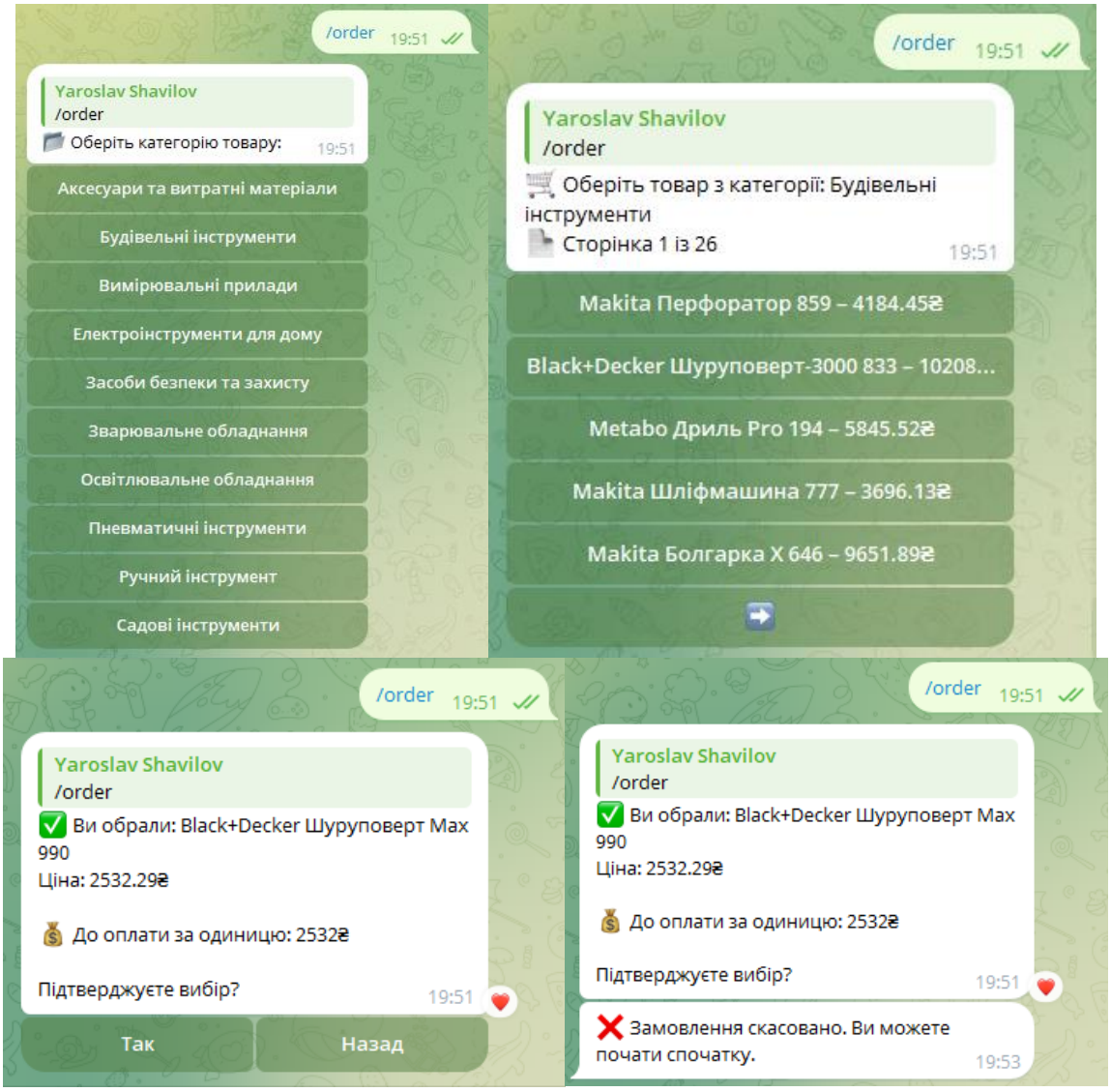


Рисунок 5.6 – Перевірка виходу із сценарію замовлення на етапі «підтвердження деталей товару»

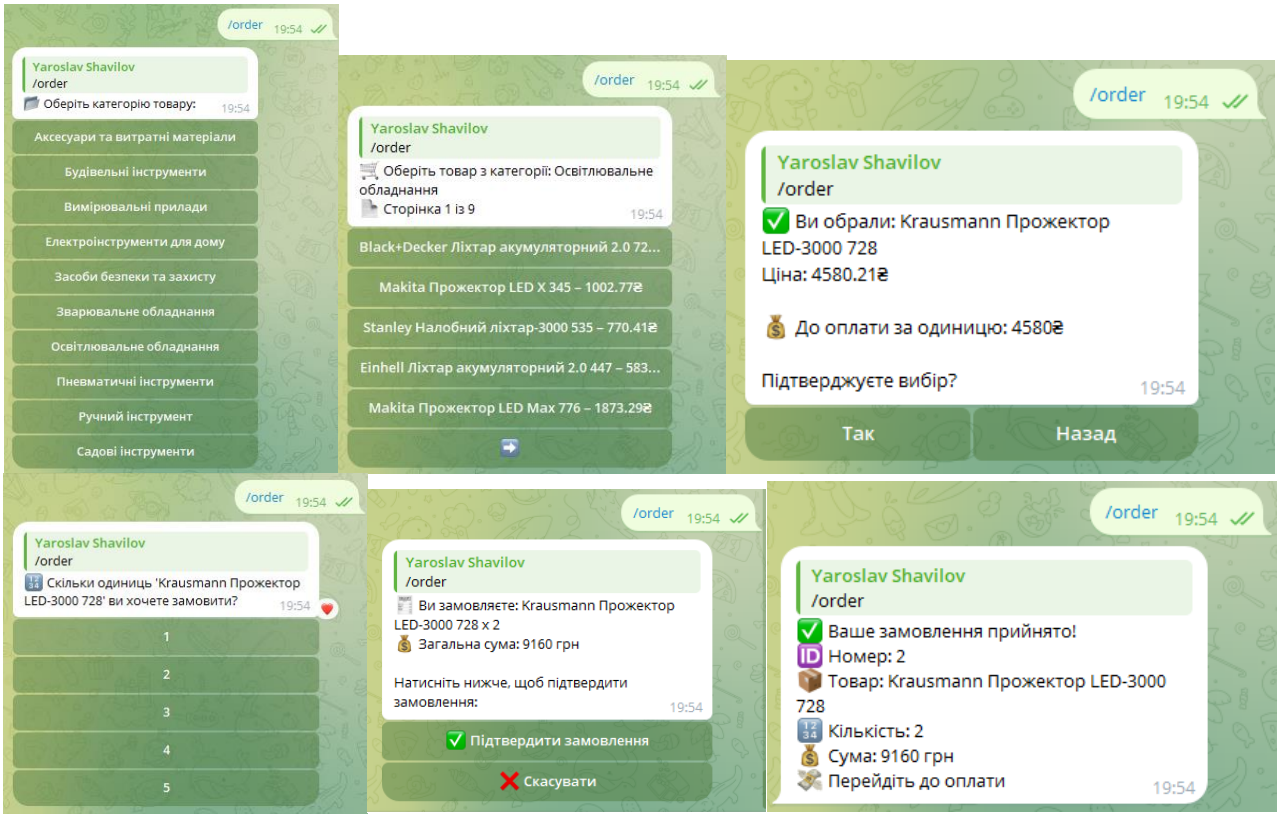
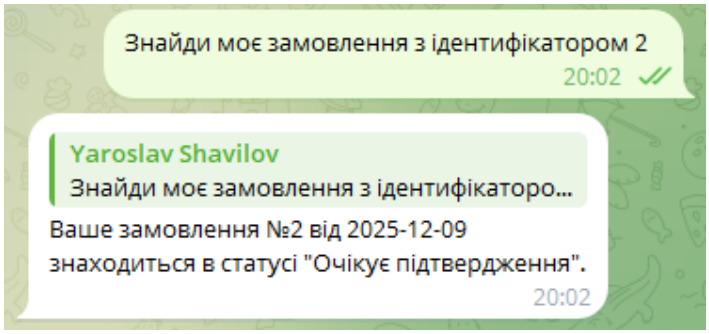


Рисунок 5.7 – Проходження всього сценарію




```
_id: ObjectId('693863f9b1bfd0d74f69b615')
order_id : "2"
customer_id : "[redacted]3"
product_id : "1369"
quantity : 2
total_price : 9160
order_date : "2025-12-09"
status : "Очікує підтвердження"
shipping_address : "Не вказано"
▾ contact_info : Object
    phone : ""
    email : ""
```


Рисунок 5.8 – Підтвердження успішності замовлення

5.4.3 Аналіз результатів

Отримані результати експериментів у цілому підтвердили коректність побудованої моделі мережі СМО та закладених інженерних рішень щодо архітектури системи. Порівняння теоретичних оцінок часу перебування заявки в мережі з експериментальними вимірюваннями показало, що для пікового навантаження модель добре описує реальну систему: розрахунковий і фактичний час реакції практично збігаються, що свідчить про доцільність обраного підходу до формалізації черг і розподілу навантаження. У режимах меншого навантаження теоретичні значення виявилися дещо завищеними, натомість експериментальні дані демонструють кращу швидкодію, ніж «прогнозує» модель. Це можна трактувати як те, що модель задає консервативну оцінку і фактично визначає верхню межу часу відповіді, тоді як реальна система працює із запасом продуктивності і за іншими маршрутами фактичного навантаження. Важливим підсумком є те, що в усіх розглянутих режимах середній час відповіді залишається в межах заданого обмеження, тобто система здатна обробляти поточний потік запитів користувачів у прийнятні терміни.

Аналіз завантаженості окремих вузлів показав, що стримуючим фактором є саме LLM-вузол, на який накладаються жорсткі обмеження через квоти зовнішнього API. Вузли Telegram-бота та гібридного сховища знань працюють у зоні низького завантаження і мають суттєвий резерв пропускної здатності, тоді як для LLM у високонавантаженому режимі спостерігається наближення до межі стійкості. Це підтверджує правильність прийнятого рішення зосередити увагу на оптимізації цього компонента за рахунок кешування та параметрах генерації, тоді як базу даних і векторний пошук можна розглядати як достатньо «легкі» з точки зору подальшої експлуатації. Водночас поєднання результатів продуктивності з метриками якості відповідей показує, що підключення RAG-конвеєра не руйнує досягнутий баланс: додаткові звернення до бази знань практично не погіршують часові характеристики, але суттєво покращують зміст відповідей.

Щодо RAG-конвеєра та FSM-сценарію – експериментальні оцінки демонструють стабільне зростання релевантності і повноти відповідей порівняно з непідкріпленою LLM: краще спираються на фактичні дані з корпоративних документів, зменшується частка галюцинацій, підвищується узгодженість між різними запитами однієї предметної області. Це особливо важливо для задач, пов’язаних з підбором будівельного обладнання, де помилки в характеристиках товару або умовах постачання безпосередньо впливають на бізнес-процеси. Перевірка безпечності та повноти покриття тестових сценаріїв для інших компонентів моделі показали відсутність руйнівних запитів і коректне проходження всіх станів. Таким чином, система не лише генерує змістовно якісні відповіді, а й дотримується вимог безпеки доступу до даних і надійності бізнес-логіки.

Узагальнюючи, можна стверджувати, що проведені експерименти підтвердили доцільність запропонованої архітектури: обрана мережна модель дозволяє обґрунтовано прогнозувати поведінку системи при зміні навантаження; гібридне сховище знань забезпечує швидкий доступ до структурованих та неструктурованих даних; підключення RAG-модуля суттєво підвищує якість відповідей, а розроблений FSM-сценарій і механізм формування безпечних запитів гарантують коректність роботи на рівні бізнес-процесів.

Отримані результати мають як наукову, так і практичну значущість: з наукового боку для подібної системи підтримки клієнтів формалізовано інтеграцію LLM, RAG-конвеєра та бази знань у вигляді мережі СМО з верифікацією на реальних експериментальних даних; з практичного – показано, що розроблений підхід може бути використаний як типове рішення для впровадження інтелектуальних сервісів підтримки у компаніях з близькою структурою даних.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи розв’язано завдання на розробку та дослідження гібридної інтелектуальної системи автоматизованого обслуговування клієнтів на основі Telegram-бота, великої мовної моделі та спеціалізованих сховищ даних. Реалізовано ідею поєднання RAG-підходу з моделями масового обслуговування для побудови архітектури, що забезпечує узгоджену роботу підсистем діалогу, доступу до знань і зберігання даних за умов обмежених ресурсів.

Отримано й досліджено математичну модель системи масового обслуговування клієнтських запитів, яка відображає структуру сервісних вузлів і взаємодію між модулями Telegram-бота, LLM-ядра та підсистемами зберігання даних. Показано, що в розглянутому діапазоні інтенсивностей вхідного потоку система забезпечує допустимі нормативні значення. Розроблена методика експериментальної оцінки якості відповідей та продуктивності дала змогу кількісно порівняти різні конфігурації RAG-конвеєра та вибрати раціональні параметри індексування, обсягів контексту й механізмів кешування.

Сформульовані положення підтверджено експериментальними дослідженнями зразка системи, що демонструє підвищення повноти та релевантності відповідей порівняно з традиційними чат-ботами, які не використовують архітектуру доповнення контекстом.

Практичні результати роботи полягають у створенні прототипу програмно-апаратного комплексу клієнтського сервісу з алгоритмами семантичної обробки, вибору інформаційних джерел, формування відповідей та маршрутизації складних запитів. Запропоновано регламенти використання системи в корпоративному середовищі постачальника будівельного обладнання, що охоплюють етапи наповнення бази знань, підтримання її актуальності та моніторингу показників сервісу. Отримані результати можуть бути використані для впровадження інтелектуальних систем підтримки клієнтів у компаніях, які працюють з великими потоками звернень.

ПЕРЕЛІК ПОСИЛАНЬ

1. zoulinf. IBM global AI adoption index report dec. 2023 | PDF | artificial intelligence | intelligence (AI) & semantics. Scribd. URL: https://www.scribd.com/document/737650470/IBM-Global-AI-Adoption-Index-Report-Dec-2023 (дата звернення: 09.12.2025).

2. The next frontier of customer engagement: AI-enabled customer service / A. C. Das та ін. McKinsey & Company. URL: https://www.mckinsey.com/capabilities/operations/our-insights/the-next-frontier-of-customer-engagement-ai-enabled-customer-service (дата звернення: 09.12.2025).

3. The economic potential of generative AI: The next productivity frontier / M. Chui та ін. McKinsey & Company. URL: https://www.mckinsey.com/capabilities/tech-and-ai/our-insights/the-economic-potential-of-generative-ai-the-next-productivity-frontier (дата звернення: 10.12.2025).

4. How Telegram bot enhances a businesss omnichannel strategy | Yespo. yespo. URL: https://yespo.io/blog/role-telegram-bots-omnichannel-business-approach (дата звернення: 10.12.2025).

5. Integration of Telegram bots with CRM systems for improved customer service - BAZU. BAZU. URL: https://bazucompany.com/blog/integration-of-telegram-bots-with-crm-systems-for-improved-customer-service/ (дата звернення: 10.12.2025).

6. Sudeep Meduri. Revolutionizing Customer Service : The Impact of Large Language Models on Chatbot Performance. International Journal of Scientific Research in Computer Science, Engineering and Information Technology. 2024. Т. 10, № 5. С. 721–730.

7. Ramesh D., Khosla E., Bhukya S. N. Inclusion of e-commerce workflow with NoSQL DBMS: MongoDB document store. 2016 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC), м. Chennai, 15–17 груд. 2016 р. 2016.

8. What is RAG? - Retrieval-Augmented Generation AI Explained - AWS. Amazon Web Services, Inc. URL: https://aws.amazon.com/what-is/retrieval-augmented-generation/ (дата звернення: 10.12.2025).

9. AI in Customer Service: How RAG and LLMs Are Transforming Support at Scale - deepsense.ai. deepsense.ai. URL: https://deepsense.ai/blog/ai-in-customer-service-how-rag-and-llms-are-transforming-support-at-scale/ (дата звернення: 10.12.2025).

10. Skakalina O., Holub A. LARGE LANGUAGE MODELS: BUSINESS APPLICATIONS AND DEVELOPMENT PROSPECTS. Системи управління, навігації та зв'язку. Збірник наукових праць. 2025. Т. 1, № 79. С. 129–133.

11. Stefanelli G. ChatGPT-5 vs previous models: Full Report and Comparison of features, capabilities, pricing, and more. Data Studios ·Exafin. URL: https://www.datastudios.org/post/chatgpt-5-vs-previous-models-full-report-and-comparison-of-features-capabilities-pricing-and-mor (дата звернення: 10.12.2025).

12. Top 50 AI Model Benchmarks & Evaluation Metrics (2025 Guide) | Articles | O-mega. O-mega. URL: https://o-mega.ai/articles/top-50-ai-model-evals-full-list-of-benchmarks-october-2025 (дата звернення: 10.12.2025).

13. Sanchez J. Gemini 2.5 Cost and Quality Comparison | Pricing & Performance. Leanware. URL: https://www.leanware.co/insights/gemini-2-5-cost-quality-comparison (дата звернення: 10.12.2025).

14. Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities. arXiv.org. URL: https://arxiv.org/abs/2507.06261 (дата звернення: 10.12.2025).

15. NEWS P. Meta презентувала Llama 4 з мультимодальністю і reasoning-first дизайном - ProIT. ProIT: медіа для профі в IT. URL: https://proit.ua/meta-priezientuvala-llama-4-z-multimodalnistiu-i-reasoning-first-dizainom/ (дата звернення: 10.12.2025).

16. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. AI at Meta. URL: https://ai.meta.com/blog/llama-4-multimodal-intelligence/ (дата звернення: 10.12.2025).

17. Mishra A., Brahmanapally N. A Comparative Performance Analysis of Locally Deployed Large Language Models Through a Retrieval-Augmented Generation Educational Assistant Application for Textual Data Extraction. AI. 2025. Т. 6, № 6. С. 119.

18. Aniela BORCAN, Diana CAPTARI, Emanuela-Cristina CARP. Comparation of the Performance of SQL and NoSQL Databases in Modern Business Applications. Database Systems Journal. 2025. Т. XVI. С. 126–141.

19. Kausar M. A., Nasar M., Soosaimanickam A. A Study of Performance and Comparison of NoSQL Databases: MongoDB, Cassandra, and Redis Using YCSB. Indian Journal Of Science And Technology. 2022. Т. 15, № 31. С. 1532–1540.

20. Sonia Anurag Dubey. Performance Analysis of NoSQL Databases for Healthcare Applications: A Benchmarking Study of MongoDB, Cassandra and Couchbase. Journal of Information Systems Engineering and Management. 2025. Т. 10, № 25s. С. 380–398.

21. Carvalho I., Sá F., Bernardino J. Performance Evaluation of NoSQL Document Databases: Couchbase, CouchDB, and MongoDB. Algorithms. 2023. Т. 16, № 2. С. 78.

22. 8.14. JSON Types. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/current/datatype-json.html (дата звернення: 10.12.2025).

23. Dudycz O. PostgreSQL JSONB - Powerful Storage for Semi-Structured Data. Architecture Weekly | Oskar Dudycz | Substack. URL: https://www.architecture-weekly.com/p/postgresql-jsonb-powerful-storage (дата звернення: 10.12.2025).

24. Dynamo | Apache Cassandra Documentation. Apache Cassandra. URL: https://cassandra.apache.org/doc/latest/cassandra/architecture/dynamo.html (дата звернення: 10.12.2025).

25. Apache Cassandra Data Modeling Best Practices Guide. Instaclustr. URL: https://www.instaclustr.com/blog/cassandra-data-modeling/ (дата звернення: 10.12.2025).

26. EDB Postgres AI - World's First Sovereign AI and Data Platform. URL: https://info.enterprisedb.com/rs/069-ALB-339/images/PostgreSQL_MongoDB_Benchmark-WhitepaperFinal.pdf (дата звернення: 10.12.2025).

27. JSON performance: PostgreSQL vs MongoDB Comparison. Document Database Community. URL: https://documentdatabase.org/blog/json-performance-postgres-vs-mongodb (дата звернення: 10.12.2025).

28. MongoDB vs PostgreSQL: Which to Choose For Your Database Solutions. EDB. URL: https://www.enterprisedb.com/choosing-mongodb-postgresql-cloud-database-solutions-guide (дата звернення: 10.12.2025).

29. Apache Cassandra Use Cases in Big Data. Neontri. URL: https://neontri.com/blog/apache-cassandra-use-cases/ (дата звернення: 10.12.2025).

30. The AI Engineer’s Playbook: Mastering Vector Search & Management (Part 2). Medium. URL: https://medium.com/data-science-collective/the-ai-engineers-playbook-mastering-vector-search-management-part-2-7a74b8038bc5 (дата звернення: 10.12.2025).

31. Chroma Docs. Chroma Docs. URL: https://docs.trychroma.com/docs/overview/architecture (дата звернення: 10.12.2025).

32. Retrieval-Augmented Generation for AI-Generated Content: A Survey. arXiv.org. URL: https://arxiv.org/abs/2402.19473 (дата звернення: 10.12.2025).

33. A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models / W. Fan та ін. KDD '24: The 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, м. Barcelona Spain. New York, NY, USA, 2024. С. 6491–6501.

34. A Survey on RAG with LLMs / M. Arslan та ін. Procedia Computer Science. 2024. Т. 246. С. 3781–3790.

35. When Large Language Models Meet Vector Databases: A Survey. arXiv.org. URL: https://arxiv.org/abs/2402.01763 (дата звернення: 10.12.2025).

36. Gen-SQL: Efficient Text-to-SQL By Bridging Natural Language Question And Database Schema With Pseudo-Schema / Jie Shi та ін. Proceedings of the 31st International Conference on Computational Linguistics. 2025. С. 3794–3807.

37. Text-to-SQL: A Developer’s Zero-to-Hero Guide | Tiger Data. Tiger Data: PostgreSQL++ for Time Series, Analytics & AI | Creators of TimescaleDB. URL: https://www.tigerdata.com/learn/text-to-sql-a-developers-zero-to-hero-guide (дата звернення: 10.12.2025).

38. Text2VectorSQL: Bridging Text-to-SQL and Vector Search for Unified Natural Language Queries. arXiv.org e-Print archive. URL: https://arxiv.org/html/2506.23071v1 (дата звернення: 10.12.2025).

39. Arooj. Connecting RAG to SQL Databases: A Practical Guide. Chitika: Explore Retrieval Augmented Generation Trends. URL: https://www.chitika.com/rag-sql-database-integration/ (дата звернення: 10.12.2025).

40. Indexing and Routing Strategies in Retrieval-Augmented Generation (RAG) Chatbots. Medium. URL: https://blog.gopenai.com/indexing-and-routing-strategies-in-retrieval-augmented-generation-rag-chatbots-06271908e9f6 (дата звернення: 10.12.2025).

41. James A., Trovati M., Bolton S. Retrieval-Augmented Generation to Generate Knowledge Assets and Creation of Action Drivers. Applied Sciences. 2025. Т. 15, № 11. С. 6247.

42. RAGAs: Automated Evaluation of Retrieval Augmented Generation / S. Es та ін. Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations, м. St. Julians, Malta. Stroudsburg, PA, USA, 2024. С. 150–158.

43. Теорія масового обслуговування в телекомунікаціях / А.Г. Ложковський. – Одеса: ОНАЗ ім. О.С. Попова, 2010. – 112 с.: іл.

44. Теорія систем масового обслуговування : навч. посібник / А. Л. Литвинов ; Харків. нац. ун-т міськ. госп-ва ім. О. М. Бекетова. – Харків : ХНУМГ ім. О. М. Бекетова, 2018. – 141 с.

45. Формальні мови, граматики та автомати: Навчальний посібник/ Гавриленко С.Ю. – Харків: НТУ « ХПІ », 2021. – 133 с. – на укр. мові.

46. Remke A. Model Checking Structured Infinite Markov Chains : Ph.D.-thesis. Münster, 2008. 154 c.

47. AI Chatbots for Business: Transforming Your Customer Experience. Build Enterprise-Grade Customer Experience AI Agents | Jio Haptik. URL: https://www.haptik.ai/knowledge-center/ai-chatbot-for-business?utm_source=chatgpt.com (дата звернення: 28.09.2025).

48. AI-powered chatbot to transform employee experience. Leena AI- Build a zero ticket enterprise. URL: https://leena.ai/hr-chatbot (дата звернення: 28.09.2025).

49. Case Studies | Enterprise Bot. Conversational Automation for Enterprises | Enterprise Bot. URL: https://www.enterprisebot.ai/case-studies (дата звернення: 28.09.2025).

50. Kagan E., Dada M., Hathaway B. AI Chatbots in Customer Service: Adoption Hurdles and Simple Remedies. SSRN Electronic Journal. 2022. URL: https://doi.org/10.2139/ssrn.4283285 (дата звернення: 28.09.2025).

ДОДАТОК А
ТЕКСТ ПРОГРАМИ СИСТЕМИ АВТОМАТИЗОВАНОГО ОБСЛУГОВУВАННЯ КЛІЄНТІВ КОМПАНІЇ-ПОСТАЧАЛЬНИКА БУДІВЕЛЬНОГО ОБЛАДНАННЯ

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
“ДНІПРОВСЬКА ПОЛІТЕХНІКА”

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ АВТОМАТИЗОВАНОГО ОБСЛУГОВУВАННЯ КЛІЄНТІВ КОМПАНІЇ-ПОСТАЧАЛЬНИКА БУДІВЕЛЬНОГО ОБЛАДНАННЯ

Текст програми

804.02070743.25025-01 12 01

Листів 19

2025

АНОТАЦІЯ

Розроблена програма є інтелектуальною системою автоматизованого обслуговування клієнтів на основі Telegram-бота, інтегрованого з великою мовною моделлю Gemini та RAG-архітектурою для доступу до корпоративної бази знань з MongoDB та ChromaDB.

Система є багатомодульною та забезпечує обробку клієнтських запитів, пошук інформації у внутрішній документації компанії, формування відповідей та маршрутизацію складних звернень.

Програма реалізує модулі діалогу, керування даними, аналітики та моніторингу, підтримує масштабування й може бути адаптована під потреби компаній-постачальників будівельного обладнання та інших підприємств.

ЗМІСТ

С.

1 Файл BudStore.py

```
import asyncio
import logging
from aiogram.types import BotCommand, BotCommandScopeDefault
from aiogram import Bot, Dispatcher, types
from aiogram.filters import Command
from config import TELEGRAM_TOKEN
from handlers import handle_message, handle_pagination_callback, PageCallbackFilter
import db
from db import sessions, init_chroma
from order_fsm import router, start_order
from aiogram.fsm.context import FSMContext
bot = Bot(token=TELEGRAM_TOKEN)
dp = Dispatcher()
dp.callback_query.register(handle_pagination_callback, PageCallbackFilter())
dp.include_router(router)  # FSM хендлери через router
# Функція для встановлення меню команд бота
async def set_commands():
    commands = [
        BotCommand(command="/start", description="Старт бота"),
        BotCommand(command="/clear_history", description="Очистити історію чату"),
        BotCommand(command="/rule", description="Правила спілкування з ботом"),
        BotCommand(command="/order", description="Оформити замовлення"),
    ]
    # Встановлюємо команди для бота
    await bot.set_my_commands(commands, BotCommandScopeDefault())
@dp.message(Command("order"))
async def cmd_order(message: types.Message, state: FSMContext):
    await start_order(message, state)
@dp.message(Command("start"))
async def cmd_start(message: types.Message):
    await message.reply("👋 Вітаємо!\n□ У чаті використано генеративну AI-модель!\n💬 Бот
спілкується природною мовою\n✍ Просто напишіть запит, що Вас цікавить —\n□ Ми
опрацюємо його як типовий AI-чат-бот!")
@dp.message(Command("clear_history"))
```

```
async def clear_history(message: types.Message):
    user_id = message.from_user.id
    # Видаляємо історію користувача з бази
    result = await sessions.update_one({"user_id": user_id}, {"$set": {"history": []}})
    if result.matched_count: await message.reply("□ Історію чат-боту очищено.")
    else: await message.reply("🕳 Історія була порожня або не знайдена.")
@dp.message(Command("rule"))
async def send_rule(message: types.Message):
    instruction_text = (...)
    await message.reply(instruction_text, parse_mode="Markdown")
@dp.message()
async def universal_handler(message: types.Message):
    await handle_message(message)
async def main():
    await init_chroma()
    print("[bot] sees:", db.async_chroma, db.async_chroma_collection)
    await set_commands()
    await dp.start_polling(bot, skip_updates=True)  # Запуск бота для обробки повідомлень
if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO)
    asyncio.run(main())
```

2 Файл config.py

```
TELEGRAM_TOKEN = '########################################'
MONGO_URI = '########################################'
GEMINI_API_KEY = '########################################'
```

3 Файл handlers.py

```
import asyncio
from aiogram import types
from db import orders, products, stores, services
from ai_assistant import (
    analyze_action,
    analyze_message,
```

```
    respond_to_data,
    respond_to_other,
    vector_context_from_chroma,
    gemini_call
)
import traceback
import json
ACTION_MAP = {
    "find_order": ("order_template", orders),
    "find_product": ("product_template", products),
    "find_store": ("store_template", stores),
}
from aiogram.types import InlineKeyboardMarkup, InlineKeyboardButton
from aiogram.filters import BaseFilter
PAGE_SIZE = 3
user_results = {}
def make_pagination_keyboard(results: list, page: int, total: int):
    buttons = []
    if page > 0:
        buttons.append(InlineKeyboardButton(text="⬅ Назад", callback_data=f"pgn:{page-1}"))
    if (page + 1) * PAGE_SIZE < total:
        buttons.append(InlineKeyboardButton(text="➡ Далі", callback_data=f"pgn:{page+1}"))
    if not buttons:
        return None
    inline_keyboard = [buttons[i:i+2] for i in range(0, len(buttons), 2)]
    return InlineKeyboardMarkup(inline_keyboard=inline_keyboard)
async def handle_message(message: types.Message):
    try:
        user_id = message.from_user.id
        text = message.text.strip()
        # Очищення попередньої пагінації
        if user_id in user_results:
            ctx = user_results[user_id]
            msg_id = ctx.get("message_id")
            if msg_id:
```

```
            try:
                await message.bot.edit_message_reply_markup(
                    chat_id=message.chat.id, message_id=msg_id, reply_markup=None)
            except Exception as e:
                print(f"Помилка видалення старої клавіатури: {e}")
        del user_results[user_id]
    # Аналіз дії
    action_data = await analyze_action(user_id, text)
    if not action_data or not action_data.get("action"):
        await message.reply("⚠ Не вдалося визначити дію. Спробуйте інакше.")
        return
    action = action_data["action"]
    # Простий тип дій
    if action == "create_service_request":
        await message.reply("• Ми зв'яжемо вас з нашим оператором, очікуйте!")
        return
    if action == "product_order":
        await message.reply("Зацікавив продукт?\n• Для створення замовлення введіть
команду:\n/order")
        return
    if action == "other_action":
        reply = await respond_to_other(user_id, text)
        await message.reply(reply)
        return
    if action not in ACTION_MAP:
        await message.reply("⚠ Невідома дія. Спробуйте сформулювати запит інакше.")
        return
    # Якщо шукаємо замовлення - додаємо фільтр по користувачу
    if action == "find_order":
        text += f'. Запит тільки для користувача: customer_id: "{user_id}"'
    template_name, collection = ACTION_MAP[action]
    # AI формує Mongo pipeline
    pipeline = await analyze_message(user_id, text, action, template_name)
    try:
```

```
            if not pipeline or not isinstance(pipeline, list):
                raise ValueError("Invalid or empty pipeline")
            # Motor: async aggregate
            cursor = collection.aggregate(pipeline)
            result = await cursor.to_list(length=5)
            if len(result) == 0:
                raise ValueError("Empty pipeline result")
            total_results = len(result)
            first_page = result[:PAGE_SIZE]
            # Формування тексту
            reply = await respond_to_data(user_id, first_page, text)
            if total_results > PAGE_SIZE:
                keyboard = make_pagination_keyboard(result, page=0, total=total_results)
                sent_message = await message.reply(reply, reply_markup=keyboard)
                user_results[user_id] = {"data": result, "text": text,
                    "message_id": sent_message.message_id}
            else: await message.reply(reply)
        except Exception as e:
            print(f"Mongo pipeline error or no results: {e}")
            # Семантичний пошук Chroma
            vec_ctx = await vector_context_from_chroma(text)
            print(vec_ctx)
            if vec_ctx:
                prompt = f"""..."""
                reply = await gemini_call(user_id, prompt, True)
                await message.reply(reply)
            else:
                await message.reply("⚠ Не вдалося знайти результати навіть через семантичний
пошук.")
    except Exception:
        traceback.print_exc()
        await message.reply("❌ Виникла внутрішня помилка. Спробуйте сформувати запит по
іншому.")
class PageCallbackFilter(BaseFilter):
    async def __call__(self, callback: types.CallbackQuery) -> bool:
```

```
    return callback.data is not None and callback.data.startswith("pgn:")
async def handle_pagination_callback(callback_query: types.CallbackQuery):
    user_id = callback_query.from_user.id
    user_data = user_results.get(user_id)
    if not user_data:
        await callback_query.answer("⚠ Результати більше не доступні.")
        return
    try:
        page = int(callback_query.data.split(":")[1])
    except (IndexError, ValueError):
        await callback_query.answer("Помилка пагінації.")
        return
    start = page * PAGE_SIZE
    end = start + PAGE_SIZE
    data = user_data["data"]
    text = user_data["text"]
    if start >= len(data):
        await callback_query.answer("⚠ Немає такої сторінки.")
        return
    page_data = data[start:end]
    # Формуємо відповідь
    reply = await respond_to_data(user_id, page_data, text)
    keyboard = make_pagination_keyboard(data, page, len(data))
    if keyboard:
        edited_message = await callback_query.message.edit_text(reply, reply_markup=keyboard)
    else:
        edited_message = await callback_query.message.edit_text(reply)
    user_results[user_id]["message_id"] = edited_message.message_id
    await callback_query.answer()
```

4 Файл ai_assistant.py

```
import json
import re
import logging
```

```
import hashlib
from typing import Any, Dict, List, Optional
import chromadb
from chromadb.config import Settings
from config import GEMINI_API_KEY
import db
from db import sessions, templates, available_fields, collection_meta, db_mongo,
ALLOWED_FIELDS, TEMPLATE_TO_COLLECTION
from datetime import datetime
import asyncio
from google.api_core.exceptions import ResourceExhausted
GEMINI_MODELS = ["models/gemini-2.5-flash", "models/gemini-2.5-flash-lite"]
from google import genai
from google.genai import Client, types
CACHE = db_mongo["ai_cache"]
HISTORY = db_mongo["sessions"]
logger = logging.getLogger(__name__)
REPLY_SAFE_LIMIT = 4000
MAX_HISTORY = 20
# ДОПОМІЖНІ ФУНКЦІЇ
def _hash_key(uid: int, text: str) -> str:
    h = hashlib.sha1()
    h.update(f"{uid}:{text}".encode("utf-8"))
    return h.hexdigest()
async def _get_cache(key: str):
    d = await CACHE.find_one({"_id": key})
    return d["value"] if d else None
async def _set_cache(key: str, val: Any):
    await CACHE.update_one(
        {"_id": key}, {"$set": {"value": val, "created_at": datetime.utcnow()}}, upsert=True )
def _trim_text(txt: str, n=600):
    txt = txt.strip()
    return txt if len(txt) <= n else txt[:n] + "..."
def _safe_json_extract(txt: str):
    try: m = re.search(r'(\[.*\]|\{.*\})', txt, re.DOTALL)
```

```
        if m: return json.loads(m.group())
    except Exception:
        return None
    return None
# ІСТОРІЯ КОРИСТУВАЧА
async def get_user_history(uid: int):
    s = await HISTORY.find_one({"user_id": uid})
    return s.get("history", []) if s else []
async def update_user_history(uid: int, role: str, text: str):
    s = await HISTORY.find_one({"user_id": uid}) or {"user_id": uid, "history": []}
    s["history"].append({"role": role, "parts": [text]})
    s["history"] = s["history"][-MAX_HISTORY:]
    await HISTORY.update_one({"user_id": uid}, {"$set": {"history": s["history"]}}, upsert=True)
async def gemini_call(uid: int, prompt: str, use_history=False) -> str:
    # --- КЕШ ---
    key = _hash_key(uid, prompt)
    cached = await _get_cache(key)
    if cached:
        return cached
    # --- ІСТОРІЯ ---
    hist = await get_user_history(uid)
    hist = hist[-1:] if use_history else []
    if hist:
        hist_text = "\n\n".join([f"{h['role']}: {h['parts'][0]}" for h in hist])
        full_prompt = f"{hist_text}\n\nUser: {prompt}"
    else:
        full_prompt = prompt
    contents = types.Content(role="user", parts=[types.Part(text=full_prompt)])
    final_response = None
    errors = []
    client = Client(
        api_key=GEMINI_API_KEY, http_options=types.HttpOptions(api_version="v1"))
    async with client.aio as aclient:
        for model_name in GEMINI_MODELS:
            try:
```

```
            response = await aclient.models.generate_content(
                model=model_name, contents=contents)
            text = (response.text or "").strip()
            if text:
                final_response = text
                logger.info(f"[{model_name}] success")
                break
            else:
                err = f"{model_name}: empty response"
                logger.warning(err)
                errors.append(err)
        except Exception as e:
            err = f"{model_name} failed: {type(e).__name__}: {e}"
            logger.warning(err)
            errors.append(err)
    # --- Якщо нічого не вдалося ---
    if not final_response:
        final_response = "⚠ Усі моделі недоступні. Спробуйте пізніше."
        logger.error("All fallback models failed:\n" + "\n".join(errors))
    # --- Оновлюємо історію + кеш ---
    await update_user_history(uid, "user", prompt)
    await update_user_history(uid, "model", final_response)
    await _set_cache(key, final_response)
    return final_response
# ВЕКТОРНИЙ ПОШУК
async def vector_context_from_chroma(query_text: str, k: int = 4) -> str:
    if  db.async_chroma_collection is None:
        return ""
    try:
        results = await db.async_chroma_collection.query(query_texts=[query_text], n_results=k)
    except Exception as e:
        logger.warning(f"Async Chroma search error: {e}")
        return ""
    if not results or not results.get("documents"):
        return ""
```

```
    snippets = []
    docs = results["documents"][0]
    metas = results["metadatas"][0]
    for doc, meta in zip(docs, metas):
        part = (...)
        snippets.append(part)
    return "\n\nСемантично схожі товари:\n" + "\n".join(snippets)
# АНАЛІЗ ДІЙ
async def analyze_action(uid: int, msg: str):
    p = f"""..."""
    r = await gemini_call(uid, p)
    return _safe_json_extract(r)
# Аналіз повідомлення (Mongo pipeline)
async def analyze_message(uid: int, msg: str, action: str, template_name: str):
    fields = available_fields.get(template_name, [])
    meta = collection_meta.get(template_name, {})
    meta_short = {k: v[:5] if isinstance(v, list) else v for k, v in meta.items()}
    p = f"""..."""
    r = await gemini_call(uid, p)
    pipeline = _safe_json_extract(r)
    if isinstance(pipeline, dict):
        pipeline = [pipeline]
    if not isinstance(pipeline, list):
        return None
    collection_name = TEMPLATE_TO_COLLECTION.get(template_name)
    def _clean_pipeline(pipeline: list, collection_name: str): ...
    pipeline = _clean_pipeline(pipeline, collection_name)
    return pipeline
async def respond_to_data(uid: int, data: Any, msg: str):
    if isinstance(data, list): data = data[:5]
    json_data = json.dumps(data, ensure_ascii=False, indent=2, default=str)
    MAX_JSON_CHARS = 1200
    if len(json_data) > MAX_JSON_CHARS:
        json_data = json_data[:MAX_JSON_CHARS] + "... (скорочено)"
    vec_ctx = await vector_context_from_chroma(msg)
```

```
    prompt = f"""..."""
    reply = await gemini_call(uid, prompt, use_history=True)
    return reply[:REPLY_SAFE_LIMIT]
async def respond_to_other(uid: int, msg: str):
    p = f"""..."""
    return await gemini_call(uid, p, use_history=False)
```

5 Файл db.py

```
from motor.motor_asyncio import AsyncIOMotorClient
from config import MONGO_URI
import chromadb
from chromadb.config import Settings
client = AsyncIOMotorClient(MONGO_URI)
db_mongo = client["dniprom"]
orders = db_mongo["orders"]
products = db_mongo["products"]
stores = db_mongo["stores"]
services = db_mongo["service_requests"]
sessions = db_mongo["sessions"]
async_chroma = None
async_chroma_collection = None
async def init_chroma():
    global async_chroma, async_chroma_collection
    async_chroma = await chromadb.AsyncHttpClient(
        host="localhost", port=8000, settings=Settings())
    async_chroma_collection = await async_chroma.get_or_create_collection(name="products_rag")
    print("[Chroma] Async client initialized")
```

6 Файл order_fsm.py

```
from aiogram.fsm.state import State, StatesGroup
from aiogram import types, Router, F
from aiogram.fsm.context import FSMContext
from aiogram.types import InlineKeyboardButton, InlineKeyboardMarkup, CallbackQuery
from db import products, orders
```

```
from bson.objectid import ObjectId
from datetime import datetime
from aiogram.filters import Command
class OrderStates(StatesGroup):
    waiting_for_category = State()
    waiting_for_product = State()
    waiting_for_price_confirmation = State()
    waiting_for_quantity = State()
    waiting_for_confirmation = State()
router = Router()
# Глобальний словник для мапування короткого ID кнопки на повну назву категорії
category_map = {}
CATEGORY_PREFIX = "cat" # Префікс
PAGE_SIZE = 5
### 1. СТАРТ: обрати категорію ###
async def start_order(message: types.Message, state: FSMContext):
    data = await state.get_data()
    old_msg = data.get("active_message")
    if old_msg:
        try: await message.bot.edit_message_text(...)
        except: pass
    await state.clear()  # Повний reset FSM
    await state.set_state(OrderStates.waiting_for_category)
    global category_map
    category_map.clear()
    categories = await products.distinct("category")
    keyboard_buttons = []
    for i, cat in enumerate(categories):
        short_id = f"{CATEGORY_PREFIX}{i}"
        category_map[short_id] = cat
        keyboard_buttons.append([
            InlineKeyboardButton(text=cat, callback_data=f"category:{short_id}")])
    keyboard = InlineKeyboardMarkup(inline_keyboard=keyboard_buttons)
    sent = await message.reply("📂 Оберіть категорію товару:", reply_markup=keyboard)
    await state.update_data(active_message=sent.message_id)
```

```
### 2. Вибір продукту з категорії ###
@router.callback_query(F.data.startswith("category:"))
async def select_category(callback: CallbackQuery, state: FSMContext):
    global category_map
    short_id = callback.data.split(":")[1]
    category = category_map.get(short_id)
    if not category:
        await callback.message.edit_text("⚠ Категорію не знайдено.")
        await state.update_data(active_message=callback.message.message_id)
        return
    # Загальна кількість товарів у категорії
    total_count = await products.count_documents({"category": category})
    if total_count == 0:
        await callback.message.edit_text("⚠ У цій категорії немає товарів.")
        await state.update_data(active_message=callback.message.message_id)
        return
    await state.update_data(category=category, page=0, total_count=total_count)
    # Показуємо першу сторінку
    await state.set_state(OrderStates.waiting_for_product)
    await render_product_page(callback, state)
async def render_product_page(callback: CallbackQuery, state: FSMContext):
    data = await state.get_data()
    category = data["category"]
    page = data["page"]
    total_count = data["total_count"]
    start = page * PAGE_SIZE
    # Motor: skip + limit
    cursor = (products.find({"category": category}).skip(start).limit(PAGE_SIZE))
    products_list = await cursor.to_list(length=PAGE_SIZE)
    # Формуємо кнопки товарів
    keyboard_rows = [[InlineKeyboardButton(
            text=f"{p['name']} – {p['price']}₴", callback_data=f"product:{str(p['_id'])}")]
        for p in products_list]
    # ---- кнопки пагінації ----
    pagination = []
```

```
    if page > 0:
        pagination.append(InlineKeyboardButton(text="⬅", callback_data=f"prodpage:{page-1}"))
    if (page + 1) * PAGE_SIZE < total_count:
        pagination.append(InlineKeyboardButton(text="➡", callback_data=f"prodpage:{page+1}"))
    if pagination:
        keyboard_rows.append(pagination)
    keyboard = InlineKeyboardMarkup(inline_keyboard=keyboard_rows)
    await callback.message.edit_text(...)
    await state.update_data(active_message=callback.message.message_id)
@router.callback_query(F.data.startswith("prodpage:"))
async def paginate_products(callback: CallbackQuery, state: FSMContext):
    new_page = int(callback.data.split(":")[1])
    data = await state.get_data()
    total_pages = (data["total_count"] - 1) // PAGE_SIZE
    if new_page < 0 or new_page > total_pages:
        await callback.answer("⚠ Сторінка недоступна.")
        return
    await state.update_data(page=new_page)
    await render_product_page(callback, state)
### 3. Підтвердження ціни ###
@router.callback_query(F.data.startswith("product:"))
async def select_product(callback: CallbackQuery, state: FSMContext):
    product_id = callback.data.split(":")[1]
    product = await products.find_one({"_id": ObjectId(product_id)})
    if not product:
        await callback.message.edit_text("⚠ Товар не знайдено.")
        await state.update_data(active_message=callback.message.message_id)
        return
    if product.get("stock", 0) == 0:
        await callback.message.edit_text("⚠ На жаль, цей товар наразі відсутній в наявності.")
        await state.update_data(active_message=callback.message.message_id)
        return
    await state.update_data(product=product)
    price = product["price"]
```

```
    discount = product.get("discount", 0)
    final_price = round(price * (1 - discount))
    msg = (...)
    keyboard = InlineKeyboardMarkup(inline_keyboard=[
        [InlineKeyboardButton(text="Так", callback_data="confirm_price"),
            InlineKeyboardButton(text="Назад", callback_data="restart")]])
    await state.set_state(OrderStates.waiting_for_price_confirmation)
    await callback.message.edit_text(msg, reply_markup=keyboard)
    await state.update_data(active_message=callback.message.message_id)
### 4. Введення кількості ###
@router.callback_query(F.data.startswith("confirm_price"))
async def confirm_price(callback: CallbackQuery, state: FSMContext):
    data = await state.get_data()
    product = data["product"]
    stock = product["stock"]
    max_quantity = min(5, stock)
    keyboard = InlineKeyboardMarkup(inline_keyboard=[
        [InlineKeyboardButton(text=str(i), callback_data=f"quantity:{i}")]
        for i in range(1, max_quantity + 1)])
    await state.set_state(OrderStates.waiting_for_quantity)
    await callback.message.edit_text(f"🔢 Скільки одиниць '{product['name']}' ви хочете
замовити?", reply_markup=keyboard )
    await state.update_data(active_message=callback.message.message_id)
@router.callback_query(F.data.startswith("quantity:"))
async def set_quantity(callback: CallbackQuery, state: FSMContext):
    quantity = int(callback.data.split(":")[1])
    data = await state.get_data()
    product = data["product"]
    price = product["price"]
    discount = product.get("discount", 0)
    final_price = round(price * (1 - discount)) * quantity
    await state.update_data(quantity=quantity, total_price=final_price)
    keyboard = InlineKeyboardMarkup(inline_keyboard=[
        [InlineKeyboardButton(text="✅ Підтвердити замовлення", callback_data="place_order")],
        [InlineKeyboardButton(text="❌ Скасувати", callback_data="restart")]])
```

```
    await state.set_state(OrderStates.waiting_for_confirmation)
    await callback.message.edit_text(...)
    await state.update_data(active_message=callback.message.message_id)
### 5. Підтвердження та запис замовлення ###
@router.callback_query(F.data == "place_order")
async def place_order(callback: CallbackQuery, state: FSMContext):
    data = await state.get_data()
    user_id = str(callback.from_user.id)
    product = data["product"]
    quantity = data["quantity"]
    total_price = data["total_price"]
    last_order = await orders.find_one(sort=[("order_id", -1)])
    if last_order and last_order.get("order_id") and last_order["order_id"].isdigit():
        next_order_id = str(int(last_order["order_id"]) + 1)
    else: next_order_id = "1"
    order = {...}
    await orders.insert_one(order)
    await products.update_one({"_id": product["_id"]}, {"$inc": {"stock": -quantity}})
    await state.clear()
    await state.update_data(active_message=callback.message.message_id)
    await callback.message.edit_text(...)
@router.callback_query(F.data == "restart")
async def restart_order(callback: CallbackQuery, state: FSMContext):
    current_state = await state.get_state()
    if current_state is None:
        await callback.answer("Немає активної операції, яку можна скасувати.", show_alert=True)
        return
    await state.clear()
    await callback.message.edit_reply_markup(reply_markup=None)
    await callback.message.answer("❌ Замовлення скасовано. Ви можете почати спочатку.")
@router.message()
async def catch_all_messages(message: types.Message, state: FSMContext):
    current_state = await state.get_state()
    if current_state is not None:
        await message.reply("Використовуйте кнопки нижче для вибору.")
```