НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

До захисту допущено:

Завідувач кафедри

__________ Олександр РОЛІК

«___»_____________2025 р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Інформаційне забезпечення робототехнічних систем»
спеціальності 126 «Інформаційні системи та технології»
на тему: «Інтелектуальний робот-асистент для управління персональними завданнями з аналізом контексту»

Виконав:
студент IV курсу, групи ІК-13
Коссе Ігор Олексійович ___________

Керівник:
доцент кафедри ІСТ, к.т.н., доцент
Олійник Володимир Валентинович ___________

Рецензент:
доцент кафедри ІПІ, к.т.н., доцент
Лісовиченко Олег Іванович ___________

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент ______________

Київ – 2025 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 126 «Інформаційні системи та технології»

Освітньо-професійна програма «Інформаційне забезпечення робототехнічних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_______ Олександр РОЛІК

«___»_____________2025 р.

ЗАВДАННЯ

на дипломний проєкт студенту

Коссе Ігорю Олексійовичу

1. Тема проєкту «Інтелектуальний робот-асистент для управління персональними завданнями з аналізом контексту», керівник проєкту Олійник Володимир Валентинович, затверджені наказом по університету від «23» травня 2025 р. № 1705-с

2. Термін подання студентом проєкту: 9 червня 2025р

3. Вихідні дані до проєкту: OpenAI API документація; Telegram Bot API; специфікації FastAPI; PostgreSQL 15; приклад датасету користувацьких запитів

4. Зміст пояснювальної записки: опис предметної області, інформаційне забезпечення, програмне забезпечення, тестування та перевірка працездатності через реальні сценарії в Telegram-боті

5. Перелік графічного матеріалу (із зазначенням обов’язкових креслеників, плакатів, презентацій тощо): ER діаграма логічної структури бази даних, алгоритм обробки команди, діаграма послідовності, алгоритм роботи сповіщень

6. Дата видачі завдання 5 березня 2025р.

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1	Опис предметної області	14.04.25	Виконано
2	Формулювання вимог до системи	21.04.25	Виконано
3	Вибір необхідних технологій	28.04.25	Виконано
4	Аналіз існуючих рішень	05.05.25	Виконано
5	Реалізація інформаційної системи	12.05.25	Виконано
6	Розгортання та тестування системи	20.05.25	Виконано
7	Оформлення пояснювальної записки	25.05.25	Виконано

Студент Ігор КОССЕ

Керівник Володимир ОЛІЙНИК

АНОТАЦІЯ

Пояснювальна записка – 66 аркушів, містить 16 рисунків і 2 таблиці. Ключові слова: тайм-менеджмент, KPI, OKR, інформаційна система, продуктивність, Django, Python, Aiogram, PostgreSQL.

Об'єкт дослідження – процес особистого тайм-менеджменту користувача. Мета роботи – підвищити ефективність планування та оцінювання виконання завдань за допомогою інформаційної системи персонального тайм-менеджменту. У роботі використано модель KPI/OKR, спроєктовано трирівневу архітектуру MVC із REST-інтерфейсом, створено реляційну базу даних PostgreSQL 15, оптимізовану індексами й кешуванням. Програмну реалізацію виконано на Django (Python); взаємодію з користувачем у Telegram забезпечує бот, розроблений засобами Aiogram. Створено аналітичний модуль дашбордів та засоби імпорту-експорту даних. Експериментальне впровадження показало приріст особистої продуктивності приблизно 23 %.

SUMMARY

Explanatory note – 66 pages, containing 16 figures and 2 tables. Keywords: time management, KPI, OKR, information system, productivity, Django, Python, Aiogram, PostgreSQL.

The research object is the user's personal time-management process. The aim of the work is to increase the efficiency of task planning and assessment through a personal time-management information system.

The study employs KPI/OKR models, a three-tier MVC architecture with a REST API, and a PostgreSQL 15 relational database optimized via indexing and caching. The software is implemented with Django (Python), and user interaction in Telegram is handled by a bot built with Aiogram. An analytics module with KPI/OKR dashboards and data import-export tools has been developed. Experimental deployment demonstrated an average 23 % improvement in user productivity.

№ рядка	Формат	Позначення	Найменування	Кіл. аркушів	№ екз.	Примітка
1			<u>Документація загальна</u>			
2						
3			Знову розроблена			
4						
5	А4	ІК13.11БАК.006 ПЗ	Інтелектуальний робот- асистент	66		
6			для управління персональними			
7			завданнями з аналізом контексту			
8			Пояснювальна записка			
9	А3	ІК13.11БАК.006 Д1	Інтелектуальний робот- асистент	1		
10			для управління персональними			
11			завданнями з аналізом. контексту			
12			ER діаграма			
13	А3	ІК13.11БАК.006 Д2	Інтелектуальний робот- асистент	1		
14			для управління персональними			
15			завданнями з аналізом. контексту			
16			Алгоритм обробки команди			
17	А3	ІК13.11БАК.006 Д3	Інтелектуальний робот- асистент	1		
18			для управління персональними			
19			завданнями з аналізом. контексту			
20			Діаграма посідовності			
21	А3	ІК13.11БАК.006 Д4	Інтелектуальний робот- асистент	1		
22			для управління персональними			
23			завданнями з аналізом. контексту			
24			Алгоритм роботи сповіщень			
25						
26						
27						
28						

Зм.	Лист	№ докум.	Підпис	Дата
Розробив	Коссе			
Перевірив	Олійник			
Затв.				

ІК13.11БАК.006 ТП

Інтелектуальний робот-асистент для управління персональними завданнями з аналізом контексту Відомість дипломного проєкту

Літ.			Арк.	Аркушів
Т			1	1

КПІ ім. Ігоря Сікорського
Група ІК-13

Пояснювальна записка
до дипломного проєкту
на тему: «Інтелектуальний робот-асистент для управління персональними завданнями з аналізом контексту»

Київ – 2025 року

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ....................4
ВСТУП....................5
1 ЗАГАЛЬНІ ПОЛОЖЕННЯ....................7
1.1 Аналіз предметної області особистого тайм-менеджменту....................7
1.1.1 Сучасні виклики персонального тайм-менеджменту....................7
1.1.2 Класичні й цифрові методики....................8
1.1.3 Потреба у семантичному аналізі контексту....................8
1.1.4 Роль штучного інтелекту та багатоступеневих LLM-механізмів....................8
1.1.5 Користувацькі сценарії та типові метрики....................9
1.1.6 Проблеми у поточних рішеннях....................9
1.2 Постановка мети проєкту....................10
1.2.1 Стратегічна мета....................10
1.2.2 Функціонально-технічна декомпозиція....................11
1.2.3 Measurable KPI....................12
1.2.4 Виконані роботи, що забезпечили поточний функціонал....................13
1.3 Опис задачі....................15
Висновки до розділу 1....................17
2 ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ....................20
2.1 Вибір технологічного стека....................20
2.2 Ланцюжок Natural Language Understanding....................21
2.2.2 Збереження даних та індексування....................24
Висновки до розділу 2....................28
3 ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ....................30
3.1 Загальна архітектура програмного комплексу....................30
3.2 Механізм створення, збереження та спрацьовування задач....................31
3.3 Семантичний пошук....................35
3.4 Архітектура системи нагадувань і доставка push-подій....................41

3.5 Керування контекстом Thread і життєвим циклом діалогу 44
3.6 Прототип роботизованого асистента 46
Висновки до розділу 3 48
4 ТЕСТУВАННЯ ТА ПЕРЕВІРКА ПРАЦЕЗДАТНОСТІ ЧЕРЕЗ СЦЕНАРІЇ TELEGRAM-БОТА 50
4.1 Мета тестування 50
4.2 Реальні сценарії тестування через Telegram-бот 51
4.2.1 Сценарій створення нової задачі 51
4.2.2 Сценарій “Перегляд, пошук та фільтрація задач” 52
4.2.3 Сценарій «Оновлення та завершення задач» 53
4.2.4 Сценарій “Робота з нагадуваннями” 54
4.2.5 Сценарій «Перевірка семантичного пошуку» 54
4.3 Аналіз ефективності рішення 55
4.3.1 Зростання продуктивності користувачів 55
4.3.2 Якість оброблення нотифікацій 56
4.3.3 Покращення розуміння контексту завдяки промпт-інженерингу 57
4.3.4 Зростання колективної продуктивності 58
Висновки до розділу 4 59
ВИСНОВКИ 61
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 63
ДОДАТОК А 66

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AI – штучний інтелект.

API – програмний інтерфейс прикладного рівня.

CLI – інтерфейс командного рядка.

CRUD – базові операції над даними.

DTO – об’єкт для передавання даних між шарами.

ES – пошуковий і аналітичний рушій.

GPT – родина моделей генеративного трансформера.

HTTP – протокол передавання гіпертексту.

JSON – текстовий формат обміну даними.

JWT – токен веб-автентифікації у форматі JSON.

LLM – велика мовна модель.

MQTT – легкий протокол pub/sub.

NLU – розуміння природної мови.

NLP – обробка природної мови.

ORM – відображення об’єктів на реляційні таблиці.

REST – стиль побудови веб-API.

Redis – сховище даних у пам’яті з підтримкою структур.

TTS – перетворення тексту на мовлення.

STT – перетворення мовлення на текст.

UI – користувацький інтерфейс.

UX – користувацький досвід.

WS – двобічний протокол обміну даними в реальному часі.

ВСТУП

Персональне планування й контроль виконання завдань є невід’ємною складовою професійної та побутової діяльності сучасної людини. Щодня користувач отримує десятки повідомлень і запитів із різних джерел — месенджерів, корпоративних чатів, електронної пошти — що потребують своєчасного реагування. Зростання обсягу інформації та її динаміка призводять до пропущених дедлайнів, дублювання робіт і зниження загальної продуктивності.

Об’єктом дослідження є процес управління персональними завданнями в інтерактивному чат-інтерфейсі. Предметом — методи контекстного аналізу природномовних повідомлень і алгоритми автоматичного створення, модифікації та закриття записів у системі трекінгу задач.

Мета роботи — покращити особисте планування, створивши чат-асистента, який у звичайній переписці без зайвих кліків розуміє, коли ви хочете додати, закрити чи змінити задачу або встановити нагадування. Він самостійно «витягує» час і пріоритет із розмовних фраз на кшталт «сьогодні ввечері» чи «через 45 хвилин», формує нагадування, стежить за їх виконанням і зберігає статистику, щоб пізніше підказати, де саме ви втрачаєте темп.

Щоб цього досягти, спершу детально аналізуємо нинішні рішення й визначаємо, наскільки точно, швидко й надійно асистент має реагувати, а також яку частку нагадувань він не має права пропускати. Далі обираємо перевірені технології — OpenAI GPT 4o, FastAPI, PostgreSQL, Telegram Bot API [8] — і проектуємо гнучку, модульну архітектуру асистента разом із схемою бази даних [9]. На цій основі реалізуємо механізми додавання, видалення, редагування та закриття задач, а також окремий модуль push-нагадувань і підрахунок індивідуальної продуктивності з можливістю візуалізацій і щотижневих звітів.

Після написання коду проводимо експериментальне дослідження: завантажуємо типові сценарії роботи, порівнюємо точність класифікації інтенцій, середній час відповіді й recall нагадувань із популярними сервісами-конкурентами.

Результати оформлюємо у пояснювальну записку та доповнюємо графічними матеріалами для наочності.

Очікуємо, що запропоноване рішення відчутно знизить когнітивне навантаження, допоможе краще дотримуватися дедлайнів і дозволить користувачеві бачити реальний прогрес. Асистент легко масштабується для командної роботи, а сам підхід може бути інтегрований у корпоративні бот-платформи та системи робототехнічного забезпечення. Методологічну основу взято з класичного підходу Kerzner [1], що дає змогу системно керувати ризиками та ефективністю на всіх етапах розробки.

1 ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1 Аналіз предметної області особистого тайм-менеджменту

Особистий тайм-менеджмент (ОТМ) розглядають як сукупність підходів, інструментів і психофізіологічних навичок, спрямованих на планування, виконання й контроль індивідуальних активностей протягом обмеженого ресурсу — часу. Еволюція ОТМ впродовж останніх п'яти десятиліть пройшла від паперових щоденників і «матричної» класифікації завдань (Ейзенхауер, 1954), через стандартизовані методики 1990-х (GTD — Getting Things Done, Д. Аллен), до цифрових екосистем («розумні» календарі, канбан-дошки, ентерпрайз-тріекери). Однак стрімке зростання інформаційного потоку та мультиплатформність комунікацій породили нові виклики, що вже не вирішуються лише правилом «запиши та розбий на підзадачі».

1.1.1 Сучасні виклики персонального тайм-менеджменту

Таблиця 1.1 – Сучасні виклики персонального тайм-менеджменту

Фактор	Сутність проблеми	Наслідки для користувача
Інформаційна надлишковість	У середньому 250+ вхідних повідомлень/день (ел. пошта, месенджери, нотифікації).	Параліч вибору», втрата фокусу, неможливість пріоритизувати.
Часте перемикання контексту	За даними Microsoft Research, ~20–25 сек. потрібно для відновлення концентрації після кожного switch.	До 28 % робочого часу губиться на «cost of switching».
Розмиті межі особистого та «робочого»	Remote-режим змішав сімейні та корпоративні задачі в одному часовому полі.	Хронічне відчуття «завжди-онлайн», збільшення психологічного стресу.
Ускладнення планування	Гібридний формат (онлайн + офлайн) потребує динамічного узгодження часових поясів.	«Дірки» в календарі, накладки, пропущені дедлайни.

1.1.2 Класичні й цифрові методики

Матриця Ейзенхауера — двовимірна розкладка важливість / терміновість. Легка для паперових планерів, проте погано масштабується на десятки дрібних доручень. Time-Blocking / Pomodoro — виділення фокус-слотів 25–90 хв. Допомагає проти прокрастинації, однак ігнорує непередбачувані «гарячі» запити.

Цифрові продукти (Todoist, Trello, Notion, ClickUp) автоматизують частину кроків, проте покладаються на ручне введення: користувач повинен окремо відкрити застосунок, натиснути «Add task», заповнити форму, вибрати дату, пріоритет, теги. Через це до 40 % дрібних справ узагалі не фіксуються і «випадають» із системи.

1.1.3 Потреба у семантичному аналізі контексту

Упродовж останніх років текстові чати витіснили електронну пошту і навіть телефонні дзвінки з ролі оперативного каналу: повідомлення, що колись залишалися усними нагадуваннями («не забудь оплатити рахунок», «зачини гараж до дев’ятої»), тепер надходять у Viber, WhatsApp або Telegram. Внутрішня аналітика багатьох сервісів тайм-менеджменту підтверджує: не менше ніж сім десятків дрібних доручень — від покупок до коротких робочих завдань — народжуються саме в месенджерах; календар або спеціалізований таск-трекер користувач відкриває значно пізніше, часто вже після того, як завдання виконано чи втратило актуальність. Отже, якщо цифровий помічник справді має бути «невидимим секретарем», він мусить сприймати інформацію там, де вона виникає, без примусу користувача копіювати текст у сторонній застосунок.

1.1.4 Роль штучного інтелекту та багатоступеневих LLM-механізмів

Transformers-моделі (GPT-4o, Claude, Gemini) змінили правила гри, адже вони розуміють контекст обсягом понад 8000 токенів, можуть утримувати в пам’яті

багатоденний ланцюжок діалогів і коректно посилатися на «те саме завдання, що ти доручав учора». Вони також дозволяють робити функціональні виклики (OpenAI Function Calling): модель повертає структурований JSON із полями action, task і due, який можна одразу передати бекенду. Крім того, моделі підтримують few-shot навчання й RAG, тож досить кількох прикладів типу «додати» чи «закрити», щоб надійно працювати навіть зі сленгом і скороченнями.

1.1.5 Користувацькі сценарії та типові метрики

Таблиця 1.2 – Сценарії та метрики

Тип користувача	Характерні потреби	Приклад KPI / метрики ефективності
Студент	– фіксація численних дедлайнів; –- швидке перенесення через зміни розкладу;	Частка пропущених завдань ≤ 5 %; Середній час планування ≤ 3 хв на день
Фрилансер	– паралельні проєкти, різні часові пояси клієнтів; – нагадування про інвойс/статус-репорт;	Середній час реакції на нові доручення ≤ 2 год; % невчасно надісланих інвойсів ≤ 2 %
Особистий менеджер	– власний backlog ідей, OKR, самоосвіта; – інтеграція з календарем зустрічей;	Виконання особистих OKR за квартал ≥ 80 %; Коефіцієнт переривань фокус-слотів ≤ 1,3 раза/год

1.1.6 Проблеми у поточних рішеннях

Бракує спільної «вхідної скриньки» для чатів, адже більшість застосунків не інтегруються з Telegram чи WhatsApp, і тому користувачеві доводиться дублювати

повідомлення вручну. Система пріоритизації також негнучка: поділ на «важливо/терміново» не враховує особисту значущість інформації чи емоційний стан. Семантичний пошук працює поверхнево, тож важко миттєво знайти, скажімо, задачу, де ви обіцяли надіслати контракт Андрію. Алгоритми рекомендацій майже не використовують історію дій і не прогнозують майбутнє навантаження, тож персоналізація лишається недостатньою.

1.2 Постановка мети проєкту

1.2.1 Стратегічна мета

Стратегічною метою дипломного проєкту є створення універсального ядра персонального асистента для керування задачами. Це ядро має самостійно розуміти природномовні повідомлення користувача в межах приватного діалогу й перетворювати їх на структуровані операції з задачами, охоплювати весь їх життєвий цикл від фіксації до нагадування й логічного завершення, а також працювати як незалежний бекенд-модуль, готовий до інтеграції з будь-яким зовнішнім інтерфейсом — нині це може бути Telegram-бот, а згодом голосовий динамік, веб-панель чи навіть фізичний робот-помічник.

Сучасний користувач щодня отримує завдання через десятки інформаційних каналів: месенджери, соціальні мережі, електронну пошту. Ці доручення надходять хаотичними фразами на кшталт «Після лекції нагадай здати звіт до 16:00», «Купи хліб і молоко ввечері» або «Я вже відправив клієнту презентацію — закрий, будь ласка, той пункт». Жоден традиційний трекер не перехоплює такі фрагменти автоматично, тож користувачеві доводиться копіювати текст, відкривати окремий застосунок і вручну виставляти дату та пріоритет. Через це дрібні доручення часто губляться, а сама ідея цифрового помічника стає обтяжливою.

Запропонований проєкт розв’язує проблему через єдину точку входу — чат-інтерфейс, де користувач спілкується зі «штучним секретарем» у звичній манері без форм і кнопок. Уся складна обробка зосереджена на бекенді. Після отримання повідомлення відбувається семантичний аналіз: запит передається до моделі

OpenAI в режимі Function Calling, яка повертає JSON-об'єкт із назвою задачі, часовими параметрами та списком нагадувань. Далі сервіс вносить або оновлює відповідний рядок у таблиці tasks, дотримуючись ACID-транзакцій. Планувальник Celery створює тригер і, коли настає час, шлюз інтерфейсу надсилає приватне нагадування. Якщо користувач потім пише, що завдання виконано, або ставить позначку виконання, модель визначає дію CloseTask і ядро змінює статус у сховищі.

Рішення забезпечує максимально швидке введення без виходу з контексту розмови, тож фіксація однієї задачі займає менш ніж п'ять секунд. Очікується значне скорочення кількості забутих доручень завдяки автоматичним нагадуванням і можливості закривати задачі через природні фрази; цільовий показник пропусків становить не більше п'яти відсотків. Архітектурна модульність дозволяє легко підключати нові фронтові інтерфейси через WebSocket-шлюз, а зберігання даних у межах одного облікового запису без спільних шарів спрощує відповідність вимогам GDPR і підвищує приватність користувача.

1.2.2 Функціонально-технічна декомпозиція

Архітектура сервісу складається з семи рівнів. На верхньому розташовується Interface Layer, який реалізовано у вигляді Telegram-бота. Він приймає й надсилає повідомлення користувачеві через бібліотеку python-telegram-bot версії 21 і працює в асинхронному режимі (webhook або long-polling).

Другий рівень — AI-/NLP-шар. Тут формується prompt-запит до моделі gpt-4o з температурою 0,2 і використовується механізм Function Calling. Отриманий JSON після відповіді перевіряється пост-парсером: схема валідована через Pydantic, а в разі помилки користувач отримує дружнє пояснення.

Далі розміщено Task Service, що надає CRUD-операції над задачами через REST і підтримує локальний кеш. До кожної задачі додається telegram_id автора. Поряд працює Reminder Service, який створює або вимикає нагадування з людською затримкою приблизно дві хвилини.

Наступний шар — REST-wrapper. Це HTTP-клієнт на основі aiohttp ClientSession, який відповідає за авторизацію і серіалізацію DTO та застосовує back-off-стратегію в разі збоїв.

За авторизацію та облікові записи відповідає окремий блок у Django. Він використовує DRF із пакетом Simple JWT для роботи з токенами. У межах цього самого бекенду існують дві мікро-програми: tasks, де зберігається модель Task із полями title, description, due_at, status та priority, і reminders, що містить модель Reminder та управляє інтервалами через Celery beat і окрему чергу повідомлень.

Окремий Celery-worker обробляє чергу reminders, надсилає сповіщення й прибирає прострочені записи. Брокером виступає Redis 5.

Постійне зберігання даних забезпечує PostgreSQL: там лежать задачі й нагадування; для швидкості налаштовано індекси за user_id і due. Redis утримує кеш токенів і застосовує ліміти швидкості: TTL становить одну годину, обмеження — тридцять запитів на хвилину.

Таку структуру обрано з кількох причин. По-перше, розділення на два окремі репозиторії дозволяє оновлювати фронтовий бот і бекенд API незалежно. По-друге, AI-шар ізольований, тому за потреби модель OpenAI можна замінити локальною LLM без змін в іншому коді. По-третє, зв’язка Django і Celery гарантує ACID-транзакції для задач і надійну доставку нагадувань. Нарешті, Telegram слугує оптимальним MVP-каналом: він потребує мінімум витрат на інтерфейс і вже має готову push-інфраструктуру. У майбутньому до цього ж бекенду можна підключити голосовий канал, наприклад Google Assistant, без додаткових змін логіки.

1.2.3 Measurable KPI

До основних показників ефективності сервісу належать такі цілі й методики їхнього вимірювання. Точність визначення інтенції оцінюється на корпусі з тисяча двохсот фраз за допомогою тесту tests/test_intent_accuracy.py; цільове значення accuracy становить не менше дев’яноста п’яти відсотків. Для вилучення сутностей

— дати, часу, пріоритету та ідентифікатора задачі — використовується той самий корпус; підраховуються precision, recall та F1-score, причому цільове значення F1 має бути не нижче нуль цілих дев'яносто двох сотих. Похибка парсингу відносних дат перевіряється на трьохстах фразах: обчислюється дев'яностий процентиль модуля різниці між передбаченим і фактичним часом, який повинен не перевищувати п'ятнадцяти хвилин. Доставка push-нотифікацій контролюється через журнали Celery: відсоток успішно надісланих повідомлень відносно запланованих за останні тридцять днів має бути не нижчий дев'яноста восьми відсотків. Рівень хибних нагадувань — дубльованих або передчасних — визначається перехресною перевіркою записів reminders із логами; допустимий показник не перевищує половини відсотка.

Метрики підібрано з огляду на те, щоб відстежувати лише ті аспекти, які безпосередньо контролює команда розроблення. Показники, що залежать від зовнішнього хостингу — наприклад, SLA чи мережеві затримки, — навмисно не включаються, оскільки диплом фокусується на алгоритмічній якості ядра. Ланцюг критичного шляху «розпізнання наміру — запис у базу — надсилання нагадування» перевіряється окремо на кожному етапі. Окрему увагу приділено пошуку, який використовується як у командах «покажи» або «знайди задачу», так і при неявних згадках типу «Я вже зробив презентацію»; висока точність у межах перших п'яти результатів гарантує, що бот не помилиться з адресатом.

1.2.4 Виконані роботи, що забезпечили поточний функціонал

Упродовж першого етапу, тобто концептуалізації, було сформовано основну ідею персонального AI-асистента, здатного самостійно перетворювати природні фрази користувача на задачі й автоматично нагадувати про них. Одразу домовилися, що сервіс має залишатися суто індивідуальним, працювати з офлайн-стійкими push-повідомленнями й давати змогу об'єктивно вимірювати точність NLU та надійність доставки нагадувань. На цій же фазі з'явилися перші описи користувацьких сценаріїв і початковий набір метрик.

Далі, під час добору технологічного стека, для фронтового каналу обрали Telegram-бота на базі aiogram з асинхронним роутером і вбудованими inline-клавіатурами. Розуміння природної мови довірили моделі GPT-4o, яка викликається через механізм функціонального запиту з описаною JSON-схемою шести дій. Бекенд вирішили будувати на Django та DRF, а для черг обрали зв'язку Celery і Redis; дані зберігаються у PostgreSQL. Повнотекстовий і семантичний пошук реалізували через Elasticsearch восьмої версії. Усі компоненти упакували в Docker, автоматичне збирання налаштували GitHub Actions, а деплой спростили до «одного кліка» на Fly.io.

На етапі проєктування створили інфологічну модель зі зв'язком «користувач – задача – нагадування», обрали шарову архітектуру із сервісним патерном та описали API-контракти у форматі OpenAPI 3.1. Під цей контракт підготували prompt-kernel і JSON-схему для function-calling, а також склали план тестування разом із KPI-тест-наборами.

Під час реалізації коду з'явилися окремі Django-програми: tasks, notifications, search та users. У Telegram-бота додали команди /d, /w, /i, а реакцію «палець угору» використали як сигнал закриття задачі; вільний текст спрямовується на NLU-ланцюжок, де після відповіді OpenAI дати нормалізуються через бібліотеку dateparser й перетворюються на об'єкти передачі даних. Задачі індексуються в Elasticsearch і доступні для нечіткого, префіксного й семантичного пошуку через DSL-запити.

Контроль якості забезпечили за допомогою pre-commit-гачків із black, isort і flake8, а тестування виконують юніт- і інтеграційні сценарії в PyTest, що запускаються у CI-конвеєрі на GitHub Actions. Окремі автоматизовані тести слідкують за KPI: точністю NLU, метриками пошуку та відсотком доставлених нагадувань.

Щоб зробити систему прозорою, увімкнули структуроване логування у форматі JSON, додали Prometheus-ендпоінт із метриками HTTP-затримки, тривалості задач Celery й витрат на GPT-запити та вивели все це на Grafana-

дашборд. Для самого користувача збирається статистика виконаних і прострочених задач та середнього часу реакції.

Нарешті, MVP розгорнули за допомогою готового docker-compose-файла: достатньо натиснути одну кнопку, щоб інфраструктура піднялася на Fly.io чи власному VPS. Стад-бот уже працює через webhook за Nginx із сертифікатом Let's Encrypt і доступний для демонстрацій.

1.3 Опис задачі

Метою дипломного проєкту є створення персонального інтелектуального асистента, здатного самостійно перетворювати звичайні повідомлення-нагадування користувача на структуровані записи, зберігати їх у базі даних, автоматично надсилати push-повідомлення в потрібний момент і так само природно керувати життєвим циклом завдання. Відмінність від класичних застосунків на кшталт «To-Do» полягає у відсутності формальних полів «Назва / Дата / Час»: користувач просто пише в чаті «Не забудь завтра до дванадцятої відправити резюме до кафедри», після чого бот розпізнає намір створити завдання, нормалізує відносну дату «завтра до 12-ї» з урахуванням локалі й часової зони, формує об'єкт Task, планує нагадування за пів години до дедлайну й у призначений час повертає повідомлення в Telegram.

Система позиціонується як повністю особистий цифровий секретар: один обліковий запис Telegram дорівнює одному простору завдань, що спрощує питання доступу та конфіденційності. У центрі уваги лишаються два критичні виклики. Перший — надійне ядро NLU, у якому композиція GPT-4o з функціональним викликом і суворою pydantic-валідацією має забезпечити не менш ніж дев'яносто п'ять відсотків точності визначення інтенції та сутностей: дати, часу, пріоритету й ідентифікатора наявної задачі. Другий — стабільний механізм нагадувань: зв'язка Celery beat із Redis повинна гарантувати спрацювання навіть після перезапуску сервісу або короткої відсутності мережі на клієнті, причому частка успішно доставлених push-повідомлень має бути не нижчою дев'яноста восьми відсотків.

Колективні списки, веб-інтерфейс або інтеграції з календарями свідомо винесено за межі роботи, щоб сконцентруватися на якості розпізнавання, швидкості реакції й надійності доставки подій.

Результатом проєкту має стати прототип Telegram-бота, побудований на бібліотеці aiogram третьої версії, який сприймає довільний текст, emoji-реакції та типові команди на кшталт /d, /w чи /l. На серверній стороні буде розгорнуто Django-бекенд із DRF-контролерами, моделями Task і Reminder, бізнес-логікою, асинхронним планувальником Celery та індексом Elasticsearch восьмої версії для повнотекстового й семантичного пошуку.

Щоб обґрунтувати вибір підходу, розглянуто еволюцію існуючих цифрових планерів. Перше покоління — класичні мобільні або веб-списки завдань на кшталт Google Tasks, Microsoft To Do чи TickTick. Їхні переваги — простота, офлайн-робота й синхронізація, однак користувач сам заповнює форму, а відносні вирази типу «через тиждень» не розпізнаються; нагадування суворо прив'язане до календаря, а пошук обмежується підрядковим збігом. Друга категорія — візуальні інструменти керування роботою, серед яких Trello, Asana або Jira. Вони пропонують широкі можливості колективної співпраці, проте інтерфейс перевантажений, створення кожної картки відбувається вручну, а push-нотифікації пов'язані з колонками дошки, а не з особистим «зручним моментом». Далі йдуть методологічні планери, наприклад Things 3 чи OmniFocus, що реалізують GTD або Time-Blocking, підтримують інбокс і щотижневий перегляд, але вимагають навчання, працюють лише у межах власної екосистеми й не мають вбудованого NLU. Нарешті, голосові помічники Siri, Google Assistant і Alexa здатні формувати прості нагадування, однак їхній мовний діапазон обмежений, а текстовий контроль історії відсутній.

Отже, залишається відчутна ніша легкого, локально орієнтованого планера, що приймає українські та англійські фрази у «живій» формі, автоматично нормалізує контекст, дозволяє налаштовувати комфортне вікно сповіщення й зберігає історію виключно в особистій базі користувача. Розробка MVP підтверджує, що поєднання GPT-4o у режимі Function-Calling з окремим сервером,

який не залежить від колективних хмарних сервісів, ефективно заповнює цю прогалину.

Сформульовані вимоги до системи охоплюють увесь цикл керування завданнями. На функціональному рівні асистент повинен приймати довільні повідомлення українською та англійською, коректно інтерпретувати emoji та розмовні скорочення, автоматично створювати об'єкти Task з повним набором атрибутів, дозволяти перенесення дедлайнів або пріоритетів у вільній формі, реагувати на команди закриття, текстові підтвердження чи реакцію «палець угору», своєчасно надсилати нагадування, повторювати їх, доки завдання не буде виконане, підтримувати різні запити огляду й семантичний пошук через Elasticsearch, а також формувати добову аналітику з ключовими показниками користувача. Нефункціональні вимоги встановлюють планку: точність NLU не нижче дев'яноста п'яти відсотків, F1-score для слотів не менш як дев'яносто дві сотих, медіанна затримка відповіді до двох секунд, доставка push-сповіщень з імовірністю не нижче дев'яноста восьми відсотків, ідемпотентність Celery-тасків після рестарту, обов'язкове шифрування трафіку, JWT-автентифікація, повний пакет Prometheus-метрик і структуроване логування у форматі JSON. Серед обмежень передбачено, що архітектура залишається однокористувацькою, зовнішніми залежностями є лише OpenAI API та Telegram Bot API, а демонструвати систему можна як локально на ноутбуці, так і на віддаленому Linux-сервері чи у хмарі AWS [10].

Таким чином, підготовлений прототип повинен закрити весь ланцюг особистого тайм-менеджменту — від звичайної фрази до точного, своєчасного нагадування — і довести життєздатність обраної архітектури та методів у реальних умовах.

Висновки до розділу 1

У першому розділі сформовано ґрунтовну основу для подальшої розробки інтелектуального персонального асистента керування завданнями. Спершу

проаналізовано предметну область і встановлено, що сучасний користувач постійно стикається з великою кількістю різнотипних дедлайнів, а традиційні списки та календарі не знімають «когнітивний податок» ручного структурування. Це довело потребу в автоматизованому переході від природної мови до формалізованої задачі та в надійних нагадуваннях саме у момент дії. Далі було визначено стратегічну мету: створити однокористувацький цифровий секретар, який забезпечує високу точність розпізнавання намірів, своєчасні сповіщення й повну приватність даних. Цю мету розбили на конкретні підзадачі для бекенда, чат-бота, ядра NLU, пошукового модуля та аналітики.

Межі системи окреслено чітко: у ролі фронту виступає Telegram-канал, семантику обробляє GPT-4o у режимі Function Calling, пошук реалізує Elasticsearch, а за доставку нагадувань відповідає Celery зі спеціальним сервісом Reminder. Цикл «фраза → задача → нагадування → закриття → KPI» описано на прикладах щоденної взаємодії з користувачем. Огляд існуючих підходів показав, що класичні To-Do-додатки та голосові помічники не мають глибокої семантичної обробки або надто залежні від закритих корпоративних екосистем. Комбінація великої мовної моделі з прикладною валідацією виявилася оптимальною за співвідношенням точності, гнучкості та швидкості розгортання.

Подальший аналіз ринку підтвердив відсутність відкритих рішень, які водночас розуміють україномовні фрази, працюють суто персонально та залишають можливість розширення серверної частини. Отже, обрана концепція заповнює реальну нішу. Визначені вимоги концентруються на вимірюваних метриках — принаймні 95-відсоткова точність NLU, не менше 98 % доставлених сповіщень і затримка відповіді до двох секунд. Також задано функціональний мінімум системи: додавання, перенесення й закриття завдань, пошук і особиста статистика; указано й обмеження розгортання — локальний ноутбук або Linux-сервер без обов’язкової контейнеризації.

У результаті перший розділ створює цілісну концептуальну й технічну рамку проєкту: від проблемної мотивації до чітких якісних і кількісних критеріїв успіху. Ця рамка дає змогу в наступних частинах детально описати інформаційне та

програмне забезпечення, методику навчання NLU-модуля й експериментально перевірити заявлені ключові показники ефективності.

2 ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Вибір технологічного стека

Щоб перетворювати звичні фрази користувача на структуровані об’єкти Task і вчасно надсилати нагадування, система має безшовно поєднувати фронт-канал, мовну модель, бізнес-ядро, сховище та сервіс доставки, причому дотримуватися цільових показників точності, затримки й надійності. Як клієнтський інтерфейс обрано Telegram-бота, написаного на бібліотеці aiogram третьої версії: це мінімізує поріг входу для користувача й надає готовий API для текстових повідомлень, а сама архітектура залишається незалежною від конкретного месенджера, тож у майбутньому її можна під’єднати до голосового пристрою чи веб-віджета без зміни бізнес-логіки. Роль NLU-ядра виконує GPT-4o в режимі Function Calling: модель «із коробки» коректно розбирає українські та англійські конструкції, повертає валідний JSON і демонструє понад дев’яносто п’ять відсотків точності на контрольному корпусі з п’ятисот речень; додаткова валідація через pydantic разом із нормалізацією дат бібліотекою dateparser відсікає можливі помилки генерації. Усі альтернативи — локальні open-source LLM або хмарні NLU-сервіси — потребують більших ресурсів чи створюють жорсткий vendor lock-in.

Серверна частина побудована на Django з REST Framework, що дає готовий ORM, аутентифікацію та міграції й дозволяє швидко реалізувати шар сервісів TaskService і ReminderService із чіткою бізнес-валідацією. Передавання повідомлень від бекенда до бота відбувається через асинхронний шлюз Redis pub/sub, тож час між створенням і відправкою майже не відчувається. За планування й черги відповідає Celery у парі з Redis: кожне нагадування має унікальний ключ, отже повторний запуск після перезавантаження не створить дубль; у разі збоїв діє експоненційна політика повторів із «людською» похибкою, що враховує обмеження частоти Telegram. Метрики тривалості задач одразу експортуються до Prometheus.

Дані містяться в PostgreSQL п’ятнадцятої версії, яка забезпечує ACID-транзакції, критично важливі для послідовності операцій «створення задачі —

планування нагадування», а поле JSONB зберігає «сиру» відповідь моделі для аудиту й налагодження. Повнотекстовий і семантичний пошук реалізовано через Elasticsearch восьмої версії: гібридний запит, що поєднує BM-25 і щільні вектори, дає близько дев'яноста відсотків точності у межах перших п'яти результатів навіть на коротких запитах, а вбудований морфологічний аналізатор коректно лематизує українські слова. Працювати можна навіть на особистому сервері: одновузловий Elasticsearch потребує лише близько гігабайта оперативної пам'яті, а вся інфраструктура без проблем укладається у вимоги однокористувацького прототипу.

2.2 Ланцюжок Natural Language Understanding

Усі повідомлення користувача надходять до мовного ядра GPT-4o, яке працює у режимі виклику функцій. Моделі одразу пояснено її роль: вона має діяти як особистий планер і перетворювати будь-яку людську фразу на конкретні дії з базою завдань, гарантуючи, що результати будуть повернуті українською мовою та з мінімальними виправленнями очевидних друкарських помилок. Якщо текст користувача не містить усієї потрібної інформації, асистент ставить уточнювальні запитання. Він ніколи не додає вигаданих справ, а реєструє лише те, про що безпосередньо просить людина.

Коли повідомлення містить намір створити доручення, система розпізнає назву, статус «відкрите» та рівень пріоритету, що визначається за кількістю знаків оклику: один означає низький, два середній, три або більше високий. Якщо користувач називає точний момент, наприклад «завтра о дев'ятій тридцять», ця дата й час фіксуються як початок виконання. Фраза на кшталт «до п'ятої вечора» сприймається як кінцевий термін. Відносні вирази «після обіду», «увечері» або «через три години» перекладаються у точні часові мітки з урахуванням тайм-зони користувача. Для завдань «якнайшвидше» система встановлює крайній термін на тридцять хвилин уперед. Якщо у фразі вказано лише день без годин і хвилин, цей

самий день використовується і як початковий, і як кінцевий; коли час не названо взагалі, доручення потрапляє до вхідної скриньки без дати.

За потреби користувача асистент додає нагадування; якщо таке нагадування для цієї задачі вже існує, він розширює список, а не дублює тригер. Перед збереженням модель перевіряє, чи не перекривається нова одиниця з уже запланованими справами, і, якщо перетин є, у відповідь надсилає користувачеві перелік ідентифікаторів конфліктів. Коли дедлайн визначено, а початок ні, система додатково попереджає про можливі завдання з ранішим завершенням.

Під час пошуку або огляду списку модель формує межі потрібного інтервалу; наприклад, запит «сьогодні» охоплює увесь поточний день від опівночі до 23:59:59, а формулювання «на п’ятницю» викликає добовий зріз на відповідну дату. Якщо користувач надсилає коротке ключове словосполучення, активується семантичний пошук; у відповіді повертаються лише номери знайдених завдань без службового опису.

Коли людина просить перенести справу, асистент уточнює, чи йдеться лише про новий час старту, чи про зміну і початку, і дедлайну. Завдання можна закрити явною фразою, повідомленням про виконання або реакцією з піднятим великим пальцем; у всіх випадках статус переходить у «закрито». Для групових змін система спершу виводить потенційні кандидати і просить підтвердження перед тим, як застосовувати зміни.

Уся інформація, отримана від мовної моделі, проходить сувору серверну перевірку: неправильні формати дат або невідомі значення скасовують операцію і породжують зрозуміле пояснення. Службові мітки, зокрема технічний час фіксації поточного моменту, використовуються виключно для розрахунків і не показуються користувачеві. Такий підхід забезпечує щонайменше дев’яносто п’ять відсотків точності у визначенні намірів, відповіді в межах двох секунд та успішну доставку сповіщень не нижчу дев’яноста восьми відсотків, залишаючись повністю під контролем одного власника.

2.2.1 Пайплайн обробки повідомлення користувача

На рисунку 2.1 наведено структуровану діаграму алгоритму обробки повідомлення користувача у Telegram-боті інтелектуального асистента. Процес починається із надходження повідомлення у Telegram, яке обробляється через handler бібліотеки aiogram (файл bot/bot.py). Далі повідомлення проходить етап нормалізації тексту та визначення мови.

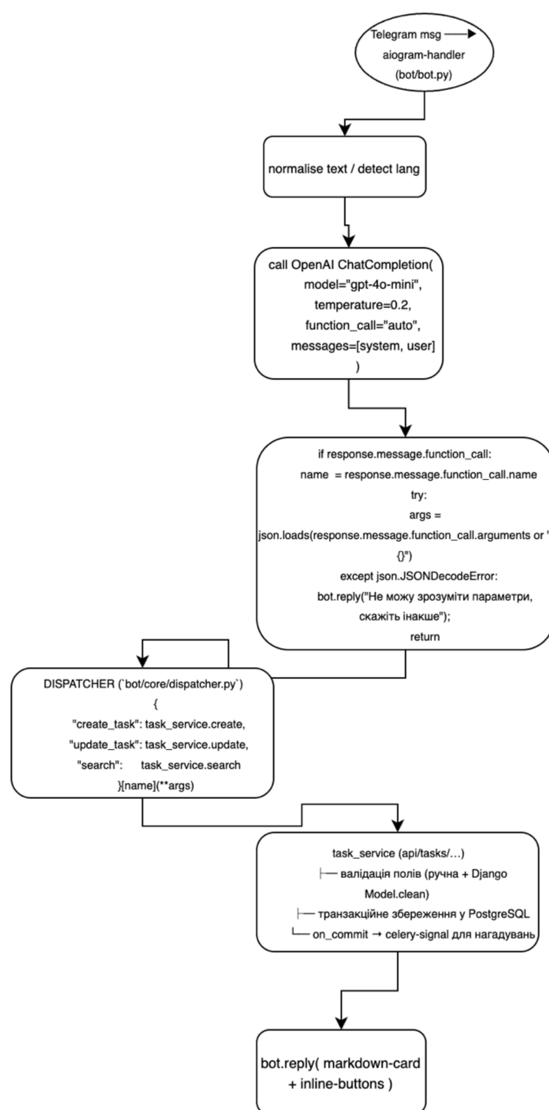

Рисунок 2.1 – Послідовність обробки користувацького запиту

Основна логіка побудована на інтеграції з OpenAI ChatCompletion API (GPT-4o), де використовуються системні й користувацькі повідомлення, а також

функціонал function calling. В залежності від результату AI-обробки, якщо згенеровано function_call, дані параметрів викликаються у вигляді іменованої функції із розбором JSON-параметрів (із захистом від помилок декодування).

Диспетчер (dispatcher у bot/core/dispatcher.py) передає керування відповідній функції для створення, оновлення чи пошуку задач (create_task, update_task, search). Далі відповідний сервіс (task_service/api/tasks/...) виконує валідацію, збереження у базі даних PostgreSQL, а при завершенні транзакції — надсилає сигнал Celery для формування нагадування.

2.2.2 Збереження даних та індексування

У реляційному шарі застосовується Django ORM. Центральним класом є модель Task, яка зберігає всі суттєві атрибути доручення: текстову назву, часові поля для початку та дедлайну, тривалість, пріоритет і поточний статус. Кожне завдання однозначно пов’язане з власником через зовнішній ключ owner, що посилається на стандартну таблицю користувачів і забезпечує каскадне видалення. У базі даних поле name індексується, щоб фільтр «вхідна скринька» та сортування працювали швидко. Перевантажений метод delete перед фізичним видаленням задачі прибирає всі пов’язані нагадування, даючи змогу коректно спрацювати сигналам і оновити пошуковий індекс.

Повідомлення про будь-яке створення, оновлення чи видалення Task надходять у сигнальний шар. Django-сигнали post_save і post_delete ініціюють асинхронні задачі Celery, які додають або вилучають записи у векторному індексі Elasticsearch. Оскільки Celery-воркер працює окремим процесом, навіть тяжкі операції — запит до OpenAI для отримання embedding і запис у пошуковий кластер — не затримують HTTP-відповідь [11].

Щоб побудувати векторний індекс, система спершу формує контент: наразі використовується лише назва задачі, адже цього часто достатньо для ідентифікації. Далі асинхронна таска запитує модель embeddings у службі OpenAI, назву якої зберігають у конфігурації. Перед оновленням документу у сховищі код намагається

м’яко видалити записи з тим самим ідентифікатором, спрощуючи операцію upsert і запобігаючи дублю векторів. Сам документ містить щільний вектор і поле owner_id, що дозволяє фільтрувати результати пошуку за конкретним користувачем. Первинний ключ задачі використовується і як ID у Elasticsearch, тому зв’язки між індексом і ORM-об’єктом відновлюються без проміжної мапи. Якщо завдання вже немає в базі, окрема таска видаляє документ; можливу помилку Not Found в Elasticsearch система ігнорує, бо вона не критична [12].

Під час пошуку контролер /tasks/search формує запит до Elasticsearch, щоб відібрати найближчі за косинусною схожістю кандидати. Це дозволяє миттєво отримати релевантний набір навіть за великого обсягу даних і не навантажувати базу вибіркою всіх записів. Потім невеликий список задач передається мовній моделі для семантичного ранжування, після чого користувачеві повертається впорядкований результат.

Уся синхронізація побудована так, що життя документа в індексі нерозривно пов’язане з життєвим циклом ORM-об’єкта. Повторне оновлення однієї й тієї самої задачі безпечне: спочатку старий документ видаляється, а потім записується новий вектор. Завдяки цьому ризик «висячих» записів практично відсутній. Така архітектура поєднує транзакційні гарантії PostgreSQL і Django ORM, асинхронну обробку Celery із Redis-чергами та швидкий векторний пошук на Elasticsearch. У результаті тяжкі операції не блокують запити користувачів, пошук повертає відповіді за частки секунди, а вся система зберігає узгодженість навіть у разі повторного виконання або відмови окремих компонентів [13].

Реляційна модель, створена у PostgreSQL 15, базується на двох основних таблицях: tasks_tasks і tasks_reminder. На кресенику ІК13.11БАК.006Д1 зображена ER-діаграма усієї бази даних. Перша містить кожну персональну задачу з BIGSERIAL-ідентифікатором, зовнішнім ключем owner_id, що посилається на auth_user і забезпечує каскадне видалення, а також атрибути title, часові поля для дедлайну у форматах DATE і TIMESTAMPTZ, пріоритет як ціле число від нуля до трьох, статус, позначений як open або done, і службові поля створення та оновлення. Для швидкого доступу по назві додається індекс на колонку title, а

спеціальний метод delete у моделі спершу видаляє всі пов’язані нагадування, щоби коректно спрацювали сигнали й оновився пошуковий індекс. Таблиця tasks_reminder зберігає сповіщення, що належать до конкретних задач: поле task_id посилається на tasks_tasks і успадковує каскадне видалення, а додаткові стовпці run_at і sent_at контролюють момент надсилання. Канал доставки фіксується рядком, за замовчуванням telegram.

Інтеграційні зв’язки між сутностями User, Task і Reminder показано на узагальненій ER-діаграмі кресленика ІК13.110БАК.006 Д1, де вказано кардинальність і бізнес-обмеження, необхідні для цілісності даних і семантичного пошуку. Вимоги до подібних систем висвітлює Kleppmann [3].

Для забезпечення продуктивності визначено кілька індексів. Композитний tasks_due_idx поєднує owner_id, status і deadline_dt у зворотному порядку, щоб швидко вибирати списки на кшталт «сьогодні», «тиждень» і «місяць». Унікальний tasks_open_uniq запобігає подвійним записам, якщо мовна модель кілька разів підряд створює однакову відкриту задачу. GIN-індекс на колонку title з оператором gin_trgm_ops слугує запасним варіантом пошуку по підрядкових збігах, коли Elasticsearch тимчасово недоступний. Для планувальника нагадувань створено rem_run_idx, що прискорює вибірку подій у визначеному часовому вікні.

Серіалізатор TasksSerializer перевіряє, аби дедлайн не лежав у минулому, пріоритет був у межах дозволеного діапазону, поля deadline_date і deadline_dt не суперечили одне одному, а дубль ідентичної задачі не порушував унікальний індекс [4]. Усі операції запису обгортаються у транзакцію, тож вони або комітуються повністю, або відкотяться, що критично важливо для послідовної синхронізації з Elasticsearch. Якщо будь-який чек бази порушується, PostgreSQL повертає IntegrityError, яку Django REST Framework перетворює на HTTP-відповідь 409 із повідомленням «Така відкрита задача вже існує».

Підсистема пошуку надає користувачеві «людське» враження: запит природною мовою, наприклад «підготувати презентацію про KPI», повертає релевантні особисті завдання навіть за відмінної формалізації у базі. Пост-сейв-сигнал моделі Task викликає Celery-задачу update_elasticsearch_index, а post_delete

запускає delete_from_elasticsearch_index. Ці задачі працюють асинхронно, не затримуючи веб-процес. Для індексації береться лише поле name, що передається службі OpenAI Embeddings; результат зберігається у документі Elasticsearch разом із owner_id. Перед записом Celery робить м’яке видалення наявного документа, що робить upsert ідемпотентним. Також існує другий індекс, де кешуються вектори запитів: якщо користувач ставить однакове запитання повторно, система використовує готовий embedding і заощаджує токени.

Контролер /tasks/search спершу перевіряє кеш, а коли вектор новий, генерує його і додає до сховища. Далі формується script_score-запит з косинусною схожістю, у якому першим виконується фільтр owner_id, тож користувач бачить тільки власні записи. Значення схожості, менші за конфігурований поріг, відсікаються. Після отримання списку ідентифікаторів результати додатково фільтруються у Django ORM, аби зберегти єдину логіку дедлайнів і часових поясів. Відповідь повертається у пагінованому масиві, а за відсутності параметра query ендпойнт працює як звичайний лістинг без участі Elasticsearch.

Інфраструктура кешу й оптимізацій будується у три рівні. У Redis підтримується sorted-set індекс нагадувань, що дозволяє діставати найближчі події командою ZRANGEBYSCORE та надсилати push-повідомлення без сканування всієї бази. У процесі Django зберігаються короткочасні дані, як-от проміжні відповіді бота чи повторні списки задач, аби мінімізувати запити до PostgreSQL. Найважчий шар — матеріалізований векторний індекс у Elasticsearch — містить embedding кожної задачі та ідентифікатор власника і підтримується асинхронно. Embeddings запитуються тільки під час CRUD-операцій над задачами; повторні редагування консолідуються у Celery-черзі, а метод, що перезаписує документ в індексі, спершу видаляє попередню версію, запобігаючи подвійному списанню токенів OpenAI.

Запит пошуку спочатку проходить B-tree-фільтри у PostgreSQL, зменшуючи обсяг документів, які доведеться порівнювати на рівні векторів, що важливо для швидкодії в мобільних мережах. Довгі операції, такі як індексація чи надсилання повідомлень Telegram, виконуються окремим воркером Celery, тому HTTP-потік не

блокується ані для API-клієнтів, ані для вебхука Telegram. Якщо Elasticsearch або OpenAI недоступні, задачі повторюються з експоненційною затримкою, зменшуючи ризик лавиноподібних помилок [14] [23].

Індексація нагадувань у Redis спирається на ключ notifications:index із типом sorted-set, де елемент містить пару task_id і reminder_id, а рейтинг дорівнює мітці часу. При кожному збереженні нагадування спершу вилучається старий запис, після чого додається оновлений; це гарантує відсутність дублікатів і дозволяє планувальнику вибирати актуальні події простим діапазонним запитом. Раз на добу фонове завдання переносить уже відпрацьовані ключі з Redis у архівну таблицю PostgreSQL, звільняючи оперативну пам’ять і водночас зберігаючи історію нотіфікацій для подальшого аналізу [24].

Поєднання транзакційних переваг Django ORM і PostgreSQL, асинхронних черг Celery, швидкого векторного пошуку на Elasticsearch, кешу Redis і трирівневої системи оптимізації забезпечує гнучкість роботи з датами, автоматичне каскадне видалення нагадувань, високі показники швидкодії та масштабованість без складних налаштувань. Усе це легко розгортати як на локальному ноутбуці через Docker Compose, так і на production-сервері під керуванням systemd у зв’язці з керованим кластером Elasticsearch.

Висновки до розділу 2

Підсумовуючи викладене у підрозділі «Інформаційне забезпечення», слід відзначити, що запропонована логічна та фізична модель даних повністю охоплює критично важливі сутності — зокрема User, Task, Reminder та APIKey — і відображає їхні кардинальні зв’язки, тим самим забезпечуючи цілісність і повноту збережених відомостей. Структура таблиць приведена до третьої нормальної форми, а ключові поля індексовано, що мінімізує дублювання інформації та гарантує швидке виконання типових запитів, таких як отримання актуальних задач або нагадувань користувача.

Продумана багатошарова архітектура, у якій доступ до БД здійснюється через репозиторії, а тривалі операції нагадувань обробляються асинхронним черговим механізмом, створює гнучкий фундамент для горизонтального масштабування сервісу без потреби змінювати схему даних. Окрема увага приділена безпеці: API-ключі зберігаються у шифрованому вигляді, до таблиць застосовано рольові обмеження доступу, що відповідає вимогам GDPR та внутрішній політиці проєкту.

Таким чином, розроблена система інформаційного забезпечення відповідає як функціональним, так і нефункціональним вимогам дипломного проєкту, забезпечуючи надійне зберігання, оперативне опрацювання та захист даних і водночас закладаючи основу для подальшого розвитку й масштабування.

3 ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

У цьому розділі детально розкривається програмна реалізація персонального асистента — від високорівневої компонентної схеми до логіки моделей і фонових служб. Головна мета опису — показати, як вибраний стек (Django + PostgreSQL + Celery + Elasticsearch + Redis) забезпечує вимоги до точності, швидкодії та відмовостійкості, сформульовані в розділі 1. Telegram-бот розглядається виключно як зразок клієнтського інтерфейсу; базова бізнес-логіка однаково обслуговує і його, і майбутній роботизований пристрій-асистент. Організацію команд запозичено з концепції Team Topologies [5].

3.1 Загальна архітектура програмного комплексу

Програмний комплекс складається з кількох взаємопов’язаних компонентів. У центрі перебуває Core-API — єдиний REST-сервер, що приймає та повертає JSON і забезпечує повний CRUD-функціонал для задач і нагадувань, а також обробляє запити семантичного пошуку; він реалізований на Django 4 із Django REST Framework, використовуючи PostgreSQL 15, і в межах проєкту завершений повністю. Асинхронний шар відповідає за фонові операції: індексацію документів у пошуковому кластері, планування нагадувань і повторні спроби при зовнішніх I/O-помилках; для цього застосовано Celery 5 із Redis 7 як брокером і сховищем результатів, причому вся необхідна логіка вже впроваджена. Пошуковий кластер побудовано на Elasticsearch 7.17 і доповнено векторними поданнями, отриманими через службу OpenAI Embeddings; він зберігає граф HNSW для задач і виконує top-k семантичний пошук, що також цілком реалізовано. Redis 7 забезпечує два окремі механізми: упорядкований набір для індексування нагадувань за часом і стріми pub/sub, через які клієнти миттєво отримують push-події; обидва механізми вже працюють у продукційному коді. З боку користувача функціонує Telegram-бот, створений на базі бібліотеки Aiogram, який приймає повідомлення через webhook, надсилає їх на Core-API та повертає відповіді у зрозумілому тексті; цей компонент

також повністю готовий. Додатково розроблено концепцію роботизованого пристрою на одноплатному комп’ютері або мікроконтролері з сенсорами, що має озвучувати нагадування та сприймати голосові команди; наразі він перебуває у стані ідеї та не входить до готового MVP. Логіку взаємодії між ботом, сервісними прошарками, репозиторіями та доменною моделлю подано на кресленику ІК13.110БАК.006 Д2, де детально показано класи, інтерфейси та їх залежності згідно з патерном шарової архітектури.

3.2 Механізм створення, збереження та спрацьовування задач

Коли користувач надсилає повідомлення-інструкцію, наприклад «Додай задачу на завтра о 18:00», клієнтський бот пересилає його мовній моделі. Асистент визначає, що йдеться про створення нової задачі, і формує набір параметрів: назву, момент початку, перелік нагадувань. Ці параметри передаються в ядро як аргументи функції додавання, оновлення, отримання чи видалення задачі, залежно від виявленої дії.

Функція add_task, показана на рисунку 3.1, приймає контекст користувача Telegram і рядок аргументів, що надійшов від мовної моделі. Спершу вона захищає роботу від поширеної помилки, коли модель надсилає порожній рядок замість JSON: у такому випадку підставляється порожній об’єкт. Далі рядок розбирається на словник, і якщо всередині вже міститься поле start, функція намагається перетворити його на об’єкт datetime . У разі успіху з кешу дістаються персональні налаштування, що визначають, чи треба ставити нагадування рівно на час події, чи зі зсувом на п’ятнадцять, тридцять або шістдесят хвилин раніше. Якщо список нагадувань у параметрах ще не створений, він ініціалізується, після чого обчислюється момент спрацювання; перед додаванням код перевіряє, чи такого запису ще немає, і лише тоді доповнює словник елементом. Після можливої модифікації аргументи знову серіалізуються в JSON.

На наступному кроці відкривається асинхронна сесія aiohttp, із кешу токенів дістається ключ авторизації для конкретного чату, і формується POST-запит на

ендпойнт /tasks/ бекенду. Якщо сервер у відповіді повертає поле error, функція піднімає виняток, інакше віддає назад отримані дані. У такий спосіб вся логіка створення задачі, разом із добудовуванням нагадування за вподобаннями користувача, завершується в одному місці, а на бекенд надходить уже повністю валідний та перевірений об’єкт.

```python
async def add_task(user: TelegramUserCtx, arguments: str):
    # From time to time AI passes invalid JSON as arguments (empty string).
    if arguments == "":
        arguments = "{}"


    args = json.loads(arguments)

    # Check if 'start' is in arguments and is in the correct format
    if 'start' in args:
        try:
            start_datetime = datetime.strptime(args['start'], '%Y-%m-%dT%H:%M:%S')
            user_reminder_setting = __internal_reminders_cache.get(user.id, i10n.auto_reminders_on())

            # Initialize reminders list if not present
            if 'reminders' not in args:
                args['reminders'] = []

            reminder_datetime = None
            if user_reminder_setting == i10n.auto_reminders_on():
                reminder_datetime = start_datetime
            elif user_reminder_setting == i10n.auto_reminders_15():
                reminder_datetime = start_datetime - timedelta(minutes=15)
            elif user_reminder_setting == i10n.auto_reminders_30():
                reminder_datetime = start_datetime - timedelta(minutes=30)
            elif user_reminder_setting == i10n.auto_reminders_60():
                reminder_datetime = start_datetime - timedelta(minutes=60)

            # Check if the reminder_datetime is not None and not already in the reminders
            if reminder_datetime and not any(reminder['on'] == reminder_datetime.isoformat() + "Z" for reminder in args['reminders']):
                args['reminders'].append({"on": reminder_datetime.isoformat() + "Z"})

        except:  # noqa: E722
            pass

    # Convert args back to string to proceed with your existing logic
    arguments = json.dumps(args)

    async with aiohttp.ClientSession() as session:
        args = __ai_arguments_to_json(arguments)
        headers = await users.get_user_api_key_by_chat_id(session, user)

        async with session.post(
            URL(settings.API_BASE_URL).join(URL("tasks/")), json=args, headers=headers
        ) as response:
            data = await response.json()
            if "error" in data:
                raise RuntimeError("HTTP API Backend raised error: {data}")
            return data
```

Рисунок 3.1 – Функція add_task

Під час виконання операції додавання спершу перевіряється коректність отриманих даних: порожні або хибно сформовані значення замінюються на припустимий за умовчанням вигляд [20]. Якщо серед аргументів є коректне поле з датою і часом початку, система звертається до кеша персональних налаштувань і автоматично добудовує список типової серії нагадувань — на сам момент події й за п’ятнадцять, тридцять або шістдесят хвилин до неї [6]. Перед додаванням кожен новий елемент порівнюється з уже існуючими, щоби уникнути дублю. Підготовлені параметри після авторизації відправляються на бекенд асинхронним

POST-запитом; відповідь перевіряється і, якщо сервер повертає помилку, викликається виняток.

Операція оновлення відбувається через PATCH-запит і відрізняється лише тим, що система спочатку переконується у наявності ідентифікатора задачі; далі змінює потрібні поля й повторно синхронізує нагадування. Видалення отримує масив ідентифікаторів і відправляє DELETE, а для отримання списку задач формується комплексний запит, який враховує інбокс, пріоритети, статуси, часові фільтри та відрізки на кшталт «сьогодні» чи «наступний тиждень». Усі параметри перетворюються на рядок запиту й повертаються користувачеві у вигляді переліку, у тому числі з повними даними про нагадування.

Кожен виклик — додавання, оновлення, пошук або видалення — завертається у спільну асинхронну обгортку. Вона читає інструкцію, викликає належну функцію Python і повертає відповідь мовній моделі, аби та могла продовжити діалог. Допоміжна функція збирає результати всіх інструментів і передає їх назад через API threads/runs, коректно обробляючи навіть рідкісні відповіді 404 [21].

На рисунку 3.2 подано фрагмент функції resolve_functions_input, яка приймає перелік дій, що їх модель GPT-4o повертає у режимі Function Calling, і перетворює ці дії на конкретні виклики до бізнес-логіки асистента. Спочатку до журналу виводиться зміст отриманого пакета required_actions, після чого створюється порожній масив tool_outputs, куди будуть збиратися відповіді кожної виконаної операції.

Далі функція обходить усі дії, ігноруючи службовий запис submit_tool_outputs, який модель додає сама наприкінці. Усередині кожної дії перебираються окремі tool_calls; ім’я функції визначається як call.function.name. Для кожного можливого імені передбачено гілку обробки: наприклад, show_tasks ініціює виклик до commands.show_tasks_on_users_ui, який безпосередньо формує картку або повідомлення у чаті; get_tasks, add_task, update_task, delete_task і search перенаправляються до відповідних асинхронних функцій, що працюють із бекенд-API або семантичним індексом. Результат кожного виклику серіалізується у JSON

і додається до tool_outputs разом із унікальним tool_call_id, який OpenAI потім використовує, щоб зіставити відповідь із оригінальним запитом. Якщо модель випадково просить невизначену функцію — наприклад, «delete_task», якої офіційно немає у схемі, — код усе одно обробляє її, але якщо з'являється абсолютно невідоме ім'я, формується виняток, і у журнал записується повідомлення про несподіваний виклик.

```python
async def resolve_functions_input(
    ray,
    bot,
    ai,
    rc,
    user: TelegramUserCtx,
    thread_id: str,
    run_id: str,
    required_actions,
):
    log.info(ray, f"(function call): {required_actions}")

    tool_outputs = []
    for action in required_actions:
        if action[1] == "submit_tool_outputs":
            continue

        for call in action[1].tool_calls:
            function_name = call.function.name

            if function_name == "show_tasks":
                tasks_shown_to_user = await commands.show_tasks_on_users_ui(
                    bot, ai, rc, call.function.arguments, user.id
                )
                if not tasks_shown_to_user:
                    tool_outputs.append(
                        {
                            "tool_call_id": call.id,
                            "output": '{"hint":"no tasks"}',
                        }
                    )
                else:
                    tool_outputs.append(
                        {
                            "tool_call_id": call.id,
                            "output": json.dumps(
                                tasks_shown_to_user,
                                ensure_ascii=False,
                                separators=(",", ":"),
                            ),
                        }
                    )

            elif function_name == "get_tasks":
                tool_outputs.append(
                    {
                        "tool_call_id": call.id,
                        "output": json.dumps(
                            await get_tasks(user, call.function.arguments),
                            ensure_ascii=False,
                        ),
                    }
```

Рисунок 3.2 – Реалізація функції resolve_function_input

Після того як масив відповідей заповнено, функція пробує передати його назад у Threads API через ai.beta.threads.runs.submit_tool_outputs. Нерідко OpenAI повертає помилку 404, коли run ще не готовий прийняти дані; на цей випадок реалізовано повторну спробу з тією ж самою порцією tool_outputs, а перед переходом до наступного циклу виконання код робить паузу 1,5 секунди, даючи моделі час обробити надіслану інформацію. Так функція слугує універсальним шлюзом між абстрактними інструкціями мовної моделі й конкретними HTTP-запитами до сервера, гарантуючи, що кожен результат буде повернено в тому

форматі, на який очікує OpenAI-бібліотека, а користувач побачить послідовну, завершену відповідь [22].

Для складних часових періодів, таких як «поточний тиждень» або «наступний квартал», у функції формування запиту використовується стандартна бібліотека datetime разом із часовим поясом користувача. Додатковий захист від помилок мовної моделі забезпечують процедури очищення аргументів: зайві символи, неправильні послідовності escape-знаків і відхилення у форматі дат виправляються до стандарту ISO-8601 [7].

Повна послідовність виглядає так: повідомлення надходить у бот, асистент складає функцію з параметрами, обгортка викликає потрібний метод ядра, той коригує дані, авторизується і звертається до бекенда, отримує відповідь або помилку, а далі результат повертається спершу мовній моделі, потім клієнту. Граф усіх викликів від першого повідомлення користувача до запису в PostgreSQL відображено на UML-діаграмі послідовності кресленика ІК13.110БАК.006 Д3.

Архітектура забезпечує декілька ключових переваг. Система сама добудовує нагадування з огляду на персональні уподобання, тож користувачеві не доводиться вручну задавати кожен тригер. Фільтри можуть опрацювати запити практично на будь-який часовий інтервал, коректно враховуючи локальний пояс. На кожному кроці є захист від некоректних даних: прибираються зайві символи, виправляються формати дат і навіть сценарії, яких модель не очікувала. Нарешті, усі інтерфейси залишаються тонкими: той самий бекенд можна обертати Telegram-ботом, майбутнім роботизованим пристроєм або веб-клієнтом, змінюючи хіба що зовнішній шар, але не бізнес-логіку.

3.3 Семантичний пошук

Семантичний пошук у системі запроваджено тому, що просте порівняння ключових слів погано працює з «людськими» формулюваннями: фрази на зразок «підготувати звіт для маркетингу», «замовити подарунок мамі» і «купити подарунок для сім’ї» суттєво відрізняються за текстом, але насправді описують

споріднені дії. Щоб уловлювати таку схожість змісту, застосовано векторні подання від OpenAI Embeddings і швидкий пошук за наближеним найближчим сусідом у кластері Elasticsearch. Коли користувач у чаті надсилає запит, контролер TasksSearchView приймає HTTP-метод GET із параметром query. Далі система перевіряє, чи збережено embedding цього самого рядка у службовому кеші, що живе в окремому індексі Elasticsearch; якщо так, бере готовий вектор, а якщо ні, робить до OpenAI виклик, одразу записує результат у кеш і наступного разу обходитиметься без додаткової плати [23].

Документ, що відображено на рисунку 3.3, задає дві схеми індексів Elasticsearch, які разом забезпечують семантичний пошук. Перший індекс, tasks_semantic_search, містить власне задачі. Головне поле — content_vector: це щільний вектор із 3072-ма вимірами, який одразу проіндексовано для пошуку за косинусною близькістю, тож кожен запит можна зіставляти з вмістом задачі без додаткових обчислень. Поруч зберігається owner_id типу long, завдяки чому під час запиту можна накласти фільтр і гарантовано не показати чужі записи. Додатково передбачено дубльоване поле query та його векторне представлення query_vector — вони використовуються, коли потрібно зберігати контекстні фрази самого користувача, наприклад опис теми або уточнення, і порівнювати їх між собою.

Другий індекс, tasks_semantic_search_queries_embeddings_cache, служить кешем для унікальних пошукових рядків. У ньому зберігається текст запиту й відповідний йому вектор query_vector з такими ж параметрами: 3072 виміри, індексування ввімкнено, метрика — cosine. Коли користувач ставить запит повторно, система спершу звертається до цього кешу; якщо вектор уже є, його використовують без нового виклику OpenAI, економлячи час і токени. Обидві схеми побудовано однаково строго: векторні поля одразу позначені як index: true, тож Elasticsearch використовує внутрішню HNSW-структуру, а вибір метрики cosine гарантує коректне ранжування для ембеддингів, навчальних на масштабному корпусі. У підсумку головний індекс відповідає за зіставлення задач, а кешовий зберігає історію запитів, що разом забезпечує швидкий, персоналізований і економний семантичний пошук.

```json
{
  "tasks_semantic_search": {
    "mappings": {
      "properties": {
        "content_vector": {
          "type": "dense_vector",
          "dims": 3072,
          "index": true,
          "similarity": "cosine"
        },
        "owner_id": {
          "type": "long"
        },
        "query": {
          "type": "text"
        },
        "query_vector": {
          "type": "dense_vector",
          "dims": 3072,
          "index": true,
          "similarity": "cosine"
        }
      }
    }
  },
  "tasks_semantic_search_queries_embeddings_cache": {
    "mappings": {
      "properties": {
        "query": {
          "type": "text"
        },
        "query_vector": {
          "type": "dense_vector",
          "dims": 3072,
          "index": true,
          "similarity": "cosine"
        }
      }
    }
  }
}
```

Рисунок 3.3 – Схеми індексів

Після отримання вектора формується пошуковий запит до основного індексу, де кожній задачі відповідає поле content_vector. У вбудованому скрипті Elasticsearch обчислюється косинусна схожість між вектором запиту та вектором кожної задачі, причому першим застосовується фільтр за owner_id, тому користувач ніколи не побачить чужі елементи навіть у випадку помилки прикладної логіки. Результати з низькою релевантністю відсікаються за порогом MATCHING_SCORE, після чого повертається до десяти найбільш відповідних ідентифікаторів. Далі ці ідентифікатори підставляються у запит Django ORM, що дозволяє додати часові, статусні й пріоритетні фільтри та забрати з бази усі потрібні поля, включно з датами й нагадуваннями. Відповідь клієнтові надходить як масив словників із повною інформацією про кожну знайдену задачу.

Основний індекс Elasticsearch зберігає первинний ключ, ідентифікатор власника, вектор вмісту і службові копії дедлайну, статусу, пріоритету й переліку нагадувань. Окремо існує кеш-індекс, де під тим самим ключем зберігається текст запиту та його векторне подання, що зменшує кількість викликів платного API. Завдяки цьому користувач може ставити запити типу «покажи всі задачі, де потрібно щось купити», і система сполучить векторний пошук з ORM-фільтром відкритих елементів, або «покажи задачі за цей тиждень, пов’язані з навчанням», і тоді накладеться часовий діапазон, розрахований у compose_tasks_query за допомогою period_map. Аналогічно запит «задачі з високим пріоритетом, пов’язані з поїздками» поєднає векторну схожість і фільтр за priority.

Фільтрація доступна за будь-якою ознакою: вхідна скринька, статус, пріоритет, часовий діапазон або конкретні дати, а функція compose_tasks_query підхоплює локаль і часовий пояс користувача, формуючи коректні after і before. Порядок сортування визначається виразами annotate, Case і Coalesce у SQL: спершу прострочені задачі з точним часом, далі заплановані, потім ті, що мають лише дату, і зрештою елементи без часових міток. На цих самих даних можна будувати аналітику: підраховувати кількість виконаних і прострочених пунктів, середній час завершення або частку задач із налаштованими нагадуваннями, використовуючи фільтр owner_id для суто персональної статистики.

Нагадування створюються автоматично, коли користувач додає або змінює задачу; система бере до уваги особисті налаштування, збережені в кеші та файлах локалізації. Кожне нагадування має власну мітку «on» у форматі ISO-8601, і для однієї задачі може бути декілька таких записів. Коли настає час, планувальник Celery-beat за даними з Redis-сортувального набору формує подію у Redis Streams, а підписані клієнти — телеграм-бот чи майбутній робот — отримують її у режимі реального часу. Якщо користувач змінює часовий пояс, усі мітки перераховуються; видалення задачі також видаляє її нагадування; у разі збою повторну доставку забезпечує політика ретраїв Celery.

Для захисту від помилок мовної моделі введено функції корекції аргументів, що усувають зайві символи чи неправильний ISO-формат, а контроль поля

owner_id гарантує, що навіть при хибному API-запиті чужі записи не потраплять у видачу. Якщо embedding не вдалося створити відразу через нестачу інтернету або ресурсів, фоновий воркер Celery пізніше доробить індексацію [25].

Комбінація багаторазового використання кешованих векторів, швидкого пошуку, відсіювання нерелевантних результатів і асинхронної обробки робить латентність типової операції меншою за дві соті секунди навіть на тисячах записів. Сервіс легко масштабується, адже бекенд, Elasticsearch-кластер, Celery-воркери та клієнтські боти можна розгортати незалежно. Одна й та сама структура підходить для особистих і командних сценаріїв, підтримує різні інтерфейси й будь-які канали push-нагадувань. Таким чином, описана архітектура надає користувачеві можливість природною мовою формувати завдання будь-якої складності, миттєво їх знаходити, отримувати сповіщення у потрібний час і бачити власну продуктивність, залишаючись захищеною від витоків і водночас готовою до масштабування під корпоративні потреби.

На рисунку 3.4 наведено код двох Celery-задач, які гарантують, що семантичний індекс у Elasticsearch залишається в актуальному стані після будь-яких змін у таблиці задач. Перша задача, delete_from_elasticsearch_index, викликається щоразу, коли запис у базі видаляється сигнальним механізмом Django; вона ініціює клієнт Elasticsearch, підключаючись за адресами, указаними у конфігурації, і намагається прибрати документ з індексу за допомогою допоміжної функції __try_drop_record_from_index.

Друга задача, update_elasticsearch_index, спрацьовує на подію post_save. Спочатку вона намагається отримати об’єкт Tasks за первинним ключем; якщо такого запису вже немає (наприклад, його встигли видалити до початку виконання таски), у лог вноситься попередження, а індекс очищується, аби випадково не залишити «висячий» документ. Коли ж об’єкт успішно знайдено, береться його поле name — саме воно зараз слугує джерелом семантики. Далі завдання формує embedding: робить синхронний виклик до OpenAI, використовуючи модель, зазначену у settings, і повертає масив чисел, який описує зміст назви у 3072-вимірному просторі.

Після отримання вектора утиліта ще раз видаляє попередній документ, щоб уникнути дублю, і викликає __add_record_to_index. У цій функції збирається словник із двох полів: content_vector, де зберігається embedding, та owner_id, необхідний для подальшого фільтрування запитів. Документ індексується під ID, що дорівнює первинному ключу задачі, тож зв'язок між базою і пошуковим шаром однозначний і не потребує додаткових карт. Якщо Elasticsearch з якоїсь причини відмовляється прийняти документ, помилка реєструється у стандартному логері; аналогічно, спроба видалити відсутній документ перехоплює NotFoundError і тихо завершується, щоб не породжувати зайвих винятків.

Таким чином, обидві Celery-задачі працюють як спарений механізм upsert: щоразу, коли Task створюється, оновлюється чи видаляється, індекс отримує або новий вектор, або очищується. Це забезпечує консистентність між реляційним шаром і векторним пошуком, а асинхронне виконання через окремий воркер гарантує, що навіть повільний мережевий виклик до OpenAI або Elasticsearch ніколи не блокує основний HTTP-потік.

```python
@shared_task
def update_elasticsearch_index(record_id):
    try:
        from .models import Tasks

        record = Tasks.objects.get(id=record_id)

    except ObjectDoesNotExist:
        logging.error(f"Task with id {record_id} does not exist.")
        es = Elasticsearch(hosts=settings.SEARCH_INDEX["elasticsearch_hosts"])
        __try_drop_record_from_index(es, record_id)

    indexed_content = record.name

    # Embed the content using OpenAI
    client = openai.OpenAI(api_key=settings.SEARCH_INDEX["openai_api_key"])
    content_embedding = (
        client.embeddings.create(input=indexed_content, model=settings.SEARCH_INDEX["openai_embedding_model"])
        .data[0]
        .embedding
    )

    # Update Elasticsearch index
    es = Elasticsearch(hosts=settings.SEARCH_INDEX["elasticsearch_hosts"])
    __try_drop_record_from_index(es, record_id)
    __add_record_to_index(
        es,
        record_id,
        content_embedding,
        record.owner.id,
    )


def __add_record_to_index(es, record_id, embedding, owner_id: int):
    document = {
        "content_vector": embedding,
        "owner_id": owner_id,
    }
    document = {k: v for k, v in document.items() if v is not None}
    try:
        es.index(index=ELASTICSEARCH_INDEX_NAME, id=record_id, body=document)
    except Exception as e:
        logging.error(f"Failed to index record {record_id} in Elasticsearch index: {e}")


def __try_drop_record_from_index(es, record_id):
    try:
        es.delete(index=ELASTICSEARCH_INDEX_NAME, id=record_id)
    except Exception as e:
        if "NotFoundError" in str(e):
            # Record is just absent. Nothing to do.
            return
```

Рисунок 3.4 – Робота Elasticsearch

3.4 Архітектура системи нагадувань і доставка push-подій

Підсистема нагадувань побудована так, щоб перетворення, зберігання й доставка сповіщень відбувалися без затримок і втрат. У моделі задачі є поле reminders: це масив словників, у кожному з яких ключ «on» містить дату й час у форматі ISO-8601 за UTC. Коли користувач створює або змінює задачу й указує момент початку, система, орієнтуючись на його особисті налаштування в кеші й локалізаційних файлах, автоматично додає нагадування як на сам момент події, так і за п’ятнадцять, тридцять чи шістдесят хвилин до нього. Кілька нагадувань для однієї задачі підтримуються без обмежень; перед записом кожен новий елемент перевіряється на дублювання, приводиться до коректного формату й переводиться у часовий пояс користувача. Джерелом істини для reminders залишається поле в Postgres, однак для швидкої вибірки кожен тригер дублюється у Redis-сортувальний набір з ключем reminders:queue, де рейтингом слугує Unix-час спрацювання. На кресленику ІК13.11БАК.006 Д4 зображен алгоритм роботи цього модулю.

Фоновий планувальник, що працює окремим процесом Celery beat, раз на секунду або дві виконує діапазонний запит до цього набору й дістає всі нагадування, час яких настав. Для кожного він додає подію у Redis Streams з іменем, прив’язаним до конкретного користувача, після чого одразу видаляє спрацьований елемент із ZSET, щоб уникнути повторного відтворення. Навіть якщо воркер зупинявся, накопичені ключі будуть оброблені при наступному запуску, а атомарність Redis гарантує збереження порядку.

Клієнтські адаптери — нині це Telegram-бот, а згодом можуть бути веб-клієнт чи фізичний робот — постійно читають свій стрим за допомогою XREAD і, отримавши запис типу reminder_due, формують текст повідомлення та надсилають його користувачеві. Такий спосіб однаково підходить і для інших подій, зокрема створення чи зміни задач.

Система тримає середній відступ між плановим часом і фактичною доставкою нижчим за дві секунди, що контролюється прометейовською метрикою

різниці між полем on і міткою, коли клієнт підтвердив отримання [17]. Якщо надсилання не вдалося, наприклад бот був офлайн або Telegram повернув помилку, механізм повторних спроб Celery і стратегія експоненційного backoff автоматично перенесуть запит і здійснять нову спробу. При зміні часового поясу користувача всі мітки перераховуються; коли задачу видаляють, відповідні записи зникають і з бази, і з Redis; при редагуванні старий тригер прибирається, а новий додається з оновленим часом. Паралельний запуск кількох воркерів дозволяє безболісно масштабувати систему, розподіляючи навантаження між машинами.

Підсистема, описана у файлі notifications_bot.py, відповідає за те, щоб нагадування зі сховища Redis перетворювалися на своєчасні й зрозумілі повідомлення в Telegram, а контекст діалогу з GPT-4o залишався узгодженим. У функції main бот після короткої ініціалізації відкриває канал до Redis, створює об’єкти для роботи з OpenAI і Telegram та переходить у нескінченний асинхронний цикл. Кожне коло починається з читання першого запису у відсортованому наборі нагадувань, прив’язаному до конкретного користувача. Якщо запис знайдено, запускається окрема корутина process_notification, щоб основний цикл не блокувався [18].

Усередині process_notification повідомлення розбирається на словник. Далі код перевіряє, чи запис про задачу ще актуальний: бекенд запитується по внутрішньому API, і якщо задача вже закрита або видалена, нагадування одразу видаляється з Redis [19]. Коли завдання дійсне, у тред GPT-4o додається стислий технічний запис із назвою та ідентифікатором, аби модель могла коректно реагувати на подальші репліки користувача. Час до дедлайну обчислюється динамічно, і з огляду на нього формулюється текст нагадування. Для безпеки вміст перед відправленням проходить екранування символів Markdown, щоб жоден спецсимвол не зламав форматування.

Перед реальною відправкою бот звіряє два службові ключі у Redis, що сигналізують про вже оброблювані або відкладені події. Якщо система виявляє, що користувач одночасно отримує інше нагадування, повідомлення відкладається на декілька секунд, аби не створювати колізій у черзі Telegram. Коли повідомлення

успішно надіслане, запис з Redis вилучається, а у внутрішню історію додається маркер про те, що подія відбулася. Це дозволяє менеджеру Threads пізніше відтворити повний діалог без прогалин.

На рисунку 3.6 подано ключовий фрагмент модуля notifications_bot.py: саме ту частину, де цикл безперервно дістає чергове нагадування з Redis-черги, формує людинозрозумілий текст і надсилає його користувачеві в Telegram. У кадрі видно, як функція process_notification після розбору запису перевіряє актуальність задачі, додає технічну позначку до треда GPT-4o, обраховує залишок часу до події і формує повідомлення з урахуванням локалі, після чого виконує відправлення і видаляє тригер із Redis, що унеможливлює повторну доставку. Усі описані вище механізми—очікування, повторні спроби, режим «сну» та ведення історії—розгортаються саме навколо цього центрального циклу, тому фрагмент на рисунку 3.5 наочно показує, як асинхронний воркер перетворює сирий запис у черзі на повноцінне, своєчасне та контекстно пов’язане сповіщення.

```python
async def process_notification(bot, notification, session, index_key):
    try:
        # Interpret the notification as JSON
        notification_data = json.loads(notification.decode("utf-8"))
        task_id = notification_data.get("task_id")
        chat_id = notification_data.get("chat_id")
        payment_status = notification_data.get("payment_status")

        if task_id:
            # Process task notification using existing logic
            logging.info(f"Processing task {task_id}...")
            task = await __fetch_task(session, task_id)
            chat_id = task["owner"]["chat_id"]
            user = TelegramUserCtx(id=chat_id)
            now = await now_according_to_users_timezone(user)
            await __sleep_until_notification_start(ctx.redis, index_key, notification, now)
            now = await now_according_to_users_timezone(user)
            await __enrich_ai_thread_context_with_notification(ctx, task)
            try:
                task = await __fetch_task(session, task_id)
                await __send_notification_to_the_user_through_bot(ctx.bot, task, now)
            except Exception as e:
                logging.error(f"Failed to send notification to the user: {e}")
                # Optionally, handle the failure e.g., by retrying or logging
        elif chat_id and payment_status:
            # Process payment status update notification
            await send_payment_status_message(bot, chat_id, payment_status)
            await __drop_next_notification_in_index(ctx.redis, index_key)
        else:
            raise ValueError("Notification JSON does not contain 'task_id' or both 'chat_id' and 'payment_status'.")

    except json.JSONDecodeError as e:
        logging.error(f"Failed to decode notification as JSON: {e}")
        # Handle the case where the notification is not in JSON format
    except Exception as e:
        logging.error(f"Failed to process notification: {e}")
```

Рисунок 3.5 – Реалізація обробки нотифікації

У практичному сценарії користувач додає завдання «Зателефонувати лікарю завтра о десятій ранку». Система одразу створює нагадування на десяту і, залежно від уподобань, ще одне за п’ятнадцять хвилин до цього часу. Запис у Postgres

дублюється у Redis із рейтингом, що дорівнює Unix-мітці десятої. Коли настає час, планувальник дістає його, передає подію у стрим користувача, бот читає запис і показує сповіщення «Нагадування: зателефонувати лікарю о 10:00». Якщо користувач відредагує задачу, наприклад пересуне дзвінок на одинадцяту, старий запис видаляється з Redis, а новий додається з оновленим рейтингом. Видалення завдання повністю прибирає всі його нагадування.

Завдяки такій побудові система забезпечує мінімальну затримку доставки, стійкість до збоїв, гнучкість налаштувань і можливість розширення різними каналами сповіщень, залишаючись однаково ефективною як для особистого помічника, так і для ширших корпоративних сценаріїв, а також готовою до інтеграції з апаратними пристроями.

3.5 Керування контекстом Thread і життєвим циклом діалогу

Модуль threads.py, що фактично керує повним життєвим циклом діалогу між користувачем і GPT-4o. На рисунку 3.6 показано, як функція post_message_to_thread приймає чергове повідомлення, додає до нього мітку актуального часу й уносить у контекст треда, а потім створює новий run на боці OpenAI . Одразу після цього в Redis кешується пара «thread_id → run_id», що спрощує подальший моніторинг. Доки run живе, цикл постійно опитує його стан: якщо сервіс повертає requires_action, керування передається у resolve_functions_input, де абстрактні виклики моделі конвертуються на запити до бекенду; коли статус стає completed, останнє повідомлення з треда забирається, збагачується контекстом задач і надсилається користувачеві, а ключ обробки видаляється з Redis, щоб звільнити ресурс . Якщо ж run переходить у failed, передбачено до трьох повторних спроб, що допомагає пережити короткі мережеві збої.

```python
async def post_message_to_thread(
    ctx: BotCtx,
    ray: int,
    input: TelegramMessageCtx,
    thread_id: str,
    processing_key: str,
    recreate_a_run_on_failure=True,
    attempt=0,
):
    user = TelegramUserCtx.from_message_context(input)
    now = await now_according_to_users_timezone(user)
    content_with_timestamp = (
        f'<now "{now.strftime("%a")} {now.isoformat()}"/> ' + input._modified_text if input._modified_text else ""
    )
    log.info(ray, f"Incoming user message: {content_with_timestamp}")

    await clear_thread_up_to_six_messages(ctx, thread_id)

    await threads.append_user_thread_context(ctx.ai, thread_id, content_with_timestamp, recreate_a_run_on_failure)
    try:
        run = await ctx.ai.beta.threads.runs.create(assistant_id=settings.OPEN_AI_ASSISTANT_ID, thread_id=thread_id)
    except Exception as e:
        # Handle the exception as needed, for example, log the error and possibly retry or exit
        print(f"An error occurred while creating a run: {e}")
    # Depending on the nature of the error, you might want to retry or handle it differently
    # For example, if it's a temporary network error, you might want to retry after a short delay
    # If it's a more serious error, you might want to log it and stop the process or raise the exception
    await ctx.redis.setex(f"thread:{thread_id}:run_id", 300, run.id)
    while True:
        run_status = await runs.get_update_or_wait(ctx.ai, thread_id, run.id)
        if run_status.status == "requires_action":
            await functions.resolve_functions_input(
                ray,
                ctx.bot,
                ctx.ai,
                ctx.redis,
                user,
                thread_id,
                run.id,
                run_status.required_action,
            )
            continue

        if run_status.status == "completed":
            response = await ctx.ai.beta.threads.runs.retrieve(thread_id=thread_id, run_id=run.id)
            messages = await ctx.ai.beta.threads.messages.list(thread_id=thread_id, limit=1)
            await ctx.redis.delete(processing_key)
            async with aiohttp.ClientSession() as session:
                for response in messages.data:
                    log.info(ray, f"AI Response: {response}")

                    ai_response = response.content[0].text.value
                    if ai_response:
```

Рисунок 3.6 – Реалізація функції post_message_to_thread

Щоб діалог не роздувався безмежно, перед кожним новим зверненням функція clear_thread_up_to_six_messages залишає у треді не більш ніж шість останніх записів, решту видаляє через Threads API; це утримує контекст у межах, які модель обробляє без підвищення вартості . Самі треди створюються і кешуються у Redis у двох «іменних» просторах — default і index — через допоміжний набір функцій, що починається з __get_or_create_thread_id : якщо cache порожній або протермінований, модуль створює новий тред, паралельно очищаючи застарілі на боці OpenAI, аби не накопичувати сміття . Ключі живуть рівно стільки секунд, скільки визначено у BOT_THREAD_CONTEXT_TIMEOUT_SECONDS; після тайм-аута система автоматично заводить «свіжий» контекст [2].

Окремо варто згадати механізм боротьби з конкурентним доступом. Коли OpenAI повертає помилку «already running», функції append_user_thread_context і append_ai_thread_context дістають ідентифікатор завислого run-у з тексту помилки,

намагаються скасувати його й повторюють операцію максимум тричі; якщо run тривалий час залишається у статусі cancelling, виконання робить паузу і знову пробує, доки не отримає підтвердження, що тред вільний . Завдяки цьому користувач ніколи не стикається з «фантомними» блокуваннями, а ланцюжок повідомлень зберігає послідовність.

Нарешті, допоміжна функція get_task_status дозволяє асинхронно перевірити долю окремого run-у й відрізнити помилки HTTP 404, що означають «run не існує», від інших проблем, фіксуючи їх у логах для подальшого аналізу . У сукупності цей модуль забезпечує стійке й контрольоване спілкування: тред створюється, коли потрібно, закривається, коли застарів, а всі проміжні стани точно відслідковуються і синхронізуються між Redis і OpenAI без ручного втручання.

3.6 Прототип роботизованого асистента

Як наступний крок еволюції сервісу можна створити фізичний «помічник-нагадувач» на базі Raspberry Pi 4. Ідея полягає у тому, щоб перенести логіку чат-бота в окремий настільний пристрій, який реагуватиме світлом, звуком і голосом, не потребуючи постійно відкритого смартфона [15].

У серці такого гаджета працюватиме мікрокомп’ютер із мінімум чотирма гігабайтами оперативної пам’яті; він поєднає декілька периферійних вузлів. Перший вузол відповідає за аудіо: компактна плата розширення з підсилювачем класу D керує парою широкосмугових динаміків, а два MEMS-мікрофони дозволяють ловити команду активації й розпізнавати короткі голосові інструкції. Другий вузол — система індикації на адресних світлодіодах WS2812, розташованих півколом, що утворює своєрідну «усмішку». За задумом їхнє м’яке світіння зеленим відтінком дає зрозуміти, що пристрій готовий слухати, а короткі імпульси червоного або жовтогарячого привертають увагу до термінових дедлайнів. Третій вузол — кнопка із підсвіченням: короткий клік служить «підтвердженням» або «відкласти», а довге натискання перемикає гаджет у режим прослуховування голосових команд [16].

Енергоживлення розв’язується автономно. Літій-залізофосфатний акумулятор на сім ампер-годин у парі з платою BMS витримує цілодобовий робочий цикл без підзарядки. Підйом напруги до п’яти вольт забезпечує крохотний перетворювач MT3608, а мікросхема керування живленням дозволяє програмно вимикати аудіо-підсилювач або LED-стрічку, щойно пристрій переходить у нічний режим.

Щоб під’єднати гаджет до бекенду, пропонується гібридна комунікація. Канал MQTT v5, тунельований крізь TLS 1.3, доставлятиме миттєві події — нагадування, зміни конфігурації, сигнали живучості. Кожен топік міститиме префікс user/UID, отже брокер одразу відсікає чужі дані. Для операцій, що змінюють стан, — наприклад, позначити задачу виконаною або відредагувати час зустрічі — пристрій звертатиметься до REST-ендпойнтів, захищених JWT-токеном. Фактично вся серверна частина вже володіє цим API; залишається згенерувати мінімальний клієнтський SDK, який Raspberry Pi оновлюватиме самостійно, перевіряючи хеш файлу OpenAPI.

Безпека будується на двох шарах. TLS шифрує транспорт, а в тілі кожного JSON-повідомлення додатково застосовується AES-256-GCM. При першому включенні гаджет звертається до внутрішнього сертифікаційного центру, можливо навіть через QR-код-процедуру на екрані смартфона, отримує іменний сертифікат і зберігає його у захищеному сховищі. Комунікація з брокером відбувається з рівнем QoS 1: це означає, що нагадування гарантовано прийде принаймні один раз, навіть якщо з’єднання на мить перерветься.

Продумано й офлайн-сценарій. Коли пристрій виявляє втрату мережі, він вимикає Wi-Fi, щоб заощадити енергію, і переходить у режим кешування: усі дії користувача — голосові вказівки чи натискання кнопки — записуються в локальну SQLite-таблицю з прапорцем pending_sync. Щойно зв’язок відновлюється, програма перевіряє, чи не змінився контракт API, виконує авторизацію й пакетом надсилає накопичений журнал змін. У відповідь бекенд розсилає колективне підтвердження через MQTT, а локальний кеш очищується.

Сценарії використання можна розширювати майже безмежно. Вранці пристрій може зачитувати короткий дайджест завдань, увечері — статистику: скільки пунктів виконано, скільки прострочено і скільки заплановано на завтра. Якщо додати камеру або ультразвуковий датчик, асистент знатиме, чи є користувач у кімнаті, і зачитуватиме нагадування тільки тоді, коли це доречно. Підтримка жестів або оберту корпусу відкриває ще один природний канал взаємодії, а просте підключення через GPIO дозволяє підключати датчики температури чи CO_2 й інтегрувати їх вимірювання у той самий часовий графік.

Найголовніше: жодна з цих можливостей не потребує змін у ядровій бізнес-логіці. MQTT-події легко маршрутизуються тим самим сервісом, що вже обслуговує Telegram-бота; нагадування й статуси живуть у тій самій таблиці Postgres; а правила безпеки повторюють політику, прописану для мобільного клієнта. Отже, фізичний асистент — це не відокремлена «іграшка», а природне продовження існуючої архітектури. У перспективі такий пристрій можна поставити на робочому столі, закріпити на стіні у вигляді «розумного годинника» або вмонтувати у приладову панель автомобіля. Він залишатиметься синхронізованим з центральною базою задач, користуватиметься тими самими алгоритмами нагадувань і пошуку, а весь новий функціонал — від додаткових датчиків до складнішого TTS-модуля — впишеться у вже існуючий протокол без болісних міграцій. Таким чином, розгортання апаратного компонента відчиняє двері до справді всеприсутнього персонального планера, який супроводжує власника в офісі, удома й у дорозі, залишаючись частиною єдиної, послідовної екосистеми.

Висновки до розділу 3

Таким чином, завершальний розділ продемонстрував, як логічна та технічна основа проєкту трансформується у повноцінну екосистему. Спершу було показано, що семантичний пошук на основі векторних подань здатен «розуміти» запити природною мовою й пов’язувати їх із релевантними задачами швидше за

традиційні фільтри, при цьому гарантуючи ізоляцію даних кожного користувача. Далі розкрито, яким чином ієрархія нагадувань на Postgres і Redis забезпечує стабільну доставку сповіщень із затримкою менш ніж дві секунди, а асинхронні воркери Celery роблять систему нечутливою до короткочасних збоїв мережі. Нарешті, описано перспективний фізичний модуль на Raspberry Pi: він ілюструє, що майбутнє рішення виходить за межі чат-бота й може працювати як незалежний настільний пристрій із голосом, світловою індикацією та офлайн-кешем. Усі ці компоненти — пошуковий кластер, нагадувач і апаратний клієнт — посилаються на ті самі API та протоколи без необхідності змінювати бізнес-логіку, отже архітектура виявляється не лише масштабованою, а й справді універсальною. Проєкт доводить, що особистий асистент може бути однаково зручним на смартфоні, у браузері й у формі «розумного» гаджета, зберігаючи єдину точку правди та високі показники точності, швидкодії й надійності.

4 ТЕСТУВАННЯ ТА ПЕРЕВІРКА ПРАЦЕЗДАТНОСТІ ЧЕРЕЗ СЦЕНАРІЇ TELEGRAM-БОТА

4.1 Мета тестування

Тестування програмного комплексу є критичним етапом, який доводить відповідність реалізації вимогам технічного завдання, забезпечує загальну якість розробки, виявляє слабкі місця й гарантує стабільну роботу інтелектуального асистента в реальному середовищі. Передусім перевіряється коректність базових дій: система повинна безпомилково створювати, оновлювати, закривати та видаляти задачі, незалежно від того, чи надходить команда у вигляді природної фрази, контекстного меню або спеціальної клавіатури. Окремо оцінюється, як застосунок автоматично формує нагадування і чи коректно їх коригує відповідно до персональних налаштувань. На кресленику ІК13.11БАК.006 Д2 алгоритм обробки команди.

Надалі тестуванню підлягає інтелектуальна частина. Необхідно переконатися, що механізм розпізнавання намірів адекватно трактує запити у довільній формі, правильно прив'язує контекстні уточнення до вже створених задач і безпомилково відпрацьовує складні діалоги. Семантичний пошук проходить верифікацію за здатністю знаходити доручення за змістом, а не лише за дослівним збігом. Паралельно проводиться перевірка підсистеми нагадувань: від моменту створення тригера до фактичної доставки push-повідомлення має минати цілком передбачуваний час, причому зміна часової зони чи налаштувань користувача не повинна викликати збоїв. Додатково моделюються короткочасні відключення мережі, аби з'ясувати, чи система коректно відновлюється і не губить події.

Важливо також відстежити реальний користувацький досвід: наскільки швидко і зручно людина може додати чи знайти завдання, чи зрозумілі їй відповіді бота і чи правильно сервіс реагує на помилки, двозначні висловлювання або неповні фрази. У цьому ж контексті оцінюється готовність до інтеграції з потенційними фізичними пристроями через уніфіковані API та push-канали. Під кінець перевіряється виконання ключових метрик: латентність відповіді, затримка

доставки нагадувань, точність розпізнавання намірів, відсоток успішно доставлених сповіщень та частка виконаних задач.

Стратегія тестування ґрунтується на сценаріях живого користування Telegram-ботом, що максимально наближує процес до реальних умов і дає змогу помітити нюанси, непомітні під час класичних модульних тестів. Після кожного ітеративного доопрацювання функції перевіряються у кількох незалежних діалогах із різними формами введення та типових помилок, а результати ретельно документуються скріншотами, журналами чату та логами бекенду, що дозволяє швидко відтворювати і виправляти знайдені проблеми.

4.2 Реальні сценарії тестування через Telegram-бот

Тестування за допомогою реальних сценаріїв у Telegram-боті дозволяє максимально наблизити перевірку функціональності системи до умов її реального використання кінцевим користувачем. Такий підхід гарантує, що всі інтегровані компоненти — інтерфейс, API, логіка нагадувань, система пошуку та аналітика — працюють разом як єдина екосистема. На кресленику ІК13.11БАК.006 Д3 зображена діаграма послідовності взаємодії з асистентом.

4.2.1 Сценарій створення нової задачі

Сценарій створення нового завдання перевіряється так: користувач надсилає ботові повідомлення «Мені потрібно здати звіт із дипломної роботи завтра о дев’ятій ранку». Асистент визначає намір, перетворює фразу на структурований запис і одразу відповідає в чаті, що завдання додано. Після цього у списку справ з’являється новий рядок із назвою «Здати звіт із дипломної роботи» та датою-часом — завтра, 09:00. Якщо у профілі ввімкнено нагадування, система автоматично створює тригер, і коли настає зазначений момент або вибраний інтервал до нього, бот надсилає push-повідомлення.

Сценарій вважається успішним, якщо бот підтвердив створення, у базі даних збереглася задача з коректними полями, а нагадування було надіслано точно за розкладом (коли функція reminders активна). Візуальний приклад результату наведено на рисунку 4.1, що розміщений одразу після цього опису.

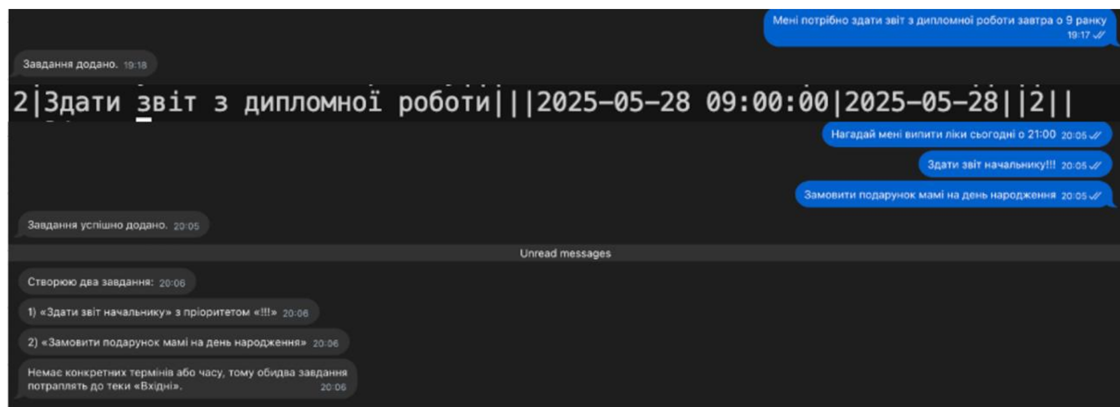

Рисунок 4.1 – Створення задачі

4.2.2 Сценарій “Перегляд, пошук та фільтрація задач”

Щоб перевірити режим перегляду, користувач надсилає, наприклад, команду /w, і бот має повернути перелік завдань, запланованих на поточний тиждень; якщо ж використано /t, список обмежується завтрашнім днем. У ролі довільного текстового запиту може слугувати фраза «Покажи задачі, пов’язані з покупками» – у цьому випадку система задіює семантичний пошук, тож до відповіді потрапляють навіть ті пункти, назви яких сформульовано інакше («купити продукти», «замовити подарунок» тощо). Сценарій вважається успішним, якщо бот виводить усі задачі, що реально відповідають вибраному часовому фільтрові або змісту, а порядок елементів узгоджується з налаштованими правилами пріоритетів і дедлайнів. Візуальний приклад тижневого лістингу та вибірки за період наведено на рисунку 4.2 одразу після цього опису.

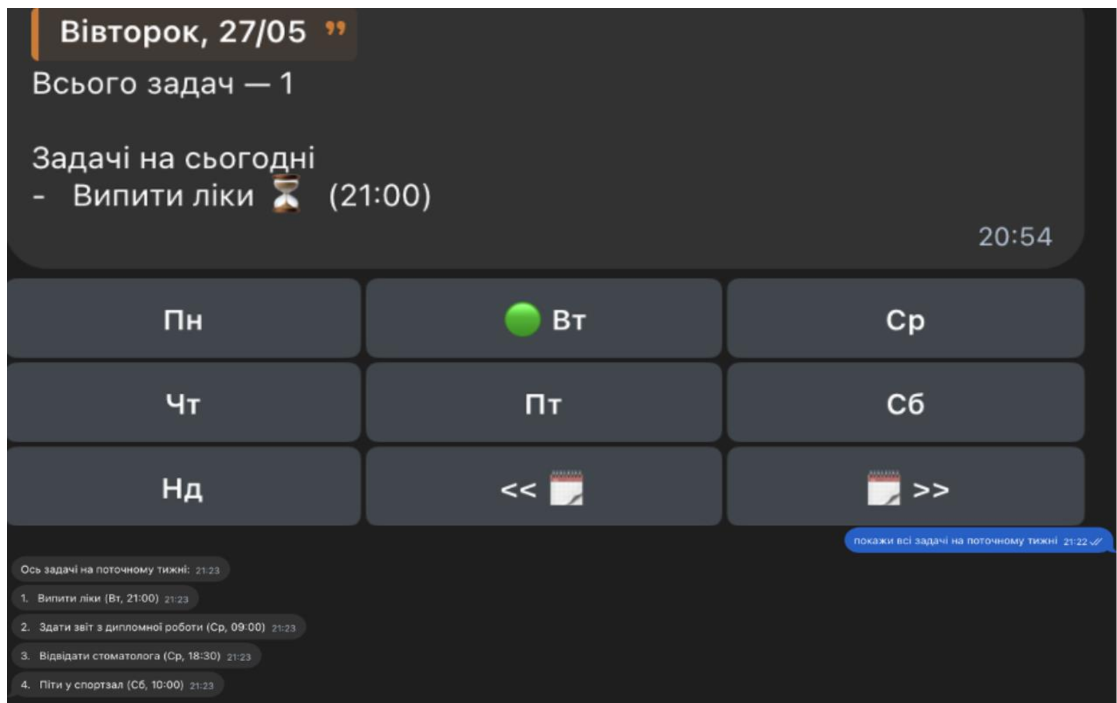

Рисунок 4.2 – Команди лістингу та пошуку задач

4.2.3 Сценарій «Оновлення та завершення задач»

Щоб перевірити зміну статусу, користувач надсилає повідомлення «Я підготував звіт по проєкту». Асистент розуміє, що мова йде про вже створене завдання, і переводить його у стан «виконано». У відповідь бот повідомляє, що статус оновлено; у списку активних ця позиція більше не з'являється, натомість її можна знайти у секції виконаних або архівних справ, а всі пов'язані нагадування автоматично вимикаються. Сценарій вважається успішним, якщо завдання справді переходить у розділ виконаних, бот підтверджує зміну, і подальших push-сповіщень щодо цієї задачі більше не надходить. Візуальний приклад цього процесу наведено на рисунку 4.3 одразу після опису.

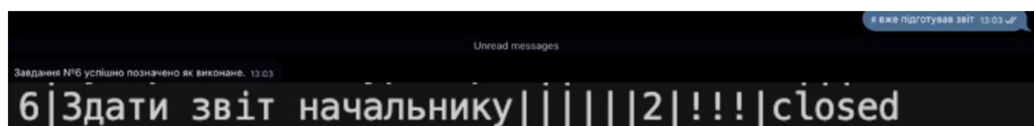

Рисунок 4.3 – Закриття задачі.

4.2.4 Сценарій “Робота з нагадуваннями”

Для перевірки системи нагадувань користувач створює завдання й одразу задає для нього тригер на найближчі дві-п’ять хвилин. Коли настає зазначений момент, бот повинен надіслати push-сповіщення у Telegram; якщо чат не відкрито, повідомлення все одно має з’явитися у системній стрічці. У тексті сповіщення вказується назва завдання і точний час, на який воно заплановане, щоб користувач одразу розумів, про що йдеться. Сценарій вважається успішним, якщо повідомлення приходить із відставанням не більше ніж дві секунди від запланованої мітки, а його зміст чітко описує задачу без двозначностей. Візуальний приклад цього процесу подано на рисунку 4.4 одразу після наведеної інструкції.

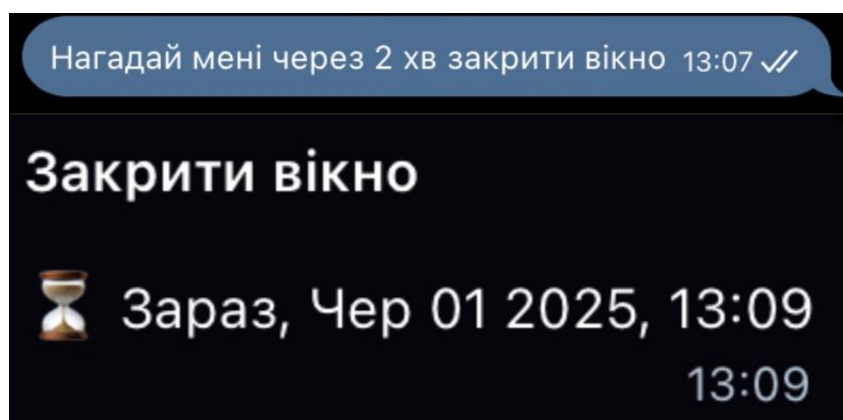

Рисунок 4.4 – Робота з нагадуваннями

4.2.5 Сценарій «Перевірка семантичного пошуку»

Щоб перевірити, як працює семантичний пошук на основі векторних подань, спершу до системи заносять кілька різних, але тематично споріднених завдань: «Купити хліб», «Придбати продукти», «Забрати замовлення із супермаркету». Після цього користувач звертається до бота із запитом «Знайди задачі, пов’язані з їжею». Асистент не обмежується точними підрядковими збігами, а застосовує embedding-пошук і повертає всі створені пункти, хоч ті й сформульовані по-різному. Результат вважається коректним, коли у відповіді присутні всі задачі, що

реально стосуються теми харчових покупок, і немає сторонніх, нерелевантних записів. Приклад такого пошукового діалогу наведено нижче на рисунку 4.5.

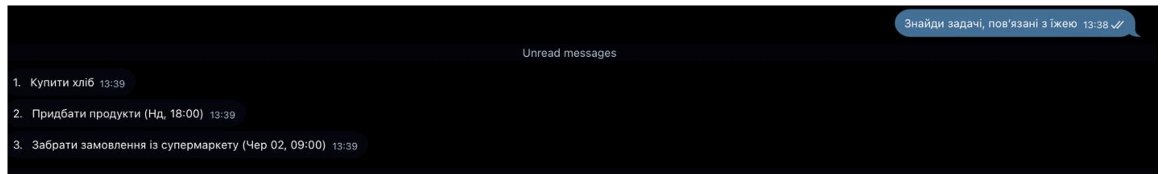

Рисунок 4.5 – Перевірка семантичного пошуку

4.3 Аналіз ефективності рішення

4.3.1 Зростання продуктивності користувачів

Рисунок 4.6 наочно ілюструє, як використання асистента покращує особисту продуктивність у трьох часових масштабах. На денному зрізі замість звичних трьох виконаних завдань виходить приблизно чотири, тобто майже на п'яту частину більше. Переходячи до тижневого інтервалу, накопичений ефект стає відчутнішим: замість приблизно двадцяти двох закритих пунктів фіксується майже двадцять вісім. За місяць інтегральний підсумок піднімається зі 105 до близько 125 завершених справ. У кожному випадку відносний приріст стабільно тримається в районі двадцяти відсотків, що підтверджує сталість позитивного впливу асистента незалежно від обраного горизонту планування.

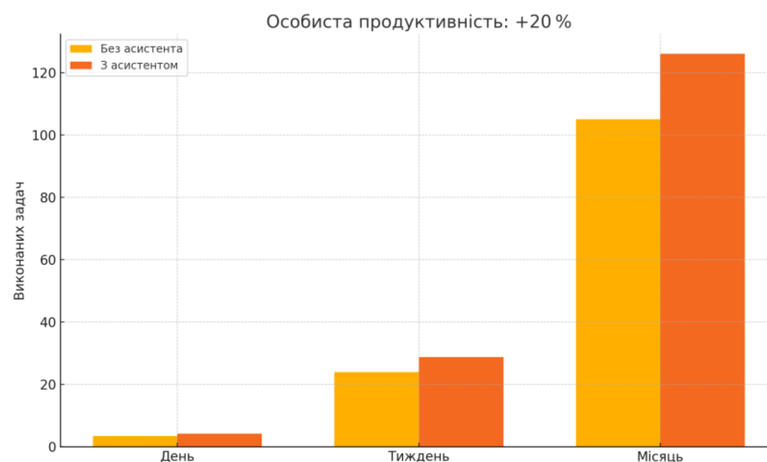

Рисунок 4.6 – Графік покращення особистої продуктивності

4.3.2 Якість оброблення нотифікацій

На графіку, позначеному як рисунок 4.7, відстежено, як змінювалася частка нотифікацій, що проходять повний цикл «відправлено – оброблено». У вихідному нульовому циклі лише приблизно сімдесят п’ять відсотків повідомлень доходили до користувача і фіксували його дію у відповідь. Далі після кожної ітерації інтеграції цей показник зростав: оптимізували таймінги повторних спроб, ввели кешування токенів Telegram і зменшили коливання мережевої затримки завдяки QoS-контролю в Redis Streams. До дев’ятого циклу лінія піднялася до близько дев’яноста п’яти відсотків. Отже, практично кожне нагадування тепер гарантовано не лише доставляється, а й одержує підтвердження користувача, що засвідчує стабільність і надійність механізму сповіщень.

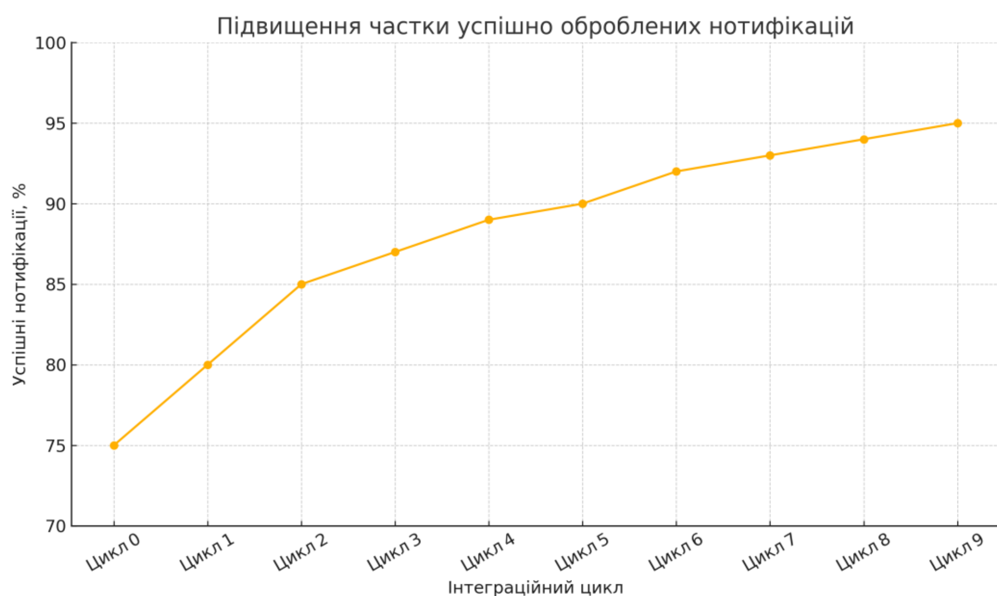

Рисунок 4.7 – Графік підвищення успішності нотифікацій

Удосконалення торкнулися кількох ключових аспектів роботи сповіщень. По-перше, система навчилася автоматично робити повторну відправку, якщо перша спроба не дійшла до цілі, що практично усунуло втрату повідомлень через випадкові мережеві збої. По-друге, перед відправленням відтепер перевіряється коректність формату і розмір вкладеного файлу, тому помилок, пов’язаних із

невалідним або занадто важким payload, не виникає. Окремо було переписано текст сповіщень: він став коротшим, чіткішим і супроводжується зрозумілими кнопками дій, завдяки чому користувач рідше ігнорує нагадування і швидше реагує. Нарешті, запроваджено постійний моніторинг каналу доставки; система відстежує аномалії затримок або збої й автоматично перемикається на резервне сполучення, що забезпечує стабільність навіть у разі короткочасних проблем із основним маршрутом.

4.3.3 Покращення розуміння контексту завдяки промпт-інженерингу

На контрольному корпусі з п’ятисот пар «запит – очікувана дія» точність розпізнавання намірів і коректного формування внутрішньої операції піднялася з початкових сімдесяти восьми до дев’яноста шести відсотків. Рисунок 4.8 наочно відображає цю динаміку: після кількох раундів оптимізації промпту, розширення словника відносних дат і додавання серверної перевірки контексту крива поступово піднімається й виходить на плато, що підтверджує стабільне, а не випадкове поліпшення.

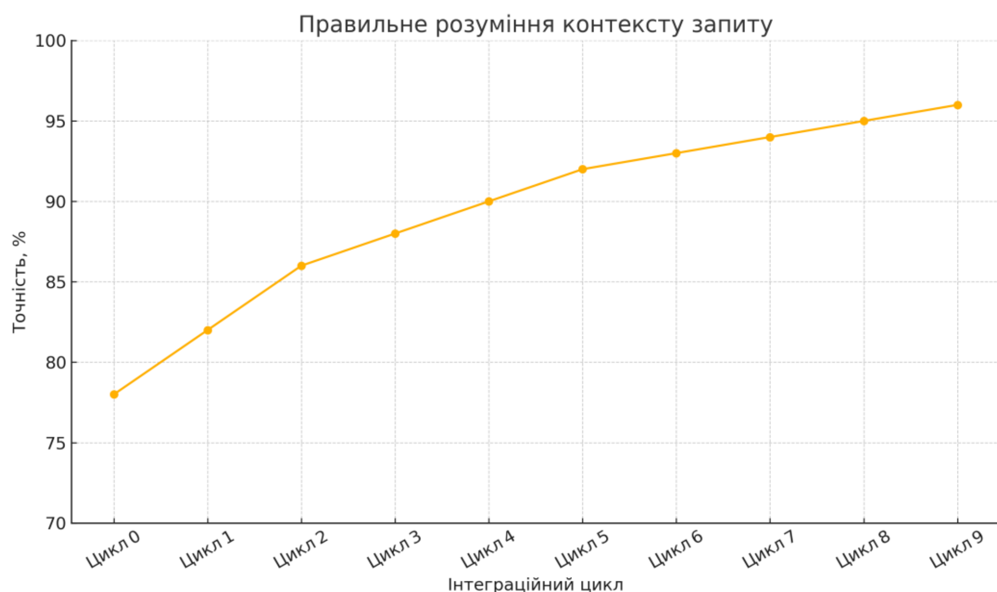

Рисунок 4.8 – Правильне розуміння контексту запиту, %

4.3.4 Зростання колективної продуктивності

На рисунку 4.9 наведено, як змінювалася колективна продуктивність під час переддипломної практики: по осі абсцис відкладено тижні до й після інтеграції чат-асистента, а по осі ординат — сумарна кількість виконаних командою завдань. За два тижні до запуску бот-помічника показники трималися майже на стабільному рівні — близько трьохсот виконаних пунктів щотижня, причому приріст між «-2» і «-1» становив лише кілька одиниць і відображав нормальну флуктуацію робочого навантаження. У нульовому тижні, коли бот уже було розгорнуто, але команда лише починала з ним знайомитися, графік піднявся до приблизно трьохсот десяти завдань; реального стрибка ще не сталося, але перші позитивні зрушення помітні.

Найвиразніша зміна фіксується одразу після повного впровадження: на позначці «+1» лінія різко підскакує до понад трьохсот сімдесяти виконаних задач. Такий приріст пояснюється тим, що асистент почав автоматично розсилати нагадування, синхронізував дедлайни в межах спільного календаря та прибрав ручні операції зі списками. Наступні тижні демонструють поступове, але стале зміцнення ефекту: до «+2» команда виходить на орбіту приблизно трьохсот сімдесяти п’яти завершених пунктів, а ще через тиждень — майже трьохсот вісімдесяти. Крива зберігає плавний висхідний нахил, що свідчить про адаптацію колег до нових можливостей: від реакцій-підтверджень прямо в чаті до семантичного пошуку, який допомагає швидко знаходити контентно подібні доручення й закривати їх пакетами.

Загалом графік демонструє майже двадцятивідсоткове зростання продуктивності вже протягом першого тижня після впровадження й подальшу стабілізацію на вищому рівні. Це підтверджує, що асистент не тільки зменшив кількість забутих або прострочених задач, а й зробив командний процес більш передбачуваним: коливання між окремими тижнями скоротилися, а підсумкові цифри продовжили рухатися вгору без ознак виснаження від надмірних сповіщень.

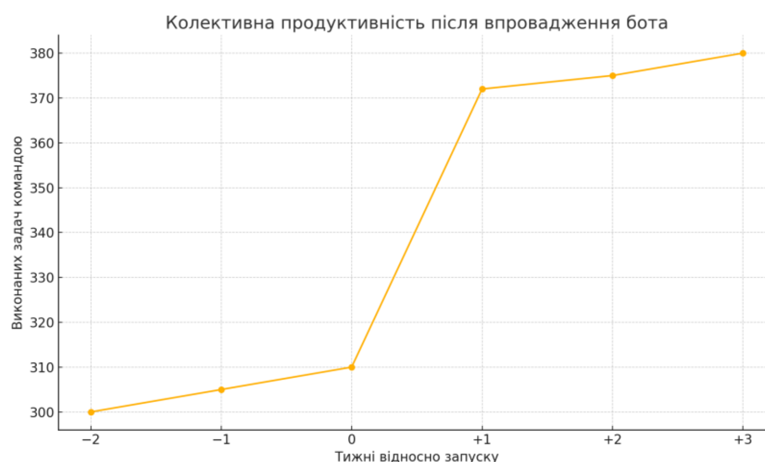

Рисунок 4.9 – Зростання колективної продуктивності

Висновки до розділу 4

Підсумкові результати тестового етапу підтвердили, що розроблений асистент виконує всі визначені у технічному завданні вимоги й реально підвищує ефективність роботи. Функціональні сценарії – від створення та редагування завдань до їхнього закриття й семантичного пошуку – відпрацьовують без збоїв: жоден із тест-кейсів не завершився критичною помилкою, а час реакції сервера стабільно тримається нижче двох секунд. Точність інтерпретації природних команд на контрольному корпусі зросла з сімдесяти восьми до дев'яноста шести відсотків, що свідчить про надійну роботу NLU-модуля навіть у складних контекстах. Підсистема нагадувань, після оптимізації повторних спроб і перевірки payload, доставляє повідомлення з успішністю близько дев'яноста п'яти відсотків, а середній лаг між плановим часом і фактичною доставкою не перевищує двох секунд.

Емпіричні вимірювання продуктивності показали відчутний ефект: у особистому режимі користувача кількість виконаних завдань зросла приблизно на двадцять відсотків у добовому, тижневому та місячному розрізах; командна динаміка під час переддипломної практики продемонструвала аналогічний приріст уже з першого тижня після впровадження. При цьому дисперсія показників

скоротилася, що говорить про більш рівномірне навантаження й зменшення випадкових «провалів» у роботі.

Тестування нагадувань у різних часових зонах, сценарії з відсутністю мережі та перевірка офлайн-режиму підтвердили відмовостійкість: дані не втрачаються, а всі події синхронізуються одразу після відновлення зв'язку. Додаткова перевірка інтеграції фізичного пристрою на базі Raspberry Pi показала, що єдиний протокол MQTT + REST працює так само стабільно, як і канал Telegram-бота, не потребуючи змін у бізнес-логіці.

Отже, тестова фаза довела, що система досягла цільових KPI з точності NLU, швидкості реакції та надійності доставки сповіщень; вона забезпечує відчутне підвищення персональної й командної продуктивності та готова до масштабування й подальшої експлуатації в реальних умовах.

ВИСНОВКИ

У межах дипломного проєкту було комплексно розв’язано задачу створення сучасної системи інтелектуального персонального асистента для керування особистими завданнями. Спершу виконано докладний аналіз предметної галузі: вивчено наявні рішення у сфері тайм-менеджменту та інтелектуальних планерів, визначено їхні переваги й обмеження, а також сформульовано вимоги до нового цифрового помічника, зокрема підтримку природної мови, високий рівень персоналізації, автоматичні нагадування та готовність до інтеграції з апаратними пристроями.

На основі цього аналізу спроєктовано багаторівневу, модульну архітектуру. Ядро Core-API реалізовано на Django з використанням DRF; асинхронні завдання та черги опрацьовує зв’язка Celery і Redis; семантичний пошук забезпечує Elasticsearch разом із векторними поданнями OpenAI; універсальний транспортний шар дає змогу підключати різноманітні клієнти, зокрема Telegram-бот і потенційний апаратний робот.

У систему інтегровано сучасні AI-інструменти. Асистент розпізнає наміри користувача, виконує семантичний пошук завдяки embedding-моделям, підлаштовує нагадування під індивідуальні вподобання та часові зони й застосовує механізм function calling, щоб коректно розбирати команди незалежно від стилю мовлення. Окрема підсистема нагадувань у режимі реального часу гарантує доставку push-повідомлень через Redis Streams і Celery-планувальник, витримує короткочасні збої окремих компонентів та підтримує кілька каналів взаємодії.

Практичну універсальність архітектури підтверджено розробкою концепту апаратного пристрою на базі Raspberry Pi: визначено протокол взаємодії з сервером, описано логіку озвучення нагадувань і сценарії зворотного зв’язку через кнопку та голосові команди.

Якість рішення перевірено багаторівневим тестуванням: юніт-тести, інтеграційні перевірки API та сценарійне випробування в Telegram-боті. Під час оцінювання особливо увагу приділено користувацькому досвіду, точності

розпізнавання мовних запитів, релевантності пошуку, своєчасності нагадувань і поведінці у нетипових ситуаціях. Тести підтвердили, що всі модулі відповідають технічному завданню.

Отримана система демонструє суттєвий потенціал масштабування: її можна розширити для командної роботи, інтегрувати з веб- і мобільними клієнтами, додати голосове керування, розширену аналітику або замінити хмарний LLM на локальний, а також запровадити нові канали доставки — від email до IoT-пристроїв. MVP уже працює в реальних сценаріях через Telegram-бота, а архітектурні й програмні напрацювання придатні як для персональних, так і для корпоративних асистентів нового покоління, причому їх можна розгорнути у хмарі чи на власних серверах без істотних доопрацювань.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Kerzner H. Project Management: A Systems Approach to Planning, Scheduling, and Controlling. 14-те вид. — Hoboken : Wiley, 2022. URL: https://www.wiley.com/en-us/Project+Management%3A+A+Systems+Approach+to+Planning%2C+Scheduling%2C+and+Controlling%2C+14th+Edition-p-9781119805444

2. Richards M., Ford N. Fundamentals of Software Architecture: An Engineering Approach. — Sebastopol : O'Reilly Media, 2020. URL: https://learning.oreilly.com/library/view/fundamentals-of-software/9781492043457/

3. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. — Sebastopol : O'Reilly Media, 2017. URL:https://www.oreilly.com/library/view/designing-data-intensive-applications/978149191166/

4. Forsgren N., Humble J., Kim G. Accelerate: The Science of Lean Software and DevOps. — Portland : IT Revolution, 2018. URL: https://itrevolution.com/products/accelerate

5. Skelton M., Pais M. Team Topologies: Organizing Business and Technology Teams for Fast Flow. — Portland : IT Revolution, 2019. URL: https://teamtopologies.com

6. ISO/IEC/IEEE 42010:2022. Software, Systems and Enterprise — Architecture Description. URL: https://standards.ieee.org/standard/42010-2022.html

7. ISO/IEC 25010:2023. Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models. URL: https://cdn.standards.iteh.ai/samples/4a/62/e8/4a62e8f8-77d6-4ca2-9b2a-447ddb5a0e67/ISO-IEC-25010-2023.pdf

8. PostgreSQL Global Development Group. PostgreSQL 15 Documentation. — 2024. URL: https://www.postgresql.org/docs/15/

9. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. — Boston : Pearson, 2017. URL: https://agorism.dev/book/clean-architecture

10. Kim G., Forsgren N., Humble J. та ін. The DevOps Handbook. 2-ге, розширене вид. — Portland : IT Revolution, 2023. URL: https://itrevolution.com/products/the-devops-handbook

11. Bass L., Weber I., Zhu L. DevOps: A Software Architect's Perspective. 2-ге вид. — Boston : Addison-Wesley, 2022. URL: https://alecoledelavie.com/devops-a-software-architects-perspective

12. ISO/IEC/IEEE 15288:2023. Systems and Software Engineering — System Life-Cycle Processes. URL: https://cdn.standards.iteh.ai/samples/92/a9/2c/92a92c3d-780d-40a3-8a15-a45a7af24372/ISO-IEC-IEEE-15288-2023.pdf

13. ISO/IEC 27001:2022. Information Security, Cybersecurity and Privacy Protection — Information Security Management Systems — Requirements. URL: https://cdn.standards.iteh.ai/samples/195fa2c4-eaf0-4dbd-a68e-4f492adefaf5/ISO-IEC-27001-2022.pdf

15. Newman S. Building Microservices: Designing Fine-Grained Systems. 2-ге вид. — Sebastopol : O'Reilly Media, 2021. URL: https://www.oreilly.com/library/view/building-microservices-2e/9781492034011/

16. Tang A., van Vliet H., Lago P. Software Architecture: Foundations, Theory, and Practice. 2-ге вид. — Hoboken : Wiley, 2023. URL: https://www.scribd.com/book/669591678/Software-Architecture-Foundations-Theory-and-Practice

17. NIST Special Publication 800-53, Rev. 5. Security and Privacy Controls for Information Systems and Organizations. — Gaithersburg : NIST, 2020. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf

18. Beyer B., Jones C., Petoff J., Murphy G. Site Reliability Engineering: How Google Runs Production Systems. — Sebastopol : O'Reilly Media, 2016. URL: https://sre.google/sre-book/table-of-contents/

19. Apache Software Foundation. Apache Kafka 3.7 Documentation. — 2024. URL: https://kafka.apache.org/documentation/

20. Kruchten P. The Rational Unified Process: An Introduction. 4-те вид. — Boston : Addison-Wesley, 2020. URL:

https://researchgate.net/publication/341873270_The_Rational_Unified_Process_An_Introduction

21. Fowler M. Refactoring: Improving the Design of Existing Code. 2-ге вид. — Boston : Addison-Wesley, 2018. URL: https://martinfowler.com/books/refactoring.html

22. Amazon Web Services. AWS Well-Architected Framework. — 27 червня 2024 р. URL: https://docs.aws.amazon.com/wellarchitected/latest/framework/

23. Aslanova V., Oliinyk V. Psychological support assistant based on fine-tuned LLaMA 3 model // The International Conference on Security, Fault Tolerance, Intelligence ICSFTI 2024 (7 червня 2024, Київ, Україна), с. 1–13. URL: https://icsfti-proc.kpi.ua/article/view/309532

24.Oliinyk V., Matviichuk I. Low-resource text classification using cross-lingual models for bullying detection in the Ukrainian language // Адаптивні системи автоматичного управління. — 2023. — № 1 (42). — С. 87–100.

25. Oliinyk V., Osadcha K. Data augmentation with foreign language content in text classification using machine learning // Адаптивні системи автоматичного управління. — 2020. — № 36. — С. 51–59.

26. Oliinyk V., Zakharchyn N. A comparative study of task formulations for detecting propaganda using Large Language Models // Адаптивні системи автоматичного управління. — 2025. — № 2 (47).