МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра комп'ютерних наук

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«ТЕХНОЛОГІЇ ІНТЕГРАЦІЇ ТЕЛЕГРАМ-БОТІВ З ЗОВНІШНІМИ СЕРВІСАМИ»**

Виконав: студент 4 курсу, групи 6.1221
спеціальності 122 Комп'ютерні науки
(шифр і назва спеціальності)
освітньої програми Комп'ютерні науки
(назва освітньої програми)
Р.Ю. Назарук
(ініціали та прізвище)
Керівник професор кафедри комп'ютерних наук, д.е.н., доцент, Полуектова Н.Р
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент завідувач кафедри програмної інженерії, к.ф.-м.н., доцент Лісняк А. О.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра комп'ютерних наук

Рівень вищої освіти бакалавр

Спеціальність 122 Комп'ютерні науки
(шифр і назва)

Освітня програма Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних наук,
д.т.н., професор

_____ Шило Г.М..
(підпис)

" ____ " _____ 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Назаруку Роману Юрійовичу
(прізвище, ім'я та по-батькові)

1. Тема роботи Технології інтеграції телеграм-ботів з зовнішніми сервісами

керівник роботи Полуектова Наталія Робертівна., д.е.н., доцент
(прізвище, ім'я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 20 » 12 2024 року № 2001-с

2. Строк подання студентом роботи 15.05.2024 р

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.
3. Розробка додатку для інтеграції телеграм боту з зовнішніми сервісами.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
презентація

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання _____

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	01.02.2025	
2.	Збір вихідних даних.	01.03.2025	
3.	Обробка методичних та теоретичних	25.03.2025	
	джерел.		
4.	Розробка першого та другого розділу.	10.04.2025	
5.	Розробка третього розділу.	05.05.2025	
6.	Оформлення та нормоконтроль	14.05.2025	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	18.06.2025	

Студент _____ Р.Ю. Назарук
(підпис) (ініціали та прізвище)

Керівник роботи _____ Н.Р. Полуектова
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ О.Г.Спиця
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Технології інтеграції телеграм-ботів з зовнішніми сервісами»: 59 с., 9 рис., 1 табл., 13 джерел, 7 додатків.

API, DEEPSEEK, GOOGLE SHEETS, KAFKA, ЗОВНІШНІ СЕРВІСИ, ІНТЕГРАЦІЯ, ТЕЛЕГРАМ-БОТ.

Об’єкт дослідження: технології автоматизації взаємодії з користувачами соціальних мереж.

Предмет дослідження: технології та інструменти для взаємодії Telegram-ботів із зовнішніми ресурсами.

Мета роботи: розробка та впровадження методів інтеграції із різноманітними зовнішніми сервісами для автоматизації обміну даними в боті.

Методи дослідження: індукція; дедукція; опитування та спостереження; порівняння; узагальнення.

В роботі досліджені методи та засоби підключення Telegram-ботів до зовнішніх сервісів та розроблений додаток, який дозволяє звертатися до корисних сервісів і має прикладне значення.

SUMMARY

Bachelor's qualification work "Technologies for Integrating Telegram Bots with External Services": 59 p., 9 fig., 1 tables, 13 sources, 7 appendices.

API, DEEPSEEK, GOOGLE SHEETS, KAFKA, EXTERNAL SERVICES, INTEGRATION, TELEGRAM BOT.

Object of research: Technologies for automating interaction with users of social networks.

Subject of research: technologies and tools for the interaction of Telegram bots with external platforms.

Purpose of work: development and implementation of methods for integrating Telegram bots with various external services to automate data exchange in the bot.

Research method: induction; deduction; survey and observation; comparison; generalization.

The work explores methods and means of connecting Telegram bots to external services and develops an application that allows you to access useful services and has applied value.

ЗМІСТ

Завдання на кваліфікаційну роботу……..…..2
Реферат 4
Summary 5
Вступ..... 8
1 Аналіз технологій створення та інтеграції телеграм-ботів..... 9
1.1 Огляд telegram api та архітектури телеграм-ботів 9
1.2 Протоколи та механізми взаємодії з ботами 10
1.2.1 Long polling..... 10
1.2.2 Webhooks 11
1.3 Огляд технологій та платформ для створення ботів 11
1.3.1 Python-боти..... 12
1.3.2 Node.js-боти..... 12
1.3.3 Serverless-рішення..... 12
2 Способи інтеграції телеграм-ботів з зовнішніми сервісами..... 14
2.1 Використання rest api для взаємодії з веб-сервісами 14
2.2 Інтеграція ботів із базами даних (postgresql, mysql, mongodb) 15
2.2.1 Postgresql..... 16
2.2.2 Mysql 17
2.2.3 Mongodb..... 17
2.3 Використання сторонніх сервісів..... 18
2.3.1 Google sheets..... 18
2.3.2 Openweather 19
2.3.3 Gpt-моделі..... 19
2.3.4 Платіжні системи..... 20
2.4 Взаємодія з мікросервісною архітектурою та брокерами повідомлень 20
2.4.1 Asyncio 21

2.4.2 Kafka 21
3 Розробка телеграм-бота та його інтеграція з зовнішніми сервісами 23
3.1 Постановка задачі та вимоги до системи 23
3.2 Вибір технологій та інструментів 23
3.3 Розробка серверної частини бота 24
3.4 Інтеграція з базою даних та зовнішніми api 25
3.5 Тестування та оптимізація роботи бота 28
Висновки 34
Перелік посилань 36
Додаток А Лістинг файлу «app.py» 38
Додаток Б Лістинг файлу docker-compose.yml 40
Додаток В Лістинг файлу dockerfile 42
Додаток Г Лістинг файлу requirements.txt 43
Додаток Д Лістинг файлів директорії handlers 44
Додаток Е Лістинг файлів директорії services 53
Додаток Ж Лістинг файлу «.env» 59

ВСТУП

У контексті сучасних інформаційних систем швидкість та надійність обміну даними між користувачем та зовнішніми платформами набуває все більшої значущості. Telegram-боти, завдяки своїй гнучкості та широким можливостям API, виступають ефективним засобом автоматизації інформаційних процесів, інтегруючи месенджер із REST-сервісами, базами даних, електронними таблицями й інтелектуальними моделями. Однак застосування цих технологій ставить перед розробником завдання вибору оптимальних архітектурних підходів, забезпечення коректності передачі даних і підтримки масштабованості системи.

Метою даної роботи є проєктування й реалізація прототипу Telegram-бота, здатного здійснювати асинхронні запити до зовнішніх ресурсів та надійно зберігати історію взаємодій у реляційній базі даних MySQL. В рамках дослідження передбачається виконати системний аналіз існуючих методів інтеграції, обґрунтувати вибір стеку технологій і реалізувати механізми обміну даними з REST API, Google Sheets, OpenWeather, OpenAi.

Для виконання дипломної роботи було поставлено такі задачі: дослідити архітектуру та можливості Telegram API, описати ключові методи, обмеження та вимоги до авторизації й безпеки; порівняти механізми взаємодії з ботом – polling та webhook, проаналізувати їх налаштування, продуктивність і рівень захищеності; здійснити огляд основних платформ і фреймворків для розробки Telegram-ботів (Python-бібліотеки, Node.js-рішення та serverless-архітектури).

В результаті досліджені методи та засоби підключення Telegram-ботів до зовнішніх сервісів та розроблений додаток, який дозволяє звертатися до корисних сервісів і має прикладне значення.

1 АНАЛІЗ ТЕХНОЛОГІЙ СТВОРЕННЯ ТА ІНТЕГРАЦІЇ ТЕЛЕГРАМ-БОТІВ

1.1 Огляд Telegram API та архітектури телеграм-ботів

Telegram API – це набір інструментів, який дозволяє розробникам створювати додатки для роботи з Telegram. Цей API дозволяє створювати ботів, які отримують і надсилають повідомлення, відповідають на запити користувачів і взаємодіють з іншими сервісами. Головна перевага – простота використання, завдяки якій розробка навіть для початківців не стає важкою задачею. Насправді, Telegram-боти обмінюються даними з серверами Telegram за допомогою HTTP-запитів.

Є два основні способи отримання оновлень:

– long polling: бот надсилає запит і чекає, поки сервер не надішле нову інформацію;
– webhooks: Telegram сам надсилає повідомлення на заздалегідь вказаний URL кожного разу, коли трапляється якась подія.

В першому випадку якщо повідомлення з’являється, воно відразу доставляється, а якщо ні – через певний час запит завершується, і бот робить запит знову. Цей метод простий у налаштуванні і добре підходить для роботи на стандартних серверах.

Другий спосіб дозволяє отримувати дані миттєво, без необхідності постійного опитування, але вимагає наявності сервера з підтримкою HTTPS.

Telegram API пропонує безліч функцій для роботи з фото, відео, групами, каналами та багатьма іншими елементами платформи. Це робить його дуже універсальним інструментом для створення як простих інформаційних ботів, так і складних сервісів, що інтегруються з іншими системами. Бот працює як окремий користувач, якому можна надсилати повідомлення та який може надсилати відповіді, використовуючи HTTP API [1].

Взагалі, завдяки гнучкій архітектурі, телеграм-боти легко масштабуються і можуть працювати як з базами даних, так і з різними зовнішніми сервісами. Розробники можуть використовувати вже готові бібліотеки або писати власний код, що дозволяє адаптувати бот під конкретні потреби проекту.

1.2 Протоколи та механізми взаємодії з ботами

Коли бот спілкується з сервером Telegram, йому потрібно отримувати нові повідомлення та відправляти відповіді. Як згадувалось вище, існують два популярні підходи для цього.

1.2.1 Long Polling

Метод Webhook дозволяє Telegram надсилати оновлення безпосередньо на сервер бота, що значно зменшує затримку [2]. Якщо протягом цього періоду надходить нове повідомлення, сервер негайно передає його клієнту. У разі відсутності даних запит завершується з таймаутом, після чого бот ініціює новий запит.

Переваги такого підходу наступні:

– простота реалізації навіть на базових хостингових рішеннях або локальних серверах;
– відсутність вимог до використання складних SSL-сертифікатів (достатньо базового HTTPS, або HTTP на етапі розробки).
– До недоліків такого підходу можна віднести:
– неминучу затримку між завершенням одного запиту та ініціюванням наступного;
– зростання навантаження на сервер при масштабуванні через постійні запити від клієнтів.

1.2.2 Webhooks

Даний метод передбачає автоматичне надсилання оновлень з сервера Telegram на сервер у момент виникнення події (наприклад, отримання повідомлення).

Для реалізації необхідно:

– зареєструвати публічну URL-адресу сервера (наприклад, https://mybot.com/webhook) у налаштуваннях бота [3];
– налаштувати сервер для обробки вхідних POST-запитів від Telegram.

При надходженні повідомлення сервер Telegram ініціює запит на вказаний вебхук, передаючи відповідні дані. Після успішної обробки сервер має повернути статусний код 200 OK для підтвердження отримання інформації.

Переваги цього методу:

– відсутність надлишкових запитів: оновлення передаються у реальному часі;
– оптимізація ресурсів: знижується навантаження на сервер та мережу.

Недоліки методу:

– обов’язкова наявність публічного сервера з дійсним SSL-сертифікатом (HTTPS є обов’язковою вимогою);
– складність початкової інтеграції та необхідність постійного моніторингу (наприклад, обробка простоїв сервера або відновлення з’єднання).

1.3 Огляд технологій та платформ для створення ботів

Огляд технологій та платформ для створення ботів дозволяє визначити найоптимальніші інструменти для розробки залежно від вимог проєкту та технічної компетентності розробника. Сучасні рішення охоплюють широкий спектр підходів — від класичних серверних бібліотек на Python і Node.js до

гнучких serverless-платформ, що спрощують масштабування та розгортання ботів без необхідності керування інфраструктурою.

1.3.1 Python-боти

Для розробки ботів на Python найчастіше використовують готові бібліотеки: наприклад, python-telegram-bot з базовим синхронним API або aiogram з підтримкою asyncio. За допомогою цих інструментів можна швидко реалізувати обробку команд, клавіатури, роботи з мультимедіа та інтеграцію з базами даних чи іншими сервісами на Python.

1.3.2 Node.js-боти

У середовищі JavaScript широке застосування знаходять бібліотеки node-telegram-bot-api та Telegraf. Дані рішення надають інтуїтивний інтерфейс для конфігурації режимів отримання оновлень (Long Polling або Webhook), а також підтримують архітектуру з використанням проміжного програмного забезпечення (middleware). Інтеграція з іншими модулями екосистеми npm здійснюється без додаткових труднощів, що спрощує розширення функціоналу.

Робота з Node.js Node.js дозволяє створювати ефективні серверniless-додатки завдяки асинхронному I/O та легкій інтеграції з хмарними сервісами [4].

1.3.3 Serverless-рішення

Сучасні технології бессерверних обчислень, такі як Amazon Web Services (AWS) Lambda, Google Cloud Functions, Microsoft Azure Functions та Vercel Functions, дозволяють розгортати логіку бота без необхідності керування серверною інфраструктурою. Користувач завантажує код функції, налаштовує вебхук, а завдання масштабування, резервування ресурсів та обслуговування системи повністю делегуються відповідному хмарному провайдеру. Для

підвищення безпеки Telegram застосовує MTProto — власний протокол шифрування, що забезпечує захист повідомлень у процесі передачі [5].

Цей підхід є особливо ефективним для проектів, де критичними є гнучкість витрат та мінімізація адміністративних накладних витрат. Оплата здійснюється за моделлю pay-as-you-go, тобто лише за фактичний час виконання коду, що виключає необхідність постійної підтримки віртуальних машин або виділених серверів.

2 СПОСОБИ ІНТЕГРАЦІЇ ТЕЛЕГРАМ-БОТІВ З ЗОВНІШНІМИ СЕРВІСАМИ

У даному розділі досліджуються основні стратегії інтеграції Telegram-ботів із зовнішніми платформами. На початку аналізується використання REST API для організації двостороннього обміну даними між ботом та веб-сервісами. Далі розглядається підключення до сучасних баз даних, таких як PostgreSQL, Firebase, MongoDB та MySQL, що забезпечує ефективне зберігання, пошук і модифікацію інформації.

В проєкті буде реалізована взаємодія з Google Sheets API для автоматизації роботи з таблицями та даними, OpenWeather API для отримання актуальної метеорологічної інформації, GPT-моделі для обробки текстів та генерації контенту на основі штучного інтелекту, а також платіжні системи (наприклад, Stripe чи PayPal) для реалізації транзакцій та фінансових операцій (хоча платіжна складова в проєкті не реалізуватиметься через нелоступність платних послуг).

На завершення висвітлюється взаємодія ботів із мікросервісною архітектурою за допомогою брокерів повідомлень (RabbitMQ, Apache Kafka). Це дозволяє забезпечити масштабованість, асинхронну обробку подій та надійність системи.

2.1 Використання REST API для взаємодії з веб-сервісами

REST API (Representational State Transfer) є стандартним протоколом для організації обміну даними між Telegram-ботами та зовнішніми веб-сервісами. Принцип роботи базується на використанні HTTP-методів (наприклад, GET, POST, PUT, DELETE), які направляються до специфічних URL-ендпоінтів. Параметри запиту передаються у форматі JSON або через рядок запиту (query

string), після чого сервіс повертає структуровані дані (зазвичай також у форматі JSON) для подальшої обробки ботом.

REST-інтеграція реалізується за допомогою спеціалізованих бібліотек, що залежать від мови програмування та парадигми виконання запитів. Для Python основним інструментом є бібліотека requests, яка використовується для синхронної взаємодії, тоді як aiohttp застосовується в асинхронних сценаріях для підвищення продуктивності. У середовищі Node.js популярними рішеннями виступають axios або вбудований метод fetch, що дозволяють ефективно працювати з HTTP-запитами.

Процес взаємодії з API починається з формування запиту, де розробник вказує цільовий ендпоінт, метод (GET, POST тощо) та додає необхідні заголовки – зокрема, заголовок авторизації з токеном доступу для ідентифікації клієнта. Після відправки запиту система аналізує статус-код відповіді: код 200 свідчить про успішне виконання, коди 4xx (наприклад, 404 або 401) вказують на помилку з боку клієнта (невірний запит або відсутність прав), а коди 5xx (500, 503) – на проблеми серверної частини.

На остаточному етапі відбувається обробка отриманих даних, яка включає парсинг JSON-відповіді, валідацію структури даних відповідно до очікуваної схеми та інтеграцію результатів у логіку роботи бота, наприклад, формування текстового повідомлення для користувача або оновлення бази даних.Такий підхід забезпечує стандартизовану та гнучку інтеграцію, що робить його основним вибором для взаємодії з веб-сервісами у сучасній розробці.

2.2 Інтеграція ботів із базами даних

Далі були розглянуті методи зберігання та отримання даних із різних типів сховищ, включаючи традиційні реляційні системи та сучасні NoSQL-рішення.

Ефективна організація взаємодії з базами даних є критичною для забезпечення стабільної роботи бота, оскільки дозволяє:

– зберігати історію повідомлень;
– управляти налаштуваннями користувачів;
– обробляти інші операційні дані, необхідні для функціонування системи.

Інтеграція з сучасними базами даних реалізується через використання різних платформ, кожна з яких має унікальні характеристики.

PostgreSQL або MySql, як реляційнs СУБД, забезпечують надійну роботу з транзакціями та підтримують виконання складних SQL-запитів, що робить їх ідеальним вибором для структурованих даних з чіткими зв’язками.

Для сценаріїв, де потрібна робота в реальному часі та автоматичне масштабування, використовується NoSQL-рішення Firebase, яке дозволяє синхронізувати дані між клієнтами без серверної логіки.

Окрему категорію становить документоорієнтована MongoDB, оптимізована для гнучкого зберігання неструктурованих даних у форматі JSON-подібних документів, що спрощує роботу з динамічними схемами та швидку ітерацію в умовах змінних вимог.

2.2.1 PostgreSQL

PostgreSQL – потужна реляційна база даних з підтримкою складних транзакцій і SQL-запитів.

Налаштування підключення: зазвичай використовують бібліотеки psycopg2 (для синхронного доступу) або asyncpg (для асинхронного).

Створення структури: описує таблиці через SQL-скрипти або ORM (наприклад, SQLAlchemy) – визначає поля для зберігання користувацьких налаштувань, логу подій, тощо.

Запити з бота: через виправлені запити SELECT/INSERT/UPDATE бот читає й записує дані. Завдяки транзакціям можна гарантувати цілісність даних навіть при одночасному доступі кількох користувачів.

2.2.2 MySQL

MySQL – реляційна СУБД з підтримкою структурованих запитів (SQL), що широко використовується для зберігання даних у класичних веб-додатках та ботах.

Ініціалізація клієнта. Встановлюємо бібліотеку для роботи з базою (наприклад, mysql-connector-python або aiomysql для асинхронного доступу) та налаштовуємо підключення, вказуючи хост, порт, ім’я користувача, пароль та назву бази даних.

Структура даних. Дані організовано в таблиці зі строгою схемою (рядки та стовпці з визначеними типами). Це забезпечує цілісність та стабільність, але потребує попереднього проєктування структури.

Робота з ботом. Через SQL-запити (SELECT, INSERT, UPDATE, DELETE) бот може зчитувати й змінювати інформацію. Для складніших сценаріїв використовуються транзакції. Оновлення даних не відбуваються в реальному часі автоматично – для цього потрібна додаткова логіка (наприклад, періодичне опитування або WebSocket-підписка через окремий шар).

Однак, слід зазначити, що з менш відомих проблем при використанні MySQL є те, що первинні ключі з автоінкрементом за замовчуванням створюються як цілі числа зі знаком, і діапазон можливих значень для ключа може бути вичерпаний [6].

Саме її було використано для написання телеграм-бота.

2.2.3 MongoDB

MongoDB – документоорієнтована NoSQL-система, заснована на зберіганні BSON-документів.

Підключення: використовуємо офіційну бібліотеку pymongo або асинхронну motor.

Опис колекцій: замість таблиць створюємо “колекції”, а замість рядків – “документи” з гнучкою схемою даних.

CRUD-операції: методи insert_one(), find(), update_one() і delete_one() дозволяють ефективно оперувати даними. Підтримка індексів та агрегаційних запитів допомагає масштабувати бот навіть при великому обсязі інформації.

2.3 Використання сторонніх сервісів

Використання сторонніх сервісів значно розширює функціональні можливості Telegram-бота, дозволяючи йому виконувати складні завдання без необхідності розробки всього з нуля.

По-перше, бот може отримувати актуальні дані в реальному часі – наприклад, через інтеграцію з OpenWeather для надання користувачам актуальної інформації про погоду.

Подруге, за допомогою Google Sheets можлива автоматизація створення звітів, ведення обліку та надсилання сповіщень без потреби у власному бекенді. Також бот може використовувати штучний інтелект – GPT-моделі дозволяють йому аналізувати тексти, генерувати відповіді або створювати контент на запит.

Нарешті, через інтеграцію з платіжними системами, такими як Stripe або PayPal, можна реалізувати фінансові операції, що відкриває шлях до платних функцій, пожертв або повноцінних онлайн-сервісів. Такий підхід робить бота гнучким, масштабованим і здатним легко адаптуватися до потреб користувачів.

2.3.1 Google Sheets

Для роботи з електронними таблицями Telegram-бот може використовувати Google Sheets API, що дозволяє автоматизувати обмін даними без необхідності розгортання власної бази. Процес інтеграції починається з налаштування автентифікації через Google Cloud Console: створюється проект,

активується Google Sheets API, а потім генерується ключ для сервісного акаунта або налаштовується OAuth 2.0 для сценаріїв з участю користувачів. Сервісному акаунту надається доступ до конкретної таблиці через її ідентифікатор або URL, що забезпечує безпеку взаємодії [7].

Для роботи з даними застосовуються спеціалізовані бібліотеки, такі як googleapis для Node.js або gspread для Python. Вони дозволяють виконувати операції читання через метод spreadsheets.values.get, звертаючись до певних діапазонів комірок, а також записувати або оновлювати інформацію за допомогою spreadsheets.values.update. Дані передаються у структурованому форматі, наприклад JSON, що спрощує їх подальшу обробку.

2.3.2 OpenWeather

Інтеграція з OpenWeather API дозволяє отримувати актуальні дані про погоду в будь-якому місті світу. Бот формує HTTP-запит із передачею ключа API та координат або назви населеного пункту, після чого отримує JSON-відповідь з температурою, вологістю, прогнозом на декілька днів тощо [8].

2.3.3 GPT-моделі

Використання GPT-моделей (наприклад, OpenAI GPT) значно розширює можливості Telegram-бота: бот формує структурований HTTP-запит до API, передаючи у запиті промпт (завдання або питання) та параметри відповіді (тон, максимальну довжину, ступінь креативності тощо), а сервіс повертає згенерований текст, який потім відображається користувачу або використовується для подальших операцій.

В роботі було реалізовано взаємодію через бібліотеку openai для Python: після отримання API-ключа та налаштування, він відправляє запити до моделі deepseek/deepseek-r1:free та обробляє відповідь, яка інтегрується в логіку бота

[9]. GPT-моделі особливо корисні для автоматичних довідкових систем, корекції та стилізації тексту, перекладу повідомлень і творчої генерації ідей.

2.3.4 Платіжні системи

Для прийому оплат у чат-боті використовують API платіжних провайдерів (Stripe, PayPal, LiqPay тощо). Бот формує посилання чи інвойс, відсилає його користувачу, відстежує статус транзакції через webhook або опитування API, а після успішної оплати надає доступ до послуг чи товарів. Така інтеграція спрощує продаж цифрових продуктів і сервісів безпосередньо в Telegram.

2.4 Взаємодія з мікросервісною архітектурою та брокерами повідомлень

Telegram-боти можуть ефективно інтегруватися у складні системи, побудовані на основі мікросервісної архітектури, використовуючи асинхронні підходи, такі як asyncio. Цей підхід дозволяє боту делегувати обробку складних бізнес-процесів спеціалізованим сервісам, зосередившись виключно на відправці та отриманні подій. Завдяки цьому система стає більш масштабованою, модульною та стійкою до збоїв.

Asyncio, як стандартна асинхронна бібліотека Python, забезпечує неблокуючу взаємодію між компонентами всередині одного процесу або між сервісами через асинхронні інтерфейси, такі як HTTP, WebSocket чи TCP. Він ідеально підходить для сценаріїв, де важлива швидкість реакції, легка вага та контроль над потоками виконання. Наприклад, бот може помістити запит у asyncio.Queue, а окрема задача у фоновому режимі буде асинхронно обробляти його та повертати результат – усе без потреби у зовнішньому брокері.

Apache Kafka, на відміну від asyncio, орієнтований на стрімінг даних у реальному часі та обробку великих обсягів інформації. Він використовує модель

логів, де повідомлення зберігаються протягом певного часу, що дозволяє підписникам повторно обробляти дані або відновлювати роботу після збоїв. Це корисне для систем, де потрібна аналітика в реальному часі, наприклад, відстеження активності користувачів або агрегація статистики.

2.4.1 Asyncio

Asyncio – це стандартна бібліотека Python для написання асинхронного коду на основі event loop.

Встановлення та налаштування: Не потребує окремого встановлення (входить до стандартної бібліотеки з Python 3.4+). Для налаштування – створюється цикл подій (event loop), у якому виконуються корутини (async def функції) [10].

Обмін повідомленнями: Корутини можуть обмінюватися даними через черги asyncio. Queue, які дозволяють організувати асинхронну передачу повідомлень між задачами без блокування основного потоку.

Отримання повідомлень: Задача (наприклад, бот або сервіс) асинхронно очікує повідомлення з черги за допомогою await queue.get(). Після обробки повідомлення можна вручну підтвердити його обробку (якщо реалізовано логіку ACK) або просто продовжити роботу, оскільки asyncio.Queue працює всередині однієї програми і не потребує мережевих підтверджень.

2.4.2 Kafka

Apache Kafka – розподілена стрімінг-платформа для обробки потоків подій.

Архітектура: повідомлення організовані в топіки (topics) і розділи (partitions), що забезпечує горизонтальне масштабування.

Відправка даних: бот працює як продюсер (producer), публікуючи події в потрібний топік через клієнтські бібліотеки (kafka-python або kafkajs) [11].

Обробка потоків: сервіси-споживачі (consumers) читають повідомлення з початку або з останньої позиції, групуються в consumer groups для паралельної обробки та гарантують принаймні одноразову доставку.

Система побудована на основі мікросервісної архітектури з використанням Apache Kafka для асинхронного обміну повідомленнями.

Основні компоненти:

- телеграм-бот (Producer) – генерує події користувачів;
- apache Kafka – брокер повідомлень;
- сервіс обробки даних (Consumer) – зберігає дані в БД;
- сервіс аналітики (Consumer) – генерує звіти.

3 РОЗРОБКА ТЕЛЕГРАМ-БОТА ТА ЙОГО ІНТЕГРАЦІЯ З ЗОВНІШНІМИ СЕРВІСАМИ

3.1 Постановка задачі та вимоги до системи

У цьому підрозділі формалізуємо основні цілі та вимоги до нашого Telegram-бота. Головна задача – створити бот, який надає користувачам актуальну інформацію з різних зовнішніх сервісів (погода, дані з Google Sheets, генерація текстів через deepseek) та зберігає історію взаємодій у базі даних, з якими після можна щось зробити. Він має реагувати на стандартні команди (/start, /check_db і т.д.) і забезпечувати простий інтерфейс з командами та кнопками.

3.2 Вибір технологій та інструментів

Для розробки Telegram-бота було обрано стек, який забезпечує швидкий старт, простоту підтримки та гнучкість для подальшого розвитку:

Мова програмування та фреймворк

В якості основи я використав Python разом із бібліотекою aiogram. Цей фреймворк побудований на asyncio та дозволяє легко реалізувати як long polling, так і webhook-режим. Завдяки асинхронній моделі обробка одночасних запитів відбувається без блокувань, що особливо важливо для інтеграції з кількома зовнішніми сервісами.

HTTP-клієнт для REST API

Для взаємодії із зовнішніми веб-сервісами (OpenWeather, Google Sheets, OpenAI) оберемо бібліотеку aiohttp. Вона підтримує асинхронні HTTP-запити та дозволяє коректно обробляти таймаути й помилки мережі.

Сховище даних

Для зберігання інформації про користувачів та історії запитів підійде MySQL. Підключенная буде реалізоване через aiomysql. Параметри з’єднання (хост, порт, ім’я користувача, пароль і назва бази) задаються через змінні оточення, щоб не зберігати секретні дані в коді.

Керування конфігурацією та секретами

Усі ключі доступу (Telegram Bot API Token, ключі до сторонніх сервісів) зберігаються у середовищі виконання як змінні оточення. Для локальної розробки застосовується файл .env та бібліотеку python-dotenv.

Середовище програмування

Середовищем розробки (IDE) було обрано PyCharm та VScode, через зручне редагування файлів пайтону та запуск контейнера здійснюється в Docker Desktop. Я використовував також Docker Desktop для реалізації контейнеру з базою даних, яка генерується через docker-compose.yml.

3.3 Розробка серверної частини бота

Розділив код на модулі (Рисунок 3.1):

- bot – Головна папка проекту з конфігурацією та docker-файлами;
- app.py – точка входу, налаштовка бота (long polling або webhook), підключення до бази та зовнішніх API;
- handlers/ – функції-обробники команд та повідомлень (start, check_db, кастомні меню, callback’и);
- services/ – клієнти для роботи з REST API (погода, Google Sheets, OpenAI) та DAO-шари для читання/запису в базу;
- secrets/ – API таблиці Google Sheets;
- .venv/ – виртуальне середовище для пайтона, створене через PyCharm.

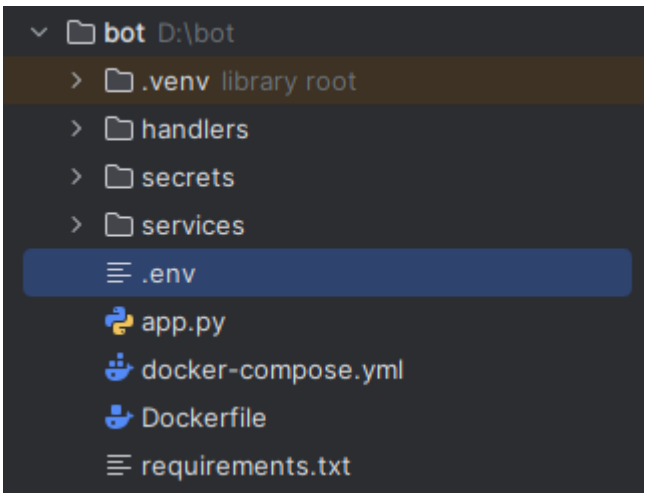


Рисунок 3.1 – Структура файлів проекту Telegram-бота

Обробка оновлень здійснювалася через обраний механізм для бота (polling або webhook). Після прийому кожного Update обробник перевіряє тип (текст, команда, натискання кнопки) і викликає відповідну функцію. Всі помилки всередині обробників потрібно ловити та логувати, щоб сервіс не впав при некоректних даних, що я намагався робити.

До обробки оновлень належать наступні файли.

Обробник, наприклад, команди weather, викликає сервісний клас OpenWeatherClient, який через aiohttp робить асинхронний HTTP-запит і повертає готовий текст для відправки. Після цього результат записується в MySQL через DAO (insert історії запиту) і відправляється користувачу (Рисунок 3.2 – Потік даних між ботом, Kafka та зовнішніми API).

Всі ключі та URL токенів зберігаються в змінних оточення. Для локального зберігання в мене використовується .env із python-dotenv.

3.4 Інтеграція з базою даних та зовнішніми API

Система Telegram-бота реалізує комплексну взаємодію з різними зовнішніми сервісами та базою даних, забезпечуючи ефективне зберігання, обробку та передачу інформації. Основу архітектури складають асинхронні

технології, що дозволяють уникнути блокування потоків при виконанні запитів, що особливо важливо для високонавантажених систем.

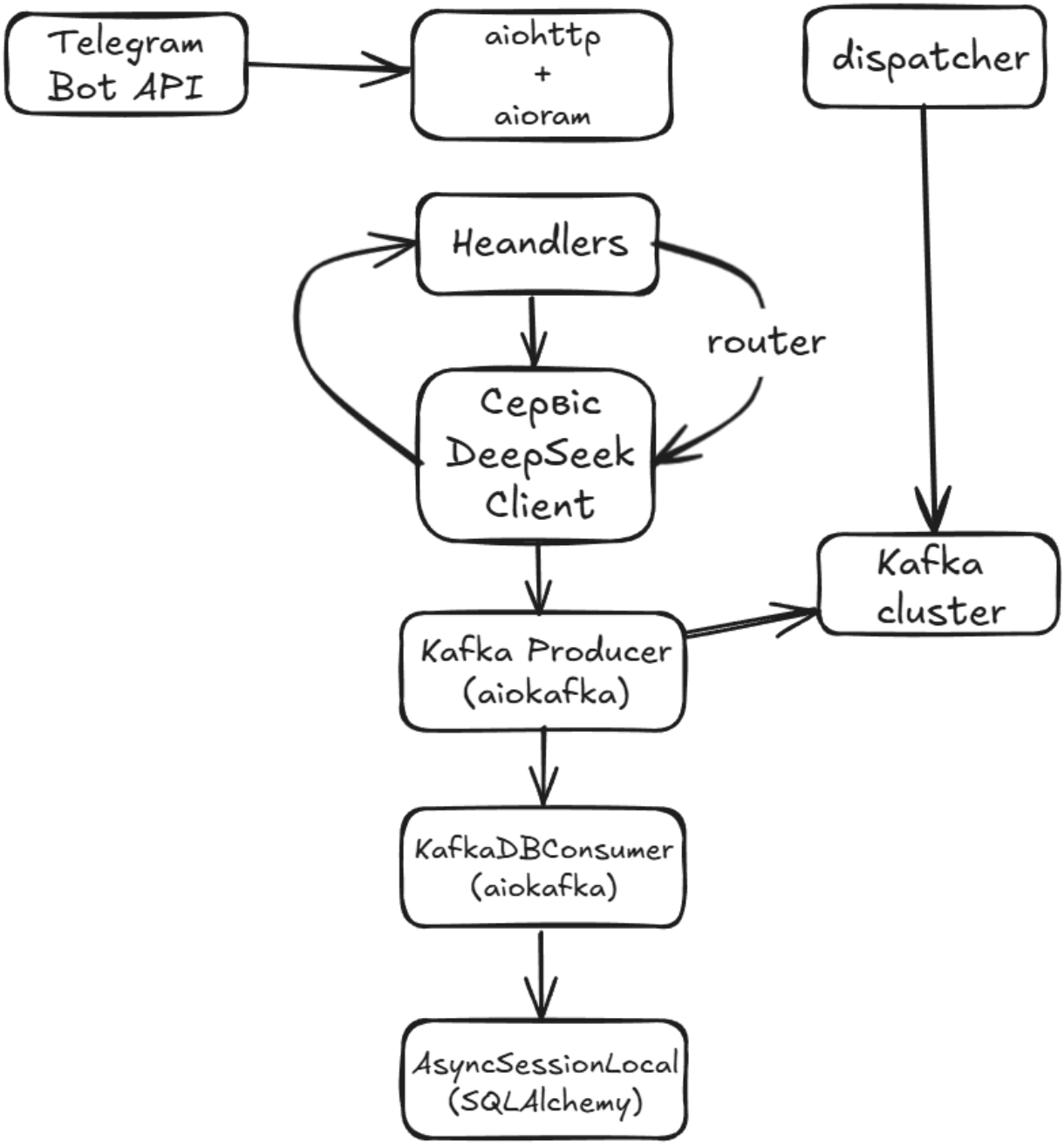


Рисунок 3.2 – Потік даних між ботом, Kafka та зовнішніми API

Робота з базою даних організована через SQLAlchemy ORM з використанням асинхронного драйвера для MySQL. Моделі даних, такі як історія дій користувачів та обрані відео, зберігаються в структурованих таблицях, що

забезпечує швидкий доступ та надійність. Автоматична генерація схем та використання змінних середовища для конфігурації підключення спрощують розгортання системи в різних умовах.

Інтеграція з YouTube API [12] дозволяє здійснювати пошук відео за запитом користувача. Асинхронні HTTP-запити через бібліотеку aiohttp забезпечують отримання даних у форматі JSON, які потім трансформуються у зручну структуру з назвою, посиланням та каналом. Обмеження результатів пошуку до п’яти позицій оптимізує вибірку та відображення інформації.

Для роботи з погодними даними використовується OpenWeatherMap API. Система отримує поточні метеорологічні показники для вказаного міста, включаючи температуру в градусах Цельсія та короткий опис умов. Асинхронність запитів забезпечує оперативну реакцію бота навіть при великій кількості одночасних звернень.

Взаємодія з Google Sheets реалізована через сервісний акаунт Google, що дозволяє читати дані з аркушів за їхніми назвами. Щоб реалізувати API, потрібно створити проєкт у Google Cloud Console, увімкнути Sheets API, створити облікові дані (API key або OAuth 2.0), а потім використовувати бібліотеки для взаємодії з таблицями. Сама таблиця буде знаходитись за посиланням [13]. Нижче наведена таблиця з незміненими даними, які зараз на сайті (Таблиця 3.1).

Таблиця 3.1 – Дані в Google таблиці

	A	B	C
1	Назва витрат	Витрати	Частина
2	Харчування	100	=ОКРУГЛ(B2/$B$8;3)
3	Транспорт	200	=ОКРУГЛ(B3/$B$8;3)
4	Розваги	300	=ОКРУГЛ(B4/$B$8;3)
5	Комунальні послуги	400	=ОКРУГЛ(B5/$B$8;3)
6	Інші послуги	500	=ОКРУГЛ(B6/$B$8;3)
8	Загальна сума	=СУММ(B2:B6)	

Обмін подіями через Kafka відіграє ключову роль у архітектурі. Події, такі як пошук відео або збереження даних, надсилаються у відповідні топіки, після чого споживачі асинхронно зберігають їх у базі даних. Це забезпечує розподілене обслуговування запитів, підвищує відмовостійкість системи та дозволяє масштабувати її для обробки великих обсягів даних.

Інтеграція з AI-моделлю Deepseek через OpenRouter API дає змогу генерувати текстові відповіді на запити користувачів. Система автоматично визначає мову вхідного тексту та забезпечує переклад при необхідності, що розширює аудиторію бота.

Загалом є близько 10000 безкоштовних запитів, які може виконувати бот (Рисунок 3.3 – Запити до бота в онлайн часі через GoogleAPI).

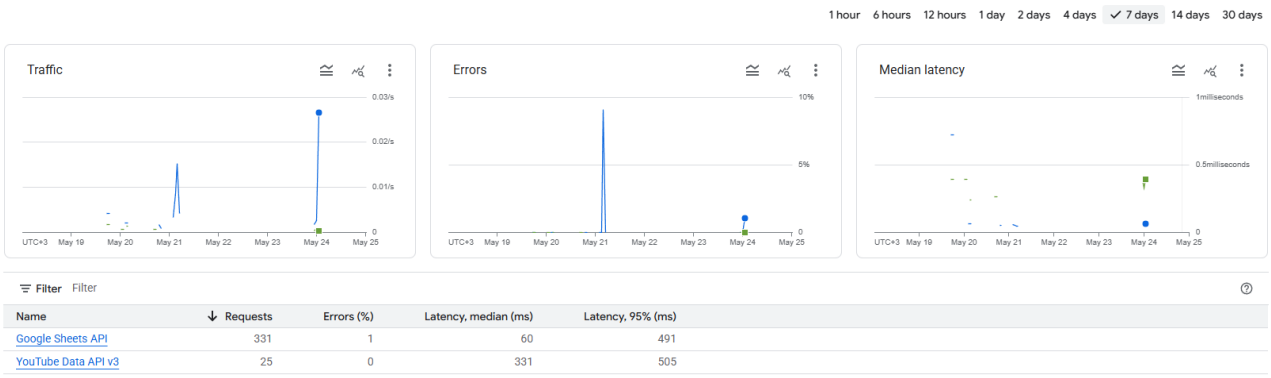


Рисунок 3.3 – Запити до бота в онлайн часі через GoogleAPI

3.5 Тестування та оптимізація роботи бота

Під час етапу тестування бот був перевірений у реальних умовах різними користувачами, що дозволило виявити критичні та другорядні помилки. Серед них – обмеження Telegram на довжину повідомлень (до 4096 символів), яке інколи призводило до обрізання відповідей, особливо під час роботи з AI-генерацією текстів. Також були випадки, коли некоректна мова запитів або спеціальні символи спричиняли помилки обробки, а деякі команди взагалі виводили систему з ладу через відсутність належної валідації вхідних даних. Для

усунення цих проблем було впроваджено механізми обмеження довжини текстів, автоматичну корекцію мови запитів за допомогою бібліотек визначення мови, а також додано інструменти відловлювання винятків для запобігання непередбачених падінь.

Бот виводить п’ять основних кнопок у головному меню, після команди /start (див. Рисунок 3.4**Error! Reference source not found.**, кожна з яких відповідає за певний функціонал.

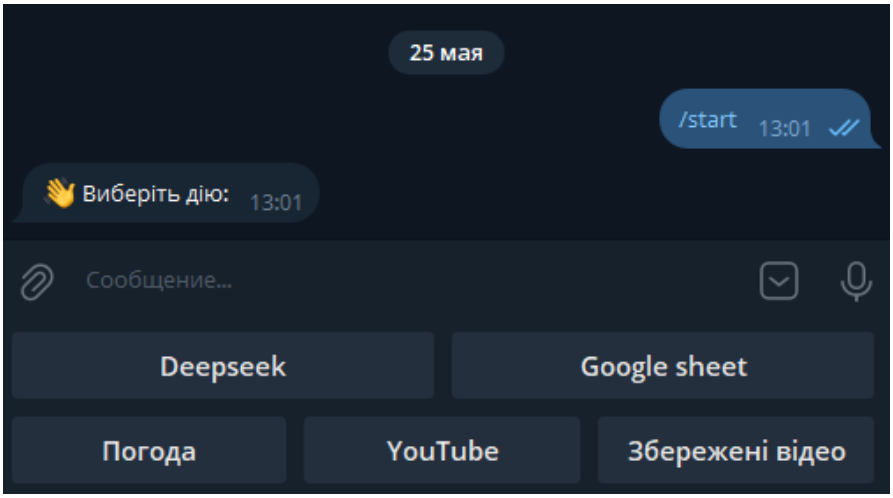


Рисунок 3.4 – Бот після запуску команди /start

deepseek.py: Відповідає за взаємодію з нейромережею Deepseek. Коли користувач обирає кнопку "Deepseek", бот запитує текстовий запит, передає його через API та повертає відповідь (Рисунок 3.5). Тут обробляється логіка запитів та відправка результатів.

common.py: Містить загальні елементи: стартове меню з кнопками, функцію перегляду збережених відео ("Збережені відео") та інструменти для роботи з базою даних (Рисунок 3.6). Також тут реалізовано видалення відео через інлайн-кнопки та перевірку стану БД (/check_db).

youtube.py: Забезпечує пошук відео на YouTube (кнопка "YouTube"). Користувач вводить запит, бот показує результати з можливістю зберегти відео під власною назвою (Рисунок 3.7). Тут обробляються інлайн-кнопки, збереження даних та комунікація з YouTube API.

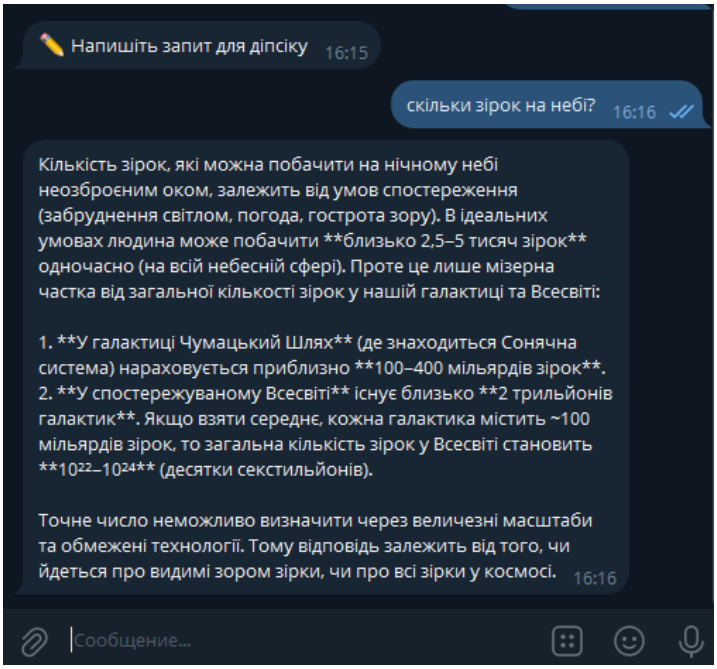


Рисунок 3.5 – Запити до deepseek

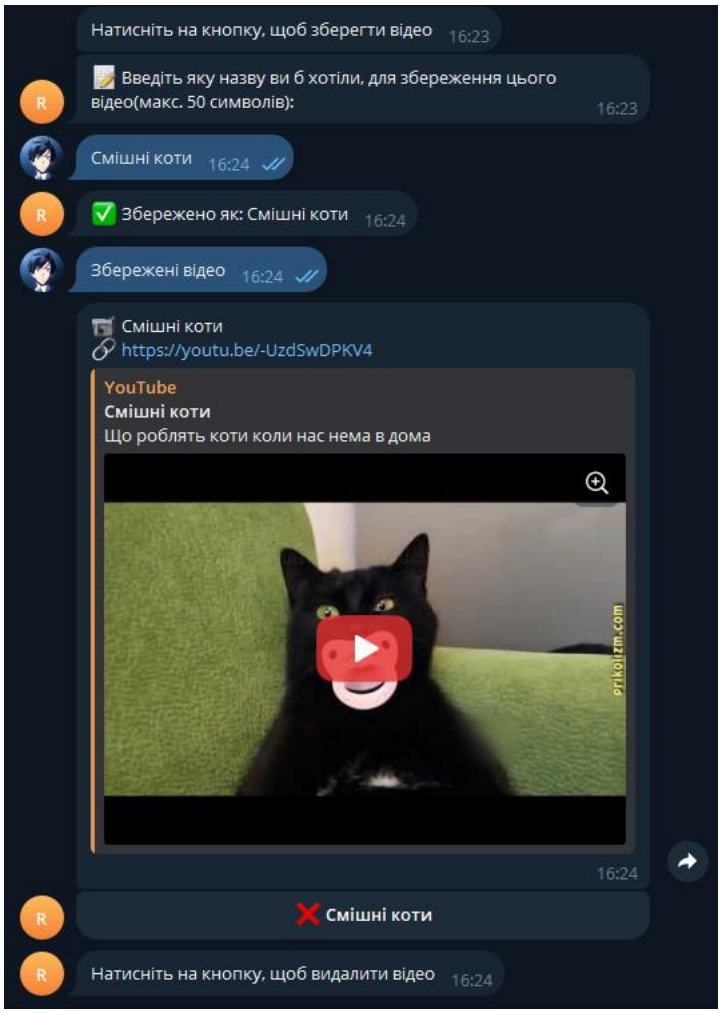


Рисунок 3.6 – Перегляд збережених відео

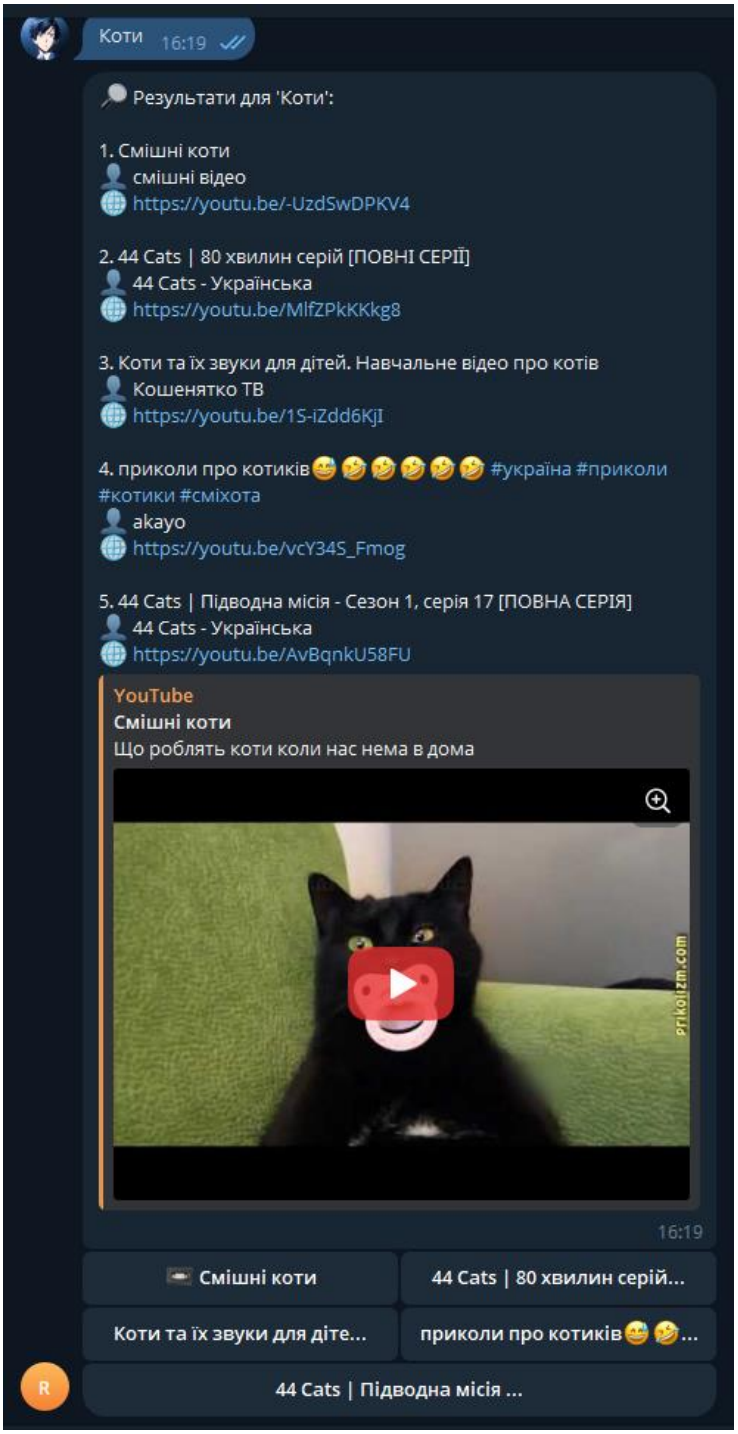


Рисунок 3.7 – Запит до YouTube

sheets.py: Відповідає за роботу з Google Таблицями (кнопка "Google sheet"). Бот відображає дані Таблиця 3.1, дозволяє редагувати комірки від B2 до B6 для визначення витрат та після цього виводить результат всієї таблиці (Рисунок 3.8). Тут реалізовано читання та оновлення даних, а також інтеграцію з API Google Sheets.

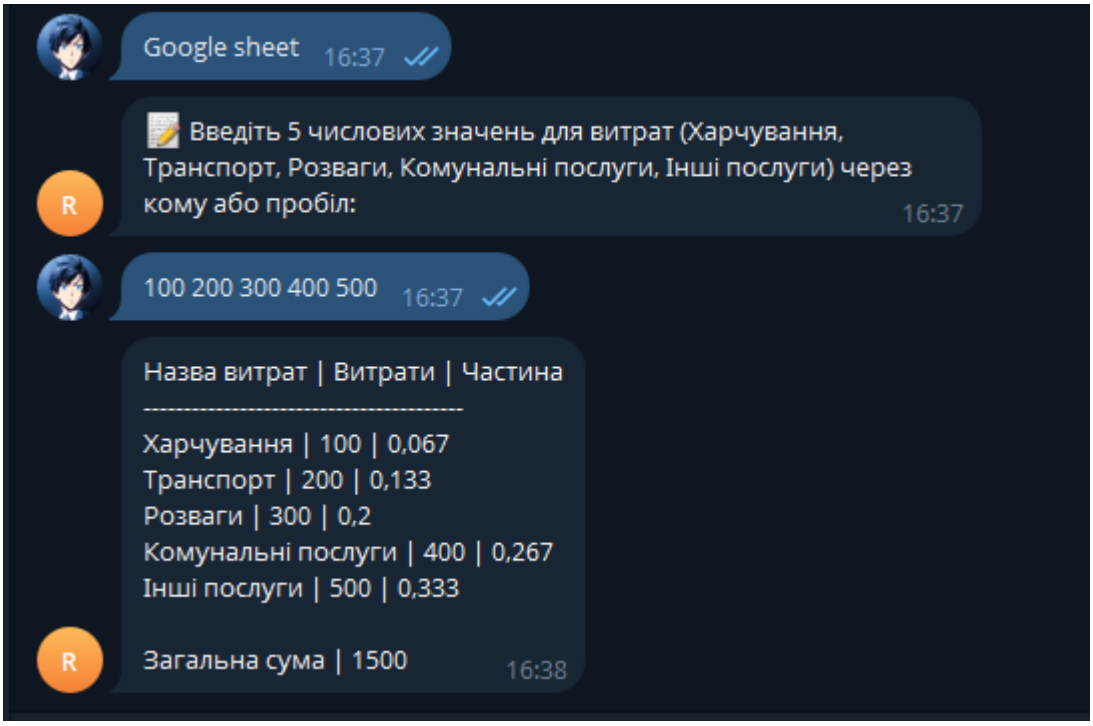


Рисунок 3.8 – Запит до Google Sheets

weather.py: Обробляє запити про погоду (кнопка "Погода"). Користувач вводить назву міста, бот отримує дані з погодного API та відправляє результат (Рисунок 3.9). Тут також відбувається обробка помилок та форматування відповіді.

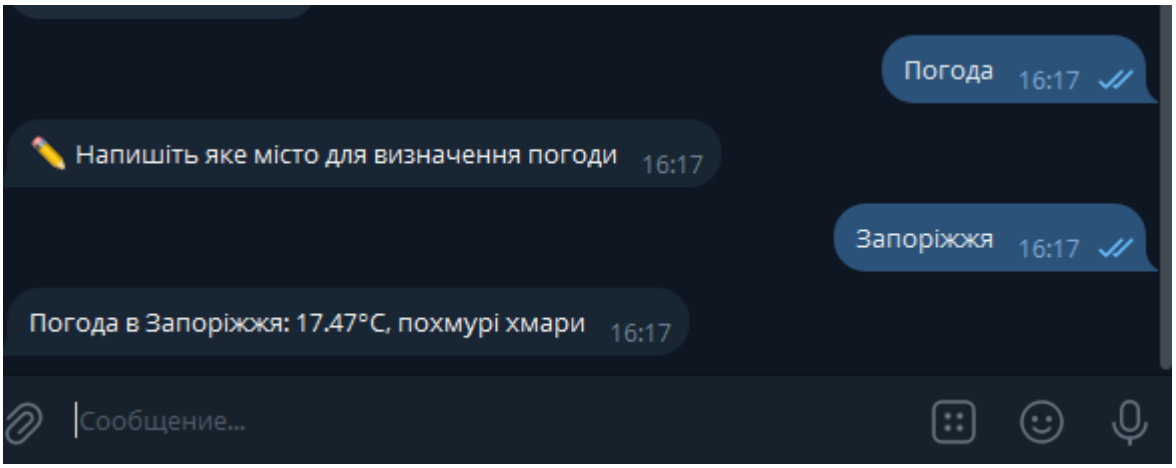


Рисунок 3.9 – Запит до openweathermap

Після серії оптимізацій бот демонструє стабільну роботу на локальному сервері без перебоїв навіть під час пікових навантажень. Ключовим етапом оптимізації стало рефакторинг коду: зайві фрагменти було видалено, а логіку спрощено за рахунок використання вбудованих функцій Python та спеціалізованих бібліотек, таких як aiohttp для асинхронних HTTP-запитів і SQLAlchemy для ефективної роботи з базою даних.

ВИСНОВКИ

У кваліфікаційній роботі було проведено комплексне дослідження технологій інтеграції Telegram-ботів із зовнішніми сервісами, що дозволило розробити функціональний прототип бота з широким спектром можливостей. Головною метою роботи було продемонструвати ефективність сучасних інструментів для автоматизації обміну даними, і ця мета була успішно досягнута. Використання асинхронного фреймворку aiogram та бібліотеки aiohttp забезпечило швидку обробку запитів, а інтеграція з Google Sheets, OpenWeather та Deepseek показала, як боти можуть взаємодіяти з різноманітними платформами для надання користувачам актуальної інформації.

Важливим аспектом роботи став аналіз механізмів Long Polling та Webhook, який підкреслив переваги асинхронного підходу для забезпечення стабільної роботи бота. Розгортання бази даних MySQL за допомогою Docker дозволило створити надійне середовище для зберігання історії запитів і користувацьких налаштувань.

Безпека залишалася пріоритетом на всіх етапах розробки. Збереження конфіденційних даних, таких як API-ключі та токени, у змінних оточення мінімізувало ризики витоку інформації. Цей підхід, поєднаний із ретельним тестуванням, забезпечив стабільність і надійність системи.

Перспективи розвитку проекту включають розширення функціоналу шляхом інтеграції з платіжними системами, що відкриває можливості для монетизації послуг бота. Також варто розглянути додавання аналітичних інструментів для відстеження користувацької активності та вдосконалення локалізації для підтримки додаткових мов.

Практична цінність роботи полягає в тому, що розроблений бот може використовуватися як інструмент автоматизації в бізнес-процесах, наприклад, для формування звітів у Google Sheets або надання метеорологічних даних. Крім того, він слугує навчальним прикладом, який ілюструє принципи інтеграції

сучасних технологій. Досвід, отриманий під час розробки, підкреслює важливість вибору правильних інструментів для досягнення ефективності, безпеки та адаптивності інформаційних систем.

Telegram-боти, поєднані з зовнішніми сервісами, є потужним інструментом для оптимізації комунікації та обробки даних.

ПЕРЕЛІК ПОСИЛАНЬ

1. Моджрик Н. Building Telegram Bots: Develop Bots in 12 Programming Languages using the Telegram Bot API. Springer. URL : https://link.springer.com/book/10.1007/978-1-4842-4197-4 (дата звернення : 27.05.2025).

2. Пилипчук В. В. Webhooks and Long Polling methods for hosting a Telegram bot. *Науковий вісник Луцького НТУ*. URL : https://konfkafptl.lntu.edu.ua/index.php/cit/article/view/397 (дата звернення : 27.05.2025).

3. Webhook.site Frequently Asked Questions. Webhook. URL : https://docs.webhook.site/ (дата звернення : 14.05.2025).

4. Стоянович С. Serverless Applications with Node.js. *Manning Publications*. URL : https://www.manning.com/books/serverless-applications-with-node-js/ (дата звернення : 27.05.2025).

5. Sahrin M., Zainal A., Al-Qarafi A. Investigating User Privacy and Security on Telegram Messenger : A Wireshark Analysis. ResearchGate. URL : https://www.researchgate.net/publication/382056690_Investigating_User_Privacy_and_Security_on_Telegram_Messenger_A_Wireshark_Analysis (дата звернення : 27.05.2025).

6. Шварц Б., Ткаченко В., Зайцев П. *MySQL по максимуму*. Coollib. URL : https://coollib.cc/b/638682-silviya-botros-mysql-po-maksimumu/read/ (дата звернення : 28.05.2025).

7. Google Cloud Documentation. Google Cloud. URL : https://cloud.google.com/docs (дата звернення : 11.05.2025).

8. Current weather data. WeatherAPI. URL : https://openweathermap.org/current (дата звернення : 02.05.2024).

9. API Reference. OpenRouter. URL : https://openrouter.ai/docs/api-reference/overview (дата звернення: 15.05.2025).

10. Asyncio – Asynchronous I/O. Asyncio. URL : https://docs.python.org/3/library/asyncio.html (дата звернення : 01.05.2025).

11. Kafka-python. Kafka-python. URL : https://kafka-python.readthedocs.io/en/master/index.html (дата звернення : 10.05.2025).

12. YouTube Data API Overview. YouTube API. URL : https://developers.google.com/youtube/v3/getting-started (дата звернення : 04.05.2024).

13. Google Dogs. Google Sheets. URL : https://docs.google.com/spreadsheets/d/11ZtCFG3_q6ZvSd32cIcZBBjMvHDuv64ux9lrXorqV_c/edit?pli=1&gid=0#gid=0 (дата звернення : 28.05.2025)

ДОДАТОК А

Лістинг файлу «app.py»

```
import os
import asyncio
from aiogram import Bot, Dispatcher
from aiogram.webhook.aiohttp_server import SimpleRequestHandler, setup_application
from aiohttp import web

from services.db import init_db
from services.kafka_consumer_db import KafkaDBConsumer
from handlers.common import router as common_router
from handlers.youtube import router as youtube_router
from handlers.deepseek import router as deepseek_router
from handlers.weather import router as weather_router
from handlers.sheets import router as sheets_router
from services.kafka_producer import KafkaProducer

from pyngrok import ngrok

from dotenv import load_dotenv

async def main():
    load_dotenv()
    bot = Bot(token=os.getenv('TG_BOT_TOKEN'))

    # 1) Создаём экземпляр
    my_producer = KafkaProducer(bootstrap_servers=os.getenv("DB_KAFKA_BOOTSTRAP",
"kafka:9092"))
    # 2) Запускаем продюсер
    await my_producer.start()

    dp = Dispatcher(kafka_producer=my_producer)

    # Реєстрація рутерів
    dp.include_router(youtube_router)
    dp.include_router(common_router)
    dp.include_router(weather_router)
    dp.include_router(sheets_router)
    dp.include_router(deepseek_router)

    # Ініціалізація БД
    await init_db()

    ngrok.set_auth_token(os.getenv("NGROK_AUTHTOKEN"))
    public_url = ngrok.connect(8443, 'http').public_url
```

```
    webhook_url = f"{public_url}/webhook"

    # Встановлення вебхука
    await bot.set_webhook(webhook_url)

    # Запуск сервера
    app = web.Application()
    SimpleRequestHandler(dispatcher=dp, bot=bot).register(app, path="/webhook")
    setup_application(app, dp, bot=bot)

    runner = web.AppRunner(app)
    await runner.setup()
    site = web.TCPSite(runner, host='0.0.0.0', port=8443)
    await site.start()

    # Инициализация консьюмера для БД
    db_consumer = KafkaDBConsumer(
        bootstrap_servers="localhost:9092",
        topic="bot-events"
    )
    await db_consumer.start()

    # Запуск консьюмера в фоне
    asyncio.create_task(db_consumer.consume_and_save())
    # Очікування завершення
    await asyncio.Event().wait()

if __name__ == '__main__':
    asyncio.run(main())
```

ДОДАТОК Б

Лістинг файлу Docker-compose.yml

```
services:
  bot:
    build: .
    env_file: .env
    environment:
      – KAFKA_BOOTSTRAP_SERVERS=kafka:9092
    networks:
      – app-network
    depends_on:
      – kafka
      – mysql

  mysql:
    image: mysql:8.0
    networks:
      – app-network
    environment:
      MYSQL_ROOT_PASSWORD: ${DB_PASS}
      MYSQL_DATABASE: ${DB_NAME}
      MYSQL_USER: ${DB_USER}
      MYSQL_PASSWORD: ${DB_PASS}
    ports:
      – "3306:3306"
    volumes:
      – mysql_data:/var/lib/mysql

  kafka:
    image: bitnami/kafka:4.0.0
    networks:
      – app-network
    environment:
      – KAFKA_ENABLE_KRAFT=yes
      – KAFKA_CFG_PROCESS_ROLES=controller,broker
      – KAFKA_CFG_NODE_ID=1
      – KAFKA_INTER_BROKER_LISTENER_NAME:PLAINTEXT
      – KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=1@kafka:9093  # Добавлено
      – KAFKA_CFG_CONTROLLER_LISTENER_NAMES=CONTROLLER
      – KAFKA_CFG_LISTENERS=PLAINTEXT://0.0.0.0:9092,CONTROLLER://0.0.0.0:9093
      –
KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=CONTROLLER:PLAINTEXT,PLA
INTEXT:PLAINTEXT
      – KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka:9092
      – KAFKA_CFG_INTER_BROKER_LISTENER_NAME=PLAINTEXT
```

```
      – KAFKA_KRAFT_CLUSTER_ID=my-kafka-cluster
    ports:
      – "9092:9092"
    volumes:
      – kafka_data:/bitnami

networks:
  app-network:
    driver: bridge

volumes:
  mysql_data:
  kafka_data:
```

ДОДАТОК В

Лістинг файлу Dockerfile

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
CMD ["python", "app.py"]
```

ДОДАТОК Г

Лістинг файлу requirements.txt

```
typing-extensions==4.13.2
aiogram==3.20.0
SQLAlchemy==2.0.15
asyncmy==0.2.10
aiohttp==3.11.18
gspread==6.1.4
oauth2client==4.1.3
openai==1.78.1
python-dotenv==1.1.0
pyngrok==7.2.8
cryptography==42.0.0
langdetect==1.0.9
googletrans==4.0.2
aiokafka==0.12.0
google-api-python-client==2.169.0
```

ДОДАТОК Д

Лістинг файлів директорії handlers

common.py

```
from aiogram import Router, F
from aiogram.types import Message, CallbackQuery, InlineKeyboardMarkup,
InlineKeyboardButton, ReplyKeyboardMarkup, KeyboardButton
from aiogram.filters import CommandStart
from sqlalchemy import select, delete
from services.db import AsyncSessionLocal, History, LoveVideo

router = Router()

menu_keyboard = ReplyKeyboardMarkup(
    keyboard=[
        [KeyboardButton(text="Deepseek"), KeyboardButton(text="Google sheet")],
        [KeyboardButton(text="Погода"), KeyboardButton(text="YouTube"),
KeyboardButton(text="Збережені відео")]
    ],
    resize_keyboard=True,
    one_time_keyboard=False
)

@router.message(CommandStart())
async def cmd_start(message: Message):
    await message.answer(
        "👋 Виберіть дію:",
        reply_markup=menu_keyboard
    )


@router.message(F.text == "/check_db")
async def check_db(message: Message):
    async with AsyncSessionLocal() as session:
        result = await session.execute(select(History))
        records = result.scalars().all()

    if not records:
        await message.answer("База даних порожня.")
        return

    response = "📝 Вміст бази даних:\n\n"
    for record in records:
        response += (
            f"ID: {record.id}\n"
```

```
            f"User: {record.user_id}\n"
            f"Command: {record.command}\n"
            f"Response: {record.response[:50]}\n"
            f"MessageID: {record.message_id}\n"
            f"Date: {record.created_at}\n\n"
        )

    await message.answer(response)

@router.message(F.text == "Збережені відео")
async def show_saved_videos(message: Message):
    async with AsyncSessionLocal() as session:
        result = await session.execute(select(LoveVideo).where(LoveVideo.user_id ==
message.from_user.id))
        records = result.scalars().all()

    if not records:
        await message.answer("Збереженних відео немає")
        return

    items = []
    for record in records:
        items.append({
            "text": (
                f"📹 {record.name_video}\n"
                f"🔗 https://youtu.be/{record.response[:50]}"
            ),
            "button": InlineKeyboardButton(
                text=f"❌ {record.name_video}",
                callback_data=f"delete_video:{record.id}"
            )
        })

    parts = []
    current_text = ""
    current_buttons = []
    for item in items:
        entry = item["text"] + "\n\n"
        if len(current_text) + len(entry) > 4000:
            parts.append((current_text.strip(), current_buttons.copy()))
            current_text = entry
            current_buttons = [item["button"]]
        else:
            current_text += entry
            current_buttons.append(item["button"])

    if current_text:
        parts.append((current_text.strip(), current_buttons.copy()))

    for text, btns in parts:
        inline_rows = []
        row = []
```

```
        for i, btn in enumerate(btns, 1):
            row.append(btn)
            if i % 2 == 0:
                inline_rows.append(row)
                row = []
        if row:
            inline_rows.append(row)

        keyboard = InlineKeyboardMarkup(inline_keyboard=inline_rows)
        await message.answer(text, reply_markup=keyboard)

    await message.answer("Натисніть на кнопку, щоб видалити відео")

@router.callback_query(F.data.startswith("delete_video:"))
async def delete_video_callback(callback: CallbackQuery):
    video_id = int(callback.data.split(":")[1])
    async with AsyncSessionLocal() as session:
        await session.execute(delete(LoveVideo).where(LoveVideo.id == video_id,
LoveVideo.user_id == callback.from_user.id))
        await session.commit()

    await callback.answer("✅ Відео видалено.")
    await callback.message.delete()
```

deepseek.py

```
from aiogram import Router, F
from aiogram.fsm.context import FSMContext
from aiogram.types import Message
from aiogram.filters import StateFilter
from services.deepseek_client import ask_deepseek
import json

router = Router()

@router.message(F.text == "Deepseek")
async def handle_ask_start(message: Message, state: FSMContext):
    await message.answer("➖· Напишіть запит для діпсіку")
    await state.set_state("waiting_for_prompt2")

@router.message(F.text, StateFilter("waiting_for_prompt2"))
async def handle_ask_prompt(message: Message, state: FSMContext, kafka_producer):
    prompt = message.text

    await message.bot.send_chat_action(message.chat.id, 'typing')

    response = await ask_deepseek(prompt)

    if not response.strip():
        response = "❌ Пуста відповідь від API. Перевірте запит."

    msg = await message.answer(response[:4000])
```

```
    event = {
        "user_id": message.from_user.id,
        "command": "Deepseek",
        "message_id": msg.message_id,
        "response": prompt[:50]
    }

    await kafka_producer.send_message("bot-events", json.dumps(event))

    await state.clear()
```

sheets.py

```
from aiogram import Router, F
from aiogram.types import Message
from services.sheets_client import get_sheet_data, send_sheet_data
import json
from aiogram.fsm.context import FSMContext
from aiogram.fsm.state import State, StatesGroup

router = Router()


class SheetStates(StatesGroup):
    waiting_for_expenses = State()


def split_text(text, max_length=4000):
    parts = []
    while len(text) > max_length:
        split_index = text.rfind('\n', 0, max_length)
        if split_index == -1:
            split_index = max_length
        parts.append(text[:split_index])
        text = text[split_index:].lstrip('\n')
    parts.append(text)
    return parts


@router.message(F.text == 'Google sheet')
async def sheet_handler(message: Message, state: FSMContext, kafka_producer):
    await message.answer(
        "📝 Введіть 5 числових значень для витрат (Харчування, Транспорт, Розваги,
Комунальні послуги, Інші послуги) через кому або пробіл:")
    await state.set_state(SheetStates.waiting_for_expenses)

    event = {
        "user_id": message.from_user.id,
        "command": "sheets_start",
        "message_id": message.message_id,
        "response": "waiting_for_expenses"
```

```
    }
    await kafka_producer.send_message("bot-events", json.dumps(event))


@router.message(SheetStates.waiting_for_expenses, F.text)
async def process_expenses_values(message: Message, state: FSMContext, kafka_producer):
    try:
        parts = message.text.replace(',', ' ').split()
        if len(parts) != 5:
            await message.answer("❗ Потрібно ввести рівно 5 числових значень!")
            return

        try:
            expenses = [float(part.strip()) for part in parts]
        except ValueError:
            await message.answer("❗ Усі значення повинні бути числами!")
            return

        worksheet = send_sheet_data('Sheet1')
        data = [[value] for value in expenses]
        worksheet.update('B2:B6', data)

        rows = get_sheet_data('Sheet1')

        table_lines = []

        headers = [rows[0][0] if len(rows[0]) > 0 else "",
                   rows[0][1] if len(rows[0]) > 1 else "",
                   rows[0][2] if len(rows[0]) > 2 else ""]
        table_lines.append(f"{headers[0]} | {headers[1]} | {headers[2]}")
        table_lines.append("-" * 40)

        for i in range(1, 6):
            row = rows[i] if i < len(rows) else []
            name = row[0] if len(row) > 0 else ""
            value = row[1] if len(row) > 1 else ""
            share = row[2] if len(row) > 2 else ""
            table_lines.append(f"{name} | {value} | {share}")

        table_lines.append("")

        if len(rows) > 7:
            total_row = rows[7]
            total_name = total_row[0] if len(total_row) > 0 else ""
            total_value = total_row[1] if len(total_row) > 1 else ""
            table_lines.append(f"{total_name} | {total_value}")

        full_text = "\n".join(table_lines)
        text_parts = split_text(full_text)

        for part in text_parts:
            await message.answer(part)
```

```
        event = {
            "user_id": message.from_user.id,
            "command": "sheets_updated",
            "message_id": message.message_id,
            "response": "success"
        }
        await kafka_producer.send_message("bot-events", json.dumps(event))

    except Exception as e:
        error_msg = f"❌ Помилка: {str(e)}"
        await message.answer(error_msg)

        event = {
            "user_id": message.from_user.id,
            "command": "sheets_error",
            "message_id": message.message_id,
            "response": error_msg
        }
        await kafka_producer.send_message("bot-events", json.dumps(event))
    finally:
        await state.clear()
```

weather.py

```
# handlers/weather.py
from aiogram import Router, F
from aiogram.types import Message
from services.weather_client import get_weather
from aiogram.fsm.context import FSMContext
from aiogram.filters import StateFilter
import json
from googletrans import Translator

router = Router()

async def translate_text(text):
    async with Translator() as translator:
        result = await translator.translate(text, src='en', dest='uk')
    return result

async def text(text):
    data = await get_weather(text)
    description = await translate_text(data['description'])
    response_text = f"{description.text}"
    return response_text

@router.message(F.text == 'Погода')
async def weather_handler(message: Message, state: FSMContext):
    await message.answer("➖· Напишіть яке місто для визначення погоди")
    await state.set_state("waiting_for_prompt")
```

```
@router.message(F.text, StateFilter("waiting_for_prompt"))
async def handle_ask_prompt(message: Message, state: FSMContext, kafka_producer):

    prompt = message.text
    await message.bot.send_chat_action(message.chat.id, 'typing')
    try:
        data = await get_weather(prompt)
        try:
            translated_prompt = await text(prompt)
        except Exception:
            translated_prompt = data['description']

        response_text = f"Погода в {prompt}: {data['temp']}°C, {translated_prompt}"

        if not response_text.strip():
            response_text = "❌ Пуста відповідь від API. Перевірте запит."

        msg = await message.answer(response_text)

        event = {
            "user_id": message.from_user.id,
            "command": "Weather",
            "message_id": msg.message_id,
            "response": response_text
        }

        await kafka_producer.send_message("bot-events", json.dumps(event))

    except KeyError as e:
        response_text = "❌ Пуста відповідь від API. Перевірте запит."
        await message.answer(response_text)

    await state.clear()
```

youtube.py

```
from aiogram import Router, F
from aiogram.types import Message, CallbackQuery
from aiogram.fsm.context import FSMContext
from aiogram.filters import StateFilter
from aiogram.utils.keyboard import InlineKeyboardBuilder
from aiogram.fsm.state import State, StatesGroup
import json
from urllib.parse import urlparse
from services.youtube_client import search_youtube_videos

router = Router()

class VideoStates(StatesGroup):
    waiting_for_video_name = State()
```

```
@router.message(F.text == 'YouTube')
async def youtube_handler(message: Message, state: FSMContext):
    await message.answer("🔍 Введіть пошуковий запит для YouTube:")
    await state.set_state("waiting_youtube_query")


@router.message(F.text, StateFilter("waiting_youtube_query"))
async def handle_youtube_search(message: Message, state: FSMContext, kafka_producer):
    search_query = message.text

    try:
        search_data = await search_youtube_videos(search_query)

        if not search_data['results']:
            await message.answer("🔍 *Нічого не знайдено... Спробуйте інший запит!*",
parse_mode="Markdown")
            return

        builder = InlineKeyboardBuilder()
        for video in search_data['results']:
            video_url = video['url']
            video_id = urlparse(video_url).path.split('/')[-1]

            button_text = f"{video['title'][:25]}..." if len(video['title']) > 28 else f"📼 {video['title']}"

            builder.button(
                text=button_text,
                callback_data=f"save_{video_id}"
            )

        builder.adjust(2, repeat=True)

        response_text = f"🔎 Результати для '{search_data['query']}':\n\n"
        for idx, video in enumerate(search_data['results'], 1):
            response_text += f"{idx}. {video['title']}\n👤 {video['channel']}\n🌐 {video['url']}\n\n"

        msg = await message.answer(response_text, reply_markup=builder.as_markup())
        await message.answer("Нажмите на кнопку чтобы сохранить видео")

        event = {
            "user_id": message.from_user.id,
            "command": "Youtube",
            "message_id": msg.message_id,
            "response": video_id
        }

        await kafka_producer.send_message("bot-events", json.dumps(event))

    except Exception as e:
        await message.answer(f"⚠ Помилка: {str(e)}")
```

```
    await state.clear()


@router.callback_query(F.data.startswith('save_'))
async def save_video_handler(callback: CallbackQuery, state: FSMContext):
    try:
        video_id = callback.data.split('_', 1)[1]

        await state.update_data(video_id=video_id)
        await callback.message.answer("📝 Введіть яку назву ви б хотіли, для збереження цього
відео(макс. 50 символів):")
        await state.set_state(VideoStates.waiting_for_video_name)
        await callback.answer()

    except Exception as e:
        await callback.answer(f"❌ Помилка: {str(e)}")


@router.message(VideoStates.waiting_for_video_name, F.text)
async def process_video_name(message: Message, state: FSMContext, kafka_producer):
    try:
        user_data = await state.get_data()
        video_id = user_data['video_id']
        custom_name = message.text.strip()[:100]

        event = {
            "user_id": message.from_user.id,
            "name_video": custom_name[:50],
            "response": video_id
        }

        await kafka_producer.send_message("bot-events", json.dumps(event))

        await message.answer(f"✅ Збережено як: {custom_name}")

    except Exception as e:
        await message.answer(f"⛔ Помилка збереження: {str(e)}")

    await state.clear()
```

ДОДАТОК Е

Лістинг файлів директорії services

db.py

```
from dotenv import load_dotenv
load_dotenv()  # Завантаження змінних середовища перед створенням движка

import os
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
from sqlalchemy.orm import sessionmaker, declarative_base
from sqlalchemy import Column, Integer, BigInteger, String, Text, DateTime, func

# Підключення до БД
DATABASE_URL = (
    f"mysql+asyncmy://{os.getenv('DB_USER')}:{os.getenv('DB_PASS')}@"
    f"{os.getenv('DB_HOST')}:{os.getenv('DB_PORT')}/{os.getenv('DB_NAME')}"
)

engine = create_async_engine(
    DATABASE_URL,
    echo=True,  # для дебагу SQL-запитів
    future=True,
)

# Створення сесій
AsyncSessionLocal = sessionmaker(
    bind=engine,
    class_=AsyncSession,
    expire_on_commit=False,
)

Base = declarative_base()

# Опис моделі
class History(Base):
    __tablename__ = 'history'
    id = Column(Integer, primary_key=True, index=True)
    user_id = Column(BigInteger, nullable=False, index=True)
    command = Column(String(64), nullable=False)
    response = Column(Text, nullable=False)
    message_id = Column(BigInteger, nullable=False, index=True)
    created_at = Column(DateTime(timezone=True), server_default=func.now())

class LoveVideo(Base):
    __tablename__ = 'lovevideo'
```

```
    id = Column(Integer, primary_key=True, index=True)
    user_id = Column(BigInteger, nullable=False, index=True)
    name_video = Column(String(64), nullable=False)
    response = Column(Text, nullable=False)
    created_at = Column(DateTime(timezone=True), server_default=func.now())

# Ініціалізація схеми
async def init_db():
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.create_all)

# Функція отримання сесії
async def get_db():
    async with AsyncSessionLocal() as session:
        yield session
```

deepseek_client.py

```
import os
import asyncio
from openai import OpenAI
from dotenv import load_dotenv
from langdetect import LangDetectException
from googletrans import Translator

load_dotenv()

client = OpenAI(
    base_url="https://openrouter.ai/api/v1",
    api_key=os.getenv("OPENROUTER_API_KEY")
)

translator = Translator()

async def ask_deepseek(prompt: str) -> str:
    try:

        loop = asyncio.get_event_loop()
        response = await loop.run_in_executor(None, lambda: client.chat.completions.create(
            model="deepseek/deepseek-r1:free",
            messages=[{"role": "user", "content": prompt}]
        ))

        if not response.choices or not response.choices[0].message.content:
            return "□♂ Не вдалося отримати відповідь. Спробуйте ще раз."

        answer = response.choices[0].message.content

        return answer

    except LangDetectException:
        return "⚠ Помилка визначення мови запиту"
```

```
    except Exception as e:
        return f"🚨 Помилка: {str(e)}"
```

kafka_consumer_db.py

```
import os
import json
from aiokafka import AIOKafkaConsumer
from services.db import AsyncSessionLocal, History, LoveVideo

class KafkaDBConsumer:
    def __init__(self, bootstrap_servers: str, topic: str):
        # Инициализируем consumer один раз
        self.consumer = AIOKafkaConsumer(
            topic,
            bootstrap_servers=os.getenv("DB_KAFKA_BOOTSTRAP", "kafka:9092"),
            group_id="db-consumer-group"
        )

    async def start(self):
        # Запускаем consumer
        await self.consumer.start()
        try:
            # Читаем сообщения и сохраняем их
            async for msg in self.consumer:
                await self._process_message(msg)
        finally:
            await self.consumer.stop()

    async def stop(self):
        await self.consumer.stop()

    async def _process_message(self, msg):
        try:
            event = json.loads(msg.value.decode())
            if not 'name_video' in event:
                await self.save_to_db(event)
            else:
                await self.save_to_db_love(event)
            # Подтверждаем, что сообщение обработано
            await self.consumer.commit()
        except Exception as e:
            print(f"❌ Ошибка обработки сообщения: {e}")

    async def save_to_db(self, message: dict):
        async with AsyncSessionLocal() as session:
            try:
                history = History(
                    user_id=message["user_id"],
                    command=message["command"],
                    response=message["response"],
                    message_id=message["message_id"]
```

```
            )
            session.add(history)
            await session.commit()
        except Exception as e:
            print(f"❌ Ошибка сохранения: {e}")
            await session.rollback()

async def save_to_db_love(self, message: dict):
    async with AsyncSessionLocal() as session:
        try:
            lovevideo = LoveVideo(
                user_id=message["user_id"],
                name_video=message["name_video"],
                response=message["response"]
            )
            session.add(lovevideo)
            await session.commit()

        except Exception as e:
            print(f"❌ Ошибка сохранения: {e}")
            await session.rollback()
```

kafka_producer.py

```
# services/kafka_producer.py
from aiokafka import AIOKafkaProducer

class KafkaProducer:
    def __init__(self, bootstrap_servers: str):
        self.bootstrap_servers = bootstrap_servers
        self.producer = None

    async def start(self):
        self.producer = AIOKafkaProducer(
            bootstrap_servers=self.bootstrap_servers
        )
        await self.producer.start()

    async def stop(self):
        await self.producer.stop()

    async def send_message(self, topic: str, message: str):
        await self.producer.send_and_wait(topic, message.encode('utf-8'))
```

sheets_client.py

```
import os
import gspread
from gspread.exceptions import SpreadsheetNotFound, WorksheetNotFound
from dotenv import load_dotenv
load_dotenv()
```

```
def get_sheet_data(sheet_name: str):
    """
    Повертає всі рядки з аркуша sheet_name.
    Викидає зрозумілу помилку, якщо документ або аркуш не знайдені.
    """
    service_account_json = os.getenv('GSPREAD_SERVICE_ACCOUNT_JSON')
    spreadsheet_name = os.getenv('GSPREAD_SHEET_NAME')

    # Перевіряємо наявність змінних
    if not service_account_json or not spreadsheet_name:
        raise EnvironmentError(
            "У .env має бути GSPREAD_SERVICE_ACCOUNT_JSON і
GSPREAD_SHEET_NAME"
        )
    # Перевіряємо існування файлу ключа
    if not os.path.isfile(service_account_json):
        raise FileNotFoundError(
            f"Файл service account '{service_account_json}' не знайдено."
        )

    # Ініціалізуємо клієнта всередині функції
    gc = gspread.service_account(filename=service_account_json)

    # Спроба відкрити документ
    try:
        sh = gc.open_by_key(spreadsheet_name)
    except SpreadsheetNotFound:
        raise SpreadsheetNotFound(
            f"Google Spreadsheet з ім’ям '{spreadsheet_name}' не знайдено."
        )

    # Спроба отримати аркуш
    try:
        worksheet = sh.worksheet(sheet_name)
    except WorksheetNotFound:
        available = [ws.title for ws in sh.worksheets()]
        raise WorksheetNotFound(
            f"Аркуш '{sheet_name}' не знайдено. Доступні аркуші: {available}"
        )

    # Повертаємо дані
    return worksheet.get_all_values()

def send_sheet_data(sheet_name: str):
    spreadsheet_id = os.getenv('GSPREAD_SHEET_NAME')  # это должен быть именно ID
таблицы
    service_account_json = os.getenv('GSPREAD_SERVICE_ACCOUNT_JSON')
    gc = gspread.service_account(filename=service_account_json)
    sh = gc.open_by_key(spreadsheet_id)
    worksheet = sh.worksheet(sheet_name)
    return worksheet
```

weather_client.py

```
import os
import aiohttp
from dotenv import load_dotenv
load_dotenv()  # Завантаження змінних середовища перед створенням движка

BASE_URL = 'http://api.openweathermap.org/data/2.5/weather'

async def get_weather(city: str) -> dict:
    params = {'q': city, 'appid': os.getenv('OPENWEATHER_API_KEY'), 'units': 'metric'}
    async with aiohttp.ClientSession() as session:
        async with session.get(BASE_URL, params=params) as resp:
            data = await resp.json()
    return {
        'city': data['name'],
        'temp': data['main']['temp'],
        'description': data['weather'][0]['description']
    }
```

youtube_client.py

```
import os
import aiohttp
from dotenv import load_dotenv

load_dotenv()
BASE_URL = 'https://www.googleapis.com/youtube/v3/search'


async def search_youtube_videos(query: str) -> dict:
    params = {
        'q': query,
        'key': os.getenv('YOUTUBE_API_KEY'),
        'part': 'snippet',
        'type': 'video',
        'maxResults': 5
    }

    async with aiohttp.ClientSession() as session:
        async with session.get(BASE_URL, params=params) as resp:
            data = await resp.json()

    return {
        'query': query,
        'results': [
            {
                'title': item['snippet']['title'],
                'url': f"https://youtu.be/{item['id']['videoId']}",
                'channel': item['snippet']['channelTitle']
            } for item in data.get('items', [])
        ]    }
```

ДОДАТОК Ж

Лістинг файлу «.env»

```
DB_KAFKA_BOOTSTRAP=kafka:9092

# Telegram
TG_BOT_TOKEN=7909475197:AAGzq0kDK0kR56OQj_yFPFpnq0zqlM-7oYQ
# MySQL
DB_HOST=mysql
DB_PORT=3306
DB_USER=botuser
DB_PASS=12341
DB_NAME=telegram_bot_db
# OpenWeather
OPENWEATHER_API_KEY=8e0a0052b4d296f41d213b8786491b6e
# Google Sheets
YOUTUBE_API_KEY=AIzaSyBNfWr9-GnJHAmr0eH9B5aTqeNg7lRJMSw
GSPREAD_SHEET_NAME=Токен GSPREAD
GSPREAD_SERVICE_ACCOUNT_JSON=/app/secrets/creds.json
# OpenAI
NGROK_AUTHTOKEN=токен NGROK
OPENROUTER_API_KEY=токен OPENROUTER
```