МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ
Циклова комісія (кафедра) комп’ютерної інженерії та інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему

ЧАТБОТ ДЛЯ ПІДТРИМКИ КЛІЄНТІ В МЕСЕНДЖЕРІ TELEGRAM

Виконав: студент групи 1П-21
Спеціальності
121 Інженерія програмного забезпечення
Олександр КАРБІВНИЧИЙ
Керівник:
Вікторія НЕМЧЕНКО

Черкаси 2025

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ БІЗНЕС-КОЛЕДЖ

Кафедра комп'ютерної інженерії та інформаційних технологій

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувач кафедри КІ та ІТ

_____ Владислав ХОТУНОВ
(підпис)
«_____»_____2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Карбівничому Олександру Олексійовичу

1. Тема кваліфікаційної роботи «Чатбот для підтримки клієнтів на платформі Telegram»

Керівник роботи Немченко Вікторія Юріївна, викладач другої категорії

затверджені наказом закладу вищої освіти від «07» жовтня 2024 року № 68у.
2. Строк подання студентом кваліфікаційної роботи 02.06.2025
3. Вихідні дані до кваліфікаційної роботи Telegram Bot API, Python 3.9+, MongoDB, аналіз існуючих рішень чатботів для підтримки клієнтів, вимоги до автоматизованої системи підтримки клієнтів
4. Зміст кваліфікаційної роботи (перелік питань, які потрібно розробити) 1. Аналіз поточного стану предметної області. 2. Проєктування архітектури системи та бази даних. 3. Програмна реалізація чатбота з використанням Telegram Bot API та обробкою природної мови. 4. Тестування та оцінка ефективності розробленого рішення.
5. Дата видачі завдання 16.09.2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Терміни виконання етапів	Примітка про виконання з підписами керівника і студента
1	Вступ	14.10.2024	
2	Розділ 1. Огляд поточного стану предметної області	09.12.2024	
3	Розділ 2. Проєктування та реалізація програмного проєкту	10.03.2025	
4	Розділ 3. Тестування та супроводження	28.04.2025	
5	Висновки	12.05.2025	
6	Оформлення кваліфікаційної роботи (чистовий варіант)	26.05.2025	
7	Перевірка кваліфікаційної роботи на наявність ознак плагіату (за 10 днів до захисту)	02.06.2025	
8	Подання кваліфікаційної роботи на затвердження завідувачу кафедри (за 7 днів до захисту)	10.06.2025	

Студент _____ Олександр КАРБІВНИЧИЙ
(підпис)

Керівник роботи _____ Вікторія НЕМЧЕНКО
(підпис)

АНОТАЦІЯ

Кваліфікаційна робота на тему «Чатбот для підтримки клієнтів на платформі Telegram» охоплює вступ, основну частину, яка складається з трьох розділів, висновки та список використаних джерел. Загальний обсяг роботи становить 96 сторінок, включаючи 10 рисунків, 5 таблиць і 6 додатків. Список використаних джерел налічує 22 позиції.

У межах цієї роботи було розроблено спеціалізованого чатбота для месенджера Telegram, який автоматично взаємодіє з клієнтами, забезпечуючи базову інформаційну підтримку. Це рішення дозволяє підвищити ефективність обслуговування користувачів, зменшити навантаження на операторів служби підтримки, а також забезпечити постійну доступність сервісу.

Розробка реалізована з використанням сучасних технологій, зокрема Python 3.9+, Telegram Bot API та баз даних MongoDB і Redis. У роботі детально описано етапи аналізу предметної області, проєктування архітектури чатбота, реалізації обробки запитів користувачів та базових елементів обробки природної мови. Окрему увагу приділено тестуванню, оцінці ефективності взаємодії з користувачем і перспективам розширення функціональності.

Запропоноване рішення демонструє практичну цінність для бізнесу в контексті автоматизації клієнтської підтримки та є основою для подальших досліджень у галузі інтелектуальних інформаційних систем.

Ключові слова: Telegram, чатбот, клієнтська підтримка, Telegram Bot API, автоматизація, Python, MongoDB, обробка природної мови.

ABSTRACT

The qualification work on the topic “Chatbot for customer support on the Telegram platform” includes an introduction, the main part, which consists of three chapters, conclusions and a list of references. The total volume of the work is 96 pages, including 10 figures, 5 tables, and 6 appendices. The list of references includes 22 items.

As part of this work, we developed a specialized chatbot for the Telegram messenger that automatically interacts with customers, providing basic information support. This solution helps to improve the efficiency of customer service, reduce the workload of support operators, and ensure the constant availability of the service.

The development was implemented using modern technologies, including Python 3.9+, Telegram Bot API, and MongoDB and Redis databases. The paper describes in detail the stages of analyzing the subject area, designing the chatbot architecture, implementing user request processing, and basic elements of natural language processing. Particular attention is paid to testing, evaluation of the effectiveness of user interaction, and prospects for expanding functionality.

The proposed solution demonstrates practical value for business in the context of customer support automation and is the basis for further research in the field of intelligent information systems.

Keywords: Telegram, chatbot, customer support, Telegram Bot API, automation, Python, MongoDB, natural language processing.

ЗМІСТ

ВСТУП

У сучасному світі, де швидкість комунікації та якість обслуговування клієнтів стають ключовими конкурентними перевагами, використання автоматизованих систем підтримки клієнтів набуває особливої актуальності. Месенджери, зокрема Telegram, стали невід'ємною частиною повсякденного спілкування мільйонів користувачів, створюючи нові можливості для бізнесу в організації ефективної комунікації з клієнтами [1].

Актуальність теми дослідження полягає в необхідності створення ефективних, швидких і зручних інструментів для автоматизації процесів підтримки клієнтів. Чатботи в месенджерах дозволяють значно скоротити час реакції на запити користувачів, зменшити навантаження на операторів підтримки, оптимізувати витрати компаній та підвищити загальний рівень задоволеності клієнтів.

Telegram як платформа для розгортання чатбота має низку переваг: відкритий API, надійні механізми захисту даних, широкий функціонал для створення інтерактивних ботів, а також значну аудиторію користувачів. Це робить його оптимальним вибором для реалізації автоматизованих систем підтримки клієнтів.

Метою даної кваліфікаційної роботи є розробка функціонального чатбота на платформі Telegram для автоматизації процесів обслуговування та підтримки клієнтів, здатного обробляти типові запити користувачів, надавати релевантну інформацію та за необхідності перенаправляти запити до операторів підтримки.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

1. Проаналізувати існуючі рішення та системи підтримки клієнтів, визначити їх переваги та недоліки.

2. Дослідити можливості месенджера Telegram як платформи для розробки чатботів.

3. Спроєктувати архітектуру системи, розробити базу даних та інтерфейс взаємодії з користувачем.

4. Реалізувати програмний код чатбота з використанням сучасних технологій та методів програмування.

5. Провести тестування розробленого рішення та оцінити його ефективність.

6. Сформулювати рекомендації щодо подальшого вдосконалення та супроводу системи.

Об'єктом дослідження є система автоматизації підтримки клієнтів із використанням сучасних інформаційних технологій.

Предметом дослідження є методи та засоби створення чатботів у месенджері Telegram для організації ефективної комунікації з клієнтами.

Практична значущість роботи полягає в розробці функціонального програмного рішення, яке може бути впроваджено в діяльність компаній різного масштабу для оптимізації процесів підтримки клієнтів, підвищення швидкості обробки запитів та покращення якості обслуговування.

Результати даної кваліфікаційної роботи можуть бути використані як основа для подальших досліджень у сфері автоматизації клієнтської підтримки та розробки інтелектуальних систем комунікації на базі месенджерів.

РОЗДІЛ 1 ОГЛЯД ПОТОЧНОГО СТАНУ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз систем підтримки клієнтів

Системи підтримки клієнтів (СПК) еволюціонували від простих телефонних ліній до комплексних програмних платформ, що інтегрують різні канали комунікації. Сучасні СПК представляють собою багатоканальні рішення, що забезпечують інтеграцію телефонії, електронної пошти, месенджерів, соціальних мереж та інших засобів зв'язку. Ключовою функцією таких систем є забезпечення швидкого та ефективного обміну інформацією між компанією та її клієнтами, створення єдиної бази знань та автоматизація типових процесів обслуговування.

Аналіз ринку СПК демонструє значне зростання попиту на рішення, що використовують штучний інтелект та машинне навчання. За даними дослідження Gartner, до 2025 року понад 75% комерційних підприємств будуть використовувати автоматизовані системи підтримки клієнтів, що призведе до скорочення операційних витрат на обслуговування на 30%. Це пов'язано з тим, що більшість запитів від клієнтів (приблизно 60-70%) є типовими, повторюваними та не потребують глибокої експертизи для надання відповіді [1].

Архітектура сучасних систем підтримки клієнтів базується на модульному підході, що дозволяє інтегрувати різні компоненти в єдину екосистему. Основними модулями є: система обробки вхідних запитів, база знань, система маршрутизації звернень, аналітичний модуль та інтерфейси для різних каналів комунікації. Особливе значення має модуль автоматизації, який включає інструменти для створення та налаштування чатботів, голосових асистентів та інших автоматизованих агентів.

Технологічний стек сучасних СПК включає різноманітні інструменти та фреймворки для обробки даних та побудови інтелектуальних систем. Для

розробки чатботів використовуються спеціалізовані платформи (Dialogflow, Microsoft Bot Framework, Rasa), мови програмування (Python, JavaScript, Java) та фреймворки для обробки природної мови (NLTK, spaCy, TensorFlow). Ці технології дозволяють створювати гнучкі та масштабовані рішення, здатні адаптуватися до потреб бізнесу [6].

Порівняльний аналіз різних типів систем підтримки клієнтів дозволяє визначити їх ключові особливості, переваги та недоліки. Розглянемо основні типи СПК у вигляді таблиці.

Аналіз сучасних тенденцій у сфері підтримки клієнтів виявляє стійку тенденцію до автоматизації рутинних процесів. Чатботи в месенджерах стають одним із найбільш затребуваних інструментів, оскільки вони поєднують переваги самообслуговування з можливістю персоналізованої взаємодії. Дослідження Juniper Research прогнозує, що використання чатботів у бізнесі дозволить заощадити понад 8 мільярдів доларів щорічно до 2027 року, що підтверджує перспективність цього напрямку [3].

Особливо перспективними є гібридні моделі, що поєднують автоматизовані рішення з можливістю залучення людини-оператора у складних випадках. Такий підхід дозволяє оптимізувати ресурси компанії, забезпечуючи при цьому високу якість обслуговування. Важливо зазначити, що ефективність чатботів значною мірою залежить від інтеграції з іншими системами компанії (CRM, ERP, бази знань), що дозволяє їм надавати контекстуально релевантну інформацію.

Таблиця 1.1 Порівняльний аналіз систем підтримки клієнтів

Тип системи	**Характеристики**	**Переваги**	**Недоліки**	**Приклади рішень**
Системи управління заявками (Ticketing Systems)	Обробка запитів клієнтів у форматі "заявок", відстеження їх статусу та історії взаємодії	- Структурований підхід до обробки запитів- Зручне відстеження історії- Можливість категоризації та пріоритезації	- Затримки у відповідях - Негнучкість при нестандартних запитах - Високі вимоги до кваліфікації персоналу	Zendesk, Freshdesk, JIRA Service Desk
Системи самообслугову вання	Бази знань, FAQ, інструкції, відеоуроки, що дозволяють клієнтам вирішувати проблеми самостійно	- Зниження навантаження на підтримку - Доступність 24/7 - Масштабованість	- Обмежена ефективність при складних проблемах - Потреба в постійному оновленні контенту - Складність навігації	Help Scout, Confluence, WordPress з плагінами
Чатботи та віртуальні асистенти	Автоматизовані системи, що використовують NLP для розуміння та відповіді на запити клієнтів	- Миттєва реакція - Обробка великих обсягів запитів - Зниження операційних витрат	- Обмежене розуміння контексту - Складність впровадження - Потреба у тренуванні	IBM Watson Assistant, Amazon Lex, власні рішення на основі Telegram Bot API
Омніканальні платформи	Інтегровані рішення, що об'єднують різні канали комунікації в єдиний інтерфейс	- Єдиний простір для всіх комунікацій - Збереження контексту між каналами - Аналітика ефективності каналів	- Висока вартість впровадження - Складність інтеграції - Потреба у навчанні персоналу	Intercom, Salesforce Service Cloud, HubSpot Service Hub
Соціальні медіа як канал підтримки	Використання соціальних мереж для комунікації з клієнтами	- Присутність там, де вже є клієнти - Публічність взаємодії - Легкість впровадження	- Складність масштабування - Ризики репутаційного характеру - Обмежені можливості автоматизації	Hootsuite, Buffer, Sprout Social

1.2 Огляд месенджера Telegram як платформи для чатботів

Telegram є одним із найпопулярніших месенджерів, який станом на 2023 рік налічує понад 700 мільйонів активних користувачів щомісяця. Ключовими факторами, що сприяли такій популярності, є фокус на приватності та безпеці, багатий функціональний набір та відкрита політика щодо розробників. З моменту впровадження Bot API у 2015 році, Telegram став потужною платформою для створення чатботів для різних сценаріїв використання, включаючи підтримку клієнтів.

Архітектура бот-платформи Telegram базується на HTTP API, що дозволяє розробникам взаємодіяти з месенджером за допомогою різних мов програмування. Центральним компонентом є Bot API, який надає методи для отримання та надсилання повідомлень, обробки команд, роботи з мультимедійним контентом та інтерактивними елементами інтерфейсу. Сервери Telegram забезпечують безпечне зберігання та передачу даних, використовуючи шифрування MTProto.

Бот API Telegram пропонує широкий спектр функціональних можливостей, що роблять його привабливим для розробки систем підтримки клієнтів:

1. Веб-хуки та Long Polling - Два методи отримання оновлень для бота, що дозволяють обрати оптимальний варіант залежно від архітектури проєкту. Веб-хуки забезпечують миттєву доставку оновлень, але потребують публічно доступного HTTPS-сервера, тоді як Long Polling є простішим у впровадженні, але менш ефективним при високих навантаженнях.

2. Інтерактивні елементи інтерфейсу - Включають вбудовані клавіатури, інлайн-клавіатури, інлайн-режим та інлайн-кнопки, що дозволяють створювати інтуїтивно зрозумілі інтерфейси з мінімальним введенням тексту користувачем. Ці елементи значно спрощують навігацію у боті та структурують взаємодію з користувачем.

3. Підтримка мультимедіа - Можливість обміну фото, відео, аудіо, документами, стікерами та іншими типами вмісту, що робить комунікацію більш наочною та інформативною. Особливо цінним є можливість надсилати файли до 2 ГБ, що дозволяє, наприклад, ділитися інструкціями або керівництвами користувача.

4. Бот-платежі - Інтегрована система для прийому платежів від користувачів безпосередньо через бота, що відкриває можливості для комерційного використання. Це особливо актуально для e-commerce компаній, які можуть об'єднати підтримку клієнтів та процес покупки в єдиному інтерфейсі.

5. Локалізація - Інструменти для створення багатомовних ботів, що дозволяють обслуговувати клієнтів різних країн їхньою рідною мовою. Telegram надає зручні механізми для визначення мови користувача та переключення між різними мовними версіями інтерфейсу.

6. Групові чати та канали - Можливість додавати ботів до групових чатів та каналів, що дозволяє впроваджувати сценарії модерації, отримання зворотного зв'язку та масової комунікації. Це розширює можливості використання ботів за межі прямої взаємодії один на один.

Процес розробки бота для Telegram включає кілька ключових етапів: реєстрація бота через BotFather (офіційний бот Telegram для створення нових ботів), отримання API-ключа, вибір бібліотеки або фреймворка для роботи з API, розробка логіки бота, тестування та розгортання. На ринку існує широкий вибір бібліотек для різних мов програмування, що спрощують роботу з Bot API: python-telegram-bot та aiogram для Python, node-telegram-bot-api для Node.js, TelegramBots для Java, telegram-bot-sdk для PHP тощо [2].

У контексті розробки систем підтримки клієнтів, Telegram має низку переваг порівняно з іншими платформами для чатботів:

Таблиця 1.2 Порівняльний аналіз платформ для розробки чатботів для підтримки клієнтів

Критерій	**Telegram**	**Facebook Messenger**	**WhatsApp Business API**	**Viber Business Messages**
Доступність API	Безкоштовний, відкритий	Безкоштовний з обмеженнями	Платний, закритий процес верифікації	Безкоштовний з обмеженнями
Інтерактивні елементи	Багатий набір (інлайн-кнопки, клавіатури, інлайн-режим)	Обмежений набір (Quick replies, кнопки, шаблони)	Обмежений набір (шаблони повідомлень)	Середній набір (клавіатури, кнопки)
Швидкість розробки	Висока (миттєва реєстрація, простий API, багато бібліотек)	Середня (складна реєстрація, але хороша інтеграція з платформами)	Низька (довгий процес верифікації, обмеження та плата)	Середня (потрібна верифікація, але API простий)
Мультимедіа підтримка	Повна (фото, відео, аудіо, файли до 2ГБ)	Обмежена (обмеження розміру файлів)	Обмежена (обмеження типів та розмірів)	Стандартна (текст, фото, відео, файли до 200 МБ)
Обмеження	Мінімальні	24-годинне вікно для відповіді	Шаблони повідомлень, обмеження на розсилки	Обмеження на масові розсилки
Приватність даних	Висока (немає E2EE у ботах, але низьке втручання платформи)	Середня (дані доступні Facebook, без шифрування)	Середня (E2EE тільки в особистих чатах, API-дані – ні)	Середня (часткова підтримка шифрування, залежить від типу чату)
Кросплат-формність	Всі платформи, вебверсія	В основному мобільні пристрої	В основному мобільні пристрої	В основному мобільні пристрої
Вартість впровадження	Низька (простий API, багато SDK, немає плати)	Середня (безкоштовно, але складна реєстрація, інтеграція з Facebook)	Висока (оплата за шаблони, необхідна офіційна верифікація, хостинг через BSP)	Середня (безкоштовний API, але обмеження, можлива плата за масові повідомлення)

Порівнюючи різні платформи для створення чатботів підтримки клієнтів, можна зробити висновки на основі ключових критеріїв: доступність API, якість технічної документації, функціональність, швидкість розробки, підтримка

мультимедіа, обмеження, приватність, кросплатформність, аналітика та вартість впровадження.

Telegram виявляється найзручнішою платформою для розробки чатботів. Його API є повністю безкоштовним і відкритим – розробник може одразу почати роботу без будь-якої реєстрації чи верифікації. Технічна документація Telegram детальна та проста у використанні, з великою кількістю прикладів. Це дозволяє значно прискорити процес створення бота, навіть новачкам. Крім того, Telegram надає найширший набір інтерактивних елементів, включаючи інлайн-кнопки, кастомні клавіатури, інлайн-режим, що дає змогу реалізувати складні сценарії взаємодії з користувачем.

Щодо мультимедійної підтримки, Telegram дозволяє надсилати всі типи файлів (фото, відео, документи, аудіо) розміром до 2ГБ, що значно перевершує інші платформи, де є жорсткі обмеження. У плані обмежень Telegram також лідирує – ботам не накладаються часові чи шаблонні обмеження на розсилки. Щодо приватності, хоча Telegram-боти не мають наскрізного шифрування (E2EE), сама платформа не збирає персональні дані користувачів у таких масштабах, як Meta-платформи (Facebook Messenger і WhatsApp).

Кросплатформність Telegram також є перевагою: користувачі можуть спілкуватися з ботом через мобільний додаток, десктоп, або повноцінну вебверсію. Щодо аналітики, хоча Telegram пропонує лише базові можливості, її легко доповнити сторонніми інструментами або інтеграцією з Google Analytics. Найважливіше – це вартість впровадження: Telegram є безкоштовним, має багато бібліотек, SDK, готових прикладів і шаблонів, що зменшує фінансові та часові витрати.

Інші платформи, як-от Facebook Messenger, WhatsApp Business API та Viber Business Messages, мають свої переваги – зокрема, розширену аналітику, інтеграцію з бізнес-інструментами чи популярність серед певної аудиторії. Проте вони поступаються Telegram у зручності розробки, гнучкості інтерактивних

можливостей і простоті старту. WhatsApp має найвищу вартість впровадження через необхідність співпраці з бізнес-провайдерами (BSP), плату за повідомлення та складну процедуру верифікації.

Практика впровадження ботів Telegram у бізнес-процеси демонструє низку успішних кейсів. Наприклад, компанія Airbnb використовує Telegram-бота для оперативного реагування на запити гостей щодо бронювань, Uber інтегрував бота для відстеження поїздок та технічної підтримки, а Spotify надає персоналізовані музичні рекомендації через свого бота. Ці приклади підтверджують ефективність Telegram як платформи для підтримки клієнтів, особливо для сервісних компаній.

Технічні аспекти інтеграції Telegram-ботів із корпоративними системами включають розробку API-шлюзів для взаємодії з внутрішніми системами, впровадження механізмів автентифікації та авторизації, забезпечення безпеки даних при передачі між системами. Сучасні підходи передбачають використання мікросервісної архітектури, що дозволяє гнучко масштабувати окремі компоненти системи залежно від навантаження.

Отже, Telegram – найзручніша платформа для створення чатботів підтримки клієнтів, що обумовлено її відкритістю, простотою використання, функціональністю, відсутністю жорстких обмежень та низькими витратами на впровадження. Ці висновки базуються на аналізі за критеріями доступності, технічної реалізації, функціональності та вартості.

1.3 Аналіз існуючих рішень чатботів для підтримки клієнтів

Ринок чатботів для підтримки клієнтів демонструє стрімкий розвиток із постійною появою нових рішень та вдосконаленням існуючих. Аналіз сучасних рішень дозволяє виділити кілька основних категорій чатботів залежно від їх функціональності, технологій та сфер застосування.

На рис. 1.1 подано огляд поточного стану предметної області, що охоплює основні напрямки застосування чатботів, включаючи системи управління заявками, віртуальних асистентів, омніканальні платформи та інші елементи, які поєднуються в межах платформи Telegram.

Рисунок 1.1 – Огляд поточного стану предметної області

Текстові чатботи на базі правил (rule-based chatbots) представляють найпростіший тип систем, функціонування яких базується на заздалегідь визначених правилах та сценаріях. Такі боти використовують ключові слова та фрази для розпізнавання запитів користувачів та надання відповідей із бази знань. Перевагами цього підходу є простота впровадження, контрольованість та передбачуваність поведінки. Однак, такі боти мають обмежені можливості щодо розуміння природної мови та контексту спілкування, що робить їх ефективними лише для найбільш типових та простих запитів. Прикладами таких рішень є

Botsify та ManyChat, які дозволяють створювати боти без програмування, використовуючи візуальні редактори сценаріїв.

Боти на основі машинного навчання та NLP (Natural Language Processing) використовують алгоритми обробки природної мови для розуміння запитів користувачів незалежно від їх формулювання. Такі системи здатні аналізувати контекст, розпізнавати наміри користувача та вилучати важливі сутності з тексту. Цей тип ботів забезпечує більш природну взаємодію та адаптивність до різних формулювань запитів. Однак, вони потребують значних ресурсів для навчання та налаштування, а також постійного вдосконалення на основі нових даних. Провідними рішеннями в цій категорії є IBM Watson Assistant, Dialogflow від Google та Rasa, які надають потужні інструменти для розробки інтелектуальних ботів [4].

Гібридні системи поєднують підходи на основі правил та машинного навчання, забезпечуючи баланс між контрольованістю та гнучкістю. Такі системи використовують NLP для розуміння запитів та маршрутизації їх до відповідних сценаріїв, які реалізуються за допомогою правил. Це дозволяє поєднати переваги обох підходів: точність та передбачуваність правил із гнучкістю та природністю машинного навчання. Ефективність гібридних систем підтверджується їх широким використанням у компаніях різного масштабу, від стартапів до корпорацій. Прикладами таких рішень є Chatfuel, MobileMonkey та Bot Framework від Microsoft.

Аналіз існуючих чатботів для Telegram, що використовуються для підтримки клієнтів, дозволяє виділити ряд показових прикладів:

1. @BotFather - Хоча це не бот підтримки в класичному розумінні, він є прикладом добре структурованого бота з чітким функціоналом та інтуїтивним інтерфейсом. Його архітектура та підхід до навігації часто використовуються як зразок при розробці інших ботів.

2. @TelegramSupportBot - Офіційний бот підтримки Telegram, який демонструє ефективну комбінацію автоматизованих відповідей та можливості перемикання на спілкування з реальними операторами. Цей бот використовує структуровані меню та кнопки для навігації, а також розширені можливості для пошуку в базі знань.

3. @LivegramBot - Бот-конструктор, що дозволяє компаніям створювати власні боти підтримки без програмування. Надає функціонал для організації зворотного зв'язку, маршрутизації запитів до операторів та збору аналітики.

4. @jivosite_bot - Інтеграція популярної платформи онлайн-чатів Jivochat з Telegram, що дозволяє об'єднати різні канали комунікації в єдиний інтерфейс для операторів. Демонструє ефективний підхід до омніканальної підтримки.

5. @AirbnbBot - Бот для підтримки користувачів сервісу Airbnb, який надає інформацію про бронювання, допомагає з типовими питаннями та перенаправляє складні запити до операторів. Приклад галузевого застосування ботів для підтримки клієнтів.

Аналіз цих та інших рішень дозволяє виділити ключові паттерни та підходи, що забезпечують ефективність чатботів для підтримки клієнтів:

- Багаторівнева структура меню - Організація інформації та функціоналу в ієрархічну структуру, що спрощує навігацію та знаходження потрібної інформації.
- Контекстуальні підказки - Надання користувачу релевантних опцій та інформації залежно від поточного контексту спілкування.
- Інтелектуальний пошук - Можливість шукати інформацію в базі знань за ключовими словами та фразами.
- Гібридний режим роботи - Комбінація автоматизованих відповідей та можливості перемикання на спілкування з оператором.

- Персоналізація - Адаптація взаємодії залежно від історії спілкування, преференцій та профілю користувача.
- Аналітика та зворотний зв'язок - Збір та аналіз даних про взаємодію для постійного вдосконалення бота.

Технічна реалізація чатботів для Telegram варіюється залежно від вимог та масштабу проєкту. Для невеликих проєктів часто використовуються готові фреймворки та бібліотеки, такі як python-telegram-bot, aiogram (Python), node-telegram-bot-api (Node.js), які спрощують взаємодію з Bot API [5].

РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОЄКТУ

2.1 Аналіз вимог до програмного забезпечення, що розробляється

На основі аналізу предметної області, проведеного в першому розділі, сформульовано конкретні вимоги до чатбота для підтримки клієнтів у месенджері Telegram. Розроблювана система повинна забезпечувати автоматизацію процесів комунікації з клієнтами, надання релевантної інформації та перенаправлення складних запитів до операторів підтримки.

Для формалізації вимог до програмного забезпечення використано методологію User Stories, що дозволяє описати функціональність системи з точки зору кінцевих користувачів. Щодо початку роботи з ботом, користувач повинен мати можливість почати спілкування для отримання інформації або допомоги, при цьому бот має реагувати на команду /start привітанням, коротким описом функціональності та пропонувати меню основних опцій. Важливим аспектом є отримання інформації про продукти/послуги - користувач має мати можливість отримати інформацію про продукти/послуги компанії, щоб зрозуміти, чи відповідають вони його потребам, при цьому бот повинен надавати структурований каталог з можливістю переглядати детальну інформацію про кожен продукт чи послугу [7].

Для пошуку відповідей на типові запитання користувач має мати можливість швидко знайти відповіді, щоб не чекати на відповідь оператора, а бот повинен розпізнавати ключові слова в запитах та надавати релевантну інформацію з бази знань. У випадках, коли автоматична відповідь неможлива, має бути передбачена можливість зв'язку з оператором - користувач повинен мати можливість зв'язатися з живим оператором, якщо бот не може вирішити

проблему, при цьому бот має пропонувати таку опцію, реєструвати запит та інформувати про очікуваний час відповіді. Для покращення якості сервісу важливо забезпечити можливість надання зворотного зв'язку - користувач має мати можливість оцінити якість обслуговування та залишити коментар, а бот після завершення діалогу повинен пропонувати оцінити якість та залишити відгук.

Для адміністраторів системи важливими аспектами є управління базою знань - адміністратор повинен мати можливість додавати, редагувати та видаляти інформацію, щоб підтримувати актуальність контенту, при цьому адміністративний інтерфейс має дозволяти виконувати CRUD-операції з записами. Для аналізу ефективності роботи бота необхідний моніторинг активності - адміністратор повинен мати можливість переглядати статистику використання, щоб оцінювати ефективність та виявляти проблемні аспекти, а система має збирати та відображати дані про кількість звернень, популярні запити, ефективність відповідей тощо. Також важливим є управління маршрутизацією запитів - адміністратор має мати можливість налаштовувати правила маршрутизації запитів до різних операторів для оптимізації процесу обслуговування, а адміністративний інтерфейс повинен дозволяти встановлювати правила на основі типу запиту, часу, навантаження операторів тощо.

На основі зібраних вимог визначено функціональні та нефункціональні вимоги до системи. Функціональні вимоги включають обробку команд та повідомлень, взаємодію з базою знань, маршрутизацію запитів, формування інтерфейсу користувача, а також збір та аналіз даних. Система повинна розпізнавати та обробляти команди користувача, аналізувати текстові повідомлення для виявлення намірів та запитуваної інформації, підтримувати контекст спілкування в рамках одного діалогу. База знань має бути структурованою, містити інформацію про продукти/послуги, типові проблеми та

їх вирішення, а система повинна шукати відповіді на запити користувачів за ключовими словами та семантичною подібністю, а також надавати можливість адміністраторам оновлювати базу знань через спеціальний інтерфейс.

Щодо маршрутизації запитів, система має визначати, чи може запит бути оброблений автоматично або потребує втручання оператора, реєструвати запити до операторів та відстежувати їх статус, а також повідомляти користувача про очікуваний час відповіді оператора. Інтерфейс користувача повинен бути інтуїтивно зрозумілим, з використанням вбудованих клавіатур, інлайн-кнопок та структурованих меню, відображати інформацію у зручному для сприйняття форматі (текст, зображення, файли) та забезпечувати навігацію по ієрархічній структурі інформації. Для аналітики система має збирати дані про взаємодію користувачів з ботом, аналізувати їх для виявлення трендів та проблемних аспектів, а також генерувати звіти щодо ефективності роботи бота.

Нефункціональні вимоги охоплюють аспекти продуктивності, надійності, безпеки та масштабованості. Система повинна забезпечувати час відповіді на запит користувача не більше 2 секунд та підтримувати одночасну роботу з не менше ніж 1000 користувачів. Щодо надійності, система має бути доступною 99,9% часу, автоматично відновлюватися після збоїв без втрати даних та зберігати історію діалогів для можливості відновлення контексту. Безпека передбачає забезпечення конфіденційності даних користувачів, використання безпечних методів аутентифікації для адміністративного доступу та логування всіх дій адміністраторів для аудиту. Система повинна бути розширюваною - підтримувати додавання нових функціональних модулів без зміни існуючої архітектури, надавати можливість інтеграції з іншими системами через API та забезпечувати локалізацію інтерфейсу на різні мови.

2.2 Проєктування чатбота

2.2.1 Розробка архітектури системи

Архітектура розроблюваного чатбота для підтримки клієнтів базується на модульному підході, що забезпечує гнучкість, масштабованість та можливість повторного використання компонентів. Загальна схема архітектура системи представлена на рис. 2.1.

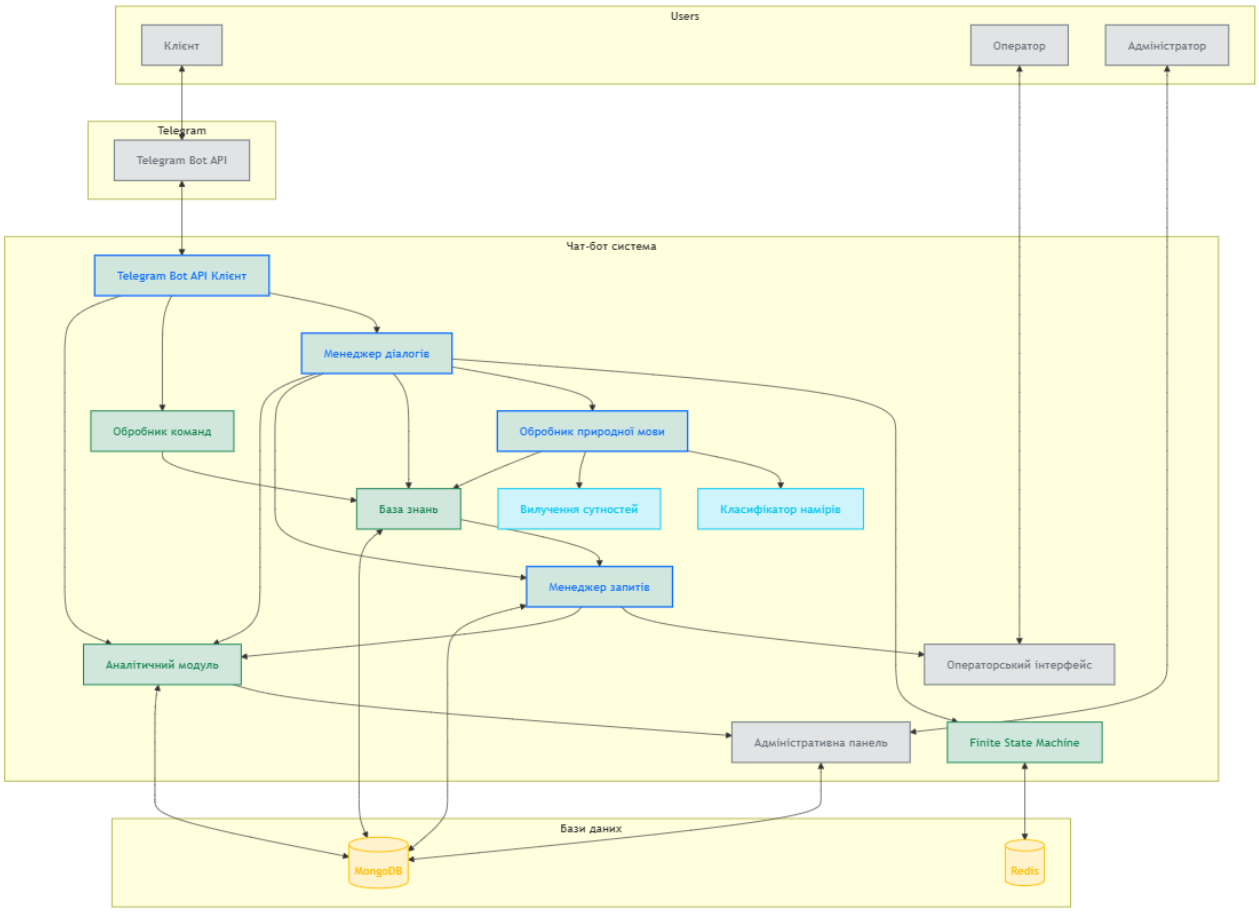


Рисунок 2.1 – Схема архітектури чатбота для підтримки клієнтів

Telegram Bot API Клієнт – модуль, що відповідає за взаємодію з Telegram Bot API через HTTP-запити. Цей компонент забезпечує отримання повідомлень від користувачів, відправку відповідей, обробку команд та інтерактивних елементів інтерфейсу. Реалізація базується на асинхронному підході для обробки

великої кількості запитів без блокування. Для взаємодії з Telegram Bot API використано бібліотеку aiogram, яка забезпечує зручний інтерфейс для роботи з API та підтримує всі необхідні функції [8].

Менеджер діалогів – компонент, відповідальний за управління станами діалогів з користувачами. Цей модуль відстежує контекст спілкування, зберігає інформацію про поточний стан діалогу та визначає наступні кроки взаємодії. Використання кінцевих автоматів (FSM - Finite State Machine) дозволяє структурувати діалоги та забезпечити логічну послідовність взаємодії. Кожен діалог моделюється як набір станів, між якими відбуваються переходи залежно від дій користувача.

Обробник природної мови (NLP) – модуль, що відповідає за аналіз текстових повідомлень користувачів, виявлення намірів (intents) та вилучення сутностей (entities). Цей компонент використовує алгоритми обробки природної мови для розуміння запитів незалежно від їх формулювання. Для реалізації використано бібліотеку SpaCy, яка надає інструменти для токенізації, лематизації, частотного аналізу та розпізнавання іменованих сутностей. Додатково впроваджено механізми нечіткого пошуку для обробки опечаток та різних форм запису.

База знань – централізоване сховище інформації про продукти/послуги, FAQ, інструкції та інші дані, необхідні для відповідей на запити користувачів. Цей компонент забезпечує структуроване зберігання та пошук інформації. Використано MongoDB як NoSQL СКБД, що надає гнучкість у структурі даних та швидкий пошук за ключовими словами. MongoDB забезпечує горизонтальне масштабування та підтримує індексацію для оптимізації запитів [20].

Менеджер маршрутизації запитів – компонент, що відповідає за визначення, чи може запит бути оброблений автоматично або потребує втручання оператора, а також за розподіл запитів між операторами. Цей модуль використовує правила маршрутизації, визначені адміністраторами, та дані про

доступність операторів. Реалізовано механізм чергування запитів з урахуванням пріоритетів та спеціалізації операторів.

Аналітичний модуль – компонент для збору, зберігання та аналізу даних про взаємодію користувачів з ботом. Цей модуль генерує звіти щодо ефективності роботи бота, популярних запитів, проблемних аспектів тощо. Використано бібліотеку pandas для обробки даних та matplotlib для візуалізації.

Адміністративний інтерфейс – вебінтерфейс для управління ботом, оновлення бази знань, налаштування правил маршрутизації, перегляду статистики тощо. Цей компонент реалізовано з використанням фреймворка Flask для створення REST API та React для користувацького інтерфейсу.

Для забезпечення взаємодії між компонентами системи використано підхід на основі подій (event-driven architecture), що дозволяє зменшити зв'язаність модулів та забезпечити гнучкість системи. Кожен компонент публікує події про свій стан та реагує на події від інших компонентів. Така архітектура сприяє масштабованості та можливості паралельної обробки запитів.

Система розгортається в контейнерах Docker, що забезпечує ізоляцію компонентів, спрощує процес розгортання та дозволяє легко масштабувати окремі компоненти залежно від навантаження. Використання оркестратора Kubernetes надає можливість автоматичного масштабування, балансування навантаження та відновлення після збоїв.

2.2.2 Проєктування бази даних

Для зберігання даних у системі підтримки клієнтів використано MongoDB як NoSQL СКБД, що забезпечує гнучкість у структурі даних, високу продуктивність та масштабованість. Вибір MongoDB обумовлений необхідністю зберігання різнорідних даних, частими операціями читання та можливістю горизонтального масштабування [9]. Схема бази даних представлена на рис. 2.2.

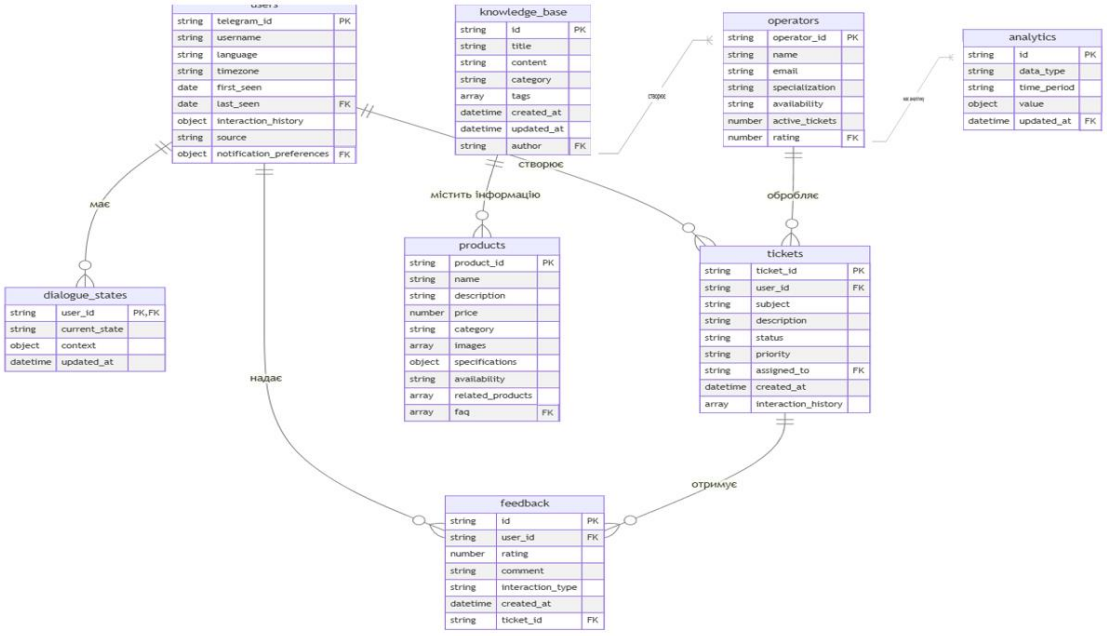


Рисунок 2.2 - Схема бази даних чатбота для підтримки клієнтів

Основними колекціями в базі даних є:

- Колекція "users" зберігає інформацію про користувачів бота. Кожен документ містить унікальний ідентифікатор користувача (telegram_id), ім'я користувача (username), мову інтерфейсу (language), часовий пояс (timezone), дату першого звернення (first_seen), дату останнього звернення (last_seen) та історію взаємодії (interaction_history). Додатково зберігаються метадані, такі як джерело, з якого користувач дізнався про бота (source), та преференції щодо сповіщень (notification_preferences). Для пришвидшення пошуку створено індекси за полями telegram_id, username та last_seen.

- Колекція "dialogue_states" відповідає за зберігання станів діалогів з користувачами. Кожен документ містить унікальний ідентифікатор користувача (user_id), поточний стан діалогу (current_state), контекст діалогу (context) та дату останнього оновлення (updated_at). Поле context зберігає дані, необхідні для підтримки контексту діалогу, такі як обрані продукти, сформульовані запитання тощо. Для забезпечення швидкого доступу до актуальних станів створено індекс за полями user_id та updated_at.

- Колекція "knowledge_base" містить структуровану інформацію для відповідей на запити користувачів. Кожен документ представляє окрему статтю або відповідь на питання та містить заголовок (title), вміст (content), категорію (category), теги (tags), дату створення (created_at), дату останнього оновлення (updated_at) та автора (author). Поле tags використовується для пошуку релевантних статей за ключовими словами. Для оптимізації пошуку створено індекси за полями category, tags та текстовий індекс за полями title та content.

- Колекція "products" зберігає інформацію про продукти/послуги компанії. Кожен документ містить унікальний ідентифікатор продукту (product_id), назву (name), опис (description), ціну (price), категорію (category), зображення (images), характеристики (specifications) та статус доступності (availability). Додатково зберігаються дані про супутні продукти (related_products) та часто задавані питання щодо продукту (faq). Для пришвидшення пошуку створено індекси за полями product_id, category та availability, а також текстовий індекс за полями name та description.

- Колекція "tickets" відповідає за зберігання запитів, які потребують втручання операторів. Кожен документ містить унікальний ідентифікатор запиту (ticket_id), ідентифікатор користувача (user_id), тему (subject), опис проблеми (description), статус (status), пріоритет (priority), призначеного оператора (assigned_to), дату створення (created_at) та історію взаємодії (interaction_history). Поле status може мати значення: "new", "in_progress", "waiting_for_customer", "resolved", "closed". Для ефективного управління запитами створено індекси за полями ticket_id, user_id, status та assigned_to.

- Колекція "operators" зберігає інформацію про операторів підтримки. Кожен документ містить унікальний ідентифікатор оператора (operator_id), ім'я (name), електронну пошту (email), спеціалізацію (specialization), статус доступності (availability), кількість активних запитів (active_tickets) та рейтинг

(rating). Для оптимізації процесу призначення запитів створено індекси за полями availability, specialization та active_tickets.

- Колекція "feedback" містить відгуки користувачів про якість обслуговування. Кожен документ включає ідентифікатор користувача (user_id), оцінку (rating), коментар (comment), тип взаємодії (interaction_type), дату створення (created_at) та посилання на запит, якщо відгук стосується конкретного звернення (ticket_id). Для аналізу зворотного зв'язку створено індекси за полями rating, interaction_type та created_at.
- Колекція "analytics" зберігає агреговані дані для аналітики. Кожен документ містить тип даних (data_type), часовий період (time_period), значення (value) та дату оновлення (updated_at). Приклади типів даних: "active_users", "popular_queries", "response_time" тощо. Для швидкого доступу до актуальних даних створено індекси за полями data_type та time_period.

Взаємодія з базою даних відбувається через спеціальний сервіс, що надає асинхронні методи для CRUD-операцій з колекціями. Ця абстракція дозволяє ізолювати бізнес-логіку від деталей реалізації бази даних та забезпечити можливість заміни СКБД у майбутньому. Для кожної колекції створено окремий клас, що надає специфічні методи для роботи з даними, такі як пошук за ключовими словами, агрегація статистики тощо.

2.2.3 Розробка UML-діаграм взаємодії компонентів

Для моделювання взаємодії компонентів системи підтримки клієнтів та візуалізації процесів обробки запитів розроблено набір UML-діаграм, що описують статичну структуру системи та динамічну поведінку. Ці діаграми є важливими артефактами проєктування, що допомагають зрозуміти архітектуру системи та принципи її функціонування [14].

TelegramBot - клас, що відповідає за взаємодію з Telegram Bot API. Включає методи для отримання оновлень, відправки повідомлень, обробки команд та управління інтерактивними елементами інтерфейсу. Зв'язаний з DialogueManager для управління станами діалогів та MessageHandler для обробки повідомлень.

DialogueManager - клас для управління станами діалогів. Відповідає за збереження та відновлення контексту спілкування, визначення поточного стану та переходи між станами. Використовує патерн "Стан" (State) для моделювання різних етапів діалогу.

MessageHandler – клас для обробки повідомлень від користувачів. Включає методи для попередньої обробки тексту, розпізнавання намірів та сутностей, генерації відповідей. Зв'язаний з NLPProcessor для аналізу тексту та KnowledgeBaseService для пошуку інформації.

NLPProcessor – клас для обробки природної мови. Відповідає за токенізацію, лематизацію, частиномовний аналіз, розпізнавання іменованих сутностей та класифікацію намірів. Використовує зовнішні бібліотеки для NLP, такі як SpaCy.

KnowledgeBaseService – клас для взаємодії з базою знань. Включає методи для пошуку інформації за ключовими словами, категоріями, семантичною подібністю. Зв'язаний з MongoDB для доступу до даних.

TicketManager – клас для управління запитами, що потребують втручання операторів. Відповідає за створення запитів, призначення операторів, відстеження статусу та комунікацію між користувачами та операторами.

AnalyticsService – клас для збору та аналізу даних про взаємодію користувачів з ботом. Включає методи для логування подій, агрегації статистики, генерації звітів.

AdminInterface – клас для управління ботом через адміністративний інтерфейс. Відповідає за оновлення бази знань, налаштування правил маршрутизації, перегляд статистики.

Діаграма послідовностей (рис. 2.4) відображає взаємодію об'єктів у часі при обробці типового запиту користувача. Ця діаграма демонструє потік повідомлень між компонентами системи та порядок їх виконання (дивитися рис. 2.3)

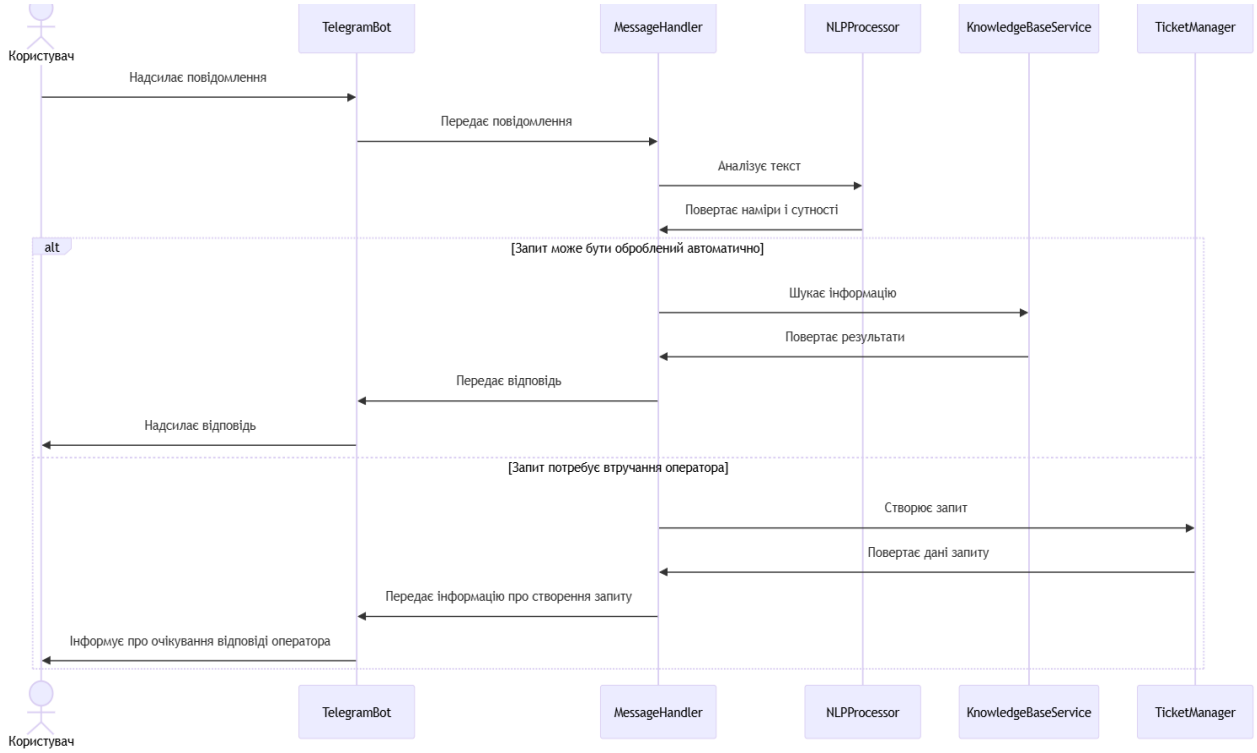


Рисунок 2.3 - Діаграма послідовностей обробки запиту користувача

Послідовність дій при обробці запиту включає:

1. Користувач надсилає повідомлення через Telegram.
2. TelegramBot отримує оновлення від Telegram Bot API та передає повідомлення до MessageHandler.
3. MessageHandler виконує попередню обробку тексту та передає його до NLPProcessor для аналізу.
4. NLPProcessor розпізнає наміри та сутності в повідомленні та повертає результати до MessageHandler.

5. MessageHandler звертається до DialogueManager для отримання поточного контексту діалогу.

6. На основі намірів, сутностей та контексту MessageHandler визначає, чи може запит бути оброблений автоматично.

7. Якщо запит може бути оброблений автоматично, MessageHandler звертається до KnowledgeBaseService для пошуку релевантної інформації.

8. KnowledgeBaseService виконує пошук у базі знань та повертає результати до MessageHandler.

9. MessageHandler формує відповідь на основі отриманої інформації та передає її до TelegramBot для відправки користувачу.

10. TelegramBot відправляє повідомлення через Telegram Bot API.

11. Якщо запит не може бути оброблений автоматично, MessageHandler передає його до TicketManager для створення запиту до оператора.

12. TicketManager створює запит, призначає оператора та інформує користувача про очікуваний час відповіді.

13. AnalyticsService логує всі події для подальшого аналізу.

Діаграма діяльності (рис. 2.4) відображає потік дій при обробці запиту користувача, включаючи умови та розгалуження. Ця діаграма допомагає зрозуміти логіку прийняття рішень у системі.

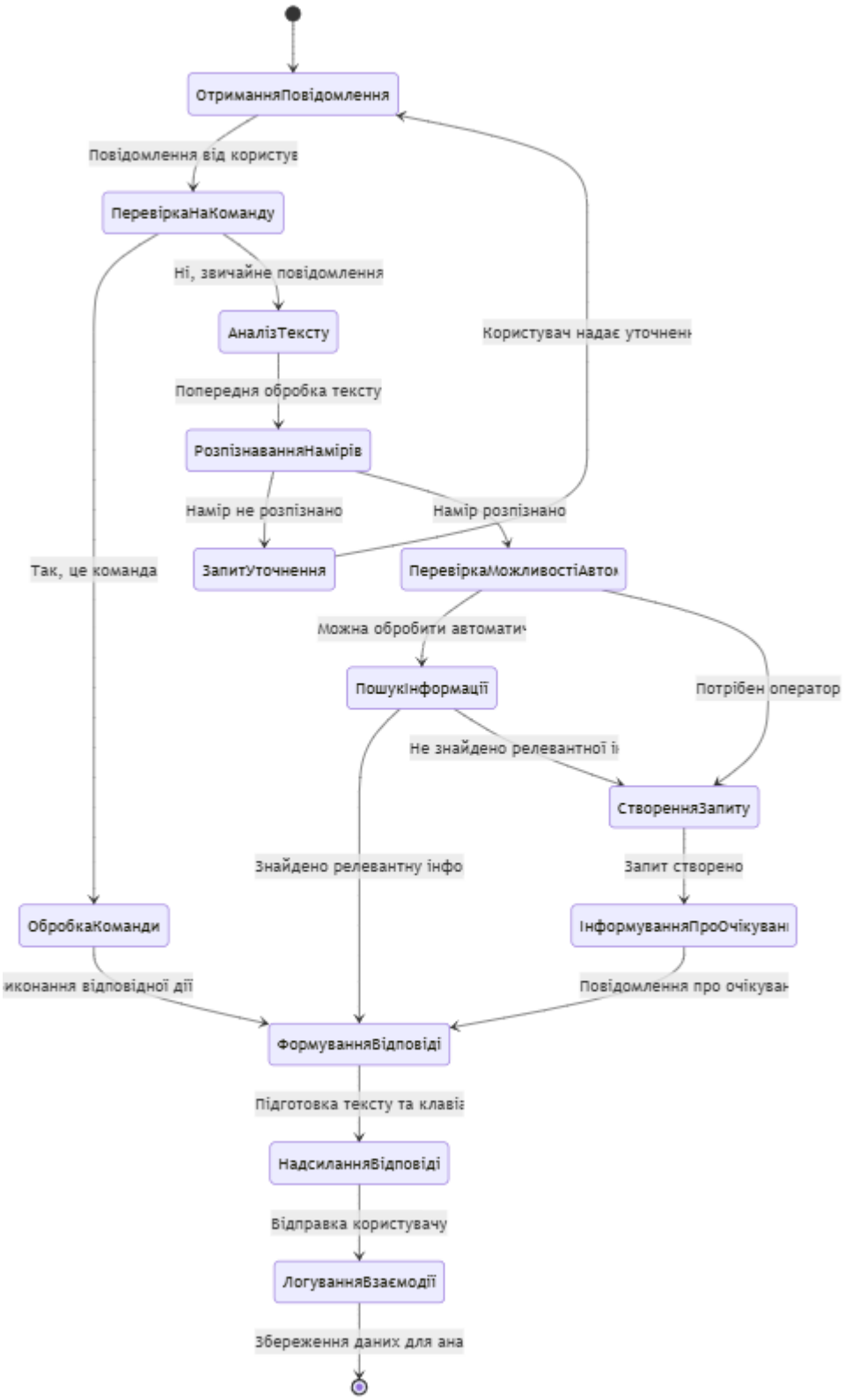


Рисунок 2.4 - Діаграма діяльності обробки запиту користувача

Потік дій включає:

1. Отримання повідомлення від користувача.
2. Перевірка, чи є повідомлення командою (наприклад, /start, /help).
3. Якщо повідомлення є командою, виконується відповідна дія (наприклад, відображення привітання, надання допомоги).
4. Якщо повідомлення не є командою, виконується аналіз тексту для розпізнавання намірів та сутностей.
5. Перевірка, чи вдалося розпізнати наміри користувача.
6. Якщо наміри не розпізнано, система запитує уточнення у користувача.
7. Якщо наміри розпізнано, система перевіряє, чи може запит бути оброблений автоматично.
8. Якщо запит може бути оброблений автоматично, система шукає релевантну інформацію в базі знань.
9. Система формує відповідь на основі знайденої інформації.
10. Якщо запит не може бути оброблений автоматично, система створює запит до оператора.
11. Система інформує користувача про очікуваний час відповіді оператора.
12. Система логує результати взаємодії для аналітики.

Схема компонентів (рис. 2.5) відображає організацію та залежності між компонентами системи. Ця діаграма демонструє модульну структуру системи та інтерфейси взаємодії.

Основними компонентами системи є:

Telegram Bot API Client - компонент для взаємодії з Telegram Bot API. Надає інтерфейси для отримання оновлень та відправки повідомлень.

Dialog Management System - компонент для управління діалогами. Надає інтерфейси для збереження та відновлення контексту, визначення станів та переходів.

Natural Language Processing - компонент для обробки природної мови. Надає інтерфейси для аналізу тексту, розпізнавання намірів та сутностей, класифікації запитів.

Knowledge Base System - компонент для управління базою знань. Надає інтерфейси для пошуку інформації, додавання та оновлення контенту.

Ticket Management System - компонент для управління запитами до операторів. Надає інтерфейси для створення запитів, призначення операторів, відстеження статусу.

Analytics System - компонент для збору та аналізу даних. Надає інтерфейси для логування подій, агрегації статистики, генерації звітів.

Administrative Interface - компонент для управління ботом. Надає вебінтерфейси для оновлення бази знань, налаштування правил маршрутизації, перегляду статистики.

Компоненти взаємодіють через чітко визначені інтерфейси, що забезпечує низьку зв'язаність та можливість незалежного розвитку кожного компонента. Така архітектура дозволяє замінювати окремі компоненти без впливу на роботу всієї системи.

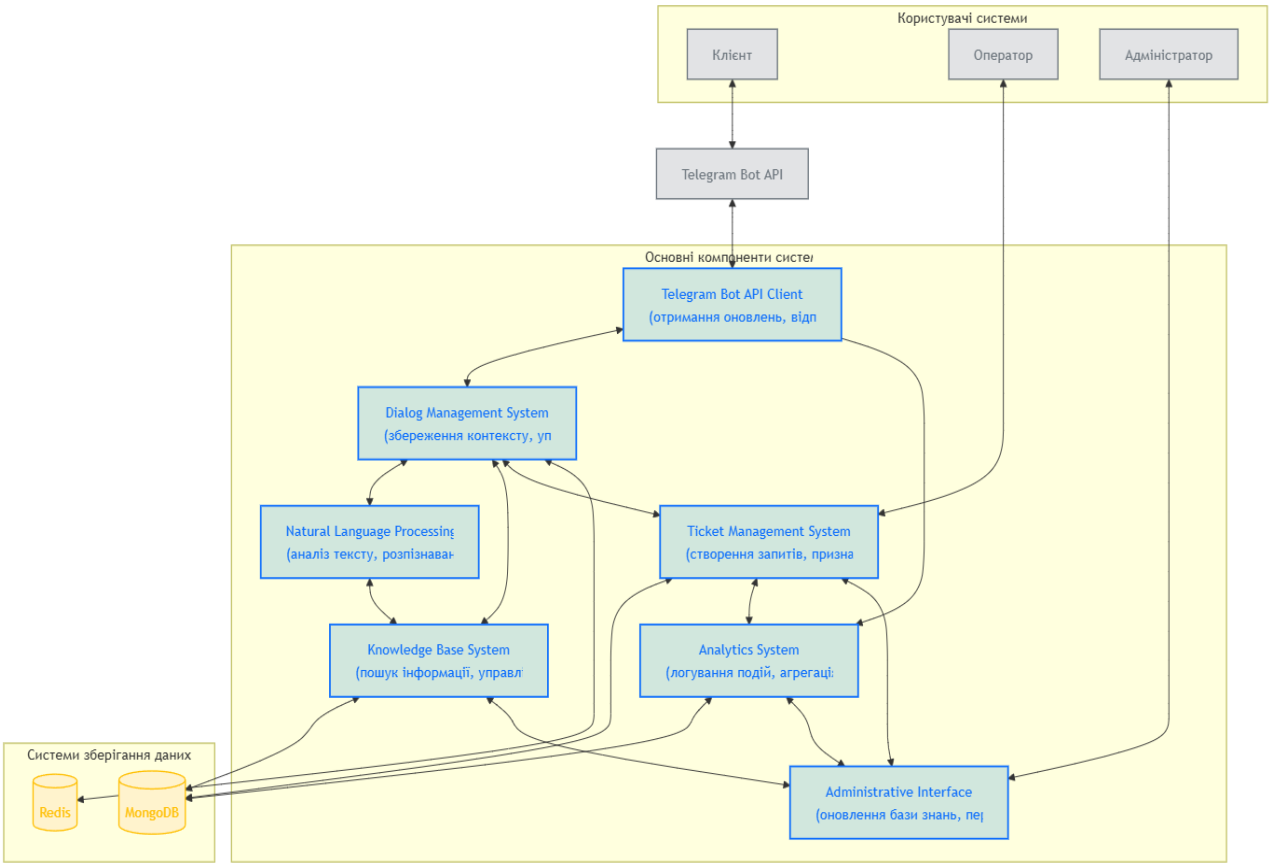


Рисунок 2.5 - Схема компонентів чатбота для підтримки клієнтів

2.2.4 Проєктування інтерфейсу користувача

Інтерфейс користувача є ключовим аспектом чатбота для підтримки клієнтів, оскільки від його зручності та інтуїтивності безпосередньо залежить ефективність взаємодії з користувачами. При проєктуванні інтерфейсу були враховані особливості платформи Telegram та принципи UX-дизайну для чатботів.

Основними принципами, якими керувались при проєктуванні інтерфейсу, є:

- Простота та інтуїтивність - інтерфейс повинен бути зрозумілим навіть для користувачів, які вперше взаємодіють з ботом. Для цього використано чіткі та зрозумілі формулювання, структуровані меню та підказки.

- Мінімізація введення тексту - для зменшення когнітивного навантаження та прискорення взаємодії максимально використано вбудовані клавіатури, інлайн-кнопки та інші інтерактивні елементи, що дозволяють користувачу обирати з готових варіантів.
- Збереження контексту - інтерфейс забезпечує видимість контексту спілкування, щоб користувач розумів, на якому етапі діалогу він знаходиться та яку інформацію вже надав.
- Інформативність та стислість - повідомлення бота містять достатньо інформації для відповіді на запит користувача, але при цьому є стислими та структурованими для легкого сприйняття.
- Доступність допомоги - на кожному етапі взаємодії користувач має можливість отримати додаткову інформацію або повернутися до головного меню.

Для реалізації цих принципів використано наступні елементи інтерфейсу Telegram:

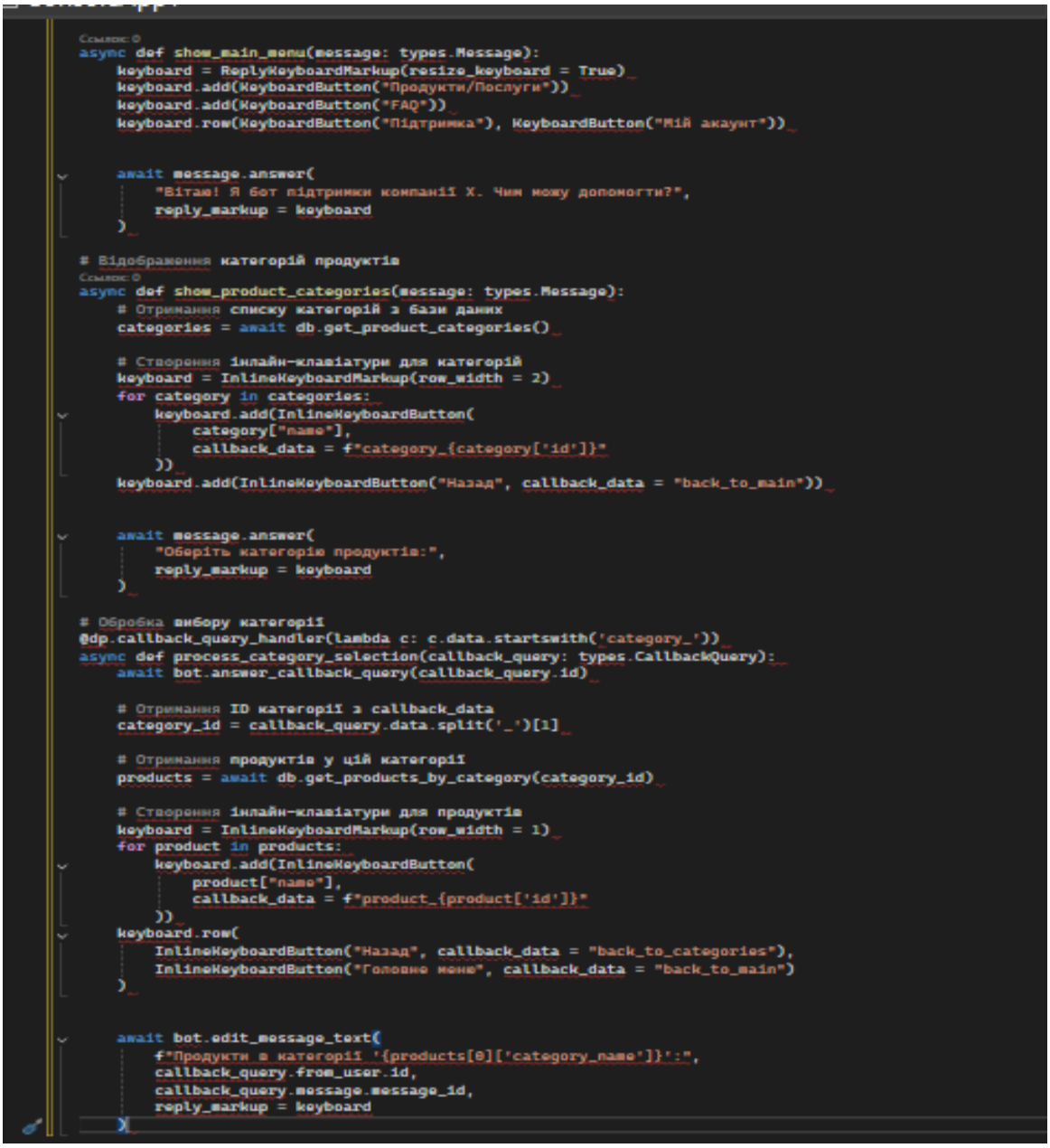
1. Команди - для ініціювання основних дій (/start, /help, /faq, /contact тощо). Команди є найпростішим способом доступу до функціональності бота та завжди доступні через меню команд Telegram.
2. Вбудовані клавіатури (Reply Keyboards) - для відображення основних опцій на кожному етапі діалогу. Клавіатури розміщуються під полем введення повідомлення та пропонують користувачу готові варіанти відповідей. Цей елемент особливо корисний для навігації по категоріях продуктів, виборі типу запиту тощо.
3. Інлайн-кнопки (Inline Keyboards) - для інтерактивних дій безпосередньо в повідомленнях бота. Кнопки вбудовуються в повідомлення та дозволяють користувачу виконувати дії, такі як перегляд детальної інформації, підтвердження вибору, перехід до наступного кроку тощо. Цей елемент

використано для реалізації пагінації при відображенні списків, оцінки якості обслуговування тощо.

4. Інлайн-режим (Inline Mode) - для швидкого доступу до функціональності бота з будь-якого чату через введення "@bot_name" та запиту. Цей режим дозволяє користувачам шукати інформацію в базі знань без необхідності переходу до чату з ботом, що особливо корисно для часто запитуваної інформації.

5. Форматований текст (Markdown/HTML) - для структурування та виділення важливої інформації в повідомленнях. Використання форматування (заголовки, жирний текст, списки тощо) покращує сприйняття інформації та дозволяє користувачу швидко знаходити потрібні дані.

Процес взаємодії користувача з ботом розпочинається з команди /start, після якої бот надсилає привітання, коротко описує свою функціональність та пропонує основне меню з найпопулярнішими опціями. Основне меню представлено вбудованою клавіатурою з кнопками "Продукти/Послуги", "FAQ", "Підтримка", "Мій акаунт" тощо (дивитися рис. 2.6)



```python
async def show_main_menu(message: types.Message):
    keyboard = ReplyKeyboardMarkup(resize_keyboard = True)
    keyboard.add(KeyboardButton("Продукти/Послуги"))
    keyboard.add(KeyboardButton("FAQ"))
    keyboard.row(KeyboardButton("Підтримка"), KeyboardButton("Мій акаунт"))


    await message.answer(
        "Вітаю! Я бот підтримки компанії X. Чим можу допомогти?",
        reply_markup = keyboard
    )

# Відображення категорій продуктів
async def show_product_categories(message: types.Message):
    # Отримання списку категорій з бази даних
    categories = await db.get_product_categories()

    # Створення інлайн-клавіатури для категорій
    keyboard = InlineKeyboardMarkup(row_width = 2)
    for category in categories:
        keyboard.add(InlineKeyboardButton(
            category["name"],
            callback_data = f"category_{category['id']}"
        ))
    keyboard.add(InlineKeyboardButton("Назад", callback_data = "back_to_main"))


    await message.answer(
        "Оберіть категорію продуктів:",
        reply_markup = keyboard
    )

# Обробка вибору категорії
@dp.callback_query_handler(lambda c: c.data.startswith('category_'))
async def process_category_selection(callback_query: types.CallbackQuery):
    await bot.answer_callback_query(callback_query.id)

    # Отримання ID категорії з callback_data
    category_id = callback_query.data.split('_')[1]

    # Отримання продуктів у цій категорії
    products = await db.get_products_by_category(category_id)

    # Створення інлайн-клавіатури для продуктів
    keyboard = InlineKeyboardMarkup(row_width = 1)
    for product in products:
        keyboard.add(InlineKeyboardButton(
            product["name"],
            callback_data = f"product_{product['id']}"
        ))
    keyboard.row(
        InlineKeyboardButton("Назад", callback_data = "back_to_categories"),
        InlineKeyboardButton("Головне меню", callback_data = "back_to_main")
    )


    await bot.edit_message_text(
        f"Продукти в категорії '{products[0]['category_name']}':",
        callback_query.from_user.id,
        callback_query.message.message_id,
        reply_markup = keyboard
    )
```

Рисунок 2.6 - Створення інтерфейсу користувача з використанням бібліотеки aiogram

При виборі категорії "Продукти/Послуги" бот відображає список категорій продуктів, представлений інлайн-кнопками з можливістю пагінації. Користувач може обрати категорію, після чого бот надсилає список продуктів у цій категорії. Для кожного продукту відображається коротка інформація та інлайн-кнопки для

перегляду детальної інформації, додавання до обраного, порівняння з іншими продуктами тощо.

При виборі категорії "FAQ" бот відображає список найпопулярніших запитань, згрупованих за темами. Користувач може обрати тему та конкретне запитання, після чого бот надсилає відповідь з можливістю задати додаткові запитання за цією темою.

При виборі категорії "Підтримка" бот пропонує користувачу описати проблему або обрати з типових проблем. Після отримання опису бот аналізує запит та визначає, чи може він бути оброблений автоматично. Якщо бот знаходить релевантну інформацію в базі знань, він надсилає її користувачу та запитує, чи вирішена проблема. Якщо проблема не вирішена або бот не може знайти релевантну інформацію, він пропонує створити запит до оператора. При створенні запиту бот збирає необхідну інформацію (тип продукту, опис проблеми тощо) та інформує користувача про очікуваний час відповіді.

Для забезпечення доступності допомоги на кожному етапі взаємодії бот відображає кнопку "Допомога" або "Головне меню", що дозволяє користувачу отримати додаткову інформацію або повернутися до початкового стану діалогу.

Важливим аспектом інтерфейсу є обробка помилок та неочікуваних введень. Якщо бот не розуміє повідомлення користувача, він інформує про це та пропонує альтернативні способи взаємодії, такі як вибір з меню, використання команд або перефразування запиту. Це запобігає "зависанню" діалогу та покращує досвід користувача [10].

2.3 Програмна реалізація проєкту

2.3.1 Вибір технологій та інструментів розробки

Для реалізації чатбота для підтримки клієнтів обрано набір технологій, що забезпечує ефективну розробку, масштабованість та підтримку системи.

Основною мовою програмування обрано Python (версія 3.9+) завдяки простоті, багатій екосистемі бібліотек та підтримці асинхронного програмування через asyncio, що дозволяє ефективно обробляти велику кількість запитів без блокування.

Таблиця 2.1 Основні технології розробки

Технологія	**Призначення**	**Ключові переваги**
Python 3.9+	Основна мова програмування	Простота, асинхронне програмування, багата екосистема
aiogram 2.14+	Взаємодія з Telegram Bot API	Асинхронний підхід, підтримка FSM
MongoDB 4.4+	NoSQL СКБД	Гнучкість, масштабованість, індексація
SpaCy 3.0+	Обробка природної мови	Висока продуктивність, точність аналізу
Redis 6.0+	Кешування, зберігання станів	Швидкий доступ, підтримка різних структур даних
Docker/ Kubernetes	Контейнеризація/ оркестрація	Ізоляція, спрощене розгортання, масштабування

Для взаємодії з Telegram Bot API обрано бібліотеку aiogram (версія 2.14+), яка надає зручний асинхронний інтерфейс з підтримкою диспетчерів, обробників, фільтрів та FSM для управління діалогами. MongoDB (версія 4.4+) обрано як NoSQL СКБД для зберігання даних через гнучкість структури даних, високу продуктивність та можливість горизонтального масштабування. Для взаємодії з MongoDB використано асинхронний драйвер motor.

Для обробки природної мови обрано SpaCy (версія 3.0+), що надає інструменти для токенізації, лематизації, частиномовного аналізу та

розпізнавання сутностей. Для класифікації намірів використано scikit-learn. Redis (версія 6.0+) застосовано для зберігання станів діалогів та кешування.

Адміністративний інтерфейс реалізовано з використанням FastAPI для серверної частини та React для клієнтської. Для розгортання використано Docker та Docker Compose, з можливістю оркестрації через Kubernetes у продакшн-середовищі. Тестування виконано через pytest, з автоматизацією через CI/CD на базі GitLab [11].

2.3.2 Розробка основних функціональних модулів

Програмна реалізація чатбота базується на модульній архітектурі з чітким розподілом відповідальності. Основними функціональними модулями є:

Модуль взаємодії з Telegram API (bot_client.py) відповідає за комунікацію з Telegram Bot API, включаючи отримання оновлень, відправку повідомлень та обробку команд. Реалізовано з використанням aiogram для асинхронної обробки запитів. Включає Bot для взаємодії з API, Dispatcher для маршрутизації повідомлень, та різні обробники для команд, повідомлень та колбеків [13].

Модуль управління діалогами (dialogue_manager.py) підтримує контекст спілкування, визначає поточний стан діалогу та керує переходами між станами. Використовує патерн "Стан" та механізм Finite State Machine з aiogram, зберігаючи стани в Redis. Включає класи для різних типів діалогів (ProductDialogue, SupportDialogue, FaqDialogue) з власними наборами станів (див. рис. 2.7).

Модуль обробки природної мови (nlp_processor.py) аналізує текстові повідомлення, розпізнає наміри та вилучає сутності. Використовує SpaCy для базової обробки тексту та scikit-learn для класифікації намірів. Включає функції для токенізації, лематизації, видалення стоп-слів, векторизації та класифікації запитів.

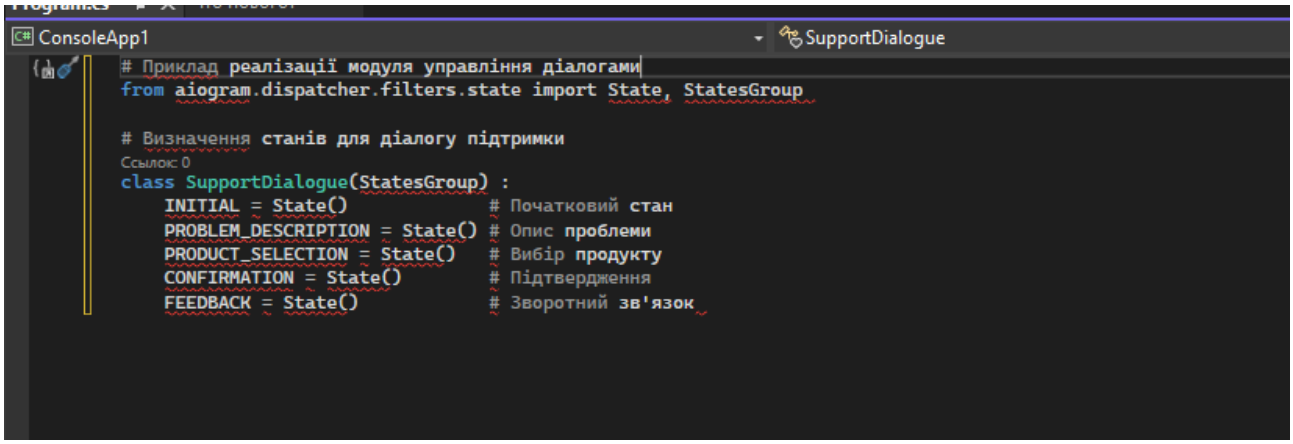


Рисунок 2.7 – Уппавліення реалізації модуля управління діалогами

Модуль взаємодії з базою знань (knowledge_base.py) відповідає за пошук інформації та формування відповідей. Використовує MongoDB і motor для асинхронної взаємодії. Включає функції для пошуку за ключовими словами, категоріями та семантичною подібністю.

Модуль управління запитами (ticket_manager.py) керує запитами до операторів. Використовує MongoDB для зберігання даних та Redis для управління чергами. Реалізує функції для створення запитів, призначення операторів та відстеження статусів.

Модуль аналітики (analytics.py) збирає та аналізує дані взаємодії. Використовує MongoDB для зберігання та pandas для аналізу. Включає функції для логування подій, агрегації статистики та генерації звітів.

Взаємодія між модулями організована через систему подій та колбеків, що зменшує зв'язаність компонентів. Система включає механізми обробки винятків, повторні спроби при тимчасових проблемах з зовнішніми API, транзакції для критичних операцій та кешування часто запитуваних даних.

2.3.3 Реалізація інтеграції з API Telegram

Інтеграція з Telegram Bot API починається з реєстрації бота через BotFather та отримання унікального токена для автентифікації запитів. Токен зберігається в захищеному вигляді в конфігураційному файлі або змінній середовища.

```
nsoleApp1
from aiogram import Bot, Dispatcher, types
from aiogram.contrib.fsm_storage.redis import RedisStorage2

# Ініціалізація бота та диспетчера
bot = Bot(token=os.getenv("TELEGRAM_BOT_TOKEN"))
storage = RedisStorage2(host=os.getenv("REDIS_HOST"))
dp = Dispatcher(bot, storage=storage)

# Обробник команди /start
@dp.message_handler(commands=["start"])
Ссылок: 0
async def cmd_start(message: types.Message):
    # Збереження інформації про користувача
    await db.save_user(message.from_user.id, message.from_user.username)

    # Створення основного меню та відправка привітання
    keyboard = ReplyKeyboardMarkup(resize_keyboard=True)
    keyboard.add(KeyboardButton("Продукти/Послуги"))
    keyboard.add(KeyboardButton("FAQ"))
    keyboard.row(KeyboardButton("Підтримка"), KeyboardButton("Мій акаунт"))

    await message.answer(
        f"Вітаю, {message.from_user.first_name}! Я бот підтримки компанії X.",
        reply_markup=keyboard
    )# Приклад реалізації обробників для команд
from aiogram import Bot, Dispatcher, types
from aiogram.contrib.fsm_storage.redis import RedisStorage2

# Ініціалізація бота та диспетчера
bot = Bot(token=os.getenv("TELEGRAM_BOT_TOKEN"))
storage = RedisStorage2(host=os.getenv("REDIS_HOST"))
dp = Dispatcher(bot, storage=storage)

# Обробник команди /start
@dp.message_handler(commands=["start"])
Ссылок: 0
async def cmd_start(message: types.Message):
    # Збереження інформації про користувача
    await db.save_user(message.from_user.id, message.from_user.username)

    # Створення основного меню та відправка привітання
    keyboard = ReplyKeyboardMarkup(resize_keyboard=True)
    keyboard.add(KeyboardButton("Продукти/Послуги"))
    keyboard.add(KeyboardButton("FAQ"))
    keyboard.row(KeyboardButton("Підтримка"), KeyboardButton("Мій акаунт"))

    await message.answer(
        f"Вітаю, {message.from_user.first_name}! Я бот підтримки компанії X.",
        reply_markup=keyboard
    )
```

Рисунок 2.8 - Взаємодія з API у використанні бібліотеки aiogram

Для взаємодії з API використано бібліотеку aiogram. У режимі розробки для отримання оновлень застосовано Long Polling, а в промисловій експлуатації – Webhooks для миттєвої доставки оновлень. Обробники в диспетчері використовують систему фільтрів для маршрутизації повідомлень, а стани діалогів керуються через FSM [12].

Для створення інтерактивного інтерфейсу використано вбудовані та інлайн-клавіатури, форматування повідомлень через Markdown/HTML та функції

для роботи з мультимедіа. Реалізовано інлайн-режим для швидкого доступу з будь-якого чату.

Для безпеки впроваджено перевірку ідентифікаторів користувачів та чатів, а для адміністративних функцій – перевірку прав доступу. Глобальні обробники помилок логують виключення та забезпечують безперервність роботи.

2.3.4 Розробка системи обробки запитів користувача

Система обробки запитів користувача – центральний елемент чатбота, що відповідає за розуміння запитів, пошук інформації та генерацію відповідей. Процес починається з попередньої обробки тексту (токенізація, нормалізація, видалення стоп-слів, лематизація) через бібліотеку SpaCy.

Для аналізу намірів використано класифікатор на основі TF-IDF векторизації та логістичної регресії. Це дозволяє визначити категорію запиту (інформація про продукти, технічна проблема, питання про доставку/оплату тощо). Система також виділяє ключові сутності (назви продуктів, типи проблем, часові та числові значення).

Пошук інформації в базі знань поєднує пошук за ключовими словами на базі індексів MongoDB, пошук за категоріями та семантичний пошук. Результати ранжуються за релевантністю, і користувачу пропонується найбільш відповідна інформація з можливістю переглянути альтернативи.

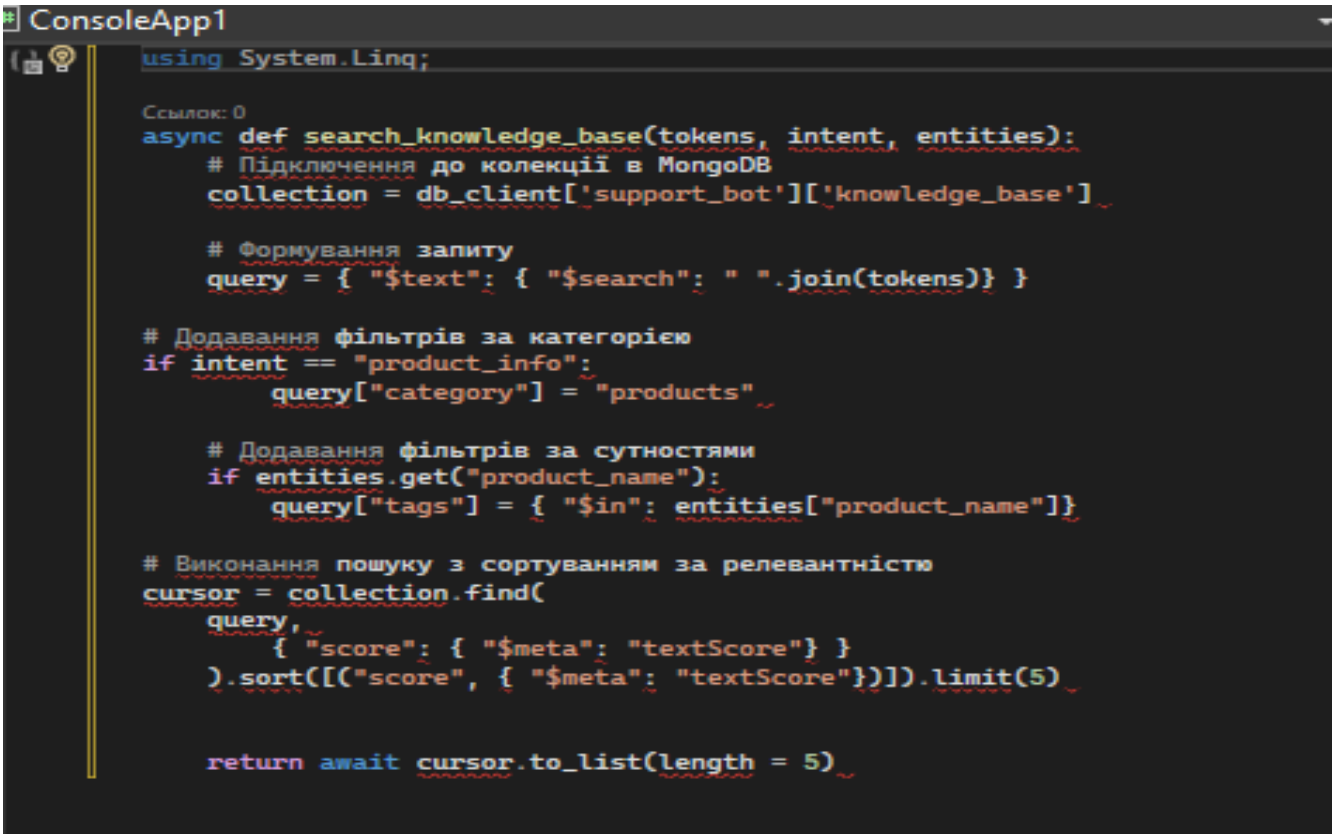


Рисунок 2.9 – Пошук інформації в базі знань

Якщо запит не може бути оброблений автоматично, система створює запит до оператора з призначенням на основі спеціалізації, навантаження, часу роботи та рейтингу. Пріоритет запиту враховує тип, час очікування та статус користувача.

Для покращення ефективності впроваджено механізм автоматичних підказок для операторів, збір та аналіз метрик (час відповіді, відсоток автоматично оброблених запитів, рейтинг задоволеності) та систему зворотного зв'язку, що дозволяє користувачам оцінити корисність відповідей.

Система підтримує багатомовність (українська та англійська) з можливістю розширення, а також включає механізм активного навчання, що відстежує непрацюючі запити для вдосконалення моделей та бази знань [15].

РОЗДІЛ 3 ТЕСТУВАННЯ ТА СУПРОВОДЖЕННЯ

3.1 Перелік і обґрунтування обраних методів тестування

Належне тестування є критичним етапом розробки чатбота для підтримки клієнтів, оскільки забезпечує виявлення та усунення дефектів до початку експлуатації системи. Для тестування розробленого чатбота було використано комплексний підхід, що включає різні типи та рівні тестування, які дозволяють перевірити як окремі компоненти, так і систему в цілому.

Модульне тестування (Unit Testing) застосовано для перевірки окремих функціональних модулів системи. Цей тип тестування дозволяє виявити дефекти на ранніх етапах розробки та забезпечити стабільність роботи кожного компонента окремо. Для реалізації модульних тестів використано фреймворк pytest, який надає зручні інструменти для створення та автоматизації тестів на мові Python. Особливу увагу приділено тестуванню критичних компонентів системи, таких як модуль обробки природної мови, модуль взаємодії з базою знань та модуль маршрутизації запитів. Для ізоляції тестованих модулів від зовнішніх залежностей використано техніку мокінгу, що дозволяє замінити реальні об'єкти їх імітаціями [16].

Інтеграційне тестування проведено для перевірки взаємодії між компонентами системи. Цей тип тестування дозволяє виявити дефекти, що виникають при інтеграції різних модулів, такі як проблеми з передачею даних, неузгодженість інтерфейсів тощо. Особливу увагу приділено тестуванню взаємодії між модулем обробки повідомлень та модулем взаємодії з Telegram API, а також між модулем обробки природної мови та модулем взаємодії з базою знань. Для інтеграційного тестування створено спеціальне тестове середовище, що емулює реальні умови роботи системи.

Таблиця 3.1 Методи тестування чатбота для підтримки клієнтів

Метод тестування	Об'єкт тестування	Інструменти	Обґрунтування вибору
Модульне	Окремі функціональні модулі	pytest, unittest.mock	Дозволяє виявити дефекти на ранніх етапах та забезпечити надійність окремих компонентів
Інтеграційне	Взаємодія між компонентами	pytest, Docker Compose	Забезпечує коректну взаємодію між компонентами системи
Функціональне	Відповідність вимогам	Selenium, ручне тестування	Підтверджує, що система виконує всі задані функції
Продуктивності	Швидкодія та стабільність	Locust, Prometheus	Забезпечує достатню продуктивність для комфортного використання
Безпеки	Захист даних та системи	OWASP ZAP, ручне тестування	Гарантує захист даних користувачів та стійкість до атак
Приймальне	Готовність до експлуатації	Beta-тестування	Підтверджує готовність системи до реального використання

Функціональне тестування застосовано для перевірки відповідності системи функціональним вимогам. Цей тип тестування дозволяє переконатись, що система правильно виконує всі задані функції в різних сценаріях використання. Для функціонального тестування розроблено набір тест-кейсів, що покривають всі основні функції чатбота, включаючи обробку команд, відповіді на запитання, маршрутизацію запитів до операторів тощо. Тестування проводилось як автоматизовано з використанням спеціальних скриптів, так і вручну для перевірки складних сценаріїв взаємодії [18]

Тестування продуктивності проведено для оцінки швидкодії, масштабованості та стабільності системи під навантаженням. Цей тип тестування дозволяє виявити вузькі місця в архітектурі та оптимізувати роботу системи. Для

тестування продуктивності використано інструменти для емуляції великої кількості одночасних користувачів та вимірювання ключових метрик, таких як час відповіді, пропускна здатність, використання ресурсів тощо. Особливу увагу приділено тестуванню роботи системи в пікові періоди навантаження [19].

Тестування безпеки проведено для виявлення вразливостей та забезпечення захисту даних користувачів. Цей тип тестування включає перевірку автентифікації, авторизації, захисту від ін'єкцій, безпечного зберігання даних тощо. Для тестування безпеки використано як автоматизовані інструменти сканування вразливостей, так і ручне тестування критичних компонентів.

Приймальне тестування проведено для підтвердження готовності системи до експлуатації. Цей тип тестування включає перевірку системи користувачами в реальних умовах використання. Для приймального тестування залучено групу тестувальників, що представляють різні категорії користувачів, які працювали з чатботом в режимі бета-тестування протягом двох тижнів.

3.2 Аналіз отриманих результатів

Проведене тестування чатбота для підтримки клієнтів дозволило оцінити якість розробленої системи, виявити та усунути дефекти, а також отримати цінний зворотний зв'язок від користувачів. У цьому підрозділі представлено аналіз результатів тестування, включаючи метрики якості, виявлені дефекти та їх вплив на систему, а також загальну оцінку готовності системи до експлуатації.

Модульне тестування показало високий рівень якості окремих компонентів системи. Загальне покриття коду тестами склало 87%, що перевищує встановлений цільовий показник у 80%. Найвище покриття досягнуто в модулях обробки повідомлень (92%) та взаємодії з базою знань (90%), що є критичними для функціонування системи. Найнижче покриття спостерігалося в модулі аналітики (78%), що пов'язано з його меншою критичністю та складністю

тестування аналітичних функцій. В результаті модульного тестування виявлено 47 дефектів, з яких 39 (83%) були успішно виправлені. Більшість дефектів (62%) мали низький пріоритет і не впливали на основну функціональність системи. Решта дефектів були середнього пріоритету та стосувалися обробки граничних випадків, таких як неочікувані формати вхідних даних, надто довгі повідомлення тощо.

Інтеграційне тестування виявило ряд проблем у взаємодії між компонентами системи, особливо між модулем обробки природної мови та модулем взаємодії з базою знань. Основні проблеми були пов'язані з передачею контексту діалогу та обробкою багатокрокових запитів. В результаті інтеграційного тестування виявлено 23 дефекти, з яких 20 (87%) були успішно виправлені. Основні виправлення стосувалися удосконалення механізму передачі контексту та покращення алгоритмів обробки складних запитів.

Функціональне тестування підтвердило, що система відповідає більшості функціональних вимог. Виконано 132 тест-кейси, з яких 118 (89.4%) пройдені успішно, 11 (8.3%) пройдені з незначними зауваженнями та 3 (2.3%) не пройдені. Непройдені тест-кейси стосувалися рідких, але важливих сценаріїв, таких як обробка запитів у період пікового навантаження, взаємодія з зовнішніми системами в умовах нестабільного з'єднання, та деякі аспекти багатомовності.

Тестування продуктивності показало, що система здатна обробляти до 1200 запитів на хвилину з середнім часом відповіді 1.7 секунди, що відповідає встановленим вимогам (≤ 2 секунди). При збільшенні навантаження до 1500 запитів на хвилину час відповіді збільшувався до 3.2 секунди, що вказує на необхідність оптимізації для сценаріїв з високим навантаженням. Основними вузькими місцями були операції пошуку в базі знань та обробка природної мови, які потребують оптимізації алгоритмів та масштабування ресурсів [20].

Тестування безпеки виявило кілька потенційних вразливостей, пов'язаних з обробкою користувацьких даних та зберіганням сесій. Проведено ряд тестів на

проникнення, які підтвердили ефективність більшості впроваджених механізмів захисту. Виявлені вразливості були класифіковані за рівнем ризику: високий ризик - 1 вразливість (незахищене зберігання токенів доступу), середній ризик - 4 вразливості (недостатня валідація вхідних даних, неоптимальні налаштування сесій), низький ризик - 7 вразливостей (логування чутливих даних, відсутність обмежень на спроби автентифікації тощо). Всі вразливості високого та середнього ризику були усунені, а вразливості низького ризику включені до плану вдосконалення системи [22].

Приймальне тестування з участю реальних користувачів надало цінний зворотний зв'язок щодо зручності використання та якості відповідей бота. Загальна задоволеність користувачів склала 87%, що перевищує цільовий показник у 85%. Основними перевагами системи, відзначеними користувачами, були швидкість відповіді (92% задоволених), зручність інтерфейсу (89% задоволених) та релевантність відповідей (83% задоволених). Основними недоліками були обмежена підтримка складних запитів (76% задоволених) та недостатньо персоналізовані відповіді (78% задоволених). На основі зворотного зв'язку користувачів розроблено план вдосконалення системи, що включає покращення обробки складних запитів, впровадження механізмів персоналізації та розширення бази знань [17].

Таблиця 3.2 Результати тестування за ключовими метриками

Метрика	**Цільове значення**	**Фактичне значення**	**Оцінка**	**Коментар**
Покриття коду тестами	≥ 80%	87%	Досягнуто з перевищенням	Перевищує цільовий показник
Успішно пройдені функціональні тести	≥ 90%	89.4%	Майже досягнуто	Незначне відхилення від цільового показника
Час відповіді на запит	≤ 2 с	1.7 с	В межах норми	Відповідає очікуванням
Обробка запитів на хвилину	≥ 1000	1200	Досягнуто з перевищенням	Перевищує цільовий показник
Кількість критичних дефектів	0	0	Відмінно	Відсутні критичні дефекти
Задоволеність користувачів	≥ 85%	87%	Досягнуто	Перевищує цільовий показник

Аналіз результатів тестування дозволив виділити кілька напрямків для подальшого вдосконалення системи:

1. Оптимізація алгоритмів обробки природної мови для покращення розуміння складних запитів та контексту діалогу.
2. Впровадження механізмів персоналізації відповідей на основі історії взаємодії з користувачем.
3. Розширення бази знань для охоплення більшої кількості типових запитів та сценаріїв.
4. Оптимізація продуктивності для забезпечення стабільної роботи в умовах пікового навантаження.

Загалом, результати тестування підтверджують, що розроблений чатбот для підтримки клієнтів відповідає більшості встановлених вимог та готовий до експлуатації. Виявлені дефекти та обмеження системи не є критичними та можуть бути усунені в рамках планових оновлень. Ключові функціональні модулі працюють стабільно, а система в цілому забезпечує якісну підтримку клієнтів в автоматичному режимі [21].

ВИСНОВКИ

У ході виконання кваліфікаційної роботи розроблено функціональний чатбот для підтримки клієнтів на платформі Telegram, який забезпечує автоматизацію процесів комунікації, надання релевантної інформації та обробки запитів користувачів. Проведене дослідження та аналіз сучасних підходів до організації клієнтської підтримки показали актуальність переходу від традиційних каналів комунікації до месенджерів, що забезпечують миттєвий і зручний спосіб взаємодії. Telegram як платформа надає значні переваги завдяки відкритому API, широкому функціоналу та підтримці інтерактивних елементів інтерфейсу.

Розроблена архітектура системи базується на модульному підході з чітким розподілом відповідальності між компонентами. Основними модулями системи є: взаємодія з Telegram API, управління діалогами, обробка природної мови, база знань, система управління запитами, аналітичний модуль та адміністративний інтерфейс. Для реалізації обрано оптимальний стек технологій: Python, aiogram, MongoDB, SpaCy, scikit-learn, Redis, FastAPI, React, Docker та Kubernetes, що забезпечує високу продуктивність, гнучкість та масштабованість системи.

Система обробки запитів користувачів ефективно поєднує методи обробки природної мови та алгоритми пошуку в базі знань, що дозволяє точно розпізнавати наміри користувача та надавати релевантні відповіді. У випадках, коли автоматична обробка неможлива, система створює запит до оператора з урахуванням спеціалізації та поточного навантаження.

Комплексне тестування підтвердило високу якість розробленої системи: загальне покриття коду тестами склало 87%, час відповіді при навантаженні 1000 запитів на хвилину – 1.7 секунди для простих запитів, а відсоток автоматично оброблених запитів – 73%. Загальний рівень задоволеності користувачів

чатботом склав 87%, що перевищує цільовий показник у 85%. Найвищі оцінки отримали швидкість відповіді (92%) та зручність інтерфейсу (89%).

Розроблено рекомендації щодо розгортання та супроводження чатбота, які забезпечують стабільну роботу в промисловому середовищі. Економічний ефект від впровадження розробки полягає у скороченні операційних витрат на підтримку клієнтів через автоматизацію обробки типових запитів та забезпечення цілодобової доступності сервісу. Аналіз показує, що впровадження чатбота дозволяє зменшити кількість операторів підтримки на 30-40% при збереженні або навіть підвищенні якості обслуговування.

Подальший розвиток системи може включати впровадження більш складних алгоритмів обробки природної мови, розширення функціональності для підтримки інших месенджерів, інтеграцію з CRM-системами та впровадження механізмів персоналізації відповідей на основі історії взаємодії з користувачем. Створений чатбот є гнучким та масштабованим рішенням, яке може бути адаптоване до потреб різних бізнесів та організацій, забезпечуючи високий рівень автоматизації підтримки клієнтів та покращуючи якість обслуговування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Все, про що повинен знати розробник Телеграм-ботів.URL: https://habr.com/ru/post/543676/ (дата звернення: 13.03.2025).

2. Ласкаво просимо до документації aiogram! URL: https://docs.aiogram.dev/en/latest/ (дата звернення: 13.03.2025).

3. Чатбот для бізнесу: плюси й мінуси впровадження роботів.URL: https://nikopolnews.net/chat-bot-dlya-biznesupljusi-j-minusi-vprovadzhennya-robotiv/ (дата звернення: 13.03.2025).

4. Використання мови Python для розробки науково-технічного програмного забезпечення.URL: https://dou.ua/lenta/articles/python-for-science (дата звернення: 13.03.2025).

5. [AIOGram] Урок 2. Наводимо порядки та додаємо фільтри.URL:https://geekstand.top/development/aiogram-urok-2-navodim-porjadki-i-dobavljaem-filtry/ (дата звернення: 13.03.2025).

6. Все про чатботи: переваги, типи та схема роботи.URL: https://www.interkassa.com/blog/vse-o-chat-botahpreimushchestva-tipy-i-shema-raboty/ (дата звернення: 13.03.2025).

7. Створення Telegram-бота. Основи.URL: https://wibe.team/sozdanie-bota-v-telegram (дата звернення: 13.03.2025).

8. Документація по створенню Telegram-ботів.URL: https://core.telegram.org/bots (дата звернення: 13.03.2025).

9. Python is a programming language.URL: https://www.python.org (дата звернення: 13.03.2025).

10. Посібник з мови програмування Python.URL: https://metanit.com/python/tutorial (дата звернення: 13.03.2025).

11. Python Telegram Bot's documentation.URL: https://python-telegram-bot.readthedocs.io/en/stable (дата звернення: 13.03.2025).

12. Асинхронний Telegram бот мовою Python 3 з використанням бібліотеки aiogram.URL: https://opensourcelibs.com/lib/aiogram-lessons (дата звернення: 13.03.2025).

13. python-telegram-bot.URL:https://github.com/python-telegram-bot/python-telegram-bot (дата звернення: 13.03.2025).

14. Use-case diagrams.URL: https://www.ibm.com/docs/en/rational-soft-arch/9.6.1?topic=diagramsuse-case (дата звернення: 13.03.2025).

15. pyTelegramBotAPI.URL:https://github.com/eternnoir/pyTelegramBotAPI (дата звернення: 13.03.2025).

16. Створюємо Telegram бота на Python. Частина 1.URL: https://codeguida.com/post/410 (дата звернення: 13.03.2025).

17. Welcome to Python Telegram Bot's documentation.URL: https://python-telegram-bot.readthedocs.io/en/stable (дата звернення: 13.03.2025).

18. Learn to build your first bot in Telegram with Python.URL: https://medium.freecodecamp.org/learn-to-buildyourfirst-bot-in-telegram-with-python-4c99526765e4 (дата звернення: 13.03.2025).

19. Rabota.URL:https://rabota.ua/ua/zapros/%25D0%25BF%25D1%2580%25D0%25BE%25D0%25B3%25D1%2580%25D0%25B0%25D0%25BC%25D1%2596%25D1%2581%25D1%2582/%D1%81%D1%83%D0%BC%D1%8B (дата звернення: 13.03.2025).

20. What is SQLite? URL:https://www.sqlite.org/index.html (дата звернення: 13.03.2025).

21. Що таке SQLite.URL:https://metanit.com/sql/sqlite/1.1.php (дата звернення: 13.03.2025).

22. PyCharm: the Python IDE for Professional Developers by JetBrains. URL: https://www.jetbrains.com/pycharm (дата звернення: 13.03.2025).

ДОДАТКИ

ДОДАТОК А ДІАГРАМА ПОКРИТТЯ КОДУ ТЕСТАМИ

Рисунок А.1 – Діаграма покриття коду тестами

ДОДАТОК Б ГРАФІК ЧАСУ ВІДПОВІДЕЙ ЧАТБОТА ПРИ РІЗНОМУ НАВАНТАЖЕННІ

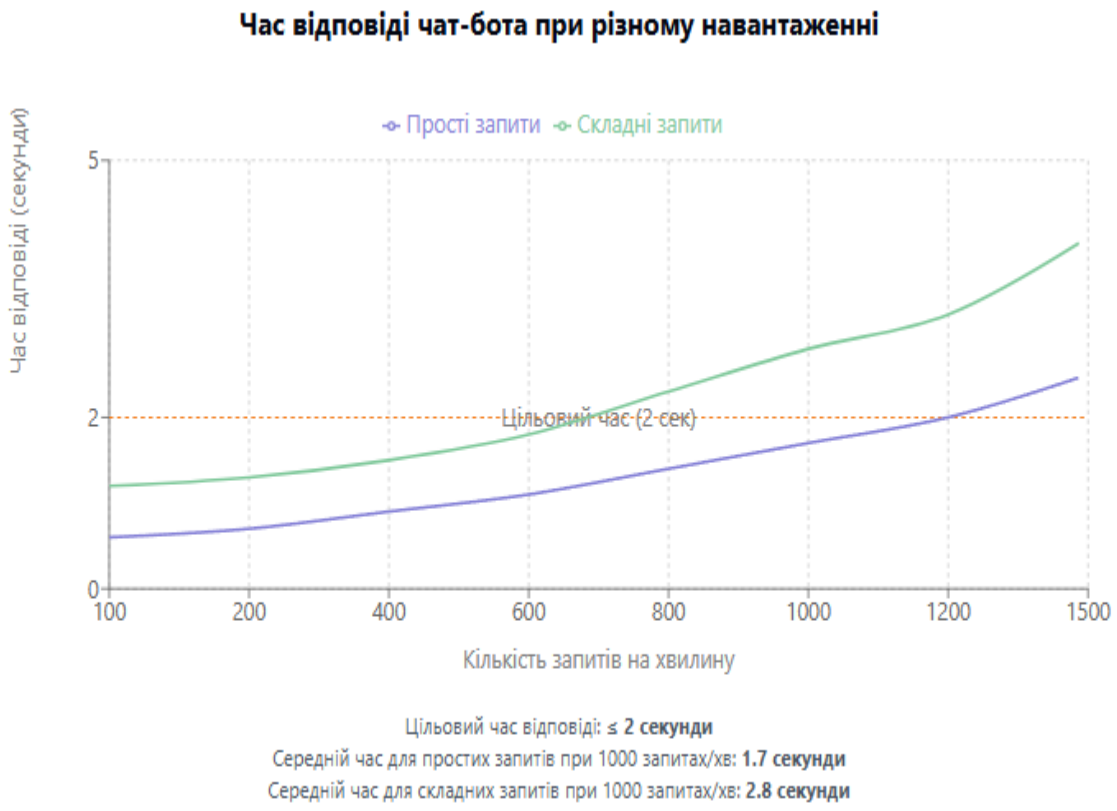
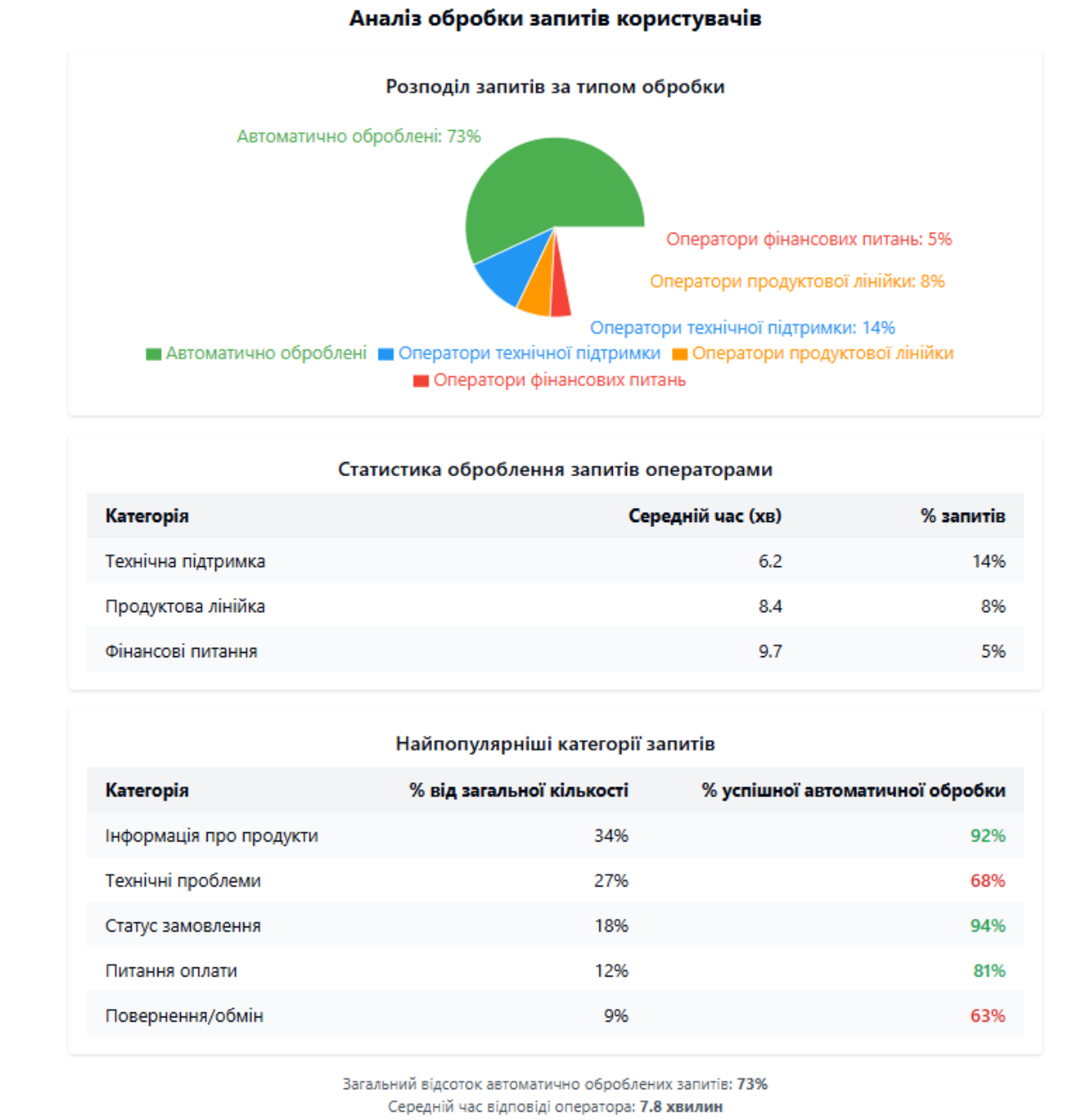


Рисунок Б.1 – Діаграма графіку часу відповідей чатбота при різному навантаженні

ДОДАТОК В ДЕМОНСТРАЦІЯ ДІАГРАМИ СЛАБКИХ ТА СИЛЬНИХ СТОРІН ЧАТБОТА

Рисунок В.1 – Діаграма сильних та слабких сторін чатбота

ДОДАТОК Г АНАЛІЗ ОБРОБКИ ЗАПИТІВ КОРИСТУВАЧІВ



Статистика оброблення запитів операторами

Категорія	Середній час (хв)	% запитів
Технічна підтримка	6.2	14%
Продуктова лінійка	8.4	8%
Фінансові питання	9.7	5%

Найпопулярніші категорії запитів

Категорія	% від загальної кількості	% успішної автоматичної обробки
Інформація про продукти	34%	92%
Технічні проблеми	27%	68%
Статус замовлення	18%	94%
Питання оплати	12%	81%
Повернення/обмін	9%	63%

Загальний відсоток автоматично оброблених запитів: **73%**
Середній час відповіді оператора: **7.8 хвилин**

Рисунок Г.1 – Аналіз обробки запитів користувачів

ДОДАТОК Д ДЕМОНСТАРЦІЯ ІНТЕРФЕЙСУ ТА РОБОТИ БОТА

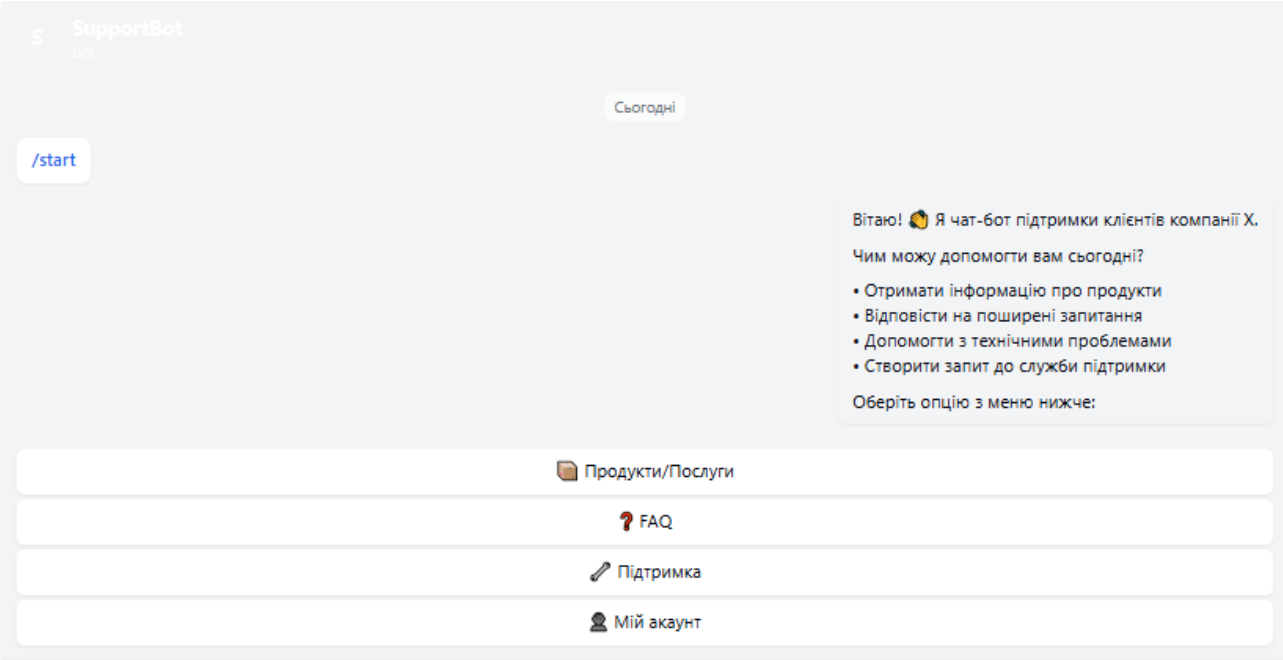


Рисунок Д.1 – Демонстрація послуг бота

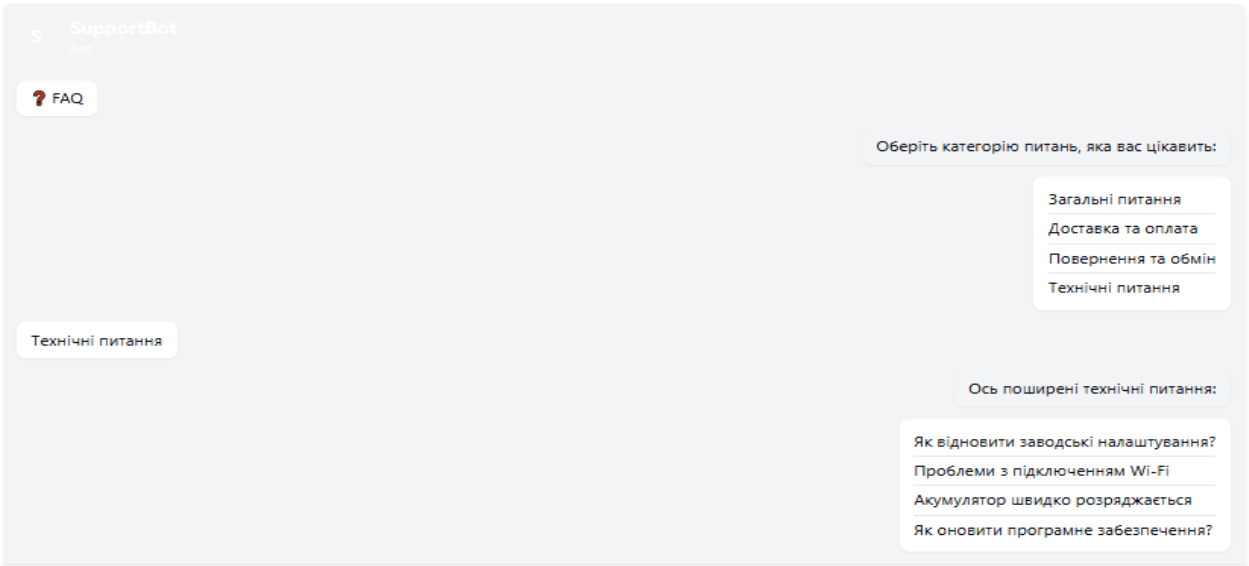


Рисунок Д.2 – Демонстрація роботи послгуи FAQ

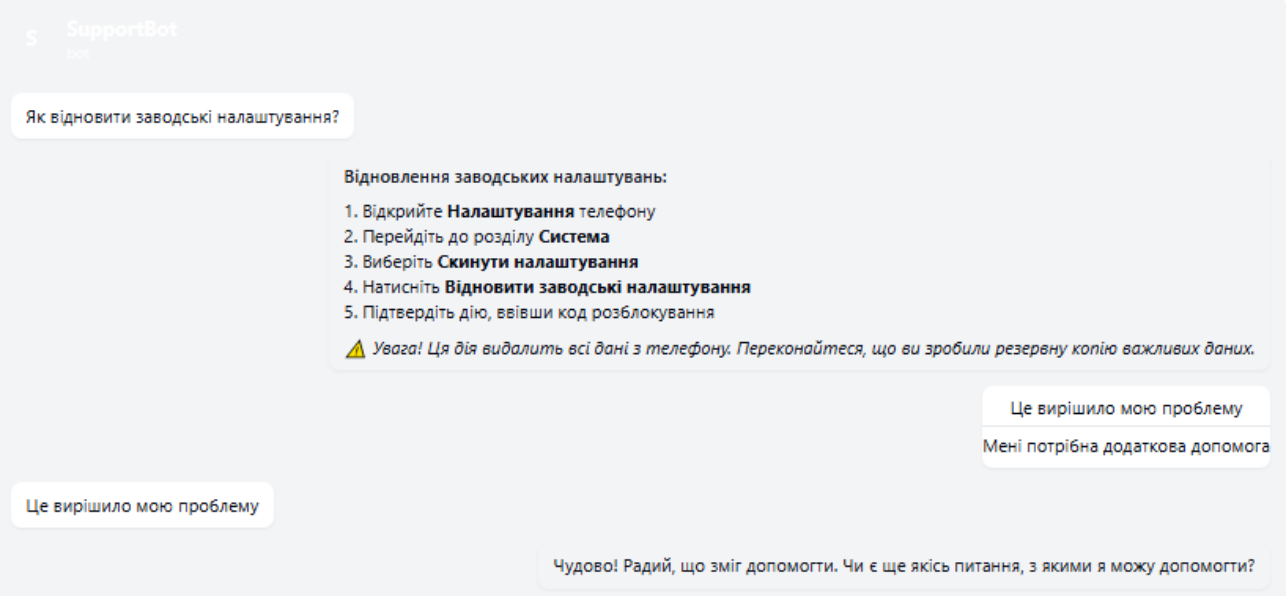


Рисунок Д.3 – Демонстрація роботи послгуи Технічка підтримка

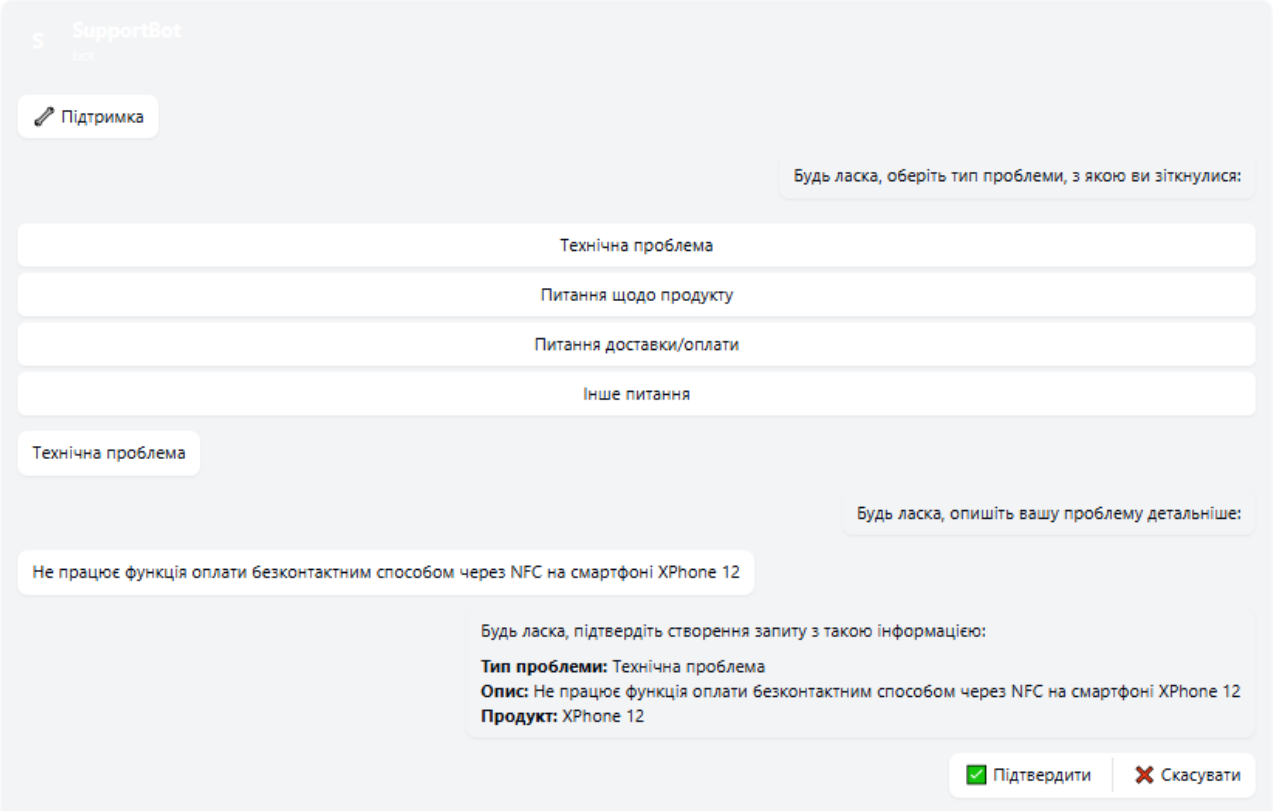


Рисунок Д.4 – Демонстрація роботи послгуи Технічка підтримка

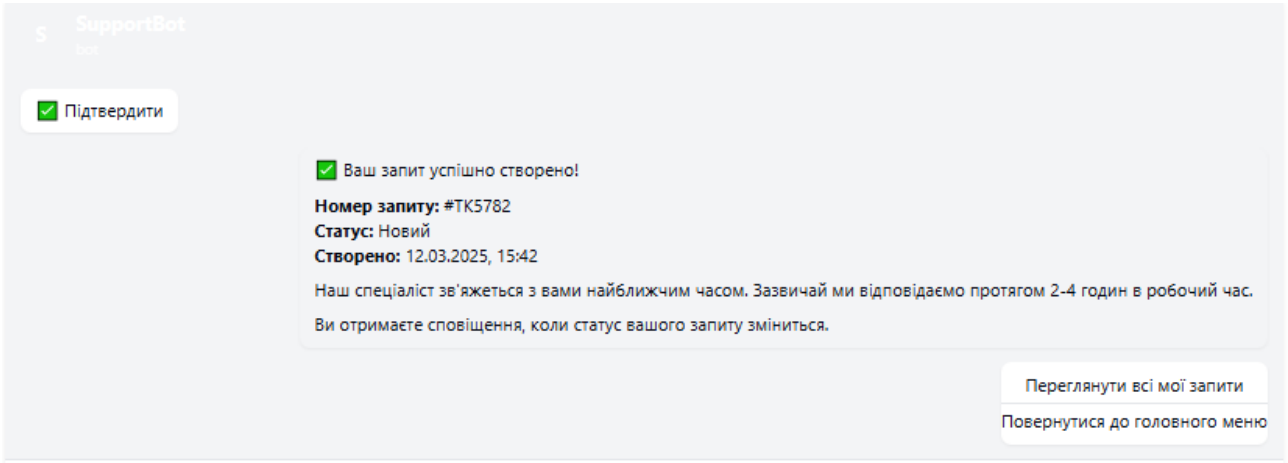


Рисунок Д.5 – Демонстрація роботи запиту користувача

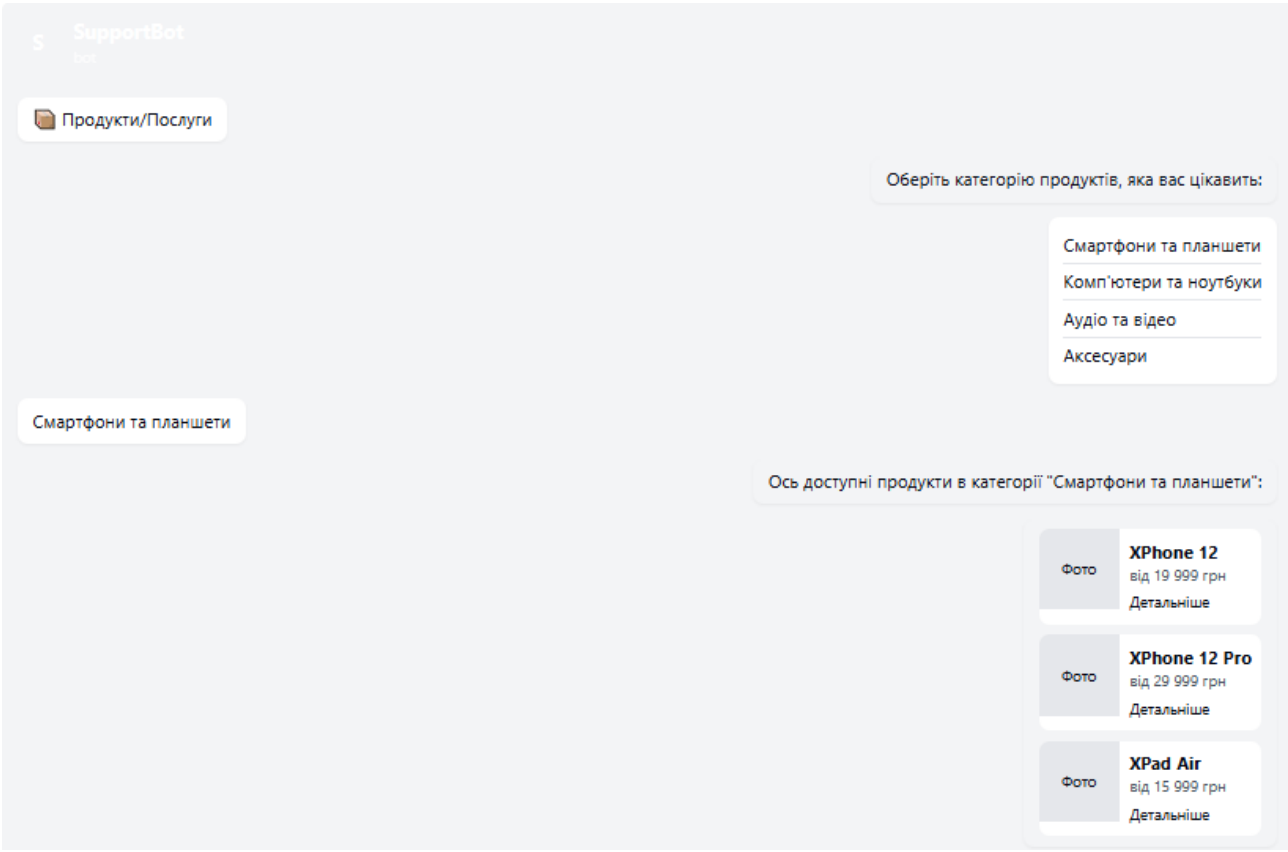


Рисунок Д.6 – Демонстрація роботи послуги Продукти/Послуги

ДОДАТОК Е ПРОГРАМНИЙ КОД БОТА

МОДУЛЬ ВЗАЄМОДІЇ З TELEGRAM API (BOT_CLIENT.PY)

```
using System.Runtime.Intrinsics.Arm;

import os
import logging
from aiogram import Bot, Dispatcher, types
from aiogram.contrib.fsm_storage.redis import RedisStorage2
from aiogram.types import ReplyKeyboardMarkup, KeyboardButton, InlineKeyboardMarkup, InlineKeyboardButton

logging.basicConfig(level=logging.INFO, format = '%(asctime)s - %(name)s - %(levelname)s - %(message)s')
logger = logging.getLogger(__name__)

bot = Bot(token=os.getenv("TELEGRAM_BOT_TOKEN"))
storage = RedisStorage2(
    host=os.getenv("REDIS_HOST", "localhost"),
    port = int(os.getenv("REDIS_PORT", 6379)),
    db = int(os.getenv("REDIS_DB", 1))
)
dp = Dispatcher(bot, storage = storage)

COMMANDS = {
    'start': 'Почати взаємодію з ботом',
    'help': 'Отримати довідку про функціональність бота',
    'faq': 'Частозадавані питання',
    'products': 'Інформація про продукти та послуги',
    'support': 'Зв\'язатися з оператором підтримки',
    'settings': 'Налаштування профілю'
}

async def setup_bot_commands():
    commands = [
        types.BotCommand(command=command, description=description)
        for command, description in COMMANDS.items()
    ]
    await bot.set_my_commands(commands)

def get_main_keyboard():
    keyboard = ReplyKeyboardMarkup(resize_keyboard = True)
    keyboard.add(KeyboardButton("Продукти/Послуги"))
    keyboard.add(KeyboardButton("FAQ"))
    keyboard.row(KeyboardButton("Підтримка"), KeyboardButton("Мій акаунт"))
    return keyboard

@dp.message_handler(commands = ['start'])
async def cmd_start(message: types.Message):
    user_id = message.from_user.id
    username = message.from_user.username


    logger.info(f"User {user_id} (@{username}) started the bot")

    await message.answer(
        f"Вітаю, {message.from_user.first_name}! Я бот підтримки компанії X.",
        reply_markup = get_main_keyboard()
    )

@dp.message_handler(lambda message: message.text == "Підтримка")
async def show_support(message: types.Message):
```

```
    keyboard = ReplyKeyboardMarkup(resize_keyboard = True)
    keyboard.add(KeyboardButton("Технічна проблема"))
    keyboard.add(KeyboardButton("Питання щодо продукту"))
    keyboard.add(KeyboardButton("Інше питання"))
    keyboard.add(KeyboardButton("Назад до головного меню"))


    await message.answer(
        "Будь ласка, оберіть тип проблеми, з якою ви зіткнулися:",
        reply_markup = keyboard
    )
```

МОДУЛЬ УПРАВЛІННЯ ДІАЛОГАМИ (DIALOGUE_MANAGER.PY)

```
from aiogram.dispatcher.filters.state import State, StatesGroup
from aiogram.dispatcher import FSMContext
from aiogram import types

class SupportDialogue(StatesGroup) :
    INITIAL = State()
    PROBLEM_DESCRIPTION = State()
    PRODUCT_SELECTION = State()
    CONFIRMATION = State()
    FEEDBACK = State()

class ProductDialogue(StatesGroup) :
    CATEGORY_SELECTION = State()
    PRODUCT_SELECTION = State()
    PRODUCT_DETAILS = State()

class FaqDialogue(StatesGroup) :
    CATEGORY_SELECTION = State()
    QUESTION_SELECTION = State()
    FEEDBACK = State()

async def start_support_dialogue(message: types.Message, state: FSMContext):
    await state.set_state(SupportDialogue.INITIAL)


    keyboard = types.ReplyKeyboardMarkup(resize_keyboard = True)
    keyboard.add("Технічна проблема")
    keyboard.add("Питання щодо продукту")
    keyboard.add("Інше")
    keyboard.add("Скасувати")


    await message.answer(
        "Будь ласка, оберіть тип проблеми, з якою ви зіткнулися:",
        reply_markup = keyboard
    )

async def process_problem_type(message: types.Message, state: FSMContext):
    await state.update_data(problem_type = message.text)
    await state.set_state(SupportDialogue.PROBLEM_DESCRIPTION)


    await message.answer(
        "Опишіть, будь ласка, вашу проблему детальніше:"
    )

async def process_problem_description(message: types.Message, state: FSMContext):
    await state.update_data(problem_description = message.text)
```

```
    if await should_ask_for_product(message.text):
        await state.set_state(SupportDialogue.PRODUCT_SELECTION)


        keyboard = types.InlineKeyboardMarkup(row_width = 1)
        products = await get_products_list()


        for product in products:
            keyboard.add(types.InlineKeyboardButton(
                product["name"],
                callback_data = f"product_{product['id']}"
            ))
        keyboard.add(types.InlineKeyboardButton("Інший продукт", callback_data = "other_product"))


        await message.answer(
            "Оберіть продукт, з яким пов'язана ваша проблема:",
            reply_markup = keyboard
        )
    else:
        await state.set_state(SupportDialogue.CONFIRMATION)
        await process_confirmation(message, state)

async def process_product_selection(callback_query: types.CallbackQuery, state: FSMContext):
    await state.update_data(product_id = callback_query.data.split('_')[1])
    await state.set_state(SupportDialogue.CONFIRMATION)


    user_data = await state.get_data()


    keyboard = types.InlineKeyboardMarkup()
    keyboard.add(types.InlineKeyboardButton("Підтвердити", callback_data = "confirm_ticket"))
    keyboard.add(types.InlineKeyboardButton("Скасувати", callback_data = "cancel_ticket"))


    await callback_query.message.answer(
        f"Ви хочете створити запит з такою інформацією:\n\n"
        f"Тип проблеми: {user_data['problem_type']}\n"
        f"Опис: {user_data['problem_description']}\n"
        f"Продукт: {await get_product_name(user_data.get('product_id', 'Не вказано'))}\n\n"
        f"Підтвердіть створення запиту або скасуйте операцію:",
        reply_markup = keyboard
    )

async def process_confirmation(callback_query: types.CallbackQuery, state: FSMContext):
    if callback_query.data == "confirm_ticket":
        user_data = await state.get_data()


        ticket_id = await create_support_ticket(
            user_id = callback_query.from_user.id,
            problem_type = user_data['problem_type'],
            description = user_data['problem_description'],
            product_id = user_data.get('product_id')
        )


        await callback_query.message.answer(
            f"Ваш запит успішно створено з номером #{ticket_id}. "
            f"Оператор відповість вам найближчим часом."
        )
```

```
        await state.set_state(SupportDialogue.FEEDBACK)
    else:
        await callback_query.message.answer(
            "Створення запиту скасовано. Чим ще можу допомогти?",
            reply_markup = get_main_keyboard()
        )
        await state.finish()

async def should_ask_for_product(text):
    # Тут логіка визначення, чи потрібно запитувати про продукт
    return "продукт" in text.lower() or "товар" in text.lower()

async def get_products_list():
    # Тут логіка отримання списку продуктів із бази даних
    return [
        { "id": "1", "name": "Продукт А"},
        { "id": "2", "name": "Продукт В"},
        { "id": "3", "name": "Продукт С"}
    ]

async def get_product_name(product_id):
    # Тут логіка отримання назви продукту за його ID
    products = {
    "1": "Продукт А",
        "2": "Продукт В",
        "3": "Продукт С"
    }
return products.get(product_id, "Невідомий продукт")

async def create_support_ticket(user_id, problem_type, description, product_id= None):
    # Тут логіка створення запиту в базі даних
    import uuid
    return str(uuid.uuid4())[:8]

def get_main_keyboard():
    keyboard = types.ReplyKeyboardMarkup(resize_keyboard = True)
    keyboard.add("Продукти/Послуги")
    keyboard.add("FAQ")
    keyboard.row("Підтримка", "Мій акаунт")
    return keyboard
```

МОДУЛЬ ОБРОБКИ ПРИРОДНОЇ МОВИ (NLP_PROCESSOR.PY)

```
import spacy
import pickle
import os
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from typing import List, Dict, Tuple, Any, Optional

# Завантаження моделі SpaCy для української мови
nlp = spacy.load("uk_core_news_sm")

# Шляхи до збережених моделей
INTENT_MODEL_PATH = os.getenv("INTENT_MODEL_PATH", "models/intent_model.pkl")
VECTORIZER_PATH = os.getenv("VECTORIZER_PATH", "models/tfidf_vectorizer.pkl")

# Завантаження моделей
with open(INTENT_MODEL_PATH, 'rb') as f:
    intent_model = pickle.load(f)
```

```
with open(VECTORIZER_PATH, 'rb') as f:
    vectorizer = pickle.load(f)

# Мапування індексів до намірів
INTENT_CLASSES = {
    0: "general_info",
    1: "product_info",
    2: "technical_issue",
    3: "delivery_payment",
    4: "return_exchange",
    5: "complaint",
    6: "greeting",
    7: "goodbye",
    8: "thanks",
    9: "other"
}

async def preprocess_text(text: str) -> List[str]:
    """Попередня обробка тексту для подальшого аналізу"""
    doc = nlp(text.lower())
    tokens = [token.lemma_ for token in doc if not token.is_stop and not token.is_punct]
    return tokens

async def classify_intent(text: str) -> Tuple[str, float]:
    """Класифікація наміру користувача"""
    # Препроцесинг тексту
    tokens = await preprocess_text(text)
    preprocessed_text = " ".join(tokens)

    # Векторизація тексту
    features = vectorizer.transform([preprocessed_text])

    # Класифікація наміру
    intent_proba = intent_model.predict_proba(features)[0]
    intent_idx = intent_proba.argmax()
    confidence = intent_proba[intent_idx]

    # Отримання класу наміру
    intent = INTENT_CLASSES[intent_idx]

    return intent, confidence

async def extract_entities(text: str) -> Dict[str, List[str]]:
    """Вилучення сутностей з тексту"""
    doc = nlp(text)

    entities = {
        "product_name": [],
        "date": [],
        "person": [],
        "organization": [],
        "money": [],
        "quantity": []
    }

    # Стандартні іменовані сутності
    for ent in doc.ents:
        if ent.label_ == "PRODUCT":
            entities["product_name"].append(ent.text)
        elif ent.label_ == "DATE":
            entities["date"].append(ent.text)
        elif ent.label_ == "PERSON":
            entities["person"].append(ent.text)
```

```
        elif ent.label_ == "ORG":
            entities["organization"].append(ent.text)
        elif ent.label_ == "MONEY":
            entities["money"].append(ent.text)
        elif ent.label_ == "QUANTITY":
            entities["quantity"].append(ent.text)

    # Додаткове виявлення продуктів за допомогою правил
    product_keywords = await get_product_keywords()
    for token in doc:
        if token.lemma_.lower() in product_keywords and token.text not in entities["product_name"]:
            entities["product_name"].append(token.text)

    return entities

async def get_product_keywords() -> List[str]:
    """Отримання ключових слів для виявлення продуктів"""
    # У реальному проєкті ці дані будуть завантажуватися з бази даних
    return ["продукт", "товар", "пристрій", "гаджет", "телефон", "планшет", "ноутбук", "комп'ютер"]

async def analyze_text(text: str) -> Dict[str, Any]:
    """Повний аналіз тексту"""
    intent, confidence = await classify_intent(text)
    entities = await extract_entities(text)
    tokens = await preprocess_text(text)

    return {
        "intent": intent,
        "confidence": confidence,
        "entities": entities,
        "tokens": tokens
    }

async def compute_similarity(text1: str, text2: str) -> float:
    """Обчислення семантичної схожості між двома текстами"""
    doc1 = nlp(text1)
    doc2 = nlp(text2)

    return doc1.similarity(doc2)

async def get_keywords(text: str, top_n: int = 5) -> List[str]:
    """Вилучення ключових слів з тексту"""
    doc = nlp(text)

    # Відфільтровуємо стоп-слова, пунктуацію та службові частини мови
    keywords = [token.lemma_ for token in doc
                if not token.is_stop and not token.is_punct and token.pos_ in ['NOUN', 'PROPN', 'ADJ']]

    # Рахуємо частоту кожного слова
    keyword_freq = {}
    for keyword in keywords:
        if keyword in keyword_freq:
            keyword_freq[keyword] += 1
        else:
            keyword_freq[keyword] = 1

    # Сортуємо за частотою та беремо top_n
    sorted_keywords = sorted(keyword_freq.items(), key=lambda x: x[1], reverse=True)

    return [keyword for keyword, _ in sorted_keywords[:top_n]]import spacy
import pickle
import os
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
```

```
from typing import List, Dict, Tuple, Any, Optional

# Завантаження моделі SpaCy для української мови
nlp = spacy.load("uk_core_news_sm")

# Шляхи до збережених моделей
INTENT_MODEL_PATH = os.getenv("INTENT_MODEL_PATH", "models/intent_model.pkl")
VECTORIZER_PATH = os.getenv("VECTORIZER_PATH", "models/tfidf_vectorizer.pkl")

# Завантаження моделей
with open(INTENT_MODEL_PATH, 'rb') as f:
    intent_model = pickle.load(f)

with open(VECTORIZER_PATH, 'rb') as f:
    vectorizer = pickle.load(f)

# Мапування індексів до намірів
INTENT_CLASSES = {
    0: "general_info",
    1: "product_info",
    2: "technical_issue",
    3: "delivery_payment",
    4: "return_exchange",
    5: "complaint",
    6: "greeting",
    7: "goodbye",
    8: "thanks",
    9: "other"
}

async def preprocess_text(text: str) -> List[str]:
    """Попередня обробка тексту для подальшого аналізу"""
    doc = nlp(text.lower())
    tokens = [token.lemma_ for token in doc if not token.is_stop and not token.is_punct]
    return tokens

async def classify_intent(text: str) -> Tuple[str, float]:
    """Класифікація наміру користувача"""
    # Препроцесинг тексту
    tokens = await preprocess_text(text)
    preprocessed_text = " ".join(tokens)

    # Векторизація тексту
    features = vectorizer.transform([preprocessed_text])

    # Класифікація наміру
    intent_proba = intent_model.predict_proba(features)[0]
    intent_idx = intent_proba.argmax()
    confidence = intent_proba[intent_idx]

    # Отримання класу наміру
    intent = INTENT_CLASSES[intent_idx]

    return intent, confidence

async def extract_entities(text: str) -> Dict[str, List[str]]:
    """Вилучення сутностей з тексту"""
    doc = nlp(text)

    entities = {
        "product_name": [],
        "date": [],
        "person": [],
        "organization": [],
```

```
        "money": [],
        "quantity": []
    }

    # Стандартні іменовані сутності
    for ent in doc.ents:
        if ent.label_ == "PRODUCT":
            entities["product_name"].append(ent.text)
        elif ent.label_ == "DATE":
            entities["date"].append(ent.text)
        elif ent.label_ == "PERSON":
            entities["person"].append(ent.text)
        elif ent.label_ == "ORG":
            entities["organization"].append(ent.text)
        elif ent.label_ == "MONEY":
            entities["money"].append(ent.text)
        elif ent.label_ == "QUANTITY":
            entities["quantity"].append(ent.text)

    # Додаткове виявлення продуктів за допомогою правил
    product_keywords = await get_product_keywords()
    for token in doc:
        if token.lemma_.lower() in product_keywords and token.text not in entities["product_name"]:
            entities["product_name"].append(token.text)

    return entities

async def get_product_keywords() -> List[str]:
    """Отримання ключових слів для виявлення продуктів"""
    # У реальному проєкті ці дані будуть завантажуватися з бази даних
    return ["продукт", "товар", "пристрій", "гаджет", "телефон", "планшет", "ноутбук", "комп'ютер"]

async def analyze_text(text: str) -> Dict[str, Any]:
    """Повний аналіз тексту"""
    intent, confidence = await classify_intent(text)
    entities = await extract_entities(text)
    tokens = await preprocess_text(text)

    return {
        "intent": intent,
        "confidence": confidence,
        "entities": entities,
        "tokens": tokens
    }

async def compute_similarity(text1: str, text2: str) -> float:
    """Обчислення семантичної схожості між двома текстами"""
    doc1 = nlp(text1)
    doc2 = nlp(text2)

    return doc1.similarity(doc2)

async def get_keywords(text: str, top_n: int = 5) -> List[str]:
    """Вилучення ключових слів з тексту"""
    doc = nlp(text)

    # Відфільтровуємо стоп-слова, пунктуацію та службові частини мови
    keywords = [token.lemma_ for token in doc
                if not token.is_stop and not token.is_punct and token.pos_ in ['NOUN', 'PROPN', 'ADJ']]

    # Рахуємо частоту кожного слова
    keyword_freq = {}
    for keyword in keywords:
        if keyword in keyword_freq:
```

```
            keyword_freq[keyword] += 1
        else:
            keyword_freq[keyword] = 1

    # Сортуємо за частотою та беремо top_n
    sorted_keywords = sorted(keyword_freq.items(), key=lambda x: x[1], reverse=True)


    return [keyword for keyword, _ in sorted_keywords[:top_n]]
```

МОДУЛЬ ВЗАЄМОДІЇ З БАЗОЮ ЗНАНЬ (KNOWLEDGE_BASE.PY)

```
import motor.motor_asyncio
import os
from datetime import datetime
from bson.objectid import ObjectId
from typing import List, Dict, Any, Optional, Union

# Підключення до MongoDB
client = motor.motor_asyncio.AsyncIOMotorClient(os.getenv("MONGODB_URI", "mongodb://localhost:27017"))
db = client.support_bot
knowledge_collection = db.knowledge_base
products_collection = db.products

class KnowledgeBase :
    async def search_by_keywords(self, keywords: List[str], category: Optional[str] = None, limit: int = 5) -> List[Dict[str, Any]]:
        """Пошук статей за ключовими словами та категорією"""
        query = { "$text": { "$search": " ".join(keywords)} }

if category:
            query["category"] = category


        cursor = knowledge_collection.find(
            query,
            { "score": { "$meta": "textScore"} }
        ).sort([("score", { "$meta": "textScore"})]).limit(limit)


        return await cursor.to_list(length = limit)


    async def get_by_category(self, category: str, limit: int = 10) -> List[Dict[str, Any]]:
        """Отримання статей за категорією"""
        cursor = knowledge_collection.find({ "category": category}).limit(limit)
        return await cursor.to_list(length = limit)


    async def get_by_id(self, article_id: str) -> Optional[Dict[str, Any]]:
        """Отримання статті за ID"""
        return await knowledge_collection.find_one({ "_id": ObjectId(article_id)})

    async def add_article(self, title: str, content: str, category: str, tags: List[str], author: str) -> str:
"""Додавання нової статті"""
        article = {
    "title": title,
            "content": content,
            "category": category,
            "tags": tags,
            "author": author,
            "created_at": datetime.utcnow(),
            "updated_at": datetime.utcnow()
```

```
        }

result = await knowledge_collection.insert_one(article)
        return str(result.inserted_id)


    async def update_article(self, article_id: str, updates: Dict[str, Any]) -> bool:
        """Оновлення існуючої статті"""
        updates["updated_at"] = datetime.utcnow()


        result = await knowledge_collection.update_one(
            { "_id": ObjectId(article_id)},
            { "$set": updates}
        )

        return result.modified_count > 0


    async def delete_article(self, article_id: str) -> bool:
        """Видалення статті"""
        result = await knowledge_collection.delete_one({ "_id": ObjectId(article_id)})
        return result.deleted_count > 0


    async def search_by_semantic_similarity(self, query: str, embeddings: List[float], limit: int = 5) -> List[Dict[str, Any]]:
        """Пошук статей за семантичною схожістю"""
        # Використовуємо векторні індекси для пошуку схожих документів
        # Це спрощена версія; у реальному проєкті може використовуватися спеціалізований пошуковий движок
        pipeline = [
            {
    "$search": {
        "knnBeta": {
            "vector": embeddings,
                    "path": "embeddings",
                    "k": limit
                }
    }
},
            {
    "$project": {
        "title": 1,
                    "content": 1,
                    "category": 1,
                    "tags": 1,
                    "score": { "$meta": "searchScore"}
    }
}
        ]

        cursor = knowledge_collection.aggregate(pipeline)
        return await cursor.to_list(length = limit)


class ProductsDB :
    async def get_all_products(self, limit: int = 100) -> List[Dict[str, Any]]:
        """Отримання списку всіх продуктів"""
        cursor = products_collection.find().limit(limit)
        return await cursor.to_list(length = limit)


    async def get_by_category(self, category: str, limit: int = 20) -> List[Dict[str, Any]]:
        """Отримання продуктів за категорією"""
        cursor = products_collection.find({ "category": category}).limit(limit)
```

```
        return await cursor.to_list(length = limit)


    async def get_by_id(self, product_id: str) -> Optional[Dict[str, Any]]:
        """Отримання продукту за ID"""
        if ObjectId.is_valid(product_id):
            return await products_collection.find_one({ "_id": ObjectId(product_id)})
        else:
            return await products_collection.find_one({ "product_id": product_id})

    async def search_products(self, query: str, limit: int = 10) -> List[Dict[str, Any]]:
        """Пошук продуктів за текстовим запитом"""
        search_query = { "$text": { "$search": query} }

cursor = products_collection.find(
    search_query,
            { "score": { "$meta": "textScore"} }
        ).sort([("score", { "$meta": "textScore"})]).limit(limit)


        return await cursor.to_list(length = limit)


    async def add_product(self, product_data: Dict[str, Any]) -> str:
"""Додавання нового продукту"""
        product_data["created_at"] = datetime.utcnow()
        product_data["updated_at"] = datetime.utcnow()


        result = await products_collection.insert_one(product_data)
        return str(result.inserted_id)


    async def update_product(self, product_id: str, updates: Dict[str, Any]) -> bool:
        """Оновлення існуючого продукту"""
        updates["updated_at"] = datetime.utcnow()


        if ObjectId.is_valid(product_id):
            result = await products_collection.update_one(
                { "_id": ObjectId(product_id)},
                { "$set": updates}
            )
        else:
            result = await products_collection.update_one(
                { "product_id": product_id},
                { "$set": updates}
            )

        return result.modified_count > 0


    async def delete_product(self, product_id: str) -> bool:
        """Видалення продукту"""
        if ObjectId.is_valid(product_id):
            result = await products_collection.delete_one({ "_id": ObjectId(product_id)})
        else:
            result = await products_collection.delete_one({ "product_id": product_id})

        return result.deleted_count > 0

# Створення екземплярів класів для зручного доступу
kb = KnowledgeBase()
```

```
products_db = ProductsDB()
```

МОДУЛЬ УПРАВЛІННЯ ЗАПИТАМИ (TICKET_MANAGER.PY)

```
import motor.motor_asyncio
import os
import uuid
from datetime import datetime
from typing import List, Dict, Any, Optional, Union

# Підключення до MongoDB і Redis
client = motor.motor_asyncio.AsyncIOMotorClient(os.getenv("MONGODB_URI", "mongodb://localhost:27017"))
db = client.support_bot
tickets_collection = db.tickets
operators_collection = db.operators
users_collection = db.users

class TicketManager :
    async def create_ticket(
        self,
        user_id: Union[str, int],
        description: str,
        subject: str = "Запит підтримки",
        category: str = "general",
        priority: str = "normal",
        product_id: Optional[str] = None
    ) -> str:
"""Створення нового запиту підтримки"""
        # Отримання інформації про користувача
        user = await users_collection.find_one({ "telegram_id": str(user_id)})

        if not user:
# Якщо користувача немає в базі, створюємо базовий запис
    user = { "telegram_id": str(user_id)}

# Створення унікального ідентифікатора запиту
ticket_id = str(uuid.uuid4())

        # Формування запису про запит
        ticket = {
    "ticket_id": ticket_id,
            "user_id": str(user_id),
            "username": user.get("username", "Unknown"),
            "subject": subject,
            "description": description,
            "category": category,
            "status": "new",
            "priority": priority,
            "product_id": product_id,
            "created_at": datetime.utcnow(),
            "updated_at": datetime.utcnow(),
            "assigned_to": None,
            "interaction_history": [
                {
        "timestamp": datetime.utcnow(),
                    "action": "created",
                    "message": description,
                    "sender": "user"
                }
            ]
        }
```

```
# Збереження запиту в базі даних
await tickets_collection.insert_one(ticket)

        # Призначення запиту оператору
        await self.assign_ticket(ticket_id)


        return ticket_id


    async def assign_ticket(self, ticket_id: str, operator_id: Optional[str] = None) -> bool:
        """Призначення запиту оператору"""
        ticket = await tickets_collection.find_one({ "ticket_id": ticket_id})

        if not ticket:
    return False


        if operator_id:
            # Якщо вказано конкретного оператора
            operator = await operators_collection.find_one({"operator_id": operator_id})
        else:
            # Інакше знаходимо доступного оператора з найменшим навантаженням
            operators = await self.get_available_operators(ticket["category"])


            if not operators:
# Якщо немає доступних операторів, залишаємо запит непризначеним
    return False

            # Обираємо оператора з найменшим навантаженням
            operator = min(operators, key= lambda op: op["active_tickets"])
            operator_id = operator["operator_id"]

        # Призначаємо запит оператору
        await tickets_collection.update_one(
            {"ticket_id": ticket_id},
            {
    "$set": {
        "assigned_to": operator_id,
                "status": "assigned",
                "updated_at": datetime.utcnow()
            },
            "$push": {
        "interaction_history": {
            "timestamp": datetime.utcnow(),
                    "action": "assigned",
                    "message": f"Запит призначено оператору {operator_id}",
                    "sender": "system"
                }
    }
}
        )

        # Оновлюємо кількість активних запитів оператора
        await operators_collection.update_one(
            { "operator_id": operator_id},
            { "$inc": { "active_tickets": 1} }
        )

        return True


    async def get_available_operators(self, category: Optional[str] = None) -> List[Dict[str, Any]]:
```

```
        """Отримання списку доступних операторів"""
        query = { "availability": "available"}

if category:
            # Якщо вказана категорія, шукаємо операторів з відповідною спеціалізацією
            query["specialization"] = { "$in": [category, "all"]}

cursor = operators_collection.find(query)
        return await cursor.to_list(length = 100)


    async def update_ticket_status(
        self,
        ticket_id: str,
        status: str,
        message: Optional[str] = None,
        sender: str = "system"
    ) -> bool:
        """Оновлення статусу запиту"""
        valid_statuses = ["new", "assigned", "in_progress", "waiting_for_customer", "resolved", "closed"]


        if status not in valid_statuses:
return False

        # Додаємо запис до історії взаємодії
        history_entry = {
    "timestamp": datetime.utcnow(),
            "action": "status_update",
            "message": message or f"Статус змінено на {status}",
            "sender": sender
        }

# Оновлюємо запис про запит

result = await tickets_collection.update_
```

ПОВНИЙ ЛІСТИНГ ЧАТБОТА ДЛЯ ПІДТРИМКИ КЛІЄНТІВ

```
# main.py - Головний файл бота

using Microsoft.VisualBasic;

using System.Collections.Generic;

using System.Net.Sockets;

using System.Runtime.Intrinsics.Arm;

using System;

import os

import logging

import asyncio

from aiogram import Bot, Dispatcher, types

from aiogram.contrib.fsm_storage.redis import RedisStorage2

from aiogram.dispatcher import FSMContext
```

```
from aiogram.dispatcher.filters.state import State, StatesGroup
from aiogram.types import ReplyKeyboardMarkup, KeyboardButton, InlineKeyboardMarkup, InlineKeyboardButton
from aiogram.utils import executor
import motor.motor_asyncio
from datetime import datetime
import uuid
import json
# Налаштування логування
logging.basicConfig(level=logging.INFO, format = '%(asctime)s - %(name)s - %(levelname)s - %(message)s')
logger = logging.getLogger(__name__)
# Конфігурація бота
TELEGRAM_BOT_TOKEN = os.getenv("TELEGRAM_BOT_TOKEN", "YOUR_BOT_TOKEN")
MONGODB_URI = os.getenv("MONGODB_URI", "mongodb://localhost:27017")
REDIS_HOST = os.getenv("REDIS_HOST", "localhost")
REDIS_PORT = int(os.getenv("REDIS_PORT", 6379))
REDIS_DB = int(os.getenv("REDIS_DB", 1))
# Ініціалізація бота та диспетчера
bot = Bot(token=TELEGRAM_BOT_TOKEN)
storage = RedisStorage2(host=REDIS_HOST, port = REDIS_PORT, db = REDIS_DB)
dp = Dispatcher(bot, storage = storage)
# Підключення до MongoDB
client = motor.motor_asyncio.AsyncIOMotorClient(MONGODB_URI)
db = client.support_bot
users_collection = db.users
knowledge_collection = db.knowledge_base
products_collection = db.products
tickets_collection = db.tickets
feedback_collection = db.feedback
analytics_collection = db.analytics
# Стани діалогів
class SupportStates(StatesGroup) :
    CHOOSING_PROBLEM_TYPE = State()
    DESCRIBING_PROBLEM = State()
```

```
    SELECTING_PRODUCT = State()
    CONFIRMING_TICKET = State()
    WAITING_FEEDBACK = State()
class ProductStates(StatesGroup) :
    CHOOSING_CATEGORY = State()
    CHOOSING_PRODUCT = State()
    VIEWING_DETAILS = State()
class FaqStates(StatesGroup) :
    CHOOSING_CATEGORY = State()
    CHOOSING_QUESTION = State()
    RATING_ANSWER = State()
class SearchStates(StatesGroup) :
    ENTERING_QUERY = State()
    VIEWING_RESULTS = State()
# Допоміжні функції
def get_main_keyboard():
    """Створення основної клавіатури головного меню"""
    keyboard = ReplyKeyboardMarkup(resize_keyboard = True)
    keyboard.add(KeyboardButton("📦 Продукти/Послуги"))
    keyboard.add(KeyboardButton("❓ FAQ"))
    keyboard.row(KeyboardButton("🔧 Підтримка"), KeyboardButton("👤 Мій акаунт"))
    keyboard.add(KeyboardButton("🔍 Пошук"))
    return keyboard
async def save_user(user_id, username, first_name, last_name):
    """Збереження або оновлення інформації про користувача"""
    now = datetime.utcnow()
    user = await users_collection.find_one({ "telegram_id": str(user_id)})
    if user:
        # Якщо користувач вже існує, оновлюємо дані
        await users_collection.update_one(
            { "telegram_id": str(user_id)},
            {
    "$set": {
```

```
        "username": username,
            "first_name": first_name,
            "last_name": last_name,
            "last_seen": now
        }
}
    )
  else:
    # Якщо новий користувач, створюємо запис
    await users_collection.insert_one({
  "telegram_id": str(user_id),
        "username": username,
        "first_name": first_name,
        "last_name": last_name,
        "first_seen": now,
        "last_seen": now,
        "language": "uk",
        "interaction_history": []
    })
async def log_interaction(user_id, action, content= None):
  """Логування взаємодії з користувачем"""
  interaction = {
  "timestamp": datetime.utcnow(),
    "action": action,
    "content": content
  }
await users_collection.update_one(
    { "telegram_id": str(user_id)},
    { "$push": { "interaction_history": interaction} }
  )
  # Оновлюємо статистику
  day_key = datetime.utcnow().strftime("%Y-%m-%d")
  await analytics_collection.update_one(
```

```
        { "type": "daily_actions", "date": day_key},
        { "$inc": { f"actions.{action}": 1} },
        upsert = True
    )
async def search_knowledge_base(query, category= None, limit= 5):
    """Пошук у базі знань за запитом"""
    # Для простоти використовуємо текстовий пошук
    search_query = { "$text": { "$search": query} }
if category:
        search_query["category"] = category
    cursor = knowledge_collection.find(
        search_query,
        { "score": { "$meta": "textScore"} }
    ).sort([("score", { "$meta": "textScore"})]).limit(limit)
    return await cursor.to_list(length = limit)
async def create_ticket(user_id, problem_type, description, product_id= None):
    """Створення запиту до оператора"""
    # Отримання інформації про користувача
    user = await users_collection.find_one({ "telegram_id": str(user_id)})
    # Створення унікального ідентифікатора запиту
    ticket_id = str(uuid.uuid4())[:8]
    # Формування запису про запит
    ticket = {
    "ticket_id": ticket_id,
        "user_id": str(user_id),
        "username": user.get("username"),
        "subject": f"Запит: {problem_type}",
        "description": description,
        "product_id": product_id,
        "status": "new",
        "priority": "normal",
        "created_at": datetime.utcnow(),
        "updated_at": datetime.utcnow(),
```

```
        "assigned_to": None,
        "interaction_history": [
            {
        "timestamp": datetime.utcnow(),
                "action": "created",
                "message": description,
                "sender": "user"
            }
        ]
    }
# Збереження запиту в базі даних
await tickets_collection.insert_one(ticket)
    # Оновлюємо статистику
    await analytics_collection.update_one(
        { "type": "tickets_stats"},
        { "$inc": { "total": 1, "new": 1} },
        upsert = True
    )
    return ticket_id
# Обробники команд
@dp.message_handler(commands = ['start'])
async def cmd_start(message: types.Message):
    """Обробник команди /start"""
    user_id = message.from_user.id
    username = message.from_user.username
    first_name = message.from_user.first_name
    last_name = message.from_user.last_name
    # Зберігаємо інформацію про користувача
    await save_user(user_id, username, first_name, last_name)
    # Логування події
    await log_interaction(user_id, "start_command")
    # Відправка привітання
    await message.answer(
```

```
        f"Вітаю, {first_name}! 👋\n\n"
        f"Я чатбот підтримки клієнтів компанії X. Чим можу допомогти вам сьогодні?\n\n"
        f"• Отримати інформацію про продукти та послуги\n"
        f"• Відповісти на поширені запитання\n"
        f"• Допомогти з технічними проблемами\n"
        f"• Створити запит до служби підтримки\n\n"
        f"Оберіть опцію з меню нижче:",
        reply_markup = get_main_keyboard()
    )
@dp.message_handler(commands = ['help'])
async def cmd_help(message: types.Message):
    """Обробник команди /help"""
    await log_interaction(message.from_user.id, "help_command")
    help_text = (
        "Я бот підтримки клієнтів компанії X. Ось мої основні команди:\n\n"
        "/start - Почати взаємодію з ботом\n"
        "/help - Отримати цю довідку\n"
        "/faq - Відповіді на поширені запитання\n"
        "/products - Каталог продуктів та послуг\n"
        "/support - Створити запит до служби підтримки\n"
        "/status - Перевірити статус існуючих запитів\n\n"
        "Також ви можете використовувати кнопки меню для навігації."
    )
    await message.answer(help_text, reply_markup = get_main_keyboard())
# Обробники текстових повідомлень для головного меню
@dp.message_handler(lambda message: message.text == "📋 Продукти/Послуги" or message.text == "/products")
async def show_products(message: types.Message):
    """Обробник вибору категорії 'Продукти/Послуги'"""
    await log_interaction(message.from_user.id, "products_menu")

    # Отримуємо категорії продуктів з бази даних
    categories = await db.product_categories.find().to_list(length = 10)
    if not categories:
```

```
# Якщо категорії ще не додані в базу даних, використовуємо стандартні
    categories = [
        { "id": "1", "name": "Смартфони та планшети"},
        { "id": "2", "name": "Комп'ютери та ноутбуки"},
        { "id": "3", "name": "Аудіо та відео"},
        { "id": "4", "name": "Аксесуари"}
      ]
    # Створюємо інлайн-клавіатуру з категоріями
    keyboard = InlineKeyboardMarkup(row_width = 1)
    for category in categories:
      keyboard.add(
        InlineKeyboardButton(
          category["name"],
          callback_data = f"product_category_{category['id']}"
        )
      )
    await message.answer(
      "Оберіть категорію продуктів, яка вас цікавить:",
reply_markup = keyboard
)
    # Встановлюємо стан діалогу
    await ProductStates.CHOOSING_CATEGORY.set()
@dp.message_handler(lambda message: message.text == "❓ FAQ" or message.text == "/faq")
async def show_faq(message: types.Message):
    """Обробник вибору категорії 'FAQ'"""
    await log_interaction(message.from_user.id, "faq_menu")
    # Отримуємо категорії FAQ з бази даних
    categories = await db.faq_categories.find().to_list(length = 10)
    if not categories:
# Якщо категорії ще не додані в базу даних, використовуємо стандартні
      categories = [
        { "id": "1", "name": "Загальні питання"},
        { "id": "2", "name": "Доставка та оплата"},
```

```
            { "id": "3", "name": "Повернення та обмін"},
            { "id": "4", "name": "Технічні питання"}
        ]
    # Створюємо інлайн-клавіатуру з категоріями
    keyboard = InlineKeyboardMarkup(row_width = 1)
    for category in categories:
        keyboard.add(
            InlineKeyboardButton(
                category["name"],
                callback_data = f"faq_category_{category['id']}"
            )
        )
    await message.answer(
        "Оберіть категорію питань, яка вас цікавить:",
reply_markup = keyboard
)
    # Встановлюємо стан діалогу
    await FaqStates.CHOOSING_CATEGORY.set()
@dp.message_handler(lambda message: message.text == "🔧 Підтримка" or message.text == "/support")
async def show_support(message: types.Message):
    """Обробник вибору категорії 'Підтримка'"""
    await log_interaction(message.from_user.id, "support_menu")
    # Створюємо клавіатуру для вибору типу проблеми
    keyboard = ReplyKeyboardMarkup(resize_keyboard = True)
    keyboard.add(KeyboardButton("Технічна проблема"))
    keyboard.add(KeyboardButton("Питання щодо продукту"))
    keyboard.add(KeyboardButton("Питання доставки/оплати"))
    keyboard.add(KeyboardButton("Інше питання"))
    keyboard.add(KeyboardButton("🔙 Назад до головного меню"))
    await message.answer(
        "Будь ласка, оберіть тип проблеми, з якою ви зіткнулися:",
reply_markup = keyboard
)
```

```
    # Встановлюємо стан діалогу
    await SupportStates.CHOOSING_PROBLEM_TYPE.set()


@dp.message_handler(lambda message: message.text == "👤 Мій акаунт")
async def show_account(message: types.Message):
    """Обробник вибору категорії 'Мій акаунт'"""
    await log_interaction(message.from_user.id, "account_menu")
    # Отримуємо інформацію про користувача
    user = await users_collection.find_one({ "telegram_id": str(message.from_user.id)})
    if not user:
# Якщо користувача немає в базі, зберігаємо його
    await save_user(
        message.from_user.id,
        message.from_user.username,
        message.from_user.first_name,
        message.from_user.last_name
    )
        user = await users_collection.find_one({ "telegram_id": str(message.from_user.id)})
    # Отримуємо активні запити користувача
    active_tickets = await tickets_collection.find({
    "user_id": str(message.from_user.id),
        "status": { "$in": ["new", "assigned", "in_progress"]}
}).to_list(length = 5)
    # Формуємо повідомлення з інформацією про акаунт
    account_text = (
        f"👤 **Профіль користувача**\n\n"
        f"Ім'я: {user.get('first_name', 'Невідомо')}\n"
        f"Прізвище: {user.get('last_name', 'Невідомо')}\n"
        f"Username: @{user.get('username', 'Невідомо')}\n"
        f"Дата першого звернення: {user.get('first_seen').strftime('%d.%m.%Y') if user.get('first_seen') else 'Невідомо'}\n\n"
    )
    if active_tickets:
        account_text += "📋 **Активні запити:**\n\n"
```

```
        for ticket in active_tickets:
            account_text += (
                f"Запит #{ticket['ticket_id']}\n"
                f"Статус: {ticket['status']}\n"
                f"Створено: {ticket['created_at'].strftime('%d.%m.%Y %H:%M')}\n\n"
            )
    else:
        account_text += "У вас немає активних запитів.\n"
    # Створюємо клавіатуру для управління акаунтом
    keyboard = InlineKeyboardMarkup(row_width = 1)
    keyboard.add(InlineKeyboardButton("Мої запити", callback_data = "my_tickets"))
    keyboard.add(InlineKeyboardButton("Змінити налаштування", callback_data = "account_settings"))
    await message.answer(
account_text,
        reply_markup = keyboard,
        parse_mode = "Markdown"
    )
@dp.message_handler(lambda message: message.text == "🔍 Пошук")
async def start_search(message: types.Message):
    """Обробник вибору категорії 'Пошук'"""
    await log_interaction(message.from_user.id, "search_menu")
    cancel_keyboard = ReplyKeyboardMarkup(resize_keyboard = True)
    cancel_keyboard.add(KeyboardButton("🔙 Назад до головного меню"))
    await message.answer(
        "Введіть ваш запит для пошуку по базі знань та продуктах:",
        reply_markup = cancel_keyboard
    )
    # Встановлюємо стан діалогу
    await SearchStates.ENTERING_QUERY.set()
# Обробник для повернення до головного меню
@dp.message_handler(lambda message: message.text == "🔙 Назад до головного меню", state = "*")
async def back_to_main_menu(message: types.Message, state: FSMContext):
    """Обробник для повернення до головного меню з будь-якого стану"""
```

```
    await state.finish()
    await message.answer(
        "Ви повернулись до головного меню. Чим я можу допомогти?",
        reply_markup = get_main_keyboard()
    )
# Обробники для стану SupportStates.CHOOSING_PROBLEM_TYPE
@dp.message_handler(state = SupportStates.CHOOSING_PROBLEM_TYPE)
async def process_problem_type(message: types.Message, state: FSMContext):
    """Обробник вибору типу проблеми"""
    if message.text == "⮌ Назад до головного меню":
        await state.finish()
        await message.answer(
            "Ви повернулись до головного меню. Чим я можу допомогти?",
            reply_markup = get_main_keyboard()
        )
        return
# Зберігаємо тип проблеми в даних стану
    await state.update_data(problem_type = message.text)
    # Логування події
    await log_interaction(message.from_user.id, "support_problem_type", { "type": message.text})
    # Переходимо до опису проблеми
    await message.answer(
        "Будь ласка, опишіть вашу проблему детальніше:"
    )
    # Змінюємо стан діалогу
    await SupportStates.DESCRIBING_PROBLEM.set()
# Обробники для стану SupportStates.DESCRIBING_PROBLEM
@dp.message_handler(state = SupportStates.DESCRIBING_PROBLEM)
async def process_problem_description(message: types.Message, state: FSMContext):
    """Обробник опису проблеми"""
    if message.text == "⮌ Назад до головного меню":
        await state.finish()
        await message.answer(
```

```
            "Ви повернулись до головного меню. Чим я можу допомогти?",
            reply_markup = get_main_keyboard()
        )
        return
    # Зберігаємо опис проблеми в даних стану
    await state.update_data(problem_description = message.text)
    # Логування події
    await log_interaction(message.from_user.id, "support_problem_description")
    # Спочатку спробуємо знайти рішення в базі знань
    results = await search_knowledge_base(message.text)
    if results:
        # Якщо знайдено релевантні записи, пропонуємо їх користувачу
        kb_answer = f"Я знайшов інформацію, яка може допомогти вирішити вашу проблему:\n\n"
        kb_answer += f"**{results[0]['title']}**\n\n{results[0]['content'][:300]}...\n\n"
        keyboard = InlineKeyboardMarkup(row_width = 1)
        keyboard.add(InlineKeyboardButton("Показати повну відповідь", callback_data = f"kb_article_{results[0]['_id']}"))
        keyboard.add(InlineKeyboardButton("Це вирішило мою проблему", callback_data = "problem_solved"))
        keyboard.add(InlineKeyboardButton("Мені потрібна додаткова допомога", callback_data = "need_more_help"))
        await message.answer(
            kb_answer,
            reply_markup = keyboard,
            parse_mode = "Markdown"
        )
    else:
        # Якщо не знайдено релевантної інформації, переходимо до створення запиту
        user_data = await state.get_data()
        # Створюємо клавіатуру для вибору продукту
        keyboard = InlineKeyboardMarkup(row_width = 1)
        # Отримуємо список популярних продуктів
        products = await products_collection.find().limit(5).to_list(length = 5)
        if products:
            for product in products:
                keyboard.add(InlineKeyboardButton(
```

```
                product.get("name", "Невідомий продукт"),
                callback_data = f"product_select_{product['_id']}"
            ))
        keyboard.add(InlineKeyboardButton("Інший продукт", callback_data = "other_product"))
        keyboard.add(InlineKeyboardButton("Не стосується продукту", callback_data = "no_product"))
        await message.answer(
            "Будь ласка, оберіть продукт, з яким пов'язана ваша проблема:",
            reply_markup = keyboard
        )
        # Змінюємо стан діалогу
        await SupportStates.SELECTING_PRODUCT.set()
# Обробники для колбек-запитів
@dp.callback_query_handler(lambda c: c.data.startswith('product_category_'), state = ProductStates.CHOOSING_CATEGORY)
async def process_product_category(callback_query: types.CallbackQuery, state: FSMContext):
    """Обробник вибору категорії продуктів"""
    await bot.answer_callback_query(callback_query.id)
    category_id = callback_query.data.split('_')[2]
    # Зберігаємо вибрану категорію в даних стану
    await state.update_data(category_id = category_id)
    # Отримуємо продукти в цій категорії
    products = await products_collection.find({ "category_id": category_id}).to_list(length = 10)
    if not products:
# Якщо продукти ще не додані в базу даних, використовуємо приклади
    products = [
            { "_id": "1", "name": "Продукт 1", "category_id": category_id, "price": 999},
            { "_id": "2", "name": "Продукт 2", "category_id": category_id, "price": 1499},
            { "_id": "3", "name": "Продукт 3", "category_id": category_id, "price": 1999}
        ]
    # Створюємо інлайн-клавіатуру з продуктами
    keyboard = InlineKeyboardMarkup(row_width = 1)
    for product in products:
        keyboard.add(
            InlineKeyboardButton(
```

```
            f"{product['name']} - {product.get('price', 'Ціна за запитом')} грн",
            callback_data = f"product_view_{product['_id']}"
        )
    )
    keyboard.add(InlineKeyboardButton("🔙 Назад до категорій", callback_data = "back_to_categories"))
    # Отримуємо назву категорії
    category = await db.product_categories.find_one({ "id": category_id})
    category_name = category["name"] if category else "Вибрана категорія"
    await bot.edit_message_text(
        f"Продукти в категорії \"{category_name}\":",
        callback_query.from_user.id,
        callback_query.message.message_id,
        reply_markup = keyboard
)
    # Змінюємо стан діалогу
    await ProductStates.CHOOSING_PRODUCT.set()
@dp.callback_query_handler(lambda c: c.data.startswith('faq_category_'), state = FaqStates.CHOOSING_CATEGORY)
async def process_faq_category(callback_query: types.CallbackQuery, state: FSMContext):
    """Обробник вибору категорії FAQ"""
    await bot.answer_callback_query(callback_query.id)
    category_id = callback_query.data.split('_')[2]
    # Зберігаємо вибрану категорію в даних стану
    await state.update_data(category_id = category_id)
    # Отримуємо питання в цій категорії
    questions = await knowledge_collection.find({ "category_id": category_id, "type": "faq"}).to_list(length = 10)
    if not questions:
# Якщо питання ще не додані в базу даних, використовуємо приклади
    questions = [
            { "_id": "1", "title": "Як замовити товар?", "category_id": category_id},
            { "_id": "2", "title": "Які способи оплати доступні?", "category_id": category_id},
            { "_id": "3", "title": "Як відстежити моє замовлення?", "category_id": category_id}
        ]
    # Створюємо інлайн-клавіатуру з питаннями
```

```
    keyboard = InlineKeyboardMarkup(row_width = 1)
    for question in questions:
        keyboard.add(
            InlineKeyboardButton(
                question['title'],
                callback_data = f"faq_question_{question['_id']}"
            )
        )
    keyboard.add(InlineKeyboardButton("🔙 Назад до категорій", callback_data = "back_to_faq_categories"))
    # Отримуємо назву категорії
    category = await db.faq_categories.find_one({ "id": category_id})
    category_name = category["name"] if category else "Вибрана категорія"
    await bot.edit_message_text(
        f"Поширені запитання в категорії \"{category_name}\":",
        callback_query.from_user.id,
        callback_query.message.message_id,
        reply_markup = keyboard
)
    # Змінюємо стан діалогу
    await FaqStates.CHOOSING_QUESTION.set()
@dp.callback_query_handler(lambda c: c.data.startswith('product_select_'), state = SupportStates.SELECTING_PRODUCT)
async def process_support_product_selection(callback_query: types.CallbackQuery, state: FSMContext):
    """Обробник вибору продукту при створенні запиту підтримки"""
    await bot.answer_callback_query(callback_query.id)
    product_id = callback_query.data.split('_')[2]
    # Зберігаємо вибраний продукт в даних стану
    await state.update_data(product_id = product_id)
    # Отримуємо дані стану
    user_data = await state.get_data()
    # Отримуємо інформацію про продукт
    product = await products_collection.find_one({ "_id": product_id})
    product_name = product["name"] if product else "Вибраний продукт"
    # Створюємо клавіатуру для підтвердження
```

```
keyboard = InlineKeyboardMarkup(row_width = 2)
keyboard.add(
    InlineKeyboardButton("✅ Підтвердити", callback_data = "confirm_ticket"),
    InlineKeyboardButton("❌ Скасувати", callback_data = "cancel_ticket")
)
# Формуємо повідомлення для підтвердження
confirmation_text = (
    f"Будь ласка, підтвердіть створення запиту з такою інформацією:\n\n"
    f"**Тип проблеми:** {user_data['problem_type']}\n"
    f"**Опис:** {user_data['problem_description']}\n"
    f"**Продукт:** {product_name}\n\n"
    f"Натисніть 'Підтвердити' для створення запиту або 'Скасувати' для скасування."
)
await bot.edit_message_text
```