

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

«Застосунок для керування пристроєм голосовими командами »

Виконав студент групи ПЗ-21-4
ДУБНИЦЬКИЙ Ярослав Володимирович

Керівник роботи:
ТОЛСТОЙ Ігор Анатолійович

Рецензент:
ЛОКТИКОВА Тамара Миколаївна

Житомир – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Зав. кафедри інженерії
програмного забезпечення
Тетяна Вакалюк

«14» лютого 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ДУБНИЦЬКИЙ** Ярослав Володимирович

Керівник роботи: **ТОЛСТОЙ** Ігор Анатолійович

Тема роботи: **«Застосунок для керування пристроєм голосовими командами»**,

затверджена Наказом закладу вищої освіти від **«14» лютого 2025 р., №61/с**

Вихідні дані для роботи: **Об'єктом дослідження є взаємодії користувача з розумними пристроями за допомогою голосових команд.**

Предметом дослідження є використання технологій автоматичного розпізнавання голосу (ASR), його обробки (NLP) та API для реалізації голосового керування пристроями (IoT).

Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Толстой І. А.	27.02.2025р.	27.02.2025р.
2	Толстой І. А.	27.02.2025р.	27.02.2025р.
3	Толстой І. А.	27.02.2025р.	27.02.2025р.

РЕФЕРАТ

Випускна кваліфікаційна робота бакалавра складається з програмного застосунку для керування пристроями голосовими командами та пояснювальної записки. Пояснювальна записка до випускної кваліфікаційної роботи містить 100 сторінок, з яких 65 становить основний текст, 32 ілюстрації та 17 таблиць.

Метою роботи є створення автономного голосового асистента, який функціонує локально без постійного підключення до інтернету, забезпечує зручне керування комп'ютером та IoT-пристроями, а також підтримує українську мову.

Система реалізована на основі архітектури MVC з використанням Python. Вона містить модулі для розпізнавання мовлення (ASR), обробки природної мови (NLP), синтезу мовлення (TTS), роботи з API та управління пристроями через MQTT і HTTP. Всі дані зберігаються локально у базі даних SQLite. Асистент підтримує офлайн-режим, активацію по ключовому слову, голосовий інтерфейс та конфігурацію команд користувачем. Передбачена можливість автоматизації дій за розкладом та подальше розширення функціональності через плагіни.

КЛЮЧОВІ СЛОВА: ГОЛОСОВИЙ АСИСТЕНТ, PYTHON, РОЗПІЗНАВАННЯ МОВЛЕННЯ, IoT, MQTT, API, TTS, NLP.

					ІПЗ.КР.Б – 121 – 25 – ПЗ			
Змн.	Арк.	№ докум.	Підпис	Дата	Застосунок для керування пристроєм голосовими командами Пояснювальна записка	Літ.	Арк.	Аркушів
Розроб.		Дубницький Я.В					3	20
Керівник		Толстой І.А.				Житомирська політехніка, група ІПЗ-21-4		
Рецензент.								
Зав. каф.								

ABSTRACT

The bachelor's qualification work consists of a software application for device control via voice commands and an explanatory report. The explanatory report contains 100 pages, of which 65 pages represent the main body of text, along with 32 illustrations and 17 tables.

The aim of the project is to develop an autonomous voice assistant that operates locally without constant internet access, enables convenient control of the computer and IoT devices, and supports the Ukrainian language.

The system is implemented using the MVC architecture in Python. It includes modules for Automatic Speech Recognition (ASR), Natural Language Processing (NLP), Text-to-Speech (TTS), API communication, and device management via MQTT and HTTP protocols. All data is stored locally in an SQLite database. The assistant supports offline mode, activation by a keyword, a voice interface, and user-configurable commands. The system also provides automation by schedule and is designed to be extended through plugins.

KEYWORDS: VOICE ASSISTANT, PYTHON, SPEECH RECOGNITION, IoT, MQTT, API, TTS, NLP.

					ІІЗ.КР.Б – 121 – 25 – ІІЗ	Арк.
						4

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	6
ВСТУП.....	7
РОЗДІЛ 1. ТЕОРЕТИКО-АНАЛІТИЧНІ ЗАСАДИ ПОБУДОВИ СИСТЕМИ ГОЛОСОВОГО КЕРУВАННЯ	9
1.1. Формулювання наукової проблеми та постановка задачі	9
1.2. Комплексний аналіз існуючих рішень у сфері голосового управління	11
1.3. Вибір ефективної архітектури програмного агента на основі сучасних підходів.....	22
1.4. Технічне обґрунтування: вибір інструментів, технологій та вимог до системи.....	24
Висновки до розділу 1	25
РОЗДІЛ 2. КОНЦЕПТУАЛЬНЕ ПРОЄКТУВАННЯ І АРХІТЕКТУРНА РЕАЛІЗАЦІЯ СИСТЕМИ.....	27
2.1. Моделювання сценаріїв використання: побудова діаграм прецедентів.....	27
2.2. Проєктування логічної структури даних та інформаційних потоків.....	33
2.3. Архітектура програмних модулів та моделювання об'єктно-орієнтованої структури.....	39
2.4. Алгоритмічна реалізація функціональних компонентів голосового інтерфейсу	43
Висновки до розділу 2	49
РОЗДІЛ 3. ІНТЕРАКТИВНА ВЗАЄМОДІЯ КОРИСТУВАЧА ЗІ СМАРТ-СИСТЕМОЮ: ІНТЕРФЕЙС І ВАЛІДАЦІЯ.....	50
3.1. Методологія встановлення та конфігурування інтелектуального застосунку	50
3.2. Структурно-функціональний аналіз інтерфейсу користувача.....	52
3.3. Методика тестування та верифікації ефективності програмної реалізації	55
3.4. Методика та результати тестування	62
Висновки до розділу 3	69
ВИСНОВКИ	71
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ :	73
ДОДАТКИ.....	76

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПЗ – програмне забезпечення.

API – application programming interface (інтерфейс прикладного програмування).

ASR (Automatic Speech Recognition) – автоматичне розпізнавання голосу.

NLP (Natural Language Processing) – обробка голосу.

TTS (Text To Speech) – конвертація тексту в голос.

STT (Speech To Text) – конвертація голосу в текст.

VAD (Voice Activity Detection) – виявлення голосової активності.

SS (Silence Suppresion) – придушення тиші.

WWD (Wake Word Detection) – виявлення «слова активації».

PC (Personal Computer) – Персональний комп'ютер.

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						6

ВСТУП

Актуальність теми зумовлена стрімким розвитком технологій голосового керування та їх інтеграцією в повсякденне життя. З кожним роком зростає потреба на зручні та інтуїтивні способи взаємодії з девайсами, де голос виступає як засіб комунікації між людиною та машиною. Голосові помічники стають важливими інструментами цифрового світу, вони застосовуються у смартфонах, розумних будинках, автомобілях і офісних системах. Також ця технологія полегшить людям з обмеженими можливостями взаємодію з девайсами. Тому, розробка власної системи управління пристроями голосовими командами на основі сучасних алгоритмів обробки голосу та машинного навчання є актуальним завданням.

Метою даної кваліфікаційної роботи є проектування та розробка застосунку для керування пристроями голосовими командами.

Для реалізації поставленої мети потрібно виконати наступні завдання:

1. Провести огляд функціональних можливостей існуючих голосових помічників та систем керування пристроями;
2. Обрати архітектуру системи та інструментальні засоби для реалізації голосового асистента;
3. Спроектувати логіку взаємодії користувача з системою, включаючи обробку команд і керування пристроями;
4. Розробити програмне забезпечення, що реалізує голосове керування пристроями в реальному часі.

Об'єктом дослідження є процес взаємодії користувача з комп'ютерними та IoT-пристроями шляхом голосових команд, що реалізується за допомогою сучасних технологій штучного інтелекту. Така взаємодія передбачає безконтактне, інтуїтивне управління цифровими сервісами, що підвищує зручність користування системами в повсякденному житті, зокрема для осіб з обмеженими можливостями.

Предметом дослідження є комплекс технологій, які забезпечують повноцінне функціонування голосового управління. До них належать: ASR, NLP, TTS, IoT-технології та Інтерфейси API.

ASR (Automatic Speech Recognition) технологія автоматичного розпізнавання мовлення. NLP (Natural Language Processing) методи обробки природної мови. TTS (Text-to-Speech) конвертування нормального друкованого тексту на аудіо. IoT-технології, засоби взаємодії з розумними пристроями. API взаємодія із зовнішніми програмними компонентами, операційними системами або мікросервісами.

Дослідження зосереджується на інтеграції цих компонентів у єдину автономну систему, яка працює локально без постійного підключення до інтернету, забезпечуючи високу продуктивність, приватність даних та персоналізацію взаємодії. [28]

Використаними методами дослідження є сукупність підходів, які забезпечують всебічний аналіз предмету дослідження. До них належать: метод порівняльного аналізу, емпіричний метод та SWOT-аналіз.

Метод порівняльного аналізу – застосований для зіставлення функціональних можливостей існуючих рішень на ринку голосових асистентів, з метою виявлення переваг та недоліків. [5]

Емпіричний метод – використано під час практичного тестування аналогів в умовах, наближених до реального використання, що дозволило об’єктивно оцінити їх ефективність. [6]

SWOT-аналіз – стратегічний метод, який дозволив проаналізувати сильні й слабкі сторони конкурентних продуктів, а також визначити можливості для розвитку та загрози впровадження.

РОЗДІЛ 1. ТЕОРЕТИКО-АНАЛІТИЧНІ ЗАСАДИ ПОБУДОВИ СИСТЕМИ ГОЛОСОВОГО КЕРУВАННЯ

1.1. Формулювання наукової проблеми та постановка задачі

В даній роботі метою є розробка застосунку для керування пристроями голосовими командами. Такий інструмент дозволить значно підвищити зручність взаємодії користувача з розумними пристроями, забезпечуючи інтуїтивне та доступне управління за допомогою голосу. [4]

Спостерігається стрімке зростання кількості інтелектуальних пристроїв, що використовуються у побуті, на виробництві та в офісному середовищі. Проте способи взаємодії з ними часто залишаються традиційними – за допомогою клавіатури, миші або сенсорного екрана. Це створює низку бар'єрів для певних категорій користувачів, уповільнює процес взаємодії та знижує загальну ефективність систем. Саме тому все більше уваги приділяється розвитку голосових інтерфейсів, які дозволяють здійснювати керування без використання рук, природним способом – мовленням. [11]

Голосове керування – це сучасна технологія, яка дає змогу користувачу контролювати пристрої без використання рук, лише промовивши команду. Така система базується на технологіях автоматичного розпізнавання мовлення (ASR), обробки природної мови (NLP), а також взаємодії з пристроями за допомогою IoT-протоколів та керує РС за допомогою скриптових команд. Подібні рішення вже широко використовуються в побуті: від голосових асистентів у смартфонах до розумних колонок, здатних вмикати світло чи музику. [1], [2].

Для реалізації такого застосунку потрібно створити декілька модулів, кожен з яких виконує конкретну функцію. Модуль розпізнавання мовлення відповідає за перетворення голосу в текст, тоді як модуль виявлення активності мовлення визначає моменти, коли користувач починає говорити. Модуль обробки природної мови аналізує зміст сказаного для забезпечення правильного тлумачення команд. Спеціальний модуль слова-пробудження активує систему у відповідь на ключову

фразу. Для забезпечення інтеграції з реальними пристроями, такими як лампи чи розетки, використовується модуль керування IoT пристроями. Завершує цей набір модуль інтерфейсу користувача, який забезпечує зручну взаємодію із системою.[23]

У результаті буде створено повноцінну систему голосового асистента, яка зможе працювати локально на комп'ютері користувача або в домашній мережі. Асистент надаватиме змогу виконувати повсякденні завдання – відкривати програми, керувати пристроями (IoT), взаємодіяти з API та навіть відповідати на запити голосом. Такий підхід зробить користування цифровими пристроями простішим, швидшим і доступнішим для широкого кола людей. [8]

Основними етапами розробки застосунку є.

1. Проектування архітектури системи:

- Визначення компонентів (ASR, NLP, Command, IoT);
- Побудова логіки взаємодії між модулями;
- Створення діаграм сценаріїв використання (use cases).

2. Розробка основного функціоналу:

- Впровадження WWD і активації голосом;
- Реалізація обробки голосових запитів і виконання команд;
- Керування IoT-пристроями через MQTT.

3. Тестування:

- Перевірка точності розпізнавання голосу;
- Тестування керування пристроями та системними командами.

Результатом реалізації поставленого завдання є функціональний голосовий помічник, що дозволяє користувачу голосом: управляти пристроями «розумного будинку», виконувати дії на комп'ютері, отримувати відповіді, допомогу або діалогову взаємодію.

1.2. Комплексний аналіз існуючих рішень у сфері голосового управління

Голосове управління, як технологія взаємодії людини з електронними пристроями, пройшло тривалий шлях розвитку. Його витoki сягають середини ХХ століття, коли перші системи розпізнавання мови були створені для військових та наукових цілей. Однак справжній прорив стався на початку 2010-х років із появою потужних алгоритмів машинного навчання, розвитком штучного інтелекту та збільшенням обчислювальних можливостей. [\[11\]](#), [\[16\]](#), [\[17\]](#).

Сьогодні голосові асистенти стали невід'ємною частиною сучасних технологій. Вони широко інтегровані в смартфони, автомобілі, побутову техніку та системи «розумного будинку». Їх популярність пояснюється кількома ключовими факторами.

Перш за все, це зручність, адже користувачі можуть керувати пристроями без фізичного контакту.

Другим аспектом є підвищення безпеки, оскільки голосове керування зменшує рівень відволікання водіїв або інших користувачів під час виконання різних завдань.

Підсумковою важливою перевагою є доступність, адже такі технології вже інтегровані у велику кількість масових пристроїв, зокрема колонки, телефони й персональні комп'ютери. Нарешті, значну роль відіграє розвиток штучного інтелекту, який сприяє підвищенню точності розпізнавання мови та покращенню контекстного аналізу запитів. [\[2\]](#)

Історичний контекст розвитку голосових асистентів починається в 1960 - 1990-ті: Перші експерименти з розпізнавання мови (наприклад, система «Shoebox» IBM). Обмежені можливості через недостатню потужність комп'ютерів. 2000-ті: Поява комерційних продуктів (наприклад, Dragon NaturallySpeaking). Технологія залишалася нішею через високу вартість і низьку точність. 2010-ті: Прорив завдяки нейромережам і хмарним обчисленням. Запуск Siri (2011), Google Now (2012), Alexa (2014). 2020-ті: Масове впровадження в IoT, автономні

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						11

системи з офлайн-режимом (наприклад, локальні моделі на основі Whisper від OpenAI). [10], [3], [39].

Для розробки застосунку управління пристроєм голосовими командами необхідно проаналізувати існуючі аналоги, щоб визначити їх основний функціонал, виявити переваги та недоліки, а також знайти можливості для вдосконалення. Ринок голосового керування активно розвивається, і вже існує велика кількість рішень, які можна умовно поділити на кілька категорій.

1. Автомобільні голосові помічники:

- Android Auto;
- BMW Intelligent Personal Assistant;
- Teyes Voice Control System.

2. Асистенти для керування Smart House:

- Amazon Alexa;
- Xiaomi XiaoAI.

3. Гібридні асистенти:

- Snips;
- Mycroft AI.

Усі ці рішення мають певну спеціалізацію, обмеження та переваги, які потрібно враховувати під час створення нового голосового асистента. Далі розглянемо їх більш детально. [25]

Teyes Voice Control System (рис. 1.1) – це інноваційна система голосового керування, інтегрована в мультимедійні головні пристрої TEYES. Система дозволяє водіям керувати функціями автомобіля за допомогою голосу, що забезпечує безпечніше водіння.



Рис.1.1 – Головний екран Teyes Voice

До переваг програмного продукту Teyes Voice можна віднести його здатність працювати в офлайн-режимі, що забезпечує стабільну роботу без необхідності підключення до інтернету. Також він має тісну інтеграцію з функціональністю автомобіля, підтримує декілька мов і сприяє безпечному керуванню під час руху, зменшуючи потребу у фізичному взаємодії з пристроєм.

Водночас до недоліків цієї системи належить те, що вона не підтримує роботу на персональних комп'ютерах і смартфонах, має обмежений набір доступних голосових команд і не дозволяє гнучко налаштовувати параметри користувачем.

Android Auto (рис. 1.2) – система, яка дозволяє інтегрувати смартфон Android з мультимедійною системою автомобіля. Голосове керування реалізується через Google Assistant, дозволяючи керувати навігацією, викликами, повідомленнями та музикою.



Рис.1.2 – Інтерфейс Android Auto

Програмне рішення Android Auto має свої переваги, серед яких варто виокремити глибоку інтеграцію з екосистемою Google-сервісів, підтримку української мови через Google Assistant, а також можливість безпечного голосового керування функціями під час водіння.

Проте воно має і свої недоліки. Для його використання необхідний смартфон на базі Android, а також стабільне підключення до інтернету, без якого більшість функцій стають недоступними.

BMW Intelligent Personal Assistant (рис. 1.3) – вбудований голосовий помічник у автомобілях BMW, який дозволяє керувати функціями авто, включно з клімат-контролем, навігацією, музикою тощо.

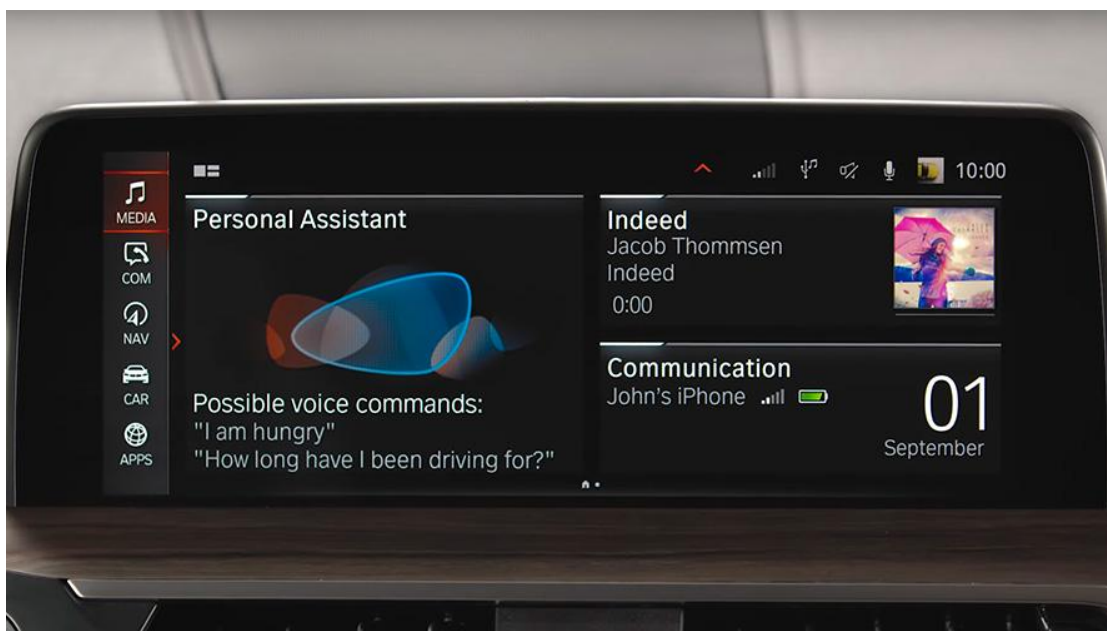


Рис.1.3 – Інтерфейс BMW Intelligent

До позитивних сторін BMW Intelligent Personal Assistant належить глибока інтеграція з електронними системами автомобіля, можливість персоналізації поведінки асистента відповідно до вподобань користувача, а також підтримка офлайн-режиму для базових функцій.

Водночас, основними обмеженнями є те, що цей асистент доступний лише в автомобілях BMW і не передбачає можливості використання на сторонніх пристроях або в інших середовищах.

Amazon Alexa (рис. 1.4) – голосовий асистент для управління розумним будинком, пристроями IoT, прослуховування музики, отримання новин та іншої інформації. Працює через фірмові пристрої Amazon Echo, Fire TV та інші.



Рис.1.4 – Девайси Amazon Alexa

До переваг Amazon Alexa можна віднести її глибоку інтеграцію з екосистемою пристроїв Інтернету речей (IoT), широкий набір доступних функцій, можливість розширення через додаткові модулі, відомі як «навики», а також підтримку голосового керування кількома пристроями одночасно.

Водночас ця система має низку суттєвих обмежень. Зокрема, вона не підтримує офлайн-режим, не має української мовної локалізації, функціонує у межах закритої платформи і потребує спеціального обладнання для повноцінного використання.

XiaoAI (рис. 1.5) – це голосовий асистент, розроблений компанією Xiaomi, який є невід’ємною частиною екосистеми MI Home. Основне призначення XiaoAI – забезпечення інтелектуального керування пристроями розумного будинку, такими як освітлення, телевізори, кондиціонери, пилососи, камери спостереження та інші сумісні смарт-пристрої.



Рис.1.5 – Екосистема XiaoAI

До переваг XiaoAI відноситься широка підтримка пристроїв екосистеми Xiaomi, ефективною інтеграцією з мобільними додатками та активним розвитком функціональності.

Проте цей голосовий асистент орієнтований переважно на китайський ринок, не підтримує українську мову та не здатен працювати в офлайн-режимі, що значно обмежує його універсальність для глобального використання.

Snips (рис. 1.6) – офлайн-асистент із відкритим кодом, який орієнтований на приватність користувача. Працює на локальному обладнанні, підтримує створення власних команд.



Рис.1.6 – Девайс який використовує Snips

Система Snips має такі беззаперечні переваги, як повністю локальна робота без необхідності підключення до інтернету, високий рівень конфіденційності даних користувача, а також підтримка створення кастомних сценаріїв взаємодії.

Утім, її розвиток практично припинився після придбання компанією Sonos, а обмежена документація створює труднощі при впровадженні та адаптації продукту до індивідуальних потреб.

Mycroft AI (рис. 1.7) – голосовий асистент з відкритим вихідним кодом. Працює на Raspberry Pi, ПК або спеціалізованих пристроях. Дає змогу створювати власні навички та адаптувати функціональність під потреби користувача.



Рис.1.7 – Девайс Mycroft AI

Асистент Microsoft AI відзначається повністю відкритим вихідним кодом, що дає змогу вільно модифікувати та розширювати функціональність системи. Він підтримує локальну роботу й забезпечує високу гнучкість у налаштуваннях.

До недоліків цього рішення можна віднести необхідність ручного налаштування та досить високий поріг входу для звичайного користувача, що ускладнює його використання без спеціальних технічних знань.

Для дослідження якості роботи аналогів потрібно використати методи порівняльного аналізу, емпіричний метод та SWOT-аналізу.

Метод порівняльного аналізу (табл. 1.1) дозволяє зіставити різні програмні продукти за заданими критеріями, щоб виявити їх сильні та слабкі сторони. Було проаналізовано 7 аналогів, а також сформована таблиця, яка дозволила побачити функціональні прогалини конкурентів. [5], [21].

У процесі аналізу оцінювались такі ключові аспекти, як: підтримка IoT-пристроїв, можливість офлайн-роботи, інтеграція з персональними комп'ютерами, підтримка української мови, гнучкість конфігурації, локальність даних та необхідність спеціального обладнання.

Таблиця 1.1

Порівняльний аналіз аналогів

Функціональні можливості	Android Auto	BMW Assistant	Teyes Voice	Alexa	XiaoAI	Snips	Mycroft AI	Lilith
Інтеграція з IoT	✗	✗	✗	✓	✓	✓	✓	✓
Інтеграція з ПК	✗	✗	✗	✓	✗	✓	✓	✓
Керування голосом	✓	✓	✓	✓	✓	✓	✓	✓
Офлайн-робота	✗	✓	✓	✗	✗	✓	✓	✓
Синтез мовлення (TTS)	✓	✓	✗	✓	✓	✓	✓	✓
Підтримка української мови	✓	✗	✗	✗	✗	✗	✗	✓

Порівняльний аналіз аналогів

Локальна безпека	✗	✓	✓	✗	✗	✓	✓	✓
Підтримка власних сценаріїв	✗	✗	✗	✓	✗	✓	✓	✓
Робота без акаунта / реєстрації	✓	✓	✓	✗	✗	✓	✓	✓
Працює повністю локально	✓	✗	✗	✓	✗	✓	✓	✓
Потреба у спец. обладнанні	✓	✓	✓	✓	✓	✗	✗	✗

Емпіричний метод у межах цієї кваліфікаційної роботи був використаний для практичного тестування голосових асистентів у максимально наближених до реальності умовах. Основна мета полягала у перевірці функціональності, точності розпізнавання мовлення, зручності використання, а також відповідності заявленим характеристикам. [6]

Тестування здійснювалося на одному ПК зі встановленою Windows 11, у звичайних побутових умовах з використанням вбудованого мікрофона ноутбука. Усі асистенти були перевірені на можливість взаємодії голосом, підтримку офлайн-режиму, відповідь українською мовою, а також зручність взаємодії без попередньої підготовки користувача.

Для оцінки голосових асистентів було визначено ряд метрик, які не лише відображають наявність певних функцій, але й характеризують якість їх реалізації. Точність розпізнавання мовлення оцінювалась у відсотках як кількість успішно виконаних команд із десяти можливих. Час реакції вимірювався як середній інтервал у секундах від моменту подачі команди до її виконання. Наявність офлайн-роботи перевірялась як здатність системи функціонувати без доступу до інтернету. Оцінювалась підтримка української мови, яка могла бути повною, частковою або відсутньою. Також визначалась необхідність реєстрації або створення облікового

запису для роботи з системою. Простота налаштування характеризувалась рівнем складності процесу, який міг бути низьким, середнім або високим. Останнім параметром виступала можливість кастомізації команд, що означало здатність користувача адаптувати систему до власних потреб, з можливими варіантами відповіді: так, ні або частково.

Результати оцінювання наведено в (таб. 1.2).

Таблиця 1.2

Емпіричний метод дослідження аналогів

Асистент	Точність розпізнавання	Час реакції	Офлайн-режим	Українська мова	Потреба в акаунті	Складність налаштування	Кастомізація команд
Android Auto	90%	1.2	Частково	Так	Так	Низька	Ні
BMW Assistant	88%	1.0	Частково	Ні	Так	Середня	Ні
Teyes Voice	78%	1.3	Так	Частково	Ні	Середня	Частково
Amazon Alexa	92%	1.5	Ні	Ні	Так	Низька	Так
Xiaomi XiaoAI	85%	1.4	Ні	Ні	Так	Висока	Частково
Snips	80%	1.1	Так	Частково	Ні	Середня	Так
Mycroft AI	84%	1.6	Так	Частково	Ні	Висока	Так
Lilith	90%	1.4	Так	Так	Так	Низька	Так

SWOT-аналіз у межах даного дослідження був використаний для оцінки сильних та слабких сторін голосових помічників, які розглядаються як потенційні аналоги розроблюваної системи. Це дозволило глибше зрозуміти конкурентне середовище, визначити функціональні прогалини, а також сформулювати стратегічне бачення для вдосконалення власного продукту. SWOT-аналіз ділиться на чотири основні аспекти:

- **S** (Strengths) – сильні сторони;
- **W** (Weaknesses) – слабкі сторони;
- **O** (Opportunities) – можливості розвитку;
- **T** (Threats) – потенційні загрози.

У межах проєкту SWOT-аналіз дозволив системно проаналізувати положення кожного з аналогів на ринку, оцінити їх переваги та недоліки, а також зрозуміти, які можливості можна реалізувати, виходячи з досвіду конкурентів.

Такий підхід допомагає визначити конкурентні переваги майбутнього продукту та сформулювати стратегію розвитку.

Таблиця 1.3

Оцінка аналогів голосових помічників за допомогою методу SWOT

Асистент	Сильні сторони (Strengths)	Слабкі сторони (Weaknesses)
Android Auto	Інтеграція з Google-сервісами; підтримка української мови; зручність інтерфейсу	Потребує смартфона; обов'язкове інтернет-з'єднання; закритість екосистеми
BMW Assistant	Глибока інтеграція з автомобілем; стабільність роботи	Обмежена доступність; відсутність української мови; неможливість кастомізації
Teyes Voice	Повноцінна офлайн-робота; швидка реакція; автоінтеграція	Працює лише у межах конкретного пристрою; обмежений функціонал
Amazon Alexa	Великий набір функцій; підтримка IoT; розширення через навички	Повна залежність від хмари; відсутність підтримки української мови
Snips	Повністю офлайн; відкритий код; приватність	Застарілий проєкт; обмежена підтримка мов; складне налаштування
Mycroft AI	Гнучкість; відкритий код; підтримка кастомних сценаріїв	Високий поріг входу; нестабільність роботи; часткова підтримка мов

Кожен із розглянутих аналогів має свою спеціалізацію та цільове середовище використання. Наприклад, Teyes, BMW Assistant та Android Auto орієнтовані на вбудовані автомобільні мультимедійні системи; Amazon Alexa і Xiaomi XiaoAI розроблені переважно для керування побутовими IoT-пристроями; тоді як Snips і Mycroft AI належать до гібридних рішень з відкритим кодом, орієнтованих як на побутове, так і персональне використання.

Проведене порівняння та SWOT-аналіз показали, що ефективний застосунок для керування пристроями голосовими командами повинен відповідати ряду ключових характеристик. Серед них – можливість офлайн-розпізнавання мовлення, що забезпечує незалежність від хмарних сервісів і покращує приватність даних. Сценарії виконання команд мають реалізовуватись локально, як для керування ПК, так і для взаємодії з IoT-пристроями. Для останніх важливо підтримувати відкриті протоколи, такі як MQTT чи HTTP API.

Також застосунок має бути розширюваним, що передбачає можливість додавання користувацьких команд. Інтерфейс повинен бути локалізований українською мовою для зручності користувачів. Крім того, важливим є підтримка взаємодії із зовнішніми API, що дозволяє здійснювати пошук інформації або виконувати інші дії на основі отриманих даних.

Саме ці принципи потрібно реалізувати в застосунку для керування пристроєм голосовими командами, який покликаний виправити функціональні недоліки, характерні для наявних рішень, та забезпечити користувачам максимальну автономність, приватність і зручність у використанні.

1.3. Вибір ефективної архітектури програмного агента на основі сучасних підходів

Для розробки голосового помічника було вибрано спроектувати застосунок на монолітній системі з модульною структурою.

Монолітна система з модульною структурою – виконується локально на клієнтському пристрої. Такий підхід дозволяє централізовано керувати всіма компонентами, зменшити складність розгортання, забезпечити високу продуктивність та автономність системи без залежності від постійного інтернет-з'єднання (за винятком запитів до зовнішніх API та IoT-пристроїв). [\[13\]](#), [\[14\]](#), [\[15\]](#)

Логіка голосового асистента реалізована як єдиний застосунок, у якому кожна функція виконується окремим модулем. Модуль голосової активації розпізнає ключове слово, яке активує асистента, дозволяючи запускати взаємодію без фізичного втручання. Модуль розпізнавання мовлення працює локально, перетворюючи голос на текст без використання хмарних сервісів, що забезпечує автономність і приватність.

Модуль обробки команд аналізує текст і визначає відповідну дію для виконання, а модуль локального виконання дій реалізує автоматизацію на рівні операційної системи, виконуючи дії, такі як запуск програм чи відкриття файлів.

Для запитів, які потребують доступу до онлайн-ресурсів, модуль роботи з API надсилає відповідні запити до зовнішніх сервісів і обробляє їх результати.

Останній компонент – IoT-модуль, що забезпечує інтеграцію з пристроями розумного будинку через протоколи, такі як HTTP або MQTT, надаючи користувачу можливість управляти такими пристроями голосовими командами. [3]

Принципи обраної архітектури забезпечують високу ефективність і зручність використання системи. Завдяки використанню локальних ресурсів клієнтського пристрою, забезпечується висока швидкість обробки запитів, що є критично важливим для голосових інтерфейсів. Відсутність залежності від хмарних сервісів сприяє автономності системи, знижуючи ризики, пов'язані з безпекою даних, і дозволяючи використовувати її у віддалених місцях без доступу до інтернету.

Масштабованість архітектури досягається через модульну структуру, яка дозволяє легко додавати нові функції або вдосконалювати існуючі без ризику порушення роботи інших компонентів. Це також забезпечує гнучкість у тестуванні та налагодженні, оскільки кожен модуль можна розробляти, тестувати й оновлювати окремо, що спрощує управління проєктом і прискорює розробку.

Окрему увагу приділено конфіденційності даних користувача: локальна обробка запитів мінімізує ризики витоку або несанкціонованого доступу до інформації, що робить систему надійною та безпечною для використання.

Обрана архітектура забезпечує ефективне використання ресурсів локального пристрою, стабільну роботу без постійного інтернет-з'єднання, гнучке масштабування та оновлення функціональності шляхом додавання нових модулів або зміни існуючих, а також створює сприятливі умови для поетапного тестування, налагодження та підтримки кожного окремого компонента в межах єдиного середовища розробки. [22]

Така структура дозволяє зберігати стабільність, простоту розробки та при цьому забезпечувати функціональну гнучкість, необхідну для сучасного голосового помічника.

1.4. Технічне обґрунтування: вибір інструментів, технологій та вимог до системи

Для реалізації голосового асистента було обрано мову програмування Python, модуль розпізнавання мовлення Vosk, модуль синтезу голосу Edge TTS, бібліотеку requests для взаємодії з API, інструменти os, subprocess і pyautogui для виконання системних дій, а також засоби paho-mqtt, socket і HTTP API для взаємодії з IoT-пристроями.

Python був обраний як основна мова розробки завдяки своїй простоті, великій кількості бібліотек для роботи з голосом, автоматизацією системи, мережевими протоколами та API. Він також добре підтримується на різних операційних системах.[\[42\]](#)

Vosk – це легка офлайн-бібліотека для автоматичного розпізнавання мовлення (ASR), яка забезпечує високу точність і підтримку кількох мов, зокрема української. Її перевагою є можливість працювати повністю без підключення до інтернету.[\[36\]](#)

Для генерації голосових відповідей використовується бібліотека edge-tts – клієнтська реалізація сучасного синтезу мовлення Microsoft Edge, яка працює через офіційні онлайн-сервіси Microsoft. Вона підтримує реалістичне відтворення мовлення багатьма мовами, включно з українською, і дозволяє легко інтегрувати якісний голосовий синтез у застосунки без необхідності глибокої налаштування.[\[33\]](#), [\[35\]](#).

Бібліотека requests використовується для надсилання HTTP-запитів до сторонніх API-сервісів. Вона забезпечує простий синтаксис, гнучке управління заголовками, параметрами та обробку відповідей, що робить її зручною для реалізації інформаційних запитів.

Для реалізації автоматизованого керування середовищем операційної системи застосовуються OS (модуль який дозволяє працювати з операційною системою) для базових файлових і системних операцій, subprocess для запуску зовнішніх процесів і скриптів, а також pyautogui для емуляції натискань клавіш,

рухів миші та взаємодії з графічним інтерфейсом, тоді як для керування IoT-пристроями використовуються протоколи `raho-mqtt` для обміну повідомленнями через MQTT-брокер.

Для розробки застосунку управління пристроєм голосовими командами потрібно використати низку інструментів, які сприяють підвищенню зручності та продуктивності розробки. Одним із ключових рішень стало використання середовища розробки PyCharm. Це потужне IDE, спеціалізоване для мови програмування Python, яке надає широкий набір інструментів для розробника – від інтелектуального автодоповнення коду до вбудованого дебагера, тестування та інтеграції з системами контролю версій.

PyCharm також забезпечує зручну навігацію по проєкту, перевірку якості коду, підтримку віртуальних середовищ і менеджерів пакетів, що значно спрощує організацію проєкту та розгортання. Завдяки цим можливостям середовище дозволяє ефективно працювати з усіма компонентами локального застосунку – від написання логіки обробки команд до інтеграції з API та управління IoT-пристроями.

Висновки до розділу 1

У даному розділі було сформульовано задачі для розробки локального голосового асистента, визначено основні функціональні вимоги та сценарії його використання. Проведено аналіз існуючих рішень, таких як Teyes Voice, Braina та Amazon Alexa, що дало змогу виділити ключові переваги та недоліки конкурентів і сформувати концепцію власного застосунку.

На основі проведеного аналізу було обґрунтовано вибір архітектури програмного забезпечення – монолітної модульної структури, яка дозволяє ефективно організувати роботу системи в межах одного пристрою з мінімальною залежністю від зовнішніх сервісів.

Також було обґрунтовано вибір інструментальних засобів: мови програмування Python, бібліотек Vosk, Edge TTS, requests, `os/subprocess/pyautogui`

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						25

для локального виконання дій, а також raio-mqtt та HTTP API для взаємодії з IoT-пристроями. Середовищем розробки було обрано PyCharm, що забезпечує зручність у написанні, тестуванні та підтримці коду.

Результатом виконаних етапів став план розробки архітектурної основи локального голосового помічника, здатного виконувати автоматизовані дії на комп'ютері, керувати розумними пристроями та отримувати інформацію через зовнішні API.

РОЗДІЛ 2. КОНЦЕПТУАЛЬНЕ ПРОЄКТУВАННЯ І АРХІТЕКТУРНА РЕАЛІЗАЦІЯ СИСТЕМИ

2.1. Моделювання сценаріїв використання: побудова діаграм прецедентів

Застосунок для керування пристроєм голосовими командами реалізує механізм природної взаємодії користувача з комп'ютером або розумними пристроями. Програма обробляє вхідні голосові запити, виконує відповідні дії в операційній системі, звертається до зовнішніх сервісів через API, а також забезпечує керування пристроями IoT через протоколи MQTT, HTTP або сокет-з'єднання.

Система функціонує в офлайн-режимі, зберігаючи основну частину обробки на локальному рівні. Це дозволяє забезпечити швидкий відгук, конфіденційність обробки запитів і гнучкість у розширенні функціональності. [\[28\]](#)

Для наочного представлення основних варіантів використання застосунку була побудована діаграма прецедентів (рис. 2.1). Вона демонструє ключові взаємодії між користувачем та функціональністю системи, включаючи розпізнавання мовлення, виконання команд, синтез відповіді, звернення до зовнішніх джерел даних та управління фізичними пристроями.

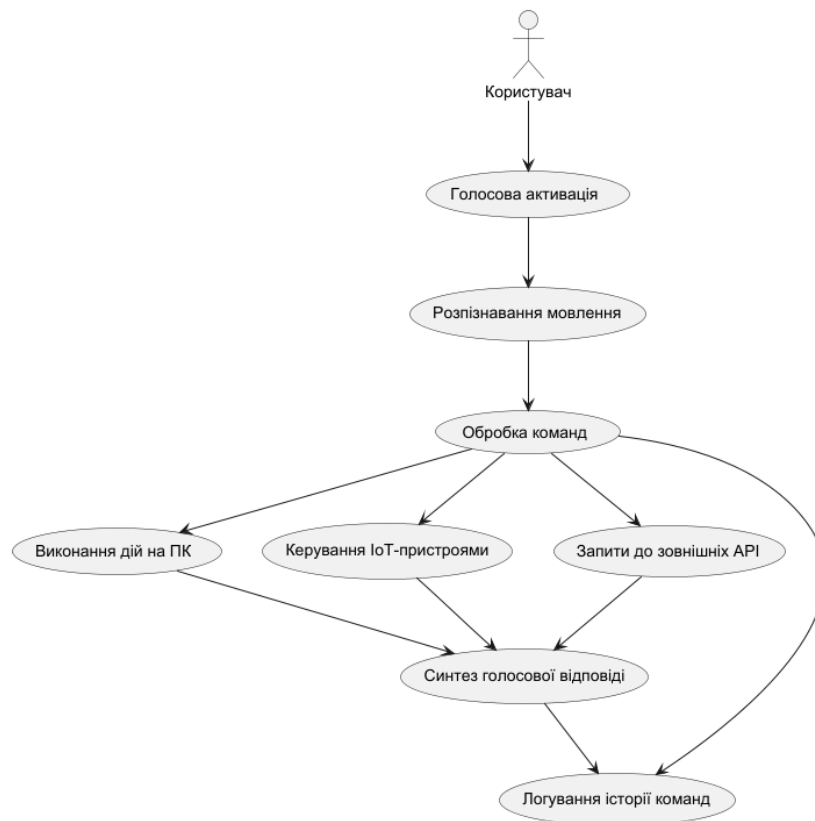


Рис. 2.1. Діаграма варіантів використання

Після побудови діаграми варіантів використання, доцільно детальніше описати основні сценарії взаємодії користувача з системою. Такий опис дозволяє краще зрозуміти логіку функціонування системи, її реакцію на вхідні події, а також забезпечити прозорість реалізації на етапі програмування.

Кожен варіант використання презентує окрему функціональність застосунку, пов'язану із перетворенням голосу на дію, відповідь або керування пристроєм. Нижче наведено таблиці з детальним описом кожного з них.

Прецедент розпізнавання мовлення (таб. 2.1) опису процес первинного перетворення мовлення користувача на текст. Цей текст є основою для подальшої інтерпретації та виконання команд. Розпізнавання мовлення є критичною функцією, що запускає увесь ланцюг обробки запиту. Лістинги модулів STT (Speech-to-Text) подано у Додатку В.

Таблиця 2.1

Опис прецеденту «Розпізнавання мовлення»

Назва варіанту використання	Розпізнавання мовлення
Опис	Користувач промовляє голосову команду, яка через мікрофон потрапляє до системи. Модуль Vosk аналізує мовний потік і перетворює його у текстову форму.
Передумови	Система активована та слухає; мікрофон підключений та справний
Післяумова	Вхідна мова перетворена на текст; отриманий текст передається для подальшої обробки
Використані технології	Vosk, Python, Microphone Input
Альтернативні потоки	Якщо мова нерозбірлива – система не реагує або просить повторити команду
Винятки	Відсутність звуку, несправність мікрофона, неправильне налаштування мови

Прецедент виконання команд на ПК (таб. 2.2) моделює взаємодію користувача із локальною операційною системою. Після успішного розпізнавання команди, система має вміти зіставити її з відомими інструкціями та виконати дію на рівні ОС.

Таблиця 2.2

Опис прецеденту «Виконання команд на ПК»

Назва варіанту використання	Виконання команд на ПК
Опис	Користувач вимовляє системну команду (наприклад, «відкрий калькулятор»). Система інтерпретує текст та виконує дію через API ОС або інструменти автоматизації.
Передумови	Система розпізнала та правильно інтерпретувала текст команди
Післяумова	Відповідна програма або дія виконана; користувач отримує підтвердження
Використані технології	Python (os, subprocess, pyautogui)
Альтернативні потоки	Якщо команда не розпізнана або відсутня у базі – асистент повідомляє про помилку

Прецедент синтезу мовлення (таб. 2.3) забезпечує зворотній зв'язок із користувачем. Після виконання дії система повинна згенерувати голосову відповідь, що підтверджує результат або надає додаткову інформацію. Це робить інтерфейс більш природним. Лістинги модулів TTS (Text-to-Speech) подано у Додатку В.

Таблиця 2.3

Опис прецеденту «Синтез мовлення»

Назва варіанту використання	Синтез мовлення
Опис	Після виконання дії система генерує аудіо-відповідь користувачеві, використовуючи текстовий результат, який озвучується через динаміки пристрою.
Передумови	Команда виконана успішно; є текст для озвучування
Післяумова	Користувач чує голосовий відгук системи
Використані технології	Edge TTS, Python, аудіо-драйвер
Альтернативні потоки	Якщо звуковий пристрій недоступний – відповідь може бути представлена текстова
Винятки	Відсутній доступ до звукових пристроїв, помилка синтезу

Прецедент взаємодії з API (таб. 2.4) демонструє, як система взаємодіє із зовнішніми веб-сервісами. У разі потреби отримання інформації, яка не зберігається локально, асистент виконує HTTP-запит, обробляє відповідь та представляє її користувачу.

Таблиця 2.4

Опис прецеденту «Запити до API»

Назва варіанту використання	Запити до API
Опис	Користувач ставить запит, система відправляє HTTP-запит до зовнішнього API, обробляє результат і озвучує/відображає відповідь.
Передумови	Інтернет-з'єднання активне; API доступне
Післяумова	Користувач отримує відповідь з API у зручній формі
Використані технології	Python (requests), сторонні API
Альтернативні потоки	У разі затримки відповіді – система повідомляє про очікування або тайм-аут
Винятки	Відсутність інтернету, помилка в API, некоректний формат відповіді

Прецедент керування IoT (таб. 2.5) пристроями демонструє можливості системи у сфері Інтернету речей (IoT). Голосова команда інтерпретується, і на її основі формується інструкція, що надсилається на підключений розумний пристрій.

Таблиця 2.5

Опис прецеденту «Керування IoT пристроями»

Назва варіанту використання	Керування IoT-пристроями
Опис	Користувач дає голосову команду на керування пристроєм. Система інтерпретує команду та надсилає відповідну інструкцію на IoT-пристрій.
Передумови	Пристрій зареєстрований у системі, підключений до локальної мережі
Післяумова	Стан пристрою змінено, користувач отримує підтвердження
Використані технології	MQTT (paho-mqtt), HTTP API, сокети
Альтернативні потоки	Якщо пристрій недоступний – система повідомляє про невдачу
Винятки	Відсутність підключення, неправильна адреса, тайм-аут підключення

Основна ціль розробки голосового асистента полягає у створенні системи, яка забезпечує зручну та ефективну взаємодію користувача з комп'ютером або IoT-пристроями голосовими командами. Такий асистент дозволяє автоматизувати повсякденні дії, полегшити виконання рутинних завдань, надати швидкий доступ до інформації та реалізувати управління розумним середовищем.

Метою проекту є розробка інструменту, що підтримує розпізнавання голосу, синтез голосу, виконання системних дій, взаємодію з API, а також управління IoT-пристроями через різні протоколи зв'язку (MQTT, сокети, HTTP). Система повинна бути зручною у використанні, легко адаптованою під потреби користувача і мати можливість масштабування.

Таким чином, головним завданням є забезпечення природної, голосової взаємодії між користувачем і технікою, що може покращити продуктивність,

зручність роботи з цифровими сервісами та підтримку розумного будинку чи робочого середовища.

Вимоги користувачів:

Єдиний внутрішній користувач – Користувач системи:

1. Виконання голосових команд для керування ПК;
2. Керування пристроями розумного будинку;
3. Отримання голосових відповідей на інформаційні запити;
4. Запуск сценаріїв або автоматизованих дій у системі;
5. Відтворення медіа, відкриття файлів, запуск програм;
6. Взаємодія з API для пошуку інформації, включно з AI-сервісами;
7. Активація помічника за ключовим словом;
8. Введення/вивід голосової інформації українською мовою.

Функціональні вимоги:

1. Розпізнавання голосових команд в офлайн-режимі;
2. Активація асистента за допомогою ключового слова;
3. Синтез мови у відповідь на команди користувача;
4. Підтримка виконання дій на рівні ОС;
5. Обробка запитів до зовнішніх API;
6. Керування IoT-пристроями через мережеві протоколи;
7. Конфігурування словника команд або дій користувачем.

Не функціональні вимоги.

1. Інтуїтивність та простота використання:
 - Користувач повинен мати змогу освоїти систему за 2-5 хвилин;
 - Голосовий інтерфейс не повинен вимагати технічних знань.
2. Швидкодія:
 - Час від розпізнавання до відповіді не більше 2 секунд;
 - Час відповіді API до 5 секунд (залежно від сервісу).
3. Надійність:
 - Система повинна бути стабільною у фоні без втручання користувача.

4. Автономність:

- Основна функціональність має працювати без доступу до інтернету (окрім API та IoT).

Системні вимоги.

1. Середовище виконання:

- Операційна система: Windows 11;
- Оперативна пам'ять: від 2 ГБ;
- Процесор: ≥ 1.4 GHz;
- Мікрофон та динамік для вводу/виводу голосу;
- Підключення до локальної мережі для IoT та до Інтернету для API.

2. Залежності:

- Python 3.10+;
- Модулі: vosk, pyttsx3, Edge TTS, paho-mqtt, requests, pyautogui, os, subprocess.

Аналіз функціональних вимог дозволив виділити наступні сутності, що забезпечать програмну реалізацію додатку.

2.2. Проектування логічної структури даних та інформаційних потоків

Для реалізації застосунку було розроблено логічну структуру бази даних, яка забезпечує зберігання та обробку необхідної інформації (рис. 2.2). База даних побудована на SQLite, що дозволяє їй працювати локально без залежності від зовнішніх серверів. Лістинг реалізації структури бази даних представлено в Додатку А. База даних застосунку для керування пристроєм голосовими командами складається з шести таблиць:

Commands – таблиця, яка зберігає доступні голосові команди, що можуть бути розпізнані та виконані асистентом.

Command_history – таблиця, яка зберігає історію виконаних команд: текст, дію, джерело та час виконання.

Settings – таблиця, яка зберігає налаштування користувача та системи (наприклад, мова, рівень гучності тощо).

Command_aliases – таблиця з альтернативними формулюваннями команд (синонімами), що підвищує точність розпізнавання.

Device_control – таблиця для збереження інформації про підключені IoT-пристрої.

User_preferences – таблиця для збереження особистих уподобань користувача, які виконуються автоматично за розкладом.[\[37\]](#)

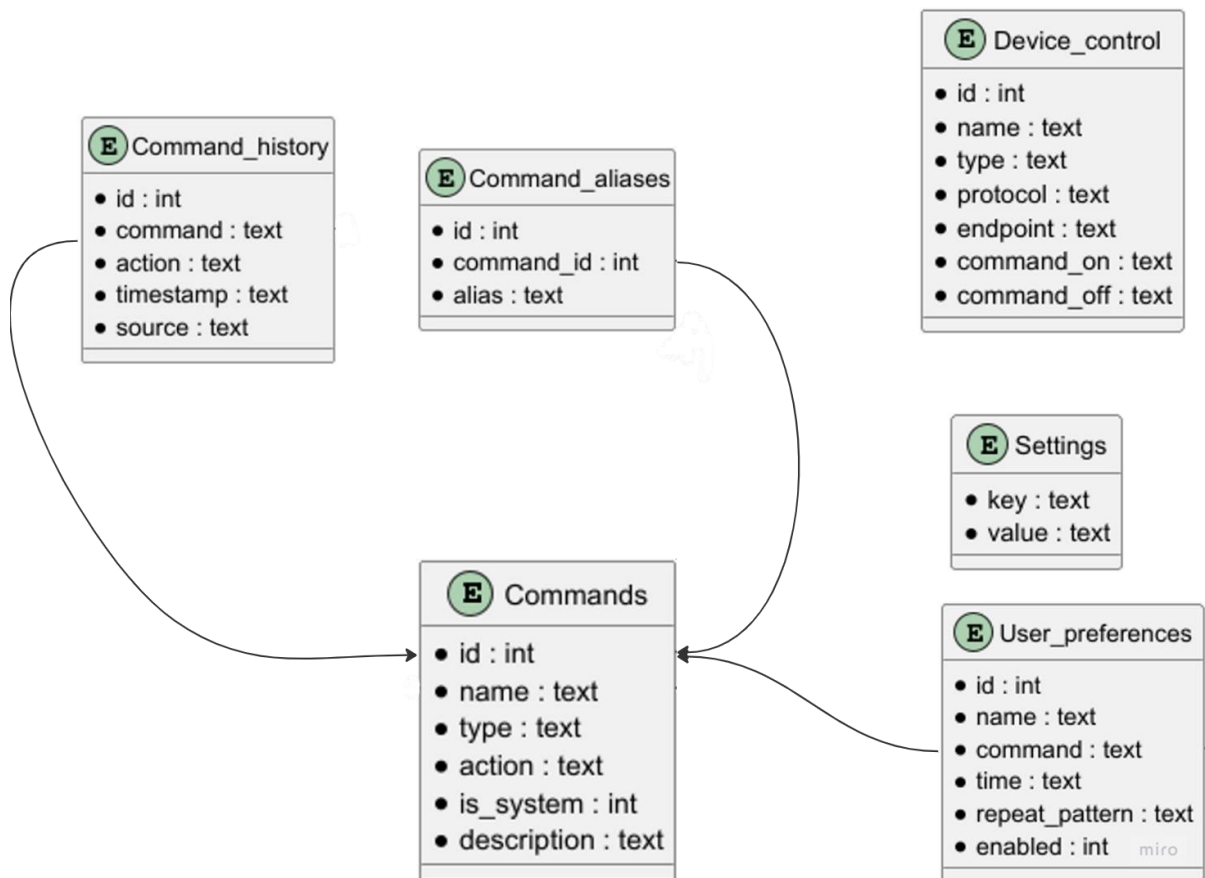


Рис. 2.2. Діаграма типів зав'язків між таблицями у базі даних

Таблиця Commands містить інформацію про всі команди, доступні в системі голосового управління. Вона використовується для зберігання ключових слів, які розпізнає система, їхніх типів, дій, що виконуються, та додаткових описів. Це основна таблиця, яка визначає функціонал і доступні команди для користувачів.

Таблиця 2.6

Опис таблиці Commands

Поле	Тип	Призначення
id	int	Ідентифікатор
name	text	Назва команди (ключове слово, яке вимовляє користувач)
type	text	Тип команди (наприклад: open, ctime.)
action	text	Конкретна дія: URL, шлях до програми або команда системи
is_system	int	Позначення, чи є команда системною (1 – так, 0 – ні)
description	text	Короткий опис дії, яку виконує команда

Таблиця Command_history використовується для зберігання історії виконаних голосових команд. Вона включає дані про те, коли команда була виконана, який текст команди було отримано, та яка дія відбулася в результаті. Це дозволяє відстежувати всі запити користувача та їх обробку.

Таблиця 2.7

Опис таблиці Command_history

Поле	Тип	Призначення
id	int	Ідентифікатор
command	text	Текст команди, сказаний користувачем
action	text	Яка дія була виконана
timestamp	text	Конкретна дія: URL, шлях до програми або команда системи

Таблиця Settings зберігає конфігураційні параметри системи. Вона використовується для налаштувань, які впливають на поведінку системи, таких як мова, гучність або інші глобальні опції. Кожен параметр має унікальний ключ і відповідне значення.

Таблиця 2.8

Опис таблиці Settings

Поле	Тип	Призначення
id	int	Ідентифікатор
key	text	Значення параметра

Таблиця `Command_aliases` містить альтернативні формулювання для існуючих команд. Вона покращує точність розпізнавання, дозволяючи системі реагувати на синоніми або варіанти ключових слів, які вводять користувачі. Зв'язок із таблицею `Commands` дозволяє визначати основну дію для кожного псевдоніма.

Таблиця 2.9

Опис таблиці `Command_aliases`

Поле	Тип	Призначення
id	int	Ідентифікатор
command_id	int	Посилання на основну команду (таблиця <code>commands</code>)
alias	text	Альтернативне формулювання команди

Таблиця `Device_control` використовується для зберігання інформації про підключені пристрої IoT. Вона визначає назву, тип пристрою, протокол взаємодії та команди для його активації або вимкнення. Це основне джерело даних для управління пристроями через систему голосових команд.

Таблиця 2.10

Опис таблиці `Device_control`

Поле	Тип	Призначення
id	int	Ідентифікатор
name	text	Назва пристрою
type	text	Тип пристрою (наприклад: light, thermostat)
protocol	text	Протокол взаємодії (MQTT, HTTP, тощо)
endpoint	text	Адреса для підключення або URL
command_on	text	Команда для активації пристрою
command_off	text	Команда для деактивації пристрою

Таблиця `User_preferences` зберігає персоналізовані налаштування користувача. Вона включає назви уподобань, команди, які потрібно виконати, час і частоту виконання, а також статус активності. Це дозволяє автоматизувати дії за розкладом і покращує користувацький досвід.

Таблиця 2.10

Опис таблиці `User_preferences`

Поле	Тип	Призначення
<code>id</code>	<code>int</code>	Ідентифікатор
<code>name</code>	<code>text</code>	Назва вподобання або дії (наприклад: «Ранкова кава»)
<code>command</code>	<code>text</code>	Яку команду потрібно виконати
<code>time</code>	<code>text</code>	Час виконання
<code>repeat_pattern</code>	<code>text</code>	Частота виконання
<code>enabled</code>	<code>int</code>	Чи увімкнене правило (1 – так, 0 – ні)

Використаний патерн проєктування MVC (рис. 2.3).

Model – модуль бази даних (`db.py`), який відповідає за взаємодію з таблицями, збереження, зчитування та оновлення інформації. Саме в моделі реалізовано логіку доступу до локальної бази даних `SQLite`, зокрема до таблиць `Commands`, `Command_history`, `User_preferences`, `Device_control`.

View – інтерфейс користувача, який може бути візуальним (графічний інтерфейс на базі `PySide6`) або голосовим (взаємодія через синтез і розпізнавання мовлення). Цей рівень відповідає за взаємодію користувача із застосунком і відображення результатів у зручній формі.

Controller – логіка обробки команд. Цей компонент приймає розпізнані текстові команди, виконує порівняння з доступними інструкціями в `config.json` або БД, визначає відповідну дію та ініціює її виконання.

Завдяки використанню цього патерну досягається чітке розділення відповідальностей, гнучкість в розробці, розширюваність та легкість у підтримці застосунку.

Такий підхід забезпечує збереження даних на пристрої, конфіденційність та гнучкість у розробці та використанні застосунку.

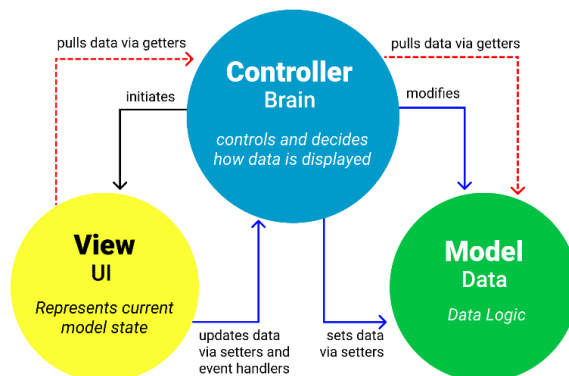


Рис. 2.3. Схема архітектури MVC патерну

Інформаційні потоки

Вхідний потік – користувач подає голосову команду через мікрофон після чого система розпізнає мовлення та перетворює його на текст і передає його до модуля обробки.

Обробка команди – система шукає відповідну дію в таблиці commands, якщо команда знайдена, вона виконується (локально, через API або IoT). Результат виконання зберігається в таблиці command_history.

Вихідний потік – Система може синтезувати голосову відповідь або виконати дію (наприклад, увімкнути світло). Користувач отримує підтвердження виконання команди.

У контексті роботи застосунку для керування пристроєм голосовими командами, дані зберігаються в локальній базі даних SQLite, яка розміщується безпосередньо на пристрої користувача. Це забезпечує повну автономність та швидкодію системи, а також дозволяє використовувати асистента без постійного підключення до мережі. [\[37\]](#)

База даних містить таблиці з командами, історією виконання та налаштуваннями. Команди можна редагувати та доповнювати безпосередньо через інтерфейс або шляхом синхронізації з зовнішніми джерелами.

Історія зберігає всі запити користувача, що дозволяє здійснювати аналітику або повторно виконувати попередні дії. Налаштування дають змогу персоналізувати поведінку асистента.

Взаємодія з базою даних відбувається через програмні методи, реалізовані в модулі роботи з БД. Ці методи дозволяють зчитувати зареєстровані голосові команди, додавати нові, зберігати історію запитів і налаштовувати поведінку системи відповідно до вподобань користувача (рис. 2.4).

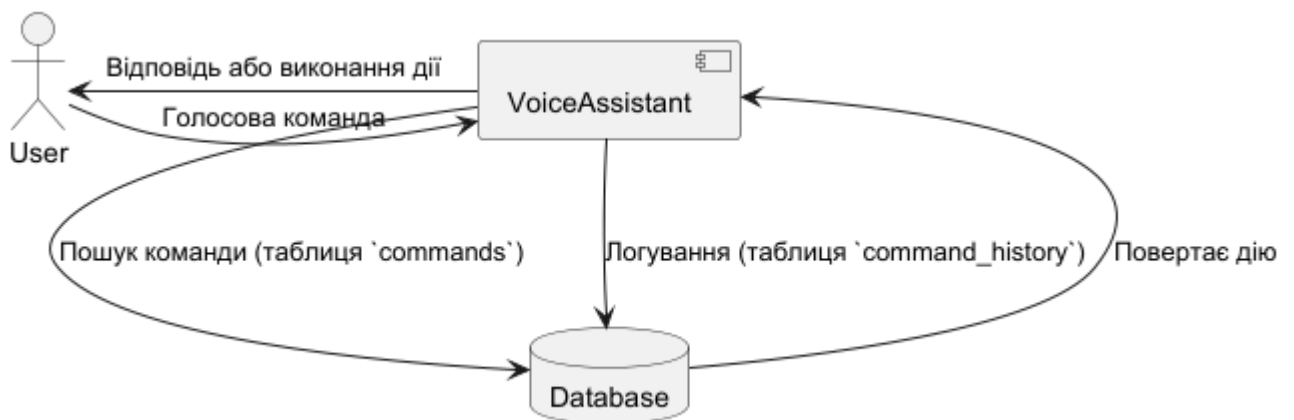


Рис. 2.4. Схема взаємодії модулів з базою даних

Такий підхід забезпечує збереження даних на пристрої, конфіденційність та гнучкість у розробці та використанні застосунку.

2.3. Архітектура програмних модулів та моделювання об'єктно-орієнтованої структури

У застосунку для керування пристроями голосовими командами архітектура побудована таким чином, що кожен модуль відповідає за окрему функціональність, що забезпечує високу гнучкість, простоту підтримки та можливість подальшого масштабування системи. Центральною частиною є модуль розпізнавання голосу, який відповідає за перетворення голосових команд у текст. Для цього використовується офлайн-бібліотека Vosk, а також реалізовано механізм активації

за ключовим словом, що дозволяє запускати взаємодію без фізичних дій користувача. [8]

Далі, після розпізнавання команди, текст обробляється модулем обробки команд, який аналізує отриману інформацію, визначає необхідні дії та виконує їх. Це може бути як взаємодія з локальними ресурсами комп'ютера, наприклад запуск програм або відкриття файлів, так і керування IoT-пристроями за допомогою мережевих протоколів MQTT, HTTP або сокетів.

Результати дій передаються до модуля синтезу мовлення, який формує голосову відповідь. Для цього використовується технологія Edge TTS, що дозволяє генерувати якісні аудіо-відповіді з підтримкою української мови, забезпечуючи комфортну взаємодію з користувачем. У разі потреби звернення до зовнішніх джерел інформації активується модуль API, який здійснює запити до вебсервісів, наприклад для отримання прогнозу погоди або відповідей на довідкові запити. Отримані дані передаються для подальшого озвучення або відображення.

Важливу роль у системі відіграє модуль керування IoT, що забезпечує зв'язок із пристроями розумного будинку. Він дозволяє надсилати команди за допомогою мережевих протоколів і забезпечує оперативне реагування на запити користувача. Усі дії, команди та налаштування зберігаються в модулі бази даних, реалізованому на основі SQLite. Це дозволяє працювати в офлайн-режимі та забезпечує конфіденційність і стабільність функціонування. [9]

Усю взаємодію між зазначеними модулями координує головний модуль системи. Він відповідає за ініціалізацію, керування потоком даних та організацію голосового інтерфейсу, через який користувач спілкується із системою. Така структура дозволяє легко додавати нові можливості, змінювати окремі компоненти без порушення загальної логіки та адаптувати застосунок під різні потреби користувачів.

Об'єктно-орієнтована структура

Система побудована на основі об'єктно-орієнтованого підходу (ООП), що забезпечує модульність, повторне використання коду, розширюваність та зручність

у підтримці проєкту. Кожен функціональний блок реалізовано у вигляді окремого класу, який інкапсулює власну відповідальність та логіку обробки. [43]

Діаграма класів (рис. 2.5) демонструє структуру класів і їх взаємозв'язки. Такий підхід не лише покращує читабельність та підтримуваність коду, але й спрощує додавання нових модулів, наприклад: підтримки інших мов, додаткових типів пристроїв або нових джерел інформації.

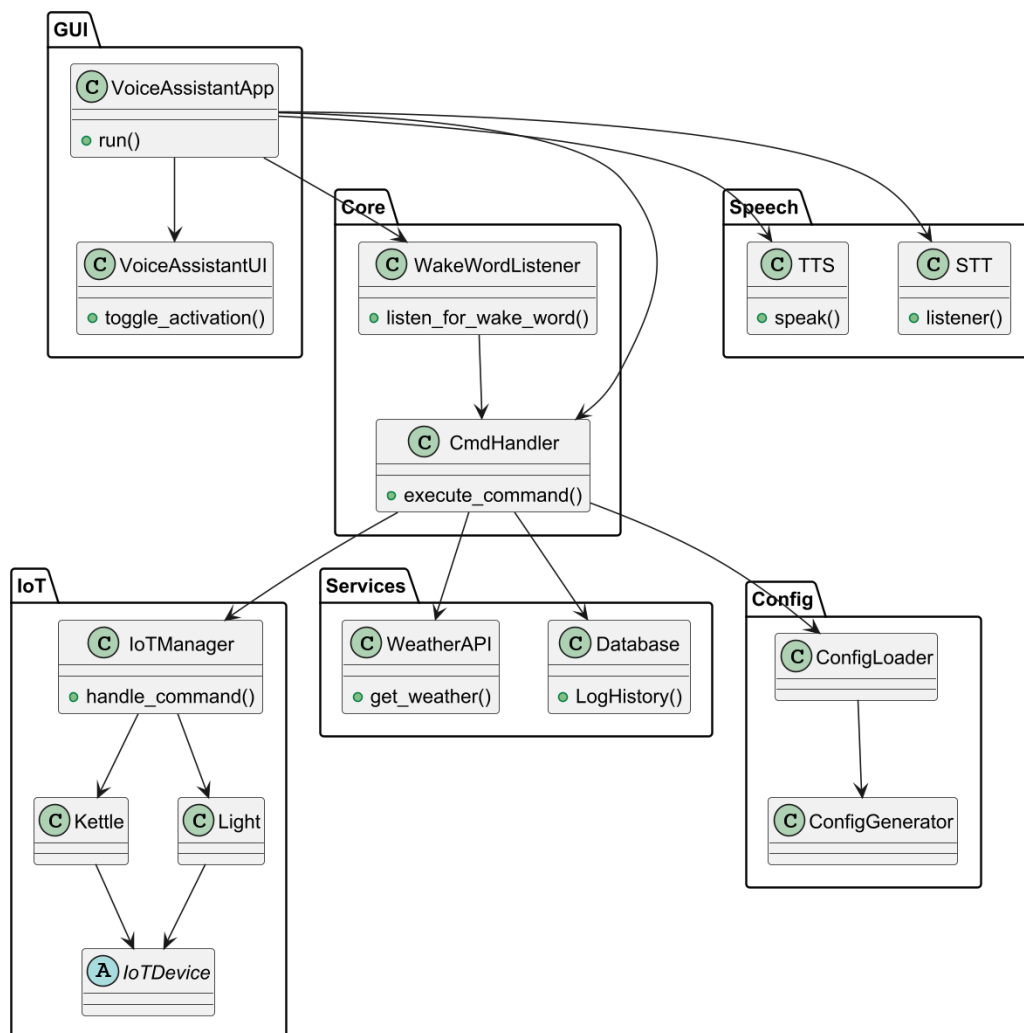


Рис. 2.5. Діаграма класів

Робота застосунку починається з ініціалізації інтерфейсу користувача. Модуль VoiceAssistantApp відповідає за запуск графічного інтерфейсу через метод run(), а інтерфейс реалізується компонентом VoiceAssistantUI, який відображає вікно з елементами керування, зокрема кнопкою активації. Після натискання цієї кнопки або при виявленні голосового тригера активується функція toggle_activation(), що розпочинає процес прослуховування.

Активация голосом реализована через модуль WakeWordListener, який безперервно прослуховує аудіопотік із мікрофона у фоновому режимі. Коли система виявляє задане ключове слово, спрацьовує ланцюг подій, що ініціює роботу модулів розпізнавання мовлення (STT) та синтезу мовлення (TTS), готуючи систему до взаємодії з користувачем. [\[12\]](#), [\[40\]](#).

Модуль STT, викликаючи метод listener(), отримує аудіо вхід і трансформує його у текстову форму. Цей текст передається на обробку до модуля CmdHandler, який відповідає за аналіз команди. CmdHandler звіряє розпізнаний текст із базою доступних команд, попередньо завантажених за допомогою модуля ConfigLoader. У разі відповідності команда передається на виконання.

У випадку, якщо розпізнана команда стосується керування пристроями Інтернету речей, активується модуль IoTManager. Цей компонент оперує конкретними пристроями, які реалізують абстрактний клас IoTDevice. Наприклад, коли користувач промовляє команду «Увімкни чайник», IoTManager викликає метод handle_command() відповідного пристрою, у даному випадку класу Kettle. Аналогічним чином відбувається керування іншими пристроями, зокрема освітленням, яке реалізоване через клас Light. Усі команди та результати їх виконання зберігаються в модулі бази даних.

Для коректної роботи системи з конфігураційними параметрами використовується ConfigLoader, який відповідає за завантаження словника команд і налаштувань із файлу config.json або з бази даних. У разі внесення змін або додавання нових команд, ConfigLoader може звертатися до модуля ConfigGenerator, який генерує оновлену конфігурацію. Такий механізм дозволяє динамічно адаптувати систему до нових умов без необхідності втручання в основну логіку роботи застосунку.

Архітектура застосунку побудована на модульній основі з чітким розмежуванням обов'язків. Використання ООП дозволяє легко додавати нові функції та адаптувати систему під потреби користувача. Кожен модуль працює незалежно, що спрощує тестування та налагодження. [\[43\]](#)

2.4. Алгоритмічна реалізація функціональних компонентів голосового інтерфейсу

Алгоритми функціонування застосунку для керування пристроєм голосовими командами реалізовано як набір взаємодіючих модулів, кожен з яких виконує свою чітко визначену функцію – від активації по ключовому слову до виконання дій на основі голосових команд, або запуску за розкладом.

Алгоритм 1 – Ініціалізація та генерація конфігурації, під час запуску головного скрипту `main.py` відбувається:

- Ініціалізація бази даних за допомогою `init_db()`, що створює всі необхідні таблиці;
- Генерація файлу конфігурації `config.json` з бази даних через `generate_config()`, що забезпечує динамічне оновлення команд користувача.

Алгоритм 2 – Слухання ключового слова активації, модуль `wake_word_listener.py` використовує модель Porcupine, яка постійно прослуховує мікрофон на предмет фрази-пробудження.

Якщо ключове слово розпізнано:

- Відтворюється сигнал готовності;
- Запускається слухач `listener()`, який передає потік в `stt.py`.

Алгоритм 3 – Розпізнавання мовлення та обробка команд, модуль `stt.py` на основі бібліотеки Vosk виконує перетворення аудіо в текст. Текст передається в обробник команд `va_respond` (рис. 2.6).

```
if rec.AcceptWaveform(data):  
    text = json.loads(rec.Result())["text"]
```

Рис. 2.6. Код для перетворення аудіо в текст

Алгоритм 4 – Пошук відповідної команди та виконання дії, функція `find_command()` з `cmd_handler.py` виконує нечітке зіставлення розпізнаного тексту

з командами з config.json (в якому команди групуються за типом). Для цього використовується бібліотека fuzzywuzzy (рис. 2.7).

```
for cmd_type, cmd_list in config.items():
    for cmd in cmd_list:
        score = fuzz.ratio(voice_input.lower(), cmd["name"].lower())
        if score > best_score:
            best_score = score
            best_result = {
                "type": cmd_type,
                "name": cmd["name"],
                "action": cmd["action"]
            }
```

Рис. 2.7. Код для нечіткого зіставлення бібліотеки «fuzzywuzzy»

Алгоритм 5 – Взаємодія з конфігурацією config.json формується автоматично на основі даних з таблиці commands бази даних (рис. 2.8). Це забезпечує швидкий доступ до команд без звернення до БД при кожному запиті, підвищуючи продуктивність.

```
{
  "open": [
    {
      "name": "відкрий браузер",
      "action": "https://www.google.com"
    }
  ],
}
```

Рис. 2.8. Конфігурація «config.json»

Алгоритм 6 – Логування команд після виконання будь-якої команди вона записується до таблиці command_history разом із часом виконання та джерелом. (рис. 2.9).

```
log_command(command="відкрий браузер", action="https://google.com")
```

Рис. 2.9. Логування команд

Розробка алгоритмів для голосового асистента передбачала ретельне проектування кожного етапу взаємодії з користувачем, починаючи від активації системи до виконання конкретних дій. Важливим аспектом стало забезпечення швидкості та точності роботи, а також можливість масштабування функціоналу.

Наприклад, алгоритм розпізнавання мовлення (ASR) базується на бібліотеці Vosk, яка забезпечує високу точність перетворення голосу в текст навіть у режимі офлайн. Це дозволило уникнути залежності від інтернет-з'єднання та зберегти конфіденційність даних користувача.

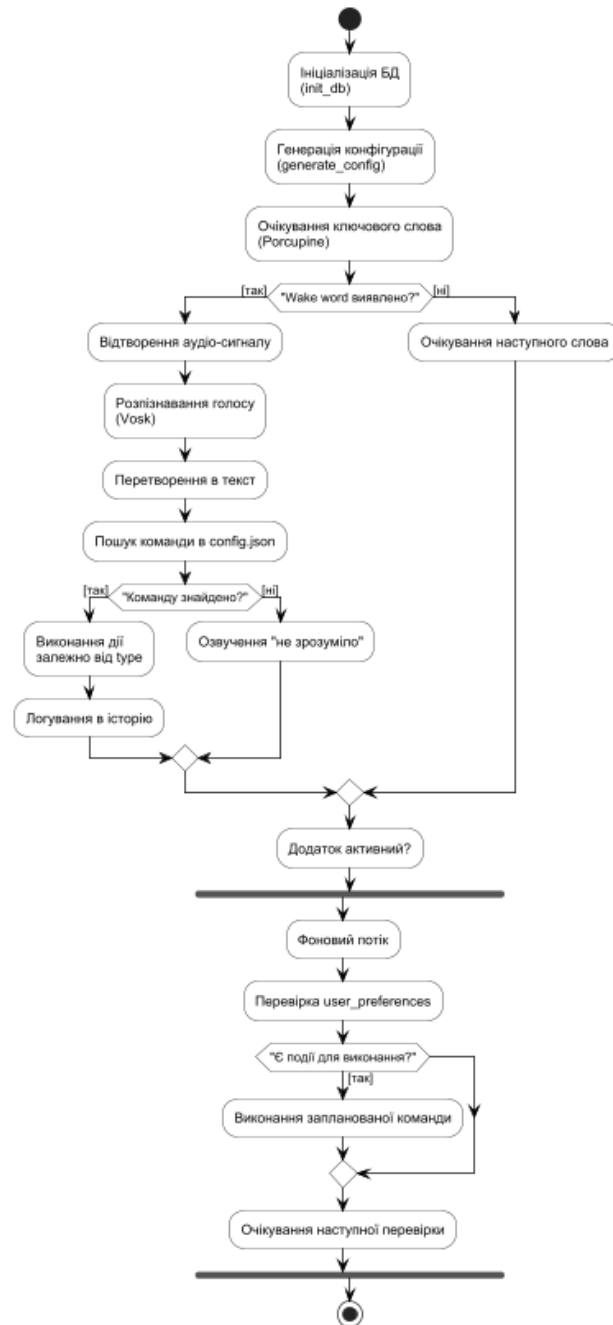


Рис. 2.10. Алгоритмічна блок-схема роботи застосунку

Алгоритмічна блок-схема (рис. 2.10) демонструє загальний процес роботи застосунку: Очікування ключового слова; Розпізнавання голосу → текст; Пошук

команди у config; Визначення дії за типом; Виконання команди; Логування результату; Запуск розкладу за user_preferences (у фоновому режимі).

На першому етапі розробки системи головним завданням стало створення мінімально життєздатного продукту (MVP – Minimum Viable Product), який демонструє основні функції голосового асистента. Цей прототип покликаний забезпечити ключову цінність системи – можливість безконтактної взаємодії з комп'ютером та IoT-пристроями за допомогою голосу, без підключення до хмарних сервісів.

До складу MVP входять наступні функціональні компоненти.

1. Активація по ключовому слову:

- Користувач викликає асистента, промовивши визначену фразу;
- Система перебуває в режимі прослуховування, поки не отримає сигнал активації.

2. Автономне розпізнавання мовлення:

- Реалізовано на основі бібліотеки Vosk, що дозволяє конвертувати голос у текст без доступу до інтернету.

3. Обробка команд і виконання дій:

- Реалізовано інтерпретацію голосових запитів, зіставлення з конфігураційним файлом та виконання відповідних дій – відкриття програм, запуск скриптів, керування файлами тощо.

4. Голосовий зворотній зв'язок:

- За допомогою Edge TTS система озвучує результати або підтвердження виконання команд.

5. Керування IoT-пристроями:

- Реалізовано взаємодію з пристроями розумного будинку через MQTT, HTTP або сокет-з'єднання.

6. Локальне логування дій:

- Усі виконані команди записуються до таблиці command_history, що дозволяє здійснювати аналіз або повторне виконання.

Важливим етапом реалізації MVP стала організація централізованої логіки обробки команд. Замість жорстко закодованих умов у коді, було розроблено підхід на основі конфігураційного файлу `config.json`, який формується з бази даних. Це дало змогу легко змінювати поведінку системи, додавати нові команди без перекомпіляції коду, що значно підвищує гнучкість та масштабованість рішення.

Ще одним ключовим компонентом стала реалізація розкладу виконання дій. Через таблицю `user_preferences` користувач може визначити дії, що мають виконуватись автоматично у заданий час, наприклад: ввімкнення світла вранці або відкриття браузера щодня о 9:00. Це дозволяє системі не лише реагувати на голосові команди, а й активно допомагати користувачеві.

Таким чином, MVP-система дозволяє вирішити основну задачу – забезпечити офлайн-голосове керування комп'ютером та IoT-пристроями, створивши базу для подальшого вдосконалення. Попри обмеженість функціоналу на даному етапі, вже реалізовані компоненти демонструють потенціал системи як універсального голосового помічника, що працює автономно, швидко й адаптивно.

Розробка MVP створила фундамент для наступних ітерацій розвитку системи, в межах яких планується значне розширення функціональності, гнучкості та масштабованості рішення. Подальші кроки спрямовані не лише на покращення користувацького досвіду, але й на вдосконалення внутрішньої архітектури, щоб система могла ефективно працювати в більш складних сценаріях.

На поточному етапі система використовує SQLite – легку реляційну базу даних, яка забезпечує збереження всієї необхідної інформації, такої як команди, історія дій, пристрої та вподобання користувача. Однак зі збільшенням обсягу даних і потреб у розширеному логуванні виникає необхідність переходу до гібридної структури бази даних. Це рішення дозволить більш ефективно управляти даними та підвищити продуктивність системи.

У рамках гібридного підходу планується зберегти реляційну базу даних (наприклад, PostgreSQL або залишити SQLite) для управління основними структурованими даними, такими як команди, налаштування та пристрої.

Одночасно передбачається інтеграція нереляційної бази даних (MongoDB або TinyDB) для зберігання динамічних і неструктурованих даних, зокрема журналів дій, статистики, сесійних даних та персоналізованих сценаріїв.

Такий підхід має ряд переваг. По-перше, це зменшить навантаження на реляційну базу даних, яка буде відповідати лише за критичну структуровану інформацію. По-друге, нереляційна база дозволить гнучко зберігати й аналізувати динамічні та неструктуровані дані, що відкриє нові можливості для розширеної аналітики. По-третє, така структура створить умови для реалізації інструментів машинного навчання, які зможуть працювати з великими обсягами історичних даних для прогнозування та адаптації системи до потреб користувача.

Таким чином, перехід до гібридної архітектури бази даних забезпечить більшу гнучкість, продуктивність і масштабованість системи, дозволяючи їй розвиватися разом зі збільшенням обсягів даних і складністю завдань.

З переходом логів у нереляційну БД відкривається можливість визначати часто вживані команди, генерувати пропозиції. Реалізувати сценарії «звичок», наприклад: автоматично запускати програму, яку користувач відкриває кожен ранок.

Ще одним важливим кроком є реалізація механізму підключення зовнішніх модулів (плагінів). Кожен плагін міститиме власний файл з описом команди, дії, додаткового інтерфейсу. Плагіни можна буде додавати через інтерфейс без редагування коду. Це дозволить стороннім розробникам створювати нові функції.

У рамках майбутнього розширення можливе підключення до GPT через OpenAI API для обробки складних запитів, синхронізація з календарями (Google Calendar, Outlook).

У подальшому планується ввести режим доступності для людей з обмеженими можливостями, який враховуватиме збільшені елементи інтерфейсу, простішу мову команд, голосові підказки, озвучування помилок, підтримку сліпих користувачів (наприклад, відсутність візуальних елементів).

Висновки до розділу 2

У другому розділі було здійснено комплексне проектування застосунку для керування пристроями голосовими командами, починаючи з визначення функціональних вимог та сценаріїв використання, і закінчуючи побудовою архітектурної структури з урахуванням майбутнього масштабування.

Було проаналізовано ключові сценарії взаємодії користувача із системою, створено діаграми прецедентів, які відображають логіку роботи голосового інтерфейсу. На основі цих сценаріїв сформовано структуру бази даних, яка охоплює всі аспекти функціонування асистента: команди, історію, налаштування, пристрої та персоналізовані вподобання користувача. Для забезпечення автономності та швидкодії було обрано SQLite як реляційну систему керування базами даних, що забезпечує повноцінну офлайн-роботу.

Здійснено моделювання архітектури на основі патерну MVC, а також реалізовано алгоритмічну модель функціональних компонентів, що охоплює ключові етапи: активацію голосом, розпізнавання мовлення, інтерпретацію команд, виконання дій, синтез мови та логування результатів. Запроваджено принципи модульності, розширюваності, інкапсуляції та автономності, що забезпечує легку адаптацію системи до змін і додавання нових можливостей без втрати стабільності.

Особливу увагу приділено реалізації MVP-рішення, яке охоплює базову функціональність голосового асистента – локальне розпізнавання мовлення, виконання команд, синтез відповідей, управління IoT-пристроями та автоматизацію через розклад. Також було закладено архітектурні основи для подальших ітерацій проєкту, серед яких – розділення баз даних на реляційну та нереляційну, впровадження персоналізованого логування, підтримка плагінів, інтеграція зі сторонніми API, а також адаптація для користувачів з обмеженими можливостями.

РОЗДІЛ 3. ІНТЕРАКТИВНА ВЗАЄМОДІЯ КОРИСТУВАЧА ЗІ СМАРТ-СИСТЕМОЮ: ІНТЕРФЕЙС І ВАЛІДАЦІЯ

3.1. Методологія встановлення та конфігурування інтелектуального застосунку

Встановлення та конфігурування інтелектуального застосунку Lilith виконується поетапно з урахуванням потреб локального середовища, офлайн-роботи, підтримки голосових команд та взаємодії з IoT (Рис. 3.1).

Методологія встановлення голосового асистента базується на принципах модульності, автономності та простоти налаштування. Це дозволяє запускати систему навіть користувачам без спеціальної технічної підготовки. На першому етапі передбачено встановлення всіх необхідних залежностей. Для початку роботи користувач повинен мати встановлену мову програмування Python версії 3.10 або вище, а також додаткові бібліотеки, перелік яких зазначено у файлі requirements.txt. Серед них – бібліотека vosk для офлайн-розпізнавання мовлення, edge-tts для синтезу мовлення, pygame і playsound для відтворення аудіо, fuzzywuzzy, json і requests для обробки команд, PySide6 для створення графічного інтерфейсу, а також paho-mqtt, socket і sqlite3 для інтеграції з IoT-пристроями та збереження даних у базі.

Після встановлення усіх залежностей і запуску головного файлу main.py (лістинг коду у Додатку Е) система автоматично виконує ініціалізацію бази даних за допомогою функції init_db() та створення початкової конфігурації команд через generate_config(). В результаті формується файл config.json, який містить перелік доступних команд, що дає змогу уникнути ручного створення таблиць і налаштувань, забезпечуючи повну готовність системи до використання одразу після інсталяції.

Наступним кроком є налаштування конфігурації. Всі доступні команди представлені у файлі config.json, який синхронізується з базою даних. Зміни до нього можуть вноситись користувачем через інтерфейс програми – зокрема, за

допомогою вкладки «Створити команду», безпосередньо у базі через графічний інтерфейс або SQL-запити, а також автоматично шляхом інтеграції з іншими сервісами.

Користувач має змогу змінювати візуальне оформлення інтерфейсу, обираючи одну з кількох доступних тем: світлу, темну, фіолетову, зелену або червону. Обрана тема зберігається в таблиці settings і використовується під час подальших запусків програми, що дозволяє персоналізувати зовнішній вигляд відповідно до власних уподобань.

Особливу увагу в архітектурі приділено підтримці підключення IoT-пристроїв. Через модуль device_control користувач може додавати нові пристрої, вказуючи їх назву, протокол взаємодії (HTTP, MQTT або socket), команди для ввімкнення та вимкнення, а також адресу відповідного сервісу. Завдяки цьому асистент набуває можливості керувати реальними об'єктами, такими як освітлення, розетки, кавоварки та інші елементи розумного будинку. [38]

Крім того, система підтримує автоматизацію дій за розкладом. Таблиця user_preferences дозволяє створювати правила, які виконуються у заданий час без участі користувача. Кожне вподобання має назву, команду, точний час активації, шаблон повторення (наприклад, щоденно або щопонеділка) та статус активності. Це відкриває можливість налаштування регулярних дій, таких як автоматичне приготування кави щодня о 8:00, що значно підвищує комфорт користування асистентом.

У фоновому режимі періодично перевіряється час, і в разі відповідності умові команда виконується автоматично.

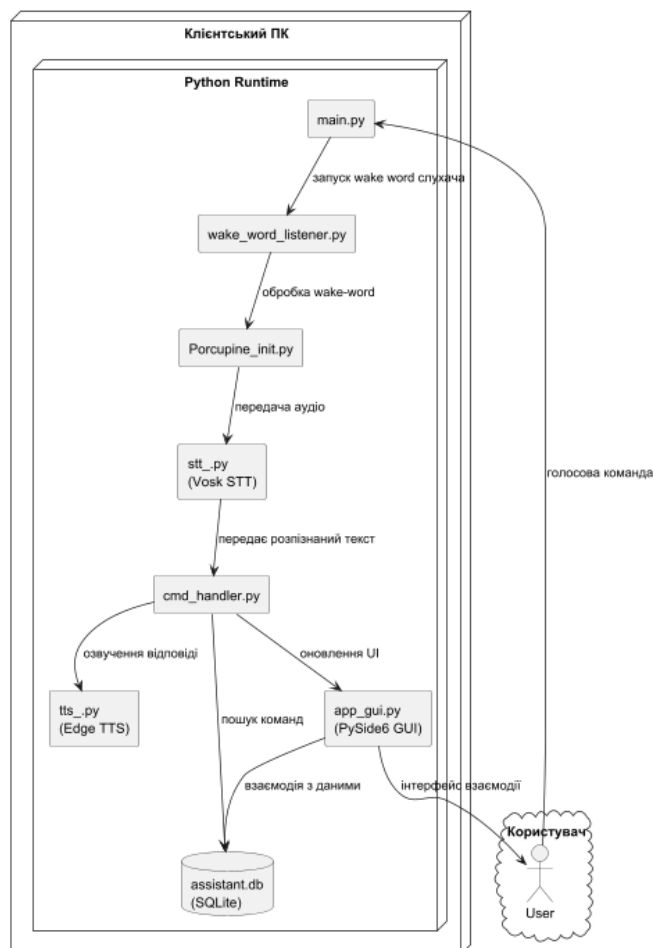


Рис. 3.1. Діаграма розгортання застосунку

Ця методологія дозволяє легко встановити, налаштувати та персоналізувати додаток для керування пристроєм голосовими командами, зберігаючи при цьому конфіденційність і повну автономність роботи без залежності від хмарних сервісів.

3.2. Структурно-функціональний аналіз інтерфейсу користувача

Графічний інтерфейс користувача є одним з ключових компонентів додатку, оскільки забезпечує зручну взаємодію з голосовим керуванням, управління командами, перегляд історії та налаштування системи. Для створення інтерфейсу використано бібліотеку PySide6, яка дозволяє реалізувати крос-платформний, сучасний дизайн. [\[13\]](#)

Інтерфейс поділено на кілька вкладок, кожна з яких відповідає за певну функціональність. Основна структура містить такі розділи як головна сторінка, список команд, історія команд, створити команду, налаштування.

Головна сторінка містить центральну анімовану візуалізацію нейромережі, яка символізує роботу голосового модуля. Крім того, на сторінці розміщено кнопку ручного запуску системи, що дозволяє ініціювати прослуховування команди користувача вручну (рис. 3.2).

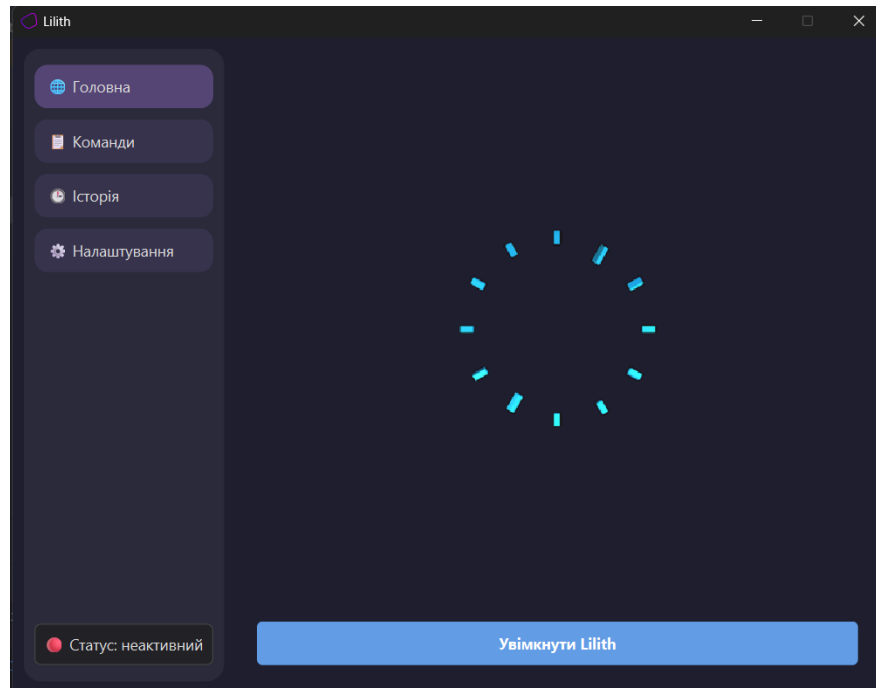


Рис. 3.2. Головна сторінка застосунку «Lilith»

Сторінка команд відображає всі наявні команди з бази даних. Для зручності реалізовано функцію пошуку, яка дозволяє швидко знаходити потрібну команду. Також передбачено відображення додаткової інформації про обрану команду. Кнопка «Додати» надає можливість користувачу створювати власні команди. (рис. 3.3)

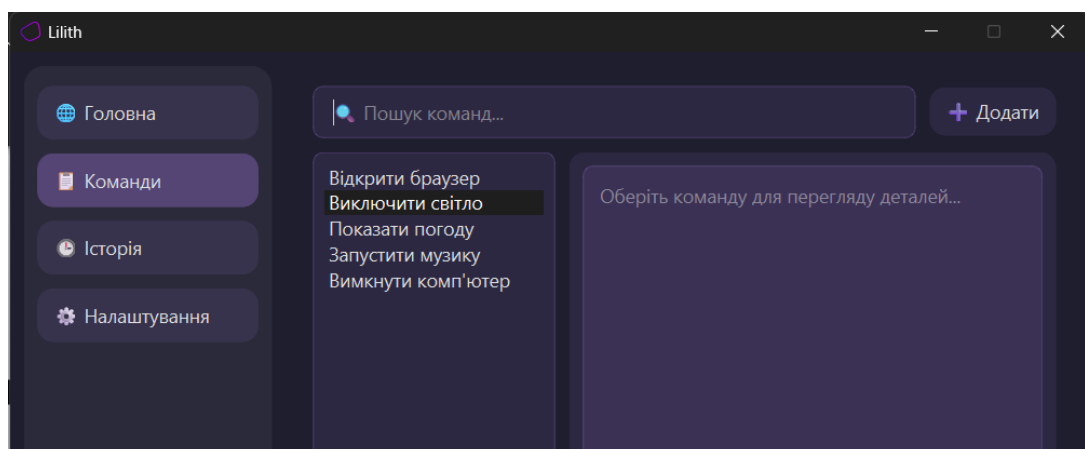


Рис. 3.3. Сторінка команд застосунку

Сторінка історії відображає таблицю з усіма виконаними командами у форматі: дата та час виконання, текст команди. У даній вкладці також передбачені фільтри, які дозволяють сортувати історію за часом та типом команди, що забезпечує зручність перегляду та аналізу. (рис. 3.4)

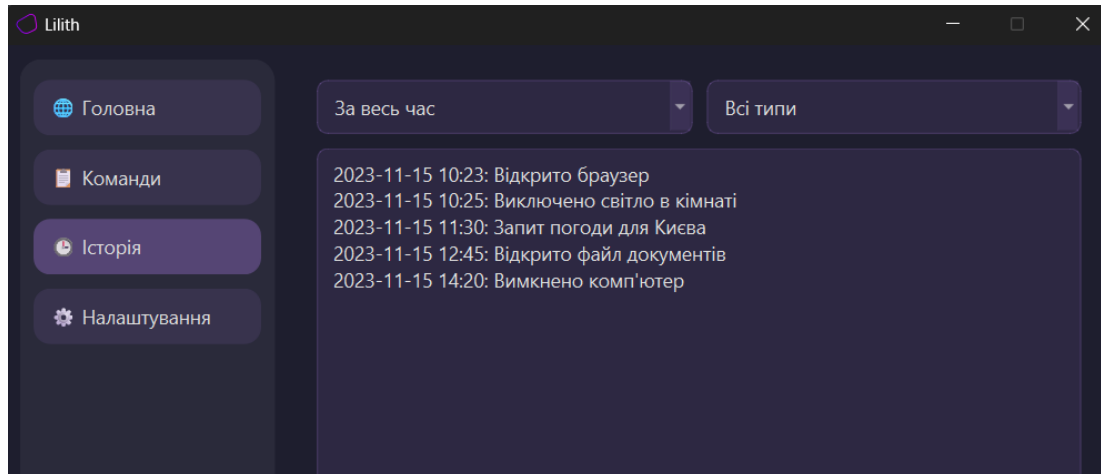


Рис. 3.4. Відображення історії запитів користувача

Сторінка налаштувань надає користувачеві можливість змінювати тему оформлення інтерфейсу, мову системи та параметри взаємодії з IoT-пристроями. Після вибору бажаних параметрів зміни можна зберегти за допомогою кнопки «Зберегти». Для повернення до початкових налаштувань передбачено кнопку «Скинути», яка відновлює стандартні параметри системи. (рис. 3.5)

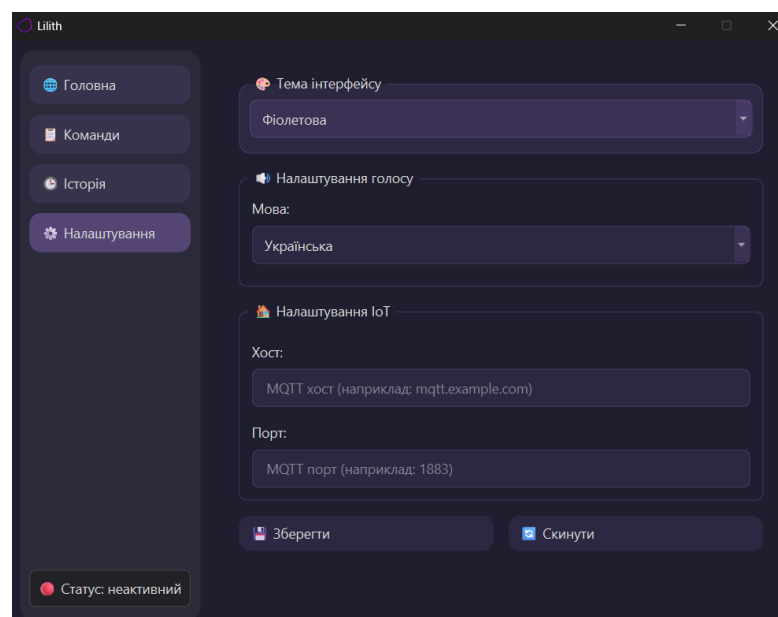


Рис. 3.5. Налаштування застосунку

Сторінка «Додати нову команду» дозволяє користувачеві створити власну голосову команду для автоматизації дій у системі. У спеціальній формі можна вказати назву команди, вибрати її тип, задати дію, та за потреби додати опис. Після заповнення форми команда зберігається в базу даних і стає доступною для виклику за допомогою голосу.(рис. 3.6)

Рис. 3.6. Форма для додавання нової команди

Інтерфейс користувача в додатку поєднує естетичність, функціональність та простоту використання. Кожен елемент розміщено логічно, а акцент на зручність взаємодії з голосовим керуванням робить застосунок доступним для широкого кола користувачів. Застосунок дозволяє не лише керувати пристроями, а й гнучко змінювати поведінку системи під потреби конкретного користувача. [\[10\]](#)

3.3. Методика тестування та верифікації ефективності програмної реалізації

Для забезпечення високої якості програмної реалізації додатку було застосовано систематичну методику тестування, яка включає функціональне, модульне, інтеграційне та користувацьке тестування. Тестування проводилось як у симульованому середовищі, так і в реальних умовах експлуатації. [\[41\]](#)

Функціональне тестування додатку мало на меті перевірку коректної роботи всіх основних функцій програмного забезпечення відповідно до специфікацій, визначених у технічному завданні та структурно-функціональній архітектурі системи.

Тестування системи проводилось поетапно та охоплювало ключові аспекти взаємодії між користувачем і програмним забезпеченням. Основну увагу було зосереджено на перевірці функціональності модулів, що відповідають за активацію, розпізнавання мовлення, обробку команд, синтез мовлення, інтеграцію з API та керування IoT-пристроями. Важливою складовою тестування була перевірка надійності та стабільності роботи кожного компонента як окремо, так і в межах єдиної системи.

Першим кроком стало тестування механізму активації за ключовим словом. Перевірки проводилися в різних умовах: у повній тиші, за наявності фонових шумів і при використанні різних голосів, що дозволило оцінити чутливість системи до зовнішнього середовища та її здатність коректно реагувати на задану фразу активації.

Далі було протестовано модуль розпізнавання мовлення, який відповідає за перетворення голосу на текст. Перевірялися команди різної довжини, слова з подібним звучанням, а також мовлення з акцентом або при пришвидшеній вимові. Це дозволило оцінити точність та адаптивність автоматичного розпізнавання в реальних умовах.

Особливу увагу приділили обробці команд. Система повинна коректно реагувати як на прості інструкції, наприклад відкриття програм чи керування файлами, так і на складні системні команди. Також перевірялися ситуації, коли користувач вводив невідомі або некоректні команди – система в таких випадках повинна була адекватно повідомляти про помилку або неможливість виконання.

Функціональність взаємодії з IoT-пристроями перевірялась шляхом увімкнення та вимкнення віртуального світильника, надсилання команд через локальний MQTT-сервер та виконання HTTP-запитів до системи автоматизації

Home Assistant для керування LED-пристроями. Таке тестування підтвердило здатність системи працювати з різними протоколами та сценаріями взаємодії з розумними пристроями.

Модуль синтезу мовлення оцінювався за параметрами природності голосу, швидкості реакції системи на запит, а також підтримки української мови. Голосовий відгук мав бути інформативним, зрозумілим і не супроводжуватись затримками чи технічними артефактами. [7]

Окремо тестувалась взаємодія з API. Було перевірено правильність формування HTTP-запитів, обробку отриманих текстових відповідей і подальше озвучування цих відповідей через голосовий інтерфейс. Система повинна була коректно виконувати запити до зовнішніх джерел і представляти результати користувачеві без затримок.

На завершення проводилась перевірка модуля роботи з історією команд та налаштуваннями. Система зберігала кожен запит у базі, відображала історію виконаних команд у графічному інтерфейсі, а також дозволяла змінювати тему оформлення інтерфейсу відповідно до вподобань користувача.

Таблиця 3.1

Результати функціонального тестування

Компонент	Очікувана поведінка	Фактична поведінка	Статус
Wake Word Detection	Реакція на ключове слово	Активується у 9 з 10 випадків	
Розпізнавання мовлення	Точний текстовий запис фрази	Точність 90%	
Обробка команд	Виконання відповідної дії	95% успішність	
Керування IoT	Увімкнення/вимкнення пристроїв	Без збоїв	
TTS	Голосова відповідь	Реалістичне озвучення	
API-запити	Отримання текстової інформації	Без збоїв	
Налаштування та історія	Запис команд, зміна тем	Відповідає очікуванням	

Усі основні функції системи були перевірені на коректність роботи при типових та граничних вхідних даних. Результати тестування підтвердили відповідність функціоналу заявленим вимогам.

Інтеграційне тестування має на меті перевірку узгодженості роботи окремих модулів системи після їх поєднання в єдину функціональну архітектуру. Це особливо важливо для застосунку Lilith, який реалізований як модульна система з чіткими зонами відповідальності – розпізнавання мовлення, обробка команд, взаємодія з базою даних, API, IoT-пристроями тощо.

У процесі інтеграційного тестування особливу увагу було приділено перевірці узгодженої взаємодії між ключовими модулями системи. Одним із найважливіших етапів стало тестування зв'язку між модулями активації по ключовому слову, розпізнавання мовлення (ASR) та обробки команд. Сценарій передбачав, що користувач промовляє ключову фразу, після чого система активується, відтворює звуковий сигнал і переходить у режим прослуховування. Успішне розпізнавання тексту та його передача до модуля обробки свідчили про правильну взаємодію всіх залучених компонентів.

Наступним важливим напрямом стала перевірка інтеграції модуля обробки команд із базою даних. Тестовий сценарій полягав у тому, що користувач вимовляє команду на зразок «Відкрий браузер». У відповідь система спочатку шукає відповідну інструкцію в конфігураційному файлі config.json, а в разі її відсутності – звертається до бази даних. Якщо знайдено збіг, команда виконується, що свідчить про належне функціонування механізму пошуку та збереження інструкцій.

Особливу увагу також приділено комунікації між обробником команд та модулями, відповідальними за інтеграцію з IoT-пристроями та зовнішніми сервісами. Так, наприклад, при команді «Увімкни світло» система формувала та надсилала MQTT-запит до відповідного пристрою. У випадку запиту типу «Яка погода?» здійснювався HTTP-запит до погодного API, а команда на кшталт «Увімкни кавоварку» запускала сценарій керування розумною технікою через

платформу Home Assistant. Усі ці перевірки підтвердили стабільність роботи мережевих протоколів і модульної логіки обробки.

Крім того, проводилась перевірка інтеграції графічного інтерфейсу з механізмами конфігурації команд. Користувач мав змогу створити нову голосову інструкцію безпосередньо через GUI. Після цього система перевіряла, чи збережено нову команду в конфігурації та базі даних, і чи здатна вона розпізнаватись голосовим інтерфейсом. Успішне виконання цих умов свідчило про повноцінну інтеграцію між інтерфейсом, базою даних і логікою обробки.

Завершальним етапом стало тестування взаємодії фонових процесів із головним циклом роботи асистента. Наприклад, для перевірки автоматизації було створено заплановану дію – «увімкнути світло о 8:00». Система у фоновому режимі перевіряла поточний час і, щойно настав запланований момент, виконувала відповідну команду. Така перевірка підтвердила здатність асистента працювати стабільно в автономному режимі та реагувати на заплановані події без участі користувача.

Таблиця 3.2

Результати інтеграційного тестування

Компонентна пара	Результат	Примітка
Wake Word ↔ ASR		Низька затримка, стабільна робота
ASR ↔ Command Handler		Текст передається без втрат
Command Handler ↔ IoT/API		Залежність від сторонніх сервісів
GUI ↔ Config/DB		Зміни доступні без перезапуску
User Preferences ↔ Scheduler		Висока точність виконання розкладу

Інтеграційне тестування підтвердило, що модулі системи працюють узгоджено, із мінімальною затримкою та без конфліктів. Найбільш чутливими виявилися компоненти, що залежать від зовнішніх джерел (API), але для офлайн-функцій інтеграція пройшла стабільно.

Тестування користувачем є важливою складовою оцінки якості програмного забезпечення, що дозволяє перевірити, наскільки зручною, інтуїтивною та ефективною є система з погляду реальних або умовних користувачів. Цей тип тестування дає змогу виявити недоліки, які часто залишаються непоміченими під час автоматизованих або модульних перевірок. У межах проєкту було проведено серію сеансів користувацького тестування із залученням двох осіб, які мали базовий досвід роботи з персональними комп'ютерами, але не були знайомі з технічними деталями реалізації асистента.

Основною метою тестування було перевірити зручність графічного інтерфейсу, оцінити зрозумілість голосового керування, логіку виконання дій, а також зібрати зворотний зв'язок стосовно загального враження від використання застосунку. Окрім цього, тестування допомогло виявити потенційні недоліки як у функціональності, так і в елементах інтерфейсу, які могли ускладнити взаємодію з програмою.

Тестування проводилось в мануальному режимі з використанням операційної системи Windows 11. Для взаємодії використовувався вбудований мікрофон. Система була попередньо налаштована: встановлено базовий набір голосових команд і конфігурацію керування IoT-пристроями, що забезпечувало імітацію реального сценарію використання.

Під час тестування учасники виконували низку завдань. Зокрема, вони активували асистента за допомогою ключової фрази, вимовляли базові команди, наприклад «відкрий браузер» або «котра година», взаємодіяли з віртуальним пристроєм шляхом голосового керування, додавали нову голосову команду через графічний інтерфейс, змінювали тему оформлення застосунку, переглядали історію виконаних дій та створювали правило автоматизації за розкладом.

Кожне завдання оцінювалося за кількома критеріями: наскільки інтуїтивно зрозумілий інтерфейс, чи легко користуватися голосовими функціями, як швидко система реагує на команди, наскільки точно виконує запити користувача, а також за рівнем загального задоволення від використання програми. Отримані результати

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						60

надали цінну інформацію для подальшого вдосконалення застосунку та підтвердили його готовність до використання реальними користувачами.

Таблиця 3.2

Результати інтеграційного тестування

Параметр	Середня оцінка (з 5)	Коментар
Зрозумілість інтерфейсу	4.5	Зручна структура, прості кнопки
Точність голосового розпізнавання	92%	Іноді не спрацьовує при тихому голосі
Швидкодія при виконанні дій	4.0	Відгук до 2 сек, затримка при API-запитах
Додавання нових команд	5.0	Повністю інтуїтивна форма
Зміна налаштувань (теми)	4.5	Все зрозуміло, але не вистачає підказок
Загальне враження	4.7	Користувачі оцінили додаток як «зрозумілий і сучасний»

У ході користувацького тестування було виявлено кілька недоліків, які впливають на зручність взаємодії з програмним продуктом. Зокрема, користувачі не одразу розуміли, чи система активна і готова сприймати голосові команди. Це спонукало до рішення додати візуальний індикатор активності, який чітко сигналізує про стан очікування.

Також було виявлено, що в темній темі оформлення окремі текстові елементи відображались з недостатньою контрастністю, що ускладнювало читання. У зв'язку з цим рекомендовано провести перегляд дизайну інтерфейсу для забезпечення кращої візуальної доступності. Крім того, при створенні нової команди через графічний інтерфейс список команд не оновлювався автоматично, що створювало ілюзію, ніби зміни не збережено. Цей недолік було усунуто шляхом додавання механізму автоматичного оновлення GUI після внесення змін.

Загалом проведене тестування підтвердило, що застосунок є зручним у користуванні, доступним навіть для людей без технічного досвіду та достатньо зрілим у функціональному плані для використання в реальних умовах. Отримані результати та відгуки дозволили зробити низку доопрацювань, які покращили якість продукту та підвищили комфорт користувачів.

3.4. Методика та результати тестування

У процесі створення та впровадження застосунку для керування пристроями голосовими командами особливу увагу було приділено тестуванню, що дозволяє виявити потенційні помилки, перевірити стабільність функціонування системи та переконатися у відповідності фактичної поведінки очікуваним результатам.

Тестування системи проводилося поетапно з використанням різних підходів, що охоплюють як технічні, так і користувацькі аспекти функціонування. На початковому етапі було застосовано unit-тестування для перевірки роботи окремих функцій і класів. Особливу увагу приділено модулям обробки голосових команд, перетворення мовлення в текст, а також взаємодії з API. Далі проводилося інтеграційне тестування, мета якого – перевірити взаємодію між ключовими модулями системи: ASR, NLP, IoT, UI та TTS. Це дозволило виявити і усунути помилки у їх взаємодії до початку повномасштабного тестування користувачем.

Основним методом перевірки з боку користувача стало ручне (manual) тестування, яке передбачало перевірку реальних сценаріїв використання: запуск програм, керування пристроями, отримання відповідей на запити. У процесі ручного тестування додатково застосовувалося exploratory testing, тобто вільна взаємодія з системою без попередньо визначених сценаріїв. Це дозволило виявити нестандартні ситуації та граничні випадки, які важко було передбачити заздалегідь.

Після внесення змін у логіку обробки команд проводилось регресійне тестування, мета якого – переконатися, що нові зміни не вплинули негативно на вже реалізовану функціональність.

Ці методи дозволили повністю охопити функціональність застосунку – від технічної реалізації до взаємодії з користувачем, та забезпечити впевненість у стабільній роботі всіх компонентів. [\[41\]](#)

Перевірка роботи голосових команд

Цей етап тестування був спрямований на перевірку базової функціональності: розпізнавання, інтерпретація та виконання команд користувача.

Тестувалися прості побутові команди, що імітують повсякденне використання системи.

Таблиця 3.3

Перевірка роботи голосових команд

Назва тесту	Умова виконання	Очікувано	Фактично	Статус
Запуск програми	Команда: «відкрий блокнот»	Запускається блокнот Windows	Блокнот відкрився	Пройдено
Закриття програми	Команда: «закрий калькулятор»	Вікно калькулятора закривається	Програма закрита	Пройдено

Для демонстрації результатів успішного виконання тест-кейсу було проведено послідовність дій із взаємодії з голосовим асистентом «Lilith». На початковому етапі користувач відкриває головне вікно застосунку та натискає кнопку «Увімкнути Lilith». У відповідь система подає голосове сповіщення про успішну активацію та готовність до виконання голосових команд (рис. 3.7).

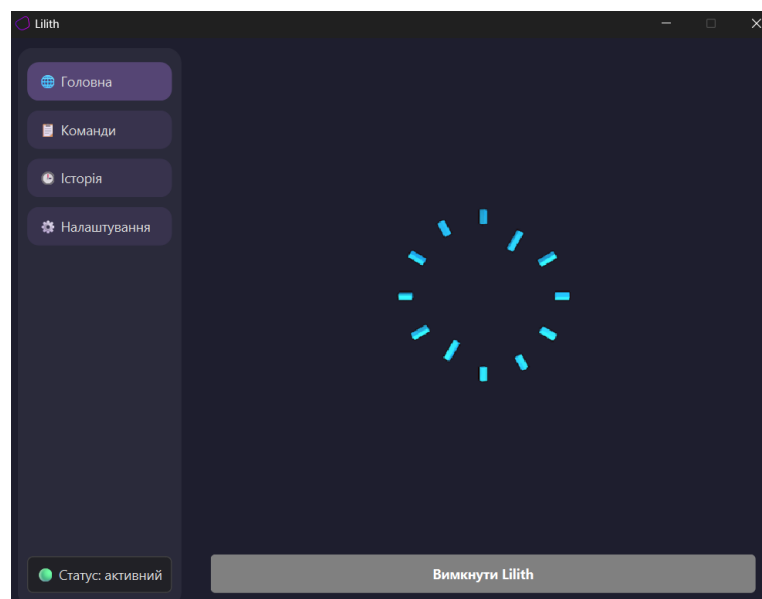


Рис. 3.7. Активація асистента «Lilith» на головній сторінці застосунку

Після активації, користувач промовляє ключове слово «Lilith», що активує режим очікування голосової команди. Система підтверджує готовність до взаємодії. Далі промовляється команда: «Відкрий блокнот», яка успішно розпізнається та виконується. У консолі виводиться текст розпізнаної команди, а також підтвердження виконаної дії (рис. 3.8).

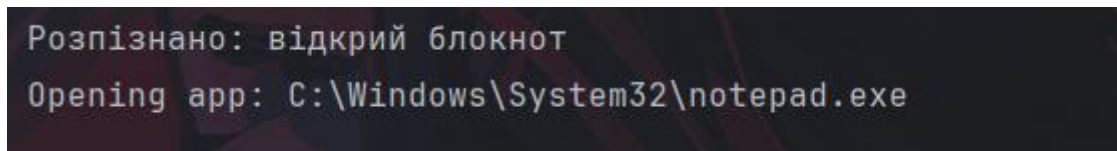


Рис. 3.8. Виведення розпізнаної команди в консоль та запуск відповідної дії

У результаті виконання команди, на екрані користувача відкривається стандартний застосунок «Блокнот», що підтверджує коректну роботу механізму виконання дій за голосовими запитами (рис. 3.9).

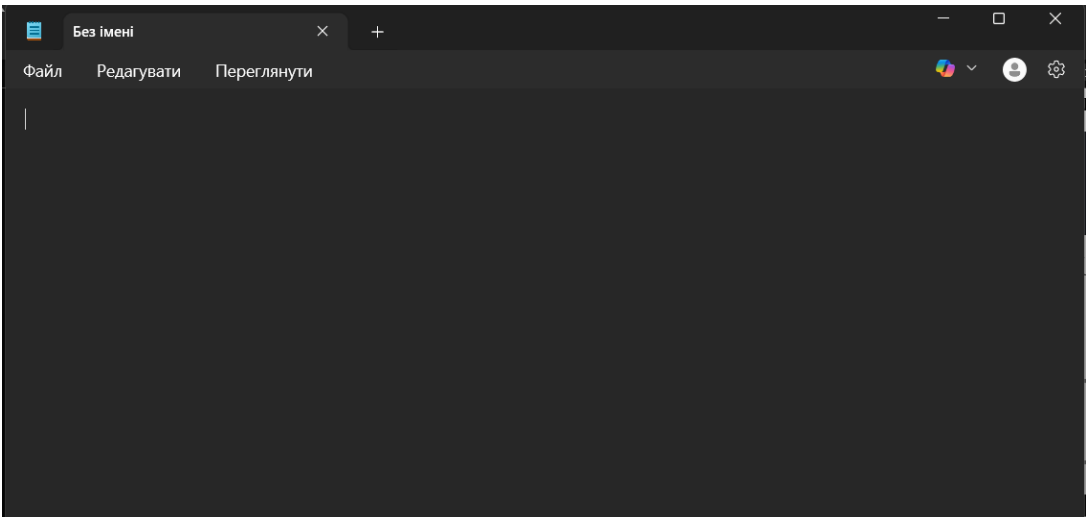


Рис. 3.9. Вікно блокноту, відкритого через голосову команду

Після виконання команди, її запис автоматично фіксується в базі даних у вигляді історії. Відповідна інформація відображається у графічному інтерфейсі у вкладці з журналом виконаних команд, що дозволяє користувачеві переглядати історію взаємодії з системою (рис. 3.10).

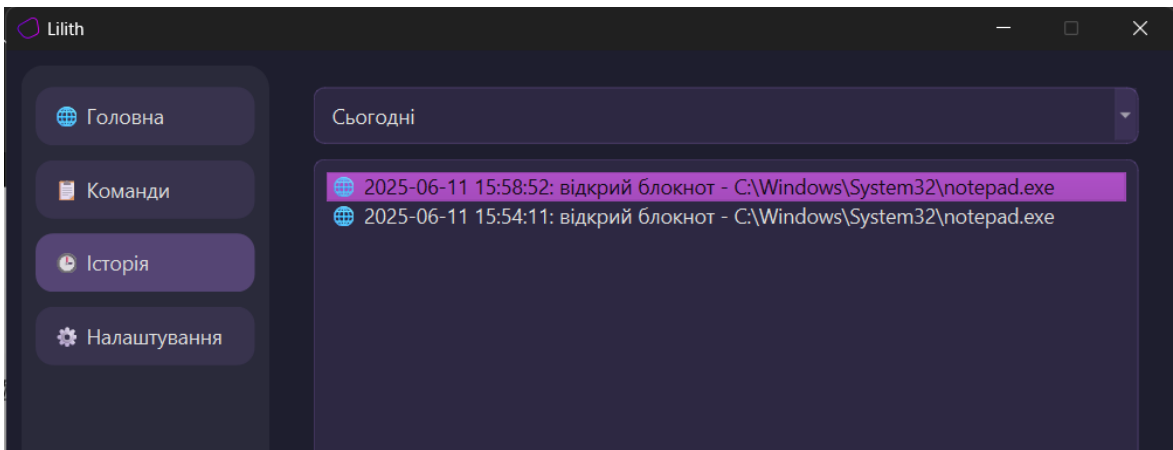


Рис. 3.10. Вікно з відображенням історії виконаних команд у застосунку

Результат виконання тесту – успішний. Система стабільно активується за допомогою клавіші та ключового слова, коректно розпізнає голосову команду українською мовою, виконує дію на рівні операційної системи та фіксує результат у базі даних.

Тестування синтезу мовлення (TTS)

Цей тип тестів охоплює модуль озвучування відповідей системи. Мета – перевірити, чи система може коректно передавати результати виконання команди користувачу через голосовий канал.

Таблиця 3.4

Тестування модуля синтезу мовлення

Назва тесту	Вхідні дані	Очікувано	Фактично	Статус
Озвучення погоди	«Погода»	«Температура 20 градусів»	Голосова відповідь з прогнозом	Пройдено
Помилка – не знайдено команду	«Що таке ластик?»	«Команду не знайдено»	Озвучено про помилку	Пройдено

Одним із тест-кейсів, реалізованих у рамках перевірки функціональності голосового асистента, є отримання голосового прогнозу погоди. Для цього користувач активує систему та промовляє команду: «Погода». Система розпізнає ключове слово та ініціює запит до відповідного API.

У результаті користувач отримує голосову відповідь із погодними даними яка генерується у реальному часі за допомогою модуля синтезу мовлення. Після виконання команди, вона також фіксується і заноситься до бази даних (рис. 3.11).

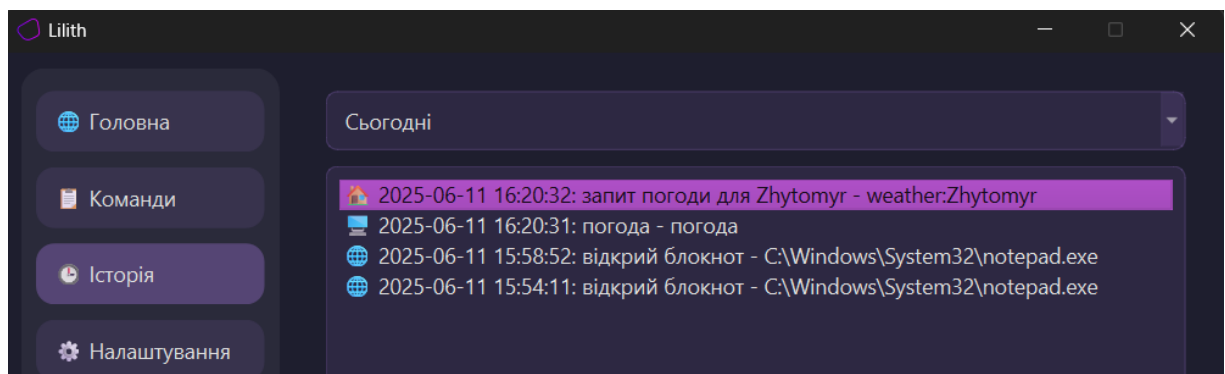


Рис. 3.11. Відображення виконаної команди в історії

Результат виконання тесту – успішний. Асистент правильно обробляє команду «Погода», здійснює запит до зовнішнього API, генерує голосову відповідь синтезованою українською мовою та забезпечує швидке озвучення актуальної інформації.

Тестування керування IoT-пристроями

Особлива увага приділялась взаємодії з розумними пристроями через мережу. Застосунок має підтримувати як MQTT-протокол, так і HTTP-запити. Нижче наведено результати перевірки IoT-функціональності.

Таблиця 3.5

Тестування взаємодії з IoT-пристроями

Назва тесту	Умова	Очікувано	Фактично	Статус
Вимкнення лампи	«Вимкни світло»	MQTT-повідомлення + OFF	Пристрій відповів OFF	Пройдено
Увімкнення чайнику	«Увімкни чайник»	MQTT-повідомлення + ON	Пристрій відповів ON	Пройдено

Окремий етап тестування був присвячений перевірці можливостей голосового асистента щодо керування IoT-пристроями.

У межах даного тест-кейсу було здійснено команду керування пристроєм «розумний чайник». Користувач після активації асистента промовляє фразу «Увімкни чайник» (рис. 3.12).

```
[MQTT] Спроба підключення до test.mosquitto.org:1883...
[MQTT] Успішно підключено до брокера test.mosquitto.org:1883
Розпізнано: увімкни чайник
```

Рис. 3.12. Виведення статусу підключення в консоль та розпізнана команда

Система коректно розпізнає голосову команду, виконує аналіз через модуль обробки команд та надсилає MQTT-повідомлення з параметром ON на адресу пристрою (рис. 3.13).

```
[IoTManager] Чайник увімкнено
[MQTT] Відправлено до home/kettle: on
[MQTT] Повідомлення опубліковано (ID: 1)
```

Рис. 3.13. Виведення в консоль результату виконання

Результат виконання тесту – успішний. Система продемонструвала стабільну роботу з протоколом MQTT, підтримку реального часу та інтеграцію з фізичними пристроями у локальній мережі.

Тестування інтерфейсу користувача (UI)

Користувацький інтерфейс був протестований на предмет зручності, доступності, а також коректної реакції на взаємодії користувача, зокрема при помилках.

Таблиця 3.6

Тестування інтерфейсу користувача

Назва елементу	Дія користувача	Очікувано	Фактично	Статус
Кнопка активації	Натискання в GUI	Асистент активується	Активовано прослуховування	Пройдено
Повідомлення про статус	Система не слухає/слухає	Показано поточний стан	Відображається коректно	Пройдено
Кнопка додавання команди	Натискання в GUI	Відкривається відповідне вікно	Відкривається вікно додавання команди	Пройдено

У межах перевірки функціональності графічного інтерфейсу користувача (UI) було протестовано сценарій додавання нової голосової команди. Метою є перевірка логіки відкриття відповідних інтерфейсних елементів, їх доступності та стабільності роботи при послідовних діях користувача.

На першому етапі користувач відкриває вкладку «Команди» у головному меню застосунку. Інтерфейс реагує миттєво, відображаючи таблицю наявних голосових інструкцій (рис. 3.14).

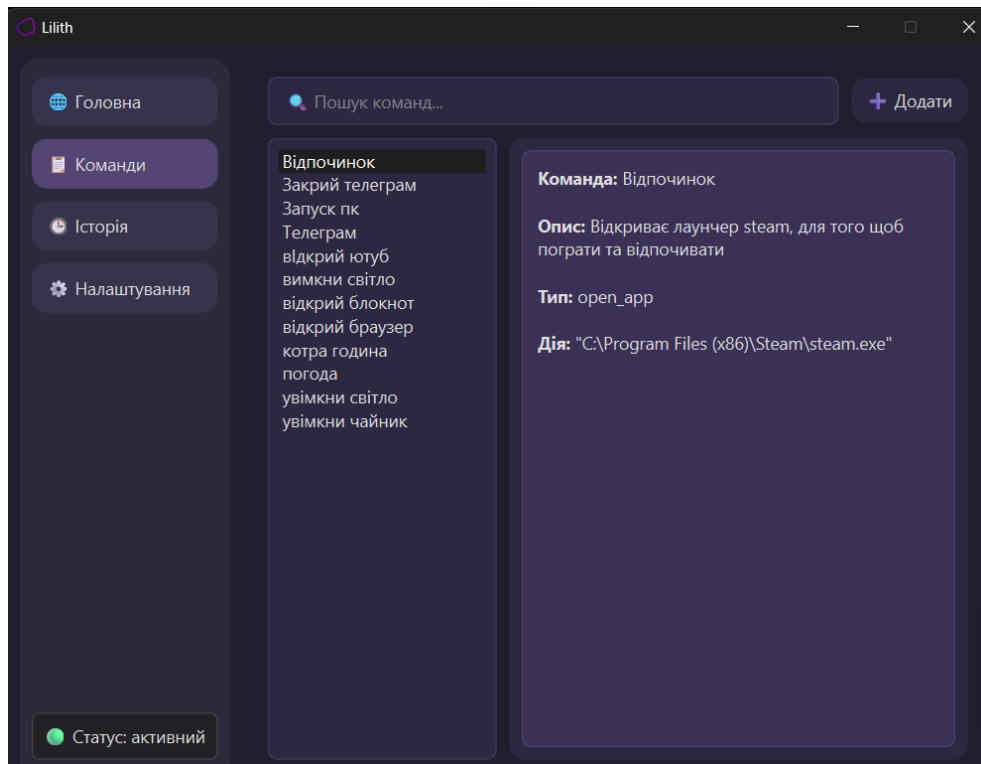


Рис. 3.14 – Перехід до вкладки з наявними голосовими командами

Далі користувач натискає кнопку «Додати команду», що розміщена у верхній частині відповідної вкладки. Система реєструє подію та виконує ініціалізацію інтерфейсу для введення нової команди. У відповідь відкривається форма додавання нової голосової команди, яка містить усі необхідні поля для заповнення (рис. 3.15).

Рис. 3.15 – Форма для створення нової голосової команди

Результати тестування показали, що всі основні модулі системи – від обробки голосу до інтеграції з IoT-пристроями – функціонують стабільно та відповідають очікуваному сценарію використання. Використання різних методів тестування (unit, integration, manual, exploratory, regression) дозволило виявити і усунути низку дрібних недоліків ще на етапі розробки, що позитивно вплинуло на загальну якість продукту.

Система показала високу точність у розпізнаванні команд, коректну обробку запитів та надійність у взаємодії з локальними пристроями. Інтерфейс користувача продемонстрував простоту та зручність використання, що робить застосунок доступним навіть для користувачів без спеціальної технічної підготовки.

Таким чином, проведене тестування підтвердило функціональну готовність застосунку до використання в реальних умовах та забезпечило необхідну впевненість у його стабільній роботі.

Висновки до розділу 3

У третьому розділі було розглянуто процес розгортання, налаштування, валідації та тестування застосунку, призначеного для голосового управління пристроями. Основна увага приділялася практичному підтвердженню ефективності розробленої системи, її зручності для користувача та функціональній стабільності.

Було описано поетапну методологію встановлення програмного забезпечення, включаючи ініціалізацію бази даних, налаштування голосових команд, конфігурацію IoT-пристроїв, персоналізацію інтерфейсу та запуск сценаріїв за розкладом. Така структурована процедура дозволяє швидко адаптувати застосунок під конкретного користувача без потреби у технічних знаннях.

Проаналізовано графічний інтерфейс, реалізований за допомогою PySide6. Інтерфейс виявився інтуїтивним, логічно структурованим і достатньо гнучким для індивідуальних налаштувань. Користувач може легко додавати нові команди, керувати історією запитів, змінювати тему оформлення та параметри взаємодії з пристроями.

У рамках функціонального тестування підтверджено правильну роботу ключових модулів системи – від розпізнавання мовлення (ASR) до виконання дій, звернення до API та генерації голосових відповідей. Проведено модульне тестування, яке забезпечило виявлення помилок на рівні окремих компонентів. Інтеграційне тестування засвідчило, що всі модулі взаємодіють стабільно та узгоджено в єдиному середовищі.

					ІПЗ.КР.Б – 121 – 25 – ІПЗ	Арк.
						70

ВИСНОВКИ

У рамках даної роботи було розроблено застосунок для керування пристроєм голосовими командами, який забезпечує безконтактну взаємодію з комп'ютером та пристроями (IoT). Головною метою проєкту стало створення системи, яка працює локально, без потреби у постійному підключенні до хмарних сервісів, підтримує українську мову, дозволяє кастомізувати сценарії взаємодії та орієнтована на потреби користувачів з обмеженими можливостями.

На початку дослідження було визначено низку проблем сучасного стану голосового керування: недостатня автономність рішень, відсутність підтримки української мови, обмежена адаптивність до користувача, а також закритість більшості екосистем. В процесі дослідження було досліджено понад 7 аналогів, проведено SWOT-аналіз і порівняльне тестування за ключовими критеріями: офлайн-робота, підтримка IoT, розширюваність, локальність даних, підтримка мов.

У технічній частині було реалізовано повноцінну архітектуру застосунку, що складається з декількох незалежних модулів: розпізнавання мовлення (ASR), синтезу мовлення (TTS), обробки команд, виконання дій на рівні ОС, керування IoT-пристроями та логування результатів. [\[1\]](#)

Було розроблено MVP (Minimum Viable Product), який уже забезпечує повноцінну взаємодію з користувачем через голосові команди: запуск програм, виконання дій на ПК, керування пристроями розумного дому, генерація голосових відповідей, ведення журналу дій, підтримка автоматизації через розклад. Це створює платформу для подальших ітерацій.

Особливу увагу приділено проектуванню бази даних: розроблено 6 основних таблиць для збереження команд, історії, уподобань, синонімів та налаштувань. Застосунок реалізовано за шаблоном MVC, що забезпечує гнучкість та масштабованість архітектури. Команди конфігуруються через config.json, який генерується на основі БД, що дозволяє змінювати логіку роботи без перекомпіляції або перезапуску системи.

Визначено і описано напрями подальшого розвитку системи: розгалуження бази даних на реляційну та нереляційну, підтримка плагінів, інтеграція з AI-сервісами, створення режиму доступності та модуль аналітики.

Розроблений застосунок має не лише прикладне значення, але й високу соціальну значущість. Його реалізація дозволяє створити умови для використання комп'ютера або розумного середовища людьми з інвалідністю, зменшує бар'єри цифрової взаємодії, сприяє інклюзії та доступності технологій. Крім того, система повністю дотримується принципу приватності: дані не передаються у хмару, не потребують реєстрації або авторизації.

Таким чином, результати цієї роботи свідчать про успішну реалізацію технічно складної, функціонально повної та соціально орієнтованої системи. Створений застосунок може бути основою для майбутніх досліджень, комерційних або open-source проєктів, а також використаний як базова платформа для подальшого впровадження в навчальних, побутових та професійних сферах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ :

1. Ковальчук О. М. Сучасні технології розпізнавання мовлення. Київ : Наукова думка, 2021. 245 с.
2. Петренко В. І. Штучний інтелект в обробці природної мови. Львів : Видавництво Львівської політехніки, 2020. 312 с.
3. Іванов П. С. Розвиток IoT технологій у сучасних інформаційних системах // Вісник НТУУ «КПІ». 2022. № 4. С. 56–63.
4. Сидоренко Л. М. Вплив голосових технологій на взаємодію з розумними пристроями // Наукові записки УКУ. 2023. № 7. С. 112–125.
5. Мельник А. В. Методи порівняльного аналізу програмних продуктів : дис. ... канд. техн. наук. Київ, 2022. 180 с.
6. Шевченко І. П. Емпіричні методи дослідження програмного забезпечення : автореф. дис. ... канд. техн. наук. Харків, 2021. 24 с.
7. Білецький А. В. Алгоритми розпізнавання мовлення для української мови // Вісник НТУУ «КПІ». 2021. № 3. С. 45–52.
8. Григоренко С. П. Архітектура голосових асистентів // Наукові записки УКУ. 2022. № 12. С. 112–125.
9. Сидоренко М. К. Розробка системи голосового керування для IoT-пристроїв : дис. ... канд. техн. наук. Київ, 2021. 180 с.
10. Kumar A., Tripathi A., Saini R. A Survey on Voice User Interfaces: Design, Development, and Challenges // Journal of Information and Optimization Sciences. 2022. Vol. 43, No. 6. P. 1261–1276.
11. Матеріали Міжнародної конференції з штучного інтелекту (ICAI 2023). Київ, 2023. 456 с.
12. Спосіб розпізнавання мовлення : пат. 12345 Україна. № UA 12345; опубл. 2021.
13. Гнатюк С. Моделювання користувацьких інтерфейсів у ПЗ // Наука та інновації. 2020. № 4. С. 43–50.
14. Остапенко В. І. Розробка розумних систем керування для побутових пристроїв // Інформаційні технології і засоби навчання. 2022. № 3(91). С. 21–29.
15. Яковенко П. І. Інтелектуальні системи в інтерфейсах «людина–машина». Харків : ХНУРЕ, 2021. 240 с.

- 16.Назаренко А. О. Голосове управління: архітектура та технології // Вісник ХНУРЕ. 2023. № 5. С. 18–24.
- 17.ДСТУ ISO 9241-210:2019 Ергономіка взаємодії людини з системою.
- 18.ДСТУ 8302:2015 Бібліографічне посилання. Загальні положення.
- 19.Про інформацію : Закон України від 02.10.1992 № 2657-XII.
URL: <https://zakon.rada.gov.ua/laws/show/2657-12> (дата звернення: 15.11.2023).
- 20.Про захист персональних даних : Закон України від 01.06.2010 № 2297-VI.
URL: <https://zakon.rada.gov.ua/laws/show/2297-17> (дата звернення: 15.11.2023).
- 21.Андрушко Ю. С. Порівняльний аналіз мовних інтерфейсів // Інформаційні технології. 2022. № 2. С. 70–77.
- 22.Вакарчук Л. І. Системи локального голосового керування: переваги та обмеження // Науковий журнал НУ «Львівська політехніка». 2023. № 6. С. 90–96.
- 23.Jurafsky D., Martin J. H. Speech and Language Processing. 3rd ed. Stanford University, 2023. URL: <https://web.stanford.edu/~jurafsky/slp3/> (дата звернення: 15.11.2023).
- 24.Rabiner L. R. A tutorial on hidden Markov models // IEEE Proceedings. 1989. Vol. 77(2). P. 257–286.
- 25.Goodfellow I., Bengio Y., Courville A. Deep Learning. Cambridge : MIT Press, 2016. 775 p.
- 26.Alkhateeb F. et al. Smart Home Voice Assistants // IEEE Access. 2020. Vol. 8. P. 153–169.
- 27.Zhou Y. et al. Design of IoT-based Smart Home Systems // IEEE Internet of Things Journal. 2021. Vol. 8(12). P. 9921–9934.
- 28.Tang Y. et al. An Offline Voice Assistant Using Edge AI // Sensors. 2021. Vol. 21(5). P. 1–18.
- 29.Nielsen J. Usability Engineering. San Francisco : Morgan Kaufmann, 1993. 362 p.
- 30.LeCun Y., Bengio Y., Hinton G. Deep Learning // Nature. 2015. Vol. 521. P. 436–444.
- 31.Sommerville I. Software Engineering. 10th ed. Boston : Pearson, 2016. 816 p.

- 32.Boehm B. Spiral Model of Software Development // ACM SIGSOFT Notes. 1986. Vol. 11(4). P. 14–24.
- 33.Smith J. Text-to-Speech Synthesis. Cambridge : MIT Press, 2019. 320 p.
- 34.Smith J. Natural Language Processing in Action. New York : Manning, 2019. 544 p.
- 35.Microsoft Azure. Speech Services. URL: <https://azure.microsoft.com/en-us/products/ai-services/ai-speech/> (дата звернення: 15.11.2023).
- 36.AlphaCephei. Vosk Speech Toolkit. URL: <https://alphacephei.com/vosk/> (дата звернення: 15.11.2023).
- 37.SQLite Documentation. URL: <https://sqlite.org/docs.html> (дата звернення: 15.11.2023).
- 38.MQTT Protocol Specification. URL: <https://mqtt.org/> (дата звернення: 15.11.2023).
- 39.OpenAI Whisper Model. URL: <https://github.com/openai/whisper> (дата звернення: 15.11.2023).
- 40.Picovoice Porcupine Wake Word.
URL: <https://github.com/Picovoice/porcupine> (дата звернення: 15.11.2023).
- 41.ISO/IEC/IEEE 29119-1:2013 Software testing standard.
- 42.Lutz M. Learning Python. 5th ed. Sebastopol : O'Reilly Media, 2013. 1648 p.
- 43.Budd T. Understanding Object-Oriented Programming. 3rd ed. Boston : Addison-Wesley, 2002. 384 p.

ДОДАТКИ

					ІІЗ.КР.Б – 121 – 25 – ІІЗ	Арк.
						76

Додаток А – Лістинг коду бази даних:

```
import sqlite3
import os

DB_NAME = "assistant.db"
BASE_DIR = os.path.dirname(os.path.abspath(__file__))
DB_PATH = os.path.join(BASE_DIR, DB_NAME)

def get_conn():
    return sqlite3.connect(DB_PATH)

def init_db():
    conn = get_conn()
    cursor = conn.cursor()

    cursor.execute("""
        CREATE TABLE IF NOT EXISTS commands (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL,
            type TEXT NOT NULL,
            action TEXT NOT NULL,
            is_system INTEGER DEFAULT 0,
            description TEXT
        )
    """)

    cursor.execute("""
        CREATE TABLE IF NOT EXISTS command_history (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            timestamp TEXT DEFAULT CURRENT_TIMESTAMP,
            command TEXT,
            action TEXT,
            source TEXT
        )
    """)

    cursor.execute("""
        CREATE TABLE IF NOT EXISTS settings (
            key TEXT PRIMARY KEY,
            value TEXT
        )
    """)

    cursor.execute("""
        CREATE TABLE IF NOT EXISTS command_aliases (
```

```

        id INTEGER PRIMARY KEY AUTOINCREMENT,
        command_id INTEGER NOT NULL,
        alias TEXT NOT NULL,
        FOREIGN KEY (command_id) REFERENCES commands(id) ON DELETE CASCADE
    )
    """)

```

```

cursor.execute("""
CREATE TABLE IF NOT EXISTS device_control (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    type TEXT,
    protocol TEXT,
    endpoint TEXT,
    command_on TEXT,
    command_off TEXT
)
""")

```

```

cursor.execute("""
CREATE TABLE IF NOT EXISTS user_preferences (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    command TEXT NOT NULL,
    time TEXT NOT NULL,
    repeat_pattern TEXT,
    enabled INTEGER DEFAULT 1
)
""")

```

```

conn.commit()
conn.close()

```

```

def add_default_commands(conn):

```

```

    cursor = conn.cursor()
    commands = [
        ("відкрий браузер", "open", "https://google.com", 1, "Відкриває браузер Google"),
        ("покажи погоду", "api", "weather", 0, "Показує поточну погоду"),
        ("вимкни світло", "iot", "light_off", 0, "Вимикає світло в кімнаті"),
        ("котра година", "ctime", "time", 1, "Повідомляє поточний час"),
        ("закрий програму", "system", "close_app", 1, "Закриває активну програму")
    ]

```

```

    for name, type_, action, is_sys, desc in commands:
        cursor.execute("INSERT INTO commands (name, type, action, is_system, description) VALUES (?, ?, ?, ?, ?)",
            (name, type_, action, is_sys, desc))

```

```

def find_command_action_by_name(full_name):
    conn = get_conn()
    cursor = conn.cursor()
    cursor.execute("SELECT action FROM commands WHERE LOWER(name) = LOWER(?)", (full_name.strip(),))
    row = cursor.fetchone()
    conn.close()
    return row[0] if row else None

def log_command(command, action, source="system"):
    conn = get_conn()
    cursor = conn.cursor()
    cursor.execute("INSERT INTO command_history (command, action, source) VALUES (?, ?, ?)",
        (command, action, source))
    conn.commit()
    conn.close()

def find_all_commands():
    conn = get_conn()
    cursor = conn.cursor()
    cursor.execute("SELECT name, type, action, is_system, description FROM commands")
    rows = cursor.fetchall()
    conn.close()

    return [{
        'name': r[0],
        'type': r[1],
        'action': r[2],
        'is_system': r[3],
        'description': r[4]
    } for r in rows]

```

Додаток Б – Лістинг модуля Wake Word:

Ініціалізація бібліотеки Porcupine:

```
import pvporcupine
import pyaudio
import os
import numpy as np
from scipy.signal import resample
from dotenv import load_dotenv

load_dotenv()
ACCESS_KEY = os.getenv('ACCESS_KEY')
KEYWORD_PATH = "app/conf/key_word/Lilith_en_windows_v3_0_0.ppn"

if not os.path.exists(KEYWORD_PATH):
    raise FileNotFoundError(f"Keyword file not found at: {KEYWORD_PATH}")

porcupine = pvporcupine.create(
    access_key=ACCESS_KEY,
    keyword_paths=[KEYWORD_PATH]
)

p = pyaudio.PyAudio()

input_device_index = 1
input_sample_rate = 44100

audio_stream = p.open(
    rate=input_sample_rate,
    channels=1,
    format=pyaudio.paInt16,
    input=True,
    frames_per_buffer=int(porcupine.frame_length * input_sample_rate / porcupine.sample_rate),
    input_device_index=input_device_index
)

def get_audio_data():
    pcm = audio_stream.read(
        int(porcupine.frame_length * input_sample_rate / porcupine.sample_rate),
        exception_on_overflow=False
    )
    pcm = np.frombuffer(pcm, dtype=np.int16)
    return resample(pcm, porcupine.frame_length).astype(np.int16)
```

Лістинг коду функції wake_word_listener:

```
import pvporcupine
import pyaudio
import os
import numpy as np
from scipy.signal import resample
from dotenv import load_dotenv

load_dotenv()
ACCESS_KEY = os.getenv('ACCESS_KEY')
KEYWORD_PATH = "app/conf/key_word/Lilith_en_windows_v3_0_0.ppn"

if not os.path.exists(KEYWORD_PATH):
    raise FileNotFoundError(f"Keyword file not found at: {KEYWORD_PATH}")

porcupine = pvporcupine.create(
    access_key=ACCESS_KEY,
    keyword_paths=[KEYWORD_PATH]
)

p = pyaudio.PyAudio()

input_device_index = 1
input_sample_rate = 44100

audio_stream = p.open(
    rate=input_sample_rate,
    channels=1,
    format=pyaudio.paInt16,
    input=True,
    frames_per_buffer=int(porcupine.frame_length * input_sample_rate / porcupine.sample_rate),
    input_device_index=input_device_index
)

def get_audio_data():
    pcm = audio_stream.read(
        int(porcupine.frame_length * input_sample_rate / porcupine.sample_rate),
        exception_on_overflow=False
    )
    pcm = np.frombuffer(pcm, dtype=np.int16)
    return resample(pcm, porcupine.frame_length).astype(np.int16)
```

Додаток Г – Лістинг модуля IoT :

Лістинг коду init.py:

```
from .iot_manager import IoTManager

manager = IoTManager()

def handle_command(command):
    return manager.handle_command(command)

def get_device_status(device_name):
    return manager.get_device_status(device_name)
```

Лістинг коду devices.py:

```
class IoTDevice:
    def __init__(self, name, device_type, status=False):
        self.name = name
        self.type = device_type
        self.status = status

    def toggle(self):
        self.status = not self.status
        return self.status

    def turn_on(self):
        self.status = True
        return self.status

    def turn_off(self):
        self.status = False
        return self.status

    def is_on(self):
        return self.status

    def get_status(self):
        return "увімкнено" if self.status else "вимкнено"

class Light(IoTDevice):
    def __init__(self, name):
        super().__init__(name, "light")

class Kettle(IoTDevice):
```

```
def __init__(self, name):
    super().__init__(name, "kettle")
```

Лістинг коду `iot_manager.py`:

```
from app.iot.devices import Light, Kettle
from app.iot.mqtt_handler import MQTTHandler
from app.DB.db import log_command
from app.stt_tts_script.tts_ import speak
import time
```

```
class IoTManager:
    _instance = None
```

```
def __new__(cls):
    if cls._instance is None:
        cls._instance = super().__new__(cls)
        cls._instance._initialized = False
    return cls._instance
```

```
def __init__(self):
    if self._initialized:
        return
```

```
    self._initialized = True
    self.devices = {
        "light": Light("Світло в кімнаті"),
        "kettle": Kettle("Чайник")
    }
    self.mqtt = MQTTHandler()
    self._connect_silently()
```

```
def _connect_silently(self):
    try:
        if not self.mqtt.connect():
            time.sleep(5)
            self.mqtt.connect()
    except:
        pass
```

```
def handle_command(self, command):
    device_name = command.get("device")
    action = command.get("action")
```

```
    if device_name not in self.devices:
        speak("Пристрій не знайдено")
```

```

        return False

    device = self.devices[device_name]

    try:
        if action == "on":
            device.turn_on()
            speak(f"{device.name} увімкнено")
        elif action == "off":
            device.turn_off()
            speak(f"{device.name} вимкнено")
        else:
            speak("Невідома дія")
            return False

    try:
        if hasattr(self.mqtt, 'is_connected') and self.mqtt.is_connected():
            self.mqtt.publish(f"home/{device_name}", "on" if device.is_on() else "off")
        except:
            pass

    log_command(f"{action} {device_name}", f"{device_name}:{action}")
    return True

except Exception as e:
    print(f"Помилка обробки команди: {e}")
    return False

def get_device_status(self, device_name):
    if device_name in self.devices:
        return self.devices[device_name].get_status()
    return None

```

Лістинг коду mqtt_handler.py:

```

import paho.mqtt.client as mqtt
import time

class MQTTHandler:
    def __init__(self, host="test.mosquitto.org", port=1883):
        self.host = host
        self.port = port
        self.connected = False
        self.client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION1, "iot_simulator")

        self.client.on_connect = self.on_connect
        self.client.on_disconnect = self.on_disconnect

```

```

self.client.on_publish = self.on_publish

def on_connect(self, client, userdata, flags, rc):
    if rc == 0:
        print(f"[MQTT] Успішно підключено до брокера {self.host}:{self.port}")
        self.connected = True
    else:
        print(f"[MQTT] Помилка підключення: код {rc}")

def on_disconnect(self, client, userdata, rc):
    print(f"[MQTT] Відключено від брокера")
    self.connected = False

def is_connected(self):
    # Імітація підключення, якщо вказано хост і порт
    if self.host and self.port:
        return True
    return self.client and self.client.is_connected()

def on_publish(self, client, userdata, mid):
    print(f"[MQTT] Повідомлення опубліковано (ID: {mid})")

def connect(self):
    try:
        print(f"[MQTT] Спроба підключення до {self.host}:{self.port}...")
        if self.host and self.port:
            self.connected = True
            return True

        self.client.connect(self.host, self.port, 60)
        self.client.loop_start()
        time.sleep(0.1)
        return self.connected
    except Exception as e:
        print(f"[MQTT] Помилка підключення: {str(e)}")
        return False

def publish(self, topic, message):
    if not self.is_connected():
        if not self.connect():
            print("[MQTT] Немає підключення, не вдалося відправити повідомлення")
            return False

    try:
        print(f"[MQTT] Імітація публікації: {topic} - {message}")
        if self.host and self.port:
            return True

        result = self.client.publish(topic, message, qos=1)

```

```

    if result.rc == mqtt.MQTT_ERR_SUCCESS:
        print(f"[MQTT] Відправлено до {topic}: {message}")
        return True
    else:
        print(f"[MQTT] Помилка відправки: код {result.rc}")
        return False
except Exception as e:
    print(f"[MQTT] Помилка публікації: {str(e)}")
    return False

def disconnect(self):
    if hasattr(self.client, 'loop_stop'):
        self.client.loop_stop()
    if hasattr(self.client, 'disconnect'):
        self.client.disconnect()
    self.connected = False

```

Додаток В – Лістинг модуля STT та TTS :

Лістинг коду функції TTS:

```
import edge_tts
import tempfile
import os
import asyncio
import pygame

def speak(TEXT):
    VOICE = 'uk-UA-PolinaNeural'
    temp_path = os.path.join(tempfile.gettempdir(), "temp_audio.mp3")
    asyncio.run(edge_tts.Communicate(text=TEXT, voice=VOICE).save(temp_path))

    pygame.mixer.init()
    pygame.mixer.music.load(temp_path)
    pygame.mixer.music.play()

    while pygame.mixer.music.get_busy():
        pygame.time.Clock().tick(10)

    pygame.mixer.quit()
    os.remove(temp_path)
```

Лістинг коду функції STT:

```
import vosk
import sys
import sounddevice as sd
import queue
import json

model = vosk.Model("app/conf/uk_model")

sample_rate = 16000
device = 1

q = queue.Queue()

def q_callback(indata, frames, time, status):
    if status:
        print(status, file=sys.stderr)
    q.put(bytes(indata))

def listener(callback):
    with sd.RawInputStream(samplerate=sample_rate, blocksize=8000, device=device, dtype='int16',
```

```

        channels=1, callback=q_callback):
rec = vosk.KaldiRecognizer(model, sample_rate)
while True:
    data = q.get()
    if rec.AcceptWaveform(data):
        text = json.loads(rec.Result())["text"]
        if text.strip():
            callback(text)
            break

```

Додаток Г – Лістинг модуля API :

Лістинг коду __init__.py:

```
from .weather_api import WeatherAPI

weather = WeatherAPI()

def get_weather(city_name=None):
    return weather.get_weather(city_name)

def speak_weather(city_name=None):
    return weather.speak_weather(city_name)
```

Лістинг коду weather_api.py:

```
import requests
from app.stt_tts_script.tts_ import speak
from app.DB.db import log_command
import json
from datetime import datetime
from dotenv import load_dotenv
import os

load_dotenv()
WEATHER_API_KEY = os.getenv('WEATHER_API_KEY')
class WeatherAPI:
    def __init__(self, api_key= WEATHER_API_KEY):
        self.api_key = api_key
        self.weather_url = "http://api.openweathermap.org/data/2.5/weather"
        self.geo_url = "http://ip-api.com/json/"
        self.units = "metric"
        self.lang = "ua"

    def _get_location(self):
        try:
            response = requests.get(self.geo_url)
            response.raise_for_status()
            data = response.json()

            if data['status'] == 'success':
                return data['city']
            return None
        except Exception as e:
            print(f"[WeatherAPI] Помилка отримання локації: {e}")
            return None
```

```

def get_weather(self, city_name=None):
    if not city_name:
        city_name = self._get_location()
    try:
        params = {
            'q': city_name,
            'appid': self.api_key,
            'units': self.units,
            'lang': self.lang
        }

        response = requests.get(self.weather_url, params=params)
        response.raise_for_status()

        data = response.json()
        return self._parse_weather_data(data)

    except requests.exceptions.RequestException as e:
        print(f"[WeatherAPI] Помилка запиту: {e}")
        speak("Не вдалося отримати дані про погоду. Перевірте підключення до інтернету.")
        return None
    except Exception as e:
        print(f"[WeatherAPI] Помилка: {e}")
        speak("Сталася помилка при отриманні погоди.")
        return None

def _parse_weather_data(self, data):
    try:
        weather_data = {
            'city': data.get('name', 'Невідоме місто'),
            'temp': round(data['main']['temp']),
            'feels_like': round(data['main']['feels_like']),
            'description': data['weather'][0]['description'].capitalize(),
            'humidity': data['main']['humidity'],
            'wind_speed': data['wind']['speed'],
            'icon': data['weather'][0]['icon']
        }
        return weather_data
    except KeyError as e:
        print(f"[WeatherAPI] Помилка парсингу: {e}")
        return None

def speak_weather(self, city_name=None):
    weather = self.get_weather(city_name)
    if not weather:
        return False

    message = (
        f"Температура {weather['temp']} градусів, "

```

```

        f"Відчувається як {weather['feels_like']}. "
        f"Вологість {weather['humidity']}%. "
    )

    log_command(f"запит погоди для {weather['city']}", f"weather:{weather['city']}")
    speak(message)
    return True

```

Додаток Д – Лістинг модуля Command :

Лістинг коду cmd_handler.py:

```
import datetime
import json
import subprocess
import webbrowser
from fuzzywuzzy import fuzz
from playsound import playsound
from num2words import num2words
from app.api.weather_api import WeatherAPI
from app.iot.iot_manager import IoTManager
from app.stt_tts_script.tts_ import speak
from app.command.config_loader import load_config
from app.DB.db import log_command

class CommandHandler:
    def __init__(self):
        self._config = None
        self.iot_manager = IoTManager()
        self.api_weather = WeatherAPI()
        self.reload_config()
    @property
    def config(self):
        return self._load_fresh_config()

    def _load_fresh_config(self):
        try:
            with open("app/command/config.json", "r", encoding="utf-8") as f:
                return json.load(f)
        except Exception as e:
            print(f"Помилка завантаження конфігурації: {e}")
            return {}

    def reload_config(self):
        self._config = self._load_fresh_config()

    def find_command(self, voice_input):
        best_match = None
        best_score = 0

        for cmd_type, cmd_list in self.config.items():
            for cmd in cmd_list:
                score = fuzz.ratio(voice_input.lower(), cmd["name"].lower())
                if score > best_score:
                    best_score = score
```

```

        best_match = {
            "type": cmd_type,
            "name": cmd["name"],
            "action": cmd["action"]
        }
    return best_match if best_score > 70 else None

def execute_command(self, command):
    cmd_type = command["type"]
    action = command["action"]
    name = command["name"]

    log_command(name, action)

    if cmd_type == "open_web":
        print(f"Opening web: {action}")
        if not action.startswith(('http://', 'https://')):
            action = 'https://' + action
        webbrowser.open(action)
        playsound("app/sound/doing.mp3")

    elif cmd_type == "open_app":
        print(f"Opening app: {action}")
        try:
            subprocess.Popen(action, shell=True)
            playsound("app/sound/doing.mp3")
        except Exception as e:
            speak(f"Не вдалося відкрити додаток: {str(e)}")

    elif cmd_type == "system":
        print(f"Executing system command: {action}")
        try:
            subprocess.run(
                action,
                shell=True,
                check=True
            )

        except subprocess.CalledProcessError as e:
            print(f"System command failed with return code {e.returncode}")
            playsound("app/sound/doing.mp3")

        except Exception as e:
            print(f"Error executing system command: {e}")
            playsound("app/sound/not_fount.mp3")

    elif cmd_type == "close":
        print(f"[CLOSE] Обробка команди закриття: {action}")
        try:

```

```

        process_name = action if action.endswith('.exe') else f'{action}.exe'
        result = subprocess.run(
            f"taskkill /f /im {process_name}",
            shell=True,
            capture_output=True,
            text=True
        )
        if "not found" in result.stderr or "не найдено" in result.stderr:
            speak("Додаток не знайдено")
        else:
            speak("Додаток закрито")
    except Exception as e:
        print(f"Помилка закриття: {e}")

elif cmd_type == "ctime":
    now = datetime.datetime.now()
    time_str = f"Зараз {num2words(now.hour, lang='uk')} {num2words(now.minute, lang='uk')}"
    speak(time_str)

elif cmd_type == "iot":
    parts = action.split(":")
    if len(parts) == 2:
        device, action = parts
        self.iot_manager.handle_command({
            "device": device,
            "action": action
        })

elif cmd_type == "api":
    if name == "погода":
        self.api_weather.speak_weather()

def va_respond(self, voice_input):
    print("Розпізнано:", voice_input)
    match = self.find_command(voice_input)
    if match:
        self.execute_command(match)
    else:
        playsound("app/sound/not_fount.mp3")

```

Лістинг коду config_loader.py:

```

import json
import os

```

```

CONFIG_PATH = os.path.join(os.path.dirname(__file__), "config.json")

```

```

def load_config():
    with open(CONFIG_PATH, "r", encoding="utf-8") as f:
        return json.load(f)

def save_config(config_data):
    with open(CONFIG_PATH, "w", encoding="utf-8") as f:
        json.dump(config_data, f, indent=2, ensure_ascii=False)

def update_commands_from_db(commands):
    config = load_config()

    for cmd in commands:
        cmd_type = cmd["type"]
        if cmd_type not in config:
            config[cmd_type] = []

        # Перевіряємо, чи команда вже існує
        exists = any(c["name"] == cmd["name"] for c in config[cmd_type])
        if not exists:
            config[cmd_type].append({
                "name": cmd["name"],
                "action": cmd["action"]
            })

    save_config(config)

```

Лістинг коду load_config_from_db.py:

```

import json
import os
from app.DB.db import get_conn
from app.command.config_loader import save_config

CONFIG_PATH = os.path.join(os.path.dirname(__file__), "config.json")

def load_commands_from_db():
    conn = get_conn()
    cursor = conn.cursor()
    cursor.execute("SELECT name, type, action FROM commands")
    rows = cursor.fetchall()
    conn.close()

    config_dict = {}

```

```
for name, cmd_type, action in rows:
    if cmd_type not in config_dict:
        config_dict[cmd_type] = []
    config_dict[cmd_type].append({
        "name": name,
        "action": action
    })

return config_dict

def generate_config():
    config_data = load_commands_from_db()
    save_config(config_data)
```

Додаток Е – Лістинг коду main.py :

```
import sqlite3
import sys
from PySide6.QtWidgets import QApplication, QMessageBox
from interface.app_gui import VoiceAssistantUI
from app.wake_word.wake_word_listener import listen_for_wake_word
from playsound import playsound
from app.DB.db import init_db, get_conn
from app.command.load_config_from_db import generate_config
import threading
from app.command.cmd_handler import CommandHandler

class VoiceAssistantApp:
    def __init__(self):
        init_db()
        generate_config()

        self.command_handler = CommandHandler()

        self.app = QApplication(sys.argv)
        self.app.setStyle("Fusion")

        self.window = VoiceAssistantUI()
        self.window.command_handler = self.command_handler

        self.load_saved_theme()

        self.window.dashboard_page.main_activate_btn.clicked.connect(self.toggle_activation)
        self.window.destroyed.connect(self.on_window_close)

        # Thread for wake word listener
        self.listener_thread = None
        self.listener_active = False

    def load_saved_theme(self):
        conn = get_conn()
        cursor = conn.cursor()
        cursor.execute("SELECT value FROM settings WHERE key = 'theme'")
        row = cursor.fetchone()
        conn.close()

        if row:
            theme = row[0]
            themes = ["dark", "light", "purple", "red", "green"]
            theme_names = ["Темна", "Світла", "Фіолетова", "Червона", "Зелена"]
            if theme in themes:
                index = themes.index(theme)
```

```

        self.window.settings_page.theme_selector.setCurrentIndex(index)
        self.window.apply_style(theme)

def load_saved_settings(self):
    conn = get_conn()
    cursor = conn.cursor()

    cursor.execute("SELECT value FROM settings WHERE key = 'theme'")
    row = cursor.fetchone()
    if row:
        theme = row[0]
        themes = ["dark", "light", "purple", "red", "green"]
        theme_names = ["Темна", "Світла", "Фіолетова", "Червона", "Зелена"]
        if theme in themes:
            index = themes.index(theme)
            self.window.settings_page.theme_selector.setCurrentIndex(index)
            self.window.apply_style(theme)

    cursor.execute("SELECT value FROM settings WHERE key = 'voice_lang'")
    row = cursor.fetchone()
    if row:
        lang_index = 0 if row[0] == "uk" else 1
        self.window.voice_lang.setCurrentIndex(lang_index)

    cursor.execute("SELECT value FROM settings WHERE key = 'voice_speed'")
    row = cursor.fetchone()
    if row:
        self.window.voice_speed.setValue(int(row[0]))

    cursor.execute("SELECT value FROM settings WHERE key = 'iot_enabled'")
    row = cursor.fetchone()
    if row:
        self.window.iot_enabled.setChecked(row[0] == "1")

    cursor.execute("SELECT value FROM settings WHERE key = 'mqtt_host'")
    row = cursor.fetchone()
    if row:
        self.window.mqtt_host.setText(row[0])

    cursor.execute("SELECT value FROM settings WHERE key = 'mqtt_port'")
    row = cursor.fetchone()
    if row:
        self.window.mqtt_port.setText(row[0])

    conn.close()

def save_theme(self, theme):
    conn = get_conn()
    cursor = conn.cursor()

```

```

        cursor.execute("REPLACE INTO settings (key, value) VALUES (?, ?)", ("theme", theme))
        conn.commit()
        conn.close()

def toggle_automation(self):
    if self.window.dashboard_page.main_activate_btn.isChecked():
        self.listener_active = True
        self.listener_thread = threading.Thread(target=self.run_listener, daemon=True)
        self.listener_thread.start()
        playsound("app/sound/activate.mp3")
    else:
        self.listener_active = False

def run_listener(self):
    while self.listener_active:
        listen_for_wake_word()

def on_window_close(self):
    themes = ["dark", "light", "purple", "red", "green"]
    current_index = self.window.settings_page.theme_selector.currentIndex()
    if 0 <= current_index < len(themes):
        self.save_theme(themes[current_index])

    self.listener_active = False

def save_new_command(self, command_data):
    conn = get_conn()
    cursor = conn.cursor()

    try:
        cursor.execute(
            "INSERT INTO commands (name, type, action, description) VALUES (?, ?, ?, ?)",
            (command_data["name"], command_data["type"], command_data["action"],
            command_data["description"])
        )
        conn.commit()

        generate_config()
        self.command_handler.reload_config() # Оновлюємо конфігурацію в пам'яті
        self.load_commands_from_db()

    except sqlite3.Error as e:
        QMessageBox.warning(self, "Помилка", f"Не вдалося зберегти команду: {str(e)}")
    finally:
        conn.close()

def run(self):
    self.window.show()
    sys.exit(self.app.exec())

```

```
if __name__ == "__main__":  
    assistant = VoiceAssistantApp()  
    assistant.run()
```