Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

РОЗРОБЛЕННЯ КРОСПЛАТФОРМЕННОГО ГОЛОСОВОГО АСИСТЕНТА ДЛЯ КЕРУВАННЯ КОМП'ЮТЕРОМ З ВИКОРИСТАННЯМ МЕТОДІВ ОБРОБЛЕННЯ ПРИРОДНЬОЇ МОВИ
(тема)

Виконав:
здобувач 4 року навчання,
групи ІТІНФ-22-1
Ковальчук В. О.
(прізвище, ініціали)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник ст. викл. Кобилін І. О.
(посада, прізвище, ініціали)

Допускається до захисту

Завідувач кафедри інформатики _____ Кобилін О. А.
(підпис) (прізвище, ініціали)

2026 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту

Кафедра Інформатики

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Ковальчуку Владиславу Олексійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розроблення кросплатформенного голосового асистента для керування комп'ютером з використанням методів оброблення природньої мови

затверджена наказом університету від 26 травня 2026 року № 435Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії 23 червня 2026 р.
3. Вихідні дані до роботи науково-методична та науково-технічна література, матеріали конференцій, дані інтернет-мережі, офіційна документація .NET, .NET MAUI, Whisper.net та ONNX Runtime, набір прикладів голосових команд українською мовою, локальні моделі розпізнавання мовлення та векторного представлення тексту.

4. Перелік питань, що потрібно опрацювати в роботі
1. Аналіз підходів до обробки голосових команд та обґрунтування архітектурних рішень кросплатформеного голосового асистента для керування комп'ютером.
2. Моделі, методи та інформаційні технології обробки голосових команд у кросплатформеному голосовому асистенті.
3. Програмна реалізація та тестування кросплатформеного голосового асистента.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) актуальність та завдання, загальний алгоритм роботи системи, гібридне визначення наміру користувача, токенізація та підготовка вхідних даних, формування embedding-вектора, попереднє тестування системи, результати роботи та перспективи розвитку, апробація результатів.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	13.04.2026	
2	Аналіз завдання, підбір літератури	14.04.26-16.04.26	
3	Аналіз літератури з проблеми, що вирішується	17.04.26-19.04.26	
4	Аналіз існуючих методів розпізнавання мови	20.04.26-22.04.26	
5	Формування кроків вирішення проблеми	23.04.26-05.05.26	
6	Програмна реалізація	26.04.26-20.05.26	
7	Аналіз отриманих результатів	21.05.26-04.06.26	
8	Оформлення пояснювальної записки	05.06.26-09.06.26	
9	Перевірка на нормоконтроль	10.06.26-19.06.26	
10	Перевірка на плагіат	11.06.26-30.06.26	
11	Рецензування	20.06.26-30.06.26	
12	Підготовка презентації та доповіді	20.06.26-30.06.26	
13	Занесення роботи в електронний архів	30.06.26-09.07.26	
14	Попередній захист кваліфікаційної роботи	30.06.26-09.07.26	

Дата видачі завдання 13 квітня 2026 р.

Здобувач _____
(підпис)

Керівник роботи _____ _____ ст. викл. Кобилін І. О._____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 91 с., 10 табл., 4 рис., 30 джерел.

ГОЛОСОВИЙ АСИСТЕНТ, ОБРОБКА ПРИРОДНОЇ МОВИ, ІНТЕНЦІЯ, ЛОКАЛЬНЕ РОЗПІЗНАВАННЯ МОВЛЕННЯ, EMBEDDINGS, WHISPER.NET, .NET.

Об’єктом роботи є процес обробки голосових команд користувача в системі керування комп’ютером.

Метою роботи є розробка кросплатформеного голосового асистента для керування комп’ютером з використанням методів обробки природної мови.

У роботі проведено аналіз підходів до розпізнавання мовлення, визначення інтенції та виконання системних дій. Для реалізації застосунку використано локальне розпізнавання мовлення, векторне представлення тексту, rule-based обробку параметрів команд, .NET 8 та .NET MAUI.

У результаті роботи здійснено програмну реалізацію голосового асистента, що забезпечує активацію голосового введення, розпізнавання команди, визначення інтенції, виділення параметрів та виконання відповідної системної дії в середовищі Windows.

VOICE ASSISTANT, NATURAL LANGUAGE PROCESSING, INTENT DETECTION, LOCAL SPEECH RECOGNITION, EMBEDDINGS, WHISPER.NET, .NET.

The object of this work is the process of processing user voice commands in a computer control system.

The goal of this work is to develop a cross-platform voice assistant for computer control using natural language processing methods.

The work analyzes approaches to speech recognition, intent detection, and execution of system actions. The application uses local speech recognition, text vector representation, rule-based command parameter processing, .NET 8, and .NET MAUI.

As a result of the work, a voice assistant software application has been implemented. It provides voice input activation, command recognition, intent detection, parameter extraction, and execution of the corresponding system action in the Windows environment.

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів 7
Вступ 9
1 Аналіз підходів до обробки голосових команд та обґрунтування архітектурних рішень кросплатформеного голосового асистента для керування комп'ютером 10
1.1 Актуальність розробки кросплатформених голосових інтерфейсів для керування комп'ютером 10
1.2 Особливості обробки коротких голосових команд українською мовою 12
1.3 Аналіз підходів до визначення інтенції в системах голосового керування 13
1.3.1 Rule-based підхід до інтерпретації голосових команд 14
1.3.2 Classifier-based підхід до визначення інтенції команд 15
1.3.3 Embedding-based підхід до семантичної інтерпретації голосових команд 17
1.4 Обґрунтування вибору рішень для інтерпретації голосових команд у межах проєкту 19
1.5 Постановка задачі 22
2 Моделі, методи та інформаційні технології обробки голосових команд у кросплатформеному голосовому асистенті 25
2.1 Інформаційна технологія обробки голосової команди 25
2.2 Архітектура програмної системи голосового асистента 28
2.3 Модель голосового введення та локального розпізнавання мовлення 31
2.4 Математична модель векторного представлення голосової команди 34
2.5 Метод визначення інтенції на основі embedding-подібності 36
2.6 Гібридна модель прийняття рішення 39

2.7 Об’єктно-орієнтована модель системи та обґрунтування вибору технологій реалізації ... 43
3 Програмна реалізація та тестування кросплатформеного голосового асистента ... 47
3.1 Обґрунтування інструментальних засобів реалізації ... 47
3.2 Структура програмного проєкту ... 50
3.3 Реалізація основних функціональних модулів ... 53
3.3.1 Реалізація ядра обробки команд ... 53
3.3.2 Реалізація підсистеми голосового введення та локального розпізнавання мовлення ... 57
3.3.3 Реалізація embedding-модуля ... 59
3.3.4 Реалізація модуля визначення інтенції ... 62
3.3.5 Реалізація гібридної обробки команд і параметрів ... 65
3.3.6 Реалізація виконавчого шару системних дій ... 68
3.4 Реалізація користувацького інтерфейсу та діагностичного режиму ... 71
3.5 Інструкція користувача ... 75
3.6 Тестування функціональних можливостей голосового асистента 78
3.7 Аналіз результатів тестування та перспективи вдосконалення ... 83
Висновки ... 86
Перелік джерел посилання ... 88

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ШІ – штучний інтелект

API – Application Programming Interface (інтерфейс програмування застосунків)

ASR – Automatic Speech Recognition (автоматичне розпізнавання мовлення)

Cosine similarity – косинусна подібність, міра близькості між векторними представленнями

DTO – Data Transfer Object (об’єкт передавання даних між частинами програмної системи)

Embedding – векторне представлення тексту або команди

Intent – інтенція, тобто намір користувача, закладений у команді

ML – Machine Learning (машинне навчання)

ML.NET – фреймворк машинного навчання для екосистеми .NET

NAudio – бібліотека .NET для роботи з аудіопотоком

.NET MAUI – Multi-platform App UI, кросплатформений фреймворк для створення застосунків на платформі .NET

NLP – Natural Language Processing (обробка природної мови)

NLU – Natural Language Understanding (розуміння природної мови)

ONNX – Open Neural Network Exchange (відкритий формат обміну нейронними мережами)

ONNX Runtime – середовище виконання моделей у форматі ONNX

Porcupine – інструмент для локального виявлення wake word у голосових застосунках

Rule-based підхід – підхід, заснований на правилах, шаблонах і словниках

Sentence embeddings – векторні представлення речень або коротких текстових висловлювань

STT – Speech-to-Text (перетворення мовлення в текст)

UI – User Interface (інтерфейс користувача)

VUI – Voice User Interface (голосовий інтерфейс користувача)

Wake word – активаційна фраза, після якої система переходить до повного розпізнавання мовлення

Whisper.net – бібліотека для локального розпізнавання мовлення у .NET-застосунках

ВСТУП

У сучасних умовах цифровізації зростає потреба у природних і зручних способах взаємодії користувача з комп’ютерними системами. Традиційні засоби введення, зокрема клавіатура і миша, залишаються основними, проте не завжди забезпечують достатню швидкість і гнучкість керування. Одним із перспективних напрямів є голосове керування, за якого команди передаються у природній мовній формі.

Розвиток методів обробки природної мови, машинного навчання та автоматичного розпізнавання мовлення розширив можливості створення голосових асистентів. Водночас значна частина таких рішень орієнтована на хмарну інфраструктуру, що спричиняє залежність від мережевого з’єднання, додаткові затримки та передавання даних до зовнішніх сервісів.

Актуальність роботи полягає у створенні голосового асистента, здатного працювати локально, швидко реагувати на команди користувача та виконувати типові дії в операційному середовищі комп’ютера. Додаткового значення набуває підтримка української мови, оскільки її морфологічна варіативність ускладнює визначення наміру користувача під час обробки коротких голосових команд.

Особливістю задачі є поєднання кількох компонентів у межах єдиної системи: активації голосового введення, перетворення мовлення в текст, визначення інтенції, виділення параметрів і виконання системної дії.

Отже, робота охоплює не лише розпізнавання мовлення, а й інтерпретацію природної мови та прикладну автоматизацію користувацьких процесів.

1 АНАЛІЗ ПІДХОДІВ ДО ОБРОБКИ ГОЛОСОВИХ КОМАНД ТА ОБҐРУНТУВАННЯ АРХІТЕКТУРНИХ РІШЕНЬ КРОСПЛАТФОРМЕНОГО ГОЛОСОВОГО АСИСТЕНТА ДЛЯ КЕРУВАННЯ КОМП'ЮТЕРОМ

1.1 Актуальність розробки кросплатформених голосових інтерфейсів для керування комп'ютером

У сучасних програмних системах зростає потреба в природних і швидких способах взаємодії користувача з комп'ютером. Одним із таких способів є голосовий інтерфейс, який дає змогу ініціювати виконання команд без використання традиційних засобів введення. Для задач керування комп'ютером це має практичне значення, оскільки значна частина типових дій, зокрема запуск програм, відкриття системних розділів, створення нотаток або зміна системних параметрів, може бути подана у формі коротких голосових команд. У дослідженнях з проєктування voice user interfaces підкреслюється, що за останнє десятиліття такі інтерфейси суттєво поширилися, а їх розвиток став окремим напрямом у межах людино-машинної взаємодії [1, 2].

Актуальність голосового керування визначається не лише зручністю такого способу взаємодії, а й можливістю скорочення кількості проміжних операцій між наміром користувача та виконанням системної дії. Для прикладних сценаріїв керування комп'ютером це особливо важливо, оскільки короткі команди дають змогу швидко ініціювати типові дії без переходу через багаторівневі елементи графічного інтерфейсу. Водночас практична цінність голосового асистента залежить не тільки від якості перетворення мовлення в текст, а й від здатності системи коректно встановлювати інтенцію команди та виокремлювати параметри, потрібні для виконання дії.

Окремого значення набуває локальна обробка мовлення. У наукових роботах, присвячених on-device automatic speech recognition, зазначається, що

сучасні end-to-end нейронні підходи зробили локальне розпізнавання мовлення практично досяжним завдяки меншому обсягу пам’яті порівняно з традиційними багатокомпонентними системами. Це створює передумови для побудови голосових застосунків, які можуть працювати без постійної залежності від зовнішньої серверної інфраструктури [3].

Для системи керування комп’ютером такий підхід є доцільним з кількох причин. По-перше, локальна обробка зменшує затримки між вимовленням команди та реакцією системи. По-друге, вона знижує залежність від стабільності мережевого з’єднання. По-третє, вона дозволяє обмежити обробку мовних даних межами самого пристрою, що є важливим для прикладних систем персонального використання. У поєднанні з механізмом активації після спеціальної тригерної фрази це формує практичну модель голосової взаємодії, за якої постійне прослуховування та повне розпізнавання мовлення розділяються функціонально.

Доцільність локальної обробки підтверджується також результатами власного дослідження, у якому розглядалося порівняння локальної та серверної обробки природної мови у кросплатформенному голосовому асистенті. У цій роботі показано, що локальний підхід зменшує залежність від мережевих умов, забезпечує кращий рівень конфіденційності та усуває додаткові затримки, пов’язані з передаванням даних між клієнтом і сервером [4].

Актуальність теми посилюється також кросплатформеним характером сучасної розробки програмного забезпечення. Для голосового асистента модулі активації, перетворення мовлення в текст, визначення інтенції та виділення параметрів команди доцільно проєктувати як максимально платформонезалежні компоненти. Водночас виконання системних дій зберігає платформно-специфічний характер, оскільки безпосередньо залежить від можливостей конкретної операційної системи. Саме тому розробка кросплатформеного голосового асистента є актуальною як з позиції

сучасних підходів до побудови програмних систем, так і з позиції практичної автоматизації типових дій користувача.

1.2 Особливості обробки коротких голосових команд українською мовою

Короткі голосові команди становлять окремий тип мовних вхідних даних, для якого характерні стисліcть, мінімальний контекст і висока змістова концентрація. На відміну від розгорнутих текстових запитів, такі висловлювання зазвичай містять лише кілька слів, однак саме в них має бути закладена вся інформація, необхідна для виконання дії. Через це задача їх інтерпретації є складнішою, ніж може здаватися на перший погляд.

Складність обробки коротких команд зумовлена тим, що обмежений обсяг тексту зменшує кількість контекстних ознак, за якими система може встановити зміст висловлювання. У дослідженнях коротких текстів підкреслюється, що саме брак контексту та розрідженість семантичної інформації ускладнюють їхню класифікацію та зіставлення за змістом [5]. Для голосового асистента це означає, що буквальний збіг окремих слів не може вважатися достатньою основою для надійного визначення інтенції.

Для прикладної системи керування комп’ютером це має практичний наслідок. Один і той самий намір користувача може бути виражений різними короткими формулюваннями, які відрізняються лексично, але залишаються близькими за змістом. Відповідно, система повинна враховувати не лише поверхневу текстову форму команди, а й її семантичну близькість до інших можливих варіантів формулювання.

Для української мови ця задача додатково ускладнюється її морфологічною та синтаксичною специфікою. Українська мова має складну морфологію, категорії роду, числа й відмінка, а також більш вільний порядок слів, що впливає на варіативність мовного вираження [6]. У контексті

голосових команд це означає, що одна й та сама дія може бути сформульована через різні граматичні форми, різне розташування компонентів команди та різні словесні конструкції без зміни базового наміру користувача.

Додаткову складність створює те, що коротка голосова команда часто поєднує два рівні інформації: інтенцію та параметри її виконання. У межах одного висловлювання можуть одночасно міститися вказівка на дію, назва об’єкта, числове значення або напрям зміни. Тому система має не лише віднести команду до певного класу, а й виділити змінні компоненти, які безпосередньо впливають на виконання дії.

Також складність обробки українських голосових команд підтверджується й у власній роботі, присвяченій семантичному аналізу мовленнєвих команд для голосових асистентів. У ній зазначено, що українська мова характеризується розвиненою морфологією, варіативністю словоформ і підвищеними вимогами до точного розпізнавання слів та сутностей, що безпосередньо впливає на якість інтерпретації команд [7].

Отже, особливості обробки коротких голосових команд українською мовою визначаються поєднанням семантичної стислості коротких текстів і мовної варіативності української мови. Саме тому для таких даних більш доцільними є підходи, здатні враховувати змістову близькість різних формулювань, а не лише формальний збіг окремих слів.

1.3 Аналіз підходів до визначення інтенції в системах голосового керування

Визначення інтенції є центральною задачею інтерпретації голосових команд, оскільки саме на цьому етапі система встановлює, яку дію має виконати у відповідь на мовний запит користувача. У дослідженнях з natural language understanding intent detection розглядається як одна з базових задач

поряд із виділенням змістових компонентів висловлювання, що безпосередньо впливають на подальшу реакцію системи [8].

Для систем голосового керування комп’ютером ця задача має прикладний характер. Після перетворення мовлення в текст система повинна не лише обробити отриманий рядок, а й віднести його до одного з допустимих типів команд. Від якості цього етапу залежить коректність подальшого виконання дії, а також стійкість роботи асистента в умовах варіативних формулювань коротких команд.

У прикладних системах можна виокремити три основні підходи до визначення інтенції: rule-based, classifier-based та embedding-based. Вони відрізняються способом представлення тексту, механізмом прийняття рішення та рівнем чутливості до мовної варіативності. Саме тому доцільно розглянути їх окремо та оцінити придатність кожного з них для задачі інтерпретації коротких голосових команд українською мовою.

1.3.1 Rule-based підхід до інтерпретації голосових команд

Rule-based підхід ґрунтується на використанні наперед заданих правил, шаблонів, словників і умовних відповідностей між текстовою формою команди та дією, яку повинна виконати система. У найпростішому випадку це може бути перевірка наявності певних ключових слів, фраз або конструкцій, які однозначно співвідносяться з конкретною інтенцією. Такий підхід є одним із найпростіших з погляду реалізації та контролю логіки роботи системи.

Основною перевагою rule-based підходу є його передбачуваність. Розробник чітко визначає, які саме мовні конструкції належать до певного класу команд, тому поведінка системи є прозорою і легко пояснюваною. Крім того, такий підхід добре працює в задачах, де набір можливих формулювань є обмеженим, а команди мають чітко структурований характер.

Для окремих типів запитів, зокрема тих, що містять фіксовані слова-тригери або прості параметри, rule-based логіка може бути ефективною та обчислювально недорогою

Водночас для задачі голосового керування комп’ютером rule-based підхід має істотні обмеження. По-перше, він чутливий до мовної варіативності. Якщо користувач формулює той самий намір іншими словами, система може не розпізнати команду без додатково передбаченого правила. По-друге, зі збільшенням кількості інтенцій, можливих формулювань і параметрів кількість правил швидко зростає, що ускладнює підтримку та масштабування системи. По-третє, такий підхід погано враховує семантичну близькість між різними фразами, оскільки орієнтується переважно на поверхневий текстовий збіг.

Для українських коротких голосових команд ці недоліки стають ще помітнішими через наявність морфологічної варіативності та відносно вільного порядку слів. Унаслідок цього система, побудована лише на правилах, потребує або великої кількості шаблонів, або дуже жорсткого обмеження допустимих формулювань. Це знижує природність взаємодії та робить асистента менш стійким до реальної мовної практики користувача.

Отже, rule-based підхід доцільно розглядати як корисний інструмент для обробки окремих фіксованих шаблонів, виявлення параметрів команди або уточнення результату, але не як універсальне рішення для повноцінного визначення інтенції у варіативних голосових командах.

1.3.2 Classifier-based підхід до визначення інтенції команд

Classifier-based підхід ґрунтується на розгляді визначення інтенції як задачі класифікації тексту, у межах якої кожна команда відноситься до одного з наперед заданих класів. У системах natural language understanding intent classification належить до базових задач, а її реалізація може спиратися

як на традиційні алгоритми машинного навчання, так і на нейронні моделі, що працюють із векторними ознаками тексту [8, 9]. Прикладом класичного алгоритму такого типу є Support Vector Machine, який може застосовуватися для багатокласової класифікації та використовує ядрові функції для роботи з нелінійно роздільними даними [10, 11].

Перевагою такого підходу є можливість навчання моделі на множині прикладів команд, що належать до різних інтенцій. На відміну від rule-based логіки, classifier-based модель не потребує жорсткого опису кожного допустимого формулювання у вигляді окремого правила. За наявності достатньої кількості репрезентативних прикладів вона може виявляти статистичні закономірності між текстовими ознаками команди та її цільовим класом. Це підвищує гнучкість системи й дозволяє частково подолати проблему мовної варіативності.

Разом із тим ефективність classifier-based підходу істотно залежить від способу подання тексту та якості навчальних даних. Якщо модель спирається переважно на поверхневі лексичні ознаки, вона може бути чутливою до змін формулювання, порядку слів або морфологічної форми. Для коротких голосових команд це є суттєвим обмеженням, оскільки такі висловлювання містять мало контексту, а їх інтенція часто виражається не через набір унікальних слів, а через загальний зміст команди.

Ще одним обмеженням classifier-based підходу є орієнтація на віднесення команди до одного з відомих класів без глибшого семантичного зіставлення близьких за змістом формулювань. Через це модель може працювати нестійко, якщо користувач формулює намір інакше, ніж у прикладах навчальної вибірки. Тому для коротких українських голосових команд цей підхід є корисним, але не завжди достатнім як самостійне рішення.

Отже, classifier-based підхід є більш гнучким порівняно з rule-based схемами та дозволяє автоматизувати визначення інтенції на основі навчальних даних. Проте для коротких варіативних команд його

результативність значною мірою залежить від якості текстового представлення, обсягу прикладів і здатності моделі враховувати не лише формальну, а й змістову подібність команд.

1.3.3 Embedding-based підхід до семантичної інтерпретації голосових команд

Embedding-based підхід ґрунтується на поданні текстової команди у вигляді вектора фіксованої розмірності, який відображає її змістові характеристики. У такій схемі визначення інтенції здійснюється не через жорсткий набір правил і не лише через належність до класу за поверхневими ознаками, а через порівняння семантичної близькості між векторним представленням поточної команди та векторними представленнями еталонних прикладів. Для задач natural language understanding це відповідає переходу від формального зіставлення тексту до роботи зі змістовою подібністю висловлювань [12, 8].

Практична цінність такого підходу полягає в тому, що близькі за змістом команди можуть розпізнаватися як пов’язані навіть за відмінності їхньої поверхневої текстової форми. У роботі Sentence-BERT показано, що моделі sentence embeddings дають змогу формувати семантично осмислені векторні представлення речень, придатні для порівняння за cosine similarity, semantic search і кластеризації. Для багатомовних сценаріїв важливим є також те, що контрастивне навчання може формувати універсальні міжмовні sentence embeddings, придатні для порівняння змісту речень у різних мовах [13, 14]. Саме ця властивість робить embedding-based підхід придатним для інтерпретації коротких голосових команд, де один і той самий намір може бути виражений різними лексичними та синтаксичними конструкціями [12].

$$sim(A,B) = \frac{A \cdot B}{\|A\|\|B\|}, \tag{1.1}$$

де A – векторне представлення першої команди;

B – векторне представлення другої команди;

$A \cdot B$ – скалярний добуток векторів;

$\|A\|$ – норма вектор A;

$\|B\|$ – норма вектора B;

$sim(A,B)$ – значення семантичної подібності між командами.

На відміну від classifier-based підходу, embedding-based схема визначає інтенцію не через жорсткі межі класів, а шляхом зіставлення нової команди з опорними прикладами за семантичною близькістю. Для коротких українських команд це є перевагою, оскільки зменшує чутливість до словоформ, порядку слів і різних способів формулювання одного наміру. Водночас якість такого підходу залежить від властивостей embedding-моделі та репрезентативності набору прикладів [12].

Разом із перевагами embedding-based підхід має і власні обмеження. Він не усуває потреби у прикладній логіці, яка повинна виділяти параметри команди, перевіряти коректність інтерпретації та передавати результат до шару виконання дії. Крім того, сам факт семантичної близькості між командами ще не гарантує повної тотожності їхньої функціональної ролі, тому в практичних системах такий підхід доцільно поєднувати з додатковими механізмами контролю й уточнення результату [12].

Отже, порівняльний аналіз підходів, наведений у табл. 1.1, показує, що embedding-based підхід є найбільш придатним для семантичної інтерпретації коротких голосових команд у тих випадках, коли система повинна враховувати не лише буквальний збіг слів, а й змістову близькість різних формулювань.

Таблиця 1.1 – Порівняльна характеристика підходів до визначення інтенції голосових команд

Підхід	**Принцип роботи**	**Переваги**	**Обмеження**	**Придатність для коротких українських голосових команд**
Rule-based	Правила, шаблони, ключові слова	Простота, прозорість, передбачуваність	Чутливість до формулювань, складне масштабування	Доцільний для фіксованих команд і параметрів
Classifier-based	Класифікація команди за навчальними прикладами	Гнучкіший за правила, навчається на даних	Залежить від вибірки, слабше враховує семантику	Придатний як базовий варіант, але нестійкий до нових формулювань
Embedding-based	Вектор команди та порівняння з прикладами	Ураховує змістову близькість, стійкіший до варіативності	Залежить від embedding-моделі та якості прикладів	Найбільш придатний для інтерпретації коротких українських голосових команд завдяки врахуванню семантичної подібності

1.4 Обґрунтування вибору рішень для інтерпретації голосових команд у межах проєкту

У межах даного проєкту доцільним є використання локального підходу до обробки голосових команд, оскільки система орієнтована на керування функціями комп’ютера в режимі безпосередньої взаємодії з користувачем.

Для такого сценарію принципове значення мають мала затримка, автономність роботи та відсутність залежності від зовнішньої серверної інфраструктури. У працях, присвячених on-device automatic speech recognition, підкреслюється, що сучасні end-to-end моделі розпізнавання мовлення є придатними до локального використання саме завдяки зменшенню обчислювальних і пам'яттєвих витрат порівняно з традиційними багатокомпонентними системами [3].

Вибір локальної схеми є обґрунтованим і з погляду загальної логіки взаємодії. У прикладній системі керування комп'ютером немає потреби виконувати повне розпізнавання мовлення безперервно. Раціональнішим є поділ процесу на два етапи: попередню активацію системи та подальшу інтерпретацію команди після спрацювання тригерного механізму. Такий підхід дозволяє зменшити обчислювальне навантаження на систему та забезпечити більш керований режим голосової взаємодії. У контексті даного проєкту це узгоджується з вимогою локальної роботи асистента та з практичною необхідністю обмежити повне мовне оброблення лише тими фрагментами, які дійсно є командами користувача.

Для визначення інтенції голосових команд обрано embedding-based підхід, оскільки він краще відповідає природі коротких варіативних висловлювань. На відміну від rule-based та classifier-based схем, цей підхід дозволяє порівнювати команди за змістовою близькістю, а не лише за буквальним збігом слів або жорстко заданими межами класів. У роботі Sentence-BERT показано, що sentence embeddings можуть ефективно використовуватись для semantic similarity та semantic search на основі косинусної подібності [12]. Саме ця властивість є визначальною для задачі інтерпретації коротких голосових команд, у яких один і той самий намір може бути виражений різними формулюваннями.

Вибір embedding-based схеми в межах проєкту також обґрунтовується практичними результатами попереднього аналізу. Для коротких команд українською мовою недостатньо покладатися лише на поверхневі текстові

ознаки, оскільки вони не забезпечують належної стійкості до морфологічної варіативності та різних синтаксичних форм подання наміру. Натомість векторне представлення команди дає змогу зіставляти нове висловлювання з набором еталонних прикладів і визначати інтенцію за максимальною семантичною близькістю. Це робить обраний підхід більш придатним саме для прикладної задачі голосового керування, а не лише для формальної текстової класифікації [12].

Разом із тим у межах проєкту недоцільно повністю відмовлятися від rule-based логіки. Хоча вона не є оптимальною як основний механізм визначення інтенції, її використання є виправданим на етапі виділення параметрів команди, перевірки окремих умов виконання та уточнення результату, отриманого ML-шаром. Таким чином, найбільш обґрунтованим для даного проєкту є поєднання семантичного визначення інтенції на основі embeddings із прикладною rule-based обробкою параметрів і службових умов виконання. Така схема забезпечує баланс між гнучкістю інтерпретації та керованістю практичної реалізації.

У власній роботі, присвяченій застосуванню ML.NET у семантичному аналізі мовленнєвих команд для голосових асистентів, також показано, що використання засобів екосистеми .NET є доцільним для побудови локальних рішень без додаткової серверної інфраструктури. У цій роботі підкреслюється інтеграційна зручність ML.NET, підтримка векторизації та можливість підключення трансформерних моделей через ONNX, що узгоджується з архітектурною логікою поточного проєкту [7].

Отже, вибір рішень для інтерпретації голосових команд у межах проєкту визначається трьома основними міркуваннями: доцільністю локальної обробки мовлення, потребою у семантично стійкому визначенні інтенції та необхідністю зберегти контрольовану прикладну логіку для обробки параметрів команд. Саме тому в межах роботи обрано локальний підхід до мовної обробки, embedding-based визначення інтенції та допоміжне використання rule-based механізмів для прикладного уточнення результату.

Узагальнену концептуальну схему інтерпретації голосової команди в межах проєкту показано на рисунку 1.1.

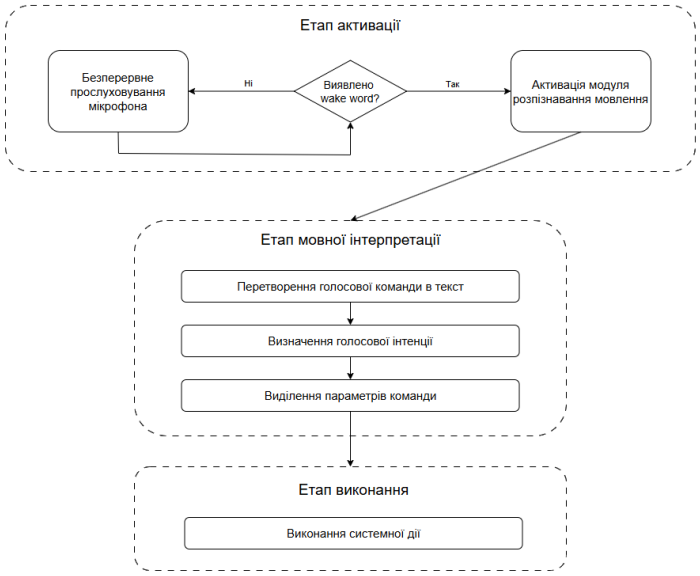

Рисунок 1.1 – Концептуальна схема інтерпретації голосової команди в межах проєкту

1.5 Постановка задачі

Отже, розробка голосового асистента для керування комп’ютером є актуальною задачею прикладної інформатики, що поєднує питання розпізнавання мовлення, обробки природної мови, визначення інтенції коротких команд і виконання системних дій у межах програмного середовища. Проведений аналіз показав, що для такої задачі особливого значення набувають локальність обробки, стійкість до варіативних формулювань команд, швидкість реакції системи та можливість безпосередньої взаємодії з операційним середовищем. Водночас важливо враховувати кросплатформену програмну основу застосунку, оскільки вона забезпечує гнучкість архітектури, тоді як виконання конкретних системних дій потребує адаптації до можливостей певної операційної системи.

У межах даної роботи задача полягає не лише у використанні готового засобу розпізнавання мовлення, а у побудові повного програмного конвеєра обробки голосової команди. Такий конвеєр має забезпечувати послідовний перехід від аудіосигналу користувача до структурованої команди, що містить визначену інтенцію, необхідні параметри та результат виконання. Для цього необхідно поєднати локальний speech-to-text модуль, механізм нормалізації розпізнаного тексту, embedding-based визначення інтенції, rule-based обробку параметрів і виконавчий шар системних дій. Саме така побудова дає змогу розглядати голосовий асистент як цілісну програмну систему, а не як окремий модуль розпізнавання мовлення.Об’єктом роботи є процес обробки голосових команд користувача в системі керування комп’ютером.

Метою роботи є розробка кросплатформеного голосового асистента для керування комп’ютером з використанням методів обробки природної мови.

Для досягнення поставленої мети необхідно вирішити такі завдання:

– проаналізувати особливості коротких голосових команд як об’єкта обробки;

– дослідити підходи до визначення інтенції в системах голосового керування;

– обґрунтувати вибір локального підходу до обробки мовлення та семантичного визначення інтенції на основі векторного представлення команд;

– сформувати набір інтенцій і параметрів, необхідних для прикладного керування функціями комп’ютера;

– розробити кросплатформену архітектуру голосового асистента з локальним розпізнаванням мовлення, модулем визначення інтенції та rule-based шаром виділення параметрів;

– реалізувати програмний застосунок на основі обраного стеку технологій;

– забезпечити практичне виконання системних дій у середовищі Windows як експериментальній платформі апробації;

– провести перевірку працездатності розробленого рішення на типових голосових командах користувача.

Таким чином, у межах кваліфікаційної роботи ставиться задача розробити локальний кросплатформений голосовий асистент, у якому поєднуються механізми активації голосової взаємодії, перетворення мовлення в текст, визначення інтенції команди, виділення її параметрів і виконання відповідної системної дії. Розроблювана система має забезпечувати повний цикл обробки голосової команди та надавати користувачеві можливість керувати типовими функціями комп’ютера у природній мовній формі.

2 МОДЕЛІ, МЕТОДИ ТА ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ОБРОБКИ ГОЛОСОВИХ КОМАНД У КРОСПЛАТФОРМЕНОМУ ГОЛОСОВОМУ АСИСТЕНТІ

2.1 Інформаційна технологія обробки голосової команди

Інформаційна технологія обробки голосових команд у межах розроблюваного асистента охоплює послідовність етапів, що забезпечують перехід від мовного сигналу користувача до виконання конкретної системної дії. На відміну від звичайного розпізнавання мовлення, така технологія не завершується отриманням тексту, а передбачає подальше визначення інтенції, виділення параметрів команди, перевірку результату та передавання його до модуля виконання.

У даній роботі обробка голосової команди організована як локальний конвеєр, у якому основні етапи виконуються без звернення до зовнішньої серверної інфраструктури. Доцільність такого підходу обґрунтовується тим, що для системи керування комп’ютером важливими є швидкість реакції, автономність роботи та обмеження передавання мовних даних за межі пристрою користувача. У власному дослідженні локальної та серверної обробки природної мови було показано, що локальна архітектура зменшує залежність від мережевих умов і усуває додаткові затримки, пов’язані з передаванням даних між клієнтом і сервером [4].

Загальна послідовність обробки команди в системі має такий вигляд: активація голосового введення, запис аудіофрагмента, локальне розпізнавання мовлення, нормалізація отриманого тексту, визначення інтенції, виділення параметрів команди, виконання системної дії та відображення результату в інтерфейсі користувача. У поточній реалізації активація голосового введення виконується через глобальну гарячу клавішу. Одним із поширених підходів до автоматичної активації голосових систем є keyword spotting, тобто виявлення ключового слова або короткої активаційної

фрази, після якої система переходить до розпізнавання повної команди [15]. Режим wake word передбачений як напрям розвитку інтерфейсу, однак не використовується як завершений backend-механізм у поточній версії системи.

Загальний час обробки голосової команди можна подати як суму тривалостей основних етапів в формулі:

$$T_{total} = T_{rec} + T_{stt} + T_{norm} + T_{\text{int}ent} + T_{params} + T_{exec}, \quad (2.1)$$

де T_{total} – загальний час обробки команди;

T_{rec} – час запису аудіофрагмента;

T_{stt} – час перетворення мовлення в текст;

T_{norm} – час нормалізації текстової команди;

$T_{\text{int}ent}$ – час визначення інтенції;

T_{params} – час виділення параметрів команди;

T_{exec} – час виконання системної дії.

Формула (2.1) відображає загальну модель часових витрат у локальному режимі роботи. Для серверної архітектури до цієї моделі необхідно було б додати час передавання даних до сервера та отримання відповіді, однак у межах розроблюваного застосунку ці складові відсутні завдяки локальній обробці.

На рисунку 2.1 подано узагальнену блок-схему інформаційної технології обробки голосової команди. Вона відображає послідовність переходу від активації голосового введення до виконання системної дії. Така структура дає змогу розділити процес на незалежні функціональні етапи, що спрощує подальше проєктування архітектури системи та програмну реалізацію окремих модулів.

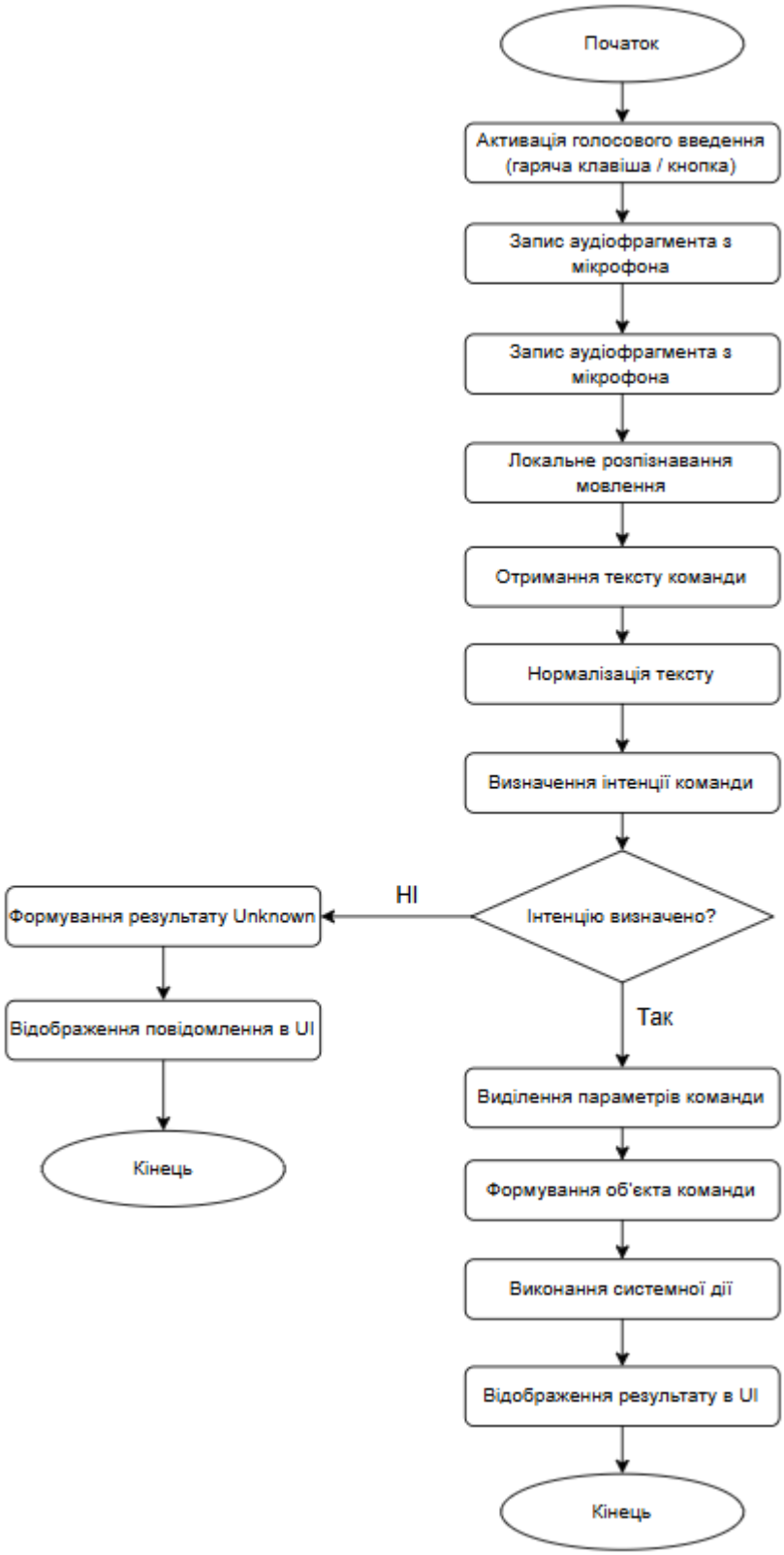

Рисунок 2.1 – Блок-схема інформаційної технології обробки голосової команди

2.2 Архітектура програмної системи голосового асистента

Архітектура розроблюваного голосового асистента побудована за модульним принципом, що дозволяє розділити процес обробки команди на незалежні функціональні частини. Такий підхід спрощує підтримку системи, дає змогу окремо розвивати модулі розпізнавання мовлення, визначення інтенції, виділення параметрів і виконання системних дій, а також забезпечує кращу керованість програмної реалізації.

У межах проєкту центральним компонентом є AssistantEngine, який виконує роль головної точки входу для обробки команди. Він приймає текстову команду, передає її до модуля інтерпретації, отримує структурований результат і ініціює виконання відповідної системної дії. Така побудова дозволяє відокремити загальну логіку роботи асистента від окремих технічних сервісів, що відповідають за машинне навчання, роботу з аудіо або взаємодію з операційною системою.

Логіка визначення команди реалізована через HybridCommandResolver, який поєднує rule-based та ML-підхід. Спочатку команда перевіряється за допомогою RuleBasedCommandResolver, що дає змогу швидко обробити очевидні або формально задані команди. Якщо правило не забезпечує надійного результату, використовується ML-шар, реалізований через TopKIntentPredictor. Такий підхід дозволяє поєднати передбачуваність правил із гнучкістю семантичного визначення інтенції на основі embedding-подібності. Доцільність використання ML.NET та пов’язаних із ним засобів для побудови локальних рішень семантичного аналізу мовленнєвих команд була обґрунтована у попередньому дослідженні [7].

ML-частина системи складається з кількох спеціалізованих компонентів. OnnxEmbeddingService відповідає за формування векторного представлення текстової команди за допомогою ONNX-моделі. VectorSimilarityService виконує обчислення подібності між embeddings. IntentExampleEmbeddingBuilder, IntentPrototypeBuilder,

NearestExampleIntentPredictor та TopKIntentPredictor забезпечують підготовку прикладів команд, порівняння нової команди з еталонними прикладами та вибір найбільш імовірної інтенції. Додаткові службові компоненти EmbeddingFilesValidator і OnnxModelInspector використовуються для перевірки наявності необхідних файлів моделі та контролю її структури.

Після визначення інтенції команда передається до виконавчого шару. У системі цю роль виконує WindowsCommandExecutor, який координує виконання дій у середовищі Windows. Безпосереднє виконання окремих операцій винесено в окремі сервіси: AppLauncher відкриває прикладні програми, NoteService створює локальні нотатки, VolumeControlService керує системною гучністю, а BrightnessControlService змінює рівень яскравості. Такий розподіл дозволяє уникнути надмірної концентрації логіки в одному класі та спрощує подальше розширення набору команд.

Окрему групу становлять компоненти, пов’язані з інтерфейсом, голосовим введенням та допоміжними можливостями застосунку. Ці сервіси забезпечують взаємодію між користувачем і внутрішньою логікою асистента, тому їхня роль полягає не лише в технічному прийманні вхідних даних, а й у створенні зручного сценарію використання програми. WhisperModelService забезпечує роботу з локальною моделлю розпізнавання мовлення та відповідає за її підготовку до використання в процесі перетворення аудіо в текст. MicrophoneRecorderService виконує запис аудіофрагмента з мікрофона, який надалі передається до модуля розпізнавання. SpeechToTextService здійснює безпосереднє перетворення мовлення в текстову форму, а SpeechTextNormalizer нормалізує отриманий результат перед подальшою інтерпретацією, зменшуючи вплив випадкових відмінностей у записі або розпізнаванні команди. Компоненти WindowsTrayIconService та WindowsHotkeyService відповідають за допоміжні механізми взаємодії із застосунком: роботу із системним треєм, згортання та відновлення вікна, а також активацію голосового введення через глобальну гарячу клавішу. Завдяки цьому асистент може працювати у фоновому режимі та залишатися

доступним для користувача без постійної взаємодії з основним вікном програми. Узагальнену структуру програмної системи показано в таблиці 2.1.

Таблиця 2.1 – Основні компоненти архітектури голосового асистента

Компонент	**Призначення**
AssistantEngine	Координує повний процес обробки команди
HybridCommandResolver	Поєднує rule-based та ML-підхід до інтерпретації
RuleBasedCommandResolver	Обробляє очевидні команди за правилами
TopKIntentPredictor	Визначає інтенцію на основі найближчих embedding-прикладів
OnnxEmbeddingService	Формує векторне представлення текстової команди
VectorSimilarityService	Обчислює подібність між векторними представленнями
WindowsCommandExecutor	Передає розпізнану команду до відповідного виконавчого сервісу
AppLauncher	Забезпечує відкриття програм
NoteService	Створює локальні текстові нотатки
VolumeControlService	Керує системною гучністю
BrightnessControlService	Керує яскравістю екрана
SpeechToTextService	Перетворює мовлення в текст
MicrophoneRecorderService	Записує аудіофрагмент з мікрофона
SpeechTextNormalizer	Нормалізує розпізнаний текст команди
AppSettingsService	Зберігає налаштування застосунку

Наведений розподіл компонентів демонструє модульну організацію системи, у якій окремо виділено сервіси введення, інтерпретації команд, формування векторних представлень та виконання системних дій. Такий підхід спрощує супровід програмного забезпечення, дозволяє ізольовано

змінювати окремі модулі та створює основу для подальшого розширення функціональності голосового асистента.

2.3 Модель голосового введення та локального розпізнавання мовлення

Модель голосового введення в розроблюваному застосунку призначена для перетворення мовної команди користувача в текстове подання, яке надалі передається до модуля інтерпретації. На цьому етапі система ще не визначає інтенцію і не виконує дію, а лише забезпечує коректне отримання аудіосигналу, його локальне розпізнавання та попередню нормалізацію результату.

У поточній реалізації активація голосового введення виконується через глобальну гарячу клавішу або відповідний елемент інтерфейсу. Такий підхід дозволяє уникнути постійного повного розпізнавання мовлення й запускати запис лише тоді, коли користувач явно ініціює голосову взаємодію. Режим wake word передбачений у логіці інтерфейсу як можливий напрям подальшого розвитку, проте в поточній версії backend-механізм безперервного виявлення активаційної фрази не використовується.

Основними компонентами цього етапу є WindowsHotkeyService, MicrophoneRecorderService, WhisperModelService, SpeechToTextService та SpeechTextNormalizer. WindowsHotkeyService забезпечує активацію запису через глобальну гарячу клавішу. MicrophoneRecorderService відповідає за отримання аудіофрагмента з мікрофона. WhisperModelService забезпечує доступ до локальної моделі розпізнавання мовлення, а SpeechToTextService виконує перетворення записаного аудіо в текст. Після цього SpeechTextNormalizer здійснює нормалізацію отриманого рядка, зокрема приведення тексту до зручнішої форми для подальшого аналізу.

Використання локального розпізнавання мовлення відповідає загальній архітектурній логіці проєкту. У дослідженнях on-device automatic speech recognition підкреслюється, що локальна обробка дає змогу зменшити залежність від серверної інфраструктури та виконувати основні етапи розпізнавання без передавання мовних даних до зовнішніх сервісів [3]. Для голосового асистента, орієнтованого на керування комп'ютером, це є важливим з погляду швидкодії, автономності та контролю над користувацькими даними. Аналогічний висновок було зроблено і у власному дослідженні локальної та серверної обробки природної мови, де локальна архітектура розглядалася як більш придатна для сценаріїв, у яких критичними є затримка відповіді, конфіденційність і стабільність роботи [4].

Формально етап голосового введення можна подати як послідовність перетворень:

$$A \xrightarrow{rec} S \xrightarrow{stt} T \xrightarrow{norm} T', \tag{2.2}$$

де A – вхідний аудіосигнал з мікрофона;

S – записаний аудіофрагмент;

T – текст, отриманий після розпізнавання мовлення;

T' – нормалізований текст команди, який передається до модуля визначення інтенції;

rec – операція запису аудіо;

stt – операція перетворення мовлення в текст;

$norm$ – операція нормалізації тексту.

Час виконання етапу голосового введення можна подати як суму тривалостей його основних складових:

$$T_{voice} = T_{act} + T_{rec} + T_{stt} + T_{norm}, \tag{2.3}$$

де T_{voice} – загальний час обробки голосового введення;

T_{act} – час активації голосового введення користувачем;

T_{rec} – час запису аудіофрагмента;

T_{stt} – час локального розпізнавання мовлення;

T_{norm} – час нормалізації отриманого тексту.

Формула (2.3) показує, що тривалість голосового введення найбільше залежить від запису аудіофрагмента та локального розпізнавання мовлення. Нормалізація тексту є коротшим етапом, однак вона важлива для подальшого визначення інтенції, оскільки зменшує вплив випадкових відмінностей у тексті команди.

Для конкретизації функцій компонентів, що беруть участь у перетворенні голосової команди в текст, їх доцільно подати у вигляді таблиці. Це дає змогу відокремити підсистему голосового введення від загальної архітектури асистента та простежити шлях команди від активації запису до передавання нормалізованого тексту. Основні компоненти моделі голосового введення наведено в табл. 2.2.

Таблиця 2.2 – Компоненти моделі голосового введення

Компонент	**Функція**
WindowsHotkeyService	Активація голосового введення через глобальну гарячу клавішу
MicrophoneRecorderService	Запис аудіофрагмента з мікрофона
WhisperModelService	Завантаження та обслуговування локальної моделі розпізнавання мовлення
SpeechToTextService	Перетворення записаного мовлення в текст
SpeechTextNormalizer	Нормалізація розпізнаного тексту перед подальшою інтерпретацією
AssistantEngine	Передавання текстової команди до модуля інтерпретації та виконання

2.4 Математична модель векторного представлення голосової команди

Після перетворення мовлення в текст голосова команда подається системі у вигляді короткого текстового висловлювання. Для подальшого визначення інтенції та порівняння команд між собою такий текст необхідно перетворити у числову форму. У межах розроблюваного голосового асистента для цього використовується векторне представлення команди, сформоване за допомогою моделі paraphrase-multilingual-MiniLM-L12-v2, що запускається локально через ONNX Runtime. Доцільність використання цієї моделі пов’язана з тим, що MiniLM застосовується для задач семантичної подібності речень і забезпечує компроміс між якістю векторного представлення та обчислювальною ефективністю [16].

Векторне представлення дає змогу перейти від поверхневої текстової форми команди до її семантичного подання. Це важливо для коротких українських голосових команд, оскільки один і той самий намір користувача може бути виражений різними словами або граматичними формами. Використання sentence embeddings є доцільним для таких задач, оскільки відповідні моделі формують векторні представлення речень і коротких текстів, придатні для подальшого зіставлення за семантичною подібністю [12].

Процес формування векторного представлення починається з токенізації текстової команди. Нормалізований текст T' перетворюється на послідовність токенів:

$$T' = \{t_1, t_2, ..., t_n\}, \tag{2.4}$$

де T' – нормалізований текст голосової команди;

t_i – i-й токен команди;

n – кількість токенів у послідовності.

Після токенізації послідовність токенів передається на вхід ONNX-моделі. На виході модель формує набір векторів прихованих станів

last_hidden_state, де кожному токену відповідає власне векторне представлення:

$$H = \{h_1, h_2, ..., h_n\}, \tag{2.5}$$

де H – нормалізований текст голосової команди;

h_i – i -й токен команди;

n – кількість токенів у послідовності.

Оскільки для подальшого порівняння потрібне не окреме представлення кожного токена, а єдиний вектор усієї команди, застосовується операція усереднення векторів токенів. З урахуванням attention mask вектор команди можна подати так:

$$E_x = \frac{\sum_{i=2}^{n} h_i \cdot m_i}{\sum_{i=2}^{n} m_i}, \tag{2.6}$$

де E_x – векторне представлення голосової команди *x*;

h_i – вектор прихованого стану i -го токена;

m_i – значення attention mask для i -го токена;

n – кількість токенів у послідовності.

Використання attention mask необхідне для того, щоб під час усереднення враховувалися лише змістові токени команди, а службові або доповнювальні елементи послідовності не спотворювали підсумкове представлення. Такий підхід відповідає загальній логіці sentence embeddings, де короткий текст або речення подається як один щільний вектор фіксованої розмірності, придатний для подальшого обчислення семантичної подібності [12].

Для забезпечення стабільнішого порівняння отриманий вектор може бути нормалізований:

$$\hat{E}_x = \frac{E_x}{\|E_x\|}, \tag{2.7}$$

де $\hat{E}_x$ – нормалізований вектор команди;

E_x – початкове векторне представлення команди;

$\|E_x\|$ – евклідова норма вектора.

Отримане векторне представлення використовується як основа для подальшого визначення інтенції. На цьому етапі система ще не виконує команду, а лише перетворює її текстову форму в числову структуру, з якою можуть працювати алгоритми порівняння. Таким чином, математична модель векторного представлення є проміжною ланкою між розпізнаним текстом команди та методом визначення інтенції на основі семантичної подібності.

2.5 Метод визначення інтенції на основі embedding-подібності

Після формування векторного представлення голосової команди система переходить до етапу визначення її інтенції. У межах розроблюваного асистента інтенція розглядається як узагальнений тип дії, яку користувач очікує від системи: відкриття програми, створення нотатки, зміна гучності, зміна яскравості, відкриття налаштувань або виконання іншої підтримуваної операції. У задачах Natural Language Understanding визначення інтенції є одним із ключових етапів, оскільки саме воно пов’язує природномовний запит користувача з подальшою дією програмної системи [8].

У проєкті інтенція визначається на основі семантичної подібності між embedding-представленням нової команди та embedding-представленнями еталонних прикладів. Для кожної інтенції формується набір типових команд, що відображають різні способи вираження одного наміру. Такий підхід відповідає логіці пошуку за запитом, де рішення приймається шляхом

порівняння вхідного опису з підготовленими еталонами [17]. Наприклад, для OpenApp використовуються команди «відкрий калькулятор», «запусти браузер», а для VolumeControl – «зроби гучніше», «зменш гучність», «постав звук на 50 відсотків».

Семантична подібність між новою командою та еталонним прикладом визначається за допомогою косинусної подібності:

$$sim(\hat{E}_x, \hat{E}_i) = \frac{\hat{E}_x \cdot \hat{E}_i}{\left\|\hat{E}_x\right\| \left\|\hat{E}_i\right\|}, \tag{2.8}$$

де $\hat{E}_x$ – нормалізоване векторне представлення нової команди;

$\hat{E}_i$ – нормалізоване векторне представлення *i*-го еталонного прикладу;

$sim(\hat{E}_x, \hat{E}_i)$ – значення подібності між командою та прикладом.

На основі отриманих значень подібності обирається еталонний приклад, який має найбільшу семантичну близькість до нової команди:

$$i^* = \arg\max_i sim(\hat{E}_x, \hat{E}_i), \tag{2.9}$$

де i^* – індекс найближчого еталонного прикладу;

i – індекс прикладу в наборі еталонних команд.

Відповідно, попередня інтенція нової команди визначається як інтенція найближчого еталонного прикладу:

$$Intent(x) = Intent(e_{i^*}), \tag{2.10}$$

де $Intent(x)$ – інтенція нової команди;

e_{i^*} – еталонний приклад, що має найбільшу подібність до команди *x*.

У практичній реалізації використовується не лише найближчий приклад, а й логіка top-k найближчих прикладів. Це дозволяє оцінювати не одиничний збіг, а загальну надійність результату серед кількох найближчих команд. Формально множину найближчих прикладів можна подати так:

$$K_x = TopK(sim(\hat{E}_x, \hat{E}_i), k), \tag{2.11}$$

де K_x – множина k найближчих еталонних прикладів до команди x;

$TopK$ – операція вибору k найбільших значень подібності;

k – кількість прикладів, що враховуються під час перевірки результату.

Застосування top-k логіки є важливим для підвищення стійкості системи. Якщо серед найближчих прикладів переважає одна інтенція, результат можна вважати більш надійним. Якщо ж найближчі приклади належать до різних інтенцій або максимальна подібність є недостатньою, система може віднести команду до класу Unknown. Це дає змогу уникати виконання випадкових або неоднозначно розпізнаних команд.

Перевірку надійності результату можна подати через порогове правило:

$$Intent(x) = \begin{cases} Intent(e_{i*}), sim(\hat{E}_x, \hat{E}_{i^*}) \geq \theta, \\ Unknown, sim(\hat{E}_x, \hat{E}_{i^*}) < \theta, \end{cases} \tag{2.12}$$

де θ – мінімально допустимий поріг семантичної подібності.

Перевага такого методу полягає в тому, що він дозволяє враховувати змістову близькість команд, а не лише буквальний збіг слів. У дослідженні Sentence-BERT показано, що sentence embeddings є придатними для задач semantic similarity та semantic search, тобто для порівняння речень або коротких текстів за змістом [12]. Це особливо важливо для голосових команд українською мовою, де один і той самий намір може бути поданий різними словоформами та синтаксичними конструкціями.

Доцільність використання embedding-представлень для визначення інтенції також пов’язана з тим, що sentence embeddings дозволяють порівнювати короткі тексти не лише за набором слів, а й за змістовою близькістю. У роботі Sentence-BERT показано, що такі векторні представлення можуть ефективно застосовуватися для задач semantic similarity та semantic search [12]. Це обґрунтовує використання embedding-based підходу як основного ML-механізму визначення інтенції в межах даного проєкту.

У реалізації голосового асистента цей метод представлений компонентами TopKIntentPredictor, NearestExampleIntentPredictor, OnnxEmbeddingService та VectorSimilarityService. OnnxEmbeddingService формує embedding нової команди, VectorSimilarityService обчислює подібність між векторами, а TopKIntentPredictor визначає найбільш імовірну інтенцію з урахуванням найближчих прикладів і перевірки надійності результату.

Отже, метод визначення інтенції на основі embedding-подібності забезпечує гнучку інтерпретацію коротких голосових команд, підвищує стійкість системи до варіативних формулювань і створює основу для подальшого розширення набору підтримуваних команд без повної перебудови логіки розпізнавання.

2.6 Гібридна модель прийняття рішення

У розроблюваному голосовому асистенті прийняття рішення щодо команди не обмежується лише одним методом інтерпретації. Для підвищення надійності використовується гібридна модель, яка поєднує rule-based підхід і ML-підхід на основі embedding-подібності [18]. Така побудова дозволяє швидко обробляти очевидні команди за правилами, а для варіативних

формулювань застосовувати семантичне зіставлення з еталонними прикладами.

Загальна логіка роботи гібридної моделі полягає в послідовній перевірці команди. Спочатку текст команди передається до RuleBasedCommandResolver, який намагається визначити очевидну інтенцію або параметри на основі правил, ключових слів і шаблонів. Якщо отриманий результат є достатньо надійним, система використовує його без звернення до ML-шару. Якщо rule-based підхід не дає коректного результату, команда передається до TopKIntentPredictor, де інтенція визначається на основі семантичної подібності до найближчих прикладів.

Формально вибір механізму інтерпретації можна подати так:

$$Intent(x) = \begin{cases} C_{rule}(x), q_{rule} = 1, \\ C_{ml}(x), q_{rule}(x) = 0 \wedge q_{ml}(x) = 1, \\ C_{unknown} = q_{rule}(x) = 0 \wedge q_{ml}(x) = 0 \end{cases}, \quad (2.13)$$

де $C(x)$ – результат інтерпретації команди x;

$C_{rule}(x)$ – результат, отриманий rule-based методом;

$C_{ml}(x)$ – результат, отриманий ML-методом;

$C_{unknown}$ – результат для невідомої або ненадійно розпізнаної команди;

$q_{rule}(x)$ – ознака успішності rule-based інтерпретації;

$q_{ml}(x)$ – ознака достатньої надійності ML-результату.

Після визначення інтенції система переходить до виділення параметрів команди. У задачах Natural Language Understanding цей етап відповідає виділенню змістових компонентів висловлювання, які уточнюють дію користувача [8]. У межах даного проєкту такими параметрами можуть бути назва програми, текст нотатки, розділ налаштувань, числове значення або напрям зміни системного параметра. Саме тому результат обробки команди

повинен містити не лише інтенцію, а й набір додаткових полів, потрібних для виконання дії.

Структурований результат команди можна подати у вигляді:

$$C = \{I, P, S\}, \tag{2.14}$$

де C – сформована команда;

I – визначена інтенція;

P – множина параметрів команди;

S – службовий статус успішності інтерпретації.

Множина параметрів залежить від типу інтенції:

$$P = \{p_1, p_2, \ldots, p_m\}, \tag{2.15}$$

де p_i – окремий параметр команди;

m – кількість параметрів, необхідних для виконання певної інтенції.

У практичній реалізації результат обробки команди формується через об’єкт AssistantExecutionResult. Він містить початковий текст команди, структурований результат її розпізнавання та ознаку фактичного виконання системної дії. Усередині об’єкта команди зберігаються поля Intent, IsSuccessful, AppName, NoteText, SettingsSection, ValuePercent і Direction, які описують тип дії та необхідні параметри її виконання. Така структура забезпечує передавання до виконавчого шару не лише визначеної інтенції, а й додаткових даних, потрібних для коректного запуску програми, створення нотатки, відкриття налаштувань або зміни системних параметрів.

Для узагальнення підтримуваних інтенцій, їхніх параметрів і відповідних системних дій доцільно подати таблицю. Вона дозволяє показати, які саме типи команд підтримує асистент і дані для їх виконання. Відповідність інтенцій, параметрів і системних дій показано в таблиці 2.3.

Таблиця 2.3 – Відповідність інтенцій, параметрів і системних дій

Інтенція	**Параметри**	**Приклад команди**	**Системна дія**
OpenApp	AppName	відкрий калькулятор	запуск програми
CreateNote	NoteText	створи нотатку купити хліб	створення локальної .txt-нотатки
VolumeControl	ValuePercent, Direction	зроби гучніше	зміна системної гучності
BrightnessControl	ValuePercent, Direction	зменш яскравість	зміна яскравості екрана
OpenSettings	SettingsSection	відкрий налаштування звуку	відкриття розділу налаштувань Windows
OpenTaskManager	-	відкрий диспетчер задач	запуск диспетчера задач
OpenExplorer	-	відкрий провідник	запуск провідника Windows
LockComputer	-	заблокуй комп’ютер	блокування комп’ютера
Unknown	-	нерозпізнана команда	дія не виконується

Гібридна модель прийняття рішення має кілька переваг для задачі голосового керування. Rule-based частина забезпечує контрольовану обробку очевидних команд і параметрів, тоді як ML-шар підвищує гнучкість системи в умовах варіативних формулювань. Такий підхід узгоджується з попереднім дослідженням застосування ML.NET у семантичному аналізі мовленнєвих

команд, де підкреслювалася доцільність поєднання засобів векторизації, класифікації та локальної інтеграції в C#-застосунках [7].

Отже, гібридна модель прийняття рішення дозволяє поєднати точність формальних правил із гнучкістю семантичного аналізу. У результаті система може коректно обробляти як прості передбачувані команди, так і варіативні висловлювання користувача, а також безпечно повертати результат Unknown, якщо інтерпретація не є достатньо надійною.

2.7 Об’єктно-орієнтована модель системи та обґрунтування вибору технологій реалізації

Об’єктно-орієнтована модель розроблюваного голосового асистента ґрунтується на поділі системи на окремі сервіси, кожен із яких відповідає за конкретний етап обробки команди. Такий підхід дозволяє уникнути концентрації всієї логіки в одному програмному модулі та забезпечує кращу підтримуваність, розширюваність і тестованість системи. У межах проєкту окремо виділено сервіси голосового введення, розпізнавання мовлення, інтерпретації команд, формування embeddings, порівняння векторів, виконання системних дій і збереження налаштувань.

Центральним елементом об’єктно-орієнтованої моделі є AssistantEngine, який координує загальний процес обробки команди. Він отримує текстову команду, передає її до HybridCommandResolver, отримує структурований результат інтерпретації та передає команду до WindowsCommandExecutor для виконання. Така структура відповідає принципу розділення відповідальностей, оскільки AssistantEngine не виконує безпосередньо ні ML-обчислень, ні системних дій, а лише організовує взаємодію між відповідними компонентами.

Модуль інтерпретації команди реалізований через HybridCommandResolver, який поєднує RuleBasedCommandResolver та

TopKIntentPredictor. Перший компонент відповідає за обробку очевидних команд і параметрів на основі правил, тоді як другий використовує embedding-представлення й механізм пошуку найближчих прикладів. Така побудова дає змогу поєднати керованість rule-based підходу з гнучкістю ML-методу. Доцільність використання засобів ML.NET і пов’язаних технологій у задачах семантичного аналізу мовленнєвих команд обґрунтовувалась у попередньому дослідженні, де було показано переваги інтеграції таких рішень у C#-застосунки [7].

Окрему групу становлять компоненти embedding-рівня. OnnxEmbeddingService відповідає за запуск ONNX-моделі та формування векторного представлення команди. VectorSimilarityService виконує обчислення подібності між векторами, а IntentExampleEmbeddingBuilder, IntentPrototypeBuilder, NearestExampleIntentPredictor і TopKIntentPredictor забезпечують підготовку прикладів, пошук найближчих команд і вибір інтенції. Службові компоненти EmbeddingFilesValidator та OnnxModelInspector використовуються для перевірки коректності файлів моделі й контролю її структури.

Виконавчий рівень системи представлений компонентом WindowsCommandExecutor, який отримує сформовану команду після етапу інтерпретації та передає її до відповідного сервісу виконання. Для відкриття програм використовується AppLauncher, для створення локальних нотаток – NoteService, для керування гучністю – VolumeControlService, а для зміни яскравості – BrightnessControlService. Кожен із цих сервісів відповідає лише за окремий тип системної дії, що забезпечує чіткий розподіл відповідальностей між компонентами. Такий підхід спрощує підтримку програмного коду та дозволяє розширювати набір підтримуваних дій без зміни загальної логіки обробки команд.Для узагальнення об’єктно-орієнтованої моделі системи доцільно подати основні класи та їхню роль у вигляді таблиці. Основні елементи об’єктно-орієнтованої моделі голосового асистента наведено в таблиці 2.4.

Таблиця 2.4 – Основні елементи об’єктно-орієнтованої моделі системи

Клас / сервіс	**Роль у системі**
AssistantEngine	Координує повний процес обробки команди
HybridCommandResolver	Поєднує rule-based та ML-інтерпретацію
RuleBasedCommandResolver	Обробляє очевидні команди та параметри за правилами
TopKIntentPredictor	Визначає інтенцію за найближчими embedding-прикладами
OnnxEmbeddingService	Формує векторне представлення текстової команди
VectorSimilarityService	Обчислює подібність між векторними представленнями
SpeechToTextService	Перетворює мовлення в текст
SpeechTextNormalizer	Нормалізує розпізнаний текст команди
WindowsCommandExecutor	Передає команду до відповідного виконавчого сервісу
AppLauncher	Виконує запуск програм
NoteService	Створює локальні текстові нотатки
VolumeControlService	Керує системною гучністю
BrightnessControlService	Керує яскравістю екрана
AppSettingsService	Зберігає налаштування застосунку
AssistantExecutionResult	Зберігає результат обробки та виконання команди

З технологічної точки зору основою реалізації обрано платформу .NET та мову C#, що забезпечує єдине середовище для побудови користувацького інтерфейсу, локальної логіки обробки команд і взаємодії з системними функціями. Використання .NET MAUI відповідає вимозі кросплатформеної основи застосунку, оскільки дозволяє проєктувати інтерфейс і частину прикладної логіки незалежно від конкретної операційної системи. Водночас

виконання системних дій реалізується через Windows-орієнтований шар, оскільки такі функції безпосередньо залежать від можливостей операційного середовища.

Для локального розпізнавання мовлення використовується Whisper.net, що дозволяє виконувати speech-to-text без звернення до зовнішніх серверів. Такий вибір узгоджується із загальною вимогою автономності системи та зменшення залежності від мережевого з'єднання. У власному дослідженні локальної та серверної обробки природної мови було показано, що локальна архітектура є доцільною для голосових асистентів, де важливими є швидкість реакції, конфіденційність і стабільність роботи незалежно від мережевих умов [4].

Для семантичної інтерпретації команд використовується ONNX Runtime та модель paraphrase-multilingual-MiniLM-L12-v2. Такий підхід дозволяє запускати модель локально, формувати sentence embeddings і визначати інтенцію команди за семантичною близькістю до еталонних прикладів. Водночас rule-based шар залишається важливим для виділення параметрів команд і обробки очевидних сценаріїв.

Отже, обрана об'єктно-орієнтована модель і технологічний стек відповідають вимогам розроблюваного голосового асистента. Модульна структура системи забезпечує розділення відповідальностей між компонентами, використання .NET і MAUI формує кросплатформену основу застосунку, Whisper.net забезпечує локальне розпізнавання мовлення, а ONNX-модель sentence embeddings дає змогу реалізувати семантичне визначення інтенції голосових команд.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ КРОСПЛАТФОРМЕНОГО ГОЛОСОВОГО АСИСТЕНТА

3.1 Обґрунтування інструментальних засобів реалізації

Програмна реалізація кросплатформеного голосового асистента виконувалася з використанням інструментальних засобів, які забезпечують розробку користувацького інтерфейсу, локальну обробку мовлення, семантичну інтерпретацію команд та виконання системних дій. Вибір технологій здійснювався з урахуванням теми кваліфікаційної роботи, вимог до локальної роботи застосунку, можливості інтеграції засобів машинного навчання та необхідності практичного керування функціями комп’ютера.

Основною мовою програмування обрано C#, а базовою платформою реалізації – .NET 8. Такий вибір зумовлений тим, що .NET забезпечує розвинену екосистему для створення настільних застосунків, роботи з асинхронними процесами, підключення зовнішніх бібліотек і взаємодії з операційною системою. Крім того, використання C# дозволяє реалізувати всі основні компоненти асистента в єдиному технологічному середовищі без необхідності винесення логіки обробки команд в окремі зовнішні сервіси.

Доцільність використання екосистеми .NET для задач обробки текстових даних підтверджується також досвідом реалізації NLP-метрик у середовищі ML.NET, де модульна структура обробки дає змогу поєднувати попередню підготовку тексту, обчислення кількісних показників і подальший аналіз результатів [19].

Для побудови інтерфейсу користувача використано .NET MAUI. Цей фреймворк призначений для створення кросплатформених застосунків з використанням C# і XAML та дає змогу формувати спільну кодову базу для різних платформ [20]. У межах даної роботи .NET MAUI використовується як технологічна основа для побудови інтерфейсу, налаштувань застосунку та організації взаємодії користувача з голосовим асистентом. Водночас у межах

цієї роботи виконання системних дій реалізовано для середовища Windows, оскільки відкриття програм, керування гучністю, яскравістю, налаштуваннями та блокуванням комп’ютера залежать від можливостей конкретної операційної системи.

Для локального розпізнавання мовлення використано бібліотеку Whisper.net. Вона дає змогу застосовувати Whisper-моделі в середовищі .NET та виконувати перетворення мовлення в текст без звернення до зовнішніх серверів [21]. У контексті розроблюваного голосового асистента це є важливим, оскільки система орієнтована на локальну обробку команд, зменшення залежності від мережевого з’єднання та забезпечення швидкої реакції на дії користувача.

Для формування векторних представлень голосових команд використано ONNX Runtime. ONNX Runtime є кросплатформеним середовищем для виконання моделей машинного навчання та дозволяє запускати моделі, підготовлені в різних ML-фреймворках [22]. У роботі цей інструмент застосовується для локального запуску моделі paraphrase-multilingual-MiniLM-L12-v2, яка формує sentence embeddings для подальшого визначення інтенції команди.

Окрему роль у реалізації відіграють допоміжні засоби, пов’язані з аудіовведенням, системним треєм і глобальною гарячою клавішею. Для запису аудіофрагментів використовується механізм роботи з мікрофоном, реалізований у відповідному сервісі застосунку. Активація голосового введення здійснюється через інтерфейс або глобальну гарячу клавішу, що дозволяє запускати розпізнавання без постійної взаємодії з основним вікном програми. Робота із системним треєм забезпечує можливість залишати асистент активним у фоновому режимі.

Для узагальнення використаних технологій доцільно подати їх у вигляді таблиці. Це дозволяє показати, які саме інструментальні засоби застосовано в роботі та яку роль вони виконують у програмній системі.

Основні технології та засоби реалізації голосового асистента наведено в табл. 3.1.

Таблиця 3.1 – Інструментальні засоби реалізації голосового асистента

Засіб / технологія	**Призначення в роботі**
C#	Основна мова програмування застосунку
.NET 8	Базова платформа реалізації програмної системи
.NET MAUI	Побудова кросплатформеної основи інтерфейсу користувача
XAML	Опис структури та елементів користувацького інтерфейсу
Whisper.net	Локальне перетворення мовлення в текст
ONNX Runtime	Локальний запуск embedding-моделі
paraphrase-multilingual-MiniLM-L12-v2	Формування векторних представлень голосових команд
NAudio / аудіосервіси	Запис аудіофрагментів з мікрофона
Windows API / системні засоби Windows	Виконання системних дій
Системний трей і глобальна гаряча клавіша	Фонова робота застосунку та швидка активація голосового введення

Загалом обрані інструментальні засоби відповідають поставленим вимогам до програмної реалізації. Вони забезпечують створення графічного інтерфейсу, локальне розпізнавання мовлення, запуск моделі машинного навчання, семантичну інтерпретацію команд і виконання системних операцій у середовищі Windows. Така комбінація технологій дозволяє реалізувати практичний голосовий асистент без використання зовнішньої серверної інфраструктури.

3.2 Структура програмного проєкту

Структура програмного проєкту голосового асистента сформована за модульним принципом, що дозволяє розділити основні функції системи між окремими сервісами. Такий підхід спрощує супровід програмного коду, полегшує тестування окремих компонентів і створює умови для подальшого розширення функціональності застосунку. У межах проєкту програмна логіка поділена на декілька основних груп: ядро обробки команд, модулі машинного навчання та векторного представлення тексту, сервіси голосового введення, компоненти виконання системних дій і засоби користувацького інтерфейсу.

Центральною частиною програмної системи є ядро обробки команд, яке координує перехід від текстової команди до виконання відповідної дії. Основним компонентом цього рівня є AssistantEngine, що приймає команду, передає її до модуля інтерпретації та отримує результат у вигляді структурованого об’єкта. Для поєднання різних підходів до обробки команд використовується HybridCommandResolver, який взаємодіє з RuleBasedCommandResolver і TopKIntentPredictor. Завдяки цьому система може обробляти як очевидні команди за правилами, так і варіативні формулювання за допомогою ML-підходу.

Окремий рівень програмної структури становлять компоненти, пов’язані з формуванням векторних представлень команд. До них належать OnnxEmbeddingService, VectorSimilarityService, IntentExampleEmbeddingBuilder, IntentPrototypeBuilder, NearestExampleIntentPredictor, TopKIntentPredictor, EmbeddingFilesValidator та OnnxModelInspector. Ці компоненти забезпечують підготовку embedding-представлень команд, перевірку файлів моделі, обчислення косинусної подібності та вибір найбільш імовірної інтенції.

Підсистема голосового введення містить сервіси, які забезпечують отримання аудіосигналу, його перетворення в текст і попередню підготовку результату до інтерпретації. MicrophoneRecorderService відповідає за запис аудіофрагмента з мікрофона, WhisperModelService забезпечує роботу з локальною моделлю розпізнавання мовлення, SpeechToTextService виконує перетворення мовлення в текст, а SpeechTextNormalizer нормалізує отриманий текст команди. Активація голосового введення в поточній реалізації здійснюється через елементи інтерфейсу або глобальну гарячу клавішу.

Виконання системних дій винесено в окремий шар, що забезпечує взаємодію з операційним середовищем Windows. Компонент WindowsCommandExecutor приймає розпізнану команду та передає її до відповідного виконавчого сервісу. Для запуску програм використовується AppLauncher, для створення локальних нотаток – NoteService, для керування гучністю – VolumeControlService, а для зміни яскравості – BrightnessControlService. Такий розподіл дозволяє розширювати перелік підтримуваних дій без суттєвої зміни загальної логіки асистента.

Допоміжні компоненти застосунку забезпечують збереження налаштувань і зручну взаємодію користувача з програмою. AppSettingsService відповідає за збереження параметрів запуску, режиму активації голосового введення та поведінки застосунку під час згортання або закриття. WindowsTrayIconService забезпечує роботу із системним треєм, що дозволяє залишати асистент активним у фоновому режимі. WindowsHotkeyService реалізує швидку активацію голосового введення через глобальну гарячу клавішу, завдяки чому користувач може запускати обробку команди під час роботи з іншими програмами.

Для узагальнення структури програмного проєкту доцільно подати основні групи компонентів у вигляді таблиці. Це дозволяє показати логічну організацію застосунку та роль кожної групи модулів у загальному процесі обробки голосової команди. Структуру основних компонентів програмного

проєкту наведено в табл. 3.2. Це забезпечує більш наочне подання структури реалізованого застосунку.

Таблиця 3.2 – Структура основних компонентів програмного проєкту

Група компонентів	**Основні класи / сервіси**	**Призначення**
Ядро обробки команд	AssistantEngine, HybridCommandResolver, RuleBasedCommandResolver	Координація обробки команди та поєднання rule-based і ML-підходу
ML та embeddings	OnnxEmbeddingService, VectorSimilarityService, TopKIntentPredictor, NearestExampleIntentPredictor	Формування векторних представлень, обчислення подібності та визначення інтенції
Перевірка ML-файлів	EmbeddingFilesValidator, OnnxModelInspector	Перевірка наявності та коректності файлів embedding-моделі
Голосове введення	MicrophoneRecorderService, WhisperModelService, SpeechToTextService, SpeechTextNormalizer	Запис аудіо, локальне розпізнавання мовлення та нормалізація тексту
Виконання системних дій	WindowsCommandExecutor, AppLauncher, NoteService, VolumeControlService,	Виконання дій у середовищі Windows
Інтерфейс і допоміжні функції	WindowsTrayIconService, WindowsHotkeyService, AppSettingsService	Робота з треєм, глобальною гарячою клавішею та налаштуваннями застосунку
Результат виконання	AssistantExecutionResult, об’єкт команди	Зберігання початкового тексту, визначеної інтенції, параметрів і статусу

Загалом така структура програмного проєкту відповідає логіці поетапної обробки голосової команди. Кожна група компонентів виконує окрему функцію: отримання мовлення, перетворення його в текст, семантичну інтерпретацію, формування структурованої команди та виконання системної дії. Це забезпечує модульність програмної реалізації та дозволяє в подальшому доповнювати асистент новими інтенціями, сервісами виконання або альтернативними моделями розпізнавання та інтерпретації команд.

3.3 Реалізація основних функціональних модулів

3.3.1 Реалізація ядра обробки команд

Ядро обробки команд є центральною частиною програмної реалізації голосового асистента, оскільки саме воно координує взаємодію між модулем інтерпретації команди та виконавчим шаром системи. У розробленому застосунку цю роль виконує клас AssistantEngine, який не працює безпосередньо з аудіосигналом, а отримує вже готовий текст після етапу локального розпізнавання мовлення та нормалізації. Такий підхід забезпечує чітке розділення відповідальностей між підсистемою голосового введення та логікою обробки команд.

На рівні архітектури AssistantEngine виступає координатором основного сценарію роботи асистента. Після отримання текстової команди він передає її до гібридного модуля інтерпретації, отримує структурований результат у вигляді об’єкта ResolvedCommand і, за потреби, ініціює виконання відповідної системної дії через WindowsCommandExecutor. Таким чином, клас не виконує самостійно ні розпізнавання інтенції, ні запуск конкретних системних операцій, а забезпечує узгоджену взаємодію між окремими сервісами.

Під час створення об’єкта AssistantEngine виконується ініціалізація основних компонентів, необхідних для подальшої обробки команд. Насамперед перевіряється наявність CSV-файлу з прикладами команд, який використовується для побудови embedding-представлень еталонних висловлювань. Далі створюється OnnxEmbeddingService, сервіс обчислення векторної подібності VectorSimilarityService, а також IntentExampleEmbeddingBuilder, який формує набір прикладів інтенцій на основі даних із CSV-файлу. На основі цих даних ініціалізується TopKIntentPredictor, що використовується для визначення інтенції команди за embedding-подібністю.

Для поєднання rule-based і ML-based підходів у ядрі створюється об’єкт HybridCommandResolver. Він отримує два основні компоненти: RuleBasedCommandResolver, який відповідає за обробку очевидних команд і виділення параметрів за правилами, та TopKIntentPredictor, який забезпечує визначення інтенції на основі семантичної близькості команди до еталонних прикладів. Окремо ініціалізується WindowsCommandExecutor, який використовується для фактичного виконання системних дій після успішної інтерпретації команди.

Основним методом класу AssistantEngine є ProcessCommand. Він приймає початковий текст команди та параметр execute, який визначає, чи потрібно виконувати системну дію після розпізнавання команди. Якщо значення execute дорівнює false, система лише виконує розбір команди без її фактичного виконання. Такий режим використовується в діагностичному режимі застосунку, де необхідно перевірити визначену інтенцію, параметри команди та статус успішності без відкриття програм.

Фрагмент реалізації методу ProcessCommand наведено в лістингу 3.1.

Лістинг 3.1 – Фрагмент методу обробки команди в класі AssistantEngine

public AssistantExecutionResult ProcessCommand(string rawText, bool execute = true)

```
{
    var command = _hybridResolver.Resolve(rawText);

    bool executed = false;

    if (execute && command.IsSuccessful)
    {
        executed = _executor.Execute(command);
    }

    return new AssistantExecutionResult
    {
        RawText = rawText ?? string.Empty,
        Command = command,
        Executed = executed
    };
}
```

Як видно з лістингу 3.1, обробка команди складається з трьох основних етапів. На першому етапі текст передається до методу Resolve гібридного resolver-а, який повертає об’єкт ResolvedCommand. На другому етапі перевіряється, чи дозволено виконання команди та чи була команда успішно розпізнана. Якщо обидві умови виконуються, команда передається до WindowsCommandExecutor. На третьому етапі формується об’єкт AssistantExecutionResult, який містить початковий текст, результат інтерпретації та ознаку фактичного виконання дії.

Для збереження результату обробки використовується клас AssistantExecutionResult. Він містить три основні поля: RawText, Command і Executed. Поле RawText зберігає початковий текст команди, отриманий після розпізнавання мовлення або введений у діагностичному режимі. Поле

Command містить структурований результат інтерпретації у вигляді об’єкта ResolvedCommand. Поле Executed показує, чи була відповідна системна дія фактично виконана.

Об’єкт ResolvedCommand є основною моделлю розпізнаної команди. Він містить інтенцію користувача, початковий текст і набір можливих параметрів, які можуть бути використані під час виконання дії. До таких параметрів належать назва програми AppName, текст нотатки NoteText, розділ налаштувань SettingsSection, відсоткове значення ValuePercent та напрям зміни Direction. Властивість IsSuccessful визначає успішність розпізнавання команди: команда вважається успішною, якщо її інтенція не дорівнює Unknown.

Перелік підтримуваних інтенцій задано в переліку AssistantIntent. У поточній реалізації система підтримує такі інтенції: OpenApp, CreateNote, VolumeControl, BrightnessControl, OpenSettings, OpenTaskManager, OpenExplorer, LockComputer та Unknown. Інтенція Unknown використовується для команд, які не вдалося надійно віднести до жодного з підтримуваних сценаріїв. Це дозволяє уникнути випадкового виконання некоректної або неповної команди.

Важливою особливістю ядра обробки команд є те, що воно працює саме з текстовими даними. Повний шлях голосової команди до ядра має такий вигляд: запис аудіо з мікрофона виконується у MicrophoneRecorderService, далі аудіофрагмент передається до SpeechToTextService, де за допомогою Whisper формується текст, після чого результат нормалізується в SpeechTextNormalizer. Лише після цього нормалізована текстова команда передається до AssistantEngine.ProcessCommand. Завдяки цьому підсистема розпізнавання мовлення та підсистема інтерпретації команд залишаються незалежними одна від одної.

Отже, реалізація ядра обробки команд забезпечує зв’язування основних частин голосового асистента в єдиний процес. AssistantEngine виконує роль

центрального координатора, який приймає текстову команду, передає її до гібридного модуля інтерпретації, контролює можливість виконання дії та повертає структурований результат. Така реалізація спрощує налагодження програми, підтримує діагностичний режим і створює основу для подальшого розширення функціональності системи.

3.3.2 Реалізація підсистеми голосового введення та локального розпізнавання мовлення

Підсистема голосового введення забезпечує перетворення усної команди користувача у текстову форму, яка надалі передається до ядра обробки команд. У програмній реалізації цей етап відокремлено від модуля визначення інтенції, що дозволяє незалежно обробляти аудіосигнал, розпізнаний текст і подальшу логіку інтерпретації команди. Основними компонентами цього рівня є MicrophoneRecorderService, WhisperModelService, SpeechToTextService та SpeechTextNormalizer.

У поточній версії застосунку активація голосового введення виконується через кнопку інтерфейсу або глобальну гарячу клавішу. Повноцінна backend-реалізація wake word у межах цієї версії не використовується і розглядається як напрям подальшого розвитку системи.

Запис голосової команди виконується у сервісі MicrophoneRecorderService. Для роботи з мікрофоном використовується бібліотека NAudio, за допомогою якої створюється аудіопотік у форматі WAV з параметрами 16000 Hz, 16 bit, 1 channel. Записані аудіофрагменти зберігаються у локальній папці застосунку FileSystem.AppDataDirectory/recordings, а назва файлу формується автоматично з урахуванням дати й часу запису. За замовчуванням тривалість запису становить 5 секунд, однак у застосунку передбачено можливість вибору іншого значення: 3, 5, 7 або 10 секунд.

Локальне розпізнавання мовлення реалізовано у сервісі SpeechToTextService з використанням бібліотеки Whisper.net; загалом моделі сімейства Whisper застосовуються для автоматичного транскрибування мовлення та подальшого перетворення аудіоданих у текстову форму [23, 24]. Для роботи застосунку використовується модель Whisper Small, яка зберігається у файлі ggml-small.bin. Завантаження та підготовку моделі забезпечує WhisperModelService: якщо модель уже наявна в локальній папці FileSystem.AppDataDirectory/models, вона використовується повторно, а якщо файл відсутній – виконується його завантаження. Після першого завантаження модель зберігається локально, тому подальше розпізнавання мовлення може виконуватися без підключення до інтернету.

Фрагмент ініціалізації Whisper-процесора наведено в лістингу 3.2.

Лістинг 3.2 – Ініціалізація Whisper-процесора для розпізнавання українських команд

```
string modelPath = await _whisperModelService.EnsureModelAsync();

_whisperFactory = WhisperFactory.FromPath(modelPath);

_processor = _whisperFactory
   .CreateBuilder()
   .WithLanguage("uk")
   .Build();
```

У лістингу 3.2 показано, що перед початком розпізнавання сервіс перевіряє наявність локального файлу моделі за допомогою методу EnsureModelAsync. Після цього модель завантажується через WhisperFactory, а параметр .WithLanguage("uk") задає українську мову розпізнавання. Це означає, що система не використовує автоматичне визначення мови, а одразу орієнтується на обробку українських голосових команд.

Після отримання тексту з Whisper результат передається до SpeechTextNormalizer. Цей сервіс приводить текст до нижнього регістру, прибирає зайві пробіли та розділові знаки, а також виправляє типові помилки розпізнавання. Попередня підготовка тексту є типовим етапом задач NLP, оскільки токенізація, нормалізація та фільтрація допомагають привести текстові дані до форми, придатної для подальшої автоматизованої обробки [19]. Наприклад, варіанти на кшталт відкрай, віткрий, откpий нормалізуються до форми відкрий, а розділені слова типу на лашту вання об’єднуються у налаштування. Додатково нормалізатор використовує словник командної лексики та відстань Левенштейна для виправлення близьких за написанням слів.

Окремо в нормалізаторі враховано команди створення нотаток. Після виявлення слів-маркерів, таких як нотатка, нотатку, замітка або нагадування, подальший текст зберігається без агресивного виправлення. Це важливо, оскільки вміст нотатки може містити довільні слова, які не належать до словника команд, і їх небажано автоматично змінювати.

Таким чином, реалізована підсистема голосового введення забезпечує повний перехід від аудіосигналу до нормалізованої текстової команди. Її роботу можна подати як послідовність: запис аудіо з мікрофона, збереження WAV-файлу, локальне розпізнавання мовлення за допомогою Whisper.net, нормалізація тексту та передавання результату до AssistantEngine. Завдяки такому розділенню підсистема розпізнавання мовлення залишається незалежною від модуля визначення інтенції, що спрощує підтримку й подальше вдосконалення застосунку.

3.3.3 Реалізація embedding-модуля

Embedding-модуль призначений для перетворення текстової команди у числове векторне представлення, яке надалі використовується для

визначення інтенції користувача. У програмній реалізації цей функціональний блок винесено в окремий сервіс OnnxEmbeddingService, що забезпечує завантаження токенізатора, запуск ONNX-моделі та формування єдиного вектора для всієї команди. Такий підхід дозволяє відокремити етап семантичного представлення тексту від логіки класифікації інтенцій.

Для формування embeddings використовується модель paraphrase-multilingual-MiniLM-L12-v2, збережена у форматі ONNX. Використання трансформерної моделі є доцільним, оскільки такі архітектури застосовують механізм self-attention для врахування взаємозв'язків між токенами послідовності та формують embedding-представлення вхідного тексту [25]. Шляхи до файлів моделі та токенізатора визначаються в класі EmbeddingPaths. У ньому передбачено два варіанти розташування файлів: локальний шлях у папці assets для зібраного застосунку та резервний шлях до папки models під час розробки. Така реалізація дозволяє використовувати одну й ту саму логіку як у середовищі розробки, так і після збирання програми.

Під час створення об'єкта OnnxEmbeddingService виконується перевірка наявності файлів tokenizer.json і model.onnx. Якщо один із необхідних файлів відсутній, генерується виняток FileNotFoundException, що дозволяє одразу виявити помилку конфігурації. Після цього токенізатор завантажується за допомогою Tokenizer.FromFile, а ONNX-модель – через InferenceSession.

Основний метод embedding-модуля – GetEmbedding. Він приймає текст команди, виконує токенізацію та формує вхідні тензори для ONNX-моделі. Під час підготовки вхідних даних використовується обмеження максимальної довжини послідовності до 128 токенів: коротші команди доповнюються за допомогою padding, а довші скорочуються за допомогою truncation. До моделі передаються input_ids, які містять числові ідентифікатори токенів, та attention_mask, що позначає значущі токени й дозволяє відокремити їх від службового доповнення. Додатковий тензор

token_type_ids використовується лише за наявності відповідного входу в експортованій ONNX-моделі.

Фрагмент формування embedding-вектора наведено в лістингу 3.3.

Лістинг 3.3 – Формування embedding-вектора на основі виходу ONNX-моделі

```
var output = results
    .First(x => x.Name == "last_hidden_state")
    .AsTensor<float>();

int hiddenSize = output.Dimensions[2];
float[] embedding = new float[hiddenSize];

long validTokenCount = 0;

for (int tokenIndex = 0; tokenIndex < sequenceLength; tokenIndex++)
{
    if (attentionMask[0, tokenIndex] == 0)
        continue;
    validTokenCount++;
    for (int hiddenIndex = 0; hiddenIndex < hiddenSize; hiddenIndex++)
    {
        embedding[hiddenIndex] += output[0, tokenIndex, hiddenIndex];
    }
}
if (validTokenCount > 0)
{
    for (int i = 0; i < embedding.Length; i++)
    {
        embedding[i] /= validTokenCount;
```

```
    }
}
```

Як видно з лістингу 3.3, для отримання векторного представлення використовується вихід ONNX-моделі last_hidden_state, який містить векторні представлення окремих токенів. Оскільки для подальшого визначення інтенції потрібен один вектор для всієї команди, у сервісі реалізовано усереднення векторів токенів. При цьому враховуються лише значущі токени, для яких значення attention_mask не дорівнює нулю. У результаті формується один embedding-вектор, який узагальнює семантичний зміст текстової команди.

Додатково в проєкті реалізовано допоміжні сервіси EmbeddingFilesValidator та OnnxModelInspector. Перший використовується для перевірки наявності основних файлів embedding-моделі, зокрема model.onnx, tokenizer.json, config.json, special_tokens_map.json, tokenizer_config.json та unigram.json. Другий дозволяє отримати інформацію про входи та виходи ONNX-моделі, що є корисним під час перевірки коректності підключення моделі до застосунку.

Таким чином, embedding-модуль забезпечує перехід від нормалізованого тексту команди до її числового векторного представлення. Отриманий embedding не виконує системну дію самостійно, а передається до модуля визначення інтенції, де порівнюється з векторними представленнями еталонних прикладів команд. Завдяки цьому система може враховувати не лише точний збіг слів, а й семантичну близькість різних формулювань користувацьких команд.

3.3.4 Реалізація модуля визначення інтенції

Модуль визначення інтенції призначений для встановлення наміру

користувача на основі текстової команди, отриманої після розпізнавання мовлення та нормалізації. У розробленому застосунку цей процес реалізовано не через точний збіг із заздалегідь заданими фразами, а через порівняння embedding-вектора поточної команди з embedding-векторами еталонних прикладів. Основними компонентами цього рівня є IntentExampleEmbeddingBuilder, TopKIntentPredictor та VectorSimilarityService.

Підготовка еталонних прикладів виконується у класі IntentExampleEmbeddingBuilder. Він зчитує CSV-файл із прикладами команд, у якому кожен рядок містить текст команди та відповідну інтенцію, наприклад OpenApp, OpenExplorer або LockComputer. Після зчитування кожен текстовий приклад передається до OnnxEmbeddingService, де перетворюється у векторне представлення. У результаті формується список об’єктів IntentExampleEmbedding, які містять початковий текст прикладу, його інтенцію та відповідний embedding-вектор.

Безпосереднє визначення інтенції виконується у класі TopKIntentPredictor. Для нової команди спочатку формується embedding-вектор, після чого він порівнюється з усіма еталонними прикладами. Обчислення близькості між векторами здійснюється у VectorSimilarityService за допомогою косинусної подібності. Далі приклади сортуються за спаданням значення подібності, і для подальшого аналізу обираються k найближчих прикладів. У поточній реалізації за замовчуванням використовується значення k = 5. Такий підхід дає змогу враховувати не лише один найближчий приклад, а сукупність кількох найбільш подібних команд. Завдяки цьому зменшується ризик помилкового визначення інтенції у випадку, якщо окремий еталонний приклад має випадково високе значення подібності.

Фрагмент логіки вибору інтенції на основі найближчих embedding-прикладів, обчислення їх подібності та подальшого групування за інтенціями наведено в лістингу 3.4.

Лістинг 3.4 – Фрагмент визначення інтенції у TopKIntentPredictor

```
var ranked = _examples
    .Select(example => new
    {
        Example = example,
        Similarity = _similarityService.CosineSimilarity(embedding, example.Vector)
    })
    .OrderByDescending(x => x.Similarity)
    .Take(k)
    .ToList();

var grouped = ranked
    .GroupBy(x => x.Example.Intent)
    .Select(group => new
    {
        Intent = group.Key,
        AverageSimilarity = group.Average(x => x.Similarity),
        BestSimilarity = group.Max(x => x.Similarity)
    })
    .OrderByDescending(x => x.AverageSimilarity)
    .ThenByDescending(x => x.BestSimilarity)
    .ToList();
```

Як видно з лістингу 3.4, після обчислення подібності система не обмежується лише одним найближчим прикладом. Спочатку вибираються найближчі приклади, після чого вони групуються за інтенціями. Для кожної групи обчислюється середнє значення подібності та найкраще значення подібності серед знайдених прикладів. Це дозволяє враховувати не тільки

одиничний збіг, а й загальну близькість команди до групи прикладів певної інтенції.

Для запобігання помилковому визначенню інтенції в модулі використовується перевірка впевненості. У класі TopKIntentPredictor задано два порогові значення: MinAverageSimilarity = 0.70 та MinBestSimilarity = 0.82. Команда вважається достатньо надійно розпізнаною, якщо середня подібність для найкращої групи не менша за 0.70 або найкращий окремий збіг не менший за 0.82. Якщо ці умови не виконуються, результат позначається як недостатньо впевнений і надалі може бути оброблений як Unknown.

Окремо в проєкті реалізовано клас NearestExampleIntentPredictor, який визначає інтенцію за одним найближчим прикладом. Однак у центральному ядрі системи використовується саме TopKIntentPredictor, оскільки він враховує групу найближчих прикладів і тому є стійкішим до випадкових збігів. Такий підхід є доцільним для коротких голосових команд, оскільки одна й та сама дія може бути сформульована різними словами, але мати близьке семантичне значення.

Модуль визначення інтенції забезпечує семантичне зіставлення команди користувача з набором еталонних прикладів. CSV-файл у цій реалізації виконує роль джерела прикладів інтенцій, embedding-модуль перетворює текст у векторну форму, а TopKIntentPredictor визначає найбільш імовірний намір користувача на основі косинусної подібності. Це дозволяє системі працювати з варіативними формулюваннями команд і не обмежуватися лише фіксованими шаблонами.

3.3.5 Реалізація гібридної обробки команд і параметрів

Гібридна обробка команд у розробленому застосунку реалізована для поєднання двох підходів: rule-based обробки та визначення інтенції за

допомогою ML-модуля. Такий підхід дає змогу, з одного боку, швидко й передбачувано обробляти очевидні команди, а з іншого – використовувати embedding-подібність для варіативних формулювань, які можуть не збігатися з наперед заданими шаблонами.

Центральним компонентом цього рівня є клас HybridCommandResolver. Він спочатку передає команду до RuleBasedCommandResolver. Якщо rule-based обробка змогла визначити інтенцію, результат одразу повертається до ядра системи. Якщо ж команда не була розпізнана правилами, застосовується TopKIntentPredictor, який визначає інтенцію на основі семантичної близькості до еталонних прикладів. Фрагмент реалізації цієї логіки наведено в лістингу 3.5.

Лістинг 3.5 – Гібридна обробка команди у HybridCommandResolver

```
public ResolvedCommand Resolve(string rawText)
{
    var ruleResult = _ruleResolver.Resolve(rawText);
    if (ruleResult.IsSuccessful)
    {
        return ruleResult;
    }
    var mlResult = _topKPredictor.Predict(rawText, k: 5);
    if (!mlResult.IsConfident)
    {
        return new ResolvedCommand
        {
            RawText = rawText ?? string.Empty,
            Intent = AssistantIntent.Unknown
        };
    }
    return new ResolvedCommand
```

```
{
    RawText = rawText ?? string.Empty,
    Intent = mlResult.Intent
};}
```

Як видно з лістингу 3.5, першим етапом є перевірка команди за правилами. Якщо результат має інтенцію, відмінну від Unknown, команда вважається успішно розпізнаною. Якщо правило не спрацювало, система переходить до ML-рівня. При цьому результат ML-модуля використовується лише за умови достатньої впевненості. Якщо TopKIntentPredictor не підтверджує надійність визначення інтенції, повертається команда зі станом Unknown, і системна дія не виконується.

Rule-based обробка реалізована у класі RuleBasedCommandResolver. Його призначення полягає не лише у визначенні очевидних інтенцій, а й у виділенні параметрів, необхідних для виконання дії. Наприклад, для інтенції OpenApp визначається параметр AppName, для CreateNote – текст нотатки NoteText, для OpenSettings – розділ налаштувань SettingsSection, а для команд керування гучністю та яскравістю – напрям зміни Direction і, за наявності, відсоткове значення ValuePercent.

У RuleBasedCommandResolver використовується набір маркерів, за якими система визначає тип команди. Наприклад, наявність слів нотат, заміт або нагадув приводить до визначення інтенції CreateNote; слова гучн, звук, голосн, тих використовуються для інтенції VolumeControl; слова яскрав, екран, диспле, підсв – для BrightnessControl. Для відкриття програм використовується словник застосунків AppDictionary, у якому мовні назви програм зіставляються з їхніми системними іменами або командами запуску.

Окрему роль у rule-based обробці відіграє виділення параметрів. Метод ExtractPercent знаходить числове значення в команді, методи ResolveVolumeDirection і ResolveBrightnessDirection визначають напрям зміни параметра, а ExtractSettingsSection перетворює природні назви розділів налаштувань на внутрішні значення, наприклад sound, display, bluetooth або

network. Для створення нотаток використовується метод ExtractNoteText, який виділяє текст після слів-маркерів на кшталт нотатку, замітку або нагадування.

Важливо, що ML-модуль у цій реалізації переважно відповідає за визначення інтенції, тоді як параметри команди обробляються правилами. Це є доцільним рішенням для голосового асистента, оскільки намір користувача може бути сформульований варіативно, але параметри системної дії повинні бути виділені контрольовано. Наприклад, назва програми, напрям зміни гучності або розділ налаштувань мають бути визначені достатньо точно, і для цього краще підходять словники, маркери та шаблони.

Отже, гібридна обробка команд забезпечує баланс між гнучкістю ML-підходу та передбачуваністю rule-based логіки. Якщо команда має очевидну структуру, вона швидко обробляється правилами разом із параметрами. Якщо ж правило не спрацьовує, система використовує embedding-based визначення інтенції. За відсутності достатньої впевненості повертається стан Unknown, що зменшує ризик помилкового виконання системної дії.

3.3.6 Реалізація виконавчого шару системних дій

Виконавчий шар голосового асистента відповідає за перетворення структурованої команди на конкретну системну дію в середовищі Windows. На цьому етапі система вже не аналізує природну мову, а працює з об’єктом ResolvedCommand, у якому визначено інтенцію користувача та необхідні параметри. Центральним компонентом цього рівня є клас WindowsCommandExecutor, який отримує розпізнану команду й передає її до відповідного сервісу виконання.

У методі Execute спочатку перевіряється, чи команда не є порожньою та чи її інтенція не дорівнює Unknown. Якщо команда не була розпізнана, системна дія не виконується. Далі використовується конструкція switch, яка

зіставляє інтенцію з конкретним способом виконання: відкриття програми, створення нотатки, зміна гучності, зміна яскравості, відкриття провідника, запуск диспетчера задач, відкриття налаштувань або блокування комп'ютера. Фрагмент маршрутизації команди наведено в лістингу 3.6.

Лістинг 3.6 – Маршрутизація системних дій у WindowsCommandExecutor

```
public bool Execute(ResolvedCommand command)
{
    if (command is null || command.Intent == AssistantIntent.Unknown)
    {
        return false;
    }

    return command.Intent switch
    {
        AssistantIntent.OpenApp => _appLauncher.Launch(command.AppName),
        AssistantIntent.CreateNote => _noteService.CreateNote(command.NoteText),
        AssistantIntent.VolumeControl => _volumeControlService.Execute(command.Direction, command.ValuePercent),
        AssistantIntent.BrightnessControl => _brightnessControlService.Execute(command.Direction, command.ValuePercent),
        AssistantIntent.OpenExplorer => OpenExplorer(),
        AssistantIntent.OpenTaskManager => OpenTaskManager(),
        AssistantIntent.OpenSettings => OpenSettings(command.SettingsSection),
        AssistantIntent.LockComputer => LockComputer(),
        _ => false
```

```
    };
}
```

Як видно з лістингу 3.6, виконавчий шар не визначає інтенцію самостійно, а лише реагує на вже сформований результат інтерпретації. Для відкриття програм використовується сервіс AppLauncher, у якому внутрішні назви застосунків зіставляються з виконуваними файлами, наприклад Calculator – з calc.exe, Notepad – з notepad.exe, Paint – з mspaint.exe, VS Code – з Code.exe. Запуск виконується через Process.Start із параметром UseShellExecute = true, що дозволяє відкривати відповідні програми засобами операційної системи

Створення нотаток реалізовано у класі NoteService. Якщо команда містить текст нотатки, сервіс створює папку VoiceAssistantNotes у каталозі документів користувача, формує ім’я файлу з урахуванням дати й часу та записує вміст у текстовий файл формату .txt з кодуванням UTF-8. Після створення файл відкривається стандартними засобами Windows. Така реалізація дозволяє швидко створювати прості локальні нотатки без використання зовнішніх сервісів.

Керування гучністю реалізовано в VolumeControlService з використанням засобів NAudio. Сервіс отримує стандартний аудіопристрій відтворення та змінює параметр MasterVolumeLevelScalar. Підтримуються команди вимкнення звуку, увімкнення звуку, встановлення конкретного рівня гучності, а також збільшення або зменшення поточного значення. Якщо користувач не вказує точний відсоток, для зміни використовується стандартний крок.

Зміна яскравості екрана реалізована у BrightnessControlService через WMI-запити до простору root\WMI. Спочатку сервіс отримує поточне значення яскравості, після чого залежно від напряму команди збільшує, зменшує або встановлює нове значення. Підсумкове значення обмежується діапазоном від 0 до 100. Такий механізм залежить від підтримки відповідних

WMI-методів конкретним дисплеєм або пристроєм, тому найбільш коректно працює на системах, де Windows надає доступ до керування яскравістю.

Окремі системні дії реалізовано без додаткових сервісів безпосередньо в WindowsCommandExecutor. Для відкриття провідника використовується запуск explorer.exe, для відкриття диспетчера задач – taskmgr.exe. Відкриття налаштувань Windows виконується через URI-схеми ms-settings, наприклад ms-settings:sound, ms-settings:display, ms-settings:bluetooth або ms-settings:network. Якщо конкретний розділ не визначено, відкривається загальна сторінка налаштувань. Блокування комп'ютера реалізовано через виклик функції LockWorkStation з бібліотеки user32.dll.

Усі виконавчі методи повертають логічний результат виконання: true у разі успішного запуску дії та false у разі помилки або відсутності необхідних параметрів. Це значення передається назад до AssistantExecutionResult і може бути використане інтерфейсом для відображення результату користувачу або для діагностики в режимі розробника. Завдяки цьому система не лише виконує команду, а й фіксує, чи була дія фактично реалізована.

Таким чином, виконавчий шар забезпечує практичне керування комп'ютером на основі результатів інтерпретації голосової команди. Його реалізація побудована за сервісним принципом: окремі типи дій винесені у відповідні класи, а WindowsCommandExecutor виконує роль маршрутизатора між інтенцією та конкретним системним механізмом. Це спрощує подальше розширення асистента, оскільки для додавання нової дії достатньо реалізувати новий сервіс і додати відповідну гілку обробки інтенції.

3.4 Реалізація користувацького інтерфейсу та діагностичного режиму

Користувацький інтерфейс голосового асистента реалізовано з використанням .NET MAUI та XAML. Його основним призначенням є забезпечення зручної взаємодії користувача із застосунком, відображення

поточного стану системи, запуск голосового введення, налаштування режимів роботи та перегляд результатів виконання команд. Під час розробки інтерфейсу враховувалася необхідність відокремлення основного користувацького сценарію від допоміжних засобів тестування, тому в застосунку передбачено звичайний режим роботи та окремий діагностичний режим.

У звичайному режимі користувач взаємодіє із застосунком через основні елементи керування, призначені для активації голосового введення та перегляду результату виконання команди. Основний сценарій роботи передбачає запуск програми, активацію запису голосової команди через кнопку або глобальну гарячу клавішу, перетворення мовлення в текст, визначення інтенції та виконання відповідної системної дії. Після завершення обробки результат відображається у зрозумілій для користувача формі, без надмірної технічної інформації.

Важливою особливістю реалізації є те, що ручне введення команди не розглядається як основний спосіб використання голосового асистента. Воно винесене до діагностичного режиму та призначене для перевірки роботи модуля інтерпретації команд без необхідності кожного разу виконувати запис аудіо. Такий підхід дозволяє зберегти основну логіку застосунку як голосового інтерфейсу, але водночас забезпечує зручний інструмент для налагодження, тестування та демонстрації роботи ML-модуля.

Діагностичний режим, або режим розробника, реалізовано як допоміжний інструмент для перевірки внутрішніх результатів обробки команди. У цьому режимі користувач або розробник може ввести команду вручну, окремо виконати її розбір, перевірити визначену інтенцію, переглянути статус успішності розпізнавання та значення параметрів, які були виділені з команди. До таких параметрів належать назва програми, текст нотатки, розділ налаштувань, відсоткове значення, напрям зміни гучності або яскравості та ознака виконання системної дії. Використання такого режиму узгоджується з підходами до онлайн-моніторингу та

діагностики, де фіксація поточних результатів обробки застосовується для своєчасного виявлення помилок і підвищення надійності роботи системи [26].

Для відображення результату обробки використовується структурована інформація, отримана з об’єкта AssistantExecutionResult. У звичайному режимі вона подається у спрощеному вигляді, наприклад як повідомлення про розпізнану або виконану дію. У режимі розробника відображаються технічні поля, зокрема RawText, Intent, IsSuccessful, Executed, AppName, NoteText, SettingsSection, ValuePercent і Direction. Це дає змогу перевірити не лише факт виконання команди, а й правильність її внутрішньої інтерпретації.

Для наочного подання реалізованого інтерфейсу доцільно використати скриншоти основних режимів роботи застосунку. Це дозволяє не лише описати функції окремих елементів, а й показати фактичний вигляд програмного засобу, структуру вікна, спосіб відображення результатів та розмежування звичайного й діагностичного режимів. Головне вікно голосового асистента у звичайному режимі подано на рис. 3.1.

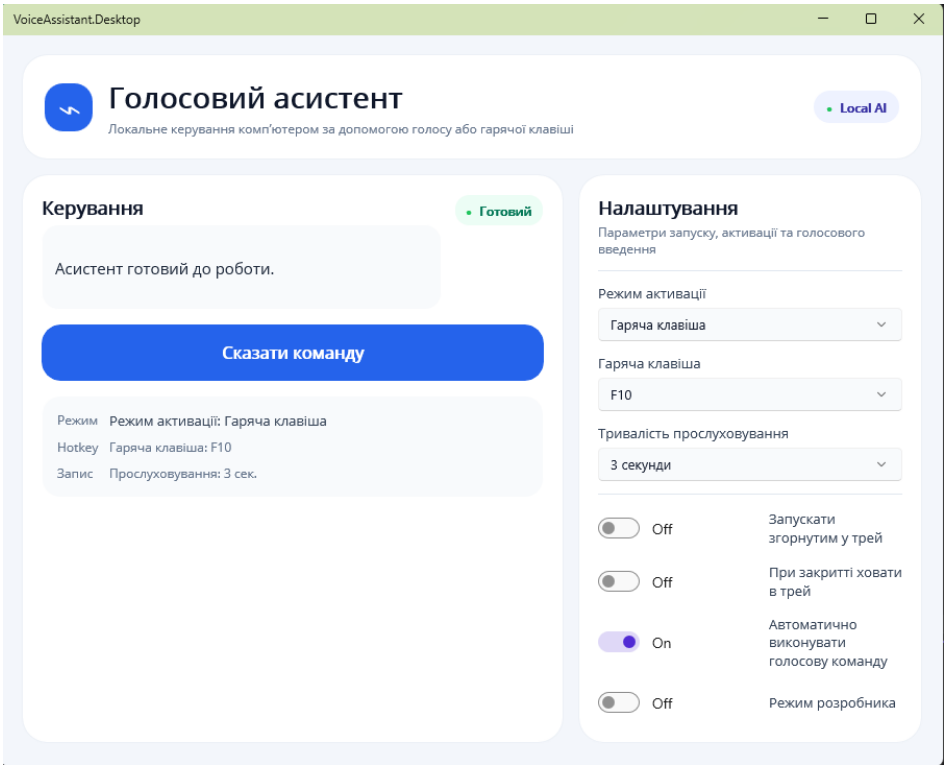

Рисунок 3.1 – Головне вікно голосового асистента у звичайному режимі

У звичайному режимі інтерфейс містить лише основні елементи, необхідні для користувацької взаємодії із застосунком. Користувач може активувати голосове введення, переглянути поточний статус асистента та отримати результат виконання команди у спрощеній формі. Такий підхід дозволяє не перевантажувати інтерфейс технічною інформацією та зберегти основний сценарій використання саме як голосове керування комп’ютером.

Діагностичний режим голосового асистента подано на рис. 3.2.

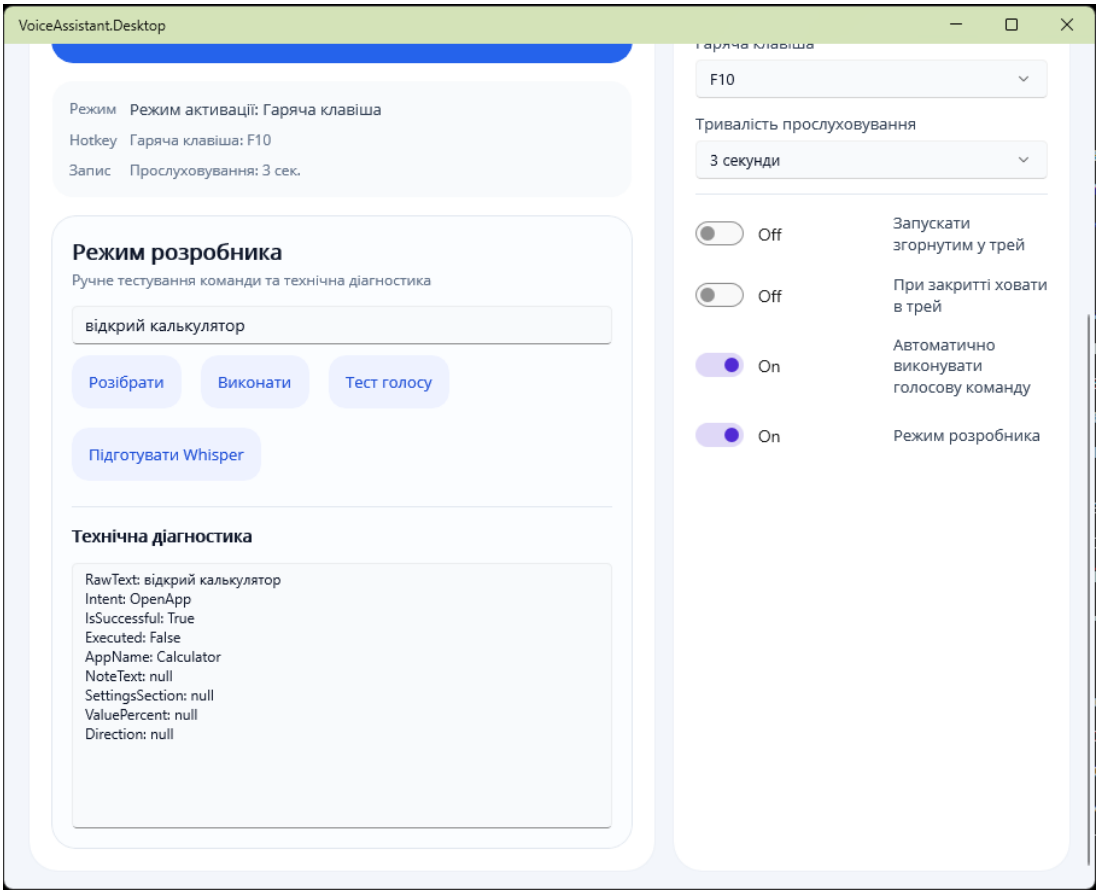

Рисунок 3.2 – Відображення діагностичного режиму голосового асистента

У діагностичному режимі відображаються додаткові елементи, призначені для перевірки внутрішньої логіки обробки команд. Зокрема, доступне ручне введення текстової команди, перегляд визначеної інтенції, статусу успішності, ознаки виконання дії та параметрів, виділених із команди. При цьому ручне введення не є основним способом використання

асистента, а застосовується лише як допоміжний інструмент тестування та налагодження.

Для підвищення зручності використання в застосунку реалізовано підтримку глобальної гарячої клавіші. Вона дозволяє активувати голосове введення без необхідності постійно відкривати головне вікно програми. Такий механізм є важливим для голосового асистента, оскільки користувач може взаємодіяти з ним у фоновому режимі, не перериваючи роботу з іншими програмами. Налаштування гарячої клавіші зберігається в параметрах застосунку та використовується під час наступних запусків.

Окрему роль у реалізації інтерфейсу відіграє робота із системним треєм. Застосунок може згортатися в трей і залишатися активним у фоновому режимі. Це відповідає характеру голосового асистента, який не повинен постійно займати місце на екрані, але має залишатися доступним для швидкої активації. Через трей користувач може відновити головне вікно або завершити роботу програми. Додатково в налаштуваннях передбачено параметри поведінки під час запуску та закриття вікна.

В цілому реалізований інтерфейс поєднує два рівні взаємодії із системою. Перший рівень орієнтований на звичайного користувача та забезпечує простий сценарій голосового керування комп’ютером. Другий рівень призначений для розробника і дає змогу перевіряти внутрішню логіку обробки команд, аналізувати визначену інтенцію та контролювати правильність виділення параметрів. Такий підхід дозволяє одночасно забезпечити зручність використання застосунку та можливість його подальшого тестування й удосконалення.

3.5 Інструкція користувача

Інструкція користувача призначена для опису основних дій, необхідних для запуску та використання розробленого голосового асистента. Оскільки

застосунок орієнтований на голосове керування комп’ютером, основним сценарієм взаємодії є активація голосового введення, промовляння команди, її розпізнавання, визначення інтенції та виконання відповідної системної дії. Ручне введення команди використовується лише в діагностичному режимі та не розглядається як основний спосіб роботи із системою.

Перед початком використання користувач запускає застосунок голосового асистента у середовищі Windows. Після запуску відкривається головне вікно програми, у якому відображається поточний стан асистента, доступні елементи керування та результат останньої обробленої команди. Якщо в налаштуваннях увімкнено запуск у згорнутому режимі, застосунок може відкриватися одразу в системному треї, залишаючись активним у фоновому режимі.

Основний порядок використання голосового асистента складається з таких етапів:

Етап 1. Запустити застосунок голосового асистента.

Етап 2. Переконатися, що мікрофон підключений і доступний для використання.

Етап 3. Активувати голосове введення за допомогою кнопки в інтерфейсі або глобальної гарячої клавіші.

Етап 4. Промовити голосову команду природною мовою, наприклад: «відкрий калькулятор», «створи нотатку купити хліб», «зроби гучніше», «відкрий диспетчер задач».

Етап 5. Дочекатися завершення запису та обробки команди.

Етап 6. Переглянути результат виконання у вікні застосунку.

Етап 7. За потреби повторити команду або змінити налаштування роботи асистента.

Після активації голосового введення застосунок записує аудіофрагмент з мікрофона та передає його до модуля локального розпізнавання мовлення. Розпізнаний текст нормалізується і передається до модуля інтерпретації, де визначається інтенція команди та, за потреби, виділяються її параметри.

Якщо команда розпізнана успішно, система виконує відповідну дію в операційному середовищі Windows. У разі недостатньої впевненості або невідповідності команди підтримуваним інтенціям дія не виконується, а користувач отримує повідомлення про те, що команду не вдалося коректно розпізнати.

Користувач може застосовувати голосовий асистент для виконання типових системних дій. До таких дій належать відкриття програм, створення локальних нотаток, керування гучністю, зміна яскравості, відкриття налаштувань Windows, запуск диспетчера задач, відкриття провідника та блокування комп’ютера. Команди можуть формулюватися у природній мовній формі, однак для підвищення точності роботи доцільно використовувати чіткі та короткі формулювання.

Окремо в застосунку передбачено можливість роботи через глобальну гарячу клавішу. Цей механізм дозволяє активувати голосове введення без відкриття головного вікна програми. Такий спосіб є зручним у випадках, коли користувач працює з іншими програмами, але хоче швидко подати голосову команду асистенту. Після натискання заданої комбінації клавіш застосунок переходить до запису голосової команди, після чого виконує стандартний цикл обробки.

Для забезпечення фонової роботи реалізовано підтримку системного трею. Користувач може згорнути застосунок у трей, не завершуючи його роботу. У такому режимі асистент не займає місце на панелі задач, але залишається доступним для подальшого використання. Через значок у системному треї можна відновити головне вікно або завершити роботу програми. Такий режим відповідає логіці роботи голосового асистента, який має бути доступним у фоновому режимі та не вимагати постійної присутності основного вікна на екрані.

У налаштуваннях застосунку користувач може визначити поведінку програми під час запуску та закриття. Зокрема, передбачено параметри запуску у згорнутому вигляді, згортання в трей під час закриття вікна та

вибір способу активації голосового введення. Такі налаштування дозволяють адаптувати застосунок до індивідуального сценарію використання та зробити взаємодію з асистентом більш зручною.

Діагностичний режим призначений для перевірки правильності роботи внутрішніх модулів системи. У цьому режимі доступне ручне введення текстової команди, її розбір без запису аудіо, перегляд визначеної інтенції, статусу успішності, ознаки виконання та параметрів команди. Цей режим використовується переважно для тестування, налагодження та демонстрації роботи модуля інтерпретації команд. Для звичайного користувацького сценарію основним способом взаємодії залишається саме голосове введення.

Загалом порядок використання розробленого застосунку є послідовним і не потребує від користувача спеціальних технічних знань. Основна взаємодія зводиться до запуску асистента, активації голосового введення та промовляння команди. Додаткові можливості, такі як системний трей, глобальна гаряча клавіша та діагностичний режим, розширюють зручність використання програми й полегшують її тестування та подальший розвиток.

3.6 Тестування функціональних можливостей голосового асистента

Тестування розробленого голосового асистента виконувалося з метою перевірки працездатності основних функціональних модулів, правильності визначення інтенцій користувача, коректності виділення параметрів команд та виконання відповідних системних дій. Оскільки застосунок поєднує декілька етапів обробки, тестування охоплювало як повний сценарій роботи з голосовою командою, так і окрему перевірку модуля інтерпретації в діагностичному режимі.

Основним об’єктом тестування є процес проходження команди через усі етапи системи: активацію голосового введення, запис аудіофрагмента, локальне розпізнавання мовлення, нормалізацію тексту, визначення інтенції,

виділення параметрів і виконання дії. Додатково перевірялася робота допоміжних можливостей застосунку, зокрема глобальної гарячої клавіші, системного трею, режиму розробника та відображення результату обробки в інтерфейсі.

Під час тестування використовувалися команди, що відповідають основним інтенціям, підтримуваним у програмній реалізації. Тестові приклади добиралися так, щоб охопити як команди виконання системних дій, так і випадки нерозпізнаних або непідтримуваних запитів. Для кожної команди перевірялися очікувана інтенція, фактично визначена інтенція, виділені параметри та результат виконання системної дії.

Результати функціонального тестування голосового асистента наведено в табл. 3.4.

Таблиця 3.4 – Результати функціонального тестування голосового асистента

№	**Тестова команда**	**Очікувана інтенція**	**Визначена інтенція**	**Результат виконання**
1	**2**	**3**	**4**	**5**
1	відкрий калькулятор	OpenApp	OpenApp	Калькулятор відкрито
2	відкрий блокнот	OpenApp	OpenApp	Блокнот відкрито
3	створи нотатку купити хліб	CreateNote	CreateNote	Створено локальну текстову нотатку
4	зроби гучніше	VolumeControl	VolumeControl	Рівень гучності збільшено
5	зменш гучність	VolumeControl	VolumeControl	Рівень гучності зменшено

Продовження таблиці 3.4

1	2	3	4	5
6	зменш яскравість	BrightnessControl	BrightnessControl	Рівень яскравості зменшено
7	відкрий налаштування звуку	OpenSettings	OpenSettings	Відкрито відповідний розділ налаштувань
8	відкрий диспетчер задач	OpenTaskManager	OpenTaskManager	Відкрито диспетчер задач
9	відкрий провідник	OpenExplorer	OpenExplorer	Відкрито провідник Windows
10	заблокуй комп’ютер	LockComputer	LockComputer	Виконано блокування комп’ютера
11	скажи мені погоду на завтра	Unknown	Unknown	Системна дія не виконувалася
12	запусти щось цікаве	Unknown	Unknown	Системна дія не виконувалася

Як видно з табл. 3.4, система коректно визначає основні інтенції, передбачені в програмній реалізації, та виконує відповідні системні дії. Зокрема, асистент правильно обробляє команди відкриття програм, створення нотаток, керування гучністю, зміни яскравості, відкриття налаштувань, запуску провідника, диспетчера задач і блокування комп’ютера. Для команд, що не належать до підтримуваних сценаріїв або

мають неоднозначне формулювання, система повертає стан Unknown. Це зменшує ризик випадкового виконання некоректної дії та підвищує безпечність роботи голосового асистента.

Окремо перевірялося виділення параметрів команд, оскільки для виконання системної дії недостатньо лише визначити інтенцію. Для OpenApp основним параметром є назва програми, для CreateNote – текст нотатки, для OpenSettings – розділ налаштувань, а для команд керування гучністю та яскравістю – напрям зміни або відсоткове значення. Перевірка виконувалася через діагностичний режим, у якому відображаються внутрішні поля розпізнаної команди. Це дозволило переконатися, що система не лише правильно визначає тип дії, а й передає до виконавчого шару необхідні дані для її коректного виконання.Результати перевірки виділення параметрів наведено в табл. 3.5.

Таблиця 3.5 – Перевірка виділення параметрів команд

№	**Команда**	**Інтенція**	**Виділений параметр**	**Результат**
1	відкрий калькулятор	OpenApp	AppName = калькулятор	Визначено коректно
2	відкрий блокнот	OpenApp	AppName = блокнот	Визначено коректно
3	створи нотатку купити хліб	CreateNote	NoteText = купити хліб	Визначено коректно
4	відкрий налаштування звуку	OpenSettings	SettingsSection = звук	Визначено коректно
5	зроби гучніше	VolumeControl	Direction = increase	Визначено коректно
6	зменш яскравість	BrightnessControl	Direction = decrease	Визначено коректно

Проведена перевірка показала, що параметри коректно виділяються для основних типів команд. Це підтверджує працездатність поєднання ML-модуля визначення інтенції з rule-based обробкою параметрів, оскільки намір користувача визначається за семантичною подібністю, а конкретні параметри – за допомогою правил, словників і шаблонів.

Додатково було перевірено допоміжні функції застосунку: активацію голосового введення через глобальну гарячу клавішу, згортання в системний трей, відновлення головного вікна, роботу діагностичного режиму та збереження налаштувань між запусками. Результати перевірки допоміжних можливостей наведено в табл. 3.6.

Таблиця 3.6 – Перевірка допоміжних можливостей застосунку

№	**Функція**	**Очікуваний результат**	**Результат перевірки**
1	Активація голосового введення кнопкою	Початок запису голосової команди	Виконано
2	Активація через глобальну гарячу клавішу	Початок запису без відкриття основного вікна	Виконано
3	Згортання в системний трей	Застосунок залишається активним у фоновому режимі	Виконано
4	Відновлення з трею	Головне вікно відкривається повторно	Виконано
5	Завершення роботи через трей	Застосунок коректно закривається	Виконано
6	Увімкнення режиму розробника	Відображаються діагностичні елементи	Виконано
7	Ручний розбір команди в режимі розробника	Відображається інтенція та параметри	Виконано

Результати тестування підтверджують, що реалізований застосунок виконує основні функції голосового асистента та забезпечує взаємодію з операційним середовищем Windows. Система здатна приймати команду, перетворювати її в текст, визначати інтенцію, виділяти необхідні параметри та виконувати відповідну системну дію. Діагностичний режим дає змогу додатково перевіряти внутрішню логіку інтерпретації команд, що є корисним під час налагодження й подальшого розширення функціональності.

3.7 Аналіз результатів тестування та перспективи вдосконалення

Проведене тестування показало, що розроблений голосовий асистент виконує основні функції, передбачені метою кваліфікаційної роботи. Застосунок забезпечує активацію голосового введення, запис аудіофрагмента, локальне перетворення мовлення в текст, нормалізацію отриманої команди, визначення інтенції, виділення параметрів та виконання відповідної системної дії в середовищі Windows. Результати перевірки підтвердили працездатність обраної архітектури та доцільність поділу системи на окремі функціональні модулі.

Найбільш стабільно система працює з командами, що мають чітку структуру та належать до заздалегідь визначених інтенцій. До таких команд належать відкриття програм, створення нотаток, керування гучністю, відкриття системних налаштувань, запуск диспетчера задач, відкриття провідника та блокування комп’ютера. Для таких сценаріїв асистент коректно визначає намір користувача, виділяє необхідні параметри та передає команду до відповідного виконавчого сервісу.

Окремо слід відзначити ефективність поєднання rule-based і ML-based підходів. Rule-based механізми доцільно використовуються для обробки очевидних команд і виділення параметрів, оскільки такі дії потребують контрольованості та передбачуваності. ML-шар, побудований на embedding-

подібності, забезпечує більшу гнучкість під час визначення інтенції, оскільки дозволяє обробляти команди, сформульовані не тільки в одному фіксованому вигляді. Завдяки цьому система може працювати з варіативними короткими командами, що є важливим для голосового інтерфейсу.

Результати тестування також показали важливість стану Unknown. Якщо команда не належить до підтримуваних інтенцій або не може бути надійно інтерпретована, система не повинна виконувати випадкову дію. Повернення стану Unknown дозволяє зменшити ризик помилкового виконання системної операції. Це особливо важливо для голосового асистента, оскільки під час розпізнавання мовлення можливі помилки, неповні фрази, шумові спотворення або випадкові висловлювання користувача.

Разом із цим у процесі тестування було виявлено певні обмеження реалізації. Якість роботи системи залежить від чіткості вимови користувача, якості мікрофона, рівня фонового шуму та коректності результату локального розпізнавання мовлення. Якщо модуль speech-to-text некоректно перетворює аудіо в текст, це може вплинути на подальше визначення інтенції. Таким чином, навіть за правильно реалізованого модуля інтерпретації загальна якість роботи асистента значною мірою залежить від якості початкового розпізнавання мовлення [24, 27].

Ще одним обмеженням є фіксований набір підтримуваних інтенцій. У поточній реалізації асистент виконує основні системні дії, необхідні для демонстрації працездатності запропонованого підходу, однак функціональність може бути розширена. Додавання нових інтенцій потребує поповнення набору прикладів команд, оновлення правил обробки параметрів та реалізації відповідних виконавчих сервісів. Завдяки модульній архітектурі таке розширення може здійснюватися без повної перебудови системи.

Окремого вдосконалення потребує механізм активації голосової взаємодії. У поточній версії основним реалізованим способом активації є кнопка інтерфейсу або глобальна гаряча клавіша. Режим wake word

передбачений у логіці розвитку застосунку, однак його повноцінна backend-реалізація може бути винесена в подальшу роботу. Запровадження стабільного механізму wake word дозволить зробити взаємодію з асистентом більш природною, оскільки користувач зможе активувати систему голосовою фразою без використання клавіатури або інтерфейсу.

Перспективним напрямом розвитку є також удосконалення діагностичних можливостей застосунку. Доцільним є виведення показника впевненості визначення інтенції, збереження історії команд, фіксація результатів розпізнавання та розширення інформації про найближчі знайдені приклади в ML-модулі. Це дозволить точніше аналізувати помилки, коригувати набір навчальних прикладів і підбирати оптимальні порогові значення для віднесення команди до певної інтенції або стану Unknown. У подальшому накопичена історія команд і результатів їх обробки може розглядатися як потік даних для виявлення типових груп помилок або сценаріїв використання, що узгоджується з підходами до adaptive clustering у задачах data stream mining [28–30].

Крім того, подальший розвиток системи може передбачати розширення підтримки природних формулювань команд. Для цього доцільно збільшити кількість прикладів у CSV-наборі, додати синонімічні конструкції, розширити словники назв програм і системних розділів, а також удосконалити правила виділення параметрів. Це дозволить підвищити стійкість асистента до різних способів формулювання однієї й тієї самої дії користувачем.

Результати тестування підтвердили, що розроблений застосунок виконує основні функції голосового асистента для керування комп’ютером. Система забезпечує повний цикл обробки команди – від отримання мовлення до виконання системної дії – та демонструє працездатність гібридного підходу. Виявлені обмеження визначають напрями подальшого вдосконалення програмного рішення.

ВИСНОВКИ

У кваліфікаційній роботі розв’язано задачу розроблення кросплатформеного голосового асистента для керування комп’ютером з використанням методів обробки природної мови. У процесі виконання роботи проаналізовано особливості побудови голосових інтерфейсів, специфіку обробки коротких голосових команд українською мовою, а також основні підходи до визначення інтенції користувача. За результатами аналізу обґрунтовано доцільність локальної обробки мовлення та використання embedding-based підходу для семантичної інтерпретації команд.

У межах роботи спроєктовано архітектуру програмної системи, що поєднує модулі голосового введення, локального розпізнавання мовлення, нормалізації тексту, визначення інтенції, виділення параметрів і виконання системних дій. Запропонована структура дозволяє розглядати голосового асистента як цілісний програмний конвеєр, у якому команда проходить шлях від аудіосигналу до виконання конкретної дії в операційному середовищі.

Для програмної реалізації використано платформу .NET, мову C#, фреймворк .NET MAUI, бібліотеку Whisper.net для локального розпізнавання мовлення та ONNX Runtime для запуску embedding-моделі. Семантична інтерпретація команд реалізована на основі моделі paraphrase-multilingual-MiniLM-L12-v2, яка формує векторні представлення тексту. Визначення інтенції виконується шляхом порівняння embedding-представлення нової команди з еталонними прикладами за косинусною подібністю.

Розроблено гібридну модель прийняття рішення, яка поєднує rule-based підхід і ML-шар. Rule-based механізми використовуються для обробки очевидних команд і виділення параметрів, а embedding-based підхід – для визначення інтенції у випадках варіативних формулювань. Таке поєднання забезпечує контрольованість обробки параметрів і водночас підвищує стійкість системи до різних способів формулювання одного й того самого наміру.

Практичним результатом роботи є програмний застосунок голосового асистента, що забезпечує локальне голосове керування типовими функціями комп’ютера. У застосунку реалізовано відкриття програм, створення локальних нотаток, керування гучністю та яскравістю, відкриття налаштувань Windows, запуск диспетчера задач, відкриття провідника та блокування комп’ютера. Загалом система підтримує 8 основних інтенцій користувача та окремий стан Unknown для невідомих або ненадійно розпізнаних команд.

Проведене тестування підтвердило працездатність розробленого рішення. Система коректно визначає основні інтенції, виділяє необхідні параметри команд і виконує відповідні системні дії. Наявність стану Unknown дозволяє уникати випадкового виконання некоректних або неоднозначних команд, що є важливою умовою безпечної роботи голосового асистента.

Розроблений застосунок має практичне значення, оскільки забезпечує локальну обробку голосових команд без використання зовнішньої серверної інфраструктури. Запропонована архітектура може бути використана як основа для подальшого розширення функціональності асистента, зокрема додавання нових інтенцій, удосконалення параметризації команд, розширення набору прикладів і впровадження повноцінного механізму активації за wake word.

Результати роботи апробовано у вигляді двох тез доповідей та однієї наукової статті. Зокрема, опубліковано тези, присвячені застосуванню ML.NET у семантичному аналізі мовленнєвих команд [7] та оцінюванню продуктивності локальної і серверної обробки природної мови у кросплатформенному голосовому асистенті [4]. Також результати, пов’язані з аналізом текстових даних у задачах NLP, відображено у науковій статті, присвяченій мультиметричному оцінюванню текстової надмірності, лексичного багатства та достовірності [19]. Таким чином, мету кваліфікаційної роботи досягнуто, а поставлені завдання виконано.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Murad, C., Candello, H., & Munteanu, C. (2023, July). What's the talk on VUI guidelines? A meta-analysis of guidelines for voice user interface design. In *Proceedings of the 5th international conference on conversational user interfaces*, pp. 1-16.

2. Творошенко, І. С. (2024). Особливості застосування методів оптимізації бізнес-процесів, реалізованих засобами машинного навчання та нейронних мереж, In *The 11th International scientific and practical conference "Modern generation: current problems, experience, development prospects"(November 12–15, 2024) Seville, Spain. International Science Group. 2024. 415 p.* (p. 369).

3. Kim C., Gowda D., Lee D. [та ін.]. A Review of On-Device Fully Neural End-to-End Automatic Speech Recognition Algorithms. *arXiv*. 2020. URL: https://arxiv.org/abs/2012.07974 (дата звернення 04.05.2026).

4. Ковальчук В.О., Кобилін І.О. (2026). Оцінювання продуктивності локальної та серверної обробки природної мови у кросплатформенному голосовому асистенті, У V студентська науково-практична конференція «прикладні комп'ютерні науки» (с. 217–219).

5. Wang, X., Jiang, W., & Luo, Z. (2016). Combination of convolutional and recurrent neural network for sentiment analysis of short texts.*Proceedings of COLING 2016, the 26th international conference on computational linguistics: Technical papers*, pp. 2428-2437.

6. Hamotskyi, S., Levbarg, A. I., & Hänig, C. (2024, May). Eval-UA-tion 1.0: benchmark for evaluating Ukrainian (large) language models. In *Proceedings of the Third Ukrainian Natural Language Processing Workshop (UNLP)@ LREC-COLING 2024*, pp. 109-119.

7. Ковальчук, В.О., Кобилін, І.О. (2025). Застосування ML.NET у семантичному аналізі мовленнєвих команд для голосових асистентів, *It-простір сьогодення: тенденції, інновації та перспективи,* С. 217–219.

8. Weld, H., Huang, X., Long, S., Poon, J., & Han, S. C. (2022). A survey of joint intent detection and slot filling models in natural language understanding. *ACM Computing Surveys*, 55(8), 1-38.

9. Pomazan V., Tvoroshenko I., and Gorokhovatskyi V. (2023) Handwritten character recognition models based on convolutional neural networks, *International Journal of Academic Engineering Research*, 7(9), pp. 64-72.

10. Kharchenko, A. I., & Kobylin, I. O. (2024). Performance analysis of support vector machine for vehicle classification, *Ministry of Education and Science of Ukraine vn Karazin Kharkiv National University*, *101*.

11. Кобилін, І. О., & Харченко, А. І. (2024). Classification techniques for computer vision, *Системи обробки інформації*, (3 (178)), 33-41.

12. Reimers, N., & Gurevych, I. (2019, November). Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, pp. 3982-3992.

13. Wang, Y., Wu, A., & Neubig, G. (2022, December). English contrastive learning can learn universal cross-lingual sentence embeddings. *Proceedings of the 2022 conference on empirical methods in natural language processing*, pp. 9122-9133.

14. Madabushi, H. T., Gow-Smith, E., Garcia, M., Scarton, C., Idiart, M., & Villavicencio, A. (2022). SemEval-2022 task 2: Multilingual idiomaticity detection and sentence embedding. *Proceedings of the 16th International Workshop on Semantic Evaluation (SemEval-2022)*, pp. 107-121.

15. López-Espejo, I., Tan, Z. H., Hansen, J. H., & Jensen, J. (2021). Deep spoken keyword spotting: An overview. *IEEE Access*, *10*, pp. 4169-4199.

16. Ніколайчук, А., Кобилін, О., Кобилін, І., & Путятіна, О. (2026). A comprehensive evaluation of transformer models for sentence-level semantic similarity in english and ukrainian. *Автоматизовані системи управління та прилади автоматики*, (189), 284-296.

17. Gorokhovatskyi V., Tvoroshenko I. (2023) Identification of visual objects by the search request. *International scientific symposium «INTELLIGENT SOLUTIONS-S», Computational intelligence (results, problems and perspectives). Decision making theory: proceedings of the international symposium, September 28, 2023, Kyiv-Uzhorod, Ukraine*, pp. 25-27.

18. Daradkeh, Y. I., & Tvoroshenko, I. (2020). Technologies for making reliable decisions on a variety of effective factors using fuzzy logic, *International Journal of Advanced Computer Science and Applications*, *11*(5).

19. Kobylin, I. O., Biehunova, V. D., Tsyban, D. S., & Kovalchuk, V. O. (2026). Measuring Textual Redundancy, Lexical Richness, and Veracity: a Multi-Metric Approach to Text Evaluation. *Information Technologies and Systems*, *8*(2), 25-45.

20. Microsoft. (n.d.). *.NET Multi-platform App UI documentation*. Microsoft Learn. Retrieved May 13, 2026, from https://learn.microsoft.com/en-us/dotnet/maui/

21. Hanea, S. (n.d.). *Whisper.net* [Computer software]. GitHub. Retrieved May 13, 2026, from https://github.com/sandrohanea/whisper.net

22. Microsoft. (n.d.). *Welcome to ONNX Runtime*. ONNX Runtime. Retrieved May 13, 2026, from https://onnxruntime.ai/docs/

23. Bain, M., Huh, J., Han, T., & Zisserman, A. Whisperx: Time-accurate speech transcription of long-form audio. *arXiv*. 2023. URL: https://arxiv.org/abs/2303.00747 (дата звернення 13.05.2026).

24. Radford, A., Kim, J. W., Xu, T., Brockman, G., McLeavey, C., & Sutskever, I. (2023). Robust speech recognition via large-scale weak supervision. *International Conference on Machine Learning*, pp. 28492-28518.

25. Боденчук-Пастухов, Є. В., Кобилін, І. О. (2026). Оцінка моделей трансформерів машинного перекладу на низько-ресурсних, азійсько-українських мовних парах, *Системи обробки інформації,* (1 (184)), 116-123.

26. Кобилін, І. О., & Ніколайчук, А. І. (2024). Monitoring and diagnosing faults in online mode using time series data, *Системи обробки інформації*, (3 (178)), 27-32.

27. Gandhi, S., Von Platen, P., & Rush, A. M. (2023). Distil-whisper: Robust knowledge distillation via large-scale pseudo labelling. *arXiv.* 2023. URL: https://arxiv.org/abs/2311.00430 (дата звернення 14.05.2026).

28. Бодянський, Є., Винокурова, О., Кобилін, І., & Мулеса, П. (2017). Адаптивна матрична нейро-фаззі самоорганізовна мережа для кластеризації багатовимірних потоків даних, *Вісник Національного університету Львівська політехніка. Комп'ютерні науки та інформаційні технології*, (864), С.314-319.

29. Bodyanskiy Y., Vynokurova O., Kobylin, I., & Kobylin O. (2016) Adaptive fuzzy clustering of short time series with unevenly distributed observations in Data Stream Mining tasks, *Information Technology and Management Science*. pp. 23-28

30. Бодянський, Є. В., Винокурова, О. А., Ізонін, І. В., Кобилін, І. О., Мулеса, П. П. (2017). Кластеризація багатовимірних часових рядів на основі адаптивної матричної нейро-фаззі самоорганізовної мережі, *Intellectual Systems For Decision Making and Problems of Computational Intelligence*, С.247.