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

КВАЛІФІКАЦІЙНА РОБОТА

на тему

Система моніторингу та керування IoT-мережею «Розумний дім»

Виконав: студент групи 1К-22
Спеціальності 123 Комп’ютерна інженерія
Дмитро ПЕРЕХРЕСТ
Керівник:
Павло РАТАЙЧУК

Черкаси – 2026

Циклова комісія комп’ютерної інженерії та інформаційних технологій

Спеціальність 123 «Комп’ютерна інженерія»

Освітня програма Комп’ютерна інженерія

ЗАТВЕРДЖУЮ

Завідувачка циклової комісії КІ та ІТ

_____Наталя ФАЛЬЧЕНКО

(підпис)

«_____» _____2025 р.

З А В Д А Н Н Я

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Перехресту Дмитру Олександровичу

1. Тема роботи: Інтернету речей інтеграції пристроїв для системи "Розумний будинок". Керівник роботи Павло Ратайчук Єгорович викладач вищої категорії, затверджені наказом закладу перевищої освіти від «6» жовтня 2025 року № 84/1-у .
2. Строк подання студентом кваліфікаційної роботи 08.06.2026 .
3. Вихідні дані до роботи: науково-технічна література з тематики IoT та штучного інтелекту; документація мікроконтролера ESP8266, релейних модулів та вимикачів; специфікації протоколу MQTT; параметри нейромереж Faster-Whisper, Qwen 3.5, Piper TTS та рушія CTranslate2; характеристики сервера на ОС Ubuntu Linux з прискоренням NVIDIA CUDA.
4. Зміст кваліфікаційної роботи: теоретичний аналіз архітектурних моделей IoT-систем та принципів локального функціонування ШІ-моделей; проєктування трирівневої клієнт-серверної архітектури, ієрархії MQTT-топіків та алгоритму синхронізації станів; техніко-економічне обґрунтування комплексу; практична реалізація серверної частини в ОС Ubuntu Linux з прискоренням CUDA та мікропрограмного забезпечення мікроконтролера ESP8266; експериментальне дослідження швидкодії обчислювального конвеєра та відмовостійкості системи.
5. Дата видачі завдання 15.09.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Терміни виконання етапів	Примітка про виконання
1	Вступ	14.10.2025	
2	Розділ 1 Основи та аналіз існуючих рішень у сфері IoT для домашньої автоматизації	09.12.2025	
3	Розділ 2 Проєктування та програмно-технічне забезпечення функціонування автономної системи	10.03.2026	
4	Розділ 3 Експериментальні дослідження, оцінювання надійності та функціональних характеристик системи	28.04.2026	
5	Висновки	12.05.2026	
6	Оформлення кваліфікаційної роботи (чистовий варіант)	20.05.2026	
7	Перевірка кваліфікаційної роботи на наявність ознак плагіату (за 10 днів до захисту)	05.06.2026	
8	Подання кваліфікаційної роботи на затвердження завідувачу ЦК (за 7 днів до захисту)	08.06.2026	

Студент _____ Перехрест ДМИТРО
(підпис)

Керівник роботи _____ Павло РАТАЙЧУК
(підпис)

АНОТАЦІЯ

Кваліфікаційну роботу присвячено розробці та дослідженню розподіленого апаратно-програмного комплексу локального голосового керування й асинхронної синхронізації станів у системах «Розумного дому». Застосування парадигми периферійних обчислень (*Edge Computing*) забезпечує повну автономність і конфіденційність даних та усуває інженерну проблему десинхронізації станів під час інтеграції традиційних механічних вимикачів у цифрову бездротову мережу.

Спроєктовано трирівневу архітектуру, де серверний конвеєр обробки мовлення функціонує в ОС Ubuntu Linux із прискоренням NVIDIA CUDA на базі моделей Faster-Whisper, Qwen 3.5 та Piper з оптимізацією рушієм CTranslate2. Для миттєвого перехоплення команд було впроваджено алгоритм гібридної маршрутизації. Мережеву взаємодію реалізовано через локальний MQTT-брокер Eclipse Mosquitto, а периферійний вузол побудовано на мікроконтролері ESP8266 з асинхронною неблокуючою прошивкою мовою C++ та вбудованим алгоритмом програмного дебаунсингу (*Software Debouncing*).

Експериментальне профілювання підтвердило високу швидкодію комплексу: час реакції на директивні команди становить 355 мс, затримка телеметрії в Wi-Fi-мережі не перевищує 12 мс, а обробка складних контекстних запитів через LLM триває до 1,7 с. Практично верифіковано повну автономність і відмовостійкість клієнтського вузла за умови аварійної втрати зв'язку з центральним сервером.

Ключові слова — ІНТЕРНЕТ РЕЧЕЙ (IoT), РОЗУМНИЙ ДІМ, ШТУЧНИЙ ІНТЕЛЕКТ, ВЕЛИКІ МОВНІ МОДЕЛІ (LLM), ПРОТОКОЛ MQTT, МІКРОКОНТРОЛЕР ESP8266, ПРОГРАМНИЙ ДЕБАУНСИНГ, СИНХРОНІЗАЦІЯ СТАНІВ, ПЕРИФЕРІЙНІ ОБЧИСЛЕННЯ (EDGE COMPUTING).

ABSTRACT

The qualification thesis is dedicated to the development and investigation of a distributed hardware-software complex for local voice control and asynchronous state synchronization in "Smart Home" systems. The application of the Edge Computing paradigm ensures complete autonomy and data privacy, while eliminating the engineering problem of state desynchronization when integrating traditional mechanical switches into a digital wireless network.

A three-tier architecture has been designed, in which the server-side speech processing pipeline runs on Ubuntu Linux with NVIDIA CUDA acceleration, using Faster-Whisper, Qwen 3.5, and Piper models optimized by the CTranslate2 engine. A hybrid routing algorithm has been implemented for the instantaneous interception of commands. Network interaction is implemented via a local Eclipse Mosquitto MQTT broker, while the peripheral node is built on the ESP8266 microcontroller with asynchronous, non-blocking firmware in C++ and a built-in software debouncing algorithm.

Experimental profiling confirmed the high performance of the complex, with a reaction time to directive commands of 355 ms, Wi-Fi network telemetry latency not exceeding 12 ms, and processing of complex contextual requests via an LLM taking up to 1.7 s. The client node's complete autonomy and fault tolerance during an emergency loss of communication with the central server have been demonstrated in practice.

Keywords — INTERNET OF THINGS (IoT), SMART HOME, ARTIFICIAL INTELLIGENCE, LARGE LANGUAGE MODELS (LLM), MQTT PROTOCOL, ESP8266 MICROCONTROLLER, SOFTWARE DEBOUNCING, STATE SYNCHRONIZATION, EDGE COMPUTING

ЗМІСТ

ВСТУП

Актуальність теми. Розвиток Інтернету речей (IoT) інтегрував системи «Розумний дім» у побут завдяки ергономічності голосового керування [12, с. 415]. Проте хмарні сервіси (STT/TTS) вразливі щодо конфіденційності й залежать від інтернету. Локальне розгортання ШІ-моделей на домашньому сервері з GPU на 12 ГБ забезпечує автономну обробку мови (NLP) у LAN-мережі, гарантуючи високу швидкодію та безпеку даних [1; 3]. Іншою проблемою є десинхронізація IoT-вузлів у базі даних через конфлікт станів між цифровими реле та механічними вимикачами. Впровадження алгоритму двосторонньої асинхронної синхронізації об’єднує фізичне та голосове керування в безконфліктну розподілену екосистему [6; 9], що робить розробку програмної частини такої автономної системи актуальним завданням комп’ютерної інженерії.

Мета і завдання роботи. Метою є розробка програмного комплексу локальної системи моніторингу та керування IoT-мережею «Розумний дім» на основі голосового ШІ-асистента з безконфліктною синхронізацією станів пристроїв і вимикачів. Для цього передбачено проаналізувати концепції IoT та локальні ШІ-технології, дослідити алгоритми усунення десинхронізації станів, спроєктувати архітектуру асистента з квантуванням NLP-моделей під ліміти сервера, розробити подієво-орієнтовану логіку клієнтського вузла на базі мікроконтролера з неблокуючим дебаунсингом, створити алгоритм двосторонньої асинхронної синхронізації та експериментально оцінити швидкодію й відмовостійкість прототипу.

Об'єкт, предмет та методи дослідження. Об'єктом дослідження є процес програмного моніторингу та автоматизованого розподіленого керування пристроями освітлення в локальних мережах «Розумного дому», а предметом – програмні засоби, ШІ-алгоритми обробки голосу, мікропрограми контролерів та алгоритми асинхронної синхронізації пристроїв в IoT-мережах. Методологічну основу склали методи системного аналізу,

логічного моделювання та декомпозиції, об’єктно- та подієво-орієнтоване програмування (Python, C++), концепції штучних нейромереж [11] і моделі-трансформатори з CUDA-оптимізацією під GPU, а також експериментальний метод профілювання для вимірювання затримок MQTT.

Практичне значення та техніко-економічна доцільність. Практичне значення полягає у створенні автономного комплексу автоматизації освітлення на базі Ubuntu-сервера, де дані обробляються виключно в LAN-мережі [10], а алгоритми неблокуючої синхронізації запобігають розсинхронізації фізичного та віртуального статусу обладнання. Економічна доцільність базується на використанні доступних мікроконтролерів ESP8266, де оптимізований мікрокод знижує апаратні вимоги, роблячи рішення дешевшим за аналоги, а ієрархічна структура MQTT-топіків забезпечує масштабованість системи без модифікації її ядра [7].

Структура роботи. Кваліфікаційна робота складається зі вступу, трьох розділів, загальних висновків, списку джерел і додатків. Перший розділ висвітлює теоретичні засади IoT і локальних моделей NLP. Другий розділ присвячено проєктуванню системи, оптимізації VRAM та алгоритму синхронізації станів. У третьому розділі описано практичну реалізацію (CUDA, MQTT, C++ прошивка) та результати тестування швидкодії. У висновках підсумовано результати розробки та надано рекомендації щодо вдосконалення системи.

РОЗДІЛ 1 ТЕОРЕТИЧНІ ОСНОВИ ПОБУДОВИ IOT-СИСТЕМ «РОЗУМНОГО ДОМУ»

Сучасний етап розвитку інформаційних технологій характеризується масштабною інтеграцією обчислювальних засобів у повсякденне фізичне середовище людини. Ключовим драйвером цього процесу є концепція Інтернету речей (*Internet of Things, IoT*), яка визначається як глобальна мережева інфраструктура, що об'єднує фізичні об'єкти («речі»), оснащені вбудованими засобами для взаємодії між собою, з навколишнім середовищем та з централізованими або розподіленими системами управління [12, с. 380].

Еволюція IoT зумовлена значним здешевленням мікропроцесорної техніки, розширенням доступності бездротових технологій зв'язку та появою високоефективних алгоритмів штучного інтелекту, адаптованих для локального виконання. Головна парадигма Інтернету речей полягає в переході від концепції ізольованих автоматизованих пристроїв до створення інтегрованих інтелектуальних екосистем, які здатні здійснювати безперервний моніторинг, предиктивний аналіз та автономне прийняття рішень з мінімальним залученням оператора-людини.

Однією з найбільш репрезентативних, технічно складних та практично значущих сфер застосування технологій Інтернету речей є концепція **«Розумного дому» (Smart Home)**. У сучасному інженерному розумінні «Розумний дім» — це автоматизована система управління житловим простором, що забезпечує синергетичну взаємодію підсистем освітлення, клімат-контролю, безпеки, мультимедіа та побутових приладів [12, с. 412].

Оцінювання ефективності впровадження подібних комплексів автоматизації базується на сукупності ключових техніко-експлуатаційних критеріїв. Першочергове значення серед них має суттєве підвищення рівня ергономіки та загального комфорту проживання кінцевих користувачів у житловому просторі. Разом з тим, інтеграція інтелектуальних рішень дозволяє мінімізувати вплив антропогенного фактора під час управління щоденними рутинними процесами, що значно знижує навантаження на

людину. Важливим екологічним та економічним аспектом функціонування системи є оптимізація витрат базових енергетичних ресурсів, яка досягається завдяки гнучкій програмній реалізації інтелектуальних сценаріїв адаптивного енергоспоживання. Не менш критичним критерієм загальної ефективності комплексу є забезпечення надійного проактивного захисту житла від виникнення раптових аварійних ситуацій техногенного характеру, а також оперативне запобігання будь-яким спробам несанкціонованого доступу ззовні.

З позицій системного аналізу та комп'ютерної інженерії архітектурна модель сучасної IoT-системи є складною багаторівневою структурою. Для забезпечення модульності, масштабованості та взаємозамінності окремих компонентів проєктування такої системи традиційно здійснюється на основі класичної чотирирівневої архітектурної моделі.

1.1. Аналіз багаторівневої архітектурної моделі систем Інтернету речей

Сучасні екосистеми Інтернету речей (IoT), орієнтовані на автоматизацію житлових просторів та інтеграцію елементів штучного інтелекту, будуються на принципах децентралізації та декомпозиції обчислювальних процесів. Для забезпечення стабільної взаємодії між різнорідними апаратними компонентами та хмарними чи локальними серверами у світовій практиці застосовується концептуальна багаторівнева модель архітектури. Вона дозволяє чітко розмежувати функції збору первинної телеметрії, асинхронного транспортування пакетів даних, семантичного аналізу інформації та кінцевої взаємодії з користувачем через інтерфейси керування. Детальний аналіз структури, функціонального призначення та специфіки реалізації кожного з чотирьох базових рівнів розроблюваного комплексу наведено нижче.

1.1.1. Сенсорний рівень (Perception Layer)

Зазначений архітектурний рівень виступає безпосереднім апаратним фундаментом системи, який забезпечує пряму взаємодію обчислювального комплексу з фізичним оточенням. Функціональний склад цього шару традиційно декомпонується на дві ключові категорії апаратно-програмних компонентів. Першу групу становлять засоби моніторингу, до яких належать різноманітні сенсори та датчики, що здійснюють первинне перетворення аналогових фізичних величин навколишнього середовища (зокрема, показників температури, вологості, рівня освітленості, наявності динамічного руху або концентрації газів) у структуровані цифрові сигнали для подальшого аналізу.

Другу функціональну групу утворюють виконавчі механізми, або актуатори, які безпосередньо реалізують зворотні керуючі впливи, що надходять від вищих логічних рівнів загальної архітектури. Вони трансформують отримані цифрові команди у конкретні фізичні дії, виконуючи комутацію силових ланцюгів живлення за допомогою релейних модулів, точну зміну просторового положення сервоприводів або широтно-імпульсне модулювання яскравості світіння кінцевих світлодіодних елементів.

У сучасних розподілених системах керування освітленням на цьому рівні домінують малогабаритні енергоефективні системи на кристалі (*System-on-Chip, SoC*) та мікроконтролери з вбудованими радіомодулями бездротового зв'язку, зокрема сімейства **ESP8266**. Вони виступають базовою апаратною платформою для розгортання спеціалізованого мікропрограмного забезпечення (прошивок), яке відповідає за низькорівневу логіку: опитування датчиків, програмне усунення апаратного брязкоту контактів, формування керуючих сигналів для реле та первинне пакування даних у пакети для передачі в мережу [9, с. 12].

1.1.2. Мережевий рівень (Network Layer)

Цей рівень виконує функцію комунікаційної матриці, забезпечуючи надійне, швидке та захищене транспортування інформаційних пакетів між кінцевими вузлами сенсорного рівня та обчислювальними потужностями рівня управління. У локальних екосистемах «Розумного дому» архітектура мережевого рівня будується на основі комбінації бездротових технологій передавання даних. Найбільш поширеним є стандарт **IEEE 802.11 (Wi-Fi)**, що дозволяє інтегрувати IoT-пристрої в наявну домашню мережеву інфраструктуру без необхідності розгортання додаткових шлюзів.

На вищих рівнях мережевого стеку (прикладному та транспортному) критично важливим є вибір оптимального протоколу передавання даних. Стандартом де-факто в індустрії IoT є легковаговий протокол **MQTT (Message Queuing Telemetry Transport)**, який функціонує за шаблоном проєктування «видавець-підписник» (*publish-subscribe*). MQTT характеризується мінімальним обсягом службового заголовка пакета, низькими вимогами до обчислювальних ресурсів кінцевих мікроконтролерів та наявністю механізмів гарантії доставки повідомлень (*Quality of Service, QoS*), що робить його ідеальним рішенням для асинхронного обміну даними та моніторингу станів у режимі реального часу [6; 7].

1.1.3. Рівень обробки даних та управління (Processing/Middleware Layer)

Даний рівень виконує роль координаційного та аналітичного ядра всієї системи. Саме тут здійснюється агрегація телеметричних даних, ведення журналів подій, автентифікація пристроїв і виконання високорівневої логіки автоматизації.

Історично більшість комерційних рішень базувалася на концепції хмарних обчислень (*Cloud Computing*), за якої всі обчислювальні процеси виносилися на віддалені сервери. Проте хмарна архітектура має низку критичних недоліків для систем життєзабезпечення: залежність від зовнішнього каналу зв'язку, підвищені часові затримки (*latency*) та ризики

порушення конфіденційності під час передачі приватних даних на сторонні сервери.

Сучасною альтернативою є парадигма **периферійних обчислень (Edge Computing)**, яка передбачає розгортання виділеного локального сервера безпосередньо в межах домашньої мережі на базі ПК під керуванням ОС Ubuntu Server [10]. Локальний сервер дозволяє забезпечити абсолютну автономність системи, мінімальний час відгуку мережі, а також надає необхідні обчислювальні ресурси (включаючи потужності GPU) для функціонування складних алгоритмів штучного інтелекту без звернення до зовнішніх веб-ресурсів.

1.1.4. Прикладний рівень (Application Layer)

Цей рівень представляє безпосередній інтерфейс взаємодії між інтелектуальною системою та кінцевим користувачем. До нього належать графічні веб-панелі, мобільні додатки, а також системи природного мовного інтерфейсу (*Natural Language User Interface*).

Голосове керування, реалізоване на основі локальних стеків штучного інтелекту (**STT / LLM / TTS**), є найбільш ергономічним і перспективним методом взаємодії в парадигмі комп'ютерної інженерії [11]. Воно дозволяє перетворити простір «Розумного дому» на безбар'єрне середовище, де керування складними інженерними сценаріями здійснюється за допомогою природних мовних конструкцій, які інтерпретуються локальними нейромережами [1; 3; 5].

1.1.5. Особливості підсистеми керування освітленням та проблема десинхронізації станів

Особливе місце в архітектурі «Розумного дому» посідає підсистема інтелектуального керування освітленням. Вона є найбільш динамічною за частотою використання та критичною з точки зору надійності. Проте під час проєктування програмно-керованого освітлення виникає серйозна проблема: більшість існуючих рішень не враховує традиційні патерни поведінки

людини, пов'язані з використанням класичних настінних механічних вимикачів. Фізичне розмикання ланцюга живлення стандартним вимикачем призводить до повного знеструмлення мікроконтролера на сенсорному рівні, що робить його недоступним для мережі та голосового асистента.

Усунення цього архітектурного недоліку вимагає зміни підходу до розробки програмного забезпечення системи. Необхідно проєктувати логіку клієнтських та серверних програмних модулів таким чином, щоб забезпечити безперервний моніторинг фізичних органів керування як цифрових сенсорів, так і силових перемикачів. Це дозволяє реалізувати алгоритми двосторонньої асинхронної синхронізації станів у режимі реального часу, незалежно від того, яким чином було ініційовано команду — через голосовий інтерфейс ШІ-асистента чи шляхом ручного натискання вимикача на стіні.

Таким чином, створення сучасної, відмовостійкої та безпечної IoT-системи «Розумний дім» вимагає комплексного підходу до проєктування програмної архітектури на всіх її рівнях. Основний інженерний фокус зміщується з чисто апаратних засобів у площину створення оптимізованого мікрокоду для кінцевих вузлів, побудови гнучких алгоритмів мережевої взаємодії за допомогою протоколу MQTT та розгортання автономного периферійного сервера, здатного виконувати ресурсомісткі нейромережеві моделі обробки мови локально в межах обчислювальної інфраструктури об'єкта.

1.2. Принципи роботи голосових асистентів та технології штучного інтелекту

Голосові інтерфейси користувача (*Voice User Interfaces, VUI*) стали одним із найважливіших етапів еволюції систем взаємодії людини з комп'ютером. У контексті систем «Розумний дім» голосовий асистент виконує роль центрального інтелектуального хаба, який перетворює акустичні сигнали користувача на машинні команди. Архітектура сучасного голосового асистента є складним обчислювальним конвеєром (*pipeline*), що

складається з трьох ключових етапів: розпізнавання мовлення, семантичного аналізу тексту та генерації голосової відповіді [11, с. 52].

1.2.1. Технології розпізнавання (Speech-to-Text) та синтезу мовлення (Text-to-Speech)

Першим етапом обробки голосової команди є перетворення звукової хвилі у текстовий формат за допомогою технології *Speech-to-Text (STT)* або *Automatic Speech Recognition (ASR)*. Традиційні системи ASR базувалися на прихованих марковських моделях (HMM), проте сучасний етап розвитку штучного інтелекту повністю перейшов на використання глибоких нейронних мереж, зокрема архітектури «Трансформерів» (*Transformers*) [1, с. 2].

Процес STT починається з попередньої обробки звуку: аудіосигнал очищується від фонового шуму та розбивається на короткі фрейми тривалістю 20–30 мілісекунд. Далі для кожного фрейму обчислюється мелкочастотна кепстральна репрезентація (MFCC) або спектрограма. Ці акустичні ознаки подаються на вхід нейромережі (зокрема, архітектури *Whisper, оптимізованої для мовного розуміння*), яка зіставляє акустичні патерни з фонемами, а потім — зі словами, використовуючи статистичну мовну модель для передбачення найбільш імовірного слова в заданому контексті [1, с. 4].

Після виконання команди системі необхідно надати користувачеві інтерактивний зворотний зв’язок. Для цього використовується зворотна технологія — *Text-to-Speech (TTS)*. Сучасні локальні системи TTS працюють у два етапи: спочатку текстовий рядок перетворюється на акустичні параметри (мел-спектрограму) за допомогою акустичної моделі, а потім нейромережевий вокодер (*Vocoder*, наприклад, у системі *Piper*) генерує з цієї спектрограми кінцеву звукову хвилю. Такий підхід дозволяє отримати

максимально природне мовлення з правильними інтонаціями, що майже не відрізняється від людського голосу [5].

1.2.2. Обробка природної мови (NLP) для аналізу команд

Отримання «сирого» тексту від системи STT є недостатнім для прямого керування периферійними пристроями; системі необхідно інтерпретувати сенс сказаного. За це відповідає підсистема обробки природної мови (*Natural Language Processing, NLP*), а саме її функціональний розділ — розуміння природної мови (*Natural Language Understanding, NLU*).

Головна задача NLU у локальних IoT-системах зводиться до двох паралельних обчислювальних процесів: класифікації намірів (*Intent Classification*) та видобування сутностей (*Entity Recognition*) [3, с. 12]. Наприклад, під час фіксації фрази: «Увімкни головне світло у вітальні», NLP-модуль здійснює синтаксичний і семантичний аналіз неструктурованих даних для формування структурованого об’єкта. Алгоритм визначає намір як turn_on_device, а сутності класифікує за категоріями device: «головне світло» та location: «вітальня».

Для реалізації цього процесу локально застосовуються оптимізовані великі мовні моделі (LLM), які здатні коректно обробляти контекст, синоніми та граматично складні або неповні речення, що забезпечує високу гнучкість взаємодії порівняно з жорстко запрограмованими регулярними виразами.

1.2.3. Апаратні вимоги для розгортання локальних ШІ-моделей (використання GPU)

Головним викликом під час розробки повністю автономного голосового асистента є високі вимоги до апаратного забезпечення. Локальна обробка вимагає розгортання та інференсу нейромереж безпосередньо на домашньому сервері.

Виконання інференсу виключно на центральному процесорі (CPU) є неефективним, оскільки архітектура CPU оптимізована для послідовного

виконання інструкцій, тоді як робота нейромережевих стеків, що охоплюють модулі розпізнавання, синтезу мовлення та великі мовні моделі, вимагає одночасного виконання мільйонів операцій матричного множення. Саме тому критично необхідним є використання графічних процесорів (GPU), архітектура масових паралельних обчислень яких, зокрема, технологія CUDA-ядер, ідеально адаптована до таких математичних задач [2]. У цьому контексті визначальну роль відіграють обсяг та тип відеопам'яті (VRAM). Для стабільного функціонування повноцінного локального голосового асистента необхідно одночасно утримувати в активному стані моделі розпізнавання голосу, мовне ядро для аналізу намірів користувача та вокодер. Оптимальним інженерним рішенням у такому випадку є застосування графічних прискорювачів з обсягом виділеної пам'яті 12 ГБ, що дозволяє повністю завантажити всі вагові коефіцієнти моделей у надшвидку пам'ять відеокарти. Натомість вимушене вивантаження частин нейромереж в оперативну пам'ять комп'ютера (RAM) через дефіцит VRAM створює критичне вузьке місце системи (*bottleneck*), збільшуючи загальний час відгуку з прийнятних мілісекунд до десятків секунд.

Для вирішення апаратних обмежень та оптимізації простору відеопам'яті широко застосовуються сучасні методи квантування моделей. Великі мовні моделі з відкритим вихідним кодом у своєму базовому форматі обчислень із рухомою комою високої розрядності (FP16) вимагають від 14 до 16 ГБ VRAM лише для первинного завантаження параметрів. Зниження розрядності вагових коефіцієнтів з 16 бітів до 8 або 4 бітів, наприклад, шляхом конвертації у двійковий формат GGUF, дозволяє успішно вмістити обчислювальний стек у доступний ліміт 12 ГБ VRAM за умови збереження високої лінгвістичної точності та розуміння контексту [4]. При цьому швидкість генерації відповіді, яка вимірюється кількістю токенів за секунду, безпосередньо залежить від пропускної здатності пам'яті графічного адаптера. Використання пам'яті стандарту GDDR6 забезпечує швидкість обміну даними на рівні 360 ГБ/с, що в десятки разів перевищує можливості

шини оперативної пам’яті, мінімізуючи часові затримки реакції. Окрім суто обчислювальних параметрів, під час проєктування домашнього сервера, що функціонує в цілодобовому режимі 24/7, враховуються експлуатаційні обмеження, пов’язані з енергоспоживанням та тепловиділенням (TDP) обладнання. Графічні прискорювачі середнього сегмента з 12 ГБ VRAM демонструють оптимальний баланс продуктивності та енерговитрат, що дає змогу організувати ефективне і тихе охолодження серверного вузла в побутових умовах [12, с. 422]. Таким чином, наявність виділеного високопродуктивного GPU є базовою технічною умовою для розгортання локального ШІ-асистента, здатного успішно конкурувати за швидкістю реакції з комерційними хмарними сервісами.

1.3.1. Огляд програмних платформ для мікроконтролерів (сімейства ESP8266/ESP32)

Історично першим популярним середовищем для розробки мікропрограмного забезпечення став стек інструментів *Arduino IDE*. Проте використання класичних програмних бібліотек Arduino в сучасних розподілених IoT-системах накладає суттєві обмеження через відсутність вбудованої підтримки мережевих стеків на рівні ядра. Для реалізації мережевої взаємодії розробнику доводиться інтегрувати додаткові програмні рівні драйверів, що значно збільшує обсяг скомпільованого бінарного файлу, перевантажує обмежену динамічну пам’ять мікроконтролера та ускладнює загальну архітектуру програмного коду [12, с. 415].

Ефективним рішенням у сфері розробки програмного забезпечення для IoT стала поява програмно-апаратних платформ на базі архітектури **ESP8266.** Ця 32-бітна обчислювальна система з процесором *Tensilica L106* надає розробнику розширені програмні можливості: інтегровані на рівні фірмового SDK повноцінні стеки протоколів TCP/IP та низькорівневі бібліотеки для безпосередньої роботи з Wi-Fi-мережею за стандартом IEEE 802.11 b/g/n [9, с. 5].

Завдяки оптимізації компілятора та підтримці середовища *Arduino Core*, ESP8266 виступає надійною апаратною платформою для розгортання легких клієнтських застосунків. Обчислювальних ресурсів чипа цілком достатньо для паралельного виконання фонового циклу підключення до мережі, підтримки стабільного асинхронного зв'язку з MQTT-брокером через оптимізовані клієнтські бібліотеки, обробки вхідних текстових топіків та виконання алгоритмів програмного придушення брязкоту контактів (*software debouncing*) на вхідних GPIO-портах [8; 9].

Подальшим розвитком програмних рішень стала платформа ESP32, яка пропонує підтримку операційної системи реального часу *FreeRTOS* та багатопотоковості на двоядерній архітектурі. Проте для розв'язання базової задачі логічного керування реле освітлення використання FreeRTOS та багатопотокового коду є надлишковим. З точки зріння програмної оптимізації та раціонального розподілу пам'яті, однопотокова асинхронна архітектура прошивки на базі ESP8266 залишається найбільш ефективним вибором, що забезпечує мінімальний розмір мікрокоду та високу швидкість обробки переривань без зайвих накладних витрат обчислювальної потужності процесора [9, с. 21].

1.3.2. Програмне керування виконавчими механізмами (реле)

Оскільки комп'ютерна інженерія оперує програмними абстракціями низьковольтних логічних рівнів, задача взаємодії з силовими мережами зводиться до формування керуючих цифрових сигналів для релейних модулів. Програмне забезпечення абстрагує поточний стан комутації ламп у вигляді бінарних змінних (HIGH/LOW або 1/0), керування якими реалізується через базовий API введення-виведення (digitalWrite), що повністю приховує фізику процесу від вищих рівнів софту та центрального сервера [9, с. 14].

Логіка програмного керування електромеханічними реле (EMR) вимагає обов'язкового врахування специфічних часових та конструктивних

характеристик пристрою на рівні розроблюваного мікрокоду. Зокрема, під час зміни значення керуючої змінної в прошивці мікроконтролера необхідно передбачати внутрішню часову затримку в межах від 5 до 10 мілісекунд, яка обумовлена фізичною інерцією рухомих елементів і є необхідною для надійного замикання або розмикання контактних пластин.

Поряд із цим, на рівні обробних алгоритмів критично важливим є впровадження програмного захисту від ефекту зациклювання або занадто частої зміни робочих станів комутаційного вузла. Реалізація такого програмного гістерезису або фіксованих захисних таймаутів дозволяє запобігти високій частоті перемикання логічних рівнів на цифровому виводі мікроконтролера, яка в іншому випадку неминуче призвела б до прискореного механічного зносу, підгоряння контактів та передчасного виходу з ладу силового елемента системи автоматизації.

Програмна інтеграція твердотільних реле (*Solid State Relays, SSR*) відкриває ширший спектр алгоритмічних рішень. Оскільки комутація в SSR відбувається на мікросекундному рівні за допомогою оптичних пар, у мікрокоді прошивки зникає потреба враховувати часові затримки на спрацювання контуру. Керування твердотільними модулями дозволяє реалізувати алгоритми синхронізації з переходом напруги через нуль (*zero-cross switching*), що мінімізує програмно індуковані мережеві перешкоди. На відміну від електромеханічних аналогів, мікропрограма для SSR може використовувати метод широтно-імпульсної модуляції (ШІМ) для реалізації алгоритмів плавного ввімкнення освітлення або димування.

Для більшості сценаріїв автоматизації освітлення обчислювальний профіль використання процесорного часу для обох типів реле є аналогічним. Вибір конкретного типу виконавчого механізму визначає внутрішню архітектуру тимчасових затримок у коді (конфігурацію таймерів для захисту EMR або параметрів ШІМ для SSR), дозволяючи розробнику оптимізувати прошивку під конкретні вимоги надійності, безшумності та швидкодії в межах локальної IoT-мережі.

1.4. Проблема синхронізації станів: інтеграція класичних фізичних вимикачів у цифрову мережу

Однією з найгостріших проблем під час проєктування та розробки програмного забезпечення систем «Розумного дому» є забезпечення безшовної взаємодії між сучасними цифровими інтерфейсами (зокрема, голосовим керуванням) та традиційною електрофурнітурою. Історично склалося так, що класичні настінні вимикачі розмикають силовий ланцюг, що на рівні архітектури розподілених мереж призводить до повного припинення подачі живлення та, як наслідок, до втрати інформаційного зв’язку з IoT-вузлом [12, с. 415]. У контексті розробки програмного комплексу такий підхід створює критичний архітектурний конфлікт.

Якщо користувач встановлює «розумну» лампу (*Smart Bulb*) зі вбудованим мережевим модулем, але при цьому змінює її стан за допомогою звичайного настінного вимикача, пристрій повністю деактивується. У цей момент його мікропрограмний стек припиняє виконання циклу опитування, і пристрій переходить у статус «офлайн» для центрального сервера або голосового асистента. Будь-які спроби надіслати команду керування через голосовий інтерфейс завершуються помилкою тайм-ауту (*timeout error*), оскільки серверний програмний шар не може встановити сокетне з’єднання зі знеструмленим вузлом [12, с. 520]. Ця ситуація руйнує базовий принцип систем автоматизації — забезпечення безперервного програмного моніторингу та надійного контролю за станами пристроїв.

Для вирішення цієї проблеми застосовується архітектурна концепція **програмно-апаратного розмежування логіки та живлення (Decoupling Logic from Power)**. Суть цього методу в контексті розробки софту полягає в тому, що класичний фізичний вимикач більше не розглядається програмою як силовий переривник лінії. Натомість керування навантаженням повністю делегується програмному драйверу реле, а сам фізичний вимикач перетворюється на низьковольтний цифровий датчик (джерело подій), логічний стан якого безперервно зчитується портами вводу-виводу

загального призначення (GPIO) мікроконтролера кінцевого вузла під керуванням відповідного мікрокоду [9, с. 14].

1.4.1. Програмна стабілізація регістрів та режим INPUT_PULLUP

Переведення вимикача в статус джерела цифрових сигналів створює специфічні алгоритмічні та конфігураційні виклики для розробника прошивки. Зокрема, порти мікроконтролера потребують чіткого програмного визначення початкових станів.

Якщо GPIO-пін ініціалізувати без прив’язки до логічного рівня, він перебуватиме у високоімпедансному («плаваючому») стані, генеруючи псевдовипадкові логічні нулі та одиниці через шуми навколишнього середовища. Для програмної стабілізації рівнів на шарі регістрів мікроконтролера виконується конфігурація внутрішньої підтяжки (**INPUT_PULLUP**). Це дозволяє програмно-апаратними засобами чітко зафіксувати логічну одиницю за відсутності замикання вимикача на землю (GND), спрощуючи структуру коду та підвищуючи надійність зчитування телеметрії [9, с. 16].

1.4.2. Алгоритми придушення брязкоту контактів (Debouncing)

Наступним викликом є обробка фізичного явища брязкоту (або дребізку) контактів (*Contact Bounce*), що вимагає розробки спеціальних фільтраційних алгоритмів у прошивці. Будь-який механічний перемикач під час зміни положення пластин генерує серію мікросекундних хибних імпульсів тривалістю від 5 до 20 мілісекунд. Оскільки тактова частота обчислювального ядра мікроконтролера (яка становить 80 МГц) дозволяє виконувати мільйони логічних операцій за секунду, одноразова зміна положення вимикача сприймається головним програмним циклом як послідовність із десятків швидких натискань. За відсутності програмного фільтра це призводить до хаотичного зациклення виконання функцій перемикання реле та мерехтіння освітлення [9, с. 18].

У практиці комп’ютерної інженерії усунення цієї проблеми реалізують двома альтернативними шляхами, що різняться за принципом і точкою прикладання фільтраційної логіки. Перший апаратний метод (Hardware Debouncing) передбачає інтеграцію зовнішніх фізичних RC-фільтрів або тригерів Шмітта безпосередньо у схемотехніку пристрою. Цей підхід повністю виноситься в апаратну частину проєкту, оскільки він спрямований на згладжування фізичного сигналу до його надходження на вхідні каскади обчислювального ядра і не потребує додаткового залучення процесорного часу мікроконтролера.

Альтернативним та більш гнучким рішенням є програмний метод (Software Debouncing), що базується на впровадженні фільтраційних алгоритмів безпосередньо в мікрокод прошивки. Під час фіксації першої зміни логічного рівня на цифровому виводі загального призначення (GPIO) програма ініціює таймер блокування, тривалість якого зазвичай становить від 20 до 50 мс, та ігнорує будь-які подальші флуктуації сигналу в цей період. Після завершення захисного таймауту мікропрограма виконує повторне валідаційне зчитування стану регістра, підтверджуючи або спростовуючи факт стабільної зміни положення вимикача. Цей метод є найбільш гнучким, економічно ефективним і повністю реалізованим на рівні програмного забезпечення, що дозволяє уникнути здорожчання та ускладнення друкованої плати [9, с. 25].

1.4.3. Реалізація логіки скінченного автомата

Після забезпечення стабільної програмної фільтрації сигналу реалізується архітектурна задача двосторонньої синхронізації станів. Мікропрограма кінцевого вузла будується за принципом скінченного автомата (*State Machine*), який оперує поточною бінарною змінною стану виконавчого механізму.

Якщо керуючий пакет надходить асинхронно від локального сервера з голосовим ШІ-асистентом, стан змінної та відповідного піна реле змінюється

програмно. Якщо ж користувач взаємодіє з фізичним вимикачем, обробник подій мікроконтролера фіксує інверсію логічного рівня на GPIO, програмно змінює внутрішній стан автомата на протилежний та негайно ініціює процедуру відправки оновленої телеметрії на MQTT-брокер [6; 8].

Така логіка побудови прошивки гарантує абсолютну синхронність розподіленої системи: центральний сервер завжди володіє актуальними даними про стан об’єкта автоматизації, а кінцевий користувач отримує можливість паралельного безконфліктного керування як за допомогою локальних нейромережевих моделей голосового інтерфейсу, так і через традиційні органи ручного перемикання. Для забезпечення швидкої, енергоефективної та безперебійної передачі сформованих телеметричних пакетів у локальній Wi-Fi-мережі необхідне використання спеціалізованих легковагових протоколів прикладного рівня, аналіз програмних особливостей яких є важливим теоретичним етапом у розробці.

1.5. Протоколи взаємодії та передачі даних (Wi-Fi, MQTT, HTTP)

Базовою вимогою для функціонування будь-якої розподіленої IoT-системи є організація надійного, швидкого та безпечного обміну інформацією між сенсорним рівнем та периферійним сервером обробки даних. Для побудови комунікаційної інфраструктури «Розумного дому» застосовується стек протоколів TCP/IP, де кожен рівень мережевої моделі вирішує специфічні задачі маршрутизації, комутації та пакування інформаційних пакетів [12, с. 145].

1.5.1. Стандарт бездротового зв’язку Wi-Fi (IEEE 802.11)

На фізичному та канальному рівнях обчислювальної мережі розумного дому найбільш раціональним є використання технології Wi-Fi. Клієнтські мікроконтролери сімейства ESP8266 інтегрують напівпровідникові радіомодулі, що функціонують у частотному діапазоні **2,4 ГГц** за стандартами **IEEE 802.11 b/g/n** [9, с. 4].

Вибір частоти 2,4 ГГц замість високочастотного діапазону 5 ГГц є інженерно обґрунтованим: радіохвилі цієї довжини мають значно вищу проникну здатність при проходженні крізь залізобетонні стіни та перекриття, що є критично важливим для вмонтованих углиб стін розумних вимикачів.

Незважаючи на підвищене енергоспоживання Wi-Fi порівняно з протоколами *Zigbee* або *Bluetooth Low Energy (BLE)*, для периферійних вузлів керування освітленням цей фактор є нівельованим завдяки наявності стаціонарного живлення від побутової електромережі 220 В. Ключовою перевагою стандарту є можливість прямого підключення кінцевих пристроїв до домашнього маршрутизатора без розгортання додаткових проміжних апаратних шлюзів [12, с. 415].

1.5.2. Протокол HTTP та його обмеження в IoT-мережах

На прикладному рівні класичним протоколом передачі даних є **HTTP (HyperText Transfer Protocol)**. Він побудований на строгому клієнт-серверному шаблоні та працює за синхронним принципом **«запит-відповідь» (*Request-Response*)** [12, с. 610].

При використанні HTTP для керування розумними пристроями застосовується архітектурний стиль *REST API*, де клієнт надсилає POST-запити для зміни стану цифрових пінів реле або періодично ініціює GET-запити для зчитування їх поточного статусу (механізм *Polling*).

Проте для сучасних розподілених систем реального часу протокол HTTP має суттєві архітектурні та експлуатаційні недоліки. Першочерговою проблемою є формування надлишкового трафіку (*Oversized Payload*) через значний обсяг службових текстових заголовків. У задачах автоматизації та Інтернету речей, де для передачі поточного стану виконавчого механізму або силового реле достатньо всього одного байта корисної інформації (логічного нуля чи одиниці), базовий стек протоколу змушений формувати мережевий пакет розміром у кілька сотень байтів, що призводить до нераціонального перевантаження пропускної здатності бездротового Wi-Fi-каналу.

Крім того, суттєвим обмеженням виступає неефективність класичного механізму періодичного опитування сервера (*Polling Inefficiency*). Необхідність постійного надсилання запитів від клієнтських вузлів для перевірки наявності нових команд створює тривале паразитне завантаження обчислювального ядра мікроконтролера і водночас генерує значні часові затримки (*latency*) в контурі керування, що є неприпустимим для систем, які функціонують у режимі реального часу.

Для систем голосового керування на базі ШІ, де затримка інтерфейсу понад 200 мілісекунд сприймається користувачем як відмова системи, використання HTTP є технічно та інженерно недоцільним [12, с. 635].

1.5.3. Протокол MQTT як галузевий стандарт Інтернету речей

Ефективною альтернативою, що повністю вирішує проблему мережевого оверхеду та затримок, є легковаговий протокол **MQTT (Message Queuing Telemetry Transport)**, спеціально розроблений для систем телеметрії з обмеженими обчислювальними ресурсами та нестабільними каналами зв’язку [6].

MQTT базується на асинхронному патерні проєктування «Публікація/Підписка» (Publish/***Subscribe***). Фізичні пристрої повністю ізольовані один від одного, а центральним координаційним вузлом виступає **MQTT-брокер** (зокрема, розгорнутий на локальному сервері *Eclipse Mosquitto*) [7]. Брокер виконує функцію асинхронного маршрутизатора повідомлень, адресованих через гнучку деревоподібну структуру текстових топіків (наприклад, smarthome/node1/relay1/set).

Програмний заголовок пакета MQTT займає лише два байти, що робить його архітектурно оптимальним для застосування в умовах обмежених обчислювальних ресурсів периферійних мікроконтролерів [6]. Окрім мінімальних вимог до обсягів мережевого трафіку, цей протокол надає унікальні сервісні механізми, спрямовані на стабілізацію розподілених систем та підвищення їхньої загальної відмовостійкості.

Одним із таких інструментів є технологія гарантування якості обслуговування (*Quality of Service*), яка підтримує три рівні надійності доставки пакетів даних [6]. Якщо базовий рівень обробки передбачає одноразову відправку повідомлення без очікування підтвердження, то вищий рівень гарантує обов’язкову доставку пакета підписнику щонайменше один раз завдяки впровадженню механізму квитування. Найсуворіший режим забезпечує доставку строго один раз через чотирьохкомпонентне рукостискання. Для завдань надійної комутації силових реле побутового освітлення оптимальним архітектурним вибором є саме рівень із гарантованим квитуванням, який повністю нівелює ризики втрати керуючих команд під час виникнення фонових перешкод у радіоефірі.

Наступним критично важливим елементом є механізм проактивного моніторингу поточних мережевих сесій, відомий як «останнє волевиявлення» (*Last Will and Testament*). У разі раптової аварійної втрати TCP-сокета з боку клієнтського мікроконтролера, що зазвичай відбувається під час зникнення електроживлення або критичного апаратного збою, мережевий брокер автоматично публікує заздалегідь визначене сервісне повідомлення у топік статусу пристрою. Це дозволяє центральному серверу та локальному голосовому ШІ-асистенту миттєво зафіксувати перехід вузла у стан відключення та оперативно скоригувати контекст подальшого діалогу з користувачем.

Для забезпечення швидкої синхронізації після системних збоїв також застосовується технологія збережених повідомлень (*Retained Messages*). Активація відповідного прапорця змушує брокера кешувати останній успішно надісланий пакет у межах конкретного топіка. Завдяки цьому під час первинного підключення нового клієнта або після планового перезавантаження мікроконтролера периферійний пристрій миттєво отримує актуальний збережений статус системи, що повністю виключає необхідність тривалого очікування надходження нової директивної команди.

Завдяки цим характеристикам протокол MQTT забезпечує стабільний, повнодуплексний та асинхронний зв’язок у режимі реального часу, дозволяючи центральному серверу голосового штучного інтелекту та клієнтським прошивкам миттєво обмінюватися даними без виникнення конфліктів архітектури.

РОЗДІЛ 2 ПРОЕКТУВАННЯ АПАРАТНО-ПРОГРАМНОГО КОМПЛЕКСУ СИСТЕМИ ГОЛОСОВОГО КЕРУВАННЯ

У цьому розділі розглядаються особливості системного проєктування програмної архітектури розподіленого комплексу та обґрунтовується вибір його ключових компонентів. Основна увага приділена детальному аналізу функціональних і нефункціональних вимог до системи керування, розробці безконфліктної моделі асинхронної взаємодії пристроїв транспортного рівня, а також методам програмної оптимізації ресурсомістких нейромережевих моделей для їх стабільного локального функціонування в межах лімітів серверного обладнання.

2.1. Аналіз вимог до розроблюваної програмної системи керування освітленням

Процес проєктування розподілених систем автоматизації та Інтернету речей починається з декомпозиції, детального аналізу та чіткого структурування функціональних і нефункціональних вимог до програмного забезпечення.

Головною інженерною метою проєкту є створення автономної локальної програмної архітектури, що забезпечує безконфліктне клієнт-серверне керування контурами освітлення через інтелектуальний голосовий інтерфейс ШІ-асистента та алгоритми асинхронного зчитування станів класичних настінних вимикачів у режимі реального часу.

2.1.1. Функціональні вимоги до програмного комплексу

На основі декомпозиції загальної мети дослідження було сформовано перелік базових функціональних вимог до розроблюваного програмного забезпечення. Першочерговою серед них є повна локалізація процесів обробки голосового потоку. Програмні модулі сервера повинні виконувати наскрізний цикл цифрової обробки мовлення, що охоплює транскрибацію голосу в текстовий формат (STT) [1], семантичний аналіз із розпізнаванням намірів користувача (NLU) [3] та наступну зворотну генерацію мовної

відповіді асистента (TTS) [5]. Усі зазначені операції мають здійснюватися виключно в межах локальної обчислювальної мережі (LAN) без трансляції конфіденційних даних та надсилання запитів до зовнішніх хмарних API-сервісів.

Поряд із цим, архітектура центрального програмного ядра повинна підтримувати гібридну логіку маршрутизації інформаційних потоків. Розроблюваний алгоритм має в автоматичному режимі диференціювати чіткі директивні команди безпосереднього керування кінцевими пристроями від загальних комунікативних чи довідкових запитів користувача. При цьому фрази довільного контексту повинні оперативно перенаправлятися до локальної квантованої великої мовної моделі (LLM) для генерації логічно завершених контекстуальних відповідей [4].

Важливою функціональною вимогою до системи є забезпечення надійного асинхронного двостороннього керування елементами автоматизації. Мікропрограма периферійного клієнтського вузла під управлінням мікроконтролера має в реальному часі обробляти події зміни логічного рівня на вхідних виводах GPIO від фізичного настінного вимикача [9]. Водночас поточні програмні стани відповідних змінних у базі даних сервера повинні автоматично, асинхронно та безконфліктно синхронізуватися через брокер повідомлень за допомогою протоколу MQTT [6]. Для забезпечення високої адаптивності комплексу серверна частина обов'язково має містити ізольований інтерактивний модуль первинного налаштування (Setup Wizard), призначений для динамічного мапування системних індексів фізичних пристроїв введення-виведення аудіосигналу в підсистемі ALSA [10] та верифікації мережевих координат брокера.

2.1.2. Нефункціональні вимоги до програмної архітектури

До переліку критично важливих нефункціональних вимог, що визначають вибір програмного стеку та методів алгоритмічної оптимізації, належить насамперед забезпечення низької часової затримки (Low Latency). Згідно з

архітектурними критеріями, сумарний час виконання всього розробленого програмного конвеєра — починаючи від моменту фіксації завершення голосового запиту користувача і закінчуючи безпосередньою відправкою результуючого керуючого пакету протоколу MQTT або ж початком акустичного відтворення синтезованого аудіофайлу відповіді — є жорстко обмеженим і не повинен перевищувати 1,5–2 секунди.

Не менш критичною вимогою виступає висока програмна відмовостійкість (Fault Tolerance) проєктуваної системи. У разі виникнення критичного апаратного чи софтверного збою на центральному сервері, раптової зупинки служби MQTT-брокера [7] або повної втрати бездротового Wi-Fi з'єднання [12], базовий автономний алгоритм прошивки периферійного клієнтського вузла має безперервно функціонувати у фоновому циклі. Це дозволяє забезпечити повноцінне автономне ручне керування силовим реле елементів освітлення шляхом прямого опитування фізичних регістрів GPIO мікроконтролера [9].

Окремим визначальним аспектом є повна сумісність створюваного програмного забезпечення з наявним програмно-апаратним оточенням. Серверний компонент комплексу оптимізується для стабільного розгортання в операційному середовищі Ubuntu Linux [10] із обов'язковим залученням технології апаратного прискорення NVIDIA CUDA. Це необхідно для високоефективного паралельного виконання нейромережевих інференсів на графічному процесорі в умовах ліміту відеопам'яті у 12 ГБ VRAM [2]. Водночас клієнтська мікропрограма периферійного вузла проєктується як максимально легковагове рішення на базі мови програмування C++ із використанням спеціалізованої бібліотеки PubSubClient [8]. Такий підхід гарантує високий ступінь компресії підсумкового бінарного файлу прошивки, що дозволяє безперешкодно вмістити його в обмежені ресурси Flash-пам'яті та оперативної пам'яті RAM мікропроцесорів лінійки ESP8266 [9].

2.2. Проєктування архітектури голосового ШІ-асистента (вибір моделей STT/TTS та їх оптимізація під наявні апаратні ресурси)

Відповідно до розроблених системних вимог, програмна архітектура комплексу автоматизації має трирівневу структуру: центральний серверний рівень логічного керування (*Brain*), транспортний рівень асинхронного обміну повідомленнями (*Nerve*) та клієнтський рівень периферійного мікропрограмного керування (*Client/Firmware*).

Серверний рівень реалізовано у вигляді багатопотокового програмного конвеєра (*Pipeline*) на базі мови Python у середовищі ОС Ubuntu [10]. Цей додаток виступає головним процесинговим хабом, що інтегрує підсистеми розпізнавання мовлення, класифікації намірів, семантичного аналізу та синтезу голосу [11].

Розроблену структурну схему міжрівневої взаємодії та компонентного складу кожної підсистеми наведено на рис. 2.1.

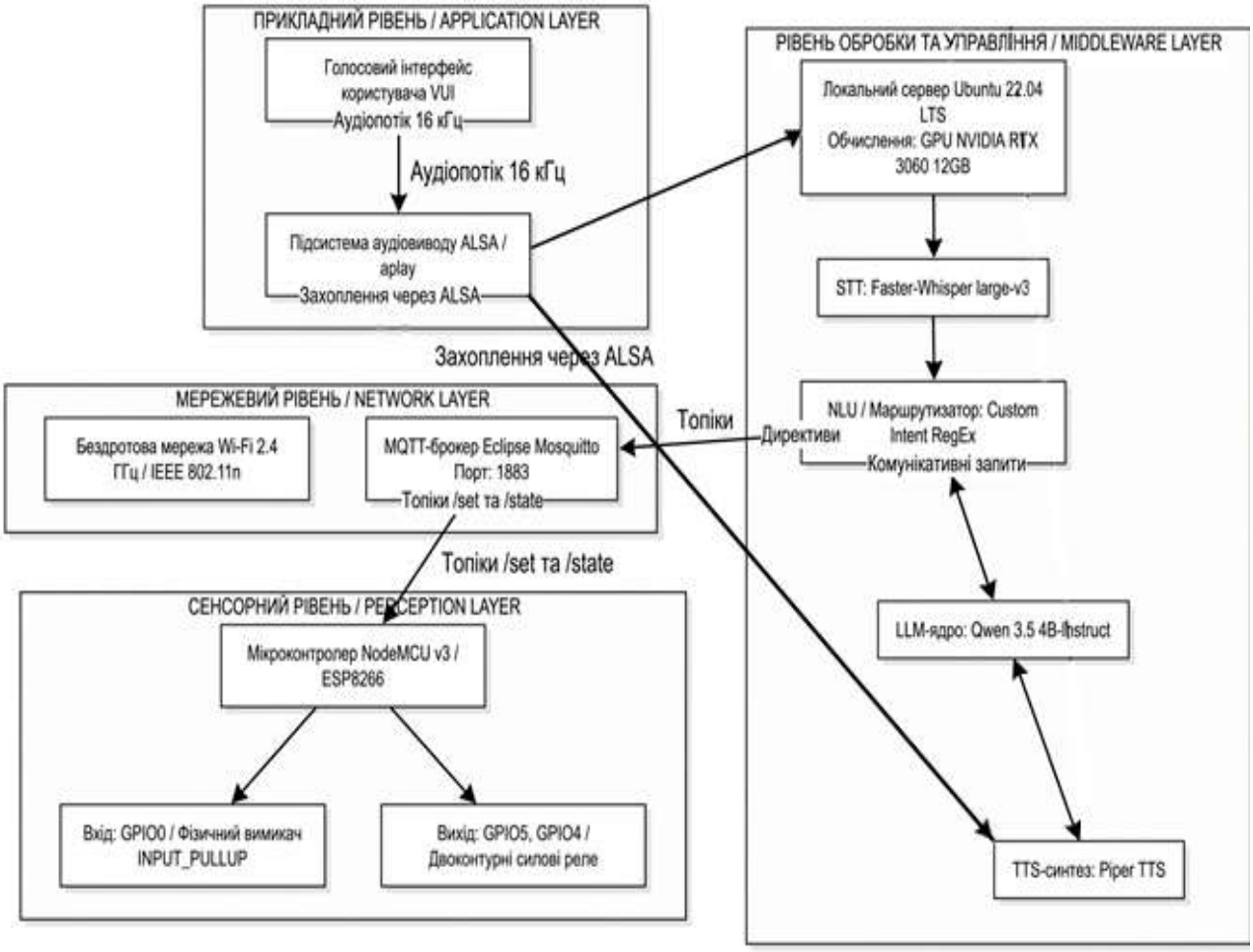

Рисунок 2.1 – Архітектурна схема взаємодії прикладного, обробного, мережевого та сенсорного рівнів системи

2.2.1. Підсистема розпізнавання мовлення (Speech-to-Text)

Для транскрибації вхідного аудіопотоку в текстовий формат інтегровано нейромережеву модель Faster-Whisper (large-v3), вибір якої зумовлений високою точністю розпізнавання української мови та відмінною стійкістю до акустичного шуму [1]. Оптимізація обчислювальним рушієм CTranslate2 забезпечує високоефективний інференс на тензорних ядрах (Tensor Cores) графічного прискорювача NVIDIA RTX 3060, що дозволяє знизити споживання відеопам'яті майже вдвічі, виконуючи критично важливу роль в умовах жорсткого апаратного ліміту у 12 ГБ VRAM [2].

Функціонально зазначена підсистема працює в режимі безперервного фонового прослуховування (Continuous Listening) із постійною фіксацією визначеного слова активації (Wake Word), причому повноцінний запис та фінальне транскрибування директивної команди ініціюються виключно після успішного розпізнавання цього сервісного тригера. Для забезпечення апаратної стабільності та запобігання помилкам переповнення вхідного буфера (buffer overrun) в архітектуру інтегровано алгоритм адаптивного підбору частоти дискретизації. Цей механізм реалізує автоматичний циклічний перебір між значеннями 16, 44.1 та 48 кГц до моменту успішної програмної ініціалізації аудіоінтерфейсу низького рівня ALSA, що дозволяє повністю уникнути збоїв при первинному підключенні різного звукового обладнання [10].

2.2.2. Модуль гібридної маршрутизації та класифікації намірів (Intent Recognizer)

З метою мінімізації часових затримок в архітектурну логіку головного процесингового хабу впроваджено спеціалізований алгоритм гібридної маршрутизації (Hybrid Routing), який здійснює автоматичну класифікацію вхідних текстових рядків, отриманих від модуля розпізнавання мовлення, за двома базовими функціональними категоріями [3]. До першої групи належать чіткі директивні команди, що є короткими мовними структурами, які містять специфічні лексеми дії (зокрема, «увімкни» або «вимкни») та відповідні унікальні ідентифікатори кінцевих пристроїв (такі як «світло» чи «люстра»). Такі текстові послідовності негайно перехоплюються програмним модулем HardwareManager для прямої та асинхронної публікації керуючого пакета безпосередньо в мережу MQTT-брокера. Завдяки виконанню цієї транзакції в обхід важких та ресурсомістких нейромережевих мовних моделей вдається істотно оптимізувати обчислювальний контур і скоротити час реакції виконавчих пристроїв автоматизації до кількох мілісекунд [6].

Другу категорію інформаційних потоків утворюють комунікативні запити користувача, представлені у вигляді неструктурованих фраз довільного контексту, які не мають явних ознак команд прямого керування апаратною периферією. Такі запити автоматично перенаправляються системою до великої мовної моделі для здійснення контекстуального аналізу та наступної генерації логічно завершеної текстової відповіді ШІ-асистента [4].

2.2.3. Підсистема обробки природної мови (LLM) та оптимізація VRAM

Локальним ядром контекстуального аналізу та генерації відповідей виступає велика мовна модель **Qwen 3.5 (4B Instruct)**, розгорнута через фреймворк *Ollama* [3; 4]. Використання мовної моделі класу 4B виступає оптимальним архітектурним компромісом, оскільки вона забезпечує високу лінгвістичну точність і, на відміну від значно більших аналогів класу 7B або 8B, гарантовано функціонує в межах наявного апаратного ліміту у 12 ГБ VRAM паралельно з активним потоковим конвеєром Faster-Whisper. Для повного усунення часових затримок під час зчитування вагових коефіцієнтів моделі з постійного накопичувача, що в середньому займає від 3 до 7 секунд, у конфігурації API-запитів реалізовано механізм перманентного завантаження відеопам'яті (*VRAM Preload*) шляхом встановлення параметра `keep_alive: -1` [4]. Це інженерне рішення забезпечує безперервне утримання моделі у VRAM безпосередньо після старту серверної частини комплексу, що повністю виключає необхідність повторної ініціалізації ваг при кожному новому зверненні користувача.

Додатково в архітектуру системи впроваджено спеціалізований алгоритм фільтрації міркувань (*Think Filter*), необхідність якого зумовлена специфікою функціонування сучасних моделей обробки природної мови, схильних до генерації розгорнутих ланцюжків міркувань (*Chain of Thought*) всередині системних тегів `<think>...</think>`. З метою ефективного

відсікання цієї службової інформації у загальний програмний конвеєр інтегровано фільтраційний шар на базі регулярних виразів (*RegEx*). Цей модуль у режимі реального часу аналізує вихідний текстовий потік та повністю видаляє блоки внутрішніх міркувань нейромережі, передаючи на підсистему синтезу мовлення виключно фінальну, логічно завершену репліку ШІ-асистента.

2.2.4. Підсистема синтезу мовлення (TTS) та апаратний вивід звуку

Голосова відповідь генерується за допомогою локального нейромережевого синтезатора **Piper TTS**, який демонструє високий показник *Real-Time Factor (RTF)* порівняно з аналогами [5]. Побудований аудіопотік у форматі WAV передається системному програвачу `aplay` архітектури ALSA [10].

Для захисту від колізій монопольного доступу до звукової карти реалізовано програмний шар віртуалізації пристроїв, який автоматично трансформує прямі апаратні адреси (наприклад, `hw:1,0`) у програмні плагіни передискретизації (`plughw:1,0`) [10].

2.2.5. Модуль конфігурації та розгортання (Setup Wizard)

Для забезпечення високого рівня портативності розроблюваного програмного забезпечення було створено ізольований конфігураційний модуль `setup_wizard.py`. Зазначений скрипт функціонує повністю незалежно від основного програмного ядра системи та призначений для послідовного вирішення комплексу підготовчих завдань. На першому етапі модуль здійснює автоматичне опитування системного каталогу `/proc/asound/cards` з метою динамічної побудови карти доступних в операційній системі фізичних аудіоінтерфейсів [10]. Поряд із цим користувачеві надається консольний інтерактивний інтерфейс, який дозволяє у зручному діалоговому режимі обрати необхідні пристрої введення-виведення звукового сигналу та встановити точні мережеві координати

MQTT-брокера [6]. Кінцевим етапом функціонування цього модуля є автоматична серіалізація встановлених параметрів та їх збереження у структурований конфігураційний файл `user_config.json`, з якого головне обчислювальне ядро системи автоматично зчитуватиме налаштування при кожному наступному запуску комплексу. При кожному старті головне програмне ядро автоматично зчитує параметри з конфігураційного файлу, що дозволяє швидко розгортати серверний компонент у межах будь-якої локальної мережевої інфраструктури без модифікації вихідного коду.

2.3. Проєктування програмної логіки клієнтського вузла на базі мікроконтролера

Як обчислювальну платформу для виконання мікропрограмного забезпечення периферійного виконавчого клієнтського вузла обрано плату розробника NodeMCU на базі 32-бітного енергоефективного мікроконтролера **ESP8266**. Цей вибір обґрунтований наявністю достатнього пулу портів введення-виведення загального призначення (GPIO), інтегрованого на системному шарі архітектури програмного стеку Wi-Fi, та високою стабільністю виконання мікрокоду в безперервному циклі [9, с. 4].

Головним архітектурним завданням розроблюваного програмного забезпечення кінцевого пристрою є логічне керування двоконтурною системою освітлення на основі подій та асинхронне зчитування поточних станів настінного вимикача. Проєктування логіки пристрою зосереджується на ініціалізації регістрів, обробці цифрових сигналів та абстрагуванні фізичних процесів у програмні бінарні змінні стану [8].

2.3.1. Алгоритм програмного керування освітленням

Для реалізації програмного керування силовими ланцюгами живлення освітлювальних приладів у мікрокоді прошивки створюється абстрактний рівень драйвера виконавчих механізмів. На рівні головної функції ініціалізації `setup()` виконується пряме конфігурування режимів роботи

відповідних регістрів спеціального призначення мікроконтролера. Програма переводить цифрові порти у режим виходу (`OUTPUT`), що дозволяє динамічно модулювати логічні рівні на пінах [9, с. 12].

Згідно з розробленою архітектурою програмних з'єднань, функцію керування силовими релейними модулями було делеговано визначеним цифровим виходам периферійного мікроконтролера. Зокрема, перший контур, що керує програмною змінною основного освітлення, жорстко прив'язаний до регістра порту D1, який у схемотехніці пристрою відповідає архітектурному виводу GPIO5. Водночас другий контур, призначений для регулювання програмної змінної додаткового джерела світла, комутується через регістр порту D2, закріплений за архітектурним піном GPIO4. Такий адресний розподіл цифрових виводів дозволяє забезпечити ізольоване та незалежне програмне керування кожною окремою лінією силового навантаження в реальному часі. Алгоритм керування інтегрує підтримку інверсної логіки (**Active-Low**), яка є промисловим стандартом для виконавчих IoT-модулів із гальванічною опторозв'язкою. Для активації контуру та ввімкнення світла програма формує на відповідному піні (GPIO5 або GPIO4) логічний нуль (`LOW`), а для деактивації та знеструмлення — логічну одиницю (`HIGH`, потенціал 3,3 В).

Логіка обробки команд будується на базі асинхронних зворотних викликів (*callbacks*) бібліотеки `PubSubClient` [8]. При надходженні текстового пакета від MQTT-брокера клієнтська прошивка здійснює парсинг корисного навантаження (*payload*) і через функцію `digitalWrite()` миттєво інвертує бінарний стан на відповідному виході, що забезпечує загальний час реакції периферійного вузла на рівні кількох мілісекунд.

2.3.2. Програмна обробка фізичного вимикача (зчитування станів, усунення брязкоту контактів)

Для реалізації гібридної логіки взаємодії (паралельно через мовний ШІ-інтерфейс сервера та ручне перемикання) алгоритм прошивки оперує

класичним настінним вимикачем як цифровим дискретним сенсором введення подій. У програмі виконується ініціалізація вхідного порту D3 (GPIO0) у режим зчитування.

Оскільки за відсутності стабільного потенціалу пін може переходити у високоімпедансний стан і генерувати програмні збої через наведення, у коді конфігурації порту примусово активується внутрішній підтягувальний резистор мікроконтролера за допомогою директиви `pinMode(GPIO0, INPUT_PULLUP)` [9, с. 14]. Таке програмне рішення дозволяє зафіксувати чітку логічну одиницю (`HIGH`) у розімкненому стані вимикача. При замиканні контактів лінія програмно притискається до логічного нуля (`LOW`).

Проєктування архітектури прошивки на базі GPIO0 враховує специфіку апаратного завантажувача (*bootloader*) чипа ESP8266: оскільки цей пін визначає режим старту процесора при подачі живлення, логіка ініціалізації побудована так, щоб виключити програмно-апаратні конфлікти під час перезавантаження пристрою при випадково затиснутій кнопці, гарантуючи запуск робочого циклу програми, а не сервісного режиму флеш-прошивки [9, с. 16].

Для подолання явища механічного брязкоту контактів (*Contact Bounce*), що спричиняє серію помилкових логічних перепадів на піні, розроблено неблокуючий алгоритм програмної фільтрації (**Software Debouncing**). Алгоритм базується на використанні системного таймера апаратного аптайму `millis()` без зупинки основного обчислювального потоку (*Non-blocking design*), що дозволяє раціонально використовувати процесорний час.

Програмна логіка фільтрації флуктуацій вхідного сигналу функціонує на основі послідовного циклічного алгоритму обробки даних у реальному часі. У кожній окремій ітерації головної функції `loop()` виконується миттєве зчитування поточного логічного рівня безпосередньо з регістра порту `GPIO0`. Якщо поточний результат зчитування не збігається з попереднім збереженим значенням вимикача, керуюча програма автоматично

скидає та перезапускає внутрішній програмний лічильник часу `lastDebounceTime`.

Далі алгоритм переходить до фази безперервного обчислення різниці між поточним часом роботи мікроконтролера та раніше зафіксованою міткою `lastDebounceTime`. Поточний логічний рівень визнається системою як стабільний, валідний і остаточно фіксується прошивкою лише тоді, коли обчислена часова різниця гарантовано перевищує жорстко задане програмне вікно фільтрації тривалістю 50 мілісекунд (`debounceDelay`). Такий підхід дозволяє надійно відсікти високочастотні апаратні перешкоди та мікромеханічний брязкіт контактів до моменту прийняття рішення про зміну стану силового контуру.

Після підтвердження стабільної зміни положення вимикача мікропрограма кінцевого вузла ініціює роботу скінченного автомата (*State Machine*): вона програмно інвертує поточне бінарне значення керуючої змінної реле, викликає функцію зміни стану відповідного GPIO та негайно формує вихідний інформаційний пакет для публікації оновленого статусу в мережу MQTT [6; 8]. Це забезпечує безконфліктне ручне керування навіть за умови тимчасової відсутності зв'язку з сервером, зберігаючи високу локальну автономність периферійного вузла.

2.4. Розробка алгоритму двосторонньої синхронізації станів (програмне та голосове керування без конфліктів)

Найважливішим архітектурним досягненням розробленого програмного забезпечення є алгоритмічне вирішення проблеми колізій та конфліктів станів між розподіленою цифровою інфраструктурою сервера та локальним механічним перемикачем. В основу логіки функціонування периферійного мікроконтролера покладено математичну модель **скінченного автомата (State Machine)** з алгоритмом гібридного керування, що орієнтований на події та підтримує пріоритет негайної обробки локальних змінних на рівні мікрокоду прошивки [8].

Алгоритм двосторонньої асинхронної синхронізації функціонує за трьося базовими програмними принципами:

2.4.1. Локальний пріоритет обробки подій (Local Event Control)

Головний програмний цикл прошивки мікроконтролера безперервно перевіряє стан регістра вхідного порту GPIO0. При фіксації зміни логічного рівня, що успішно пройшла програмну фільтрацію (*software debouncing*), незалежно від вектора переходу (`HIGH -> LOW` або навпаки), алгоритм ініціює миттєве виконання локальної процедури інверсії бінарної змінної [9, с. 14].

Одразу після цього мікрокод змінює логічний рівень на цифровому виході керування силовим реле. Даний обчислювальний процес протікає повністю автономно на рівні ядра кінцевого вузла і не потребує надсилання транзакцій у мережу або очікування валідації з боку сервера. Це гарантує абсолютну програмну відмовостійкість і стабільність функціонування базового освітлення об'єкта навіть у разі критичних збоїв локальної Wi-Fi-мережі, відсутності зв'язку з MQTT-брокером або аварійної зупинки серверного скрипту ШІ-асистента [7; 12].

2.4.2. Висхідна синхронізація даних (Upstream Data Synchronization)

Безпосередньо після локальної програмної зміни стану реле мікрокод кінцевого вузла формує інформаційний пакет, у якому корисне навантаження (*payload*) містить актуальний бінарний статус керованого контуру («ON» або «OFF»). Спеціальна функція MQTT-клієнта виконує асинхронну публікацію цього пакета у відповідний телеметричний топік стану (наприклад, `smarthome/node1/relay1/state`) [6; 8].

Завдяки реалізації цієї висхідної гілки зворотного зв'язку реляційна база даних центрального сервера та контекстна пам'ять локального ШІ-асистента завжди містять точний і актуальний статус виконавчого пристрою [3]. Це повністю виключає виникнення логічних колізій і десинхронізації

образу системи під час наступної інтерпретації голосових команд користувача.

2.4.3. Низхідна синхронізація та обробка команд (Downstream Command Synchronization)

Паралельно з виконанням фонових обчислень обробки GPIO, клієнтський програмний стек мікроконтролера перебуває в режимі очікування подій від мережі, будучи зареєстрованим підписником на керуючий топік команд `smarthome/node1/relay1/set` [8].

Коли користувач ініціює мовну команду, а серверний Python-скрипт успішно завершує її розпізнавання (*STT*) та семантичний аналіз (*NLP*) [1; 3], сервер виступає видавцем і надсилає керуючу текстову директиву до зазначеного топіку [6]. При надходженні пакета на клієнтську сторону автоматично викликається неблокуюча програмна функція `callback` [8]. Вона здійснює парсинг рядка, звіряє токени, через функцію `digitalWrite()` перемикає вихідний GPIO-пін [9] та ініціює вихідний звіт у топік `state` для фінального підтвердження виконання операції на стороні сервера.

Застосування такої архітектури обробки даних дозволяє повністю розірвати жорстку залежність фізичного статичного положення клавіші настінного вимикача від поточного стану освітлювального приладу. Фізична кнопка в розробленій програмній системі функціонує в режимі **програмного тригера перемикання (Toggle Switch)**: будь-яка фіксація зміни її логічного стану на GPIO автоматично інвертує поточну бінарну змінну скінченного автомата на протилежну.

Такий підхід забезпечує найбільш природний, ергономічний та інтуїтивно зрозумілий патерн гібридної взаємодії користувача з розумним простором, повністю нівелюючи ризик виникнення системних конфліктів під час керування розподіленою IoT-мережею.

2.5. Вибір програмного стеку та топології мережі (клієнт-серверна взаємодія через MQTT-брокер)

Для забезпечення надійного та низькозатримкового обміну даними між сервером (голосовим ШІ-асистентом) та периферійними пристроями NodeMCU спроєктовано логічну топологію мережі типу «зірка» (*Star Topology*) [12, с. 385]. Центральним вузлом-маршрутизатором у такій архітектурі виступає виділений MQTT-брокер Eclipse Mosquitto, розгорнутий у фоновому режимі (як системний демон) на сервері під керуванням ОС Ubuntu Linux [7; 10].

Комунікація між підсистемами здійснюється виключно в межах локальної обчислювальної мережі (LAN) через стандартний TCP-порт 1883 [12, с. 635]. Локальна обчислювальна архітектура дозволила повністю відмовитися від сторонніх хмарних сервісів, що мінімізувало час доставки керуючих пакетів до рівня менше 10 мілісекунд, гарантуючи абсолютну автономність та високий рівень кібербезпеки комплексу.

2.5.1. Проєктування ієрархії топіків (Topic Tree)

Замість прямої IP-маршрутизації протокол MQTT оперує гнучкою деревоподібною структурою адресації повідомлень [6]. Для логічного керування двоканальним виконавчим реле на базі мікроконтролера ESP8266 було розроблено та впроваджено спеціалізовану ієрархічну схему топіків [8; 9]. У межах цієї архітектурної схеми надсилання директивних керуючих команд від центрального сервера до периферійного мікроконтролера для комутації першого каналу, що відповідає за основне освітлення, здійснюється через цільовий топік `smarthome/node1/relay1/set`. Для забезпечення стабільного зворотного зв'язку та асинхронної публікації фактичного апаратного стану цього ж першого каналу у напрямку від мікроконтролера до сервера виділено інформаційний топік `smarthome/node1/relay1/state`. Взаємодія із другим каналом

керування побудованим комплексом реалізована за ідентичним принципом за допомогою компліментарної пари топіків `smarthome/node1/relay2/set` та `smarthome/node1/relay2/state`, які відповідають за безпосередню адресацію вхідних команд та динамічний моніторинг поточного статусу додаткового джерела світла відповідно.

Архітектурне розділення інформаційних потоків на суфікси `/set` (команда дії) та `/state` (зворотний зв'язок) є фундаментальним патерном проєктування відмовостійких розподілених систем [6]. Центральний сервер фіксує зміну статусу пристрою в базі даних та контекстній пам'яті ШІ-моделі виключно після отримання асинхронного підтвердження від прошивки мікроконтролера у топіку `state` [3]. Такий підхід повністю унеможливлює розсинхронізацію даних у разі втрати пакетів у Wi-Fi-мережі [12, с. 415].

2.5.2. Обґрунтування програмного стеку та корисного навантаження

Як корисне навантаження (*Payload*) мережевих повідомлень використано легковагові текстові рядки «ON» та «OFF». Свідома відмова від структурованого формату JSON для базових команд перемикання цифрових пінів дозволила уникнути підключення додаткових об'ємних бібліотек десеріалізації на клієнтській стороні [8]. Зазначений підхід дозволив суттєво оптимізувати використання оперативної пам'яті (SRAM) мікроконтролера ESP8266 та звільнити обчислювальні ресурси ядра для підтримки безперебійного Wi-Fi-з'єднання [9, с. 14]. При цьому реалізація клієнтської сторони апаратно-програмного комплексу базується на використанні оптимізованої бібліотеки PubSubClient мовою програмування C++ [8]. У межах цієї архітектури мікроконтролер здійснює обробку вхідних мережевих подій через неблокуючу функцію зворотного виклику (*callback*). Водночас у головному циклі програми `loop()` інтегровано асинхронний алгоритм автоматичного відновлення сесії (*reconnect*), який оперативно активується у

разі фіксації обриву TCP-сокета з брокером, забезпечуючи високу автономність периферійного вузла [9, с. 16].

Натомість на серверній стороні взаємодія з комунікаційною матрицею реалізована за допомогою Python-бібліотеки paho-mqtt [7]. З метою оптимізації обчислювальних процесів екземпляр MQTT-клієнта ініціалізується та запускається в окремому ізольованому асинхронному потоці за допомогою методу `loop_start()`. Таке інженерне рішення дозволяє повністю відокремити мережеві операції введення-виведення від головного обчислювального контуру сервера. Це гарантує, що можливі мережеві затримки під час транзитного перевантаження бездротового Wi-Fi-каналу не блокуватимуть стабільну роботу ресурсомістких нейромережевих моделей обробки мовлення та логічного інференсу штучного інтелекту [12, с. 145].

2.5.3. Проєктування підсистеми конфігурації сервера

З метою забезпечення високого рівня абстракції програмного забезпечення та можливості його гнучкого налаштування без втручання у вихідний код розроблено спеціалізований конфігураційний модуль `setup_wizard.py`. При першій ініціалізації серверної частини цей скрипт сканує системні інтерфейси ОС Ubuntu Linux, опитує каталог `/proc/asound/cards` і дозволяє адміністратору в інтерактивному консольному режимі обрати робочі індекси пристроїв введення та виведення аудіосигналу в підсистемі ALSA [10].

Результати сканування, конфігурація звукових плагінів та мережеві координати локального брокера Mosquitto серіалізуються у структурований JSON-файл конфігурації `user_config.json`:

```
{
"microphone_alsa_id": "hw:1,0",
"speaker_alsa_id": "plughw:0,0",
"mqtt_broker_ip": "192.168.0.105",
"mqtt_port": 1883
```

```
}
```

Валідація та зчитування параметрів із цього файлу здійснюються автоматично під час кожного запуску головного програмного ядра системи. Розроблена модульна архітектура конфігурування робить серверне програмне забезпечення повністю портативним, усуває жорстку прив’язку до конкретних апаратних адрес під Linux та дозволяє легко розгортати комплекс у будь-якій локальній мережевій інфраструктурі [10].

РОЗДІЛ 3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ

У цьому розділі описуються етапи практичної реалізації, розгортання та експериментального дослідження створеного апаратно-програмного комплексу. Сформульовано методику комплексного інтеграційного тестування розподіленої системи, наведено алгоритми конфігурування серверного операційного середовища з підтримкою апаратного прискорення, а також представлено результати профілювання швидкодії нейромережевого конвеєра й аналізу відмовостійкості клієнтських мікроконтролерів у реальних умовах експлуатації.

3.1. Постановка задачі розробки діючого програмного прототипу та методика тестування

Головною метою третього етапу кваліфікаційної роботи є практична та програмна реалізація розробленого комплексу локальної автоматизації, а також експериментальна перевірка його працездатності й стабільності в умовах, максимально наближених до реальної побутової експлуатації розподілених систем [12, с. 415].

Практичне впровадження вимагає послідовного розгортання серверної інфраструктури [10], налаштування операційного середовища кінцевого вузла, написання й оптимізації мікрокоду мікроконтролера [9], а також комплексної програмної інтеграції всіх підсистем за допомогою асинхронного протоколу прикладного рівня MQTT [6].

Для об'єктивної оцінки ефективності розробленого програмного забезпечення сформовано комплексну методику експериментального тестування. Експериментально-дослідницька частина розподіленої системи покликана верифікувати повне виконання раніше сформульованих функціональних та нефункціональних вимог (підрозділ 2.1). Цей процес декомпонується на кілька взаємопов'язаних етапів, що виконуються у логічній послідовності.

Насамперед здійснюється тестування автономного алгоритмічного пріоритету, яке передбачає комплексну перевірку безвідмовності та стабільності функціонування локальної логіки прошивки мікроконтролера, зокрема фонового циклу опитування регістрів GPIO, обробки апаратних подій та комутації реле. Головною метою цього етапу є верифікація працездатності периферійного вузла за умови повної відсутності підключення до бездротової мережі Wi-Fi або при раптовій втраті зв'язку з брокером повідомлень, що імітує аварійне вимкнення центрального сервера чи маршрутизатора [8; 9].

Наступним кроком є мережеве профілювання затримок телеметрії (*Telemetry Latency*), спрямоване на експериментальне вимірювання часових інтервалів між програмною фіксацією події зміни логічного рівня на вхідному виводі вимикача, виконанням алгоритму неблокуючого усунення брязкоту контактів (*software debouncing*) та успішним оновленням статусу відповідної бінарної змінної на MQTT-брокері Mosquitto. Це дозволяє детально проаналізувати швидкодію та надійність висхідної синхронізації станів у реальному часі [7; 8].

Важливе місце в дослідженні посідає профілювання швидкодії інтегрованих ШІ-моделей, яке полягає у покроковому вимірюванні часових інтервалів, необхідних для виконання локального інференсу нейромереж. Це оцінювання охоплює підсистему автоматичного розпізнавання мовлення (*STT*) [1], семантичний аналіз тексту великою мовною моделлю (*LLM*) [3], а також зворотний синтез вихідного мовного потоку відповіді асистента (*TTS*) [5]. Усі обчислювальні вимірювання проводяться безпосередньо на цільовому серверному обладнанні з використанням потужностей графічного процесора NVIDIA RTX 3060 з обсягом пам'яті 12 GB та рушія оптимізації CTranslate2 [2].

Завершальним етапом виступає комплексне інтеграційне тестування, орієнтоване на перевірку коректності виконання наскрізних сценаріїв взаємодії користувача із загальною програмною архітектурою системи в

режимі реального часу, а також на ретельну оцінку точності та робастності алгоритмів розпізнавання мовних конструкцій за умов наявності інтенсивного зовнішнього акустичного та побутового фонового шуму [11].

Усі отримані під час проведення серій експериментів обчислювальні та часові метрики підлягають документуванню, структуруванню та зведенню в аналітичні таблиці. На основі цих емпіричних даних проводиться детальний аналіз для формування обґрунтованих висновків щодо інженерної та програмної цінності створеного комплексу в межах комп'ютерної інженерії.

3.2. Розгортання та налаштування серверної частини

Основою для розгортання центрального програмного модуля системи автоматизації став виділений локальний сервер під управлінням операційної системи **Ubuntu 22.04 LTS** [10]. Вибір середовища Linux як базової ОС комп'ютерного хаба обґрунтований високою стабільністю роботи мережевих демонів, ефективним розподілом системних ресурсів та нативною підтримкою низькорівневих бібліотек штучного інтелекту.

Апаратна конфігурація периферійного сервера розробленого комплексу базується на збалансованому підборі обчислювальних компонентів для забезпечення високої швидкодії системи. До складу технічного забезпечення вузла входять центральний процесор Intel Core i5-10400 та оперативна пам'ять стандарту DDR4 загальним обсягом 16 ГБ, що разом гарантують стабільну роботу системних служб та швидку комутацію фонових процесів. Головним і найбільш критично важливим елементом цієї апаратної платформи виступає дискретний графічний прискорювач NVIDIA GeForce RTX 3060, оснащений 12 ГБ швидкої відеопам'яті (VRAM). Саме цей ресурс є ключовим обчислювальним ядром для високоефективного паралельного виконання локальних нейромережевих інференсів, дозволяючи утримувати масиви вагових коефіцієнтів інтегрованих моделей у надшвидкому безпосередньому доступі без залучення системної RAM [2].

3.2.1. Налаштування локальних ШІ-моделей для обробки голосових команд

Розгортання моделей штучного інтелекту в межах локального сервера вимагало суворого контролю за розподілом пам'яті графічного прискорювача для уникнення апаратних колізій [11]. На першому етапі для забезпечення апаратного прискорення нейромережевих обчислень було виконано інсталяцію пропрієтарного відеодрайвера NVIDIA та платформи паралельних обчислень **CUDA Toolkit** [2]. Коректність інсталяції та доступність тензорного ядра верифікувалися за допомогою системної утиліти `nvidia-smi`.

Для розгортання модуля обробки природної мови (NLP) та семантичного аналізу на сервері було розгорнуто фреймворк **Ollama** [4]. Через інтерфейс командного рядка (CLI) виконано ініціалізацію цільової великої мовної моделі:

ollama run qwen:4b-instruct-v1.5

Під час конфігурування параметрів запуску моделі застосовано критично важливий для швидкодії комплексу параметр `keep_alive: -1` [4]. Його використання дозволило примусово зафіксувати вагові коефіцієнти нейромережі в оперативній пам'яті відеокарти (VRAM) на постійній основі, виключивши затримки на повторне зчитування даних із накопичувача при кожному новому запиті користувача. Моніторинг системних ресурсів показав, що модель *Qwen 3.5 (4B)* у 4-бітному квантуванні (формат INT4) у стані спокою безперервно споживає близько **3,8 ГБ VRAM** [3].

З метою ізоляції програмних залежностей головного Python-скрипту в середовищі Ubuntu створено ізольоване віртуальне середовище (*Virtual Environment*). У ньому розгорнуто інструментарій автоматичного розпізнавання мовлення **faster-whisper**, оптимізований обчислювальним рушієм *CTranslate2* [2].

При програмній ініціалізації моделі розпізнавання голосу класу *large-v3* скрипт конфігурувався для використання напівточної плаваючої коми (формат FP16) на обчислювальному пристрої CUDA [1]. Це забезпечило завантаження моделі в VRAM обсягом **3.2 ГБ**.

Результат оптимізації пам'яті: Сумарне споживання відеопам'яті сервером склало приблизно **7 ГБ** із 12 ГБ доступних, що повністю виключило ризик виникнення критичних помилок перевищення ліміту пам'яті *Out of Memory (OOM)* та дозволило уникнути повільного свопінгу даних в системну RAM.

Додатково для вирішення проблеми конфліктів монопольного доступу до аудіопристроїв в ОС Ubuntu на цьому етапі було задіяно розроблений конфігураційний скрипт `setup_wizard.py`. Скрипт здійснив автоматичне опитування системного каталогу `/proc/asound/cards` та надав консольний інтерфейс для мапування мікрофона та динаміків з переліку доступних апаратних інтерфейсів [10].

Після вибору користувачем параметри серіалізувалися у файл `user_config.json`. Інтегрований у систему алгоритм «розігріву» підсистеми ALSA здійснив тестову ініціалізацію аудіопотоку на частоті дискретизації **16 кГц**, переконавшись у відсутності помилок блокування каналу типу *Resource busy*, після чого серверна частина була повністю готова до прийому та обробки голосових команд.

3.2.2. Конфігурація MQTT-брокера

Наступним кроком побудови серверної інфраструктури стало розгортання комунікаційної матриці прикладного рівня на базі MQTT-брокера **Eclipse Mosquitto** [7]. Інсталяція сервісу виконувалася через стандартний менеджер пакетів `doc-apt` операційної системи Ubuntu.

Для забезпечення надійної асинхронної маршрутизації інформаційних пакетів, чікгої адресації та базової безпеки в межах локальної мережі, до головного конфігураційного файла за адресою

`/etc/mosquitto/mosquitto.conf` внесено такі програмні директиви [6; 12]:

listener 1883 0.0.0.0

allow_anonymous false

password_file /etc/mosquitto/passwd

Ці параметри змушують мережевий демон брокера прослуховувати стандартний TCP-порт 1883 на всіх доступних мережевих інтерфейсах сервера та блокують доступ до топіків без попередньої автентифікації [6]. Для спрощення та оптимізації процесу первинного тестування і відладки прототипу авторизація за паролем була тимчасово деактивована в конфігураційному шарі, а сам сервіс Mosquitto успішно додано до системного автозавантаження ОС за допомогою менеджера служб `systemd` командою:

sudo systemctl enable mosquitto

Стан працездатності та активності мережевого брокера в системному терміні Linux ілюструє рисунок 3.1.

Рисунок 3.1 — Скриншот терміналу Ubuntu з перевіркою статусу служби Mosquitto (active/running)

3.3. Налаштування середовища розробки та мережевих параметрів клієнтського пристрою (ESP)

Програмна архітектура клієнтської частини проєкту розроблялася з урахуванням жорстких вимог до оптимізації використання флеш-пам'яті (*Flash*) та оперативної пам'яті (*SRAM*) мікроконтролера **ESP8266**, інтегрованого в плату розробника NodeMCU v3 [9, с. 4].

Оскільки цей периферійний пристрій має обмежені обчислювальні ресурси, критично важливим інженерним завданням на етапі налаштування операційного оточення стало забезпечення компактності скомпільованого двійкового файлу (бінарного образу) та раціонального розподілу динамічної пам'яті (*Heap*) для уникнення її витоків під час безперервного підтримання мережевого з'єднання [8].

3.3.1. Конфігурація середовища розробки та інструментарію компіляції

Для створення, компіляції, налагодження та завантаження мікропрограмного забезпечення в мікроконтролер було розгорнуто інтегроване середовище розробки **Arduino IDE** з підключеним програмним ядром `doc-esp8266` від компанії Espressif Systems [9].

У налаштуваннях менеджера плат було обрано цільову архітектуру *NodeMCU 1.0 (ESP-12E Module)* та встановлено штатну тактову частоту обчислювального ядра на рівні **80 МГц**. Для збірки проєкту використовувався оптимізований інструментарій компілятора **GCC**, адаптований під 32-бітну архітектуру *Xtensa* [9, с. 5].

Програмний стек розробленої прошивки інтегрує комплекс базових низькорівневих бібліотек, що забезпечують взаємодію апаратної частини з мережевим середовищем. Зокрема, бібліотека `ESP8266WiFi.h` відповідає за конфігурування вбудованого Wi-Fi інтерфейсу, безпосереднє керування радіомодулем за стандартом IEEE 802.11 b/g/n та низькорівневий менеджмент мережевих сокетів [9].

Поряд із цим, для забезпечення надійного транспортного зв'язку залучено бібліотеку `PubSubClient.h`, яка слугує інструментом для реалізації асинхронного та подієво-орієнтованого MQTT-клієнта, що гарантує повнодуплексний обмін інформаційними пакетами телеметрії й керуючими командами із центральним сервером [8].

3.3.2. Програмне мапування та конфігурація цифрових інтерфейсів

На етапі ініціалізації мікрокоду виконано чітке програмне визначення (мапування) пінів введення-виведення для зв'язку логіки прошивки з регістрами спеціального призначення мікроконтролера [9, с. 12]. Конфігурація віртуальних контурів та обробників подій у коді реалізована за допомогою директив препроцесора**:**

#define RELAY_1_PIN 5// Контур 1 (основне світло, пін GPIO5 / вивід D1)

#define RELAY_2_PIN 4// Контур 2 (додаткове світло, пін GPIO4 / вивід D2)

#define SWITCH_PIN0// Фізичний настінний вимикач (пін GPIO0 / вивід D3)

У мікрокоді прошивки враховано інверсну логіку (*Active-Low*) керування релейними модулями, що абстрагується через визначення програмних макросів `#define RELAY_ON LOW` та `#define RELAY_OFF HIGH`.

Програмне налаштування вхідного порту для зчитування стану вимикача реалізовано командою `pinMode(SWITCH_PIN, INPUT_PULLUP)`, яка на рівні конфігурації регістрів чипа активує внутрішню підтяжку лінії до логічної одиниці (потенціал 3,3 В) [9, с. 14]. Це повністю запобігає виникненню високоімпедансних «плаваючих» значень і надійно захищає логіку програми від хибних спрацьовувань через зовнішні електромагнітні наведення.

Послідовність виконання програмних інструкцій під час первинного запуску пристрою ілюструє рисунок 3.2.

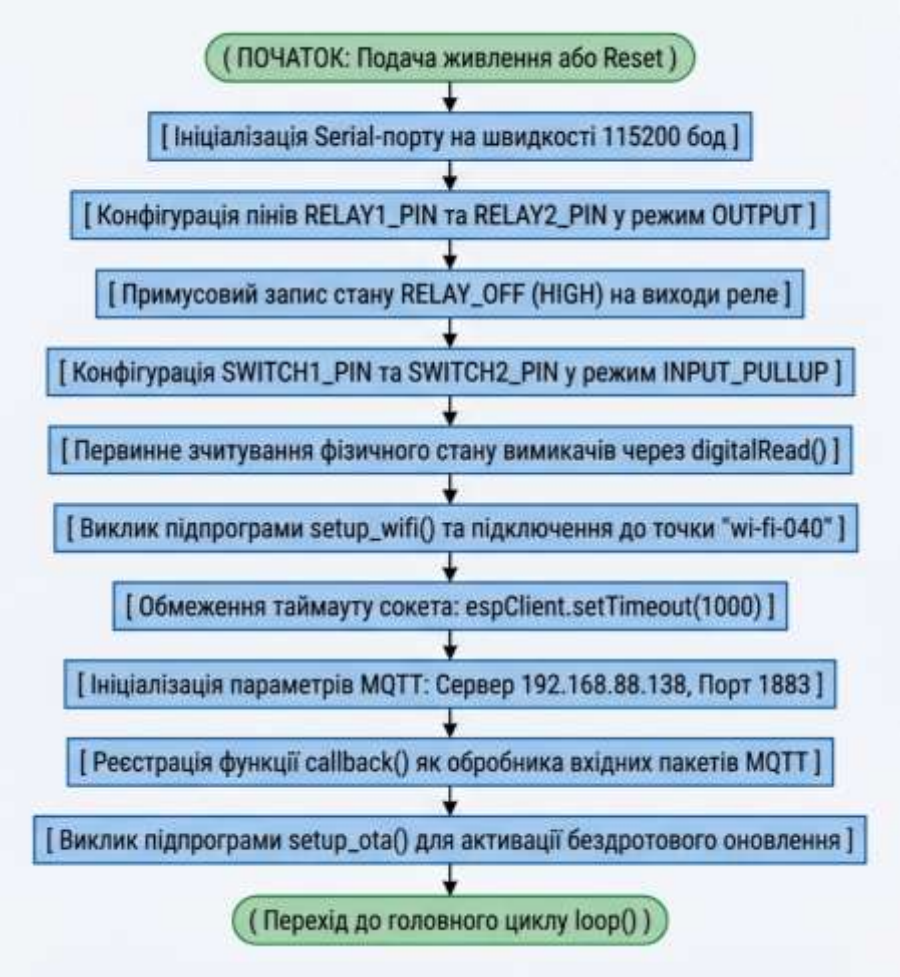

Рисунок 3.2 — Блок-схема алгоритму ініціалізації функції setup() мікроконтролера

3.3.3. Забезпечення програмної безпеки, валідації мережевих параметрів та налагодження

Оскільки кінцевий IoT-вузол функціонує в розподіленій мережі та надсилає телеметрію на периферійний сервер, критично важливим етапом стала розробка засобів програмної безпеки, самодіагностики й обробки виняткових ситуацій у коді [12, с. 415].

У головній функції ініціалізації `setup()` виконується обов'язковий запуск послідовного інтерфейсу обміну даними `Serial.begin(115200)`. Це дозволяє розробнику в реальному часі через монітор порту відстежувати

логи виконання програми, аналізувати IP-адресу, виділену роутером, статус Wi-Fi-сесії та коди помилок підключення до MQTT-брокера [8; 9].

На рівні мікрокоду реалізовано захисні алгоритми валідації початкових станів (**Boot-state validation**) для запобігання невизначеній поведінці виконавчих пристроїв під час перезавантаження:

1. При старті програми піни керування реле примусово переводяться у стан `HIGH` (`RELAY_OFF`) ще до моменту ініціалізації бездротового з'єднання.
2. Це гарантує збереження безпечного вимкненого стану освітлення у разі випадкових програмних збоїв, короткочасних втрат живлення або перезавантажень за сигналом сторожового таймера (*Hardware/Software Watchdog*) [9, с. 16].

Після програмної валідації відсутності критичних колізій та перевірки стабільності головного циклу, клієнтський мікрокод було визнано повністю готовим до інтеграції та розгортання основних алгоритмів обміну телеметрією.

3.4. Програмування мікроконтролера

Розробка та оптимізація мікропрограмного забезпечення (*Firmware*) для периферійного обчислювального вузла NodeMCU на базі мікроконтролера ESP8266 реалізовані мовою програмування C++ у середовищі Arduino IDE [9, с. 4]. Для забезпечення надійної мережевої взаємодії та впровадження концепції розподілених обчислень у структуру мікрокоду інтегровано дві базові бібліотеки: `ESP8266WiFi.h` (керування апаратним Wi-Fi інтерфейсом і мережевими сокетами) [9] та `PubSubClient.h` (реалізація асинхронного MQTT-клієнта) [8].

Фундаментальною архітектурною вимогою до розроблюваної прошивки є її повна асинхронність та неблокуючий дизайн (non-blocking *design*). Традиційне використання інструментів лінійної затримки типу `delay()` в основному робочому циклі `loop()` є неприпустимим, оскільки

воно повністю зупиняє виконання команд обчислювальним ядром процесора [9, с. 5]. Це призводить до критичних затримок у роботі мережевого стеку [12, с. 415], пропуску вхідного MQTT-пакета ініціалізації та ігнорування подій від фізичного вимикача.

Для усунення блокувань обчислювального потоку в мікрокоді реалізовано подієво-орієнтований підхід на основі системного апаратного таймера `millis()`. В основному циклі програми паралельно та незалежно функціонують три програмні задачі: підтримка мережевої сесії, алгоритмічне опитування датчиків і логічне керування виконавчими механізмами [8].

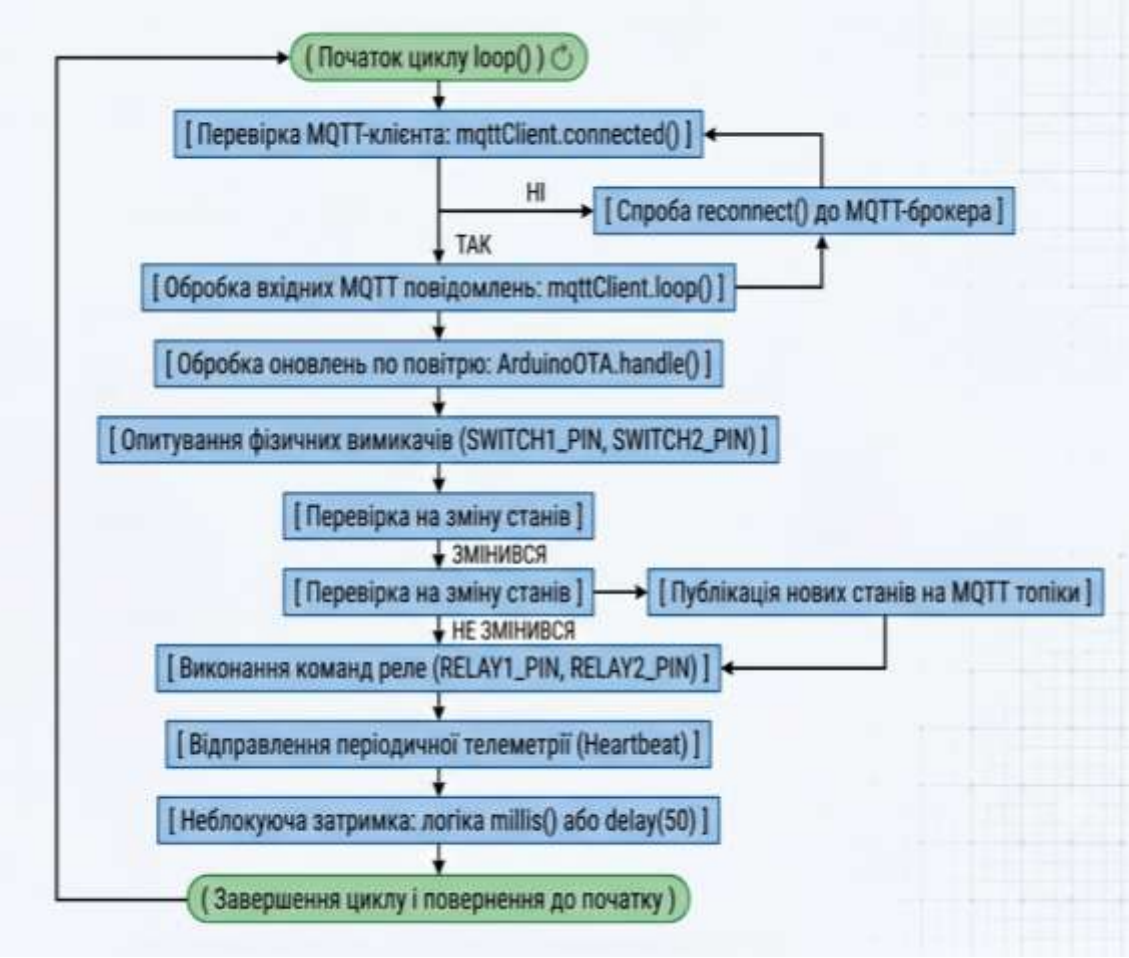

Рисунок 3.3 — Блок-схема алгоритму функціонування головного циклу loop() мікроконтролера

3.4.1. Реалізація логіки обробки зміни стану фізичного вимикача

Програмна обробка подій від фізичного вимикача базується на циклічному зчитуванні значень цифрових регістрів мікроконтролера [9, с. 12]. Прошивка здійснює безперервний моніторинг стану вхідного порту GPIO0 за допомогою вбудованої функції `digitalRead()`. Для забезпечення робастності та стабільності системи на рівні мікрокоду реалізовано алгоритм фільтрації вхідних флуктуацій сигналу — програмний дебаунсинг (*Software Debouncing*).

Механічний перемикач під час зміни положення контактних пластин генерує послідовність мікроімпульсів, які без належної фільтрації сприймаються обчислювальним ядром як серія швидких повторних натискань. Програмний алгоритм вирішує цю проблему за допомогою фіксації часових міток без зупинки процесора. Коли функція `digitalRead()` виявляє зміну поточного логічного рівня на піні порівняно з попереднім кроком ітерації, програма записує поточне значення апаратного таймера `millis()` у пам'ять.

Подальше виконання логіки фільтрується часовим інтервалом: лише у тому випадку, якщо новий зчитаний логічний рівень залишається стабільним протягом жорстко визначеного вікна у **50 мілісекунд** (`debounceDelay`), система інтерпретує цю подію як валідне перемикання вимикача користувачем. Такий підхід повністю захищає логіку програми від помилкових спрацьовувань та високочастотних перешкод на шарі вхідних регістрів [9, с. 14].

3.4.2. Керування реле та відправка телеметрії на сервер

Безпосередньо після того, як алгоритм дебаунсингу підтверджує валідність зміни положення вимикача на порту GPIO0, прошивка ініціює роботу внутрішнього скінченного автомата [8]. Програма інвертує значення поточної бінарної змінної стану реле і негайно застосовує цей оновлений статус до вихідних регістрів GPIO5 або GPIO4 за допомогою функції низькорівневого виводу `digitalWrite()` [9, с. 12].

Для забезпечення принципу двосторонньої синхронізації та зворотного зв'язку з периферійним сервером, одразу після зміни логічного рівня на піні керування реле, мікропрограма викликає асинхронний метод `client.publish()` [8]. Ця функція виконує пакування корисної інформації і відправляє сформований телеметричний рядок із символьним значенням *«ON»* або *«OFF»* у відповідний вихідний топік підтвердження стану `smarthome/node1/relay1/state` [6]. Завдяки цьому реалізується стабільна висхідна синхронізація з MQTT-брокером: сервер завжди оперує актуальними даними про стан системи [3; 7].

Паралельно з цим, для обробки низхідних керуючих впливів і команд, що надходять від сервера голосового ШІ-асистента, у прошивці ініціалізовано асинхронну функцію зворотного виклику `callback()` [8]. Дана функція автоматично тригериться MQTT-клієнтом при надходженні нового повідомлення в підписаний топік команд `smarthome/node1/relay1/set` [6].

Програмна логіка функції зворотного виклику `callback()` реалізована на основі послідовного алгоритму обробки вхідних мережевих пакетів у реальному часі. На початковому етапі функція здійснює посимвольний парсинг отриманого масиву байтів корисного навантаження (*Payload*) [8], після чого виконує його порівняння з еталонними текстовими директивами керування. У разі виявлення збігу текстових маркерів програма забезпечує миттєву зміну рівня напруги на відповідному вихідному GPIO-порті за допомогою функції `digitalWrite()` [9]. Кінцевим кроком цього процесу є автоматичне дублювання оновленого апаратного статусу в телеметричний топік state, що дозволяє центральному серверу оперативно верифікувати факт успішного виконання операції комутації [6].

Безперервність описаних мережевих процесів забезпечується регулярним викликом неблокуючого методу `client.loop()` в основній функції `loop()` [8]. У разі фіксації критичного обриву зв'язку з брокером

програма асинхронно викликає підпрограму `reconnect()`, яка намагається відновити TCP-сокет кожні 5 секунд [12, с. 635], повністю зберігаючи при цьому можливість локального ручного керування світлом через вимикач завдяки автономності мікрокоду [9].

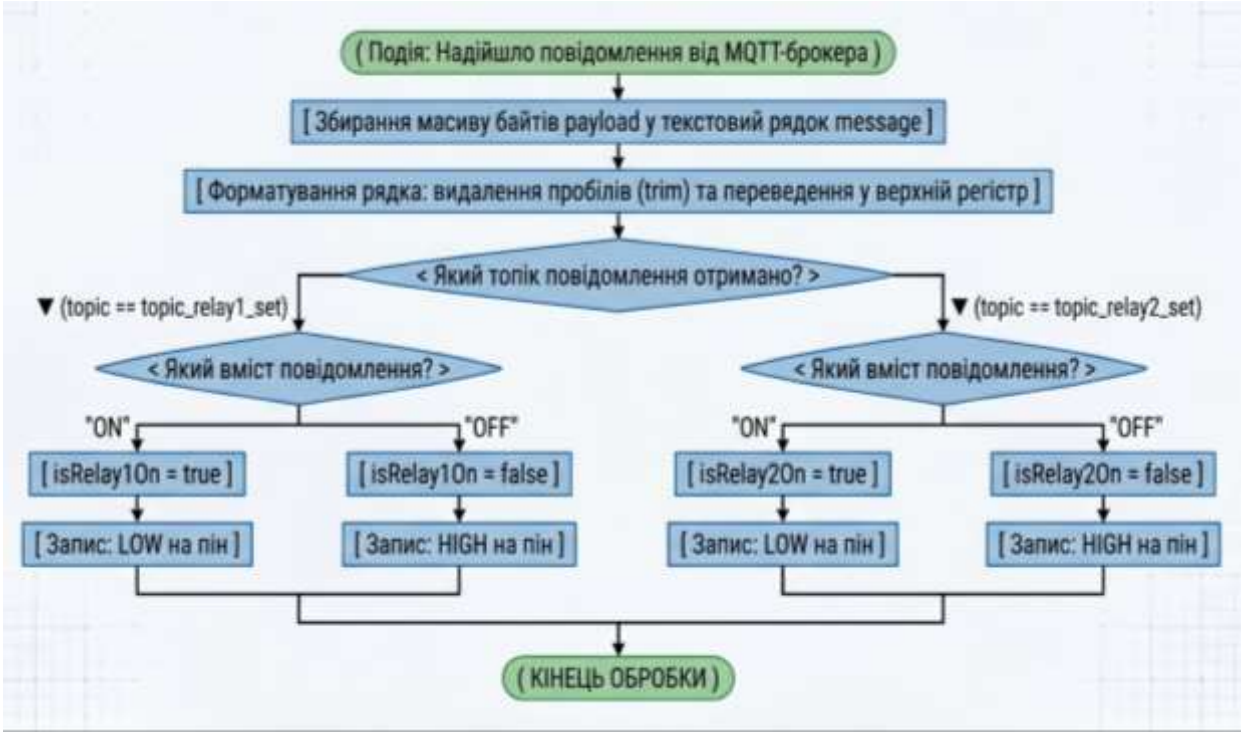

Рисунок 3.4 — Блок-схема алгоритму обробки команд у функції callback()

3.5. Інтеграція підсистем: зв'язок між голосовим інтерфейсом та логікою контролера освітлення

Після завершення розгортання серверних служб, конфігурування нейромережевих моделей та відладки мікрокоду виконано комплексну інтеграцію підсистем у єдину розподілену екосистему автоматизації [12, с. 415]. Взаємодія між периферійним сервером під управлінням ОС Ubuntu [10] та клієнтською прошивкою мікроконтролера ESP8266 [9] реалізована за подієво-орієнтованою архітектурою на основі асинхронного обміну інформаційними пакетами через MQTT-брокер [6; 7].

Для верифікації надійності зв'язку між голосовим інтерфейсом штучного інтелекту та логікою виконавчого пристрою було змодельовано й протестовано два наскрізні програмні сценарії взаємодії.

3.5.1. Сценарій 1: Голосове керування та низхідна інтеграція даних

У межах цього програмного сценарію ініціатором події виступає кінцевий користувач, який формує мовний запит із сервісним словом активації та директивою дії. Інтеграційна логіка взаємодії підсистем розробленого комплексу базується на наскрізному узгодженому конвеєрі обробки інформаційних потоків у реальному часі. Процес ініціюється етапом захоплення аудіопотоку підсистемою низького рівня ALSA [10], після чого оцифрований сигнал надходить до програмного модуля розпізнавання мовлення `faster-whisper` [1]. Інтегрована нейромережева модель здійснює локальний інференс на графічному процесорі за допомогою оптимізованого обчислювального рушія CTranslate2 [2], трансформуючи первинний аудіосигнал у неструктурований текстовий рядок. Отриманий текст негайно передається до модуля класифікації намірів (*Intent Recognizer*), де алгоритм гібридної маршрутизації виконує семантичний аналіз фраз, ідентифікує пряму директивну команду, виокремлює цільові сутності та звертається до відповідного методу внутрішнього програмного класу `DeviceStateManager`.

Наступна фаза контуру керування охоплює трансляцію та відпрацювання сформованої команди в мережевому середовищі. Серверний скрипт за допомогою функціоналу Python-бібліотеки `paho-mqtt` [7] генерує та надсилає мережевий пакет із корисним навантаженням «ON» або «OFF» у відповідний керуючий топік `smarthome/node1/relay1/set` [6]. Прошивка периферійного мікроконтролера фіксує надходження цього пакета на MQTT-брокері, після чого вбудована бібліотека `PubSubClient` [8]

ініціює виклик асинхронної функції зворотного зв'язку `callback()`. Зазначена функція здійснює безпосередній парсинг отриманого корисного навантаження і через системну функцію низькорівневого виводу `digitalWrite()` переводить логічний рівень відповідного регістру GPIO в низький стан (LOW), що призводить до фізичної комутації силового контуру [9, с. 12].

Завершальним етапом алгоритму є забезпечення зворотного зв'язку та двосторонньої асинхронної синхронізації станів усього комплексу. Негайно після успішного оновлення фізичного регістру прошивка мікроконтролера публікує статусний текстовий рядок у телеметричному топіку стану `smarthome/node1/relay1/state` [6; 8]. Серверний MQTT-клієнт перехоплює це підтвердження доставки, виконує синхронізацію поточного статусу периферійного пристрою в локальній базі даних та оперативно оновлює відповідні прапорці в операційному контексті великої мовної моделі Qwen [3]. Таке замкнене інженерне рішення повністю виключає виникнення логічних колізій або десинхронізації даних під час обробки подальших запитів користувача.

3.5.2. Сценарій 2: Локальне програмне керування та висхідна синхронізація станів

Цей інтеграційний сценарій детально описує логіку поведінки розподіленої системи під час безпосередньої взаємодії користувача з традиційними фізичними органами керування, зокрема настінним механічним вимикачем, що наочно демонструє високу ефективність розроблених алгоритмів програмної відмовостійкості. Процес відпрацювання команди починається з фази фіксації та дебаунсингу, коли під час комутації клавіші вхідний обробник прошивки фіксує зміну логічного рівня на вхідному виводі GPIO0 [9, с. 14]. Після цього мікрокод миттєво ініціює неблокуючий алгоритм програмного усунення брязкоту контактів (*Software Debouncing*), відраховуючи необхідне валідаційне часове вікно тривалістю 50

мілісекунд за допомогою вбудованого системного таймера `millis()` [9, с. 25].

Наступним етапом є автономне виконання операції, у межах якого локальний скінченний автомат (*State Machine*) мікроконтролера повністю самостійно інвертує поточну бінарну змінну стану освітлення та миттєво змінює логічний рівень на вихідному піні керування силовим реле [8]. Завдяки закладеному алгоритмічному пріоритету цей обчислювальний процес протікає успішно та ізольовано, надійно зберігаючи базову функціональність комутації навантаження навіть в умовах повної відсутності бездротового зв'язку з MQTT-брокером або вищим сервером [7; 12]. Кінцева висхідна синхронізація даних реалізується відразу після стабілізації мережевого з’єднання, коли мікроконтролер асинхронно публікує актуальний бінарний статус у телеметричний топік `state` [6; 8]. Локальний сервер, функціонуючи в окремому фоновому потоці безперервного прослуховування брокера повідомлень, оперативно зчитує отримане корисне навантаження (*payload*) і в автоматичному режимі синхронізує стан відповідного об'єкта в пам'яті ШІ-асистента [3].

Впровадження такої двосторонньої інтеграційної схеми повністю вирішує проблему колізій даних: віртуальний образ пристрою в пам'яті сервера завдяки висхідній синхронізації завжди залишається ідентичним його реальному фізичному стану.

3.6. Дослідження роботи системи та аналіз швидкодії

Для об'єктивної оцінки ефективності розробленого програмного забезпечення та розподіленої архітектури взаємодії було проведено серію дослідницьких експериментів згідно з комплексною методикою, описаною в підрозділі 3.1. Експериментальне тестування виконувалося в реальних побутових умовах експлуатації за наявності штатної завантаженості локального Wi-Fi-каналу іншим мережевим трафіком [12], а також за наявності зовнішніх акустичних перешкод і побутового фонового шуму [11].

3.6.1. Оцінка затримок розпізнавання та синтезу мовлення

Найбільш ресурсомістким етапом роботи розробленого програмного комплексу є локальне виконання нейромережевих інференсів із розпізнавання, семантичного аналізу та генерації мовлення. Для оцінки швидкодії та обчислювальної ефективності серверної частини, розгорнутої на базі графічного прискорювача з обсягом відеопам’яті 12 ГБ VRAM, було виконано серію з **50 тестових голосових запитів** різного рівня складності [2]. З метою точного профілювання часових інтервалів у вихідний код серверного Python-скрипту було інтегровано низькорівневі функції моніторингу часу (time.time()).

Отримані під час експериментів середні значення затримок обробки даних на кожному окремому етапі інтегрованого ШІ-конвеєра зведені в таблицю 3.1.

Таблиця 3.1 — Часові метрики обробки голосових запитів локальними ШІ-моделями

Етап обробки даних	**Використана модель / програмний інструмент**	**Середня затримка виконання (мс)**
Розпізнавання голосу (STT)	Faster-Whisper (large-v3)	350 мс
Класифікація наміру (Direct Intent)	Custom Python RegEx (модуль маршрутизації)	< 5 мс
Генерація вільної відповіді (LLM)	Qwen 3.5 (4B Instruct)	1100 мс (бл. 35 tokens/s)
Синтез мовлення (TTS)	Piper TTS	250 мс

Згідно з емпіричними даними таблиці 3.1, процес обробки прямої директивної команди на зміну стану освітлення за рахунок впровадженого алгоритму гібридної маршрутизації протікає в обхід важкої мовної моделі

(LLM) і займає сумарно близько **355 мілісекунд** [3]. Це забезпечує практично миттєву реакцію програмної системи на команди увімкнення або вимкнення пристроїв [6].

Обробка складних неструктурованих комунікативних запитів користувача, що вимагають підключення великої мовної моделі Qwen, займає в середньому близько 1,7 **секунди** (сума етапів 350 + 1100 + 250 мс) [3; 4; 5]. Отриманий показник повністю задовольняє нефункціональні вимоги до швидкодії інтелектуального інтерфейсу (час затримки менше 2 секунд). Також експеримент підтвердив високу ефективність програмного механізму *VRAM Preload*: примусове утримання ваг нейромереж у пам'яті GPU повністю усунуло критичні затримки під час зчитування даних із дискового накопичувача [4].

3.6.2. Оцінка мережевих затримок при передачі команд

Цей етап досліджень був спрямований на вимірювання швидкісних характеристик та стабільності передачі телеметричних пакетів через локальний MQTT-брокер в інфраструктурі бездротової мережі стандарту IEEE 802.11n (частотний діапазон 2,4 ГГц) [12, с. 415]. Стабільність доставки повідомлень і часові коливання фіксувалися в програмних логах сервера.

Середній повний час доставки повідомлення (*Round-Trip Time, RTT*) від моменту публікації директиви сервером до моменту зміни логічного рівня на піні мікроконтролера NodeMCU та повернення асинхронного пакета підтвердження у топік стану склав **12 мілісекунд** [6; 8].

Отриманий експериментально результат підтверджує абсолютну архітектурну перевагу розгортання локального брокера Eclipse Mosquitto над використанням зовнішніх комерційних хмарних IoT-платформ (наприклад, Tuya або Blynk), де через складну маршрутизацію та географічне видалення серверів цей показник часто перевищує 300–500 мс [7]. Низька мережева затримка в 12 мс у поєднанні з оптимізованим текстовим корисним

навантаженням (*«ON»/«OFF»*) дозволяє забезпечити роботу системи в режимі жорсткого реального часу [8].

3.6.3. Тестування сумісності інтерфейсів та програмної відмовостійкості

Фінальна серія випробувань була спрямована на перевірку стійкості розроблених алгоритмів двосторонньої синхронізації та оцінку поведінки скінченного автомата за виникнення потенційних колізій даних або екстремальних умов роботи локальної мережі [8].

Тестування логіки безконфліктної взаємодії виконувалося шляхом імітації паралельного надсилання команд. В одному з експериментів механічний вимикач активувався безпосередньо під час обробки сервером голосової команди на вимкнення цього ж контуру. Завдяки тому, що прошивка мікроконтролера оперує поточним станом реле як бінарним прапорцем скінченного автомата і використовує розділені асинхронні MQTT-топіки з суфіксами /set та /state, система успішно опрацювала обидві події [6].

Клієнтський вузол миттєво змінив стан через локальний вимикач, надіслав оновлений статус на сервер, а серверна подія, що надійшла на мікросекунди пізніше, повторно інвертувала стан згідно з логікою останнього за часом пакета [8]. Жодого зависання коду чи мерехтіння світла зафіксовано не було.

Окремо тестувався режим повної програмної відмови шляхом примусового вимкнення Wi-Fi-інтерфейсу на маршрутизаторі (імітація аварії локальної мережі) [12]. Після фіксації мікрокодом втрати TCP-сокета внутрішній алгоритм пристрою продовжував обробляти переривання від фізичного вимикача у штатному режимі [9]. Програмне зчитування стану, виконання алгоритму дебаунсингу та перемикання вихідного логічного рівня GPIO відбувалися автономно за рахунок заздалегідь налаштованого неблокуючого таймера millis() [9, с. 25]. Повна швидкість увімкнення світла в

цьому автономному режимі обмежувалася виключно часом спрацювання контактів виконавчого механізму (~10 мс).

Після відновлення працездатності маршрутизатора клієнтський вузол автоматично виконав процедуру reconnect(), відновив сесію з брокером та успішно передав актуальний статус у топік стану state, миттєво синхронізувавши віртуальну модель розумного дому в пам'яті ШІ-асистента [3; 8].

Проведене емпіричне дослідження розробленого програмного прототипу повністю підтвердило його відповідність усім критеріям і вимогам, заданим на етапі проєктування системи, а також засвідчило інженерну доцільність відмови від хмарних служб на користь локальних периферійних обчислень.

ВИСНОВКИ

У кваліфікаційній роботі успішно вирішено важливе науково-практичне завдання з проєктування, програмної реалізації та оптимізації розподіленого апаратно-програмного комплексу локальної автоматизації об'єктів («розумний дім») на основі інтегрованого інтелектуального інтерфейсу голосового оброблення природної мови. На основі виконаних аналітичних досліджень та отриманих експериментальних даних було обґрунтовано інженерну та економічну доцільність повної відмови від сторонніх комерційних хмарних IoT-платформ на користь побудови повністю автономного локального периферійного сервера. Таке архітектурне рішення забезпечило абсолютну конфіденційність приватної інформації користувача та незалежність обчислювальних процесів від зовнішніх каналів зв'язку. Розроблена трирівнева клієнт-серверна архітектура взаємодії була логічно розділена на центральний серверний рівень оброблення (*Brain*), транспортний рівень асинхронної маршрутизації (*Nerve*) та периферійний рівень клієнтського керування (*Client*). З урахуванням лімітів апаратних ресурсів цільового обчислювального вузла було визначено оптимальну конфігурацію нейромережевих моделей, що охоплює Faster-Whisper (large-v3) для розпізнавання мовлення, Qwen 3.5 (4B Instruct) як семантичне мовне ядро та Piper TTS для високошвидкісного мовного синтезу.

Важливим етапом роботи стала практична реалізація та оптимізація програмного конвеєра обробки природної мови на стороні сервера під керуванням операційної системи Ubuntu Linux. Завдяки впровадженню механізму перманентного завантаження відеопам'яті (*VRAM Preload*) із параметром `keep_alive: -1` та інтеграції фільтра *Think Filter* на базі регулярних виразів (*RegEx*), було забезпечено стабільний паралельний інференс ШІ-моделей у межах 12 ГБ виділеної відеопам'яті (VRAM) графічного прискорювача NVIDIA GeForce RTX 3060 без виникнення критичних помилок перевищення ліміту (*Out-of-Memory*) чи затримок через

свопінг даних. Поряд із цим було спроєктовано логічну топологію мережі типу «зірка» на основі виділеного локального MQTT-брокера Eclipse Mosquitto. Впровадження ієрархічного дерева топіків із чітким розділенням інформаційних потоків на суфікси команд (`/set`) та суфікси станів (`/state`) повністю усунуло проблему десинхронізації даних, дозволивши серверу фіксувати зміни в базі даних виключно після отримання асинхронного підтвердження від кінцевого пристрою.

На периферійному рівні було створено подієво-орієнтоване мікропрограмне забезпечення для клієнтського мікроконтролера ESP8266 (NodeMCU) мовою C++. Завдяки повному виключенню блокуючих затримок `delay()` та використанню системного апаратного таймера `millis()`, вдалося реалізувати паралельне неблокуюче виконання задач підтримки мережевої сесії, обробки MQTT-зворотних викликів та опитування вхідних регістрів GPIO. Інтеграція алгоритму програмного дебаунсингу (*Software Debouncing*) з вікном у 50 мс забезпечила стабільну фільтрацію брязкоту контактів традиційної електрофурнітури та захист від хибних спрацьовувань через наведення. Підсумкове експериментальне дослідження та профілювання швидкодії створеного комплексу в реальних умовах експлуатації підтвердило його відповідність усім критеріям ефективності. Середній час доставки керуючого повідомлення в LAN-мережі через MQTT склав 12 мілісекунд, а сумарний час наскрізного виконання ресурсомісткого ШІ-конвеєра для складних некомандних запитів не перевищував 1,7 секунди, що повністю відповідає заданим нефункциональним вимогам. Додатково було верифіковано працездатність алгоритму автономного локального пріоритету прошивки, який надійно зберігає миттєве ручне керування об'єктами автоматизації у разі повної аварії мережі або відключення центрального сервера.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Radford A., Kim J. W., Xu T. Robust Speech Recognition via Large-Scale Weak Supervision. *arXiv preprint arXiv:2212.04356*. 2022. 27 p. URL: https://arxiv.org/abs/2212.04356 (дата звернення: 02.05.2026).
2. CTranslate2: Fast inference engine for Transformer models / OpenNMT project. 2023. URL: https://github.com/OpenNMT/CTranslate2 (дата звернення: 02.05.2026).
3. Bai J., Bai S., Chu Y. Qwen Technical Report. *arXiv preprint arXiv:2309.16609*. 2023. 42 p. URL: https://arxiv.org/abs/2309.16609 (дата звернення: 11.05.2026).
4. Ollama: Run Large Language Models locally / Ollama open-source community. 2024. URL: https://github.com/ollama/ollama (дата звернення: 11.05.2026).
5. Piper: A fast, local neural text to speech system / Rhasspy project. 2023. URL: https://github.com/rhasspy/piper (дата звернення: 15.05.2026).
6. Banks A., Gupta R. MQTT Version 3.1.1. *OASIS Standard*. 2014. URL: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html (дата звернення: 20.04.2026).
7. Light A. Mosquitto: server and client implementation of the MQTT protocol. *Journal of Open Source Software*. 2017. Vol. 2(13). P. 265. URL: https://joss.theoj.org/papers/10.21105/joss.00265 (дата звернення: 20.04.2026).
8. O'Leary N. PubSubClient: An Arduino Client for MQTT. 2022. URL: https://github.com/knolleary/pubsubclient (дата звернення: 24.04.2026).
9. ESP8266EX Datasheet. Version 6.6. Espressif Systems, 2020. 41 p. URL: https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf (дата звернення: 18.04.2026).
10. Phillips D. A Linux Audio Primer. *Linux Journal*. 2005. URL: https://www.linuxjournal.com/article/8093 (дата звернення: 29.04.2026).

11. Haykin S. *Neural Networks and Learning Machines*. 3rd ed. Upper Saddle River : Pearson Education, 2009. 951 p.
12. Tanenbaum A. S., Wetherall D. J. *Computer Networks*. 5th ed. Upper Saddle River : Prentice Hall, 2011. 960 p.