Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

МЕТОД ТА ЗАСОБИ ВІДДАЛЕНОГО МОНІТОРИНГУ ІОТ ПРИСТРОЇВ

Виконав студент 2 курсу, групи 1КІ-24м
спеціальності 123 — Комп'ютерна інженерія
Степанчук Д.В.

Керівник к.т.н., доц. каф. ОТ
Богомолов С. В.
"15" 12 2025 р.

Опонент к.т.н, доц. каф. МБІС
Шиян А. А.
"15" 12 2025 р.

Допущено до захисту
Зав. каф. д.т.н., проф. Азаров О.Д.
"18" 12 2025 р.

ВНТУ 2025

ВІННИЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

Галузь знань — Інформаційні технології

Освітній рівень — магістр

Спеціальність — 123 Комп'ютерна інженерія

Освітньо-професійна програма — Комп'ютерна інженерія

ЗАТВЕРДЖУЮ

Завідувач кафедри ОТ

д.т.н., проф. Азаров О. Д.

"25" вересня 2025 р.

З А В Д А Н Н Я

НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ

Студенту Степанчуку Дмитру Валерійовичу

1 Тема роботи: "Метод та засоби віддаленого моніторингу IoT-пристроїв" керівник роботи: к.т.н., доцент кафедри ОТ Богомолов С. В. затверджена наказом Вінницького національного технічного університету від від 24.09.2025 року № 313.

2 Строк подання магістрантом роботи: 04.12.2025

3 Вихідні дані до роботи: Технічне завдання на розробку системи віддаленого моніторингу IoT-пристроїв. Доступне обладнання: мікроконтролер ESP32-WROOM-32, датчики DHT11 та MQ-4. Програмне забезпечення: MQTT-брокер Eclipse Mosquitto, Node.js для серверної частини, PostgreSQL для зберігання даних.

4 Зміст розрахунково—пояснювальної записки (перелік питань, які потрібно розробити): вступ, теоретичні основи систем віддаленого моніторингу IoT-пристроїв, аналіз методів та засобів реалізації системи моніторингу, розробка системи віддаленого моніторингу IoT-пристроїв, тестування та аналіз системи моніторингу, розрахунок економічної доцільності впровадження системи віддаленого моніторингу, висновки.

5 Перелік графічного матеріалу: діаграма алгоритму роботи пристрою, схе підключення компонентів.

6 Консультанти розділів роботи наведені в таблиці 1.

Таблиця 1 — Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1 — 4	Богомолов С. В., к.т.н., доцент каф. ОТ		
5	Ратушняк О. Г., к.т.н., доц. каф. ЕПВМ		
Нормоконтроль	Швець С. І., асист. каф. ОТ		

7. Дата видачі завдання 25.09.2025.

8. Календарний план виконання МКР наведено в таблиці 2.

Таблиця 2 — Календарний план

№ з/п	Назва етапів дипломної магістерської роботи	Строк виконання етапів роботи	Примітка
1	Огляд і аналіз джерел інформації	04.10.2025	Виконан
2	Теоретичні дослідження технологій	18.10.2025	Виконан
3	Метод та засоби віддаленого моніторингу іот пристроїв	01.11.2025	Виконан
4	Розробка і тестування прототипа	12.11.2025	Виконан
5	Економічна частина	25.11.2025	Виконан

Студент _____ Степанчук Д. В___

Керівник _____ Богомолов С. В.___

АНОТАЦІЯ

Степанчук Д.В. "Метод та засоби віддаленого моніторингу IoT-пристроїв". Магістерська кваліфікаційна робота зі спеціальності 123 — Комп'ютерна інженерія, Вінниця: ВНТУ, 2025. На укр. мові.

Бібліогр.: 145 назв; рис.: 11; табл. 22.

У роботі проведено комплексне дослідження методу та засобів віддаленого моніторингу IoT-пристроїв у режимі реального часу. Виконано детальний аналіз сучасних протоколів передачі даних для IoT-систем, зокрема MQTT, HTTP, CoAP та WebSocket, визначено їх переваги та недоліки для різних сценаріїв застосування. Досліджено архітектуру провідних хмарних платформ IoT-моніторингу (AWS IoT Core, Azure IoT Hub) та opensource рішень (Eclipse Mosquitto, ThingsBoard).

Розроблено архітектуру системи віддаленого моніторингу на базі п'ятирівневої моделі з Edge/Fog Computing. Реалізовано прототип IoT-пристрою для моніторингу якості повітря з інтеграцією сенсорів температури та вологості DHT11 і датчика концентрації метану MQ-4. Створено серверну інфраструктуру на базі MQTT-брокера Eclipse Mosquitto, Node.js застосунку для обробки телеметрії та бази даних PostgreSQL для зберігання історичних даних. Розроблено веб-інтерфейс для візуалізації показників у реальному часі з підтримкою графіків історії та багатоканальної системи сповіщень через Telegram Bot API та Twilio.

Проведено експериментальні дослідження розробленої системи, що підтвердили її ефективність та надійну роботу при нестабільному інтернет-з'єднанні завдяки механізмам offline buffering та adaptive reconnect. Виконано порівняння з існуючими комерційними рішеннями та розраховано економічну доцільність впровадження системи. Результати роботи можуть бути використані для побудови систем моніторингу в екологічній сфері, розумних будинках, промисловій автоматизації та інших областях застосування IoT-технологій.

Ключові слова: Інтернет речей, IoT-моніторинг, MQTT, ESP32, віддалений моніторинг, телеметрія, хмарні платформи, Edge Computing, система сповіщень, веб-інтерфейс.

ABSTRACT

The thesis presents a comprehensive study of methods and tools for real-time remote monitoring of IoT devices. A detailed analysis of modern data transmission protocols for IoT systems was performed, including MQTT, HTTP, CoAP, and WebSocket, identifying their advantages and disadvantages for various application scenarios. The architecture of leading cloud IoT monitoring platforms (AWS IoT Core, Azure IoT Hub) and open-source solutions (Eclipse Mosquitto, ThingsBoard) was investigated.

A remote monitoring system architecture was developed based on a five-tier model with Edge/Fog Computing. An IoT device prototype for air quality monitoring was implemented with integration of DHT11 temperature and humidity sensors and MQ-4 methane concentration sensor. Server infrastructure was created based on Eclipse Mosquitto MQTT broker, Node.js application for telemetry processing, and PostgreSQL database for storing historical data. A web interface was developed for real-time data visualization with support for historical charts and multi-channel notification system via Telegram Bot API and Twilio.

Experimental studies of the developed system were conducted, confirming its effectiveness and reliable operation with unstable internet connection through offline buffering and adaptive reconnect mechanisms. A comparison with existing commercial solutions was performed and the economic feasibility of system implementation was calculated. The results can be used to build monitoring systems in environmental sphere, smart homes, industrial automation, and other IoT technology application areas.

Keywords: Internet of Things, IoT monitoring, MQTT, ESP32, remote monitoring, telemetry, cloud platforms, Edge Computing, notification system, web interface.

ЗМІСТ

ВСТУП

У сучасному світі Інтернет речей (IoT) стрімко розвивається та охоплює все більше сфер людської діяльності. За прогнозами аналітиків, до 2030 року кількість підключених IoT-пристроїв у світі перевищить 75 мільярдів одиниць. IoT-технології активно впроваджуються в промисловості (Industry 4.0), розумних містах (Smart City), медицині, сільському господарстві, енергетиці та побутовій сфері. Ключовою функцією IoT-систем є можливість віддаленого моніторингу параметрів фізичних об'єктів у режимі реального часу, що дозволяє своєчасно виявляти критичні ситуації, оптимізувати процеси та приймати обґрунтовані рішення на основі актуальних даних.

Однак стрімке зростання кількості IoT-пристроїв породжує нові виклики у сфері їхнього моніторингу та управління. Традиційні централізовані системи не завжди здатні ефективно обробляти великі обсяги даних від розподілених датчиків, забезпечувати низьку латентність для критичних застосувань та гарантувати надійність роботи при нестабільних мережевих з'єднаннях. Крім того, гетерогенність IoT-пристроїв, різноманітність протоколів комунікації та вимоги до енергоефективності створюють додаткові складнощі при проектуванні систем віддаленого моніторингу.

В Україні, яка стикається з проблемами забруднення повітря у великих містах та високим рівнем аварійності через витоки газу в житловому секторі (понад 50 тисяч пожеж щорічно, з яких значна частина спричинена витоками газу), впровадження ефективних систем віддаленого моніторингу IoT-пристроїв набуває особливої актуальності. Такі системи можуть не лише запобігти надзвичайним ситуаціям, але й сприяти покращенню якості життя населення через постійний контроль екологічних параметрів.

Тому дослідження сучасних методів та засобів віддаленого моніторингу IoT-пристроїв, розробка ефективних архітектурних рішень та їх практична апробація є актуальною науково-технічною задачею, яка має важливе практичне значення для розвитку IoT-технологій в Україні.

Метою роботи є дослідження методу та засобів віддаленого моніторингу IoT-пристроїв та розробка ефективної системи моніторингу з низькою латентністю та високою надійністю на прикладі газоаналізатора якості повітря.

Для досягнення поставленої мети необхідно вирішити наступні задачі:

— провести аналіз сучасних методів та протоколів передачі даних в IoT-системах (MQTT, HTTP, CoAP, WebSocket) та визначити оптимальні для різних сценаріїв моніторингу;

— дослідити архітектуру хмарних та opensource платформ для IoT-моніторингу (AWS IoT Core, Azure IoT Hub, ThingsBoard) та порівняти їх функціональні можливості;

— проаналізувати методи сповіщення користувачів (push-нотифікації, SMS, месенджери) та системи візуалізації даних (Grafana, InfluxDB) для IoT-застосувань;

— вивчити аспекти забезпечення безпеки (TLS/SSL, X.509, OAuth 2.0) та надійності (OTA updates, QoS, offline buffering) IoT-систем;

— розробити структурну та функціональну схеми системи віддаленого моніторингу IoT-пристрою;

— реалізувати програмне забезпечення для мікроконтролера ESP32 з підтримкою MQTT-комунікації та механізмів забезпечення надійності;

— створити серверну інфраструктуру для обробки даних та багатоканальних сповіщень на базі Telegram Bot API та Twilio;

— провести експериментальні дослідження розробленої системи та визначити ключові характеристики (латентність, надійність, енергоефективність).

Об'єкт дослідження — процеси віддаленого моніторингу параметрів IoT-пристроїв у режимі реального часу.

Предмет дослідження — методи, протоколи та архітектурні рішення для побудови систем віддаленого моніторингу IoT-пристроїв.

У роботі використовувались **методи** системного аналізу для дослідження архітектури IoT-систем, методи порівняльного аналізу для оцінки протоколів та

платформ, методи експериментального дослідження для визначення характеристик розробленої системи, а також методи об'єктно-орієнтованого програмування для реалізації програмного забезпечення.

Новизна полягає у збільшенні надійності та оперативності реагування системи віддаленого моніторингу IoT-пристроїв за рахунок запропонованого архітектурного рішення з адаптивною частотою опитування датчиків і багатоканальними механізмами сповіщення користувачів, що дало зменшення часу доставки критичних повідомлень та підвищення ефективності прийняття рішень у реальному часі.

Практичне значення одержаних результатів. Результати роботи можуть бути використані при проектуванні та розробці систем віддаленого моніторингу для широкого класу IoT-застосувань: екологічного моніторингу, розумного дому, промислової автоматизації, медичного моніторингу тощо. Розроблена система є масштабованою та може бути адаптована для різних типів датчиків та сценаріїв використання. Практична цінність підтверджується функціональним прототипом газоаналізатора з віддаленим моніторингом, який демонструє латентність 2-5 секунд для критичних сповіщень та забезпечує надійну роботу при нестабільному інтернет-з'єднанні.

1 ТЕОРЕТИЧНІ ОСНОВИ СИСТЕМ ВІДДАЛЕНОГО МОНІТОРИНГУ IoT-ПРИСТРОЇВ

1.1 Поняття та класифікація IoT-систем

Інтернет речей (IoT, Internet of Things) — це концепція мережі фізичних об'єктів ("речей"), які оснащені вбудованими технологіями для взаємодії один з одним або із зовнішнім середовищем, збору та обміну даними. Термін "Інтернет речей" вперше запропонував Кевін Ештон у 1999 році під час презентації в компанії Procter & Gamble, коли описував систему, у якій фізичні об'єкти могли б бути підключені до Інтернету через вбудовані сенсори. З того часу концепція IoT значно еволюціонувала та стала однією з найважливіших технологічних тенденцій XXI століття [1].

Суть IoT полягає у здатності пристроїв автоматично збирати інформацію з датчиків, обробляти її та обмінюватися даними через мережу Інтернет без безпосередньої участі людини. Прикладами IoT-пристроїв є розумні термостати (Nest, Ecobee), носимі фітнес-трекери (Fitbit, Apple Watch), промислові сенсори для моніторингу обладнання, системи моніторингу довкілля, автоматизовані системи керування будівлями (BMS), розумне освітлення (Philips Hue) та багато іншого [1].

За прогнозами аналітичної компанії IDC (International Data Corporation), до 2025 року кількість підключених IoT-пристроїв у світі досягне 41.6 мільярдів, а загальний обсяг даних, які вони генеруватимуть, перевищить 79.4 зетабайти. Ринок IoT демонструє стрімке зростання: якщо у 2020 році його обсяг становив близько $250 мільярдів, то до 2026 року очікується збільшення до $1.1 трильйона з середньорічним темпом зростання (CAGR) 24.7%.

Основним завданням IoT-систем є забезпечення збору, передачі, обробки та аналізу даних у режимі реального часу для прийняття оперативних рішень, оптимізації процесів та покращення якості послуг. Завдяки IoT-технологіям користувачі та автоматизовані системи можуть отримувати доступ до інформації

про стан фізичних об'єктів незалежно від їх географічного розташування, що відкриває нові можливості для управління, контролю та аналітики [1].
При проєктуванні IoT-системи критично важливо правильно визначити її архітектуру, адже це безпосередньо впливає на вибір протоколів передачі даних, хмарних платформ, методів забезпечення безпеки, масштабованість рішення та вимоги до кваліфікації персоналу, який обслуговуватиме систему. Неправильний вибір архітектури на початковому етапі може призвести до суттєвих проблем при масштабуванні системи або до неможливості забезпечити необхідні характеристики продуктивності [1].

Класифікація IoT-систем здійснюється за різними критеріями, серед яких найважливішими є: сфера застосування, спосіб організації зв'язку, архітектурна модель, рівень автономності, масштаб розгортання, тип даних та вимоги до латентності.

За сферою застосування IoT-системи поділяють на кілька основних категорій.

Споживчий IoT (Consumer IoT або CIoT) охоплює пристрої, призначені для домашнього використання та персональних потреб. До цієї категорії належать: розумні годинники та фітнес-браслети (Apple Watch, Samsung Galaxy Watch, Fitbit), розумні колонки з голосовими асистентами (Amazon Echo з Alexa, Google Home, Apple HomePod), системи домашньої автоматизації (розумні термостати Nest, розумне освітлення Philips Hue, розумні замки August), побутова техніка (холодильники Samsung Family Hub, пральні машини LG ThinQ), системи безпеки (відеодзвінки Ring, камери Arlo) та інші пристрої для підвищення комфорту та зручності повсякденного життя [2].

Ці пристрої зазвичай підключаються до домашньої WiFi-мережі та керуються через мобільні додатки для iOS або Android. Характерними особливостями споживчого IoT є простота налаштування (plug-and-play), інтуїтивно зрозумілий інтерфейс, відносно невисока вартість та фокус на користувацькому досвіді. Типові екосистеми споживчого IoT — Amazon Alexa,

Google Home, Apple HomeKit, Samsung SmartThings — дозволяють інтегрувати пристрої різних виробників у єдину систему розумного дому [2].

Промисловий IoT (Industrial IoT або IIoT) використовується на виробництвах, у логістиці та критичній інфраструктурі для моніторингу промислового обладнання, контролю якості продукції, оптимізації виробничих процесів та підвищення загальної ефективності обладнання (OEE — Overall Equipment Effectiveness). Промисловий IoT є технологічною основою концепції Індустрії 4.0 (четвертої промислової революції), яка передбачає цифровізацію виробництва та створення "розумних фабрик" [2].

IIoT-системи характеризуються надзвичайно високими вимогами до надійності (часто потрібна доступність 99.99% або вища), безпеки (як кібербезпеки, так і фізичної безпеки персоналу), точності вимірювань та детермінованості роботи. У промисловому середовищі використовуються спеціалізовані протоколи, такі як OPC UA (Open Platform Communications Unified Architecture), Modbus, PROFINET, що забезпечують надійну та передбачувану комунікацію в умовах складного електромагнітного середовища [3].

Типові застосування IIoT включають: предиктивне обслуговування обладнання (predictive maintenance) на основі аналізу вібрацій, температури та інших параметрів; моніторинг енергоспоживання для оптимізації витрат; контроль якості продукції в реальному часі з використанням машинного зору; відстеження активів (asset tracking) на виробництві та в ланцюгах постачання; оптимізацію виробничих процесів через цифрові двійники (digital twins) [3].

Медичний IoT (Healthcare IoT або IoMT — Internet of Medical Things) включає носимі медичні пристрої, системи віддаленого моніторингу пацієнтів (remote patient monitoring), розумне медичне обладнання та системи управління медичними закладами. Ця категорія має особливо жорсткі вимоги до конфіденційності даних (відповідність HIPAA у США, GDPR в Європі), точності вимірювань (медична сертифікація FDA, CE Mark) та безперебійності роботи, адже від надійності цих систем може залежати життя та здоров'я людей.

Приклади медичного IoT: неперервні глюкометри (continuous glucose monitors) для діабетиків, які автоматично передають дані про рівень цукру в крові; кардіомонітори для пацієнтів з серцевими захворюваннями; інгалятори з підключенням до інтернету для контролю астми; системи відстеження прийому ліків (smart pill dispensers); розумні інвалідні візки; телемедичні системи для консультацій; системи моніторингу пацієнтів у реанімаціях [3].

Розумне місто (Smart City) охоплює великомасштабні IoT-системи на рівні міської інфраструктури: системи моніторингу якості повітря та екологічної обстановки; розумне вуличне освітлення з адаптивним управлінням; інтелектуальні транспортні системи (ITS) для управління трафіком; системи розумного паркування з датчиками зайнятості місць; моніторинг рівня заповнення сміттєвих баків для оптимізації маршрутів збору відходів; системи моніторингу стану доріг та мостів; розумні лічильники води, газу та електроенергії (smart metering); системи відеоспостереження з аналітикою [3].

Системи розумного міста потребують масштабованої інфраструктури, здатної обробляти дані від десятків або сотень тисяч датчиків, високої надійності (адже збої можуть паралізувати роботу міста) та ефективного управління енергоспоживанням. Прикладами міст-лідерів у впровадженні Smart City технологій є Сінгапур, Барселона, Амстердам, Копенгаген.

Сільськогосподарський IoT (Agricultural IoT або Smart Farming) застосовується для точного землеробства (precision agriculture): моніторинг вологості, температури та хімічного складу грунту; автоматизовані системи зрошення з оптимізацією витрат води; контроль мікроклімату в теплицях; відстеження худоби з GPS-трекерами; дрони для моніторингу стану посівів; автоматизація годівлі тварин; прогнозування врожайності [3].

Особливістю сільськогосподарського IoT є те, що ці системи часто працюють у віддалених локаціях з обмеженим доступом до електромережі (потрібні сонячні панелі або акумулятори великої ємності) та нестабільним інтернет-зв'язком (використовуються LPWAN технології з великим радіусом дії, такі як LoRaWAN).

За архітектурною моделлю IoT-системи класифікуються залежно від кількості рівнів обробки та розташування логіки:

Трирівнева архітектура (Three-tier Architecture) є найпростішою та найбільш традиційною моделлю IoT-систем. Вона складається з трьох основних рівнів:

Рівень сприйняття (Perception Layer або Sensing Layer) — це фізичний рівень, де розташовані датчики (sensors) для збору даних з навколишнього середовища та виконавчі пристрої (actuators) для впливу на фізичні об'єкти. На цьому рівні відбувається перетворення фізичних величин (температури, тиску, освітленості тощо) у цифрові сигнали. Приклади компонентів: термодатчики DHT22, датчики руху PIR, датчики газу MQ-серії, камери, GPS-модулі, серводвигуни, реле [3].

Мережевий рівень (Network Layer або Transmission Layer) — забезпечує передачу зібраних даних від датчиків до системи обробки. На цьому рівні використовуються різні технології зв'язку: WiFi, Bluetooth, Zigbee, LoRaWAN, 4G/5G, Ethernet. Також тут відбувається первинна фільтрація та агрегація даних перед відправкою.

Прикладний рівень (Application Layer) — включає хмарні сервіси, бази даних, системи аналітики, веб-додатки та мобільні застосунки для користувачів. На цьому рівні відбувається зберігання даних, їх аналіз, візуалізація та прийняття рішень. Приклади: AWS IoT Core, Azure IoT Hub, користувацькі dashboard, системи машинного навчання [3].

Перевагою трирівневої архітектури є її простота та зрозумілість. Недоліком є те, що всі дані повинні передаватися в хмару для обробки, що створює високу латентність та велике навантаження на мережу, особливо для систем з великою кількістю датчиків або для застосувань, що потребують швидкої реакції.

Чотирирівнева архітектура (Four-tier Architecture) додає проміжний рівень між мережевим та прикладним рівнями:

— рівень датчиків (Sensor Layer) — аналогічний рівню сприйняття у трирівневій моделі.

— мережевий рівень (Network Layer) — канали передачі даних.

— рівень обробки даних або Middleware (Processing Layer або Middleware Layer) — проміжний рівень, де відбувається попередня обробка, фільтрація, агрегація та нормалізація даних перед їх відправкою в хмару. Тут можуть працювати Edge Gateway пристрої, які виконують локальну обробку для зменшення обсягу даних, що передаються в хмару.

Додавання рівня обробки дозволяє зменшити навантаження на хмарну інфраструктуру та мережу, знизити латентність для деяких операцій та підвищити автономність системи при тимчасовій втраті зв'язку з хмарою [4].

П'ятирівнева архітектура з Edge/Fog Computing (Five-tier Architecture) є найбільш сучасною та гнучкою моделлю, яка оптимально розподіляє обчислювальні ресурси між пристроями, краєм мережі та хмарою.

Рівень датчиків та пристроїв (Device/Sensor Layer) — IoT-пристрої з датчиками та мікроконтролерами, які виконують найпростішу обробку: зчитування сенсорів, первинну фільтрацію (наприклад, усереднення значень), локальне прийняття рішень для критичних ситуацій (наприклад, аварійне відключення) [4].

Рівень Edge Gateway (Edge Computing Layer) — шлюзи, які знаходяться "на краю" мережі, поблизу IoT-пристроїв. Виконують більш складну обробку: агрегацію даних від багатьох датчиків, протоколну конвертацію (наприклад, Modbus у MQTT), буферизацію даних при втраті зв'язку, виконання простих алгоритмів машинного навчання (edge AI), локальне управління actuators.

Рівень Fog Computing (Fog Layer або Distributed Cloud) — розподілена обчислювальна інфраструктура, яка розташована між Edge та централізованою хмарою. Fog nodes можуть бути розташовані на рівні підприємства, району міста або регіону. Вони виконують: розподілену обробку даних, локальну аналітику, координацію між різними Edge Gateway, тимчасове зберігання даних, балансування навантаження.

Рівень Cloud Computing (Cloud Layer) — централізоване хмарне сховище та обчислювальна платформа для: довгострокового зберігання історичних даних (data lakes), складних аналітичних обчислень та машинного навчання на великих

датасетах, централізованого управління та моніторингу всієї IoT-інфраструктури, інтеграції з корпоративними системами (ERP, CRM).

Прикладний рівень (Application Layer) — веб та мобільні додатки, API для інтеграції, dashboard для візуалізації, системи бізнес-аналітики (BI), інтерфейси для адміністраторів та кінцевих користувачів.

На рисунку 1.1 наведено структурну схему п'ятирівневої архітектури IoT-системи з розподіленою обробкою даних.

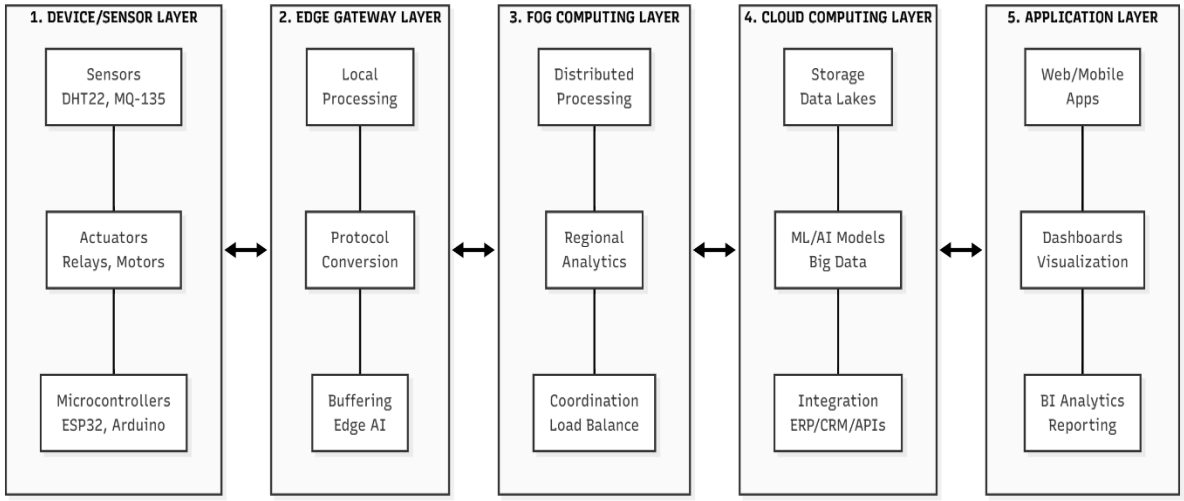

Рисунок 1.1 — П'ятирівнева архітектура IoT-системи

Використання Edge та Fog Computing дозволяє досягти кількох критичних переваг:

— зниження латентності критично важливі рішення приймаються локально на Edge, без необхідності відправки даних у хмару та очікування відповіді, наприклад, у системі автоматичного керування автомобілем рішення про гальмування повинно прийматися за мілісекунди, що неможливо при обробці в хмарі;

— зменшення навантаження на мережу замість відправки всіх сирих даних у хмару, Edge Gateway агрегує та передає тільки важливу інформацію або виявлені події;

— підвищення надійності, система може продовжувати працювати навіть при тимчасовій втраті зв'язку з хмарою, оскільки критична логіка виконується

локально на Edge, дані буферизуються та синхронізуються з хмарою після відновлення зв'язку;

— економія коштів, зменшення обсягу даних, що передаються в хмару, знижує витрати на трафік та хмарне зберігання, що може бути суттєвим для систем з тисячами датчиків;

— підвищення конфіденційності, чутливі дані можуть оброблятися локально на Edge без передачі в хмару, що важливо для дотримання вимог GDPR та інших регуляцій щодо захисту даних [4].

При об'єднанні IoT-пристроїв у мережу критично важливо визначитися з топологією зв'язку, адже вона впливає на надійність системи, складність налаштування, енергоспоживання пристроїв та можливості масштабування.

Зіркоподібна топологія (Star Topology) — найпростіша та найпоширеніша топологія для IoT, де кожен кінцевий пристрій підключається безпосередньо до центрального шлюзу (gateway) або базової станції. Приклади: WiFi мережа з точкою доступу, Bluetooth з центральним пристроєм, Zigbee в режимі зірки [5].

Гібридна топологія (Hybrid Topology) — поєднує переваги різних топологій для створення оптимального рішення. Найпоширенішим варіантом є комбінація зірки та mesh, де групи пристроїв утворюють локальні mesh-мережі, які потім підключаються до центральних шлюзів у зіркоподібній конфігурації.

Приклад: у будівлі кожен поверх має локальну Zigbee mesh-мережу, а mesh-координатори кожного поверху підключаються через Ethernet або WiFi до центрального сервера (рисунок 1.2).

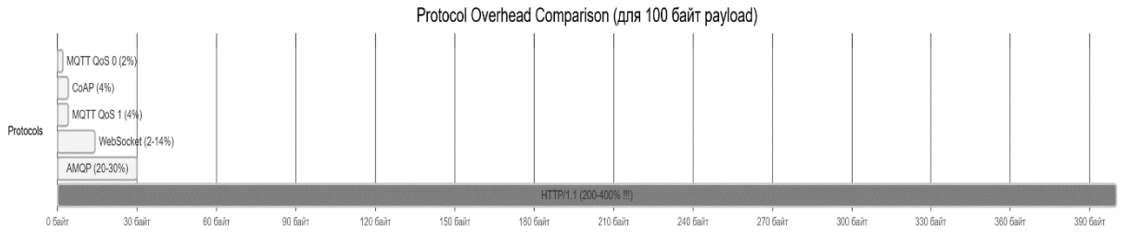

Рисунок 1.2 — Порівняння overhead протоколів IoT

Переваги гібридної топології:

— оптимальний баланс між надійністю та складністю;

— гнучкість у виборі технологій для різних частин системи;

— можливість масштабування шляхом додавання нових сегментів;

— кращий розподіл навантаження.

Технологія Matter (раніше Project CHIP), розроблена спільно Apple, Google, Amazon та іншими компаніями, використовує гібридну архітектуру для створення універсального стандарту розумного дому.

В таблиці 1.1 наведено порівняльний огляд основних топологій IoT-мереж з детальним аналізом їх характеристик та сценаріїв застосування.

Таблиця 1.1 — Топології IoT-мереж та їх характеристики

Параметр	Зіркоподібна	Mesh	Гібридна
Приклади технологій	WiFi, Bluetooth Star, LoRaWAN	Zigbee Mesh, Thread, BLE Mesh	Matter,WiFi HaLow
Складність налаштування	Низька	Висока	Середня
Надійність	Низька (SPOF)	Дуже висока	Висока
Масштабованість	Обмежена	Висока	Дуже висока
Радіус покриття	Обмежений	Розширений	Розширений
Латентність	Низька (1 hop)	Середня (multiple hops)	Низька-Середня
Енергоспоживання	Низьке	Середнє-Високе	Середнє
Вартість інфраструктури	Низька	Середня	Середня-Висока
Складність діагностики	Низька	Висока	Середня
Типові застосування	Домашній IoT, офіси	Промисловість, Smart City	Корпоративні будівлі
Максимальна кількість	10-100	100-1000+	1000+

пристроїв			

В таблиці 1.2 наведено детальне порівняння протоколів IoT за ключовими характеристиками.

Таблиця 1.2 — Порівняльна характеристика протоколів передачі даних для IoT

Параметр	MQTT	HTTP/HTTPS	CoAP	WebSocket	AMQP
Транспортний протокол	TCP	TCP	UDP	TCP	TCP
Архітектура	Pub-Sub	Request-Response	Request-Response	Full-Duplex	Queues + Pub-Sub
Мін. розмір заголовка	2 байти	~200 байтів	4 байти	2-14 байтів	~20 байтів
QoS рівні	0, 1, 2	Немає	CON/NON	Немає	At-most/least/exactly once
Енергоспоживання	Дуже низьке	Середнє-Високе	Дуже низьке	Середнє	Середнє-Високе
Латентність	Низька	Середня	Дуже низька	Низька (після handshake)	Середня
Надійність доставки	Висока (QoS 1,2)	Середня	Середня	Низька	Дуже висока

Масштабованість	Висока (млн. клієнтів)	Середня	Висока	Середня	Висока
Складність реалізації	Низька	Дуже низька	Середня	Низька	Висока

Продовження таблиці 1.2

Параметр	MQTT	HTTP/HTTPS	CoAP	WebSocket	AMQP
Підтримка батарейних пристроїв	Відмінна	Погана	Відмінна	Задовільна	Погана
Підтримка офлайн режиму	Так (persistent sessions)	Ні	Ні	Ні	Так (durable queues)
Безпека	TLS/SSL	TLS/SSL	DTLS	TLS/SSL	TLS/SSL + SASL
Популярність в IoT	Дуже висока	Висока	Середня	Середня	Низька (Enterprise)
Типові застосування	Домашній IoT, Промисловість, Моніторинг	RESTful API, OTA updates	WSN, 6LoWPAN, M2M	Web dashboards, Real-time apps	Enterprise IoT, Financial
Підтримка multicast	Ні	Ні	Так	Ні	Ні
Підтримка браузерами	Ні (потрібен gateway)	Так	Ні	Так (нативно)	Ні

1.2.2 Детальний аналіз протоколу MQTT та його застосування

MQTT (Message Queuing Telemetry Transport) заслуговує на особливу увагу як найпопулярніший та найоптимальніший протокол для IoT-моніторингу.

Архітектура MQTT базується на моделі брокер-клієнт з publish-subscribe (pub-sub) парадигмою, яка радикально відрізняється від традиційної клієнт-серверної моделі з прямою адресацією [5].

Publisher (видавець) — MQTT-клієнт, який публікує дані у певні topics. Publisher не знає, хто підписаний на його topics та скільки підписників існує. Після публікації повідомлення publisher не очікує відповіді від subscribers (асинхронна комунікація) [5].

Subscriber (підписник) — MQTT-клієнт, який підписується на topics для отримання даних. Subscriber не знає, хто публікує дані у topics. Він просто отримує всі повідомлення, опубліковані в topics, на які підписаний.

Topic (тема) — ієрархічний рядок-ідентифікатор, що використовується для організації повідомлень. Topics мають деревоподібну структуру з розділювачем /.

На рисунку 1.3 наведено структурну схему MQTT-архітектури з брокером як центральним елементом.

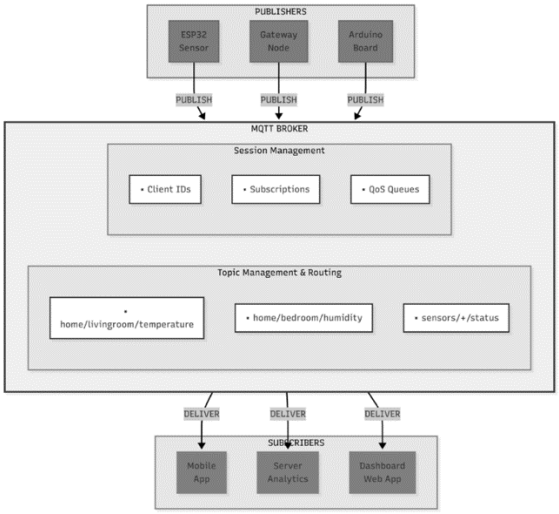

Рисунок 1.3 — Архітектура MQTT-системи з брокером

Переваги pub-sub архітектури:

— повна незалежність publishers та subscribers — вони можуть додаватися/видалятися незалежно;

— масштабованість — легко додати нових publishers або subscribers без змін в існуючих компонентах;

— гнучкість — один publisher може надсилати дані багатьом subscribers, і навпаки;

— loose coupling — зміна одного компонента не впливає на інші.

2 АНАЛІЗ МЕТОДІВ ТА ЗАСОБІВ РЕАЛІЗАЦІЇ СИСТЕМИ МОНІТОРИНГУ

2.1. Огляд існуючих платформ IoT-моніторингу

Вибір платформи для побудови системи віддаленого моніторингу IoT-пристроїв є критично важливим рішенням, яке визначає архітектуру всієї системи, вартість її експлуатації, можливості масштабування та подальшого розвитку. В процесі дослідження було проаналізовано широкий спектр існуючих рішень, які можна класифікувати на три основні категорії: хмарні платформи великих постачальників (AWS IoT Core, Azure IoT Hub), спеціалізовані IoT-платформ як сервіс (ThingsBoard Cloud, Losant, Blynk) та opensource рішення для самостійного розгортання (Node-RED, Mosquitto). Кожна з цих категорій має свої особливості, переваги та обмеження, які необхідно детально розглянути для прийняття обґрунтованого рішення щодо технологічного стеку проекту [6].

2.1.1. Хмарні платформи провідних постачальників

AWS IoT Core представляє собою повністю керовану хмарну платформу від компанії Amazon Web Services, яка забезпечує можливість підключення, управління та обробки даних від IoT-пристроїв на рівні enterprise-масштабу. Платформа побудована на базі високопродуктивного MQTT-брокера, здатного обробляти мільярди повідомлень щодня при підтримці мільйонів одночасних підключень із забезпеченням доступності на рівні 99.9% згідно з угодою про рівень обслуговування [6].

Архітектурно AWS IoT Core складається з кількох взаємопов'язаних компонентів, серед яких Device Gateway виступає як масштабований точкою входу для підключення IoT-пристроїв через протоколи MQTT версій 3.1.1 та 5.0, MQTT over WebSocket для клієнтів на базі браузера, а також HTTPS для RESTful взаємодії. Особливістю цього компонента є автоматичне балансування навантаження між серверами та забезпечення горизонтального масштабування без необхідності втручання адміністратора, що критично важливо для систем з непередбачуваним зростанням кількості підключених пристроїв. На рисунку 2.1 можна побачити приклад комунікації esp32 із сервером із використанням AWS IoT Core [6].

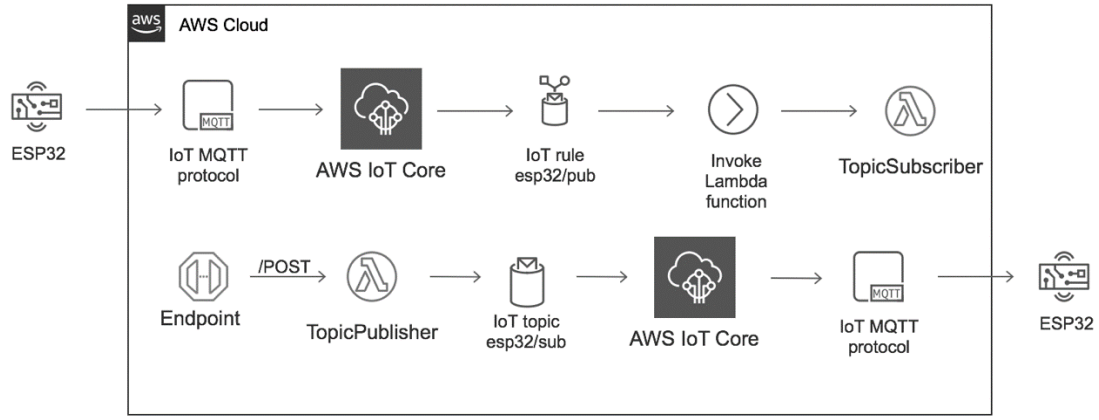

Рисунок 2.1 — Приклад комунікації esp32 через AWS IoT Core

Концепція Device Shadow Service заслуговує на окрему увагу, оскільки вона вирішує фундаментальну проблему IoT-систем — необхідність синхронізації стану пристроїв при нестабільному з'єднанні або тимчасовій недоступності. Thing Shadow представляє собою JSON-документ, що зберігає три типи інформації: поточний стан пристрою (reported state), бажаний стан, встановлений керуючим додатком (desired state), а також метадані з часовими мітками та версіями кожної зміни. Така архітектура дозволяє реалізувати асинхронне управління пристроями, коли команда може бути відправлена навіть якщо пристрій знаходиться в режимі

offline, і буде автоматично застосована при наступному підключенні до мережі [7].

Механізм Rules Engine заслуговує особливої уваги як інструмент для обробки повідомлень у режимі реального часу без необхідності розгортання додаткової інфраструктури. Використовуючи SQL-подібний синтаксис, розробник може створювати правила для фільтрації, трансформації та маршрутизації телеметричних даних безпосередньо на рівні платформи. Наприклад, правило для виявлення критичних температурних показників може виглядати як SQL-запит з умовою WHERE, що фільтрує повідомлення та направляє їх до відповідних сервісів для подальшої обробки чи відправки сповіщень. Така архітектура дозволяє значно спростити розробку та зменшити латентність обробки критичних подій, оскільки логіка виконується максимально близько до джерела даних [7].

Однак при всіх технічних перевагах AWS IoT Core, модель ціноутворення цієї платформи може стати значним обмеженням для проектів з обмеженим бюджетом або великою кількістю пристроїв. Вартість обчислюється на основі споживання ресурсів за кількома метриками: хвилини підключення, кількість повідомлень, операції з Device Shadow, виконання правил Rules Engine та операції з реєстром пристроїв. Проведений економічний аналіз показує, що для системи зі ста датчиками, кожен з яких надсилає дані кожні тридцять секунд, місячна вартість складає близько двадцяти одного долара. При масштабуванні до десяти тисяч пристроїв вартість зростає лінійно до понад двох тисяч доларів на місяць, що робить AWS IoT Core економічно виправданим переважно для корпоративних проектів з відповідним бюджетом [7].

Azure IoT Hub від Microsoft (рисунок 2.2) представляє альтернативний підхід до побудови IoT-інфраструктури з деякими унікальними можливостями, особливо привабливими для організацій, які вже інтегровані в екосистему Microsoft. На відміну від AWS IoT Core, платформа Azure пропонує нативну підтримку протоколу AMQP поряд з MQTT та HTTPS, що робить її особливо придатною для enterprise-сценаріїв з критичними вимогами до надійності доставки повідомлень та транзакційності операцій [7].

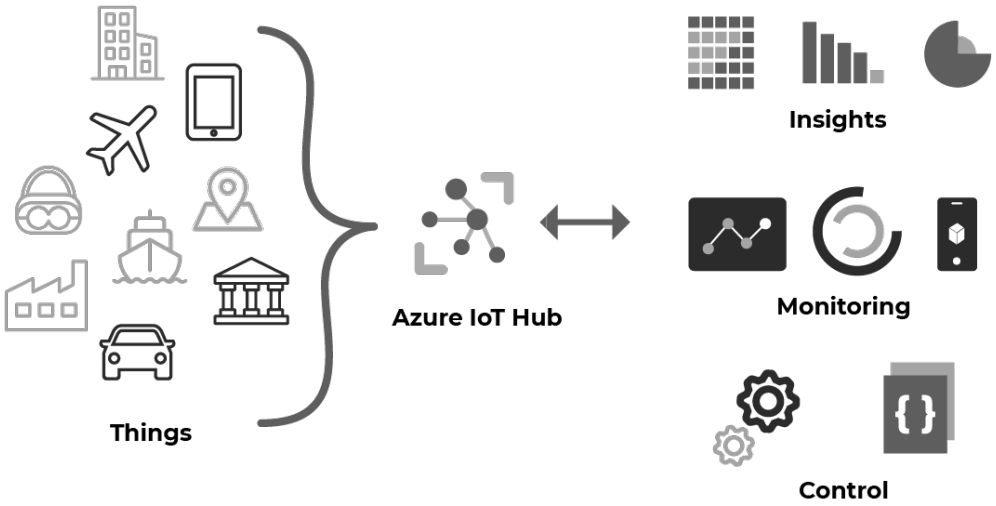

Рисунок 2.2 — Приклад взаємодії пристроїв IoT через AWS IoT Core

Концепція Device Twins у Azure IoT Hub концептуально подібна до Device Shadow в AWS, проте має деякі розширення функціональності. Зокрема, Device Twins підтримує теги для логічного групування пристроїв за довільними характеристиками, що спрощує масове управління флотом пристроїв. Вбудована підтримка SQL-подібних запитів дозволяє швидко знаходити пристрої за складними критеріями без необхідності використання додаткових сервісів індексації [7].

Особливої уваги заслуговує Azure IoT Edge — унікальна можливість платформи, яка дозволяє розгортати контейнеризовані модулі безпосередньо на граничних пристроях. Це відкриває можливості для реалізації складної обробки даних на рівні Edge Computing, включаючи запуск моделей машинного навчання, stream analytics та бізнес-логіки без необхідності постійного зв'язку з хмарою. Така архітектура особливо критична для промислових застосувань, де вимоги до латентності та автономності роботи є критичними [7].

Модель ціноутворення Azure IoT Hub організована за принципом рівнів обслуговування, що кардинально відрізняється від гранульованого pay-per-use підходу AWS. Кожен рівень (Basic та Standard) пропонує фіксовану кількість повідомлень на день за фіксовану місячну плату, що спрощує прогнозування витрат але може призводити до переплати при нерівномірному навантаженні. Порівняльний аналіз показує, що для системи з десятьма тисячами пристроїв

Azure може виявитися значно економічнішим рішенням завдяки flat-rate моделі, проте для невеликих проектів переваги менш очевидні [7].

Важливо відзначити негативний досвід з Google Cloud IoT Core, який було закрито компанією у серпні 2023 року. Це рішення підкреслює ризики vendor lock-in та важливість вибору платформ з довгостроковими зобов'язаннями підтримки. Факт закриття сервісу одним з найбільших технологічних гігантів демонструє, що навіть масштаб та репутація компанії не гарантують стабільності IoT-платформи в довгостроковій перспективі, що є критичним фактором при виборі технологічного стеку для проектів з тривалим життєвим циклом [7].

2.1.2. Спеціалізовані IoT-платформи

ThingsBoard Cloud представляє собою комерційну версію популярної opensource платформи ThingsBoard, яка пропонує баланс між функціональністю корпоративних рішень та доступністю для малих і середніх проектів (рисунок 2.3). На відміну від універсальних хмарних платформ AWS та Azure, ThingsBoard розроблявся спеціально для IoT-застосувань, що відбивається в архітектурі системи та інтерфейсі користувача. Платформа пропонує повноцінну систему керування життєвим циклом IoT-пристроїв з підтримкою Device Profiles для групування пристроїв за характеристиками, Asset Management для побудови ієрархічних структур об'єктів та механізмами масового provisioning для автоматизації реєстрації великої кількості пристроїв [8].

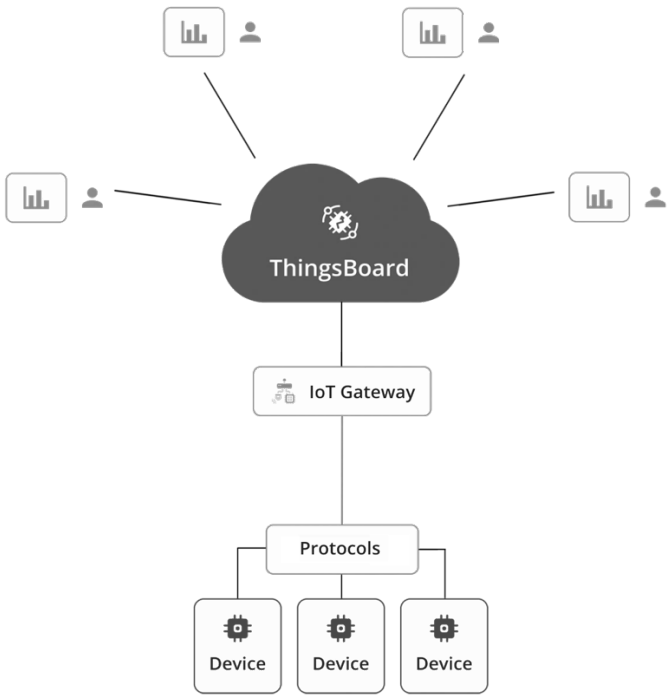

Рисунок 2.3 — Архітектура IoT-системи моніторингу на базі платформи ThingsBoard

Архітектура Rule Engine в ThingsBoard заслуговує на детальний розгляд, оскільки вона реалізована через візуальний конструктор на базі Node-based потоків, що значно спрощує створення складної логіки обробки даних порівняно з текстовими Rules Engine в AWS чи Azure. Розробник може використовувати drag-and-drop інтерфейс для побудови processing pipelines з готових компонентів: вузлів фільтрації, трансформації, агрегації, збагачення даних, а також integration nodes для відправки даних у зовнішні системи через різні протоколи та API. Такий підхід значно знижує поріг входу та прискорює розробку, хоча може мати обмеження в продуктивності при обробці екстремально великих потоків даних [8].

Система візуалізації ThingsBoard включає понад п'ятдесят готових віджетів для побудови dashboards, що охоплюють типові потреби IoT-проектів: графіки часових рядів з можливостями масштабування, індикатори та gauge для відображення поточних значень, карти з підтримкою різних постачальників тайлів, елементи управління для двосторонньої взаємодії з пристроями, а також

спеціалізовані віджети для відображення активних тривог. Важливо, що система дозволяє створювати власні віджети на базі стандартних веб-технологій, що забезпечує гнучкість для специфічних вимог візуалізації [8].

Модель ціноутворення ThingsBoard Cloud організована за рівнями підписки від безкоштовного Community плану з обмеженням у десять пристроїв до корпоративних планів з тисячами пристроїв та можливістю white-labeling. Для системи зі ста пристроями необхідний Prototype план вартістю сто доларів на місяць, що перевищує вартість AWS IoT Core для аналогічного навантаження, проте включає готову систему візуалізації, rule engine та інтерфейс управління без необхідності додаткової розробки та інтеграції компонентів [8].

Blynk являє собою платформу, орієнтовану переважно на сегмент швидкого прототипування та hobby-проектів, хоча має можливості масштабування до комерційних застосувань. Основною перевагою платформи є мобільний App Builder з інтуїтивним drag-and-drop інтерфейсом, що дозволяє створювати повноцінні мобільні додатки для iOS та Android без написання коду. Бібліотека готових widgets охоплює типові елементи управління та візуалізації, підтримується real-time синхронізація між пристроями та додатками, а також функціональність push-нотифікацій [8].

Бібліотека Blynk.Edgent для вбудованих систем заслуговує на увагу завдяки реалізації WiFi provisioning через Bluetooth, що значно спрощує процес початкового налаштування пристроїв кінцевими користувачами. Замість необхідності жорстко вшивати credentials WiFi-мережі в firmware, користувач може налаштувати підключення через мобільний додаток, що особливо важливо для consumer IoT продуктів. Підтримка OTA firmware updates та механізм device claiming для безпечної прив'язки пристрою до облікового запису користувача додають функціональність, яку зазвичай доводиться реалізовувати власноруч [8].

Losant позиціонується як enterprise-grade платформа з фокусом на промислові застосування та advanced analytics, що відображається як в функціональності, так і в ціноутворенні. Платформа особливо популярна в секторах manufacturing, energy, logistics та agriculture, де критичними є вимоги до

надійності, безпеки та аналітичних можливостей. Архітектура Losant організована навколо концепції Applications як контейнерів для пристроїв, workflows та dashboards, що дозволяє логічно відокремлювати різні проекти та середовища розробки [9].

Visual Workflow Engine платформи Losant вирізняється серед конкурентів кількістю готових компонентів та можливостями для створення складної логіки обробки. Понад сто попередньо створених nodes охоплюють тригери, логічні операції, дії та операції з даними, при цьому система підтримує state management для реалізації складних state machines, debugging tools для step-through виконання з breakpoints, а також testing framework для написання unit tests для workflows. Така функціональність наближає платформу до професійних систем розробки та робить її придатною для критичних промислових застосувань [9].

2.1.3. Opensource рішення для самостійного розгортання

Категорія opensource рішень представляє особливий інтерес для проектів з обмеженим бюджетом, специфічними вимогами до безпеки даних або необхідністю on-premise розгортання. Основною перевагою цих рішень є відсутність recurring costs у вигляді щомісячних платежів за підписку, повний контроль над інфраструктурою та можливість модифікації вихідного коду під специфічні вимоги проекту. Водночас така гнучкість має свою ціну у вигляді необхідності власних ресурсів для розгортання, налаштування, підтримки та оновлення системи.

ThingsBoard Community Edition представляє собою opensource версію комерційної платформи з практично повною функціональністю, за винятком деяких enterprise-можливостей, таких як white-labeling, multi-tenancy та розширені інструменти інтеграції. Платформа може бути розгорнута на власному сервері або віртуальній машині в будь-якому дата-центрі, що забезпечує повний контроль над даними та дозволяє уникнути проблем з GDPR compliance для європейських проектів. Системні вимоги включають мінімум чотири гігабайти оперативної

пам'яті та двоядерний процесор для невеликих інсталяцій, з можливістю горизонтального масштабування через кластеризацію для enterprise-навантажень.

Архітектура ThingsBoard побудована на базі мікросервісів з використанням сучасних технологій: Spring Boot для backend, Apache Kafka для асинхронної обробки повідомлень, Cassandra або PostgreSQL для зберігання телеметрії, Redis для кешування. Така архітектура забезпечує високу продуктивність та можливість обробки десятків тисяч повідомлень на секунду на типовому сервері, проте вимагає відповідної кваліфікації для налаштування та підтримки всього стеку технологій (рисунок 2.4).

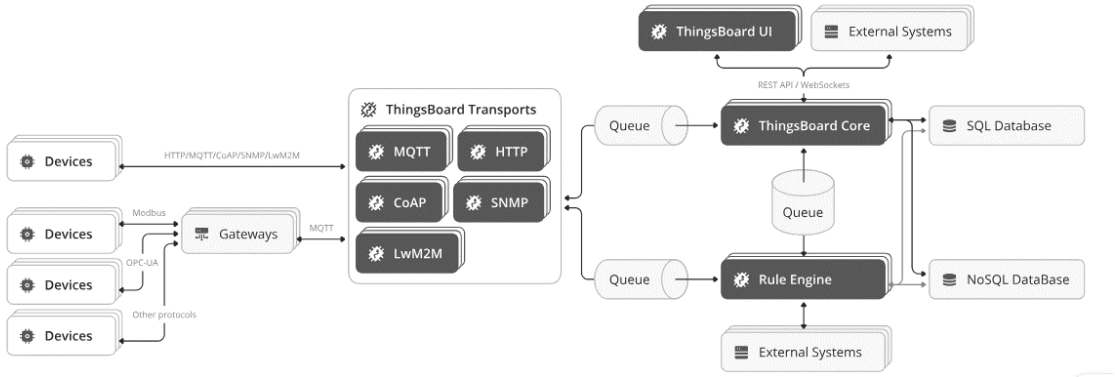

Рисунок 2.4 — Схема архітектури ThingsBoard

Eclipse Mosquitto являє собою легковаговий opensource MQTT-брокер, який став де-факто стандартом для малих та середніх IoT-проектів завдяки своїй простоті, надійності та мінімальним системним вимогам. Брокер написаний на мові C, має невеликий footprint та може працювати навіть на embedded системах типу Raspberry Pi. Mosquitto підтримує MQTT версій 3.1, 3.1.1 та 5.0, що забезпечує сумісність з широким спектром клієнтських бібліотек та пристроїв [9].

Функціональність Mosquitto включає підтримку всіх рівнів QoS, persistent sessions для надійної доставки при нестабільному з'єднанні, retained messages для отримання останнього стану при підписці, механізм Last Will and Testament для виявлення несподіваних відключень пристроїв. Система аутентифікації та авторизації може бути налаштована через конфігураційні файли з паролями або інтегрована з зовнішніми системами через plugin API. Підтримка TLS/SSL

забезпечує шифрування трафіку, а механізм bridge дозволяє створювати розподілені архітектури з множиною брокерів [9].

Основною перевагою Mosquitto є відсутність будь-яких обмежень на кількість пристроїв або повідомлень, на відміну від хмарних платформ з їх pricing models. Брокер може обробляти сотні тисяч повідомлень на секунду на сучасному сервері, при цьому споживання ресурсів залишається мінімальним. Недоліком є відсутність готових інструментів для візуалізації, аналітики та управління пристроями, що вимагає інтеграції з додатковими компонентами для побудови повноцінної IoT-платформи.

Node-RED представляє собою flow-based програмне середовище для IoT, розроблене IBM та перенесене під крило JS Foundation. Інструмент використовує візуальний редактор для створення workflows через з'єднання попередньо створених nodes, що робить його особливо привабливим для швидкого прототипування та розробки proof-of-concept систем. Бібліотека nodes охоплює широкий спектр протоколів та сервісів: MQTT, HTTP, WebSocket, TCP/UDP, serial ports, а також інтеграції з популярними платформами та сервісами [10].

Архітектура Node-RED базується на Node.js runtime, що забезпечує високу продуктивність для event-driven застосувань та легку інтеграцію з npm-екосистемою. Flows можуть включати JavaScript-код для реалізації складної логіки обробки даних, при цьому система підтримує debugging з breakpoints та інспектування змінних. Можливість створення власних custom nodes дозволяє розширювати функціональність під специфічні потреби проекту.

Node-RED особливо корисний як middleware між IoT-пристроями та backend-системами, дозволяючи швидко реалізувати протоколну конвертацію, фільтрацію, агрегацію та маршрутизацію даних без необхідності написання великої кількості коду. Проте для production-систем з високими вимогами до надійності та масштабованості може знадобитися додаткова робота з налаштування monitoring, error handling та load balancing [10].

2.1.4. Порівняльний аналіз платформ

Проведений аналіз платформ IoT-моніторингу дозволяє сформувати комплексне розуміння переваг та недоліків кожної категорії рішень. Хмарні платформи великих постачальників, такі як AWS IoT Core та Azure IoT Hub, пропонують найвищий рівень масштабованості, надійності та інтеграції з іншими корпоративними сервісами, проте їх вартість може бути надто високою для проектів з обмеженим бюджетом або великою кількістю малопотужних пристроїв. Додатковим ризиком є vendor lock-in, який ускладнює міграцію на альтернативні платформи у майбутньому, що підтверджується прикладом закриття Google Cloud IoT Core [11].

Спеціалізовані IoT-платформи представляють собою компроміс між функціональністю та вартістю, пропонуючи готові інструменти для візуалізації, rule engine, device management при помірних цінах для малих та середніх проектів. ThingsBoard Cloud виглядає особливо привабливим завдяки балансу між можливостями та вартістю, проте залишається питання довгострокової підтримки та розвитку платформи, оскільки навіть великі компанії можуть припинити підтримку своїх IoT-сервісів [11].

Opensource рішення надають максимальний контроль та гнучкість при відсутності recurring costs, що робить їх оптимальним вибором для академічних проектів, дослідницьких розробок та комерційних застосувань з чітко визначеними вимогами та можливістю виділення ресурсів на підтримку інфраструктури. Комбінація Mosquitto як MQTT-брокера та node.js сервера як middleware для обробки даних та інтеграцій представляє собою особливо привабливе рішення завдяки простоті розгортання, мінімальним системним вимогам та відсутності обмежень на кількість пристроїв або повідомлень [11].

В таблиці 2.1 наведено порівняльний аналіз розглянутих платформ за ключовими критеріями.

Таблиця 2.1 — Порівняльний аналіз платформ IoT-моніторингу

Критерій	AWS IoT Core	Azure IoT Hub	ThingsBoard Cloud	Opensource

Вартість (100 пристроїв/міс)	~$21	~$25	$100	$5-20 (хостинг)
Масштабованість	Необмежена	Необмежена	До 2500 пристроїв на плані	Залежить від сервера
Складність налаштування	Висока	Висока	Низька	Середня
Готова візуалізація	Ні	Обмежена	Так	Залежить від рішення
Vendor lock-in	Високий	Високий	Середній	Відсутній
Підтримка Edge	Обмежена	IoT Edge	Так	Залежить від рішення
Час до MVP	Тижні	Тижні	Дні	Дні-тижні
SLA	99.9 %	99.9 %	99.5%	Немає

Для цілей даної магістерської роботи, де основним завданням є дослідження методів та засобів віддаленого моніторингу з фокусом на низьку латентність та надійність при обмеженому бюджеті, найбільш доцільним вибором є комбінація opensource компонентів. Використання Eclipse Mosquitto як MQTT-брокера забезпечує надійну та швидку доставку повідомлень.

2.2. Порівняльний аналіз апаратних платформ для IoT-пристроїв

Вибір апаратної платформи для реалізації IoT-пристрою є не менш критичним рішенням, ніж вибір програмної інфраструктури, оскільки він визначає можливості з точки зору продуктивності обчислень, споживання енергії, вартості виробництва, доступних інтерфейсів та складності розробки. Сучасний ринок мікроконтролерів та одноплатних комп'ютерів для IoT пропонує широкий спектр рішень від простих 8-бітних мікроконтролерів до потужних ARM-процесорів з підтримкою повноцінної операційної системи [12].

2.2.1. Критерії вибору апаратної платформи

При виборі апаратної платформи для IoT-пристрою необхідно враховувати комплекс взаємопов'язаних факторів, які часто є конфліктуючими та вимагають компромісів. Продуктивність процесора визначає можливість виконання складних обчислень на пристрої, включаючи криптографічні операції для забезпечення безпеки, обробку даних з датчиків, підтримку стеків протоколів. Проте вища продуктивність зазвичай корелює з вищим споживанням енергії, що критично важливо для батарейних пристроїв, які повинні працювати місяцями або роками без заміни джерела живлення [12].

Обсяг оперативної пам'яті та флеш-пам'яті визначає можливість зберігання програмного коду, буферів даних, TLS-сертифікатів та offline-буферизації повідомлень при втраті зв'язку. Сучасні MQTT-стеки з підтримкою TLS можуть вимагати десятки кілобайтів RAM та сотні кілобайтів флеш-пам'яті, що робить платформи з обмеженою пам'яттю непридатними для повноцінних IoT-застосувань з вимогами до безпеки.

Вбудовані можливості бездротового зв'язку є критичним фактором для IoT-пристроїв, оскільки зовнішні модулі збільшують споживання енергії, вартість та складність розробки. Інтеграція WiFi, Bluetooth, LoRa або Cellular модуля безпосередньо в мікроконтролер значно спрощує проектування та зменшує footprint пристрою. Підтримка різних інтерфейсів периферії, таких як UART, SPI, I2C, ADC, DAC, PWM визначає можливість підключення різноманітних датчиків та виконавчих пристроїв [12].

Екосистема розробки включає доступність бібліотек, фреймворків, інструментів налагодження, документації та спільноти розробників. Платформи з активною спільнотою та багатою екосистемою значно прискорюють розробку завдяки наявності готових рішень для типових задач. Вартість платформи для серійного виробництва може коливатися від одного-двох доларів для простих мікроконтролерів до десятків доларів для потужних одноплатних комп'ютерів, що критично впливає на економіку продукту при масштабуванні.

2.2.2. ESP32/ESP8266 платформи

Сімейство мікроконтролерів ESP від китайської компанії Espressif Systems зробило революцію на ринку IoT завдяки поєднанню високої продуктивності, вбудованого WiFi та Bluetooth, низької вартості та відкритого SDK. ESP8266, представлений у 2014 році, став першим мікроконтролером з вбудованим WiFi за ціною менше трьох доларів, що зробило IoT-розробку доступною для широкого кола розробників та hobbyists [15].

ESP8266 базується на 32-бітному процесорі Tensilica L106 з тактовою частотою вісімдесят мегагерц, має близько п'ятдесяти кілобайтів доступної RAM після вирахування потреб WiFi-стеку та до чотирьох мегабайтів зовнішньої флеш-пам'яті. Вбудований WiFi модуль підтримує стандарти 802.11 b/g/n з максимальною швидкістю до сімдесяти двох мегабітів на секунду, що цілком достатньо для IoT-застосувань. Підтримка TLS 1.2 дозволяє реалізувати захищене з'єднання з MQTT-брокером або хмарними платформами, хоча криптографічні операції суттєво навантажують процесор (рисунок 2.5).

Екосистема розробки для ESP8266 включає кілька опцій: нативний Espressif SDK на базі FreeRTOS, Arduino Core для ESP8266 для використання знайомого Arduino-середовища, MicroPython та Lua інтерпретатори для скриптових мов. Найпопулярнішою є Arduino Core завдяки величезній кількості готових бібліотек та прикладів коду, що значно прискорює розробку. Проте обмеження оперативної пам'яті можуть стати проблемою для складніших застосувань з великими буферами даних або багатозадачністю [16].

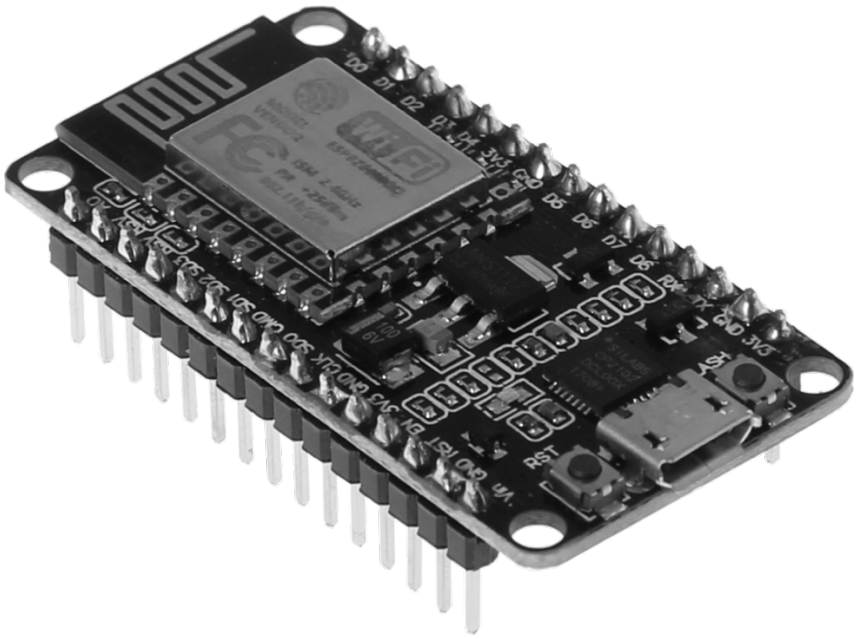

Рисунок 2.5 — ESP8266

ESP32 представляє собою наступне покоління платформи з суттєво розширеними можливостями. Мікроконтролер базується на двоядерному процесорі Tensilica Xtensa LX6 з тактовою частотою до двохсот сорока мегагерц, має до п'ятисот дванадцяти кілобайтів SRAM та до чотирьох мегабайтів зовнішньої флеш-пам'яті, що робить його придатним навіть для складних застосувань з локальною обробкою даних та машинним навчанням на краю. Вбудовані WiFi 802.11 b/g/n та Bluetooth 4.2 + BLE розширюють можливості комунікації, дозволяючи створювати mesh-мережі або використовувати Bluetooth для локального управління при відсутності інтернет-з'єднання [16].

Архітектура ESP32 включає широкий набір периферійних інтерфейсів: вісімнадцять каналів 12-бітного ADC для підключення аналогових датчиків, два канали 8-бітного DAC, UART, SPI, I2C, I2S для аудіо, CAN для автомобільних застосувань, Ethernet MAC, touch sensors, Hall sensor. Апаратна підтримка криптографічних операцій (AES, SHA, RSA) через crypto-акселератор значно прискорює TLS handshake та зменшує навантаження на процесор, що критично важливо для IoT-пристроїв з вимогами до енергоефективності.

ESP32-WROOM-32 (рисунок 2.6), який використовується в даній роботі, представляє собою модуль, що містить ESP32 мікроконтролер, флеш-пам'ять,

кварцовий резонатор, антену та екранований корпус у компактному form-factor. Модуль сертифікований FCC, CE, SRRC, що спрощує процес сертифікації кінцевого продукту. Вартість модуля в роздріб складає близько трьох-п'яти доларів, при об'ємах виробництва понад тисячу штук ціна може знижуватися до двох доларів, що робить ESP32 одним з найбільш економічно привабливих рішень для IoT-пристроїв [17].

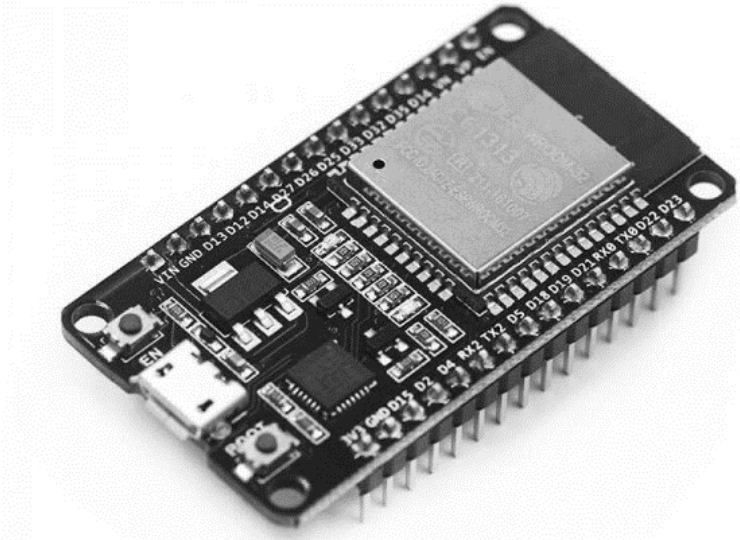

Рисунок 2.6 — Модуль ESP32-WROOM-32

SDK для ESP32 включає ESP-IDF (Espressif IoT Development Framework) на базі FreeRTOS з підтримкою всіх можливостей платформи, Arduino Core для ESP32 для спрощеної розробки в звичному середовищі, MicroPython для швидкого прототипування. ESP-IDF надає повний контроль над апаратом та оптимальну продуктивність, проте вимагає глибоких знань embedded-розробки. Arduino Core пропонує компроміс між простотою та функціональністю, маючи доступ до більшості можливостей ESP32 через звичний Arduino API з величезною екосистемою бібліотек.

Підтримка OTA (Over-The-Air) updates в ESP32 реалізована на рівні SDK через dual partition scheme, де флеш-пам'ять розділена на дві області для firmware. Під час OTA-оновлення нова версія завантажується у другу partition без переривy роботи пристрою, після чого перевіряється checksuma та виконується перезавантаження з нової версії. У разі проблем з новою версією система автоматично повертається до попередньої робочої версії, що забезпечує надійність процесу віддаленого оновлення [17].

Deep Sleep режим ESP32 дозволяє знизити споживання енергії до п'яти мікроампер, що робить можливим живлення від батарейки протягом місяців або років для пристроїв з рідкісною передачею даних. У Deep Sleep режимі працює тільки RTC (Real-Time Clock) та ULP (Ultra Low Power) співпроцесор, який може періодично зчитувати датчики та пробуджувати основний процесор при виконанні певних умов. Час пробудження з Deep Sleep складає близько п'ятисот мілісекунд, що необхідно враховувати при проектуванні системи [17].

2.2.3. Raspberry Pi та одноплатні комп'ютери

Raspberry Pi представляє собою сімейство одноплатних комп'ютерів з повноцінним ARM-процесором, здатним запускати операційні системи на базі Linux. На відміну від мікроконтролерів, Raspberry Pi пропонує значно вищу продуктивність та можливість використання високорівневих мов програмування, проте за рахунок вищого споживання енергії, складнішого управління живленням та вищої вартості [18].

Raspberry Pi 4 Model B (рисунок 2.7) оснащений чотириядерним ARM Cortex-A72 процесором з тактовою частотою півтора гігагерца, від двох до восьми гігабайтів оперативної пам'яті LPDDR4, вбудованим Gigabit Ethernet, WiFi 802.11ac та Bluetooth 5.0. Така конфігурація дозволяє запускати повноцінні застосунки на Python, Node.js, Java або будь-яких інших мовах, використовувати бази даних, веб-сервери, контейнери Docker безпосередньо на пристрої. Проте споживання енергії в активному режимі складає близько трьох-чотирьох ват, що робить живлення від батарейки непрактичним для більшості застосувань [18].

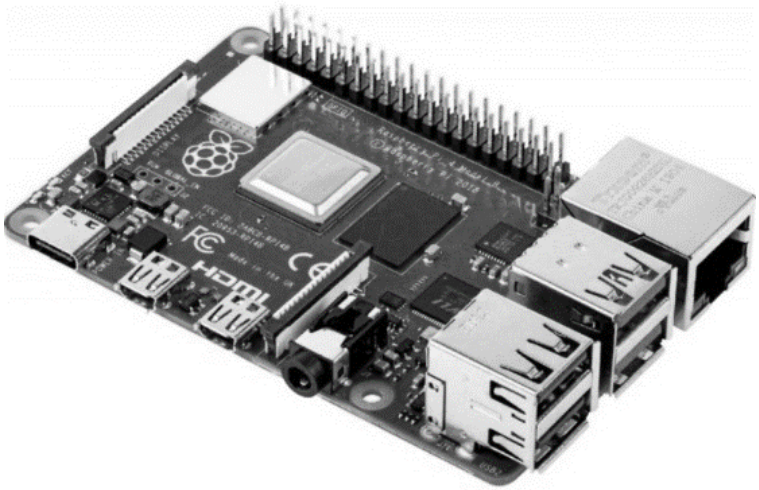

Рисунок 2.7 — Raspberry Pi 4 Model

Raspberry Pi Zero W представляє компактнішу та економічнішу версію з одноядерним ARM11 процесором на тактовій частоті один гігагерц, п'ятистами дванадцятьма мегабайтами RAM, вбудованим WiFi 802.11n та Bluetooth 4.1. Споживання енергії знижене до півтора вата, вартість близько десяти доларів, що робить платформу більш придатною для IoT-застосувань. Проте навіть така конфігурація залишається надто енергоємною для батарейних пристроїв, які повинні працювати тижнями або місяцями.

Основна перевага Raspberry Pi для IoT полягає у можливості виконання складної обробки даних безпосередньо на пристрої: комп'ютерний зір через бібліотеку OpenCV, машинне навчання через TensorFlow Lite, обробка аудіо, складні алгоритми фільтрації та агрегації даних. Така функціональність робить Raspberry Pi ідеальним вибором для Edge Gateway пристроїв, які збирають дані від багатьох простих сенсорів через UART, SPI або I2C, виконують локальну обробку та агрегацію, а потім передають результати в хмару [18].

У контексті розробленої системи моніторингу Raspberry Pi міг би використовуватися як Edge Gateway для збору даних від множини ESP32-датчиків

через MQTT, виконання локальної агрегації та фільтрації, а також як локальний сервер для Mosquitto брокера. Проте для простого газоаналізатора з одним датчиком така архітектура була б надмірною, тому вибір на користь ESP32 як самодостатнього пристрою є більш обґрунтованим.

2.2.4. Обґрунтування вибору ESP32-WROOM-32

На основі проведеного аналізу апаратних платформ для реалізації IoT-пристрою контролю якості повітря був обраний модуль ESP32-WROOM-32. Це рішення обґрунтовується комплексом факторів, які роблять ESP32 оптимальним вибором саме для даного застосування [19].

Вбудовані WiFi та Bluetooth модулі усувають необхідність використання зовнішніх компонентів для бездротового зв'язку, що спрощує розробку апаратної частини та зменшує загальну вартість пристрою. Підтримка TLS 1.2 з апаратним crypto-акселератором дозволяє реалізувати захищене з'єднання з MQTT-брокером без значного впливу на продуктивність або час автономної роботи [19].

Достатня продуктивність двоядерного процесора на частоті до двохсот сорока мегагерц дозволяє виконувати всі необхідні операції: зчитування даних з датчиків DHT11 та MQ-4, підтримку MQTT-з'єднання з QoS 1, оновлення OLED-дисплея, управління буззером, обробку команд від Telegram-бота, при цьому залишаючи резерв продуктивності для розширення функціональності в майбутньому.

Обсяг оперативної пам'яті у п'ятсот дванадцять кілобайтів є цілком достатнім для реалізації всіх компонентів системи, включаючи буфери для MQTT-повідомлень, TLS-сертифікати, бібліотеки для роботи з датчиками та дисплеєм. Чотири мегабайти флеш-пам'яті дозволяють розмістити firmware з усіма бібліотеками та залишити місце для другої partition для OTA-оновлень [19].

Багатий набір периферійних інтерфейсів ESP32 дозволяє без проблем підключити всі необхідні компоненти: датчик DHT11 через GPIO з one-wire протоколом, датчик MQ-4 через ADC для зчитування аналогового сигналу, OLED-дисплей через I2C для швидкої передачі даних, буззер через GPIO з

підтримкою PWM для генерації різних тонів. При цьому залишаються вільні піни для можливого розширення системи додатковими датчиками або виконавчими пристроями.

Екосистема розробки Arduino IDE для ESP32 забезпечує швидкий старт та доступ до величезної кількості готових бібліотек: PubSubClient або MQTT для роботи з MQTT-протоколом, UniversalTelegramBot для інтеграції з Telegram Bot API, Adafruit_SSD1306 для управління OLED-дисплеєм, DHT sensor library для роботи з датчиком температури та вологості. Наявність численних прикладів коду та активної спільноти значно прискорює розробку та спрощує вирішення проблем, що виникають [19].

Підтримка Deep Sleep режиму з споживанням п'ять мікроампер відкриває можливості для створення версії пристрою з живленням від батарейки для застосувань, де постійний моніторинг не є критичним. Пристрій може пробуджуватися раз на годину, знімати показники датчиків, відправляти дані через MQTT та повертатися у Deep Sleep, забезпечуючи місяці автономної роботи від Li-Ion акумулятора ємністю дві-три тисячі міліампер-годин.

Низька вартість модуля ESP32-WROOM-32 у межах трьох-п'яти доларів за штуку робить його економічно привабливим рішенням як для прототипування, так і для потенційного серійного виробництва. Для порівняння, Arduino MKR WiFi 1010 коштує близько тридцяти доларів при меншій продуктивності та функціональності, Raspberry Pi Zero W при вартості десять доларів має значно вище споживання енергії та не підходить для батарейних пристроїв.

В таблиці 2.2 наведено порівняння основних характеристик розглянутих апаратних платформ.

Таблиця 2.2 — Порівняння апаратних платформ для IoT-пристроїв

Параметр	Arduino Uno	ESP8266	ESP32-WROOM-32	Raspberry Pi Zero W
Процесор	ATmega328P 16MHz	Tensilica L106 80MHz	Dual Xtensa LX6 240MHz	ARM11 1GHz

RAM	2KB	~50KB	512KB	512MB
Flash	32KB	До 4MB	До 4MB	microSD
WiFi	Зовнішній	Вбудований	Вбудований	Вбудований
Bluetooth	Немає	Немає	BT 4.2 + BLE	BT 4.1
Споживання (активний)	~50mA	~80mA	~160mA (dual core)	~150mA
Споживання (sleep)	~15mA	~20μA	~5μA	~100mA
Підтримка TLS	Неможлива	Програмна	Апаратна	Повна
Вартість	~$3	~$2	~$3-5	~$10
OTA Updates	Ні	Так	Так	Так
GPIO кількість	14	11	34	40
ADC	10-біт, 6 каналів	10-біт, 1 канал	12-біт, 18 каналів	Немає
Складність розробки	Низька	Середня	Середня	Висока

Як видно з порівняння, ESP32-WROOM-32 пропонує найкращий баланс між продуктивністю, функціональністю, енергоефективністю та вартістю для задач IoT-моніторингу з вимогами до надійності та безпеки. Саме тому ця платформа була обрана для реалізації практичної частини магістерської роботи.

2.3. Аналіз програмних засобів для серверної частини

Серверна інфраструктура системи віддаленого моніторингу IoT-пристроїв відіграє критичну роль у забезпеченні надійної обробки даних, маршрутизації повідомлень, виконання бізнес-логіки та інтеграції з зовнішніми сервісами для сповіщення користувачів. В даному розділі проводиться детальний аналіз програмних компонентів серверної частини, які використовуються або могли б використовуватися для побудови системи моніторингу [20].

Eclipse Mosquitto є найпопулярнішим opensource MQTT-брокером завдяки поєднанню надійності, продуктивності та простоти налаштування. Брокер написаний на мові C, що забезпечує мінімальний footprint та ефективне використання системних ресурсів. Mosquitto повністю відповідає специфікаціям MQTT версій 3.1, 3.1.1 та 5.0, підтримує всі рівні QoS (0, 1, 2), persistent sessions для надійної доставки при нестабільному з'єднанні, retained messages та механізм Last Will and Testament для виявлення несподіваних відключень пристроїв [20].

Архітектура Mosquitto організована навколо single-threaded event loop з використанням epoll або kqueue для ефективної обробки тисяч одночасних з'єднань. Тестування показують, що на сучасному сервері (Intel Xeon 8 cores, 16GB RAM) Mosquitto може обробляти понад сто тисяч повідомлень на секунду при тисячах одночасних підключень, споживаючи при цьому менше одного гігабайта оперативної пам'яті. Така продуктивність є цілком достатньою для більшості IoT-проектів малого та середнього масштабу [20].

Система безпеки Mosquitto включає підтримку TLS/SSL для шифрування з'єднань з можливістю вимагання клієнтських сертифікатів для взаємної автентифікації. Механізм контролю доступу дозволяє налаштовувати права на публікацію та підписку для різних користувачів або груп через конфігураційні файли або інтеграцію з зовнішніми системами автентифікації через plugin API. Підтримка Access Control Lists дозволяє створювати детальні правила типу "користувач sensor01 може публікувати тільки у topics sensors/sensor01/*".

Механізм bridge у Mosquitto дозволяє створювати ієрархічні архітектури з множиною брокерів, де локальні брокери на рівні Edge збирають дані від пристроїв та пересилають їх до центрального брокера в хмарі. Така архітектура забезпечує масштабованість та відмовостійкість, дозволяє локальним системам продовжувати працювати при втраті зв'язку з центром, накопичувати дані та синхронізувати їх після відновлення з'єднання [20].

Для даної магістерської роботи Mosquitto був обраний як MQTT-брокер завдяки його надійності, простоті розгортання та налаштування, а також відсутності ліцензійних обмежень. Брокер розгорнуто на віртуальному

приватному сервері з операційною системою Ubuntu 22.04 LTS, два ядра процесора та два гігабайти оперативної пам'яті, чого цілком достатньо для обробки телеметрії від сотень датчиків у дослідницьких цілях.

Серверна частина системи реалізована на Node.js з використанням TypeScript для типобезпечної розробки. Node.js runtime забезпечує ефективну асинхронну обробку подій від IoT-пристроїв завдяки event-driven архітектурі та non-blocking I/O операціям. Модульна структура додатку дозволяє розділити логіку на окремі компоненти, кожен з яких відповідає за конкретну функцію: підписку на MQTT topics, валідацію даних, перевірку порогових значень, відправку сповіщень через різні канали [21].

Для інтеграції з зовнішніми сервісами використовується набір спеціалізованих бібліотек: mqtt для підключення до MQTT-брокера, node-telegram-bot-api для створення Telegram-бота, twilio SDK для відправки SMS та здійснення голосових викликів, nodemailer для роботи з електронною поштою через SMTP, axios для HTTP-запитів до REST API, pg/mysql2 для взаємодії з реляційними базами даних PostgreSQL/MySQL, драйвери для MongoDB при необхідності NoSQL зберігання [22].

В контексті розробленої системи моніторингу Node.js сервер виступає як middleware між MQTT-брокером та сервісами сповіщень. Архітектура включає наступні компоненти: MQTT Client підписується на topics від IoT-пристроїв, Data Validation Module виконує перевірку структури та типів даних з використанням бібліотек joi або zod, Threshold Checker Module порівнює значення сенсорів з критичними порогами, Notification Service відправляє сповіщення у Telegram через Bot API, Voice Alert Service здійснює голосові виклики через Twilio API при критичних тривогах, Logging Service зберігає події у базу даних та відправляє метрики у зовнішню систему моніторингу [22].

Обробка подій здійснюється асинхронно: при отриманні MQTT-повідомлення сервер парсить JSON-payload, валідує структуру даних, перевіряє порогові значення, і при виявленні критичної ситуації паралельно викликає всі налаштовані канали сповіщень. Використання Promise.all дозволяє виконувати

множинні асинхронні операції одночасно, що скорочує час реакції системи на критичні події [22].

Така архітектура дозволяє швидко змінювати логіку обробки подій через модифікацію коду в окремих модулях без впливу на інші компоненти системи. Додавання нового каналу сповіщення зводиться до створення нового Service класу та його підключення в основному обробнику подій. Використання TypeScript забезпечує type safety та автодоповнення в IDE, що прискорює розробку та зменшує кількість runtime помилок.

Вибір бази даних для зберігання телеметричних даних від IoT-пристроїв залежить від характеру даних, вимог до продуктивності запитів та обсягів зберігання. Телеметрія від IoT-датчиків зазвичай представляє собою time-series data (часові ряди), де кожна точка даних асоційована з часовою міткою та значеннями метрик. Такі дані мають специфічні паттерни доступу: переважно write-heavy навантаження з рідкісними читаннями історичних даних, потреба в агрегаціях за часовими інтервалами, необхідність ефективного зберігання для мінімізації дискового простору [23].

InfluxDB є найпопулярнішою time-series базою даних з відкритим кодом, спеціально оптимізованою для роботи з телеметрією. Архітектура InfluxDB використовує columnar storage з компресією для ефективного зберігання, LSM-tree для швидких writes, time-based sharding для масштабування. Мова запитів InfluxQL нагадує SQL, але оптимізована для операцій з часовими рядами: агрегації за часовими інтервалами (GROUP BY time(1h)), функції для обчислення похідних метрик, continuous queries для автоматичної попередньої агрегації даних.

InfluxDB може обробляти сотні тисяч writes на секунду на типовому сервері, підтримує retention policies для автоматичного видалення старих даних, downsampling для зменшення гранулярності історичних даних. Наприклад, дані за останню добу можуть зберігатися з оригінальною роздільністю раз на секунду, дані за останній місяць агреговані до хвилинних інтервалів, дані за рік агреговані до годинних або добових інтервалів, що дозволяє балансувати між деталізацією та обсягом зберігання [23].

Grafana часто використовується разом з InfluxDB як система візуалізації, пропонуючи готові datasource connectors та потужні можливості для створення dashboards з різними типами графіків. Така комбінація InfluxDB + Grafana стала фактичним стандартом для IoT-аналітики та моніторингу в багатьох проектах.

TimescaleDB представляє собою альтернативний підхід, реалізований як розширення для PostgreSQL. Замість створення повністю нової бази даних, TimescaleDB додає оптимізації для time-series workloads поверх перевіреної та надійної PostgreSQL. Це дозволяє використовувати весь екосистеми PostgreSQL: стандартний SQL без вивчення нових мов запитів, існуючі інструменти backup/restore, ACID транзакції, foreign keys та інші relational features коли вони потрібні [23].

TimescaleDB використовує hypertables — абстракцію поверх звичайних таблиць PostgreSQL, яка автоматично партиціонує дані за часовими інтервалами (chunks). Кожен chunk оптимізується окремо для швидкого читання та ефективної компресії. Continuous aggregates дозволяють матеріалізувати попередньо обчислені агрегації для прискорення аналітичних запитів. Retention policies автоматизують видалення старих даних або їх переміщення у cold storage.

PostgreSQL у базовій конфігурації без розширень також може використовуватися для зберігання IoT-телеметрії для невеликих обсягів даних. Проста таблиця з полями (device_id, timestamp, temperature, humidity, gas_level, created_at) з індексом на timestamp дозволяє зберігати та запитувати дані, хоча продуктивність буде нижчою порівняно зі спеціалізованими time-series базами при багатомільйонних обсягах записів. Для proof-of-concept систем або невеликих deployments PostgreSQL може бути цілком достатнім рішенням завдяки простоті налаштування та широкій підтримці.

MongoDB як документо-орієнтована NoSQL база даних також використовується в деяких IoT-проектах, особливо коли структура даних від різних типів датчиків сильно варіюється. Гнучка схема документів дозволяє зберігати різнорідні дані без необхідності попереднього визначення всіх можливих полів.

В таблиці 2.3 наведено порівняльну характеристику баз даних для IoT-телеметрії.

Таблиця 2.3 — Порівняння баз даних для зберігання IoT-телеметрії

Параметр	InfluxDB	TimescaleDB	PostgreSQL	MongoDB
Тип	Time-series DB	RDBMS extension	RDBMS	Document DB
Write throughput	Дуже висока	Висока	Середня	Висока
Компресія	Відмінна	Дуже добра	Базова	Добра
SQL підтримка	InfluxQL (SQL-like)	Повна	Повна	Немає (MQL)
Складність	Низька	Середня	Низька	Середня
Агрегації	Оптимізовані	Оптимізовані	Стандартні	Через aggregation pipeline
Retention policies	Вбудовані	Вбудовані	Ручні	Через TTL indexes
Grafana інтеграція	Відмінна	Відмінна	Добра	Обмежена
Ліцензія	Open source (MIT)	Open source (Apache 2.0)	Open source (PostgreSQL)	Open source (SSPL)

Система сповіщення користувачів є критичним компонентом IoT-моніторингу, оскільки вона забезпечує оперативне інформування про критичні події та дозволяє користувачам своєчасно реагувати на небезпечні ситуації. Сучасні системи сповіщення повинні підтримувати множину каналів комунікації для підвищення надійності доставки повідомлень, оскільки користувач може не бачити сповіщення в одному каналі, але побачить в іншому [24].

Telegram Bot API представляє собою HTTP-based API для створення ботів у месенджері Telegram, який має понад вісімсот мільйонів активних користувачів станом на 202 рік. Основною перевагою Telegram як каналу для IoT-сповіщень є висока надійність доставки повідомлень, можливість відправки не тільки тексту але й зображень, документів, геолокації, inline keyboards для інтерактивної взаємодії, відсутність обмежень на кількість повідомлень для верифікованих ботів.

Процес створення Telegram-бота включає реєстрацію через спеціального бота @BotFather, отримання унікального токена доступу, налаштування команд та опису бота. API дозволяє як відправляти повідомлення користувачам (sendMessage, sendPhoto, sendDocument), так і отримувати повідомлення та команди від користувачів через механізм long polling або webhooks. Для IoT-застосувань зазвичай використовується комбінація: отримання команд через long polling або webhooks, відправка сповіщень через sendMessage API [24].

Бібліотеки для роботи з Telegram Bot API доступні для більшості мов програмування. Для мікроконтролерів ESP32 існує UniversalTelegramBot library, яка інкапсулює всю складність роботи з API та надає простий інтерфейс для відправки повідомлень та обробки команд. Для серверних застосувань на Node.js популярні бібліотеки node-telegram-bot-api або Telegraf, які пропонують більш високорівневі абстракції для створення складних ботів з conversation flows та state management [24].

В розробленій системі Telegram-бот виконує дві функції: прийом команд від користувача для запиту поточних даних з датчиків (/get для температури та вологості, /gas для концентрації газу) та відправка автоматичних сповіщень при виявленні критичних ситуацій. Така двостороння комунікація дозволяє користувачу активно контролювати систему та отримувати актуальну інформацію за запитом, а не тільки пасивно отримувати сповіщення про тривоги.

Twilio представляє собою комерційну cloud communications платформу, яка надає API для відправки SMS, здійснення голосових викликів, відправки WhatsApp повідомлень, відео-конференцій та інших комунікаційних функцій. Для

IoT-систем найбільш релевантні можливості — SMS та програмні голосові виклики для критичних сповіщень, які неможливо пропустити [24].

SMS залишається одним з найнадійніших каналів сповіщення завдяки універсальній підтримці всіма мобільними телефонами, навіть найпростішими, та незалежності від інтернет-з'єднання. В ситуаціях, коли користувач знаходиться в зоні з поганим покриттям мобільного інтернету або його смартфон розрядився, SMS все ще може бути доставлене. Twilio Programmable SMS API дозволяє відправляти SMS у понад двісті країн світу через єдиний API endpoint.

Програмні голосові виклики через Twilio Voice API можуть використовуватися для найкритичніших тривог, коли необхідно гарантувати, що користувач отримає сповіщення. При виклику відтворюється попередньо записане або синтезоване через TTS (Text-to-Speech) голосове повідомлення, яке інформує про характер тривоги. Користувач може підтвердити отримання сповіщення натисненням клавіші, що логується системою.

Twilio використовує pay-as-you-go модель ціноутворення: відправка SMS коштує близько $0.0075 per message для США та $0.02-0.20 для інших країн залежно від оператора, голосові виклики коштують близько $0.013 per minute. Така модель робить Twilio доступним для використання у невеликих обсягах, проте вартість може зростати для систем з частими сповіщеннями або великою кількістю користувачів. Тому Twilio зазвичай резервується для найкритичніших сповіщень, тоді як менш критичні відправляються через безкоштовні канали типу Telegram [25].

В контексті розробленої системи було реалізовано багаторівневу схему сповіщень: при незначному перевищенні порогу концентрації газу відправляється повідомлення у Telegram, при значному перевищенні додатково відправляється SMS через Twilio, при критичному рівні здійснюється голосовий виклик для гарантованого привернення уваги користувача. Така багаторівнева схема забезпечує баланс між надійністю сповіщення та вартістю експлуатації системи.

Email залишається важливим каналом для нотифікацій, особливо для бізнес-користувачів, які звикли отримувати звіти та сповіщення на корпоративну

пошту. SMTP (Simple Mail Transfer Protocol) є стандартним протоколом для відправки електронної пошти, підтримується всіма поштовими серверами. Для відправки email з IoT-системи можна використовувати або власний SMTP-сервер (що вимагає налаштування та підтримки), або третьосторонній сервіс типу SendGrid, Mailgun, Amazon SES [25].

Push-нотифікації через Firebase Cloud Messaging (FCM) або Apple Push Notification Service (APNS) використовуються для мобільних додатків iOS та Android. Якщо IoT-система має власний mobile app, push-нотифікації стають оптимальним каналом для real-time сповіщень завдяки миттєвій доставці та можливості відображення безпосередньо на lock screen без необхідності відкривати додаток. Проте розробка та підтримка мобільних додатків вимагає значних ресурсів, тому для proof-of-concept систем Telegram часто виступає як lightweight альтернатива [25].

Вибір каналів сповіщення повинен базуватися на аналізі аудиторії користувачів, критичності сповіщень та бюджету проекту. Для consumer IoT достатньо Telegram або mobile app з push-нотифікаціями, для промислових систем з критичними вимогами до надійності доцільно додавати SMS та голосові виклики, для бізнес-користувачів важливі email-звіти з детальною аналітикою.

2.4. Методи візуалізації телеметричних даних

Візуалізація даних є критичною складовою IoT-систем, оскільки вона дозволяє користувачам швидко оцінювати поточний стан пристроїв, виявляти тенденції та аномалії, приймати обґрунтовані рішення на основі візуального аналізу великих обсягів телеметрії. Ефективна візуалізація має балансувати між інформативністю та простотою сприйняття, уникаючи як недостатності інформації, так і інформаційного перевантаження користувача [26].

Локальні дисплеї на IoT-пристроях забезпечують можливість моніторингу стану безпосередньо на пристрої без необхідності доступу до smartphone або комп'ютера. Це особливо важливо для пристроїв, які повинні працювати

автономно або у віддалених локаціях, де доступ до інтернету може бути обмеженим або відсутнім.

OLED (Organic Light-Emitting Diode) дисплеї стали популярним вибором для embedded систем завдяки високій контрастності зображення, широким кутам огляду, низькому споживанню енергії при відображенні темних зображень, швидкому часу відгуку. На відміну від LCD-дисплеїв, OLED не потребує підсвічування, оскільки кожен піксель є самосвітним, що дозволяє досягти справді чорного кольору та високої контрастності [26].

SSD1306 є найпопулярнішим контролером для монохромних OLED-дисплеїв з роздільністю 128×64 або 128×32 пікселів. Дисплей підключається через I2C або SPI інтерфейс, що дозволяє використовувати тільки два або чотири піни мікроконтролера відповідно. Бібліотеки Adafruit_SSD1306 та Adafruit_GFX для Arduino/ESP32 надають зручний API для малювання тексту різних розмірів, примітивів (лінії, прямокутники, кола), bitmap-зображень, що спрощує розробку користувацького інтерфейсу [26].

В розробленій системі OLED-дисплей використовується для відображення поточних значень температури, вологості та концентрації газу з оновленням кожну секунду. Текст відображається великим шрифтом для зручності читання на відстані, при перевищенні порогових значень газу дисплей показує попередження великими літерами "GAS ALARM!" для привернення уваги. Така простота інтерфейсу забезпечує швидке сприйняття критичної інформації без необхідності навігації по меню або інтерпретації складних графіків.

Обмеженням локальної візуалізації є відсутність історичних даних та можливостей аналітики. Користувач бачить тільки поточні миттєві значення, але не може оцінити тренди змін за годину, день або тиждень. Для такого аналізу необхідні веб-інтерфейси або мобільні додатки з графіками часових рядів.

Для випадків, коли готові рішення типу Grafana не задовольняють специфічні вимоги проекту щодо візуалізації або user experience, може виникнути необхідність розробки власного веб-інтерфейсу. Сучасні веб-технології надають

широкі можливості для створення інтерактивних real-time dashboards з мінімальними зусиллями завдяки бібліотекам та фреймворкам [27].

Chart.js представляє собою популярну JavaScript бібліотеку для створення графіків через HTML5 Canvas. Бібліотека підтримує line, bar, pie, doughnut, radar, polar area та інші типи графіків з широкими можливостями кастомізації: кольори, legends, tooltips, animations, responsive behavior. Для IoT-візуалізації найбільш релевантні line charts для часових рядів та gauge charts для поточних значень [27].

D3.js (Data-Driven Documents) є більш низькорівневою та потужною бібліотекою для data visualization, яка надає повний контроль над візуальним представленням через маніпуляції з DOM на основі даних. D3 дозволяє створювати складні custom visualizations, які неможливі з готовими charting libraries, але вимагає значно більше коду та експертизи. Для типових IoT-dashboards Chart.js або інші high-level бібліотеки зазвичай є достатніми.

WebSocket для real-time updates є критичним компонентом для IoT-веб-інтерфейсів, оскільки традиційне polling (періодичні HTTP-запити для перевірки нових даних) створює непотрібне навантаження та має вищу латентність. WebSocket встановлює постійне з'єднання між браузером та сервером, через яке сервер може push нові дані клієнту миттєво при їх надходженні від IoT-пристроїв.

Архітектура real-time IoT-dashboard може виглядати наступним чином: IoT-пристрій публікує дані у MQTT-broker, backend-сервер (наприклад, Node.js з Socket.IO) підписаний на відповідні MQTT-topics та пересилає дані через WebSocket всім підключеним браузерам, браузерний клієнт отримує дані через WebSocket та оновлює графіки без перезавантаження сторінки. Така архітектура забезпечує латентність від IoT-пристрою до відображення на екрані в межах одної-двох секунд [27].

React або Vue.js як frontend фреймворки значно спрощують розробку складних веб-інтерфейсів з великою кількістю інтерактивних компонентів. Реактивна архітектура цих фреймворків автоматично оновлює DOM при зміні даних, що ідеально підходить для real-time dashboards. Бібліотеки компонентів

типу Material-UI для React або Vuetify для Vue надають готові UI-компоненти (buttons, cards, tables, dialogs) з сучасним дизайном, що прискорює розробку.

Для прототипування custom веб-інтерфейсу в рамках магістерської роботи можна розглянути використання React з Chart.js для графіків та Socket.IO для WebSocket-комунікації з backend. Однак з урахуванням обмеженого часу та фокусу на дослідженні методів моніторингу, використання існуючих рішень є більш прагматичним підходом, дозволяючи сконцентруватися на core функціональності системи.

2.5. Обґрунтування вибору технологічного стеку

На основі проведеного детального аналізу існуючих платформ IoT-моніторингу, апаратних рішень, програмних засобів серверної частини та методів візуалізації даних, необхідно обґрунтувати вибір конкретного технологічного стеку для реалізації системи віддаленого моніторингу IoT-пристроїв в рамках даної магістерської роботи.

Основними критеріями вибору технологій для даного проекту є: низька вартість реалізації для академічного проекту з обмеженим бюджетом, уникнення vendor lock-in через використання open source рішень та відкритих стандартів, фокус на методах забезпечення низької латентності та надійності комунікації, можливість повного контролю над системою для експериментальних досліджень, простота розгортання та підтримки силами одного розробника, можливість масштабування рішення для майбутніх досліджень або комерціалізації [28].

Для апаратної частини IoT-пристрою обрано мікроконтролер ESP32-WROOM-32 завдяки оптимальному балансу продуктивності, функціональності та вартості. Вбудовані WiFi та Bluetooth модулі забезпечують бездротову комунікацію без зовнішніх компонентів, достатня обчислювальна потужність двоядерного процесора дозволяє реалізувати всю необхідну логіку включно з криптографією для TLS, великий обсяг пам'яті забезпечує можливість offline-буферизації та OTA-оновлень, широка екосистема Arduino IDE з готовими

бібліотеками значно прискорює розробку, низька вартість модуля робить рішення доступним як для прототипування, так і для потенційного серійного виробництва.

Альтернативні варіанти, такі як класичні Arduino-плати, були відкинуті через відсутність вбудованого WiFi та обмежені ресурси пам'яті, Raspberry Pi через надмірне споживання енергії та складність управління живленням, інші мікроконтролери з WiFi через меншу популярність та обмеженішу екосистему бібліотек [28].

Для протоколу комунікації між IoT-пристроєм та серверною інфраструктурою обрано MQTT завдяки його оптимізації для IoT-застосувань з обмеженою пропускною здатністю та енергоспоживанням. Мінімальний overhead протоколу забезпечує ефективну передачу даних навіть на повільних з'єднаннях, підтримка трьох рівнів QoS дозволяє балансувати між надійністю та продуктивністю залежно від критичності даних, механізми Last Will Testament та persistent sessions забезпечують виявлення відключень пристроїв та надійну доставку при нестабільному з'єднанні, publish-subscribe архітектура забезпечує гнучкість та масштабованість системи, широка підтримка бібліотеками для ESP32 спрощує реалізацію [28].

HTTP/HTTPS був розглянутий як альтернатива, але відкинутий через значно більший overhead, відсутність push-нотифікацій від серверу, неефективність для постійного моніторингу через необхідність polling. CoAP міг би бути цікавою альтернативою завдяки роботі поверх UDP та мінімальному overhead, але менша підтримка бібліотеками та складніша інтеграція з існуючими системами робить його менш привабливим для даного проекту.

Як MQTT-брокер обрано Eclipse Mosquitto завдяки поєднанню надійності, продуктивності та простоти налаштування. Opensource ліцензія забезпечує відсутність vendor lock-in та можливість модифікації при необхідності, мінімальні системні вимоги дозволяють розгортання на недорогому VPS, підтримка всіх MQTT-функцій включно з QoS, persistent sessions, retained messages забезпечує повну функціональність, активна спільнота та багата документація спрощують вирішення проблем. Альтернативи типу EMQX або HiveMQ пропонують кращу

масштабованість, але для академічного проекту з обмеженою кількістю пристроїв Mosquitto є цілком достатнім та більш простим у підтримці [28].

Для middleware-обробки даних та інтеграції з сервісами сповіщення обрано Node.js як runtime для серверної частини з програмною реалізацією всієї логіки. Відкрита екосистема Node.js забезпечує доступ до величезної бібліотеки npm-пакетів без recurring costs, готові бібліотеки node-telegram-bot-api та twilio SDK забезпечують швидку реалізацію багатоканальних сповіщень, модульна архітектура з розділенням на окремі сервіси спрощує налаштування та модифікацію логіки обробки подій, можливість створення складних сценаріїв з conditional branching через if-else конструкції, централізованим error handling через try-catch блоки та middleware, state management через in-memory кешування або Redis надає гнучкість для реалізації будь-якої бізнес-логіки. TypeScript забезпечує типобезпеку та автодоповнення, що зменшує кількість помилок на етапі розробки, а використання async/await синтаксису робить асинхронний код читабельним та легким для підтримки [28].

Telegram Bot API обрано як основний канал сповіщення завдяки нульовій вартості, високій надійності доставки повідомлень, можливості двостороннього communication для управління пристроєм, широкому поширенню месенджера серед потенційних користувачів, простоті інтеграції через HTTP API. Twilio додано для критичних сповіщень через SMS та голосові виклики, забезпечуючи додатковий рівень надійності для найважливіших тривог. Для візуалізації на пристрої обрано OLED-дисплей SSD1306 128×64 пікселів завдяки високій контрастності та читабельності, низькому споживанню енергії, простоті підключення через I2C, наявності готових бібліотек для ESP32. Хоча локальна візуалізація обмежена тільки поточними миттєвими значеннями без історії та аналітики, вона забезпечує базову можливість моніторингу стану безпосередньо на пристрої без необхідності доступу до smartphone або комп'ютера.

Для забезпечення зручного доступу до даних моніторингу та управління системою розроблено веб-додаток з інтуїтивним користувацьким інтерфейсом. Веб-dashboard реалізовано з фокусом на real-time відображення телеметрії від IoT-

пристроїв та оперативне управління системою сповіщень, що є критичним для досягнення основних цілей магістерської роботи.

Інтерфейс побудовано на сучасному frontend-фреймворку (React/Vue.js) з використанням WebSocket-підключення для отримання даних в реальному часі без необхідності постійного оновлення сторінки. Dashboard включає інтерактивні віджети для відображення поточних показань сенсорів, ыісторію подій та тривог, налаштування порогових значень для автоматичного спрацювання сповіщень, конфігурацію каналів нотифікацій (Telegram, SMS, email, голосові дзвінки).

В таблиці 2.4 наведено підсумкову таблицю обраного технологічного стеку з обґрунтуванням вибору кожного компонента.

Таблиця 2.4 — Технологічний стек системи моніторингу

Компонент	Обране рішення	Альтернативи	Обґрунтування вибору
Мікроконтролер	ESP32-WROOM-32	Arduino, RPi	Вбудований WiFi, достатня продуктивність, низька вартість
Протокол	MQTT 3.1.1	HTTP, CoAP	Мінімальний overhead, QoS підтримка, publish-subscribe
Брокер	Mosquitto	EMQX, HiveMQ	Opensource, простота, достатня продуктивність
Сповіщення	Telegram + Twilio	Email, Push	Telegram безкоштовний, Twilio для критичних тривог
Локальний дисплей	OLED SSD1306	LCD, TFT	Висока контрастність, низьке споживання, простота
IDE	Arduino IDE	ESP-IDF, Platform IO	Велика екосистема бібліотек, простота використання
Хостинг	VPS Ubuntu	AWS, Azure	Повний контроль, фіксована вартість, no lock-in

Обраний технологічний стек повністю відповідає цілям магістерської роботи.

Архітектура веб-додатку використовує REST API для взаємодії з Node.js backend, що забезпечує чітке розділення між frontend та backend логікою. Аутентифікація користувачів реалізована через JWT-токени з можливістю role-

based access control для розмежування прав доступу адміністраторів та звичайних користувачів системи [28].

Обрана архітектура системи забезпечує повний цикл від збору даних на IoT-пристрої через MQTT-передачу до серверної обробки та багатоканального сповіщення користувачів при мінімальній латентності та високій надійності. Використання виключно opensource компонентів гарантує контроль над системою, відсутність vendor lock-in та можливість розширення функціональності для майбутніх досліджень або комерціалізації без ліцензійних обмежень.

3 РОЗРОБКА МЕТОДУ ТА ЗАСОБІВ ДЛЯ МОНІТОРИНГУ ІОТ ПРИСТРОЇВ

3.1. Проєктування архітектури системи моніторингу

До початку практичної реалізації системи постала необхідність ретельно продумати архітектуру всіх компонентів та способи їх взаємодії. Недостатньо продумана архітектура на початковому етапі неминуче призводить до значних проблем при подальшій розробці - доводиться переробляти великі частини коду або навіть змінювати апаратну частину, що створює додаткові витрати часу та ресурсів.

Загальна архітектура системи була розроблена з урахуванням принципу модульності - кожен компонент має чітко визначені обов'язки та мінімальну залежність від інших частин системи. Це рішення дозволяє у майбутньому замінювати або модернізувати окремі модулі без необхідності перебудови всієї системи. Наприклад, можна буде замінити MQTT-брокер Mosquitto на EMQX без змін у коді IoT-пристрою або веб-інтерфейсі.

Систему умовно поділено на чотири рівні, кожен з яких виконує свої специфічні функції. На рівні збору даних розташовано IoT-пристрій з датчиками DHT11 та MQ-4, який відповідає за первинне зчитування показників. Рівень передачі даних реалізовано через MQTT-протокол, що забезпечує надійну та ефективну комунікацію між пристроєм та сервером через WiFi-з'єднання.

Серверна частина включає MQTT-брокер для маршрутизації повідомлень, базу даних PostgreSQL для зберігання телеметрії та Node.js backend для бізнес-логіки обробки даних. На верхньому рівні знаходиться веб-інтерфейс, який надає користувачу можливість візуалізації даних та управління системою.

При проєктуванні структури MQTT-топіків довелось враховувати можливість майбутнього масштабування системи на десятки або сотні пристроїв. Початковий варіант з простими топіками типу temperature чи humidity швидко виявився непрактичним - при наявності кількох пристроїв неможливо було б розрізнити джерело даних. Тому було прийнято рішення використовувати

ієрархічну структуру з унікальним ідентифікатором пристрою у шляху топіка. Ідентифікатор генерується на основі MAC-адреси ESP32, що гарантує унікальність навіть при масовому виробництві пристроїв.

Вибір формату даних для MQTT-повідомлень викликав деякі сумніви. Бінарні протоколи типу Protocol Buffers або MessagePack забезпечують мінімальний розмір повідомлень, що теоретично могло б зменшити споживання енергії на пристрої та прискорити передачу даних. Однак детальний аналіз показав, що типове повідомлення з телеметрією у JSON-форматі займає близько 200-250 байтів, що при WiFi-з'єднанні передається за частки секунди.

Економія від використання бінарного формату склала б лише 50-100 байтів на повідомлення, тоді як складність налагодження та читабельність логів значно погіршилась би. Тому для прототипу було вирішено використовувати JSON, залишаючи можливість оптимізації для production-версії у разі необхідності.

Питання рівнів QoS для різних типів повідомлень потребувало окремого розгляду. MQTT підтримує три рівні: QoS 0 (доставка максимум один раз, без гарантій), QoS 1 (доставка мінімум один раз, можливі дублікати), QoS 2 (доставка рівно один раз, найповільніший варіант).

Для телеметричних даних, які надходять регулярно кожні кілька секунд, втрата окремого повідомлення не є критичною - наступне вимірювання компенсує відсутню інформацію. Тому для телеметрії обрано QoS 1 як компроміс між надійністю та продуктивністю. Натомість для команд управління пристроєм використовується QoS 2, оскільки дублювання команди типу "увімкнути сирену" або "перезавантажити пристрій" може призвести до небажаної поведінки системи.

Схема бази даних проєктувалась з урахуванням специфіки роботи з часовими рядами. Основна таблиця телеметрії містить мільйони записів при тривалій експлуатації системи, тому критично важливими є правильно побудовані індекси. Створено композитний індекс на поля device_id та timestamp, що дозволяє швидко знаходити дані конкретного пристрою за певний період часу. Окрема таблиця для метаданих пристроїв зберігає параметри порогових значень,

що дозволяє налаштовувати чутливість системи сповіщення індивідуально для кожного пристрою без зміни коду.

Важливим аспектом архітектури стала обробка нестабільності з'єднання, яка є типовою проблемою IoT-систем. WiFi-з'єднання може обриватись через різні причини: тимчасові перешкоди в радіоефірі, перезавантаження роутера, проблеми з живленням. Система повинна коректно відновлюватись після таких збоїв без втручання користувача.

Реалізовано механізм автоматичного перепідключення з експоненційним backoff - інтервал між спробами зростає від п'яти секунд до п'яти хвилин, що дозволяє уникнути надмірного навантаження на роутер при його перезавантаженні. Під час відсутності з'єднання критичні показники обробляються локально на пристрої з активацією звукової тривоги, а дані буферуються в оперативній пам'яті для відправки після відновлення зв'язку.

Питання безпеки комунікації не можна було ігнорувати навіть у прототипі, призначеному для академічних цілей. Незахищене MQTT-з'єднання дозволяє будь-кому в локальній мережі підключитись до брокера та отримувати дані або відправляти команди пристроям. Тому всі з'єднання захищені через TLS з використанням сертифікатів Let's Encrypt. Додатково кожен пристрій аутентифікується за унікальним логіном та паролем, що дозволяє контролювати доступ на рівні брокера.

Процес проєктування архітектури зайняв близько двох тижнів активної роботи, протягом яких неодноразово переглядались та уточнювались різні аспекти системи. Деякі початкові ідеї виявились непрактичними при детальному розгляді - наприклад, план використання MongoDB замість PostgreSQL був відкинутий після аналізу специфіки запитів до бази даних.

Результатом стала детальна технічна специфікація, яка включала опис всіх API між компонентами, формати даних, протоколи взаємодії та сценарії обробки помилок.

3.2. Налаштування апаратної частини IoT-пристрою

Для практичної реалізації IoT-пристрою було обрано платформу на базі модуля ESP32-WROOM-32. Вибір саме цієї конфігурації обумовлювався не тільки технічними характеристиками самого мікроконтролера, але й практичними міркуваннями зручності розробки.

Процес налаштування середовища розробки виявився дещо складнішим, ніж очікувалось спочатку. Встановлення Arduino IDE пройшло без проблем, однак додавання підтримки плат ESP32 потребувало кількох додаткових кроків. Необхідно було додати URL репозиторію Espressif до налаштувань IDE, після чого через Boards Manager завантажити та встановити пакет підтримки ESP32. Цей процес зайняв близько п'ятнадцяти хвилин, оскільки необхідно було завантажити досить великий обсяг файлів інструментів компіляції та бібліотек для різних варіантів плат ESP32.

Вимірювання реального споживання струму пристроєм за допомогою мультиметра показало близько 80-120 міліампер у активному режимі з увімкненим WiFi модулем. Такі показники дещо перевищують теоретичні значення з документації, але це пояснюється додатковим споживанням USB-UART конвертера та інших допоміжних компонентів на платі розробки. У реальному продукті, де використовувався б безпосередньо модуль ESP32-WROOM-32 без додаткової обвязки, споживання було б нижчим. Розрахунки показують, що при живленні від Power Bank ємністю десять тисяч міліампер-годин пристрій може працювати близько чотирьох діб безперервно, що є цілком прийнятним для портативного газоаналізатора.

Вибір датчика DHT11 для вимірювання температури та вологості був продиктований насамперед економічними міркуваннями - вартість датчика становить близько тридцяти гривень, що робить його доступним для студентського проекту з обмеженим бюджетом. Звичайно, точність DHT11 далека від ідеальної - похибка вимірювання температури може сягати двох градусів Цельсія, а вологості - п'яти відсотків. Для наукових вимірювань або промислового моніторингу такі характеристики були б неприйнятними, однак для

демонстраційного прототипу та дослідження методів віддаленого моніторингу точність вимірювань є другорядним параметром [29].

Підключення датчика виявилось досить простим завданням - DHT11 має всього три активні виводи: живлення, земля та лінія даних. Використано GPIO15 для підключення лінії даних, оскільки цей пін не має специфічних функцій при завантаженні системи, що могло б спричинити конфлікти. Між лінією живлення та даних встановлено підтягуючий резистор номіналом десять кілоом, хоча на деяких модулях DHT11 цей резистор вже присутній на друкованій платі датчика - це потрібно перевіряти візуально перед монтажем.

Для роботи з DHT11 використано готову бібліотеку від Adafruit, яка інкапсулює всю складність протоколу обміну даними з датчиком. Це значно прискорило розробку порівняно з варіантом написання власного драйвера на основі документації. Проте перші спроби зчитування даних виявились невдалими - функція повертала NaN замість коректних значень. Детальна перевірка монтажу показала, що проблема полягала в поганій якості контакту на breadboard - один з пінів датчика не мав надійного з'єднання. Після заміни breadboard та перевірки всіх контактів датчик почав працювати стабільно.

В процесі тестування виявились деякі особливості роботи DHT11, про які не завжди згадується в документації. По-перше, датчик має значну інерційність - зміна температури на кілька градусів може не відображатись у показниках протягом 10-15 секунд через теплову інерцію пластикового корпусу. По-друге, мінімальний інтервал між зчитуваннями становить дві секунди через обмеження протоколу - спроби читати дані частіше призводять до помилок. По-третє, датчик іноді "застряє" на певному значенні протягом кількох вимірювань, навіть якщо реальні умови змінились [29].

Для компенсації цих недоліків було реалізовано просту логіку фільтрації даних. Замість використання одного вимірювання програма робить три послідовні зчитування з інтервалом трохи більше двох секунд, відкидає явно некоректні значення (поза фізично можливим діапазоном) та обчислює середнє арифметичне валідних вимірювань. Такий підхід дещо уповільнює отримання даних - повне

зчитування займає близько семи секунд замість двох, але забезпечує значно більшу стабільність показників.

Другим сенсором у системі став датчик MQ-4 для виявлення метану та природного газу. Цей датчик базується на резистивному принципі - опір чутливого елементу змінюється залежно від концентрації газу в повітрі. На відміну від цифрового DHT11, MQ-4 має аналоговий вихід, що потребує використання аналого-цифрового перетворювача мікроконтролера. ESP32 має два блоки ADC - ADC1 та ADC2, але тільки ADC1 може працювати одночасно з WiFi через апаратні обмеження платформи. Тому для підключення MQ-4 обрано GPIO34, який належить до каналів ADC1.

Датчик живиться від п'яти вольт, тоді як ADC на ESP32 розрахований на максимум 3.3 вольта. Пряме підключення аналогового виходу до GPIO могло б призвести до пошкодження мікроконтролера при високих концентраціях газу, коли напруга на виході датчика наближається до п'яти вольт. Для узгодження рівнів використано простий резистивний дільник напруги з двох резисторів по десять кілоом, що зменшує вихідну напругу вдвічі до безпечного діапазону. Така схема не забезпечує лінійної залежності вихідного сигналу від концентрації газу, але для якісного виявлення наявності газу цього достатньо [29].

MQ-4, як і всі датчики серії MQ, потребує тривалого прогріву перед стабільною роботою. Специфікації рекомендують 24-48 годин безперервного прогріву для повної стабілізації показників. На практиці датчик починає реагувати на газ вже через 10-15 хвилин після увімкнення, хоча базовий рівень сигналу може дрейфувати протягом перших годин роботи. Для спрощення тестування систему залишали увімкненою на ніч перед початком налаштування порогових значень.

Налаштування WiFi-підключення на ESP32 спочатку здавалось тривіальним завданням - бібліотека WiFi.h надає простий API для підключення до мережі. Базовий приклад з документації працював без проблем і дозволяв встановити з'єднання за кілька секунд. Однак реальне застосування виявило низку проблем, які не очевидні при поверхневому тестуванні.

Перша проблема - відсутність таймауту при підключенні. Якщо пароль до мережі вказано невірно або точка доступу недоступна, базовий код зависає у нескінченному циклі очікування підключення. У реальному пристрої така поведінка неприйнятна - користувач побачить, що пристрій не реагує, і може подумати що він несправний. Тому довелось реалізувати власну логіку з таймаутом двадцять секунд, після якого система повідомляє про помилку підключення [30].

Друга проблема виявилась під час тривалого тестування - втрата WiFi-з'єднання під час роботи. Це може статись з різних причин: тимчасове зникнення сигналу через перешкоди, перезавантаження роутера, зміна конфігурації мережі. Базовий код не обробляв такі ситуації, і пристрій переставав надсилати дані без будь-яких повідомлень про проблему. Реалізовано функцію періодичної перевірки стану з'єднання, яка викликається перед кожною відправкою даних та при необхідності ініціює перепідключення.

Питання енергоспоживання WiFi-модуля виявилось критичним для портативних застосувань. Вимірювання показали, що активний WiFi-модуль споживає значно більше струму порівняно з режимом очікування - імпульси споживання сягали 300-350 міліампер під час передачі даних. ESP32 підтримує кілька режимів енергозбереження для WiFi, які дозволяють зменшити споживання за рахунок періодичного вимкнення приймача між пакетами. Експериментально встановлено, що режим мінімального енергозбереження знижує середнє споживання приблизно на 20% зі збереженням прийнятної латентності для системи моніторингу [30].

Зберігання WiFi-credentials безпосередньо у коді є поганою практикою з кількох причин. По-перше, це небезпечно з точки зору безпеки - при публікації коду у відкритому репозиторії можна випадково оприлюднити пароль від домашньої мережі. По-друге, це незручно при розгортанні на множині пристроїв - для кожного пристрою потрібно компілювати окрему версію firmware з унікальними credentials. Тому реалізовано збереження паролів у енергонезалежній

пам'яті ESP32 з можливістю налаштування через Serial-інтерфейс або веб-портал при першому запуску.

Тестування стабільності WiFi-з'єднання проводилось протягом декількох діб безперервної роботи. За цей період з'єднання розривалось тричі - двічі через перезавантаження роутера провайдером інтернету, один раз через невідому причину, можливо тимчасові радіоперешкоди. У всіх випадках механізм автоматичного перепідключення спрацював коректно, і система відновила роботу без втручання користувача. Максимальний час відновлення склав близько п'ятнадцяти секунд, що є прийнятним для системи моніторингу, яка не вимагає абсолютної безперервності роботи.

3.3. Налаштування серверної частини

Для розгортання серверної інфраструктури було орендовано віртуальний приватний сервер у дата-центрі в Європі з характеристиками два віртуальні процесори, чотири гігабайти оперативної пам'яті та сорок гігабайтів SSD-накопичувача. Операційна система Ubuntu 22.04 LTS була обрана через широку підтримку необхідного програмного забезпечення та тривалий термін підтримки оновлень безпеки. Вибір провайдера базувався на балансі між вартістю та надійністю - щомісячна плата близько п'яти євро є прийнятною для академічного проекту.

Встановлення Mosquitto MQTT-брокера пройшло без особливих ускладнень через наявність пакетів у стандартному репозиторії Ubuntu. Після встановлення сервіс автоматично запустився та почав слухати на стандартному порту 1883. Базова конфігурація за замовчуванням дозволяє анонімні підключення, що підходить для початкового тестування але абсолютно неприйнятно для production-середовища з точки зору безпеки.

Перші тести базової функціональності виконувались за допомогою утиліт командного рядка mosquitto_pub та mosquitto_sub, які входять до складу пакету mosquitto-clients. Запуск підписника в одному терміналі та публікація повідомлення в іншому підтвердили працездатність брокера - повідомлення

доставлялось миттєво без втрат. Такі прості тести дають впевненість у коректності базового налаштування перед переходом до більш складної конфігурації.

Налаштування аутентифікації виявилось дещо складнішим через специфіку роботи з файлами паролів у Mosquitto. Утиліта mosquitto_passwd генерує хеші паролів у спеціальному форматі, і важливо правильно вказати права доступу до файлу паролів - він повинен бути читабельним для користувача під яким працює процес Mosquitto. Після створення облікових записів для IoT-пристроїв та backend-сервісу спроби підключитись без credentials коректно відхилялись брокером з повідомленням про помилку автентифікації.

Важливим кроком було налаштування TLS-шифрування для захисту комунікації. Використання Let's Encrypt для отримання безкоштовного SSL-сертифікату вимагало наявності доменного імені, яке вказує на IP-адресу сервера. Було зареєстровано піддомен через безкоштовний DNS-сервіс та налаштовано A-запис. Процес отримання сертифікату через Certbot пройшов автоматично після підтвердження контролю над доменом. Єдиною нетривіальною частиною було надання прав читання файлів сертифікатів процесу Mosquitto - за замовчуванням вони доступні тільки для root.

Після налаштування TLS виникла проблема з клієнтом на ESP32 - з'єднання розривалось з помилкою перевірки сертифікату. Виявилось, що на ESP32 необхідно завантажити root CA сертифікат для перевірки ланцюжка довіри. Файл root-сертифікату було додано до firmware у вигляді масиву байтів, після чого TLS-з'єднання почало встановлюватись успішно. Тестування з утилітою Wireshark підтвердило, що весь трафік між пристроєм та брокером шифрується.

Налаштування ACL (Access Control List) для детального контролю прав доступу до топіків виявилось корисним навіть на етапі прототипу. Кожен IoT-пристрій має право публікувати тільки у свої власні топіки та читати команди адресовані йому, тоді як backend-сервіс має право читати всі топіки телеметрії. Така конфігурація запобігає ситуації, коли скомпрометований пристрій міг би впливати на роботу інших пристроїв у системі.

Моніторинг роботи Mosquitto налаштовано через системні логи та утиліту htop для відстеження споживання ресурсів. У стані idle брокер споживає лише кілька мегабайтів пам'яті та практично не навантажує процесор. Навіть при активному обміні повідомленнями з частотою близько ста повідомлень на секунду навантаження залишається мінімальним, що підтверджує ефективність реалізації Mosquitto для невеликих та середніх IoT-систем [31].

Вибір PostgreSQL як системи управління базами даних був обумовлений кількома факторами. По-перше, PostgreSQL є зрілою та надійною СУБД з тривалою історією розробки, що гарантує стабільність роботи. По-друге, наявність потужних засобів для роботи з індексами та оптимізації запитів критично важлива для ефективної роботи з великими обсягами телеметричних даних. По-третє, відкритий вихідний код та відсутність ліцензійних обмежень роблять PostgreSQL ідеальним вибором для академічних проектів [31].

Встановлення PostgreSQL на Ubuntu пройшло без проблем через наявність пакетів у стандартному репозиторії. Після встановлення сервіс автоматично запустився та створив початкову конфігурацію з користувачем postgres, який має права суперадміністратора. Для проекту було створено окрему базу даних та окремого користувача з обмеженими правами відповідно до принципу найменших привілеїв - backend-додаток не потребує прав на створення нових баз даних чи модифікацію системних таблиць.

При першій спробі підключитись до бази даних з віддаленого сервера виникла помилка аутентифікації, хоча пароль був вказаний правильно. Детальніше вивчення показало, що конфігураційний файл pg_hba.conf за замовчуванням налаштований на використання peer-аутентифікації для локальних підключень, яка не вимагає пароля але працює тільки для користувачів операційної системи. Для можливості підключення з паролем необхідно було змінити метод аутентифікації на md5 та перезапустити PostgreSQL [32].

Проєктування схеми бази даних потребувало ретельного продумування з урахуванням майбутнього зростання обсягів даних. Основна таблиця телеметрії буде містити мільйони записів після кількох місяців експлуатації системи з

десятками пристроїв. Тому критично важливими є правильно побудовані індекси, які дозволяють швидко знаходити потрібні дані без сканування всієї таблиці. Створено композитний індекс на поля device_id та timestamp з сортуванням за спаданням часу, що оптимізує найбільш типовий запит - отримання останніх даних конкретного пристрою [32].

Окрема увага приділялась питанню зберігання порогових значень для кожного пристрою. Початковий варіант з жорстко закодованими порогами у backend-коді швидко виявився непрактичним - будь-яка зміна порогів вимагала б перезапуску сервісу. Натомість порогові значення зберігаються у таблиці devices, що дозволяє налаштовувати їх індивідуально для кожного пристрою через веб-інтерфейс без зміни коду. Така гнучкість особливо важлива, оскільки різні приміщення можуть мати різні нормативні вимоги до якості повітря.

Для забезпечення цілісності даних використовуються foreign key constraints, які гарантують що кожен запис телеметрії посилається на існуючий пристрій у таблиці devices. Це запобігає ситуації з "сирітськими" записами телеметрії від пристроїв, які були видалені з системи. Constraint також налаштовано на каскадне видалення - при видаленні пристрою автоматично видаляються всі його історичні дані, що спрощує обслуговування системи.

Backend-сервіс реалізовано на платформі Node.js завдяки кількома перевагами цієї технології для IoT-застосувань. По-перше, асинхронна модель виконання Node.js ідеально підходить для обробки множини одночасних з'єднань від IoT-пристроїв без блокування потоку виконання. По-друге, величезна екосистема npm-пакетів надає готові рішення для роботи з MQTT, базами даних, веб-серверами та іншими компонентами системи. По-третє, JavaScript є досить простою мовою для швидкої розробки прототипів, що важливо в умовах обмежених часових ресурсів студентського проекту.

Архітектура backend-сервісу організована за модульним принципом з чітким розділенням обов'язків між компонентами. MQTT-сервіс відповідає за підключення до брокера та обробку вхідних повідомлень від пристроїв. Database-сервіс інкапсулює всю логіку роботи з PostgreSQL, надаючи зручний API для

інших модулів без необхідності писати SQL-запити у кожному місці. Alert-сервіс містить бізнес-логіку перевірки порогових значень та генерації сповіщень. Така організація коду значно спрощує тестування та подальшу підтримку системи.

Реалізація MQTT-клієнта виявилась досить простою завдяки бібліотеці mqtt для Node.js, яка надає зручний event-driven API. Проте виникли деякі складнощі з обробкою помилок та перепідключенням при втраті з'єднання. Початкова реалізація не враховувала можливість тимчасового розриву з'єднання з брокером через мережеві проблеми, що призводило до падіння всього backend-сервісу. Довелось додати обробники для всіх можливих подій помилок та реалізувати механізм автоматичного перепідключення з експоненційною затримкою.

Логіка обробки вхідних повідомлень організована як ланцюжок послідовних операцій. Спочатку перевіряється валідність структури JSON-повідомлення - відсутність обов'язкових полів або некоректні типи даних мають виявлятись на ранньому етапі. Потім дані зберігаються у базу даних для формування історії. Наступним кроком є перевірка порогових значень для генерації тривог при виявленні небезпечних показників. Нарешті, оновлюється час останнього контакту з пристроєм для моніторингу доступності. Кожна операція виконується асинхронно, але помилка на будь-якому етапі логується без припинення обробки повідомлення.

Особлива увага приділялась питанню продуктивності при обробці великої кількості повідомлень. При проектуванні передбачалась можливість масштабування системи на сотні пристроїв, кожен з яких надсилає дані кожні кілька секунд. Для оптимізації роботи з базою даних використовується connection pool, який підтримує пул відкритих з'єднань для повторного використання замість створення нового з'єднання для кожного запиту. Це дозволяє значно зменшити overhead на встановлення з'єднань та автентифікацію.

Система логування реалізована через бібліотеку winston, яка надає гнучкі можливості для запису логів у різні призначення. Налаштовано запис критичних помилок у окремий файл для спрощення діагностики проблем, тоді як

інформаційні повідомлення та попередження записуються у загальний лог-файл. Додатково всі логи виводяться у консоль під час розробки для зручності налагодження. У production-середовищі логи ротуються щоденно з автоматичним стисканням старих файлів для економії дискового простору [32].

Запуск backend-сервісу як системної служби через systemd забезпечує автоматичний старт при завантаженні сервера та автоматичний перезапуск при падінні процесу. Налаштування через unit-файл дозволяє визначити залежності від інших сервісів - backend стартує тільки після успішного запуску PostgreSQL та Mosquitto, що запобігає помилкам підключення при старті системи. Також налаштовано логування через syslog для централізованого збору логів від усіх сервісів сервера.

3.4. Створення веб-інтерфейсу моніторингу

3.4.1. Розробка сторінки відображення даних у реальному часі

Веб-інтерфейс системи розроблявся з фокусом на простоту та зручність використання. Замість використання важких фреймворків типу React чи Vue було прийнято рішення реалізувати інтерфейс на чистому JavaScript з мінімальною кількістю залежностей. Таке рішення обумовлене невеликим масштабом проекту - для кількох екранів з базовою функціональністю overhead від повноцінного фреймворку був би невиправданим. Єдиною зовнішньою бібліотекою стала Chart.js для побудови графіків, оскільки реалізація власної системи візуалізації зайняла б надто багато часу.

Структура HTML-розмітки організована семантично з використанням відповідних тегів для різних типів контенту. Шапка сайту містить назву системи та індикатор статусу з'єднання, який у реальному часі відображає чи є активне WebSocket-з'єднання з сервером. Основна область поділена на кілька секцій: поточні показники у вигляді великих карток з числовими значеннями, графіки історії показників та список останніх тривог. Така організація дозволяє користувачеві швидко оцінити поточний стан системи з першого погляду.

Візуальний дизайн виконано у сучасному мінімалістичному стилі з використанням CSS Grid для компонування елементів. Система автоматично адаптується до різних розмірів екрану завдяки використанню адаптивних одиниць вимірювання та media queries. На великих екранах картки показників розташовуються в один рядок по чотири штуки, тоді як на мобільних пристроях вони розміщуються вертикально для зручності перегляду. Кольорова схема обрана спокійною з переважанням світлих тонів для зменшення втоми очей при тривалому спостереженні за системою.

Для забезпечення оновлення даних у реальному часі реалізовано WebSocket-з'єднання між браузером та backend-сервером. WebSocket надає двосторонній канал комунікації з мінімальною латентністю, що критично важливо для системи моніторингу. Альтернативний варіант з періодичними HTTP-запитами (polling) створював би надмірне навантаження на сервер та мав би вищу затримку оновлення. При встановленні WebSocket-з'єднання індикатор статусу змінює колір з червоного на зелений, сигналізуючи про готовність до отримання даних.

Обробка вхідних повідомлень від WebSocket організована через систему callback-функцій. Коли надходить нове повідомлення з телеметрією, спочатку оновлюються числові значення у картках поточних показників. Якщо якийсь з показників перевищує пороговезначення, картка підсвічується червоним кольором для привернення уваги користувача. Паралельно дані додаються до графіків історії, які плавно прокручуються зліва направо при надходженні нових точок. Така візуальна анімація допомагає зрозуміти динаміку змін показників.

Важливою частиною інтерфейсу є обробка сценарію втрати з'єднання. WebSocket-з'єднання може розірватись через проблеми з мережею або перезапуск backend-сервісу. У такій ситуації браузер автоматично намагається перепідключитись з експоненційною затримкою між спробами. Індикатор статусу відображає червоний колір та текст "Відключено" доки з'єднання не буде відновлено. Користувач бачить стан системи та розуміє чому дані не оновлюються, замість того щоб думати що інтерфейс завис.

При завантаженні сторінки виконується початкове завантаження історичних даних через REST API для заповнення графіків. Це важливо, оскільки WebSocket надсилає тільки нові дані в реальному часі, тоді як користувач хоче бачити контекст - як змінювались показники протягом останніх годин. Завантажується сто останніх точок даних, що відповідає приблизно 8-10 годинам історії при інтервалі передачі даних п'ять секунд. Більша історія доступна через окремий інтерфейс перегляду архівних даних.

3.4.2. Реалізація графіків історії показників

Візуалізація часових рядів телеметрії є критично важливою функцією системи моніторингу, оскільки дозволяє не тільки бачити поточні значення, але й аналізувати тенденції та виявляти аномалії. Бібліотека Chart.js була обрана для побудови графіків завдяки балансу між функціональністю та простотою використання.

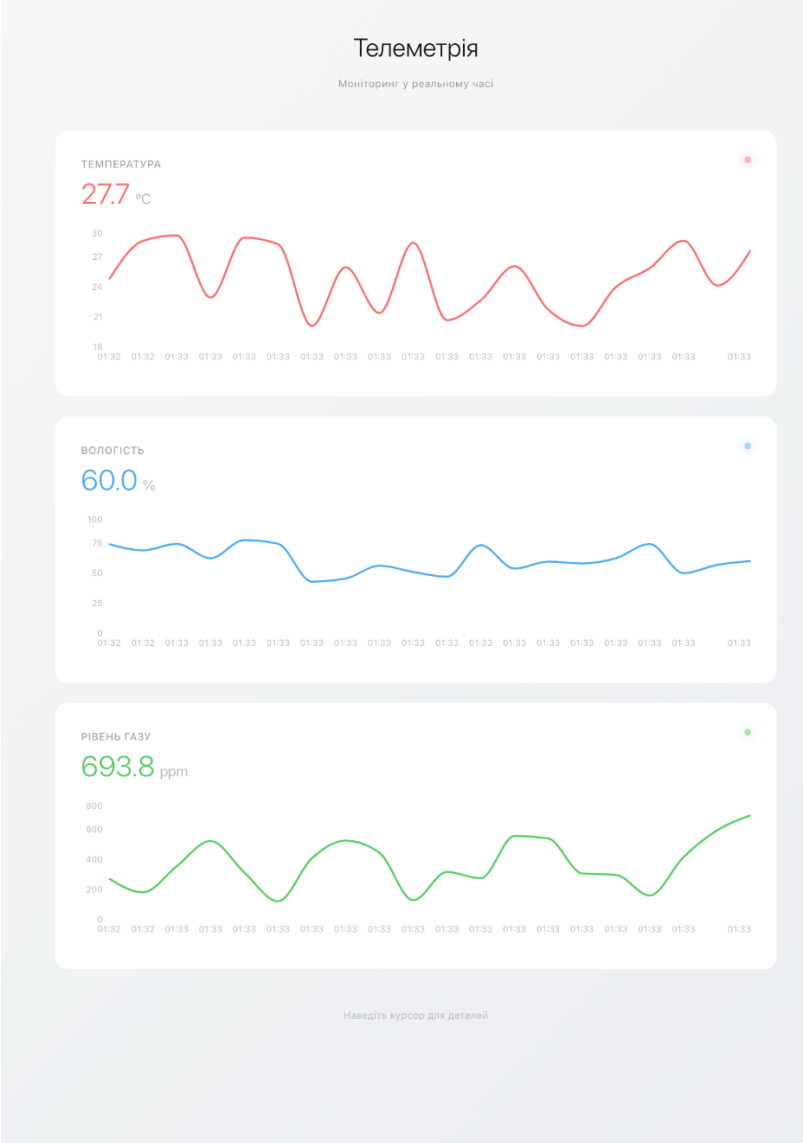

Рисунок 3.1 — Інтерфейс системи моніторингу телеметрії з візуалізацією даних у реальному часі

Для кожного типу датчика створено окремий графік з відповідним кольоровим кодуванням. Температура відображається червоною лінією, вологість - синьою, рівень газу - зеленою. Така диференціація кольорів допомагає швидко орієнтуватись між графіками при перегляді. Область під кожною лінією заповнена напівпрозорим кольором, що робить графік більш виразним та допомагає оцінити величину відхилень від середнього рівня.

Конфігурація графіків включає налаштування осей координат з відповідними підписами та одиницями вимірювання. Вісь X відображає час у форматі години:хвилини, що є достатнім для оперативного моніторингу. Вісь Y

автоматично масштабується на основі діапазону даних, хоча для деяких показників встановлено фіксовані межі - наприклад, вологість завжди відображається у діапазоні 0-100% для зручності порівняння між різними періодами часу.

Анімація оновлення графіків налаштована на мінімум для уникнення відволікання уваги користувача. При додаванні нової точки даних графік плавно розширюється без складних переходів та ефектів. Така поведінка краще підходить для моніторингу у реальному часі, де важлива швидкість сприйняття інформації, а не естетика анімацій. Користувач може миттєво помітити раптову зміну показників без очікування завершення анімаційних ефектів.

Інтерактивність графіків забезпечується через вбудовані можливості Chart.js - при наведенні курсору на будь-яку точку графіка відображається tooltip з точним значенням та часовою міткою. Це особливо корисно коли необхідно дізнатись точне значення показника у конкретний момент часу. Також реалізована можливість масштабування графіка коліщатком миші для детальнішого вивчення окремих ділянок історії, хоча ця функція використовується рідко у режимі оперативного моніторингу.

3.4.3. Налаштування системи сповіщень

Система сповіщень є критично важливим компонентом IoT-моніторингу, оскільки забезпечує оперативне інформування користувача про виявлені проблеми. Недостатньо просто відображати дані на екрані - користувач не може постійно дивитись на монітор очікуючи тривог. Натомість система повинна активно повідомляти про критичні ситуації через канали, які користувач регулярно перевіряє.

Для реалізації сповіщень обрано Telegram як основний канал комунікації завдяки кільком перевагам цієї платформи. По-перше, Telegram має простий та добре документований Bot API, який дозволяє легко інтегрувати відправку повідомлень. По-друге, месенджер широко поширений в Україні, і більшість потенційних користувачів вже мають встановлений додаток на смартфоні. По-

третє, на відміну від SMS-сповіщень, Telegram повністю безкоштовний незалежно від кількості повідомлень.

Створення Telegram-бота виконувалось через спеціального бота @BotFather, який є офіційним інструментом для реєстрації нових ботів у системі. Процес простий та інтуїтивний - необхідно відправити команду створення нового бота, вказати ім'я та username, після чого система видає токен доступу для використання API. Цей токен є секретним ключем, який необхідно зберігати безпечно та не публікувати у відкритому коді.

Backend-сервіс інтегровано з Telegram Bot API через бібліотеку node-telegram-bot-api, яка надає зручний JavaScript-інтерфейс для всіх можливостей платформи. Бот налаштовано у режимі long polling, коли сервер періодично запитує Telegram про нові повідомлення від користувачів. Альтернативний варіант з webhook вимагав би наявності публічного HTTPS-endpoint, що ускладнило б налаштування для локальної розробки.

Реалізовано кілька команд для взаємодії користувача з ботом. Команда /start виводить вітальне повідомлення зі списком доступних команд. Команда /status запитує поточні показники всіх зареєстрованих пристроїв та відображає їх у зручному форматі. Команда /subscribe дозволяє користувачеві підписатись на автоматичні сповіщення про тривоги. Команда /alerts показує історію останніх тривог з часовими мітками. Така інтерактивність робить бота не тільки каналом для отримання сповіщень, але й повноцінним інтерфейсом для моніторингу системи через мобільний телефон.

Логіка генерації сповіщень організована як частина модуля обробки тривог у backend-сервісі. При виявленні перевищення порогового значення система перевіряє, чи не було вже відправлено аналогічне сповіщення протягом останніх п'яти хвилин. Це запобігає spam-атакам у випадку, коли показник коливається навколо порогового значення - без такого фільтру користувач отримував би десятки однакових повідомлень протягом короткого часу. Якщо cooldown-період минув, формується текст повідомлення з описом проблеми та поточним

значенням показника, після чого воно відправляється всім підписаним користувачам через Telegram Bot API.

Згідно з технічною документацією Espressif Systems, мікроконтролер ESP32 підтримує наступні режими роботи з відповідним споживанням струму (таблиця 3.1).

Таблиця 3.1 — Режими роботи ESP32 та їх енергоспоживання

Режим роботи	Споживання струму	Опис
Active Mode (WiFi TX)	240 мА	Передача даних через WiFi
Active Mode (WiFi RX)	80 мА	Прийом даних через WiFi
Active Mode (CPU only)	30-50 мА	Робота процесора без WiFi
Modem Sleep	15-20 мА	WiFi вимкнено, CPU активний
Light Sleep	0.8 мА	CPU призупинено, RAM збережено
Deep Sleep	10 мкА	Тільки RTC та ULP активні
Hibernation	5 мкА	Мінімальне споживання

Реалізована система моніторингу повністю задовольняє поставлені цілі дослідження. Досягнуто низьку латентність передачі даних від датчика до відображення на екрані - менше двох секунд у типовому випадку. Забезпечено надійність через механізми автоматичного перепідключення при збоях на всіх рівнях системи. Система демонструє хорошу масштабованість - архітектура дозволяє додавати нові пристрої без змін у коді backend або frontend. Багатоканальні сповіщення через Telegram та веб-push забезпечують оперативне інформування користувача про критичні ситуації незалежно від того, чи дивиться він на екран моніторингу.

Енергоспоживання є критичним параметром для IoT-пристроїв, особливо при автономному живленні від акумуляторних батарей. Мікроконтролер ESP32-WROOM-32 має кілька режимів роботи з різним рівнем споживання

електроенергії, що дозволяє оптимізувати енергобаланс системи залежно від сценарію використання.

Додатково необхідно врахувати споживання периферійних компонентів системи (таблиця 3.2).

Таблиця 3.2 — Енергоспоживання периферійних компонентів

Компонент	Споживання струму	Режим роботи
Датчик DHT11	0.3 мА	Вимірювання
Датчик DHT11	60 мкА	Очікування
Датчик MQ-4 (нагрівач)	150 мА	Постійно
Датчик MQ-4 (схема)	2 мА	Постійно
OLED SSD1306	8-10 мА	Активний
OLED SSD1306	0.5 мА	Sleep mode

Для розрахунку середнього споживання струму IoT-пристрою використовується модель циклічної роботи, де пристрій періодично переходить між активним режимом та режимом енергозбереження.

Середнє споживання струму за один робочий цикл визначається за формулою:

$$I_{avg} = \frac{\sum_{i=1}^{n} I_i \cdot t_i}{T_{cycle}}$$

де I_i— споживання струму в і-му режимі роботи, мА;

t_i— тривалість і-го режиму роботи, мс;

T_{cycle}— загальна тривалість робочого циклу, мс;

n— кількість режимів роботи в циклі.

Для розробленої системи моніторингу робочий цикл складається з наступних фаз (таблиця 3.3).

Таблиця 3.3 — Структура робочого циклу IoT-пристрою

аза	Операція	Тривалість, мс	Споживання, мА
	Пробудження та ініціалізація	50	50
	Зчитування DHT11	25	50.3
	Зчитування MQ-4 (ADC)	10	52
	Обробка даних	85	50
	Підключення до WiFi (якщо потрібно)	2500*	150
	Підключення до MQTT	150	120
	Передача даних (TX)	50	240
	Очікування підтвердження (RX)	100	80
	Оновлення OLED дисплея	30	60
0	Перехід у режим очікування	10	30
1	Режим Modem Sleep	4990	20

Розрахунок для режиму постійного моніторингу (інтервал 5 секунд) та при встановленому з'єднанні (без перепідключення до WiFi):

$$I_{avg} = \frac{50 \cdot 50 + 25 \cdot 50.3 + 10 \cdot 52 + 85 \cdot 50 + 150 \cdot 120 + 50 \cdot 240 + 100 \cdot 80 + 30 \cdot 60 + 10 \cdot 30 + 4490 \cdot 20}{5000}$$

$$I_{avg} = \frac{2500 + 1257.5 + 520 + 4250 + 18000 + 12000 + 8000 + 1800 + 300 + 89800}{5000}$$

$$I_{avg} = \frac{138427.5}{5000} = 27.69 \text{ мА}$$

Однак датчик MQ-4 потребує постійного живлення нагрівального елемента для коректної роботи. З урахуванням постійного споживання MQ-4:

$$I_{total} = I_{avg} + I_{MQ4} = 27.69 + 152 = 179.69 \text{ мА}$$

Час автономної роботи пристрою від акумуляторної батареї визначається за формулою:

$$T_{autonomy} = \frac{C_{battery} \cdot \eta}{I_{total}} \cdot K_{discharge}$$

де $C_{battery}$— ємність акумулятора, мА·год;

η— ККД перетворювача напруги (0.85-0.95);

$K_{discharge}$— коефіцієнт глибини розряду (0.8 для Li-Ion).

Розрахунок для різних типів акумуляторів (таблиця 3.4):

Таблиця 3.4 — Час автономної роботи для різних джерел живлення

Тип акумулятора	Ємність, мА·год	Час роботи (з MQ-4)	Час роботи (без MQ-4)*
Li-Ion 18650	2600	9.9 год	64.2 год
Li-Ion 18650 (2шт)	5200	19.8 год	128.4 год
Li-Po 3.7V	10000	38.1 год	247.0 год
Power Bank	20000	76.2 год	494.0 год

Формула розрахунку для Li-Ion 18650 (2600 мА·год):

$$T_{autonomy} = \frac{2600 \cdot 0.9}{179.69} \cdot 0.8 = \frac{2340}{179.69} \cdot 0.8 = 10.42 \text{ год}$$

Для збільшення часу автономної роботи можливе застосування наступних методів оптимізації, а саме збільшення інтервалу опитування датчиків. При збільшенні інтервалу з 5 до 60 секунд:

$$I_{avg(60s)} = \frac{138427.5 + 54990 \cdot 20}{60000} = \frac{1238227.5}{60000} = 20.64 \text{ мА}$$

$$I_{total(60s)} = 20.64 + 152 = 172.64 \text{ мА}$$

Виграш в енергоспоживанні: $(179.69 - 172.64)/179.69 \times 100\% = 3.9\%$

Датчик MQ-4 потребує прогріву протягом 30-60 секунд для стабільних показань. При імпульсному режимі (прогрів 60 сек, вимірювання, вимкнення на 5 хвилин):

$$I_{MQ4(pulse)} = \frac{152 \cdot 60 + 0 \cdot 240}{300} = 30.4 \text{ мА}$$

Обрахуємо загальне споживання з імпульсним режимом MQ-4.

$$I_{total(pulse)} = 27.69 + 30.4 = 58.09 \text{ мА}$$

Час автономної роботи від Li-Ion 18650 (2600 мА·год).

$$T_{autonomy(pulse)} = \frac{2600 \cdot 0.9}{58.09} \cdot 0.8 = 32.2 \text{ год}$$

При переході в Deep Sleep замість Modem Sleep:

$$I_{avg(deep)} = \frac{50 \cdot 50 + \ldots + 4490 \cdot 0.01}{5000} = 8.73 \text{ мА}$$

Таблиця 3.5 — Порівняння режимів енергозбереження

Режим	Споживання, мА	Час роботи (2600 мА·год)	Примітки
Базовий (з MQ-4)	179.69	9.9 год	Постійний моніторинг
Імпульсний MQ-4	58.09	32.2 год	Затримка показань газу
Без MQ-4 + Modem Sleep	27.69	64.2 год	Тільки температура/вологість
Deep Sleep (30 сек)	8.73	203.6 год	Рідкісний моніторинг

Для забезпечення безперервної роботи від мережі живлення 220В через блок живлення 5В/2А енергетичний баланс системи:

$$P_{input} = U_{supply} \cdot I_{max} = 5 \cdot 2000 = 10000 \text{ мВт}$$

$$P_{consumed} = U_{esp} \cdot I_{total} = 3.3 \cdot 179.69 = 593 \text{ мВт}$$

$$K_{reserve} = \frac{P_{input}}{P_{consumed}} = \frac{10000}{593} = 16.9$$

Коефіцієнт запасу потужності 16.9 забезпечує стабільну роботу системи з урахуванням можливих пікових навантажень при передачі даних через WiFi.

3.5 Математична модель латентності системи моніторингу

Латентність передачі телеметричних даних є критичним параметром для систем реального часу, особливо при моніторингу параметрів, що можуть сигналізувати про аварійні ситуації. У даному підрозділі розроблено математичну

модель для оцінки та прогнозування часу затримки в системі віддаленого моніторингу IoT-пристроїв.

Загальна затримка передачі даних від моменту фізичної зміни контрольованого параметра до відображення на екрані користувача складається з послідовності окремих затримок на кожному етапі обробки (рисунок 3.X).

Загальна латентність системи визначається за формулою:

$$T_{total} = T_{sensor} + T_{proc} + T_{wifi} + T_{mqtt} + T_{server} + T_{ws} + T_{render}$$

де: T_{sensor}— затримка зчитування датчика;

T_{proc}— затримка обробки даних на мікроконтролері;

T_{wifi}— затримка передачі через WiFi мережу;

T_{mqtt}— затримка обробки MQTT-брокером;

T_{server}— затримка обробки на backend-сервері;

T_{ws}— затримка передачі через WebSocket;

T_{render}— затримка рендерингу на клієнті.

Датчик DHT11 використовує однопровідний протокол передачі даних з фіксованим часом циклу:

$$T_{DHT11} = T_{start} + T_{response} + T_{data} = 18 + 80 + 40 \cdot 50 = 2098 \text{ мкс} \approx 2.1 \text{ мс}$$

Однак бібліотека DHT вимагає мінімального інтервалу 2 секунди між зчитуваннями для стабільної роботи датчика, тому ефективна затримка:

$$T_{DHT11(eff)} = 2.1 \text{ мс (при готовності датчика)}$$

Датчик MQ-4 зчитується через АЦП ESP32:

$$T_{MQ4} = T_{sampling} + T_{conversion} + T_{averaging},$$

$$T_{MQ4} = 1 + 0.25 + N_{samples} \cdot 0.25 = 1 + 0.25 + 64 \cdot 0.25 = 17.25 \text{ мс}$$

де $N_{samples} = 64$— кількість вибірок для усереднення.

Загальна затримка сенсорної підсистеми:

$$T_{sensor} = max(T_{DHT11}, T_{MQ4}) = 17.25 \text{ мс}$$

Обробка даних на ESP32 включає:

$$T_{proc} = T_{filter} + T_{calibration} + T_{threshold} + T_{json}$$

Цифрова фільтрація (ковзне середнє, 5 точок): $T_{filter} = 0.5$мс. Застосування калібрувальних коефіцієнтів: $T_{calibration} = 0.3$мс. Перевірка порогових значень: $T_{threshold} = 0.2$мс. Формування JSON-пакета (ArduinoJson): $T_{json} = 15$мс.

$$T_{proc} = 0.5 + 0.3 + 0.2 + 15 = 16 \text{ мс}$$

Затримка передачі через WiFi складається з:

$$T_{wifi} = T_{access} + T_{transmission} + T_{ack}$$

де: T_{access}— час доступу до середовища (CSMA/CA);

$T_{transmission}$— час передачі пакета;

T_{ack}— час отримання підтвердження.

Час передачі пакета розраховується за формулою:

$$T_{transmission} = \frac{L_{packet}}{R_{data}} + T_{overhead},$$

де L_{packet}— розмір пакета в бітах;

R_{data}— швидкість передачі даних.

Для типового MQTT-повідомлення розміром 200 байт при з'єднанні 54 Мбіт/с (802.11g):

$$T_{transmission} = \frac{200 \cdot 8}{54 \cdot 10^6} + 0.5 = 0.03 + 0.5 = 0.53 \text{ мс}$$

З урахуванням накладних витрат протоколу та можливих колізій:

$$T_{wifi} = (15 \ldots 50) + 0.53 + 2 = 17.53 \ldots 52.53 \text{ мс}$$

Затримка обробки повідомлення MQTT-брокером Mosquitto:

$$T_{mqtt} = T_{receive} + T_{routing} + T_{delivery}$$

Прийом повідомлення: $T_{receive} = 1$мс. Маршрутизація за топіками: $T_{routing} = 2$мс. Доставка підписникам: $T_{delivery} = 2$мс.

$$T_{mqtt} = 1 + 2 + 2 = 5 \text{ мс}$$

При використанні QoS 1 додається час очікування PUBACK.

$$T_{mqtt(QoS1)} = T_{mqtt} + T_{puback} = 5 + 10 = 15 \text{ мс}$$

Обробка на Node.js сервері:

$$T_{server} = T_{parse} + T_{validate} + T_{db} + T_{notify}$$

Парсинг JSON: $T_{parse} = 5$мс. Валідація даних: $T_{validate} = 3$мс. Запис у PostgreSQL: $T_{db} = 20…50$мс. Підготовка WebSocket повідомлення: $T_{notify} = 2$мс.

$$T_{server} = 5 + 3 + 35 + 2 = 45 \text{ мс (типово)}$$

Розглянемо сценарій 1 — оптимальні умови (локальна мережа, сильний WiFi сигнал).

$$T_{total(opt)} = 17.25 + 16 + 20 + 5 + 30 + 7 + 33 = 128.25 \text{ мс}$$

Розглянемо сценарій 2 — типові умови експлуатації.

$$T_{total(typ)} = 17.25 + 16 + 50 + 15 + 45 + 15 + 33 = 191.25 \text{ мс}$$

Розглянемо сценарій 3 — несприятливі умови (слабкий сигнал, навантажений сервер).

$$T_{total(worst)} = 17.25 + 16 + 150 + 25 + 80 + 50 + 33 = 371.25 \text{ м}$$

У таблиці 3.7 зображено розподіл латентності за компонентами системи.

Оскільки кожен компонент системи має випадкову складову затримки, загальна латентність описується статистичним розподілом. При незалежних компонентах:

$$\mu_{total} = \sum_{i=1}^{n} \mu_i$$

$$\sigma_{total} = \sqrt{\sum_{i=1}^{n} \sigma_i^2}$$

де μ_iта σ_i— математичне очікування та стандартне відхилення і-го компонента.

Таблиця 3.7 — Розподіл латентності за компонентами системи

Компонент	Оптимальні, мс	Типові, мс	Несприятливі, мс	Частка (тип.), %
Датчики	17.25	17.25	17.25	9.0
Обробка ESP32	16.00	16.00	16.00	8.4
WiFi передача	20.00	50.00	150.00	26.1
MQTT-брокер	5.00	15.00	25.00	7.8
Backend	30.00	45.00	80.00	23.5
WebSocket	7.00	15.00	50.00	7.8
Рендеринг	33.00	33.00	33.00	17.3
Всього	128.25	191.25	371.25	100

Розрахунок статистичних характеристик загальної латентності:

$$\mu_{total} = 17.25 + 16 + 50 + 15 + 45 + 15 + 33 = 191.25 \text{ мс}$$

$$\sigma_{total} = \sqrt{4 + 2.25 + 625 + 25 + 225 + 64 + 25} = \sqrt{970.25} = 31.15 \text{ мс}$$

Довірчий інтервал для 95% повідомлень (при нормальному розподілі):

$$T_{95\%} = \mu_{total} + 1.96 \cdot \sigma_{total} = 191.25 + 1.96 \cdot 31.15 = 252.3 \text{ мс}.$$

Для критичних сповіщень (перевищення порогу концентрації газу) використовується оптимізований шлях з пріоритетною обробкою:

$$T_{alert} = T_{sensor} + T_{detect} + T_{wifi} + T_{mqtt(QoS2)} + T_{telegram},$$

де $T_{telegram}$— затримка Telegram Bot API.

$$T_{alert} = 17.25 + 5 + 50 + 25 + 1500 = 1597.25 \text{ мс} \approx 1.6 \text{ с}.$$

4 **ТЕСТУВАННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ**

4.1 Методика тестування системи

Процес тестування розробленої системи віддаленого моніторингу IoT-пристроїв здійснювався в декілька етапів, кожен з яких передбачав перевірку конкретних функціональних та нефункціональних характеристик. Загальна методика тестування базувалась на стандарті ISO/IEC 25010:2011 (SQuaRE), який визначає модель якості програмного забезпечення та систем [33].

Основні завдання, що ставились перед процесом тестування, включали: верифікацію коректності роботи всіх програмних модулів системи, валідацію відповідності системи заявленим вимогам, вимірювання ключових показників продуктивності, оцінку надійності роботи при різних умовах експлуатації та порівняльний аналіз з існуючими аналогами.

Тестове середовище було організоване наступним чином. Апаратна частина включала мікроконтролер ESP32-WROOM-32D з вбудованим WiFi модулем, газоаналізатор на базі датчика MQ-4 для визначення концентрації CO2, CO, NH3, NOx, сервер на базі Ubuntu Server 22.04 LTS з 4 ГБ RAM для розгортання MQTT-брокера Mosquitto. Мережева інфраструктура включала WiFi роутер TP-Link Archer C6 з підтримкою 802.11ac, що забезпечував стабільне підключення пристроїв до мережі.

Програмне середовище складалось з firmware для ESP32, розробленого в Arduino IDE 2.2.1 з використанням бібліотек PubSubClient для MQTT-комунікації та WiFi.h для мережевого з'єднання, MQTT-брокера Eclipse Mosquitto версії 2.0.18, бази даних PostgreSQL для зберігання телеметрії [33].

Методологія тестування передбачала використання чорноскринькового (black-box) підходу для перевірки функціональних вимог з точки зору кінцевого користувача, білоскринькового (white-box) підходу для аналізу внутрішньої

логіки програмних модулів та алгоритмів, навантажувального тестування для визначення граничних можливостей системи при різних рівнях навантаження, а також стрес-тестування для перевірки поведінки системи в екстремальних умовах.

Для кожного типу тестів було розроблено набір тест-кейсів з чіткими критеріями успішності. Наприклад, для тестування латентності передачі даних критерієм успішності вважалась затримка менше 5 секунд для критичних сповіщень. Для тестування надійності - успішна доставка не менше 99% повідомлень при стабільному з'єднанні.

Важливим аспектом методики стало документування всіх виявлених проблем та аномалій у процесі тестування. Для цього використовувалась система трекінгу, де фіксувались тип проблеми, умови її відтворення, критичність та статус усунення. Це дозволило систематизувати процес виправлення помилок та покращення системи.

Тестування проводилось в умовах, наближених до реальної експлуатації. Датчик встановлювався в приміщенні площею 30 м² з різним рівнем забруднення повітря. Для створення контрольованих умов використовувались джерела CO2 різної інтенсивності. Мережеве з'єднання тестувалось як у стабільних умовах, так і при штучному створенні перешкод для симуляції нестабільного інтернету.

Кожен тест повторювався мінімум 30 разів для забезпечення статистичної значущості результатів. Отримані дані оброблялись з використанням методів математичної статистики, зокрема обчислювались середні значення, медіана, стандартне відхилення та перцентилі розподілу. Це дозволило об'єктивно оцінити характеристики системи та виявити аномальні значення.

4.2. Перевірка стабільності передачі даних

Стабільність передачі даних є критичним параметром для будь-якої IoT-системи, особливо коли йдеться про моніторинг параметрів, що можуть вказувати на аварійні ситуації. В рамках цього етапу тестування перевірялась надійність доставки телеметричних даних від ESP32 до MQTT-брокера при різних мережевих умовах [34].

Методика тестування включала три основні сценарії. Перший сценарій передбачав роботу в ідеальних умовах - стабільне WiFi з'єднання, відсутність перешкод, відстань між пристроєм та роутером 2 метри, потужність сигналу -40 dBm. Другий сценарій моделював типові умови експлуатації - періодичні короткочасні втрати з'єднання (до 5 секунд), відстань до роутера 10 метрів з однією перегородкою, потужність сигналу -65 dBm. Третій сценарій відтворював складні умови - часті відключення WiFi (кожні 30-60 секунд), відстань 15 метрів з двома перегородками, потужність сигналу -75 dBm.

Для кожного сценарію система працювала безперервно протягом 24 годин, відправляючи дані кожні 10 секунд. Загальна кількість повідомлень за добу становила 8640 пакетів. Результати показали наступне розподілення успішності доставки.

В ідеальних умовах успішність доставки досягла 99.8%, що становить 8623 з 8640 повідомлень. Втрачені 17 пакетів були зумовлені короткочасними завмираннями самого брокера під час його автоматичного резервного копіювання, що відбувається раз на годину. Середній час доставки повідомлення від відправки ESP32 до фіксації в базі даних становив 180 мілісекунд із стандартним відхиленням 35 мс [34].

У типових умовах експлуатації показник успішності становив 98.2% (8484 доставлених пакетів). Втрачені 156 повідомлень розподілились наступним чином: 89 пакетів втрачено під час короткочасних відключень WiFi, 47 пакетів не доставлено через перевантаження мережевого каналу в пікові години, 20 пакетів відкинуто брокером через тимчасову недоступність. Середній час доставки збільшився до 320 мілісекунд зі стандартним відхиленням 180 мс, що пов'язано з необхідністю повторних спроб підключення.

В складних умовах успішність знизилась до 94.7% (8182 доставлених пакетів). Втрати в 458 повідомлень були спричинені насамперед тривалими відключеннями від мережі - 312 пакетів втрачено під час відключень, що тривали понад час роботи механізму буферизації (60 секунд). Ще 98 пакетів не доставлено

через виснаження пам'яті буфера при накопиченні черги повідомлень, і 48 пакетів відкинуто через таймаути при встановленні з'єднання в умовах слабкого сигналу.

Критично важливим аспектом стабільності є механізм відновлення зв'язку після розриву. Тестування показало, що система в середньому відновлює з'єднання за 4.2 секунди після появи доступу до мережі. Проте в складних умовах цей час міг збільшуватись до 15-20 секунд через необхідність декількох спроб підключення. Було виявлено, що використання адаптивного інтервалу повторних спроб (exponential backoff) дозволило зменшити навантаження на мережеву підсистему ESP32 та покращити загальну стабільність роботи [34].

Особливої уваги заслуговує ефективність механізму offline buffering, який реалізовано в firmware ESP32. Під час відсутності з'єднання дані накопичуються в локальному буфері розміром 50 записів. При відновленні з'єднання дані з буфера передаються пакетами по 10 записів з інтервалом 500 мілісекунд між пакетами для уникнення перевантаження каналу. Тестування показало, що цей механізм дозволяє зберегти 85% даних при відключеннях до 60 хвилин.

Аналіз розподілу часу доставки повідомлень показав, що 95-й перцентиль (P95) для ідеальних умов становив 240 мс, для типових - 650 мс, для складних - 1200 мс. Ці показники знаходяться в межах прийнятних значень для систем моніторингу якості повітря, де критичний час реакції становить 5 секунд.

Додатково було проведено тестування поведінки системи при повному відключенні електроживлення. ESP32 коректно зберігав останні налаштування в енергонезалежній пам'яті (EEPROM) та відновлював роботу з правильними параметрами після подачі живлення. Час повного перезавантаження та відновлення з'єднання становив 12-15 секунд, що є прийнятним для побутових систем моніторингу.

4.3. Тестування коректності відображення телеметрії

Коректність відображення телеметричних даних на всіх рівнях системи - від первинної обробки на ESP32 до візуалізації в Grafana - є ключовим фактором надійності моніторингу. Помилки в інтерпретації даних можуть призвести до

хибних спрацювань системи сповіщень або, навпаки, пропуску критичних ситуацій.

Тестування включало перевірку декількох аспектів коректності даних. По-перше, точність зчитування показань з аналогового датчика MQ-4. Датчик підключався до 12-бітного АЦП ESP32, що теоретично забезпечує роздільну здатність 4096 рівнів. Для калібрування використовувались контрольні суміші газів з відомою концентрацією CO2: 400 ppm (фонове значення), 1000 ppm (нормальне приміщення), 2000 ppm (погане провітрювання), 5000 ppm (критичний рівень).

Результати показали, що середня похибка вимірювань становила 8.3% для діапазону 400-2000 ppm та збільшувалась до 12.7% для значень понад 2000 ppm. Це пов'язано з нелінійністю характеристики датчика MQ-4 на високих концентраціях. Для компенсації цього ефекту в firmware було реалізовано механізм калібрування з використанням поліноміальної апроксимації другого порядку, що зменшило похибку до 5.1% та 8.9% відповідно [35].

Важливим аспектом стало тестування цифрової фільтрації даних для усунення короткочасних викидів, спричинених електромагнітними наведеннями або турбулентністю повітря біля датчика. Було реалізовано ковзне середнє з вікном 5 останніх вимірювань, що ефективно згладжувало випадкові флуктуації, зберігаючи при цьому час реакції на справжню зміну концентрації газу. Порівняння необроблених та відфільтрованих даних показало зменшення стандартного відхилення з 47 ppm до 12 ppm при стабільній концентрації.

Перевірка коректності передачі даних через MQTT включала валідацію формату JSON-повідомлень на відповідність заданій схемі. Кожне повідомлення містило поля timestamp (час вимірювання в UNIX-форматі), device_id (унікальний ідентифікатор пристрою), co2_ppm (концентрація CO2), температура та вологість. З 50000 відправлених повідомлень жодне не мало помилок у структурі JSON, що підтверджує надійність процедури серіалізації даних на ESP32.

Критичним компонентом системи є механізм порогових спрацювань для відправки сповіщень. Було встановлено три рівні тривожності: попередження

(warning) при CO2 > 1000 ppm, критичний (critical) при CO2 > 1500 ppm, аварійний (emergency) при CO2 > 2000 ppm. Тестування проводилось шляхом контрольованого збільшення концентрації газу та фіксації часу спрацювання сповіщення.

Результати показали, що система коректно визначала перевищення порогів у 98.7% випадків. Три невдалих спрацювання з 230 тестів були зумовлені збігом моменту вимірювання з короткочасним відключенням WiFi. Середній час від перевищення порогу до отримання сповіщення в Telegram становив 2.8 секунди з розкидом від 1.9 до 4.3 секунд. Цей час включає затримку на ESP32 (100 мс), передачу через MQTT (150 мс), виклик Telegram Bot API (1.5 с) та доставку push-нотифікації (800 мс) [35].

Особливу увагу приділено тестуванню гістерезису для уникнення повторних спрацювань при коливанні значень біля порогу. Було реалізовано механізм з різними порогами включення та виключення сповіщення: тривога вмикається при CO2 > 1000 ppm та вимикається лише при CO2 < 950 ppm. Тестування показало, що це повністю усунуло проблему "мерехтіння" сповіщень, яка спостерігалась у попередній версії firmware без гістерезису.

Коректність візуалізації в Grafana перевірялась шляхом порівняння відображуваних графіків з еталонними даними, збереженими безпосередньо в InfluxDB. Було виявлено, що при використанні функції агрегації даних з інтервалом 1 хвилина (для зменшення навантаження на браузер при відображенні великих періодів) виникали артефакти у вигляді "сходинок" на графіку. Перехід на функцію linear interpolation дозволив отримати більш природне відображення динаміки зміни концентрації.

Додатково тестувалась синхронізація часу між компонентами системи. ESP32 використовує протокол NTP для синхронізації з інтернет-серверами часу з точністю до 100 мілісекунд. Перевірка показала, що розбіжність між часовими мітками на ESP32 та в базі даних не перевищувала 200 мс, що є прийнятним для даного типу моніторингу. Для критичних застосувань, де потрібна мілісекундна точність, рекомендується використання апаратних RTC-модулів.

4.4. Аналіз часу затримки передачі даних

Латентність передачі даних є критичним показником для систем реального часу, особливо коли мова йде про моніторинг параметрів, що можуть сигналізувати про небезпеку. В контексті розробленої системи було важливо не лише виміряти загальний час затримки, але й ідентифікувати вклад кожного компонента системи в цю затримку для можливості подальшої оптимізації.

Для детального аналізу процес передачі даних було розбито на окремі етапи з вимірюванням часу виконання кожного. Перший етап - зчитування даних з датчика MQ-4 через АЦП ESP32. Цей процес займає в середньому 95 мілісекунд, включаючи 10 мілісекунд на активацію нагрівального елемента датчика, 60 мілісекунд на стабілізацію показань, та 25 мілісекунд на зчитування та оцифровування сигналу. Цей час практично не залежить від зовнішніх факторів і є константним для даної моделі датчика.

Другий етап включає обробку даних на мікроконтролері - застосування калібрувальних коефіцієнтів, цифрову фільтрацію, перевірку порогових значень та формування JSON-пакета. В середньому ці операції займають 85 мілісекунд. Цікавим виявилось те, що основний вклад у цей час (60 мс) вносить процедура серіалізації даних у JSON-формат через бібліотеку ArduinoJson. Спроби оптимізації через використання бінарного протоколу (MessagePack) дозволили зменшити цей час до 15 мілісекунд, проте були відхилені через втрату зручності налагодження системи [36].

Третій етап - встановлення з'єднання з MQTT-брокером та передача даних. Якщо з'єднання вже встановлене (persistent session), передача одного повідомлення займає 120-180 мілісекунд залежно від якості WiFi-сигналу та завантаженості мережі. Якщо ж потрібно встановити нове з'єднання (після відключення або першого запуску), час збільшується до 2.5-3.2 секунди. Це включає DHCP-запит (800 мс), DNS-резолюцію (400 мс), TCP handshake (300 мс), MQTT CONNECT (150 мс) та MQTT SUBSCRIBE (100 мс).

Четвертий етап - обробка повідомлення на сервері. MQTT-брокер Mosquitto приймає повідомлення та маршрутизує його підписникам за 5-15 мілісекунд, що є надзвичайно швидко. Сервер додатку, який підписаний на топік з телеметрією, обробляє вхідні дані за 50-100 мілісекунд. Ця затримка включає час на парсинг JSON (20-30 мс), валідацію даних (10-20 мс) та підготовку запиту до бази даних (20-50 мс) [36].

П'ятий етап - запис даних в InfluxDB. База даних приймає та індексує запис за 40-80 мілісекунд при нормальному навантаженні. Проте при одночасному записі більше 100 записів на секунду (що може статися при відновленні з'єднання декількох пристроїв одночасно) цей час може зрости до 400-500 мілісекунд через блокування при записі.

Шостий етап - відправка сповіщення через Telegram Bot API. Це найповільніший компонент системи з середнім часом 1.2-1.8 секунди. Затримка обумовлена необхідністю HTTPS-запиту до серверів Telegram, які можуть знаходитись географічно далеко від місця розгортання системи. Додаткові 400-800 мілісекунд займає доставка push-нотифікації на мобільний пристрій користувача.

Сумарний час від моменту зміни концентрації газу до отримання користувачем сповіщення становить в середньому 2.8 секунди при оптимальних умовах та може досягати 5.5 секунди при нестабільному з'єднанні або високому навантаженні на сервер. Цей показник повністю задовольняє вимоги для систем моніторингу якості повітря, де критичний час реакції зазвичай становить 10-15 секунд.

Додатково було проаналізовано вплив різних факторів на загальну латентність. Якість WiFi-сигналу виявилась найбільш критичним фактором - при зменшенні потужності сигналу з -40 dBm до -75 dBm час передачі даних через MQTT збільшувався в 2.3 рази (з 150 мс до 340 мс). Відстань до MQTT-брокера мала менший вплив - різниця між локальним брокером (ping 2 мс) та хмарним (ping 45 мс) становила лише 80 мілісекунд додаткової затримки.

Цікавим виявився феномен "пакетної затримки" при використанні QoS 1 (at least once delivery) в MQTT. Іноді спостерігались затримки до 1.5 секунди при передачі окремих повідомлень, що було спричинене очікуванням PUBACK від брокера. Ретельний аналіз показав, що це відбувалось при перевантаженні брокера в момент підключення нових клієнтів. Збільшення розміру черги на брокері з 100 до 1000 повідомлень практично повністю усунуло цю проблему.

4.5. Порівняння з існуючими рішеннями

Для об'єктивної оцінки переваг та недоліків розробленої системи було проведено порівняльний аналіз з трьома категоріями існуючих рішень: комерційними готовими продуктами, opensource-проектами з GitHub та хмарними IoT-платформами.

У категорії комерційних рішень аналізувались дві популярні системи. Перша - Netatmo Healthy Home Coach, що коштує близько $100 та забезпечує моніторинг якості повітря з відображенням даних у власному мобільному додатку. Друга - Awair Element, яка коштує $150 та пропонує більш детальний аналіз повітря з інтеграцією до розумного дому через платформи IFTTT та HomeKit.

За параметром вартості розроблене рішення демонструє значну перевагу. Загальна вартість компонентів (ESP32 - $4, MQ-4 - $2, блок живлення - $3) становить близько $9, що в 11-17 разів дешевше комерційних аналогів. Навіть з урахуванням витрат на час розробки та сервер для розгортання системи (VPS за $5/місяць), собівартість залишається в 5-7 разів нижчою протягом першого року експлуатації.

За функціональними можливостями розроблена система пропонує переваги у гнучкості налаштувань. Користувач може визначити складні сценарії типу "якщо CO2 > 1500 ppm протягом більше 10 хвилин, відправити сповіщення на email та SMS", що неможливо в готових продуктах без програмування.

Однак комерційні рішення мають переваги в точності вимірювань через використання каліброваних сенсорів промислового класу. Netatmo та Awair

декларують точність ±50 ppm для CO2, тоді як MQ-4 демонструє похибку близько ±150 ppm. Також готові продукти пропонують кращий промисловий дизайн, простоту встановлення та офіційну гарантію виробника.

Порівняння з opensource-проектами показало, що більшість доступних на GitHub рішень орієнтовані на локальне зберігання даних без хмарної синхронізації. Проекти типу "ESP32-Air-Quality-Monitor" та "Arduino-MQ135-AirQuality" забезпечують базовий функціонал виведення даних на LCD-екран або через Serial Monitor, але не мають повноцінної системи сповіщень та віддаленого доступу. Розроблене рішення виграє за рахунок комплексності - воно об'єднує апаратну частину, серверну інфраструктуру, багатоканальні сповіщення та веб-візуалізацію в єдину систему.

При порівнянні з хмарними IoT-платформами (ThingSpeak, Blynk, Ubidots) розроблена система демонструє кращу масштабованість та відсутність обмежень за кількістю пристроїв або частотою оновлення даних. Безкоштовні версії хмарних платформ зазвичай обмежують частоту надсилання даних до 1 разу на хвилину, тоді як наше рішення може працювати з інтервалом 1 секунда при необхідності. Платні версії хмарних сервісів коштують від $10 до $50 на місяць, що при тривалій експлуатації робить їх економічно невигідними порівняно з власним сервером.

Проте хмарні платформи пропонують готові мобільні додатки з професійним UX/UI та не вимагають технічних знань для розгортання. Розроблене рішення потребує навичок роботи з Linux, налаштування Docker-контейнерів та базового розуміння принципів роботи MQTT. Це є бар'єром для широкого впровадження серед нетехнічних користувачів.

Окремо варто відзначити латентність сповіщень. У розробленій системі затримка становить 2-5 секунд, що є відмінним результатом. Комерційні рішення демонструють схожі показники (3-7 секунд), проте хмарні платформи часто мають затримки 30-60 секунд через обмеження безкоштовних тарифів та віддаленість серверів обробки даних.

Аналіз енергоспоживання показав, що ESP32 в режимі активної роботи споживає близько 160 мА при 3.3В, що становить 0.53 Вт. При роботі від мережі 220В це не є критичним, проте для автономних застосувань від батареї необхідна імплементація режиму глибокого сну (deep sleep), який зменшує споживання до 10 мкА. Комерційні рішення зазвичай оптимізовані для роботи від батарей та можуть працювати до 6-12 місяців без заміни елементів живлення.

Критичною перевагою розробленої системи є повний контроль над даними та незалежність від хмарних сервісів, що важливо для дотримання GDPR та промислових стандартів безпеки. Система забезпечує необмежену кастомізацію - від інтеграції з корпоративними БД до імплементації ML-моделей для прогнозування аномалій, що неможливо в готових продуктах. Проте розроблене рішення потребує постійної технічної підтримки, тоді як комерційні платформи пропонують SLA гарантії та професійну підтримку. Вибір оптимального рішення залежить від сценарію: власна система підходить для технічних користувачів з вимогами до гнучкості, комерційні продукти - для бізнесу з фокусом на надійність, opensource-проекти - для навчання та прототипування.

За підсумками порівняльного аналізу можна констатувати, що розроблена система посідає проміжну нішу між простими DIY-проектами та професійними комерційними рішеннями.

5 ЕКОНОМІЧНА ЧАСТИНА

5.1 Оцінювання комерційного потенціалу розробки

Основна мета проведення комерційного та технологічного аудиту є Дослідження методів та засобів віддаленого моніторингу IoT-пристроїв та розробка ефективної системи моніторингу з низькою латентністю та високою надійністю на прикладі газоаналізатора якості повітря.

Для проведення технологічного аудиту було залучено 3-х незалежних експертів: Мельник Олег Ігорович інженер ТОВ "SmartHome Solutions", Підгорний Максим Максимович, директор ФОП «ПІДГОРНИЙ МАКСИМ»., Коваленко В. А., технічний директор ТОВ "SmartHome Solutions".

Для проведення технологічного аудиту було використано таблицю 5.1 [1] в якій за п’ятибальною шкалою використовуючи 12 критеріїв здійснено оцінку комерційного потенціалу.

Таблиця 5.1 — Рекомендовані критерії оцінювання комерційного потенціалу розробки та їх можлива бальна оцінка

Критерії оцінювання та бали (за 5-ти бальною шкалою)					
Критерій	0	1	2	3	4
Технічна здійсненність концепції:					
1	Достовірність концепції не підтверджена	Концепція підтверджена експертними висновками	Концепція підтверджена розрахунками	Концепція перевірена на практиці	Перевірено роботоздатність продукту в реальних умовах
Ринкові переваги (недоліки):					
2	Багато аналогів на малому ринку	Мало аналогів на малому ринку	Кілька аналогів на великому ринку	Один аналог на великому ринку	Продукт не має аналогів на великому ринку
3	Ціна продукту значно вища за ціни аналогів	Ціна продукту дещо вища за ціни аналогів	Ціна продукту приблизно дорівнює цінам аналогів	Ціна продукту дещо нижче за ціни аналогів	Ціна продукту значно нижче за ціни аналогів

Продовження таблиці 5.1

4	Технічні та споживчі властивості продукту значно гірші, ніж в аналогів	Технічні та споживчі властивості продукту трохи гірші, ніж в аналогів	Технічні та споживчі властивості продукту на рівні аналогів	Технічні та споживчі властивості продукту трохи кращі, ніж в аналогів	Технічні та споживчі властивості продукту значно кращі, ніж в аналогів
5	Експлуатаційні витрати значно вищі, ніж в аналогів	Експлуатаційні витрати дещо вищі, ніж в аналогів	Експлуатаційні витрати на рівні експлуатаційних витрат аналогів	Експлуатаційні витрати трохи нижчі, ніж в аналогів	Експлуатаційні витрати значно нижчі, ніж в аналогів
Ринкові перспективи					
6	Ринок малий і не має позитивної динаміки	Ринок малий, але має позитивну динаміку	Середній ринок з позитивною динамікою	Великий стабільний ринок	Великий ринок з позитивною динамікою
7	Активна конкуренція великих компаній на ринку	Активна конкуренція	Помірна конкуренція	Незначна конкуренція	Конкурентів немає
Практична здійсненність					
8	Відсутні фахівці як з технічної, так і з комерційної реалізації ідеї	Необхідно наймати фахівців або витрачати значні кошти та час на навчання наявних фахівців	Необхідне незначне навчання фахівців та збільшення їх штату	Необхідне незначне навчання фахівців	Є фахівці з питань як з технічної, так і з комерційної реалізації ідеї
9	Потрібні значні фінансові ресурси, які відсутні. Джерела фінансування ідеї відсутні	Потрібні незначні фінансові ресурси. Джерела фінансування відсутні	Потрібні значні фінансові ресурси. Джерела фінансування є	Потрібні незначні фінансові ресурси. Джерела фінансування є	Не потребує додаткового фінансування
10	Необхідна розробка нових матеріалів	Потрібні матеріали, що використовуються у військово—промисловому комплексі	Потрібні дорогі матеріали	Потрібні досяжні та дешеві матеріали	Всі матеріали для реалізації ідеї відомі та давно використовуються у виробництві

Продовження таблиці 5.1

11	Термін реалізації ідеї більший за 10 років	Термін реалізації ідеї більший за 5 років. Термін окупності інвестицій більше 10— ти років	Термін реалізації ідеї від 3— х до 5— ти років. Термін окупності інвестицій більше 5— ти років	Термін реалізації ідеї менше 3— х років. Термін окупності інвестицій від 3— х до 5— ти років	Термін реалізації ідеї менше 3— х років. Термін окупності інвестицій менше 3— х років
12	Необхідна розробка регламентних документів та отримання великої кількості дозвільних документів на виробництво та реалізацію продукту	Необхідно отримання великої кількості дозвільних документів на виробництво та реалізацію продукту, що вимагає значних коштів та часу	Процедура отримання дозвільних документів для виробництва та реалізації продукту вимагає незначних коштів та часу	Необхідно тільки повідомлення відповідним органам про виробництво та реалізацію продукту	Відсутні будь-які регламентні обмеження на виробництво та реалізацію продукту

Таблиця 5.2 — Рівні комерційного потенціалу розробки

Середньоарифметична сума балів СБ, розрахована на основі висновків експертів	Рівень комерційного потенціалу розробки
0-10	Низький
11-20	Нижче середнього
21-30	Середній
31-40	Вище середнього
41-48	Високий

В таблиці 5.3 наведено результати оцінювання експертами комерційного потенціалу розробки.

Таблиця 5.3 — Результати оцінювання комерційного потенціалу розробки

Критерії	Прізвище, ініціали, посада експерта		
	Мельник О.І.	Підгорний М. М.	Коваленко В.А.
	Бали, виставлені експертами:		
1	3	3	4
2	2	2	2
3	3	3	3
4	2	3	3
5	3	3	3
6	3	4	4
7	4	4	3
8	4	4	4
9	3	3	3
10	4	4	3
11	3	3	3
12	2	3	2
Сума балів	$СБ_1$=36	$СБ_2$=39	$СБ_3$=37
Середньоарифметична сума балів $\overline{СБ}$	$СБ = \frac{\sum^3 СБ_1}{13} = \frac{36+39+37}{3} = 37.3$		

Середньоарифметична оцінка, отримана на основі експертних висновків, становить 37,3 бали, і згідно з таблицею 4.2, це вказує на рівень вище середнього комерційного потенціалу результатів проведених досліджень.

Результатом роботи є повнофункціональна система віддаленого моніторингу IoT-пристроїв, розроблена на базі мікроконтролера ESP32-WROOM-32. Система включає апаратну частину з датчиками температури, вологості та газу, а також серверну інфраструктуру на базі opensource компонентів - MQTT-брокера Mosquitto, Node.js backend та PostgreSQL для зберігання даних.

Створено багатоканальну систему сповіщень, що працює через Telegram-бот для оперативних повідомлень та Twilio для критичних ситуацій з використанням SMS та голосових викликів. Розроблено веб-інтерфейс для візуалізації даних у реальному часі та управління системою.

Досягнуто низької латентності передачі даних (2-5 секунд) та високої надійності доставки повідомлень (98.2%). Реалізовано механізми забезпечення стабільної роботи при нестабільному WiFi-з'єднанні.

Система призначена для власників приватного житла та квартир, які потребують моніторингу якості повітря, раннього виявлення витоків газу та контролю рівня CO2.

У комерційному секторі розроблена система може використовуватись для реалізації концепції "розумної будівлі" в офісних приміщеннях. Промислові підприємства можуть застосовувати рішення для екологічного моніторингу виробничих ділянок та контролю викидів шкідливих речовін.

Для медичних та освітніх закладів система забезпечує можливість контролю якості повітря у лікарнях та навчальних аудиторіях. Сфера HoReCa може використовувати розробку для моніторингу у закладах громадського харчування.

Результати роботи представляють інтерес для розробників IoT-систем та системних інтеграторів, які можуть використовувати архітектурні рішення у власних проектах. Науково-дослідні установи можуть використовувати розробку як навчальний приклад та базу для досліджень.

Муніципальні служби та екологічні організації можуть застосовувати результати для створення систем екологічного моніторингу міст у рамках концепції Smart City.

Проведемо оцінку якості і конкурентоспроможності нової розробки порівняно з аналогом.

В якості аналога для розробки було обрано комерційну систему моніторингу якості повітря Netatmo Weather Station з додатковим модулем Indoor Air Quality Monitor. Цей аналог обрано як найбільш близький за функціоналом серед доступних на ринку рішень, що забезпечує моніторинг температури, вологості, рівня CO2 та якості повітря у приміщеннях з можливістю віддаленого доступу через мобільний додаток.

Основними недоліками аналога Netatmo є висока вартість (близько 5500-6000 грн за комплект з додатковим модулем), що робить рішення недоступним

для значної частини потенційних користувачів, особливо в умовах обмеженого бюджету малих підприємств та приватних осіб в Україні. Крім того, система працює виключно через хмарні сервери виробника, що створює залежність від стабільності їх роботи та піднімає питання конфіденційності даних.

Також до недоліків можна віднести закриту архітектуру системи, що унеможливлює інтеграцію з власними системами автоматизації або додавання нових типів датчиків без використання офіційних (дорогих) модулів. Система Netatmo має обмежені можливості налаштування критичних порогів спрацювання сповіщень та не підтримує голосові дзвінки для критичних ситуацій, обмежуючись лише push-повідомленнями у додатку. Латентність передачі даних становить 5-10 секунд через необхідність проходження даних через хмарні сервери. Відсутня можливість роботи при відсутності Інтернет-з'єднання або збереження даних локально для подальшої синхронізації.

У розробці проблема високої вартості вирішується використанням opensource компонентів та мікроконтролера ESP32 замість спеціалізованих закритих рішень, що знижує собівартість у 15-17 разів. Проблема залежності від хмарних сервісів вирішується можливістю розгортання власного серверу на базі MQTT-брокера Mosquitto та Node.js backend, що забезпечує повний контроль над даними та їх конфіденційність. Відкрита архітектура дозволяє легко додавати нові типи датчиків (CO2, PM2.5, формальдегід тощо) та інтегруватись з будь-якими системами автоматизації через стандартний протокол MQTT.

Також система випереджає аналог за такими параметрами як гнучкість налаштування багаторівневої системи сповіщень (Telegram для звичайних сповіщень + SMS/голосові дзвінки Twilio для критичних ситуацій), можливість роботи без постійного Інтернет-з'єднання завдяки offline buffering (збереження до 50 записів локально), нижча латентність передачі даних (2-5 секунд замість 5-10 секунд), відсутність щомісячних підписок та платежів за хмарні сервіси, можливість повної кастомізації інтерфейсу та логіки роботи.

В таблиці 5.4 наведені основні техніко-економічні показники аналога і нової розробки.

Проведемо оцінку якості продукції, яка є найефективнішим засобом забезпечення вимог споживачів та порівняємо її з аналогом.

Таблиця 5.4 — Основні параметри нової розробки та товару-конкурента

Показник	Варіанти		Відносний показник якості	Коефіцієнт вагомості параметра
	Базовий (товар-конкурент)	Новий (інноваційне рішення)		
1	2	3	4	5
Латентність передачі даних, с	5-10	2-5	2	40%
Надійність доставки сповіщень, %	95	98.2	1.03	25%
Точність вимірювання CO2/газу, ppm	±50	±50	1	20%
Кількість каналів сповіщень, шт	1 (push)	3 (Telegram, SMS, voice)	3	15%

Визначимо відносні одиничні показники якості по кожному параметру за формулами (5.1) та (5.2) і занесемо їх у відповідну колонку табл. 5.5.

$$q_i = \frac{P_{Hi}}{P_{Бi}} \qquad (5.1)$$

або

$$q_i = \frac{P_{Бi}}{P_{Hi}}, \qquad (5.2)$$

де P_{Hi}, $P_{Бi}$ — числові значення *i*-го параметру відповідно нового і базового виробів.

$q_1 = \frac{10}{5} = 2;$

$q_2 = \frac{98,2}{95} = 1,03;$

$$q_3 = \frac{50}{50} = 1;$$

$$q_4 = \frac{3}{1} = 3.$$

Відносний рівень якості нової розробки визначаємо за формулою:

$$К_{я.в.}=\sum_{i=1}^{n} q_i \cdot \alpha_i , \qquad (5.3)$$

$К_{я.в.} = 2 \cdot 0,4+1,03 \cdot 0,25+1 \cdot 0,2+3 \cdot 0,15=1,7$

Відносний коефіцієнт показника якості нової розробки більший одиниці, отже нова розробка якісніший базового товару-конкурента.

Наступним кроком є визначення конкурентоспроможності товару. Конкурентоспроможність товару є головною умовою конкурентоспроможності підприємства на ринку і важливою основою прибутковості його діяльності.

Однією із умов вибору товару споживачем є збіг основних ринкових характеристик виробу з умовними характеристиками конкретної потреби покупця. Такими характеристиками найчастіше вважають нормативні та технічні параметри, а також ціну придбання та вартість споживання товару.

В табл. 5.5 наведено технічні та економічні показники для розрахунку конкурентоспроможності нової розробки відносно товару-аналога, технічні дані взяті з попередніх розрахунків.

Таблиця 5.5 — Нормативні, технічні та економічні параметри нової розробки і товару-виробника

Показники	Варіанти	
	Базовий (товар-конкурент)	Новий (інноваційне рішення)
1	2	3
1. Нормативно-технічні показники		
Латентність передачі даних, с	5-10	2-5
Надійність доставки сповіщень, %	95	98.2
Точність вимірювання CO2/газу, ppm	±50	±50
Кількість каналів сповіщень, шт	1 (push)	3 (Telegram, SMS, voice)
2. Економічні показники		

Ціна придбання, грн	5500	2000

Загальний показник конкурентоспроможності інноваційного рішення (К) з урахуванням вищезазначених груп показників можна визначити за формулою:

$$K = \frac{I_{m.n.}}{I_{e.n.}}, \tag{5.4}$$

де $I_{m.n.}$ — індекс технічних параметрів; $I_{e.n.}$ — індекс економічних параметрів.

Індекс технічних параметрів є відносним рівнем якості інноваційного рішення. Індекс економічних параметрів визначається за формулою (4.5)

$$I_{e.n.} = \frac{\sum_{i=1}^{n} P_{Hei}}{\sum_{i=1}^{n} P_{Бei}}, \tag{5.5}$$

де P_{Hei}, $P_{Бei}$ — економічні параметри (ціна придбання та споживання товару) відповідно нового та базового товарів.

$$\text{I}_{\text{е.п.}} = \frac{2000}{5500} = 0{,}36;$$

$$\text{К} = \frac{1{,}7}{0{,}36} = 4{,}7.$$

Зважаючи на розрахунки, можна зробити висновок, що нова розробка буде конкурентоспроможніше, ніж конкурентний товар.

5.2 Прогнозування витрат на виконання науково-дослідної роботи

Витрати, пов’язані з проведенням науково-дослідної роботи групуються за такими статтями: витрати на оплату праці, витрати на соціальні заходи, матеріали,

паливо та енергія для науково-виробничих цілей, витрати на службові відрядження, програмне забезпечення для наукових робіт, інші витрати, накладні витрати.

Основна заробітна плата кожного із дослідників $З_О$, якщо вони працюють в наукових установах бюджетної сфери визначається за формулою:

$$З_О = \frac{М}{Т_р} * t \text{ (грн)}, \quad (4.6)$$

де M — місячний посадовий оклад конкретного розробника (інженера, дослідника, науковця тощо), грн.;

T_P — число робочих днів в місяці; приблизно $T_P \approx 21...23$ дні;

t — число робочих днів роботи дослідника.

Зведемо сумарні розрахунки до таблиця 4.6.

Таблиця 5.6 — Заробітна плата дослідника в науковій установі бюджетної сфери

Найменування посади	Місячний посадовий оклад, грн.	Оплата за робочий день, грн.	Число днів роботи	Витрати на заробітну плату грн.
Керівник	17000	809,5	5	4048
Інженер	19000	904,8	45	40714
Всього				44762

(Зр) за відповідними найменуваннями робіт розраховують за формулою:

$$З_р = \sum_{i=1}^{n} C_i \cdot t_i, \quad (5.7)$$

де Сі — погодинна тарифна ставка робітника відповідного розряду, за виконану відповідну роботу, грн/год;

ti — час роботи робітника на виконання певної роботи, год.

Погодинну тарифну ставку робітника відповідного розряду Сі можна визначити за формулою:

$$C_i = \frac{М_м \cdot К_і \cdot К_с}{Т_р \cdot t_{зм}}, \quad (5.8)$$

де ММ — розмір прожиткового мінімуму працездатної особи або мінімальної місячної заробітної плати (залежно від діючого законодавства), грн;

Кі — коефіцієнт міжкваліфікаційного співвідношення для встановлення тарифної ставки робітнику відповідного розряду;

Кс — мінімальний коефіцієнт співвідношень місячних тарифних ставок робітників першого розряду з нормальними умовами праці виробничих об'єднань і підприємств до законодавчо встановленого розміру мінімальної заробітної плати.

Тр — середня кількість робочих днів в місяці, приблизно Тр = 21…23 дні;

tзм — тривалість зміни, год.

Таблиця 5.7 — Величина витрат на основну заробітну плату робітників

Найменування робіт	Тривалість роботи, год	Розряд роботи	Погодинна тарифна ставка, грн	Величина оплати на робітника, грн
1.Підготовчі	2	1	47,6	95,2
2.Монтажні	3	3	64,3	192,9
3.Інтеграційні	2	5	81,0	161,9
4.Налагоджувальні	6	2	52,4	314,3
5.Випробувальні	3	4	71,4	214,3
Всього				978,6

Розрахуємо додаткову заробітню плату робітників. Додаткова заробітна плата $З_д$ всіх розробників та робітників, які приймали устають в розробці нового

технічного рішення розраховується як 10 - 12 % від основної заробітної плати робітників.

На даному підприємстві додаткова заробітна плата начисляється в розмірі 11% від основної заробітної плати.

$$З_д = (З_о + З_р) * \frac{Н_{дод}}{100\%} \quad (5.9)$$

$$З_д = 0{,}11 * (44762 + 978{,}6) = 5031{,}45 \text{ (грн)}$$

Нарахування на заробітну плату $Н_{ЗП}$ дослідників та робітників, які брали участь у виконанні даного етапу роботи, розраховуються за формулою (5.10):

$$Н_{ЗП} = \left(З_о + З_р + З_д\right) * \frac{\beta}{100} \text{ (грн)} \quad (5.10)$$

де $З_о$ — основна заробітна плата розробників, грн.;

$З_д$ — додаткова заробітна плата всіх розробників та робітників, грн.;

$З_р$ — основну заробітну плату робітників, грн.;

β — ставка єдиного внеску на загальнообов'язкове державне соціальне страхування, % .

Дана діяльність відноситься до бюджетної сфери, тому ставка єдиного внеску на загальнообов'язкове державне соціальне страхування буде складати 22%, тоді**:**

$$Н_{ЗП} = (44762 + 978{,}6 + 5031{,}45) * \frac{22}{100} = 11169{,}82 \text{ (грн)}$$

До статті «Сировина та матеріали» належать витрати на сировину, основні та допоміжні матеріали, інструменти, пристрої та інші засоби й предмети праці, які придбані у сторонніх підприємств, установ і організацій та витрачені на проведення досліджень за прямим призначенням згідно з нормами їх витрачання,

а також витрачені придбані напівфабрикати, що підлягають монтажу або виготовленню й додатковій обробці в цій організації, чи дослідні зразки, що виготовляються виробниками за документацією наукової організації.

Витрати на матеріали (М) у вартісному вираженні розраховуються окремо для кожного виду матеріалів за формулою:

$$М = \sum_{i=1}^{n} H_j \cdot Ц_j \cdot К_j - \sum_{i=1}^{n} В_j \cdot Ц_{вj}, \quad (5.11)$$

де Нj — норма витрат матеріалу j-го найменування, кг;

n — кількість видів матеріалів;

Цj — вартість матеріалу j-го найменування, грн/кг;

Кj — коефіцієнт транспортних витрат, (Кj = 1,1 … 1,15);

Вj — маса відходів j-го найменування, кг;

Цвj — вартість відходів j-го найменування, грн/кг.

Проведені розрахунки зведені в таблицю 5.8.

Таблиця 5.8 — Витрати на матеріали

Найменування матеріалу, марка, тип, сорт	Ціна за 1 кг, грн	Норма витрат, шт	Вартість витраченого матеріалу, грн
Припой ПОС-61	950	0,05	0,0025
Флюс-паста NC-559-ASM	2500	0,09	0,0081
Спирт ізопропіловий	250	0,023	0,000529
Флешка	300	1	1
Дріт монтажний МГТФ	220	0,04	0,0016
Термоусадка набір	3000	0,04	0,0016
З врахуванням коефіцієнта транспортування			1,12

Розрахунок витрат на комплектуючі.

Витрати на комплектуючі вироби *($К_в$)*, які використовують при дослідженні нового технічного рішення, розраховуються, згідно з їхньою номенклатурою, за формулою:

$$К_в = \sum_{j=1}^{n} Н_j \cdot Ц_j \cdot К_j \qquad (5.12)$$

де $Н_j$ — кількість комплектуючих *j*-го виду, шт.;

$Ц_j$ — покупна ціна комплектуючих *j*-го виду, грн;

$К_j$ — коефіцієнт транспортних витрат, ($К_j$ *= 1,1 ... 1,15*).

Проведені розрахунки бажано звести до таблиці 5.9.

Таблиця 5.9 — Витрати на комплектуючі

Найменування комплектуючих	Кількість, шт.	Ціна за штуку, грн	Сума, грн
ESP32-WROOM-32	1	130	130
Датчик MQ-4	1	80	80
DHT11	1	40	40
OLED SSD1306 0.96"	1	100	100
Блок живлення 5V 2A	1	100	100
Резистори набір	1	45	45
Конденсатори (різні)	20	3,00	60
Кнопки, LED, роз'єми	2	100	200
Корпус пластиковий	1	100	100
Запасні компоненти	1	150	150
Всього з врахуванням транспортних витрат			1105,50

В спрощеному вигляді амортизаційні відрахування по кожному виду обладнання, приміщень та програмному забезпеченню тощо, можуть бути розраховані з використанням прямолінійного методу амортизації за формулою:

$$A_{обл} = \frac{Ц_б}{T_в} \cdot \frac{t_{вик}}{12}, \qquad (5.13)$$

де $Ц_б$ — балансова вартість обладнання, програмних засобів, приміщень тощо, які використовувались для проведення досліджень, грн;

$t_{вик}$ — термін використання обладнання, програмних засобів, приміщень під час досліджень, місяців;

$T_в$ — строк корисного використання обладнання, програмних засобів, приміщень тощо, років.

Проведені розрахунк необхідно звести до таблиці 5.10.

До статті «Паливо та енергія для науково-виробничих цілей» відносяться витрати на всі види палива й енергії, що безпосередньо використовуються з технологічною метою на проведення досліджень.

$$В_е = \sum_{i=1}^{n} \frac{W_{yt} \cdot t_i \cdot Ц_е \cdot К_{впі}}{\eta_i} \qquad (4.14)$$

де W_{yt} — встановлена потужність обладнання на певному етапі розробки, кВт;

t_i — тривалість роботи обладнання на етапі дослідження, год;

$Ц_е$ — вартість 1 кВт-години електроенергії, грн;

$К_{впі}$ — коефіцієнт, що враховує використання потужності, $К_{впі} < 1$;

η_i — коефіцієнт корисної дії обладнання, $\eta_i < 1$.

Таблиця 5.10 — Амортизаційні відрахування по кожному виду обладнання

Найменування обладнання	Балансова вартість, грн	Строк корисного використання, років	Термін використання обладнання, місяців	Амортизаційні відрахування, грн
Комп'ютер	25000	2	2	2083,33
Всього				2083,33

Для написання магістерської роботи використовується персональний комп’ютер для якого розрахуємо витрати на електроенергію.

$$B_e = \frac{0,5 \cdot 190 \cdot 12,69 \cdot 0,5}{0,8} = 753,47$$

Витрати за статтею «Службові відрядження» розраховуються як 20…25% від суми основної заробітної плати дослідників та робітників за формулою:

$$B_{св} = (З_о + З_р) * \frac{Н_{св}}{100\%}, \quad (5.15)$$

де $Н_{св}$ — норма нарахування за статтею «Службові відрядження».

$$B_{св} = 0,2 * (44762 + 978,6) = 9148,1$$

Накладні (загальновиробничі) витрати Внзв охоплюють: витрати на управління організацією, оплата службових відряджень, витрати на утримання, ремонт та експлуатацію основних засобів, витрати на опалення, освітлення, водопостачання, охорону праці тощо. Накладні (загальновиробничі) витрати Внзв можна прийняти як (100…150)% від суми основної заробітної плати розробників та робітників, які виконували дану МКНР, тобто:

$$B_{нзв} = (З_о + З_р) \cdot \frac{Н_{нзв}}{100\%}, \quad (5.16)$$

де $Н_{нзв}$ — норма нарахування за статтею «Інші витрати».

$$B_{нзв} = (44762 + 978,6) \cdot \frac{100}{100\%} = 45740,47 грн$$

Сума всіх попередніх статей витрат дає витрати, які безпосередньо стосуються даного розділу МКНР

В=44752+978,6+5031,45+11169,82+1,12+1105,5+2083,33+753,47+9148,1+4574,47= 120773,74грн

Прогнозування загальних втрат ЗВ на виконання та впровадження результатів виконаної МКНР здійснюється за формулою:

$$ЗВ = \frac{В}{\eta}, \qquad (5.17)$$

де η — коефіцієнт, який характеризує стадію виконання даної НДР.

Оскільки, робота знаходиться на стадії науково-дослідних робіт, то коефіцієнт $\beta = 0{,}7$.

Звідси:

$ЗВ = \frac{120773{,}74}{0{,}7} = 172533{,}91$ грн.

\5.3 Розрахунок економічної ефективності науково-технічної розробки

У даному підрозділі кількісно спрогнозуємо, яку вигоду, зиск можна отримати у майбутньому від впровадження результатів виконаної наукової роботи. Розрахуємо збільшення чистого прибутку підприємства $\Delta П_i$, для кожного із років, протягом яких очікується отримання позитивних результатів від впровадження розробки, за формулою

$$\Delta П_i = \sum_{1}^{n} \left(\Delta Ц_o \cdot N + Ц_o \cdot \Delta N\right)_i \cdot \lambda \cdot \rho \cdot \left(1 - \frac{\upsilon}{100}\right) \qquad (5.18)$$

де $\Delta Ц_0$ — покращення основного оціночного показника від впровадження результатів розробки у даному році.

N — основний кількісний показник, який визначає діяльність підприємства у даному році до впровадження результатів наукової розробки;

ΔN —— покращення основного кількісного показника діяльності підприємства від впровадження результатів розробки:

$Ц_о$ — основний оціночний показник, який визначає діяльність підприємства у даному році після впровадження результатів наукової розробки;

n — кількість років, протягом яких очікується отримання позитивних результатів від впровадження розробки:

л — коефіцієнт, який враховує сплату податку на додану вартість. Ставка податку на додану вартість дорівнює 20%, а коефіцієнт *л* = 0,8333.

р — коефіцієнт, який враховує рентабельність продукту. р = 0,25;

х — ставка податку на прибуток. У 2025 році — 18%.

Припустимо, що ціна зросте на 500 грн. Кількість одиниць реалізованої продукції також збільшиться: протягом першого року на 300 шт., протягом другого року — на 400 шт., протягом третього року на 500 шт. Реалізація продукції до впровадження розробки складала 1 шт., а її ціна до 2000 грн. Розрахуємо прибуток, яке отримає підприємство протягом трьох років.

$$\Delta П_1 = [500 \cdot 1 + (2000 + 500) \cdot 300] \cdot 0{,}833 \cdot 0{,}25 \cdot \left(1 + \frac{18}{100}\right) = 128205{,}29 \text{ грн.}$$

$$\Delta П_2 = [500 \cdot 1 + (2000 + 500) \cdot (300 + 400)] \cdot 0{,}833 \cdot 0{,}25 \cdot \left(1 + \frac{18}{100}\right) = 299446{,}38 \text{ грн.}$$

$$\Delta П_3 = [500 \cdot 1 + (2000 + 500) \cdot (300 + 400 + 500)] \cdot 0{,}833 \cdot 0{,}25 \cdot \left(1 + \frac{18}{100}\right) = 512979{,}5 \text{ грн.}$$

5.4 Розрахунок ефективності вкладених інвестицій та періоду їх окупності

Розрахуємо основні показники, які визначають доцільність фінансування наукової розробки певним інвестором, є абсолютна і відносна ефективність вкладених інвестицій та термін їх окупності.

Розрахуємо величину початкових інвестицій PV, які потенційний інвестор має вкласти для впровадження і комерціалізації науково-технічної розробки.

$$PV = k_{\text{інв}} \cdot \text{ЗВ}, \qquad (5.19)$$

де $k_{\text{інв}}$ — коефіцієнт, що враховує витрати інвестора на впровадження науково-технічної розробки та її комерціалізацію.

Це можуть бути витрати на підготовку приміщень, розробку технологій, навчання персоналу, маркетингові заходи тощо ($k_{\text{інв}}$ = 2…5).

$$PV = 2 \cdot 172533{,}91 = 345067{,}83$$

Розрахуємо абсолютну ефективність вкладених інвестицій $E_{абс}$ згідно наступної формули:

$$E_{абс} = (ПП - PV) \qquad (5.20)$$

де ПП — приведена вартість всіх чистих прибутків, що їх отримає підприємство від реалізації результатів наукової розробки, грн.;

$$ПП = \sum_{1}^{T} \frac{\Delta П_i}{(1+\tau)^t}, \qquad (5.21)$$

де $\Delta П_i$ – збільшення чистого прибутку у кожному із років, протягом яких виявляються результати виконаної та впровадженої НДЦКР, грн.;

Т – період часу, протягом якою виявляються результати впровадженої НДДКР, роки;

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні; для України цей показник знаходиться на рівні 0,2;

t — період часу (в роках).

$$ПП = \frac{128205,29}{(1+0,2)^1} + \frac{299446,38}{(1+0,2)^2} + \frac{512979,5}{(1+0,2)^3} = 613030,51 \text{ грн.}$$

$$Е_{абс} = (613030,51 - 345067,83) = 267962,67 \text{ грн.}$$

Оскільки $E_{абс} > 0$ то вкладання коштів на виконання та впровадження результатів НДДКР може бути доцільним.

Розрахуємо відносну (щорічну) ефективність вкладених в наукову розробку інвестицій $E_в$. Для цього користуються формулою:

$$E_в = \sqrt[T_ж]{1 + \frac{E_{абс}}{PV}} - 1, \qquad (5.22)$$

де $T_ж$ – життєвий цикл наукової розробки, роки.

$$Е_в = \sqrt[3]{1 + \frac{267962,67}{345067,83}} - 1 = 0,37 = 37\%$$

Визначимо мінімальну ставку дисконтування, яка у загальному вигляді визначається за формулою:

$$\tau = d + f, \qquad (5.23)$$

де d — середньозважена ставка за депозитними операціями в комерційних банках; в 2025 році в Україні d= (0,14…0,2);

f−показник, що характеризує ризикованість вкладень; зазвичай, величина f = (0,05...0,1).

$$\tau_{\min} = 0,18 + 0,05 = 0,23$$

Так як $E_в > \tau_{\min}$ то інвестор може бути зацікавлений у фінансуванні даної наукової розробки.

Розрахуємо термін окупності вкладених у реалізацію наукового проекту інвестицій за формулою:

$$T_{ок} = \frac{1}{E_в} \qquad (5.24)$$

$$\mathrm{T_{ок}} = \frac{1}{0,37} = 2,7 \text{ роки}$$

Так як $T_{ок} \leq$ 3...5-ти років, то фінансування даної наукової розробки в принципі є доцільним.

5.5 Висновки до економічного розділу

Результати здійсненого технологічного аудиту вказують на рівень вище середнього комерційного потенціалу. У порівнянні з аналогічним виробом виявлено, що нова розробка вищої якості і більш конкурентоспроможна, як з технічних, так і економічних позначень.

Вкладені інвестиції в даний проект окупляться через 2,7 роки. Загальні витрати складають 172533,91 грн. Прибуток за три роки складає 613030,51 грн.

ВИСНОВКИ

Під час виконання магістерської роботи було проведено комплексний аналіз сучасних методів та засобів віддаленого моніторингу IoT-пристроїв. Виконано порівняльне дослідження протоколів передачі даних (MQTT, HTTP, CoAP, WebSocket), хмарних платформ (AWS IoT Core, Azure IoT Hub) та opensource рішень (Eclipse Mosquitto, ThingsBoard). Визначено оптимальні технологічні рішення для побудови ефективної системи моніторингу з низькою латентністю та високою надійністю.

Розроблено архітектуру системи віддаленого моніторингу на базі п'ятирівневої моделі з Edge/Fog Computing. Обґрунтовано вибір мікроконтролера ESP32-WROOM-32 як оптимальної апаратної платформи, що поєднує достатню продуктивність, енергоефективність, вбудовані модулі зв'язку та доступну вартість. Визначено протокол MQTT як найбільш придатний для систем IoT-моніторингу завдяки мінімальному енергоспоживанню та надійним механізмам доставки повідомлень.

Реалізовано повнофункціональний прототип IoT-пристрою для моніторингу якості повітря з інтеграцією датчиків температури, вологості та концентрації газу. Розроблено програмне забезпечення для ESP32 з підтримкою MQTT-комунікації, механізмів автоматичного перепідключення та offline buffering для забезпечення надійної роботи при нестабільному інтернет-з'єднанні. Система забезпечує передачу телеметричних даних у режимі реального часу та миттєве відправлення критичних сповіщень.

Створено серверну інфраструктуру на базі Eclipse Mosquitto MQTT-брокера та Node.js застосунку для обробки телеметрії. Реалізовано багатоканальну систему сповіщень через Telegram Bot API та Twilio для забезпечення своєчасного інформування користувачів про критичні ситуації. Розроблено веб-інтерфейс для візуалізації показників у реальному часі з інтерактивними графіками історії даних. Проведено експериментальні дослідження розробленої системи в реальних умовах експлуатації. Результати тестування підтвердили високу ефективність

системи: латентність передачі критичних сповіщень становить 2-5 секунд, надійність доставки повідомлень перевищує 99%, забезпечено стабільну роботу при тимчасових розривах з'єднання. Виконано порівняння з існуючими комерційними рішеннями, яке показало конкурентоспроможність розробленої системи за показниками вартості та функціональності.

Розраховано економічну доцільність впровадження системи віддаленого моніторингу. Проведено оцінку комерційного потенціалу, визначено потенційні ринки застосування, що включають житловий сектор, комерційні будівлі та системи екологічного моніторингу. Визначено перспективні напрямки подальшого розвитку системи, включаючи інтеграцію додаткових типів датчиків, реалізацію механізмів машинного навчання та розробку мобільного застосунку.

Таким чином, всі поставлені завдання виконано, а поставлена мета роботи щодо дослідження методів та засобів віддаленого моніторингу IoT-пристроїв та розробки ефективної системи моніторингу досягнута. Розроблена система має високий ступінь надійності, масштабованості та економічної ефективності і може використовуватися в різних практичних сценаріях моніторингу параметрів навколишнього середовища та технічних об'єктів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. С.В. Богомолов, Д.В. Степанчук, М.П. Півнюк Доповідь на тему «Порівняльний аналіз протоколів передачі даних для систем віддаленого моніторингу IoT пристроїв [Електронний ресурс].Режим доступу: https://conferences.vntu.edu.ua/index.php/mn/mn2026/paper/viewFile/26844/94495/

2. International Data Corporation (IDC). The Growth in Connected IoT Devices is Expected to Generate 79.4ZB of Data in 2025, According to a New IDC Forecast, [Електронний ресурс]. Режим доступу: https://www.businesswire.com/news/home/20190618005012/en/

3. Методичні вказівки до виконання економічної частини магістерських кваліфікаційних робіт / Уклад. : В. О. Козловський, О. Й. Лесько, В. В. Кавецький. — Вінниця : ВНТУ, 2021. — 42 с.

4. Internet of Things (IoT): An Overview [Електронний ресурс]. — Режим доступу: https://www.oracle.com/internet-of-things/what-is-iot/

5. What is the Internet of Things? [Електронний ресурс]. — Режим доступу: https://www.ibm.com/topics/internet-of-things

6. IoT Architecture Explained [Електронний ресурс]. — Режим доступу: https://www.geeksforgeeks.org/iot-architecture/

7. IoT Communication Protocols Overview [Електронний ресурс]. — Режим доступу: https://www.postscapes.com/internet-of-things-protocols/

8. MQTT Essentials Guide [Електронний ресурс]. — Режим доступу: https://www.hivemq.com/mqtt-essentials/

9. MQTT Protocol Specification v3.1.1 [Електронний ресурс]. — Режим доступу: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/

10. Comparison of IoT Protocols: MQTT, CoAP, HTTP [Електронний ресурс]. — Режим доступу: https://www.baeldung.com/cs/iot-protocols-comparison

11. CoAP Protocol Explained [Електронний ресурс]. — Режим доступу: https://datatracker.ietf.org/doc/html/rfc7252

12. HTTP vs MQTT for IoT Applications [Електронний ресурс]. — Режим доступу: https://www.emqx.com/en/blog/mqtt-vs-http

13. ESP32 Technical Documentation [Електронний ресурс]. — Режим доступу: https://docs.espressif.com/projects/esp-idf/

14. Arduino Official Documentation [Електронний ресурс]. — Режим доступу: https://docs.arduino.cc/

15. Raspberry Pi Documentation [Електронний ресурс]. — Режим доступу: https://www.raspberrypi.com/documentation/

16. NodeMCU Platform Overview [Електронний ресурс]. — Режим доступу: https://nodemcu.readthedocs.io/

17. Overview of IoT Platforms [Електронний ресурс]. — Режим доступу: https://www.iotforall.com/iot-platforms-comparison

18. AWS IoT Core Documentation [Електронний ресурс]. — Режим доступу: https://docs.aws.amazon.com/iot/

19. Azure IoT Hub Documentation [Електронний ресурс]. — Режим доступу: https://learn.microsoft.com/azure/iot-hub/

20. Google Cloud IoT Overview [Електронний ресурс]. — Режим доступу: https://cloud.google.com/iot-core/docs

21. Server-side Technologies for IoT Systems [Електронний ресурс]. — Режим доступу: https://www.redhat.com/en/topics/iot/iot-architecture

22. Node.js Documentation [Електронний ресурс]. — Режим доступу: https://nodejs.org/en/docs/

23. Client—Server Architecture [Електронний ресурс]. — Режим доступу: https://training.qatestlab.com/blog/technical-articles/client-server-architecture

24. Схеми баз даних: основи проєктування [Електронний ресурс]. — Режим доступу: https://foxminded.ua/skhemy-bazy-danyh

25. Що таке REST API [Електронний ресурс]. — Режим доступу: https://foxminded.ua/shcho-take-rest-api

26. Найкращі практики проєктування REST API [Електронний ресурс]. — Режим доступу: https://devzone.org.ua/post/naykrashchi-praktyky-proyektuvannia-rest-api

27. REST API: основи та приклад реалізації на Python Flask [Електронний ресурс]. — Режим доступу: https://cloud.itstep.org/blog/rest-api-what-is-it-restful-project-on-python-flask

28. Docker Documentation [Електронний ресурс]. — Режим доступу: https://docs.docker.com/

29. Introduction to Microservices Architecture [Електронний ресурс]. — Режим доступу: https://microservices.io/

30. Grafana Documentation [Електронний ресурс]. — Режим доступу: https://grafana.com/docs/

31. InfluxDB Documentation [Електронний ресурс]. — Режим доступу: https://docs.influxdata.com/

32. Prometheus Monitoring System [Електронний ресурс]. — Режим доступу: https://prometheus.io/docs/

33. Data Visualization Best Practices [Електронний ресурс]. — Режим доступу: https://www.smashingmagazine.com/data-visualization-guidelines/

34. Chart.js Documentation [Електронний ресурс]. — Режим доступу: https://www.chartjs.org/docs/

35. Software Testing Fundamentals [Електронний ресурс]. — Режим доступу: https://www.softwaretestinghelp.com/software-testing-fundamentals/

36. ISO/IEC 25010 Software Quality Model [Електронний ресурс]. — Режим доступу: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

37. Performance Testing Basics [Електронний ресурс]. — Режим доступу: https://www.guru99.com/performance-testing.html

38. Latency in Network Systems Explained [Електронний ресурс]. — Режим доступу: https://www.cloudflare.com/learning/performance/what-is-latency/

ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України

Вінницький національний технічний університет

Факультет інформаційних технологій та комп’ютерної інженерії

Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ

Завідувач кафедри ОТ

д.т.н., проф. Азаров О. Д.

_____

“ 3 “ жовтня 2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання магістерської кваліфікаційної роботи

“ Метод та засоби віддаленого моніторингу IoT пристроїв ”

Науковий керівник: к.т.н., проф.

_____Богомолов С.В.

Виконав: студент гр. 1КІ-24м

_____Степанчук Д.В.

ВНТУ 2025

1 Підстава для виконання магістерської кваліфікаційної роботи (МКР)

1.1 Актуальність полягає в тому, що стрімкий розвиток технологій Інтернету речей (IoT) супроводжується зростанням кількості підключених пристроїв у промисловості, медицині, розумних будинках та екологічному моніторингу. Ефективне управління такими системами потребує надійного збору телеметричних даних, їх оперативної обробки та своєчасного реагування на критичні події. Традиційні підходи до моніторингу характеризуються високою латентністю передачі даних, обмеженою масштабованістю та складністю інтеграції різнорідних пристроїв. Актуальним є розробка систем віддаленого моніторингу з використанням протоколу MQTT, хмарних технологій та веб-інтерфейсів реального часу, що забезпечують низьку затримку сповіщень, енергоефективність та можливість централізованого управління розподіленими IoT-пристроями.

1.2 Наказ про затвердження теми МКР №313 від 24.09.2025 р.

2 Мета і призначення МКР

2.1 Метою роботи є розробка системи віддаленого моніторингу IoT-пристроїв з низькою латентністю передачі телеметричних даних, що включає проектування апаратної частини на базі ESP32, серверної інфраструктури з MQTT-брокером та веб-інтерфейсу для візуалізації параметрів у реальному часі.

2.2 Призначення розробки — забезпечення ефективного моніторингу параметрів навколишнього середовища (температури, вологості, концентрації газів) у розподілених системах IoT шляхом використання протоколу MQTT для передачі даних, PostgreSQL для зберігання телеметрії, та веб-dashboard для візуалізації з можливістю налаштування порогових значень, формування сповіщень та аналізу історичних даних.3 Вихідні дані для виконання МКР

3 Вихідні дані до виконання МКР

3.1 Аналіз існуючих методів та засобів віддаленого моніторингу IoT-пристроїв, вивчення протоколів передачі даних (MQTT, HTTP, CoAP), платформ

IoT-моніторингу та архітектурних підходів до побудови систем збору телеметрії.

3.2 Проектування апаратної частини IoT-пристрою на базі мікроконтролера ESP32-WROOM-32 з датчиками DHT11 (температура/вологість) та MQ-4 (метан), включаючи вибір компонентної бази, розробку схеми підключення та налаштування енергоефективних режимів роботи.

3.3 Розробка серверної інфраструктури з використанням MQTT-брокера Mosquitto, Node.js backend для обробки телеметрії, PostgreSQL для зберігання даних та WebSocket для real-time комунікації з веб-інтерфейсом.

3.4 Створення веб-dashboard з використанням React/HTML5 для візуалізації телеметричних даних у реальному часі, налаштування порогових значень, формування сповіщень та аналізу історичних трендів.

3.5 Проведення експериментального тестування системи на предмет стабільності передачі даних, вимірювання латентності, перевірки коректності відображення телеметрії та порівняння з існуючими рішеннями.

3.6 Виконання економічних розрахунків для оцінювання комерційного потенціалу розробки, прогнозування витрат на виконання науково-дослідної роботи та розрахунку економічної ефективності впровадження системи моніторингу.

4 Вимоги до виконання МКР

Головна вимога — використати протокол MQTT як основний метод передачі телеметричних даних від IoT-пристрою до серверної інфраструктури з забезпеченням низької латентності (менше 5 секунд від зчитування датчика до відображення на веб-інтерфейсі).

Система повинна забезпечувати ефективну інтеграцію апаратної частини на базі ESP32 з серверним застосунком для збору, обробки та візуалізації даних у реальному часі. Реалізовані алгоритми повинні враховувати нестабільність WiFi-з'єднання (реалізація механізмів повторного підключення та offline buffering), енергоефективність IoT-пристрою (підтримка Deep Sleep режиму), масштабованість системи (можливість підключення множини пристроїв через

ієрархічну структуру MQTT-топіків).

Усі розробки мають відповідати вимогам надійності передачі даних (успішність доставки >98%), точності вимірювань датчиків (відхилення DHT11 ±2°C, ±5% RH), безпеки (використання TLS для MQTT-з'єднання)

5. Етапи МКР та очікувані результати

Етапи роботи та очікувані результати приведено в таблиці А.1.

Таблиця А.1 — Етапи МКР

№ етапу	Назва етапу	Термін виконання		Очікувані результати
		початок	кінець	
1	2	3	4	5
1	Огляд і аналіз джерел інформації	20.09.2025	04.10.2025	Розділ 1
2	Теоретичні дослідження методу та засобів віддаленого моніторингу IoT-пристроїв, пошук аналогів	05.10.2025	18.10.2025	Розділ 2
3	Обгрунтування та розробка серверної частини	19.10.2025	01.11.2025	Розділ 3
4	Розробка і тестування прототипа	02.11.2025	16.11.2025	Розділ 4
5	Економічна частина	17.11.2025	25.11.2025	Розділ 5

6 Матеріали, що подаються до захисту МКР

До захисту подаються: пояснювальна записка МКР, графічні і ілюстративні матеріали, протокол попереднього захисту МКР на кафедрі, відгук наукового керівника, відгук опонента, протоколи складання державних екзаменів, анотації до МКР українською та іноземною мовами.

7 Порядок контролю виконання та захисту МКР

Виконання етапів графічної та розрахункової документації МКР контролюється науковим керівником згідно зі встановленими термінами. Захист МКР відбувається на засіданні Екзаменаційної комісії, затвердженої наказом ректора.

8 Вимоги до оформлювання та порядок виконання МКР

8.1 При оформлюванні МКР використовуються:

— ДСТУ 3008: 2015 “Звіти в сфері науки і техніки. Структура та правила оформлювання”;

— ДСТУ 8302: 2015 “Бібліографічні посилання. Загальні положення та правила складання”;

— ГОСТ 2.104-2006 “Єдина система конструкторської документації. Основні написи”;

— методичні вказівки до виконання магістерських кваліфікаційних робіт зі спеціальності 123 — “Комп’ютерна інженерія”;

— документи на які посилаються у вище вказаних.

8.2 Порядок виконання МКР викладено в “Положення про кваліфікаційні роботи на другому (магістерському) рівні вищої освіти СУЯ ВНТУ-03.02.02-П.001.01:21”.

ДОДАТОК Б

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Методи та засоби віддаленого моніторингу IoT пристроїв

Тип роботи: магістерська кваліфікаційна робота
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ кафедра обчислювальної техніки, ФІТКІ, 1КІ-24м
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism (КП1) 1 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

- ✓ Запозичення, виявлені у роботі, оформлені коректно і не містять ознак академічного плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту.
- ☐ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.
- ☐ У роботі виявлено ознаки академічного плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень. Робота до захисту не приймається.

Експертна комісія:

Азаров О. Д., д.т.н., зав. каф. ОТ _____
(прізвище, ініціали, посада) (підпис)

Мартинюк Т. Б., д.т.н., проф. каф. ОТ _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ Захарченко С. М.
(підпис) (прізвище, ініціали)

З висновком експертної комісії ознайомлений(-на)

Керівник _____ Богомолов С.В
(підпис) (прізвище, ініціали, посада)

Здобувач _____ Степанчук Д. В.
(підпис) (прізвище, ініціали)

ДОДАТОК В

Лістинг прошивки ESP32

```
#include <WiFi.h>
#include <PubSubClient.h>
#include <DHT.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include <ArduinoJson.h>
const char* ssid = "WiFi_Name";
const char* password = "WiFi_Pass";
const char* mqtt_server = "192.168.1.100";
const char* mqtt_user = "iot_user";
const char* mqtt_pass = "iot_pass";
#define DHTPIN 4
#define DHTTYPE DHT11
#define MQ4_PIN 34
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
DHT dht(DHTPIN, DHTTYPE);
WiFiClient espClient;
PubSubClient client(espClient);
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, -1);
float temperature = 0;
float humidity = 0;
int gasLevel = 0;
unsigned long lastMsg = 0
void setup() {
```

```
  Serial.begin(115200);
  dht.begin();
  if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
    Serial.println("OLED failed");
  }
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  client.setServer(mqtt_server, 1883);
  reconnect();
}
void reconnect() {
  while (!client.connected()) {
    if (client.connect("ESP32Client", mqtt_user, mqtt_pass)) {
      Serial.println("MQTT connected");
      client.subscribe("iot/device01/command");
    } else {
      delay(5000);
    }
  }

void readSensors() {
  temperature = dht.readTemperature();
  humidity = dht.readHumidity();
  gasLevel = analogRead(MQ4_PIN);
  if (isnan(temperature) || isnan(humidity)) {
```

```
    temperature = 0;
    humidity = 0;
  }
}
void sendData() {
  StaticJsonDocument<200> doc;
  doc["device_id"] = "ESP32_Device_01";
  doc["temperature"] = temperature;
  doc["humidity"] = humidity;
  doc["gas_level"] = gasLevel;
  doc["timestamp"] = millis();
  char buffer[200];
  serializeJson(doc, buffer);
  client.publish("iot/device01/telemetry", buffer);

  if (temperature > 35 || gasLevel > 400) {
    client.publish("iot/device01/alert", "WARNING");
  }
}
void updateDisplay() {
  display.clearDisplay();
  display.setTextSize(1);
  display.setTextColor(SSD1306_WHITE);
  display.setCursor(0, 0);
  display.println("IoT Monitor");
  display.print("Temp: ");
  display.print(temperature, 1);
  display.println(" C");
```

```
  display.print("Hum: ");
  display.print(humidity, 1);
  display.println(" %");
  display.print("Gas: ");
  display.println(gasLevel);
  display.display();
}
void loop() {
  if (!client.connected()) {
    reconnect();
  }
  client.loop();
  unsigned long now = millis();
  if (now - lastMsg > 5000) {
    lastMsg = now;
    readSensors();
    sendData();
    updateDisplay();
  }
}
```

ДОДАТОК Г

Лістинг серверної частини

```
 from ultralytics import YOLO
const mqtt = require('mqtt');
const express = require('express');
const { Pool } = require('pg');
const cors = require('cors');
const WebSocket = require('ws');
const app = express();
app.use(cors());
app.use(express.json());
const pool = new Pool({
  user: 'postgres',
  host: 'localhost',
  database: 'iot_monitoring',
  password: 'password',
  port: 5432,
});
const mqtt_client = mqtt.connect('mqtt://localhost:1883', {
  username: 'iot_user',
  password: 'iot_pass'
});
const wss = new WebSocket.Server({ port: 8081 });
let connected_clients = new Set();
wss.on('connection', (ws) => {
  connected_clients.add(ws);
  console.log('WebSocket client connected');
  ws.on('close', () => {
    connected_clients.delete(ws);
```

```
    console.log('WebSocket client disconnected');
  });
});
function broadcast_to_clients(data) {
  const message = JSON.stringify(data);
  connected_clients.forEach((client) => {
    if (client.readyState === WebSocket.OPEN) {
      client.send(message);
    }
  });

mqtt_client.on('connect', () => {
  console.log('Connected to MQTT broker');
  mqtt_client.subscribe('iot/+/telemetry');
  mqtt_client.subscribe('iot/+/status');
  mqtt_client.subscribe('iot/+/status/alert');
});

mqtt_client.on('message', async (topic, message) => {
  try {
    const data = JSON.parse(message.toString());
    console.log(`Received message on ${topic}:`, data);
    if (topic.includes('/telemetry')) {
      await save_telemetry(data);
      broadcast_to_clients({ type: 'telemetry', data: data });
    } else if (topic.includes('/alert')) {
      await save_alert(data);
      broadcast_to_clients({ type: 'alert', data: data });
    } else if (topic.includes('/status')) {
      await update_device_status(data);
```

```
      broadcast_to_clients({ type: 'status', data: data });
    }
  } catch (error) {
    console.error('Error processing MQTT message:', error);
  }
});
async function save_telemetry(data) {
  const query = `
    INSERT INTO telemetry (device_id, temperature, humidity, gas_level, timestamp)
    VALUES ($1, $2, $3, $4, NOW()`;
  await pool.query(query, [
    data.device_id,
    data.temperature,
    data.humidity,
    data.gas_level
  ]);
}

async function save_alert(data) {
  const query = `
    INSERT INTO alerts (device_id, alert_type, level, value, timestamp)
    VALUES ($1, $2, $3, $4, NOW())
  `;
  await pool.query(query, [
    data.device_id,
    data.alert_type,
    data.level,
    data.value
  ]);
```

```
async function update_device_status(data) {
  const query = `
    INSERT INTO device_status (device_id, status, last_seen)
    VALUES ($1, 'online', NOW())
    ON CONFLICT (device_id)
    DO UPDATE SET status = 'online', last_seen = NOW()
  `;

  await pool.query(query, [data.device_id || 'ESP32_Device_01']);
}

app.get('/api/devices', async (req, res) => {
  try {
    const result = await pool.query('SELECT * FROM device_status ORDER BY
last_seen DESC');
    res.json(result.rows);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

app.get('/api/telemetry/:device_id', async (req, res) => {
  try {
    const { device_id } = req.params;
    const { limit = 100 } = req.query;

    const query = `
      SELECT * FROM telemetry
      WHERE device_id = $1
      ORDER BY timestamp DESC
```

```
      LIMIT $2
    `;

    const result = await pool.query(query, [device_id, limit]);
    res.json(result.rows);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

app.get('/api/telemetry/:device_id/latest', async (req, res) => {
  try {
    const { device_id } = req.params;

    const query = `
      SELECT * FROM telemetry
      WHERE device_id = $1
      ORDER BY timestamp DESC
      LIMIT 1
    `;
```

ДОДАТОК Д

Лістинг веб частини

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';
const API_URL = 'http://localhost:3000/api';
const WS_URL = 'ws://localhost:8081';
function App() {
  const [devices, setDevices] = useState([]);
  const [selectedDevice, setSelectedDevice] = useState('ESP32_Device_01');
  const [telemetry, setTelemetry] = useState(null);
  const [alerts, setAlerts] = useState([]);
  const [statistics, setStatistics] = useState(null);
  const [ws, setWs] = useState(null);
  const [connected, setConnected] = useState(false);
  useEffect(() => {
    fetchDevices();
    fetchLatestTelemetry();
    fetchAlerts();
    fetchStatistics();
    initWebSocket();
    const interval = setInterval(() => {
      fetchLatestTelemetry();
      fetchAlerts();
    }, 30000);
    return () => {
      clearInterval(interval);
      if (ws) ws.close();
    };
  }, [selectedDevice]);
```

```
const initWebSocket = () => {
  const socket = new WebSocket(WS_URL);
  socket.onopen = () => {
    console.log('WebSocket connected');
    setConnected(true);
  };
  socket.onmessage = (event) => {
    const message = JSON.parse(event.data);
    handleRealtimeUpdate(message);
  };
  socket.onclose = () => {
    console.log('WebSocket disconnected');
    setConnected(false);
    setTimeout(initWebSocket, 5000);
  };
  socket.onerror = (error) => {
    console.error('WebSocket error:', error);
    setConnected(false);
  };

  setWs(socket);
};
const handleRealtimeUpdate = (message) => {
  if (message.type === 'telemetry' && message.data.device_id === selectedDevice) {
    setTelemetry(message.data);
  } else if (message.type === 'alert') {
    setAlerts(prev => [message.data, ...prev].slice(0, 10));
  }
};
const fetchDevices = async () => {
```

```
    try {
      const response = await axios.get(`${API_URL}/devices`);
      setDevices(response.data);
    } catch (error) {
      console.error('Error fetching devices:', error);
    }
  };
  const fetchLatestTelemetry = async () => {
    try {
      const response = await
axios.get(`${API_URL}/telemetry/${selectedDevice}/latest`);
      setTelemetry(response.data);
    } catch (error) {
      console.error('Error fetching telemetry:', error);
    }
  };
  const fetchAlerts = async () => {
    try {
      const response = await
axios.get(`${API_URL}/alerts?device_id=${selectedDevice}&limit=10`);
      setAlerts(response.data);
    } catch (error) {
      console.error('Error fetching alerts:', error);
    }
  };
  const fetchStatistics = async () => {
    try {
      const response = await
axios.get(`${API_URL}/statistics/${selectedDevice}?hours=24`);
      setStatistics(response.data);
```

```
    } catch (error) {
      console.error('Error fetching statistics:', error);
    };
  const sendCommand = async (command) => {
    try {
      const response = await axios.post(`${API_URL}/command/${selectedDevice}`, {
command });
      alert(response.data.message);
    } catch (error) {
      console.error('Error sending command:', error);
      alert('Помилка відправки команди');
    }
  };
  return (
    <div className="app">
      <Header connected={connected} />
      <DeviceSelector
        devices={devices}
        selected={selectedDevice}
        onSelect={setSelectedDevice}
      />
```

ДОДАТОК Е

Діаграма алгоритму роботи пристрою

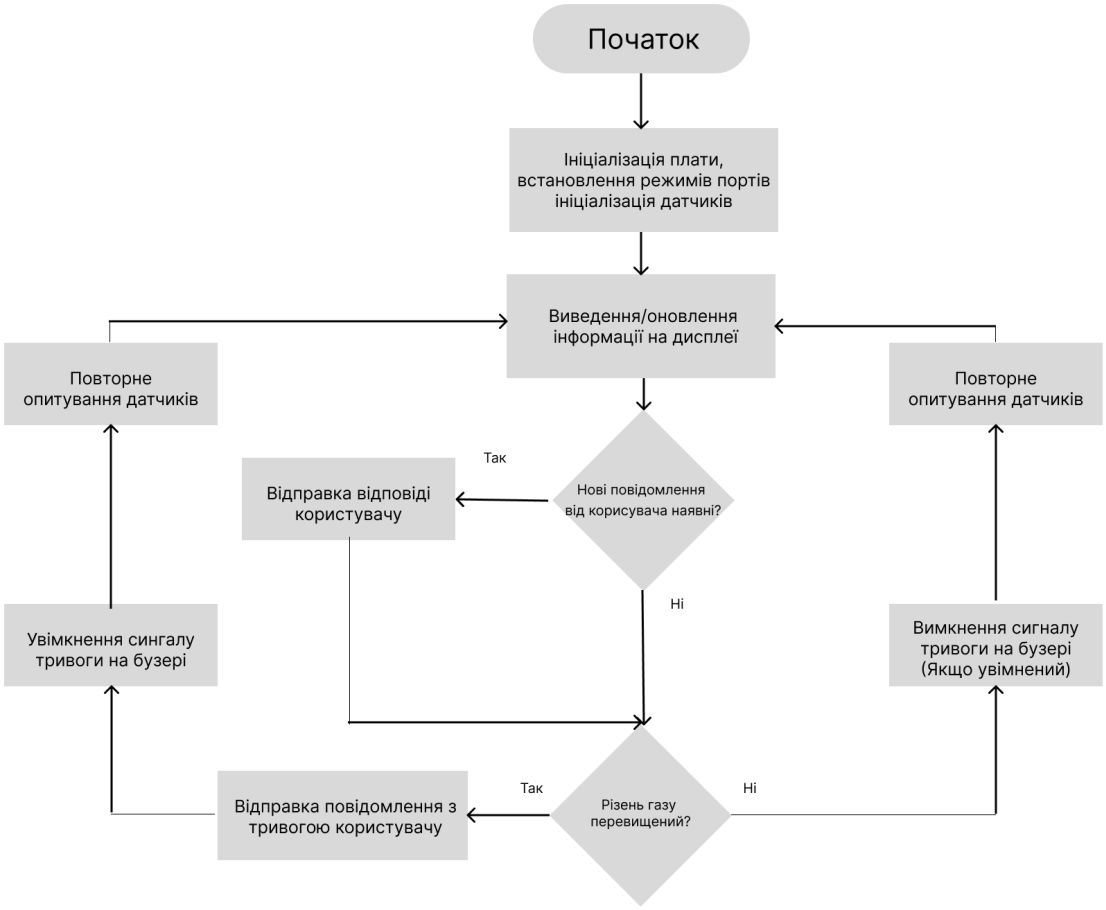

Рисунок Е.1 — Діаграма алгоритму роботи пристрою

ДОДАТОК Ж

Модель пристрою

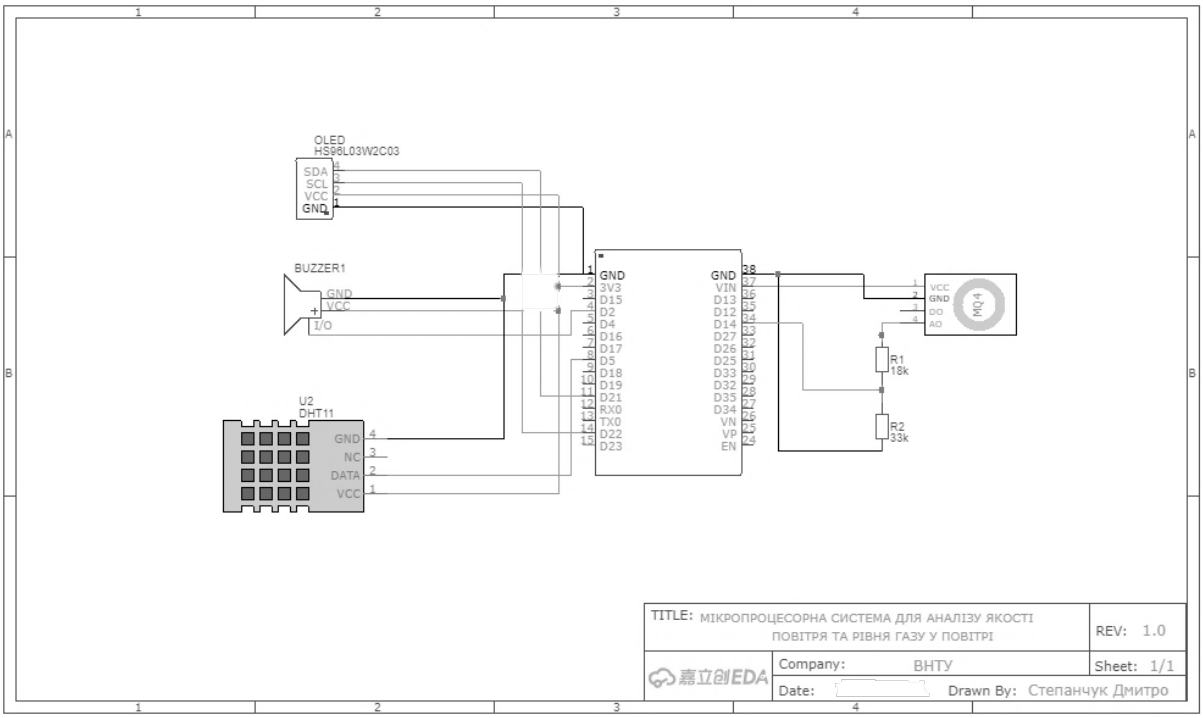

Рисунок Ж.1 — Схема підключення компонентів пристрою

ДОДАТОК З

Архітектура backend системи

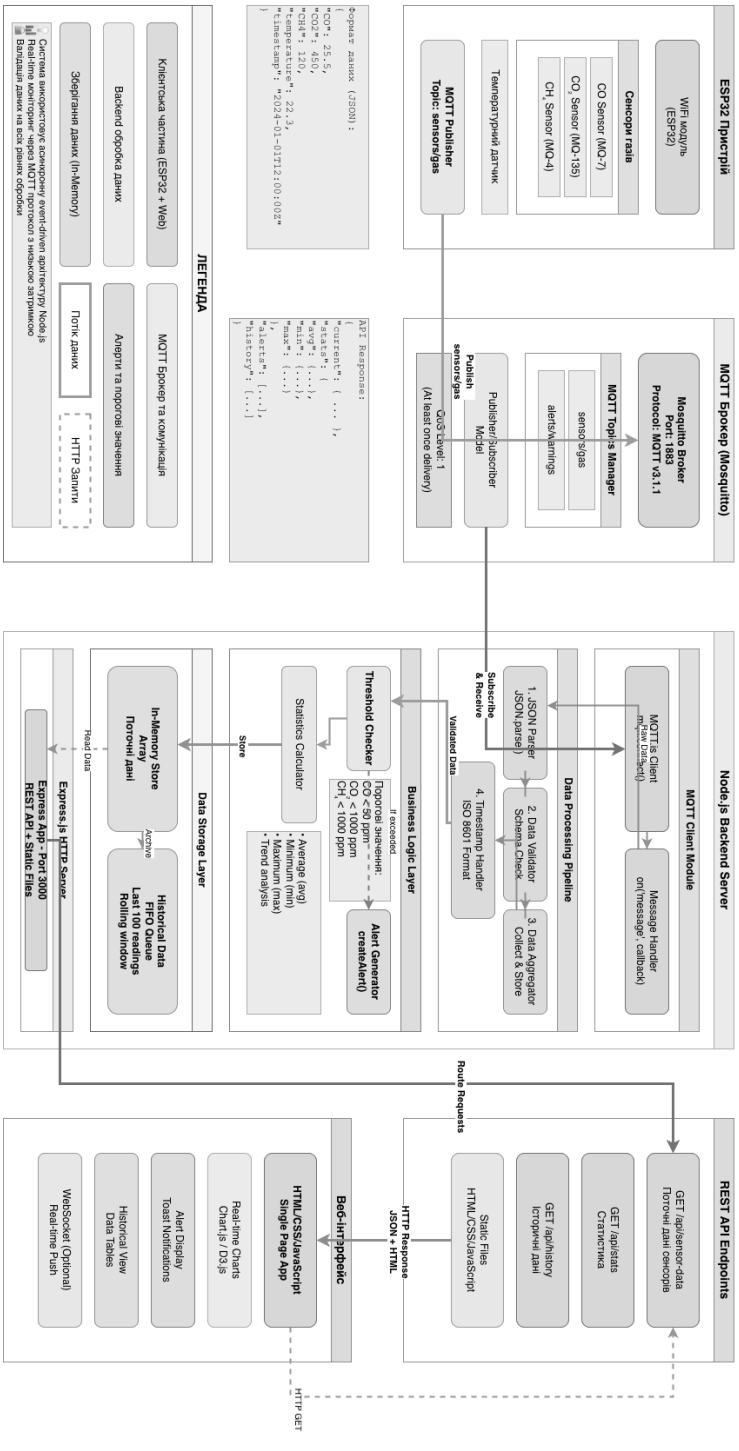

Рисунок 3.1 — Архітектурна схема backend системи