ІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп’ютерних наук

Кафедра комп'ютерної інженерії

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувачка кафедри,
д-р техн. наук, проф.

_____Ірина ЖУРАВСЬКА

« __ » _____ 202__ р.

КВАЛІФІКАЦІЙНА РОБОТА
НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ МАГІСТРА

Система емуляції та візуалізації IoT-даних у реальному часі з використанням протоколу WebSocket

Спеціальність 123 Комп’ютерна інженерія

Освітня програма «Комп’ютерна інженерія»

Здобувач _____Микита КАЙДАНОВИЧ
підпис

«__» _____ 202__ р.

Керівник д-р техн. наук, проф.,
зав. кафедри комп’ютерної інженерії _____Ірина ЖУРАВСЬКА
підпис
«__» _____ 202__ р.

Миколаїв – 2025

Факультет	Комп’ютерних наук
Кафедра	Комп’ютерної інженерії
Рівень вищої освіти	Другий (магістерський)
Освітній ступень	Магістр
Спеціальність	123 Комп’ютерна інженерія
Освітня програма	Комп’ютерна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри комп’ютерної інженерії
_____ Ірина ЖУРАВСЬКА
«_____» _____ 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу здобувача

Кайданович Микита Вячеславович
(прізвище, ім’я, по батькові здобувача)

1. Тема кваліфікаційної роботи
Система емуляції та візуалізації IoT-даних у реальному часі з використанням протоколу WebSocket

Затверджена наказом по ЧНУ ім. Петра Могили від 23.06.2025 № 165/1.

2. Строк представлення кваліфікаційної роботи « 10 » грудня 2025 р.

3. Очікуваний результат роботи та початкові дані, якщо такі потрібні
Проєкт передбачає розробку системи для роботи з потоковими IoT-даними в режимі soft real-time. Головна ідея полягає у використанні WebSocket для організації безперервного каналу зв'язку. Система має бути реалізована у вигляді клієнт-серверного застосунку, де серверна частина виступає емулятором сенсорної мережі, а клієнтська сторона реалізує візуалізацію отриманих масивів даних через динамічні графічні інтерфейси.

4. Перелік питань, що підлягають розробці:
Аналіз методів та протоколів обміну даними в IoT-системах та обґрунтування вибору WebSocket; розроблення архітектури системи емуляції та візуалізації IoT-даних у режимі реального часу; проєктування структури повідомлень та моделі генерації сенсорних даних; реалізація серверної частини на платформі Node.js із використанням бібліотеки ws; розроблення клієнтського інтерфейсу з інтерактивною візуалізацією на основі Chart.js; інтеграція механізмів збереження історії даних у базі SQLite та налаштування параметрів емуляції; проведення тестування системи на стійкість, масштабованість та затримку передачі даних.

5. Перелік графічних матеріалів

_____
_____
_____

6. Консультанти:

Консультант	Кафедра (організація)	Частина роботи

Керівник роботи _____ ___Ірина ЖУРАВСЬКА___

Особистий підпис *Власне ім'я ПРІЗВИЩЕ*

Здобувач _____ ___Микита КАЙДАНОВИЧ__

Особистий підпис *Власне ім'я ПРІЗВИЩЕ*

Дата видачі завдання «_01_» ___липня_____ 2025 р.

КАЛЕНДАРНИЙ ПЛАН

виконання кваліфікаційної магістерської роботи

Тема: Система емуляції та візуалізації IoT-даних у реальному часі з використанням протоколу WebSocket

№ з/п	Найменування роботи	Початок	Закінчення	Примітки
1	Розробка та затвердження завдання на виконання КМР	01.10.2025	04.10.2025	Виконано
2	Огляд літератури за темою роботи	05.10.2025	09.10.2025	Виконано
3	Складання календарного плану КМР	10.10.2025	13.10.2025	Виконано
4	Аналіз предметної області	14.10.2025	17.10.2025	Виконано
5	Розробка проєктних рішень	18.10.2025	22.10.2025	Виконано
6	Моделювання та конструювання АПЗ	23.10.2025	29.10.2025	Виконано
7	Перевірка працездатності, тестування та апробація розробленого АПЗ, аналіз результатів тестування	01.11.2025	13.11.2025	Виконано
8	Оформлення КМР та презентації	14.11.2025	24.11.2025	Виконано
9	Попередній захист	11.11.2025	25.11.2025	Виконано
10	Відгук керівника КМР	03.12.2025	07.12.2025	Виконано
11	Рецензування	05.12.2025	09.12.2025	Виконано
12	Завершення оформлення КМР та презентації	03.12.2025	10.12.2025	Виконано
13	Захист кваліфікаційної магістерської роботи	16.12.2025	17.12.2025	Виконано

Керівник роботи _____ ___Ірина ЖУРАВСЬКА___

Особистий підпис *Власне ім'я ПРІЗВИЩЕ*

Здобувач _____ ___Микита КАЙДАНОВИЧ__

Особистий підпис *Власне ім'я ПРІЗВИЩЕ*

«____» _____ 20__ р.

АНОТАЦІЯ

до кваліфікаційної магістерської роботи
«Система емуляції та візуалізації IoT-даних у реальному часі
з використанням протоколу WebSocket»
Студент гр. 605: Кайданович Микита Вячеславович
Керівник: д-р техн. наук, проф. ЖУРАВСЬКА І. М.

Магістерська кваліфікаційна робота присвячена розробці підходів до роботи з IoT-даними в режимі soft real-time без використання фізичного обладнання. Стрімкий розвиток сенсорних мереж робить натурні випробування архітектур обробки даних занадто дорогими та складними. Впровадження систем емуляції вирішує цю дилему, надаючи розробникам інструмент для варіативного моделювання сценаріїв та точного відтворення умов експерименту.

Об’єкт дослідження – процес обміну та обробки даних у системах Інтернету речей.

Предмет дослідження – технологічні підходи до емуляції сенсорних мереж, а також механізми транспортування та візуалізації телеметрії в режимі реального часу.

Мета роботи спрямована на створення повнофункціональної системи, яка поєднує генератор даних (віртуальні сенсори) та візуальний інтерфейс. Основою комунікації обрано протокол WebSocket, що забезпечує безперервний потік даних між компонентами системи.

Практична значимість роботи полягає у створенні інструмента, який вирішує проблему доступності апаратного забезпечення: він дозволяє тестувати IoT-сценарії та алгоритми у повністю віртуальному середовищі. Це робить систему незамінною для навчальних курсів та лабораторних робіт, а також дозволяє використовувати її як платформу для прототипування при розробці нових промислових рішень.

Робота пройшла апробацію на XXVIII Всеукраїнській науково-практичної конференції «Могилянські читання – 2025» (Миколаїв, листопад 2025 р.).

Основні результати роботи за розділами:

У першому розділі проведено аналіз сучасних підходів до організації обміну IoT-даними, розглянуто особливості потокової передачі сенсорної інформації, а також виконано порівняння існуючих інструментів для емуляції та моніторингу показників у реальному часі. Обґрунтовано вибір протоколу WebSocket як основи для реалізації двонаправленого каналу передачі.

У другому розділі розроблено архітектуру системи, що включає серверну частину для генерації синтетичних даних, клієнтський модуль для візуалізації та сховище історичних даних. Побудовано логічні та функціональні схеми взаємодії компонентів, визначено структуру JSON-повідомлень і механізми керування параметрами емуляції.

У третьому розділі реалізовано програмний прототип системи на основі платформи Node.js із використанням бібліотеки ws для WebSocket-з’єднань. На стороні клієнта виконано інтеграцію з Chart.js для оновлюваної візуалізації потоків

даних. Здійснено збереження історії вимірювань до бази SQLite, а також реалізовано механізми імітації мережевих збоїв та шумових варіацій сигналів.

У четвертому розділі проведено тестування системи щодо затримки передачі, стабільності при підключенні множинних клієнтів і навантажувальної стійкості. Результати підтвердили, що запропонована система забезпечує передачу даних у реальному часі з мінімальною затримкою та може бути масштабована в межах лабораторних і навчальних сценаріїв.

Робота складається з 73 сторінок, 7 таблиць, 19 рисунків та 3 додатків. У процесі дослідження використано 23 джерела посилання.

Ключові слова: *IoT, емуляція даних, WebSocket, візуалізація в реальному часі, Node.js, Chart.js, сенсорні системи.*

ABSTRACT

of the Master's Thesis

" System for Real-Time Emulation and Visualization of IoT Data Using the WebSocket Protocol"

Student, Group 605: Mykyta Kaidanovych

Supervisor: Dr.Sc. (Techn.), Prof. ZHURAVSKA Iryna

The Master's thesis focuses on developing approaches to handle IoT data in soft real-time mode without the use of physical hardware. The rapid development of sensor networks makes physical testing of data processing architectures excessively expensive and complex. The implementation of emulation systems resolves this dilemma by providing developers with a tool for flexible scenario modeling and accurate reproduction of experimental conditions.

Object of the study – the process of data exchange and processing in IoT systems.

The subject of research is technological approaches to sensor network emulation, as well as mechanisms for real-time telemetry transport and visualization.

The aim of the thesis is to create a fully functional system that combines a data generator (virtual sensors) and a visual interface. The WebSocket protocol was chosen as the basis for communication, ensuring a continuous data stream between system components.

The practical significance of the work lies in the creation of a tool that solves the issue of hardware availability: it enables the testing of IoT scenarios and algorithms in a completely virtual environment. This makes the system indispensable for educational courses and laboratory assignments, while also allowing it to serve as a prototyping platform for the development of new industrial solutions.

The results of the work were presented at the XXVIII All-Ukrainian Scientific and Practical Conference "Mohylian Readings – 2025" (Mykolaiv, November 2025).

Main results of the thesis:

In Chapter 1, modern approaches to IoT data transmission were analyzed, along with challenges of real-time streaming and existing IoT emulation platforms. The WebSocket protocol was justified as an optimal solution for establishing low-latency bidirectional communication.

In Chapter 2, the system architecture was developed, including a server-side module for data generation, a client interface for visualization, and a storage subsystem for historical datasets. Logical and functional interaction schemes were constructed, and the JSON message structure was defined.

In Chapter 3, a software prototype was implemented using Node.js and the ws library for WebSocket communication. The client-side visualization was developed with Chart.js, supporting dynamic data updating. Data logging into SQLite was implemented, along with simulated network disturbances and noise variation mechanisms.

In Chapter 4, system testing was performed to assess latency, stability under multiple concurrent clients, and performance under increased load. The results confirmed that the system ensures real-time data delivery with minimal delay and can be scaled within laboratory and educational use cases.

The thesis consists of 73 pages, 7 tables, 19 figures, and 3 appendices. The research is based on 23 references.

Keywords: IoT, data emulation, WebSocket, real-time visualization, Node.js, Chart.js, sensor systems.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ 4
ВСТУП 5
1 АНАЛІЗ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАВДАННЯ РОЗРОБКИ СИСТЕМИ ЕМУЛЯЦІЇ ТА ВІЗУАЛІЗАЦІЇ IOT-ДАНИХ У РЕЖИМІ РЕАЛЬНОГО ЧАСУ 7
1.1 Актуальність теми 7
1.2 Аналіз сучасного стану проблеми 9
1.3 Огляд технологій і підходів 11
1.4 Аналіз існуючих аналогів систем емуляції IoT-даних 13
1.5 Характеристика IoT-даних та вимоги до емуляції 15
1.6 Специфікації вимог до системи 19
1.7 Постановка завдання дослідження 20
Висновки до розділу 1 22
2 ПРОЄКТУВАННЯ СИСТЕМИ ЕМУЛЯЦІЇ ТА ВІЗУАЛІЗАЦІЇ IOT-ДАНИХ 25
2.1 Загальна архітектура системи 25
2.1.1 Моделювання апаратних інтерфейсів та абстракція типів сенсорів 27
2.2 Механізм роботи WebSocket 28
2.3 Використані технології та середовище розробки 31
2.4 Логіка обробки даних 33
2.4.1 Генерація даних 33
2.4.2 Формування пакета 34
2.4.3 Передача 34
2.4.4 Обробка 34
2.4.5 Збереження 35
2.5 Запланована реалізація інтерфейсу візуалізації 37
2.6 Перспективи тестування та очікувані результати 38
Висновки до розділу 2 40
3 РЕАЛІЗАЦІЯ РОЗРОБЛЕНОЇ СИСТЕМИ 42

3.1 Обґрунтування вибору середовища розробки та структура проєкту 42
3.2 Реалізація серверного модуля та логіки емуляції 44
3.2.1 Алгоритмічна реалізація емуляції фізичних процесів 45
3.2.2 Структура даних та реалізація збереження історії 47
3.3 Особливості реалізації клієнтського інтерфейсу та візуалізації даних 49
Висновки до розділу 3 53
4 ТЕСТУВАННЯ ТА АНАЛІЗ ЕФЕКТИВНОСТІ РОЗРОБЛЕНОЇ СИСТЕМИ 55
4.1 Методика та результати функціонального тестування 55
4.2 Аналіз продуктивності та мережевих затримок 58
4.3 Перевірка стабільності та масштабованості системи 60
Висновки до розділу 4 62
ВИСНОВКИ 64
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 66
ДОДАТОК А Код програми 68
ДОДАТОК Б Матеріали апробації 93
ДОДАТОК В Перевірка на унікальність 95

ПЕРЕЛІК СКОРОЧЕНЬ

БД	– База даних
СКБД	– Система управління базами даних
API	– Application Programming Interface
BOM	– Byte Order Mark
DB	– Database (база даних)
FIFO	– First In, First Out
HAL	– Hardware Abstraction Layer
HTTP	– HyperText Transfer Protocol
IoT	– Internet of Things (Інтернет речей)
JSON	– JavaScript Object Notation
MQTT	– Message Queuing Telemetry Transport
R&D	– Research and Development
SCADA	– Supervisory Control and Data Acquisition
SPA	– Single Page Application

ВСТУП

З появою Інтернету речей грань між фізичними об'єктами та програмним кодом майже зникла. IoT-рішення стали стандартом для "розумних" міст, агропромисловості та виробництва. Проте інженери стикаються з "вузьким місцем" розробки – тестуванням. Побудувати фізичний полігон із сотнями датчиків для перевірки гіпотези часто неможливо технічно або невиправдано дорого. Саме тому індустрія переходить до використання віртуальних двійників (емуляторів), які дозволяють моделювати навантаження та поведінку мережі без витрат на апаратне забезпечення.

Разом з тим, створення таких екосистем неможливе без ретельної перевірки каналів передачі даних. Виконувати такі тести на реальному обладнанні – це ресурсомістке завдання, яке часто перетворюється на логістичний кошмар. Саме тому індустрія переходить до використання віртуальних аналогів сенсорів. Емуляція стає тим фундаментом, який дозволяє проєктувати та налагоджувати IoT-рішення без ризиків, пов'язаних з фізичним обладнанням.

Візуалізація отриманих даних у режимі реального часу має не менше значення. Вона дозволяє не лише відслідковувати параметри середовища, а й оперативно реагувати на відхилення. Для забезпечення миттєвого обміну інформацією між сервером-емулятором та клієнтом-візуалізатором найдоцільніше використовувати протокол WebSocket, що забезпечує постійне двонаправлене з’єднання між клієнтом і сервером без необхідності багаторазових HTTP-запитів.

З технічної точки зору, актуальність проєкту зумовлена потребою в ефективних інструментах емуляції, які працюють через WebSocket. Це дозволяє вирішити тривалу проблему високої вартості тестування та затримок передачі даних. Розроблена система стає гнучким інструментом, що закриває потреби як розробників, так і освітян.

Мета роботи фокусується на реалізації архітектурного рішення, яке поєднує генерацію синтетичних даних та їх миттєвий вивід на дашборд. Головна задача –

забезпечити безшовну інтеграцію віртуальних сенсорів із вебінтерфейсом у режимі soft real-time.

Об’єкт дослідження: процес обміну даними між сенсорними пристроями та системами візуалізації даних.

Предмет дослідження: методи емуляції, передавання та графічного відображення потоків IoT-даних.

Завдання дослідження:

1) виконати порівняльний аналіз технологій передачі даних (HTTP, MQTT, WebSocket), виявивши "вузькі місця" кожного протоколу в контексті вебзастосунків;

2) обґрунтувати вибір клієнт–серверної архітектури з двонаправленим з’єднанням через протокол WebSocket як оптимальної для систем реального часу та сформулювати вимоги до її реалізації;

3) спроєктувати універсальну модель даних, здатну генерувати валідне навантаження, імітуючи поведінку різнорідних фізичних сенсорів.

4) реалізувати базову систему взаємодії «сервер–клієнт» на платформі Node.js з використанням бібліотек *Express* і *ws* для підтримки WebSocket-з’єднань;

5) забезпечити механізми горизонтального масштабування та інтеграції з базами даних для збереження історії.

Прикладна цінність роботи полягає у створенні інструменту, що усуває бар'єр "заліза": система дозволяє моделювати навантаження від тисяч сенсорів та візуалізувати телеметрію без витрат на фізичне обладнання.

Апробація результатів дослідження.

Результати роботи планується апробувати на науково-практичній конференції «Могилянські читання – 2025», що відбулася на базі Чорноморського національного університету імені Петра Могили (листопад 2025 р.) [1].

1 АНАЛІЗ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАВДАННЯ РОЗРОБКИ СИСТЕМИ ЕМУЛЯЦІЇ ТА ВІЗУАЛІЗАЦІЇ IOT-ДАНИХ У РЕЖИМІ РЕАЛЬНОГО ЧАСУ

1.1 Актуальність теми

Сучасний технологічний ландшафт докорінно змінився завдяки Інтернету речей – парадигмі, що забезпечила безшовну інтеграцію фізичних об’єктів у цифровий простір. Його концепція ґрунтується на взаємодії величезної кількості пристроїв – сенсорів, контролерів, мікрокомп’ютерів, побутової техніки, транспортних засобів, медичного обладнання – що обмінюються даними між собою та з хмарними платформами. У результаті формується новий тип інфраструктури, де інформаційні потоки постійно циркулюють між об’єктами, користувачами та аналітичними системами. Технології IoT знаходять практичне застосування практично в усіх сферах людської діяльності:

1) у промисловості – для моніторингу стану обладнання та енергоефективності;
2) у сільському господарстві – для контролю мікроклімату, вологості ґрунту, освітлення [14];
3) у транспорті – для відстеження маршрутів і керування потоками;
4) у медицині – для моніторингу життєвих показників пацієнтів у реальному часі.

Гнучкість архітектури зробила Інтернет речей безальтернативною основою для побудови інтелектуальних середовищ. Сьогодні на цій базі функціонують як глобальні системи ("розумні" міста, підприємства), так і локальні приватні екосистеми [15].

Разом з тим, по мірі зростання кількості підключених пристроїв постає проблема складності розробки, налагодження та тестування подібних систем. Кожен реальний сенсор вимагає власного джерела живлення, мережевого інтерфейсу, протоколу зв’язку, налаштування безпеки та каналів передачі[16].

Тестування системи з десятками або сотнями сенсорів у лабораторних умовах стає практично неможливим без значних фінансових витрат. З огляду на це, наявність віртуального полігону стає обов'язковою передумовою успішної розробки. Таке середовище дозволяє не лише верифікувати коректність алгоритмів та моделей даних, а й піддати стрес-тестам канали зв'язку, точно відтворюючи динаміку фізичних пристроїв [17].

У цьому розрізі критичного значення набуває технологія імітаційного моделювання. Мова йде про створення програмних генераторів, які продукують синтетичний трафік, що статистично не відрізняється від реального. Це дозволяє інженерам "вийти за межі" фізичної лабораторії: перевіряти стійкість алгоритмів до пікових навантажень та мережевих колізій без ризику для обладнання. Можна стверджувати, що віртуалізація IoT-периметра стала фундаментом для побудови надійних високонавантажених систем.

Особливе значення в таких системах має реальний час обміну даними, коли затримка між генерацією інформації та її відображенням не перевищує кількох мілісекунд. Для досягнення цього ефекту використовується протокол WebSocket, який забезпечує постійне двонаправлене з’єднання між клієнтом і сервером. На відміну від традиційних HTTP-запитів, що виконуються періодично, WebSocket дозволяє серверу самостійно ініціювати передачу даних, щойно вони з’являються. Такий архітектурний підхід відкриває шлях до побудови реактивних дашбордів: показники сенсорів оновлюються на клієнті автоматично, нівелюючи потребу у постійному надсиланні запитів до сервера.

Інтеграція протоколу WebSocket у процеси емуляції створює синергетичний ефект, дозволяючи будувати високоточні моделі IoT-систем. Таке середовище стає ідеальним полігоном як для наукових пошуків, так і для навчання, адже воно нівелює ризики роботи з фізичним обладнанням. Крім очевидної економії бюджету, цей підхід надає інженеру тотальний контроль над умовами експерименту, що неможливо в реальних умовах.

У підсумку, запит індустрії на інструменти, здатні візуалізувати дані в режимі реального часу без затримок, визначає беззаперечну актуальність обраного напрямку досліджень у контексті глобальної цифровізації.

1.2 Аналіз сучасного стану проблеми

Ринок технологій Інтернету речей активно розвивається, охоплюючи дедалі більше секторів економіки та наукових досліджень. Сьогодні практично кожна галузь має власні специфічні вимоги до систем збору, обробки та візуалізації даних. У промисловості використовуються SCADA-рішення для контролю технологічних процесів і запобігання аваріям, у побутовій сфері поширені «розумні» пристрої, що дозволяють керувати освітленням, температурою чи безпекою житла, у науці застосовуються складні вимірювальні комплекси, які потребують точної синхронізації великої кількості сенсорів (рис. 1.1). В усіх цих випадках об’єднує одна спільна проблема – необхідність перевірки того, як система реагує на різноманітні сценарії: втрату пакетів даних, перевантаження мережі, пікові потоки, непередбачувані збої в роботі окремих модулів. Без імітації таких ситуацій розробка та вдосконалення систем IoT є практично неможливою [9].

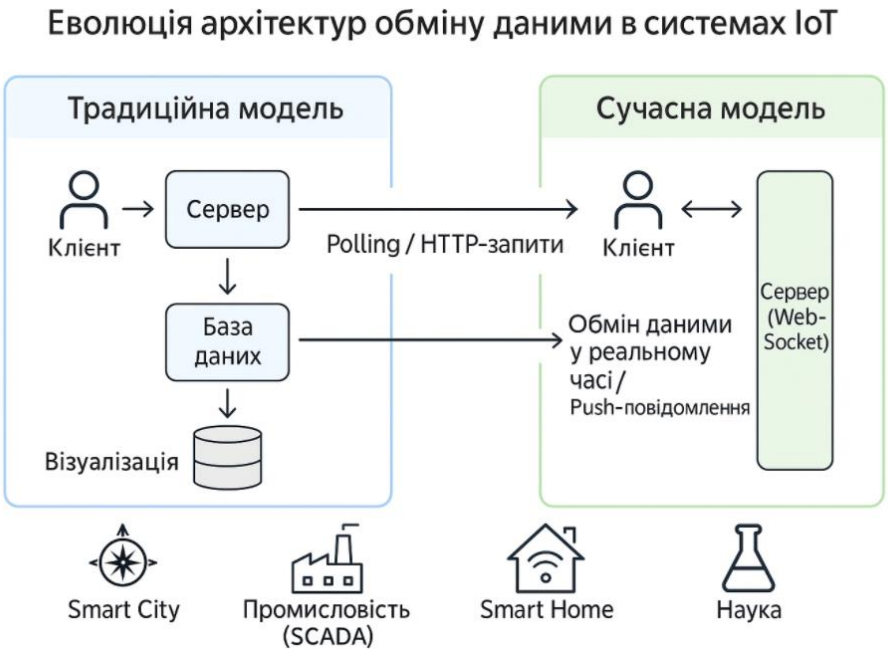


Рисунок 1.1 – Еволюція архітектур обміну даними в системах IoT

Більшість сучасних IoT-рішень ґрунтуються на класичній моделі «клієнт → сервер → база даних → інтерфейс відображення». Така схема зручна для централізованого керування, проте має обмеження при роботі в режимі реального часу. Передавання даних через періодичні HTTP-запити (polling) створює додаткове навантаження на канал і сервер, а також призводить до збільшення затримки оновлення інформації. При масштабуванні зростає потреба в буферизації вимірювань, відновленні пакетів при втраті сигналу та вирівнюванні часових рядів. Для систем реального часу ці фактори критичні, оскільки навіть затримка у 200–300 мс може спотворити динаміку сенсорних даних і унеможливити коректний аналіз [1; 4]. На сучасному етапі розвитку з’являються альтернативні підходи, спрямовані на усунення цих недоліків. Вибір на користь WebSocket продиктований його здатністю забезпечувати безшовний двонаправлений зв'язок. Технологія дозволяє серверу відправляти дані клієнту асинхронно (push-повідомлення), що усуває зайві накладні витрати на встановлення з'єднання. У результаті ми отримуємо систему з миттєвою реакцією на події та рівномірним розподілом навантаження. Це забезпечує мінімальні затримки передачі, рівномірне навантаження на мережу та більш природну взаємодію між компонентами системи. Саме тому технології WebSocket дедалі частіше використовуються в IoT-проєктах [7], що потребують оперативного оновлення даних: у системах розумного міста, транспортних трекінг-сервісах, медичних платформах для відстеження показників пацієнтів, аналітичних панелях підприємств.

Водночас більшість готових рішень, які сьогодні існують на ринку, залишаються закритими або комерційними продуктами. Це обмежує можливість їхнього використання в освітніх чи дослідницьких цілях, де потрібна гнучкість, відкритий вихідний код [5] і можливість адаптації під конкретне завдання. У таких умовах актуальним стає створення власних універсальних середовищ для емуляції передачі IoT-даних. Вони повинні бути достатньо простими для налаштування, але водночас гнучкими, щоб відтворювати різні режими роботи сенсорів, підтримувати

різні типи даних (температура, вологість, тиск, освітленість тощо) та працювати з різними клієнтськими інтерфейсами.

Додатковим бар'єром є фрагментація екосистеми IoT. Відсутність єдиного стандарту змушує розробляти адаптери для уніфікації даних від різних датчиків, що включає етапи фільтрації та верифікації. Перехід на універсальні формати типу JSON полегшує агрегацію, проте не скасовує необхідності в математичній обробці рядів: виявленні артефактів вимірювання, інтерполяції пропусків та часовій синхронізації

Отже, існує очевидний дисбаланс між складністю IoT-мереж та засобами їх діагностики. Створення системи, яка дозволяє синхронно емулювати роботу безлічі пристроїв та обробляти їх телеметрію без затримок, є відповіддю на цей виклик. Саме розробка такого уніфікованого механізму становить основну цінність роботи в контексті комп'ютерної інженерії.

1.3 Огляд технологій і підходів

У сучасній розробці систем реального часу використовується кілька підходів до організації передачі даних між клієнтом і сервером. Найбільш поширеними серед них є три основні моделі (табл. 1.1), кожна з яких має власні переваги, недоліки та сфери застосування.

Таблиця 1.1 – Порівняння технологій передавання даних у реальному часі

Технологія	Характеристика	Недоліки
HTTP Polling	Клієнт періодично запитує дані з сервера через HTTP	Високе навантаження на мережу, затримки
Long Polling	Сервер тримає запит відкритим, доки не з’являться нові дані	Неекономне використання ресурсів
WebSocket	Постійне двонаправлене з’єднання між клієнтом і сервером	Потрібна додаткова конфігурація безпеки

Порівняння підтверджує висновки, наведені в роботах [1; 4; 7]. Традиційні технології, такі як HTTP Polling або Long Polling, тривалий час залишалися базовими рішеннями для обміну даними між клієнтом і сервером. Проте зі збільшенням обсягів інформації та кількості підключень вони почали демонструвати суттєві недоліки. У моделі HTTP Polling клієнт з певним інтервалом надсилає запити до сервера, навіть якщо нових даних немає. Це призводить до зайвого навантаження на мережу, підвищеного споживання ресурсів і затримок у відображенні інформації. Long Polling частково вирішує цю проблему, оскільки сервер утримує запит відкритим до моменту появи нових даних, однак така модель не забезпечує повної асинхронності, потребує великої кількості відкритих з’єднань і створює додаткове навантаження на процесор.

На відміну від зазначених підходів, WebSocket забезпечує повноцінний двонаправлений канал зв’язку, який залишається відкритим протягом усього сеансу комунікації. Протокол WebSocket базується на TCP-з’єднанні, що дозволяє клієнту та серверу обмінюватися повідомленнями у форматі фреймів без необхідності повторного встановлення з’єднання. Завдяки цьому вдається досягти мінімальної затримки, стабільного обміну даними та значного зменшення навантаження на мережеві ресурси. Власне, завдяки цим характеристикам WebSocket стає безальтернативним вибором для чутливих до затримок доменів. Це стосується сценаріїв, де кожна мілісекунда має значення: від високочастотного трейдингу та multiplayer-ігор до систем життєзабезпечення в медицині та SCADA- систем на виробництві.

Технологія WebSocket визначена у специфікації RFC 6455, розробленій Internet Engineering Task Force (IETF). Вона підтримується більшістю сучасних браузерів і серверних платформ. Сутність її роботи полягає у встановленні початкового з’єднання через HTTP (так званий handshake), після чого канал переходить у постійний двонаправлений режим. Відтепер сервер перестає бути пасивним учасником, який лише "відбиває" запити, і отримує інструмент для активної розсилки інформації. Саме ця властивість перетворює звичайний вебканал

на повноцінну магістраль для миттєвого обміну даними, що є критично важливим для динамічних середовищ.

У межах цього дослідження обрано реалізацію на основі середовища Node.js, оскільки воно забезпечує неблокувальну модель введення-виведення (non-blocking I/O) [12] та асинхронне виконання коду. Node.js має розвинену екосистему бібліотек, яка спрощує створення масштабованих серверних застосунків. Для реалізації базових HTTP-запитів використовується бібліотека Express.js, що надає зручний інтерфейс маршрутизації, обробки запитів та конфігурації сервера. Для встановлення й підтримання WebSocket-з’єднань застосовується легка бібліотека *ws*, яка є однією з найпопулярніших серед розробників JavaScript-сервісів реального часу. Разом ці інструменти забезпечують стабільну, швидкодіючу й гнучку основу для побудови серверної частини системи.

З боку клієнта для візуалізації даних використовується бібліотека Chart.js, що дає можливість створювати інтерактивні графіки, діаграми та динамічні панелі моніторингу [11]. Вона працює на основі HTML5 Canvas, підтримує плавні оновлення без перезавантаження сторінки та дозволяє одночасно відображати кілька наборів даних. Chart.js відзначається простотою інтеграції з JavaScript-кодом і високим рівнем продуктивності навіть при великій кількості точок даних, що робить її оптимальним вибором для клієнтської частини системи емуляції IoT.

У сукупності вибір саме такого технологічного стеку – Node.js, Express.js, ws і Chart.js – забезпечує можливість побудови кросплатформної системи, яка не залежить від операційної системи користувача, легко масштабується, має відкритий код і підтримується широким співтовариством розробників [18]. Крім того, відкритість і безкоштовність зазначених інструментів дозволяють застосовувати їх у навчальних і дослідницьких цілях без додаткових витрат.

Таким чином, аналіз сучасних технологій показує, що поєднання асинхронної архітектури Node.js, ефективного двонаправленого протоколу WebSocket і гнучкої бібліотеки візуалізації Chart.js створює оптимальне середовище для розроблення

системи реального часу, яка здатна забезпечити швидке, стабільне та масштабоване передавання IoT-даних і їхнє наочне відображення.

1.4 Аналіз існуючих аналогів систем емуляції IoT-даних

Проблематика верифікації IoT-систем посідає одне з центральних місць у сучасній комп'ютерній інженерії. Хоча ринок насичений інструментами для моделювання роботи сенсорів, більшість із них страждає від "хвороби вузької спеціалізації" або має суттєві функціональні обмеження. Серед існуючих рішень помітно виділяється IoTIFY – хмарний сервіс, що дозволяє генерувати віртуальні пристрої у вебсередовищі, проте навіть він не є панацеєю [8]. Користувач може створювати сценарії поведінки сенсорів, задавати типи протоколів (MQTT, CoAP, HTTP) і візуалізувати потоки даних у реальному часі. Недоліком цієї системи є закритість вихідного коду та необхідність платної підписки для повнофункціональної роботи. Аналогічні функції реалізовано у ThingsBoard Simulator, модулі однойменної платформи, який дозволяє надсилати тестові повідомлення через MQTT або HTTP і має інтегровану панель моніторингу [5]. Однак ця система не підтримує WebSocket-з’єднання, що обмежує її використання для вебвізуалізації у реальному часі.

Іншим відомим рішенням є MATLAB IoT Toolbox, який має засоби моделювання даних від віртуальних пристроїв і гнучкі аналітичні можливості. Проте цей інструмент є закритим, потребує ліцензії та не підходить для легкого розгортання у вебсередовищі [6]. Для навчальних і дослідницьких цілей широко застосовуються Simulink IoT Emulator, Node-RED Dashboard із вбудованими блоками inject і debug, а також Kaa IoT Platform, що дозволяють створювати тестові пристрої з мінімальним налаштуванням. Попри це, усі перелічені рішення мають спільний недолік – складність масштабування та відсутність універсального підходу до генерації різнорідних даних із можливістю гнучкої візуалізації.

Таблиця 1.2 – Порівняльна характеристика існуючих емуляторів IoT-даних

Платформа	Тип середовища	Переваги	Недоліки
IoTIFY	Хмарна вебплатформа	Просте створення сценаріїв; підтримка MQTT, HTTP, CoAP; візуалізація потоків у реальному часі	Платна підписка; закритий код; залежність від сервісу
ThingsBoard Simulator	Модуль у складі ThingsBoard	Інтеграція з аналітичними панелями; підтримка MQTT	Відсутність WebSocket; складність локального розгортання
MATLAB IoT Toolbox	Комерційне середовище моделювання	Потужні аналітичні засоби; точність генерації даних	Ліцензійна програма; закрита архітектура
Node-RED Dashboard	Відкрите середовище	Простота інтеграції; підтримка локального тестування; велика спільнота	Обмежені можливості масштабування; відсутність складної емуляції
Kaa IoT Platform	Відкрита серверна платформа	Гнучке налаштування; підтримка різних типів пристроїв	Потребує значних ресурсів; складна інсталяція
Simulink IoT Emulator	Модуль MATLAB Simulink	Детальне моделювання фізичних процесів	Висока складність; залежність від MATLAB

Порівняльний аналіз показує, що більшість існуючих аналогів зосереджені або на моделюванні протоколів (як MQTT-брокери-тестери), або на побудові хмарних демонстраційних середовищ. При цьому бракує відкритих і простих у використанні систем, які б дозволяли одночасно емулювати дані від численних сенсорів і візуалізувати їх через WebSocket у браузері без додаткових серверів чи брокерів. Отже, розробка власного рішення з відкритою архітектурою, реалізованого на базі технологій Node.js, WebSocket і Chart.js, є доцільною й має потенціал заповнити нішу між складними комерційними емуляторами та обмеженими навчальними інструментами [19].

1.5 Характеристика IoT-даних та вимоги до емуляції

Системи Інтернету речей характеризуються надзвичайно різноманітними типами даних, що залежать від призначення сенсорів, середовища їхнього використання та особливостей конкретного завдання. До найпоширеніших типів належать температурні показники [21], рівень вологості, атмосферний тиск, освітленість, рівень шуму, споживання електроенергії, положення у просторі (GPS-координати), показники вібрацій, швидкість руху або концентрація газів у повітрі. У промислових застосуваннях додатково збираються дані про стан машин і механізмів, а в розумних містах – про транспортні потоки, якість повітря, рівень заповненості контейнерів зі сміттям тощо. Кожен тип даних має власні одиниці вимірювання, діапазони, точність і частоту оновлення.

Багато IoT-пристроїв працюють із частотою вимірювання від 1 до кількох сотень вимірів за секунду. Це означає, що навіть у невеликий проміжок часу утворюється величезний потік інформації, який потрібно передавати, обробляти й зберігати без втрат. У таких умовах основними викликами стають стабільність каналу зв’язку, оптимізація передачі та гарантована доставка повідомлень. Особливо критично це для систем, що функціонують у реальному часі – наприклад, у дистанційному моніторингу виробничих процесів, де затримка навіть у кілька секунд може призвести до помилкових рішень або втрати важливих показників [2].

При проєктуванні моделі даних стало очевидно, що передавати "голі" значення недостатньо. Критично важливим є метадані: унікальний ідентифікатор вузла, точний час події (timestamp), розмірність величини та допустимі межі похибки. Це формує вимогу до повідомлення: воно має бути атомарним і самодостатнім. Вибір зупинився на JSON – фактичному стандарті індустрії. Його головний козир – це гнучкість схеми, що дозволяє "безболісно" додавати нові ключі без порушення зворотної сумісності з існуючими модулями системи.

Приклад структури повідомлення від емулятора наведено на рис. 1.2.

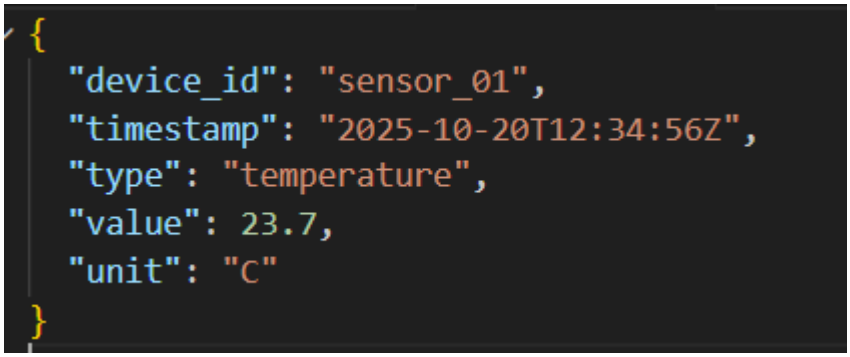


Рисунок 1.2 – Приклад структури JSON-повідомлення від емулятора IoT-даних

Оскільки дані надходять безперервним потоком, головним викликом стає збереження їхньої хронологічної структури. Різна частота опитування сенсорів та нестабільність каналу можуть призвести до спотворення картини вимірювань. Вирішенням проблеми є впровадження алгоритмів буферизації та нормалізації: вони дозволяють "склеїти" розрізнені пакети в єдиний логічний ланцюжок, відсіявши при цьому випадкові викиди значень. Це є фундаментом для коректної роботи підсистеми візуалізації.

Наведена структура є універсальною моделлю, яка може бути розширена додатковими полями залежно від типу пристрою або специфіки задачі. Наприклад, до неї можна додати параметри калібрування, стан батареї, рівень сигналу Wi-Fi, ідентифікатор геопозиції або службову інформацію для синхронізації з іншими вузлами мережі. Такий формат даних не лише забезпечує сумісність між різними платформами, а й значно спрощує подальший аналіз і візуалізацію, оскільки JSON легко інтегрується у більшість сучасних мов програмування (Python, JavaScript, C#, Java тощо).

Завдяки стандартизації структури пакетів ми досягаємо безшовної розширюваності: система автоматично адаптується до зростання навантаження, обробляючи вхідний потік як уніфікований набір даних. Це також відкриває широкі можливості для інтеграції. Зовнішні сервіси, такі як хмарні аналітичні платформи або документ-орієнтовані бази даних, можуть споживати такий JSON, що значно спрощує побудову інфраструктури зберігання.

Невід'ємною складовою якісної емуляції є імплементація стохастичних моделей похибок. Ідеальних сенсорів не існує: на практиці ми завжди маємо справу з дрейфом показників, електромагнітними завадами та втратами пакетів. Тому генератор даних повинен підмішувати до «чистого» сигналу синтетичний шум та імітувати зовнішні збурення. Це єдина можливість перевірити, чи здатні алгоритми фільтрації та механізми ретрансмісії встояти перед хаосом реальної експлуатації.

Функціонал системи дозволяє вийти за межі штатного режиму роботи та імітувати складні патерни поведінки. Мова йде про стрес-тестування: симуляцію масового відключення вузлів або high-load навантаження на канал зв'язку. Аналіз реакції сервера на такі події є ключем до оцінки його стабільності та визначення часу, необхідного для автоматичного відновлення.

Для досягнення максимальної автентичності симуляції в систему закладено алгоритми генерації перешкод: від накладання білого шуму на корисний сигнал до імітації "мережевого шторму" або повної втрати пакетів. Варіативність інтервалів передачі дозволяє відтворити нестабільність реальних каналів зв'язку. Такий підхід забезпечує глибоке стрес-тестування архітектури, дозволяючи локалізувати та усунути "вузькі місця" ще на фазі прототипування, задовго до інтеграції з фізичним обладнанням.

Фундаментальна цінність емулятора полягає в його адаптивності. Інструмент надає можливість маніпулювати ключовими змінними середовища: задавати щільність сенсорів, рівень зашумленості каналу та інтенсивність трафіку. Це перетворює його на універсальний полігон, де можна відтворити будь-який сценарій – як стабільну роботу "розумного будинку", так і критичні перевантаження промислової мережі.

До основних вимог до емуляції IoT-даних належать:

- можливість задавати кількість сенсорів і частоту надсилання даних;
- підтримка різних типів сигналів (температурні, аналогові, дискретні);
- відтворення випадкових збоїв та втрат пакетів;

- формування достовірних часових рядів із реалістичними варіаціями;
- підтримка швидкого оновлення даних у реальному часі;
- простота масштабування без змін основного коду.

Фактично, емулятор виступає універсальним бенчмарком для IoT-систем. Він забезпечує замкнений цикл тестування, дозволяючи перевірити ефективність коду та надійність каналів зв'язку. Це рішення радикально знижує поріг входження в розробку складних систем, оскільки дозволяє моделювати high-load навантаження без інвестицій у фізичну інфраструктуру.

Інтеграція такого інструментарію в pipeline розробки дозволяє кардинально оптимізувати таймінг проєкту. Замість тривалого налагодження на фізичних стендах, команда отримує швидкий зворотний зв'язок у віртуальному середовищі. Це створює фундаментальну базу для розгортання складних алгоритмів керування, здатних працювати у жорсткому реальному часі.

1.6 Специфікації вимог до системи

Ключем до успішної реалізації платформи стало визначення жорстких технічних нормативів. Сукупність функціональних та якісних вимог виступає своєрідним "каркасом" системи. Їх виконання є обов'язковою умовою для забезпечення надійної роботи WebSocket-з'єднань та коректної візуалізації потоків без лагів і збоїв.

Оскільки фундаментом IoT є робота з безперервними потоками телеметрії, архітектура системи повинна бути оптимізована під специфіку стримінгу. Дані сенсорів мають часову прив’язку, можуть надходити з різною частотою та із затримками, що потребує механізмів синхронізації часових рядів, виявлення пропусків і фільтрації аномальних значень. Для забезпечення цілісності інформації система повинна підтримувати контроль структури повідомлень та гарантувати їх доставку без дублювання. Окрему увагу необхідно приділити вирівнюванню

потоків при надходженні даних від декількох сенсорів, а також мінімізації затримок, що є критичним у режимі реального часу.

Функціональні вимоги:

- система має забезпечувати генерацію синтетичних даних із заданими параметрами (кількість сенсорів, типи сигналів, частота оновлення, рівень шуму);
- передавання даних здійснюється через двонаправлений канал WebSocket із підтримкою кількох клієнтів одночасно;
- клієнтська частина повинна оновлювати графічні дані без перезавантаження сторінки;
- має бути реалізована можливість збереження історичних даних у локальній базі SQLite;
- користувач повинен мати змогу змінювати параметри емуляції під час роботи системи.

Нефункціональні вимоги:

1) **продуктивність.** Система повинна забезпечувати обробку не менше 100 повідомлень на секунду без втрати даних при середньому навантаженні;

2) **надійність.** Передбачено автоматичну перевірку активності з’єднання та відновлення сеансу після втрати зв’язку;

3) **безпека.** Передача даних здійснюється через захищений канал *wss*:// із використанням протоколу TLS; реалізовано базову автентифікацію клієнтів;

4) **масштабованість.** Архітектура має підтримувати підключення не менше 50 клієнтів одночасно без деградації швидкодії;

5) **зручність використання.** Інтерфейс користувача повинен бути інтуїтивно зрозумілим, із можливістю візуального спостереження за зміною параметрів у реальному часі;

6) **сумісність.** Система має бути кросплатформною, придатною для розгортання в операційних системах Windows, Linux і macOS;

7) **розширюваність.** Передбачено можливість інтеграції з базами даних, брокерами повідомлень (MQTT, Kafka) або сторонніми API;

8) **підтримуваність.** Код системи має бути структурованим, документованим і придатним для подальшої модифікації.

Резюмуючи, сформований пул вимог виконує роль технічної конституції проєкту. Він не лише фіксує метрики якості, а й слугує дорожньою картою для всіх подальших етапів: від архітектурного проєктування до фінальної валідації продукту [3].

1.7 Постановка завдання дослідження

Підсумовуючи огляд існуючих підходів, стає можливим чітко окреслити вектор подальшої роботи. Виявлені технічні обмеження та стандарти галузі диктують постановку мети дослідження та визначають спектр завдань, які необхідно вирішити в рамках розробки.

Ключовий технічний виклик полягає в усуненні залежності від апаратного рівня. Дослідження поведінки IoT-мереж на фізичному обладнанні часто впирається у фінансові та логістичні бар'єри, особливо коли йдеться про масштабні інсталяції. Вирішенням є перехід у віртуальну площину: розробка програмного комплексу, здатного генерувати синтетичний трафік "на льоту" та візуалізувати його через вебінтерфейси.

Основною ціллю є розробка програмної архітектури для наскрізної обробки IoT-даних: від їх синтетичної генерації до виводу на графіки. Використання технології WebSocket дозволяє вирішити цю задачу в режимі real-time, усуваючи обмеження класичних підходів до передачі даних.

Такий функціонал відкриває доступ до комплексної діагностики. Інженер може в реальному часі відстежувати маршрут даних, виявляти "вузькі місця" у стабільності системи та валідувати поведінку клієнтських інтерфейсів при можливій деградації якості зв'язку.

Для досягнення поставленої мети необхідно розв’язати такі основні завдання дослідження:

1) провести аналітичний огляд існуючих технологій для реалізації передачі даних у реальному часі у вебзастосунках (HTTP, MQTT, WebSocket) і визначити їхні переваги та недоліки;

2) обґрунтувати вибір архітектури клієнт-сервер із двонаправленим з’єднанням через протокол WebSocket як оптимальної для системи реального часу та сформулювати специфікації вимог;

3) розробити структуру повідомлень і модель генерації даних, що забезпечуватиме реалістичну емуляцію роботи сенсорів різного типу;

4) реалізувати базову систему взаємодії сервер-клієнт на платформі Node.js із використанням бібліотек *Express* (для обробки HTTP-запитів) та *ws* (для підтримки WebSocket-з’єднань);

5) забезпечити безперервне оновлення даних на клієнтській стороні без перезавантаження сторінки, з використанням JavaScript і бібліотеки Chart.js для візуалізації;

6) створити можливість гнучкого масштабування системи, щоб вона могла підтримувати одночасну роботу великої кількості емульйованих сенсорів, а також передбачити можливість інтеграції з іншими рішеннями – базами даних, брокерами повідомлень (наприклад, MQTT або Kafka);

7) провести тестування продуктивності, стабільності роботи та коректності відображення даних у різних режимах навантаження;

Поставлені завдання мають комплексний характер і охоплюють усі етапи життєвого циклу програмного рішення – від аналізу вимог до реалізації й тестування. У процесі виконання роботи планується реалізувати прототип платформи, здатної моделювати роботу десятків або сотень сенсорів, генерувати їхні дані з урахуванням шумів і часових зсувів, передавати інформацію в реальному часі через WebSocket-з’єднання та відображати її у вигляді інтерактивних графіків і таблиць.

Підсумком дослідження є готовий до впровадження програмний модуль. Його функціонал виходить за межі простого емулятора: це комплексний стенд для валідації серверних архітектур та навчання студентів принципам IoT-взаємодії. Ключова перевага розробки – абстрагування від фізичної природи сенсорів, що робить систему легко розширюваною та адаптивною до будь-яких прикладних сценаріїв.

Висновки до розділу 1

У першому розділі здійснено теоретичний аналіз проблеми створення системи емуляції та візуалізації IoT-даних у режимі реального часу. Розглянуто сучасний стан розвитку технологій Інтернету речей, охарактеризовано ключові тенденції, проблеми та виклики, що виникають у процесі розроблення таких систем. Визначено, що зростання кількості пристроїв, підключених до мережі, потребує нових підходів до тестування, налагодження та дослідження поведінки розподілених систем без використання фізичного обладнання. Звідси випливає нагальна потреба у створенні синтетичних середовищ. Такі інструменти повинні не просто генерувати рандомні числа, а відтворювати достовірні потоки телеметрії, зберігаючи часову структуру передачі даних "в жилу

Проаналізовано основні архітектурні моделі обміну інформацією між клієнтом і сервером у середовищі IoT, серед яких найбільш перспективною визнано архітектуру з використанням протоколу WebSocket, що забезпечує двонаправлений обмін даними та мінімальні затримки передачі. У результаті порівняння традиційних підходів – HTTP Polling і Long Polling – встановлено, що вони не відповідають сучасним вимогам до швидкодії, масштабованості та ефективного використання ресурсів. Натомість WebSocket дозволяє реалізувати постійне з’єднання між клієнтом і сервером, що є необхідною умовою для роботи систем моніторингу в реальному часі.

У ході дослідження виконано огляд інструментів і технологій, придатних для реалізації поставленої задачі. Обґрунтовано вибір Node.js як базового середовища

виконання, Express.js – як фреймворку для обробки HTTP-запитів, *ws* – для підтримки WebSocket-з’єднань, та Chart.js – для клієнтської візуалізації даних. Показано, що зазначене поєднання технологій забезпечує оптимальний баланс між продуктивністю, простотою реалізації та можливістю масштабування системи.

В роботі детально розібрано природу телеметрії: її залежність від контексту та високу швидкість генерації. Це стало підґрунтям для вибору формату даних. Наведено приклад універсального JSON-об'єкта, який виступає "транспортом" для показників емулятора. Доведено, що використання JSON є найбільш раціональним рішенням для вебсередовищ завдяки його нативній підтримці та архітектурній гнучкості.

У заключній частині розділу сформульовано мету дослідження – створення архітектури системи, що дозволяє емулювати та візуалізувати потоки IoT-даних у реальному часі, а також визначено основні завдання, необхідні для її досягнення. Ці завдання охоплюють етапи від аналітичного огляду технологій до практичної реалізації, тестування й оцінювання ефективності запропонованого рішення.

Таким чином, результати першого розділу створюють науково-методичне підґрунтя для подальшої роботи. У наступному розділі буде розглянуто проєктування архітектури системи, принципи реалізації серверної та клієнтської частин, а також механізми взаємодії між ними через WebSocket-з’єднання, що дозволить перейти від теоретичних засад до практичної реалізації запропонованої системи.

2 ПРОЄКТУВАННЯ СИСТЕМИ ЕМУЛЯЦІЇ ТА ВІЗУАЛІЗАЦІЇ ІОТ-ДАНИХ

2.1 Загальна архітектура системи

Розроблена система побудована за класичною клієнт–серверною архітектурою (рис. 2.1), у якій сервер виконує функції генератора даних і посередника між віртуальними сенсорами та клієнтами, а клієнтська частина відповідає за відображення інформації в реальному часі. Такий підхід забезпечує логічне розділення функціональності, спрощує масштабування й дає можливість легко модифікувати окремі компоненти без впливу на інші частини системи.

Основні компоненти архітектури:

1) сервер-емулятор (*Node.js, Express, ws*) – відповідає за генерацію синтетичних даних, формування JSON-повідомлень і передачу їх клієнтам через WebSocket-з’єднання. Він виконує роль центрального вузла, який підтримує одночасну комунікацію з кількома клієнтами, розподіляє потоки даних і забезпечує синхронність передавання. Сервер може працювати у двох режимах – автоматичному (з фіксованими параметрами емуляції) та ручному (з налаштовуваними параметрами, які задає користувач через інтерфейс);

2) клієнт-візуалізатор (*HTML, JavaScript, Chart.js*) – приймає потоки даних через WebSocket і динамічно оновлює графіки без перезавантаження сторінки. Його основна функція полягає у відображенні актуальної інформації про стан сенсорів, а також у наданні користувачеві інструментів для моніторингу, порівняння й аналізу. Клієнт реалізує інтерактивну панель (dashboard), на якій у реальному часі відображаються графіки температури, вологості, тиску, освітленості тощо. Chart.js використовується завдяки своїй гнучкості та підтримці оновлення даних «на льоту»;

3) сховище даних **(**SQLite**)** – застосовується для збереження історії переданих повідомлень, що дозволяє виконувати подальший аналіз, будувати статистику або проводити ретроспективне тестування. SQLite обрано завдяки її

легкості, відсутності необхідності у встановленні серверного процесу та можливості інтеграції безпосередньо у Node.js-застосунок. Такий підхід дозволяє зберігати дані у локальному середовищі без додаткових витрат на конфігурацію;

4) гнучкість системи забезпечується блоком параметризації. Користувач може динамічно змінювати характеристики потоку: додавати шумові компоненти, регулювати частоту оновлення або імітувати збої сенсорів. Це відкриває можливості для проведення повноцінних стрес-тестів: створення сценаріїв граничного навантаження для оцінки пропускної здатності та стабільності серверного бекенду.

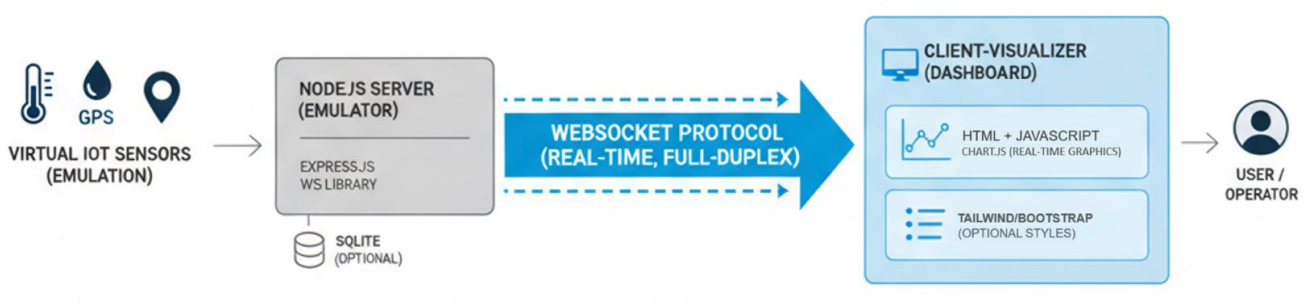


Рисунок 2.1 – Структурна схема системи емуляції та візуалізації IoT-даних

Архітектура системи передбачає двонаправлений обмін даними між сервером і клієнтом. Сервер періодично генерує набори даних, які мають структуру JSON-об’єктів, що містять ідентифікатор пристрою, тип сенсора, часову мітку, значення та одиницю вимірювання. Клієнт приймає ці повідомлення, обробляє їх за допомогою JavaScript-функцій та миттєво відображає на графіках. За рахунок використання протоколу WebSocket відпадає потреба в періодичних запитах до сервера – дані передаються асинхронно, у момент їх появи.

Для повноцінної імітації промислових умов реалізовано механізм обробки множинних підключень. Архітектура дозволяє гнучко маршрутизувати дані: від індивідуальної доставки кожному клієнту до спільного доступу. Такий підхід перетворює систему на ефективний інструмент для тестування масштабованості,

даючи змогу емпірично визначити, як зростання кількості споживачів впливає на затримки та стабільність каналу.

Закладена в основу архітектурна розширюваність гарантує життєздатність проєкту. Модульна структура дозволяє легко інтегрувати нові компоненти – від специфічних сенсорів до різноманітних бекендів (PostgreSQL, MongoDB). Завдяки цьому система стає "пластичною": її можна однаково ефективно використовувати як складний дослідницький полігон або як наочний стенд для студентських лабораторій.

Концептуально система являє собою гнучкий каркас. Вона поєднує високу швидкодію клієнт-серверної взаємодії з простотою масштабування. Це рішення створювалося з прицілом на майбутнє: воно легко адаптується під нові виклики, будь то розширення алгоритмічної бази чи інтеграція machine learning інструментів для глибокого аналізу телеметрії.

2.1.1 Моделювання апаратних інтерфейсів та абстракція типів сенсорів

Однією з ключових задач при проєктуванні системи емуляції для потреб комп’ютерної інженерії є врахування апаратних особливостей сенсорів. У реальних вбудованих системах зміна типу датчика тягне за собою зміну протоколів низького рівня (I2C, SPI, UART, OneWire) та форматів даних. Розроблена система реалізує рівень абстракції (HAL), який дозволяє моделювати поведінку різних класів пристроїв без зміни архітектури ядра.

При зміні типу датчика в системі відбуваються наступні адаптивні зміни:

1) **зміна розрядності та квантування.** Для аналогових датчиків (наприклад, терморезисторів NTC або датчиків освітленості LDR) емулюється робота АЦП (аналого-цифрового перетворювача). Система генерує «сирі» значення у діапазоні 0–1023 (для 10-бітного АЦП) або 0–4095 (для 12-бітного), які потім програмно перераховуються у фізичні величини;

2) **часові характеристики опитування.** Цифрові датчики (наприклад, DS18B20 або DHT22) мають фіксований час конвертації (від 750 мс до 2 с). Система

емуляції враховує ці затримки, блокуючи оновлення даних частіше, ніж це дозволяє фізика реального кристала;

3) **протокольна специфіка.** При імітації роботи сенсорів на шині I2C (наприклад, BMP280) система моделює пакетну передачу даних (адреса пристрою + регістр даних), що дозволяє у майбутньому замінити віртуальний модуль на реальний мікроконтролерний шлюз (на базі ESP32 або STM32), який транслюватиме дані у WebSocket.

Таким чином, система не просто генерує випадкові числа, а відтворює метрологічні характеристики реальних апаратних засобів, що відповідає вимогам до інженерних систем збору даних.

2.2 Механізм роботи WebSocket

На відміну від класичного HTTP-запиту, який ініціюється виключно клієнтом і завершується після отримання відповіді, протокол WebSocket створює постійне двонаправлене TCP-з’єднання, у межах якого обмін даними може відбуватися в обох напрямках. Це означає, що сервер має можливість самостійно надсилати повідомлення клієнту у будь-який момент часу без необхідності очікування запиту (табл. 2.1). Такий механізм дозволяє досягти мінімальної затримки між моментом генерації події та її відображенням у клієнтському інтерфейсі, що є критичним для систем реального часу.

Таблиця 2.1 – Порівняння WebSocket і HTTP:

Параметр	HTTP	WebSocket
Тип з’єднання	Клієнт надсилає запит → сервер відповідає	Постійне двонаправлене з’єднання
Затримка передачі	Висока (через періодичні запити)	Низька (постійний канал)
Навантаження на сервер	Високе при багатьох клієнтах	Низьке, ефективна передача

Параметр	HTTP	WebSocket
Формат даних	Текстові HTTP-заголовки + тіло	Легковагові фрейми даних
Використання у реальному часі	Обмежене	Оптимальне

Принцип роботи WebSocket складається з двох основних етапів: встановлення з’єднання (handshake) та обміну даними (data frames):

1) етап handshake. Клієнт надсилає спеціальний HTTP-запит на сервер, який містить заголовок Upgrade: websocket. Це сигналізує серверу про намір перейти з HTTP-протоколу на WebSocket. Якщо сервер підтримує цей протокол, він відповідає кодом 101 Switching Protocols, після чого з’єднання стає постійним;

2) етап передачі даних. Після встановлення з’єднання сторони обмінюються невеликими бінарними або текстовими фреймами, що містять лише необхідну інформацію без надлишкових заголовків. Це суттєво зменшує навантаження на канал і забезпечує високу швидкість реакції системи.

Використання WebSocket дозволяє відмовитися від традиційного механізму періодичних запитів (polling), який створює надлишковий трафік і навантаження на сервер. Замість цього система підтримує активне з’єднання, через яке клієнт і сервер можуть миттєво обмінюватися повідомленнями. У контексті розробленої системи це означає, що сервер надсилає дані з емуйованих сенсорів безпосередньо у браузер користувача одразу після їх генерації, а клієнт миттєво оновлює графіки на екрані.

Для реалізації WebSocket-з’єднань у роботі використано бібліотеку *ws* (WebSocket API for Node.js). Це легкий, швидкий і стандартизований інструмент, який дозволяє створювати серверну частину без складних налаштувань. З її допомогою можна:

- ініціювати WebSocket-сервер у середовищі Node.js;

- приймати та обробляти підключення від кількох клієнтів одночасно;
- надсилати дані окремим клієнтам або транслювати одне повідомлення всім підключеним користувачам;
- обробляти події розриву з'єднання, помилок чи повторного підключення.

Приклад створення простого WebSocket-сервера на Node.js із використанням бібліотеки *ws* наведено на рис. 2.1.

```
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 8080 });

wss.on('connection', ws => {
  console.log('Новий клієнт підключився');
  ws.on('message', message => console.log(`Отримано: ${message}`));
  setInterval(() => {
    const data = { sensor: 'temp', value: (20 + Math.random() * 5).toFixed(1) };
    ws.send(JSON.stringify(data));
  }, 1000);
});
```

Рисунок 2.1 – Приклад реалізації WebSocket-сервера на платформі Node.js

Цей приклад демонструє базовий принцип роботи: після підключення клієнта сервер щосекунди надсилає JSON-повідомлення з випадковими значеннями сенсора. Клієнтська частина, отримуючи ці дані, може негайно відобразити їх у вигляді графіків, таблиць або показників у реальному часі.

У реальній системі емуляції IoT-даних використовується аналогічний принцип, але з розширеною логікою – передбачено підтримку декількох типів сенсорів, регулювання частоти передачі, генерацію шуму та збереження історії даних у базі SQLite. Таким чином, WebSocket виступає основним комунікаційним каналом, який забезпечує синхронність і миттєву взаємодію між сервером і клієнтом.

Результатом обраної архітектури є швидка та відмовостійка система, готова до масштабування. Протокол WebSocket ефективно вирішує проблему затримок,

забезпечуючи миттєвий відгук інтерфейсу, без якого неможливо уявити повноцінну роботу "в прямому ефірі".

2.3 Використані технології та середовище розробки

Для реалізації системи було обрано сучасні кросплатформні технології, які забезпечують високу продуктивність, простоту масштабування та стабільну роботу в режимі реального часу (табл. 2.2).

Таблиця 2.2 – Використані технології та їх призначення у системі

Компонент	Технологія	Призначення
Сервер	Node.js + Express + ws	Генерація, обробка та передача даних
Клієнт	HTML + JavaScript + Chart.js	Відображення показників у реальному часі
База даних	SQLite	Збереження історії даних
Безпека	TLS (wss://)	Шифрування каналу зв’язку

Основу серверної частини складає Node.js – середовище виконання JavaScript, побудоване на рушії V8. Головна перевага Node.js полягає в асинхронній, неблокувальній моделі обробки подій, що дозволяє ефективно обслуговувати сотні одночасних з’єднань без суттєвого навантаження на процесор. Це робить його ідеальним рішенням для застосунків, які працюють у режимі реального часу, зокрема систем моніторингу, аналітичних панелей та IoT-платформ. Крім того, Node.js спирається на потужну екосистему npm. Це дає можливість миттєво розширювати можливості системи, просто підтягуючи необхідні залежності. Інструменти для автентифікації, валідації даних або моніторингу інтегруються за лічені хвилини, що робить процес масштабування системи гнучким та ефективним.

Для маршрутизації запитів, обробки HTTP-з’єднань та створення базової інфраструктури сервера використано фреймворк Express.js. Він забезпечує зручний механізм організації маршрутів, управління запитами та розширення

функціональності через middleware-компоненти. У системі Express виконує допоміжну роль – обробляє службові HTTP-запити (наприклад, налаштування або статистику), тоді як обмін даними між сервером і клієнтом здійснюється через WebSocket.

Передавання даних у реальному часі реалізовано за допомогою бібліотеки *ws* (WebSocket API for Node.js). Вона забезпечує створення легких WebSocket-серверів і дозволяє підтримувати активне двонаправлене з’єднання між клієнтом і сервером. Завдяки цій бібліотеці сервер може відправляти повідомлення відразу після генерації нових даних, а клієнти – миттєво їх отримувати без повторних запитів. Такий підхід мінімізує затримку оновлення та забезпечує відображення показників практично у реальному часі.

Клієнтська частина системи реалізована за допомогою стандартних вебтехнологій – HTML, CSS та JavaScript.

Для реалізації фронтенду використано стандартні веб-технології: HTML для побудови структури контенту та CSS для налаштування UI/UX. Головне ж навантаження лягає на JavaScript, який виступає сполучною ланкою між користувачем і сервером, відповідаючи за логіку обміну повідомленнями та інтерактивну візуалізацію.

Для візуалізації телеметрії інтегровано рішення на базі Chart.js. Ця бібліотека дозволяє перетворити сухий потік цифр на інтуїтивно зрозумілі інтерактивні діаграми. Ключові переваги – це автоматична адаптація під розмір екрана та механізм динамічного оновлення. Оператору не потрібно оновлювати сторінку: криві на графіку змінюються синхронно з надходженням нових пакетів від сенсорів, забезпечуючи безшовний досвід моніторингу.

Для зберігання даних застосовано легковагову вбудовану базу SQLite, що не потребує налаштування серверного процесу. Її особливістю є можливість зберігати всю базу даних у одному файлі, що значно спрощує розгортання системи та забезпечує швидкий доступ до історичних даних. SQLite підходить для тестових

і лабораторних проєктів, а також для невеликих серверів, де потрібна висока швидкодія та простота інтеграції.

Окрему увагу приділено питанню безпеки передавання даних. Для шифрування каналів зв’язку використовується протокол TLS (Transport Layer Security), який реалізує захищене з’єднання формату *wss*://. Це дозволяє запобігти перехопленню чи зміні переданих даних під час комунікації між сервером і клієнтами. Крім функціональності, критичну роль відіграє захист. Сервер підтримує авторизований доступ, що гарантує підключення лише довірених клієнтів.

Середовище розробки базується на відкритих інструментах:

1) Visual Studio Code як основний редактор коду, що забезпечує підтримку синтаксису JavaScript і JSON, інтеграцію з Git і налагодження в реальному часі;

2) npm (Node Package Manager) – для встановлення необхідних бібліотек і керування залежностями;

3) Google Chrome – як середовище тестування клієнтської частини з використанням інструментів розробника (DevTools).

Усі компоненти системи є кросплатформними та безкоштовними, що забезпечує проєкту повну технологічну незалежність. Широка підтримка розробників означає, що система не стане "тупиковою гілкою": вона відкрита для необмеженого вдосконалення, масштабування та інтеграції з іншими програмними продуктами без суттєвих обмежень.

Таким чином, поєднання Node.js, Express.js, ws, Chart.js і SQLite створює потужну, надійну та універсальну основу для реалізації системи емуляції та візуалізації IoT-даних, що поєднує ефективність, простоту й гнучкість у налаштуванні.

2.4 Логіка обробки даних

Передача інформації між модулями системи відбувається послідовно, згідно з принципом потокової обробки даних у режимі реального часу (рис. 2.2).

Архітектура передбачає, що всі компоненти – сервер, клієнт і база даних – взаємодіють через стандартизовані повідомлення у форматі JSON, що забезпечує узгодженість і простоту інтеграції.

2.4.1 Генерація даних

На першому етапі сервер-емулятор ініціює створення випадкових або псевдовипадкових значень, які відповідають параметрам заданого сенсора. Для кожного типу сенсора (наприклад, температурного, вологісного, тиску чи освітленості) задаються межі діапазону, одиниці вимірювання, а також коефіцієнти варіації, що дозволяють моделювати реальні фізичні умови. Наприклад, для температурного сенсора може бути задано діапазон від 15 °C до 30 °C з випадковими коливаннями ± 0,5 °C. Генерація відбувається періодично із заданим інтервалом, що імітує постійне надходження даних від реального пристрою.

2.4.2 Формування пакета

Згенеровані дані об’єднуються у структурований об’єкт у форматі **JSON**, який містить усю необхідну службову інформацію:

1) device_id – унікальний ідентифікатор сенсора або пристрою;
2) timestamp – точна часова мітка створення пакета (у форматі ISO 8601);
3) type – тип сенсора (temperature, humidity, pressure тощо);
4) value – виміряне або згенероване значення;
5) unit – одиниця вимірювання (°C, %, hPa тощо). Така структура є уніфікованою для всіх типів пристроїв, що дозволяє легко масштабувати систему або додавати нові сенсори без зміни протоколу передачі даних.

2.4.3 Передача

Сформований JSON-пакет надсилається клієнтам через протокол **WebSocket** у режимі *broadcast*. Це означає, що один сервер може одночасно передавати

інформацію всім підключеним клієнтам. Кожен новий пакет автоматично надходить на сторону клієнта без повторних запитів, що значно зменшує затримку в оновленні даних. Такий архітектурний підхід є безальтернативним для задач реального часу. Він ламає застарілий патерн постійного опитування сервера, який гальмує систему. Натомість впроваджується механізм прямої доставки даних, що забезпечує миттєву реакцію інтерфейсу та стабільний потік телеметрії до користувача.

2.4.4 Обробка

Клієнтська частина, реалізована на **JavaScript**, приймає повідомлення, розбирає JSON-структуру (JSON.parse()) і вносить нові значення у відповідні масиви даних, що відображаються за допомогою **Chart.js**. Кожен сенсор може бути представлений окремим графіком, а візуалізація оновлюється без перезавантаження сторінки завдяки внутрішнім механізмам анімації Chart.js.

Візуальний ряд доповнюється блоком базової математики. Система в реальному часі обчислює ключові метрики: середні величини, дисперсію та пікові навантаження. Це дає оператору глибший контекст, дозволяючи не просто спостерігати за змінами, а й робити обґрунтовані висновки щодо динаміки та стабільності роботи сенсорів.

2.4.5 Збереження

Усі отримані дані дублюються у локальну базу **SQLite**, де зберігаються з часовими мітками для подальшого аналізу. Збереження виконується автоматично після кожного отриманого пакета. Це дозволяє будувати статистику, виконувати порівняльний аналіз або відтворювати історію вимірювань за певний період.

Завдяки такій інтеграції серверна частина може працювати одночасно як елемент реального часу і як сховище історичних даних, що забезпечує повну автономність системи. Послідовність дій між компонентами системи представлена на рис. 2.2. Сервер-емулятор генерує дані, формує пакети та надсилає їх через

WebSocket-з’єднання. Клієнтська частина приймає пакети, візуалізує отриману інформацію й дублює її у локальну базу для подальшої обробки. Завдяки цьому утворюється замкнений цикл даних – від генерації до відображення і збереження, що забезпечує повноцінне тестування системи без участі реальних пристроїв.

Рисунок 2.2 – Логічна схема передачі та обробки IoT-даних

Особливістю такої логіки обробки є її розширюваність і незалежність від конкретних реалізацій апаратних сенсорів. Формат JSON дозволяє інтегрувати нові типи даних без зміни основного коду, а використання WebSocket – забезпечує стабільну швидкодію навіть за великої кількості клієнтів. Це робить систему універсальною платформою для дослідження, тестування або демонстрації IoT-технологій у навчальному процесі.

Резюмуючи, створена схема комунікації демонструє високу ефективність як модель реального IoT-стеку. Вона консолідує всі етапи життєвого циклу даних у безшовному програмному середовищі, що дозволяє досліджувати процеси генерації, передачі та аналізу інформації в комплексі, а не ізольовано.

2.5 Запланована реалізація інтерфейсу візуалізації

У подальшій роботі над кваліфікаційною магістерською роботою планується розроблення клієнтського вебінтерфейсу для відображення IoT-даних у режимі реального часу. Основною метою створення інтерфейсу є забезпечення зручного, інформативного та інтерактивного представлення інформації, яка надходить від емульованих сенсорів. Передбачається використання сучасних вебтехнологій HTML5, CSS3 та JavaScript із застосуванням бібліотеки Chart.js, що дозволяє реалізувати побудову гнучких динамічних графіків і діаграм з оновленням даних без перезавантаження сторінки.

Інтерфейс реалізує концепцію єдиного вікна моніторингу. Дані презентуються через графічні віджети, які оновлюються миттєво завдяки технології WebSocket Push. Це дозволяє користувачу тримати руку на пульсі динамічних змін системи. Додатковою перевагою є модуль роботи з архівами: підключення бази даних SQLite дозволяє виконувати вибірку збережених метрик для детального вивчення історії вимірів.

У процесі розроблення планується реалізувати такі функціональні можливості інтерфейсу:

1) здатність дашборда відображати метрики групи сенсорів одночасно. Кожному пристрою виділяється персональна графічна зона (чарт), що дозволяє оператору зіставляти динаміку процесів та виявляти аномалії в межах єдиної панелі управління;

2) масштабування графіків у часі – користувач зможе змінювати часовий інтервал відображення (наприклад, останні 5 хвилин, година, доба), що дає змогу детальніше аналізувати процеси;

3) зміна діапазону значень по осі Y – для кожного графіка можна буде вручну налаштувати межі відображення, що підвищить точність візуального сприйняття;

4) позначення неактивних сенсорів – у разі втрати зв’язку або відсутності оновлень даних індикатор сенсора змінюватиме колір або статус, попереджаючи користувача про проблему;

5) обчислення статистичних показників – система автоматично визначатиме середні, мінімальні та максимальні значення параметрів за вибраний інтервал часу, відображаючи їх на панелі або у вигляді підказок при наведенні на графік;

6) історичний режим перегляду – користувач зможе вибирати часові відрізки з бази даних SQLite для побудови ретроспективних графіків, що дає змогу оцінити поведінку системи у минулі періоди.

Візуальна частина інтерфейсу проєктується з урахуванням принципів мінімалізму та адаптивності. Основний екран міститиме панель із поточними параметрами (температура, вологість, тиск тощо), область графіків та блок керування параметрами відображення. Усі елементи інтерфейсу будуть реалізовані у стилі responsive design, що забезпечить коректне відображення як на комп’ютерах, так і на мобільних пристроях. Для цього використовуватимуться засоби CSS Grid і Flexbox, а також медіазапити для адаптації компонування.

2.6 Перспективи тестування та очікувані результати

Логічним продовженням розробки є розгортання тестової кампанії. Її мета – емпірично довести ефективність та відмовостійкість системи. Ми маємо переконатися, що логіка обміну повідомленнями працює безпомилково, сервер витримує заплановані навантаження, а візуалізація на клієнті залишається плавною та точною навіть при інтенсивному потоці даних.

Під час випробувань передбачається оцінювання таких параметрів функціонування системи:

- затримка передачі даних між сервером і клієнтом;

- споживання ресурсів сервером при різній кількості підключень;
- поведінка системи при втраті з’єднання або перевищенні навантаження;
- правильність відображення інформації у клієнтській частині.

Очікується, що система забезпечуватиме:

- низьку затримку (до 150 мс при середньому навантаженні);
- масштабованість при підключенні до 50 клієнтів одночасно;
- відмовостійкість за рахунок періодичної перевірки активності підключень;
- зручну візуалізацію даних у реальному часі з можливістю розширення функціоналу у майбутньому.

На основі отриманих результатів можливо сформулювати висновки щодо стабільності роботи системи та її відповідності проєктним вимогам. У разі виявлення відхилень передбачається проведення оптимізації алгоритмів генерації та обробки даних, а також удосконалення механізмів буферизації на стороні клієнта. Очікуваним результатом тестування є підтвердження того, що створена система:

- забезпечує стійку роботу в умовах динамічного оновлення даних;
- підтримує високу швидкість реакції інтерфейсу при збільшенні навантаження;
- демонструє оптимальне використання ресурсів сервера;
- є зручною у використанні та адаптивною для різних сценаріїв моніторингу.

Результати тестів мають подвійну цінність: вони верифікують працездатність поточної реалізації та формують унікальний датасет для дослідження потокових систем. Зібрана статистика може стати базисом для наукових публікацій,

присвячених оптимізації Real-Time протоколів. Крім того, гнучкість архітектури дозволяє легко масштабувати проєкт, інтегруючи модулі штучного інтелекту або адаптуючи рішення під мобільні платформи.

Отже, тестування є кульмінацією інженерного циклу. Воно закриває питання надійності поточної версії системи та водночас відкриває горизонти для її апгрейду. Це критичний момент для аналізу вузьких місць і планування масштабування технологічного ядра рішення.

Висновки до розділу 2

У другому розділі виконано детальне проєктування структури системи емуляції та візуалізації IoT-даних, визначено її основні компоненти, функціональні зв’язки та принципи взаємодії між серверною і клієнтською частинами. Запропонована архітектура побудована за класичною моделлю клієнт–сервер, у межах якої сервер виконує функцію генерації, обробки та передавання даних, а клієнтська частина відповідає за приймання, обробку й візуалізацію інформації у зручній для користувача формі. Такий підхід забезпечує високу гнучкість, масштабованість і простоту подальшої інтеграції з іншими системами.

Обґрунтовано використання WebSocket як стандарту для Real-Time комунікації. Протокол вирішує головну проблему HTTP – надлишковість службових даних у заголовках. Завдяки підтримці постійного з'єднання досягається максимальна ефективність транспорту: система здатна обробляти щільний потік подій від мультисенсорної мережі, не створюючи при цьому черг та затримок.

У процесі проєктування визначено основні технологічні засоби, які використовуватимуться при реалізації системи:

1) Node.js – для створення асинхронного серверного середовища з підтримкою багатьох одночасних з’єднань;
2) Express.js – для обробки допоміжних HTTP-запитів та маршрутизації;
3) бібліотека *ws* – для реалізації WebSocket-з’єднань;

4) Chart.js – для побудови інтерактивних графіків і динамічної візуалізації даних;

5) SQLite – для локального збереження історії вимірювань та подальшого статистичного аналізу.

Розроблено алгоритм наскрізної обробки інформації. Він об'єднує всі етапи: створення, трансляцію, візуалізацію та логування даних у єдиний потік. Це гарантує автономність системи, дозволяючи моделювати поведінку складних мереж виключно програмними засобами, без прив'язки до фізичного «заліза».

У роботі закладено фундамент візуальної ергономіки системи. Головна вимога до дашборда – реактивність: користувач має керувати масштабом графіків та бачити статистику в реальному часі. Також пропрацьовано сценарії майбутнього розширення: від адаптації верстки під різні екрани до функціоналу ретроспективного аналізу, що дозволить піднімати архіви вимірювань із бази даних.

У завершальній частині розділу окреслено підходи до тестування системи після її реалізації, які охоплюють вимірювання затримок передачі даних, оцінювання стабільності з’єднання, поведінку при перевантаженні та коректність відображення показників на клієнтському рівні. Такі випробування дадуть змогу визначити реальні можливості створеної архітектури та підтвердити її ефективність.

Отже, результати другого розділу формують технічну і методологічну основу для переходу до наступного етапу – реалізації прототипу системи, її налагодження, тестування та аналізу ефективності. Отримані напрацювання мають практичну цінність і можуть бути використані як у науково-дослідницьких, так і в навчальних цілях для демонстрації принципів роботи систем Інтернету речей у режимі реального часу.

3 РЕАЛІЗАЦІЯ РОЗРОБЛЕНОЇ СИСТЕМИ

3.1 Обґрунтування вибору середовища розробки та структура проєкту

У процесі програмної реалізації системи емуляції та візуалізації IoT-даних першочерговим завданням став вибір оптимального набору інструментальних засобів, які б дозволили забезпечити кросплатформність, високу швидкість розробки та легкість подальшого масштабування програмного продукту. З огляду на специфіку завдання, яке полягає в обробці потокових даних у режимі реального часу, розробка велася в операційній системі Windows 11, проте, завдяки використанню універсального кросплатформного середовища виконання Node.js, серверна частина системи може бути безперешкодно розгорнута на інших операційних системах, таких як Linux (зокрема, дистрибутиви Ubuntu, Debian) або macOS, без необхідності внесення будь-яких змін у вихідний програмний код.

В якості основного інтегрованого середовища розробки (IDE) було обрано Visual Studio Code (VS Code). Цей вибір зумовлений наявністю широкого спектру розширень для підтримки синтаксису сучасних стандартів JavaScript (ES6+), HTML5 та CSS3, а також вбудованими засобами для налагодження коду та роботи з терміналом. Для забезпечення контролю версій та відстеження етапів розробки використовувалася розподілена система управління версіями Git, що дозволило структурувати процес внесення змін до програмного коду.

Організація файлової системи проєкту була спроєктована з дотриманням принципів модульності та відокремлення логіки сервера від клієнтського інтерфейсу. Структура проєкту має наступний ієрархічний вигляд:

1) *server.js* – головний виконуваний файл серверної частини, який слугує точкою входу в застосунок та ініціалізує всі необхідні служби;

2) *package.json* – файл маніфесту проєкту, що містить метадані про розробку, скрипти запуску та перелік залежностей (express для маршрутизації, *ws* для роботи з WebSocket, *sqlite*3 для взаємодії з базою даних), необхідних для коректного функціонування системи;

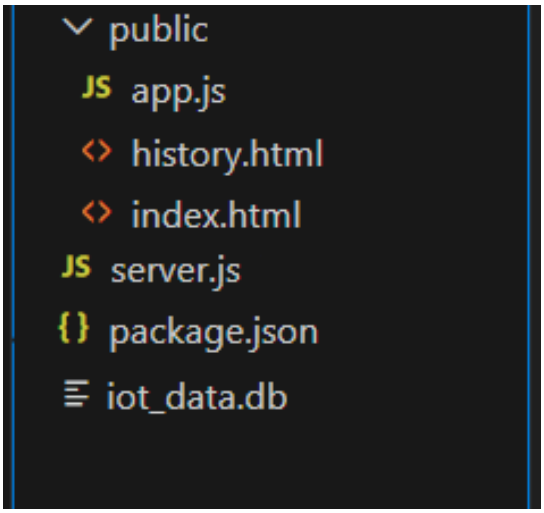


Рисунок 3.1 – Структура файлової системи проєкту в середовищі VS Code

3) *iot_data.db* – бінарний файл реляційної бази даних SQLite, який використовується для персистентного збереження історії вимірювань;

4) *public/* – директорія, що містить статичні файли клієнтської частини, які передаються браузеру користувача:

1) *index.html* – основний файл розмітки головної панелі моніторингу (Dashboard), що визначає структуру інтерфейсу;

2) *history.html* – файл розмітки сторінки журналу подій, призначений для перегляду архівних даних та експорту звітів;

3) *app.js* – файл, що містить клієнтську бізнес-логіку, алгоритми обробки вхідних повідомлень через WebSocket та функції побудови динамічних графіків.

Така організація файлової системи забезпечує чітке логічне розмежування між серверною частиною (backend), яка відповідає за обробку даних та взаємодію з базою, і клієнтською частиною (frontend), що відповідає виключно за візуалізацію. Це спрощує навігацію по проєкту, полегшує подальшу підтримку коду та дозволяє масштабувати систему шляхом додавання нових модулів без порушення цілісності існуючої архітектури.

3.2 Реалізація серверного модуля та логіки емуляції

Серверна частина програмного комплексу реалізована на базі платформи Node.js, що дозволило використати подійно-орієнтовану модель для ефективної обробки множинних з’єднань. Ключовим елементом підсистеми емуляції є спеціально розроблений клас *VirtualSensor*, який призначений для імітації фізичних процесів зміни показників навколишнього середовища.

На відміну від спрощених підходів, що базуються на генерації випадкових чисел у заданому діапазоні [20], у розробленому класі реалізовано складніший математичний апарат, який враховує механізм інерції фізичних величин та вплив зовнішніх збурюючих факторів, таких як сила вітру. Метод *update()*, який викликається циклічно, приймає як аргумент поточне значення швидкості вітру і на основі цього коригує цільові значення температури та вологості. Це дозволяє програмно імітувати ефект «видування» тепла (Wind Chill factor) та прискореного випаровування вологи при сильному вітрі.

Для забезпечення гнучкості системи було розроблено структуру конфігураційних сценаріїв, що визначають поведінку сенсорів у різних умовах (зима, літо, сауна тощо). Ініціалізація віртуальних сенсорів відбувається через конструктор класу, який задає початкові діапазони вимірювань. Фрагмент програмної реалізації конфігурації сценаріїв та ініціалізації класу наведено на рис. 3.1.

```
const SCENARIOS = {
    'normal': { name: 'Кімната (Норма)', temp: [20, 25], hum: [40, 60], press: [1010, 1015], light: [300, 600] },
    'summer': { name: 'Літо (Спека)',    temp: [30, 38], hum: [20, 40], press: [1005, 1010], light: [800, 1000] },
    'winter': { name: 'Зима (Мороз)',    temp: [-10, -2], hum: [70, 90], press: [1020, 1035], light: [50, 200] },
    'autumn': { name: 'Осінь (Дощ)',     temp: [5, 12],  hum: [85, 99], press: [990, 1000],  light: [100, 300] },
    'sauna':  { name: 'Екстрим (Сауна)', temp: [80, 100], hum: [90, 100], press: [1000, 1010], light: [50, 150] }
};

class VirtualSensor {
    constructor(id, type, unit, initialRange) {
        this.id = id; this.type = type; this.unit = unit;
        this.min = initialRange[0]; this.max = initialRange[1];
        this.value = (this.min + this.max) / 2;
    }
```

Рисунок 3.2 – Конфігурація сценаріїв емуляції та структура класу сенсора

Для забезпечення функціонування системи в режимі реального часу було інтегровано бібліотеку *ws*. Сервер сконфігуровано для роботи в режимі «broadcasting» (широкомовна передача): при кожній ітерації головного циклу емуляції, що відбувається з періодичністю 500 мс, сформований пакет даних у форматі JSON миттєво надсилається всім активним клієнтам, підключеним до сокету. Паралельно з процесом відправки здійснюється асинхронний запис параметрів вимірювання у таблицю *measurements* бази даних SQLite, що гарантує збереження історії навіть у разі аварійної зупинки сервера.

Окремою важливою складовою реалізації є система кліматичних сценаріїв («Зима», «Літо», «Сауна», «Осінь»), параметри яких (мінімальні та максимальні межі для кожного типу датчика) зберігаються в конфігураційному об’єкті SCENARIOS. Зміна активного сценарію відбувається за ініціативою клієнта шляхом надсилання відповідної керуючої команди, що дозволяє динамічно змінювати граничні умови генерації даних без необхідності перезавантаження серверного процесу.

3.2.1 Алгоритмічна реалізація емуляції фізичних процесів

Ключовою особливістю розробленої системи є відмова від генерації суто випадкових значень (Random Walk) на користь імітаційної моделі, що наближена до фізичних процесів. Для цього в класі *VirtualSensor* реалізовано алгоритм, який враховує інерційність вимірюваних величин та кореляцію між параметрами навколишнього середовища [22].

Математична модель оновлення показників базується на чотирьох компонентах:

1) цільове значення (Target Value) – визначається обраним кліматичним сценарієм (наприклад, діапазон температур для сценарію «Зима»);

2) вплив зовнішніх збурень (Disturbance) – у даній системі реалізовано вплив швидкості вітру на температурні показники та вологість;

3) інерційна складова (Inertia) – забезпечує плавність зміни показників, імітуючи теплову інерцію датчиків або повітряних мас;

4) стохастичний шум (Noise) – імітує похибку вимірювання реальних цифрових сенсорів.

Алгоритм розрахунку миттєвого значення температури T_{new} на кожній ітерації циклу емуляції можна описати формулою:

$$T_{target} = T_{scen} - k_w \times V_{wind};$$

$$T_{new} = T_{old} + (T_{target} - T_{old}) \times k_{inert} + \delta_{noise}$$

де T_{scen} – базове цільове значення температури для поточного сценарію (середина діапазону);

V_{wind} – поточна швидкість вітру (м/с);

k_w – коефіцієнт вітрового охолодження (Wind Chill Factor), у програмі прийнято k_w = 0,3;

k_{inert} – коефіцієнт інерції (швидкість реакції системи), прийнято 0,05 (тобто за один такт значення змінюється на 5% від різниці);

δ_{noise} – випадкова величина (шум), що варіюється в межах ± 10 % від діапазону вимірювання.

Аналогічний підхід застосовано до вологості, де вітер виступає фактором осушення повітря (коефіцієнт 0,5).

Така реалізація забезпечує візуальну автентичність даних. Графіки точно відтворюють динаміку фізичних явищ: вони залишаються плавними у стабільному стані, але демонструють адекватну реакцію на зміни умов. Введення фактору затримки дозволяє змоделювати природну інерційність сенсорів, що є критичним для реалістичності сприйняття.

Результатом роботи стала математична чистота графіків. Завдяки введенню інерції вдалося позбутися неприродної дискретності, яка часто видає штучне походження даних. Тепер криві еволюціонують органічно: реакція на зміну

сценарію відбувається без розривів функції, імітуючи поведінку масивного фізичного тіла або реального метеорологічного явища.

```
update(windSpeed = 0) {
    // Розрахунок базового цільового значення як середини діапазону сценарію
    let target = (this.min + this.max) / 2;

    // Моделювання фізичного впливу вітру на показники
    if (this.type === 'temperature') {
        // Коефіцієнт охолодження: чим сильніший вітер, тим нижча цільова температура
        target -= windSpeed * 0.3;
    } else if (this.type === 'humidity') {
        // Ефект висушування повітря потоками вітру
        target -= windSpeed * 0.5;
    }

    // Реалізація механізму інерції: плавне наближення поточного значення до цільового
    const diff = target - this.value;
    let step = diff * 0.05; // Крок зміни складає 5% від різниці, що забезпечує плавність

    // Додавання стохастичного шуму для імітації похибки реального вимірювального приладу
    const noise = (Math.random() - 0.5) * (this.max - this.min) * 0.1;
    this.value += step + noise;

    // Повернення результату з округленням до двох знаків після коми
    return parseFloat(this.value.toFixed(2));
}
```

Рисунок 3.3 – Програмна реалізація алгоритму емуляції фізичних процесів

Реалізована серверна архітектура повністю готує дані для передачі: вони є валідними, фізично обґрунтованими та структурованими у форматі JSON. Це створює надійну основу для наступного етапу роботи системи – візуалізації цих даних на боці клієнта, що буде розглянуто в наступному підрозділі.

3.2.2 Структура даних та реалізація збереження історії

Критично важливим етапом роботи системи є формування уніфікованого формату повідомлень для передачі через канал WebSocket та організації їх довготривалого зберігання. Для забезпечення сумісності з клієнтською частиною було розроблено структуру JSON-пакета, яка містить не лише виміряне значення, а й метадані: ідентифікатор пристрою, тип вимірювання, одиниці та точну часову мітку генерації.

Реалізація збереження даних виконана на базі СКБД SQLite. Вибір цієї технології зумовлений її «легковаговістю» (serverless architecture) та високою швидкістю запису, що є оптимальним для локальних IoT-шлюзів.

На етапі ініціалізації сервера програмно створюється таблиця *measurements* за допомогою SQL-запиту, що гарантує наявність коректної структури бази даних при першому запуску системи. Фрагмент коду, що відповідає за ініціалізацію БД та формування пакетів даних, наведено на рис. 3.4.

```
// --- ФУНКЦІЯ ЗБЕРЕЖЕННЯ ---
function saveAndBroadcast(payload) {
    // 1. Зберігає в БД тільки якщо це НЕ вітер
    if (payload.type !== 'wind') {
        db.run(`INSERT INTO measurements (device_id, type, value, unit, timestamp) VALUES (?, ?, ?, ?, ?)`,
            [payload.device_id, payload.type, payload.value, payload.unit, payload.timestamp]);
    }

    // 2. Відправляє клієнтам ВСЕ
    payload.action = 'data';
    broadcast(payload);
}

// --- 5. ЦИКЛ ---
setInterval(() => {
    if (isPaused) return;

    const now = new Date().toISOString();

    const windSensor = mySensors.find(s => s.type === 'wind');
    const currentWindSpeed = windSensor.update();
    saveAndBroadcast({ device_id: windSensor.id, timestamp: now, type: windSensor.type, value: currentWindSpeed, unit: windSensor.unit });

    mySensors.forEach(sensor => {
        if (sensor.type !== 'wind') {
            const newValue = sensor.update(currentWindSpeed);
            saveAndBroadcast({ device_id: sensor.id, timestamp: now, type: sensor.type, value: newValue, unit: sensor.unit });
        }
    });
}, 500);

process.on('SIGINT', () => { db.close(() => process.exit(0)); });
server.listen(PORT, '0.0.0.0', () => { console.log(`Сервер запущено: http://localhost:${PORT}`); });
```

Рисунок 3.4 – Реалізація схеми бази даних та структури повідомлень

Функція *saveAndBroadcast* виконує подвійну роль: вона асинхронно записує сформований об’єкт у базу даних та одночасно транслює його (broadcast) усім підключеним клієнтам. Це дозволяє розділити процеси архівування та візуалізації, забезпечуючи збереження історії навіть у випадку відсутності активних клієнтів.

Варто також зазначити, що в алгоритмі збереження реалізовано механізм оптимізації дискового простору. Система виконує селективний запис: критично важливі параметри (температура, тиск) зберігаються в базу даних для історії, тоді як високодинамічні параметри, що використовуються виключно для миттєвої

візуалізації (наприклад, анімація сили вітру), транслюються клієнтам, але виключаються з процесу архівації. Такий підхід дозволяє уникнути надмірного зростання розміру файлу бази даних при тривалій роботі системи.

3.3 Особливості реалізації клієнтського інтерфейсу та візуалізації даних

Клієнтська частина програмного комплексу спроєктована та реалізована за принципом односторінкового вебзастосунку (Single Page Application, SPA), що забезпечує високу швидкість відгуку інтерфейсу. Верстка інтерфейсу є повністю адаптивною завдяки використанню сучасних можливостей CSS, зокрема технології Flexbox та медіазапитів, що дозволяє коректно та зручно відображати панель приладів як на широкоформатних моніторах персональних комп'ютерів, так і на екранах мобільних пристроїв.

Для графічної візуалізації потоків даних було використано потужну бібліотеку Chart.js, яка працює з елементом HTML5 Canvas. У системі реалізовано три різних типи візуального представлення даних, залежно від їх фізичної природи:

1) лінійні графіки (Line Chart) – застосовуються для відображення динаміки зміни температури, атмосферного тиску та рівня освітленості у часі. Для покращення візуального сприйняття використано кубічну інтерполяцію кривих, що робить лінії графіків плавними;

2) кругова діаграма (Doughnut Chart) – використана спеціально для відображення відносної вологості повітря, оскільки цей показник має відсотковий характер (діапазон 0–100 %), і кругова діаграма інтуїтивно краще передає ступінь «заповненості». Загальний вигляд реалізованої панелі моніторингу, що об'єднує ці типи візуалізації, наведено на рис. 3.5;

Рисунок 3.5 – Головна панель моніторингу (Dashboard) у світлій темі

3) кастомні CSS-віджети – для відображення сили вітру було розроблено унікальний індикатор у вигляді шкали рівня сигналу («палички»), колір та кількість активних елементів якого змінюються динамічно залежно від числового значення інтенсивності вітру. Такий підхід дозволив заощадити простір на екрані та інтегрувати показник безпосередньо в панель інструментів (рис. 3.6).

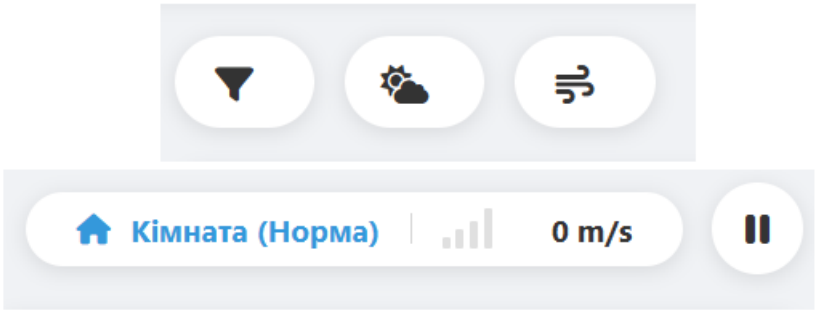


Рисунок 3.6 – Панель інструментів з інтегрованим віджетом сили вітру

Технічною особливістю клієнтської реалізації є механізм оптимізованого оновлення графіків [23]. Дані, що надходять, зберігаються в локальному масиві chartDataStorage, який функціонує за принципом черги (FIFO – First In, First Out) з жорстким обмеженням у 50 точок. Це рішення запобігає переповненню оперативної пам’яті браузера при тривалій безперервній роботі системи моніторингу.

Додатково в інтерфейсі реалізовано низку функцій для покращення користувацького досвіду (UX):

- темна тема інтерфейсу: Реалізовано механізм перемикання глобальних змінних CSS (:root) для забезпечення комфортної роботи оператора в умовах низького зовнішнього освітлення. Стан обраної теми персистентно зберігається в localStorage браузера. Приклад адаптації інтерфейсу до темного режиму наведено на рис. 3.7;

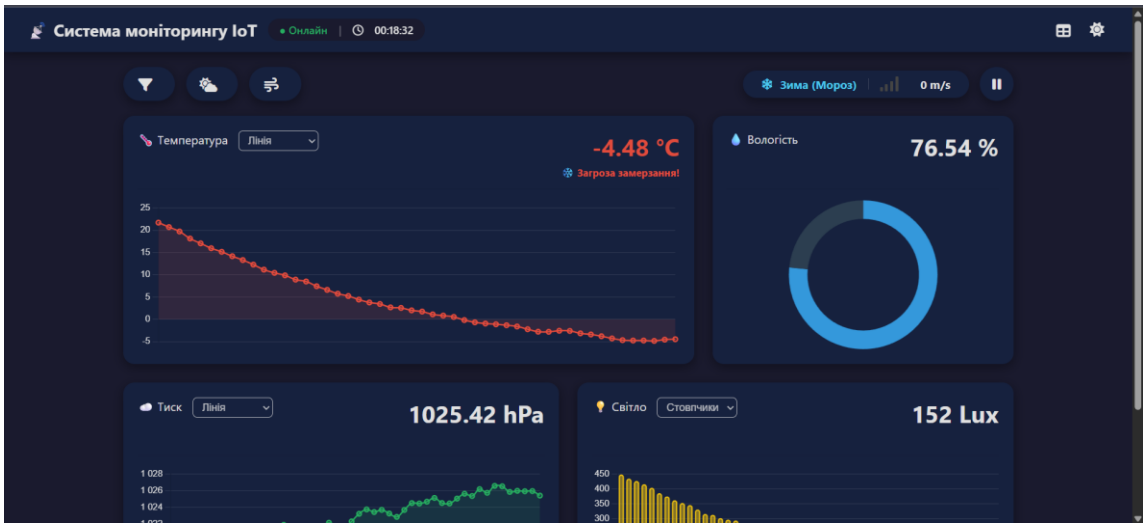


Рисунок 3.7 – Відображення інтерфейсу в темному режимі

- інтерактивна фільтрація: Користувач має можливість приховувати окремі графіки, при цьому інтерфейс плавно перебудовується завдяки використанню CSS Transitions, заповнюючи вільний простір іншими елементами;

- модуль роботи з історією: Розроблено окрему вебсторінку history.html, яка здійснює запити до REST API сервера (/api/history) для отримання повного архіву даних. Реалізовано функціонал експорту вибірки даних у форматі CSV з коректним кодуванням для подальшого аналізу в табличних процесорах, таких як Microsoft Excel (рис. 3.8).

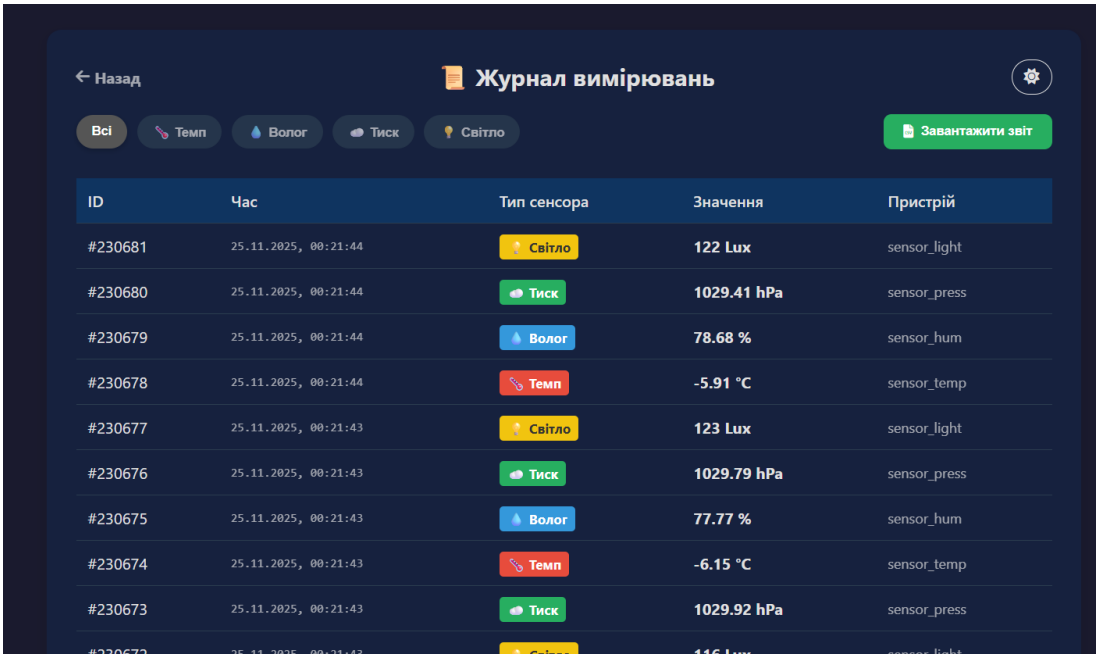


Рисунок 3.8 – Журнал історії вимірювань з функцією експорту

Окрему увагу при реалізації інтерфейсу було приділено його адаптивності. Завдяки використанню CSS-правил @media та гнучких контейнерів (Flexbox), макет сторінки автоматично підлаштовується під розмір екрана пристрою користувача. На мобільних пристроях панель інструментів перебудовується у вертикальний стек, а графіки займають 100 % ширини екрана, що забезпечує зручність моніторингу навіть зі смартфона або планшета (рис.3.9).

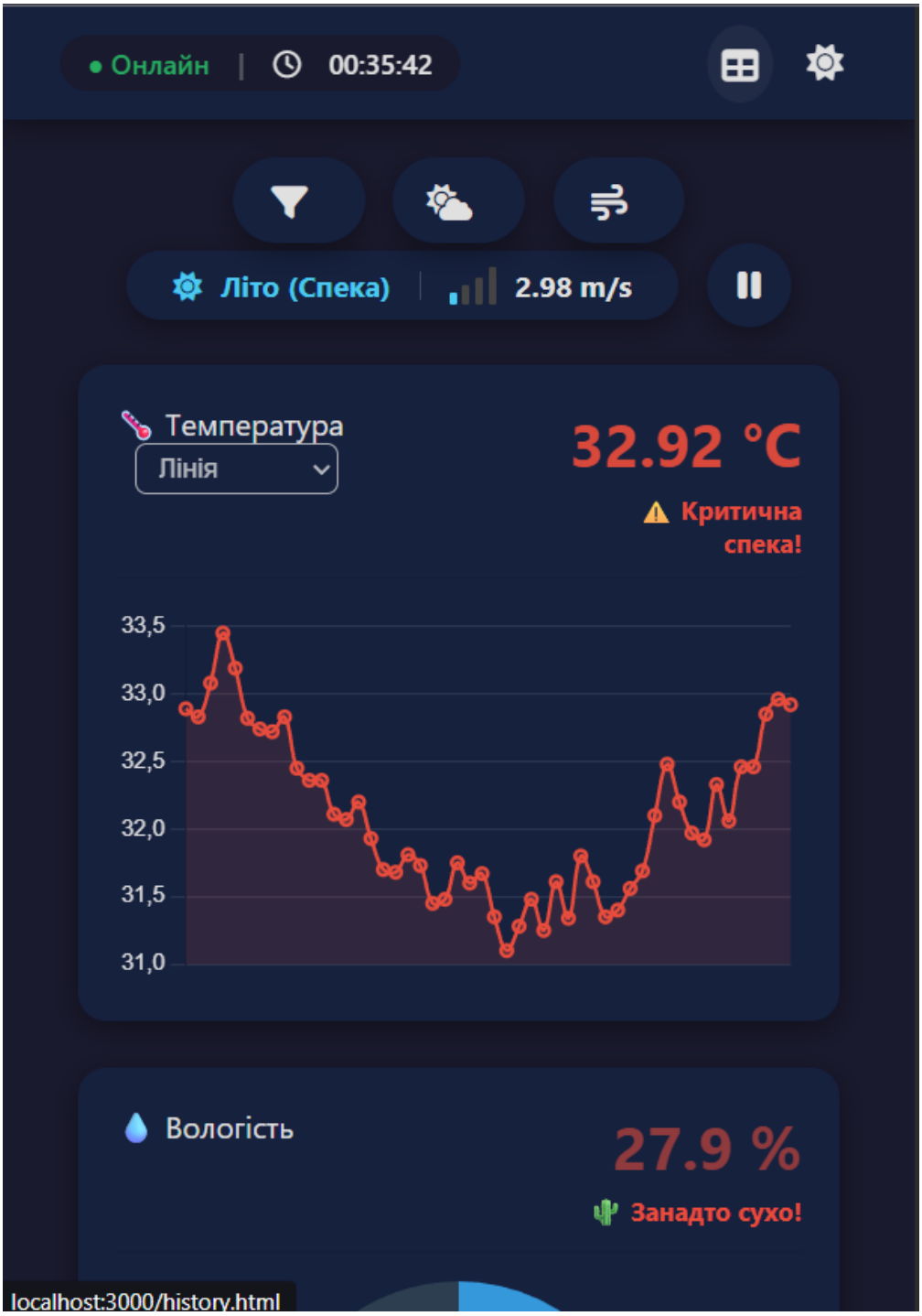


Рисунок 3.9 – Адаптивне відображення інтерфейсу на мобільному пристрої

Функція збереження даних забезпечує локалізаційну коректність. Використання мітки порядку байтів (BOM) у заголовку файлу запобігає спотворенню символів при відкритті в Excel. Такий підхід дозволяє безшовно інтегрувати результати емуляції в робочі процеси зовнішньої аналітики та підготовки наукової звітності.

Резюмуючи, клієнтська частина виступає як повноцінна SPA-екосистема. Вона гарантує замкнений цикл взаємодії з інформацією, нівелюючи розрив між спостереженням у реальному часі та аналітичною обробкою архівних записів.

Висновки до розділу 3

У третьому розділі кваліфікаційної роботи було детально розглянуто процес практичної реалізації програмного комплексу системи емуляції та візуалізації IoT-даних. У ході виконання роботи було успішно вирішено комплекс завдань, пов’язаних із розробкою як серверної, так і клієнтської частин системи, а також забезпеченням їх ефективної взаємодії.

Технологічним серцем платформи обрано середовище Node.js, але головна інновація криється в алгоритмічній площині. Було створено об'єктну модель віртуальних пристроїв, яка відтворює реальну поведінку фізичних тіл. Система емулює не просто "сухі" значення, а динамічні процеси: алгоритми враховують інерцію та взаємовплив параметрів навколишнього середовища. Такий підхід перетворює симуляцію з хаотичного набору цифр на когерентну модель реальності.

Для організації інформаційного обміну було налаштовано високоефективний канал зв’язку з використанням протоколу WebSocket. Це дозволило реалізувати режим миттєвої широкомовної передачі даних («broadcasting»), що гарантує синхронне отримання інформації всіма підключеними клієнтами з мінімальними затримками, недосяжними при використанні традиційних HTTP-запитів. Паралельно з передачею даних було інтегровано підсистему збереження історії вимірювань на основі реляційної бази даних SQLite. При цьому було застосовано методи селективного запису, що дозволило оптимізувати навантаження на дискову підсистему шляхом фільтрації високодинамічних візуальних параметрів.

Окрему увагу було приділено реалізації клієнтської частини, яка виконана у вигляді односторінкового вебзастосунку (SPA). Використання бібліотеки Chart.js дозволило створити комбіновану систему візуалізації, що включає лінійні графіки для часових рядів, кругові діаграми для відсоткових показників та спеціалізовані

кастомні віджети. Для забезпечення стабільної роботи інтерфейсу при тривалих сеансах моніторингу було розроблено та впроваджено алгоритм керування оперативною пам’яттю браузера, який працює за принципом FIFO, обмежуючи кількість точок на графіках.

Крім того, розроблений інтерфейс відповідає сучасним вимогам до юзабіліті (UX). Завдяки адаптивному дизайну система коректно відображається на пристроях з різною роздільною здатністю екрана, а підтримка темної теми та інтерактивної фільтрації графіків створює комфортні умови для роботи оператора. Функціонал системи було розширено модулем експорту звітів у формат CSV з підтримкою регіональних стандартів кодування, що забезпечує інтероперабельність розробленого комплексу із зовнішніми аналітичними інструментами, такими як Microsoft Excel.

Таким чином, результатом роботи над третім розділом є створення повністю працездатного, завершеного програмного продукту, який готовий до етапу комплексного функціонального та навантажувального тестування.

4 ТЕСТУВАННЯ ТА АНАЛІЗ ЕФЕКТИВНОСТІ РОЗРОБЛЕНОЇ СИСТЕМИ

4.1 Методика та результати функціонального тестування

Після завершення етапу розробки, для перевірки відповідності створеної системи сформульованим у технічному завданні вимогам, було проведено комплексне функціональне тестування. Тестування виконувалося методом «чорної скриньки» (Black Box Testing), який передбачає перевірку реакції системи на різноманітні дії користувача та вхідні дані без заглиблення у внутрішню структуру коду під час самого тесту.

Умови проведення тестування:

- апаратна платформа: Ноутбук (CPU AMD r5 5500U, 16 GB RAM);
- операційна система: Windows 11;
- середовище виконання сервера: Node.js v20 (LTS);
- клієнтське середовище: Браузер Google Chrome (версія 120+);
- інструменти діагностики: Chrome DevTools (вкладка Network) для моніторингу WebSocket-з'єднань.

Метою тестування було підтвердження коректності роботи основних функцій: встановлення з’єднання, візуалізації даних, зміни режимів роботи та збереження інформації. Результати перевірки основних сценаріїв роботи систематизовано та наведено в табл. 4.1.

Таблиця 4.1 – Результати функціонального тестування системи

№ з/п	Сценарій тестування	Очікуваний результат роботи системи	Фактичний результат	Статус
1	Ініціалізація підключення клієнта до сервера	Індикатор статусу змінюється на зелений «Онлайн», графіки починають оновлюватися даними в реальному часі	Індикатор змінив колір, потік даних відображається коректно	✓ OK

№ з/п	Сценарій тестування	Очікуваний результат роботи системи	Фактичний результат	Статус
2	Зміна кліматичного сценарію на «Зима»	Значення температури повинні почати плавно знижуватися до від'ємних значень	Графік температури продемонстрував плавний спад до -5°C протягом 10-15 секунд	✓ OK
3	Перевірка впливу вітру на температурні показники	При ручному увімкненні режиму «Сильний вітер» температура повинна знижуватися швидше через програмний коефіцієнт охолодження	При вітрі >15 м/с зафіксовано додаткове падіння температури на 3°C відносно норми	✓ OK
4	Аварійне відключення сервера	Клієнтський інтерфейс має зафіксувати розрив з’єднання та показати статус «Офлайн»	Статус миттєво змінився на червоний «Офлайн», оновлення графіків зупинилося	✓ OK
5	Перевірка збереження та завантаження історії	Після перезавантаження сторінки браузера графіки не повинні бути пустими, а мають відобразити попередні дані	При оновленні сторінки графіки автоматично завантажили історію з бази даних SQLite	✓ OK
6	Експорт звіту у файл	Завантажується файл у форматі .csv, який коректно відкривається в Excel з правильним кодуванням кирилиці	Файл успішно завантажено, дані структуровані по стовпчиках, текст читабельний	✓ OK

Для підтвердження коректності роботи механізму сценаріїв було зафіксовано поведінку системи при перемиканні режимів. На рис. 4.1 зображено реакцію системи на активацію сценарію «Зима», де чітко видно динаміку зниження температури (червоний графік) та стабілізацію показників у новому діапазоні.

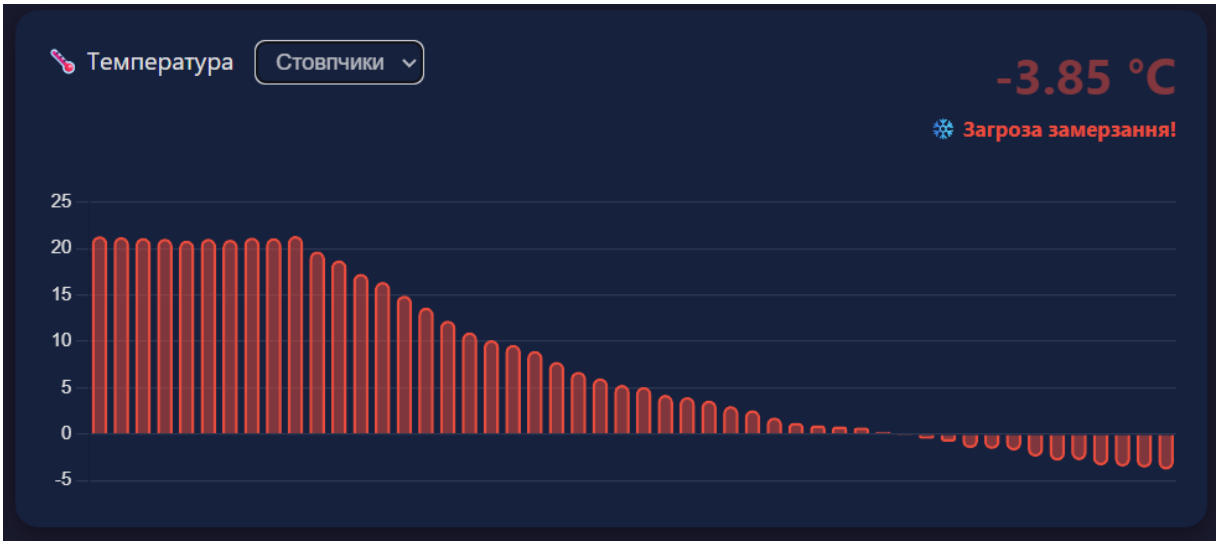


Рисунок 4.1 – Динаміка зміни показників при перемиканні кліматичного сценарію

Окрему серію тестів було проведено для перевірки реалізованої математичної моделі взаємовпливу параметрів, зокрема впливу швидкості вітру на показники температури та вологості. При ручному увімкненні режиму «Сильний вітер» (через панель керування) система продемонструвала прискорене зниження температури (ефект охолодження вітром) та зменшення вологості, що відповідає закладеним фізичним закономірностям. Візуалізація стану вітру на клієнтській стороні реалізована через динамічний віджет у панелі інструментів (рис. 4.2), який змінює колірну індикацію залежно від інтенсивності потоку.

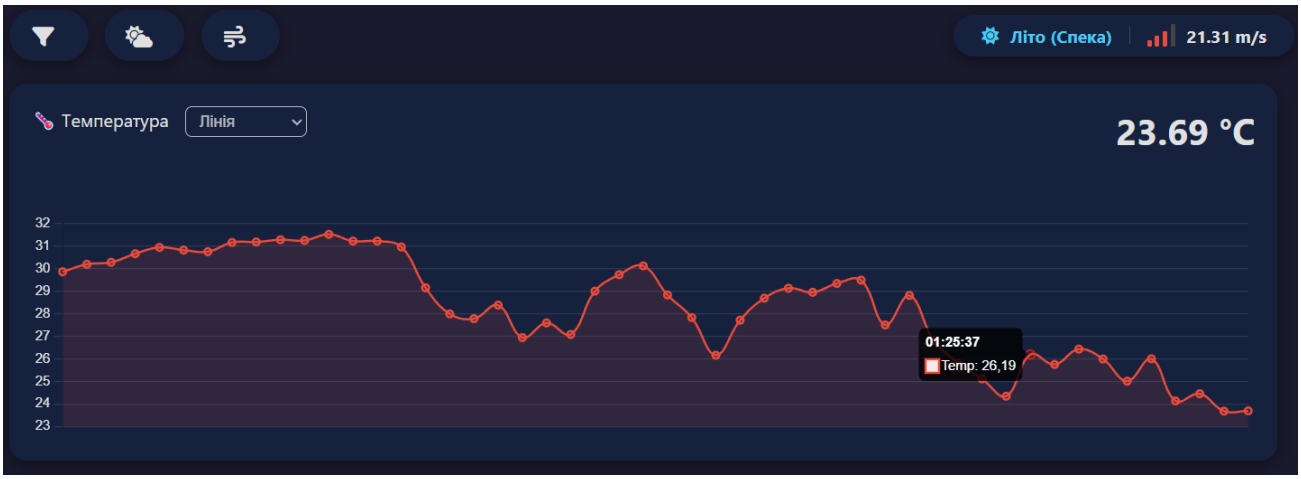


Рисунок 4.2 – Відображення критичних показників вітру та реакція системи на зміну погодних умов

Також було перевірено модуль експорту даних. Згенерований CSV-файл успішно відкривається у табличному процесорі Microsoft Excel, при цьому кодування кириличних символів відображається коректно завдяки додаванню BOM-мітки (рис. 4.3).

	A	B	C	D	E	F	G
	ID	Час	Тип сенсо	Значення	Одиниці	ID Пристрою	
	255865	25.11.202	light	143	Lux	sensor_light	
	255864	25.11.202	humidity	75,67	%	sensor_hum	
	255863	25.11.202	pressure	1029,65	hPa	sensor_press	
	255862	25.11.202	temperatu	-6,55	°C	sensor_temp	
	255861	25.11.202	pressure	1029,05	hPa	sensor_press	
	255860	25.11.202	light	147	Lux	sensor_light	
	255859	25.11.202	humidity	75,14	%	sensor_hum	
	255858	25.11.202	temperatu	-6,12	°C	sensor_temp	
	255857	25.11.202	light	153	Lux	sensor_light	
	255856	25.11.202	humidity	74,99	%	sensor_hum	
	255855	25.11.202	pressure	1028,63	hPa	sensor_press	
	255854	25.11.202	temperatu	-6,48	°C	sensor_temp	
	255853	25.11.202	pressure	1028,79	hPa	sensor_press	
	255852	25.11.202	light	149	Lux	sensor_light	
	255851	25.11.202	humidity	75,47	%	sensor_hum	

Рисунок 4.3 – Фрагмент експортованого звіту з даними сенсорів

Проведене тестування підтвердило, що всі закладені функціональні можливості працюють коректно. Механізм плавної зміни значень (інерція) функціонує згідно з розробленою математичною моделлю, а інтерфейс адекватно реагує на дії користувача без видимих затримок чи помилок відображення.

4.2 Аналіз продуктивності та мережевих затримок

Одним із ключових завдань даної роботи було забезпечення передачі IoT-даних у режимі реального часу з мінімальними затримками. Для об’єктивної оцінки ефективності використання протоколу WebSocket у порівнянні з традиційними методами було проведено серію вимірювань мережевої затримки (Latency) та

накладних витрат трафіку. Для порівняння використовувався класичний підхід HTTP Polling (періодичні запити).

Вимірювання проводилися за допомогою вбудованих інструментів розробника браузера Google Chrome (Chrome DevTools, вкладка Network). Умови тесту передбачали використання локальної мережі (Wi-Fi 5GHz) та частоту оновлення даних 2 Гц (інтервал 500 мс). Візуалізацію потоку фреймів даних через WebSocket наведено на рис. 4.4.

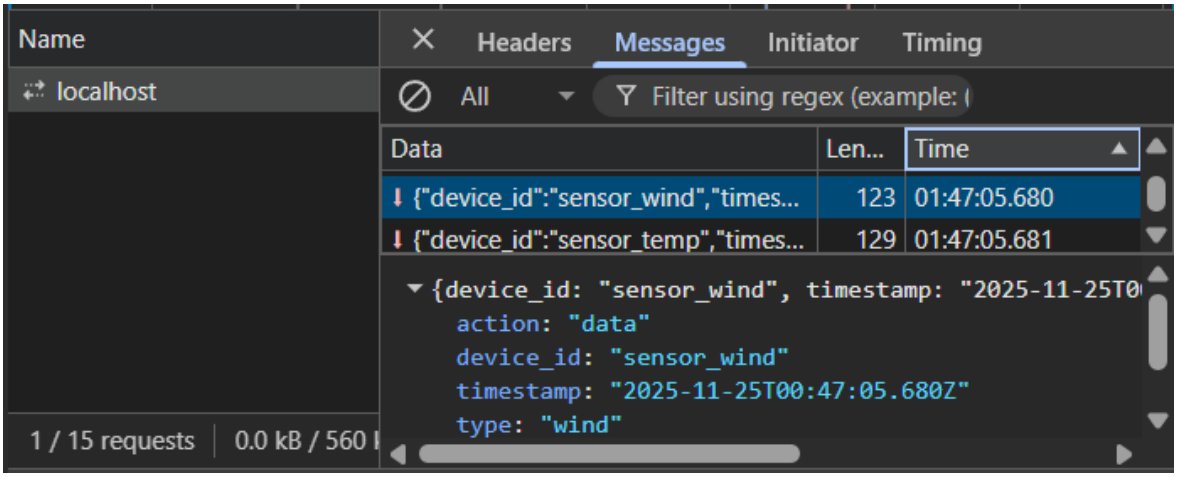


Рисунок 4.4 – Аналіз потоку даних WebSocket у інструментах розробника

Результати вимірювань показали суттєві переваги обраної архітектури:

1) **ефективність передачі даних:** як видно зі знімка екрана, загальний розмір пакета складає всього 123–129 байт. Враховуючи, що корисне навантаження (JSON-рядок із даними) займає близько 120 байт, розмір службового заголовка WebSocket складає лише 2–8 байт (менше 5 % від обсягу пакета);

2) **порівняння з HTTP**: для передачі того самого обсягу корисних даних (120 байт) через класичний HTTP-запит, довелося б передати додатково 200–800 байт HTTP-заголовків (Cookies, User-Agent, Token). Таким чином, використання WebSocket зменшує "паразитний" трафік у 5–10 разів;

3) **затримка передачі (RTT):** середня затримка (Round Trip Time) для передачі повідомлення через WebSocket склала **2–5 мс** у локальній мережі. Для методу HTTP Polling мінімальна затримка складає 20–50 мс плюс час,

необхідний на встановлення нового TCP-з’єднання (handshake) для кожного запиту, що є критичним при високій частоті оновлення.

Для оцінки економії трафіку було проведено розрахунковий експеримент для сеансу тривалістю 1 година при частоті оновлення 2 рази на секунду (7200 запитів). Результати зведено в табл. 4.2.

Таблиця 4.2 – Порівняння споживання трафіку за 1 годину роботи

Параметр	**HTTP Polling**	**WebSocket (реалізовано)**	**Економія**
Кількість повідомлень	7200	7200	-
Середній розмір заголовка	~500 байт	~6 байт	98,8 %
Корисне навантаження (JSON)	~100 байт	~100 байт	0 %
Загальний обсяг трафіку	~4.3 Мбайт	~0.76 Мбайт	~5,6 разів

Результати вимірювань доводять кратне зменшення паразитного трафіку. У проєкції на системи з тисячами вузлів, це запобігає передачі гігабайтів зайвої інформації. Така оптимізація є безальтернативною для IoT-рішень з обмеженим бюджетом енергії та трафіку, оскільки дозволяє суттєво знизити операційні витрати на мобільний зв'язок та подовжити автономність датчиків.

4.3 Перевірка стабільності та масштабованості системи

Для перевірки стабільності роботи серверної частини під навантаженням було проведено стрес-тест, що полягав у емуляції одночасного підключення значної кількості клієнтів. Для цього на тестовому стенді було відкрито 20 незалежних вкладок браузера з запущеним дашбордом, кожна з яких встановлювала окреме WebSocket-з’єднання з сервером та отримувала повний потік даних у режимі реального часу.

Моніторинг споживання системних ресурсів здійснювався за допомогою системного інструмента «Диспетчер завдань» (Task Manager). Результати вимірювань у момент пікового навантаження наведено на рис. 4.5.

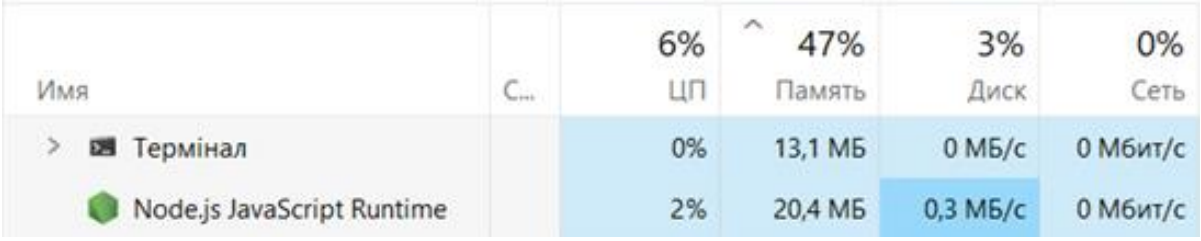


Рисунок 4.5 – Моніторинг використання системних ресурсів
під час навантаження

Моніторинг демонструє високу ефективність розробленого програмного забезпечення. Незважаючи на активний процес генерації даних, розсилку повідомлень через WebSocket та паралельний запис історії в базу даних, споживання ресурсів залишається мінімальним.

Результати спостереження за системними ресурсами:

1) **навантаження на ЦП –** процес Node.js споживав менше 5 % потужності центрального процесора стандартного ноутбука, що свідчить про високу ефективність асинхронної моделі введення-виведення;

2) **використання оперативної пам'яті –** споживання оперативної пам'яті сервером залишалося стабільним на рівні близько 40–50 Мбайт, без ознак витоку пам'яті (Memory Leak) протягом тривалого часу роботи;

3) **синхронізація даних –** дані у всіх 20 відкритих вкладках оновлювалися синхронно, без видимих затримок чи розсинхронізації між різними клієнтами;

4) **робота бази даних –** система управління базою даних SQLite успішно обробляла запис транзакцій у реальному часі без блокування основного потоку передачі даних до клієнтів.

Система продемонструвала високу стабільність роботи. Асинхронна природа платформи Node.js дозволяє стверджувати, що розроблений сервер здатний обслуговувати сотні подібних підключень на стандартному офісному обладнанні без суттєвої деградації продуктивності.

Висновки до розділу 4

В рамках четвертого розділу реалізовано програму емпіричних досліджень створеної платформи. Ключовий фокус було зміщено на верифікацію параметрів продуктивності та оцінку відмовостійкості системи в умовах пікових навантажень. Застосування стратегії тестування «чорної скриньки» дозволило забезпечити об'єктивність експертизи: ми підтвердили відповідність продукту технічному завданню та довели коректність роботи базових алгоритмів без заглиблення у внутрішню структуру коду.

Результати функціонального тестування засвідчили надійність роботи механізмів емуляції фізичних процесів. Зокрема, система продемонструвала адекватну реакцію на динамічне перемикання кліматичних сценаріїв, коректно відтворюючи інерційність зміни показників температури та вологості. Окремо слід відзначити успішну верифікацію математичної моделі взаємовпливу параметрів, де імітація посилення вітру призводила до прогнозованого зниження температурних показників, що підтверджує високий рівень реалістичності розробленої емуляційної моделі. Також було перевірено та підтверджено коректність роботи модуля експорту даних, який забезпечує формування валідних звітів, сумісних із зовнішніми аналітичними інструментами.

Важливим етапом дослідження став порівняльний аналіз ефективності мережевих протоколів. Отримані емпіричні дані наочно довели беззаперечну перевагу обраного протоколу WebSocket над традиційним підходом HTTP Polling

для задач потокової передачі даних у реальному часі. Вимірювання показали, що використання постійного з’єднання дозволило зменшити обсяг службового трафіку на 98,8 %, нівелюючи накладні витрати на заголовки запитів, та забезпечити мінімальну затримку передачі даних на рівні 2–5 мс, що є визначальним фактором для систем оперативного моніторингу.

Окремим досягненням стало підтвердження архітектурної витривалості бекенду на Node.js. Стрес-тести показали, що навіть при агресивному сценарії (масові підключення + інтенсивний запис у БД), система зберігає стабільний профіль споживання ресурсів. Відсутність витоків пам’яті та просадок продуктивності є прямим доказом ефективності неблокуючої моделі введення-виведення, що гарантує легке масштабування системи в майбутньому.

Резюмуючи результати випробувань, можна констатувати повну експлуатаційну готовність системи. Програмний продукт довів свою ефективність і може бути безпосередньо імплементований у навчальний процес або задіяний в R&D проєктах як надійний інструментарій для симуляції складних мережевих топологій.

ВИСНОВКИ

У кваліфікаційній магістерській роботі вирішено актуальне науково-практичне завдання створення системи для емуляції та візуалізації потоків даних Інтернету речей у режимі реального часу. Виконане дослідження дозволило пройти повний цикл розробки – від теоретичного обґрунтування вибору технологій до практичної реалізації та всебічного тестування готового продукту.

Аналіз доменної області виявив фундаментальні вади класичної моделі HTTP Polling: висока латентність та генерація паразитного трафіку роблять її непридатною для задач real-time моніторингу. На противагу цьому підходу, було обґрунтовано перехід до подійно-орієнтованої архітектури на базі WebSocket. Це рішення гарантує персистентне повнодуплексне з'єднання, що є стандартом де-факто для миттєвої синхронізації станів між сервером і клієнтом.

Центральним елементом практичної частини роботи стала розробка та програмна реалізація серверного ядра на платформі Node.js. Ключовим досягненням тут є відмова від спрощених алгоритмів генерації випадкових чисел на користь створення повноцінної математичної моделі віртуальних сенсорів температури, відносної вологості повітря, атмосферного тиску, рівня освітленості та швидкості вітру. Застосування об’єктно-орієнтованого підходу дозволило імплементувати механізми фізичної інерції показників, а також змоделювати складні взаємозв’язки між параметрами навколишнього середовища, зокрема, кореляцію між силою вітру, температурою та вологістю. Введення змінних погодних патернів надало інструментарію необхідної глибини. Це дозволило змоделювати реакцію IoT-мережі на контрастні умови експлуатації: від штатного режиму спокою до екстремальних навантажень, де перевіряється межа витривалості алгоритмів.

Паралельно було створено клієнтську частину у вигляді адаптивного односторінкового вебзастосунку. Використання сучасних засобів візуалізації

на базі бібліотеки Chart.js дозволило реалізувати комбіноване відображення даних через лінійні графіки, кругові діаграми та спеціалізовані віджети. Особливу увагу було приділено ергономіці інтерфейсу та оптимізації продуктивності: впроваджено механізми керування оперативною пам’яттю браузера, підтримку темної теми та інструменти для фільтрації потоків даних. Функціонал системи було розширено модулем експорту звітів, що забезпечує інтероперабельність накопичених даних із зовнішніми аналітичними інструментами.

Експериментальна перевірка розробленого комплексу підтвердила його високу ефективність та надійність. Результати навантажувального тестування засвідчили, що перехід на архітектуру WebSocket дозволив зменшити обсяг службового мережевого трафіку на 98,8 % порівняно з класичними методами, а середня затримка передачі даних склала 2–5 мс у локальній мережі. Система продемонструвала здатність стабільно обслуговувати множинні підключення з мінімальним споживанням системних ресурсів, що підтверджує правильність вибору асинхронної моделі введення-виведення.

Практична цінність проєкту криється у його архітектурній гнучкості. Розроблено не просто навчальний тренажер, а повноцінну платформу для R&D, що дозволяє валідувати архітектурні гіпотези. Ключовою перевагою є підтримка гібридного режиму: здатність системи прозоро об'єднувати віртуальні потоки з даними від реальних контролерів перетворює її на потужний інструмент для поглиблених досліджень сучасного IoT-ландшафту.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Кайданович М. В., Журавська І. М. Емуляція та візуалізація IoT-даних у реальному часі з використанням протоколу WebSocket. *Могилянські читання – 2025* : тези доп. XXVIII Всеукр. наук.-практ. конф., Миколаїв, 10–14 листоп. 2025 р. Миколаїв : Чорном. нац. ун-т ім. Петра Могили, 2025. URL: https://dspace.chmnu.edu.ua/ (дата звернення: 07.12.2025).

2. Enache B. A., Banica C. K., Bogdan Ana G. «Performance Analysis of MQTT Over WebSocket for IoT Applications». *Scientific Bulletin of Electrical Engineering Faculty*, 2023(1):46–49. URL: https://doi.org/10.2478/sbeef-2023-0008 (дата звернення: 24.10.2025).

3. Khattach O., Moussaoui O., Hassine M. «End-to-End Architecture for Real-Time IoT Analytics and Predictive Maintenance Using Stream Processing and ML Pipelines». *Sensors*, 2025; 25(9):2945. URL: https://doi.org/10.3390/s25092945 (дата звернення: 24.10.2025).

4. Lukianets M. O., Sulema Y. S. Візуалізація даних у режимі реального часу для систем мереж IoT: виклики та стратегії оптимізації продуктивності. *System Technologies*, 2023; 5(148):52–61. URL: https://doi.org/10.34185/1562-9945-5-148-2023-05. (дата звернення: 24.10.2025).

5. Amirkhanov B., Adilzhanova S., Kunelbayev M., Tokhtassyn M. Evaluating HTTP, MQTT over TCP and MQTT over WebSocket for Digital Twin Applications: A Comparative Analysis on Latency, Stability, and Integration. *International Journal of Innovative Research and Scientific Studies*, 2025; 8(1):679-694. URL: https://doi.org/10.53894/ijirss.v8i1.4414 . (дата звернення: 24.10.2025).

6. Ortiz G., Boubeta-Puig J., Criado J., Corral-Plaza D., Garcia-de-Prado A., Medina-Bulo I., Iribarne L. A Microservice Architecture for Real-Time IoT Data Processing: A Reusable Web of Things Approach for Smart Ports. *arXiv preprint*, 2024. URL: https://arxiv.org/abs/2401.15390 . (дата звернення: 24.10.2025).

7. Ficili I. Leveraging IoT, Cloud, and Edge Computing with AI». *Sensors*, 2025; 25(6):1763. URL: https://doi.org/10.3390/s25061763. (дата звернення: 24.10.2025).

8. Sembiring A.P., Faza S., Destiadi R. Design and Implementation of IoT Connection With WebSocket Using PHP. *International Journal of Research in Vocational Studies*, 2023; 2(4):94-98. URL: https://doi.org/10.53893/ijrvocas.v2i4.173 . (дата звернення: 24.10.2025).

9. Alami S. Real-Time Energy Monitoring and Anomaly Detection in IoT-Based Systems. 2025. URL: https://www.diva-portal.org/smash/get/diva2:1986256/FULLTEXT01.pdf (дата звернення: 24.10.2025).

10. Andreas R. An In-Depth Guide to IoT Protocols, 2024. URL: https://www.com4.no/en/blog/an-in-depth-guide-to-iot-protocols (дата звернення: 24.10.2025).

11. D3.js Documentation : офіційна документація, 2025. URL: https://d3js.org/ (дата звернення: 24.10.2025).

12. Chart.js Official Documentation : офіційна документація, 2025. URL: https://www.chartjs.org/ (дата звернення: 24.10.2025).

13. Node.js Official Documentation : офіційна документація, 2025. URL: https://nodejs.org/ (дата звернення: 24.10.2025).

14. Nazarenko V. «Weights-based real-time routing platform for digital agriculture in Ukraine». *Grail of Science*, 2025:185-193. URL*:* https://archive.journal-grail.science/index.php/2710-3056/issue/view/17.10.2025/45 (дата звернення: 25.10.2025).

15. Khomenko Y., Babichev S. «Modular IoT Architecture for Monitoring and Control of Office Environments Based on Home Assistant». *MDPI*, 2025. URL: https://www.mdpi.com/2624-831X/6/4/69?utm_source=researchgate.net&utm_medium=article (дата звернення: 25.10.2025).

16. Bondar O. «Graph neural network-based intrusion detection for IoT: performance and comparative analysis». *Grail of Science*, 2025:506-513. URL: https://archive.journal-grail.science/index.php/2710-3056/article/view/3033. (дата звернення: 25.10.2025).

17. Коробейнікова Т., Іськович Т. «Організація архітектури системи IoT в протокольному стекові» *Grail of Science*, 2025:341-346. URL: https://archive.journal-grail.science/index.php/2710-3056/article/view/1260 (дата звернення: 25.10.2025).

18. Zanevych O. «Advancing web development: a comparative analysis of modern frameworks for REST and GraphQL back-end services». *Grail of Science*, 2024. № 37. P. 216–227. URL: https://archive.journal-grail.science/index.php/2710-3056/article/view/2156 (дата звернення: 25.10.2025).

19. Хоменко Є., Бабічев С. «Автономна IoT-система моніторингу мікроклімату аудиторій на основі відкритої DIY-архітектури». *Journals KNTU Kherson*, 2025:245-253. URL: https://journals.kntu.kherson.ua/index.php/ppmm/article/view/961/923 (дата звернення: 25.10.2025).

20. Поперешняк С. «Інтеграція фізичних сенсорів у генерацію псевдовипадкових чисел». *Journals KNTU Kherson*, 2024:177-187. URL: https://journals.kntu.kherson.ua/index.php/ppmm/article/view/760/727 (дата звернення: 25.10.2025).

21. Малюк О., Мартинюк В. «Порівняльний аналіз сенсорів температури». *Measuring and computing devices in technological processes*, 2025:144-151. URL: https://vottp.khmnu.edu.ua/index.php/vottp/article/view/452/437 (дата звернення: 25.10.2025).

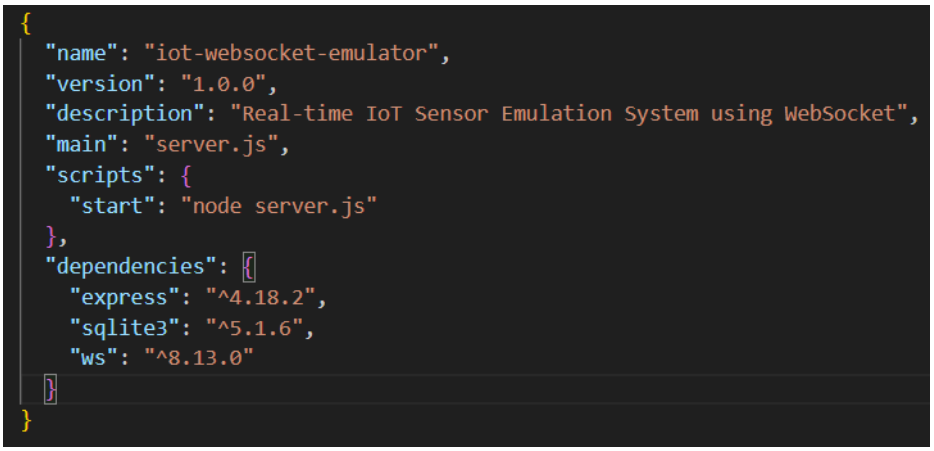
22. Ткачук В., Середюк О. «Концепція фізичного моделювання впливу вологи на роботу термоанемометричних сенсорів». *Methods and devices of quality control*, 2023:31-40. URL: https://mpky.nung.edu.ua/index.php/mpky/article/view/609/592 (дата звернення: 25.10.2025).

23. Лук’янець М. «Проблеми та можливі рішення оптимізації відображення даних мереж IoT у реальному часі». *Grundlagen der modernen wissenschaftlichen Forschung*, 2024:190-191. URL: https://archive.logos-science.com/index.php/conference-proceedings/article/view/1977/2012 (дата звернення: 25.10.2025).

ДОДАТОК А

Код програми

Лістинг файлу package.json наведено на рис. А.1.



```
{
  "name": "iot-websocket-emulator",
  "version": "1.0.0",
  "description": "Real-time IoT Sensor Emulation System using WebSocket",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "sqlite3": "^5.1.6",
    "ws": "^8.13.0"
  }
}
```

Рисунок А.1 – Конфігурація проєкту та залежності

Лістинг А.2 – Файл server.js (Серверна частина: емуляція та обробка даних)

```
const path = require("path");
const express = require("express");
const http = require("http");
const WebSocket = require("ws");
const sqlite3 = require("sqlite3").verbose();

const PORT = 3000;
const app = express();

app.use(express.static(path.join(__dirname, "public")));

const server = http.createServer(app);
const wss = new WebSocket.Server({ server });

// Ініціалізація бази даних SQLite
const db = new sqlite3.Database('./iot_data.db');
db.run(`CREATE TABLE IF NOT EXISTS measurements (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    device_id TEXT, type TEXT, value REAL, unit TEXT, timestamp TEXT
)`);

// API для отримання історії вимірювань
app.get('/api/history', (req, res) => {
    const filterType = req.query.type;
    let sql = `SELECT * FROM measurements ORDER BY id DESC LIMIT 500`;
    let params = [];

    if (filterType && filterType !== 'all') {
        sql = `SELECT * FROM measurements WHERE type = ? ORDER BY id DESC LIMIT
500`;
        params = [filterType];
    }
```

```
    db.all(sql, params, (err, rows) => {
        if (err) {
            res.status(400).json({ "error": err.message });
            return;
        }
        res.json({ "message": "success", "data": rows });
    });
});

// Налаштування сценаріїв емуляції
let isPaused = false;
const SCENARIOS = {
    'normal': { name: 'Кімната (Норма)', temp: [20, 25], hum: [40, 60], press:
[1010, 1015], light: [300, 600] },
    'summer': { name: 'Літо (Спека)',    temp: [30, 38], hum: [20, 40], press:
[1005, 1010], light: [800, 1000] },
    'winter': { name: 'Зима (Мороз)',    temp: [-10, -2], hum: [70, 90], press:
[1020, 1035], light: [50, 200] },
    'autumn': { name: 'Осінь (Дощ)',     temp: [5, 12],  hum: [85, 99], press:
[990, 1000],  light: [100, 300] },
    'sauna':  { name: 'Екстрим (Сауна)', temp: [80, 100], hum: [90, 100], press:
[1000, 1010], light: [50, 150] }
};

const WIND_MODES = { 'none': [0, 0], 'low': [1, 5], 'medium': [6, 12], 'high':
[15, 30] };

// Клас віртуального сенсора з фізичною моделлю інерції
class VirtualSensor {
    constructor(id, type, unit, initialRange) {
        this.id = id; this.type = type; this.unit = unit;
        this.min = initialRange[0]; this.max = initialRange[1];
        this.value = (this.min + this.max) / 2;
    }
    setTargetRange(min, max) { this.min = min; this.max = max; }

    update(windSpeed = 0) {
        let target = (this.min + this.max) / 2;
        // Вплив вітру на температуру та вологість
        if (this.type === 'temperature') target -= windSpeed * 0.3;
        else if (this.type === 'humidity') target -= windSpeed * 0.5;

        const diff = target - this.value;
        let step = diff * 0.05; // Коефіцієнт інерції
        const noise = (Math.random() - 0.5) * (this.max - this.min) * (windSpeed >
10 ? 0.5 : 0.1);
        this.value += step + noise;

        if (this.type === 'humidity' && this.value < 0) this.value = 0;
        if (this.type === 'wind' && this.value < 0) this.value = 0;
        if (this.type === 'light') return Math.round(this.value);
        return parseFloat(this.value.toFixed(2));
    }
}

const mySensors = [
    new VirtualSensor("sensor_wind",  "wind",        "m/s", WIND_MODES['none']),
    new VirtualSensor("sensor_temp",  "temperature", "°C",
SCENARIOS['normal'].temp),
    new VirtualSensor("sensor_hum",   "humidity",    "%",
SCENARIOS['normal'].hum),
```

```
    new VirtualSensor("sensor_press", "pressure",    "hPa",
SCENARIOS['normal'].press),
    new VirtualSensor("sensor_light", "light",       "Lux",
SCENARIOS['normal'].light)
];

// Обробка WebSocket з'єднань
wss.on("connection", (ws) => {
    // Відправка історії при підключенні
    db.all(`SELECT * FROM (SELECT * FROM measurements ORDER BY id DESC LIMIT 100)
ORDER BY id ASC`, [], (err, rows) => {
        if (!err && rows) rows.forEach(row => ws.send(JSON.stringify({ action:
'data', ...row })));
    });

    ws.send(JSON.stringify({ action: 'pause_status', value: isPaused }));

    ws.on("message", (msg) => {
        try {
            const data = JSON.parse(msg);

            if (data.action === 'set_scenario') {
                const config = SCENARIOS[data.value];
                if (config) {
                    mySensors.find(s => s.type ===
'temperature').setTargetRange(config.temp[0], config.temp[1]);
                    mySensors.find(s => s.type ===
'humidity').setTargetRange(config.hum[0], config.hum[1]);
                    mySensors.find(s => s.type ===
'pressure').setTargetRange(config.press[0], config.press[1]);
                    mySensors.find(s => s.type ===
'light').setTargetRange(config.light[0], config.light[1]);
                    broadcast({ action: 'scenario_changed', name: config.name });
                }
            }

            if (data.action === 'set_wind') {
                const range = WIND_MODES[data.value];
                if (range) mySensors.find(s => s.type ===
'wind').setTargetRange(range[0], range[1]);
            }

            if (data.action === 'toggle_pause') {
                isPaused = !isPaused;
                broadcast({ action: 'pause_status', value: isPaused });
            }
        } catch (e) { console.error(e); }
    });
});

function broadcast(data) {
    const json = JSON.stringify(data);
    wss.clients.forEach(client => { if (client.readyState === WebSocket.OPEN)
client.send(json); });
}

function saveAndBroadcast(payload) {
    if (payload.type !== 'wind') {
        db.run(`INSERT INTO measurements (device_id, type, value, unit, timestamp)
VALUES (?, ?, ?, ?, ?)`,
```

```
            [payload.device_id, payload.type, payload.value, payload.unit,
payload.timestamp]);
    }
    payload.action = 'data';
    broadcast(payload);
}

// Головний цикл генерації даних
setInterval(() => {
    if (isPaused) return;
    const now = new Date().toISOString();

    const windSensor = mySensors.find(s => s.type === 'wind');
    const currentWindSpeed = windSensor.update();
    saveAndBroadcast({ device_id: windSensor.id, timestamp: now, type:
windSensor.type, value: currentWindSpeed, unit: windSensor.unit });

    mySensors.forEach(sensor => {
        if (sensor.type !== 'wind') {
            const newValue = sensor.update(currentWindSpeed);
            saveAndBroadcast({ device_id: sensor.id, timestamp: now, type:
sensor.type, value: newValue, unit: sensor.unit });
        }
    });
}, 500);

server.listen(PORT, '0.0.0.0', () => { console.log(`Сервер запущено:
http://localhost:${PORT}`); });
```

Лістинг А.3 – Файл public/index.html (Клієнтська частина: розмітка інтерфейсу)

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>IoT Dashboard Pro</title>
    <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/6.0.0/css/all.min.css">
    <style>
        :root { --bg-color: #f0f2f5; --text-color: #333; --header-bg: #ffffff; --
card-bg: #ffffff; --shadow: 0 2px 10px rgba(0,0,0,0.1); --accent-color: #3498db; -
-bar-bg: #e0e0e0; }
        body.dark-mode { --bg-color: #1a1a2e; --text-color: #e0e0e0; --header-bg:
#16213e; --card-bg: #16213e; --shadow: 0 4px 15px rgba(0,0,0,0.4); --accent-color:
#4cc9f0; --bar-bg: #444; }
        body { font-family: 'Segoe UI', sans-serif; background: var(--bg-color);
color: var(--text-color); margin: 0; transition: 0.3s; }

        /* HEADER */
        .header { height: 60px; background: var(--header-bg); box-shadow: var(--
shadow); display: flex; justify-content: space-between; align-items: center;
padding: 0 30px; position: sticky; top: 0; z-index: 100; }
        .header-left { display: flex; align-items: center; gap: 15px; }
        .header h1 { margin: 0; font-size: 1.4em; }
        .header-status { display: flex; align-items: center; gap: 15px; font-size:
0.9em; font-weight: 600; background: var(--bg-color); padding: 5px 15px; border-
radius: 20px; }
        .online { color: #27ae60; } .offline { color: #c0392b; }
        .header-right { display: flex; align-items: center; gap: 10px; }
```

```
        .icon-btn { background: none; border: none; color: var(--text-color);
font-size: 1.2em; cursor: pointer; padding: 8px; border-radius: 50%; transition:
0.2s; }
        .icon-btn:hover { background: rgba(128,128,128,0.1); }
        /* --- TOOLBAR --- */
        .toolbar {
            display: flex;
            justify-content: space-between;
            align-items: center;
            padding: 20px 0;
            max-width: 1200px; margin: 0 auto;
        }
        .toolbar-left, .toolbar-right {
            display: flex; align-items: center; gap: 15px;
        }
        /* Меню */
        .expanding-menu { position: relative; display: flex; align-items: center;
background: var(--card-bg); border-radius: 30px; box-shadow: var(--shadow);
padding: 10px 15px; transition: all 0.4s cubic-bezier(0.25, 0.8, 0.25, 1);
overflow: hidden; width: 40px; white-space: nowrap; cursor: pointer; z-index: 50;
}
        .menu-icon { font-size: 1.2em; color: var(--text-color); min-width: 30px;
text-align: center; }
        .menu-options { display: flex; gap: 15px; opacity: 0; margin-left: 15px;
transition: opacity 0.3s ease 0.1s; pointer-events: none; }
        .expanding-menu:hover { padding-right: 20px; }
        .expanding-menu:hover .menu-options { opacity: 1; pointer-events: auto; }
        .filter-menu:hover { width: 480px; }
        .scenario-menu:hover { width: 650px; }
        .wind-menu:hover { width: 620px; }
        .menu-item { display: flex; align-items: center; cursor: pointer; user-
select: none; font-weight: 500; font-size: 0.95em; transition: 0.2s; }
        .menu-item input { margin-right: 8px; cursor: pointer; }
        /* Кнопка паузи */
        .pause-btn { background: var(--card-bg); border: none; color: var(--text-
color); font-size: 1.2em; cursor: pointer; padding: 10px; border-radius: 50%; box-
shadow: var(--shadow); width: 45px; height: 45px; display: flex; justify-content:
center; align-items: center; transition: 0.2s; }
        .pause-btn:hover { transform: scale(1.1); }
        .pause-btn.active { color: #f1c40f; }
        /* --- ОБ'ЄДНАНИЙ ВІДЖЕТ ПОГОДИ --- */
        .weather-widget {
            background: var(--card-bg); padding: 8px 25px; border-radius: 30px;
            box-shadow: var(--shadow); display: flex; align-items: center; gap:
15px;
        }
        .current-scenario { display: flex; align-items: center; gap: 10px; font-
weight: bold; color: var(--accent-color); font-size: 0.95em; }
        .vertical-divider { width: 1px; height: 15px; background: var(--text-
color); opacity: 0.2; }
        .current-wind { display: flex; align-items: center; gap: 10px; }
        .mini-bars { display: flex; align-items: flex-end; gap: 3px; height: 20px;
}
        .mini-bar { width: 4px; background: var(--bar-bg); border-radius: 1px;
transition: 0.2s; }
        .mini-bar:nth-child(1) { height: 30%; } .mini-bar:nth-child(2) { height:
50%; } .mini-bar:nth-child(3) { height: 70%; } .mini-bar:nth-child(4) { height:
100%; }
        .mini-bar.active { background: var(--accent-color); } .mini-
bar.active.high { background: #e74c3c; }
```

```
        .wind-text { font-weight: bold; font-size: 0.95em; min-width: 60px; text-
align: right; }
        /* СІТКА */
        .grid-container { display: flex; flex-wrap: wrap; justify-content: center;
gap: 25px; max-width: 1200px; margin: 0 auto 40px auto; padding: 0 20px; }
        .card { background: var(--card-bg); padding: 20px; border-radius: 15px;
box-shadow: var(--shadow); transition: all 0.6s cubic-bezier(0.25, 0.8, 0.25, 1);
opacity: 1; transform: scale(1); max-height: 500px; overflow: hidden; flex: 1 1
45%; min-width: 350px; }
        #card_temperature { flex: 2 1 60%; max-width: 100%; } #card_humidity {
flex: 1 1 30%; max-width: 400px; }
        .card.hidden { opacity: 0; transform: scale(0.8) translateY(20px); flex: 0
0 0 !important; min-width: 0 !important; max-width: 0 !important; padding: 0
!important; margin: 0 !important; max-height: 0 !important; border: none; pointer-
events: none; }
        @media (max-width: 768px) { .card, #card_temperature, #card_humidity {
flex: 1 1 100%; max-width: 100%; } .header h1 { display: none; } .toolbar { flex-
direction: column; } }
        .card-header { display: flex; justify-content: space-between; align-items:
flex-start; margin-bottom: 15px; border-bottom: 1px solid rgba(0,0,0,0.05);
padding-bottom: 10px; }
        .chart-select { background: transparent; border: 1px solid var(--text-
color); color: var(--text-color); padding: 4px 8px; border-radius: 6px; margin-
left: 10px; cursor: pointer; font-size: 0.9em; opacity: 0.8; }
        .value-box { text-align: right; } .value-display { font-size: 2em; font-
weight: bold; }
        .alarm { color: #e74c3c !important; animation: pulse 1s infinite; }
@keyframes pulse { 50% { opacity: 0.5; } }
        .warning-text { font-size: 0.85em; font-weight: bold; color: #e74c3c;
margin-top: 5px; min-height: 20px; opacity: 0; transition: opacity 0.3s; }
.warning-text.visible { opacity: 1; }
        .chart-wrapper { position: relative; height: 200px; width: 100%; display:
flex; justify-content: center; } canvas { max-height: 100% !important; max-width:
100% !important; }
    </style>
</head>
<body>
    <header class="header">
        <div class="header-left">
            <h1>· Система моніторингу IoT</h1>
            <div class="header-status">
                <span id="status" class="offline">● Офлайн</span>
                <span style="opacity: 0.3">|</span>
                <i class="far fa-clock"></i> <span id="last-update-time">--:--:--
</span>
            </div>
        </div>
        <div class="header-right">
            <a href="history.html" class="icon-btn" title="Історія вимірювань"
style="text-decoration: none;">
                <i class="fas fa-table"></i>
            </a>
            <button class="icon-btn" onclick="toggleTheme()" title="Змінити тему">
                <i class="fas fa-moon" id="theme-icon"></i>
            </button>
        </div>
    </header>
    <div style="padding: 0 20px;">

        <div class="toolbar">
```

```
            <div class="toolbar-left">
                <div class="expanding-menu filter-menu">
                    <div class="menu-icon"><i class="fas fa-filter"></i></div>
                    <div class="menu-options">
                        <label class="menu-item"><input type="checkbox" checked
onchange="toggleCard('temperature')"> Темп</label>
                        <label class="menu-item"><input type="checkbox" checked
onchange="toggleCard('humidity')"> Волог</label>
                        <label class="menu-item"><input type="checkbox" checked
onchange="toggleCard('pressure')"> Тиск</label>
                        <label class="menu-item"><input type="checkbox" checked
onchange="toggleCard('light')"> Світло</label>
                    </div>
                </div>
                <div class="expanding-menu scenario-menu">
                    <div class="menu-icon"><i class="fas fa-cloud-sun"></i></div>
                    <div class="menu-options">
                        <label class="menu-item"><input type="radio" name="scen"
checked onclick="changeScenario('normal')"> · Норма</label>
                        <label class="menu-item"><input type="radio" name="scen"
onclick="changeScenario('summer')"> ☀ Літо</label>
                        <label class="menu-item"><input type="radio" name="scen"
onclick="changeScenario('winter')"> ❄ Зима</label>
                        <label class="menu-item"><input type="radio" name="scen"
onclick="changeScenario('autumn')"> · Осінь</label>
                        <label class="menu-item"><input type="radio" name="scen"
onclick="changeScenario('sauna')"> ♨ Сауна</label>
                    </div>
                </div>
                <div class="expanding-menu wind-menu">
                    <div class="menu-icon"><i class="fas fa-wind"></i></div>
                    <div class="menu-options">
                        <label class="menu-item"><input type="radio" name="wind"
checked onclick="changeWind('none')"> ✕ Штиль</label>
                        <label class="menu-item"><input type="radio" name="wind"
onclick="changeWind('low')"> · Слабкий</label>
                        <label class="menu-item"><input type="radio" name="wind"
onclick="changeWind('medium')"> · Помірний</label>
                        <label class="menu-item"><input type="radio" name="wind"
onclick="changeWind('high')"> · Сильний</label>
                    </div>
                </div>
            </div>
            <div class="toolbar-right">

                <div class="weather-widget">
                    <div class="current-scenario" id="scen-indicator">
                        <i class="fas fa-home"></i>
                        <span id="scen-text">Кімната (Норма)</span>
                    </div>
                    <div class="vertical-divider"></div>
                    <div class="current-wind">
                        <div class="mini-bars" id="wind-bars-mini">
                            <div class="mini-bar"></div><div class="mini-
bar"></div><div class="mini-bar"></div><div class="mini-bar"></div>
                        </div>
                        <div id="val_wind" class="wind-text">-- m/s</div>
                    </div>
```

```
                </div>
                <button id="pause-btn" class="pause-btn" onclick="togglePause()"
title="Пауза/Продовжити">
                    <i class="fas fa-pause"></i>
                </button>
            </div>
        </div>
        <div class="grid-container">
            <div class="card" id="card_temperature">
                <div class="card-header">
                    <div><span>· Температура</span> <select class="chart-select"
onchange="switchChartType('temperature', this.value)"><option value="line">Лінія</option><option
value="bar">Стовпчики</option></select></div>
                    <div class="value-box"><div id="val_temperature" class="value-
display">-- °C</div><div id="msg_temperature" class="warning-text"></div></div>
                </div>
                <div class="chart-wrapper"><canvas
id="chart_temperature"></canvas></div>
            </div>
            <div class="card" id="card_humidity">
                <div class="card-header"><div><span>· Вологість</span></div><div
class="value-box"><div id="val_humidity" class="value-display">-- %</div><div
id="msg_humidity" class="warning-text"></div></div></div>
                <div class="chart-wrapper"><canvas
id="chart_humidity"></canvas></div>
            </div>
            <div class="card" id="card_pressure">
                <div class="card-header">
                    <div><span>☁ Тиск</span> <select class="chart-select"
onchange="switchChartType('pressure', this.value)"><option value="line">Лінія</option><option
value="bar">Стовпчики</option></select></div>
                    <div class="value-box"><div id="val_pressure" class="value-
display">-- hPa</div><div id="msg_pressure" class="warning-text"></div></div>
                </div>
                <div class="chart-wrapper"><canvas
id="chart_pressure"></canvas></div>
            </div>
            <div class="card" id="card_light">
                <div class="card-header">
                    <div><span>· Світло</span> <select class="chart-select"
onchange="switchChartType('light', this.value)"><option
value="line">Лінія</option><option value="bar">Стовпчики</option></select></div>
                    <div class="value-box"><div id="val_light" class="value-
display">-- Lux</div><div id="msg_light" class="warning-text"></div></div>
                </div>
                <div class="chart-wrapper"><canvas
id="chart_light"></canvas></div>
            </div>
        </div>
    </div>
    <script src="app.js"></script>
</body>
</html>
```

Лістинг А.4 – Файл public/app.js (Клієнтська частина: логіка візуалізації)

```
const statusSpan = document.getElementById('status');
const timeSpan = document.getElementById('last-update-time');
const scenarioSpan = document.getElementById('current-scenario');
```

```
const scenText = document.getElementById('scen-text');
const scenIcon = document.querySelector('#scen-indicator i');
const pauseBtn = document.getElementById('pause-btn');
const chartDataStorage = {
    temperature: { labels: [], data: [], color: '#e74c3c', label: 'Temp' },
    pressure:    { labels: [], data: [], color: '#27ae60', label: 'Press' },
    light:       { labels: [], data: [], color: '#f1c40f', label: 'Light' },
    wind:        { labels: [], data: [], color: '#607d8b', label: 'Wind' }
};
let activeCharts = {};
let isDarkMode = false;
const SCENARIO_ICONS = {
    'Кімната (Норма)': 'fa-home',
    'Літо (Спека)': 'fa-sun',
    'Зима (Мороз)': 'fa-snowflake',
    'Осінь (Дощ)': 'fa-cloud-showers-heavy',
    'Екстрим (Сауна)': 'fa-hot-tub-person'
};
// --- ЛОГІКА ТЕМИ (Синхронізація) ---
function applyTheme(dark) {
    isDarkMode = dark;
    const icon = document.getElementById('theme-icon');

    if (isDarkMode) {
        document.body.classList.add('dark-mode');
        if(icon) { icon.classList.remove('fa-moon'); icon.classList.add('fa-sun');
}
    } else {
        document.body.classList.remove('dark-mode');
        if(icon) { icon.classList.remove('fa-sun'); icon.classList.add('fa-moon');
}
    }
    // Оновлення графіків
    for (const key in activeCharts) {
        if (key !== 'humidity') initChart(key, activeCharts[key].config.type);
        else {
            const chart = activeCharts['humidity'];
            if(chart && chart.data.datasets.length > 0) {
                chart.data.datasets[0].backgroundColor = ['#3498db', isDarkMode ?
'#2c3e50' : '#ecf0f1'];
                chart.update();
            }
        }
    }
}
window.toggleTheme = function() {
    const newMode = !isDarkMode;
    applyTheme(newMode);
    localStorage.setItem('theme', newMode ? 'dark' : 'light');
};
// Запуск: читаємо тему
if (localStorage.getItem('theme') === 'dark') {
    applyTheme(true);
}
// --- UI ---
window.changeScenario = function(name) { if(ws.readyState===1)
ws.send(JSON.stringify({action:'set_scenario', value:name})); };
window.changeWind = function(val) { if(ws.readyState===1)
ws.send(JSON.stringify({action:'set_wind', value:val})); };
window.togglePause = function() { if(ws.readyState===1)
ws.send(JSON.stringify({action:'toggle_pause'})); };
```

```
window.toggleCard = function(id) {
document.getElementById(`card_${id}`).classList.toggle('hidden'); };
window.switchChartType = function(id, type) { if(id!=='humidity') initChart(id,
type); };
function updateWindWidget(value) {
    const bars = document.querySelectorAll('#wind-bars-mini .mini-bar');
    const activeCount = Math.ceil(value / 7.5);
    bars.forEach((bar, index) => {
        if (index < activeCount) {
            bar.classList.add('active');
            if (value > 15) bar.classList.add('high'); else
bar.classList.remove('high');
        } else { bar.classList.remove('active', 'high'); }
    });
}
// --- ГРАФІКИ ---
function getCommonOptions() {
    const textColor = isDarkMode ? '#e0e0e0' : '#666';
    const gridColor = isDarkMode ? 'rgba(255,255,255,0.1)' : 'rgba(0,0,0,0.05)';
    return {
        responsive: true, maintainAspectRatio: false, animation: false,
        interaction: { intersect: false }, plugins: { legend: { display: false }
},
        scales: { x: { display: false }, y: { ticks: { color: textColor }, grid: {
color: gridColor } } }
    };
}
function createConfig(type, dataStore) {
    return {
        type: type,
        data: {
            labels: dataStore.labels,
            datasets: [{
                label: dataStore.label, data: dataStore.data, borderColor:
dataStore.color,
                backgroundColor: type === 'line' ? dataStore.color + '20' :
dataStore.color + '80',
                borderWidth: 2, fill: true, tension: 0.4, borderRadius: type ===
'bar' ? 5 : 0
            }]
        },
        options: getCommonOptions()
    };
}
function createDoughnutConfig(value) {
    const empty = 100 - value;
    return {
        type: 'doughnut',
        data: {
            labels: ['Вологість', 'Пусто'],
            datasets: [{
                data: [value, empty],
                backgroundColor: ['#3498db', isDarkMode ? '#2c3e50' : '#ecf0f1'],
                borderWidth: 0, hoverOffset: 4
            }]
        },
        options: {
            responsive: true, maintainAspectRatio: false, cutout: '75%',
            plugins: { legend: { display: false }, tooltip: { enabled: false } },
            animation: { animateRotate: false, animateScale: false }
        }
```

```
    };
}
function initChart(sensorKey, type = 'line') {
    const ctx = document.getElementById(`chart_${sensorKey}`);
    if (activeCharts[sensorKey]) activeCharts[sensorKey].destroy();
    if (sensorKey === 'humidity') activeCharts[sensorKey] = new Chart(ctx,
createDoughnutConfig(0));
    else activeCharts[sensorKey] = new Chart(ctx, createConfig(type,
chartDataStorage[sensorKey]));
}
['temperature', 'humidity', 'pressure', 'light'].forEach(key => initChart(key,
'line'));
// --- WEBSOCKET ---
const ws = new WebSocket('ws://localhost:3000');
ws.onopen = () => { statusSpan.innerHTML = '<span style="color:#27ae60">●
Онлайн</span>'; };
ws.onclose = () => { statusSpan.innerHTML = '<span style="color:#c0392b">●
Офлайн</span>'; };
ws.onmessage = (event) => {
    const msg = JSON.parse(event.data);
    if (msg.action === 'pause_status') {
        if (msg.value === true) {
            statusSpan.innerHTML = '<span style="color:#f1c40f">□ Пауза</span>';
            pauseBtn.innerHTML = '<i class="fas fa-play"></i>';
            pauseBtn.classList.add('active');
        } else {
            statusSpan.innerHTML = '<span style="color:#27ae60">● Онлайн</span>';
            pauseBtn.innerHTML = '<i class="fas fa-pause"></i>';
            pauseBtn.classList.remove('active');
        }
        return;
    }
    if (msg.action === 'scenario_changed') {
        if(scenText) scenText.textContent = msg.name;
        if(scenIcon) scenIcon.className = 'fas ' + (SCENARIO_ICONS[msg.name] ||
'fa-home');
        return;
    }
    if (msg.action === 'data' || !msg.action) {
        const type = msg.type;
        const val = msg.value;
        const valEl = document.getElementById(`val_${type}`);
        const msgEl = document.getElementById(`msg_${type}`);

        if (valEl) {
            valEl.textContent = `${val} ${msg.unit}`;

            if (type === 'wind') {
                updateWindWidget(val);
                // Якщо ми тут, значить вітер прийшов, але ми його не малюємо як
графік і він не пишеться в БД
                return;
            }
            let warning = ""; let isAlarm = false;
            if (type === 'temperature') { if (val > 30) { warning = "⚠ Критична
спека!"; isAlarm = true; } else if (val < 10) { warning = "❄ Загроза замерзання!"; isAlarm = true; } }
            if (type === 'humidity') { if (val > 85) { warning = "· Занадто
вологo!"; isAlarm = true; } else if (val < 30) { warning = "· Занадто сухо!";
isAlarm = true; } }
```

```
            if (type === 'pressure') { if (val < 995) { warning = "· Низький тиск";
isAlarm = true; } if (val > 1030) { warning = "· Високий тиск"; isAlarm = true; } }
            if (type === 'light') { if (val > 900) { warning = "✹ Яскраве світло";
isAlarm = true; } if (val < 100) { warning = "· Темрява"; isAlarm = true; } }
            if (isAlarm) { valEl.classList.add('alarm');
if(msgEl){msgEl.textContent=warning; msgEl.classList.add('visible');} }
            else { valEl.classList.remove('alarm');
if(msgEl){msgEl.classList.remove('visible');} }
        }
        if (type === 'humidity') {
            const chart = activeCharts['humidity'];
            if (chart) {
                chart.data.datasets[0].data = [val, 100 - val];
                chart.data.datasets[0].backgroundColor = ['#3498db', isDarkMode ?
'#2c3e50' : '#ecf0f1'];
                chart.update();
            }
        } else if (chartDataStorage[type]) {
            const store = chartDataStorage[type];
            store.labels.push(new Date(msg.timestamp).toLocaleTimeString());
            store.data.push(val);
            if (store.labels.length > 50) { store.labels.shift();
store.data.shift(); }
            if (activeCharts[type]) activeCharts[type].update();
        }
    }
    if(timeSpan) timeSpan.textContent = new Date().toLocaleTimeString();
};
```

Лістинг А.5 – Файл public/history.html (Сторінка перегляду історії та експорту)

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Історія вимірювань | IoT Pro</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/6.0.0/css/all.min.css">
    <style>
        :root { --bg-color: #f0f2f5; --text-color: #333; --card-bg: #ffffff; --
border-color: #ddd; --table-head-bg: #f8f9fa; --hover-bg: rgba(0,0,0,0.02); }
        body.dark-mode { --bg-color: #1a1a2e; --text-color: #e0e0e0; --card-bg:
#16213e; --border-color: #2c3e50; --table-head-bg: #0f3460; --hover-bg:
rgba(255,255,255,0.05); }
        body { font-family: 'Segoe UI', sans-serif; background: var(--bg-color);
color: var(--text-color); padding: 20px; transition: 0.3s; }

        .container { max-width: 1000px; margin: 0 auto; background: var(--card-
bg); padding: 30px; border-radius: 15px; box-shadow: 0 4px 15px rgba(0,0,0,0.1); }
        .header-row { display: flex; justify-content: space-between; align-items:
center; margin-bottom: 20px; flex-wrap: wrap; gap: 15px; }
        h1 { margin: 0; font-size: 1.5em; }
        .back-btn { text-decoration: none; color: var(--text-color); opacity: 0.7;
font-weight: bold; display: flex; align-items: center; gap: 5px; transition: 0.2s;
}
        .back-btn:hover { color: #3498db; opacity: 1; }

        .icon-btn { background: none; border: 1px solid var(--text-color); color:
var(--text-color); font-size: 1em; cursor: pointer; padding: 8px 12px; border-
radius: 20px; transition: 0.2s; }
```

```
        .icon-btn:hover { background: var(--text-color); color: var(--bg-color); }
        .tools-bar { display: flex; justify-content: space-between; align-items:
center; margin-bottom: 20px; flex-wrap: wrap; gap: 15px; }
        .filter-bar { display: flex; gap: 10px; flex-wrap: wrap; }
        .filter-btn { border: none; padding: 8px 16px; border-radius: 20px;
cursor: pointer; font-weight: bold; transition: 0.2s; background: var(--border-
color); color: var(--text-color); opacity: 0.7; }
        .filter-btn:hover { opacity: 1; transform: translateY(-1px); }
        .filter-btn.active { opacity: 1; color: white; box-shadow: 0 2px 5px
rgba(0,0,0,0.2); }
        .filter-btn[data-type="all"].active { background: #555; }
        .filter-btn[data-type="temperature"].active { background: #e74c3c; }
        .filter-btn[data-type="humidity"].active { background: #3498db; }
        .filter-btn[data-type="pressure"].active { background: #27ae60; }
        .filter-btn[data-type="light"].active { background: #f1c40f; color: #333;
}
        .export-btn { background-color: #27ae60; color: white; border: none;
padding: 10px 20px; border-radius: 8px; cursor: pointer; font-weight: bold;
display: flex; align-items: center; gap: 8px; transition: 0.2s; }
        .export-btn:hover { background-color: #219150; transform: translateY(-
2px); box-shadow: 0 4px 10px rgba(39, 174, 96, 0.3); }
        table { width: 100%; border-collapse: collapse; margin-top: 10px; color:
var(--text-color); }
        th, td { padding: 12px; text-align: left; border-bottom: 1px solid var(--
border-color); }
        th { background-color: var(--table-head-bg); font-weight: 600; }
        tr:hover { background-color: var(--hover-bg); }
        .badge { padding: 4px 8px; border-radius: 4px; font-size: 0.85em; font-
weight: bold; color: white; }
        .bg-temp { background-color: #e74c3c; } .bg-hum { background-color:
#3498db; } .bg-press { background-color: #27ae60; } .bg-light { background-color:
#f1c40f; color: #333; }
        .timestamp { opacity: 0.7; font-family: monospace; font-size: 0.95em; }
    </style>
</head>
<body>
    <div class="container">
        <div class="header-row">
            <a href="/" class="back-btn"><i class="fas fa-arrow-left"></i>
Назад</a>
            <h1>· Журнал вимірювань</h1>
            <button class="icon-btn" onclick="toggleThemeHistory()">
                <i class="fas fa-moon" id="theme-icon"></i>
            </button>
        </div>
        <div class="tools-bar">
            <div class="filter-bar">
                <button class="filter-btn active" data-type="all"
onclick="setFilter('all')">Bci</button>
                <button class="filter-btn" data-type="temperature"
onclick="setFilter('temperature')">· Темп</button>
                <button class="filter-btn" data-type="humidity"
onclick="setFilter('humidity')">· Волог</button>
                <button class="filter-btn" data-type="pressure"
onclick="setFilter('pressure')">☁ Тиск</button>
                <button class="filter-btn" data-type="light"
onclick="setFilter('light')">· Світло</button>
            </div>
            <button class="export-btn" onclick="downloadReport()"><i class="fas
fa-file-csv"></i> Завантажити звіт</button>
```

```
        </div>
        <div style="overflow-x: auto;">
            <table id="historyTable">
                <thead>
                    <tr>
                        <th>ID</th>
                        <th>Час</th>
                        <th>Тип сенсора</th>
                        <th>Значення</th>
                        <th>Пристрій</th>
                    </tr>
                </thead>
                <tbody><tr><td colspan="5" style="text-
align:center">Завантаження...</td></tr></tbody>
            </table>
        </div>
    </div>
    <script>
        let currentFilter = 'all';
        let isDarkMode = false;
        let currentData = [];
        function applyTheme(dark) {
            isDarkMode = dark;
            const icon = document.getElementById('theme-icon');
            if (isDarkMode) {
                document.body.classList.add('dark-mode');
                icon.classList.remove('fa-moon'); icon.classList.add('fa-sun');
            } else {
                document.body.classList.remove('dark-mode');
                icon.classList.remove('fa-sun'); icon.classList.add('fa-moon');
            }
        }
        function toggleThemeHistory() {
            const newMode = !isDarkMode;
            applyTheme(newMode);
            localStorage.setItem('theme', newMode ? 'dark' : 'light');
        }
        if (localStorage.getItem('theme') === 'dark') applyTheme(true);
        function setFilter(type) {
            currentFilter = type;
            document.querySelectorAll('.filter-btn').forEach(btn => {
                btn.classList.remove('active');
                if (btn.getAttribute('data-type') === type)
btn.classList.add('active');
            });
            loadHistory();
        }
        async function loadHistory() {
            try {
                const response = await
fetch(`/api/history?type=${currentFilter}`);
                const result = await response.json();
                currentData = result.data;
                const tbody = document.querySelector('#historyTable tbody');
                tbody.innerHTML = '';
                if (result.data.length === 0) {
                    tbody.innerHTML = '<tr><td colspan="5" style="text-
align:center">Записів не знайдено</td></tr>';
                    return;
                }
                result.data.forEach(row => {
```

```
                    const tr = document.createElement('tr');
                    let badgeClass = 'bg-secondary';
                    let label = row.type;
                    if(row.type === 'temperature') { badgeClass = 'bg-temp'; label
= '· Темп'; }
                    else if(row.type === 'humidity') { badgeClass = 'bg-hum';
label = '· Волог'; }
                    else if(row.type === 'pressure') { badgeClass = 'bg-press';
label = '☁ Тиск'; }
                    else if(row.type === 'light') { badgeClass = 'bg-light'; label
= '· Світло'; }
                    const date = new Date(row.timestamp).toLocaleString('uk-UA');
                    tr.innerHTML = `<td>#${row.id}</td><td
class="timestamp">${date}</td><td><span class="badge
${badgeClass}">${label}</span></td><td style="font-weight:bold">${row.value}
${row.unit}</td><td style="font-size:0.9em; opacity:0.7">${row.device_id}</td>`;
                    tbody.appendChild(tr);
                });
            } catch (error) { console.error('Помилка:', error); }
        }
        function downloadReport() {
            if (!currentData || currentData.length === 0) { alert("Немає даних!");
return; }
            let csvContent = "ID;Час;Тип сенсора;Значення;Одиниці;ID Пристрою\n";
            currentData.forEach(row => {
                const date = new Date(row.timestamp).toLocaleString('uk-UA');
                const excelValue = row.value.toString().replace('.', ',');
                const line =
`${row.id};"${date}";${row.type};${excelValue};${row.unit};${row.device_id}`;
                csvContent += line + "\n";
            });
            const bom = "\uFEFF";
            const blob = new Blob([bom + csvContent], { type:
'text/csv;charset=utf-8;' });
            const link = document.createElement("a");
            const url = URL.createObjectURL(blob);
            const now = new Date();
            const fileName = `Zvit_${currentFilter}_${now.getHours()}-
${now.getMinutes()}.csv`;
            link.setAttribute("href", url);
            link.setAttribute("download", fileName);
            link.style.visibility = 'hidden';
            document.body.appendChild(link);
            link.click();
            document.body.removeChild(link);
        }
        loadHistory();
        setInterval(loadHistory, 5000);
    </script>
</body>
</html>
```

ДОДАТОК Б

Матеріали апробації

Апробація результатів магістерської кваліфікаційної роботи відбулась на XXVIII Всеукраїнській щорічній науково-практичній конференції «Могилянські читання – 2025» (Миколаїв, 10–14 листопада 2025 р);

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет імені Петра Могили
ДНУ «Інститут модернізації змісту освіти»
Південний науковий центр НАН та МОН
Інститут української археографії та джерелознавства
імені М. С. Грушевського НАН України
Первинна профспілкова організація ЧНУ імені Петра Могили

ЧНУ

XXVIII ВСЕУКРАЇНСЬКА ЩОРІЧНА НАУКОВО–ПРАКТИЧНА КОНФЕРЕНЦІЯ

«МОГИЛЯНСЬКІ ЧИТАННЯ–2025: досвід та тенденції розвитку суспільства в Україні: глобальний, національний та регіональний аспекти»

ПРОГРАМА

Миколаїв, 10–14 листопада 2025 року

Миколаїв – 2025

ПІДСЕКЦІЯ: КОМП'ЮТЕРНА ІНЖЕНЕРІЯ

Дата та час проведення: 13.11.2025 о 14:00
https://meet.google.com/pub-aamo-oyy

Керівник підсекції: **Дарнапук Є. С. –** PhD, доцент (б. в. з.) кафедри комп'ютерної інженерії

Секретар підсекції: **Худолій Є. П. –** аспірант кафедри комп'ютерної інженерії

Мета проведення: обмін науковими поглядами щодо перспектив розвитку комп'ютерної інженерії в Україні та обговорення перспективних розробок.

1. ***Chuiko G.*** (D.Sc. in Physics and Mathematics, Professor of the Computer Engineering Dep.), ***Yaremchuk O.*** (Senior Lecturer of the Department of Medical and Biological Disciplines), ***Bazhenov D.*** (MS Student, Petro Mohyla Black Sea National University, Mykolaiv, Ukraine). **Feature engineering and noise reduction for cardiovascular risk prediction in Weka.**
2. ***Охотський В.В.*** (магістрант кафедри комп'ютерної інженерії, ЧНУ імені Петра Могили, м. Миколаїв, Україна), ***Нікольський В.В.*** (д-р техн. наук, професор, ЧНУ імені Петра Могили, м. Миколаїв, Україна; Нац. ун-т «Одеська морська академія», м. Одеса, Україна). **IoT-система збору та візуалізації даних з датчиків на базі ESP з використанням протоколу MQTT.**
3. ***Павленко Б.В.*** (магістрант кафедри комп'ютерної інженерії, ЧНУ імені Петра Могили, м. Миколаїв, Україна), ***Нікольський В.В.*** (д-р техн. наук, професор, ЧНУ імені Петра Могили, м. Миколаїв, Україна; Нац. ун-т «Одеська морська академія», м. Одеса, Україна). **Комп'ютерно-інтегрована система поливу тепличного господарства.**
4. ***Тенета Є.В.*** (аспірант), ***Ситніков В.С.*** (д-р техн. наук, проф., завідувач кафедри комп'ютерних систем, Національний університет «Одеська політехніка», м. Одеса, Україна). **Нейронна компенсація інерційних коливань рівня пального з використанням LSTM і навчальних еталонів RAW → EXPECTED.**
5. ***Афоніи Ю.С.*** (аспірант), ***Савінов В.Ю.*** (канд. техн. наук, доцент, доцент кафедри комп'ютерної інженерії, ЧНУ імені Петра Могили, м. Миколаїв, Україна). Розподілена система гуманітарного розмінування з використанням глибокого навчання та комп'ютерного зору.

6. ***Гончаров Д.С.*** (аспірант кафедри комп'ютерної інженерії), ***Кандиба І.О.*** (PhD, ст. викладач кафедри інженерії програмного забезпечення, ЧНУ імені Петра Могили, м. Миколаїв, Україна). Аналіз даних сервісу Google Fit.
7. ***Гріднєв А.Ю.*** (магістрант), ***Дарнапук Є.С.*** (PhD, доцент (б. в. з.) кафедри комп'ютерної інженерії, ЧНУ імені Петра Могили, м. Миколаїв, Україна). **Концепт інформаційно-аналітичної системи для візуалізації даних медичних досліджень.**
8. ***Гюльмамедов Н.М.*** (бакалаврант), ***Бурлаченко І.С.*** (старший викладач кафедри комп'ютерної інженерії, ЧНУ імені Петра Могили, м. Миколаїв, Україна). **Контролери PWM-сигналів для керування сервомоторами у мультиагентних роботизованих системах.**
9. ***Доценко Д.В.*** (магістрант), ***Крайник Я.М.*** (канд. техн. наук, доцент, доцент кафедри комп'ютерної інженерії, ЧНУ імені Петра Могили, м. Миколаїв, Україна). **Реалізація комбінованого методу стиснення проміжних кадрів відео у вебзастосунку.**
10. ***Жуковський Д.С.*** (магістрант), ***Журавська І.М.*** (д-р техн. наук, проф., завідувач кафедри комп'ютерної інженерії, ЧНУ імені Петра Могили, м. Миколаїв, Україна). **IoT-мережа із захистом на основі алгоритмів «легкої криптографії».**
11. ***Завгородній К.С.*** (магістрант), ***Дарнапук Є.С.*** (PhD, доцент (б. в. з.) кафедри комп'ютерної інженерії, ЧНУ імені Петра Могили, м. Миколаїв, Україна). **IoT-система збору та первинної обробки біомедичних показників пацієнтів на базі Raspberry PI та ESP32.**
12. ***Кайданович М.В.*** (магістрант), ***Журавська І.М.*** (д-р техн. наук, професор, завідувач кафедри комп'ютерної інженерії, ЧНУ імені Петра Могили, м. Миколаїв, Україна). **Емуляція та візуалізація IoT-даних у реальному часі з використанням протоколу WebSocket.**
13. ***Мельников А.Г.*** (бакалаврант), ***Салтовський Б.Г.*** (старший викладач кафедри комп'ютерної інженерії, ЧНУ імені Петра Могили, м. Миколаїв, Україна). **Пристрій керування на основі датчика APDS-9960.**
14. **Невідомий Д. О.** (бакалаврант), **Пузирьов С. В.** (канд. фіз.-мат. наук, доцент кафедри комп'ютерної інженерії, ЧНУ імені Петра Могили, м. Миколаїв, Україна). **Система об'єднання відеопотоків у реальному часі на базі Raspberry PI.**
15. **Старченко В. В.** (старший викладач кафедри комп'ютерної інженерії, ЧНУ імені Петра Могили, м. Миколаїв, Україна). **Генерація фрактальних зображень з заданими просторово-**

ДОДАТОК В

Перевірка на унікальність

Звіт про перевірку схожості тексту Identific

Назва документа:
Кайданович_605_Кваліф_Робота_2025_v4-ДляАнтиплаг.docx

Ким подано: **Ірина Журавська**	Дата перевірки: **2025-12-07 22:54:18**	Дата звіту: **2025-12-07 23:05:24**
Ким перевірено: **I + U + DB + P + DOI**	Кількість сторінок: **71**	Кількість слів: **14362**

Схожість 4%	Збіг: **50 джерела**	Вилучено: **0 джерела**
Інтернет: **20 джерела**	DOI: **0 джерела**	База даних: **0 джерела**
Перефразовування 1%	Кількість: **37 джерела**	Перефразовано: **883 слова**
Цитування 0%	Цитування: **3**	Всього використано слів: **43**
Включення 0%	Кількість: **0 включення**	Всього використано слів: **0**
Питання 1%	Замінені символи: **0**	Інший сценарій: **198 слова**