Міністерство освіти і науки України
Кам’янець-Подільський національний університет імені Івана Огієнка
Фізико-математичний факультет
Кафедра комп’ютерних наук

Кваліфікаційна робота бакалавра
з теми: «**Розробка архітектури хмарного сервісу для збору та обробки даних від IoT-пристроїв**»

Виконав: здобувач вищої освіти
групи KN1-B21
спеціальності 122 Комп’ютерні науки
Костинюк Владислав Вікторович
Керівник:
Слободянюк Олександр Васильович
доцент кафедри комп’ютерних наук

Рецензент:_____

Кам’янець-Подільський – 2025 р.

АНОТАЦІЯ

Кваліфікаційна робота присвячена розробці хмарного вебсервісу для збору, зберігання, обробки та візуалізації даних, отриманих від IoT-пристроїв. У межах проєкту реалізовано повнофункціональну систему, яка охоплює всі етапи: від надсилання телеметричних даних сенсорами до їх відображення через вебінтерфейс. Розробка базується на використанні мікрофреймворку Flask, хмарного хостингу Render та бази даних MongoDB Atlas.

Застосунок підтримує приймання та обробку даних у реальному часі, формування звітів у форматі PDF, а також надсилання електронних нагадувань про критичні події. Особливу увагу приділено розгортанню системи в хмарному середовищі, що забезпечує доступність сервісу з будь-якого пристрою, підключеного до Інтернету.

Результати роботи демонструють доцільність використання відкритих вебтехнологій для створення масштабованих IoT-рішень. Розроблена система може бути адаптована для задач моніторингу довкілля, агросфери або «розумного дому», а також слугувати навчальним прикладом в освітніх курсах із вебпрограмування, IoT та хмарних технологій.

Ключові слова: IoT, вебсервіс, Flask, MongoDB, хмарні технології, PDF-звіт, сенсорні дані, автоматизація.

ABSTRACT

The qualification thesis is dedicated to the development of a cloud-based web service for collecting, storing, processing, and visualizing data received from IoT devices. The implemented system provides a full data cycle: from sensor-generated telemetry to its display in a user-friendly web interface. The application is built using the Flask microframework, hosted on Render, and integrated with MongoDB Atlas as the cloud database.

The solution supports real-time data processing, PDF report generation, and automated email notifications for critical events. Special attention is given to deploying the system in a cloud environment, ensuring high availability and accessibility from any Internet-connected device.

The results confirm the feasibility of using open-source web technologies to build scalable IoT solutions. The developed system can be adapted for environmental monitoring, smart agriculture, or smart home projects, and may also serve as a teaching tool in courses on web development, IoT, and cloud computing.

Keywords: IoT, web service, Flask, MongoDB, cloud technologies, PDF report, sensor data, automation.

ЗМІСТ

ВСТУП

Сучасна цифрова епоха характеризується стрімким розвитком технологій Інтернету речей (Internet of Things, IoT), які дозволяють автоматизовано збирати, передавати та аналізувати дані з фізичного середовища. Рішення на базі IoT усе ширше впроваджуються у сільське господарство, екологічний моніторинг, міську інфраструктуру, промисловість. У цьому контексті важливим завданням є створення таких архітектур систем, які дозволяють не лише збирати дані від сенсорів, а й ефективно їх обробляти, зберігати та візуалізувати у зручному форматі для користувача.

Особливу роль у цьому відіграють хмарні сервіси, які забезпечують доступність, масштабованість і гнучкість у розгортанні IoT-систем. Використання хмарних платформ дозволяє обробляти дані в режимі реального часу, забезпечувати централізований доступ і мінімізувати локальні ресурси для обробки інформації.

У межах кваліфікаційної роботи розроблено веборієнтовану систему, що дозволяє емулювати роботу IoT-пристрою, збирати телеметричні дані (температура, вологість), передавати їх на сервер через API, зберігати у хмарній базі даних та виводити у вебінтерфейсі. Реалізовано функціонал формування PDF-звітів, нагадувань про події, а також розгортання сервісу на платформі Render. Ознайомитися з результатом роботи можна за посиланнями:

- вебінтерфейс: https://iot-project-zy46.onrender.com
- вихідний код: https://github.com/VladislavKostyniuk/iot-project

Метою роботи є створення хмарного вебсервісу, що дозволяє приймати та обробляти дані від IoT-пристроїв, з використанням відкритого технологічного стеку (Python, Flask, MongoDB, Render).

Для досягнення цієї мети були визначені такі завдання:

- проаналізувати підходи до побудови архітектури IoT-систем;
- обґрунтувати вибір інструментів (Flask, MongoDB, хмарний хостинг);
- реалізувати серверну частину з API для приймання телеметрії;
- створити вебінтерфейс для візуалізації даних;
- реалізувати функціональність генерації PDF-звітів та сповіщень;
- розгорнути готову систему у хмарному середовищі та протестувати її роботу.

Об’єктом дослідження є архітектура систем з IoT-пристроями та хмарною обробкою даних.
Предметом дослідження — вебтехнології та інструменти побудови хмарних сервісів для обслуговування IoT-інфраструктури.

Методологія реалізації проєкту включала системний аналіз, моделювання потоків даних, застосування принципів REST-архітектури, об’єктно-орієнтованого підходу до програмування, а також тестування продуктивності й коректності роботи сервісу.

Структура кваліфікаційної роботи охоплює три розділи. Перший розділ присвячено теоретичним основам побудови IoT-архітектур. У другому розділі здійснено проєктування архітектури системи, визначення її компонентів та вибір технологій. У третьому розділі

представлено реалізацію, розгортання сервісу, результати тестування та аналіз роботи системи.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Сучасні підходи до побудови IoT-систем

Інтернет речей (Internet of Things, IoT) — це концепція побудови інформаційних систем, у яких фізичні пристрої, сенсори, виконавчі механізми та інші елементи навколишнього середовища здатні взаємодіяти між собою, збирати, передавати та аналізувати дані в автоматизованому режимі. IoT системи дедалі активніше проникають у різні сфери життя — від «розумного дому» й медицини до промисловості, логістики й аграрного виробництва. Центральне місце в таких системах займає обмін даними, який здійснюється через мережу з метою оперативного реагування, накопичення інформації та оптимізації бізнес-процесів.

Залежно від особливостей сфери застосування, типу даних і технічних обмежень, архітектура IoT-систем може будуватись на основі різних підходів. Серед сучасних моделей найпоширенішими є централізована, хмарна, периферійна (edge computing) та т.зв. туманна архітектура (fog computing). Кожен із цих підходів має свої переваги, недоліки та сфери оптимального застосування.

Централізована архітектура передбачає, що всі пристрої надсилають дані до єдиного центрального вузла — сервера або контролера, який виконує всю логіку обробки. Такий підхід простий у реалізації й підтримці, однак не завжди придатний до великих і розподілених систем. Основним недоліком централізованих рішень є їх слабка масштабованість, високе навантаження на центральний вузол та ризик виникнення єдиної точки відмови. У разі збоїв у центрі вся система може втратити функціональність.

Більш адаптивним варіантом виступає хмарна архітектура, яка набула широкого поширення завдяки появі спеціалізованих хмарних платформ, здатних обробляти мільйони повідомлень від IoT-пристроїв у реальному часі. У цій моделі пристрої підключаються до інтернету, передають дані на хмарні сервіси, де відбувається їх зберігання, обробка, агрегація та аналіз. Водночас користувачі можуть взаємодіяти з системою через вебінтерфейси, мобільні додатки або API. Основними перевагами такого підходу є масштабованість, гнучкість, централізоване управління та готова інфраструктура для авторизації, маршрутизації даних, балансування навантаження й створення резервних копій. Серед популярних рішень варто відзначити AWS IoT Core, Azure IoT Hub, Google Cloud IoT та сучасні PaaS-платформи як-от Render і Railway, які спрощують процес розгортання MVP-рішень.

Однак у певних випадках — особливо коли потрібна обробка даних із мінімальними затримками або існує ризик нестабільного інтернет-з’єднання — більш доречним є застосування edge-архітектури (edge computing). Суть цього підходу полягає в тому, що обробка даних здійснюється не в хмарі, а безпосередньо на самих пристроях або у проміжних вузлах, розташованих поблизу джерела даних. Це дає змогу знизити навантаження на мережу, забезпечити автономну роботу системи та мінімізувати час відгуку. Edge-підхід активно застосовується у транспортних системах, медицині, промислових установках та сценаріях, де критично важлива швидкість реакції. Типовим прикладом є використання Raspberry Pi, Jetson Nano або вбудованих контролерів із вбудованим програмним забезпеченням.

Проміжною ланкою між хмарою та периферією є fog computing — туманна архітектура, яка забезпечує локальну обробку даних на

рівні шлюзів, маршрутизаторів або міні-серверів. Основна ідея полягає в тому, що необроблені сирі дані не передаються в хмару повністю, а фільтруються або агрегуються безпосередньо поблизу пристроїв. Це зменшує мережеве навантаження, підвищує швидкість реакції та дозволяє гнучко комбінувати хмарні й локальні обчислення. Fog-підхід часто реалізується у великих індустріальних системах, «розумних містах» або системах моніторингу інфраструктури.

У контексті реалізації хмарного IoT-сервісу для обробки даних від пристроїв, який розглядається в цій кваліфікаційній роботі, доцільним є вибір хмарної архітектури. Такий підхід відповідає ключовим вимогам до гнучкості, масштабованості, зручності розгортання та доступності для користувачів. Хмарне середовище надає готову інфраструктуру для обробки запитів, зберігання даних і реалізації вебінтерфейсу, що дозволяє сконцентруватися на розробці функціоналу, а не на технічних деталях інфраструктури. У даному проєкті як хостинг-платформу обрано сервіс Render, що підтримує безперервне розгортання, масштабування та моніторинг серверних застосунків. Таким чином, хмарна модель забезпечує ефективну інтеграцію IoT-пристроїв, серверної логіки та візуалізації даних, що цілком відповідає цілям і завданням дослідження.

1.2. Основні компоненти IoT-архітектури

Архітектура системи Інтернету речей (IoT) передбачає побудову складної багаторівневої інфраструктури, яка охоплює фізичні пристрої, мережеві компоненти, програмні засоби, хмарні сервіси та бази даних. Кожен рівень виконує окрему функцію, забезпечуючи безперервний потік даних — від моменту їх збору в середовищі до аналітичної обробки й подання користувачу. Розуміння ролі та взаємозв’язку

компонентів є ключовим етапом проєктування ефективного IoT-рішення.

Пристрої збору даних (сенсори, контролери) — це фізичні елементи, що безпосередньо взаємодіють із навколишнім середовищем. Вони реєструють параметри, як-от температура, вологість, освітленість, тиск, рівень шуму тощо. У контексті даного проєкту можуть застосовуватись емулятори пристроїв, написані на Python, які генерують випадкові або змодельовані дані для імітації роботи реального сенсора. З технічної точки зору, пристрій повинен мати мінімальну обчислювальну потужність, стабільне мережеве з'єднання та підтримку протоколу передачі даних (наприклад, HTTP або MQTT).

Шлюзи (gateway) виступають проміжною ланкою між пристроями збору даних і хмарними платформами. Вони можуть фільтрувати, агрегувати, шифрувати або кешувати дані перед відправленням у хмару. У простих реалізаціях шлюз може бути замінений безпосереднім підключенням пристрою до API сервера, що актуально для невеликих проєктів із незначним трафіком. Проте в більш масштабних рішеннях шлюзи забезпечують локальну обробку (edge-функціональність), балансування навантаження та підвищення стійкості до збоїв.

Хмарні сервіси реалізують основну обчислювальну логіку IoT-системи. Вони приймають дані через API, зберігають їх у базі даних, виконують попередню обробку, розрахунки, перевірку допустимих меж, формування звітів тощо. У даному проєкті хмарний сервер створено з використанням фреймворку Flask, який забезпечує зручний механізм створення RESTful API та легко інтегрується з іншими компонентами, зокрема з MongoDB.

API (Application Programming Interface) — інтерфейс прикладного програмування — надає уніфікований спосіб взаємодії між пристроями, клієнтськими застосунками та серверною логікою. API-ендпоінти дозволяють надсилати дані (через POST-запити), отримувати останні вимірювання (GET-запити), автентифікувати користувачів або запускати певні сервіси. Важливою вимогою до API є стабільність, документованість, обробка помилок і захист від несанкціонованого доступу.

База даних (БД) відповідає за зберігання всієї інформації, що надходить від пристроїв або генерується в результаті обробки. Для зберігання структурованих і напівструктурованих даних в межах даного проєкту використано MongoDB — документоорієнтовану NoSQL базу, яка добре пристосована до динамічних структур даних і високої частоти записів. MongoDB дозволяє зберігати записи у форматі BSON (аналог JSON), що спрощує серіалізацію й обробку в мовах програмування, зокрема Python.

Уся взаємодія між компонентами IoT-архітектури базується на принципі децентралізації: дані передаються від сенсорів до API, обробляються сервером, зберігаються в базі та повертаються у вигляді звітів або візуалізацій. Завдяки цьому досягається висока гнучкість системи, можливість масштабування та незалежність компонентів. У проєкті реалізовано просту, але ефективну архітектуру, яка охоплює всі ці рівні, дозволяючи виконувати повний цикл: генерація → передача → обробка → збереження → відображення.

1.3 Огляд протоколів обміну даними в IoT

Протоколи обміну даними відіграють ключову роль у функціонуванні IoT-систем, оскільки саме вони забезпечують

комунікацію між пристроями, шлюзами, хмарними сервісами та користувацькими інтерфейсами. Вибір протоколу залежить від багатьох чинників: енергоспоживання, надійність зв’язку, частота обміну повідомленнями, обсяг даних і складність реалізації. У цьому підрозділі розглянуто найпоширеніші протоколи, що застосовуються в сучасних IoT-системах: MQTT, HTTP/HTTPS та CoAP.

MQTT (Message Queuing Telemetry Transport) — легковаговий публікаційно-підписний протокол, спеціально розроблений для передачі телеметричних даних у середовищах з обмеженими ресурсами. MQTT ґрунтується на принципі брокера повідомлень: пристрої (видавці) надсилають дані на брокер, а інші пристрої або сервіси (передплатники) отримують ці дані, підписавшись на відповідну тему. Основні переваги MQTT полягають у мінімальному використанні трафіку, ефективній підтримці слабких або нестабільних з’єднань, а також у можливості налаштування рівнів надійності доставки повідомлень. Недоліками протоколу є необхідність налаштування окремого MQTT-брокера, обмежена функціональність порівняно з HTTP і складність масштабування у великих інфраструктурах. MQTT чудово підходить для енергоощадних пристроїв, які періодично надсилають невеликі обсяги даних — наприклад, сенсори температури чи вологості.

HTTP/HTTPS (HyperText Transfer Protocol Secure) — універсальний протокол обміну гіпертекстом, який широко використовується не лише в IoT, а й у веброзробці загалом. У

контексті IoT він дозволяє пристроям надсилати дані безпосередньо на сервер через API-запити, зокрема метод POST для передачі показників. Переваги HTTP/HTTPS включають простоту реалізації, широке розповсюдження і підтримку, сумісність із більшістю хмарних платформ, а також можливість використання стандартних інструментів. Недоліками є більший обсяг заголовків порівняно з MQTT чи CoAP, підвищене енергоспоживання через TCP-з'єднання та відсутність механізму збереження сесії. У межах цієї дипломної роботи використано саме HTTP-протокол для надсилання даних з емулятора на сервер, що забезпечує простоту тестування та сумісність з обраною серверною платформою.

CoAP (Constrained Application Protocol) — UDP-орієнтований протокол, призначений для обміну повідомленнями між пристроями в умовах обмежених обчислювальних і енергетичних ресурсів. На відміну від HTTP, CoAP використовує значно легший стек протоколів, що знижує навантаження на мережу та забезпечує високу швидкість передавання. Його перевагами є компактність повідомлень, підтримка multicast-запитів, можливість реалізації REST-подібної моделі та швидкий обмін завдяки використанню UDP. Водночас CoAP має і свої недоліки: менш поширений, ніж MQTT і HTTP, потребує спеціалізованих бібліотек, має складнощі із забезпеченням надійності доставки. CoAP найкраще підходить для великих сенсорних мереж, де критичним є зменшення енергоспоживання та швидкість реакції, однак рідко використовується у прототипуванні через ускладнення налаштувань.

Загалом, вибір протоколу залежить від архітектури IoT-системи, типу пристроїв, потреб у безпеці, затримках та обсягах даних. У дипломній роботі використано HTTP як найбільш доступний для розробки, тестування та інтеграції з вебфреймворками та хмарними платформами.

1.4 Хмарні платформи для обробки IoT-даних

З поширенням Інтернету речей виникає потреба в інструментах, які дозволяють не лише збирати, а й ефективно зберігати, аналізувати та візуалізувати отримані дані. Одним із найважливіших компонентів сучасної IoT-екосистеми є хмарні платформи, які виступають центральною точкою обробки потоків інформації від пристроїв. Їх використання дозволяє зменшити навантаження на локальні ресурси, забезпечити масштабованість рішень і скоротити час виходу продукту на ринок.

Серед провідних рішень у цій сфері варто виділити AWS IoT Core, Azure IoT Hub, Google Cloud IoT, а також більш прості хостингові сервіси на кшталт Render і Railway, які часто використовуються в навчальних або прототипних проєктах.

AWS IoT Core — сервіс від Amazon Web Services, який надає широкий спектр можливостей для з'єднання, захисту, обробки та управління IoT-пристроями в реальному часі. Основними перевагами платформи є її масштабованість, підтримка MQTT та HTTPS, розвинені можливості аналітики (AWS IoT Analytics), інтеграція з іншими хмарними сервісами AWS (наприклад, Lambda, S3, DynamoDB) та високий рівень безпеки. Водночас, складність налаштувань і висока вартість послуг можуть бути бар'єром для малих команд або студентських проєктів.

Azure IoT Hub — рішення від Microsoft, яке дозволяє з'єднувати мільйони пристроїв з хмарою, організовувати двосторонній зв’язок і керувати пристроями. Azure підтримує як MQTT, так і AMQP, HTTP і надає зручні засоби для моніторингу стану пристроїв, а також управління оновленнями програмного забезпечення. Платформа інтегрується з іншими сервісами Microsoft, зокрема з Power BI для побудови аналітичних панелей. Як і у випадку з AWS, складність використання та ціна можуть бути обмежувальними факторами.

Google Cloud IoT Core — сервіс для збирання та аналізу даних від пристроїв через MQTT або HTTP, з подальшою інтеграцією з BigQuery, Cloud Functions або Data Studio. Цей сервіс відзначається простотою використання та гнучкістю, але, починаючи з 2023 року, Google оголосив про поступове згортання підтримки IoT Core. Це стало вагомим аргументом для пошуку альтернативних платформ у довгостроковій перспективі.

Render — сучасна платформа хостингу, яка дозволяє розгортати вебзастосунки, API та бази даних із мінімальними витратами часу. Render не є спеціалізованою IoT-платформою, але чудово підходить для хостингу серверної логіки (наприклад, Flask API), що приймає дані від IoT-пристроїв. Основними перевагами є легкість налаштування, безкоштовний тариф для невеликих проєктів, автоматичний деплой з GitHub та вбудований SSL. Саме Render був обраний для розгортання серверної частини в межах даного дипломного проєкту.

Railway — ще одна хмарна платформа для швидкого розгортання серверів і баз даних. Railway забезпечує зручне середовище для безперервної інтеграції (CI/CD), підтримує більшість сучасних мов програмування та дозволяє просто керувати змінними середовища.

Railway часто використовується як альтернатива Render для проєктів із відкритим кодом і швидким циклом розробки, хоча її можливості для масштабування все ще поступаються великим хмарним екосистемам.

Таким чином, при виборі хмарної платформи для IoT-проєкту важливо враховувати ціль проєкту (прототипування, комерційне рішення, навчальний проєкт), доступний бюджет, вимоги до масштабованості та зручність інтеграції з іншими сервісами. Для мети цієї кваліфікаційної роботи ідеальним варіантом став Render як легка, доступна і досить гнучка платформа для розгортання серверної логіки й обробки даних від IoT-пристроїв.

1.5 Приклади реалізації простих IoT-проєктів

На сьогодні існує велика кількість прикладів реалізації простих IoT-систем, які демонструють базові принципи функціонування мережі пристроїв, взаємодії з хмарними сервісами та відображення отриманих даних. Такі проєкти часто використовуються в освітньому середовищі або в процесі створення прототипів майбутніх промислових рішень.

Одним із найпоширеніших прикладів є система моніторингу температури та вологості повітря на основі датчиків DHT11 або DHT22. Такі датчики підключаються до мікроконтролерів на кшталт Arduino чи ESP8266, які періодично зчитують дані та надсилають їх через Wi-Fi до віддаленого сервера або хмарної платформи. Найчастіше для цього використовуються легкі протоколи обміну — MQTT або HTTP. Отримані дані можуть зберігатися в базі даних (наприклад, MongoDB або InfluxDB) і відображатися у вигляді графіків у вебінтерфейсі або дашборді на базі Grafana.

Інший приклад — проєкти з моніторингу ґрунтової вологи. У цьому випадку використовується вологоємний сенсор (наприклад, YL-

69), який вимірює електропровідність ґрунту та передає дані до контролера. Це дозволяє автоматично оцінювати рівень вологості та, за потреби, ініціювати полив. Такі системи можуть працювати автономно або інтегруватися з хмарними сервісами для віддаленого керування, історії змін або надсилання повідомлень на телефон користувача.

Популярними також є демонстраційні проєкти з побудови “розумного дому” (smart home), які включають керування освітленням, температурою, сигналізацією або камерами спостереження через вебінтерфейс або мобільний застосунок. У цьому випадку часто застосовується платформа Blynk або Node-RED, які дозволяють будувати IoT-логіку без написання складного коду.

У рамках даної кваліфікаційної роботи реалізовано навчальний проєкт з використанням емулятора IoT-пристрою, який генерує випадкові сенсорні значення (наприклад, температуру та вологість) і надсилає їх до хмарного серверу через HTTP-запити. Серверна частина побудована на Flask і зберігає отримані значення в базі даних MongoDB, розгорнутій на хостингу Render. Отримані дані виводяться у вебінтерфейсі, що дозволяє наочно відстежувати оновлення в режимі реального часу.

Особливість цієї реалізації полягає в її простоті та доступності для розуміння. Вона не потребує фізичного обладнання, оскільки пристрій емулюється програмно, що дозволяє зосередитися на логіці передачі, обробки та візуалізації даних. Такий підхід зручний для тестування архітектурних рішень, оптимізації структури API і підготовки до масштабування.

Таким чином, навіть прості IoT-проєкти можуть ефективно демонструвати ключові елементи екосистеми Інтернету речей,

включаючи сенсори, контролери, мережеві протоколи, хмарні платформи та вебінтерфейси. Вони формують основу для подальшої розробки більш складних систем автоматизації, моніторингу або аналітики даних у різних галузях — від сільського господарства до міської інфраструктури.

1.6 Висновки до розділу

У результаті аналізу предметної області, пов’язаної з побудовою IoT-систем та архітектурою хмарних сервісів для збору й обробки даних, було сформовано низку концептуальних висновків, що визначають основу подальшого проєктування та реалізації кваліфікаційної роботи.

По-перше, розглянуті сучасні підходи до побудови архітектури IoT-рішень продемонстрували значне розмаїття моделей — від централізованих до edge- і fog-computing-архітектур. У кожної з них є свої переваги, які визначаються специфікою задач, обсягом даних, затримками в обробці та вимогами до автономності пристроїв. У контексті проєкту оптимальним є хмарний підхід, що забезпечує масштабованість і доступність з будь-якої точки з доступом до мережі.

По-друге, було окреслено основні компоненти типових IoT-систем: сенсорні пристрої, мікроконтролери, шлюзи, канали передачі даних, хмарні платформи, API і бази даних. Визначення ролі кожного компонента дозволяє зрозуміти послідовність обробки інформації — від моменту зчитування з фізичного сенсора до її збереження і відображення користувачеві.

По-третє, аналіз протоколів передачі даних, таких як MQTT, CoAP, HTTP/HTTPS, дозволив встановити їхні переваги та недоліки. Зокрема, у випадку простих навчальних проєктів найбільш доречним є

використання HTTP як універсального та легко реалізованого протоколу, сумісного з більшістю фреймворків веброзробки.

По-четверте, досліджено функціонал провідних хмарних платформ для роботи з IoT-даними, таких як AWS IoT Core, Azure IoT Hub, Google Cloud IoT, Render та Railway. Незважаючи на широкі можливості комерційних сервісів, для освітніх і малих проєктів оптимальним рішенням є використання платформ, які пропонують безкоштовні тарифи або простоту розгортання, як-от Render.

По-п’яте, було проаналізовано приклади простих IoT-проєктів, які демонструють реальні сценарії застосування технологій Інтернету речей: моніторинг температури, вологи, керування освітленням тощо. Це дозволило окреслити практичну значущість IoT у повсякденному житті й промисловості, а також визначити базову структуру реалізації майбутньої системи.

Таким чином, на основі теоретичного аналізу сформовано методологічну базу для подальшого проєктування та реалізації хмарного вебсервісу для збору і обробки даних від IoT-пристроїв. У наступному розділі буде представлено технічне завдання, обґрунтування вибору технологій, побудова архітектури системи та опис логіки взаємодії її компонентів.

РОЗДІЛ 2. ПРОЄКТУВАННЯ СИСТЕМИ

2.1 Формулювання технічного завдання

У межах реалізації кваліфікаційної роботи сформульовано технічне завдання на розробку хмарного сервісу, орієнтованого на приймання, обробку та зберігання даних, отриманих від пристроїв Інтернету речей, із подальшою їх візуалізацією через вебінтерфейс. Актуальність такого рішення зумовлена зростанням кількості IoT-пристроїв, що генерують великі обсяги телеметричних даних, які потребують автоматизованої обробки, зберігання, аналізу та подання в доступній формі кінцевим користувачам. Особливу увагу приділено сумісності системи з хмарною інфраструктурою, що забезпечує високу доступність, масштабованість і відмовостійкість сервісу без прив’язки до конкретного фізичного обладнання.

Система має забезпечити стабільний канал зв’язку між IoT-пристроєм та сервером через HTTP POST-запити з передачею структурованих повідомлень у форматі JSON. На стороні сервера необхідно реалізувати валідацію отриманих запитів, яка включає перевірку структури, коректності форматів, наявності обов’язкових полів та відповідності типів даних. У разі виявлення помилок система має повертати інформативні повідомлення для зручної діагностики.

Зберігання даних буде реалізовано за допомогою MongoDB, оскільки ця база даних дозволяє зберігати неструктуровану або слабо структуровану інформацію у гнучкому форматі BSON. Такий підхід добре підходить для IoT-застосунків, де структура даних може змінюватися або варіюватися залежно від типу пристрою.

Важливою частиною системи є API, що забезпечуватиме доступ до збережених даних, зокрема до останніх отриманих значень або до

історичних записів. Доступ до API повинен бути захищеним. На початковому етапі достатньо використання простого механізму авторизації або API-ключа для запобігання несанкціонованому доступу.

Користувацький інтерфейс вебзастосунку має бути максимально простим та інтуїтивно зрозумілим. Основна його функція — відображення останніх отриманих даних у текстовому або графічному форматі. Інтерфейс повинен бути доступним через браузер з будь-якого пристрою, що забезпечить зручність використання у різних умовах.

Система має бути розгорнута у хмарному середовищі з підтримкою Python та MongoDB, зокрема на платформі Render або Railway. Хостинг повинен забезпечувати постійний доступ до сервера, підтримку вхідних HTTP-запитів, а також стабільне підключення до бази даних. Крім того, має бути реалізований базовий моніторинг працездатності системи та журналювання подій.

Проєкт також має передбачати можливість масштабування як на рівні серверної обробки, так і з боку бази даних. Це важливо для врахування майбутнього зростання обсягу даних або кількості підключених пристроїв.

Формулювання технічного завдання охоплює як функціональні, так і нефункціональні вимоги до системи. Враховуючи зазначене, запропонована архітектура дозволяє створити ефективне, стабільне та гнучке програмне рішення для збирання та обробки IoT-даних у хмарному середовищі.

2.2. Обґрунтування вибору інструментів

Для реалізації хмарного сервісу збору та обробки IoT-даних було обрано набір інструментів, які забезпечують простоту розгортання,

масштабованість, підтримку сучасних стандартів передачі даних і активну спільноту розробників. Основними критеріями вибору були відкритість ліцензій, сумісність між компонентами, продуктивність у реальному часі, підтримка JSON-форматів, а також наявність документації та прикладів застосування в аналогічних проєктах.

На етапі емуляції IoT-пристрою було вирішено використати мову програмування Python, яка має розвинену екосистему бібліотек для генерації псевдовипадкових чисел, формування структурованих повідомлень у форматі JSON, а також надсилання HTTP-запитів до вебсервера. Крім того, Python дозволяє швидко створювати скрипти для моделювання різних сценаріїв поведінки пристроїв, включно зі зміною інтервалу надсилання даних, параметрів або структур повідомлень. Це забезпечує гнучкість тестування системи на ранніх етапах розробки.

Для створення серверної частини був обраний мікрофреймворк Flask. Його перевагою є легкість у налаштуванні, швидкий запуск розробки та інтеграція з іншими бібліотеками Python. Flask дозволяє створювати REST API з мінімальними витратами коду, при цьому забезпечуючи розширюваність і підтримку сучасних технологій веброзробки. Завдяки своїй гнучкій структурі, Flask добре підходить для побудови мікросервісів, які в подальшому можуть бути об’єднані в більші розподілені системи.

У якості системи керування базами даних було обрано MongoDB. Цей вибір обумовлений її документно-орієнтованою природою, яка є особливо зручною для роботи з нестабільною або змінною структурою даних, характерною для IoT-повідомлень. MongoDB дозволяє зберігати кожен запис у вигляді документа BSON, що легко конвертується з і в

JSON. Така структура дозволяє динамічно додавати нові поля або змінювати схему зберігання без необхідності модифікації схеми всієї БД. Крім того, MongoDB підтримує гнучкі механізми індексації, швидкий пошук та масштабування на рівні кластерів, що є важливим для обробки великих потоків телеметричних даних.

Для хостингу серверного застосунку було обрано платформу Render. Це хмарне середовище дозволяє розгортати повнофункціональні вебзастосунки без необхідності конфігурування серверів вручну. Render підтримує деплой через GitHub, автоматичні оновлення, середовище для запуску Python-додатків та підключення до зовнішніх баз даних, зокрема MongoDB Atlas. Це забезпечує гнучке управління застосунком, можливість моніторингу, логування і масштабування у разі підвищеного навантаження.

У разі розгортання локальної версії або використання альтернативного середовища, також була розглянута платформа Railway, яка має схожі функціональні можливості, проте менш зручний інтерфейс керування проєктами.

Таким чином, поєднання Python, Flask, MongoDB і Render дозволяє створити легкий у підтримці, розширюваний та ефективний хмарний сервіс для прийому, зберігання та обробки IoT-даних у реальному часі. Такий технологічний стек оптимально підходить для реалізації дипломного проєкту, оскільки забезпечує високу швидкість розробки, прозорість архітектури та адаптивність до майбутнього масштабування.

2.3. Архітектура системи та взаємодія компонентів

Архітектура хмарного сервісу для збору та обробки даних від IoT-пристроїв базується на принципах модульності, гнучкості та

масштабованості. Вона реалізована за клієнт-серверною моделлю з чітким розмежуванням відповідальності між елементами: пристроєм (емулятором), серверною частиною (API), системою зберігання даних (MongoDB) та інтерфейсом користувача.

На першому рівні знаходиться IoT-пристрій або його емулятор, реалізований на Python. Його основною функцією є періодична генерація показників (наприклад, температури, вологості або іншого фізичного параметра) та передача цих даних через HTTP-запити на сервер. Для цього використовується формат JSON, який легко обробляється серверною частиною та зберігається у базі даних.

Серверна частина, створена з використанням Flask, виконує роль проміжного прошарку між пристроями та базою даних. Вона приймає запити від клієнтів, здійснює базову валідацію даних (перевірку формату, наявності обов’язкових полів, допустимості значень), після чого виконує запис у MongoDB. Сервер також містить реалізацію REST API, через яке відбувається обробка запитів на отримання останніх даних для відображення на фронтенді.

База даних MongoDB виступає у ролі центрального сховища всіх телеметричних даних. Її документна структура дозволяє легко масштабувати модель даних, адаптувати її під зміну структури JSON-повідомлень та ефективно виконувати агрегаційні запити, які можуть використовуватися для подальшої аналітики або побудови графіків.

На стороні клієнта реалізовано простий вебінтерфейс, який дозволяє переглядати останні отримані значення від IoT-пристроїв. За потреби інтерфейс може бути розширений функціоналом авторизації, фільтрації за датою або побудови графіків на основі збережених даних.

Комунікація між компонентами є асинхронною. IoT-пристрій не очікує відповіді на кожен запит, що дозволяє йому продовжувати роботу незалежно від стану сервера або бази даних. Такий підхід забезпечує більшу надійність системи в умовах втрати зв’язку або перевантаження сервера. У майбутньому архітектуру можна легко розширити, додавши черги повідомлень (наприклад, на базі RabbitMQ або Redis) для буферизації даних та підвищення стійкості системи.

Загальна архітектура забезпечує прозору обробку даних у режимі реального часу, що є критично важливим для IoT-рішень. Крім того, обраний підхід дозволяє гнучко масштабувати кожен компонент окремо: збільшити кількість емуляторів, додати репліки бази даних або розгорнути кілька серверів API за допомогою балансувальника навантаження.

2.4. Діаграма потоків даних

Процес взаємодії між компонентами системи побудований за принципом послідовної передачі даних від IoT-пристрою до хмарного інтерфейсу користувача з можливістю зберігання та подальшого аналізу інформації. В основі архітектури лежить логічний потік даних, що забезпечує чіткий розподіл функціональних обов’язків на кожному з етапів.

Першим джерелом даних є IoT-пристрій, у даній роботі реалізований у вигляді емулятора. Він створює симульовані сенсорні показники у форматі JSON — наприклад, значення температури, вологості або іншого обраного параметра. Емулятор періодично (наприклад, кожні 5 секунд) надсилає дані на сервер через HTTP POST-запит, вказуючи в тілі запиту сформоване повідомлення.

На наступному етапі Flask-сервер отримує цей запит через спеціально створений маршрут API, наприклад /api/data. Контролер цього маршруту перевіряє коректність структури отриманого JSON-пакету, здійснює логічну валідацію (чи присутні обов’язкові поля, чи не перевищують значення допустимі межі) та у випадку успіху передає дані до системи зберігання.

MongoDB, яка виступає як сховище даних, приймає ці записи у вигляді документів. Для кожного запиту створюється новий запис у відповідній колекції (наприклад, sensor_data), із збереженням усіх отриманих параметрів та автоматичним додаванням мітки часу (timestamp). Це дозволяє відслідковувати історію змін і здійснювати аналіз динаміки параметрів у часі.

На завершальному етапі, коли користувач відкриває вебінтерфейс, фронтенд надсилає GET-запит на той самий сервер Flask із запитом на останні дані. Сервер отримує запит, звертається до MongoDB, вибирає найновіші записи з бази й повертає їх у форматі JSON назад у браузер. Дані відображаються у вигляді текстових значень або графіків, залежно від реалізації клієнтського інтерфейсу.

Таким чином, потік даних відбувається в чіткій послідовності: Емулятор → Flask API → MongoDB → Клієнтський інтерфейс, що забезпечує реєстрацію, зберігання та доступ до сенсорної інформації в режимі майже реального часу.

Архітектура системи дозволяє легко масштабувати потік — наприклад, підтримувати кілька емуляторів одночасно або зберігати дані від різних типів пристроїв, використовуючи відповідні мітки і фільтри при збереженні та вибірці.

2.5. Висновки до розділу

У цьому розділі було детально проаналізовано підходи до проєктування архітектури системи збору та обробки IoT-даних, а також обґрунтовано вибір інструментальних засобів, які були використані при реалізації дипломного проєкту. Запропонована система поєднує в собі простоту реалізації та гнучкість масштабування завдяки розподіленню функціональності на логічні шари та використанню хмарного середовища.

Формалізоване технічне завдання дозволило чітко визначити функціональні вимоги до системи, серед яких ключовими були: приймання даних від IoT-пристрою, збереження показників у базі даних, обробка запитів через REST API, а також відображення інформації в інтерактивному вебінтерфейсі.

Вибір інструментів було здійснено на основі порівняння їх функціональних характеристик і відповідності вимогам проєкту. Flask обрано як легкий мікрофреймворк для побудови серверної логіки, MongoDB — як масштабовану документоорієнтовану базу даних, а Render — як доступну платформу для безперервного розгортання та хостингу вебсервісу. Python також використовується для створення емулятора IoT-пристрою, що дозволяє гнучко змінювати параметри генерації даних і тестувати систему в умовах, наближених до реальних.

Розроблена логічна архітектура забезпечує зрозумілий та надійний потік даних між усіма компонентами: від пристрою — до хмари й інтерфейсу користувача. Це дозволяє системі бути розширюваною, незалежною від конкретного обладнання і готовою до інтеграції з реальними IoT-платформами або сенсорами у майбутньому.

У підсумку, цей розділ створює концептуальну й технічну основу для практичної реалізації системи, яка буде детально представлена у наступному розділі.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ

3.1. Реалізація емулятора IoT-пристрою

Оскільки в межах цього проєкту IoT-пристрій є умовним і не базується на фізичному сенсорному модулі, для потреб моделювання було розроблено програмний емулятор, який генерує випадкові числові значення, що імітують реальні дані, зокрема показники температури, вологості, або інші аналогічні параметри. Емулятор написаний мовою Python, оскільки ця мова забезпечує зручний синтаксис і надає розширені можливості для роботи з мережею, часом і форматуванням даних.

Основним завданням емулятора є створення та надсилання HTTP POST-запитів до API-сервера у визначені інтервали часу. Кожен запит містить згенерований об'єкт у форматі JSON, у якому присутні поля: ідентифікатор пристрою, значення сенсора, мітка часу та допоміжна інформація, наприклад, статус з’єднання. Це дозволяє не лише перевіряти роботу API, а й моделювати навантаження системи, оцінювати поведінку бази даних під час частого запису нових даних.

Особливу увагу при розробці було приділено реалізації таймерів, які дозволяють періодично надсилати дані без потреби ручного запуску. Таким чином, система може працювати в режимі, максимально наближеному до реального середовища, в якому IoT-пристрій постійно передає нову інформацію до центрального хмарного сервісу. У подальшому емулятор може бути легко адаптований для роботи з фізичним сенсорним модулем (наприклад, на базі Raspberry Pi або ESP32) за рахунок заміни функції генерації випадкових чисел на реальні показники.

3.2. Розробка серверної частини

Серверна частина системи реалізована на основі мікрофреймворку Flask, що надає зручні засоби для створення REST API. API виконує роль посередника між IoT-пристроєм (чи емулятором), базою даних і вебінтерфейсом. Основними завданнями API є: прийом запитів, перевірка коректності переданих даних, обробка виключень і безпосереднє збереження інформації до бази даних MongoDB.

Реалізований маршрут /api/data приймає POST-запити з JSON-тілом. Перед записом API перевіряє, чи всі необхідні поля присутні, і чи мають вони допустимі значення. Якщо дані відповідають вимогам, вони зберігаються у відповідній колекції бази даних. Для запобігання втраті або дублюванню інформації передбачено перевірку унікальності ідентифікаторів та фіксацію часу надходження запису.

Також у серверній частині передбачено окремі маршрути для отримання останніх записів (GET-запити) або всієї історії даних. Це дозволяє вебінтерфейсу підключатися до API та відображати потрібну інформацію в реальному часі. Сервер підтримує обробку одночасних запитів і може бути масштабований за допомогою додаткових інструментів, таких як Gunicorn або uWSGI, при розгортанні в продакшн-середовищі.

3.3. Веб-інтерфейс

Вебінтерфейс був реалізований з використанням HTML, CSS та JavaScript для клієнтської частини, яка звертається до Flask API задля отримання і виводу даних у зручному для користувача форматі. Основна мета інтерфейсу — забезпечити можливість візуалізації

отриманих із пристрою даних, а також відображення історії вимірювань у вигляді таблиць або графіків.

Головна сторінка інтерфейсу включає просту панель перегляду останніх значень сенсорних даних. Для цього використовується AJAX-запит до ендпоінту /api/latest, який повертає останній запис із бази даних. Дані динамічно оновлюються з певною періодичністю (наприклад, кожні 10 секунд), що дозволяє реалізувати ефект моніторингу в реальному часі.

Також було реалізовано компонент історії вимірювань, де користувач може переглядати всі отримані записи за певний період. На цій сторінці використовується бібліотека Chart.js, яка дозволяє будувати графіки на основі числових даних. Кожна точка на графіку відповідає одному запису з бази даних, що дозволяє здійснювати первинний аналіз трендів — наприклад, виявляти пікові значення, середні показники або аномалії.

Інтерфейс було оформлено таким чином, щоб забезпечити максимальну простоту використання: мінімалістичний дизайн, контрастне відображення значень, адаптивна верстка для мобільних пристроїв. Також реалізовано базову авторизацію — доступ до інтерфейсу з можливістю редагування (наприклад, очистки даних) дозволено лише користувачам, які мають відповідні облікові дані. Це реалізовано через базову форму логіна, яка перевіряє введені облікові записи і встановлює сесію.

У разі виявлення помилок, таких як відсутність зв’язку з API чи невірний формат даних, користувач бачить відповідне повідомлення. Завдяки цьому забезпечується не лише функціональність, а й

інформативність системи — користувач може оперативно реагувати на технічні збої.

3.4. Розгортання сервісу в хмарі

Для забезпечення постійного доступу до IoT-сервісу з будь-якої точки світу було обрано модель хмарного розгортання. У якості платформи хостингу було використано сервіс Render.com, який дозволяє швидко деплоїти вебзастосунки, API та бази даних із підтримкою безперервного розгортання (CI/CD) через інтеграцію з GitHub.

Розгортання системи складалося з кількох основних етапів:

1. Публікація вихідного коду на GitHub у відкритому репозиторії https://github.com/VladislavKostyniuk/iot-project. Структура репозиторію включає директорії з API-кодом на Flask, статичними файлами інтерфейсу, конфігураційним файлом requirements.txt для встановлення залежностей та файлом render.yaml для опису процесу побудови й деплою.
2. Створення нового вебсервісу на Render. У налаштуваннях вказується Git-репозиторій, команда побудови (наприклад, pip install -r requirements.txt), та команда запуску (наприклад, python app.py). Оскільки Flask за замовчуванням запускається локально, у коді було використано бібліотеку gunicorn, що дозволяє адаптувати запуск сервера до умов продакшну.
3. Налаштування змінних середовища (environment variables), таких як DATABASE_URI, які необхідні для з’єднання з хмарною базою MongoDB. Для зберігання даних була використана окрема інстанція MongoDB Atlas або Railway (альтернативна

платформа), яка надає безкоштовний хостинг БД з інтерфейсом керування.

4. Деплой та моніторинг. Після завершення налаштувань сервіс автоматично зібрав і розгорнув застосунок. У випадку змін у GitHub-репозиторії відбувається автоматичне повторне розгортання. Система логів Render надає змогу відслідковувати статус сервера, HTTP-запити, помилки та навантаження.

Таким чином, завдяки Render вдалося реалізувати повноцінну хмарну інфраструктуру для IoT-сервісу, доступну за посиланням: https://iot-project-zy46.onrender.com/. Це рішення забезпечує безперервну доступність, масштабованість та простоту супроводу розробленої системи.

3.5. Тестування та аналіз роботи системи

Тестування є критично важливим етапом у розробці будь-якої інформаційної системи, а особливо — у проєктах, що взаємодіють із зовнішніми пристроями та мережею. Метою тестування розробленого IoT-сервісу було перевірити його коректність, стабільність, витривалість під навантаженням і поведінку в умовах втрати з’єднання.

Одним із перших етапів стало функціональне тестування API. Застосовуючи Postman, було перевірено, чи API приймає дані, що надсилаються у форматі JSON. Кожен HTTP POST-запит від емулятора містив ключові параметри: ідентифікатор пристрою, показник температури або вологості, мітку часу. У відповідь сервер повертав статус-код 200 або повідомлення про помилку (наприклад, при порушенні формату запиту).

Далі проводилося тестування обробки та збереження даних. Після успішного надсилання інформації її наявність перевірялась

безпосередньо в базі даних MongoDB. Для цього використовувався MongoDB Compass — графічна оболонка, що дозволяє переглядати записи в колекціях, фільтрувати їх за датою або пристроєм, а також аналізувати структуру збережених документів.

Особливу увагу приділено тестуванню візуалізації даних. На вебінтерфейсі відображалися останні значення від пристрою в реальному часі. Було реалізовано простий графік температури з використанням бібліотеки Chart.js, який динамічно оновлювався. Перевірка проводилася за сценарієм: надсилання 10–20 значень з інтервалом у кілька секунд, відображення на графіку, оновлення останнього блоку з поточним станом пристрою.

Крім того, протестовано поведінку сервісу при високому навантаженні. Для цього було запущено скрипт, який імітує паралельну роботу 10 емуляторів, що надсилають дані одночасно кожні 2 секунди. Система витримувала навантаження без втрат даних або збоїв у роботі. Серверна частина залишалась стабільною, час відповіді API не перевищував 300–400 мс.

Випробування при втраті з’єднання показали, що у разі відсутності інтернету на боці пристрою передача даних не здійснюється, однак після відновлення з’єднання скрипт продовжував надсилання без потреби в перезапуску. Це підтверджує стійкість клієнтського коду до мережевих проблем.

На завершення було проведено аналіз переваг і обмежень розробленого рішення. До ключових переваг слід віднести:

просту й гнучку архітектуру з відкритим кодом;

швидке розгортання та доступність з будь-якої точки світу;

можливість масштабування та розширення функцій;

відсутність залежності від платних платформ.

Серед обмежень — відсутність складної авторизації, базова візуалізація, залежність від стабільності Render.com при великому навантаженні.

Загалом, тестування продемонструвало високу працездатність системи, готовність до інтеграції з реальними пристроями та ефективність її роботи в типових сценаріях використання.

3.6. Висновки до розділу

У результаті реалізації та тестування системи збору та обробки даних від IoT-пристроїв були досягнуті основні цілі, поставлені на початку проєкту. Було створено повнофункціональний хмарний сервіс, що охоплює генерацію даних на стороні емулятора, їх приймання через REST API, зберігання у базі даних MongoDB та візуалізацію через вебінтерфейс.

Етап розробки емулятора IoT-пристрою продемонстрував, що навіть у середовищі без фізичних сенсорів можна імітувати надсилання достовірних даних, що є особливо корисним на етапі проєктування і тестування. Завдяки використанню Python і бібліотеки requests вдалося швидко реалізувати модуль генерації температурних та інших значень, які надсилаються у форматі JSON.

Серверна частина, розроблена з використанням фреймворку Flask, показала стабільність і гнучкість у обробці HTTP-запитів. Було реалізовано механізми перевірки валідності даних, логування помилок, а також структуру обробки запитів, придатну для масштабування.

База даних MongoDB, як нереляційне сховище, виявилася зручною для зберігання структурованих, але гнучких JSON-документів.

Це дозволило легко адаптувати модель даних у ході розробки без суттєвих змін у коді.

Хмарне розгортання на Render забезпечило доступність проєкту без необхідності в локальній інфраструктурі, а також спростило процес оновлення та обслуговування системи. Увесь стек рішень, обраний на етапі проєктування, підтвердив свою ефективність на практиці.

Тестування сервісу показало, що система здатна обробляти вхідні дані в режимі реального часу, залишаючись стійкою до помилок з’єднання, мережевих затримок і високого навантаження. Візуалізація даних у вебінтерфейсі забезпечила наочність і зручність у сприйнятті інформації.

У сукупності всі етапи реалізації продемонстрували, що побудова IoT-рішення на базі відкритих технологій можлива без значних ресурсних витрат, при цьому забезпечуючи масштабованість, гнучкість і надійність, достатні для навчального або малого комерційного використання. Результати цього розділу закладають підґрунтя для майбутньої модернізації системи, зокрема — додавання реальних сенсорів, введення багатокористувацького режиму та реалізації аналітичних модулів.

ВИСНОВКИ

У межах кваліфікаційної роботи було реалізовано хмарний сервіс для збору та обробки даних від IoT-пристроїв, який демонструє можливість побудови доступної, масштабованої та функціональної системи з використанням сучасних вебтехнологій та інфраструктур хмарних платформ. Проведений аналіз предметної області засвідчив актуальність застосування IoT-рішень у різних галузях, зокрема в моніторингових системах, «розумних» середовищах, сільському господарстві, енергетиці та екології. Розглянуто архітектурні підходи до побудови IoT-систем, що дало змогу обґрунтовано обрати хмарну модель як найоптимальнішу для даного проєкту.

Проєктування системи охоплювало формування технічного завдання, обґрунтування вибору інструментів, розробку логіки взаємодії між компонентами, опис потоків даних. Реалізацію здійснено за допомогою Python, Flask, MongoDB, а хостинг виконано на платформі Render. Вебінтерфейс забезпечив можливість відображення результатів у зручному вигляді, а серверна частина забезпечила надійний прийом і зберігання даних.

У процесі тестування було підтверджено стабільну роботу системи в умовах різного навантаження, перевірено здатність обробляти неперервний потік вхідних даних, оцінено стійкість до збоїв і втрат зв’язку. Система коректно обробляє HTTP-запити, виконує валідацію даних і забезпечує наочне виведення інформації для користувача.

Розроблений прототип підтвердив ефективність використання відкритого стеку технологій для побудови повноцінної IoT-інфраструктури. Рішення може бути розширене або модифіковане

залежно від прикладної сфери — для аграрного моніторингу, систем безпеки, енергозбереження або інтелектуального управління просторами. Підвищення складності системи в майбутньому може включати роботу з реальними сенсорами, впровадження механізмів авторизації, генерацію статистичних звітів, використання сокетів для реалізації оновлення даних у реальному часі, інтеграцію з мобільними застосунками.

Таким чином, результати виконаної роботи демонструють доцільність використання хмарних технологій у поєднанні з IoT-рішеннями та відкритими засобами розробки для створення адаптивних цифрових сервісів із перспективою подальшого масштабування та інтеграції в різні сфери людської діяльності.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Analysis of Cloud Platforms for IoT Services – Elsevier Procedia Computer Science, 2021.

2. Ashton, K. That 'Internet of Things' Thing // RFID Journal. – 2009. – URL: https://www.rfidjournal.com/that-internet-of-things-thing

3. AWS IoT Core Documentation – URL: https://docs.aws.amazon.com/iot

4. Azure IoT Hub Documentation – URL: learn.microsoft.com/en-us/azure/iot-hub

5. Bandyopadhyay, D., Sen, J. Internet of Things: Applications and Challenges in Technology and Standardization // Wireless Personal Communications. – 2011.

6. Best Practices for Cloud Deployment – Amazon Web Services Blog.

7. Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., & Brandic, I. Cloud computing and emerging IT platforms // Future Generation Computer Systems. – 2009.

8. CoAP Protocol Specification. – IETF RFC 7252. – URL: tools.ietf.org/html/rfc7252

9. Fielding, R. T. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation. – University of California, Irvine, 2000.

10. Flask Documentation – URL: https://flask.palletsprojects.com

11. Google Cloud IoT Core Documentation – URL: cloud.google.com/iot-core

12. Gubbi, J., Buyya, R., Marusic, S., & Palaniswami, M. Internet of Things (IoT): A vision, architectural elements, and future directions // Future Generation Computer Systems. – 2013.

13. HTTP/1.1 Protocol Specification. – IETF RFC 2616. – URL: https://www.w3.org/Protocols/

14. Intelligent Monitoring and Predictive Maintenance using IoT – IEEE Access, 2020.

15. Introduction to MongoDB – MongoDB University. – URL: https://university.mongodb.com

16. IoT Data Management Frameworks – Springer, 2022.

17. IoT Security Best Practices – CISCO White Paper, 2020.

18. IoT-based Smart Agriculture: An Overview – International Journal of Computer Applications, 2019.

19. JSON (JavaScript Object Notation) Specification – URL: www.json.org

20. Minerva, R., Biru, A., & Rotondi, D. Towards a Definition of the Internet of Things (IoT). – IEEE Internet Initiative, 2015.

21. MongoDB Documentation – URL: https://www.mongodb.com/docs/

22. MQTT Version 5.0. OASIS Standard. – URL: https://mqtt.org/mqtt-specification/

23. Python Requests Library Documentation – URL: https://docs.python-requests.org

24. Railway Cloud Platform – URL: https://railway.app

25. Render Hosting Platform – URL: https://render.com

26. REST API Design Rulebook / M. Masse. – O'Reilly Media, 2011.

27. Web API Development with Flask – Real Python Guide.

28. White Paper: Fog Computing and its Role in the Internet of Things. – OpenFog Consortium.

29. Zeng, D., Guo, S., & Cheng, Z. The Web of Things: A Survey // Journal of Communications. – 2011. – №6(6).

Zhang, Y., & Ansari, N. On Harnessing the Power of Optical Technologies in Green and Smart Cloud Computing // IEEE Communications Surveys & Tutorials. – 2013.