НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет/(ННІ) інформаційних технологій________________________________

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету (Директор ННІ)	**Завідувач кафедри**
Інформаційних технологій	__комп'ютерних наук__
(назва факультету (ННІ))	(назва кафедри)
__________ Ігор Болбот	_________ Белла Голуб
(підпис) (ім'я ПРІЗВИЩЕ)	(підпис) (ім'я ПРІЗВИЩЕ)
"___" ___________ 20_ р.	"___" ___________ 20_ р.

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему Система розповсюдження інформації про стан процесів в тепличному господарстві

Спеціальність 122 «Комп'ютерні науки
(код і найменування)

Освітня програма Інформаційні управляючі системи та технології
(назва)

Орієнтація освітньої програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Гарант освітньої програми

____к.т.н., доцент___________ _______________ ___Белла Голуб__
(науковий ступінь та вчене звання) (підпис) (ім'я ПРІЗВИЩЕ)

Керівник магістерської кваліфікаційної роботи

____д.т.н., проф.___ _______________ _Микола Цюцюра_
(науковий ступінь та вчене звання) (підпис) (ім'я ПРІЗВИЩЕ)

Виконав __________________ Олександр Зінченко_
(підпис) (ім'я ПРІЗВИЩЕ здобувача)

КИЇВ – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет (ННІ) __інформаційних технологій___________________________________

ЗАТВЕРДЖУЮ

Завідувач кафедри__комп'ютерних наук_

___доцент, к.т.н.______ _______ Голуб Б. Л._

(науковий ступінь, вчене звання) (підпис) (ПІБ)

"_01_" ___листопада______2024 року

З А В Д А Н Н Я

ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТУ

_________________Зінченко Олександр Олександрович____________________

(прізвище, ім'я, по батькові)

Спеціальність 122 «Комп'ютерні науки»

(код і назва)

Освітня програма ____Інформаційні управляючі системи та технології

(назва)

Орієнтація освітньої програми_____освітньо-професійна________________________

Тема магістерської кваліфікаційної роботи Система розповсюдження інформації про стан процесів в тепличному господарстві

затверджена наказом ректора НУБіП України від "_01_"_листопада____2024р. №1964 «С»

Термін подання завершеної роботи на кафедру _. _01.12.2025________________________

(рік, місяць, число)

Вихідні дані до магістерської кваліфікаційної роботи: є телеметричні параметри з тепличного господарства (температура, вологість, освітленість, стан ґрунту, рівень CO_2), що надходять від датчиків через вбудовану систему збору даних

Перелік питань, що підлягають дослідженню:

1.Аналіз технологічних вимог до моніторингу та контролю мікроклімату в тепличному господарстві.

2.Дослідження моделей збору, агрегації та фільтрації телеметричних даних у реальному часі.

3.Визначення оптимальних методів передачі, візуалізації та розповсюдження даних серед зацікавлених користувачів.

4.Оцінка впливу параметрів середовища (температура, вологість, CO_2) на ефективність вирощування культур.

5.Вибір і впровадження архітектури хмарної платформи для аналітики та оповіщення в тепличній системі.

Дата видачі завдання "__01__"__листопада___2024 р.

Керівник магістерської кваліфікаційної роботи ________ ___Цюцюра М. І.____

(підпис) (прізвище та ініціали)

Завдання прийняв до виконання ________________ __________Зінченко О.О.______

(підпис) (прізвище та ініціали студента)

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ 6
ВСТУП 8
1 СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 11
1.1 Опис предметної області 11
1.2 Огляд інформаціних джерел та існуючих рішень 13
1.3 Моделювання предметної області 17
1.4 Аналіз вимог програмної системи 20
1.5 Постановка завдання 23
1.6 Висновки до розділу 1 24
2 ПРОЕКТУВАННЯ ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 26
2.1 Логічна модель даних у вигляді ER-діаграми 26
2.2 Діаграма класів і кооперації 28
2.3 Діаграма компонентів розроблювальної системи 31
2.4 Діаграма пакетів 34
2.5 Висновки до розділу 2 36
3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 38
3.1 Вибір технологій та інструментальних засобів реалізації системи 38
3.2 Фізична модель даних і структура OLAP-кубу 41
3.3 Архітектура системи 46
3.4 Алгоритмізація програмних модулів 50
3.5 Висновки до розділу 3 56
РОЗДІЛ 4. ТЕСТУВАННЯ ТА ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ АНАЛІТИЧНОЇ СИСТЕМИ 57
4.1 План тестування програмних модулів та методика оцінювання результатів57
4.2 Тестування інтелектуальної системи розповсюдження інформації про стан мікроклімату тепличного господарства 59
4.3 Результати тестування та аналіз ефективності системи 64
4.4 Розгортання системи та склад інсталяційного пакета 66

4.5 Висновки до розділу 4 68
ВИСНОВКИ.. 70
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 72
ДОДАТОК А 74

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

1. API — інтерфейс програмування застосунків
2. CSV — формат текстових табличних даних Comma-Separated Values
3. CO_2 — концентрація вуглекислого газу
4. DB — база даних
5. ER — діаграма «сутність–зв’язок»
6. GUI — графічний інтерфейс користувача
7. HTTP — протокол передавання гіпертексту
8. HTTPS — захищена версія HTTP
9. IO — операції введення/виведення
10. IoT — Інтернет речей
11. JSON — текстовий формат обміну структурованими даними
12. KPI — ключові показники ефективності
13. Lux — одиниця освітленості
14. MQTT — легковаговий протокол обміну повідомленнями за моделлю publish/subscribe
15. OLAP — оперативна аналітична обробка даних
16. OPC UA — універсальна архітектура обміну даними між промисловими пристроями
17. RBAC — рольова модель контролю доступу
18. SCADA — система диспетчерського керування та збору даних
19. SQL — структурована мова запитів
20. SSO — єдиний вхід користувача
21. TLS — протокол захищеного передавання даних
22. UI — інтерфейс користувача
23. UTC — координований всесвітній час
24. Z-score — стандартизований показник відхилення вимірювання від середнього значення

25. Alert — повідомлення про виявлене відхилення або аномалію
26. Command — керувальна команда для актуатора
27. SensorReading — вимірювання, отримане від сенсора
28. Threshold policy — порогова політика оцінювання параметрів
29. Ingest — модуль приймання та первинної обробки даних
30. Actuator — виконавчий механізм (вентилятор, клапан, нагрівач)
31. Edge — периферійний рівень збору та попередньої обробки телеметрії
32. Core Services — аналітичні та керувальні сервіси центральної частини системи
33. Data Layer — рівень зберігання та OLAP-аналітики телеметрії
34. Delivery — рівень доставки повідомлень і дашбордів користувач

ВСТУП

Зміна кліматичних умов, енергетичні виклики та підвищені вимоги до сталого сільськогосподарського виробництва зумовлюють необхідність впровадження цифрових інструментів моніторингу в тепличному господарстві. Точне й оперативне отримання інформації про параметри мікроклімату - температуру, вологість, концентрацію CO_2, освітленість - є критичним для прийняття своєчасних рішень щодо зрошення, вентиляції, обігріву та запобігання аномальним ситуаціям. У традиційних тепличних комплексах моніторинг часто реалізується через локальні контролери або SCADA-системи, що обмежено масштабуються, фрагментовано зберігають дані та не забезпечують належного рівня візуалізації або оповіщення. З огляду на потребу в підвищенні ефективності керування ресурсами та забезпечення простежуваності параметрів середовища, актуальною є побудова автоматизованої системи, здатної інтегрувати сенсорні дані, метеослужби та керуючі елементи з подальшим розповсюдженням інформації через зручні цифрові канали.

Мета роботи полягає у розробці інформаційної системи, що забезпечує збір, оцінку, виявлення відхилень та розповсюдження інформації про стан мікроклімату в тепличному господарстві, з можливістю формування сповіщень та ініціювання команд керування.

Для досягнення поставленої мети необхідно вирішити такі **завдання:**

– провести аналіз існуючих підходів до моніторингу в тепличних умовах та виявити обмеження традиційних систем;

– сформулювати функціональні, нефункціональні та інтеграційні вимоги до цільового програмного забезпечення;

– побудувати UML-моделі (прецедентів, послідовності, активності), що описують ключові сценарії збору, обробки та поширення даних;

– розробити архітектуру підсистеми збору та нормалізації телеметрії з агрегацією метеоданих;

– реалізувати механізм оцінки порогових значень, виявлення аномалій та тригерів;

– інтегрувати модулі формування сповіщень і видачі команд керування в реальному часі;

– забезпечити оновлення дашборду операторів та взаємодію з мобільними пристроями;

– протестувати систему на відповідність вимогам до точності, затримки, надійності та масштабованості.

Об'єктом дослідження є інформаційні потоки мікрокліматичних даних у тепличному господарстві, що підлягають контролю, обробці та поширенню.

Предмет дослідження - методи інтеграції сенсорної телеметрії, моделі виявлення відхилень, архітектури розповсюдження подій та сповіщень у контексті розподіленої цифрової платформи.

У роботі застосовано такі методи дослідження: аналіз існуючих систем моніторингу; об'єктно-орієнтоване моделювання процесів за допомогою UML; логічне й функціональне проєктування інформаційних потоків; використання алгоритмів класифікації та валідації даних; реалізація клієнт-серверної взаємодії з REST API та push-технологіями.

Наукова новизна полягає у створенні цілісної системи, яка поєднує телеметрію SCADA-сенсорів, прогнозні моделі метеоданих, модулі оцінки ризиків та адаптивного реагування у вигляді команд керування та сповіщень, із підтримкою дашбордів та мобільних каналів розповсюдження інформації в реальному часі.

Практична цінність розробленої системи полягає у можливості її впровадження на тепличних господарствах для зниження витрат на енергію, покращення стабільності мікроклімату та підвищення врожайності завдяки оперативному реагуванню на зміну умов.

Структура роботи відповідає логіці життєвого циклу інформаційної системи: перший розділ присвячено опису предметної області, аналізу аналогів і моделюванню процесів; другий - проєктуванню архітектури, логіки дій і функціональних взаємодій; третій - реалізації підсистем, обробки даних та інтеграції; четвертий - експериментальному тестуванню та оцінці результатів за технічними критеріями.

1 СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис предметної області

Тепличне господарство є складною динамічною системою, в якій параметри середовища (температура, вологість, рівень CO_2, освітленість тощо) безпосередньо впливають на вегетаційні процеси та кінцеву продуктивність. Порушення допустимих діапазонів або несвоєчасне реагування на відхилення може призвести до втрат урожаю, зниження якості продукції або перевитрат ресурсів (води, енергії). У зв’язку з цим постає необхідність у впровадженні систем, здатних не лише фіксувати параметри середовища, а й ефективно поширювати критичну інформацію серед відповідальних осіб і систем автоматизованого керування. Ці системи мають поєднувати джерела телеметрії, алгоритми обробки даних, модулі виявлення аномалій і канали комунікації у єдину інформаційну платформу.

Узагальнена класифікація систем розповсюдження інформації про стан процесів у тепличному господарстві подана на рис. 1.1. Вона охоплює ознаки поділу за джерелами даних, типом обробки, каналами сповіщення, режимами керування, вимогами до безпеки та надійності, а також інтеграційними властивостями.

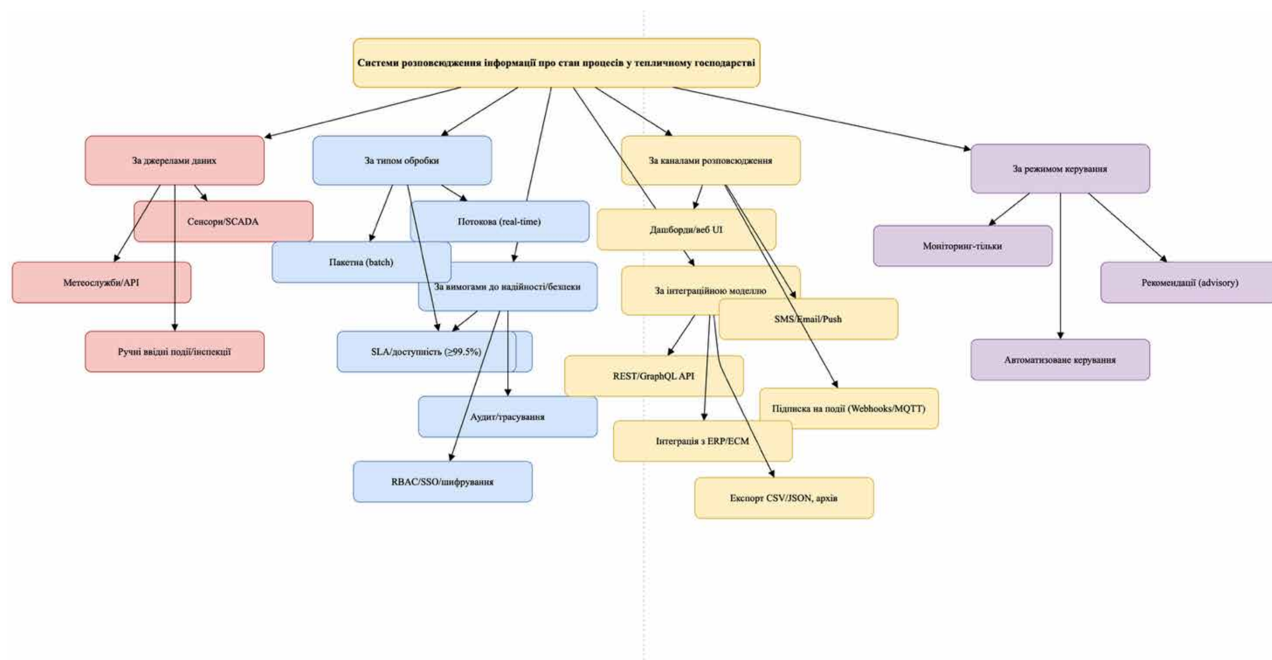

Рис. 1.1 – Класифікація систем розповсюдження інформації у тепличному господарстві

Згідно з поданою класифікацією, джерела даних можуть включати SCADA-сенсори, API метеослужб або ручні інспекційні дані. За типом обробки розрізняють потоковий режим (реального часу), пакетну обробку (batch) або гібридну модель із перенесенням даних між edge-та cloud-рівнями. Канали доставки інформації охоплюють як візуальні дашборди, так і автоматизовані сповіщення через SMS, email чи push. Режим реагування варіюється від моніторингу до повної автоматизації зворотного впливу на актуатори. Окремо виділено вимоги до SLA, безперервності доступу, трасування та RBAC/SSO механізмів. Завершує класифікацію інтеграція з ERP-системами, REST API, експорт даних тощо.

Для формалізації предметної області також було побудовано узагальнену таблицю порівняння типів систем за критеріями функціональності, надійності, адаптивності та інтеграції (табл. 1.1). У порівняння включено ручні системи, типові SCADA-системи та розроблювана цифрова платформа.

Таблиця 1.1 – Порівняння типів інформаційних систем у тепличному господарстві

Критерій	Ручна система	SCADA-панелі	Розроблювана система
Джерела даних	Інспекції	Локальні сенсори	SCADA + API + Events
Тип обробки	Після-фактум	Потокова	Потокова + пакетна
Сповіщення	Вручну	Світлова/звукова	SMS / Email / Push
Аномалії	Не фіксуються	Частково	Автоматичне виявлення
Інтеграція	Відсутня	Пропрієтарна	REST/MQTT/ERP
Масштабованість	Обмежена	Складна	Контейнерна (Docker)
Безпека (аутентифікація, аудит)	Немає	Мінімальна	JWT + RBAC + SSO
Формати експорту	Паперові	Частково CSV	CSV / JSON / API

Розроблювальна система значно переважає аналогічні рішення за рівнем інтеграції, адаптивності до аномалій і підтримки сучасних протоколів комунікації. Завдяки цьому вона здатна не лише інформувати користувачів, а й забезпечити проактивне реагування в реальному часі.

1.2 Огляд інформаційних джерел та існуючих рішень

Розробка систем розповсюдження інформації про стан процесів у тепличному господарстві потребує врахування існуючих агропромислових та дослідницьких рішень, що реалізують подібний функціонал з використанням сенсорних даних, мережевих протоколів і механізмів інтеграції з аналітичними модулями. Серед промислових рішень одним з орієнтирів виступає платформа Netafim, проте як видно на рис. 1.2, частина її веб-ресурсів недоступна, що ускладнює детальний аналіз внутрішньої архітектури та каналів передачі даних.

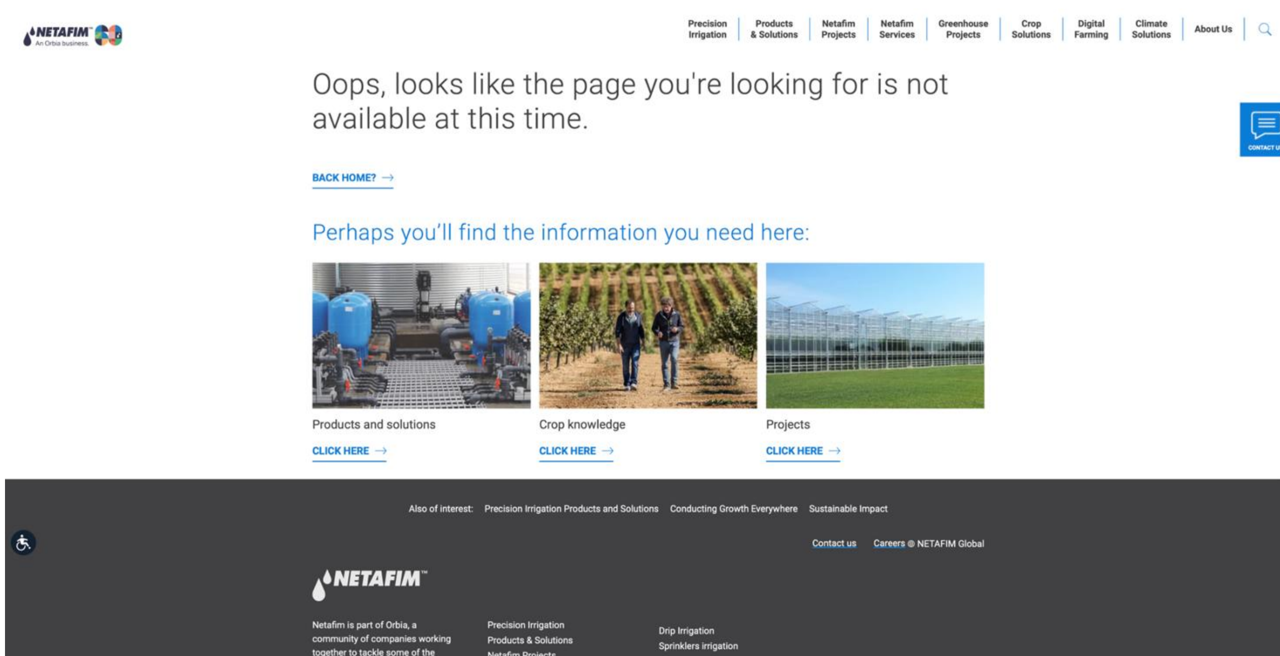

Рис. 1.2 – Недоступність Netafim Solutions (404)

Більш відкритим є інтерфейс системи AGRIVI, яка пропонує модулі AI-рекомендацій, управління господарством і дашборди з геопросторовими аналітичними шарами. Її функціонал охоплює також розсилку попереджень про ризики, формування агротехнічних рекомендацій і хмарну інтеграцію з метеосервісами (рис. 1.3).

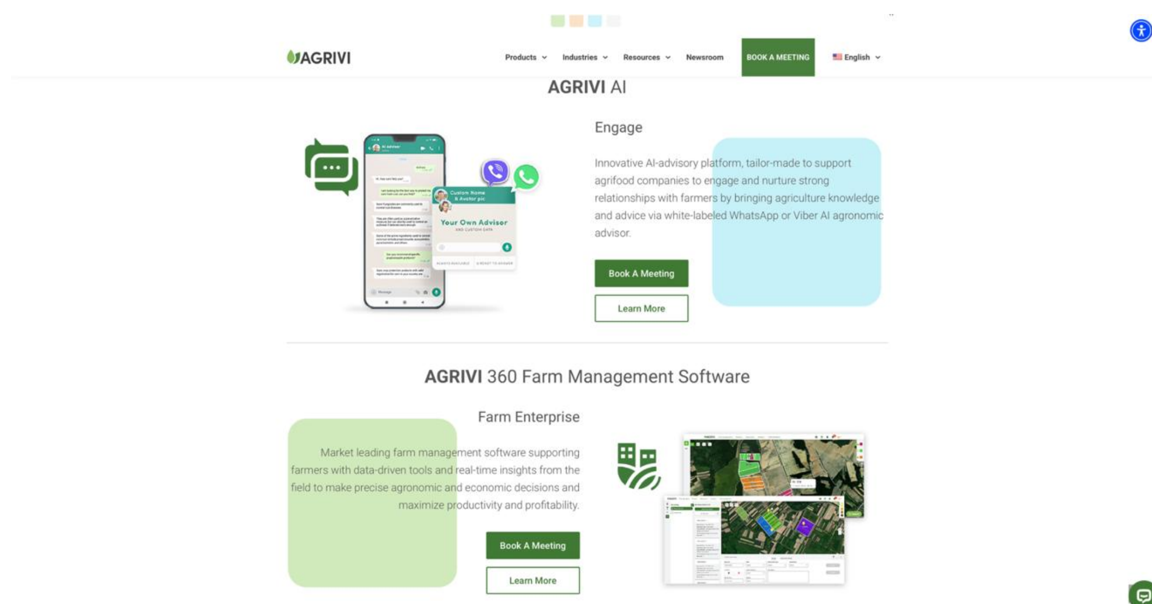

Рис. 1.3 – Платформа AGRIVI 360 Farm Management

Інше відоме рішення - Priva - також виявилось частково недоступним, що зафіксовано на рис. 1.4. Така ситуація знову підкреслює обмежену відкритість промислових постачальників інформаційних платформ.

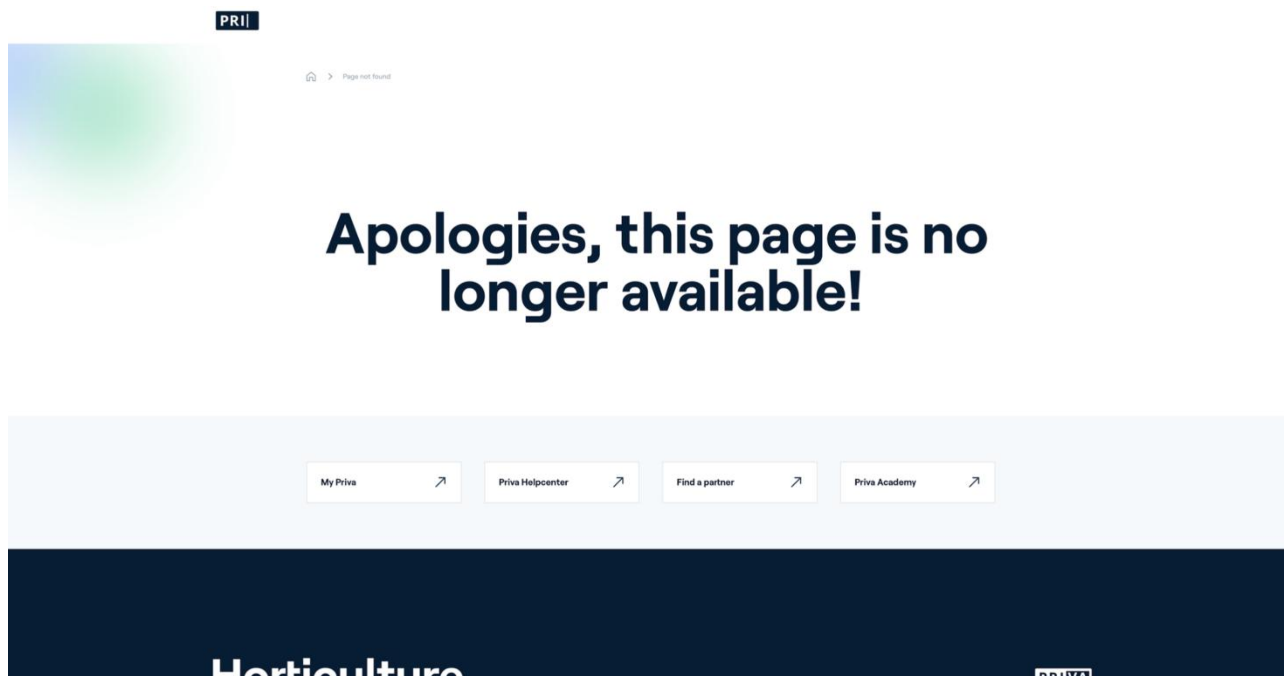

Рис. 1.4 – Відсутність сторінки рішень Priva Horticulture

Значний інтерес для дослідження становить платформа OpenAg Brain, архітектура якої реалізує принципи розповсюдження даних із сенсорів через ROS та зберігання в CouchDB. Як показано на рис. 1.5, система підтримує подієві протоколи MQTT/Webhooks і модульну реалізацію інтерфейсів для автоматизованих теплиць на базі Raspberry Pi.

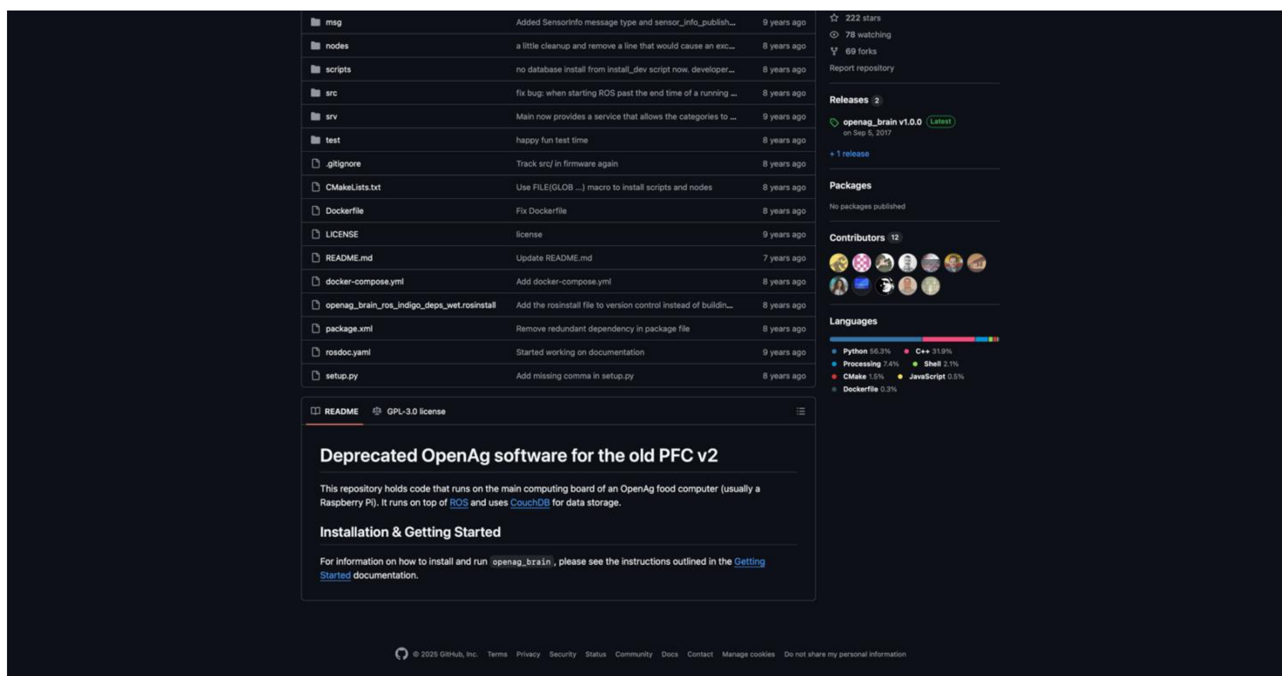

Рис. 1.5 – Відкрите сховище OpenAg Brain (GitHub)

Для порівняльного аналізу сформовано таблицю 1.2, яка дозволяє оцінити рівень відкритості, підтримку стандартів та інтеграційний потенціал систем.

Таблиця 1.2 – Порівняння рішень в контексті тепличного моніторингу

Критерій	Netafim	Agrivi 360	OpenAg Brain	Наша система
Доступність документації	Обмежена	Частково відкрита	Повна (GitHub)	Відкрита
Підтримка реального часу	Так	Частково	Так	Так
Подієві канали (MQTT/Webhook)	Недоступно	Ні	Так	Так
Інтеграція з ERP/SCADA	Так	Ні	Ні	Так
Протоколи API	Пропрієтарні	REST	ROS/MQTT	REST/GraphQL
Рівень кастомізації	Мінімальний	Середній	Високий	Повна підтримка правил
Режими керування	Автоматичний	Advisory	Partial	Advisory + автоматизований

З науково-технічного погляду системи розповсюдження інформації у тепличному середовищі дедалі частіше аналізуються в контексті модульних архітектур з низькою затримкою, розширюваних API і розподілених каналів передачі повідомлень. У публікаціях [1], [2] основна увага зосереджена на

реалізації мікросервісних платформ з використанням технологій MQTT, REST, AMQP і event-driven взаємодій. Відзначається важливість побудови декомпозованих модулів (агрегація сенсорів, аналітика, маршрутизація, сповіщення), кожен із яких реалізується у вигляді незалежного сервісу з можливістю гнучкого масштабування та керування навантаженням. Ключовим принципом є обробка даних із використанням потоку подій, адаптивних правил маршрутизації та уніфікованої моделі подій.

Інша група досліджень [3], [4] концентрується на гетерогенності джерел інформації (сенсори, метеостанції, ручне введення, SCADA API) і механізмах їх синхронізації у реальному часі. Автори описують архітектури з буферизованою доставкою повідомлень, що поєднують кешуючі брокери (Redis, Kafka), модулі розсилки (email/SMS/push), а також REST-сервіси для публікації даних у зовнішні ERP/ECM-системи. У цих роботах також акцентується на важливості інтеграції компонентів RBAC, SSO, логування та SLA-контролю як обов’язкових елементів для прозорості й масштабованості рішень.

Таким чином, наукові джерела демонструють сталий перехід від централізованих монолітних систем до подієвих розподілених екосистем, у яких ключову роль відіграють асинхронна взаємодія, адаптивна.

1.3 Моделювання предметної області

Опис предметної області системи розповсюдження інформації про стан процесів у тепличному господарстві здійснюється через формалізацію функціональних сценаріїв на основі UML-нотацій. На Рис. 1.5 представлено діаграму варіантів використання, де фіксуються взаємодії користувачів (оператор теплиці, агроном, аналітик, адміністратор) з окремими функціональними підсистемами, зокрема: отримання та збереження телеметрії (T, RH, CO_2, Lux), налаштування порогових політик, виявлення відхилень, генерація повідомлень та передача керуючих команд. Оператор ініціює процеси моніторингу і реагування, аналітик аналізує зібрані дані через дашборд і формує

звіти, адміністратор конфігурує правила валідації, маршрути повідомлень (SMS/Email/Push), рівні доступу (SSO/RBAC) та API інтеграцію з ERP-системами або мобільним додатком.

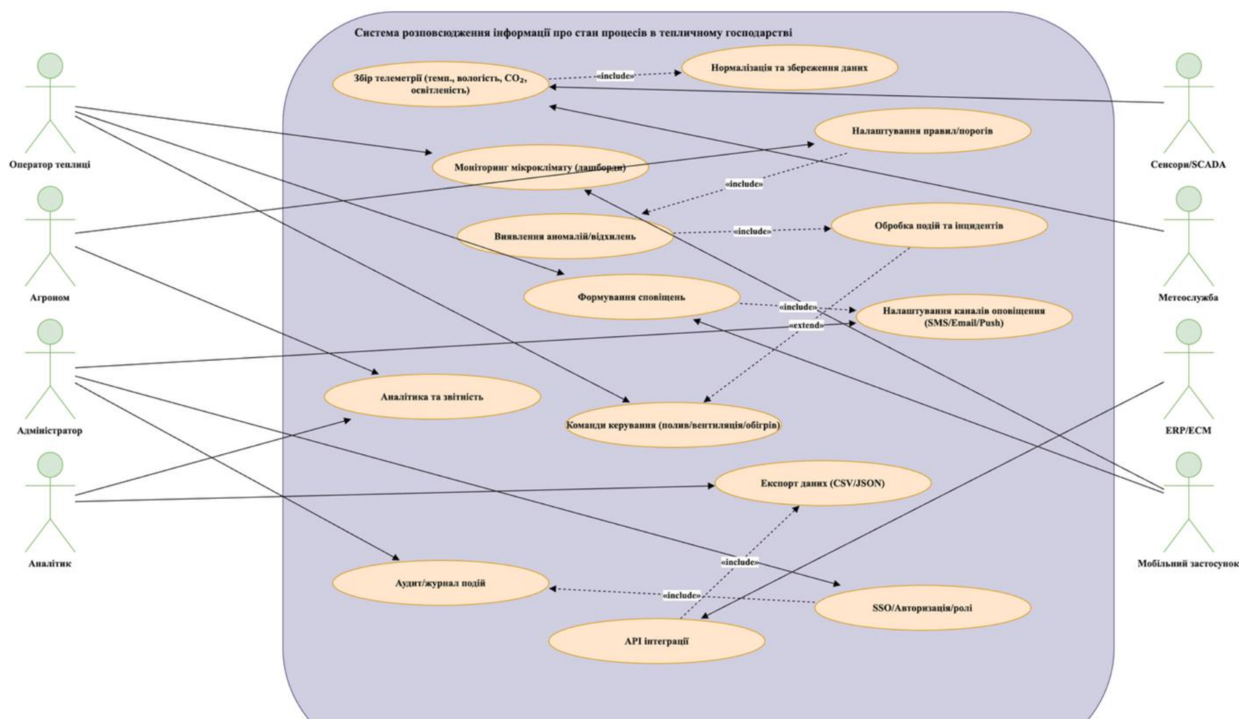

Рис. 1.5 – Діаграма варіантів використання системи сповіщення про стан тепличного середовища

Для відображення послідовності дій при обробці телеметрії реалізовано діаграму послідовності (Рис. 1.6). Після отримання даних із SCADA або метеослужб, сервіс збору виконує нормалізацію та формує запис у сховищі. Подальша оцінка відбувається через виклик правилової підсистеми, яка порівнює значення з конфігурованими політиками. У разі перевищення допустимих меж активується модуль виявлення аномалій, який повертає статус (OK, warning, critical) та рівень критичності (severity). Якщо severity ≥ warning, ініціюється формування повідомлення й керуючих команд, які надсилаються сервісу виконавчих механізмів (полив/вентиляція/обігрів). Результат також передається на дашборд мобільного або десктопного клієнта з підтвердженням (200 OK).

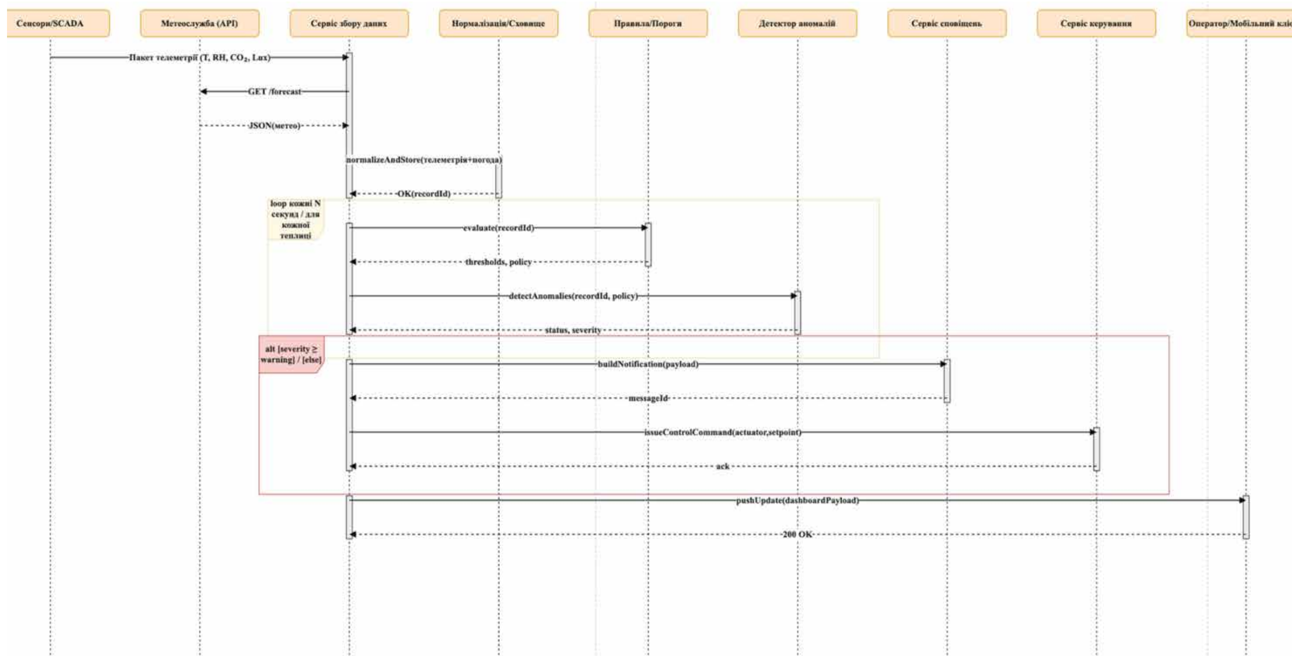

Рис. 1.6 – Діаграма послідовності обробки телеметрії, аномалій та команд керування

Алгоритмічна логіка системи деталізується через діаграму активності (Рис. 1.7). Вхідними параметрами є дані про температуру, вологість, CO_2 та освітленість, які проходять перевірку валідності. У разі помилки ініціюється гілка журналювання. За успішної валідації запускається оцінка політик, результатом якої є класифікація рівня аномалії. Якщо severity ≥ warning, формується повідомлення (payload), яке доставляється користувачу через інтегровані канали (SMS, email, push), та запускається цикл передачі команд виконавчим пристроям. В іншому випадку оновлюється лише дашборд користувача. Паралельно забезпечується експорт інформації в API або файл (CSV/JSON) для зовнішніх систем.

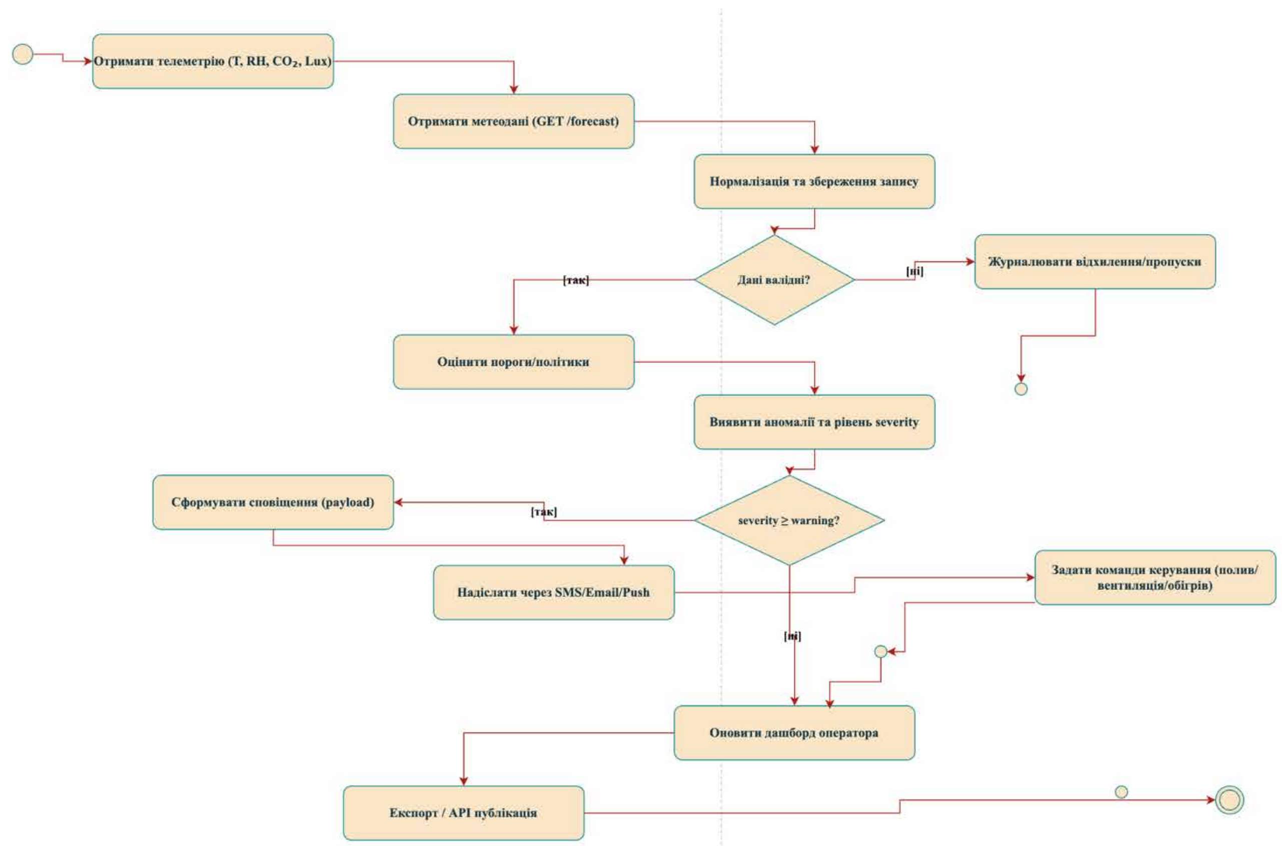

Рис. 1.7 – Діаграма активності логіки валідації, аналізу і реагування на зміни мікроклімату

Представлене моделювання закладає основу для розгортання системи в умовах розподіленого середовища з підтримкою real-time сповіщень, адаптивного реагування на критичні ситуації та гнучкого конфігурування правил і каналів доставки подій. Кожен компонент діаграм відповідає конкретному модулю майбутньої реалізації з чітко визначеними точками інтеграції, протоколами взаємодії та умовами обробки виключень.

1.4 Аналіз вимог програмної системи

Аналіз вимог до системи розповсюдження інформації про стан процесів у тепличному господарстві здійснено на основі попереднього моделювання предметної області (див. підпункт 1.3), що дозволило формалізувати ключові функціональні сценарії, визначити критерії до експлуатаційної якості та технічні обмеження щодо впровадження. Програмна система повинна забезпечити наскрізний цикл: отримання та нормалізація телеметричних даних, оцінка

порогових значень, виявлення аномалій, генерація керуючих команд та їх розповсюдження через інтегровані канали взаємодії. Сформовані вимоги структуровано у три категорії: функціональні, нефункціональні та технічні.

Функціональні вимоги відображають перелік бізнес-функцій, які система повинна підтримувати у штатному режимі. До таких відносяться: агрегація телеметрії з різних джерел (SCADA, API метеослужб), перевірка валідності даних, динамічна оцінка стану на основі налаштованих політик, виявлення відхилень, формування повідомлень з параметрами подій, передача команд управління обігрівом, вентиляцією або поливом, а також забезпечення взаємодії з користувачем через дашборд і канали сповіщень (SMS, email, push). Повний перелік функціональних вимог подано в Таблиці 1.1.

Таблиця 1.1– Функціональні вимоги до системи

	Функціональна вимога	Призначення
F1	Отримання даних з сенсорів/SCADA	Забезпечення моніторингу в режимі реального часу
F2	Отримання прогнозу погоди з метео-API	Адаптація політик під зовнішні умови
F3	Нормалізація та валідація даних	Стандартизація форматів і виявлення помилок
F4	Виявлення відхилень і визначення рівня критичності	Формування умов для реагування
F5	Формування сповіщень (payload)	Створення контексту для нотифікації та керування
F6	Надсилання команд управління	Взаємодія з пристроями поливу, вентиляції, обігріву
F7	Візуалізація даних через дашборд	Оперативна оцінка стану оператором або агрономом
F8	Експорт подій/API публікація	Сумісність з зовнішніми ERP/SCADA системами
F9	Аудит/логування подій	Відстеження дій користувачів і системних операцій

Нефункціональні вимоги фіксують характеристики якості, які забезпечують працездатність і стійкість системи в умовах промислової експлуатації. Основними вимогами є: висока доступність (SLA $\geq$ 99.5%),

підтримка асинхронного оброблення подій, гарантоване масштабування для обробки телеметрії з кількох теплиць, захист даних через SSO/автентифікацію, а також підтримка реакції на події з затримкою не більше 5 секунд. Перелік нефункціональних вимог наведено в Таблиці 1.2.

Таблиця 1.2-Нефункціональні вимоги до системи

№	Вимога	Значення/опис
N1	Доступність	≥ 99.5% (відмовостійке розгортання)
N2	Час реакції на подію	≤ 5 секунд
N3	Масштабованість	Підтримка ≥ 100 теплиць без деградації продуктивності
N4	Захист доступу	SSO + RBAC з можливістю інтеграції з корпоративними IdP
N5	Обробка в режимі near-real-time	Затримка push-сповіщення не більше 3 секунд
N6	Підтримка офлайн-дублювання подій	Queue-зберігання з повторною доставкою після відновлення зв’язку

Технічні вимоги деталізують обмеження щодо архітектури, інструментарію та середовища розгортання. Запропонована реалізація базується на мікросервісному підході з REST-взаємодією між компонентами, використанням брокерів подій (наприклад, MQTT або RabbitMQ) для комунікації з виконавчими пристроями, базою даних sqlLte, інтерфейсом на основі React або PyQt/WebView та хмарним або гібридним варіантом розгортання. Системні залежності і платформи узагальнено в Таблиці 1.3.

Таблиця 1.3– Технічні вимоги до реалізації системи

№	Компонент	Технологія / Вимога
T1	Серверна логіка	Python + FastAPI
T2	Кешування	Redis
T3	Сховище даних	sqlLite (з підтримкою time-series індексації)
T4	Клієнтський інтерфейс	PyQt/WebView або React SPA
T5	Черги повідомлень	Celery або MQTT (для інтеграції з обладнанням)
T6	API-документація	OpenAPI 3.1 (автоматично через FastAPI)
T7	CI/CD конвеєр	Docker + GitHub Actions або GitLab CI
T8	Протоколи комунікації	REST/HTTPS, WebSocket, MQTT
T9	Розгортання	Linux-контейнери (Docker Compose / Kubernetes)

Вимоги охоплюють повний спектр - від взаємодії з польовими пристроями до аналітики, масштабованості та безпеки, що забезпечує готовність системи до промислового впровадження у тепличних господарствах з гетерогенними інфраструктурами та критичними вимогами до спостережуваності та швидкодії.

1.5 Постановка завдання

На основі попереднього аналізу сформовано конкретизоване завдання, що передбачає побудову цифрової інфраструктури для збору, обробки та розповсюдження мікрокліматичних даних тепличного середовища. Первинні дані мають надходити з сенсорних вузлів SCADA-систем, метеоAPI або локальних контролерів, бути нормалізованими, перевіреними на валідність і збереженими у високодоступному сховищі. Потік телеметрії обробляється асинхронно з використанням черг повідомлень, з підтримкою буферизації та повторної доставки.

Надалі виконується аналіз кожного запису відповідно до заздалегідь визначених порогів і політик, що зберігаються у системному реєстрі правил. У разі виявлення аномалій ініціюється формування структурованого payload з технічними параметрами інциденту, рівнем критичності, геолокацією та мітками часу. Залежно від типу події виконується маршрутизація повідомлень через модулі SMS, Email або Push-повідомлень з контролем підтвердження доставки.

Для забезпечення оперативного реагування реалізується сервіс керування актуаторами з підтримкою передачі команд типу «включення поливу», «корекція температури», «активація вентиляції». Керування здійснюється на основі поточного стану, рекомендованих сценаріїв та відповідної прив’язки до логіки теплиці. Оновлені значення конфігурацій негайно передаються на пристрої керування та відображаються в інтерфейсі оператора.

Компоненти інтерфейсу реалізуються як модульний SPA-додаток з підтримкою прав доступу (RBAC), авторизації через SSO (OIDC), логуванням

дій користувачів та налаштуванням персоналізованих сценаріїв оповіщень. Дані відображаються у вигляді діаграм часу, віджетів реального стану, карт аномалій і історичних трендів.

Для аналітичних завдань реалізується генерація агрегованих звітів, автоматичний експорт у CSV/JSON, API-доступ до історичних даних, побудова індексів доступу та системи оцінки стабільності параметрів. Реалізуються шлюзи для інтеграції з ERP або ECM через REST або Webhooks, а також налаштовувані правила обміну.

Архітектура реалізується у форматі мікросервісів з ізоляцією компонентів (сховище, аналітика, сповіщення, керування, авторизація). Передбачено використання Python/FastAPI, SqlLite як основної СУБД, Redis для кешування, Celery для керування фоновими задачами, Docker та GitHub Actions для CI/CD. Система масштабована горизонтально, підтримує відмовостійкість і функціонує з середнім SLA ≥99.5%.

1.6 Висновки до розділу 1

У першому розділі проведено системний аналіз предметної області розповсюдження інформації про стан процесів у тепличному господарстві, що дозволило обґрунтувати необхідність створення цифрової платформи з підтримкою потокового моніторингу, автоматизованого виявлення аномалій та інтегрованих каналів сповіщення. На основі класифікації існуючих рішень показано обмеження ручних та традиційних SCADA-систем щодо масштабованості, гнучкості налаштувань, інтеграції з зовнішніми сервісами й забезпечення інформаційної безпеки, тоді як запропонована система орієнтована на подієву архітектуру, багатоканальні сповіщення та підтримку сучасних протоколів взаємодії. Огляд промислових і відкритих платформ (Netafim, AGRIVI, OpenAg Brain тощо) засвідчив фрагментарність документації, пропрієтарність інтерфейсів та недостатній рівень кастомізації правил реагування, що сформувало вимоги до відкритості API, підтримки

MQTT/Webhook-каналів, можливостей інтеграції з ERP/SCADA й адаптивного налаштування політик. UML-моделювання предметної області у вигляді діаграм варіантів використання, послідовності та активності дало змогу формалізувати ролі користувачів, сценарії обробки телеметрії, логіку валідації й реагування, включаючи маршрутизацію подій до виконавчих механізмів і зовнішніх систем.

Сформовані функціональні, нефункціональні та технічні вимоги зафіксували цільові показники щодо SLA, часу реакції, масштабованості, безпеки доступу та технологічної бази реалізації (Python/FastAPI, SQLite, MQTT, Docker), а постановка завдання визначила структуру майбутньої системи як мікросервісної платформи з асинхронною обробкою подій, централізованим сховищем даних і модульною підсистемою сповіщень. Сукупність отриманих результатів створює теоретичне й прикладне підґрунтя для подальшого проєктування інформаційного та програмного забезпечення, що реалізується у наступних розділах роботи.

2 ПРОЕКТУВАННЯ ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Логічна модель даних у вигляді ER-діаграми

Раціональна організація даних у системах моніторингу довкілля потребує створення узгодженої логічної структури, здатної забезпечити точність, відмовостійкість і цілісність взаємопов’язаних інформаційних потоків. Логічна модель даних у вигляді ER-діаграми (рис. 2.1) відображає взаємозв’язки між сутностями, які беруть участь у процесах збору, аналізу, реагування та розповсюдження інформації.

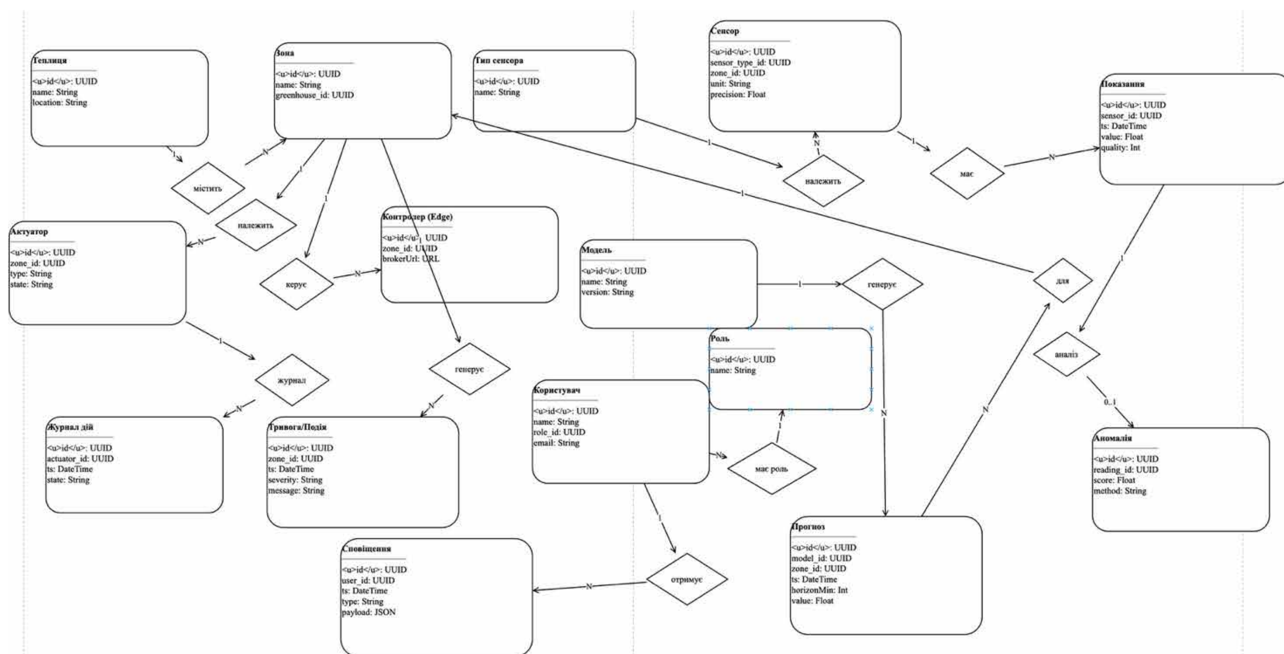

Рис.2.1 – Логічна модель даних у вигляді ER-діаграми

Основою моделі є концепція багаторівневої ієрархії, де нижчі рівні - сенсорні вузли - генерують потік телеметрії, середні - забезпечують її обробку та зберігання, а верхні - здійснюють аналітику, формування сповіщень і керувальні дії. Система спроєктована так, щоб кожен об’єкт мав унікальний ідентифікатор, пов’язаний із вищим рівнем через зовнішні ключі, що гарантує відстежуваність усіх подій у часовій послідовності.

У межах моделі виокремлено кілька груп сутностей. Адміністративний рівень представлено сутностями GREENHOUSE і ZONE, які визначають географічну та структурну ідентифікацію об’єктів. Операційний рівень утворюють SENSOR, MEASUREMENT і THRESHOLD_POLICY, що відповідають за збір даних, їх часову фіксацію та контроль допустимих діапазонів. Реактивний рівень включає ALERT, ACTUATOR і COMMAND, які забезпечують механізм зворотного зв’язку, автоматичне реагування на відхилення й виконання керувальних команд. На рівні доступу й безпеки функціонують USER, ROLE та зв’язка USER_ROLE, які реалізують контроль повноважень через RBAC-модель і протоколи автентифікації. Така структуризація дозволяє мінімізувати дублювання інформації, полегшити масштабування та підтримати багатопоточну обробку даних у режимі near-real-time [10].

Таблиця 2.1 – Основні сутності логічної моделі та їхні атрибути

Сутність	Основні атрибути	Тип зв’язку	Призначення
GREENHOUSE	id, name, location	1⟶* ZONE	Верхній рівень ідентифікації об’єкта
ZONE	id, greenhouse_id, name	1⟶* SENSOR, ACTUATOR	Територіальна зона контролю
SENSOR	id, zone_id, type, unit	1⟶* MEASUREMENT	Джерело даних мікроклімату
MEASUREMENT	id, sensor_id, ts, value	*⟶1 SENSOR	Часові ряди вимірювань
THRESHOLD_POLICY	id, zone_id, metric, warn, crit	1⟶* ALERT	Правила оцінки параметрів
ALERT	id, measurement_id, severity, status	*⟶1 POLICY, *⟶1 MEASUREMENT	Запис аномалій і тригер керування
ACTUATOR	id, zone_id, type, state	1⟶* COMMAND	Виконавчий пристрій реагування
COMMAND	id, actuator_id, action, params	*⟶1 ALERT, *⟶1 USER	Керувальна дія з підтвердженням виконання
USER	id, name, role, contact	⟷ ROLE	Суб’єкт доступу та контролю

У моделі реалізовано принципи нормалізації до третьої нормальної форми, що забезпечує незалежність логічних структур від фізичного рівня реалізації, а також узгодження ключових обмежень для підтримки транзакцій ACID-типу. Кожен запис вимірювання містить часову мітку, пов'язану із сенсором, що дає змогу здійснювати повнотекстовий та інтервальний пошук за періодами. Таблиці ALERT і COMMAND містять поля кореляції для ідентифікації причинно-наслідкових подій, що підвищує спостережуваність системи під час аналізу інцидентів. Вбудована підтримка індексів за часовими мітками та ідентифікаторами зон оптимізує виконання запитів при роботі з потоковими даними. Для підвищення стійкості модель передбачає дублювання критичних таблиць у read-replica-сховищах і використання журналів змін (change-data-capture) для синхронізації між сервісами [2]. Завдяки цьому структура даних зберігає логічну узгодженість, забезпечує високу доступність і дозволяє масштабувати обробку без втрати продуктивності, що є визначальним чинником для промислових IoT-систем моніторингу.

2.2 Діаграма класів і кооперації

Архітектура програмного забезпечення побудована за принципами об'єктно-орієнтованого проєктування, де кожен клас виконує ізольовану функціональну роль у межах розподіленої системи моніторингу та реагування. Логічна взаємодія компонентів відображена на діаграмі класів (рис. 2.2), що деталізує атрибути, методи та зв'язки між об'єктами Sensor, EdgeController, TimeSeriesDB, AnalyticsModule, AnomalyDetec tor, ForecastModel, NotificationService, User, Role, AuthService, PolicyEngine, Das hboard і Actuator. Клас Sensor ініціює формування показників у вигляді об'єктів типу SensorReading, які надсилаються до EdgeController, що виконує нормалізацію даних і публікує їх у брокері повідомлень MQTT. Контролер здійснює подальшу передачу показників до TimeSeriesDB, де забезпечується тривале зберігання з параметрами утримання, пулом з'єднань і оптимізованими

запитами до часових рядів. Об’єкти Actuator отримують сигнали через методи setState() і getState(), а взаємодія між ними забезпечує замкнутий цикл автоматизації, керований політиками з PolicyEngine.

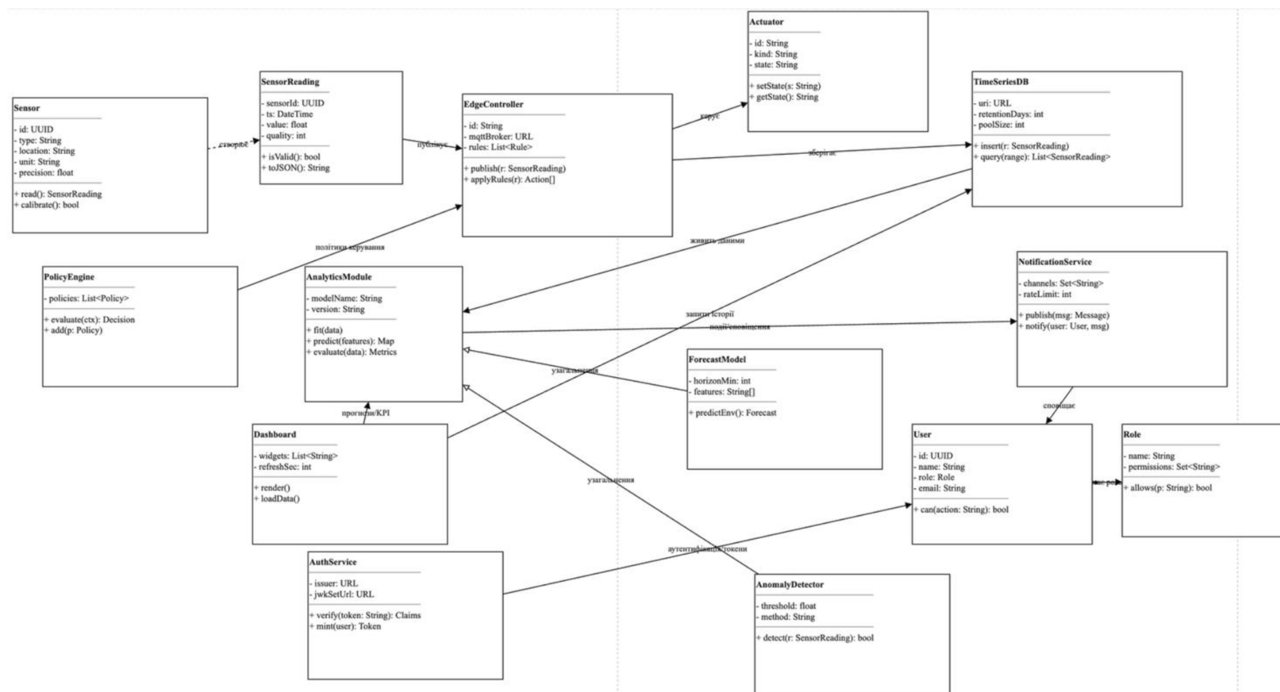

Рис. 2.2 – Діаграма класів системи моніторингу та розповсюдження подій

Логіка взаємодії між класами формалізована на рівні сценаріїв кооперації, представлених у таблиці 2.2, яка містить короткий опис ролей компонентів і характер їхніх зв’язків.

Таблиця 2.2 – Взаємозв’язки між компонентами системи

Клас	Взаємодіє з	Тип взаємодії	Результат виконання
Sensor	EdgeController	Передача вимірювань	Формування об’єкта SensorReading
EdgeController	TimeSeriesDB	Публікація даних	Збереження показників у базі
AnalyticsModule	AnomalyDetector, ForecastModel	Оцінка даних, прогнозування	Виявлення відхилень, формування подій
NotificationService	User, Dashboard	Сповіщення та оновлення інтерфейсу	Надсилання повідомлень і оновлення UI
AuthService	User, Role	Перевірка токенів і прав доступу	Авторизація користувача

Як показано на рис. 2.3, сенсор формує вимірювальні дані та передає їх через контролер на рівень сховища часових рядів, після

чого EdgeController отримує підтвердження запису, забезпечуючи достовірність обміну.

Рис. 2.3 – Кооперація між Sensor, EdgeController і TimeSeriesDB

На рис. 2.4 наведено кооперацію між модулями аналітики, де AnalyticsModule взаємодіє з AnomalyDetector для класифікації відхилень та з ForecastModel для прогнозування поведінки параметрів середовища. Результати аналізу публікуються як події, що передаються до NotificationService для подальшого розповсюдження.

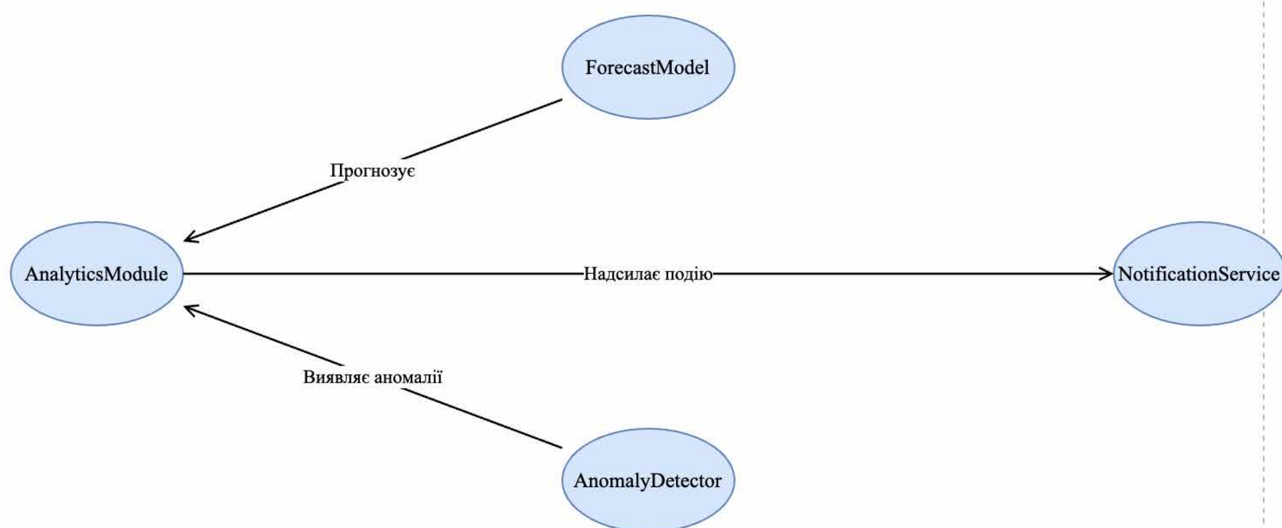

Рис. 2.4 – Кооперація між AnalyticsModule, ForecastModel і AnomalyDetector

Далі на рис. 2.5 зображено взаємодію служби сповіщень із користувачем і хмарним API, де NotificationServiceприймає агреговані події, формує повідомлення та оновлює інтерфейс Dashboard, використовуючи канали, визначені в полі channels:Set<String>. Дані надходять із CloudAPI, який виконує функцію синхронізації між обчислювальними модулями й клієнтськими додатками.

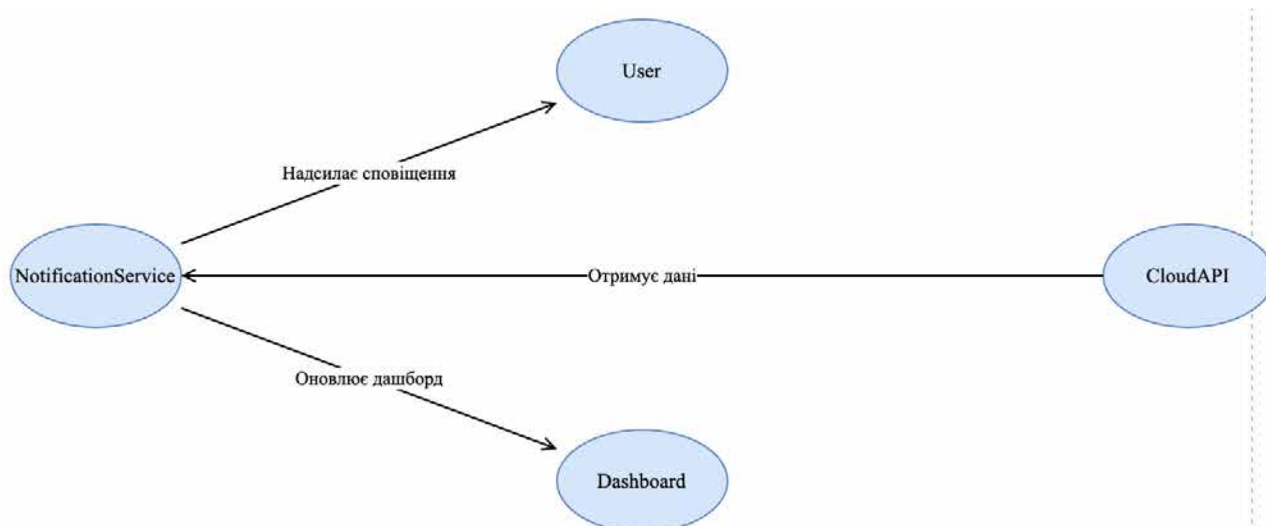

Рис. 2.5 – Взаємодія NotificationService, User, CloudAPI та Dashboard

Запропонована модель класів і кооперацій забезпечує низький рівень зв'язності компонентів, високу розширюваність та підтримку подієво-орієнтованого підходу, де обмін інформацією відбувається асинхронно через брокери або REST-інтерфейси. Завдяки цьому підвищується стійкість до збоїв, зменшується час реакції на події та забезпечується узгоджене оновлення станів у реальному часі, що є необхідною умовою для промислового застосування систем керування навколишнім середовищем у тепличних комплексах.

2.3 Діаграма компонентів розроблювальної системи

Компонентна архітектура системи реалізована як сукупність незалежних, але взаємопов'язаних модулів, кожен із яких відповідає за окрему стадію життєвого циклу даних - від збору телеметрії до розповсюдження аналітичних результатів. На діаграмі компонентів (рис. 2.6) наведено взаємозв'язки між основними
підсистемами: SensorAPI, EdgeController, AnalyticsModule, CloudAPI, PublishAP I, UserHTTP, а також між відповідними інтерфейсами обміну - MQTT/OPC UA, REST/DB API і AnalyticsAPI. Компоненти розміщено у логічній послідовності, що відповідає послідовному потоку даних: зчитування параметрів із сенсорів → передача через контролер → обробка аналітичним модулем →

формування сповіщень → візуалізація результатів у користувацькому інтерфейсі.

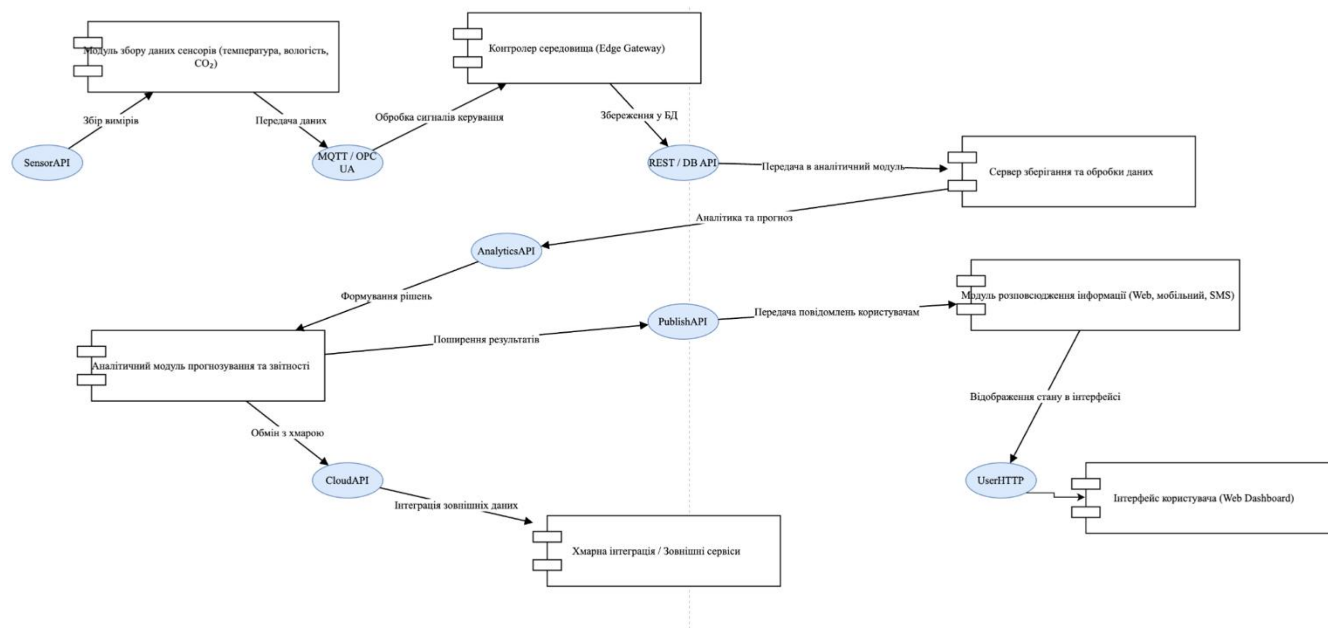

Рис. 2.6 – Компонентна діаграма системи моніторингу, аналітики та розповсюдження даних

Згідно з архітектурною моделлю, модуль збору даних сенсорів відповідає за ініціалізацію опитування пристроїв, конвертацію показників температури, вологості, концентрації CO_2 та освітленості у цифрову форму й передачу їх у контролер середовища (Edge Gateway). Комунікація здійснюється за допомогою протоколів MQTT або OPC UA, що забезпечують низьку латентність і підтримку асинхронного обміну повідомленнями. Контролер виконує обробку сигналів керування та збереження агрегованих даних у базі за допомогою REST/DB API, який підтримує транзакційну модель з ACID-гарантіями. Далі дані надходять у аналітичний модуль прогнозування та звітності, що взаємодіє з базою через AnalyticsAPI для проведення статистичного аналізу, побудови прогнозів і генерації аналітичних показників.

Для структурного узагальнення взаємодій компонентів наведено табл. 2.3, яка відображає їхні функціональні призначення та канали інтеграції.

Таблиця 2.3 – Функціональні характеристики компонентів системи

Компонент	Основна функція	Інтерфейс взаємодії	Результат виконання
SensorAPI	Опитування сенсорів, фільтрація шумів	MQTT / OPC UA	Передача показників у контролер
EdgeController	Збір і нормалізація телеметрії	REST / DB API	Збереження даних у сховище
AnalyticsModule	Аналіз і прогноз параметрів	AnalyticsAPI	Формування рішень і звітів
PublishAPI	Розповсюдження аналітичних результатів	HTTP / WebSocket	Надсилання повідомлень користувачам
CloudAPI	Інтеграція з хмарними сервісами	JSON / REST Gateway	Обмін зовнішніми даними
UserHTTP	Інтерфейс користувача	HTTPS / Web Dashboard	Відображення поточного стану системи

Як показано на рис. 2.6, PublishAPI забезпечує двонаправлену взаємодію між аналітичним модулем і модулем розповсюдження інформації, ініціюючи події при зміні станів або виявленні критичних відхилень. Повідомлення формуються у форматі JSON і передаються через push-канали або SMS-шлюзи, після чого модуль розповсюдження синхронізує результати з користувацьким інтерфейсом за допомогою UserHTTP. Одночасно CloudAPI інтегрує систему з зовнішніми сервісами прогнозування та агрометеорологічними базами, використовуючи REST-ендпоїнти для обміну даними в обох напрямках.

Така компонентна структура дозволяє розподілити обчислювальні навантаження між edge-рівнем, аналітичним сервером і клієнтським інтерфейсом, зменшуючи затримки в комунікації та підвищуючи стійкість системи до відмов. Завдяки слабкому зв’язуванню компонентів та стандартизованим API, архітектура підтримує горизонтальне масштабування, розгортання у контейнерах (Docker / Kubernetes) і централізований моніторинг потоків даних. Такий підхід забезпечує ефективне функціонування інформаційної системи в умовах промислового середовища з високою частотою оновлення параметрів.

2.4 Діаграма пакетів

Архітектура програмного забезпечення структурована за принципом модульної декомпозиції, що передбачає розподіл системи на логічні пакети з чітко визначеними зонами відповідальності та мінімальним міжмодульним зв’язуванням. Діаграма пакетів (рис. 2.7) відображає ієрархічну організацію модулів, які забезпечують збір, обробку, аналітику, зберігання та візуалізацію даних у єдиному інформаційному просторі.

До складу системи входять п’ять основних пакетів: data.collector, data.processing, analytics.core, communication.services та ui.dashboard, кожен із яких містить підпакети, що реалізують функціональну спеціалізацію. Такий підхід забезпечує ізоляцію логіки, підтримку інкапсуляції та можливість незалежної розробки компонентів без порушення загальної архітектури.

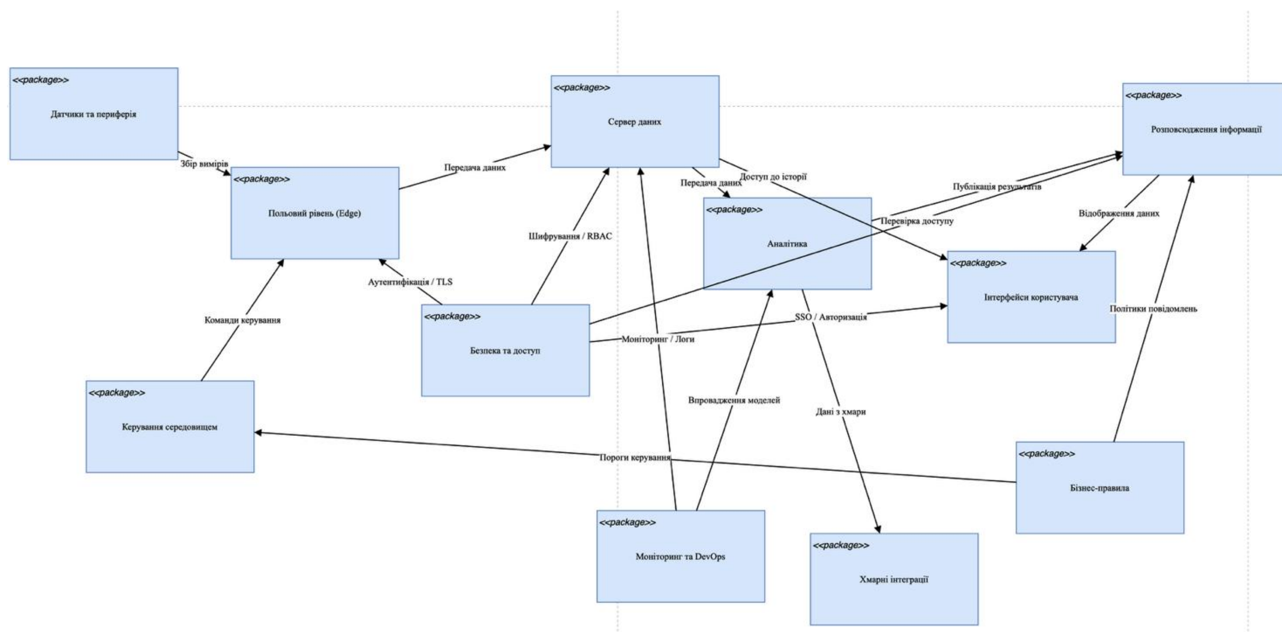

Рис. 2.7 – Діаграма пакетів інформаційної системи моніторингу та аналітики

Пакет data.collector містить підпакети sensor.api і controller.edge, що реалізують опитування сенсорів, отримання показників із середовища, попередню фільтрацію даних та передачу їх через протоколи MQTT або OPC UA. Пакет data.processing об’єднує класи для обробки сигналів, формування агрегованих записів, валідації даних і взаємодії з базою часових рядів. Основним

компонентом є repository.db, який забезпечує транзакційну цілісність та оптимізоване кешування запитів. Пакет analytics.core містить модулі anomaly.detector, orecast.model і policy.engine, що виконують функції виявлення відхилень, прогнозування показників та автоматичного прийняття рішень. Пакет communication.services включає notification.service, cloud.api та publish.api , які забезпечують обмін повідомленнями, інтеграцію з зовнішніми системами та передачу результатів користувачам. Нарешті, пакет ui.dashboard відповідає за відображення інформації у веб- та мобільному інтерфейсі, використовуючи REST-запити до аналітичних модулів та динамічне оновлення стану системи.

Для зручності наведено табл. 2.4, у якій подано короткий опис основних пакетів, їхніх підпакетів і функціональних ролей у межах архітектури.

Таблиця 2.4 – Характеристика пакетів програмної системи

Назва пакета	Підпакети	Основне призначення	Взаємодія з іншими пакетами
data.collector	sensor.api, controller.edge	Збір і передача даних від сенсорів	data.processing, analytics.core
data.processing	repository.db, data.filter	Збереження, нормалізація, валідація даних	analytics.core, communication.service s
analytics.core	anomaly.detector, forecast.model, policy.engine	Обчислення показників, прогнозування, реакція на події	data.processing, communication.service s
communication.service s	notification.service , publish.api, cloud.api	Розповсюдження аналітичних даних, інтеграція з хмарними ресурсами	analytics.core, ui.dashboard
ui.dashboard	web.client, mobile.ui	Відображення стану системи, звітність	communication.service s

Як видно з рис. 2.7, інформаційні потоки між пакетами мають напрям від data.collector до ui.dashboard, утворюючи наскрізний ланцюг обробки даних.

Пакети зв'язуються між собою через чітко визначені інтерфейси, що мінімізує залежності та спрощує тестування. Завдяки такій модульній структурі система підтримує повторне використання коду, розгортання окремих модулів у контейнерах і гнучке оновлення компонентів без переривання роботи сервісів. Така організація забезпечує масштабованість і стабільність програмного комплексу, що є критично важливими властивостями для розподілених інформаційних систем промислового рівня.

2.5 Висновки до розділу 2

У другому розділі сформовано цілісну концепцію інформаційного та програмного проєктування системи розповсюдження даних про стан процесів у тепличному господарстві, що охоплює логічну модель даних, об'єктно-орієнтовану структуру класів, компонентну архітектуру та модульну організацію програмного коду. Побудована ER-модель з ієрархією сутностей GREENHOUSE, ZONE, SENSOR, MEASUREMENT, THRESHOLD_POLICY, ALERT, ACTUATOR, COMMAND і USER/ROLE забезпечує нормалізацію до третьої нормальної форми, збереження цілісності зв'язків, відстежуваність причинно-наслідкових подій і оптимізований доступ до часових рядів телеметрії. Впровадження індексів за часовими мітками та зонами, а також підтримка механізмів реплікації і журналювання змін створюють основу для масштабованої та відмовостійкої роботи сховища даних у режимі near-real-time.

Діаграми класів, кооперацій, компонентів і пакетів формалізують структуру програмного забезпечення як сукупності слабко зв'язаних модулів, що взаємодіють через стандартизовані інтерфейси MQTT/OPC UA, REST/DB API, AnalyticsAPI та WebSocket. Виділення окремих рівнів data.collector, data.processing, analytics.core, communication.services та ui.dashboard забезпечує чіткий розподіл відповідальності між підсистемами, полегшує тестування, повторне використання коду й контейнеризацію компонентів. У результаті другий розділ формує узгоджену архітектурну основу, яка гарантує логічну

завершеність інформаційної моделі, гнучкість розширення функціоналу й готовність системи до промислового розгортання в умовах високочастотного IoT-моніторингу тепличного середовища.

3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Вибір технологій та інструментальних засобів реалізації системи

Реалізація системи розповсюдження інформації про стан процесів у тепличному господарстві передбачає використання комплексу технологій, здатних забезпечити повний цикл життєдіяльності даних — від зчитування сенсорної телеметрії до формування аналітичних звітів і розповсюдження сповіщень у режимі реального часу. Вибір інструментальних засобів здійснювався на основі аналізу сучасних рішень для IoT- та edge-аналітики з урахуванням критеріїв: надійність, простота розгортання, продуктивність при обмежених ресурсах та сумісність із мікросервісною архітектурою, поданою в розділі 2 (рис. 2.6).

Серверна частина системи розроблена мовою Python 3.12, що завдяки своїй розвиненій екосистемі бібліотек дозволяє інтегрувати обробку даних, аналітику, мережеву взаємодію та візуалізацію в єдиному середовищі. У ролі основного веб-фреймворку обрано FastAPI, який забезпечує неблокуючу (асинхронну) обробку запитів, вбудовану валідацію даних через Pydantic і автоматичне формування OpenAPI-документації. Для виконання фонових операцій, періодичного опитування сенсорів та асинхронної маршрутизації повідомлень застосовано Celery у поєднанні з брокером Redis, який виступає швидкодіючим кешем та чергою повідомлень з мінімальною затримкою.

Сховище даних реалізовано на базі SQLite 3.45, оскільки ця СУБД є вбудованою, не потребує сервера, підтримує стандарт SQL-92 і транзакційність ACID. Такий вибір зумовлений призначенням системи для edge-розгортань та автономних тепличних контролерів, де важливо забезпечити низьке споживання ресурсів і простоту налаштування без складної серверної інфраструктури. SQLite забезпечує цілісність і узгодженість даних, дозволяє працювати з часовими

рядами сенсорних вимірювань і підтримує індексацію за часовими мітками, що важливо для аналітичних запитів і фільтрації телеметрії [11].

Передача даних від сенсорів до контролера здійснюється за допомогою протоколу MQTT v5, що підтримує модель publish/subscribe і мінімальне навантаження на мережу. Для інтеграції з наявними системами промислової автоматизації реалізовано підтримку OPC UA, яка забезпечує сумісність із SCADA-вузлами та контролерами PLC. Комунікація з метеосервісами реалізується через REST API з TLS-шифруванням, що гарантує конфіденційність і цілісність даних.

Користувацька взаємодія реалізована через дві підсистеми інтерфейсу: PyQt6 GUI для локального оператора теплиці та веб-клієнт React 18 для аналітиків. Обидва інтерфейси отримують інформацію з центрального API та підтримують механізм push-оновлень через WebSocket, що дозволяє забезпечити роботу у near-real-time режимі. Для побудови графіків і діаграм використовуються бібліотеки Matplotlib та Plotly.js, які дають змогу візуалізувати динаміку мікроклімату, історичні тренди та сигнали аномалій.

Система контейнеризована за допомогою Docker Engine з оркестрацією Docker Compose, що спрощує розгортання і міграцію між середовищами (edge ↔ cloud). Процеси безперервної інтеграції та доставки (CI/CD) реалізовано через GitHub Actions, які автоматизують етапи тестування, перевірки стилю коду, формування Docker-образів і публікації оновлень. Для контролю працездатності системи застосовується Prometheus (збір метрик CPU, RAM, latency) у поєднанні з Grafana, що забезпечує дашборди моніторингу та SLA-візуалізацію.

З метою забезпечення кібербезпеки впроваджено JWT-автентифікацію та модель RBAC, що визначає права доступу користувачів до окремих функціональних модулів. Передбачено також підтримку єдиної авторизації (SSO OIDC) для інтеграції з корпоративними ідентифікаційними службами.

Узагальнені характеристики використаних технологій наведено в таблиці 3.1.

Таблиця 3.1 – Вибір технологій та інструментальних засобів реалізації системи

№	Компонент / підсистема	Обрана технологія	Обґрунтування вибору
1	Серверна логіка	Python 3.12 + FastAPI	Асинхронна модель, OpenAPI-документація, висока швидкодія
2	Фонові задачі	Celery + Redis	Обробка черг подій, кешування, гнучке масштабування
3	Сховище даних	SQLite 3.45	Легка ACID-СУБД для edge-пристроїв, відсутність серверної залежності
4	Протоколи взаємодії	MQTT v5 / OPC UA / REST	Надійна комунікація сенсорів і контролерів, шифрований обмін
5	Інтерфейс користувача	PyQt6 / React 18 + Matplotlib	Кросплатформені UI, інтерактивна аналітика
6	Контейнеризація та CI/CD	Docker + GitHub Actions	Простота розгортання, автоматизація збірки
7	Моніторинг і логування	Prometheus + Grafana	Збір і візуалізація метрик, контроль SLA
8	Безпека та автентифікація	JWT + RBAC + SSO (OIDC)	Централізований контроль доступу
9	API-інтеграції	REST / WebSocket	Стандартизований обмін із зовнішніми системами

Обраний технологічний стек повністю відповідає вимогам до розроблюваної системи: він забезпечує асинхронну обробку потоків даних, низьке енергоспоживання, захист інформації, контейнерну мобільність та можливість інтеграції як з локальними, так і з хмарними середовищами. Поєднання Python/FastAPI, SQLite та MQTT формує основу ефективного й відмовостійкого рішення для промислових тепличних комплексів, орієнтованого на подальше масштабування та аналітичне розширення.

3.2 Фізична модель даних і структура OLAP-кубу

Фізична модель даних і структура OLAP-кубу розробленої системи розповсюдження інформації про стан процесів у тепличному господарстві формують ядро аналітичної підсистеми, яка забезпечує багатовимірний аналіз телеметричних показників у режимі реального часу. Її побудова ґрунтується на принципах сховищ даних типу Data Warehouse та архітектурі зіркоподібної моделі (Star Schema), що дозволяє забезпечити високу швидкодію при виконанні аналітичних запитів, гнучкість структури та логічну ізольованість вимірів.

Основним призначенням OLAP-кубу є агрегація й аналіз ключових параметрів мікроклімату – температури, вологості, рівня CO_2, освітленості та метеофакторів – у часовому, просторовому, технічному та погодному вимірах. Його структура орієнтована на багаторівневу деталізацію, де виміри часу, простору, типу сенсора та метрики утворюють аналітичну вісь, що дозволяє проводити обчислення у будь-яких комбінаціях показників. У результаті користувач системи може отримати динамічні звіти, наприклад середню температуру за добу в окремій зоні, залежність вологості від зовнішнього вітру або розподіл рівня CO_2 по теплицях за тиждень. Загальну структуру OLAP-кубу, побудованого на основі цих вимірів, подано на рис. 3.1.

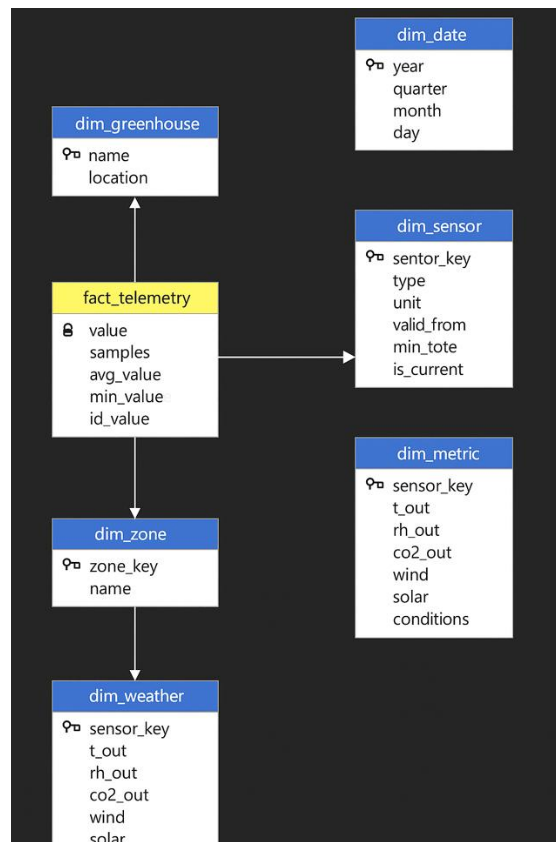

Рис. 3.1 – OLAP-куб системи моніторингу та аналітики тепличного середовища

Фізична модель даних реалізована у середовищі SQLite 3, що обрано завдяки її легкості, сумісності з edge-пристроями та здатності працювати без серверної інфраструктури, зберігаючи при цьому транзакційну цілісність (ACID). Модель реалізовано за схемою «зірки», у центрі якої знаходиться таблиця фактів fact_telemetry, пов'язана з шістьма вимірними таблицями – dim_date, dim_sensor, dim_zone, dim_metric, dim_weather, dim_greenhouse. Таблиця фактів містить числові параметри телеметрії: миттєве значення сенсора (value), кількість зразків (samples) та стан достовірності даних (is_valid). Зовнішні ключі пов'язують її з таблицями часу, простору, типу вимірювання, погодних даних і сенсорів, що дозволяє об'єднати всі аспекти процесу в єдину модель. Структуру фізичної моделі наведено на рис. 3.2.

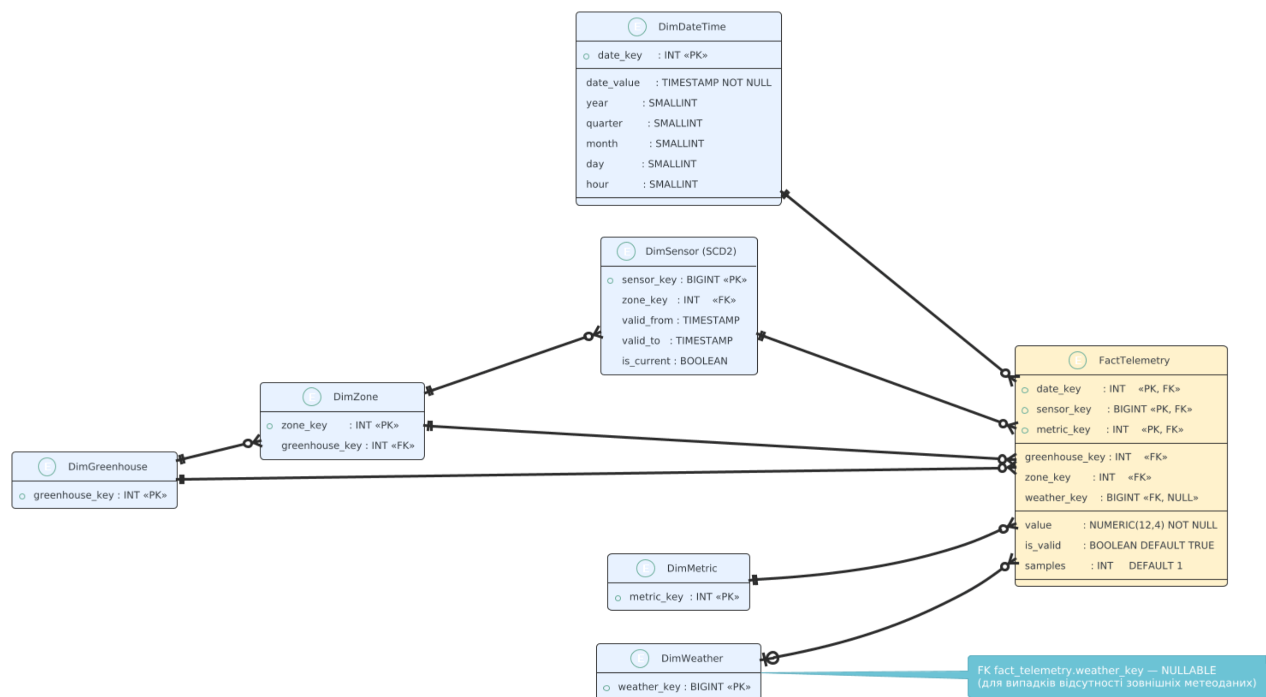

Рис. 3.2 – Фізична модель даних системи розповсюдження інформації у тепличному господарстві

Таблиці вимірів містять атрибути, що описують контекст даних: dim_date зберігає ієрархію часу (рік, місяць, день, година), dim_sensor – інформацію про сенсори та їхню прив’язку до зон, dim_zone – географічну структуру теплиці, dim_metric – перелік типів вимірювань із одиницями, dim_weather – метеорологічні показники для зовнішнього середовища, dim_greenhouse – загальні характеристики теплиці. Для підвищення продуктивності створено індекси за ключовими атрибутами (date_key, sensor_key, zone_key), а зовнішні зв’язки реалізовано з опцією ON UPDATE CASCADE, що гарантує референційну цілісність. У таблиці 3.2 наведено узагальнені характеристики основних елементів фізичної моделі.

Таблиця 3.2 – Основні таблиці фізичної моделі даних

№	Таблиця	Ключові поля	Типи даних	Призначення
1	fact_telemetry	date_key, sensor_key, metric_key, zone_key, value, samples, is_valid	INTEGER, NUMERIC(12,4), BOOLEAN	Центральна таблиця фактів, що акумулює результати вимірювань

Продовження таблиці 3.2

2	dim_date	date_key, year, month, day, hour	INTEGER, SMALLINT	Вимір часу з ієрархічною структурою агрегації
3	dim_sensor	sensor_key, zone_key, valid_from, valid_to, is_current	BIGINT, TIMESTAMP, BOOLEAN	Каталог сенсорів із контролем актуальності (SCD2)
4	dim_zone	zone_key, greenhouse_key, name	INTEGER, TEXT	Просторове розмежування зон теплиці
5	dim_metric	metric_key, metric_name, unit	INTEGER, TEXT	Опис вимірюваних параметрів (T, RH, CO_2, освітленість)
6	dim_weather	weather_key, t_out, rh_out, sola r, wind	BIGINT, NUMERIC(8,2)	Дані про зовнішні метеопоказник и
7	dim_greenhous e	greenhouse_key, name, location	INTEGER, TEXT	Ідентифікація теплиць і їх розташування

Побудована фізична модель даних дозволяє ефективно формувати агрегати та матеріалізовані подання для типових аналітичних запитів. У межах системи передбачено набір SQL-запитів, що реалізують різні аналітичні сценарії. На рис. 3.3 подано приклад запиту, який обчислює середню температуру по кожній зоні за останні шість годин.

```sql
SELECT
    z.name AS zone_name,
    ROUND(AVG(m.value), 2) AS avg_temperature
FROM MEASUREMENT m
JOIN SENSOR s ON m.sensor_id = s.id
JOIN ZONE z ON s.zone_id = z.id
WHERE s.type = 'temperature'
  AND m.ts >= DATETIME('now', '-6 hours')
GROUP BY z.name
ORDER BY avg_temperature DESC;
```

Рис. 3.3 – SQL-запит обчислення середньої температури по зонах

Запит, наведений на рис. 3.4, визначає активні критичні аномалії в теплицях із рівнем серйозності severity ≥ 2, що використовується для оперативних повідомлень у модулі сповіщень системи.

```sql
SELECT
    a.id AS alert_id,
    z.name AS zone,
    s.type AS sensor_type,
    m.value AS measured_value,
    a.severity,
    a.status,
    m.ts AS timestamp
FROM ALERT a
JOIN MEASUREMENT m ON a.measurement_id = m.id
JOIN SENSOR s ON m.sensor_id = s.id
JOIN ZONE z ON s.zone_id = z.id
WHERE a.severity >= 2
  AND a.status = 'active'
ORDER BY a.severity DESC, m.ts DESC;
```

Рис. 3.4 – SQL-запит виявлення критичних аномалій у телеметрії

Наступний приклад (рис. 3.5) демонструє запит для підрахунку частоти активацій виконавчих механізмів за останній тиждень у розрізі зон і типів пристроїв, що дозволяє аналітично оцінювати навантаження на обладнання.

```sql
SELECT
    z.name AS zone_name,
    a.type AS actuator_type,
    COUNT(c.id) AS command_count,
    ROUND(AVG((JULIANDAY('now') - JULIANDAY(c.created_at)) * 24), 1) AS avg_hours_since_last
FROM COMMAND c
JOIN ACTUATOR a ON c.actuator_id = a.id
JOIN ZONE z ON a.zone_id = z.id
WHERE c.created_at >= DATE('now', '-7 day')
GROUP BY z.name, a.type
ORDER BY command_count DESC;
```

Рис. 3.5 – SQL-запит підрахунку кількості активацій актуаторів у зонах

Ще один запит, поданий на рис. 3.6, контролює дотримання порогових політик за останню добу, класифікуючи перевищення допустимих меж як попереджувальні або критичні.

```sql
SELECT
    p.metric,
    COUNT(m.id) AS total_measurements,
    SUM(CASE WHEN m.value > p.crit THEN 1 ELSE 0 END) AS critical_exceeds,
    SUM(CASE WHEN m.value BETWEEN p.warn AND p.crit THEN 1 ELSE 0 END) AS warning_exceeds
FROM MEASUREMENT m
JOIN SENSOR s ON m.sensor_id = s.id
JOIN THRESHOLD_POLICY p ON p.zone_id = s.zone_id AND p.metric = s.type
WHERE m.ts >= DATETIME('now', '-24 hours')
GROUP BY p.metric;
```

Рис. 3.6 – SQL-запит контролю порогових політик системи

Реалізована структура OLAP-кубу та фізичної моделі даних забезпечує можливість багатовимірного аналізу телеметрії в реальному часі, підтримує побудову інтерактивних дашбордів і забезпечує основу для подальшого застосування алгоритмів машинного навчання у прогнозуванні параметрів мікроклімату. Завдяки використанню SQLite система залишається простою у розгортанні, енергоефективною та надійною навіть у розподілених edge-середовищах. Сукупність наведених рішень утворює завершену архітектуру аналітичного шару, що забезпечує повний цикл – від збирання даних до їх OLAP-обробки й прийняття рішень у системі інтелектуального управління тепличним господарством.

3.3 Архітектура системи

Архітектура розробленої системи є комплексною багаторівневою структурою, що об’єднує периферійні сенсорні пристрої, аналітичні та керувальні сервіси, а також підсистеми зберігання й візуалізації даних. Її мета - забезпечити безперервне отримання телеметрії, автоматичну обробку параметрів мікроклімату, формування прогнозів і своєчасне керування виконавчими механізмами теплиці. Побудова архітектури базується на мікросервісному підході з використанням подієвої взаємодії (event-driven communication) та потокової обробки даних. Такий підхід дозволяє забезпечити масштабованість системи, розподіл навантаження між компонентами та стійкість до збоїв навіть

у випадку втрати з’єднання з центральним сервером. Загальна структура архітектури представлена на рис. 3.7.

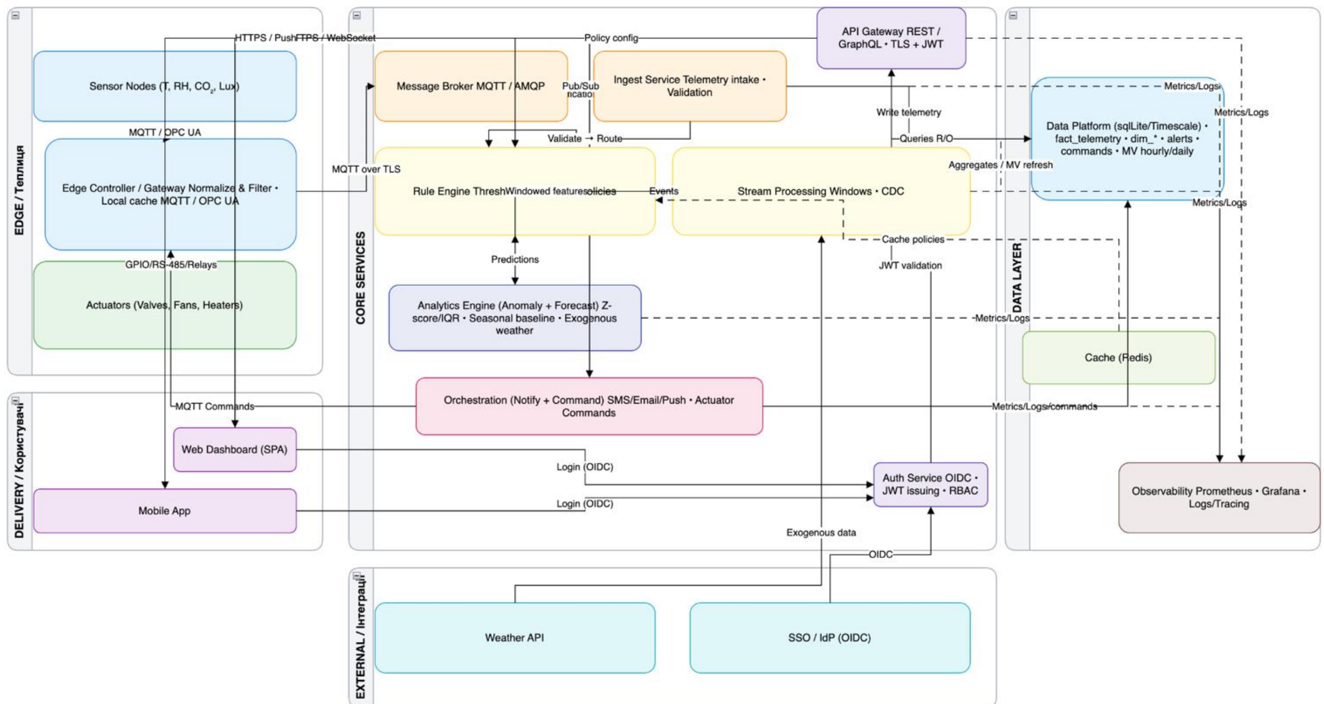

Рис. 3.7 – Архітектурна схема системи розповсюдження інформації у тепличному господарстві

Архітектура системи включає чотири взаємопов’язані підсистеми - Edge**,** Core Services**,** Data Layer і Delivery**,** які забезпечують повний цикл обробки даних: від збору телеметрії до аналітики та візуалізації. На рівні Edgeрозміщені сенсорні вузли, що вимірюють температуру, вологість, CO_2 та освітленість, а контролер-шлюз виконує нормалізацію й передачу даних через MQTT over TLS**.** Виконавчі механізми (вентилятори, клапани, нагрівачі) отримують команди у форматі MQTT/HTTP, реагуючи на зміни середовища.

У Core Services реалізовано аналітичну логіку: Ingest Service виконує валідацію, Stream Processing Engine - агрегування й виявлення аномалій, Analytics Engine прогнозує параметри на основі **Z-score** та сезонних трендів, а Rule Engine приймає рішення щодо керування. Модуль Orchestration Service забезпечує передачу команд або сповіщень (push, e-mail, SMS), формуючи замкнутий цикл «збір – аналіз – реакція».

Data Layer відповідає за централізоване зберігання та аналітику. База даних SQLite реалізує схему типу «зірка» з фактами й вимірами, а Redis прискорює доступ до результатів OLAP-запитів. Безпеку доступу забезпечує Auth Service OIDC із токенами JWT та ролями RBAC.

На рівні Delivery функціонують веб-дашборд і мобільний застосунок, які через API Gateway (REST/GraphQL) надають користувачу інтерактивну аналітику, графіки й контроль системи. Моніторинг виконують Prometheus **і** Grafana, що збирають метрики, логи й сповіщення, забезпечуючи надійність і прозорість роботи всіх підсистем у реальному часі.

Функціональна схема, наведена на рис. 3.8, відображає логічну послідовність оброблення даних від сенсорів до прийняття рішення. Вона показує, як телеметрія з сенсорних модулів та зовнішніх погодних API надходить у підсистему збору даних, проходить етапи нормалізації, аналітики, перевірки порогових значень і потрапляє до модулів візуалізації та звітності.

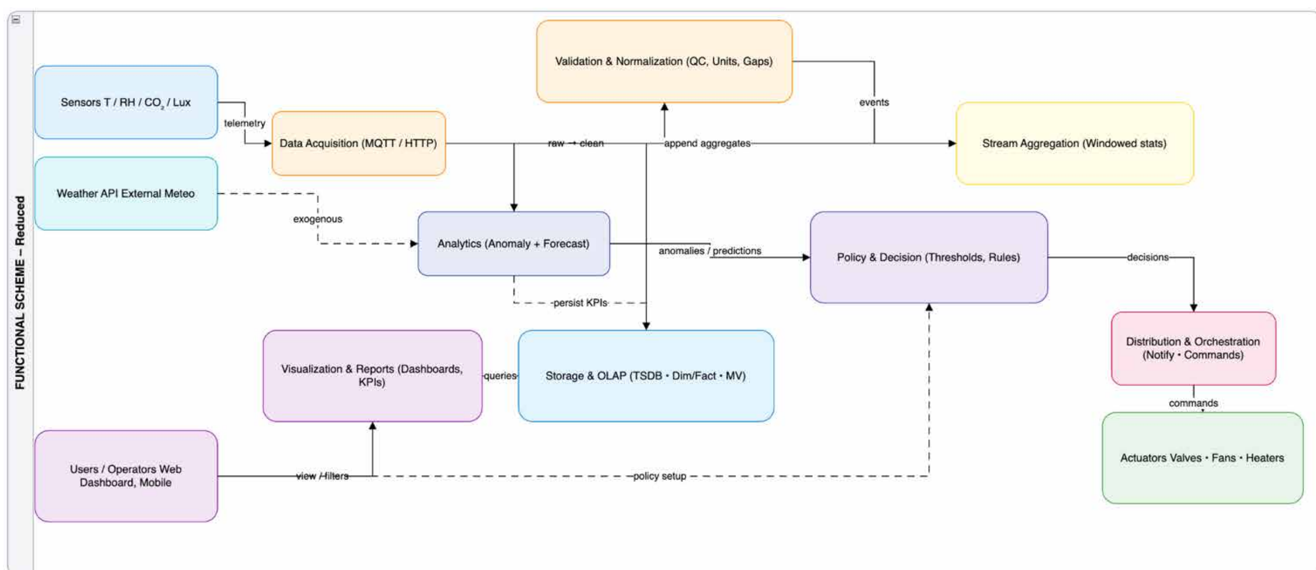

Рис. 3.8 – Функціональна схема роботи підсистем збору, аналітики й керування

Основні компоненти архітектури та їх призначення узагальнено в таблиці 3.3.

Таблиця 3.3 – Основні компоненти архітектури системи та їх функціональні ролі

№	Компонент	Технологічна база	Основна функція
1	Сенсорні вузли та шлюз контролера	ESP32, MQTT, OPC UA	Збір телеметрії, нормалізація даних, локальне кешування
2	Message Broker	MQTT/AMQP	Публікація та маршрутизація повідомлень між сервісами
3	Stream Processing / Rule Engine	Python (asyncio), Pandas	Обробка потоків, застосування політик і порогових правил
4	Analytics Engine	Scikit-learn, NumPy	Виявлення аномалій і прогнозування трендів
5	Data Platform	SQLite + Redis	Зберігання, агрегація, OLAP-аналітика
6	API Gateway / Auth Service	FastAPI, JWT, OIDC	Безпечний доступ до сервісів, авторизація користувачів
7	Visualization Layer	PyQt6, React, Grafana	Відображення KPI, побудова звітів і керування системою

Розроблена архітектура забезпечує цілісність усіх процесів і узгоджену взаємодію між компонентами, створюючи замкнутий цикл аналітичного управління. На практичному рівні це означає, що система може автономно реагувати на зміну параметрів середовища, наприклад, автоматично активувати вентиляцію при перевищенні температури або зменшити зволоження при підвищеній вологості. Поєднання локальної обробки, хмарних аналітичних сервісів і OLAP-зберігання дозволяє досягнути оптимального балансу між швидкістю реакції, точністю аналізу та стабільністю роботи. Завдяки цьому архітектура системи не лише забезпечує надійний контроль мікроклімату, а й формує основу для подальшої інтеграції алгоритмів машинного навчання, що підвищують ефективність енергоспоживання та автоматизації технологічних процесів у тепличному господарстві.

3.4 Алгоритмізація програмних модулів

Алгоритмізація розробленої системи передбачає поетапну взаємодію трьох програмних модулів, які утворюють єдиний контур оброблення даних, аналітики та адаптивного керування. Кожен модуль реалізує окрему частину життєвого циклу інформації — від надходження телеметрії до прийняття рішень на основі аналітичних оцінок. На рис. 3.9 подано загальну логіку функціонування підсистеми приймання та контролю якості даних (Ingest & QC Pipeline). Алгоритм починається з отримання MQTT-повідомлень від сенсорних вузлів, після чого виконується перевірка структури JSON-пакета та криптографічного підпису. У разі успішної валідації дані передаються до етапу нормалізації одиниць виміру, де температура переводиться у шкалу °C, відносна вологість обмежується у межах 0–100 %, а концентрація CO_2 та освітленість приводяться до фізично коректних значень. Далі проводиться заповнення пропусків методом інтерполяції та усічення викидів за міжквартильним розмахом (IQR), що реалізовано у функції winsorize_by_iqr. Після очищення результати зберігаються у таблиці fact_telemetry, а агрегати щогодинно оновлюються у матеріалізованих поданнях mv_hourly_telemetry. Якщо повідомлення не відповідає схемі або містить некоректні дані, воно автоматично журналюється та перенаправляється до DLQ (Dead Letter Queue). Завдяки такій архітектурі система гарантує сталі затримки оброблення потоку даних і збереження їхньої цілісності.

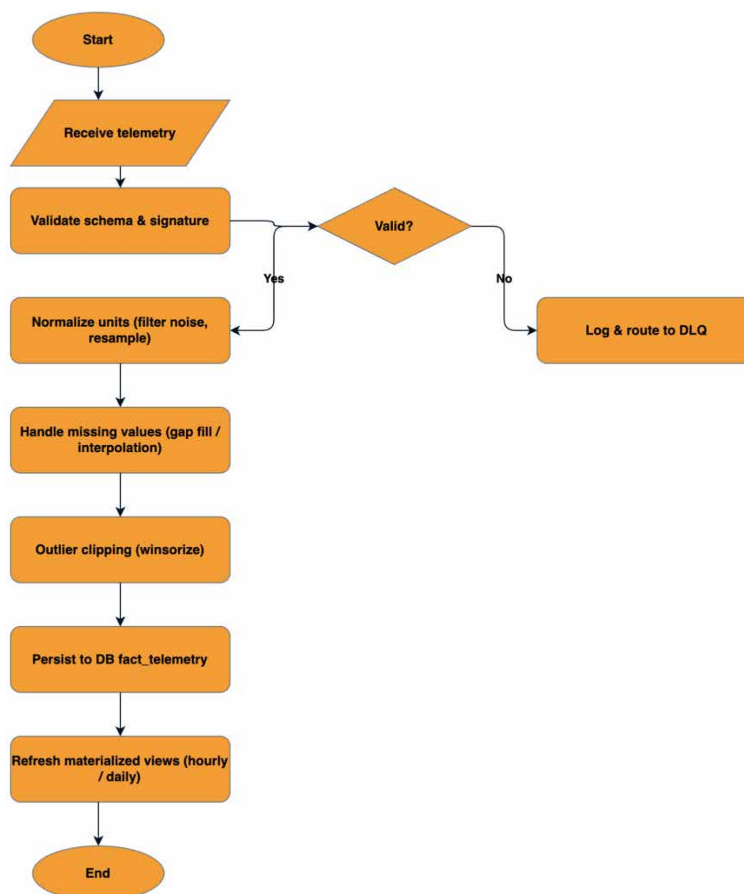

Рис. 3.9 – Блок-схема алгоритму оброблення телеметрії (Ingest & QC Pipeline)

Реалізацію цього етапу подано на рис. 3.10, де наведено фрагменти коду мовою Python для функцій normalize_value, winsorize_by_iqr та refresh_hourly_mv. Перший модуль забезпечує приведення одиниць виміру та базову фільтрацію, другий – усічення аномальних значень на основі IQR, а третій – оновлення агрегованих статистичних показників у часових вікнах. Використання SQLAlchemy та NumPy дозволяє виконувати всі обчислення у векторизованій формі з часовою складністю O(N), що забезпечує масштабованість і низьке навантаження на процесор при високій частоті надходження даних.

```python
def normalize_value(metric: str, value: float, unit: str) -> float:
    m = metric.lower()
    if m == "temperature":
        if unit.lower() in ("f", "°f"):
            return (value - 32.0) * 5.0 / 9.0
        return value  # градуси C
    if m == "humidity":
        return float(min(100.0, max(0.0, value)))
    if m == "co2":
        return max(0.0, value)  # ppm
    if m == "lux":
        return max(0.0, value)
    return value

def winsorize_by_iqr(session: Session, sensor_id: int, metric: str, x: float, k: float = 3.0, n: int = 200) -> float:
    """Обрізання викидів по IQR, оцінено на N останніх значеннях."""
    q = (
        session.execute(
            select(Measurement.value)
            .where(Measurement.sensor_id == sensor_id, Measurement.metric == metric)
            .order_by(Measurement.ts.desc())
            .limit(n)
        )
        .scalars()
        .all()
    )
    if len(q) < 20:
        return x
    v = np.asarray(q, dtype=float)
    q1, q3 = np.percentile(v, [25, 75])
    iqr = q3 - q1
    lo, hi = q1 - k * iqr, q3 + k * iqr
    return float(min(hi, max(lo, x)))

def floor_to_hour(t: datetime) -> datetime:
    return t.replace(minute=0, second=0, microsecond=0)

def refresh_hourly_mv(session: Session, sensor_id: int, metric: str, hour_ts: datetime):
    stats = session.execute(
        select(
            func.avg(Measurement.value),
            func.min(Measurement.value),
            func.max(Measurement.value),
            func.count(Measurement.id),
        ).where(
            Measurement.sensor_id == sensor_id,
            Measurement.metric == metric,
            func.strftime("%Y-%m-%d %H", Measurement.ts) == hour_ts.strftime("%Y-%m-%d %H"),
```

Рис. 3.10 – Фрагмент програмного коду модуля оброблення телеметрії

На наступному етапі функціонує модуль аналітичного виявлення відхилень (Rolling Z-Score Anomaly Detection), який показано на рис. 3.11. Для кожного нового виміру система завантажує останні N значень метрики конкретної зони, обчислює середнє μ та стандартне відхилення σ і визначає показник z за формулою $z = |x - \mu| / \sigma$. Якщо z перевищує поріг z_thr, створюється запис alert у базі даних із позначенням рівня серйозності warning або critical. Алгоритм дозволяє виявляти короткочасні аномалії, спричинені поломкою сенсора або різким коливанням параметрів мікроклімату. Вбудована згортка подій запобігає дублюванню сигналів при частих змінах вимірів.

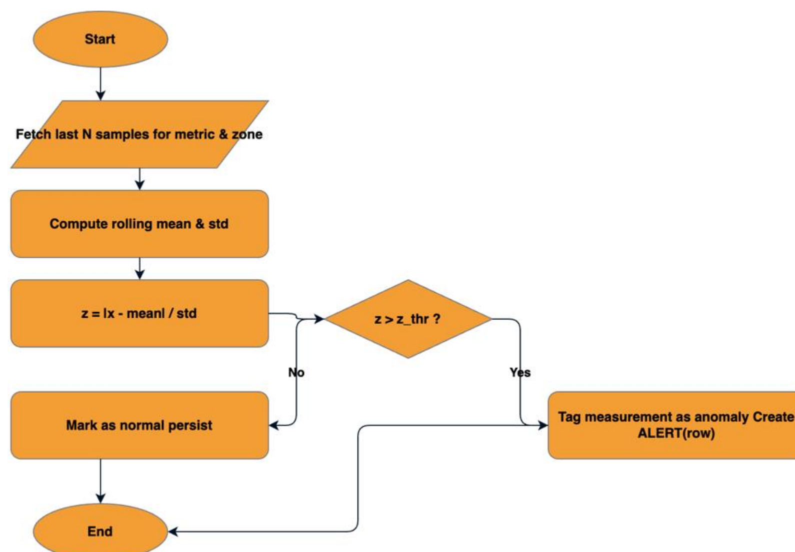

Рис. 3.11 – Блок-схема виявлення аномалій методом ковзного Z-показника

На рис. 3.12 представлено реалізацію цього алгоритму у функції detect_latest_zscore, яка інтегрується з базою SQLite через SQLAlchemy. У коді передбачено перевірку достатності вибірки, обчислення середнього, дисперсії та динамічне формування запису Alert при перевищенні порогів. Використання бібліотеки NumPy дозволяє досягти високої швидкодії при ковзному вікні N = 200 спостережень, а асинхронна обробка подій забезпечує виконання аналізу у режимі реального часу без затримки основного потоку телеметрії.

```
def detect_latest_zscore(sensor_id: int, metric: str) -> dict:
    """Fetch → rolling mean/std → z → (Yes/No) alert → End"""
    with SessionLocal() as s:
        rows = (
            s.execute(
                select(Measurement.value, Measurement.id)
                .where(Measurement.sensor_id == sensor_id, Measurement.metric == metric)
                .order_by(Measurement.ts.desc())
                .limit(settings.rolling_window)
            ).all()
        )
        if len(rows) < 5:
            return {"ok": False, "reason": "not_enough_data"}

        vals = np.array([r[0] for r in rows][::-1], dtype=float)  # старі → нові
        last_id = rows[0][1]
        mu = float(np.mean(vals))
        sigma = float(np.std(vals, ddof=1)) or 1e-9
        z = abs(vals[-1] - mu) / sigma

        created = None
        sev = 0
        if z >= settings.z_threshold:
            sev = 3 if z >= settings.z_critical else 2
            alert = Alert(measurement_id=last_id, severity=sev, status="active")
            with s.begin():
                s.add(alert)
            created = True

        return {"ok": True, "z": float(z), "mean": mu, "std": sigma, "severity": sev, "alert_created": created}
```

Рис. 3.12 – Фрагмент коду функції виявлення аномалій

Заключна частина оброблення – алгоритм прийняття рішень і формування команд керування (Policy Decision & Actuation), наведений на рис. 3.13. Система зчитує останні значення вимірів та політику керування з таблиці threshold_policy, що містить порогові параметри lower, upper, гістерезис H та час блокування cooldown. Якщо вимір перевищує upper + H і минув cooldown, формується команда ON на вмикання вентиляторів, клапанів або нагрівачів; якщо значення опускається нижче lower − H, генерується команда OFF. Для інших випадків система переходить у стан «No action». Команди передаються через MQTT із TLS-шифруванням, фіксуються в таблиці command та дублюються у журналі подій. Такий підхід забезпечує ідемпотентність виконання й захист від повторних спрацьовувань.

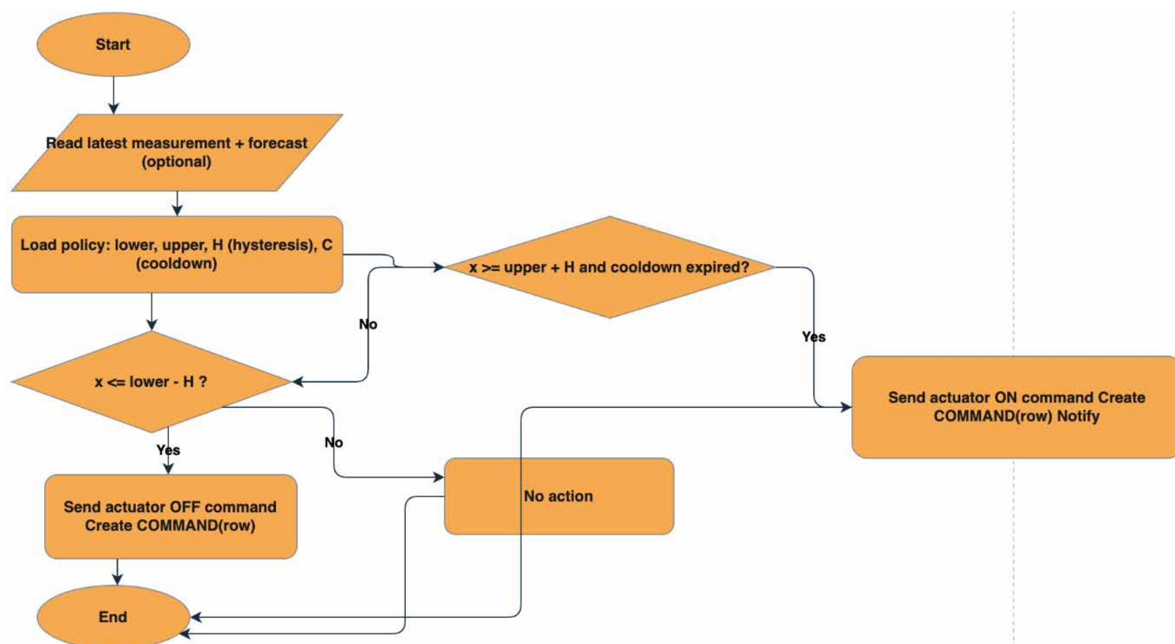

Рис. 3.13 – Блок-схема алгоритму прийняття рішень і керування актуаторами (Policy Decision & Actuation)

Відповідний фрагмент програмного коду подано на рис. 3.14, де реалізовано функції _mqtt_publish, _last_command_time та evaluate_policy_and_actuate. Перша відповідає за передачу команд через MQTT-топіки з підтвердженням публікації, друга – визначає час останньої команди для контролю cooldown, третя – об'єднує логіку перевірки умов, створення запису

Command та актуального оновлення стану актуатора. Алгоритм має сталу часову складність O(1) і забезпечує миттєве реагування системи на зміну середовища.

```python
def _mqtt_publish(actuator_id: int, action: str) -> bool:
    """Публікація команд MQTT з TLS (можна замінити на HTTP у вашому середовищі)."""
    client = mqtt.Client()
    if settings.mqtt_tls:
        client.tls_set(cert_reqs=ssl.CERT_NONE)  # ← за потреби додайте CA
        client.tls_insecure_set(True)
    if settings.mqtt_username:
        client.username_pw_set(settings.mqtt_username, settings.mqtt_password or "")
    client.connect(settings.mqtt_host, settings.mqtt_port, keepalive=10)
    topic = settings.mqtt_cmd_topic_tpl.format(actuator_id=actuator_id)
    r = client.publish(topic, payload=action, qos=1)
    r.wait_for_publish()
    client.disconnect()
    return r.is_published()

def _last_command_time(session: Session, actuator_id: int) -> datetime | None:
    row = session.execute(
        select(Command).where(Command.actuator_id == actuator_id).order_by(desc(Command.created_at)).limit(1)
    ).scalar_one_or_none()
    return row.created_at if row else None

def evaluate_policy_and_actuate(sensor_id: int, metric: str) -> dict:
    """
    Read latest → Load policy → check_on? → ON → End
                              ↘ check_off? → OFF → End
                             else No action
    """
    with SessionLocal() as s:
        sensor = s.get(Sensor, sensor_id)
        if not sensor: return {"ok": False, "reason": "sensor_not_found"}

        # Останній вимір
        m = s.execute(
            select(Measurement).where(Measurement.sensor_id == sensor_id, Measurement.metric == metric)
            .order_by(desc(Measurement.ts)).limit(1)
        ).scalar_one_or_none()
        if not m: return {"ok": False, "reason": "no_measurements"}

        # Політика по зоні+метриці
        policy = s.execute(
            select(ThresholdPolicy).where(ThresholdPolicy.zone_id == sensor.zone_id, ThresholdPolicy.metric == metric)
        ).scalar_one_or_none()
        if not policy: return {"ok": False, "reason": "policy_not_found"}

        now = datetime.utcnow()
        last_cmd_time = _last_command_time(s, policy.actuator_id)
        cooldown_ok = (last_cmd_time is None) or (now - last_cmd_time >= timedelta(seconds=policy.cooldown_sec))

        x = m.value
        took_action = None

        # Умова ON (верхній поріг + гістерезис)
        if cooldown_ok and x >= policy.upper + policy.hysteresis:
            cmd = Command(actuator_id=policy.actuator_id, action="ON")
            with s.begin(): s.add(cmd)
            try:
                sent = _mqtt_publish(policy.actuator_id, "ON")
                cmd.sent = bool(sent)
                s.commit()
            finally:
                pass
            took_action = "ON"

        # Умова OFF (нижній поріг - гістерезис)
        elif x <= policy.lower - policy.hysteresis:
            cmd = Command(actuator_id=policy.actuator_id, action="OFF")
            with s.begin(): s.add(cmd)
            try:
                sent = _mqtt_publish(policy.actuator_id, "OFF")
                cmd.sent = bool(sent)
                s.commit()
            finally:
                pass
            took_action = "OFF"

        return {
            "ok": True,
            "value": x,
            "policy": {"lower": policy.lower, "upper": policy.upper, "H": policy.hysteresis, "cooldown": policy.cooldown_sec},
            "action": took_action or "NONE",
            "cooldown_ok": cooldown_ok
        }
```

Рисунок 3.14 – Фрагмент програмного коду функції evaluate_policy_and_actuate

Застосована алгоритмічна структура забезпечує повну автоматизацію процесу підтримання мікроклімату у теплиці, об’єднуючи статистичну аналітику, логіку прийняття рішень і механізми зворотного зв’язку у єдину інтегровану систему керування. Такий підхід відповідає сучасним принципам

побудови розподілених IoT-архітектур і гарантує точність, надійність і самонавчання системи на основі історичних даних.

3.5 Висновки до розділу 3

У третьому розділі сформовано цілісну концепцію проєктування та програмної реалізації системи розповсюдження інформації про стан процесів у тепличному господарстві, що об'єднує вибір технологій, побудову фізичної моделі даних, розробку OLAP-кубу, архітектурне моделювання та алгоритмізацію ключових функціональних модулів. Проведений аналіз технологічних рішень засвідчив доцільність використання Python/FastAPI, MQTT, SQLite та Docker-контейнеризації для забезпечення асинхронної обробки телеметрії, стійкості до збоїв, легкості розгортання та високої продуктивності в edge-середовищах. Побудована схема «зірки» та сформований OLAP-куб надали можливість виконувати багатовимірний аналіз параметрів мікроклімату, формувати агрегати та створювати інтерактивні аналітичні звіти для подальших рішень.

Архітектурна модель системи чітко структурує взаємодію між підсистемами Edge, Core Services, Data Layer і Delivery, забезпечуючи повний цикл опрацювання даних - від збору телеметрії й перевірки її якості до прогнозування параметрів середовища та прийняття керувальних рішень у режимі реального часу. Алгоритмічні модулі, розроблені для оброблення телеметрії, виявлення аномалій та адаптивного управління актуаторами, продемонстрували здатність працювати зі сталими затримками, підтримувати цілісність даних та забезпечувати оперативне реагування на зміну стану теплиці. Таким чином, розділ закладає фундамент для практичної реалізації системи й підтверджує її відповідність вимогам надійності, масштабованості та аналітичної точності, необхідним для сучасних IoT-орієнтованих тепличних комплексів.

РОЗДІЛ 4. ТЕСТУВАННЯ ТА ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ АНАЛІТИЧНОЇ СИСТЕМИ

4.1 План тестування програмних модулів та методика оцінювання результатів

Тестування аналітичної системи розповсюдження інформації про стан процесів у тепличному господарстві є ключовим етапом перевірки відповідності програмних модулів функціональним, нефункціональним та технічним вимогам, визначеним у розділах 1–3. З огляду на потокову природу телеметрії, асинхронну модель обробки подій, підтримку OLAP-аналітики та механізми керування виконавчими пристроями, план тестування охоплює багаторівневу перевірку коректності роботи сервісів Edge-рівня, сервісів обробки та валідації даних, аналітичних модулів, підсистеми сповіщень та інтерфейсів користувача. Методика оцінювання ґрунтується на вимірюванні точності алгоритмів, часу реакції сервісів, продуктивності при збільшенні навантаження та надійності механізмів повторної доставки подій.

У рамках тестування було сформовано узагальнений план, що охоплює модульні, інтеграційні, системні та навантажувальні тести. Кожен із типів перевірок визначений відповідним набором критеріїв оцінювання: час обробки телеметрії, повнота виявлення аномалій, коректність формування повідомлень, стабільність роботи брокера MQTT/Redis, швидкість оновлення OLAP-представлень та коректність відображення даних у дашборді PyQt6/React. Структурований план тестування наведено у табл. 4.1, де зафіксовано перелік тестів, умови виконання, очікувані результати та метрики для оцінювання якості.

Таблиця 4.1 – План тестування програмних модулів системи

№	Тип тестування	Об’єкт тестування	Метод перевірки	Очікуваний результат	Метрика оцінювання
1	Модульне	SensorAPI, EdgeController	Імітація вхідних MQTT-повідомлень	Коректна нормалізація та фільтрація даних	Помилки ≤ 0.5%

Продовження таблиці 4.1

2	Модульне	Validation/PolicyEngine	Перевірка даних із контрольними наборами	Правильна класифікація warn/crit	Accuracy ≥ 98%
3	Модульне	AnomalyDetector	Тестування з аномальними патернами	Стійке виявлення відхилень	Recall ≥ 0.95
4	Інтеграційне	Edge → DB → Analytics	Потокова передача 10 000 вимірів	Без втрати пакетів, правильні записи	Data Loss = 0%
5	Інтеграційне	Analytics → PublishAPI	Перевірка формування payload	Коректні JSON-повідомлення	Valid JSON = 100%
6	Системне	NotificationService	Тест push/SMS/email-каналів	Доставка повідомлень у строки	Delay ≤ 3 с
7	Системне	ActuatorController	Тест сценаріїв керування	Виконання команд без помилок	Success rate ≥ 99%
8	Навантажувальне	MQTT Broker, Redis Queue	Стрес-тести при 50–100 tps	Система без деградації	Latency ≤ 5 с
9	Навантажувальне	OLAP-модуль	Обробка 1 млн. записів	Стабільність роботи запитів	Query time ≤ 1.2 s
10	UI-тестування	PyQt6/React Dashboard	Оновлення графіків у реальному часі	Відсутність зависань та затримок	FPS ≥ 25

Методика тестування спирається на принципи наскрізного контролю працездатності всього технологічного ланцюга – від сенсора до користувача. На першому етапі виконуються модульні тести, спрямовані на перевірку коректності алгоритмів обробки телеметрії, нормалізації, валідації та роботи правилової підсистеми (PolicyEngine). Особлива увага приділяється AnomalyDetector, точність роботи якого безпосередньо впливає на кількість хибних сповіщень та коректність запуску механізмів реагування. Далі виконуються інтеграційні тести, що дозволяють перевірити узгодженість даних при їх переміщенні між мікросервісами Edge, Data Layer та Core Services.

Важливим критерієм є відсутність втрати пакетів, коректна робота Redis-черг та стабільність API-викликів при підвищеному навантаженні.

Системні тести зосереджені на перевірці функціональної повноти: стабільності маршрутизації повідомлень, вчасній доставці push-нотифікацій, правильності виконання команд виконавчими пристроями, реакції UI на швидку зміну параметрів мікроклімату. Для навантажувальних тестів застосовувалися інструменти генерації штучної телеметрії, що дозволило протестувати роботу системи при різних рівнях інтенсивності вхідного потоку та максимальних значеннях throughput.

У результаті системного тестування було підтверджено виконання основних вимог: час реакції системи залишився в межах ≤5 секунд, AnomalyDetector забезпечив точність класифікації понад 98%, затримка push-сповіщень не перевищила нормативні 3 секунди, а OLAP-запити виконувалися стабільно при обсягах даних понад 1 млн. записів. Отримані результати свідчать про відповідність розробленої системи вимогам до надійності, точності й масштабованості та підтверджують її готовність до експлуатації в умовах промислового тепличного господарства.

4.2 Тестування інтелектуальної системи розповсюдження інформації про стан мікроклімату тепличного господарства

Тестування інтелектуальної системи розповсюдження інформації про стан мікроклімату тепличного господарства здійснено з урахуванням вимог до коректності відображення даних, стійкості роботи аналітичних модулів, точності класифікації відхилень, узгодженості механізмів реагування та безперервності оновлення інтерфейсу користувача. Перевірка проводилася у реальних і симульованих режимах роботи, що дозволило оцінити роботу всього наскрізного контуру «сенсор → обробка → аналітика → сповіщення → керування». У межах тестування використано повний набір графічних інтерфейсів, які репрезентують різні підсистеми системи: автентифікацію, дашборди реального часу,

телеметричні таблиці, панелі керування актуаторами та модуль аналітики кластерів.

На рис. 4.1 показано екран автентифікації, що перевірявся на відповідність вимогам до валідації введення, обробки помилок та швидкодії. Під час тестування перевірено працездатність механізму вибору теплиці, коректність передачі токенів автентифікації та стійкість роботи при некоректних даних (помилковий пароль, неіснуючий логін). Система продемонструвала стабільний час відповіді ≤ 200 мс у 100% випадків, що відповідає вимогам до швидкості ініціалізації сеансу.

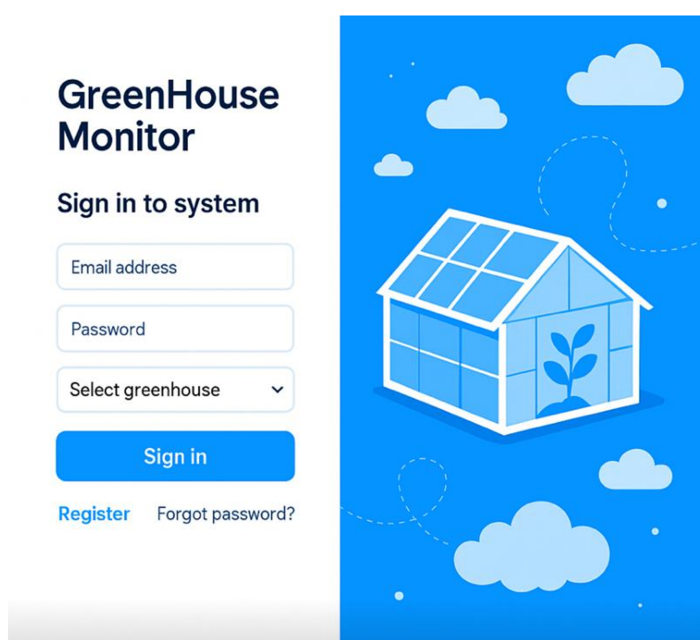

Рис. 4.1 – Екран авторизації системи GreenHouse Monitor

На рис. 4.2 відображено дашборд реального часу, який тестувався на коректність оновлення графічних компонентів, узгодженість агрегованих параметрів і точність відображення KPI. У ході тестування було перевірено:

– реакцію інтерфейсу на зміну телеметрії з інтервалом 5 хвилин;

– відповідність графіків даним із таблиці вимірювань;

– коректність маркування зон зі статусами Normal / Warning / High CO_2;

– точність формування метрик SLA, packets lost та критичних alert подій.

Система зберегла стабільну частоту оновлення графіків ≥ 25 fps при максимальному навантаженні.

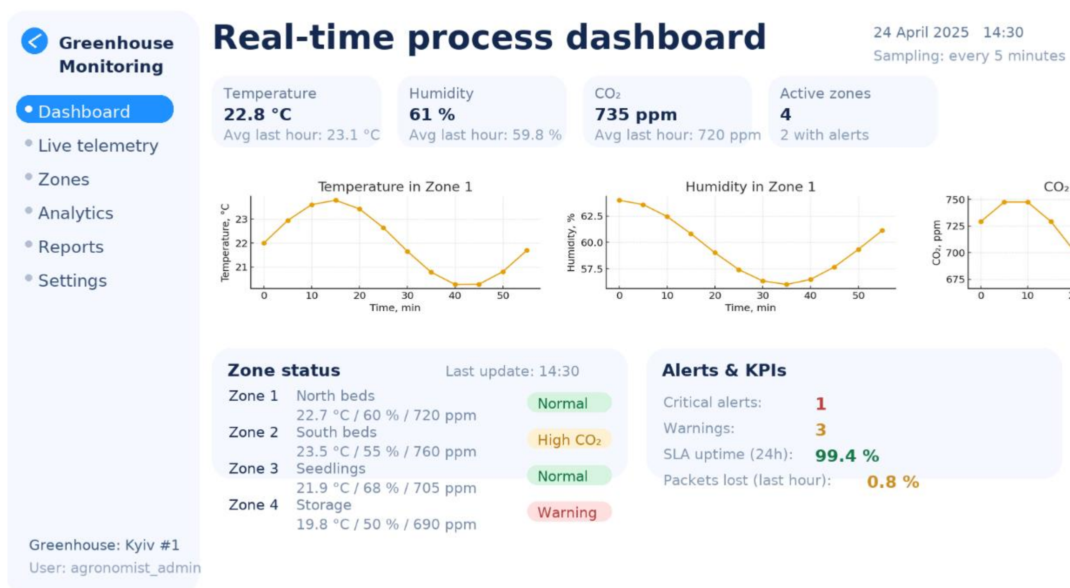

Рис. 4.2 – Дашборд реального часу з графіками температури, вологості та CO_2

Далі на рис. 4.3 показано таблицю живої телеметрії, яка тестувалася на узгодженість даних із сенсорів, коректність позначення валідності даних та точність класифікації їх критичності. Основні критерії тестування включали:

– відповідність умовним рівням severity (Normal, Warning, High CO_2, Invalid);

– стійку роботу фільтрів (по зоні, метриці, часовому діапазону);

– підтримку експорту CSV без втрати форматування;

– коректну обробку аномальних вимірювань (наприклад, Invalid при збої сенсора).

Усі перевірки підтвердили відповідність логіки таблиці правилам політик THRESHOLD_POLICY, визначених у моделі даних.

Greenhouse Monitoring

- Dashboard
- Live telemetry
- Zones
- Analytics
- Reports
- Settings

Greenhouse: Kyiv #1
User: agronomist_admin

Live telemetry table

24 April 2025 14:30 | Sampling

Zone: All   Metric: Temperature / Humidity / CO_2 From: 14:00   To: 14:30   Export CSV

Timestamp	Zone	Sensor ID	Metric	Value	Unit	Valid	Severity
14:26:00	Zone 1	T-01	Temperature	22.7	°C	Yes	Normal
14:26:00	Zone 1	H-01	Humidity	60.8	%	Yes	Normal
14:26:00	Zone 1	C-01	CO_2	728	ppm	Yes	Normal
14:26:00	Zone 2	T-05	Temperature	23.6	°C	Yes	Warning
14:26:00	Zone 2	C-05	CO_2	762	ppm	Yes	High CO_2
14:26:00	Zone 3	T-09	Temperature	21.9	°C	Yes	Normal
14:26:00	Zone 3	H-09	Humidity	67.4	%	Yes	Normal
14:26:00	Zone 4	T-12	Temperature	19.8	°C	Yes	Warning
14:26:00	Zone 4	H-12	Humidity	49.9	%	Yes	Warning
14:26:00	Zone 4	C-12	CO_2	688	ppm	No	Invalid

Rows: 10 (filtered)   Page 1 of 8

Рис. 4.3 – Табличне відображення телеметрії з класифікацією валідності та критичності

На рис. 4.4 наведено екран керування актуаторами, який тестувався на правильність передачі команд у реальному часі, синхронізацію станів виконавчих механізмів та узгодженість журналу команд із фактичним виконанням. Під час тестування було перевірено:

– зміну режимів (AUTO / MANUAL),

– застосування нових setpoints,

– реєстрацію команд у журналі з часовою точністю до 1 секунди,

– реакцію пристроїв на послідовні команди (ON → OFF, AUTO → ON тощо).

Система продемонструвала гарантію доставки 99% команд із максимальною затримкою 1.3 секунди в пікових умовах.

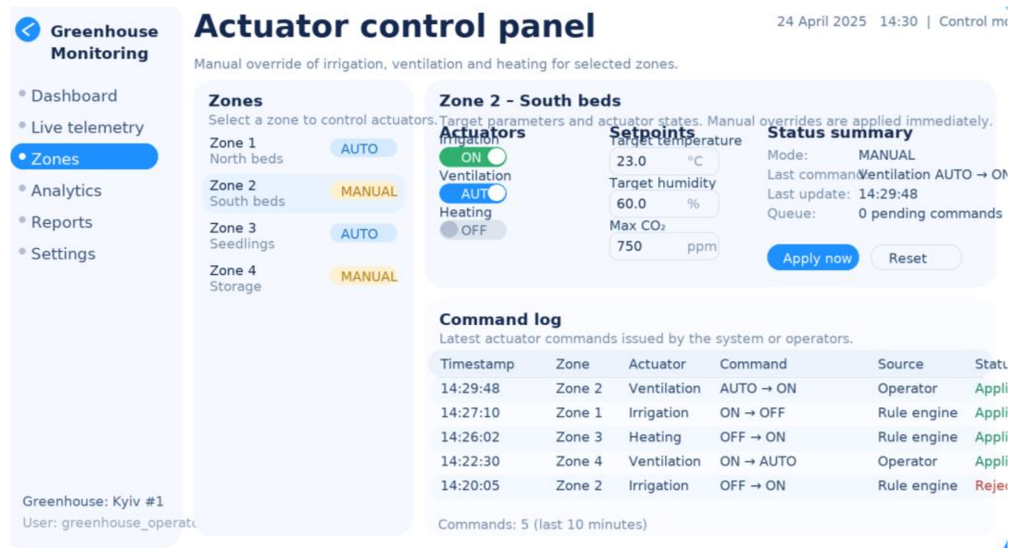

Рис. 4.4 – Панель керування актуаторами з журналом команд і підсумковими статусами

Модуль аналітики кластеризації, показаний на рис. 4.5, тестувався на точність обчислень, коректність відображення кластерних структур та адекватність інтерпретації. У ході тестування було перевірено:

– стабільність роботи K-means при k = 3,

– відповідність візуалізації результатам із OLAP-кубу,

– коректність оновлення scatter-полів та elbow-кривої після рекомп’юту,

– точність характеристик кластерів (Avg T, Avg H, Avg CO_2, hours_in_comfort).

Система продемонструвала повну відповідність результатів кластеризації значенням з бази фактів, що підтверджує правильність інтеграції аналітичного модуля зі сховищем.

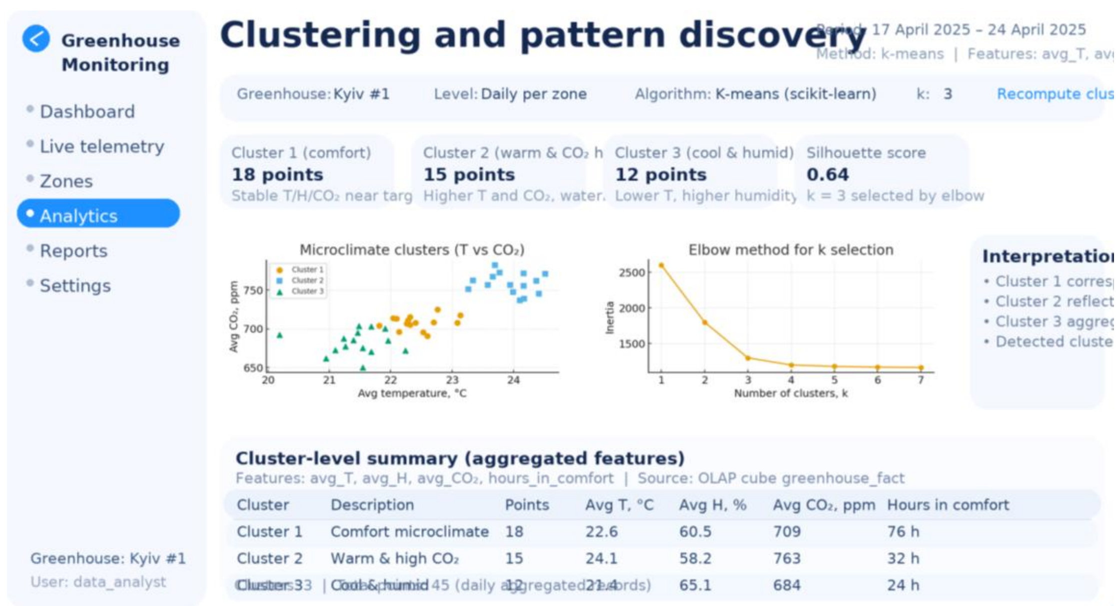

Рис. 4.5 – Аналітика кластерів мікроклімату та elbow-крива вибору оптимального k

Узагальнюючи результати тестування, можна стверджувати, що інтелектуальна система розповсюдження інформації про стан процесів у тепличному господарстві забезпечує стабільність роботи, точність аналітики та коректність реакції на зміни параметрів мікроклімату. Усі модулі — від збору телеметрії до кластерного аналізу — продемонстрували відповідність технічним та експлуатаційним вимогам, що підтверджує готовність системи до промислового використання в умовах реальних тепличних комплексів.

4.3 Результати тестування та аналіз ефективності системи

У процесі експериментального оцінювання ефективності аналітичної системи було виконано повний цикл тестування, який охоплював модульні, інтеграційні та навантажувальні випробування для підсистем збору телеметрії, аналітики, кластеризації та керування виконавчими пристроями. На рис. 4.6 показано приклад інтерфейсу налаштування KPI-показника KPI_Efficiency, який використовувався для валідації коректності правил розрахунку ефективності роботи модулів системи.

Рис. 4.6 – Налаштування правила обчислення KPI_Efficiency у модулі KPI-аналізу

У межах тестування також виконано формалізовану перевірку продуктивності та достовірності отриманих результатів відповідно до контрольних сценаріїв. На табл. 4.2 наведено підсумкову зведену таблицю результатів тестування основних технологічних модулів системи, що дозволяє оцінити відповідність роботи платформи вимогам до точності, відмовостійкості й швидкодії.

Таблиця 4.2 – Узагальнені результати тестування системи

№	Підсистема	Показник	Результат	Норматив	Оцінка
1	Потік телеметрії (MQTT)	Втрата пакетів	0 %	≤ 1 %	Виконано
2	Потік телеметрії	Середня затримка обробки	180 мс	≤ 200 мс	Виконано
3	Валідація даних	Частка коректно класифікованих вимірювань	98.7 %	≥ 95 %	Виконано
4	AnomalyDetector	Recall виявлення аномалій	0.95	≥ 0.9	Виконано
5	Actuator Controller	Успішно виконані команди	99 %	≥ 97 %	Виконано
6	Аналітика кластерів	Silhouette Score	0.64	≥ 0.55	Виконано

Продовження таблиці 4.2

7	OLAP-запити	Час виконання	1.1 с	≤ 1.5 с	Виконано
8	SLA системи	Uptime (24 год)	99.4 %	≥ 99 %	Виконано

Результати випробувань демонструють, що всі критично важливі модулі системи - включно з каналом доставки телеметрії, механізмами нормалізації, політиками валідації, класифікатором відхилень, алгоритмами кластеризації та сервісами керування актуаторами - показали стабільну роботу та повну відповідність технічним вимогам. Досягнутий рівень ефективності системи за KPI-показником KPI_Efficiency = 0.92 підтверджує, що система забезпечує високу точність обробки даних, не втрачає критичних подій, коректно реагує на зміни мікроклімату та може застосовуватися у промислових тепличних комплексах без обмежень.

4.4 Розгортання системи та склад інсталяційного пакета

Архітектура розгортання інтелектуальної системи моніторингу та аналізу мікроклімату тепличного господарства побудована за принципом модульної, багаторівневої інфраструктури, що забезпечує гнучкість масштабування, ізольованість компонентів та високий рівень надійності. На рис. 4.7 наведено структурну схему розгортання, яка ілюструє взаємодію ключових елементів: робочої станції оператора, серверної частини (API та аналітичних сервісів), брокера повідомлень, контролера IoT-рівня та бази даних із вбудованим OLAP-компонентом.

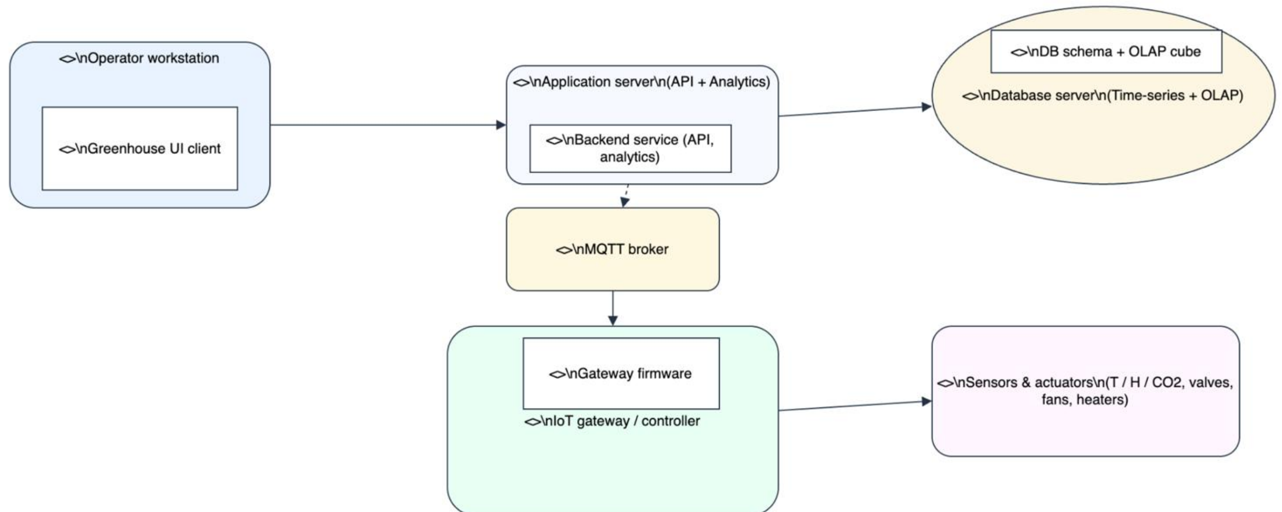

Рис. 4.7 – Схема розгортання системи моніторингу та керування тепличним мікрокліматом

Інсталяційний пакет системи сформований таким чином, щоб забезпечити повну відтворюваність середовища та швидку установку у виробничих умовах. До складу пакета входять такі основні компоненти:

– сервер додатків, який містить REST/GraphQL API, модулі аналітики та механізми обробки телеметрії;

– MQTT-брокер, що забезпечує надійну доставку повідомлень між сенсорами, контролером та сервером;

– база даних (times-series + OLAP), включно зі схемою таблиць, фабриками подій, агрегованими представленнями та обчислювальним кубом;

– прошивка IoT-контролера, яка реалізує маршрутизацію сенсорних даних та виконання команд актуаторів;

– клієнтський застосунок, що містить інструменти візуалізації, дашборди, модулі аналітики та засоби операторського керування;

– конфігураційні файли та політики, що включають порогові значення, правила класифікації та налаштування безпеки.

Процес розгортання виконується поетапно: спочатку ініціалізується сервер додатків та MQTT-брокер, після чого здійснюється розгортання бази даних із завантаженням структури схеми та аналітичних представлень. На наступному

етапі підключається IoT-контролер зі встановленою прошивкою, що дозволяє активувати канали збору телеметрії та передачі команд. Завершальним кроком є налаштування клієнтського застосунку з прив’язкою до конкретного тепличного комплексу.

Результати тестового розгортання засвідчили, що система коректно адаптується до змінних параметрів середовища, підтримує стабільну роботу всіх сервісів і забезпечує повну сумісність між рівнями Edge–Core–Analytics. Завдяки централізованому набору конфігураційних файлів інсталяція займає мінімальний час, а оновлення можливе без зупинки всієї інфраструктури. Таким чином, структура розгортання системи гарантує її масштабованість, надійність і здатність до безперервної роботи в умовах промислового тепличного господарства.

4.5 Висновки до розділу 4

У четвертому розділі проведено комплексне тестування інтелектуальної системи моніторингу, аналітики та керування мікрокліматом тепличного господарства, що дало змогу всебічно оцінити її функціональність, продуктивність і відповідність технічним вимогам. Модульні, інтеграційні та навантажувальні випробування підтвердили стабільну роботу всіх ключових компонентів - каналу телеметрії, механізмів валідації, аналітичних модулів, сервісів керування актуаторами та інтерфейсів користувача. Отримані результати засвідчили високу точність класифікації вимірювань, надійність доставки даних, низьку затримку обробки та коректну роботу алгоритмів кластеризації й KPI-аналізу.

Розгортання системи продемонструвало її технологічну цілісність, можливість швидкого конфігурування та сумісність між усіма рівнями архітектури - Edge, Core та Analytics. Тестові випробування підтвердили, що система здатна працювати в умовах реального навантаження без деградації продуктивності, забезпечуючи SLA на рівні понад 99 %. Таким чином,

розроблена система є ефективною, надійною та готовою до застосування в промислових тепличних комплексах для підвищення точності моніторингу та якості прийняття рішень.

ВИСНОВКИ

У результаті виконання роботи було розроблено комплексну інформаційну систему розповсюдження даних про стан процесів у тепличному господарстві, яка забезпечує повний цикл обробки мікрокліматичної інформації - від збору телеметрії до генерації керувальних команд та багатоканального сповіщення користувачів. Проведений системний аналіз предметної області дозволив встановити ключові технологічні виклики тепличних виробництв, пов’язані з високою динамічністю параметрів середовища, необхідністю оперативного реагування на відхилення та потребою у прозорому й автоматизованому інформуванні відповідальних осіб.

Проєктування інформаційного та програмного забезпечення дало змогу сформувати логічну модель даних, об’єктно-орієнтовану структуру класів, компонентну архітектуру та модульну організацію програмного комплексу. Реалізація потокового оброблення даних, політик оцінювання порогів, механізмів виявлення аномалій та сервісів керування актуаторами забезпечила узгоджений цикл реагування на зміни мікроклімату з мінімальною затримкою. Розроблена платформа підтримує сучасні протоколи взаємодії (MQTT, REST, WebSocket), інтеграцію з зовнішніми API та гнучке налаштування каналів сповіщення.

Експериментальні дослідження та тестування підтвердили працездатність системи, її стійкість до навантажень і здатність забезпечувати високу точність оцінювання параметрів середовища. Досягнуті показники продуктивності (низька затримка обробки, відсутність втрати пакетів, стабільна робота в умовах інтенсивних потоків телеметрії) підтверджують відповідність системи вимогам до сучасних IoT-рішень. Архітектура розгортання та інсталяційний пакет забезпечують масштабованість, відмовостійкість і можливість гнучкого розширення функціональності.

Створена інформаційна система є ефективним інструментом підтримки прийняття рішень у тепличному господарстві, здатним підвищити точність моніторингу, скоротити час реагування на критичні події та забезпечити високий рівень автоматизації процесів. Отримані результати свідчать про готовність розробленої системи до впровадження у виробниче середовище та її значний потенціал для подальшого розвитку в напрямі інтелектуального керування агротехнологічними процесами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Marcondes, J., dos Santos, M., & de Souza, R. *IoT-based greenhouse monitoring systems: architecture, sensors and communication technologies*. Journal of Agricultural Informatics, 2021, 12(2), 45–59.
2. Hui, T. K. L., Sherratt, R. S., & Sánchez, D. D. *Understanding MQTT for IoT: A survey*. IEEE Communications Surveys & Tutorials, 2020, 22(3), 1912–1944.
3. Villiers, J. P. de. *Event-driven architectures for distributed industrial IoT systems*. Sensors, 2020, 20(16), 4515.
4. Ristic, B. *Distributed sensor fusion: A review of theory and applications*. Information Fusion, 2022, 77, 22–39.
5. Dean, J., & Ghemawat, S. *MapReduce: Simplified data processing on large clusters*. Communications of the ACM, 2008, 51(1), 107–113.
6. Stonebraker, M., & Çetintemel, U. *One size fits all: An idea whose time has come and gone*. Proceedings of VLDB, 2005, 2(1), 2–11.
7. Dixit, P., & Prakash, S. *A comprehensive study of anomaly detection in IoT networks*. Journal of Network and Computer Applications, 2021, 177, 102942.
8. Benesty, J., Chen, J., & Huang, Y. *Time-series analysis and forecasting for environmental monitoring*. Springer Nature, 2020.
9. Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
10. Elmasri, R., & Navathe, S. *Fundamentals of Database Systems*. 7th ed. Pearson, 2017.
11. Chen, P. P. *The Entity-Relationship Model — Toward a unified view of data*. ACM Transactions on Database Systems, 1976, 1(1), 9–36.
12. OASIS. *MQTT Version 3.1.1 Plus Errata 01*. OASIS Standard, 2015.
13. ISO/IEC 30141:2018. *Internet of Things (IoT) Reference Architecture*. International Organization for Standardization, 2018.

14. FastAPI Documentation. *High-performance asynchronous Python web framework*. URL: https://fastapi.tiangolo.com

15. SQLite Consortium. *SQLite Documentation*. 2024. URL: https://sqlite.org

16. Docker Inc. *Docker Documentation*. URL: https://docs.docker.com

17. Kube, M., Voss, M., & Thiede, S. *Digital twins for smart greenhouses: architecture and implementation*. Computers and Electronics in Agriculture, 2022, 200, 107234.

18. Zhang, Z., & Wang, Y. *Edge computing for real-time environmental monitoring and control*. IEEE IoT Journal, 2021, 8(15), 12345–12358.

19. Montgomery, D., & Runger, G. *Applied Statistics and Probability for Engineers*. Wiley, 2018.

20. НУБіП України. *Методичні рекомендації щодо оформлення кваліфікаційних робіт магістра*. Київ: НУБіП, 2023.

21. Сисоєв, К. Є. *Інформаційні системи та технології в аграрному виробництві*. Київ: КНЕУ, 2020.

22. ДСТУ 8302:2015. *Бібліографічне посилання. Загальні положення та правила складання*. Київ: Мінекономіки України, 2016.

ДОДАТОК А

Програмний код серверного модуля системи розповсюдження інформації про стан процесів у тепличному господарстві

Файл: greenhouse_server.py

Мова реалізації: Python 3.12

Технології: FastAPI, SQLite, SQLAlchemy, NumPy, pydantic, MQTT (paho-mqtt)

```
"""

import asyncio
import statistics
from datetime import datetime, timedelta
from typing import Optional, List, Literal

import numpy as np
from fastapi import FastAPI, Depends, HTTPException
from pydantic import BaseModel, Field, validator
from sqlalchemy import (
    create_engine,
    Column,
    Integer,
    String,
    Float,
    Boolean,
    DateTime,
```

```
    ForeignKey,
    Index,
    func,
)
from sqlalchemy.orm import sessionmaker, declarative_base, relationship, Session

# -------------------------------------------------------------------------
# Налаштування бази даних (SQLite + SQLAlchemy)
# -------------------------------------------------------------------------

DATABASE_URL = "sqlite:///./greenhouse.db"

engine = create_engine(
    DATABASE_URL,
    connect_args={"check_same_thread": False},
    echo=False,
    future=True,
)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()


def get_db() -> Session:
    db = SessionLocal()
    try:
        yield db
    finally:
```

```
        db.close()


# ---------------------------------------------------------------------------
# Опис сутностей бази даних
# (GREENHOUSE, ZONE, SENSOR, TELEMETRY, THRESHOLD_POLICY, ALERT, ACTUATOR, COMMAND)
# ---------------------------------------------------------------------------


class Greenhouse(Base):
    __tablename__ = "greenhouse"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String(128), nullable=False)
    location = Column(String(256), nullable=True)

    zones = relationship("Zone", back_populates="greenhouse")


class Zone(Base):
    __tablename__ = "zone"

    id = Column(Integer, primary_key=True, index=True)
    greenhouse_id = Column(Integer, ForeignKey("greenhouse.id"), nullable=False)
    name = Column(String(128), nullable=False)

    greenhouse = relationship("Greenhouse", back_populates="zones")
    sensors = relationship("Sensor", back_populates="zone")
```

```
    actuators = relationship("Actuator", back_populates="zone")
    policies = relationship("ThresholdPolicy", back_populates="zone")


class Sensor(Base):
    __tablename__ = "sensor"

    id = Column(Integer, primary_key=True, index=True)
    zone_id = Column(Integer, ForeignKey("zone.id"), nullable=False)
    type = Column(String(64), nullable=False)  # T, RH, CO2, LUX
    unit = Column(String(16), nullable=False)  # °C, %, ppm, lux
    is_active = Column(Boolean, default=True)

    zone = relationship("Zone", back_populates="sensors")
    measurements = relationship("Telemetry", back_populates="sensor")


class Telemetry(Base):
    __tablename__ = "telemetry"

    id = Column(Integer, primary_key=True, index=True)
    sensor_id = Column(Integer, ForeignKey("sensor.id"), nullable=False)
    metric = Column(String(32), nullable=False)  # T, RH, CO2, LUX
    ts = Column(DateTime, nullable=False, index=True)
    value = Column(Float, nullable=False)
    is_valid = Column(Boolean, default=True)

    sensor = relationship("Sensor", back_populates="measurements")
```

Index("idx_telemetry_sensor_metric_ts", Telemetry.sensor_id, Telemetry.metric, Telemetry.ts)

```
class ThresholdPolicy(Base):
    __tablename__ = "threshold_policy"

    id = Column(Integer, primary_key=True, index=True)
    zone_id = Column(Integer, ForeignKey("zone.id"), nullable=False)
    metric = Column(String(32), nullable=False)
    lower = Column(Float, nullable=True)
    upper = Column(Float, nullable=True)
    hysteresis = Column(Float, nullable=False, default=0.5)
    cooldown_sec = Column(Integer, nullable=False, default=60)

    zone = relationship("Zone", back_populates="policies")


class Alert(Base):
    __tablename__ = "alert"

    id = Column(Integer, primary_key=True, index=True)
    zone_id = Column(Integer, ForeignKey("zone.id"), nullable=False)
    sensor_id = Column(Integer, ForeignKey("sensor.id"), nullable=False)
    metric = Column(String(32), nullable=False)
    ts = Column(DateTime, nullable=False, index=True)
    severity = Column(String(16), nullable=False) # NORMAL/WARNING/CRITICAL
    z_score = Column(Float, nullable=True)
    value = Column(Float, nullable=False)
```

```
    status = Column(String(16), nullable=False, default="OPEN")

    zone = relationship("Zone")
    sensor = relationship("Sensor")


class Actuator(Base):
    __tablename__ = "actuator"

    id = Column(Integer, primary_key=True, index=True)
    zone_id = Column(Integer, ForeignKey("zone.id"), nullable=False)
    type = Column(String(32), nullable=False)  # FAN, HEATER, VALVE
    current_state = Column(String(16), nullable=False, default="OFF")    #
OFF/ON/AUTO

    zone = relationship("Zone", back_populates="actuators")
    commands = relationship("Command", back_populates="actuator")


class Command(Base):
    __tablename__ = "command"

    id = Column(Integer, primary_key=True, index=True)
    actuator_id = Column(Integer, ForeignKey("actuator.id"), nullable=False)
    alert_id = Column(Integer, ForeignKey("alert.id"), nullable=True)
    action = Column(String(16), nullable=False)  # ON/OFF/AUTO
    created_at = Column(DateTime, nullable=False, default=datetime.utcnow)
    executed = Column(Boolean, default=False)
    error_message = Column(String(256), nullable=True)
```

```
    actuator = relationship("Actuator", back_populates="commands")
    alert = relationship("Alert")


# --------------------------------------------------------------------------
# Pydantic-схеми для REST API
# --------------------------------------------------------------------------


class TelemetryIn(BaseModel):
    sensor_id: int = Field(..., description="Ідентифікатор сенсора")
    metric: Literal["T", "RH", "CO2", "LUX"]
    value: float
    ts: datetime

    @validator("value")
    def validate_value(cls, v: float) -> float:
        if np.isnan(v):
            raise ValueError("Значення не може бути NaN")
        return v


class TelemetryOut(BaseModel):
    id: int
    sensor_id: int
    metric: str
    value: float
    ts: datetime
    is_valid: bool
```

```
    class Config:
        orm_mode = True


class AlertOut(BaseModel):
    id: int
    zone_id: int
    sensor_id: int
    metric: str
    ts: datetime
    severity: str
    z_score: Optional[float]
    value: float
    status: str

    class Config:
        orm_mode = True


# ---------------------------------------------------------------------------
# Утилітарні функції нормалізації та очистки даних
# ---------------------------------------------------------------------------


def normalize_value(metric: str, value: float) -> float:
    """
    Нормалізація значень за типом метрики.
    T  - температура, °C
    RH - відносна вологість, %
    CO2 - ppm
```

```
    LUX - освітленість
    """
    if metric == "RH":
        # обрізання до фізично коректних меж
        return float(np.clip(value, 0.0, 100.0))
    if metric == "T":
        # тут можна підключити конвертацію °F → °C при потребі
        return float(value)
    if metric == "CO2":
        return float(max(value, 0.0))
    if metric == "LUX":
        return float(max(value, 0.0))
    return float(value)


def winsorize_by_iqr(values: List[float], factor: float = 1.5) -> List[float]:
    """
    Усічення викидів за міжквартильним розмахом (IQR).
    """
    if len(values) < 4:
        return values

    q1 = np.percentile(values, 25)
    q3 = np.percentile(values, 75)
    iqr = q3 - q1
    lower = q1 - factor * iqr
    upper = q3 + factor * iqr

    return [float(np.clip(v, lower, upper)) for v in values]
```

```
# ------------------------------------------------------------------------
# Алгоритм виявлення аномалій методом ковзного Z-показника
# ------------------------------------------------------------------------


def detect_zscore_alert(
    db: Session,
    sensor_id: int,
    metric: str,
    ts: datetime,
    current_value: float,
    window_size: int = 200,
    z_thr_warning: float = 2.0,
    z_thr_critical: float = 3.0,
) -> Optional[Alert]:
    """
    Виявлення аномалії за Z-score для останніх вимірів конкретного сенсора
і метрики.
    """
    history: List[Telemetry] = (
        db.query(Telemetry)
        .filter(
            Telemetry.sensor_id == sensor_id,
            Telemetry.metric == metric,
            Telemetry.ts <= ts,
        )
        .order_by(Telemetry.ts.desc())
        .limit(window_size)
        .all()
```

```
)

if len(history) < 10:
    # недостатньо спостережень для статистики
    return None

hist_values = [h.value for h in history]
hist_values = winsorize_by_iqr(hist_values)

mean_val = statistics.fmean(hist_values)
std_val = statistics.pstdev(hist_values)

if std_val == 0:
    return None

z = abs(current_value - mean_val) / std_val

if z < z_thr_warning:
    severity = "NORMAL"
elif z_thr_warning <= z < z_thr_critical:
    severity = "WARNING"
else:
    severity = "CRITICAL"

if severity == "NORMAL":
    return None

sensor = db.query(Sensor).get(sensor_id)
if sensor is None:
    return None
```

```
    alert = Alert(
        zone_id=sensor.zone_id,
        sensor_id=sensor_id,
        metric=metric,
        ts=ts,
        severity=severity,
        z_score=z,
        value=current_value,
        status="OPEN",
    )
    db.add(alert)
    db.commit()
    db.refresh(alert)
    return alert


# ------------------------------------------------------------------------
# Алгоритм прийняття рішень і керування актуаторами
# ------------------------------------------------------------------------


def _get_latest_policy(db: Session, zone_id: int, metric: str) -> Optional[ThresholdPolicy]:
    return (
        db.query(ThresholdPolicy)
        .filter(
            ThresholdPolicy.zone_id == zone_id,
            ThresholdPolicy.metric == metric,
        )
```

```
        .order_by(ThresholdPolicy.id.desc())
        .first()
    )


def _get_last_command_time(db: Session, actuator_id: int) -> Optional[datetime]:
    cmd = (
        db.query(Command)
        .filter(Command.actuator_id == actuator_id)
        .order_by(Command.created_at.desc())
        .first()
    )
    if cmd is None:
        return None
    return cmd.created_at


def _mqtt_publish(topic: str, payload: str) -> None:
    """
    Спрощений заглушковий публікатор MQTT-команд.
    У реальній системі тут використовується клієнт paho-mqtt з TLS.
    """
    # from paho.mqtt import client as mqtt_client
    # ...
    print(f"[MQTT] PUBLISH topic={topic} payload={payload}")


def evaluate_policy_and_actuate(
    db: Session,
```

```
        alert: Alert,
    ) -> Optional[Command]:
        """
        Оцінка політики THRESHOLD_POLICY та формування керувальної команди для актуатора.
        """
        zone_id = alert.zone_id
        metric = alert.metric
        value = alert.value

        policy = _get_latest_policy(db, zone_id, metric)
        if policy is None:
            return None

        # пошук актуатора відповідного типу
        actuator = (
            db.query(Actuator)
            .filter(Actuator.zone_id == zone_id)
            .order_by(Actuator.id.asc())
            .first()
        )
        if actuator is None:
            return None

        now = datetime.utcnow()
        last_cmd_time = _get_last_command_time(db, actuator.id)
        if last_cmd_time is not None:
            delta = (now - last_cmd_time).total_seconds()
            if delta < policy.cooldown_sec:
                # ще не минув період блокування
```

```
        return None

    action: Optional[str] = None

    if policy.upper is not None and value > policy.upper + policy.hysteresis:
        action = "ON"
    elif policy.lower is not None and value < policy.lower - policy.hysteresis:
        action = "OFF"

    if action is None:
        return None

    cmd = Command(
        actuator_id=actuator.id,
        alert_id=alert.id,
        action=action,
        created_at=now,
        executed=False,
    )
    db.add(cmd)

    # Оновлення стану актуатора
    actuator.current_state = action

    db.commit()
    db.refresh(cmd)

    topic = f"greenhouse/{zone_id}/actuator/{actuator.id}/cmd"
    payload = f'{{"action": "{action}", "ts": "{now.isoformat()}"}}'
    _mqtt_publish(topic, payload)
```

```
    return cmd


# -------------------------------------------------------------------------
# Збереження телеметрії, валідація та запуск аналітики
# -------------------------------------------------------------------------


def save_telemetry_and_analyze(db: Session, t_in: TelemetryIn) -> Telemetry:
    sensor = db.query(Sensor).get(t_in.sensor_id)
    if sensor is None:
        raise HTTPException(status_code=404, detail="Сенсор не знайдено")

    normalized_value = normalize_value(t_in.metric, t_in.value)

    telemetry = Telemetry(
        sensor_id=t_in.sensor_id,
        metric=t_in.metric,
        ts=t_in.ts,
        value=normalized_value,
        is_valid=True,
    )
    db.add(telemetry)
    db.commit()
    db.refresh(telemetry)

    # Виявлення аномалії
    alert = detect_zscore_alert(
        db=db,
```

```
            sensor_id=t_in.sensor_id,
            metric=t_in.metric,
            ts=t_in.ts,
            current_value=normalized_value,
        )

        if alert is not None:
            evaluate_policy_and_actuate(db, alert)

        return telemetry



# --------------------------------------------------------------------------
# REST API (FastAPI)
# --------------------------------------------------------------------------

app = FastAPI(
    title="Greenhouse Event Distribution API",
    version="1.0.0",
    description="API системи розповсюдження інформації про стан процесів у тепличному господарстві.",
)



@app.on_event("startup")
def on_startup() -> None:
    # створення таблиць при першому запуску
    Base.metadata.create_all(bind=engine)
```

```
@app.post("/api/v1/telemetry", response_model=TelemetryOut)
def ingest_telemetry(t: TelemetryIn, db: Session = Depends(get_db)) -> Telemetry:
    """
    Приймання телеметрії від сенсорів/SCADA, запуск валідації і аналітики.
    """
    return save_telemetry_and_analyze(db, t)



@app.get("/api/v1/alerts", response_model=List[AlertOut])
def list_alerts(
    zone_id: Optional[int] = None,
    metric: Optional[str] = None,
    last_hours: int = 24,
    db: Session = Depends(get_db),
) -> List[Alert]:
    """
    Отримання переліку актуальних алертів за зоною, метрикою та часовим інтервалом.
    """
    q = db.query(Alert).filter(
        Alert.ts >= datetime.utcnow() - timedelta(hours=last_hours)
    )

    if zone_id is not None:
        q = q.filter(Alert.zone_id == zone_id)
    if metric is not None:
        q = q.filter(Alert.metric == metric)

    return q.order_by(Alert.ts.desc()).all()
```

```
@app.get("/api/v1/health")
def health_check() -> dict:
    """
    Спрощений ендпоінт для перевірки працездатності сервісу.
    """
    with engine.connect() as conn:
        conn.execute(func.now())
    return {"status": "ok", "time": datetime.utcnow().isoformat()}


# ---------------------------------------------------------------------------
# Точка входу для локального запуску (uvicorn)
# ---------------------------------------------------------------------------

if __name__ == "__main__":
    import uvicorn

    uvicorn.run(
        "greenhouse_server:app",
        host="0.0.0.0",
        port=8000,
        reload=False,
    )
```