Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя
Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних систем та мереж
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: *Розподілена комп'ютерна система агрегації та аналізу даних з IoT-пристроїв*

Виконав: студент 4 курсу, групи СІ-41
спеціальності 123 «Комп'ютерна інженерія»
(шифр і назва спеціальності)

(підпис) Баландюк Ю.Р. (прізвище та ініціали)

Керівник (підпис) Луцків А.М. (прізвище та ініціали)

Нормоконтроль (підпис) Луцик Н.С. (прізвище та ініціали)

Завідувач кафедри (підпис) Осухівська Г.М. (прізвище та ініціали)

Рецензент (підпис) Литвиненко Я.В. (прізвище та ініціали)

Тернопіль
2026

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет *комп'ютерно-інформаційних систем і програмної інженерії*
(повна назва факультету)

Кафедра *комп'ютерних систем та мереж*
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ *Осухівська Г.М.*
(підпис) (прізвище та ініціали)
«25» квітня 2026 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня *бакалавр*
(назва освітнього ступеня)

за спеціальністю *123 «Комп'ютерна інженерія»*
(шифр і назва спеціальності)

студента *Баландюку Юрій Руслановичу*
(прізвище, ім'я, по батькові)

1. Тема роботи *Розпоідлена комп'ютерна система агрегації та аналізу даних з IoT-пристроїв*

Керівник роботи *Луцків Андрій Мирославович, к.т.н., доцент*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «*24*» *квітня 2026 року № 4/9-188*.

2. Термін подання студентом завершеної роботи *15.06.2026*

3. Вихідні дані до роботи *Технічне завдання*

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. 1. Аналіз технічного завдання
2. Проєктна частина
3. Практична частина
4. Безпека життєдіяльності, основи охорони праці.
Висновки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1. Структурна схема компонентів розподіленої системи обробки даних
2. Структурна схема порівняння повномасштабного та реалізованого технологічних стеків в умовах апаратних обмежень
3. Структурна схема медальйоної архітектури збереження та обробки даних
4. Структурна схема опрацювання повідомлення в реальному часі

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи охорони праці	*доц. каф. МТ Сенчишин В.С.*		

7. Дата видачі завдання *25.04.2026 р.*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	*Розробка технічного завдання*	*26.01 – 02.02*	
2.	*Робота над першим розділом «Аналіз технічного завдання»*	*03.02 – 15.02*	
3.	*Робота над другим розділом «Проєктна частина»*	*20.04 – 28.04*	
4.	*Робота над третім розділом «Практична частина»*	*29.04 – 20.05*	
5.	*Робота над четвертим розділом «Безпека життєдіяльності, основи охорони праці»*	*21.05 – 25.05*	
6.	*Оформлення пояснювальної записки і графічного матеріалу*	*26.05 – 7.06*	
7.	*Перевірка на академічний плагіат, перевірка керівником та консультантами*	*8.06 – 14.06*	
8.	*Попередній захист кваліфікаційної роботи бакалавра*	*15.06 – 21.06*	
9.	*Захист кваліфікаційної роботи бакалавра*	*22.06*	

Студент _____ *Баландюк Юрій Русланович*
(підпис) (прізвище та ініціали)

Керівник роботи _____ *Луцків Андрій Мирославович*
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Баландюк Ю.Р. Розподілена комп’ютерна система агрегації та аналізу даних з IoT-пристроїв: робота на здобуття кваліфікаційного ступеня бакалавра: спец. 123 — комп’ютерна інженерія. Тернопіль: Тернопільський національний технічний університет імені Івана Пулюя, 2026.

Ключові слова: великі дані, потокова обробка, інфраструктура, On-premise рішення, контейнер, Apache Kafka, Apache Spark, SparkSQL, MinIO, Apache Iceberg, Apache Thrift Server, Apache Superset, Docker, Big Data

В ході виконання кваліфікаційної роботи бакалавра було спроєктовано та реалізовано розподілену комп’ютерну систему агрегації та аналізу даних з IoT-пристроїв. Архітектура системи базується на використанні брокера повідомлень Apache Kafka для збору телеметричних даних, кластера Apache Spark для обробки потоків у реальному часі та об’єктного сховища MinIO під керуванням формату Apache Iceberg. Візуалізація результатів аналізу забезпечується BI-платформою Apache Superset, а цілісність та масштабованість системи досягнута шляхом контейнеризації компонентів за допомогою Docker.

Пояснювальна записка кваліфікаційної роботи містить чотири розділи.

В першому розділі обґрунтовується актуальність теми розробки розподіленої комп’ютерної системи агрегації та аналізу даних з IoT-пристроїв. Визначаються вимоги до продуктивності та масштабування архітектури, здійснюється обґрунтований вибір стеку технологій. Також коротко описуються принципи роботи використовуваних технологій та концепція об’єктного зберігання даних.

Другий розділ присвячений розробці архітектури системи агрегації та функціонуванню конвеєра обробки даних у контейнеризованому середовищі.

Розглядається специфіка побудови мікросервісної структури за допомогою Docker, інтеграція брокера повідомлень Apache Kafka, системи розподілених обчислень Apache Spark та об’єктного сховища MinIO з табличним форматом Apache Iceberg, а також проєктування схеми передачі телеметрії від джерела до засобів візуалізації. Також розглядається роль Spark Thrift Server у забезпеченні доступу до даних та обґрунтовується вибір Apache Superset як інструменту бізнес-аналітики.

В третьому розділі реалізовується практичне розгортання розробленої розподіленої системи. Описано процес налаштування брокерів Kafka, конфігурування Spark-кластера для потокової обробки та створення структурованого сховища на базі MinIO, а також налаштуванню інтерактивних дашбордів для візуалізації аналітичних даних.

Четвертий розділ присвячений аспектам безпеки життєдіяльності та основи охорони праці.

ANNOTATION

Balandiuk Y.R. Distributed Computer System for Aggregation and Analysis of Data from IoT Devices: Bachelor's Graduation Thesis: speciality 123 — computer engineering. Ternopil: Ternopil Ivan Puluj National Technical University, 2026.

Keywords: Big Data, stream processing, infrastructure, On-premise solution, container, Apache Kafka, Apache Spark, SparkSQL, MinIO, Apache Iceberg, Apache Thrift Server, Apache Superset, Docker, Big Data.

During the execution of the bachelor's qualification thesis, a distributed computer system for aggregation and analysis of data from IoT devices was designed and implemented. The system architecture is based on the use of the Apache Kafka message broker for collecting telemetry data, an Apache Spark cluster for real-time stream processing, and MinIO object storage managed by the Apache Iceberg table format. Visualization of analysis results is provided by the Apache Superset BI platform, while system integrity and scalability are achieved through containerization of components using Docker.

The explanatory note of the qualification thesis consists of four sections.

The first section substantiates the relevance of developing a distributed computer system for aggregation and analysis of data from IoT devices. Performance and scalability requirements of the architecture are defined, and a justified selection of the technology stack is carried out. The operating principles of the utilized technologies and the concept of object-based data storage are also briefly described.

The second section is devoted to the development of the system architecture and the functioning of the data processing pipeline in a containerized environment. It examines the specifics of building a microservice structure using Docker, the

integration of the Apache Kafka message broker, the Apache Spark distributed computing system, and MinIO object storage with the Apache Iceberg table format, as well as the design of the telemetry transmission scheme from source to visualization tools. The role of Spark Thrift Server in providing data access is also considered, and the choice of Apache Superset as a business intelligence tool is justified.

The third section presents the practical deployment of the developed distributed system. It describes the process of configuring Kafka brokers, setting up the Spark cluster for stream processing, creating structured storage based on MinIO, and configuring interactive dashboards for visualization of analytical data.

The fourth section is devoted to occupational safety and health and safety fundamentals.

ЗМІСТ

СПИСОК СКОРОЧЕНЬ

ACID - Atomicity, Consistency, Isolation, Durability

GCP - Google Cloud Platform

AWS - Amazon Web Services

ODBC - Open Database Connectivity

JDBC - Java Database Connectivity

BI - Business Intelligence

S3 - Simple Storage Service

RBAC - Role-Based Access Control

API - Application Programming Interface

DAG - Directed Acyclic Graph

IAM - Identity and Access Management

K8S - Kubernetes

ВСТУП

У сучасному світі стрімкий розвиток концепції «Інтернету речей» (IoT) призвів до генерації колосальних обсягів телеметричних даних, обробка яких стала критично важливою для моніторингу, прогнозної аналітики та прийняття рішень у реальному часі. Це зумовлює високий попит на архітектурні рішення, здатні забезпечити мінімальну затримку при зборі даних та високу надійність їх зберігання. Найкращим підходом для вирішення таких завдань є використання розподілених систем обробки потоків, що базуються на мікросервісній архітектурі та масштабованих об'єктних сховищах.

В даній кваліфікаційній роботі реалізовується розподілена комп'ютерна система агрегації та аналізу даних з IoT-пристроїв, побудована на базі брокера повідомлень Apache Kafka та фреймворку розподілених обчислень Apache Spark. Використання Spark Structured Streaming дозволяє виконувати агрегацію та фільтрацію даних безпосередньо «на льоту», мінімізуючи час від появи події на пристрої до її відображення в аналітичній панелі. Особливістю архітектури є застосування концепції Data Lakehouse та «Медальйон» шляхом поєднання об'єктного сховища MinIO із сучасним табличним форматом Apache Iceberg. Такий підхід дозволяє поєднати гнучкість «озер даних» (datalake) із транзакційністю (ACID) та швидкістю аналітичних баз даних, що є необхідним для коректної обробки потоків IoT-даних.

Мостом для аналітики даних виступає Spark Thrift Server сервіс, який забезпечує стандартний інтерфейс JDBC/ODBC доступу до оброблених даних для BI-системи Apache Superset. Завдяки високій швидкості виконання запитів до формату Iceberg, аналітична панель забезпечує миттєвий відгук навіть при роботі з великими історичними масивами даних. Це дає змогу створювати інтерактивні дашборди та проводити глибокий історичний аналіз накопичених IoT-даних без зупинки процесу збору.

На основі технічного завдання здійснено аналіз вимог до програмного забезпечення, що дозволило спроєктувати систему, де кожен компонент ізольований у Docker-контейнері.

Розгортання системи у контейнеризованому середовищі забезпечує високу доступність та відмовостійкість, а в разі збою окремого сервісу, механізми реплікації Apache Kafka та контрольні точки (checkpoints) в Apache Spark гарантують збереження цілісності даних та автоматичне відновлення процесу обробки.

РОЗДІЛ 1 АНАЛІЗ ТЕХНІЧНОГО ЗАВДАННЯ

1.1 Аналіз вимог до розподіленої комп'ютерної системи агрегації та аналізу даних з IoT-пристроїв

Для початку проаналізуємо проєктовану систему, що стоїть перед нами, визначимо її основні характеристики та мінімальні вимоги [1]. Проєктування системи для роботи з IoT-даними вимагає розв'язання задачі «низької затримки» (low latency) при одночасному забезпеченні високої якості та доступності даних для аналізу. На відміну від класичних систем обробки, розроблювана архітектура базується на принципі реального часу (streaming), де опрацювання відбувається безпосередньо під час надходження подій.

Для ефективного керування даними та їхньою структурою в системі доцільно впровадити медальйонну модель (medallion architecture) [24], детальна логіка якої наведена на структурній схемі медальйонної архітектури збереження та обробки даних. Вона дозволяє розділити процес обробки IoT-даних на такі функціональні рівні:

– рівень отримання даних «Bronze» реалізує налаштування брокера повідомлень Apache Kafka для гарантованої доставки необроблених даних від датчиків;

– рівень обробки «Silver» використовує Spark Structured Streaming для очищення, перевірки та нормалізації потокових даних;

– аналітичний рівень «Gold» забезпечує формування фінальних даних шляхом агрегації IoT-даних за часовими проміжками, типами пристроїв або іншими ключовими метриками, що створює надійне джерело даних для бізнес-аналітики.

В ході виконання кваліфікаційної роботи повинні бути виконані такі завдання [1]:

– спроєктована загальна структура та функціональна схема розподіленої комп'ютерної системи агрегації та аналізу даних з IoT-пристроїв;

– визначені чіткі вимоги до апаратної та програмної частини, такі як кількість обчислювальних ресурсів для Spark-виконавців, обсяг RAM для Kafka-брокерів, параметри дискового простору для MinIO сервісу [22];

– вибрані та обґрунтовані інструменти для реалізації концепції Data Lakehouse (S3-сумісне сховище, Apache Iceberg) [27];

– налаштована конфігурація Apache Kafka для забезпечення надійної черги повідомлень з високою пропускною здатністю;

– реалізовані алгоритми обробки, агрегації та фільтрації IoT-даних у реальному часі за допомогою Apache Spark;

– розроблені компоненти системи ізольовані в Docker-контейнерах для забезпечення мобільності та швидкого розгортання [18];

– налаштована інтеграція Apache Superset із розподіленим сховищем для візуалізації результатів аналізу;

– проведено тестування системи на предмет стійкості до навантажень та коректності відновлення обробки після збоїв (fault tolerance).

Також важливо передбачити потреби у масштабуванні системи, оскільки обсяги телеметрії з IoT-пристроїв мають тенденцію до експоненціального зростання. Архітектура повинна підтримувати горизонтальне масштабування через збільшення кількості партицій у Kafka та збільшення обчислювальної потужності Spark-кластера без потреби в зміні логіки додатків. Особлива увага також приділяється масштабованості об'єктного сховища MinIO, яке має забезпечувати стабільно низьку затримку при доступі до даних навіть за умови значного накопичення історичних даних.

З точки зору безпеки, вимоги до системи включають повну ізоляцію мережевого трафіку між контейнерами та чітке розмежування прав доступу до

аналітичних ресурсів [9]. Для сховища MinIO це реалізується через політику керування ключами доступу та розмежування прав на читання і запис у бакети, що запобігає несанкціонованим діям із даними. Також платформа Apache Superset вимагає налаштування рольової моделі доступу (RBAC), яка дозволяє створювати персоналізовані дашборди для різних груп користувачів.

Такий комплексний аналіз дозволяє створити не просто набір скриптів, а повноцінну інженерну платформу для роботи з Big Data, де кожен елемент захищений та готовий до зростаючих навантажень [14].

1.2 Огляд процесу побудови розподілених архітектур опрацювання потокових IoT-даних

Побудова сучасних комп'ютерних систем опрацювання потокових даних (data pipelines) базується на принципі послідовної трансформації інформації від сирого вигляду до аналітичних моделей. У системах інженерії великих даних розрізняють два основні підходи до архітектури, такі як пакетний (batch) та потоковий (streaming). Традиційна Lambda-архітектура передбачає розділення цих процесів, проте сучасні вимоги до IoT-систем стимулюють перехід до архітектури де всі дані розглядаються як безперервний потік [14].

Процес побудови розподіленої комп'ютерної систем опрацювання в межах даної роботи охоплює чотири етапи:

– Етап збору даних (data ingestion) полягає у отриманні повідомлень від розподілених IoT-джерел та їх тимчасовому зберіганні. Використання брокера повідомлень дозволяє обробляти пікові навантаження та гарантувати, що дані не будуть втрачені навіть при тимчасовій недоступності обчислювальних модулів. На цьому етапі формується «Bronze» рівень даних, де зберігається оригінальний JSON-формат подій [24].

– Етап обробки та трансформації (stream processing) виконується за допомогою розподілених обчислювальних рушіїв. Розподілена комп'ютерна система реалізує логіку фільтрації некоректних даних, приведення типів (перетворення початкових текстових значень у числові формати для забезпечення коректності математичних розрахунків та оптимізації обсягу збереженої інформації) та збагачення потоку додатковими метаданими. Саме тут відбувається перехід до «Silver» рівня де дані структурується у табличний формат. Важливою характеристикою цього етапу є підтримка механізму «exactly-once», як показано на блок-схемі послідовності обробки повідомлення у реальному часі, що гарантує обробку кожної події рівно один раз [21].

– Етап збереження та транзакційного керування (storage layer) у сучасних розподілени комп'ютерних системах базується на концепції Data Lakehouse. Використання відкритого табличного формату дозволяє організувати дані у вигляді партицій із кількома версіями, що забезпечує швидкий пошук та можливість виконання аналітичних запитів паралельно із записом нових даних. Це вирішує проблему ізоляції процесів, яка була критичною у ранніх архітектурах Big Data [27].

– Етап надання даних (data serving) завершує побудову розподіленої комп'ютерної системи, створюючи інтерфейс для кінцевих споживачів [4]. Процес передбачає агрегацію даних у «Gold» рівень, де формуються готові показники (середня температура за годину, кількість активних пристроїв тощо). Доступ до цих даних організовується через високорівневі SQL-інтерфейси, що дозволяє інтегрувати розподілену комп'ютерну систему із будь-якими сучасними системами бізнес-аналітики та візуалізації [26].

Така багаторівнева структура побудови розподіленої комп'ютерної системи забезпечує не лише швидкість обробки, а й гнучкість структури, дозволяючи легко додавати нові етапи аналізу або змінювати існуючу логіку трансформації без зупинки всього потоку IoT-даних [15].

1.3 Огляд можливих рішень до організації розподілених комп’ютерних систем агрегації та аналізу IoT-даних

При розробці комп’ютерної систем обробки великих даних ключовим питанням є вибір підходу до розгортання інфраструктури. На даний момент в інженерії даних домінують три основні підходи, кожен з яких має свої технічні та економічні особливості. Розглянемо та проаналізуємо їх детальніше для визначення оптимального середовища для реалізації нашої комп’ютреної системи.

Публічні хмарні рішення (public cloud), такі як Amazon Web Service, Google Cloud або Microsoft Azure, пропонують готові керовані сервіси для Big Data (наприклад, Amazon Redshift, Google BigQuery або Azure Databricks). Головною перевагою хмар є швидкість запуску та автоматичне масштабування ресурсів (auto-scaling). Однак при роботі з безперервними потоками IoT-даних, вартість хмарних послуг стрімко зростає через високу ціну за передачу трафіку та оренду обчислювальних ресурсів у режимі 24/7 [17]. Також виникає ризик залежності від конкретного провайдера.

Гібридні рішення (hybrid cloud) поєднують локальні потужності з можливостями публічних хмар. Це дозволяє зберігати критично важливі дані на власних серверах, а пікові навантаження обробляти в хмарі. Такий підхід є найбільш гнучким, проте він значно ускладнює архітектуру системи, вимагаючи налаштування складних мережевих шлюзів, синхронізації даних та подвійного контролю безпеки [10].

Локальні рішення (on-premise) передбачають розгортання всього технологічного стеку на власних або орендованих виділених серверах. Підхід забезпечує максимальний контроль над апаратними ресурсами, мережевою безпекою та фізичним розташуванням даних. Все ж таки такий варіант вимагає початкових витрат на обладнання або налаштування середовища. В довгостроковій перспективі такий підхід є найбільш економічно вигідним для

систем із постійним стабільним навантаженням, як характерним для IoT-моніторингу.

Після порівняльного аналізу для даної кваліфікаційної роботи було обрано «on-premise» модель із використанням контейнеризації. Такий вибір дозволяє нам поєднати переваги локального контролю з гнучкістю хмарних технологій завдяки використанню Docker. Інженерна реалізація цього підходу відображена на структурній схемі порівняння повномасштабного та реалізованого технологічних стеків в умовах апаратних обмежень.

Використання Open Source стеку (Apache Kafka, Apace Spark, MinIO, Apache Superset) з використанням власних ресурсів гарантує повну незалежність від сторонніх сервісів, забезпечує високу пропускну здатність внутрішньої мережі та дозволяє оптимізувати витрати на зберігання великих обсягів IoT-даних [22].

1.4 Огляд існуючих сервісів для побудови розподілених комп'ютерних систем агрегації та аналізу IoT-даних

Розробка розподіленої комп'ютерної системи для обробки IoT-даних в реальному часі вимагає вибору інструментів, здатних обробляти безперервні набори подій із мінімальною затримкою. Для оптимізації обчислювальних ресурсів на одному вузлі було розроблено структурну схему порівняння повномасштабного та реалізованого технологічних стеків в умовах апаратних обмежень, яка демонструє зменшення надлишкових компонентів. На сучасному ринку технологій Big Data існує кілька домінуючих екосистем для роботи з потоковими даними. Розглянемо та порівняємо найбільш актуальні рішення, що могли б стати альтернативою обраному стеку [19].

Сервіси збору даних та черг повідомлень (streaming ingestion) виконують завдання тимчасового збереження потоку подій для запобігання втраті даних при пікових навантаженнях. У ролі альтернатив Apache Kafka досліджено

RabbitMQ, який є традиційним брокером з акцентом на складну маршрутизацію, однак він видаляє повідомлення відразу після підтвердження отримання, що робить неможливу повторну обробку даних. Сервіс Apache Pulsar, що пропонує багаторівневе зберігання, але є складнішим у розгортанні через залежність від додаткових компонентів, таких як BookKeeper, ZooKeeper. Також проаналізовано хмарний сервіс AWS Kinesis, який забезпечує високу масштабованість, але має жорсткі ліміти на обсяг даних та високу вартість експлуатації [4].

Обчислювальні сервіси обробки потоків (stream processing) передбачають трансформацію та агрегацію даних «на льоту» перед фінальним збереженням. Як альтернативу Apache Spark розглянуто Apache Flink, що вважається лідером у сфері обробки кожної події окремо, проте Apache Spark забезпечує перевагу у вигляді уніфікованого API для потокової та пакетної аналітики. Бібліотека Apache Kafka Streams пропонує зручну інтеграцію для простих трансформацій, але не має потужних аналітичних функцій та оптимізатора запитів. Також було досліджено Google Dataflow як потужне хмарне рішення, що повністю автоматизує ресурси, проте створює критичну залежність від інфраструктури конкретного провайдера [23].

Об’єктне сховище та табличні формати забезпечує довготривале зберігання терабайтів інформації з підтримкою транзакційності (ACID). На противагу обраним MinIO та Apache Iceberg, класичне рішення HDFS вимагає значних апаратних ресурсів та складного адміністрування. Формат Delta Lake виступає прямим конкурентом Iceberg, проте останній забезпечує кращу ізоляцію рівнів зберігання та більш гнучку еволюцію схем даних без ручної зміни таблиць. Також розглянуто Amazon S3, використання якого в «on-premise» рішенні перетворює архітектуру на гібридну, що робить MinIO єдиною повноцінною альтернативою для локального розгортання.

Платформи аналітики та візуалізації (business intelligence) представляють результати аналізу кінцевому користувачу через інтерактивні

інтерфейси. В порівнянні з Apache Superset система Grafana ідеально підходить для технічного моніторингу метрик інфраструктури в реальному часі, проте вона поступається у можливостях побудови складних бізнес-звітів та реляційного аналізу на основі SQL-запитів [26]. Платформа Metabase пропонує максимально простий інтерфейс для швидкого проектування графіків, але має суттєво обмежену кастомізацію складних віджетів при роботі з великими масивами інформації. Система з відкритим кодом Redash демонструє високу ефективність під час написання прямих SQL-запитів до різних джерел, проте її функціонал для створення багатосторінкових інтерактивних панелей моніторингу є недостатньо гнучким. Популярний інструмент аналізу даних Looker Studio надає зручні хмарні інструменти для візуалізації маркетингових звітів, але повністю залежить від зовнішніх сервісів і не дозволяє розгортати інфраструктуру локально. Професійні комерційні рішення, такі як Tableau або Power BI, забезпечують найвищий рівень аналітичного моделювання, проте вони вимагають високих витрат на ліцензування та комерційну підтримку.

Таким чином, за критеріями нульової вартості ліцензування, автономності від хмарних сервісів, наявності потужного вбудованого SQL-редактора та можливості інтеграції в контейнеризованому середовищі, для реалізації розподіленої комп’ютерної системи було обрано Apache Superset.

РОЗДІЛ 2 ПРОЄКТНА ЧАСТИНА

2.1 Загальна архітектура розподіленої комп’ютерної системи обробки IoT-даних

Структурна схема мультимедійного домашнього сервервера наведена на рис. 2.1. Перейдемо до проєктної частини і побудуємо загальну архітектуру розподіленої комп’ютерної системи обробки IoT-даних, яка зображена на структурній схемі компонентів розподіленої системи обробки даних. Оскільки розгортаємо систему як локальне «on-premise» рішення на із використанням контейнеризації [18], архітектура розділена на наступні логічні етапиі:

– етап збору даних (data ingestion);

– етап обробки та трансформації даних (stream processing);

– етап збереження та транзакційного керування даними (storage layer);

– етап надання даних (data serving).

Перший етап відповідає за збір даних, використовуючи архітектуру «publisher/subscriber». Центральним компонентом цього рівня виступає кластер Apache Kafka, розгорнутий у Docker-середовищі, який забезпечує надійне проміжне накопичення повідомлень у відповідних топіках (topics). Такий підхід дозволяє ефективно ізолювати джерела даних від обчислювальних ресурсів, гарантуючи, що жодна подія не буде втрачена навіть при критичних навантаженнях на систему. Завдяки механізмам координації Kraft, Apache Kafka забезпечує високу доступність та чітку впорядкованість вхідних даних, виконуючи роль стабільної шини повідомлень для системи [20].

Другий етап відповідає за обробку даних і базується на технології розподілених обчислень Apache Spark. На цьому рівні функціонує Spark Master, який координує роботу Spark Workers для забезпечення паралельної обробки інформації [23].

Використовуючи модуль Spark Structured Streaming, система зчитує мікро-пакети даних з топіків Apace Kafka, виконує їх очищення, перевірку та необхідні агрегації в реальному часі. Такий підхід дозволяє перетворювати необроблені IoT-дані у структурований формат безпосередньо в процесі руху подій по розподіленій комп'ютерній системі.

Третій етап реалізує стратегію довготривалого зберігання та структурування інформації згідно з сучасним підходом Data Lakehouse [27]. Цей рівень забезпечує перехід від простого накопичення неструктурованих файлів до створення продуктивного аналітичного сховища. Основними компонентами цього етапу є:

– Сервіс MinIO виступає високопродуктивним S3-сумісним об'єктним сховищем. Він забезпечує надійне фізичне зберігання об'єктів у локальній інфраструктурі, пропонуючи високу пропускну здатність та горизонтальну масштабованість, характерну для хмарних систем [22].

– Apache Iceberg відкритий табличний формат високого рівня, що інтегрується безпосередньо над рівнем MinIO. Він наділяє систему властивостями повноцінної аналітичної бази даних, впроваджуючи підтримку ACID-транзакцій, автоматичну еволюцію схем (schema evolution) та партиціювання (partitioning). Це дозволяє усунути типові проблеми «болота даних» (data swamp), забезпечуючи цілісність IoT-даних та високу швидкість виконання запитів.

Впровадження цього шару дозволяє гармонійно поєднати економічну ефективність та гнучкість об'єктного зберігання з надійністю та структурованістю системи. Такий підхід є критично важливим для коректної

обробки великих масивів історичних подій та проведення глибокого аналізу накопичених даних [13].

Четвертий етап архітектури відповідає за фінальну інтерпретацію даних та надання кінцевому користувачу інструментів для аналітики.

Цей рівень трансформує структуровані масиви інформації в аналітичні дані за допомогою таких сервісів:

– Apache Spark SQL Thrift Server виступає в ролі високоефективного аналітичного шлюзу, що реалізує інтерфейс доступу до таблиць Apache Iceberg через протокол JDBC. Це дозволяє виконувати складні аналітичні запити безпосередньо над обчислювальними потужностями Spark-кластера, забезпечуючи мінімальну затримку (latency) при обробці великих наборів даних [25].

– Apache Superset сучасна платформа бізнес-аналітики корпоративного класу, що є фінальною ланкою архітектури. Вона забезпечує візуалізацію оброблених метрик у вигляді інтерактивних дашбордів, теплових карт та динамічних графіків.

Завдяки такій логічній структурі, користувач отримує можливість не лише в реальному часі здійснювати моніторинг поточного стану IoT-інфраструктури, а й проводити глибокий історичний аналіз динаміки даних через інтуїтивно зрозумілий веб-інтерфейс. Це забезпечує повний цикл роботи з даними, а саме від моменту їх виникнення до перетворення на аналітичну звітність.

2.2 Обгрунтування вибору інструментів для побудови розподіленої комп’ютерної системи обробки IoT-даних

Ефективність розгортання та подальшого масштабування аналітичної платформи безпосередньо залежить від коректного підбору технологічного стека, що має відповідати специфіці IoT-даних. При формуванні

архітектурного рішення в межах даної кваліфікаційної роботи ключовими критеріями вибору стали не лише функціональні можливості інструментів, а й їхня здатність до легкої інтеграції та висока продуктивність в умовах реального часу [19].

Вибір оптимальних технологій дозволяє забезпечити відмовостійке керування кластером та гнучкість обробки великих масивів інформації. Зокрема, зроблено акцент на використанні відкритого програмного забезпечення, що гарантує відсутність прив’язки до конкретних хмарних провайдерів та дозволяє розгорнути систему в межах локальної «on-premise» інфраструктури.

В наступних підрозділах розглянемо функціональні особливості обраних сервісів, їхню роль і специфіку налаштування для забезпечення цілісності та високої швидкості аналізу IoT-даних.

2.2.1 Принцип роботи MQTT bridge для отримання IoT-даних

Для забезпечення стабільного потоку даних у розподілену комп’ютерну систему реалізується спеціалізований сервіс MQTT bridge. В основі роботи цього компонента лежить мережевий протокол MQTT, який використовується для передачі даних від пристроїв з обмеженими ресурсами. Взаємодія цього протоколу базується на моделі «publisher/subscriber», де всі повідомлення проходять через центрального брокера. Основна задача сервісу полягає в тому, щоб виступати активним підписником, який підключається до брокера платформи ThingSpeak за допомогою унікальних ідентифікаторів та ключів верифікації [17].

Після встановлення з’єднання сервіс ініціює підписку на конкретні топіки, які є ієрархічними шляхами для розподілу повідомлень від сенсорів. Коли зовнішній пристрій публікує нові IoT-дані в обраний топік, брокер миттєво пересилає ці дані нашому сервісу. Такий підхід дозволяє збирати дані у реальному часі та отримувати вхідні пакети в структурі, яка є зручною для

подальшого збереження та використання в нашій розподіленій комп'ютерній системі. Також використання протоколу дозволяє гарантувати доставку повідомлень навіть при нестабільному мережевому з'єднанні завдяки вбудованим механізмам підтвердження передачі [10]. Сервіс MQTT bridge автоматично підтримує зв'язок із сервером та очікує на появу нових подій у відповідних каналах.

Такий підхід забезпечує довгострокову роботу сервісу, який безперервно зчитує вхідні потоки IoT-даних та готує їх для подальшого використання. В результаті отримаємо надійний та автоматизований механізм збору даних, який працює у фоновому режимі та не потребує ручного втручання.

2.2.2 Принцип роботи брокера повідомлень Apache Kafka

Принципи функціонування сервісу Apache Kafka, зображено на рисунку 2.1, який виконує роль високонавантаженої розподіленої шини повідомлень, базується на кластерній архітектурі відповідно до структурної схеми компонентів розподіленої системи обробки даних.

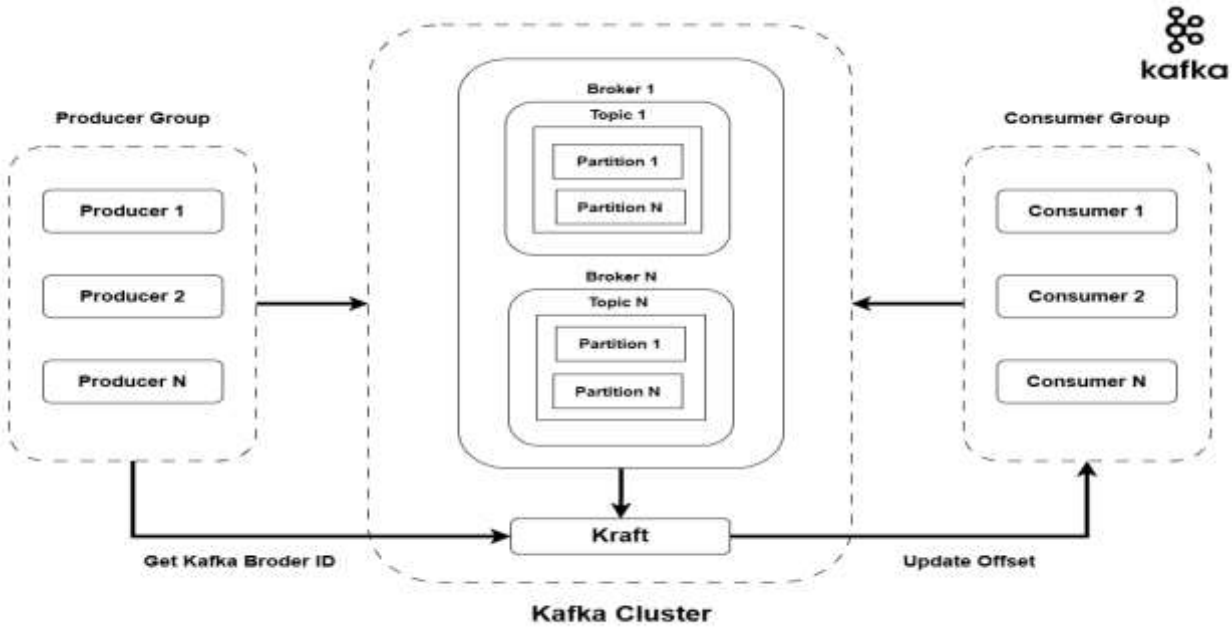

Рисунок 2.1 – Архітектура сервісу Apache Kafka

На відміну від класичних систем черг, Apache Kafka працює за принципом розподіленого логу подій (distributed commit log), що дозволяє не лише здійснювати транспортування даних, а й забезпечувати їх надійне зберігання протягом встановленого інтервалу часу. Логічна взаємодія компонентів розподіленої комп'ютерної системи базується на моделі Producer Group та Consumer Group. Такий підхід дозволяє джерелам даних функціонувати незалежно від споживачів, що забезпечує повне архітектурне роз'єднання етапу збору IoT-подій та процесу їх подальшого оброблення [2].

Внутрішня організація потоків у центральній частині схеми демонструє розподіл даних між вузлами за відповідними топіками. Кожна така логічна одиниця розділяється на декілька фізичних партицій, що виступає ключовим фактором горизонтального масштабування. Саме через партиціювання досягається ефективний розподіл обчислювального навантаження та реалізується можливість високошвидкісного паралельного зчитування IoT-даних [2].

Процес оперування даними пов'язаний із механізмом Update Offset, який дозволяє групі споживачів чітко фіксувати прогрес обробки кожного повідомлення. Завдяки присвоєнню унікальних зміщень (offset), система отримує можливість безперешкодно відновлювати обчислювальний цикл із точки останньої фіксації у разі технічних збоїв, що гарантує цілісність передачі за стандартом «at-least-once».

Ефективність споживання даних забезпечується через функціонування Consumer Group, де партиції топіків автоматично закріплюються за доступними консюмерами. Така логіка не лише підвищує загальну пропускну здатність, а й активує механізм динамічного балансування навантаження в разі зміни кількості активних споживачів у кластері.

Стабільність усієї архітектури підтримується протоколом KRaft, який інтегрує вузли у єдину керовану екосистему. Реалізація алгоритму розподілу навантаження безпосередньо всередині Apache Kafka дозволяє координувати

запити на брокерів та оновлення метаданих без залучення зовнішніх сервісів. Це мінімізує системні затримки та значно спрощує адміністрування обчислювальних потужностей у ізольованому Docker-середовищі.

2.2.3 Принцип роботи кластера розподілених обчислень Apache Spark

Функціонування розподіленої комп'ютерної системи на етапі трансформації даних базується на кластерній архітектурі розподіленого обчислення за допомогою Apache Spark. В межах даної архітектури Apache Spark виступає сервісом для трансформації IoT-даних між шиною повідомлень Apache Kafka та об'єктним сховищем MinIO, реалізуючи спосіб обробки даних у оперативній пам'яті (in-memory processing). Це дозволяє мінімізувати дискові операції введення-виведення та забезпечувати високу швидкість агрегації IoT-даних [23].

Архітектурна модель (рис. 2.2) кластера базується на взаємодії вузлів Master та Worker, де центральний координатор Cluster Manager, який відповідає за розподіл ресурсів та керування життєвим циклом завдань.

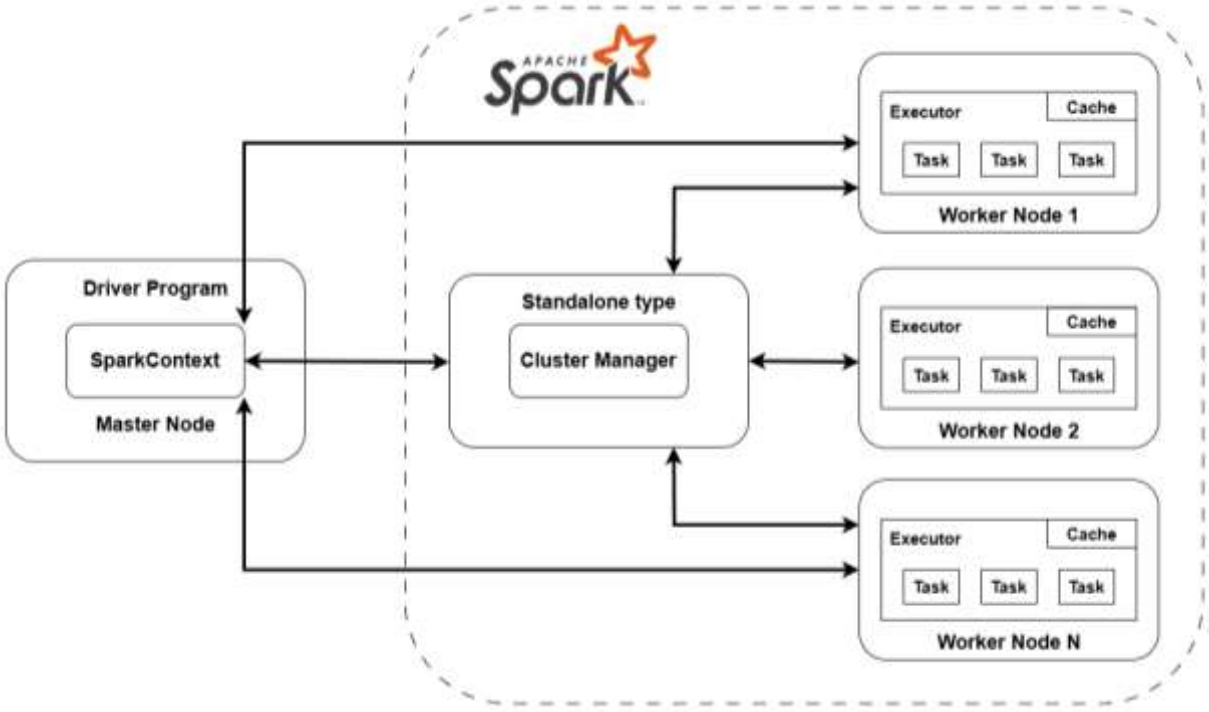

Рисунок 2.2 – Архітектура кластера Apache Spark разом із Cluster Manager

Робочі вузли виконують безпосередню обробку даних у межах ізольованого виконавця (executor). Така ієрархія дозволяє системі динамічно масштабуватися, адаптуючись до інтенсивності вхідного потоку від IoT-пристроїв без втрати продуктивності [25].

Особлива увага приділяється модулю Spark Structured Streaming, який трансформує процес зчитування з Apache Kafka в роботу з «нескінченною таблицею» (unbounded table). Дані обробляються за допомогою механізму мікро-пакетів (micro-batches), що забезпечує баланс між низькою затримкою та високою пропускною здатністю. Використання високорівневого API DataFrames на цьому етапі дозволяє впроваджувати сувору схему типізації для вхідних JSON-повідомлень, перетворюючи початкові дані на структуровані події [25].

Процес обробки IoT-даних в середині Apache Spark оптимізується шляхом побудови орієнтованого ациклічного графа (DAG). Перед фізичним виконанням операцій система будує логічний план трансформацій, що мінімізує переміщення даних між вузлами кластера (shuffling). Це критично важливо для забезпечення цілісності при виконанні віконних функцій або агрегації показників за певні часові інтервали.

2.2.4 Принцип організації і структурування даних у об’єктному сховищі MinIO та форматі Iceberg

Функціонування рівня довготривалого зберігання в межах розробленої розподіленої комп’ютерної системи базується на поєднанні об’єктного сховища MinIO та табличного формату Apache Iceberg. Поєднання таких компонентів дозволяє реалізувати сучасну концепцію Data Lakehouse, яка інтегрує високу масштабованість об’єктних систем із надійністю та аналітичними можливостями класичних реляційних баз даних.

Розглянемо роль MinIO як базового інфраструктурного елемента для розміщення інформації. Виступаючи високопродуктивним S3-сумісним

сховищем, даний сервіс забезпечує фізичне збереження об’єктів у логічно структурованих контейнерах або бакетах (buckets). Використання такої архітектури (рис. 2.3) надає можливість ефективно оперувати великими масивами неструктурованих даних і забезпечувати горизонтальне розширення системи без втрати продуктивності.

Для гарантування безпеки та розмежування прав доступу впроваджується політика IAM-ролей, що дозволяє чітко контролювати взаємодію між Apache Spark виконавцями (executors) та фізичним сховищем на рівні окремих ідентифікаторів доступу [22].

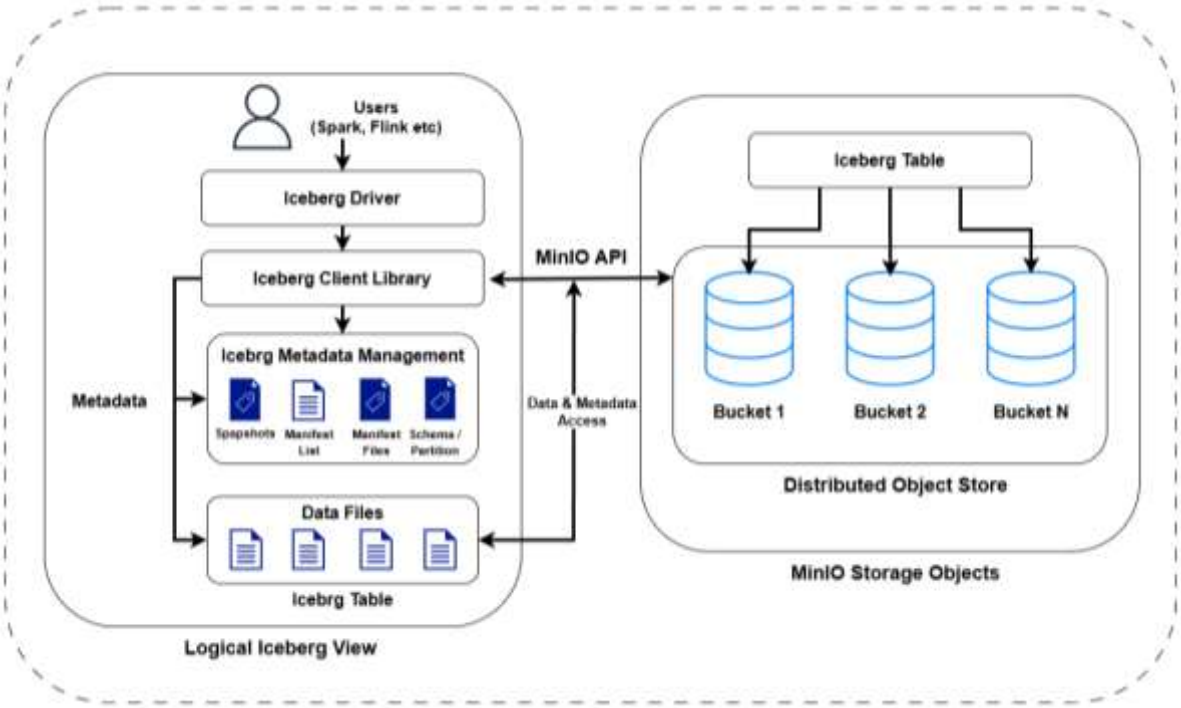

Рисунок 2.3 – Архітектура сховища MinIO разом із Apache Iceberg

Логічна організація даних реалізується через архітектуру відкритого табличного формату Apache Iceberg (рис. 2.3), що функціонує безпосередньо над рівнем зберігання MinIO. Впровадження цього формату наділяє систему властивостями повноцінної аналітичної бази даних завдяки підтримці транзакційних можливостей за стандартом ACID. Це гарантує атомарність операцій запису зі сторони Apache Spark та унеможливлює появу некоректних

або частково збережених станів даних. Важливою особливістю обраного підходу є реалізація механізму запису (append), що дозволяє безперервно додавати нові IoT-дані в кінець існуючих таблиць без ризику пошкодження структури файлів [27].

Окрему увагу варто приділити механізму еволюції схем, який дозволяє гнучко змінювати структуру таблиць без необхідності повного перезапису історичної інформації. Також Apache Iceberg автоматично оптимізує розподіл даних по фізичних директоріях усередині бакетів. Такий підхід дозволяє уникнути хаотичного накопичення файлів та значно пришвидшує виконання аналітичних запитів завдяки відкиданню не відповідних розділів під час пошуку. Масштабування системи при цьому відбувається лінійно, оскільки архітектура Iceberg мінімізує навантаження на метадані при збільшенні обсягу накопичених об’єктів у MinIO [6].

Процес керування метаданими виступає ключовою перевагою обраного формату, оскільки система оперує багаторівневими шарами замість прямого сканування файлової структури. Це забезпечує стабільно високу швидкість зчитування для інструментів візуалізації навіть при накопиченні значних обсягів історичної IoT-даних.

В результаті розгортання такої структури дозволяє отримати автономне аналітичне сховище корпоративного рівня, яке повністю відповідає вимогам щодо надійності та швидкодії в ізольованому середовищі.

2.2.5 Принцип роботи шлюзу Apache Spark SQl Thrift Server

Інтеграція оброблених даних із зовнішніми інструментами візуалізації в межах розробленої розподіленої комп’ютерної системи, забезпечується за допомогою розгортання та налаштування Apache Spark Thrift Server. Даний компонент виступає в ролі високоефективного аналітичного шлюзу, який надає можливість стороннім сервісам взаємодіяти з таблицями Apache Iceberg за допомогою стандартних SQL-запитів через протоколи JDBC або ODBC [26].

Використання такої архітектури (рис. 2.4) дозволяє повністю ізолювати від об’єктне сховище MinIO та представити IoT-дані в виглядів таблиць.

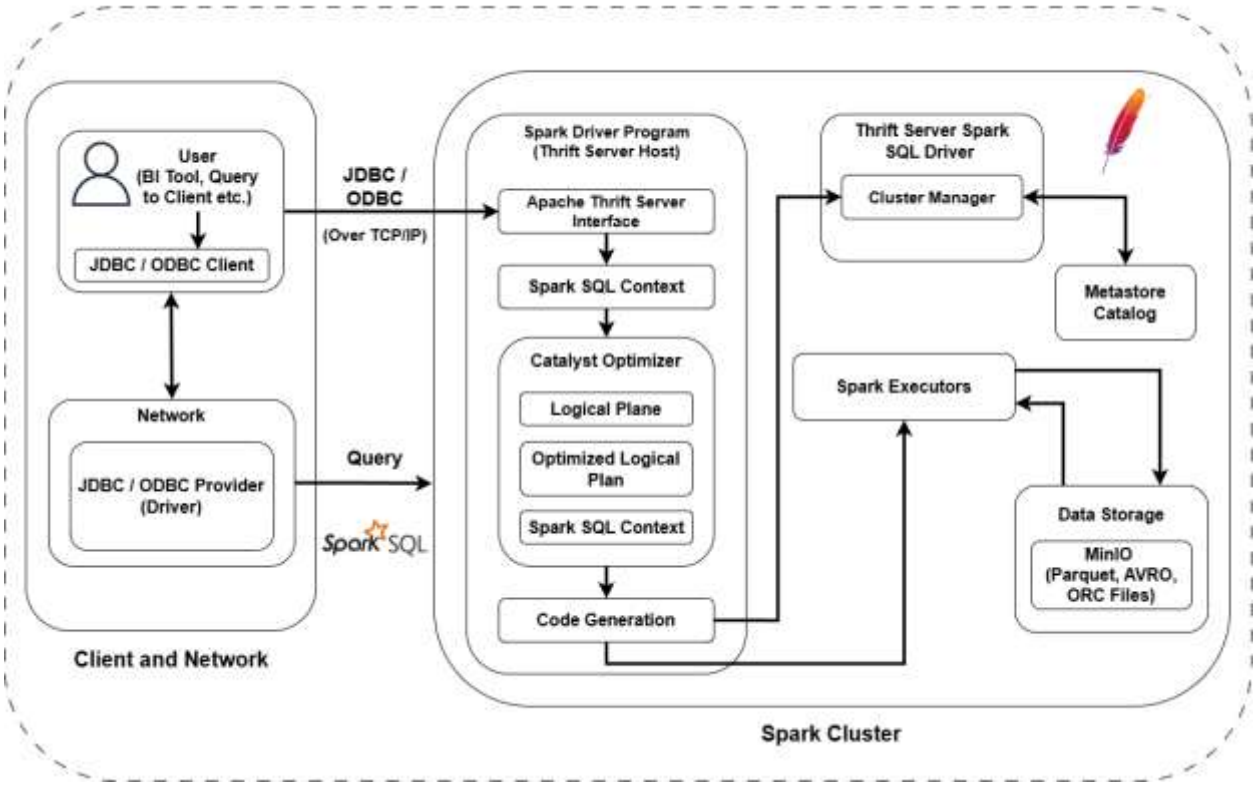

Рисунок 2.4 – Архітектура Apache Spark Thrift Server

Функціонування шлюзу базується на архітектурі тривалого сеансу, де Spark-додаток працює у фоновому режимі як багатокористувацький сервер. Це забезпечує значну перевагу у швидкодії, оскільки ресурси кластера виділяються заздалегідь, і кожен новий запит від системи візуалізації не потребує повного перезапуску Spark-сесії. Такий підхід мінімізує затримки при побудові динамічних дашбордів, дозволяючи обробляти складні аналітичні запити в режимі реального часу.

Процес обробки запитів через Thrift Server інтегрує всі раніше описані переваги обчислювального рушія Spark. Коли система візуалізації, наприклад, Apache Superset, надсилає SQL-запит, шлюз передає його на виконання оптимізатору, який будує план виконання з урахуванням метаданих Apache Iceberg [27].

Окрему увагу варто приділити ролі Apache Thrift Server в забезпеченні безпеки та розмежуванні доступу. Виступаючи єдиною точкою входу до аналітичного шару, сервіс дозволяє централізовано керувати підключеннями та контролювати навантаження на обчислювальні вузли.

В результаті впровадження даного шлюзу дозволяє розподіленій комп'ютерній системі зчитувати оброблені IoT-дані для використання в аналітичних сервісах.

2.2.6 Принцип роботи системи візуалізації Apache Superset

Візуальна інтерпретація та аналіз оброблених IoT-даних у межах розробленої розподіленої комп'ютерної системи реалізується за допомогою сервісу Apache Superset, загальна схема функціонування якого представлена на рисунку 2.5.

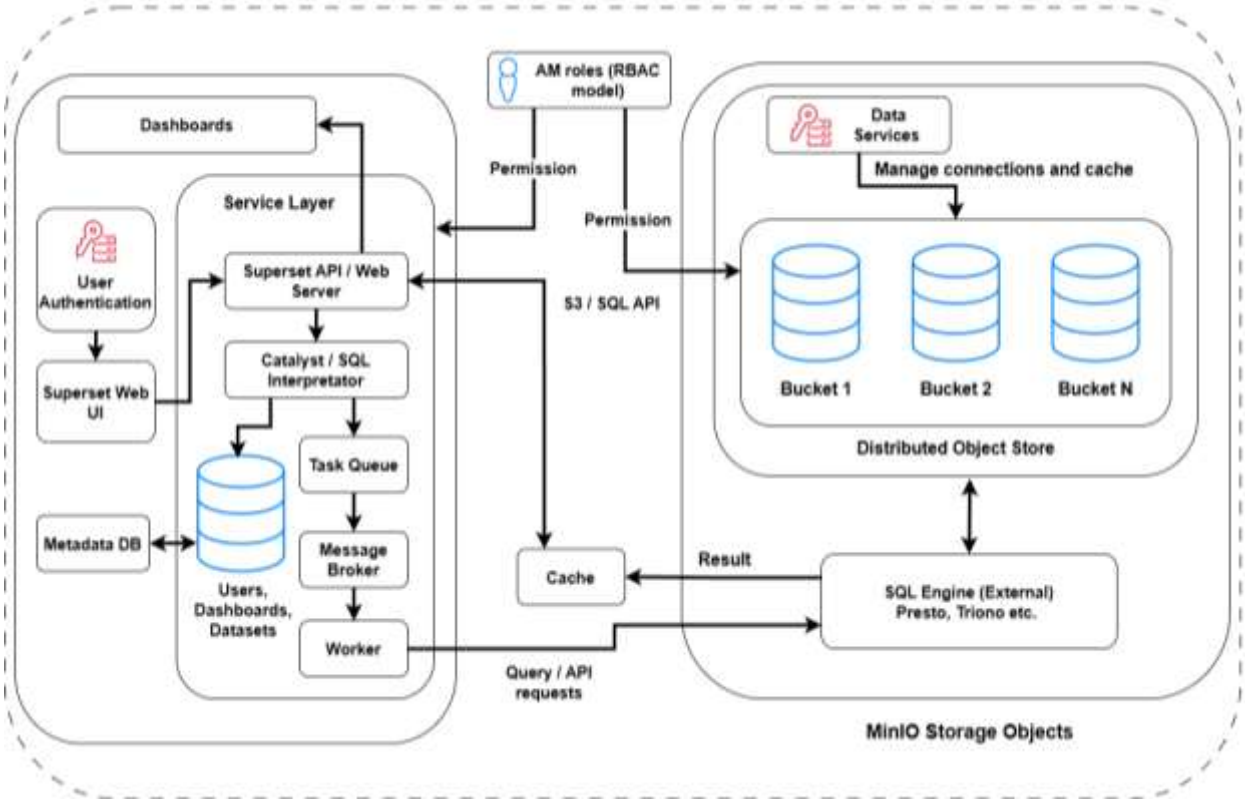

Рисунок 2.5 – Архітектура Apache Superset

Ця сучасна платформа бізнес-аналітики (BI) корпоративного класу, яка виступає фінальним інтерфейсом взаємодії користувача з IoT-даними, надаючи інструменти для побудови складних інтерактивних дашбордів. Використання Apache Superset дозволяє інтегрувати різні аналітичні можливості в єдине візуальне середовище, забезпечуючи моніторинг стану IoT-інфраструктури в режимі реального часу.

Функціонування системи візуалізації базується на безпосередньому підключенні до Spark Thrift Server через протокол SQLAlchemy, що перетворює платформу на потужний SQL-інтерфейс до об'єктного сховища. Завдяки такій інтеграції, Apache Superset не потребує власного сховища для масивів IoT-даних, а виконує запити безпосередньо над таблицями Apache Iceberg [27]. Такий підхід гарантує актуальність відображуваної інформації, оскільки будь-які зміни в бакетах MinIO миттєво стають доступними для візуалізації після завершення обробки у Apache Spark кластері.

Процес побудови аналітичних звітів реалізується через розширений інструментарій Explore, який дозволяє користувачам без написання складного коду проводити глибокий історичний аналіз. Система підтримує широкий спектр візуальних елементів, а саме від лінійних графіків реального часу до складних теплових карт розподілу навантаження на датчики. Важливою особливістю є можливість кешування результатів запитів на рівні BI-платформи, що суттєво знижує навантаження на обчислювальний кластер при повторних переглядах популярних дашбордів.

Окрему увагу варто приділити ролі Apache Superset у забезпеченні безпечного доступу до аналітичної звітності через веб-інтерфейс. Впровадження механізмів контролю доступу на основі ролей (RBAC) дозволяє чітко розмежовувати права користувачів на перегляд конкретних метрик або керування налаштуваннями системи. Механізм безпеки функціонує через гнучку систему інтеграції з протоколами автентифікації, що дозволяє використовувати як внутрішню базу користувачів, так і зовнішні сервіси.

Після успішної авторизації система порівнює ідентифікатор користувача з набором доступних йому ролей, автоматично фільтруючи контент у візуальному інтерфейсі. Додатковим рівнем захисту виступає можливість налаштування фільтрації на рівні рядків (row level security), яка гарантує, що різні групи користувачів бачитимуть лише ті сегменти IoT-даних, які відповідають їхнім правилам [9].

Розглядаючи переваги такої системи безпеки, варто виділити високу гнучкість налаштувань та можливість централізованого керування великою кількістю облікових записів. Використання перевірених стандартів шифрування та підтримка протоколів безпечного з'єднання забезпечують захист аналітичних даних від несанкціонованого перехоплення під час передачі через веб-браузер. Таклож автоматичне логування дій користувачів дозволяє проводити аудит безпеки та відстежувати історію звернень до критично важливої інформації.

Водночас складність налаштування RBAC можна віднести до певних недоліків, оскільки помилки у конфігурації великої кількості взаємопов'язаних дозволів можуть призвести до ненавмисного обмеження доступу або виникнення вразливостей. Також варто враховувати, що розширені механізми безпеки створюють додаткове навантаження на сервер метаданих Apache Superset, що вимагає ретельного планування ресурсів при масштабуванні системи на велику кількість одночасних користувачів.

2.3 Огляд інструментів розгортання сервісів розподіленої комп'ютерної системи обробки IoT-даних

Ефективне функціонування розподіленої комп'ютерної системи великих даних вимагає надійної ізоляції компонентів і гнучкого керування мережевою взаємодією між ними. Процес розгортання розробленої архітектури базується на використанні сучасних технологій контейнеризації, що дозволяє об'єднати

Apache Kafka, Apache Spark, MinIO та Apache Superset в єдину логічну екосистему [18]. Це забезпечує ідентичність середовищ розробки та виконання, гарантуючи стабільність обробки IoT-даних незалежно від конфігурації хоста.

Для реалізації локальної інфраструктури в межах даної кваліфікаційної роботи обрано технологію Docker Compose, яка надає інструменти для декларативного опису та керування ресурсами кластера. З рештою при переході до промислової експлуатації підтримується можливість використання платформи Kubernetes (K8s), яка пропонує розширені механізми автоматичного масштабування та самовідновлення сервісів.

Окрім основних рішень, існують альтернативні підходи для оркестрації, такі як HashiCorp Nomad, який орієнтований на гібридні середовища, та Docker Swarm, що пропонує простішу модель масштабування на декілька вузлів. Попри різноманітність інструментів, обраний стек дозволяє зосередити основні ресурси на логіці обробки даних, забезпечуючи при цьому достатній рівень відмовостійкості та ізоляції сервісів у межах єдиного обчислювального вузла.

2.3.1 Платформа розгортання інфраструктури Docker Compose

Реалізація архітектури відбувається за допомогою використання Docker Compose, який базується на декларативному підході до керування сервісами, де вся конфігурація кластера описується в єдиному файлі формату YAML. Це дозволяє визначити взаємозв’язки між компонентами, параметри мережі та обсяги сховищ як код (Infrastructure as Code) [18], забезпечуючи ідентичність середовища розгортання на будь-якому обчислювальному хості. В межах даної роботи Docker Compose виконує роль локального оркестратора, що координує життєвий цикл контейнерів Apache Kafka, Apache Spark, об’єктного сховища MinIO, Apache Spark Thrift Server та BI системи Apache Superset.

Однією з ключових переваг обраної платформи є можливість прямого керування ресурсами через параметри «mem_limit» та «cpus». У контексті великих даних це дозволяє оптимізовувати використання оперативної пам’яті та ядер процесора для кожного сервісу окремо. Такий поділ гарантує, що інтенсивні аналітичні обчислення у Apache Spark Streaming не спричинять дефіциту ресурсів для брокерів Apache Kafka, підтримуючи стабільність черг повідомлень та запобігаючи збоям через критичне перевантаження системи.

Важливим аспектом функціонування інфраструктури є використання ізольованих мережевих мостів (bridge networks). Docker Compose автоматично створює внутрішню мережу, де сервіси взаємодіють між собою за іменами сервісів, що значно спрощує конфігурацію підключень (наприклад, Apache Spark звертається до Apache Kafka за іменем сервісу, а не за статичною IP-адресою). Це забезпечує додатковий рівень безпеки, оскільки доступ до критичних портів брокерів або баз даних залишається закритим для зовнішніх мереж, якщо вони не вказані явно на хост-машину [9].

Для забезпечення цілісності накопиченої інформації використаємо механізм Volumes, який відокремлює обчислювальну логіку від фізичного рівня зберігання. Монтування локальних директорій сервера безпосередньо у внутрішні структури контейнерів дозволяє зберігати логи Apache Kafka, метадані Apache Iceberg та об’єкти MinIO навіть після повної зупинки або оновлення сервісів [20]. Така організація перетворює контейнери на тимчасові виконавчі середовища, тоді як стан системи залишається стійким і захищеним від випадкових втрат при перезавантаженні середовища.

2.3.2 Платформа розгортання інфраструктури Kubernetes

Перехід до обробки даних у промислових масштабах вимагає використання платформи оркестрації K8s, яка автоматизує розгортання, масштабування та керування контейнеризованими додатками в межах великих кластерів [2]. На відміну від Docker Compose, який орієнтований на роботу в межах одного

фізичного вузла, Kubernetes дозволяє об’єднувати ресурси десятків або сотень серверів у єдиний обчислювальний простір. Це робить систему стійкою до відмов окремого обладнання та забезпечує безперервність розподіленої комп’ютерної системи.

Фундаментальною особливістю платформи є механізм самовідновлення (self-healing), який автоматично перезапускає контейнери у разі збою або переносить їх на інші робочі вузли кластера. У контексті великих даних це критично важливо для брокерів Apache Kafka та виконавців Apache Spark, оскільки система самостійно підтримує задану кількість активних копій без втручання адміністратора. Використання об’єктів типу StatefulSets дозволяє зберігати ідентифікатори та стан дискових масивів для таких сервісів, як MinIO, гарантуючи цілісність даних при будь-яких змінах конфігурації [22].

Процес динамічного масштабування в Kubernetes реалізується через механізм Horizontal Pod Autoscaler. Даний інструмент дозволяє автоматично збільшувати кількість обчислювальних одиниць (pods), наприклад, Apache Spark при зростанні інтенсивності вхідного потоку IoT-даних [14]. Коли навантаження знижується, система самостійно вивільняє ресурси, що забезпечує високу економічну ефективність використання хмарної або локальної інфраструктури. Така гнучкість недоступна в межах використання Docker Compose і є ключовою перевагою для систем, що працюють із нерівномірним трафіком даних.

Окрему увагу варто приділити керуванню мережевим простором через об’єкти Services та Ingress Controller. Kubernetes забезпечує внутрішнє балансування навантаження між екземплярами споживачів, автоматично розподіляючи запити на зчитування з Apache Kafka або запис у сховище MinIO. Це дозволяє створювати складні конфігурації мережі з високим рівнем безпеки, де аналітичний шар повністю ізольований від зовнішнього середовища, а доступ до дашбордів Apache Superset надається через захищені шлюзи.

Впровадження Kubernetes у промислову експлуатацію дозволяє трансформувати розроблену розподілену комп’ютерну систему в високонадійну мікросервісну архітектуру корпоративного рівня. Попри складність початкового налаштування, платформа надає необмежені можливості для горизонтального розширення системи, дозволяючи обробляти мільйони подій за секунду без зниження продуктивності або втрати цілісності інформації.

РОЗДІЛ 3 ПРАКТИЧНА ЧАСТИНА

3.1 Підготовка інфраструктурного середовища та контейнеризація сервісів

Для забезпечення роботи контейнерів у середовищі Windows використаємо Docker Desktop, який містить в собі Docker Engine, Docker CLI та Docker Compose [18].

Для початку перевіримо та за потреби активуємо апаратну віртуалізацію на рівні BIOS або UEFI материнської плати. Без використання якої підсистема WSL 2 не зможе ініціалізувати гіпервізор, що зробить неможливим запуск контейнерів. Статус віртуалізації можна перевірити в диспетчері завдань Windows в вкладці «Perfomance», де в розділі «CPU» повинно бути описано «Virtualization: Enabled» (рис. 3.1). Якщо відповідний параметр не активувати перед встановленням Docker Desktop, то система видасть критичну помилку при спробі розпочати роботу сервісу.

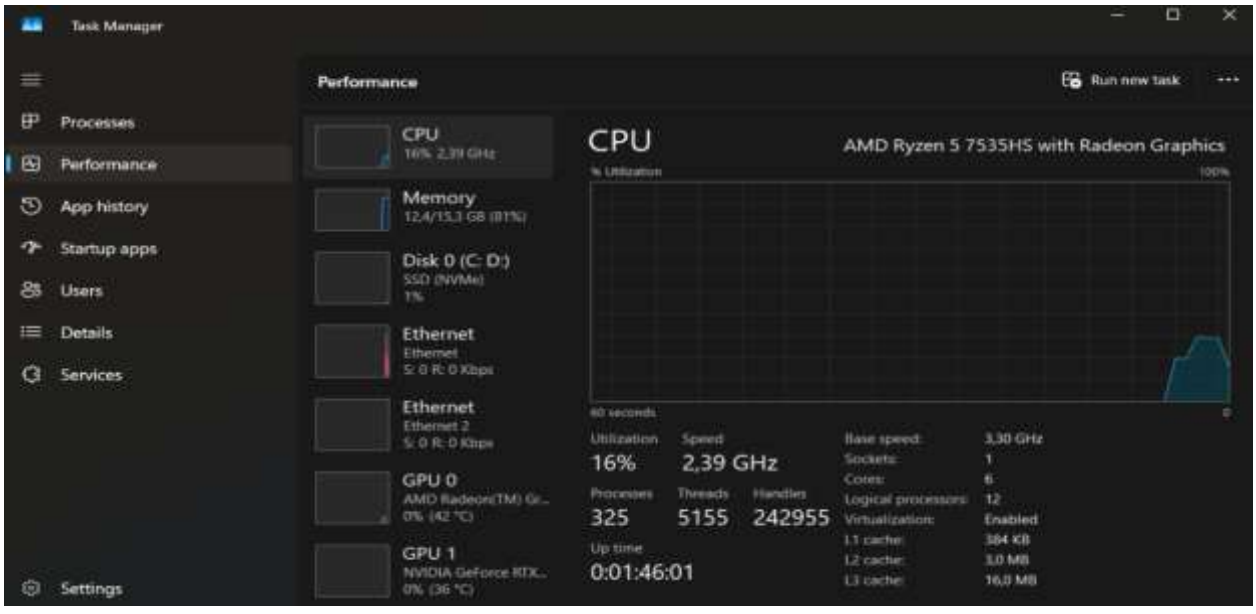

Рисунок 3.1 – Перевірка активації віртуалізації на хості

Наступним етапом встановимо на хості підсистему WSL 2 (Windows Subsystem for Linux). Це дозволяє Docker запускати Linux-контейнери безпосередньо в ядрі Linux, що працює паралельно з Windows, забезпечуючи значно вищу продуктивність та сумісність із системними викликами, необхідними для стабільної роботи Apache Kafka та Apache Spark [19]. Для цього через панель керування Windows із правами адміністратора перейдемо в панель «Windows Features» та активуємо функції, такі як «Windows Subsystem for Linux» та «Virtual Machine Platform». Підтвердження коректного налаштування середовища та відображення активної версії WSL 2 наведено на рисунку 3.2.

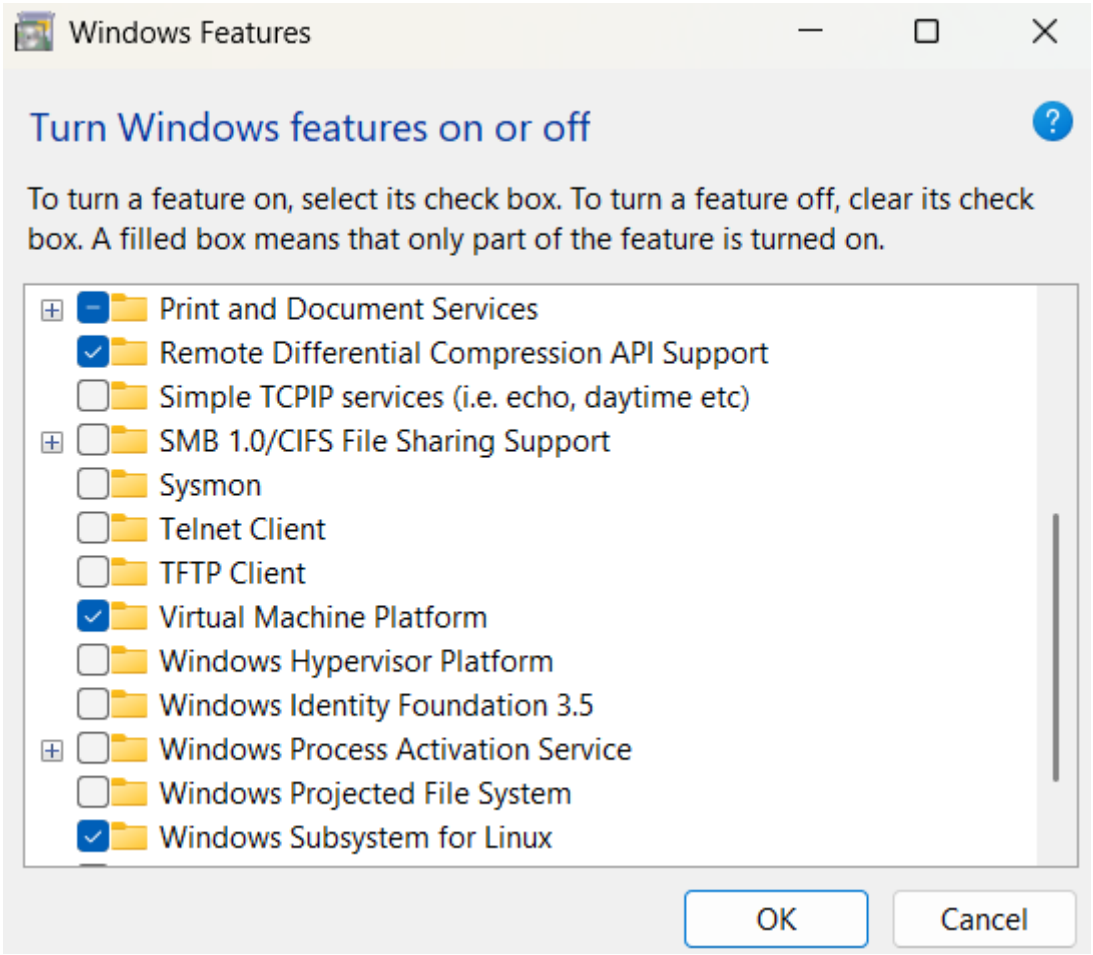

Рисунок 3.2 – Підтвердження активації компонентів віртуалізації

Після активації компонентів та перезавантаження системи необхідно встановити пакет оновлення ядра WSL 2 та налаштувати його як версію за

замовчуванням. В терміналі виконаємо команду «wsl --set-default-version 2», що гарантує використання сучасного підходу для всіх майбутніх Linux-дистрибутивів. Для перевірки налаштувань використаємо команду «wsl –list --verbose», результати якої показано на рисунку 3.3.

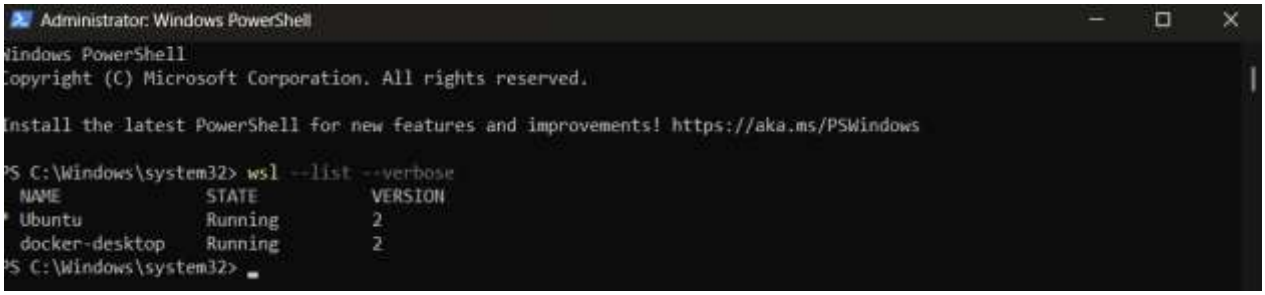

Рисунок 3.3 – Підтвердження встановлення WSL 2 як версії за замовчуванням у системному терміналі.

Для забезпечення роботи контейнерів у середовищі Windows використаємо Docker Desktop. Процес встановлення розпочнемо з завантаження актуальної версії інсталятора з офіційного сайту розробника. Після встановлення Docker Desktop в розділі «General» необхідно переконатися, що активовано опцію WSL Integration (рис. 3.4).

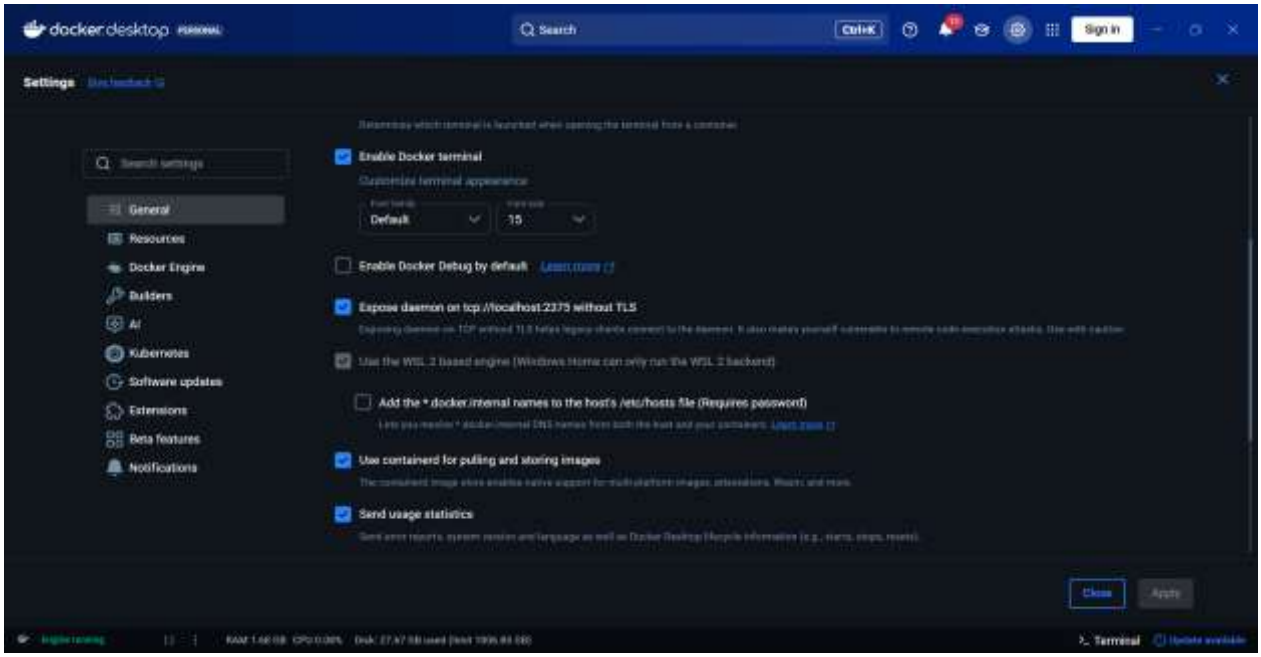

Рисунок 3.4 – Підтвердження активації WSL 2 в середовищі Docker Desktop

Це дозволяє виконувати команди Docker безпосередньо з консолі Windows (PowerShell або CMD), отримуючи повноцінний доступ до файлової системи Linux для коректної роботи монтування томів (volumes). Для перевірки успішності встановлення та коректності налаштувань використаємо команду «docker info» в терміналі. На рисунку 3.5 зображено результат команди, який підтвердужує наявність встановленого плагіна Docker Compose [18]. Належне налаштування цього середовища відіграє ключову роль, оскільки воно виступає фундаментом для розгортання всієї архітектури.

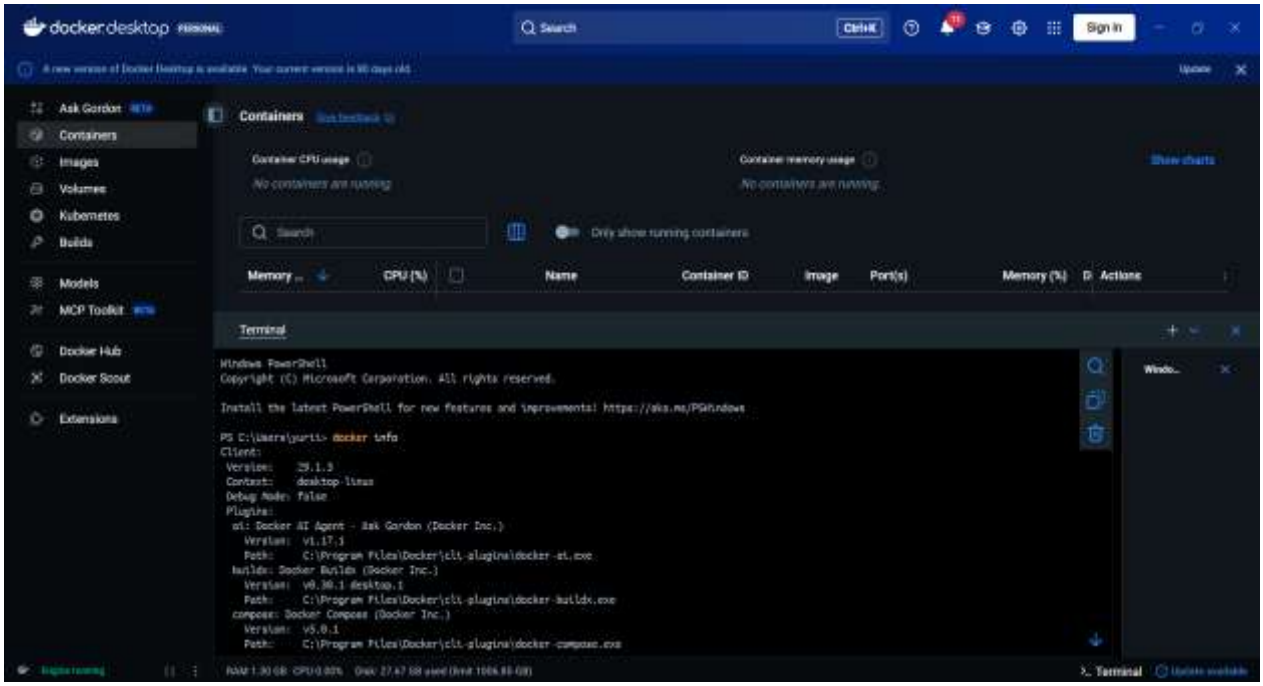

Рисунок 3.5 – Підтвердження успішного встановлення Docker Desktop

Після завершення базового налаштування системи перейдемо до етапу опису конфігурування сервісів. Практична реалізація системи базується на технології розгортання цілісного інфраструктурного середовища через Docker Compose. Такий підхід дозволяє автоматизувати запуск усіх компонентів розподіленої системи агрегації та аналізу даних з IoT-пристроїв як єдиної логічної одиниці. Повний цикл обробки даних, що охоплює етапи збору, трансформації, зберігання та візуалізації, опишемо в конфігураційному файлі «docker-compose.yml». Використання головного файлу є ключовим елементом

оркестрації, що використовує формат YAML для декларативного опису всієї інфраструктури розподіленої комп'ютернох системи. Він дозволяє в єдиному файлі визначити параметри роботи мереж, сховищ даних та взаємозв'язків між окремими сервісами, наприклад Apache Kafka брокери разом із Kafka UI (додаток Б.1), забезпечуючи їх одночасний запуск та стабільну взаємодію в ізольованому середовищі [20].

Мережеву ізоляцію та стабільність обміну даними забезпечимо за допомогою створення ізольованої внутрішньої мережі «service-network», яка використовує драйвер «bridge». Така ізольована віртуальна мережа дозволяє контейнерам здійснювати комунікацію за допомогою внутрішнього DNS-сервісу Docker. Для конфігурвання цієї мережі задама Docker-інструкції, як зображено на рисунку 3.6. Після цього отримаємо внутрішню мережу, яка буде використовуватися для взаємодії сервісів між собою.

```
...
networks:
  service-network:
    driver: bridge
```

Рисунок 3.6 – Лістинг коду оголошення внутрішньої мережі в Docker Compose

Для забезпечення цілісності накопиченої інформації та стану системи реалізуємо механізм сховищ даних (volumes). Використаємо два типи монтування. Перший іменнований тип зберігання використаємо для сервісних логів, наприклад Apache Spark Log. Другий «bind mount» тип збергіання використаємо для постінийх даних, наприклад, івентів в партиціях з Apache Kafka, які будемо фізично зберігати на диску хоста. Для цього задаємо Docker-інструкції в кожному із сервісів та в загальній конфігурації файлу «docker-compose.ylm», щоб створити внутрішнє монтування, як зображено на рисунку 3.7.

```
...
    volumes:
      - spark-master-logs:/app/logs

  volumes:
    spark-master-logs:
    spark-workers-logs:
    spark-thrift-server-logs:
...
    volumes:
      - ${PATH_TO_KAFKA_2_STORAGE}:/var/lib/kafka/data
...
```

Рисунок 3.7 – Лістинг коду оголошення монтування томів в Docker Compose

Створене таким чином інфраструктурне середовище за допомогою Docker Compose дозволяє отримати надійне середовище для розгортання сервісів [18].

3.2 Конфігурування MQTT bridge для отримання IoT-даних

Щоб забезпечити стабільний потік IoT-даних в нашій розподіленій комп'ютерній системі, налаштуємо джерела інформації в платформі ThingSpeak [17]. Цей сервіс, що базується на аналітиці MATLAB, виступає в ролі зовнішнього MQTT-брокера, який збирає дані безпосередньо з фізичних або віртуальних IoT-датчиків. Процес підготовки джерела розпочнемо з створення облікового запису та конфігурування структури даних.

На головній сторінці сервісу перейдемо до вкладки керування обліковим записом для проходження реєстрації (рис. 3.8). Для створення акаунта доцільно використовувати корпоративну пошту університету, що дозволяє отримати розширений доступ до інструментів аналізу MATLAB.

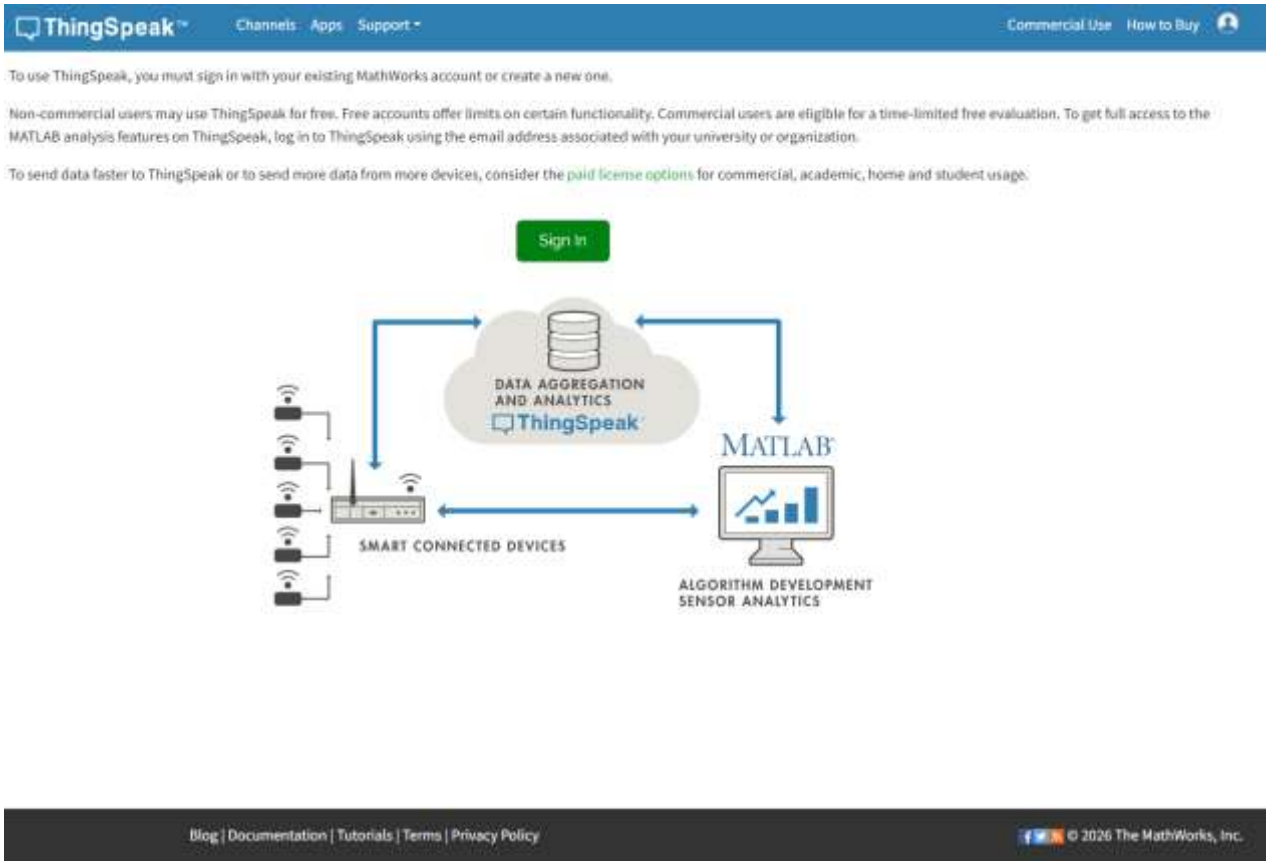

Рисунок 3.8 – Вкладки керування обліковим записом в сервісі ThingSpeak

Після успішної реєстрації та підтвердження нашої університетської пошти нам стає доступним повний функціонал платформи для створення власних каналів або підписки на вже існуючі [17].

Для організації потокового збору даних перейдемо в вкладку «Devices» та оберемо розділ «MQTT», де відображається перелік підключених пристроїв (рис. 3.9).

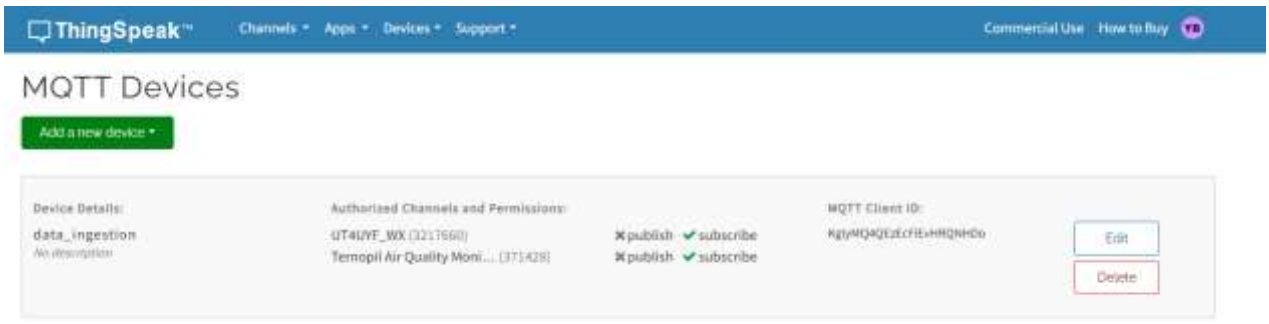

Рисунок 3.9 – Вкладки керування підпискою на зовнішні канали за допомогою MQTT протоколу

Натиснемо «Add a new device» та відкриємо вікно конфігурації нового віртуального пристрою. У цьому вікні вкажемо назву нашого клієнта, наприклад «data_ingestion», та перейдемо до авторизації каналів (рис. 3.10).

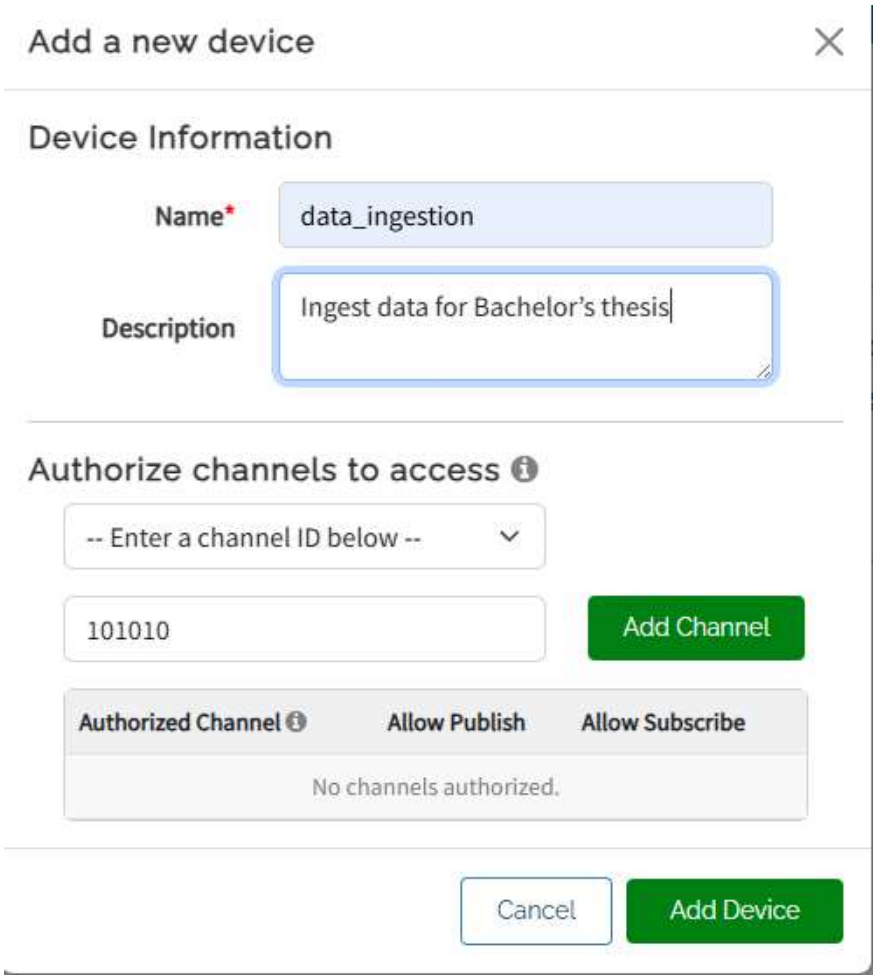

Рисунок 3.10 – Створення підписки на зовнішій канал в сервісі ThingSpeak

Нам доступна можливість обрати конкретні канали зі списку підписок або вписати наявний ID та додати їх до цього пристрою. Важливим моментом є налаштування прав доступу, де потрібно обов'язково активувати опцію «Allow Subscribe» для кожного каналу. Це дозволить нашому сервісу отримувати оновлення даних у реальному часі. Після збереження налаштувань нам стануть доступні параметри автентифікації, такі як унікальний MQTT Client ID (рис. 3.11). Отримані ключі веривікації додамо безпосередньо в програмний код для авторизації запитів до сервера ThingSpeak. Такий підхід

дозволить нам безпечно та автоматично отримувати всі необхідні IoT-дані в режимі реального часу.

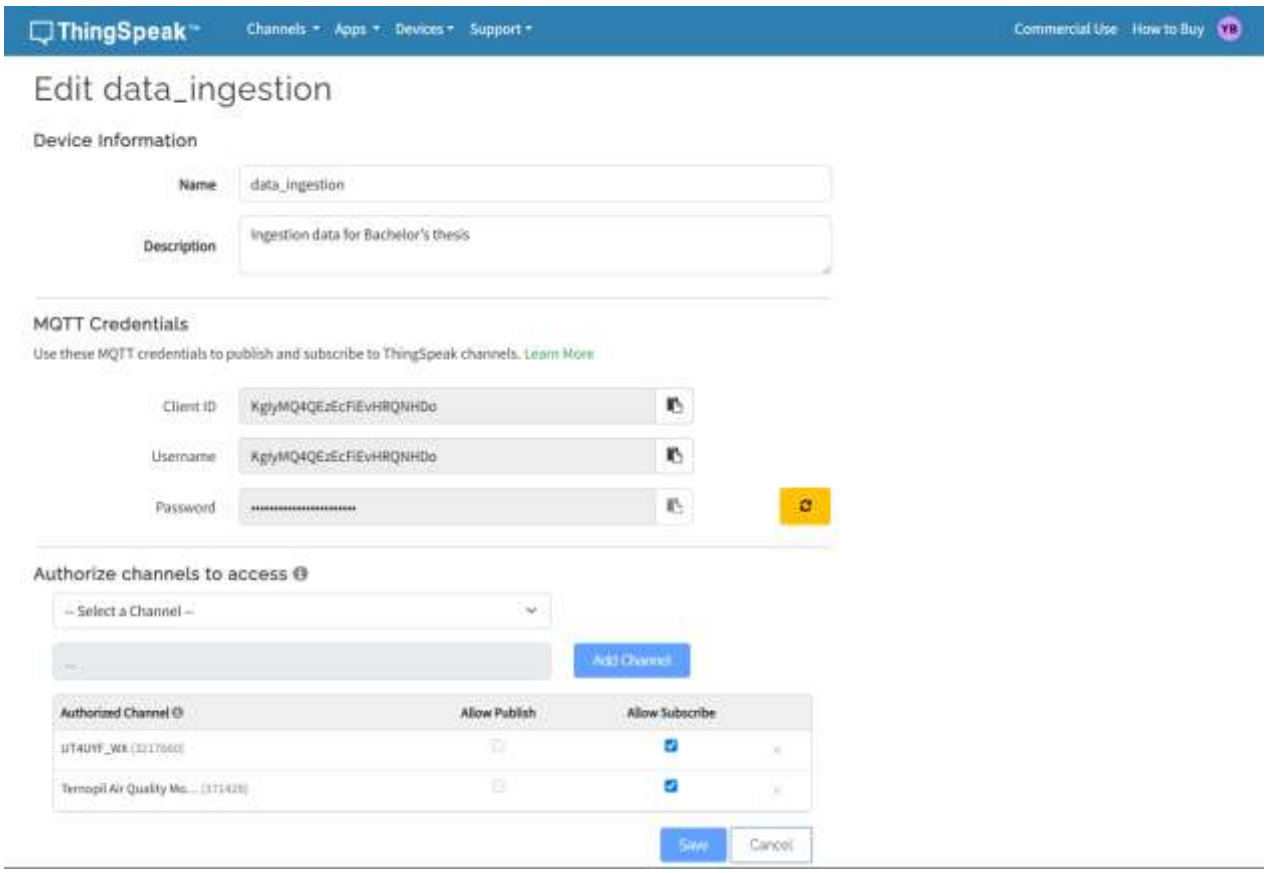

Рисунок 3.11 – Згенеровані ідентифікатори доступу для підключення до брокера ThingSpeak

Для забезпечення безперебійного надходження даних із зовнішніх IoT-пристроїв у розроблену розподілену комп'ютерну систему, реалізуємо та розгорнимо спеціалізований сервіс MQTT bridge [19]. Основним завданням цього компонента є створення стійкого інтеграційного шлюзу між хмарним брокером ThingSpeak, що працює за протоколом MQTT, та внутрішньою шиною даних Apache Kafka. Таке архітектурне рішення дозволяє трансформувати різні потоки неструктурованих IoT-даних у впорядковані повідомлення, готові до подальшої аналітичної обробки в режимі реального часу [20].

Програмна реалізація сервісу виконаємо мовою Python із використанням високоефективних бібліотек «paho-mqtt» для взаємодії з IoT-брокером та «confluent-kafka» для роботи з чергами повідомлень. В основі сервісу побудуємо клас-обгортку MQTTClientWrapper (додаток Б.2), який використаємо для інкапсуляції логіки мережевих з'єднань та обробки критичних ситуацій.

При ініціалізації об'єкта задаємо унікальний «client_id» та налаштуємо систему зворотних викликів (callbacks). Ключову роль відіграє метод «on_message», який автоматично спрацьовує при надходженні нових даних у межах загального життєвого циклу обробки подій, наведеного на блок-схемі послідовності обробки повідомлення у реальному часі. В середині цього методу застосуємо регулярні вирази для динамічного вилучення ідентифікатора каналу «channel_id» безпосередньо з назви топіка. Це дозволяє нам використовувати отриманий ID як ключ при відправці повідомлення в Apache Kafka, що є критично важливим для коректного поділу даних за конкретними пристроями та дотримання порядку повідомлень [21].

Для гарантування високої надійності передачі реалізуємо механізм попередньої перевірки доступності брокерів перед кожною операцією запису. Використаємо метод «list_topics» з короткою затримкою та переконаємося, що кластер Apache Kafka готовий до прийому, після чого викликаємо «self.producer.produce». В разі переповнення внутрішнього буфера Apache Kafka, система автоматично генерує BufferError, що дозволяє нам коректно зупинити зчитування та уникнути втрати даних. Весь процес керується через метод «start()», який запускає нескінченний цикл «loop_forever()», забезпечуючи автоматичне перепідключення при розривах зв'язку з Thingspeak [17].

Контейнеризацію цього сервісу реалізуємо на основі власного Docker-образу (рис. 3.12) та розгорнемо його в Docker Compose під назвою «mqtt-bridge».

```
FROM python:3.12-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
ENV PYTHONPATH=/app
COPY src ./src
CMD ["python", "src/ingestion/mqtt_bridge/thingspeak/data_ingestion.py"]
```

Рисунок 3.12 – Лістинг коду створеного власного Dockerfile для побудови Docker-контейнера MQTT bridge сервісу

Наступним кроком задамо послідовність запуску за допомогою директиви «depends_on» (рис. 3.13), що прив’язує старт моста до готовності обох брокерів Apache Kafka («kafka-1» та «kafka-2»).

```
...
  mqtt-bridge:
    build:
      context: .
      dockerfile: containers/thingspeak/Dockerfile
    image: mqtt-bridge:latest
    mem_limit: 256m
    cpus: 0.5
    container_name: thingspeak-mqtt-bridge

    env_file:
      - .env
    volumes:
      - mqtt-bridge-logs:/app/logs
    networks:
      - service-network
    depends_on:
      - kafka-1
      - kafka-2
  ...
```

Рисунок 3.13 – Лістинг коду оголошення залежностей для запуску сервісу MQQT bridge в Docker Compose

Для керування конфіденційними даними, такими як логіни та паролі до каналів Thingspeak, використаємо файл змінних середовища «.env», що дозволяє відокремити параметри доступу від програмного коду. Для запуску та фонового виконання сервісу MQTT bridge мвикористаємо команду «docker compose up -d –build», яка дозволяє автоматично зібрати образ та розгорнути контейнер. Моніторинг успішності передачі даних виконаємо за допомогою веб-інтерфейсу Docker Desktop, який зображено на рисунку 3.14, в режимі реального часу. В логах відображається потік вхідних повідомлень із коректно обробленими ключами та корисним навантаженням [18].

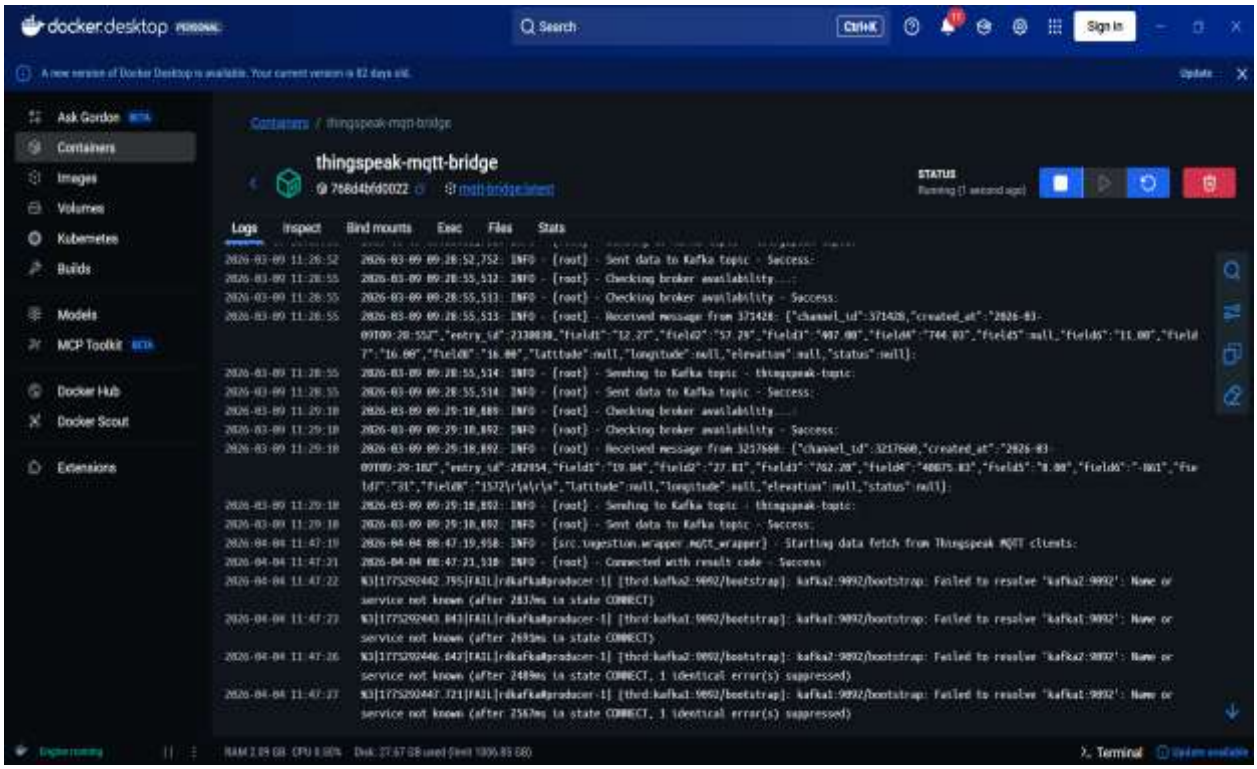

Рисунок 3.14 – Моніторинг успішності передачі даних

Розгорнутий міст стає надійним фундаментом для всієї розподіленої комп’ютерної системи, ізолюючи внутрішню інфраструктуру від специфіки зовнішніх IoT-протоколів.

3.3 Розгортання та конфігурування кластера Apache Kafka

Для створення надійного фундаменту передачі IoT-даних розгорнемо кластер Apache Kafka, що базується на двох брокерах. Використання декількох вузлів дозволяє нам забезпечити відмовостійкість системи та високу доступність даних у реальному часі. Як основу використаємо офіційні образи «confluentinc/cp-kafka» (рис. 3.15), які гарантують стабільність роботи в контейнеризованому середовищі.

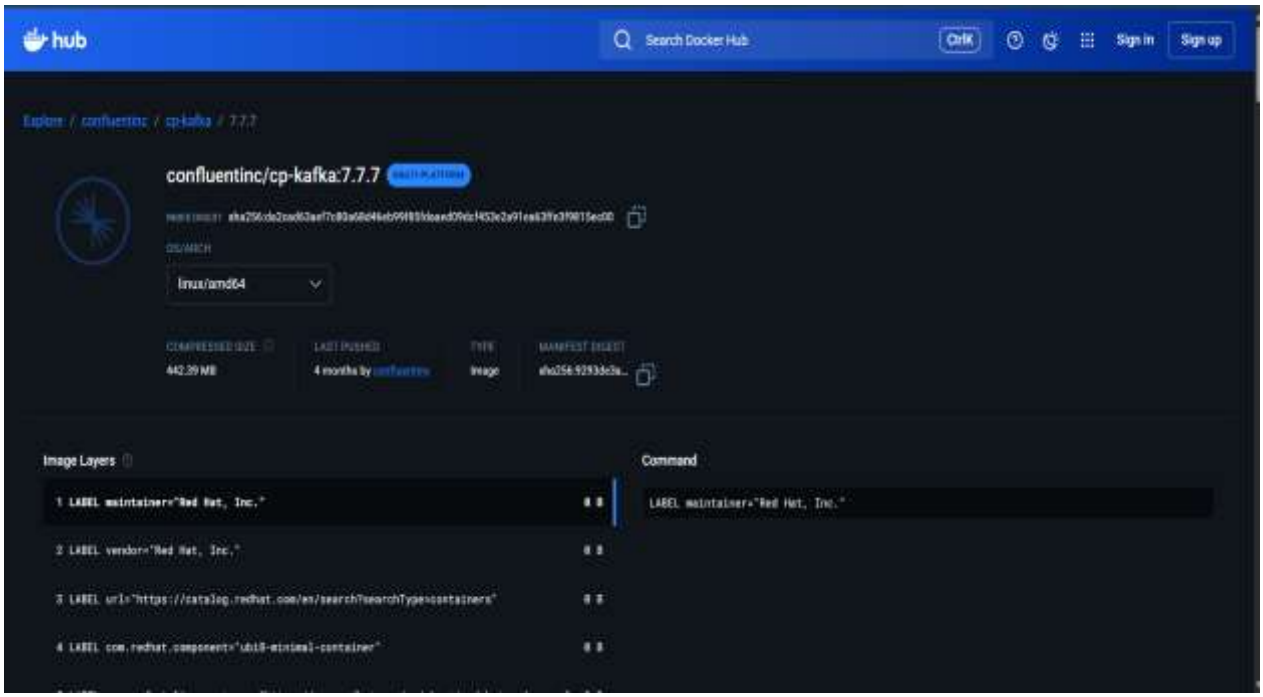

Рисунок 3.15 – Отримання офіційного образу «confluentinc/cp-kafka» від Apache Kafka

Для кожного брокера, «kafka-1» та «kafka-2», встановимо жорсткі ліміти на використання ресурсів хост-системи. Виділимо по 2 ГБ оперативної пам’яті та по 2 ядра процесора на кожен вузол, тим самим забезпечимо стабільну обробку пікових навантажень без зменшення продуктивності. Головною особливістю нашого розгортання є відмова від Zookeeper на користь сучасного режиму KRaft [20].

Перейдемо до налаштування ідентифікації та ролей вузлів (додаток Б.3), де встановимо параметри «KAFKA_NODE_ID» та «KAFKA_BROKER_ID» в значення 1 та 2 відповідно для кожного контейнера. Ключовим етапом конфігурування стане активація сучасного режиму KRaft через параметр «KAFKA_PROCESS_ROLES: "broker,controller"» (додаток Б.3), що дозволить нам відмовитися від Zookeeper і об’єднати функції зберігання даних із керуванням метаданими кластера на кожному вузлі. Для формування стабільної архітектури використаємо «KAFKA_CONTROLLER_QUORUM_VOTERS» (додаток Б.3), де чітко пропишемо адреси всіх контролерів у форматі «ID@hostname:port», а за допомогою унікального «CLUSTER_ID» (Додаток Б.3) логічно об’єднаємо брокери в єдину захищену систему [20].

Паралельно розгорнемо гнучку мережеву інфраструктуру, сконфігуруємо параметр «KAFKA_LISTENERS» (додаток Б.3) для розмежування клієнтського трафіку «PLAINTEXT» та службових команд контролера. Через «KAFKA_ADVERTISED_LISTENERS» (додаток Б.3) надамо зовнішнім сервісам, таким як MQTT-міст, коректну адресу для підключення до брокера, а за допомогою «KAFKA_LISTENER_SECURITY_PROTOCOL_MAP» (додаток Б.3) поставимо в відповідність назви слухачів та протоколи передачі даних. Внутрішня комунікація між вузлами буде забезпечена через параметри «KAFKA_CONTROLLER_LISTENER_NAMES» (додаток Б.3) та «KAFKA_INTER_BROKER_LISTENER_NAME» (додаток Б.3), що гарантує ізоляцію системного трафіку від основного потоку IoT-даних [21].

Для забезпечення максимальної відмовостійкості встановимо відповідна правила реплікації, де параметр «KAFKA_DEFAULT_REPLICATION_FACTOR: 3» (додаток Б.3) змусить систему створювати три копії кожного повідомлення. Також встановимо «KAFKA_MIN_INSYNC_REPLICAS: 2» (додаток Б.3), що гарантує

підтвердження успішного запису лише при наявності даних мінімум на двох вузлах, мінімізуючи ризики втрати інформації.

Стан споживачів захистимо через «KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 2» (додаток Б.3), а для миттєвого розподілу навантаження при старті встановимо затримку перебалансування «KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS» (додаток Б.3) в значення 0. Примусово встановимо «KAFKA_AUTO_CREATE_TOPICS_ENABLE: "false"» (додаток Б.3), що дозволить нам вручну контролювати структуру топіків та їхнє партиціювання.

Налаштуємо систему зовнішніх портів та сховищ, що забезпечить надійний доступ до брокерів та гарантує повну стійкість даних. Для першого вузла відкриємо стандартні порти «9092» та «9093» (додаток Б.3), тоді як для другого вузла використаємо порти «9094» та «9095» (додаток Б.3), щоб уникнути конфліктів на хост-машині. Використання директиви «ports» у нашому файлі «docker-compose.yml» дозволить нам підключатися до кластера не лише всередині віртуальної мережі «service-network», а й через зовнішні інструменти розробки безпосередньо з операційної системи Windows [18].

Особливу увагу приділимо фізичному зберіганню даних, оскільки за допомогою директиви «volumes» встановимо пряму відповідність між внутрішньою директорією брокера «/var/lib/kafka/data» та локальним сховищем на нашому сервері. Як представлено на рисунку 3.16, в вибраній директорії автоматично формується ієрархічна структура каталогів, де для кожного топіка створюються окремі папки партицій (наприклад, «thingspeak-topic-0») [19].

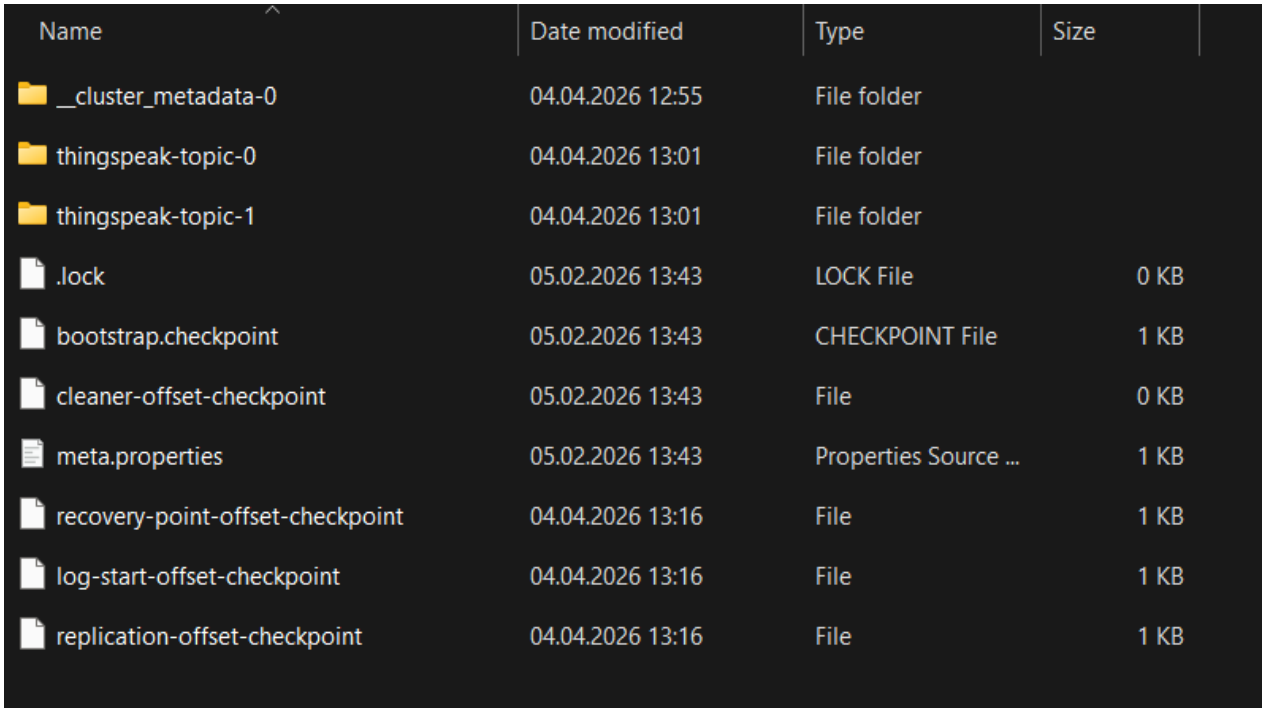

Рисунок 3.16 – Організація зберігання даних із використанням партицій в Apache Kafka

В середині відповідних каталогів Kafka зберігає сегменти повідомлень IoT-даних, що проходять через систему, забезпечуючи їхню стійкість незалежно від стану контейнера.

Окрім самих даних, на диску фіксуються критично важливі метадані та системні чекпоінти, такі як «recovery-point-offset-checkpoint» та «replication-offset-checkpoint». Такі файли відіграють ключову роль у механізмі збереження стану. Вони записують поточні зміщення повідомлень, що дозволяє кластеру точно знати, які дані вже були записані на диск або синхронізовані між репліками. Такий підхід гарантує цілісність розподіленої комп'ютерної системи, адже при перезапуску сервісів система миттєво відновлює роботу з останнього зафіксованого зміщення, уникаючи втрати або дублювання даних.

Для забезпечення зручного моніторингу та візуального адміністрування кластера розгорнемо сервіс Kafka UI, використовуючи офіційний образ «provectuslabs/kafka-ui» (рис. 3.17).

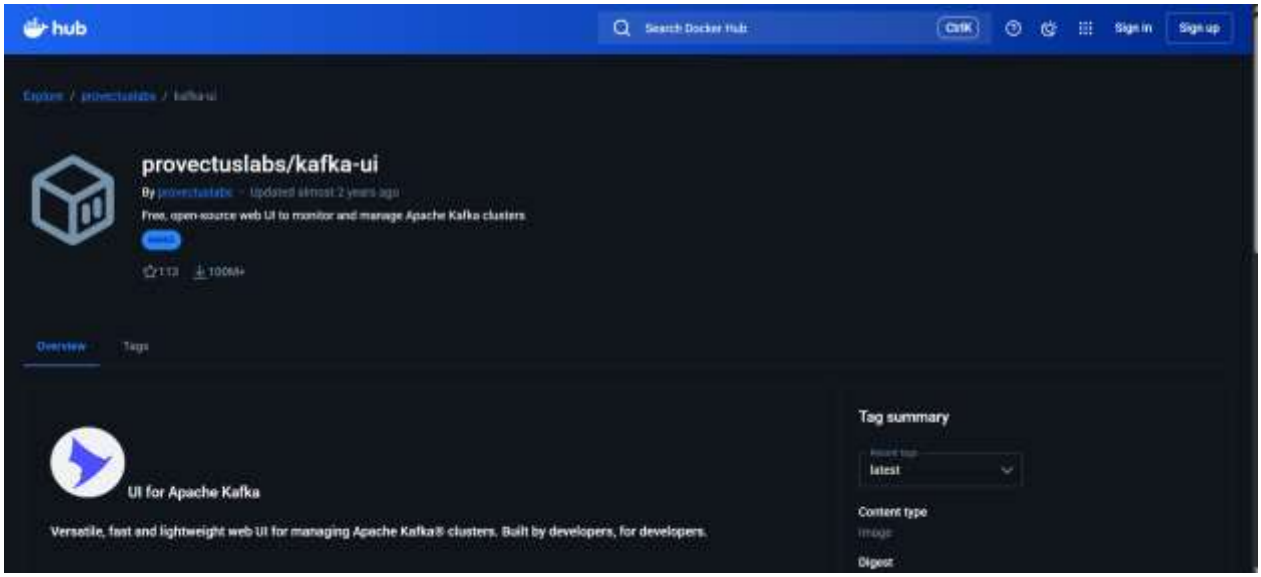

Рисунок 3.17 – Отримання офіційного образу provectuslabs/kafka-ui Apache Kafka UI

Сервіс дозволяє нам відмовитися від складних консольних команд на користь інтуїтивно зрозумілого інтерфейсу. Налаштуємо сервіс, щоб він залежав від брокерів за допомогою директиви «depends_on» (рис. 3.18), а через параметр «KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS» (рис. 3.18) вкажемо внутрішні адреси наших вузлів у мережі «service-network» [20].

```
...
 kafka-ui:
    image: provectuslabs/kafka-ui:latest
    mem_limit: 512m
    cpus: 0.5
    container_name: kafka-cluster-ui
    environment:
      KAFKA_CLUSTERS_0_NAME: local
      KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS: kafka1:9092,kafka2:9092
  ports:
      - 8080:8080
    volumes:
      - kafka-ui-logs:/app/logs
    networks:
      - service-network
    depends_on:
      - kafka-1
      - kafka-2
  ...
```

Рисунок 3.18 – Лістинг коду створення власного образу Kafka UI

Розгорнемо та активуємо інструменти моніторингу кластера через термінал з оболонкою Bash [28]. Для цього запустимо скрипт «dev.sh», який автоматично збирає всю інфраструктуру Apache Kafka разом із необхідними залежностями за допомогою команди «docker compose up -d –build» [18]. Тільки після того, як у консолі з’явиться підтвердження про успішний старт усіх компонентів, тоді отримаємо доступ до візуальних інструментів адміністрування. Процес виконання цього скрипта та успішний запуск усіх контейнерів продемонстровано на рисунку 3.19.

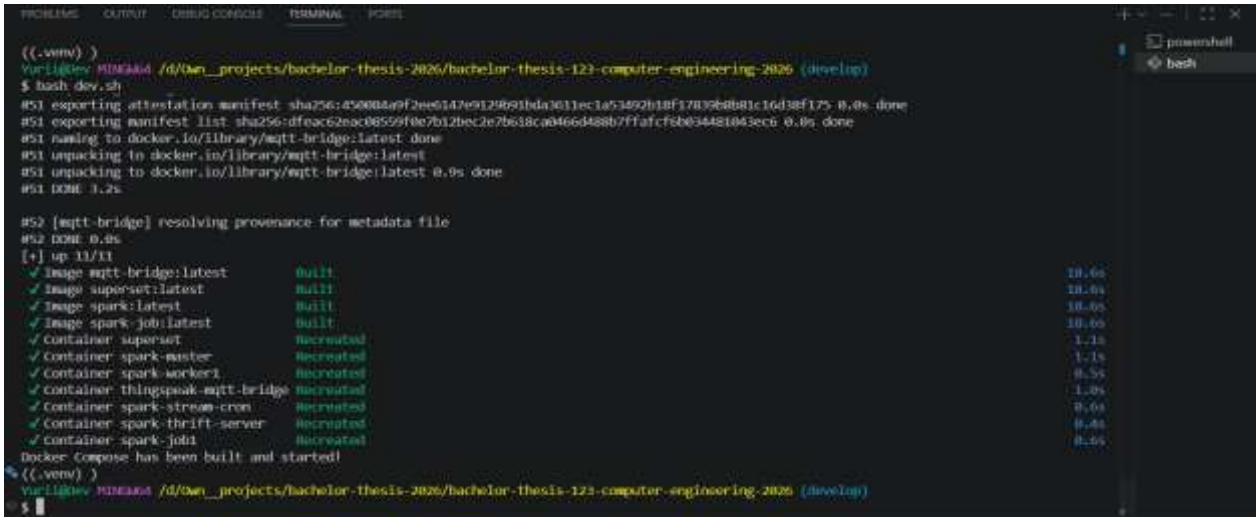

Рисунок 3.19 – Розгортання інфраструктури Apache Kafka

Застосування прапорця «--build» гарантує, що Docker спочатку перезбере всі необхідні образи, враховуючи останні зміни у конфігураційних файлах та змінних оточення. Параметр «-d» дозволить нам розгорнути брокери у фоновому режимі, що забезпечить безперервну роботу шини повідомлень навіть після завершення сесії в консолі.

Після успішного розгортання сервісів отримаємо доступ до графічної панелі керування через браузер за адресою «localhost:8080». Як зображено на рисунку 3.20, головний дашборд системи відображає загальний стан здоров’я розгорнутих брокерів, та підтверджує, що локальний кластер перебуває в статусі «Online».

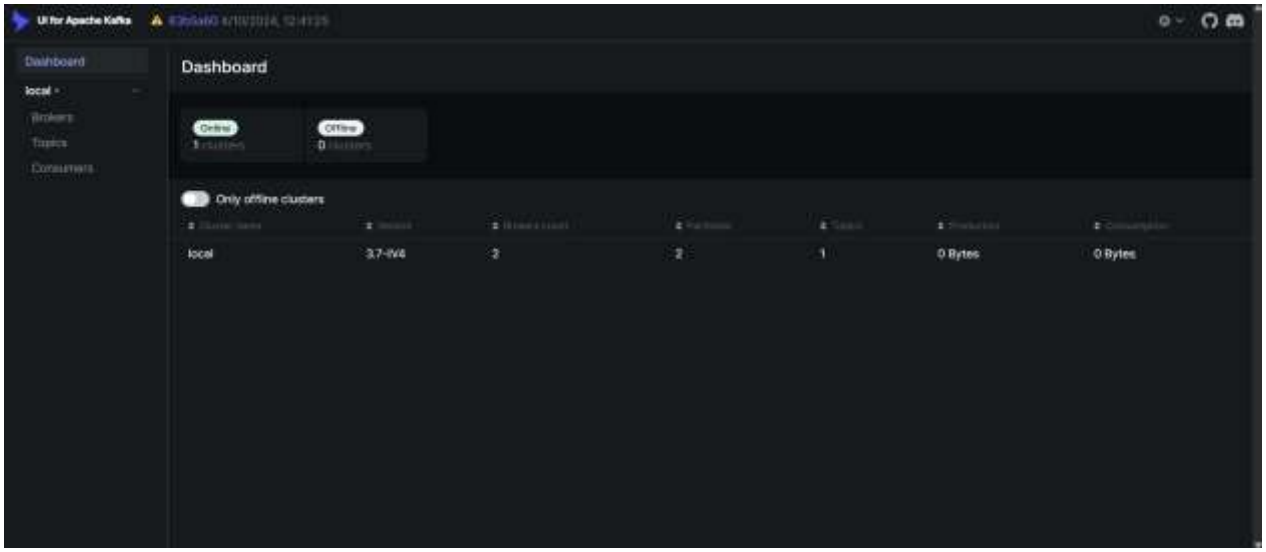

Рисунок 3.20 – Загальний стан ситеми в Kafka UI

В сервісі Kafka UI можемо побачити версію збірки «Kafka (3.7-IV4)», кількість підключених брокерів, а також загальну кількість активних топіків та партицій, що забезпечує швидку перевірку працездатності всієї шини повідомлень.

Перейдемо до детального огляду вузлів у вкладці «Brokers», в якій можемо проаналізувати стан кожного окремого сервера. На рисунку 3.21 зображено вузли «kafka1» та «kafka2», кожен з яких успішно пройшов ініціалізацію.

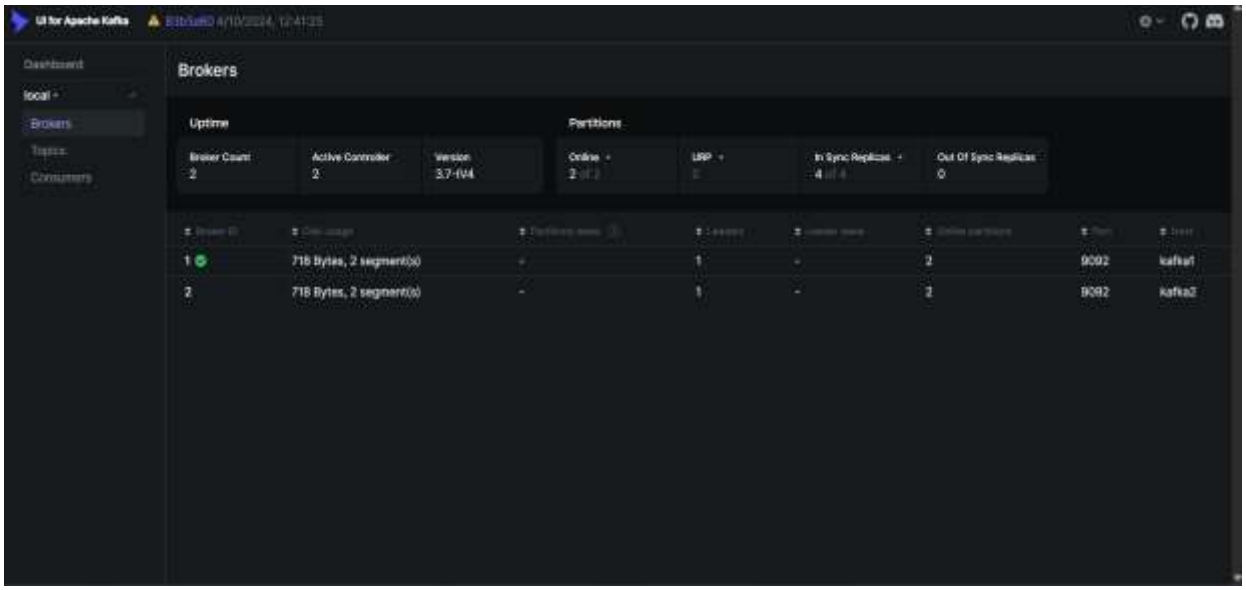

Рисунок 3.21 – Аналітика підключених брокери в Kafka UI

Система відображає їхні унікальні ідентифікатори, обсяг використаного дискового простору та кількість партицій, закріплених за кожним брокером. Наявність статусу «In Sync Replicas (4 з 4)» підтверджує, що дані успішно дублюються між вузлами згідно з налаштуваннями реплікації, що гарантує високу відмовостійку розподіленої комп'ютерної системи [21].

Для моніторингу конкретних потоків даних використаємо вкладку «Topics», де представлена детальна аналітика для кожного каналу телеметрії. Як зображено на рисунку 3.22 на прикладі топіка «thingspeak-topic», інтерфейс дозволяє нам відстежувати кількість повідомлень у реальному часі та структуру партиціювання.

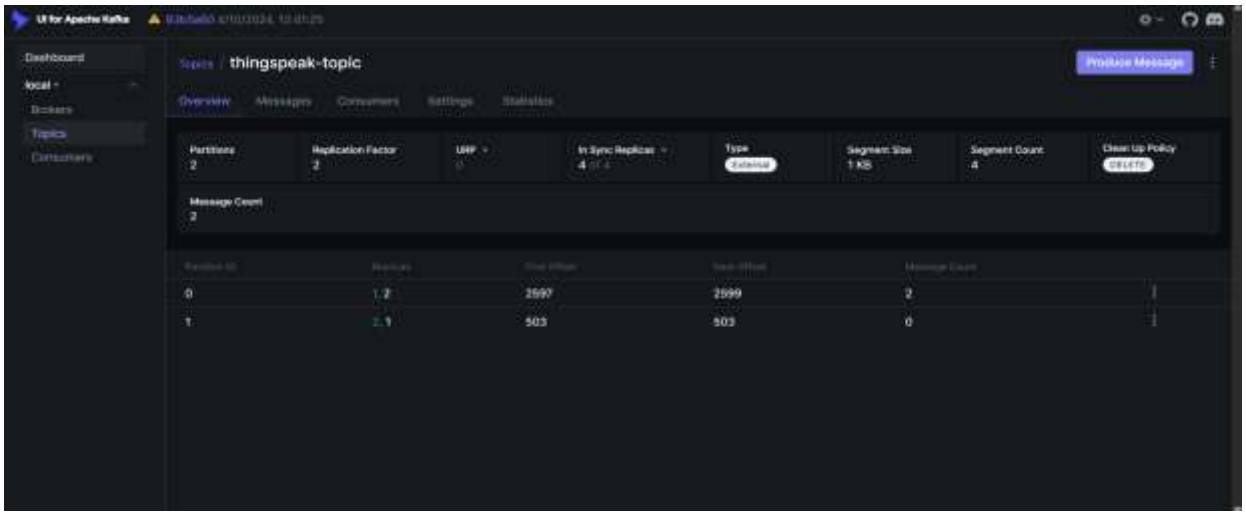

Рисунок 3.22 – Аналітика підключених топіків в Kafka UI

Можемо побачити розподіл на дві партиції («ID 0» та «ID 1») та актуальні значення зсувів («First Offset» та «Next Offset»), що дає нам повне розуміння того, на якому етапі перебуває процес зчитування даних.

Такий рівень візуалізації перетворює консольну інфраструктуру на прозоре середовище, де можемо оперативно виявляти затримки в обробці, помилки в конфігурації реплікта проблеми в роботі брокерів [19].

3.4 Розгортання та конфігурування кластера Apache Spark

Для реалізації центрального обчислювального шару нашої розподіленої комп'ютерної системи розгорнемо кластер Apache Spark в режимі «standalone». Вибір цього інструменту зумовлений його здатністю до високоефективної розподіленої обробки великих масивів даних у реальному часі. В межах нашої розподіленої комп'ютерної системи Apache Spark виконує роль «розумного ядра», яке не просто пересилає байти, а здійснює складну аналітичну трансформацію вхідного потоку IoT-даних перед його збереженням в об'єктному сховищі [23].

Використання Apache Spark дозволить нам впровадити технологію Structured Streaming, яка читає потік даних із Apache Kafka як нескінченну таблицю, що постійно доповнюється. Це надає нам можливість застосовувати звичайні SQL операції для фільтрації, агрегації та очищення даних безпосередньо «на льоту» [25]. Такий підхід забезпечує мінімальну затримку між моментом виникнення події на IoT-датчику та її появою в структурованому вигляді в об'єктному сховищі MinIO [22].

Процес підготовки обчислювального середовища розпочнемо із проєктування та створення оптимізованого власного Docker-образу. Оскільки стандартні образи Apache Spark часто не містять специфічних конекторів для роботи з сучасними форматами даних, розробимо власну конфігурацію, яка дозволить системі безперешкодно взаємодіяти з усіма компонентами нашої розподіленої комп'ютерної системи. Ключовою особливістю збірки є підхід «multi-stage build» для побудови Dockerfile (додаток Б.4) [25].

На першому етапі, використаємо образ «maven:3.9.3-eclipse-temurin-17» (додаток Б.4) та розгорнемо тимчасове середовище для підготовки залежностей. Використаємо Maven для автоматизації процесу завантаження та пакування необхідних JAR-файлів. Через файл «maven.xml» задамо чіткий перелік бібліотек, таких як конектори «spark-sql-kafka» для отримання IoT-даних, «iceberg-spark-runtime» для транзакційного зберігання та «hadoop-aws»

разом із «aws-java-sdk-bundle» для інтеграції з S3-сумісним сховищем MinIO (рис 3.23) [22].

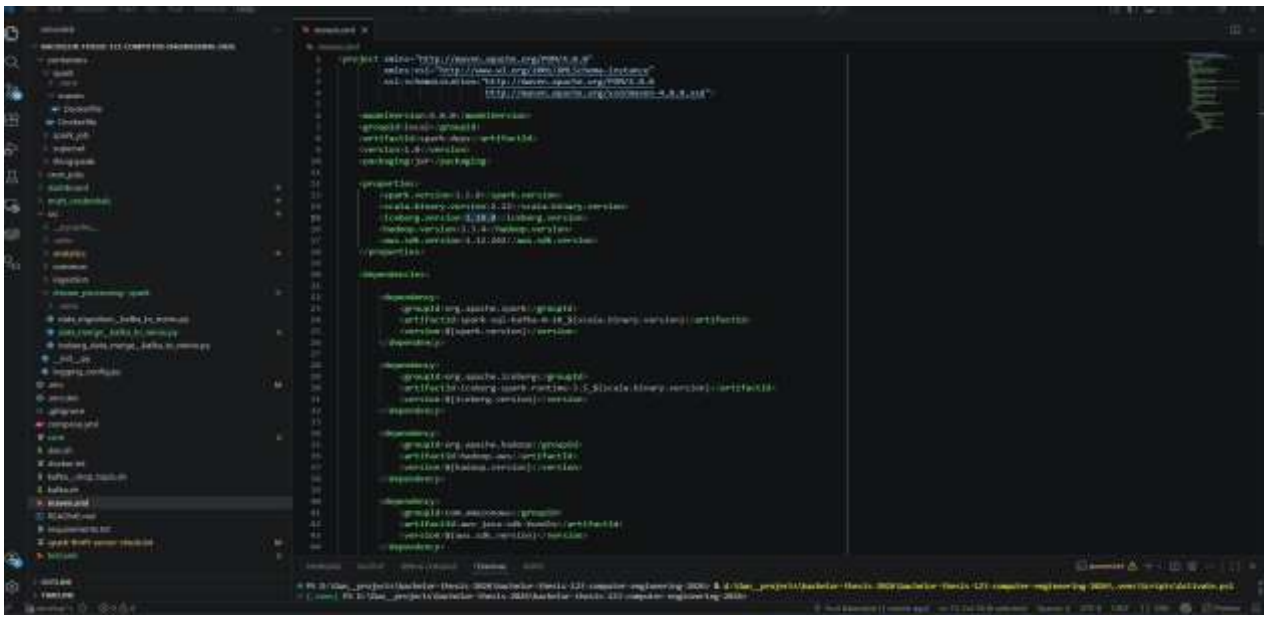

Рисунок 3.23 – Опис maven.xml файлу для збірки залежностей за домогою Maven

Це дозволить нам зібрати всі залежності в єдиний артефакт, виключаючи ризик конфлікту версій або відсутності критичних класів під час виконання завдання.

На другому етапі перейдемо до формування фінального образу на базі «python:3.12-bullseye» (додаток Б.4), що забезпечить нам легкість середовища та базову підтримку PySpark. Встановимо середовище виконання OpenJDK 17 (додаток Б.4), яке є рекомендованим для стабільної роботи Spark версії «3.5.8». Через інструкції «ENV» поставимо всі необхідні системні змінні, такі як «SPARK_HOME» та «JAVA_HOME», а також оновимо глобальний шлях «PATH», щоб забезпечити доступ до команд «spark-submit» та «pyspark» з будь-якої точки системи (додаток Б.4).

Використаємо інструкцію «COPY --from=deps» (додаток Б.4), щоб перенести підготовлені Maven бібліотеки безпосередньо в системну папку «jars/» нашого Apache Spark-кластера. Такий підхід гарантує, що вузли

матимуть повноцінну підтримку роботи з S3-та транзакційними таблицями Iceberg, не потребуючи додаткового завантаження пакетів з центарльного репозиторю [27].

Для координації обчислювальних потужностей розгорнемо Apache Spark-кластер в Docker Compose та опишемо залежні інструкції в «compose.yml» файлі, що дозволить нам гнучко керувати ресурсами та забезпечить високу доступність сервісів (додаток Б.5).

Спочатку розгорнемо вузол «spark-master», який виконуватиме роль центрального координатора ресурсів [2]. Відкриємо порт «8082» (додаток Б.5) для доступу до Web UI, що дозволить нам моніторити стан кластера, та порт «7077» (додаток Б.5), який буде внутрішнім шлюзом, через який воркери підключатимуться до драйвера для отримання завдань. За допомогою директиви «environment» встановимо параметр «SPARK_NO_DAEMONIZE: true» (додаток Б.5), щоб логи Apache Spark виводилися безпосередньо в консоль Docker, полегшуючи процес відлагодження [18].

Паралельно запустимо вузол «spark-worker», який є безпосередньою робочою нодою кластера. Виділимо для нього значні ресурси, що включає 4 ядра процесора та 6 ГБ оперативної пам’яті, використовуючи змінні «SPARK_WORKER_CORES» та «SPARK_WORKER_MEMORY» (додаток Б.5). Така конфігурація дозволить нам паралельно обробляти декілька партицій даних із Apache Kafka, забезпечуючи високу пропускну здатність нашої розподіленої комп’ютерної системи [20].

Для того, щоб кластер працював як єдина злагоджена система, налаштуємо файл «conf/spark-defaults.conf». В ньому зробимо глобальні налаштування, які будуть автоматично застосовуватися до всіх завдань. Пропишемо конфігурації для каталогу Iceberg, встановивши тип сховища «hadoop» та вказавши шлях до сховища MinIO [22]. Також розгорнемо налаштування S3 кінцеових точок, прописавши доступ до нашого локального хмарного сховища. Це дозволить Apache Spark автоматично розпізнавати структуру сховища та коректно працювати з транзакційними таблицями без

необхідності дублювати ці параметри в кожному окремому Python-скрипті (рис. 3.24).

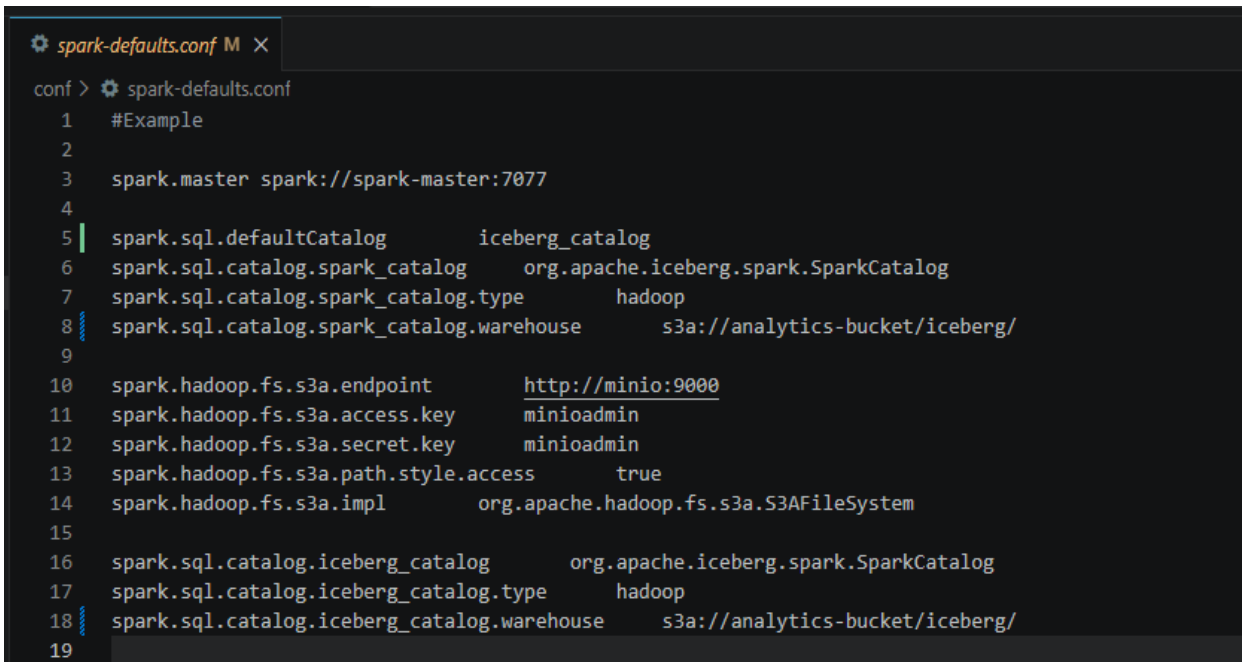

Рисунок 3.24 – Опис «spark-defaults.conf» файлу

Також розгорнемо сервіс «spark-job1», який виступатиме клієнтом для подачі нашого стрімінгового завдання [18]. Виділимо для нього 2 ядра процесора та 4 ГБ оперативної пам’яті, що забезпечить стабільну роботу драйвера (рис. 3.25).

```
...
spark-job1:
    build:
      context: .
      dockerfile: containers/spark_job/Dockerfile
    image: spark-job:latest
    mem_limit: 4g
    cpus: 2.0
    container_name: spark-job1
    env_file:
      - .env
    entrypoint: /bin/bash
        --driver-memory 1g \
     command: -c "spark-submit \
          --executor-memory 2g \
          --executor-cores 2 \
          --num-executors 2 \
          --conf spark.cores.max=2 \
          --conf spark.sql.shuffle.partitions=100 \
--conf spark.streaming.kafka.maxRatePerPartition=500 \
          --packages org.apache.spark:spark-sql-kafka-0
10_2.12:3.5.8,org.apache.hadoop:hadoop-aws:3.3.4,com.amazonaws:aws-java
sdk-bundle:1.12.262,org.apache.iceberg:iceberg-spark-runtime
3.5_2.12:1.10.0 \
/opt/spark/jobs/src/stream_processing/spark/data_ingestion__kafka_to_mini
.py"
    volumes:
      - ./conf:/opt/spark/conf
      - spark-job1-logs:/app/logs
    networks:
      - service-network
    depends_on:
      - spark-master
      - spark-worker
      - minio
...
```

Рисунок 3.25 – Лістинг коду розгортання Apache Spark Job за допомогою Docker Compose

За допомогою інструкції «env_file» підключимо файл «.env», що дозволить Apache Spark-завданню безпечно отримувати доступ до облікових даних MinIO та параметрів бази даних. Запуск самого завдання зробимо через команду «spark-submit», де розгорнемо детальне налаштування ресурсів, включаючи обмеження поділу даних «maxRatePerPartition=500» (рис. 3.25). Це дозволить нам збалансувати навантаження та забезпечити принцип «exactly-once» обробки даних [19].

Особливу увагу приділимо використанню директиви «depends_on». Налаштуємо Apache Spark залежність від сховища MinIO, гарантуючи, що обчислювальний кластер почне роботу лише тоді, коли об'єктне сховище буде повністю готове до приймання даних. Такий підхід до оркестрації дозволить нам створити стійку інфраструктуру, де кожен компонент чітко знає свою роль та параметри взаємодії з іншими частинами системи [22].

Основна логіка перетворення даних зосередиться в скрипті «data_ingestion kafka_to_minio.py». Оголосимо створення SparkSession, яку сконфігуруємо для роботи з Hadoop-каталогом Iceberg, що дозволить поєднувати гнучкість SQL-таблиць із надійністю об'єктного сховища (рис. 3.26).

```
...
spark = (
    SparkSession.builder
    .appName("thingspeak__kafka_to_minio")
    .config("spark.sql.catalog.iceberg_catalog",
    "org.apache.iceberg.spark.SparkCatalog")
    .config("spark.sql.catalog.iceberg_catalog.type", "hadoop")
    .config("spark.sql.catalog.iceberg_catalog.warehouse",
    env_config.iceberg_warehouse_catalog)
    .getOrCreate()
)
...
```

Рисунок 3.26 – Лістинг коду створення SparkSession в файлі «data_ingestion__kafka_to_minio.py»

Процес обробки розгорнемо через декілька логічних етапів. Спершу зробимо зчитування потоку, налаштувавши «readStream» для підключення до брокерів нашого внутрішнього кластера «(kafka1:9092, kafka2:9092)». Підпишемося на топік «thingspeak-topic» та встановимо параметр «startingOffsets», щоб гарантувати обробку всіх доступних повідомлень із моменту першого пуску системи. Оскільки дані надходять у форматі JSON, наступним кроком виконаємо парсинг та типізацію [23].

Застосуємо сувору схему «StructType», яка описує структуру полів Thingspeak (рис. 3.27).

```
...
schema = (
StructType()
    .add("channel_id", StringType())
    .add("entry_id", StringType())
    .add("created_at", StringType())
    .add("field1", StringType())
    .add("field2", StringType())
    .add("field3", StringType())
    .add("field4", StringType())
    .add("field5", StringType())
    .add("field6", StringType())
    .add("field7", StringType())
    .add("field8", StringType())
)
...
```

Рисунок 3.27 – Лістинг коду оголошення StructType в файлі «data_ingestion__kafka_to_minio.py»

Це дозволить нам перетворити неструктурований бінарний «payload» із Apache Kafka в чітку табличну форму, готову до аналітики.

Найважливішу частину роботи реалізуємо на етапі трансформації (ETL). Розгорнемо складну логіку очищення та нормалізації полів, що включає:

– використання функції «trim()» для видалення зайвих пробілів у текстових даних (додаток Б.6);

– приведення типів до Decimal(18,4), що забезпечить високу точність для IoT-даних (додаток Б.6);

– реалізацію умовної логіки «when/otherwise» для динамічного мапінгу полів (додаток Б.6);

– додавання технічних міток часу «dwr_extracted_at» та «dwr_created_at» для забезпечення прозорості аудиту даних (додаток Б.6).

Після всіх трасформацій зробимо запис у сховище, використовуючи формат Iceberg з режимом «append». Щоб гарантувати стабільність та дотримання принципу «exactly-once», налаштуємо параметр «checkpointLocation» безпосередньо в бакеті MinIO (рис. 3.28).

```
...
query = (
    processed_df.writeStream
    .format("iceberg")
    .outputMode("append")
    .option("path", env_config.iceberg_table_path)
    .option("checkpointLocation",
env_config.iceberg_checkpointlocation_path)
    .start()
)
...
```

Рисунок 3.28 – Лістинг коду налаштування параметра «checkpointLocation» в файлі «data_ingestion__kafka_to_minio.py»

Це дозволить нашому Apache Spark-завданню автоматично зберігати стан обробки, завдяки чому при перезапуску або технічному збої система миттєво відновить роботу з останнього успішно зафіксованого зміщення, не дублюючи дані в аналітичних таблицях [27].

Після успішного розгортання сервісів та налатшування всіх потрібних файлів, перейдемо до етапу активного моніторингу, щоб переконатися в коректності взаємодії Apache Spark із суміжними сервісами. Початкову перевірку стану кластера зробимо через Web UI Spark Master на порту «8082» (рис. 3.29).

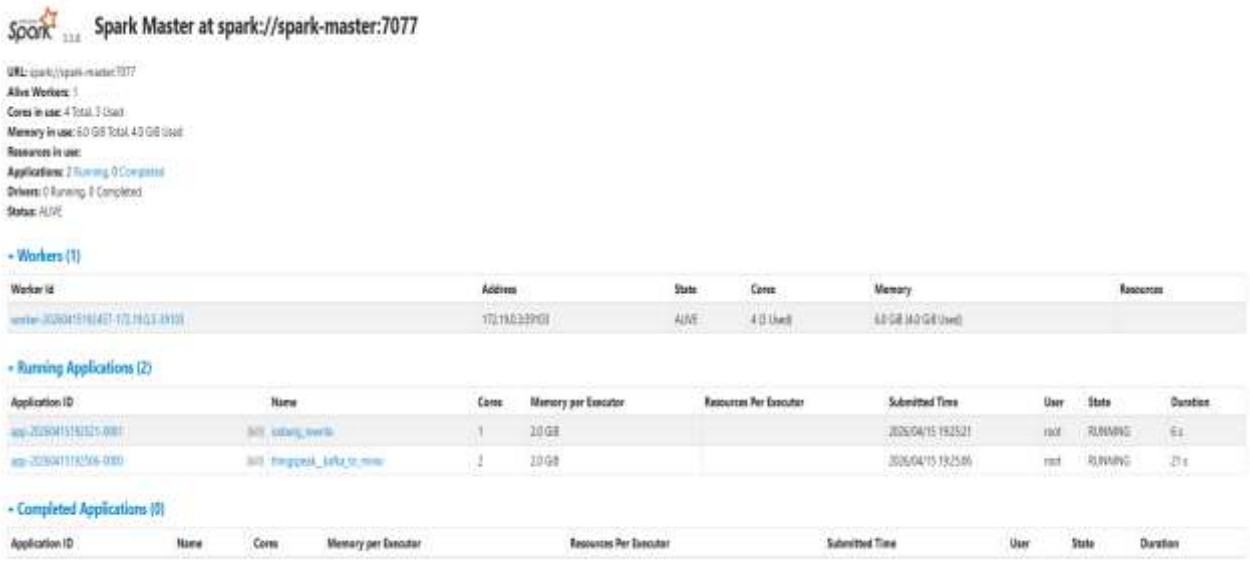

Рисунок 3.29 – Головна сторінка Web UI Spark Master

В розділі «Workers» побачимо статус «ALIVE» для нашого воркера, а в секції «Running Applications» відображається активне стрімінгове завдання «thingspeak__kafka_to_minio». Це підтверджує, що координатор успішно прийняв запит на обробку та виділив необхідні ресурси [19].

Паралельно можемо спостерігати за життєвим циклом системи через системні логи в Docker Desktop. На рисунку 3.30 зображено системний вивід інформації контейнера «spark-master», де представлено процес реєстрації вузла та призначення йому системних ресурсів.

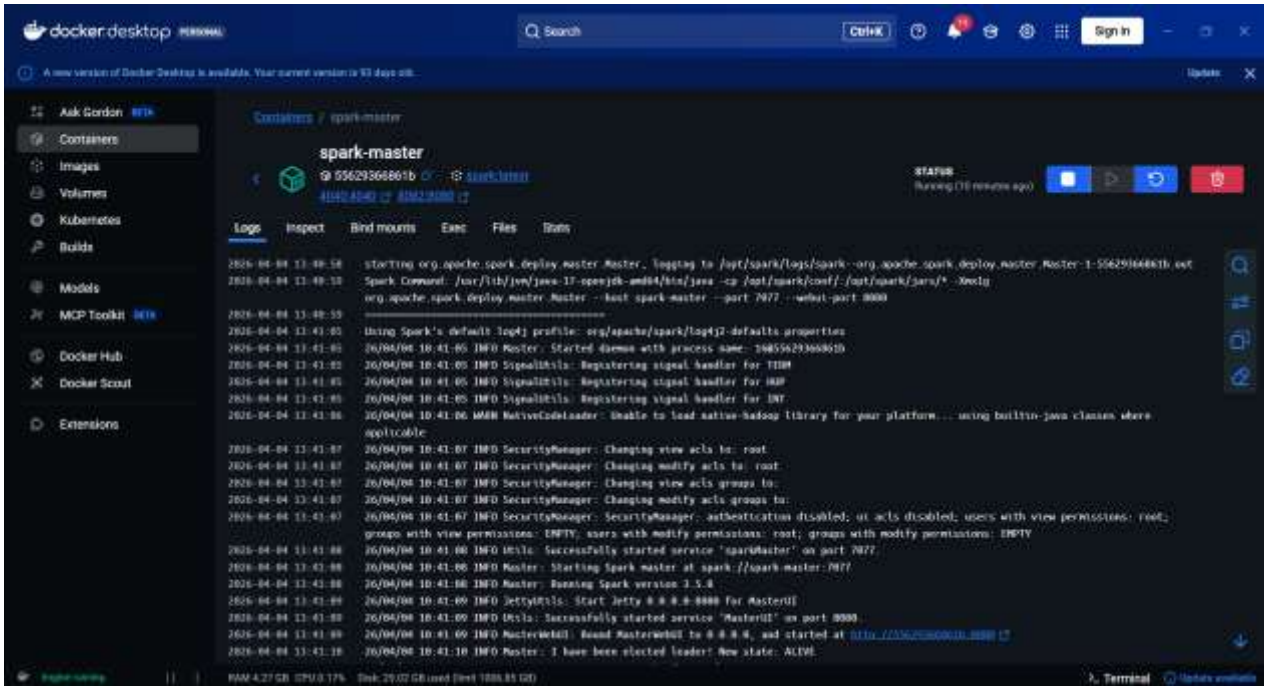

Рисунок 3.30 – Процес запуску Spark Master сервісу

На рисунку 3.31 зображено системний вивід інформації контейнера «spark-job1», де зафіксовано процес реєстрації завдання та призначення йому обчислювальних ресурсів.

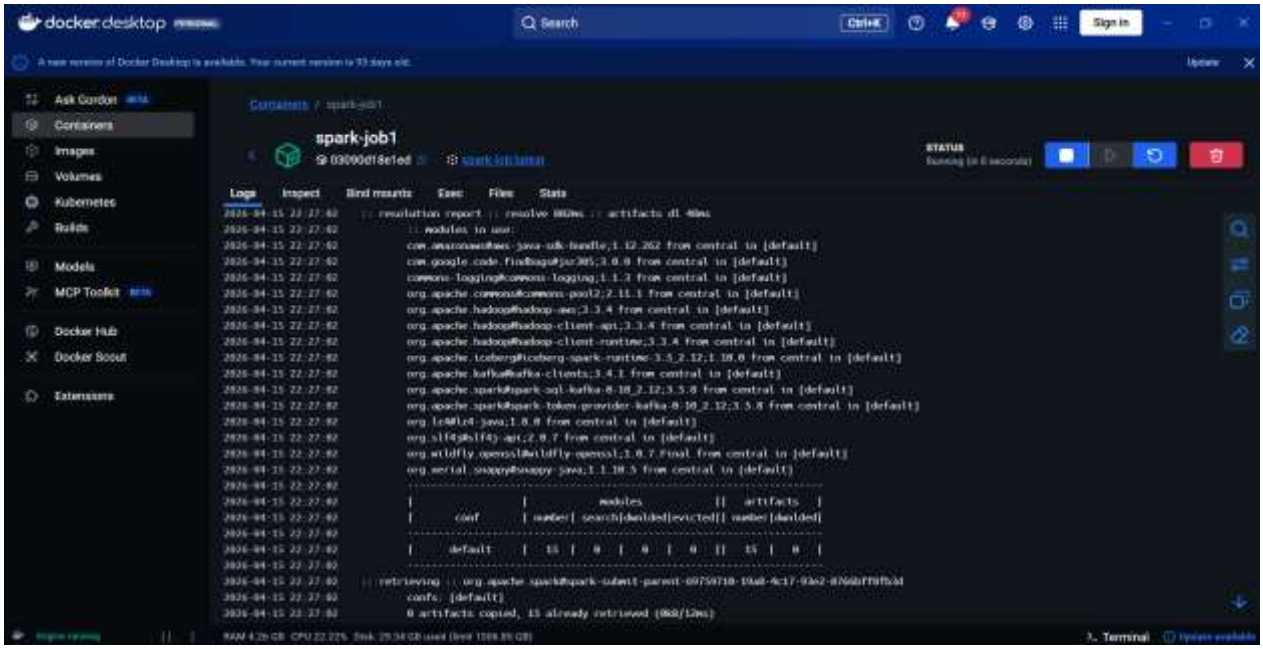

Рисунок 3.31 – Процес запуску Apache Spark Job сервісу для обробки даних

Успішний запуск та стабільний початок обробки IoT-даних підтвердимо логами воркера, які зображено на рисунку 3.32 [23].

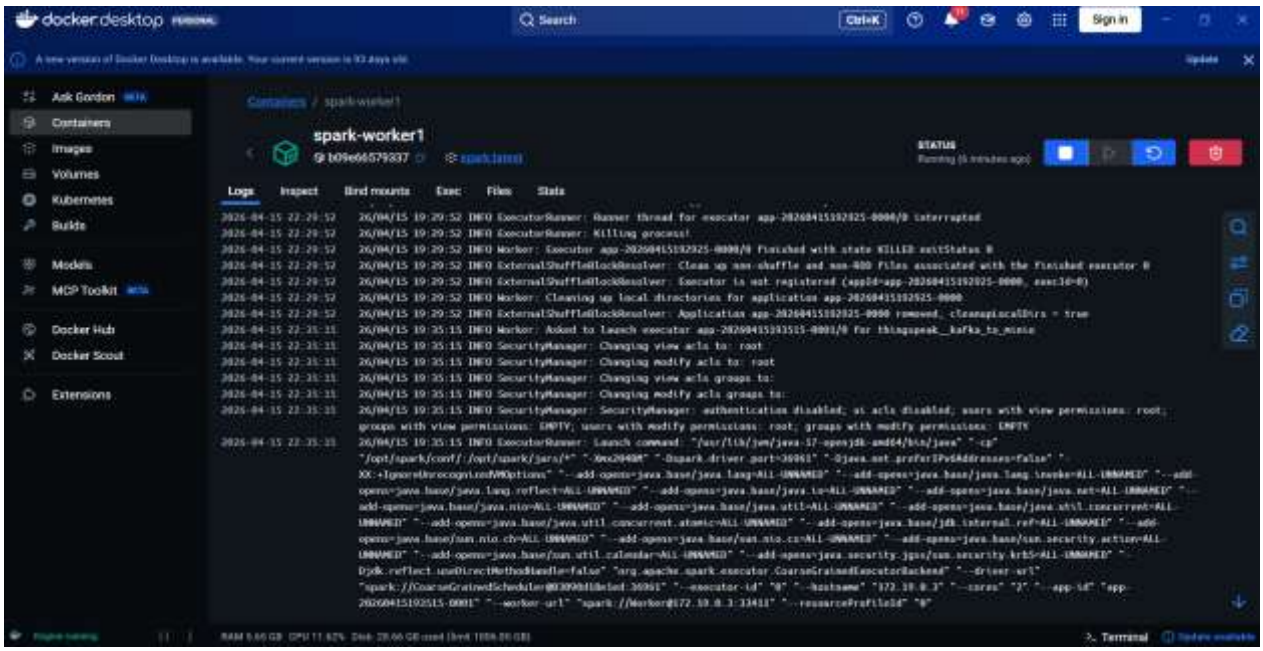

Рисунок 3.32 – Успішний запуск Apache Spark Worker сервісу

На рисунку зображено запуск «CoarseGrainedExecutorBackend» та ініціалізація «StreamExecution». Завдяки такому багаторівневому моніторингу

зможемо повністю контролювати кожен етап життєвого циклу даних у нашій розподіленій комп'ютерній системі від моменту підключення Apache Kafka до успішного запису даних у об'єктне сховище MinIO.

3.5 Налаштування об'єктного сховища MinIO із використанням технології Iceberg та ініціалізація бакетів

Для створення надійного рівня зберігання даних у нашій розподіленій комп'ютерній системі розгорнемо сервіс MinIO, який виступає високоефективним S3-сумісним об'єктним сховищем. В нашій системі обробки даних MinIO виконує роль Data Lakehouse, де поєднуються переваги дешевого об'єктного зберігання та аналітичних можливостей традиційних баз даних. Такий підхід дозволить нам не просто зберігати файли, а організувати повноцінний «срібний» рівень даних (Silver Layer), де інформація вже очищена, типізована та готова до миттєвого аналізу [24]. Завдяки високій швидкості введення-виведення та підтримці транзакційності через формат Iceberg, MinIO забезпечує цілісність даних навіть при інтенсивному потоковому навантаженні, коли тисячі записів IoT-даних щосекунди надходять із Apache Kafka через Apache Spark-виконавця. Вибір такої архітектури обумовлений повною сумісністю з протоколами AWS S3, що дозволяє нам використовувати потужні корпоративні інструменти обробки, такі як Apache Spark та Apache Iceberg, безпосередньо в локальному середовищі або на власних серверах [22].

Розгортання об'єктного сховища виконаємо за допомогою Docker Compose, та використаємо офіційний образ «quay.io/minio/minio» (рис 3.33).

```
...
  minio:
    image: quay.io/minio/minio
    mem_limit: 512m
    cpus: 1.0
    container_name: minio
    environment:
      - MINIO_ROOT_USER=minioadmin
      - MINIO_ROOT_PASSWORD=minioadmin
    command: server /data --console-address ":9001"
    volumes:
      - ${MINIO_PATH}:/data
      - minio-logs:/app/logs
    networks:
      - service-network
    ports:
      - 9000:9000
      - 9001:9001
...
```

Рисунок 3.33 – Лістинг коду розгортання MinIO за допомогою Docker Compose

Такий підхід дозволить нам інтегрувати сховище в єдину ізольовану мережу та забезпечити його автоматичний запуск разом із іншими сервісами. Щоб гарантувати стабільність роботи в умовах інтенсивного потоку даних встановимо кількість доступних ресурсів. Виділимо контейнеру 512 МБ оперативної пам’яті та 1 ядро процесора. Такий підхід не дозволить надмірному споживанню ресурсів хоста та забезпечить передбачувану продуктивність при виконанні запису IoT-даних [23].

Використовуючи змінні середовища «MINIO_ROOT_USER» та «MINIO_ROOT_PASSWORD» (рис 3.33), встановимо облікові дані, які винесемо для забезпечення безпеки доступу в файл конфігурації «.env».

Для того щоб не втратити дані після зупинки або оновлення контейнера, застосуємо директиву «volumes» (рис 3.33). Оголосимо шлях до локальної папки на хості, яку визначемо в змінній «${MINIO_PATH}» (рис 3.33), та внутрішньою робочою директорією контейнера «/data». Такий підхід гарантує фізичне збереження кожного байта отриманих даних на диску хоста незалежно від життєвого циклу сервісу [18].

Налаштуємо MinIO на роботу в внутрішній мережі «service-network», що дозволить іншим сервісам звертатися до сховища за допомогою внутрішнього DNS-іменні «minio:9000». За допомогою директиви «ports» (рис 3.33) відкриємо порт «9000» для програмного доступу до даних за протоколом S3, через який Apache Spark та Apache Kafka будуть взаємодіяти зі сховищем. Також відкриємо порт «9001», який доставить нам доступ до візуальної консолі керування MinIO Console. Особливу увагу приділимо безпеці та контролю доступу [22].

Перед початком створення бакетів через веб-інтерфейс виконаємо перевірку контейнера в середовищі Docker. На рисунку 3.34 зображено системні логи контейнера, що підтверджують успішний запуск сервісу MinIO, ініціалізації користувачів та виділення системних ресурсів, що сигналізує нам про готовність сховища до приймання IoT-даних.

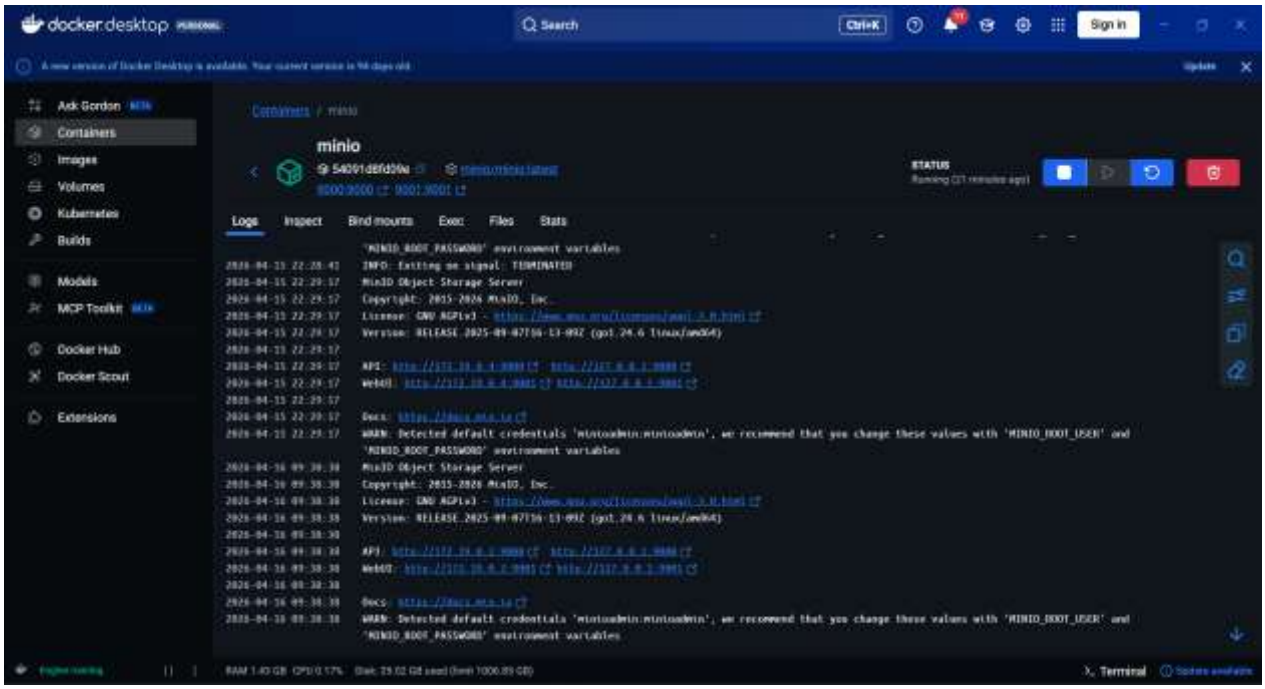

Рисунок 3.34 – Системні логи контейнера MinIO сервісу

Після успішного запуску сервісу перейдемо до веб-інтерфейсу керування за адресою «localhost:9001». На рисунку 3.35 зображено головну сторінку авторизації, яку використаємо для входу.

Рисунок 3.35 – Веб-інтерфейс головної сторінки входу в MinIO сервіс

Як тільки успішоно ввійдемо в систему, перейдемо до створення бакета під назвою «analytics-bucket», який виступатиме кореневим сховищем для всієї нашої аналітичної інфраструктури. Розділимо сховище на логічні шари, де відповідно до медальйонної архітектури на срібному рівні (Silver Layer), згідно зі структурною схемою розгортання компонентів розподіленої системи обробки даних, через змінні «ICEBERG_DB_PATH» та «ICEBERG_TABLE_PATH» в конфігураційному файлі «.env» (рис. 3.36) визначимо шлях до створеного нами бакета «s3a://analytics-bucket/iceberg/silver/kafka_stream» [18].

```
#-----------------------------------------
# MinIO
MINIO_PATH=D:/Own__projects/bachelor-thesis-2026/minio-local-file-storage
ACCESS_KEY=minioadmin
SECRET_KEY=minioadmin
S3A_ENDPOINT=http://minio:9000
S3_BUCKED=s3a://analytics-bucket/thingspeak/
S3_BUCKED_CHECKPOINTLOCATION=s3a://analytics-bucket/thingspeak/_checkpoint
ICEBERG_DB_PATH=s3a://analytics-bucket/iceberg/silver
ICEBERG_TABLE_PATH=s3a://analytics-bucket/iceberg/silver/kafka_stream
ICEBERG_CHECKPOINTLOCATION_PATH=s3a://analytics-bucket/iceberg/metadata/_checkpoint
ICEBERG_WAREHOUSE_CATALOG=s3a://analytics-bucket/iceberg/
ICEBERG_TABLE=iceberg_catalog.silver.kafka_stream
ICEBERG_DB=iceberg_catalog.silver
#-----------------------------------------
```

Рисунок 3.36 – Файл конфігурації «.env»

Саме тут Apache Spark зберігатиме вже очищені та типізовані IoT-дані в табличному форматі Iceberg. Також встановимо шлях «ICEBERG_CHECKPOINTLOCATION_PATH» (рис. 3.36) для збереження контрольних точок, що дозволить системі миттєво відновлюватися після збоїв та гарантувати обробку кожного повідомлення рівно один раз [27].

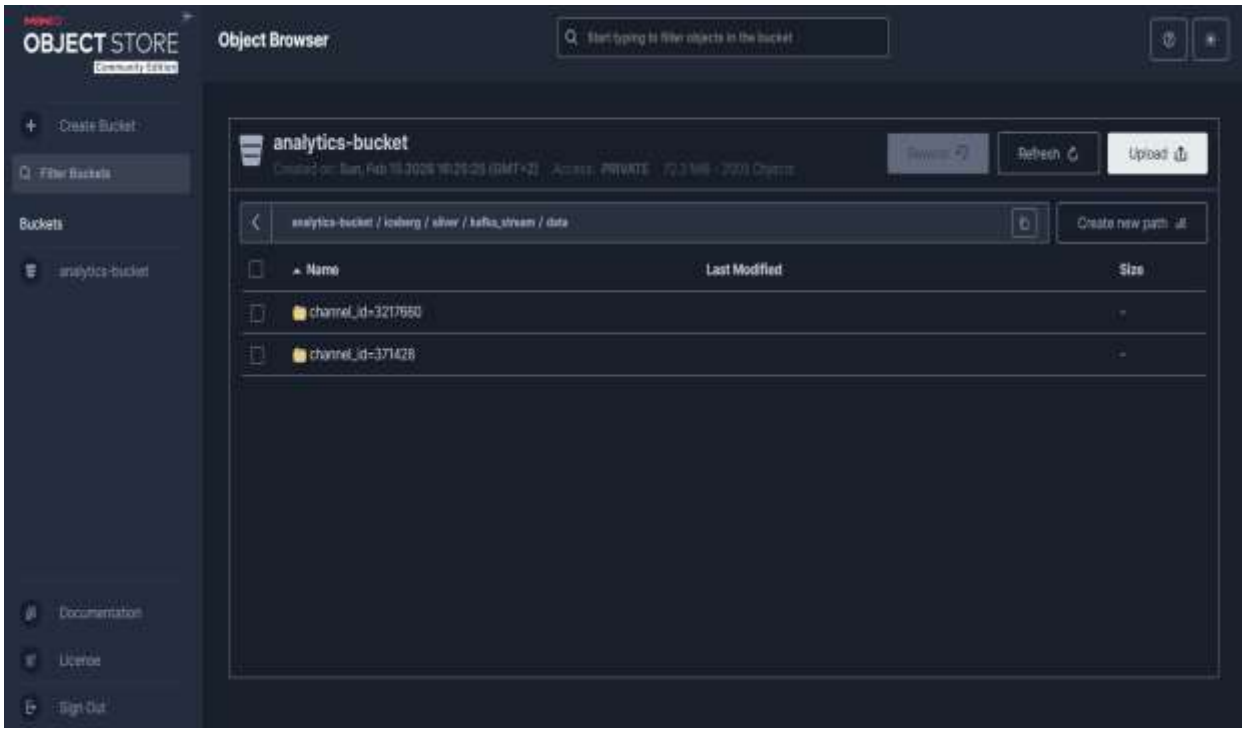

Рисунок 3.37 – Створений бакет із внутрішнім партиціюванням в MinIO сервісі

На рисунку 3.37 можемо побачити результат роботи нашої розподіленої комп’ютерної системи безпосередньо в веб-інтерфейсі MinIO. Також на рисунку зображено створений бакет в якому Apache Spark автоматично розгорнув структуру папок із застосуванням секціювання за полем «channel_id» [27]. В середині кожної директорії фізично зберігаються об’єкти у форматі Parquet, які містять оптимізовані дані за обсягом (рис. 3.38).

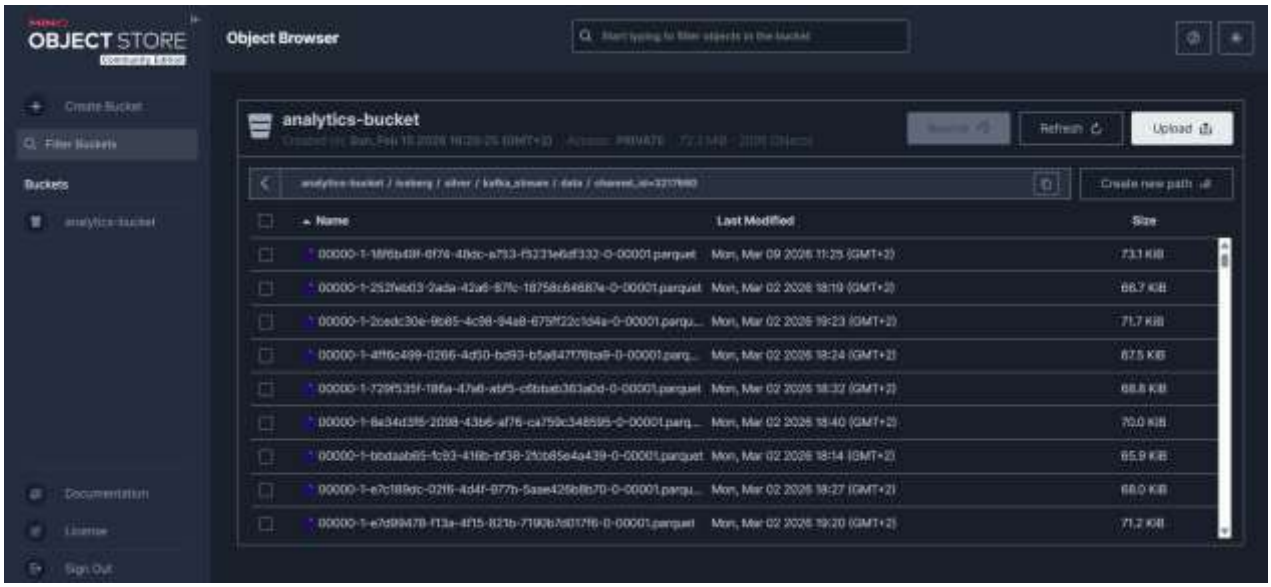

Рисунок 3.38 – Збережені дані в форматі Parquet в MinIO сервісі

Така організація сховища дозволяє нам виконувати швидкі аналітичні запити, оскільки система зчитує лише необхідні папки та фрагменти даних, що досить сильно підвищує загальну продуктивність системи.

3.6 Конфігурування шлюзу Spark Thrift Server

Для забезпечення доступу зовнішніх BI-інструментів до даних, збережених в об’єктному сховищі MinIO, розгорнемо Apache Spark Thrift Server. Цей компонент виконує роль критично важливого моста, який перетворює об’єктне сховище MinIO на повноцінне аналітичне середовище. Шлюз надає можливість виконувати SQL-запити до даних через стандартні

інтерфейси JDBC/ODBC, що робить розподілене сховище доступним для традиційних методів аналізу та візуалізації [26].

Розгорнемо сервіс в Docker Compose на базі власного образу «spark:latest» (рис. 3.39).

```
...
  spark-thrift-server:

image: spark:latest
    mem_limit: 4g
    cpus: 2.0
    container_name: spark-thrift-server
    environment:
      SPARK_NO_DAEMONIZE: "true"
    entrypoint: /bin/bash
    command: -c "/opt/spark/sbin/start-thriftserver.sh spark://spark
    master:7077
          --driver-memory 1g \
          --executor-memory 2g \
          --executor-cores 1 \
          --num-executors 1 \
          --conf spark.cores.max=1"
    ports:
      - 10000:10000
    volumes:
      - ./conf:/opt/spark/conf
      - spark-thrift-server-logs:/app/logs
    networks:
      - service-network
    depends_on:
      - spark-master
      - spark-worker
...
```

Рисунок 3.39 – Лістинг коду розгортання Apache Spark Thrift Server за допомогою Docker Compose

Виділимо контейнеру 4 ГБ оперативної пам’яті та 2 ядра процесора, що забезпечить стабільне опрацювання вхідних аналітичних запитів. Запуск виконаємо за допомогою скрипта «start-thriftserver.sh» (рис. 3.39), який підключається до існуючого Master-вузла кластера. Такий підхід дозволяє Apache Spark Thrift Server використовувати спільні обчислювальні ресурси воркерів для виконання складних обчислень безпосередньо над файлами в сховищі [25].

Конфігурацію ресурсів для виконання запитів додамо безпосередньо до команди запуску. Для драйвера виділимо 1 ГБ пам’яті, а для виконання обчислень розгорнемо Apache Spark-виконавця та виділимо йому 2 ГБ оперативної пам’яті. Окремо налаштуємо параметри паралелізму, зокрема «spark.cores.max=1» (рис. 3.39), щоб аналітична запити до об’єктного сховища не блокувала основний потік даних. Через директиву «ports» (рис. 3.39) відкриємо стандартний порт «10000», який слугує основною точкою входу для зовнішніх підключень.

Наступним кроком налаштуємо доступ до конфігурацій та метаданих. Зробимо монтування локальної директорії «./conf» у внутрішню папку «/opt/spark/conf». Це дозволяє сервісу використовувати ідентичні налаштування каталогів та S3-кінцевих точок. Завдяки цьому шлюз відображає актуальний стан даних у MinIO в реальному часі, забезпечуючи цілісність інформації для кінцевого користувача [27].

Функціонування Apache Spark Thrift Server перетворює ізольоване об’єктне сховище на відкриту аналітичну платформу. Це дозволяє реалізувати модель Data Lakehouse, де дані стають доступними для стандартних SQL-клієнтів та BI-систем, таких як Apache Superset. На рисунку 3.40 зображено системні логи в Docker Desktop, в яких описується ініціалізація HiveThriftServer2 та готовність сервісу приймати з’єднання.

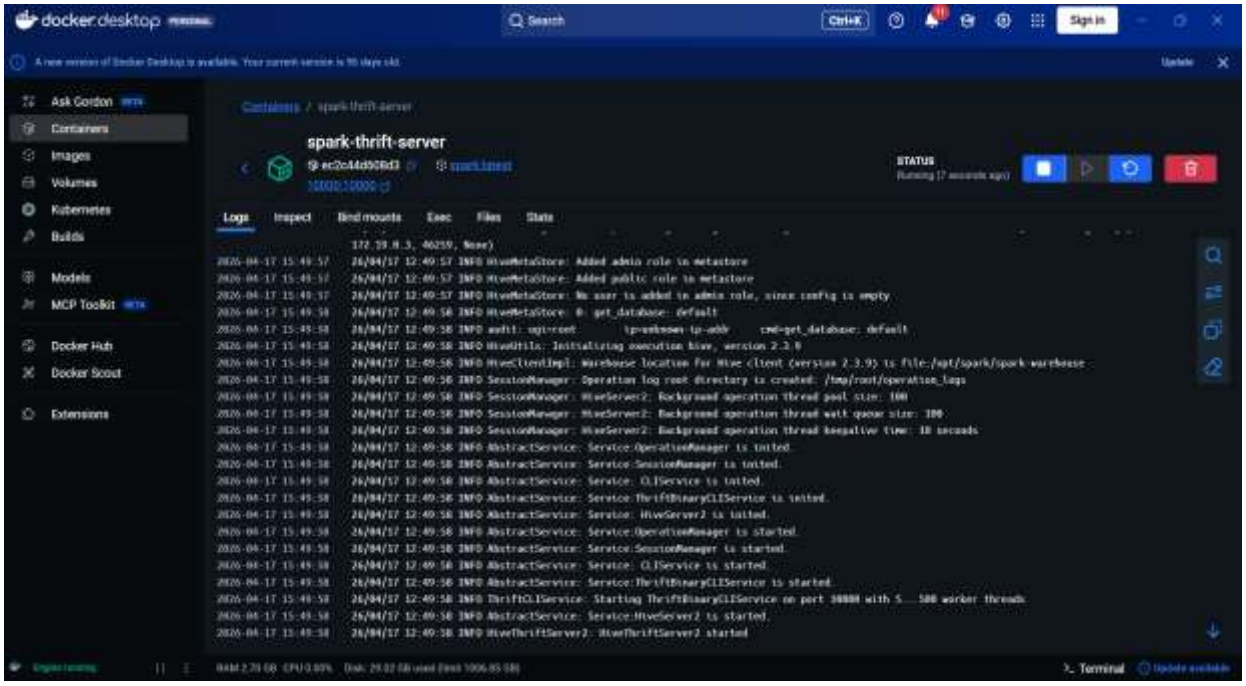

Рисунок 3.40 – Системні логи та ініціалізація Spark Thrift Server сервісу

Шоб перевірити процес підключення виконаємо перевірку шлюзу за допомогою вбудованого терміналу в Docker Desktop та утиліти Beeline. На рисунку 3.41 зображено успішне підключення до JDBC-інтерфейсу «spark-thrift-server» та виконання SQL-запитів до створеного каталогу [26].

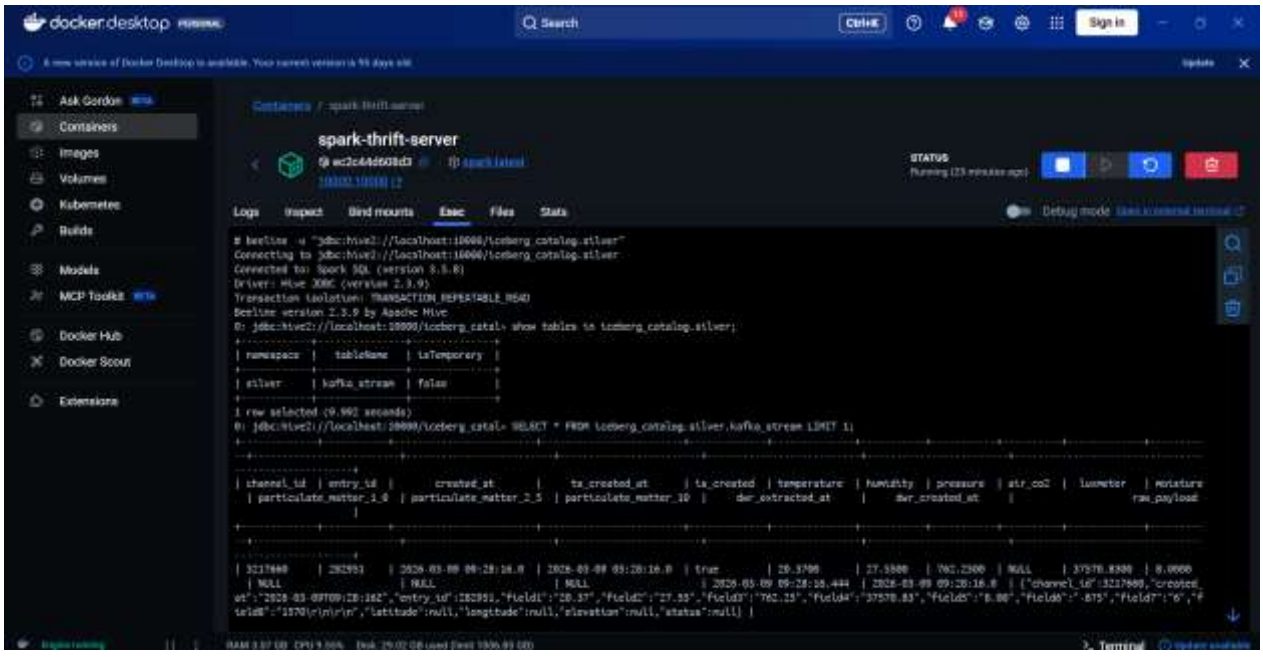

Рисунок 3.41 – Демонстрація підключення Spark Thrift Server сервісу до об’єктного сховища MinIO

Команда «show tables» підтверджує, що шлюз коректно бачить таблицю «kafka_stream» в просторі імен «silver», інтегрованому з об’єктним сховищем. Виконаємо «SELECT » запит, щоб продемонструвати доступ до структурованих IoT-даних. В терміналі відображаються заповнені поля «channel_id», показники температури, вологості та тиску, а також серіалізований початковий payload. Це підтверджує готовність обчислювального шару до роботи з BI-системами, оскільки дані вже доступні для стандартного SQL-аналізу в режимі реального часу [26].

Тепер будь-який зовнішній інструмент може виконувати аналітичні запити до збережених IoT-даних, не враховуючи складну внутрішню структуру parquet-файлів у MinIO.

3.7 Інтеграція системи візуалізації та моніторингу Apache Superset

Завершальним етапом побудови розподіленої комп’ютерної системи агрегації та аналізу даних з IoT-пристроїв виконаємо розгортання Apache Superset, що представляє собою сучасну платформу для дослідження та візуалізації даних. На цьому рівні забезпечимо кінцевому користувачу зручний графічний інтерфейс для аналізу даних, завдяки чому перетворимо структуровані записи у Data Lakehouse на інтерактивні дашборди та графіки, що працюють в реальному часі [27].

Для інтеграції Apache Superset із нашими сервісами створимо власний Docker образ на основі офіційного дистрибутива «apache/superset» (рис. 3.42).

```
FROM apache/superset:latest
USER root

RUN apt-get update && \
    apt-get install -y --no-install-recommends \
        gcc \
        build-essential \
        libsasl2-dev \
        libsasl2-2 \
        libsasl2-modules \
        libssl-dev \
        python3-dev \
    && rm -rf /var/lib/apt/lists/*
RUN . /app/.venv/bin/activate && \
    uv pip install \
        pyhive==0.7.0 \
        sasl==0.3.1 \
        thrift_sasl==0.4.3

USER superset
CMD ["/app/docker/entrypoints/run-server.sh"]
```

Рисунок 3.42 – Лістинг коду створення власного образу за допомогою Dockerfile для Apache Superset

Оскільки стандартне середовище не містить специфічних драйверів для інтеграції Apache Spark Thrift Server, в Dockerfile реалізуємо встановлення необхідних системних залежностей (рис. 3.42), таких як «libsasl2-dev» та «libssl-dev». Також встановимо спеціалізовані Python-бібліотеки «pyhive», «sasl» та «thrift_sasl» (рис. 3.42), що дозволить Apache Superset взаємодіяти з аналітичним сховищем через протокол Hive та забезпечить високу швидкість виконання SQL-запитів до нашого Data Lakehouse. Визначимо інструкцію «CMD ["/app/docker/entrypoints/run-server.sh"]» для запуску сервісу при запуску контнейра (рис. 3.42) [18].

Оркестрацію сервісу здійснимо за допомогою Docker Compose, виділивши для стабільної роботи 4 ГБ оперативної пам’яті та 2 ядра процесора (рис. 3.43).

```
...
superset:
  build:
    context: .
    dockerfile: containers/superset/Dockerfile
  image: superset:latest
  mem_limit: 4g
  cpus: 2.0
  container_name: superset
  env_file:
    - .env
  environment:
    SUPERSET_ENV: production
    SUPERSET_LOAD_EXAMPLES: "no"
    SUPERSET_USERNAME: admin
    SUPERSET_PASSWORD: admin
    SUPERSET_EMAIL: admin@example.com
  entrypoint: /bin/bash
  command: -c "superset db upgrade &&
        superset init &&
        superset fab create-admin \
        --username admin \
        --firstname Admin \
        --lastname Admin \
        --email admin@example.com \
        --password admin &&
        superset run -p 8088 --host 0.0.0.0"
  volumes:
   ./src/analytics/superset_config.py:/app/pythonpath/superset_config.py
    - ${SUPERSET_STORE_PATH}:/app/superset_home
    - superset-logs:/app/logs
  networks:
    - service-network
  ports:
    - "8088:8088"
...
```

Рисунок 3.43 – Лістинг коду розгортання Apache Superset за допомогою Docker Compose

Під час запуску контейнера налаштуємо автоматичне виконання ініціалізації бази даних метаданих, створення облікового запису адміністратора та конфігурування внутрішніх дозволів за допомогою інструкції «command» (рис. 3.43). Для збереження налаштованих дашбордів та системних логів застосуємо монтування томів за допомогою інструкції «volumes», що гарантуватиме безпечне збереження всіх створених візуалізацій навіть після повного перезапуску системи або її оновлення (рис. 3.43) [18].

Важливим елементом конфігурації є налаштування файлу «superset_config.py», який змонтуємо в внутрішню директорію сервісу (рис. 3.43). В цьому файлі пропишемо ключі безпеки та специфічні параметри автентифікації, без яких робота вебсервера в продуктивному режимі буде неможливою. Такий підхід дозволить нам відокремити конфіденційні дані від коду образу та забезпечить гнучке керування правами доступу безпосередньо через змінні середовища у файлі «.env». Також об’єднаємо Apache Superset із загальною мережею «service-network» та відкриємо порт «8088» для доступу до веб-серверу за допомогою інструкції «ports» (рис. 3.43). Такий підхід відкриває шлях для прямої взаємодії між інтерфейсом аналітика та даними у MinIO [22].

Після завершення процесу розгортання отримаємо доступ до платформи через порт «8088». На рисунку 3.44 зображено головну сторінку Apache Superset, яка виступає центральним місцем для навігації та керування аналітичними дашбордами.

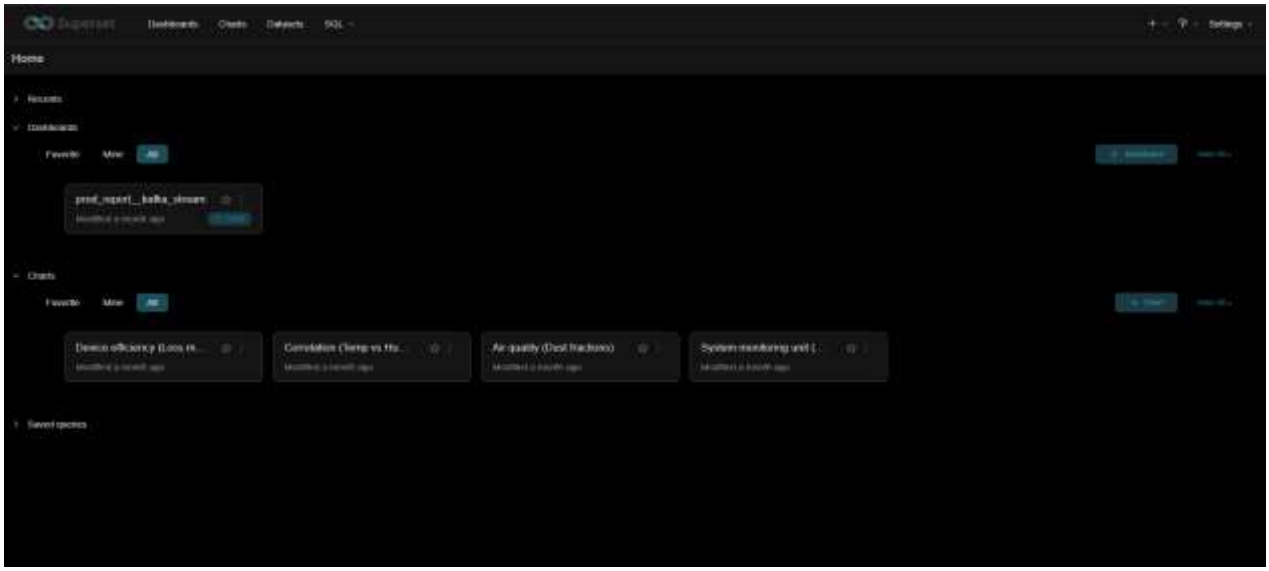

Рисунок 3.44 – Головна сторінка Apache Superset сервісу

Першим практичним кроком виконаємо підключення до джерела даних в вкладці «Database Connections», використовуючи JDBC драйвер та посилння на наш шлюз Apache Spark Thrift Server. На рисунку 3.45 зображено вікно

конфігурації підключення до бази даних, де пропишемо параметри з’єднання з хостом «spark-thrift-server» та виконаємо тестування.

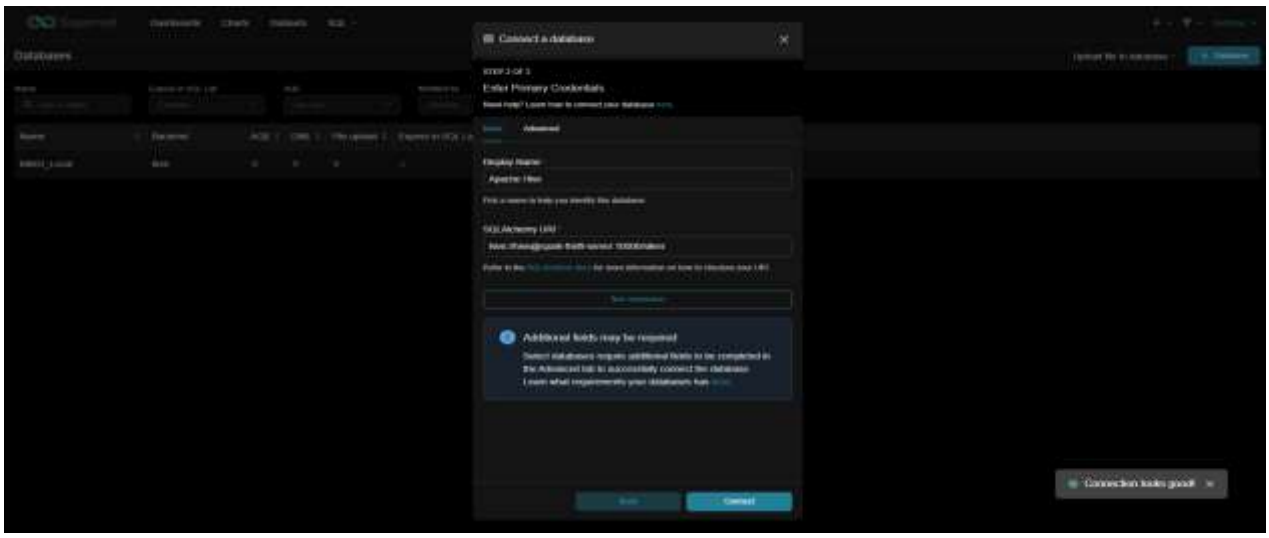

Рисунок 3.45 – Налаштування підключення до бази даних в Apache Superset сервісі

В разі успішного підключення, такий підхід дозволить нашому візуалізаційному шару отримати прямий доступ до таблиць та даних в об’єктному сховищі MinIO.

Тепер перейдемо в вкладку «Datasets» та за допомогою SQL Lab завантажемо дані з об’єктного сховища. На рисунку 3.46 зображено розділ «Datasets» в інтерфейсі Apache Superset із підключеним набором даних «trx_kafka_stream» [26].

Рисунок 3.46 – Створений «trx_kafka_stream» датасет в Apache Superset сервісі

Тепер перейдемо в вкладку «Dashboards» та натиснемо «+Dashboard» для створення аналітичних дашбордів (рис. 3.47).

Рисунок 3.47 – Створення дашборда в Apache Superset сервісі

На цьому етапі реалізуємо безпосередню розробку візуальних компонентів системи в робочому просторі. Для цього використаємо вкладку «+ Create a new chart» для побудови відповідних аналітичних представлень (рис. 3.48).

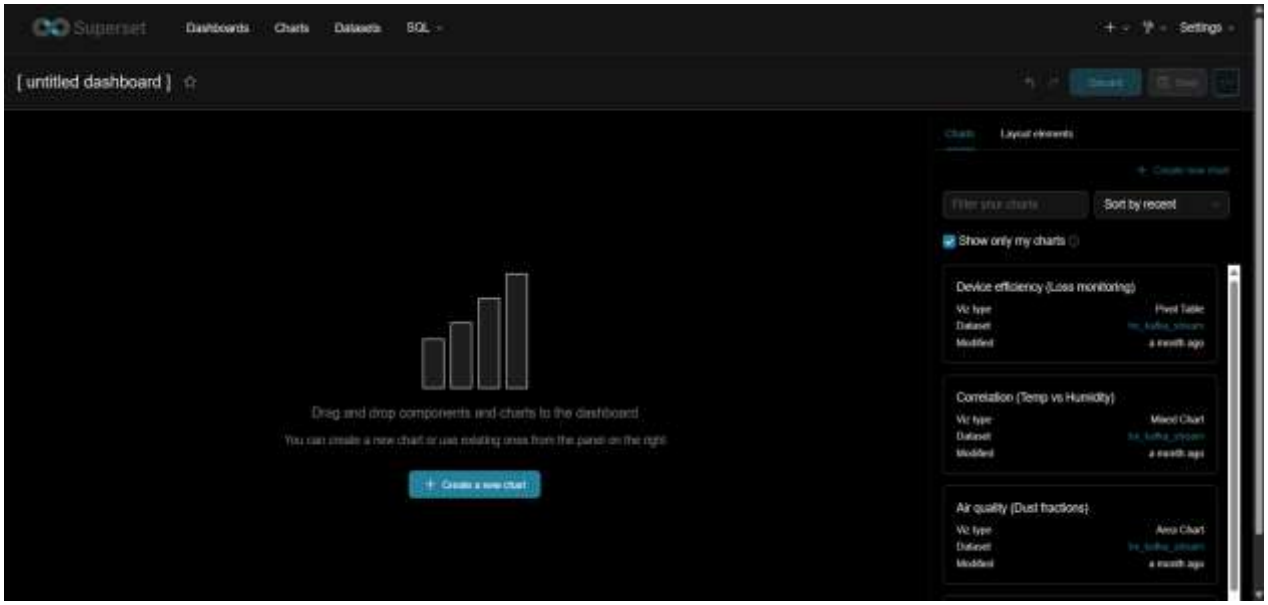

Рисунок 3.48 – Створення чарту в Apache Superset сервісі

Перейдемо до створення самого аналітичного вікна відповідно до наших аналітичних завдань. Оберемо відповідний тип графіка та згрупуємо ключові

показники ефективності IoT-пристроїв, включаючи динаміку температури, рівень вологості та показники атмосферного тиску (рис. 3.49) [17].

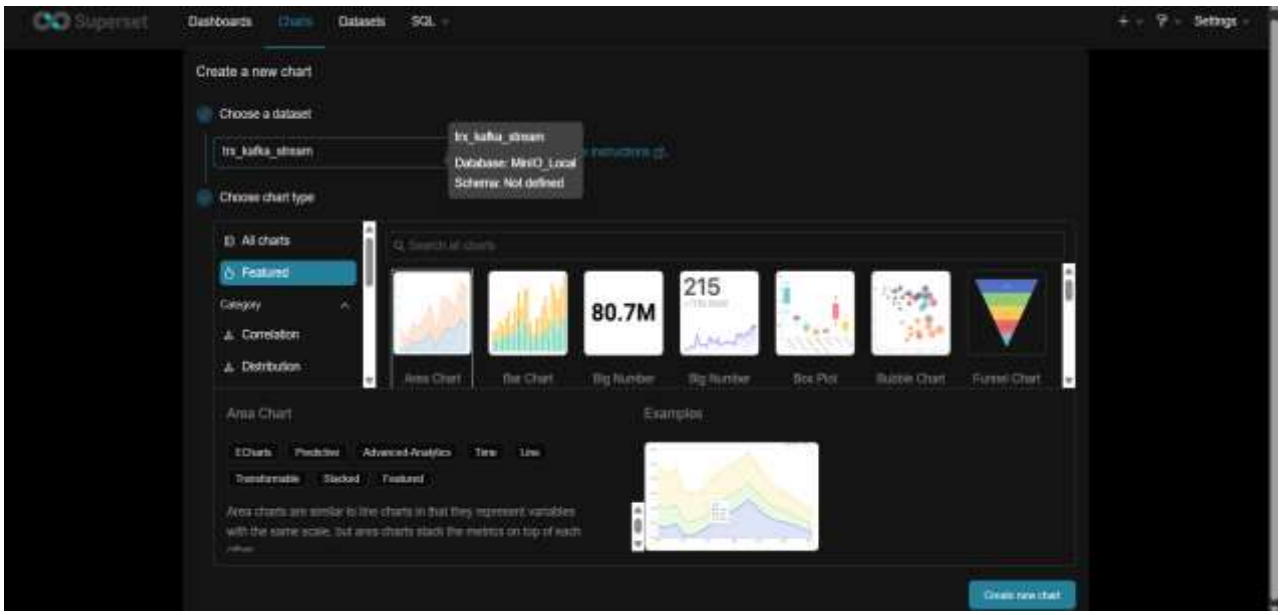

Рисунок 3.49 – Вибір аналітичного відображення в Apache Superset сервісі

Завдяки реалізованій архітектурі Data Lakehouse забезпечимо автоматичне оновлення цих графіків. Як тільки новий мікро-батч даних потрапить у сховище, зміни миттєво відобразяться на моніторі, гарантуючи повноцінний контроль над об’єктами в режимі реального часу.

На рисунку 3.50 представлено результати інтерактивної аналітики, яка дозволяє здійснювати комплексний моніторинг стану IoT-інфраструктури в режимі реального часу.

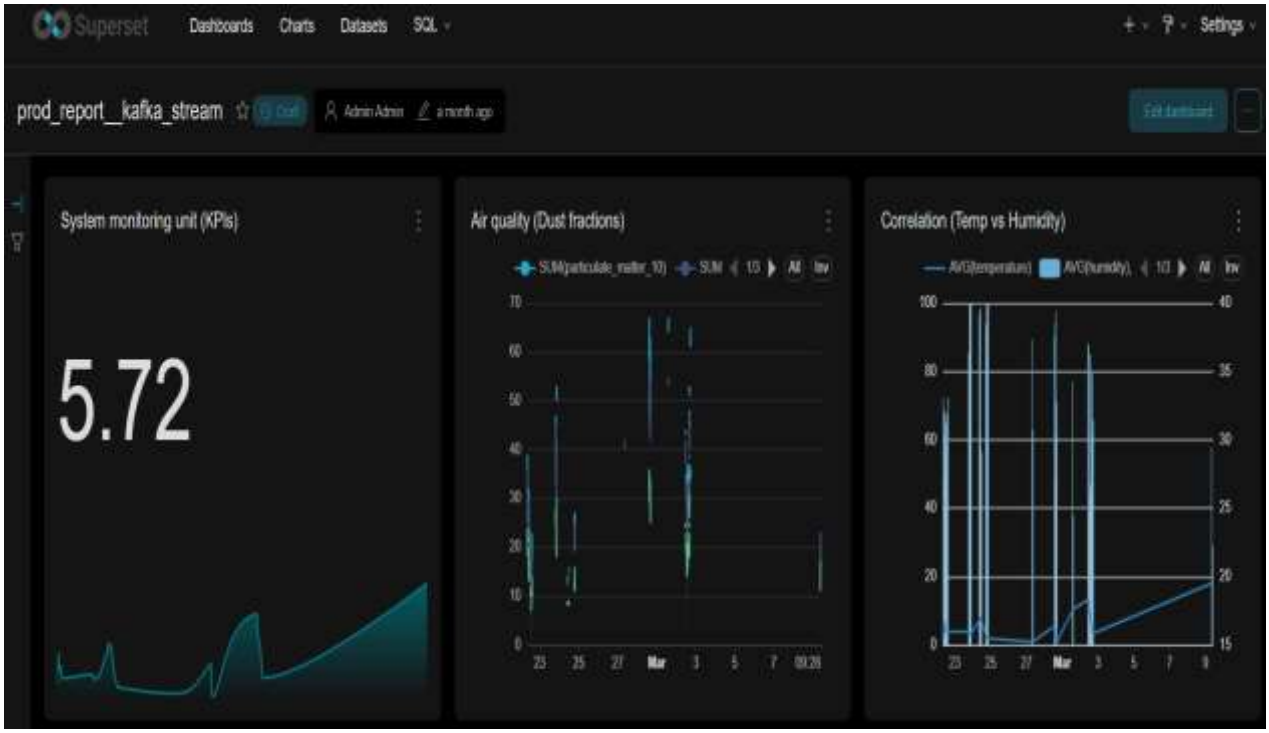

Рисунок 3.50 – Відображення створених аналітик в Apache Superset сервісі

За допомогою аналітики «System monitoring unit (KPIs)» виконується миттєва оцінка ключових показників ефективності (KPI) системи, що дозволяє оператору швидко визначити загальний стан мережі без детального вивчення кожного окремого датчика. Аналіз графіків «Air quality (Dust fractors)» та «Correlation (Temp vs Humidity)» забезпечує виявлення неявних залежностей між параметрами навколишнього середовища, зокрема впливу вологості на концентрацію пилу. Дослідження цих закономірностей дозволяє прогнозувати потенційні ризики для обладнання або погіршення умов експлуатації об’єктів, що є критично важливим для обслуговування всієї системи [23].

На рисунку 3.51 зображено зведену таблицю «Device efficiency (Loss monitoring)», яка має особливе значення для забезпечення цілісності даних в нашому Data Lakehouse [27].

Рисунок 3.51 – Відображення «Device efficiency (Loss monitoring)» аналітики в Apache Superset сервісі

Відповідна аналітична панель орієнтована на системних адміністраторів та інженерів мережі, оскільки вона відображає інтенсивність надходження повідомлень у розрізі кожної години для кожного окремого пристрою. Така деталізація дозволяє чітко розрізняти стабільну роботу сенсорів від періодів аномальної активності або повної втрати зв’язку, що допомагає локалізувати проблеми у конкретних вузлах мережі.

В результаті такий інструментарій надає можливість проводити глибокий історичний аналіз продуктивності, дозволяючи мінімізувати втрати даних та підвищити загальну надійність розробленого рішення.

РОЗДІЛ 4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

4.1 Безпека праці персоналу та експлуатації серверного обладнання

Серверні приміщення для обробки великих масивів даних є критичними компонентами сучасної інформаційної інфраструктури. Вони підтримують роботу обчислювальних кластерів, де безперебійний доступ до даних має вирішальне значення. Однак експлуатація такого високотехнологічного обладнання в умовах дата-центрів пов’язана з ризиками, які можуть поставити під загрозу як персонал, так і саму інфраструктуру. Для забезпечення безпеки праці та безперебійної роботи серверів необхідно впроваджувати комплексні заходи, які охоплюють пожежну, електричну, кліматичну безпеку, а також захист від фізичних факторів впливу та несанкціонованого доступу [32].

Однім із головних напрямків безпеки при роботі з апаратними комплексами є протипожежний захист, оскільки замикання або перегрів обчислювальних вузлів може завдати значних збитків і призвести до втрати обладнання та даних.

Для забезпечення протипожежної безпеки в інфраструктурі дата-центру відповідно до загальних вимог нормативних актів впроваджується комплексний підхід, що включає такі заходи [31]:

– встановлення систем раннього виявлення задимлення та автоматичних систем газового пожежогасіння (на основі інертних газів або хладонів), які не пошкоджують чутливу електроніку;

– розробка чітких інструкцій та планів евакуації з технологічних зон, проведення регулярних інструктажів і навчань для ІТ-спеціалістів щодо дій у разі пожежі;

– облаштування герметичних серверних залів протипожежними перегородками та дверима для локалізації вогню;

– використання негорючих матеріалів при прокладанні силових і мережевих кабельних ліній.

Електробезпека під час обслуговування серверів має першорядне значення, оскільки тривала робота під навантаженням підвищує ризики ураження струмом та виникнення аварійних ситуацій [29].

Для забезпечення належного рівня електробезпеки й захисту інженерів впроваджуються наступні заходи:

– обов’язкове заземлення та занулення всіх металевих частин серверних стійок, шаф і корпусів пристроїв відповідно до нормативних правил;

– використання потужних джерел безперебійного живлення (UPS) для захисту від перепадів напруги та забезпечення автономної роботи обладнання під час вимкнень;

– регулярне проведення технічних оглядів, вимірювання опору ізоляції та планова перевірка електричних щитів;

– чітке структурування й маркування кабельних потоків усередині комутаційних шаф для зниження ризиків аварій.

Підтримання оптимального мікроклімату (комплексного поєднання температури, вологості, швидкості руху повітря та теплового випромінювання) в технологічних зонах дата-центру є ключовим для запобігання апаратним збоям та збереження нормального теплового балансу організму працюючих. Відповідно до ДСН 3.3.6.042-99 «Санітарні норми мікроклімату виробничих приміщень» [33], для категорії легких робіт Іа в робочій зоні встановлено оптимальні параметри, які в холодний період року становлять: температура повітря 22–24 °С, відносна вологість 60–40 % та швидкість руху повітря не більше 0,1 м/с, а в теплий період року – температура повітря 23–25 °С при аналогічних показниках вологості та швидкості повітряного потоку.Порушення цього балансу при температурі повітря понад

30 °C призводить до стрімкого падіння працездатності людини, а зневоднення організму на 6 % викликає розлади розумової діяльності та зниження гостроти зору. Робоче місце дата-інженера за рівнем енерговитрат (до 120 ккал/год або 139 Вт) відноситься до категорії легких фізичних робіт Іа, що згідно з ДСН 3.3.6.042-99 [33] вимагає забезпечення оптимальних умов праці.

Основні заходи з клімат-контролю та інструментальної нормалізації робочого середовища включають:

– розгортання автоматизованих систем кондиціювання та вентиляції повітря [34] для цілодобового підтримання температури в межах 22…24 °C та відносної вологості на рівні 40…60 %;

– впровадження методів розподілу «холодних» та «гарячих» коридорів, що дозволяє утримувати швидкість переміщення повітряних потоків у межах не більше ніж 0,1 м/с та захищає персонал від відчутного руху повітря;

– контроль інтенсивності теплового опромінення працюючих від нагрітих поверхонь обладнання та освітлювальних приладів, яка у разі опромінення понад 50%% поверхні тіла не повинна перевищувати ліміт у 35 Вт/м² (і до 100 Вт/м² при опроміненні менше ніж 25 %% тіла);

– моніторинг та регулювання рівня аероіонізації в залах ЕОМ відповідно до ДСН, де мінімально необхідна кількість негативних іонів становить 600 іонів/см³, позитивних – 400 іонів/см³, а їх оптимальний діапазон коливається в межах 3000…5000 іонів/см³ (із жорстким обмеженням максимуму в 50000 іонів/см³);

– проведення планового інструментального контролю параметрів мікроклімату за допомогою психрометрів Ассмана (зі швидкістю аспірації повітря 2 м/с), гігрометрів та анемометрів не менше 3 разів за робочу зміну. Вимірювання здійснюються у робочій зоні (просторі заввишки до 2 м над рівнем підлоги) на висоті 1,0 м від підлоги для робіт у положенні сидячи та на висоті 1,5 м для робіт у положенні стоячи.

Захист від несанкціонованого фізичного доступу до серверного обладнання є критичним для запобігання викраденню даних, вандалізму та іншим загрозам. Для цього вхідні двері дата-центру обладнуються електронними системами контролю доступу (СКУД) із використанням іменних карток або біометричних сканерів. Ведення автоматичних журналів обліку дозволяє точно відстежувати та фіксувати час перебування відвідувачів у технологічній зоні. Розгортання систем відеоспостереження високої роздільної здатності забезпечує візуальний контроль усіх закритих зон, а збереження відеоархівів допомагає проводити аналіз інцидентів. Залучення відповідального персоналу для регулярного обходу та огляду кріплень обладнання забезпечує додатковий рівень захисту.

Шум та вібрація, що генеруються вентиляторами охолодження й жорсткими дисками, можуть негативно впливати на роботу обладнання та викликати передчасну втому й послаблення реакції персоналу. Основні заходи щодо їх зменшення включають використання спеціальних шумопоглинаючих матеріалів у конструкціях стін та підлоги приміщень дата-центру, а також встановлення серверних шаф на антивібраційні опори чи демпферні підставки для мінімізації коливань. Регулярний моніторинг рівнів акустичного тиску відповідно до ДСН 3.3.6.037-99 та рівнів вібрації за ДСН 3.3.6.039-99, аналіз їхнього впливу на працівників та впровадження коригуючих засобів при перевищенні допустимих норм дозволяють підтримувати належні умови праці. Експлуатація та модернізація обчислювальної інфраструктури неминуче призводять до накопичення відходів, управління якими є важливим аспекстом екологічної безпеки.

Основні заходи включають організацію роздільного збору та передачу спеціалізованим підприємствам відправатих серверів, акумуляторів UPS та кабельної продукції для їх подальшої утилізації чи переробки. Впровадження внутрішніх політик щодо використання енергоефективного обладнання також сприяє зменшенню кількості електронного сміття. Проведення навчань для IT-

персоналу щодо належного поводження з відходами підвищує обізнаність та відповідальність працівників.

4.2 Забезпечення електробезпеки захисту при експлуатації розподіленої серверної інфраструктури

Функціонування розподіленої серверної інфраструктури критично залежить від стабільності електропостачання та захищеності від вогневих ризиків. Постійна робота апаратних комплексів під високим обчислювальним навантаженням створює підвищені вимоги до надійності силових мереж та систем аварійного захисту. Будь-який збій у системі живлення або локальне займання комутаційного вузла можуть призвести до миттєвої зупинки конвеєрів обробки великих даних, незворотної втрати інформації та загрози здоров'ю обслуговуючого персоналу.

Комплекс заходів із забезпечення електробезпеки розподіленої інфраструктури базується на нормативних вимогах Правил улаштування електроустановок [29] та включає такі основні рішення:

– Захисне заземлення та занулення. Воно передбачає з'єднання всіх металевих частин серверних шаф, стійок, корпусів обчислювальних вузлів та систем кондиціювання з контуром заземлення. Згідно з ПУЕ [29], для мереж із лінійною напругою 380 В трифазного струму (або 220 В однофазного джерела) опір заземлюючого пристрою, до якого під'єднані нейтралі трансформаторів, у будь-яку пору року має бути не більше 4 Ом. Для запобігання ураженню кроковою напругою (різницею потенціалів між двома точками в зоні розтікання струму на відстані кроку 0,8 м) у разі потенційного замикання на землю, контроль параметрів контуру проводиться щонайменше один раз на рік приладами типу М416 у період найменшої провідності ґрунту (межі вимірювання від 0,1 до 1000 Ом) з урахуванням питомого електричного опору ґрунту.

– Гарантоване та безперебійне живлення. Воно реалізується шляхом інтеграції промислових джерел безперебійного живлення (UPS) з подвійним перетворенням напруги (On-Line), що захищає апаратуру від імпульсних перешкод та забезпечує стабільну роботу інфраструктури до 1000 В під час автоматичного або ручного переключення на резервні лінії.

– Автоматичний захист та моніторинг. Він забезпечується встановленням автоматичних вимикачів, пристроїв захисного вимкнення (ПЗВ) та реле контролю фаз у силових щитах для запобігання перевантаженням і коротким замиканням.

– Регулярне технічне обслуговування й діагностика ізоляції. Вона проводиться мегомметром (наприклад, типу М1101 на напругу до 1000 В зі швидкістю обертання ручки генератора 120 об/хв) при повністю вимкненій напрузі. Оскільки приміщення дата-центрів класифікуються як приміщення з підвищеною небезпекою, перевірка проводиться не менше двох разів на рік [32]. Опір ізоляції силових та комутаційних кабельних ліній на кожну фазу повинен становити не менше 0,5 МОм. При фіксації зниження опору ізоляції мережі в експлуатації до 30% від її номінального значення, лінія негайно виводиться з конвеєра обробки для проведення ремонтних робіт.

Організація протипожежного захисту технологічних приміщень із серверним обладнанням враховує неможливість використання класичних засобів гасіння (води, піни) та відповідно до чинних Правил пожежної безпеки в Україні [31] складається з наступних елементів:

– Автоматичних систем газового пожежогасіння. Вони базуються на застосуванні вогнегасних речовин на основі інертних газів, які ліквідують вогонь шляхом витіснення кисню або хімічного гальмування реакції горіння без утворення конденсату та пошкодження плат чи дискових масивів [31].

– Раннє виявлення через витяжні системи. Воно полягає в встановленні високочутливих лазерних сповіщувачів, які аналізують повітря на наявність мікрочастинок диму ще до появи відкритого полум’я чи різкого підвищення температури.

– Конструктивна ізоляція приміщень. Вона вимагає використання протипожежних дверей, стін та герметичних кабельних проходок із межею вогнестійкості не менше REI 60 відповідно до державних будівельних норм [30] для надійної локалізації можливого загорання.

– Початкові засоби та евакуація. Вони передбачають оснащення приміщень ручними вуглекислотними вогнегасниками типу ОУ-3 чи ОУ-5, розробку чітких схем евакуації та регулярне навчання персоналу алгоритмам дій при спрацюванні тривоги.

Поєднання надійного електротехнічного захисту з автоматизованими системами пожежогасіння дозволяє мінімізувати ризики аварійних ситуацій.

ВИСНОВКИ

В ході виконання кваліфікаційної роботи було розроблено архітектуру розподілену та реалізовано розподілену комп'ютерну систему збору, агрегування та аналізу даних з IoT-пристроїв. Використання сучасних засобів інженерії великих даних дозволило створити гнучку архітектуру, яка поєднує в собі переваги традиційних сховищ та аналітичних платформ.

В результаті проведеного дослідження було вирішено наступні ключові завдання:

– проєктування та розгортання інфраструктури на базі технологій контейнеризації Docker дозволило створити ізольоване середовище для всіх компонентів системи, що забезпечило стабільну роботу брокера повідомлень Apache Kafka, обчислювального кластера Apache Spark та об'єктного сховища MinIO;

– реалізація архітектури Data Lakehouse через впровадження технології Apache Iceberg надала можливість організувати зберігання даних із підтримкою транзакційності, еволюції схем та ефективного секціювання, перетворивши сховище об'єктів на високоефективний аналітичний рівень;

– автоматизація обробки даних за допомогою Spark Structured Streaming забезпечила оперативне очищення, типізацію та завантаження телеметрії у сховище, а налаштування Spark Thrift Server відкрило доступ до цих даних через стандартні SQL-інтерфейси для зовнішніх BI-інструментів;

– візуалізація та моніторинг на базі Apache Superset дозволили розгорнути інтерактивну панель для відображення стану фізичних датчиків у реальному часі, що надало можливість оперативно виявляти аномалії та аналізувати стабільність надходження даних з кожного пристрою.

Практичне значення отриманих результатів полягає у створенні готового до експлуатації прототипу системи, що здатна обробляти великі потоки даних із мінімальною затримкою. Розроблена архітектура Data Lakehouse демонструє високу надійність та відмовостійкість, забезпечуючи

цілісність даних навіть в разі мережевих збоїв. Отриманий інструментарій аналізу ефективності пристроїв надає можливість проводити глибоку діагностику IoT-мережі, що дозволяє оптимізувати витрати на обслуговування інфраструктури та підвищити якість прийняття рішень на основі реальних даних.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Жаровський Р.О., Луцик Н.С., Осухівська Г.М., Паламар А.М., Тиш Є.В. Методичні вказівки до виконання кваліфікаційної роботи бакалавра для здобувачів першого (бакалаврського) рівня вищої освіти за спеціальністю 123 «Комп'ютерна інженерія» усіх форм навчання. Тернопіль: ТНТУ, 2024. 39 с.

2. Луцків А., Лупенко С., Пасічник В. Паралельні та розподільнені обчислення. Навчальний посібник. Львів: Видавництво «Магнолія 2006», 2024. 566 с.

3. Микитишин А.Г., Митник М.М., Стухляк П.Д., Пасічник В.В. Комп'ютерні мережі. Книга 1. Львів: «Магнолія 2006», 2024. 256 с.

4. Микитишин А.Г., Митник М.М., Стухляк П.Д., Пасічник В.В. Комп'ютерні мережі. Книга 2. Львів: «Магнолія 2006», 2024. 328 с.

5. Ковальський В., Тиш Є. Прогнозування мережевих аномалій у хмарному середовищі з використанням генеративних алгоритмів. Матеріали XIII науково-технічної конференції «Інформаційні моделі, системи та технології». Тернопіль: ТНТУ, 2025. С.116.

6. Yatsyshyn V., Pastukh O., Palamar A., Zharovskyy R. Technology of relational database management systems performance evaluation during computer systems design. Scientific Journal of TNTU, Ternopil, Ukraine, 2023. Vol. 109, No 1. P. 54–65.

7. Yatsyshyn V., Pastukh O., Zharovskyi R., Shabliy N.Software tool for productivity metrics measure of relational database management system. Mathematical Modeling. No 1 (48). 2023. P. 7-17.

8. Дячук, О.А.; Жаровський, Р.О Управління потоком за критеріями доступності. Матеріали XI науково-технічна конференція Тернопільського національного технічного університету імені Івана Пулюя «Інформаційні моделі системи та технології» (13-14 грудня 2023 року). Тернопіль: ТНТУ. 2023. С. 151.

9. Ковтун Н., Жаровський Р. Аналіз засобів протидії вторгненням і атакам на комп'ютерні системи. Матеріали XII Міжнародна науково-технічна конференція молодих учених та студентів «Актуальні задачі сучасних технологій» (6-7 грудня 2023 року). Тернопіль: ТНТУ. 2023. С. 453-454.

10. Микитишин А. Г., Митник М. М., Стухляк П. Д. Телекомунікаційні системи та мережі. Тернопіль: Тернопільський національний технічний університет імені Івана Пулюя, 2017. 384 с.

11. Свергун С., Жаровський Р. Тестування програмного забезпечення побудованого на мікросервісній архітектурі. Матеріали X науково-технічної конференції Тернопільського національного технічного університету імені Івана Пулюя «Інформаційні моделі системи та технології» (7-8 грудня 2022 року). Тернопіль: ТНТУ. 2022. С. 92.

12. Lutskiv A. Adaptable Text Corpus Development for Specific Linguistic Research / Andriy Lutskiv, Nataliya Popovych // International Scientific-Practical Conference «Problems of Infocommunications. Science and Technology» (October 8-11, 2019), 2019. - C.217-223.

13. Lutskiv A. Big Data Approach to Developing Adaptable Corpus Tools /Andriy Lutskiv, Nataliya Popovych// Computational Linguistics and Intelligent Systems. Proc. 4thInt. Conf. COLINS 2020. Volume I:Workshop. Lviv, Ukraine, April23-24, 2020, CEUR-WS.org, online. pp.374-395. URL: http://ceur-ws.org/Vol-2604/.

14. Lutskiv A. Big data-based approach to automated linguistic analysis effectiveness.//A. Lutskiv, N. Popovych/ IEEE Third International Conference on Data Stream Mining & Processing August 21-25, 2020, Lviv, Ukraine pp.438–443. DSMP 2020.

15. Lutskiv A. Corpus-Based Translation Automation in Adaptable Corpus Translation Module /Andriy Lutskiv, Roman Lutsyshyn// Computational Linguistics and Intelligent Systems. Proc. 5th Int. Conf. COLINS 2021. Volume I: Workshop. Lviv, Ukraine, April 22-23, 2021, CEUR-WS.org, online. pp.374-395. URL: http://ceur-ws.org/Vol-2604/.

16. Yatsyshyn V. A Risks management method based on the quality requirements communication method in agile approaches / Vasyl Yatsyshyn, Oleh Pastukh, Andriy Lutskiv, Viktor Tsymbalistyy, Nataliia Martsenko //Information Technologies: Theoretical and Applied Problems 2022 (ITTAP 2022), Ternopil, Ukraine, November 22-24, 2022. pp.1-10.

17. ThingSpeak data collection documentation : веб-сайт. URL: https://www.mathworks.com/help/thingspeak/ (дата звернення: 10.03.2026 р.).

18. Docker Desktop for development: веб-сайт. URL: https://docs.docker.com/ (дата звернення: 13.03.2026 р.).

19. Building a Real-Time Data Pipeline with Apache Airflow, Kafka, Spark, and Cassandra: веб-сайт. URL: https://medium.com/@jushijun/building-a-real-time-data-pipeline-with-apache-airflow-kafka-spark-and-cassandra-be4ee5be8843 (дата звернення: 20.03.2026 р.).

20. Setting Up a Kafka Cluster Using Docker Compose(Kraft Mode): A Step-by-Step Guide: веб-сайт. URL: https://medium.com/@darshak.kachchhi/setting-up-a-kafka-cluster-using-docker-compose-a-step-by-step-guide-a1ee5972b122 (дата звернення: 28.03.2026 р.).

21. Setting Up Kafka Connect and Configuring MirrorMaker 2 for Replication: веб-сайт. URL: https://medium.com/@darshak.kachchhi/setting-up-kafka-connect-cluster-and-configuring-mirrormaker-2-for-replication-864123395258 (дата звернення: 05.04.2026 р.).

22. Setting up AWS S3 (Minio) and Spark using Docker Compose: веб-сайт. URL: https://medium.com/@dkalouris/setting-up-aws-s3-minio-and-spark-using-docker-compose-6a22ef26c6b0 (дата звернення: 10.04.2026 р.).

23. Optimizing PySpark Jobs: Best Practices and Techniques: веб-сайт. URL: https://medium.com/@devendra631995/optimizing-pyspark-jobs-best-practices-and-techniques-7308d2a99071 (дата звернення: 20.04.2026 р.).

24. A Deep Dive into Medallion Architecture: Key Principles and Practical Implementation: веб-сайт. URL: https://medium.com/@devendra631995/a-deep-

dive-into-medallion-architecture-key-principles-and-practical-implementation-fd9f85be02f5 (дата звернення: 25.04.2026 р.).

25. Apache Spark Configurations Precedence: веб-сайт. URL: https://bigdataenthusiast.medium.com/spark-configurations-precedence-2014535ea5be (дата звернення: 29.04.2026 р.).

26. Apache Spark SQL CLI documentation: веб-сайт. URL: https://spark.apache.org/docs/latest/sql-distributed-sql-engine-spark-sql-cli.html (дата звернення: 03.05.2026 р.).

27. Apache Iceberg documentation: веб-сайт. URL: https://spark.apache.org/docs/latest/sql-distributed-sql-engine-spark-sql-cli.html (дата звернення: 10.05.2026 р.).

28. Bash documentation: веб-сайт. URL: https://www.w3schools.com/bash/index.php (дата звернення: 13.05.2026 р.).

29. Правила улаштування електроустановок. ПУЭ. веб-сайт. URL: https://zakon.rada.gov.ua/rada/show/v0476732-17#Text (дата звернення: 28.05.2026 р.).

30. ДБН В.1.1-7-2016 «Пожежна безпека об’єктів будівництва». веб-сайт. URL: https://e-construction.gov.ua/laws_detail/3080743763845318619 (дата звернення: 28.05.2026 р.).

31. Правила пожежної безпеки в Україні. (від 30.12.2014 р.. №1417). веб-сайт. URL: https://zakon.rada.gov.ua/laws/show/z0252-15#Text (дата звернення: 28.05.2026 р.).

32. Гандзюк М.П., Желібо Є.П., Халімовський М.О. Основи охорони праці. – К.: Каравела, 2008. – 384 с.

33. ДСН 3.3.6.042-99 «Санітарні норми мікроклімату виробничих приміщень».

34. ДБН В.2.5-67:2013 «Опалення, вентиляція та кондиціонування».

Додаток А

Технічне завдання

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп’ютерно-інформаційних систем і програмної інженерії

Кафедра комп’ютерних систем та мереж

“Затверджую”

Завідувач кафедри КС

_____ Осухівська Г.М.

“ 2 ” лютого 2026 р.

РОЗПОДІЛЕНА КОМП’ЮТЕРНА СИСТЕМА АГРЕГАЦІЇ ТА АНАЛІЗУ ДАНИХ З ІОТ-ПРИСТРОЇВ

ТЕХНІЧНЕ ЗАВДАННЯ

на _13_ листках

Вид робіт: Кваліфікаційна робота

На здобуття освітнього ступеня «Бакалавр»

Спеціальність 123 «Комп’ютерна інженерія»

«УЗГОДЖЕНО»	«ВИКОНАВЕЦЬ»
Керівник кваліфікаційної роботи	Студент групи СІ-41
_____ к.т.н., доц. Луцків А.М.	_____ Баландюк Ю.Р.
“ 2 ” лютого 2026 р.	“ 2 ” лютого 2026 р.

Тернопіль 2026

1 Загальні відомості

1.1 Повна назва та її умовне позначення

Повна назва теми кваліфікаційної роботи: «Розподілена комп’ютерна система агрегації та аналізу даних з IoT-пристроїв».

Умовне позначення кваліфікаційної роботи: КС КРБ 123.142.00.00

1.2 Виконавець

Студент групи СІ-41, факультету комп’ютерно-інформаційних систем і програмної інженерії, кафедри комп’ютерних систем та мереж, Тернопільського національного технічного університету імені Івана Пулюя, Баландюк Ю.Р.

1.3 Підстава для виконання роботи

Підставою для виконання кваліфікаційної роботи є наказ по університету (№4/9-188 від 24.04.2026 р.)

1.4 Планові терміни початку та завершення роботи

Плановий термін початку виконання кваліфікаційної роботи – 26.01.2026 р.

Плановий термін завершення виконання кваліфікаційної роботи – 21.06.2026 р.

1.5 Порядок оформлення та пред’явлення результатів роботи

Порядок оформлення пояснювальної записки та графічного матеріалу здійснюється у відповідності до чинних норм та правил ISO, ЕСКД, ЕСПД та ДСТУ.

Пред’явлення проміжних результатів роботи з виконання кваліфікаційної роботи здійснюється у відповідності до графіку, затвердженого керівником роботи. Попередній захист кваліфікаційної роботи відбувається при готовності роботи – наявності пояснювальної записки та графічного матеріалу.

Пред’явлення результатів кваліфікаційної роботи відбувається шляхом захисту на відповідному засіданні ЕК, ілюстрацією основних досягнень за допомогою графічного матеріалу.

2 Призначення і цілі створення системи

2.1 Призначення системи

Розроблена система призначена для високоефективного збору, централізованої агрегації, обробки та довгострокового зберігання телеметричних даних з IoT-пристроїв у режимі реального часу. Основне призначення платформи полягає в оптимізації процесів інгерсії даних через брокер повідомлень та автоматизації їх аналізу за допомогою обчислювального кластера. Шляхом впровадження архітектури Data Lakehouse система забезпечує надійне збереження великих обсягів інформації із можливістю виконання швидких аналітичних SQL-запитів. Також система призначена для надання кінцевим користувачам інструментів інтерактивної візуалізації та оперативного моніторингу технологічних параметрів, стабільності каналів зв’язку та загальної ефективності роботи підключених пристроїв. Завдяки контейнеризації всіх компонентів, система оптимізована для гнучкого масштабування, локального керування інфраструктурою та забезпечення повної незалежності від зовнішніх хмарних провайдерів.

2.2 Мета створення системи

Мета створення системи «Розподіленої комп’ютерної системи агрегації та аналізу даних з IoT-присроїв» полягає в програмній реалізація та впроваджені високопродуктивної, горизонтально масштабованої й економічно ефективної інфраструктури на основі архітектури Data Lakehouse, що забезпечує низьку затримку обробки, транзакційне зберігання та багаторівневий аналіз потоків даних від розподіленої мережі IoT-пристроїв у реальному часі.

Основні завдання, які потрібно виконати для досягнення поставленої мети:

– спроєктувати загальну структуру та функціональну схему розподіленої комп’ютерної системи агрегації та аналізу даних з IoT-пристроїв;

– визначити чіткі вимоги до апаратної та програмної частини, такі як кількість обчислювальних ресурсів для Spark-виконавців, обсяг RAM для Kafka-брокерів, параметри дискового простору для MinIO сервісу;

– вибрати та обґрунтовати інструменти для реалізації концепції Data Lakehouse (S3-сумісне сховище, Apache Iceberg);

– налаштувати конфігурація Apache Kafka для забезпечення надійної черги повідомлень з високою пропускною здатністю;

– реалізувати алгоритми обробки, агрегації та фільтрації IoT-даних у реальному часі за допомогою Apache Spark;

– розробити компоненти системи ізольовані в Docker-контейнерах для забезпечення мобільності та швидкого розгортання;

– налаштувати Apache Superset із розподіленим сховищем для візуалізації результатів аналізу;

– провести тестування системи на предмет стійкості до навантажень та коректності відновлення обробки після збоїв (fault tolerance).

2.3 Характеристика об'єкту

Об'єктом розробки є розподілена високопродуктивна система реального часу, призначена для збирання, агрегації, потокового аналізу та персистентного збереження великих масивів телеметричних даних із датчиків IoT-пристроїв. Архітектурно об'єкт проєктування побудований на базі Docker-контейнеризованої інфраструктури, яка об'єднує MQTT-шлюзи Eclipse Mosquitto, платформу черг повідомлень Apache Kafka та аналітичний рушій Apache Spark. Первинне приймання неструктурованих часових рядів здійснюється через легковажний протокол MQTT, після чого потоки даних буферизуються в Kafka-топіках для забезпечення стійкості до пікових навантажень. Обробка та обчислення метрик відбуваються за допомогою модуля Spark Structured Streaming із подальшим записом даних у транзакційні таблиці Apache Iceberg в об'єктному сховищі MinIO. Кінцевий аналітичний рівень об'єкта представлений платформою бізнес-аналітики Apache Superset, що забезпечує низьку затримку (low latency) при візуалізації технологічних параметрів на інтерактивних вебдашбордах.

3 Вимоги до системи

3.1 Вимоги до системи в цілому

3.1.1 Вимоги до структури та функціонування системи

Структура системи агрегації та аналізу великих масивів IoT-даних в реальному часі складається з таких апаратних та програмних компонентів:

– локальна хост-система з розгорнутим середовищем контейнеризації Docker Engine та інструментами оркестрації Docker Compose;

– контейнеризований шлюз MQTT Broker для первинного приймання та маршрутизації асинхронних потоків даних від кінцевих IoT-пристроїв;

– розподілений кластер Apache Kafka у складі брокерів Kafka One та Kafka Two для надійної буферизації, реплікації та зберігання черг повідомлень;

– веб-інтерфейс Kafka UI для централізованого адміністрування, моніторингу топіків та контролю стану брокерів черг;

– обчислювальний кластер Apache Spark у складі вузлів Spark Master та Spark Worker для розподіленої обробки великих даних;

– програмні модулі Spark Job та Spark Stream Cron для автоматизації конвеєрів очищення, валідації та потокової трансформації даних за допомогою Spark Structured Streaming;

– інтегрований аналітичний шлюз Spark Thrift Server для забезпечення відкритого JDBC/ODBC доступу до даних через стандартні SQL-інтерфейси;

– об’єктне сховище MinIO, інтегроване з файловою системою хоста (Host’s File System), для тривалого зберігання даних;

– табличний формат Apache Iceberg для організації високопродуктивного транзакційного шару архітектури Data Lakehouse із підтримкою ACID-транзакцій;

– аналітична платформа Apache Superset для побудови інтерактивних панелей моніторингу, візуалізації технологічних метрик та контролю стабільності роботи IoT-мережі;

– інфраструктура системи, включаючи модулі отримання, обробки, зберігання та візуалізації IoT-даних, розгорнута як єдиний незалежний комплекс сервісів за допомогою інструменту оркестрації Docker Compose.

3.1.2 Вимоги до способів та засобів зв’язку між компонентами системи

У контейнеризованій мережі «service-network» зв’язок між Docker-компонентами забезпечують віртуальні пари інтерфейсів Veth. Телеметрія з IoT-пристроїв надходить до шлюзу за протоколом MQTT, після чого буферизується в Apache Kafka. Обчислювальні воркери Apache Spark

паралельно зчитують ці черги в реальному часі через нативні високопродуктивні конектори. Результати обробки Spark Structured Streaming записує в об’єктне сховище MinIO (через S3 API), використовуючи транзакційний табличний формат Apache Iceberg. Доступ до аналітичних шарів даних реалізовано через шлюз Spark Thrift Server за протоколами JDBC/ODBC, до якого платформа Apache Superset надсилає SQL-запити для побудови дашбордів. Зовнішній доступ користувачів до вебінтерфейсів захищено протоколом HTTPS, а зв’язок із хост-системою забезпечується через механізм прокидання портів контейнерів.

3.1.3 Вимоги до режимів функціонування системи

Для системи визначено два режими функціонування:

- нормальний режим функціонування;
- аварійний режим функціонування.

Основним режимом функціонування системи є нормальний режим, який передбачає безперервне збирання потоків IoT-телеметрії, їх буферизацію, розподілену обробку та коректне відображення аналітичних метрик на вебдашбордах. Для забезпечення стабільного нормального режиму необхідно суворо виконувати вимоги й дотримуватись умов експлуатації базового програмного забезпечення та комплексу апаратних засобів дата-центру, які вказані у відповідних інструкціях з експлуатації та технічних регламентах.

Аварійний режим функціонування системи характеризується повною або частковою відмовою одного чи декількох компонентів програмного або технічного забезпечення (наприклад, збоєм окремого брокера Kafka, воркера Spark чи втратою зв'язку з об'єктним сховищем). При цьому архітектура системи завдяки механізмам контейнеризації, реплікації та ізоляції сервісів повинна зберігати живучість, підтримувати працездатність конвеєрів збору даних та забезпечувати функціонування базових засобів моніторингу в межах критичних налаштувань.

3.1.4 Вимоги по діагностуванню системи

Діагностування даної системи варто проводити згідно зі встановленим розкладом або у випадку виникнення збоїв у роботі контейнеризованої інфраструктури та конвеєрів обробки IoT-даних.

Головні інструменти, що використовуються для проведення діагностики, тестування та моніторингу стану елементів системи:

– Docker logs – потрібен для збору системних логів контейнерів, виконання діагностичних команд та швидкого налагодження проблем роботи сервісів;

– Kafka UI – веб-інтерфейс для візуалізації стану брокерів, моніторингу затримок у топіках та контролю споживання черг повідомлень кластером Spark;

– Spark UI – інтегрована веб-консоль для детального аналізу виконання потокових завдань, моніторингу обчислювальних воркерів та діагностики затримок у конвеєрах Spark Structured Streaming.

3.1.5 Перспективи розвитку, проектування системи

Перспективи розвитку та модернізації розробленої системи включають в себе перехід від локальної оркестрації Docker Compose до повноцінного кластера Kubernetes з метою впровадження механізмів автоматичного горизонтального масштабування обчислювальних воркерів Spark та брокерів Kafka залежно від динаміки вхідного IoT-трафіку.

Сюди також входить інтеграція розподілених інструментів логування та трасування за допомогою стека Prometheus і Grafana для забезпечення більш глибокого аналізу метрик інфраструктури, розширення аналітичного шару алгоритмами машинного навчання на базі Spark MLlib для передбачуваної діагностики обладнання, а також впровадження хмарних об’єктних сховищ як гібридного рівня зберігання історичних масивів даних.

3.2 Показники призначення

Система повинна забезпечувати безперервне приймання, потокову обробку та персистентне збереження телеметричних даних у режимі реального часу з мінімальною затримкою (low latency). Показники призначення об'єкта розробки мають гарантувати стабільну пропускну здатність при пікових навантаженнях від IoT-пристроїв, а також забезпечувати високу точність агрегації метрик і швидку візуалізацію аналітичних звітів на дашбордах.

3.2.1 Вимоги до надійності

Надійність системи агрегації та аналізу великих масивів IoT-даних впроваджується шляхом забезпечення її безперервної та стабільної роботи в режимі реального часу. Контейнеризація сервісів у середовищі Docker разом із реплікацією повідомлень в середині кластера Apache Kafka забезпечує надійне функціонування і безперервну доступність даних за допомогою механізмів відмовостійкості, ізоляції можливих збоїв та резервування критичних обчислювальних компонентів. Тривале збереження транзакційних масивів інформації на рівні Apache Iceberg у поєднанні з локальним об’єктним сховищем гарантує цілісність даних, захист від втрат у моменти пікових навантажень та високу мережеву безпеку інфраструктури.

3.3 Вимоги до безпеки

Зовнішні елементи технічних засобів дата-центру, що перебувають під напругою, повинні бути обладнані надійним захистом від випадкового дотику, а самі серверні шафи й комутаційне устаткування повинні мати захисне занулення або заземлення із нормативним опором розтікання струму не більше 4 Ом.

Система електроживлення розподіленої інфраструктури повинна забезпечувати автоматичне захисне вимкнення за допомогою пристроїв захисного вимкнення та вимикачів при перевантаженнях чи коротких замиканнях в колах навантаження, а також мати доступні органи ручного аварійного знеструмлення всього залу.

Загальні вимоги пожежної безпеки технологічних приміщень повинні відповідати нормам державних будівельних регламентів. У разі виникнення локального загорання в ізоляції кабельних ліній чи модулів живлення не мають виділятися отруйні гази й високотоксичний дим, а для ліквідації вогню без пошкодження плат електроніки мають застосовуватися автоматичні системи газового пожежогасіння або ручні вуглекислотні вогнегасники.

3.3.1 Вимоги до експлуатації, технічного обслуговування, ремонту і зберігання компонентів системи

Мікроклімат у технологічних приміщеннях розподіленої серверної інфраструктури та дата-центрів повинен суворо відповідати нормам виробничого мікроклімату по ДСН 3.3.6.042-99:

– температуру повітря в межах від +10 °С до +35 °С;

– відносну вологість повітря при 25 °С в межах від 30 % до 80 %;

– атмосферний тиск 760±25 мм рт. ст.

Періодичне технічне обслуговування використовуваного апаратного забезпечення має проводитися відповідно до вимог супровідної технічної документації виробників, але не рідше ніж один раз на рік.

Комплексне технічне обслуговування та інструментальне тестування технічних засобів повинні охоплювати перевірку всіх компонентів архітектури, зокрема: серверних вузлів, мережевих комутаторів і шлюзів, систем збору та передачі даних, а також промислових джерел безперебійного живлення під навантаженням.

На підставі результатів регулярного тестування та моніторингу апаратних засобів повинен здійснюватися глибокий аналіз причин виникнення

будь-яких виявлених дефектів, збоїв в ізоляції чи відхилень у роботі обладнання, а також негайно вживатися інженерно-технічні заходи для їх повної ліквідації.

3.4 Вимоги до захисту інформації від несанкціонованого доступу

Система повинна забезпечувати захист від несанкціонованого доступу на рівні не нижче встановленого вимогами, що пред'являються до категорії 1Д по класифікації документа, що діє, «Автоматизовані системи. Захист від несанкціонованого доступу до інформації. Класифікація автоматизованих систем».

Компоненти підсистеми захисту від НСД повинні забезпечувати:

– ідентифікацію користувача;

– перевірку повноважень користувача при роботі з системою;

– розмежування доступу користувачів.

Рівень захищеності від несанкціонованого доступу засобів обчислювальної техніки, що здійснюють обробку конфіденційної інформації, повинен відповідати вимогам класу захищеності згідно вимогам документу «Засоби обчислювальної техніки. Захист від несанкціонованого доступу до інформації. Показники захищеності від несанкціонованого доступу до інформації».

3.4.1 Вимоги по збереженню інформації при аваріях

Інформація, при виникненні аварійних ситуацій повинна бути збережена на резервних носіях.

3.4.2 Вимоги по стандартизації і уніфікації

Програмні компоненти системи повинні розроблятися з використанням уніфікованих підходів до контейнеризації, відкритих стандартів взаємодії та загальноприйнятих протоколів передачі даних. Система повинна відповідати

сучасним вимогам ергономіки, інтуїтивності та зручності користування веб-інтерфейсами за умови комплектування інфраструктури високоякісним апаратним обладнанням (серверними вузлами, моніторами оператора та іншим устаткуванням), що має необхідні державні та міжнародні сертифікати відповідності, якості та безпеки.

3.4.3 Вимоги до функцій (завдань), що виконуються системою:

– зручний інтерфейс передбачає наявність інтуїтивного вебінтерфейсу та інтерактивних дашбордів для моніторингу та візуалізації метрик у реальному часі;

– зберігання медіафайлів гарантує надійне та відмовостійке збереження великих обсягів даних, включаючи телеметрію, логи, супутні медіафайли та бінарні об'єкти, у розподіленому об'єктному сховищі;

– пошук інформації забезпечує можливість швидкого пошуку, фільтрації та SQL-агрегації історичних даних за часом, типами IoT-пристроїв чи ідентифікаторами датчиків;

– висока швидкодія визначає мінімальну затримку на всьому шляху даних від їх генерації на пристрої та буферизації в чергах до відображення на графіках користувача.

4 Вимоги до документації

Документація повинна відповідати вимогам ЄСКД та ДСТУ

Комплект документації повинен складатись з:

– пояснювальної записки;

– графічного матеріалу:

а) Структурна схема компонентів розподіленої системи обробки даних.

б) Структурна схема порівняння повномасштабного та реалізованого технологічних стеків в умовах апаратних обмежень.

в) Структурна схема медальйонної архітектури збереження та обробки даних.

г) Структурна схема опрацювання повідомлення в реальному часі.

*Примітка: У комплект документації можуть вноситися зміни та доповнення в процесі розробки.

5 Стадії та етапи проектування

Таблиця 1 – Стадії та етапи виконання кваліфікаційної роботи бакалавра

№ етапу	Назва етапу виконання кваліфікаційної роботи бакалавра	Термін виконання
1	*Розробка технічного завдання*	*26.01 – 02.02*
2	*Робота над першим розділом «Аналіз технічного завдання»*	*03.02 – 15.02*
3	*Робота над другим розділом «Проєктна чатсина»*	*20.04 – 28.04*
4	*Робота над третім розділом «Практична частина»*	*29.04 – 20.05*
5	*Робота над четвертим розділом «Безпека життєдіяльності, основи охорони праці»*	*21.05 – 25.05*
6	*Оформлення пояснювальної записки і графічного матеріалу*	*26.05 – 7.06*
7	*Перевірка на академічний плагіат, перевірка керівником та консультантами*	*8.06 – 14.06*
8	*Попередній захист кваліфікаційної роботи бакалавра*	*15.06 – 21.06*
9	*Захист кваліфікаційної роботи бакалавра*	*22.06*

6 Додаткові умови виконання кваліфікаційної роботи

Під час виконання кваліфікаційної роботи у дане технічне завдання можуть вноситися зміни та доповнення.

Додаток Б

Лістинг коду

Б.1 - Лістинг коду «docker-compose.yml» із взаємодією Apache Kafka сервісів

```
services:
  kafka-1:
    image: confluentinc/cp-kafka:7.7.7
    mem_limit: 4g
    cpus: 4.0
    hostname: kafka1
    container_name: kafka1
    environment:
      KAFKA_NODE_ID: 1
      KAFKA_BROKER_ID: 1
      KAFKA_PROCESS_ROLES: "broker,controller"
      KAFKA_CONTROLLER_QUORUM_VOTERS:
     "1@kafka1:9093,2@kafka2:9093"
      KAFKA_LISTENERS:
     "PLAINTEXT://kafka1:9092,CONTROLLER://kafka1:9093"
      KAFKA_ADVERTISED_LISTENERS: "PLAINTEXT://kafka1:9092"
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
     "CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT"
      KAFKA_CONTROLLER_LISTENER_NAMES: "CONTROLLER"
      KAFKA_INTER_BROKER_LISTENER_NAME: "PLAINTEXT"
      CLUSTER_ID: "SourceClusterID-133331"
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 2
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
      KAFKA_DEFAULT_REPLICATION_FACTOR: 3
      KAFKA_MIN_INSYNC_REPLICAS: 2
      KAFKA_AUTO_CREATE_TOPICS_ENABLE: "false"
    ports:
      - 9092:9092
      - 9093:9093
    volumes:
      - ${PATH_TO_KAFKA_1_STORAGE}:/var/lib/kafka/data
      - kafka-1-logs:/app/logs
    networks:
      - service-network
kafka-2:
    image: confluentinc/cp-kafka:7.7.7
    mem_limit: 4g
    cpus: 4.0
    hostname: kafka2
    container_name: kafka2
    environment:
      KAFKA_NODE_ID: 2
      KAFKA_BROKER_ID: 2
      KAFKA_PROCESS_ROLES: "broker,controller"
```

```
        KAFKA_CONTROLLER_QUORUM_VOTERS:
       "1@kafka1:9093,2@kafka2:9093"
        KAFKA_LISTENERS:
       "PLAINTEXT://kafka2:9092,CONTROLLER://kafka2:9093"
        KAFKA_ADVERTISED_LISTENERS: "PLAINTEXT://kafka2:9092"
        KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
        "CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT"
        KAFKA_CONTROLLER_LISTENER_NAMES: "CONTROLLER"
        KAFKA_INTER_BROKER_LISTENER_NAME: "PLAINTEXT"
        CLUSTER_ID: "SourceClusterID-133331"
        KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 2
        KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
        KAFKA_DEFAULT_REPLICATION_FACTOR: 2
        KAFKA_MIN_INSYNC_REPLICAS: 2
        KAFKA_AUTO_CREATE_TOPICS_ENABLE: "false"
      ports:
        - 9094:9092
        - 9095:9093
      volumes:
        - ${PATH_TO_KAFKA_2_STORAGE}:/var/lib/kafka/data
        - kafka-2-logs:/app/logs
      networks:
        - service-network
    kafka-ui:
      image: provectuslabs/kafka-ui:latest
      mem_limit: 512m
      cpus: 0.5
      container_name: kafka-cluster-ui
      environment:
        KAFKA_CLUSTERS_0_NAME: local
        KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS:
       kafka1:9092,kafka2:9092
      ports:
        - 8080:8080
      volumes:
- kafka-ui-logs:/app/logs
      networks:
        - service-network
      depends_on:
        - kafka-1
        - kafka-2
    ...
```

Б.2 - Лістинг коду класу-обгортки «MQTTClientWrapper»

```
class MQTTClientWrapper:
    """
    Wrapper for MQTT client to consume messages and forward
them to Kafka.

    Encapsulates an MQTT client configured via
‘MQTTSubscriptionConfig’,
```

```
        subscribes to specified topics, and produces messages to
a Kafka
        topic. Handles connection, disconnection, message
reception, and
        error reporting.

        Attributes:
            config: MQTT subscription and Kafka configuration.
            client: Initialized paho-mqtt client with callbacks.
            producer: Kafka producer instance for publishing
         messages.
        """

        def __init__(self,
            config: MQTTSubscriptionConfig
        ) -> None:
            """
            Initialize MQTT client and Kafka producer.

    Sets up MQTT callbacks (connect, disconnect, message) and
            credentials if provided.

            Args:
              config: 'MQTTSubscriptionConfig' object with MQTT
              broker,topics, and Kafka target settings.
            """
            self.config = config
            try:
                self.client = mqtt.Client(
                    mqtt.CallbackAPIVersion.VERSION2,
                    client_id=config.client_id,
                    clean_session=config.clean_session,
                )
                self.client.enable_logger()
            except Exception as e:
                logging.error(f"Client object has not been
    created - {e}")

            if self.config.client_username and
    self.config.client_password:

    self.client.username_pw_set(config.client_username,
                config.client_password)
      self.client.on_connect = self.on_connect
            self.client.on_disconnect = self.on_disconnect
            self.client.on_message = self.on_message
            try:
                self.producer = Producer({'bootstrap.servers':
                config.kafka_bootstrap})
            except Exception as e:
                logging.error(f"Producer object has not been
         created - {e}")
```

Б.3 - Лістинг коду Docker образу Apache Kafka

```
...
  kafka-1:
    image: confluentinc/cp-kafka:7.7.7
    mem_limit: 4g
    cpus: 4.0
    hostname: kafka1
    container_name: kafka1
    environment:
KAFKA_NODE_ID: 1
      KAFKA_BROKER_ID: 1
      KAFKA_PROCESS_ROLES: "broker,controller"
      KAFKA_CONTROLLER_QUORUM_VOTERS:
     "1@kafka1:9093,2@kafka2:9093"
      KAFKA_LISTENERS:
     "PLAINTEXT://kafka1:9092,CONTROLLER://kafka1:9093"
      KAFKA_ADVERTISED_LISTENERS: "PLAINTEXT://kafka1:9092"
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
      "CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT"
      KAFKA_CONTROLLER_LISTENER_NAMES: "CONTROLLER"
      KAFKA_INTER_BROKER_LISTENER_NAME: "PLAINTEXT"
      CLUSTER_ID: "SourceClusterID-133331"
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 2
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
      KAFKA_DEFAULT_REPLICATION_FACTOR: 3
      KAFKA_MIN_INSYNC_REPLICAS: 2
      KAFKA_AUTO_CREATE_TOPICS_ENABLE: "false"
    ports:
      - 9092:9092
      - 9093:9093
    volumes:
      - ${PATH_TO_KAFKA_1_STORAGE}:/var/lib/kafka/data
      - kafka-1-logs:/app/logs
    networks:
      - service-network
  kafka-2:
    image: confluentinc/cp-kafka:7.7.7
    mem_limit: 4g
    cpus: 4.0
    hostname: kafka2
    container_name: kafka2
    environment:
      KAFKA_NODE_ID: 2
      KAFKA_BROKER_ID: 2
      KAFKA_PROCESS_ROLES: "broker,controller"
      KAFKA_CONTROLLER_QUORUM_VOTERS:
     "1@kafka1:9093,2@kafka2:9093"
      KAFKA_LISTENERS:
     "PLAINTEXT://kafka2:9092,CONTROLLER://kafka2:9093"
      KAFKA_ADVERTISED_LISTENERS: "PLAINTEXT://kafka2:9092"
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
      "CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT"
```

```
      KAFKA_CONTROLLER_LISTENER_NAMES: "CONTROLLER"
      KAFKA_INTER_BROKER_LISTENER_NAME: "PLAINTEXT"
      CLUSTER_ID: "SourceClusterID-133331"
     KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 2
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
      KAFKA_DEFAULT_REPLICATION_FACTOR: 2
      KAFKA_MIN_INSYNC_REPLICAS: 2
      KAFKA_AUTO_CREATE_TOPICS_ENABLE: "false"
    ports:
      - 9094:9092
      - 9095:9093
    volumes:
      - ${PATH_TO_KAFKA_2_STORAGE}:/var/lib/kafka/data
      - kafka-2-logs:/app/logs
    networks:
      - service-network
  ...
```

Б.4 - Лістинг коду Docker образу Apache Spark

```
FROM maven:3.9.3-eclipse-temurin-17 AS deps
WORKDIR /tmp/dependencies

COPY maven.xml /tmp/dependencies/pom.xml
RUN mvn clean package
# COPY maven.xml /tmp/dependencies/pom.xml
#      RUN      mvn      dependency:copy-dependencies      -
DoutputDirectory=/jars

FROM python:3.12-bullseye
# default shell is sh
# RUN apt-get update && apt-get install -y maven openjdk-17-
jdk wget curl
&& \
#     apt-get clean && rm -rf /var/lib/apt/lists/*
RUN apt-get update && \
    apt-get install -y --no-install-recommends openjdk-17-jdk
&& \
    apt-get clean && rm -rf /var/lib/apt/lists/*
ARG SPARK_VERSION=3.5.8
ARG HADOOP_VERSION=3
ENV SPARK_HOME="/opt/spark"
ENV JAVA_HOME="/usr/lib/jvm/java-17-openjdk-amd64"
ENV
PATH="${JAVA_HOME}:${SPARK_HOME}/bin:${SPARK_HOME}/sbin:${PA
TH}"
RUN mkdir -p ${SPARK_HOME}
WORKDIR ${SPARK_HOME}
# Fetch spark 3.5.8 version and download the appropriate
binary file
RUN    curl    -fSL    https://dlcdn.apache.org/spark/spark-
${SPARK_VERSION}/spark
```

```
${SPARK_VERSION}-bin-hadoop${HADOOP_VERSION}.tgz \
    -o /tmp/spark.tgz
# Unpack the file and cleanup the binary file
RUN tar -xzf /tmp/spark.tgz -C ${SPARK_HOME} --strip-
components=1 \
    && rm /tmp/spark.tgz
# Copy deps from a later build step
COPY --from=deps /tmp/dependencies/target/spark-deps-1.0.jar
${SPARK_HOME}/jars/
# COPY --from=deps /jars ${SPARK_HOME}/jars/
WORKDIR ${SPARK_HOME}
# Port master will be exposed
ENV SPARK_MASTER_PORT="7077"
# Name of master container and also counts as hostname
ENV SPARK_MASTER_HOST="spark-master"
ENTRYPOINT ["/bin/bash"]
```

Б.5 - Лістинг коду розгортання Apache Spark-кластера за допомогою Docker Compose

```
...
  spark-master:
    build:
      context: .
      dockerfile: containers/spark/maven/Dockerfile
    image: spark:latest
    mem_limit: 1g
    cpus: 1.0
    container_name: spark-master
    environment:
      SPARK_NO_DAEMONIZE: "true"
      SPARK_MASTER_MEMORY: "1g"
    entrypoint: /bin/bash
    command: -c "/opt/spark/sbin/start-master.sh"
    ports:
      - 8082:8080
      - 4040:4040
    volumes:
      - spark-master-logs:/app/logs
    networks:
      - service-network
    depends_on:
      - minio
  spark-worker:
    image: spark:latest
    mem_limit: 6g
    cpus: 4.0
    container_name: spark-worker1
    environment:
      SPARK_NO_DAEMONIZE: "true"
      SPARK_WORKER_CORES: 4
      SPARK_WORKER_MEMORY: "6g"
```

```
        entrypoint: /bin/bash
         command:       -c       "/opt/spark/sbin/start-worker.sh
         spark://spark-master:7077"
    volumes:
          - spark-workers-logs:/app/logs
        networks:
          - service-network
        depends_on:
          - minio
```

Б.6 - Лістинг коду розгортання Apache Spark-кластера за допомогою Docker Compose

```
    ...
    processed_df = parsed_df.select(

trim(col("channel_id")).cast(IntegerType()).alias("channel_id"),

trim(col("entry_id")).cast(IntegerType()).alias("entry_id"),
    to_timestamp(col("created_at")).alias("created_at"),
        from_utc_timestamp(col("created_at"),
    "America/New_York").alias("ts_created_at"),
        when(col("channel_id").isNotNull(),
True).otherwise(False)
            .alias("is_created"),

trim(col("field1")).cast(DecimalType(18,4)).alias("temperature")
,

trim(col("field2")).cast(DecimalType(18,4)).alias("humidity"),
        when(col("channel_id") == 3217660,
    col("field3").cast(DecimalType(18,4)))
            .otherwise(col("field4").cast(DecimalType(18,4)))
            .alias("pressure"),
        when(col("channel_id") == 371428,
    col("field3").cast(DecimalType(18,4)))
            .otherwise(lit(None).cast(DecimalType(18,4)))
            .alias("air_co2"),
        when(col("channel_id") == 3217660,
    col("field4").cast(DecimalType(18,4)))
            .otherwise(lit(None).cast(DecimalType(18,4)))
            .alias("luxmeter"),
        when(col("channel_id") == 3217660,
    col("field5").cast(DecimalType(18,4)))
            .otherwise(lit(None).cast(DecimalType(18,4)))
            .alias("moisture"),
    when(col("channel_id")              ==              371428,
col("field6").cast(IntegerType()))
            .otherwise(lit(None).cast(IntegerType()))
            .alias("particulate_matter_1_0"),
```

```
     when(col("channel_id")               ==               371428,
col("field7").cast(IntegerType()))
              .otherwise(lit(None).cast(IntegerType()))
              .alias("particulate_matter_2_5"),
     when(col("channel_id")               ==               371428,
col("field8").cast(IntegerType()))
              .otherwise(lit(None).cast(IntegerType()))
              .alias("particulate_matter_10"),

to_timestamp(current_timestamp()).alias("dwr_extracted_at"),
          to_timestamp(col("created_at")).alias("dwr_created_at"),
          col("raw_payload")
     )
     ...
```