Міністерство освіти і науки України
Рівненський державний гуманітарний університет
Кафедра інформаційних технологій та моделювання

КВАЛІФІКАЦІЙНА РОБОТА

за освітнім ступенем «бакалавр»

на тему:

«IoT-платформа моніторингу стану транспортного засобу з обробкою даних у хмарній інфраструктурі»

Виконав:
студент IV-го курсу
групи ІПЗ-41
спеціальності 121 «Інженерія програмного забезпечення»
Корнійчук Олександр Васильович
Керівник:
к.т.н., доцент Шинкарчук Н.В.

Рівне – 2026

АНОТАЦІЯ ДО КВАЛІФІКАЦІЙНОЇ РОБОТИ

Корнійчук О.В. IoT-платформа моніторингу стану транспортного засобу з обробкою даних у хмарній інфраструктурі. Кваліфікаційна робота на здобуття ступеня «бакалавр» за спеціальністю 121 «Інженерія програмного забезпечення» – Рівненський державний гуманітарний університет. Рівне, 2026. 55 с.

Бакалаврська робота присвячена розробці платформи моніторингу транспортних засобів на основі технологій Інтернету речей із хмарною обробкою телеметричних даних. Предметом дослідження є архітектурні та програмні рішення для збору, передачі, обробки і візуалізації даних про стан транспортних засобів у реальному часі.

На основі порівняльного аналізу IoT-протоколів і хмарних платформ обрано MQTT (QoS 1) та Microsoft Azure. Архітектура охоплює чотири рівні: Python-симулятор із фізично пов’язаними параметрами, Azure IoT Hub для прийому телеметрії, serverless-pipeline з трьох Azure Functions та сховищ Cosmos DB і Azure SQL із дашбордом Metabase. Виявлення аномалій працює на двох рівнях – порогові перевірки і статистичний z-score на ковзному вікні. Безпека охоплює три незалежних рівні: TLS 1.2, SAS-токени і HMAC-SHA256-підпис кожного повідомлення. Виявлені аномалії автоматично фіксуються у журналі Google Sheets.

Систему розгорнуто виключно на безкоштовних рівнях Azure (вартість $0). Практичне тестування підтвердило коректну роботу всіх компонентів платформи і відповідність вимогам.

Ключові слова: Інтернет речей, MQTT, Azure IoT Hub, Azure Functions, Cosmos DB, виявлення аномалій, z-score, хмарні обчислення, serverless, HMAC-SHA256.

ABSTRACT

Korniichuk O.V. IoT-Based Vehicle Condition Monitoring Platform with Cloud Data Processing Infrastructure. Bachelor's qualification thesis in specialty 121 "Software Engineering" – Rivne State University of the Humanities. Rivne, 2026. 55 p.

This bachelor's thesis addresses the design and development of a vehicle monitoring platform built on Internet of Things technologies with cloud-based processing of telemetry data. The subject of the study encompasses architectural and software solutions for real-time collection, transmission, processing, and visualization of vehicle condition parameters.

Following a comparative analysis of IoT communication protocols and cloud platforms, MQTT (QoS 1) and Microsoft Azure were selected as the primary technology stack. The proposed architecture consists of four tiers: a Python-based simulator generating physically correlated vehicle parameters, Azure IoT Hub for telemetry ingestion, a serverless processing pipeline composed of three Azure Functions, and a storage layer built on Cosmos DB and Azure SQL, with a Metabase dashboard for data visualization. Anomaly detection operates at two levels — threshold-based checks and statistical z-score analysis over a sliding window. The security model incorporates three independent layers: TLS 1.2 transport encryption, SAS token authentication, and per-message HMAC-SHA256 signature verification. Detected anomalies are automatically recorded to a Google Sheets event log.

The entire system was deployed exclusively on Azure free-tier services at zero cost. Functional testing confirmed that all platform components operate correctly and meet the defined requirements.

Keywords: Internet of Things, MQTT, Azure IoT Hub, Azure Functions, Cosmos DB, anomaly detection, z-score, cloud computing, serverless, HMAC-SHA256.

ЗМІСТ

ВСТУП

Актуальність роботи. Світ Індустрії 4.0 диктує нові правила. Автомобіль перестав бути просто залізом і перетворився на складний цифровий вузол. Проте виникає парадокс. Даних багато, але вони зазвичай просто лежать як «цифрове сміття» у пам’яті контролерів, не приносячи жодної користі власнику. Сучасний бізнес не може бути ефективним та конкурентноздатним, якщо буде дізнаватися про поломку за фактом. Потрібна проактивність. Наприклад, створивши «цифрового двійника» у хмарному середовищі, можна звільнити пам’ять та оптимізувати роботу процесора бортового комп’ютера авто. При цьому, зібрані дані акумулюються у базі даних в хмарі, цифрові сервіси якої, надають інструменти для їх подальшої організації, обробки та візуалізації. Такий підхід надає бізнесу широкий спектр можливостей для проведення аналітики, систематизації інформації, створенню цифрового бекапу, що забезпечить стратегічну перевагу на ринку.

Мета роботи: реалізація IoT-платформи моніторингу стану транспортного засобу з обробкою даних у хмарній інфраструктурі.

Об’єктом дослідження є процес збору, передачі та візуалізації даних в системах Інтернету речей.

Інструментом дослідження є мова програмування Python для розробки програми-симулятора транспортного засобу, хмарна платформа Microsoft Azure для обробки та зберігання даних, система управління базами даних Azure SQL, середовище контейнеризації Docker для розгортання сервісів, а також платформа Metabase для візуалізації даних.

Предметом дослідження є методи побудови хмарної архітектури для збору та обробки телеметрії в Microsoft Azure та інструменти візуалізації у Metabase.

Завдання дослідження:

1. Дослідити концепцію IoT у транспорті та проаналізувати протоколи передачі даних (MQTT, CoAP, AMQP).

2. Проаналізувати можливості хмарних платформ для IoT та обґрунтувати вибір платформи Microsoft Azure для розгортання проєкту.
3. Провести огляд існуючих рішень на ринку моніторингу транспортних засобів для виявлення кращих практик та обмежень.
4. Спроектувати архітектуру платформи, включаючи модель даних та логіку розподілу потоків обробки.
5. Розробити прототип системи та протестувати її здатність обробляти дані у хмарному середовищі.

Апробація результатів кваліфікаційної роботи. Результати виконання кваліфікаційної роботи, окремі її аспекти та одержані узагальнення і висновки оприлюднено на І Міжнародній науково-практичній конференції «Сучасні інформаційні технології: від теорії до практики» (м. Луцьк, 2026 р.), звітній науковій конференції викладачів, співробітників та здобувачів вищої освіти Рівненського державного гуманітарного університету за 2025 рік (м. Рівне, 2026 р.).

Публікації. Результати, які були отримані в ході кваліфікаційного дослідження опубліковано у формі тез доповідей на І Міжнародній науково-практичній конференції «Сучасні інформаційні технології: від теорії до практики»» (м. Луцьк, 2026 р.)(Додаток А).

Структура роботи. Дипломна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків. У першому розділі проаналізовано технології та протоколи інтернету речей (MQTT, CoAP, AMQP), а також особливості збору телеметрії через інтерфейс OBD-II. Другий розділ присвячено огляду хмарних обчислень як основи сучасних IoT-рішень та аналізу особливостей Microsoft Azure у порівнянні з альтернативними хмарними платформами. У межах третього розділу проведено аналіз сучасних IoT-рішень для моніторингу транспортних засобів та спроєктовано архітектуру платформи, включаючи структуру бази даних і конфігурацію хмарних сервісів. Завершальний четвертий розділ містить опис практичної реалізації системи та аналіз результатів тестування розробленого програмного забезпечення.

РОЗДІЛ 1
ТЕХНОЛОГІЇ ТА ПРОТОКОЛИ ІНТЕРНЕТУ РЕЧЕЙ ДЛЯ ЗБОРУ ТРАНСПОРТНИХ ДАНИХ І ТЕЛЕМЕТРІЇ

1.1. Основи та архітектура Інтернету речей

Термін «Internet of Things» (IoT) вперше з’явився у 1999 році. Що цікаво, термін з’вився не в академічній статті чи в технічній специфікації. Кевін Ештон, використав його у внутрішній презентації компанії Procter & Gamble про автоматичний облік товарів через RFID-мітки [1]. Таким чином, ранній концепт інтернету речей, був створений для вирішення прагматичної задачі: не витрачати час співробітників на ручний перерахунок запасів на складі.

З плином часу, IoT проходить шлях від нішевої до загальноприйнятої технології. За даними Statista, станом на 2023 рік у світі було підключено понад 19.8 мільярдів IoT-пристроїв, а в прогнозах, до 2034 року ця цифра сягне 40.6 мільярдів [2]. Відзначу, що кількість самих пристроїв тут не головне, інтернет речей змінює сам принцип, яким системи збирають, зберігають та обробляють дані. Більше того, ці системи виконують вищевказані функції без присутності людини.

Наведемо приклад: у класичному розумінні, у клієнт-серверній моделі людина через браузер чи мобільний додаток надсилає запит до серверу і, в кінцевому рахунку, отримує певну відповідь. Проте, в IoT-системі місця для людини може і не бути (рис. 1.1.). Наприклад, система в якій датчики самостійно збирають, агрегують та надсилають телеметрію в залежності від налаштувань логіки роботи пристроїв та вимірюваних параметрів.

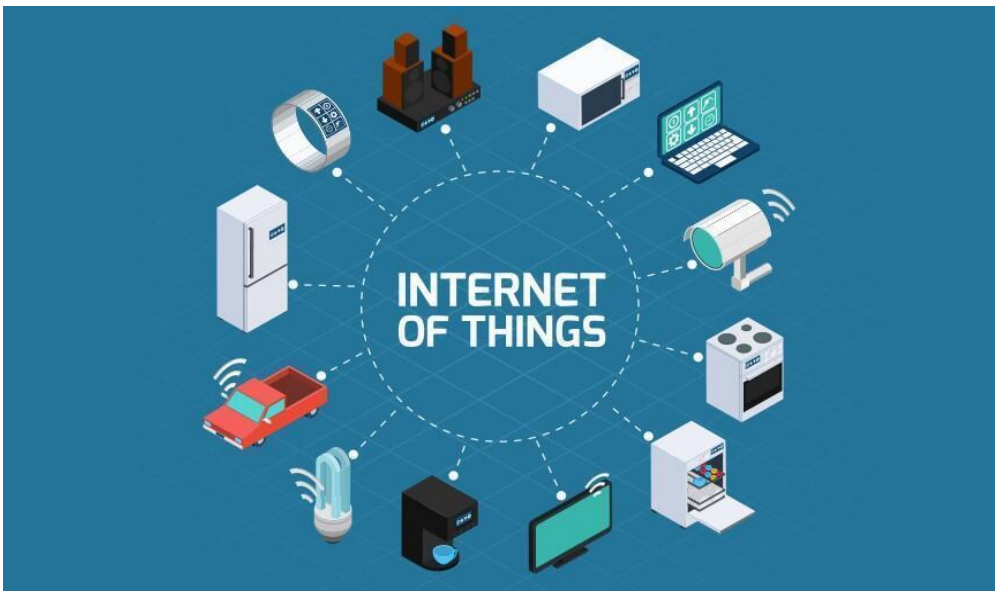


Рисунок 1.1. Інтернет речей

Для транспортної галузі це може бути проривною технологією: парк авто, що підключені у єдину систему, здатні генерувати цілу «лавину» темеметрії (координати, швидкість, температуру двигуна, оберти двигуна, тощо) без участі водія. Під'єднавши модулі для аналітики та візуалізації, логісти та диспетчери будуть здатні бачити «живу» картину стану парку авто, що підніме конкурентноспроможність бізнесу.

1.2. Архітектура Інтернету речей

Інтернет речей є складною структурою. Саме тому потрібно розгортати рішення з чітко визначеною архітектурою системи, що забезпечить безперебійну роботу між фізичними пристроями (сенсори, датчики, тощо), мережею та хмарними середовищами. Архітектура описує де і які технології використовуються, як саме дані проходять цикл: збір даних, передача через мережеву інфраструктуру, зберігання на певному пристрої пам'яті або хмарі, обробка зібраних даних та представлення користувачеві через зручний та зрозумілий інтерфейс.

На практиці, найчастіше використовується ієрархія наступного типу:

1. Шар сприйняття (Perception layer) – це найнижчий рівень, відповідає за взаємодію різного роду пристроїв: датчиків (сенсорів), RFID-міток, контролерів та камер з навколишнім світом. За допомогою згаданих пристроїв, проводиться забір даних (освітленість, вологість, температура).
2. Мережевий рівень (Network layer) – рівень, що відповідає саме за передачу даних від пристроїв до хмарних середовищ або серверів за допомогою стандартних протоколів зв'язку. Саме на цьому рівні відбувається керування мережами, трафіком, адресація, маршрутизація, інтеграція різних мережевих технологій та пристоїв для забезпечення стабільної роботи системи. Технології мережевого рівня в IoT охоплюють як короткодіапазонні мережі (Wi-Fi, Bluetooth, ZigBee), так і протоколи далекої дії (LoRa, NB-IoT, LTE-M). Також, використовуються протоколи MQTT та CoAP, що належать до рівня застосунків.
3. Рівень обробки (Processing/Middleware layer) – це рівень, що слугує «мостом» між апаратною частиною та програмною. Рівень обробки є центральним елементом для платформ, що розглядаються в цій роботі. Основна ціль – перетворити потік «сирих даних» на інформацію, зручну для подальшого використання. Основними задачами є: фільтрація та агрегація «сирих даних», Edge Computing, управління пристроями, зберігання даних.
4. Рівень застосування (Application layer) – найвищий рівень архітектури, де відбувається взаємодія з користувачами чи зовнішніми системами на основі візуалізації проаналізованої та структурованої інформації. Також, на цьому рівні користувач може налаштовувати, персоналізовувати чи впливати іншим чином на систему. Сюди можна включити інтерфейси для зручної роботи користувача, програми або додатки для проведення аналітики, а також мобільні застосунки для взаємодії «користувач-IoT».

Ієрархія архітектури IoT зображена на рисунку 1.2.

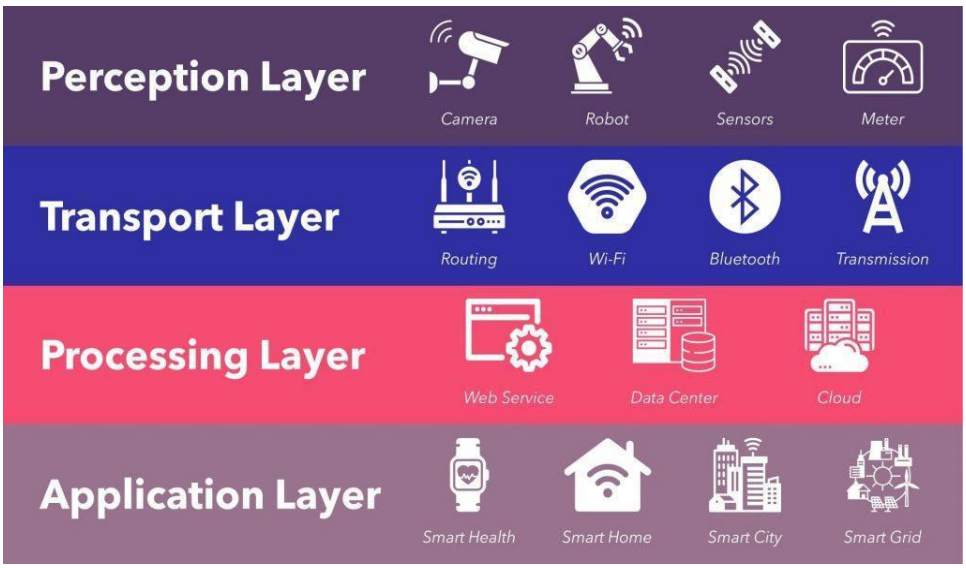


Рисунок 1.2. Архітектура IoT

Важливо відмітити: такий розподіл не є жорстким стандартом із точно прописаними межами. Наприклад, підхід, при якому обробку даних можна перенести на пристрої збору інформації (він відомий як «edge computing») активно розвивається і набуває популярності у промислових IoT-рішеннях. Проте, у даній роботі зберігання та обробка даних перенесена у хмару, такий вибір зумовлений зручністю розгортання, масштабування та налаштування IoT-рішення для моніторингу авто у реальному часі.

1.3. Джерела даних у транспортних системах

Перш ніж почати проектування системи збору даних, необхідно відповісти на декілька питань: як і які дані будуть збиратись та звідки вони беруться. У даному випадку, така відповідь може бути досить неочевидною, адже сучасне авто – складне інженерне рішення. Однак, представимо автомобіль не як інженерний проєкт, а як джерело структурованого потоку інформації. Тому представимо транспортний засіб у вигляді «blackbox». У даній роботі основний фокус не як працює двигун автомобіля, нам потрібна інформація: стан двигуна, його температура, навантаження та оберти. А джерел цих даних два: стандарт бортової діагностики OBD-II і модуль позиціонування (наприклад, GPS).

OBD-II – це уніфікована система бортової діагностики, запроваджена виробниками для контролю викидів і стану систем керування двигуном. Вона дозволяє електронному блоку фіксувати відхилення в роботі датчиків та виконавчих механізмів і зберігати коди, які механіки та автовласники можуть прочитати через стандартний діагностичний роз'єм DLC. Головна перевага OBD-II – спільна мова: коди та їх структура уніфіковані, тож базову інтерпретацію можна зробити для більшості авто [3].

Фізично це коннектор, що знаходиться, як правило, під кермом (рис. 1.3.).

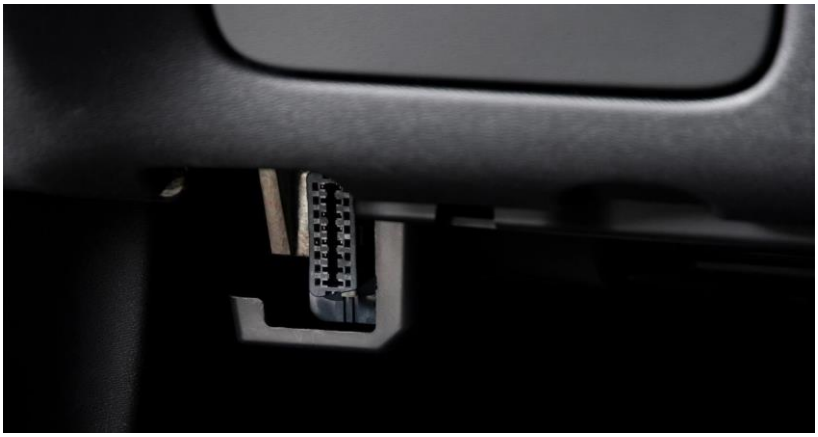


Рисунок 1.3. Коннектор OBD-II

До коннектора підключається зчитувач (рис. 1.4.) і автомобіль у відповідь надає потрібні параметри.

Рисунок 1.4. Зчитувач OBD-II

В основі, для отримання даних у стандарті OBD-II розроблена система ідентифікаторів параметрів, відома як PID (Parameter Identification). Кожен PID є числовим ключем до конкретного вузла чи вимірювання транспортного засобу. Наприклад, звернення до ідентифікатора 0x05 дозволяє отримати інформацію про температуру охолоджувальної рідини, тоді як коди 0x0C та 0x0D відповідають за частоту обертів двигуна та миттєву швидкість відповідно [4].

Система не видає описів помилок чи сигналів тривоги у текстовому вигляді, натомість інтерфейс повертає виключно числові значення усуваючи можливість неправильного трактування коду помилки, при цьому дані стають адаптованими для подальшої машинної обробки. Зокрема, такий підхід дозволяє перенести такі дані у формат JSON, що є дуже корисним для їх подальшої передачі і обробки у хмарній інфраструктурі.

Якщо OBD-II відповідає на питання «що відбувається з автомобілем», то GPS відповідає на питання «де він знаходиться». Глобальна система позиціонування визначає координати об’єкта, приймаючи сигнали від супутників і обчислюючи різницю в часі їх надходження. У результаті, отримуємо координати: широту, довготу і висоту над рівнем моря у системі відліку WGS-84.

Координати з GPS є основним джерелом для відображення маршруту на картографічному дашборді. У системі, що розробляється, GPS оновлює дані щосекунди. Поєднуючи OBD-II і GPS, можна скласти достатньо повну картину стану і місцезнаходження транспортного засобу.

1.4. Протоколи передачі телеметрії в IoT-системах

Вибір протоколу – одне з найголовніших рішень при побудові IoT-проєкту. На перший погляд, може здатись, що гарним рішенням буде вибір протоколу HTTP, адже саме він лежить в основі World Wide Web, а завдяки REST, HTTP забезпечує універсальну сумісність. На жаль, на відміну від обміну даними з вебсайтом, обмін телементрією транспорту має інші пріоритети: мінімальний

трафік, стійкість до нестабільного з'єднання, гарантія доставки при поганому сигналі.

Наведемо кілька найбільш популярних протоколів для IoT, кожен з яких вирішує конкретний клас задач, серед них:

- MQTT (Message Queuing Telemetry Transport) – легкий протокол, спрямований на роботу у мережах з низькою пропускною здатністю та пристроях з обмеженими ресурсами, який працює на основі TCP/IP. Заснований на архітектурі під назвою «publisher/subscriber» (рис. 1.5.), яка передбачає взаємодію між двома різними типами пристроїв: «видавцями» (пристрій, що надає дані) та «передплатниками» (сервер або хмара). Серед основних переваг протоколу MQTT – мінімальне споживання ресурсів і стабільна робота.
- CoAP (Constrained Application Protocol) – спроєктований для міжмашинної комунікації для пристроїв і мереж з обмеженими ресурсами. CoAP використовує UDP для компактності і оснований на моделі клієнт–сервер.
- AMQP (Advanced Message Queuing Protocol) – протокол прикладного рівня для взаємодії додатків та систем. AMQP як і протокол MQTT, базується на моделі «publisher/subscriber». Передплатник і видавець системи взаємодіють, надсилаючи та запитуючи повідомлення з «черги повідомлень».

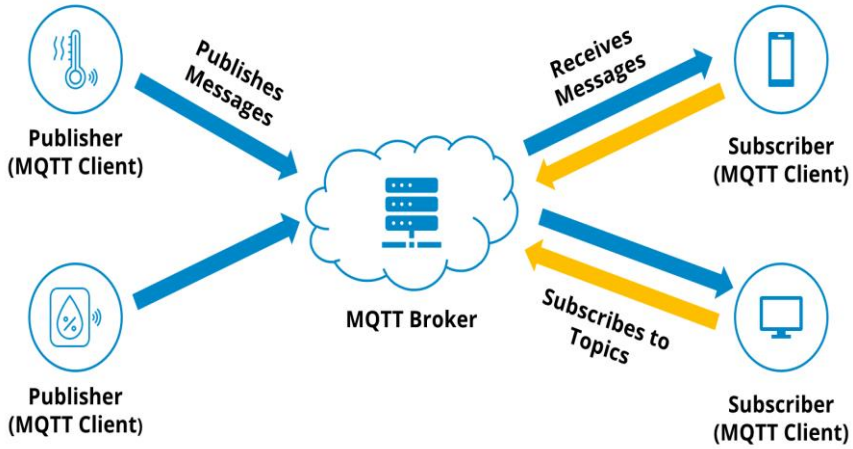


Рисунок 1.5. Схема роботи протоколу MQTT

Таблиця 1.1. Порівняльна характеристика протоколів передачі телеметрії для IoT-систем [5-8]

Критерій	**MQTT**	**AMQP**	**CoAP**
Розмір заголовка	Від 2 байтів	Приблизно 15 байт	4 байти
Транспортний протокол	TCP	TCP	UDP
Гарантія доставки	QoS 0, 1, 2	Підтвердження (Ack)	Вбудована (Confirmable)
Затримка	Низька	Низька	Мінімальна
Навантаження на канал	Мінімальне	Помірне	Мінімальне
Підтримка Azure IoT	Повна (Native)	Повна (Native)	Через шлюз

Аналіз таблиці підтверджує: MQTT має найкращу комбінацію показників для задачі, що вирішується. AMQP і CoAP у даній архітектурі не використовуються.

РОЗДІЛ 2
ОГЛЯД ХМАРНИХ ПЛАТФОРМ ДЛЯ ІНТЕРНЕТУ РЕЧЕЙ

2.1. Хмарні обчислення як основа сучасних IoT-рішень

Розгортання IoT-проєктів тісно пов’язане з вибором середовища для зберігання та обробки телеметричних даних. Впродовж тривалого часу єдиним варіантом залишався локальний сервер – власне обладнання, встановлене у серверній кімнаті чи дата-центрі. Такий підхід, навіть на сьогоднішній день, має свої переваги, проте з появою хмарних платформ обмеження локальних серверів стали надто помітними, щоб їх ігнорувати.

Розглянемо наступний приклад: транспортна компанія має десять автобусів. Кожен з них генерує певну кількість даних, які використовуються для оперативної логістики, так і для майбутньої аналітики. З таким навантаженням, локальний сервер впорається без проблем. Але, якщо компанія вирішить придбати сотню нових автобусів і розширити зону покриття транспортом? Навантаження на сервер виросте кратно і постають важливі питання: як масштабувати обчислювальні потужності? Відповідь може бути не зовсім очевидною, адже встановлення і налаштування може означити нові виклики: необхідністю значних фінансових вливань на обладнання; найм спеціаліста, який і буде обслуговувати нові потужності; простій у роботі компанії, що може принести витрати і репутаційні збитки.

Тут на виручку приходять хмарні платформи. Така хмарна інфраструктура пропонує вирішити подібні проблеми через модель оплати за фактичним споживанням (pay-as-you-go). За такою моделлю компанія платить рівно за ті ресурси, які використовує в даний момент. Навантаження зростає – платформа автоматично масштабується. Спала – ресурси вивільняються. Подібний гнучкий підхід дозволяє поглибити гнучкість управління бізнесом, адже відпадає необхідністю постійного планування «наперед» з ймовірністю втратити кошти при невдалому розширенні бізнесу.

Основні відмінності хмари та локальної інфраструктури:

1. Витрати. Як було сказано раніше, покупка нового обладнання потребуватиме значних одноразових фінансових інвестицій. З іншого боку, хмара потребує оплати щомісячно, а у разі необхідності, потужності можна знизити, або зовсім від них відмовитись без переплати.
2. Підтримка. Локальна інфраструктура потребує окремого системного адміністратора, який налаштовуватиме обладнання, оновлюватиме програмне забезпечення, створюватиме резервні копії, тощо. Хмарний провайдер бере всі ці обов’язки на себе.
3. По-третє, доступність. Провідні хмарні провайдери гарантують SLA на рівні 99,9–99,99% часу роботи[9]. Для транспортних систем, де критично важливо знати місцезнаходження автомобіля в реальному часі, така гарантія значно перевищує те, що реально досяжне силами власної IT-служби.

Ринок хмарних IoT-платформ насичений – навіть серед рішень рівня «enterprise» можна нарахувати більше десяти гравців. Тому до вибору платформи варто підходити системно, визначивши наперед критерії відбору та необхідні параметри. Для системи моніторингу транспортних засобів, що розробляється в даній роботі, сформовано перелік вимог:

1. Функціональні вимоги: підтримка MQTT для передачі телеметрії; наявність механізму обробки повідомлень у реальному часі; можливість довгострокового зберігання та генерації звітів; наявність hot та cold paths; підтримка черг повідомлень для надійної доставки алертів.
2. Нефункціональні: можливість кратної масштабованості проєкту; низька затримка від надходження повідомлення до його збереження (не більше 5 секунд); наявність безкоштовного або постійного free tier для розгортання прототипу без додаткових витрат. Наявність або

можливість інтеграції Python SDK (симулятор транспортного засобу реалізується саме на цій мові).

2.2. Microsoft Azure

Microsoft Azure (рис. 2.1.) – одна з трьох найбільших хмарних платформ у світі поряд із Amazon Web Services та Google Cloud Platform. Для розробки IoT-рішень Azure приваблює завдяки інтеграції між власними компонентами: сервіси IoT Hub, Functions, Cosmos DB, SQL Database і Service Bus сумісні між собою «Out of the box» і не потребують налаштування міжсервісних з'єднань вручну.

Рисунок 2.1. Microsoft Azure

За даними Statista, Azure займає другу позицію на ринку хмарних послуг із часткою 21% (рис. 2.2.).

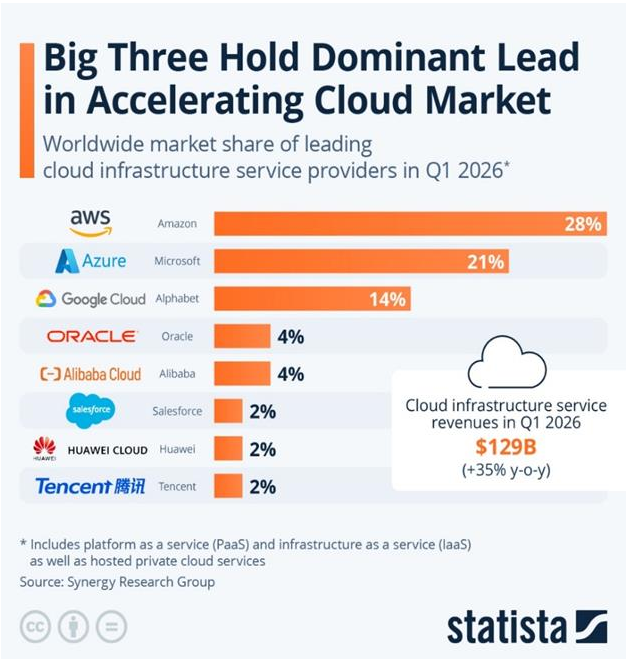


Рисунок 2.2. Статистика ринку хмарних платформ

При виборі компонентів Azure для даної роботи ключовим обмеженням є необхідність використання виключно безкоштовних рівнів сервісів. Це накладає певні обмеження: відмова від Azure Stream Analytics (сервіс потокової обробки) на користь Azure Functions.

Azure IoT Hub –керована служба, яка забезпечує надійний та безпечний двонаправлений зв’язок між пристроями Інтернету речей та серверною частиною застосунків(рис. 2.3.), Azure IoT Hub пропонує надійний обмін повідомленнями між пристроями та хмарою, забезпечує безпечний зв’язок за допомогою облікових даних безпеки та контролю доступу для кожного пристрою, а також включає бібліотеки пристроїв для найпопулярніших мов та платформ[10].

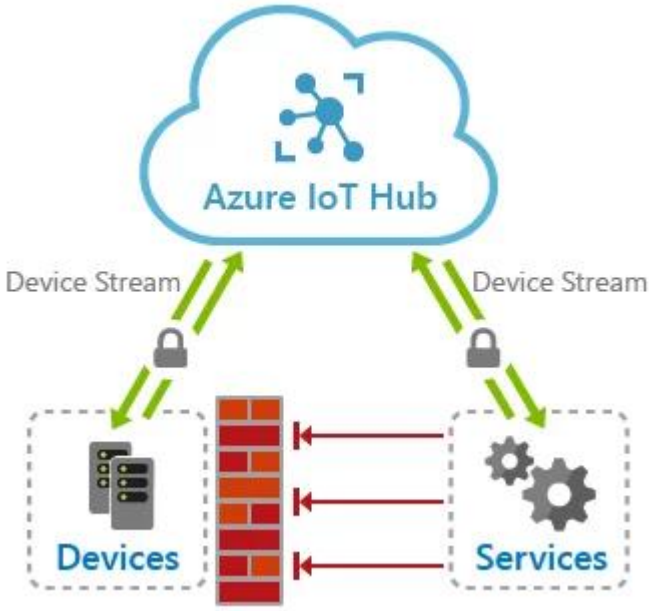


Рисунок 2.3. Microsoft Azure IoT Hub

Кожен пристрій реєструється в IoT Hub, отримує унікальний ідентифікатор і ключ аутентифікації, після чого може надсилати повідомлення через MQTT, AMQP або HTTP.

Для даної роботи використовується безкоштовний рівень F1, що дозволяє підключити один хаб з лімітом 8 000 повідомлень на добу. Важливий нюанс: повідомлення рахуються блоками по 0,5 кб. Якщо JSON-пакет телеметрії перевищує 512 байт – він займає два блоки і денний ліміт вдвічі скорочується.

IoT Hub також є сумісним із протоколом Event Hubs – і саме через цей сумісний endpoint Azure Functions підключаються до потоку телеметрії. Така архітектура дозволяє Functions «слухати» повідомлення від пристроїв без будь-якого проміжного компонента.

Також відмітимо, що Azure IoT Hub має вбудовану сумісність з протоколом Event Hubs. А наявність «built-in endpoint», сумісного з Event Hubs, дозволяє інтегрувати сервіс Azure Functions у потік даних. Таким чином, усувається необхідність у впровадженні проміжних брокерів повідомлень або додаткових черг, забезпечуючи пряму трансляцію даних від пристроїв до логіки обробки.

Azure Stream Analytics – це повністю керована служба аналітики в режимі реального часу, призначена для аналізу та обробки швидко переміщуваних потоків даних, які можна використовувати для отримання аналітичних відомостей, створення звітів або активації сповіщень та дій[11].

Stream Analytics використовує SQL-подібний синтаксис запитів. Він ідеально підходить для виробничих систем із тисячами пристроїв та яка потребує проведення складної аналітики в реальному часі. При виборі компонентів для обробки потокових даних було проведено порівняльний аналіз Azure Stream Analytics та Azure Functions, тут важливо відмітити серйозний нюанс Stream Analytics: даний сервіс не підтримує безкоштовний сервіс, а вимагає постійної підписки.

Згідно з актуальною моделлю тарифікації Azure Stream Analytics V2 (Standard), мінімальна конфігурація сервісу потребує виділення 1/3 обчислювального вузла (Streaming Unit V2). При вартості повного вузла у розмірі $0.405 за годину, мінімальні витрати на утримання сервісу складають $97.20 на місяць, незалежно від наявності чи відсутності вхідного трафіку[12].

Azure Functions вирішує ту ж задачу принципово інакше. Сервіс не використовує зарезервовані обчислювальні потужності, натомість працює у serverless-режимі та має подієво-орієнтовану архітектуру. Тобто функція запускається лише тоді, коли надходить повідомлення і зупиняється одразу після обробки.

Для даної роботи Functions виконують роль «Ingestor»:

- приймають повідомлення від IoT Hub,
- виконують валідацію та нормалізацію даних,
- перевіряють порогові значення,
- публікують алерт у чергу Service Bus.

У таблиці 2.1. наведемо порівняння сервісів Azure Stream Analytics та Azure Functions у контексті обробки даних.

Таблиця 2.1. Порівняння Azure Stream Analytics та Azure Functions для обробки IoT-телеметрії [13-14]

Критерій	**Azure Stream Analytics**	**Azure Functions**
Модель виконання	Постійно запущений процесор	Serverless (запуск за подією)
Вартість (мін.)	~$100/місяць завжди	Перший 1M викликів – $0
Віконні агрегації	Вбудовані (SQL-синтаксис)	Реалізуються вручну в коді
Затримка обробки	< 1 сек	1-3 сек
Складність налаштування	Середня (SQL)	Низька (Python)
Масштабування	Ручне (зміна SU)	Автоматичне
Підходить для прототипу	Ні (вартість)	Так (безкоштовно)
Підходить для виробництва	Так	Так (при належній логіці)

Service Bus – це хмарний сервіс обміну повідомленнями, що забезпечує надійну асинхронну комунікацію між компонентами системи. На відміну від

прямого виклику функції, черга Service Bus гарантує доставку повідомлення навіть у випадку тимчасової недоступності одержувача: повідомлення зберігається в черзі і буде оброблено, коли одержувач відновить роботу [15].

У системі моніторингу транспортних засобів Service Bus використовується як буфер для алертів. Коли «Ingestor» виявляє аномалію (наприклад, перевищення швидкості, критичну температуру двигуна, тощо), він публікує запис до черги Service Bus. Окрема функція «Notifier» підписана на цю чергу і відповідає за відправку сповіщення: додавання рядка до Google Sheets. У межах безкоштовного 12-місячного акаунту Azure, Service Bus Standard tier доступний 750 годин на місяць, що покриває потреби проєкту.

Cosmos DB – це глобально розподілена NoSQL база даних, оптимізована для операцій із низькою затримкою. Гарантована затримка читання менше 10 мс робить її ідеальним сховищем для «гарячих» даних: останніх позицій транспортних засобів, поточного стану датчиків, саме тих записів, до яких дашбоард звертається на постійній основі.

Важлива перевага Cosmos DB (рис. 2.4.) для даної роботи – наявність постійного безкоштовного рівня. На відміну від 12-місячних безкоштовних сервісів, Cosmos DB Free Tier діє безстроково і надає 1 000 RU/s та 25 ГБ сховища. Дані зберігаються у форматі JSON-документів, що спрощує логіку Function: отриманий пакет JSON записується в Cosmos DB практично без змін.

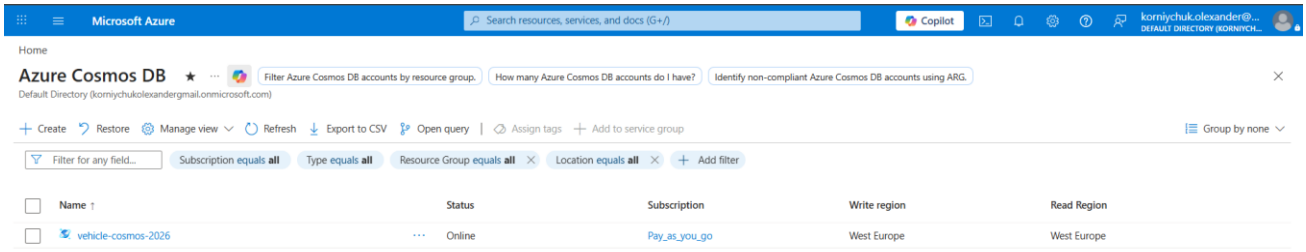


Рисунок 2.4. Microsoft Azure Cosmos DB

Разом із Cosmos DB у системі використовується Azure SQL Database (рис. 2.5.) для зберігання агрегованих даних. Якщо Cosmos DB зберігає кожен отриманий JSON-пакет, то SQL Database отримує лише підсумкові записи: середня

швидкість за годину, максимальна температура за добу, загальний пробіг, кількість алертів по кожному транспортному засобу, тощо.

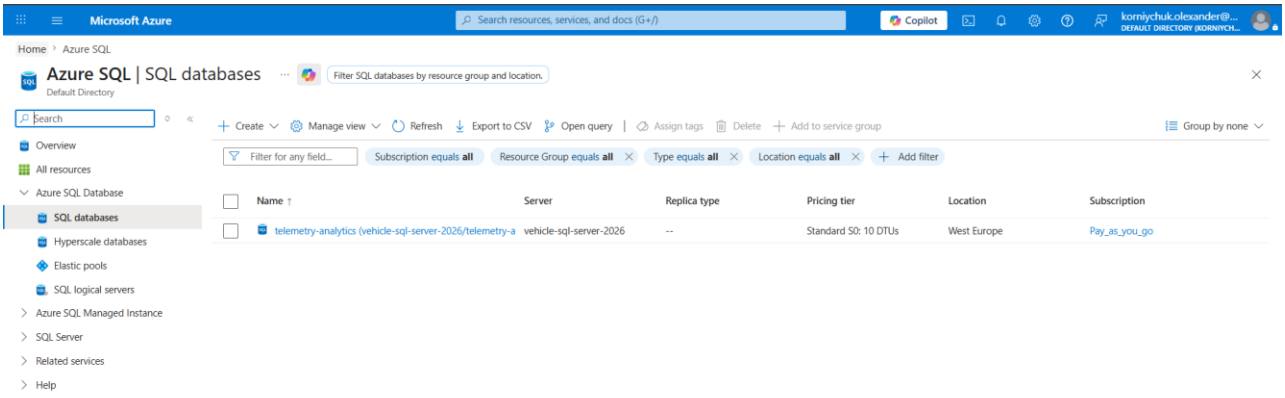


Рисунок 2.5. Microsoft Azure SQL

Саме ця реляційна база стає джерелом даних для Metabase – інструменту візуалізації, який значно краще працює зі структурованими SQL-таблицями, ніж із NoSQL-документами. В межах 12-місячного безкоштовного акаунту Azure SQL Database S0 (5 DTU) доступна 31 день на місяць – цього вистачає для безперервної роботи прототипу. Розмір бази при зберіганні лише агрегатів не перевищить кількох мегабайт навіть після тривалого тестування.

Metabase (рис. 2.6.) – це open-source платформа, що дозволяє будувати інтерактивні дашборди та виконувати SQL-запити. На відміну від Power BI, що потребує платної ліцензії для повноцінної хмарної публікації звітів, Metabase повністю безкоштовна у своїй Community Edition і розгортається як Docker-контейнер.

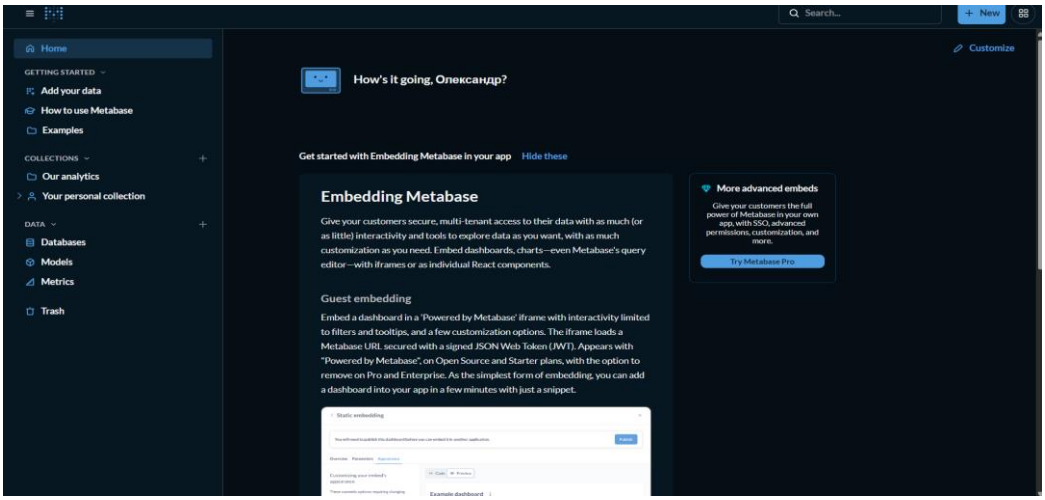


Рисунок 2.6. Головний екран Metabase

Для даної роботи Metabase запускається локально на через Docker (рис. 2.7.) і підключається до Azure SQL Database в хмарі.

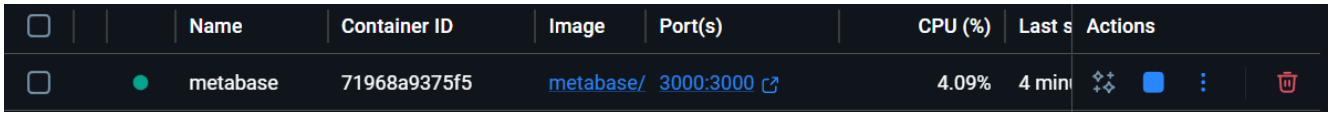


Рисунок 2.7. Запущений локально Metabase

Таблиця 2.2. Компоненти та їх роль у платформі моніторингу ТЗ[13-17]

Компонент	**Роль**	**Free tier**	**Ліміт**
IoT Hub F1	Прийом MQTT-телеметрії від транспортного засобу	Назавжди	8 000 пов./добу
Azure Functions	Serverless-обробка: валідація, нормалізація, перевірка порогів	Назавжди	1M викликів/місяць
Service Bus Standard	Черга алертів між Functions	12 місяців	750 год/місяць
Cosmos DB Free Tier	Зберігання сирої телеметрії (JSON)	Назавжди	1 000 RU/s, 25 ГБ
Azure SQL Database S0	Агреговані дані для Metabase	12 місяців	5 DTU, 250 МБ
Metabase	Дашборд: графіки, карта, таблиці	Open-source	Без обмежень
Google Sheets API	Журнал інцидентів (алерти)	Безкоштовно	Без обмежень

2.3. Огляд альтернативних хмарних платформ

Вибір Azure як основної платформи для даної роботи не означає, що альтернативи непридатні. Навпаки, кілька конкуруючих рішень мають власні помітні переваги і в іншому контексті вибір міг би бути іншим. Нижче розглянемо чотири найбільш релевантні альтернативи: AWS IoT Core, Google Cloud, ThingsBoard та Firebase.

Amazon Web Services (рис. 2.8.) – найбільший хмарний провайдер у світі і його AWS IoT Core є прямим функціональним аналогом Azure IoT Hub. Сервіс приймає повідомлення від пристроїв через MQTT, HTTP або WebSocket, надає реєстр пристроїв і Rules Engine для маршрутизації повідомлень. Безкоштовний рівень AWS IoT Core дозволяє обробляти 500 000 повідомлень на місяць.

Рисунок 2.8. Amazon Web Services

Проте є і складнощі. Цінова модель AWS IoT Core: оплата здійснюється окремо за кількість повідомлень і за кількість з’єднань. Сервіси AWS для побудови пайплайну (IoT Core – Kinesis – DynamoDB – QuickSight) не мають настільки зручного поєднання між собою: кожна інтеграція потребує явного налаштування прав IAM і ролей. Для новачка це означає додаткові витрати часу та ймовірні помилки при налаштуванні.

Google вийшов на ринок IoT-платформ дещо пізніше за конкурентів. Cloud IoT Core, запущений у 2017 році, надавав базові функції прийому телеметрії через MQTT і HTTP. Однак у серпні 2023 року Google оголосив про припинення

підтримки Cloud IoT Core (рис. 2.9.) і закрив сервіс, запропонувавши клієнтам міграцію до партнерських рішень.

Рисунок 2.9. Cloud IoT Core

ThingsBoard – платформа Інтернету речей з відкритим кодом для збору, обробки, візуалізації та управління пристроями даних. Дана платформа може розгортатись як у хмарі, так і локально, що надає гнучкість користувачу у виборі інфраструктури (рис. 2.10.).

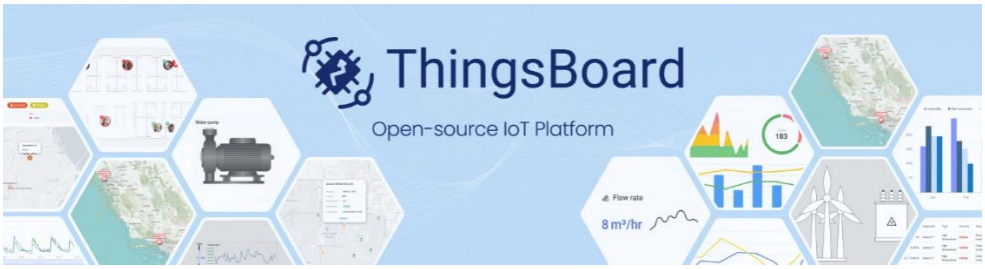


Рисунок 2.10. Сфери використання ThingsBoard

ThingsBoard підтримує: MQTT, HTTP, CoAP; платформа має гнучку систему правил із візуальним редактором; вбудований конструктор дашбордів із підтримкою карт.

Проте, слабким місцем ThingsBoard є масштабування. Розгортання широкої IoT-системи з тисячами пристроїв вимагають підключення та налаштування кластерів Kafka і Cassandra, що в свою чергу, складна задача навіть для DevOps-інженера. Якщо використовувати ThingsBoard для невеликих чи наукових проєктів, то повного контролю над інфраструктурою значно переважують переваги гнучкості.

Firebase від Google, як хмарна платформа, орієнтована на швидку розробку та масштабування застосунків. Не дивлячись на те, що Firebase першочергово створений для мобільних і веб-додатків, його структура даних, орієнтована на події модель та інтеграція з Google Cloud дозволяють використовувати цей сервіс як легку backend-платформу для IoT-проєктів.

Але Firebase не проектувався для IoT-систем. MQTT не підтримується нативно, відсутній реєстр пристроїв, немає механізму черг для алертів. Підключення значної кількості пристроїв у IoT-мережу потенційно впирається у технічні обмеження платформи.

Вибір Azure як основної платформи ґрунтується на трьох факторах:

1. Наявність free tier для сервісів Cosmos DB і Functions, що дозволяє реалізувати прийом і обробку телеметрії без будь-яких витрат.
2. Service Bus, доступний у межах 12-місячного безкоштовного акаунту, що забезпечує чергу алертів і дозволяє реалізувати event-driven архітектуру з кількох функцій.
3. Якість офіційної документації Microsoft та готові приклади для Python SDK, що дозволяє скоротити час на розгортання і дозволяє знайти ймовірне вирішення проблем у разі їх виникнення.

РОЗДІЛ 3
АНАЛІЗ СУЧАСНИХ РІШЕНЬ ІНТЕРНЕТУ РЕЧЕЙ ТА ПРОЄКТУВАННЯ ПЛАТФОРМИ МОНІТОРИНГУ ТРАНСПОРТНОГО ЗАСОБУ

3.1. Огляд сучасних IoT-рішень моніторингу транспортних засобів

Донедавна системи моніторингу транспорту працювали, в основному, з GPS- трекерами. Серед основних завдань: визначення місцеположення авто і передача даної інформації диспетчеру. Проте, технологічний розвиток змінює правила гри і сучасний бізнес мусить впроваджувати комплексні IoT-проєкти та розбудовувати бази даних, щоб мати стратегічну перевагу. Які саме платформи наккраще підходять для подібних задач?

Однією з відоміших «екосистем» у цій сфері сьогодні є Samsara. Компанія пропонує до використання телематичні шлюзи Vehicle Gateway, які виступають центральним хабом для зібраної телеметрії, а також інтеграцію комп'ютерного зору (з допомогою AI-камер) з даними CAN-шини. Такий підхід дозволяє проводити серйознішу аналітику, наприклад виявляти ознаки втоми у водія чи використання мобільного телефону за кермом [18].

Інший гравець на ринку, Geotab, використовує інший підхід: глибока орієнтація на телематику та аналітику автопарку. Основою продукту є пристрої серії GO. Вони підключаються до OBD-II або CAN-шини автомобіля та зчитують технічні параметри транспортного засобу, далі ці дані передаються до хмарної системи MyGeotab, де використовуються для побудови аналітики. Платформа орієнтована на великі автопарки та обробку значних обсягів телематичних даних [19].

Проте, є ще невеликі гравці, основна ніша яких –GPS моніторинг та диспетчеризація. Наприклад, Verizon Connectта подібні системи, які використовуються невеликим й середнім автобізнесом для контролю маршрутів, швидкості руху, місцезнаходження автомобіля та часу простою. Серед основних переваг: швидкість та низька ціна впровадження.

3.2. Формування технічних вимог до проєктованої платформи

Будь-який IT проєкт починається не з вибору платформи і не з написання коду, він починається з аналізу та формування вимог. Саме ці вимоги задають напрям, обмежують простір рішень і є єдиним об’єктивним критерієм для оцінки результату. Вимоги до платформи моніторингу транспортних засобів сформовані з урахуванням ймовірних задач у сфері автотранспорту і обмежень безкоштовних рівнів Azure.

Означимо функціональні вимоги:

1. Прийом телеметрії. Хмара має отримувати дані від транспортних засобів у реальному часі через MQTT. Кожне повідомлення має структуру JSON з набором параметрів: GPS-позиція, стан двигуна, паливо, електрика, шини, діагностичні коди, тощо.
2. Обробка та зберігання. Система повинна проводити валідацію кожного вхідного пакету (перевіряти наявність обов’язкових полів, діапазони числових значень, цілісність HMAC-підпису) і зберігати валідні дані у хмарному сховищі. Некоректні або підроблені пакети відхиляються з відповідним записом у лог Application Insights.
3. Виявлення аномалій. Кожне повідомлення (окрім запису в базу даних) проходить перевірку на відхилення від норми: перевищення порогових значень (швидкість, температура двигуна, тиск у шинах тощо) та статистичну перевірку на основі z-score для ключових параметрів. При виявленні аномалії – формується алерт і передається далі по «pipeline».
4. Сповіщення. Кожен алерт фіксується у журналі інцидентів (через інтеграцію з Google Sheets) і в реляційній базі даних Azure SQL для подальшої аналітики.
5. Візуалізація. Дані відображаються через платформу Metabase із графіками, таблицями, індикаторами, тощо.

Оформимо основні функціональні вимоги у таблицю 3.1.

Таблиця 3.1. Функціональні вимоги до системи моніторингу ТЗ

Вимога	**Пріоритет**	**Критерій виконання**
Прийом MQTT-телеметрії від ТЗ	Критичний	Повідомлення надходить до IoT Hub < 1 сек після відправки
Валідація та перевірка HMAC-підпису	Критичний	Некоректні/підроблені пакети відхиляються, а валідні записуються
Виявлення аномалій (порогові значення, z-score)	Критичний	Аномалія виявляється і передається в чергу ≤ 3 сек
Фіксація інцидентів у Google Sheets та Azure SQL	Високий	Рядок з’являється ≤ 10 сек після виявлення аномалії
Дашборд Metabase з графіками і таблицями	Високий	Дані оновлюються на основі актуальних записів у Azure SQL
Паралельна робота кількох симульованих ТЗ	Середній	>3 пристроїв одночасно без втрати повідомлень

Означимо нефункціональні вимоги:

1. Продуктивність. Загальна затримка від відправки повідомлення симулятором до його запису в Cosmos DB не більше 5 секунд. Для дашборду Metabase допустима затримка вища (він відображає дані з Azure SQL).
2. Надійність. Втрата повідомлень не повинна перевищувати 1% від загального обсягу. Це забезпечується механізмом QoS 1 протоколу MQTT. Для алертів між компонентами системи надійність гарантує черга Azure Service Bus.

3. Безпека. TLS 1.2 для шифрування каналу, SAS-токени для автентифікації пристрою і HMAC-SHA256 підпис кожного повідомлення для гарантії цілісності даних. Детально – в підрозділі 3.5.
4. Вартість розгортання. Усі компоненти функціонують у межах безкоштовних рівнів Azure. Це обмеження вплинуло на архітектурні рішення: Azure Functions замість Stream Analytics, Metabase замість Power BI.

3.3. Загальна архітектура системи

Система побудована за принципом «event-driven pipeline»: кожна подія (вхідне MQTT-повідомлення) ініціює ланцюжок обробки. Компоненти не викликають один одного напряму, сервіси передають телеметрію через брокери подій IoT Hub та Service Bus.

Організація системи розподілена на наступні рівні:

1. Програма-симулятор, що написана мовою Python. Вона генерує телеметрію по заданому транспортному засобу і публікує її через MQTT з підписом HMAC-SHA256.
2. Azure IoT Hub рівня F1. Він приймає MQTT-повідомлення, аутентифікує пристрої за SAS-токенами і поміщає їх у внутрішню чергу. Саме з цієї черги читає функція Ingestor.
3. Три Azure Functions у рамках одного Function App:
    a. Ingestor: здійснює валідацію та збереження даних з IoT Hub, паралельно ініціюючи пошук аномалій через клас AnomalyDetector. Виявлені інциденти публікуються в чергу Service Bus.
    b. AnomalyDetectorFn: виконує класифікацію алертів, зчитуючи їх із першої черги та пересилаючи до другої.

c. Notifier: реалізує фінальну фіксацію класифікованих подій у Google Sheets та Azure SQL.

4. Metabase, що підключається до Azure SQL, з допомогою налаштованого дашборду, візуалізує телеметрію.

3.4. Модель даних та формат повідомлень

Кожне повідомлення симулятора – це JSON-об’єкт (рис. 3.1.). Структура фіксована і перевіряється validators.py при кожному вхідному пакеті: відсутність будь-якої секції або обов’язкового поля призводить до відхилення.


```
"device_id": "vehicle-001",
"timestamp": "2026-05-14T13:04:57Z",
"trip_id": "930d4d3c-5ace-48a7-9a62-2879b936b133",
"gps": {
  "latitude": 50.450071,
  "longitude": 30.523299,
  "speed_kmh": 9.4,
  "heading": 248.9,
  "altitude_m": 165.7,
  "satellites": 12
},
"engine": {
  "rpm": 894,
  "temp_coolant_c": 23.1,
  "temp_oil_c": 30.8,
  "runtime_sec": 3,
  "load_pct": 1.3
},
"fuel": {
  "level_pct": 58.0,
  "consumption_l_per_100km": 7.34,
  "range_km": 434.7
},
"electrical": {
  "voltage_v": 13.26,
  "alternator_load_pct": 13.4
},
"environment": {
  "temp_outside_c": 12.0,
  "temp_inside_c": 18.1,
  "humidity_pct": 65.8
},
```


Рисунок 3.1. Частина JSON-об’єкту з телеметрією

Варто відзначити, IoT Hub F1 рахує повідомлення блоками по 0,5 КБ, тому, при перевищенні, ліміт знижується з 8000 до 4000 повідомлень на добу.

У Cosmos DB кожен пакет зберігається як окремий документ. Ingestor-функція перед записом прибирає поле msg_signature і додає поле id, як комбінацію device_id і timestamp, що забезпечує унікальність.

Azure SQL зберігає два типи записів. Таблиця telemetry_aggregates (рис. 3.2.) отримує дані безпосередньо з Ingestor: при кожному валідному повідомленні функція викликає upsert_telemetry_aggregate і записує ключові параметри пакету.

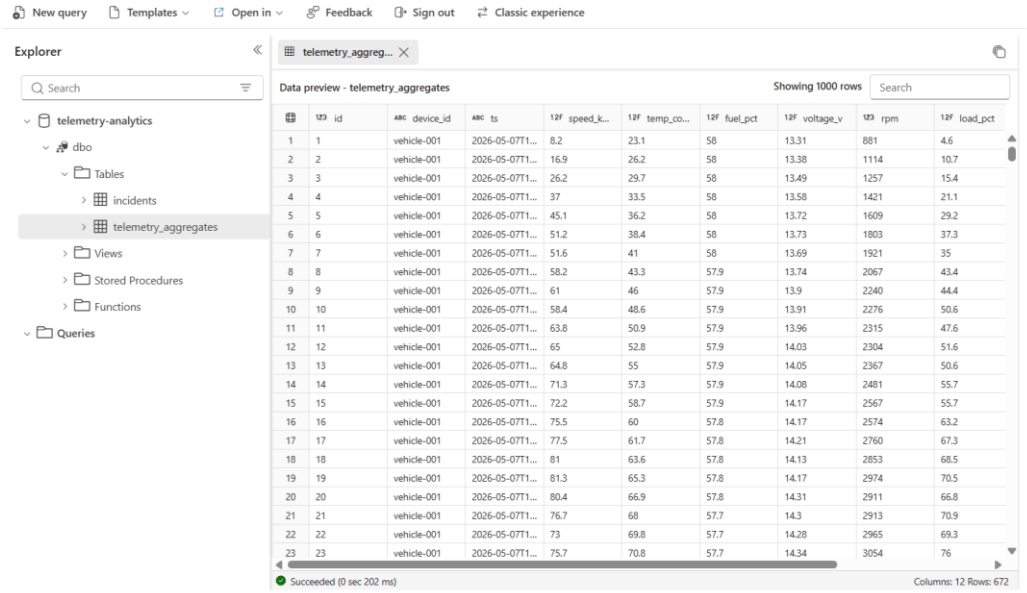


Рисунок 3.2. Таблиця telemetry_aggregates

Таблиця incidents (рис. 3.3.) заповнюється Notifier-функцією при кожному отриманому алерті. Ці дві таблиці є єдиним джерелом для Metabase.

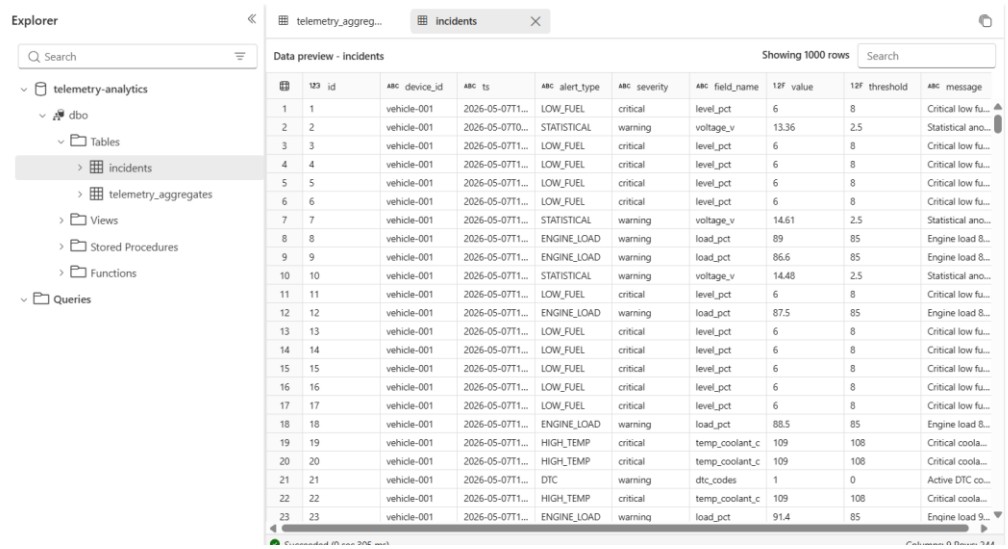


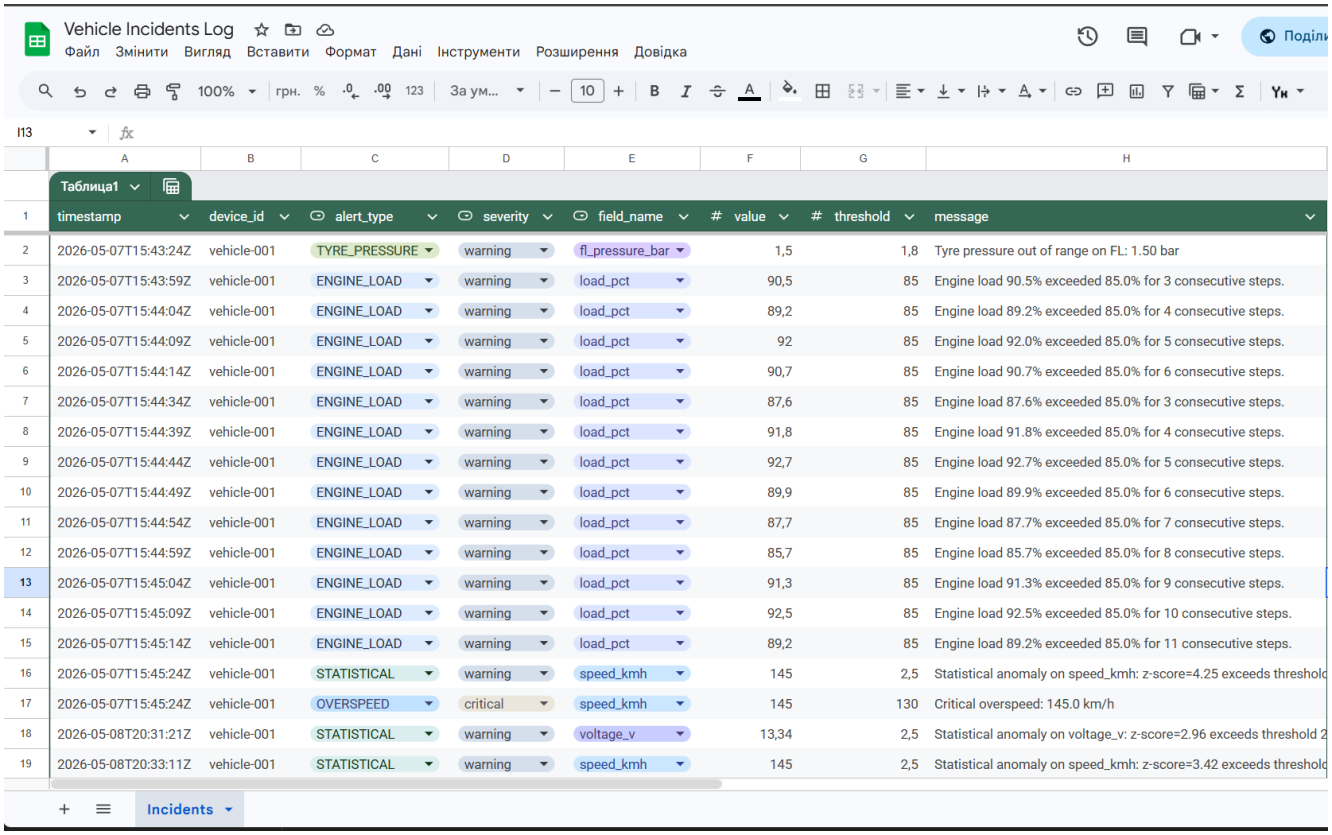
Рисунок 3.3. Таблиця incidents

«Pipeline» реалізовано у трьох Azure Functions в рамках однієї програми. Логіка виявлення аномалій зосереджена у класі AnomalyDetector і виконується в Ingestor при кожному вхідному повідомленні. Дві інші Functions відповідають за маршрутизацію алертів і запис результатів.

Ingestor запускається тригером EventHubTrigger, прив’язаним до Event Hub-сумісного endpoint IoT Hub. Функція обробляє одне повідомлення за виклик і виконує такі кроки:

1. Верифікація підпису. Клас MessageSigner забезпечує цілісність даних шляхом формування цифрового підпису за алгоритмом HMAC-SHA256. Для запобігання помилок валідації через різний порядок полів у повідомленні, вхідний JSON-об’єкт попередньо приводиться до канонічного вигляду. Якщо підписи не збігаються – повідомлення відхиляється.
2. Схемна валідація через validate_payload(). Функція перевіряє наявність обов’язкових секцій і полів, типи даних. При будь-якій помилці – ValidationError і повернення без запису.
3. Виявлення аномалій. Клас AnomalyDetector викликається методом detect() і виконує три незалежні перевірки: порогові значення, z-score для ключових параметрів і відстеження тривалого перевантаження двигуна.
4. Запис і маршрутизація. Верифікований пакет очищується від підпису та паралельно маршрутизується: у Cosmos DB (upsert_item) для збереження стану, в Azure SQL (upsert_telemetry_aggregate) для агрегації статистики та, за умови виявлення аномалій, у чергу Service Bus для генерації миттєвих сповіщень.

Notifier підписаний на чергу «alerts-classified» і виконує два незалежних записи при кожному алерті. Перший – до таблиці incidents в Azure SQL через pyodbc. Другий – до Google Sheets (рис. 3.4.).



	timestamp	device_id	alert_type	severity	field_name	# value	# threshold	message
2	2026-05-07T15:43:24Z	vehicle-001	TYRE_PRESSURE	warning	fl_pressure_bar	1,5	1,8	Tyre pressure out of range on FL: 1.50 bar
3	2026-05-07T15:43:59Z	vehicle-001	ENGINE_LOAD	warning	load_pct	90,5	85	Engine load 90.5% exceeded 85.0% for 3 consecutive steps.
4	2026-05-07T15:44:04Z	vehicle-001	ENGINE_LOAD	warning	load_pct	89,2	85	Engine load 89.2% exceeded 85.0% for 4 consecutive steps.
5	2026-05-07T15:44:09Z	vehicle-001	ENGINE_LOAD	warning	load_pct	92	85	Engine load 92.0% exceeded 85.0% for 5 consecutive steps.
6	2026-05-07T15:44:14Z	vehicle-001	ENGINE_LOAD	warning	load_pct	90,7	85	Engine load 90.7% exceeded 85.0% for 6 consecutive steps.
7	2026-05-07T15:44:34Z	vehicle-001	ENGINE_LOAD	warning	load_pct	87,6	85	Engine load 87.6% exceeded 85.0% for 3 consecutive steps.
8	2026-05-07T15:44:39Z	vehicle-001	ENGINE_LOAD	warning	load_pct	91,8	85	Engine load 91.8% exceeded 85.0% for 4 consecutive steps.
9	2026-05-07T15:44:44Z	vehicle-001	ENGINE_LOAD	warning	load_pct	92,7	85	Engine load 92.7% exceeded 85.0% for 5 consecutive steps.
10	2026-05-07T15:44:49Z	vehicle-001	ENGINE_LOAD	warning	load_pct	89,9	85	Engine load 89.9% exceeded 85.0% for 6 consecutive steps.
11	2026-05-07T15:44:54Z	vehicle-001	ENGINE_LOAD	warning	load_pct	87,7	85	Engine load 87.7% exceeded 85.0% for 7 consecutive steps.
12	2026-05-07T15:44:59Z	vehicle-001	ENGINE_LOAD	warning	load_pct	85,7	85	Engine load 85.7% exceeded 85.0% for 8 consecutive steps.
13	2026-05-07T15:45:04Z	vehicle-001	ENGINE_LOAD	warning	load_pct	91,3	85	Engine load 91.3% exceeded 85.0% for 9 consecutive steps.
14	2026-05-07T15:45:09Z	vehicle-001	ENGINE_LOAD	warning	load_pct	92,5	85	Engine load 92.5% exceeded 85.0% for 10 consecutive steps.
15	2026-05-07T15:45:14Z	vehicle-001	ENGINE_LOAD	warning	load_pct	89,2	85	Engine load 89.2% exceeded 85.0% for 11 consecutive steps.
16	2026-05-07T15:45:24Z	vehicle-001	STATISTICAL	warning	speed_kmh	145	2,5	Statistical anomaly on speed_kmh: z-score=4.25 exceeds threshold
17	2026-05-07T15:45:24Z	vehicle-001	OVERSPEED	critical	speed_kmh	145	130	Critical overspeed: 145.0 km/h
18	2026-05-08T20:31:21Z	vehicle-001	STATISTICAL	warning	voltage_v	13,34	2,5	Statistical anomaly on voltage_v: z-score=2.96 exceeds threshold 2
19	2026-05-08T20:33:11Z	vehicle-001	STATISTICAL	warning	speed_kmh	145	2,5	Statistical anomaly on speed_kmh: z-score=3.42 exceeds threshold

Рисунок 3.4. Інтеграція з Google Sheets

Порогові перевірки покривають очевидні порушення, але не бачать аномалій (наприклад, різких відхилень у технічно допустимих межах). Для цього AnomalyDetector застосовує метод z-score.

Для параметрів швидкості, температури охолодження і напруги, детектор накопичує чергу з останніх ZSCORE_WINDOW_SIZE = 20 вимірювань. При кожному новому значенні обчислюється за формулою (3.1):

$$z = \frac{|x - \mu|}{\sigma} \qquad (3.1)$$

Деμ – середнє арифметичне значень телеметрії у часовому вікні;**σ**– стандартне відхилення показників.

Якщо z > 2.5 і стандартне відхилення перевищує 1×10^{-9} (тобто вікно містить різноманітні значення) – генерується алерт типу STATISTICAL. Вікно розміром 20 вимірювань при частоті раз на 30 секунд відповідає 10 хвилинам спостережень – достатньо для виявлення коротких аномалій.

3.5. Безпека системи

Весь трафік між симулятором і Azure IoT Hub шифрується через TLS 1.2 на порту 8883. IoT Hub вимагає TLS для всіх підключень і не приймає незашифровані з’єднання на порту 1883.

Після встановлення зашифрованого каналу IoT Hub перевіряє право пристрою на підключення через SAS-токен.

TLS і SAS-токен не захищають від підробки даних всередині з’єднання. Тому, кожне повідомлення підписується на рівні payload. Клас MessageSigner серіалізує повідомлення у JSON і обчислює HMAC-SHA256 від отриманого рядка з використанням MESSAGE_SIGNING_SECRET (рис. 3.5.). Підпис додається у поле msg_signature.

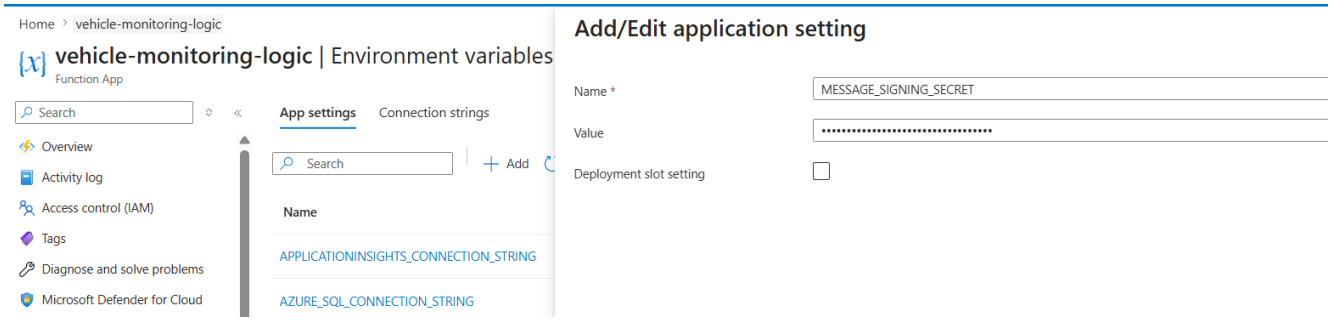


Рисунок 3.5. Конфігурація секретних ключів

РОЗДІЛ 4
РЕАЛІЗАЦІЯ IOT-ПЛАТФОРМИ ЗБОРУ ТА АНАЛІТИКИ ТРАНСПОРТНИХ ДАНИХ У ХМАРНІЙ ІНФРАСТРУКТУРІ

4.1. Структура проєкту

Проєкт організовано у два незалежних модулі з різним функціональним призначенням. Таке розділення дозволяє розробляти і тестувати симулятор незалежно від хмарних компонентів і навпаки: достатньо підняти лише один модуль. Залежності Python зафіксовані у окремих файлах requirements.txt для кожного модуля – це дозволяє уникнути конфліктів версій між симулятором і Functions. Конфігурація зберігається у файлах .env, які не потрапляють до системи контролю версій. Структуру файлів проекту наведено у таблиці 4.1.

Таблиця 4.1. Структура файлів проєкту

Файл	Модуль	Призначення
vehicle_simulator.py	simulation/	Клас VehicleSimulator: генерація фізично телеметрії
telemetry_publisher.py	simulation/	Клас TelemetryPublisher: MQTT-з'єднання з IoT Hub через TLS
security.py	simulation/ та functions/	SasTokenGenerator та MessageSigner – модуль безпеки
main.py	simulation/	Точка входу: CLI-аргументи, threading для кількох пристроїв
config.py	simulation/	Константи симулятора: фізичні коефіцієнти, порогові значення
routes/kyiv_route.py	simulation/	GPS-маршрут Київ: список Waypoint з координатами і висотою

function_app.py	functions/	Три Azure Functions: Ingestor, AnomalyDetectorFn, Notifier
anomaly.py	functions/	Клас AnomalyDetector: порогові перевірки, z-score, ENGINE_LOAD
validators.py	functions/	Функція validate_payload: схемна валідація і перевірка діапазонів
notifier_sheets.py	functions/	append_incidents_batch: запис алертів у Google Sheets
notifier_sql.py	functions/	upsert_telemetry_aggregate, insert_incident: запис в Azure SQL
config.py	functions/	Порогові значення аномалій, назви таблиць і черг

Проаналізувавши таблицю, відзначимо, що модуль simulation/ містить симулятор транспортного засобу і є автономним Python-застосунком, що запускається локально. Другий модуль – functions/ – містить логіку для Azure Functions і розгортається у хмару. Модулі написані мовою Python.

4.2. Симулятор транспортного засобу

Симулятор відтворює рух транспортного засобу у м. Київ. Маршрут визначено у routes/kyiv_route.py як список об’єктів Waypoint – іменованих кортежів з полями latitude, longitude, altitude_m і name. Усього 25 точок з координатами WGS-84.

Клас VehicleSimulator реалізований як Python dataclass. Внутрішній стан – поточна швидкість, температура, рівень палива, координати тощо – зберігається у приватних полях між кроками симуляції. При кожному виклику методу step() симулятор просуває автомобіль вздовж маршруту, оновлює всі параметри з урахуванням фізичних залежностей і повертає готовий словник телеметрії.

Просування по маршруту реалізоване через ping-pong: досягнувши останньої точки, автомобіль починає рух у зворотному напрямку.

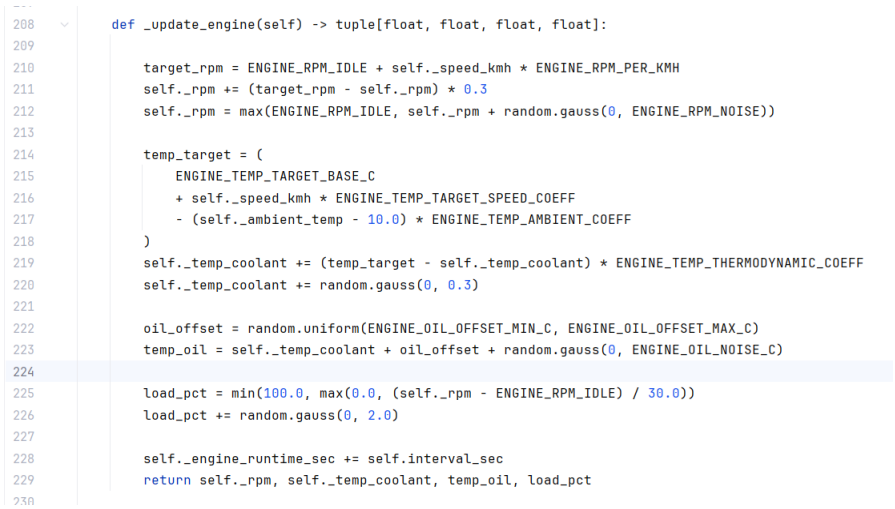
Ключовою особливістю симулятора є те, що параметри не є незалежними. Вони пов’язані. Це дуже важливо для тестування алгоритмів виявлення аномалій: якщо параметри не пов’язані між собою, детектор не зможе відрізнити справжню аномалію від випадкового збігу незалежних значень.

Швидкість і RPM пов’язані лінійною залежністю: цільове значення обертів двигуна обчислюється як target_rpm = ENGINE_RPM_IDLE + speed x ENGINE_RPM_PER_KMH, де ENGINE_RPM_IDLE =800 об/хв, ENGINE_RPM_PER_KMH = 28. Реальне значення наближається до цілі з коефіцієнтом інерції 0.3 і статистичним шумом N(0, 50). Отже при швидкості 60 км/год очікуваний RPM становить приблизно 2480 об/хв.

Температура двигуна прямує до рівноважного значення з термодинамічним коефіцієнтом 0.05 (за кожен крок різниця між поточною і цільовою температурою зменшується на 5%). Рівноважна температура залежить від швидкості (більше навантаження – більше тепла) і зовнішньої температури (холодне повітря охолоджує).

Напруга бортової мережі залежить від RPM: при холостому ходу генератор дає 13.2 В, при підвищених обертах – до 14.4 В. Витрата палива залежить від швидкості, а рівень палива зменшується на кожному кроці пропорційно пройденій відстані.

На рисунку 4.1. представлено програмну реалізацію методу, що відповідає за розрахунок динамічних параметрів двигуна.



```python
def _update_engine(self) -> tuple[float, float, float, float]:

    target_rpm = ENGINE_RPM_IDLE + self._speed_kmh * ENGINE_RPM_PER_KMH
    self._rpm += (target_rpm - self._rpm) * 0.3
    self._rpm = max(ENGINE_RPM_IDLE, self._rpm + random.gauss(0, ENGINE_RPM_NOISE))

    temp_target = (
        ENGINE_TEMP_TARGET_BASE_C
        + self._speed_kmh * ENGINE_TEMP_TARGET_SPEED_COEFF
        - (self._ambient_temp - 10.0) * ENGINE_TEMP_AMBIENT_COEFF
    )
    self._temp_coolant += (temp_target - self._temp_coolant) * ENGINE_TEMP_THERMODYNAMIC_COEFF
    self._temp_coolant += random.gauss(0, 0.3)

    oil_offset = random.uniform(ENGINE_OIL_OFFSET_MIN_C, ENGINE_OIL_OFFSET_MAX_C)
    temp_oil = self._temp_coolant + oil_offset + random.gauss(0, ENGINE_OIL_NOISE_C)

    load_pct = min(100.0, max(0.0, (self._rpm - ENGINE_RPM_IDLE) / 30.0))
    load_pct += random.gauss(0, 2.0)

    self._engine_runtime_sec += self.interval_sec
    return self._rpm, self._temp_coolant, temp_oil, load_pct
```

Рисунок 4.1. Фрагмент коду з алгоритмом математичного моделювання динаміки параметрів двигуна транспортного засобу

Для тестування виявлення аномалій симулятор підтримує режим --inject-anomalies. При запуску з цим прапором через кожні задані N хвилин активується одна випадкова аномалія з п’яти можливих: OVERSPEED (speed = 145 км/год на 3 кроки), HIGH_TEMP (temp_coolant = 109 C на 5 кроків), LOW_FUEL (fuel_pct = 6%, незворотньо), TYRE_PRESSURE (fl_pressure = 1.5 бар на 1 крок), DTC (код P0101 на 5 кроків). Такий підхід дозволяє проводити тестування як логіки детектора, а також те, чи дійшло аномальне значення через мережу до хмари і було інтерпретоване.

Приклад згенерованої аномалії наведено на рисунку 4.2.

```
2026-05-15 02:07:37,886 [WARNING] vehicle_simulator: Injecting anomaly: HIGH_TEMP
2026-05-15 02:07:37,887 [INFO] __main__: [vehicle-001] FULL PAYLOAD: {
  "device_id": "vehicle-001",
  "timestamp": "2026-05-14T23:07:37Z",
  "trip_id": "3effbfd3-6e9d-4a25-a924-3fca3cf4e148",
  "gps": {
    "latitude": 50.449771,
    "longitude": 30.522248,
    "speed_kmh": 64.4,
    "heading": 251.8,
    "altitude_m": 163.3,
    "satellites": 8
  },
  "engine": {
    "rpm": 2505,
    "temp_coolant_c": 109.0,
    "temp_oil_c": 115.0,
    "runtime_sec": 60,
    "load_pct": 56.9
```

Рисунок 4.2. Приклад вихідних даних симулятора при ініціації критичного стану двигуна

Клас TelemetryPublisher відповідає за встановлення MQTT-з’єднання і публікацію повідомлень.

4.3. Реалізація Azure Functions

Три Functions розміщені в одному Function App на Python 3.11. Весь код functions/ розгортається в хмару однією командою func azure functionapp publish vehicle-monitoring-logic --python. Перевагою такого підходу є те, що локальне тестування і хмарне розгортання використовують один і той самий код.

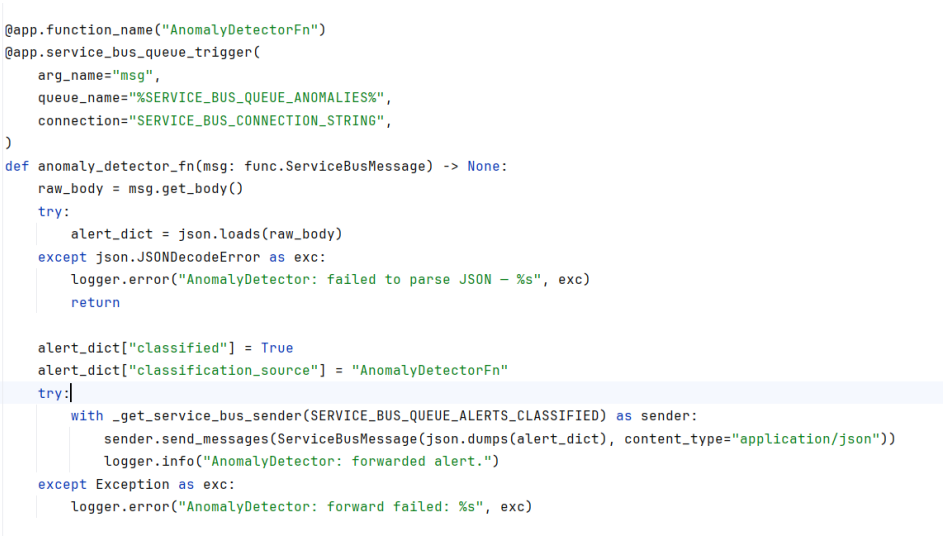
Ingestor (рис. 4.3.) є першою і найважливішою з трьох функцій. Вона обробляє кожне повідомлення від кожного пристрою. Логіка функції задається декораторами function_name (ім’я функції) та event_hub_message_trigger (тригер вхідних повідомлень).

```python
@app.function_name("Ingestor")
@app.event_hub_message_trigger(
    arg_name="event",
    event_hub_name="%IOTHUB_EVENT_HUB_NAME%",
    connection="IOTHUB_EVENT_HUB_CONNECTION_STRING",
    cardinality="one",
    consumer_group="$Default",
)
def ingestor(event: func.EventHubEvent) -> None:
    raw_body = event.get_body()
    try:
        payload = json.loads(raw_body)
    except json.JSONDecodeError as exc:
        logger.error("Ingestor: failed to parse JSON — %s", exc)
        return

    device_id = payload.get("device_id", "unknown")
    signer = _get_message_signer()
    if not signer.verify(payload):
        logger.warning("Ingestor: HMAC verification failed for device '%s'.", device_id)
        return
```

Рисунок 4.3. Фрагмент коду функції Ingestor: прийом, парсинг та перевірка цілісності даних

AnomalyDetectorFn (рис. 4.4.) підписана на чергу anomalies через service_bus_queue_trigger. Її логіка наступна: функція збагачує отриманий алерт двома полями (classified: True і classification_source: AnomalyDetectorFn) і пересилає до черги alerts-classified. Такий дизайн залишає місце для майбутнього розширення.



```python
@app.function_name("AnomalyDetectorFn")
@app.service_bus_queue_trigger(
    arg_name="msg",
    queue_name="%SERVICE_BUS_QUEUE_ANOMALIES%",
    connection="SERVICE_BUS_CONNECTION_STRING",
)
def anomaly_detector_fn(msg: func.ServiceBusMessage) -> None:
    raw_body = msg.get_body()
    try:
        alert_dict = json.loads(raw_body)
    except json.JSONDecodeError as exc:
        logger.error("AnomalyDetector: failed to parse JSON — %s", exc)
        return

    alert_dict["classified"] = True
    alert_dict["classification_source"] = "AnomalyDetectorFn"
    try:
        with _get_service_bus_sender(SERVICE_BUS_QUEUE_ALERTS_CLASSIFIED) as sender:
            sender.send_messages(ServiceBusMessage(json.dumps(alert_dict), content_type="application/json"))
            logger.info("AnomalyDetector: forwarded alert.")
    except Exception as exc:
        logger.error("AnomalyDetector: forward failed: %s", exc)
```

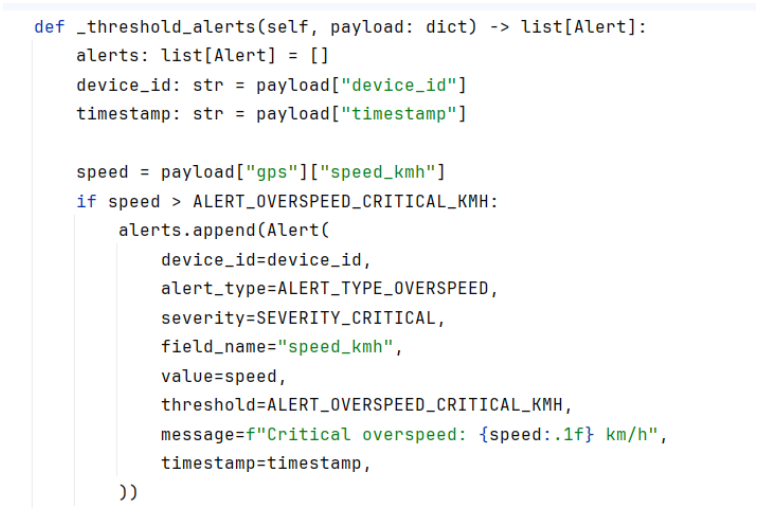
Рисунок 4.4. Програмна логіка модуля AnomalyDetectorFn: збагачення даних та маршрутизація подій

Функція Notifier підписана на чергу alerts-classified. При кожному виклику вона виконує два незалежних записи: insert_incident до таблиці incidents в Azure SQL і append_incidents_batch до Google Sheets. Незалежність записів реалізована через окремі блоки try/except – помилка одного запису не перериває інший. В такому випадку, якщо Google Sheets API тимчасово недоступний, дані все одно потраплять у базу даних.

4.4. Модуль виявлення аномалій

Клас AnomalyDetector в anomaly.py об’єднує три стратегії виявлення, що виконуються послідовно при кожному виклику detect().

Метод _threshold_alerts() (рис. 4.5.) проходить по всіх параметрах і порівнює їх з константами з config.py. Логіка наступна: для OVERSPEED і HIGH_TEMP спочатку перевіряється критичний поріг, потім – попередній рівень попередження. Для LOW_FUEL – навпаки, оскільки значення нижче критичного важливіше.



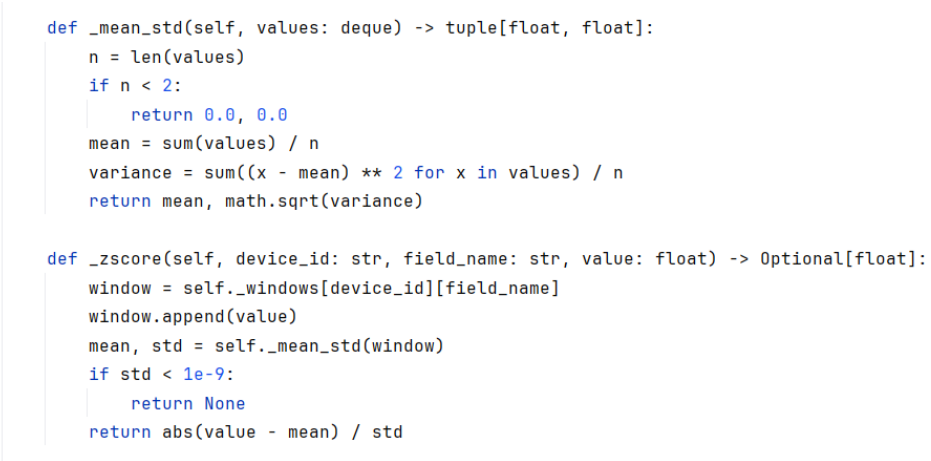
```python
def _threshold_alerts(self, payload: dict) -> list[Alert]:
    alerts: list[Alert] = []
    device_id: str = payload["device_id"]
    timestamp: str = payload["timestamp"]

    speed = payload["gps"]["speed_kmh"]
    if speed > ALERT_OVERSPEED_CRITICAL_KMH:
        alerts.append(Alert(
            device_id=device_id,
            alert_type=ALERT_TYPE_OVERSPEED,
            severity=SEVERITY_CRITICAL,
            field_name="speed_kmh",
            value=speed,
            threshold=ALERT_OVERSPEED_CRITICAL_KMH,
            message=f"Critical overspeed: {speed:.1f} km/h",
            timestamp=timestamp,
        ))
```

Рисунок 4.5. Фрагмент коду модуля аналітики для ідентифікації перевищення швидкості

Метод _engine_load_alerts() відстежує кількість послідовних кроків, де engine.load_pct перевищує 85%. Лічильник для кожного device_id зберігається у _engine_load_streak. Алерт ENGINE_LOAD формується лише якщо лічильник досяг ALERT_ENGINE_LOAD_CONSECUTIVE_STEPS = 3. При першому кроці нижче порогу, лічильник обнуляється.

Метод _statistical_alerts() обчислює z-score (рис. 4.6.) для трьох параметрів: speed_kmh, temp_coolant_c, voltage_v. Ковзне вікно реалізовано через collections.deque з maxlen = ZSCORE_WINDOW_SIZE = 20. При кожному виклику нове значення додається до черги, старе автоматично витісняється, після чого обчислюється середнє і стандартне відхилення. Якщо z-score перевищує ZSCORE_THRESHOLD = 2.5, генерується алерт «STATISTICAL».



```python
def _mean_std(self, values: deque) -> tuple[float, float]:
    n = len(values)
    if n < 2:
        return 0.0, 0.0
    mean = sum(values) / n
    variance = sum((x - mean) ** 2 for x in values) / n
    return mean, math.sqrt(variance)

def _zscore(self, device_id: str, field_name: str, value: float) -> Optional[float]:
    window = self._windows[device_id][field_name]
    window.append(value)
    mean, std = self._mean_std(window)
    if std < 1e-9:
        return None
    return abs(value - mean) / std
```

Рисунок 4.6. Реалізація розрахунку z-score для динамічного виявлення аномалій

Відмітимо: у полі value алерту зберігається саме поточне значення параметра, а не z-score. Текстове поле message містить: назву поля, обчислений z-score і поріг. Це було зроблено для зручності і кращої читабельності. Google Sheets міститиме конкретне значення, а не абстрактне статистичне відхилення.

Функція validate_payload() у validators.py виконує перевірку кожного вхідного пакета ще до того, як той потрапляє до детектора аномалій. Спочатку, проводиться перевірка чи є всі обов'язкові поля та чи відповідають їх типи очікуваним. Далі, йде перевірка чи знаходяться значення в реалістичних межах.

Приклад фізичного обмеження: температура охолоджувальної рідини не може перевищувати 150 градусів за цельсієм або бути від'ємною.

4.5. Запис даних та інтеграція з зовнішніми сервісами

Телеметрія, що пройшла валідацію та аналіз, потрапляє до декількох різних сховищ, при чому, кожне закриває свою задачу.

У Cosmos DB потрапляє кожен пакет, що пройшов HMAC-перевірку і валідацію. Cosmos DB-клієнт ініціалізується у функції _get_cosmos_container() через CosmosClient з endpoint і ключем з Application Settings. Ingestor записує кожен валідний пакет методом upsert_item: якщо документ із таким id вже існує - він оновлюється; якщо ні – створюється новий. Поле id формується як device_id + '_' + timestamp, що гарантує унікальність у межах одного пристрою і часової відмітки.

На відміну від Cosmos DB, в Azure SQL йдуть не всі сирі пакети підряд, а впорядковані дані: швидкість, температура, координати та інші показники з прив'язкою до конкретного пристрою і часу. Саме ця таблиця є основою для аналітики в Metabase.

Запис у Azure SQL реалізований через SQL MERGE, що показано на рисунку 4.7. Логіка проста: якщо рядок з такою парою (device_id, ts) вже є, то нічого не відбувається. Якщо немає – вставляємо новий.

```
sql = f"""
    MERGE {SQL_TABLE_AGGREGATES} AS target
    USING (SELECT ? AS device_id, ? AS ts) AS source
    ON target.device_id = source.device_id AND target.ts = source.ts
    WHEN NOT MATCHED THEN
        INSERT (device_id, ts, speed_kmh, temp_coolant_c, fuel_pct,
                voltage_v, rpm, load_pct, odometer_km, latitude, longitude)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?);
"""
```

Рисунок 4.7. Формування SQL-запиту для агрегації та збереження поточних параметрів стану транспортного засобу

Відмітимо, що Azure SQL Database дозволяє не лише накопичувати телеметрію, а й здійснювати пряме керування даними безпосередньо у хмарі (рис. 4.8.).Наявність вбудованого редактора запитів та підтримка стандартного SQL-синтаксису дає можливість проводити оперативний аналіз, фільтрацію та агрегацію даних без завантаження їх на локальну машину. Це перетворює базу даних з пасивного сховища на активний інструмент аналітики.

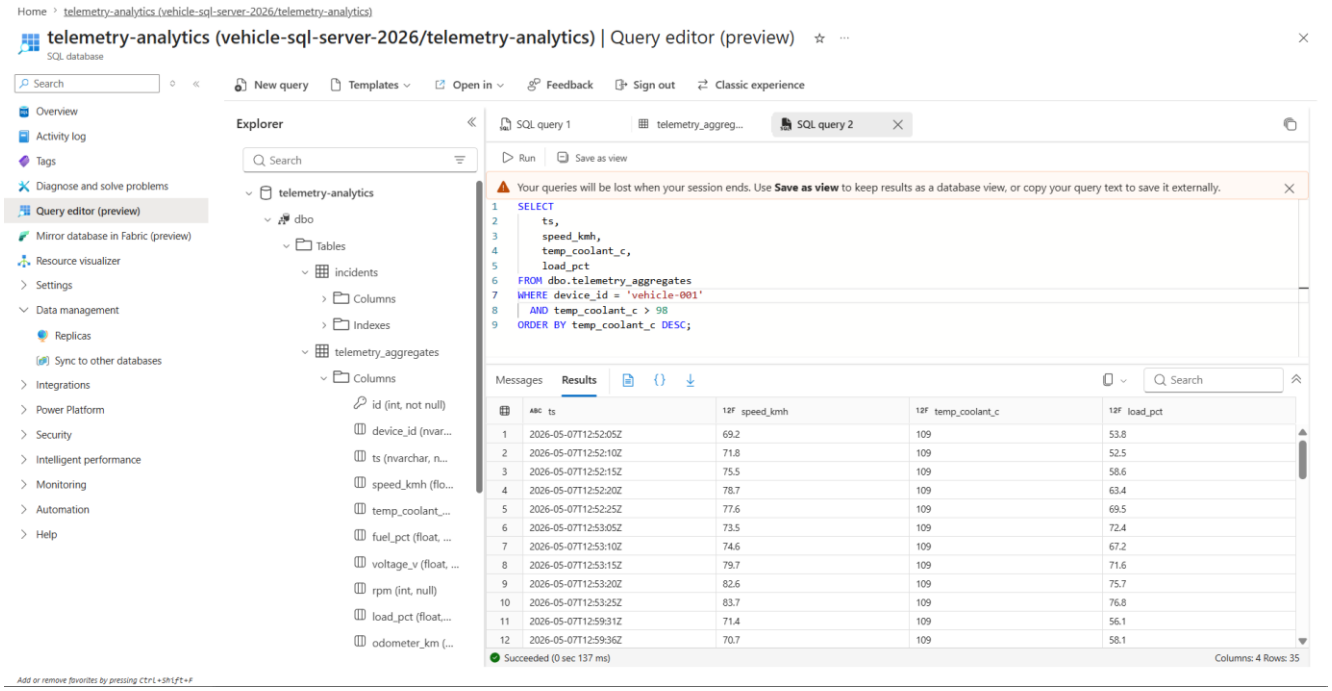


Рисунок 4.8. Запит для вибірки показників телеметрії ТЗ при граничних температурних режимах роботи двигуна

Інтеграція з Google Sheets реалізована у notifier_sheets.py. Автентифікація відбувається через Service Account: JSON-ключ зчитується зі змінної

GOOGLE_SERVICE_ACCOUNT_JSON, десеріалізується і передається в Credentials.from_service_account_info(). Дана інтерграція дозволяє диспетчеру або менеджеру парку не відкривати Metabase щохвилини, а Google Sheets він бачить у браузері постійно. Алерт, що з’явився у спільній таблиці, помітять одразу (рис. 4.9.). Крім того, дані що потрапляють у таблицю, можуть бути миттєво використані користувачем для побудови звітів, фільтрації та сортування інцидентів за допомогою стандартних інструментів електронних таблиць, що не потребують знань мов запитів.

2026-05-14T23:08:48Z	vehicle-001	OVERSPEED	critical	speed_kmh	145	130	Critical overspeed: 145.0 km/h

Рисунок 4.9. Приклад алерту у таблиці

4.6. Дашборд, візуалізація даних. Тестування системи

Metabase підключається до Azure SQL і дозволяє будувати інтерактивні графіки та звіти без написання окремого фронтенд-застосунку. Усі запити йдуть до «холодного» сховища в Azure SQL та не навантажують основний «pipeline» обробки.

Головний екран Metabase зображено на рисунку 4.10.

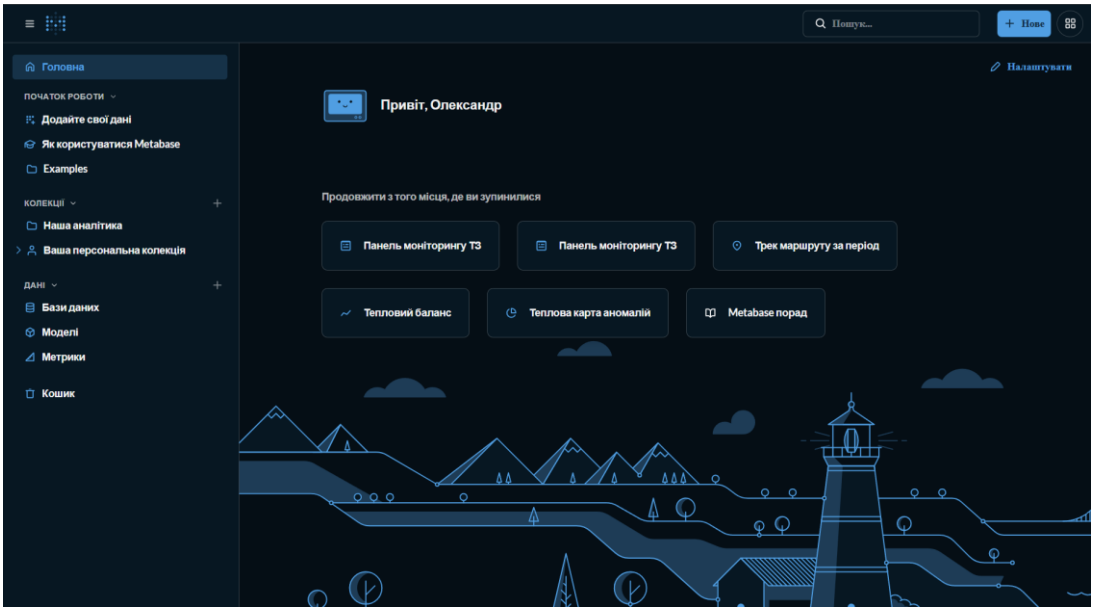


Рисунок 4.10. Головний екран Metabase

Вибір цього інструмента обумовлений кількома причинами: відкритий вихідний код, простота інтеграції з базами даних, підтримка інтерактивних dashboard-панелей та можливість швидкого створення аналітики без потреби у складному налаштуванні BI-систем.

На рисунку 4.11 продемонстровано процес створення SQL-запиту, що реалізує логіку вибору останнього запису для кожного зареєстрованого пристрою.

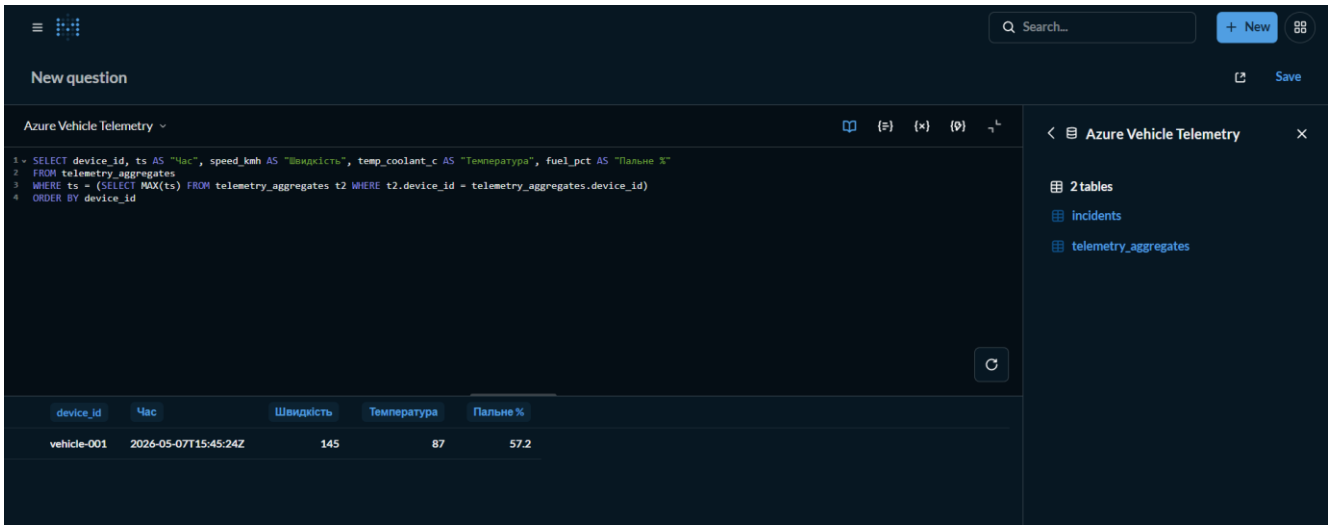


Рисунок 4.11. SQL-запит через Metabase

Рисунок 4.12. демонструє панель, призначена для миттєвої оцінки стану автомобіля в поточний момент.

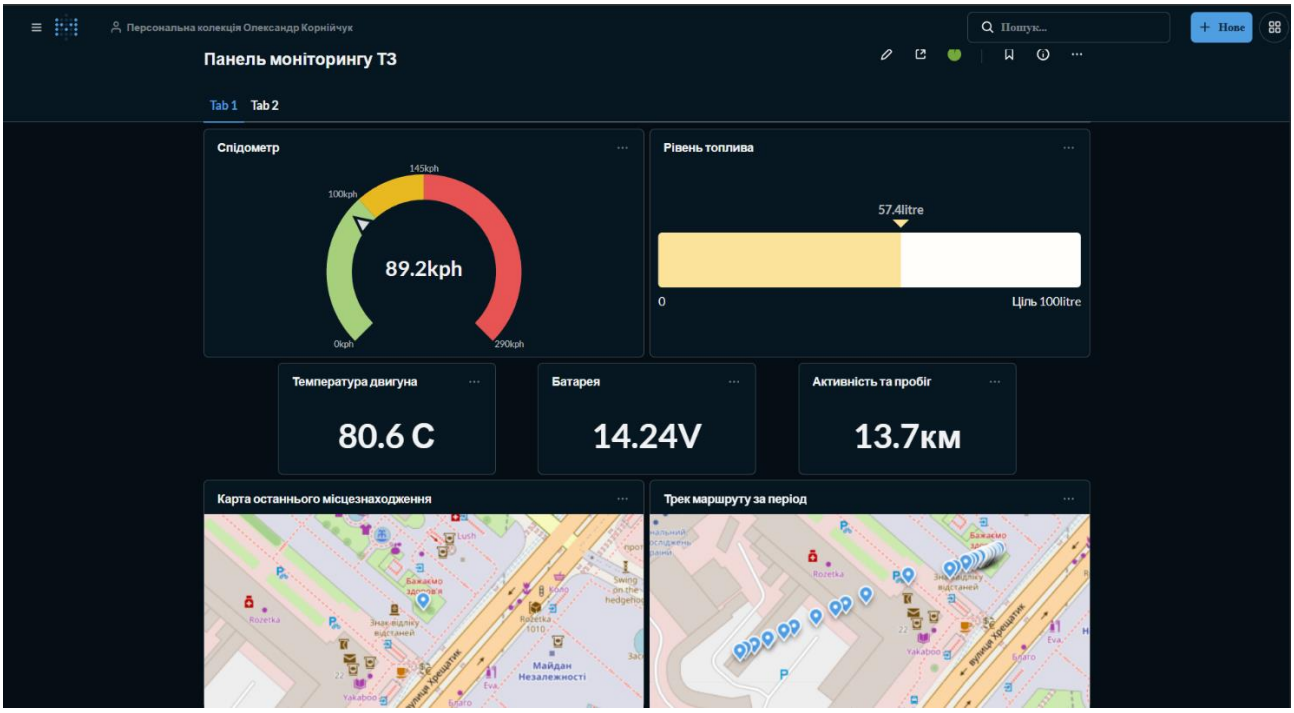


Рисунок 4.12. Панель оперативного моніторингу транспортного засобу

Проте, Metabase пропонує дещо більше, аніж візуалізація актуальної інформації. Особливу роль у системі відіграє блок автоматичної класифікації подій (рис. 4.13.), що дозволяє виокремити критичні аномалії з загального потоку телеметрії. Візуалізація цього процесу дає чітке розуміння структури проблем: на дашборді наочно представлено розподіл алертів за рівнем важливості та типами несправностей.

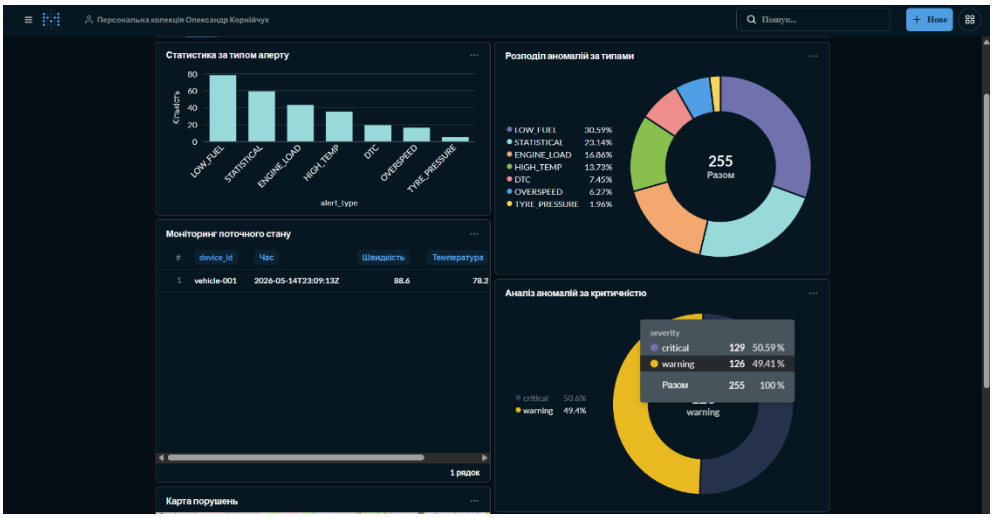


Рисунок 4.13. Статистичний розподіл аномалій за типами

Графіки динаміки дозволяють побачити тренди у стані авто. На рисунку 4.14. видно, як поєднання показників швидкості, напруги та температури допомагає вчасно помітити аномалії ще до того, як вони стануть критичними.

Рисунок 4.14. Аналітична панель технічного стану

Для підтвердження надійності розробленого рішення проведено серію випробувань. Наприклад, на рисунку 4.15 видно, що передача даних від симулятора до IoTHub займає <1с.

Рисунок 4.15. Тест швидкості доставки повідомлення у IoTHub

Другий сценарій: пристрій vehicle-001, інтервал 30 секунд, тривалість 2 години. Мета – перевірити повний ланцюжок від відправки до відображення в Metabase. Перевірялась наявність JSON-документів у Cosmos DB Data Explorer, рядків у telemetry_aggregates в Query Editor та відображення графіків у Metabase.

Ланцюжок спрацював повністю. За 2 години надіслано 240 повідомлень, у Cosmos DB з’явилось 240 документів – втрат не зафіксовано. Затримка від публікації до появи рядка в Cosmos DB становила в середньому 1.9 секунди, максимум – 4.2 секунди при cold start Function після паузи.

Система працює згідно заданих вимог і специфікацій.

ВИСНОВКИ

У результаті виконання практичної частини кваліфікаційної роботи спроєктовано та реалізовано IoT-платформу для моніторингу стану транспортних засобів з обробкою даних у хмарній інфраструктурі Microsoft Azure. Платформа складається з двох основних компонентів: симулятора транспортного засобу та хмарного «pipeline» обробки, що включає три Azure Functions, баз даних (Cosmos DB та Azure SQL), системи сповіщень через Google Sheets та візуалізації на основі Metabase.

Протокол MQTT забезпечив надійну передачу телеметрії. Безпека реалізована поєднанням TLS-шифрування та HMAC-SHA256 підпису для кожного повідомлення. Архітектура Hot/Cold Path дозволила одночасно досягти оперативного реагування на аномалії в реальному часі та довгострокового зберігання даних для аналітики.

Симулятор з корельованими параметрами забезпечив стабільний потік телеметрії для її подальшої обробки та зберігання. Тришарова логіка виявлення аномалій (порогові перевірки, детекція стійкого перевантаження і z-score) продемонструвала здатність правильно виявляти різні типи відхилень.

Використання Python та Azure дозволило створити проєкт, що забезпечує стабільний моніторинг та функціонує виключно на безкоштовних ресурсах хмарної платформи. Логіка обробки повідомлень забезпечує коректну доставку телеметрії до баз даних. Робота над проєктом дала можливість відточити навички розробки мовою Python та навчитися конфігурувати хмарну інфраструктуру для реальних задач. У підсумку маємо надійний інструмент для контролю стану авто, який працює без збоїв і не потребує фінансових витрат на підтримку.

Практична цінність роботи полягає у тому, що система є повністю функціональним прототипом. Платформа легко адаптується для реального розгортання шляхом заміни симулятора на фізичний OBD-II трекер з MQTT-підтримкою, при цьому вся логіка хмарної частини залишиться без змін.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Інтернет речей — Вікіпедія. URL: https://uk.wikipedia.org/wiki/%D0%86%D0%BD%D1%82%D0%B5%D1%80%D0%BD%D0%B5%D1%82_%D1%80%D0%B5%D1%87%D0%B5%D0%B9 (дата звернення: 03.02.2026).
2. IoT connections worldwide 2034| Statista. URL: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/ (дата звернення: 03.02.2026).
3. Як читати помилки OBD-II: гайд для новачків. URL: https://www.ids.kiev.ua/obdii-guide-ua/ (дата звернення: 08.03.2026).
4. OBDII ECU — Open Vehicles documentation. URL: https://docs.openvehicles.com/en/latest/userguide/ecu.html (дата звернення: 10.02.2026).
5. Azure IoT Hub communication protocols and ports | Microsoft Learn. URL: https://learn.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-protocols (дата звернення: 11.02.2026).
6. MQTT Version 3.1.1. URL: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html (дата звернення: 11.03.2026).
7. OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0, Part 2: Transport. URL: https://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-transport-v1.0-os.html (дата звернення: 11.02.2026).
8. RFC 7252 - The Constrained Application Protocol (CoAP). URL: https://datatracker.ietf.org/doc/html/rfc7252 (дата звернення: 22.02.2026).
9. Azure SLA Board. URL: https://azurecharts.com/sla(дата звернення: 22.02.2026).
10. Microsoft Azure IoT Edge. URL: https://azure.github.io/iot-edge-v1/(дата звернення: 03.03.2026).

11. Azure Stream Analytics documentation - Azure Stream Analytics | Microsoft Learn. URL: https://learn.microsoft.com/en-us/azure/stream-analytics/(дата звернення: 05.03.2026).
12. Pricing - Stream Analytics | Microsoft Azure. URL: https://azure.microsoft.com/en-us/pricing/details/stream-analytics/ (дата звернення: 06.03.2026).
13. Azure IoT Hub Documentation | Microsoft Learn. URL: https://learn.microsoft.com/en-us/azure/iot-hub/(дата звернення: 08.03.2026).
14. Azure Functions documentation | Microsoft Learn. URL: https://learn.microsoft.com/en-us/azure/azure-functions/(дата звернення: 08.03.2026).
15. Azure Service Bus Messaging documentation | Microsoft Learn. URL: https://learn.microsoft.com/en-us/azure/service-bus-messaging/(дата звернення: 10.03.2026).
16. Unified AI Database - Azure Cosmos DB | Microsoft Learn. URL: https://learn.microsoft.com/en-us/azure/cosmos-db/overview(дата звернення: 10.03.2026).
17. Metabase documentation | Metabase Documentation. URL: https://www.metabase.com/docs/latest/(дата звернення: 12.03.2026).
18. Fleet Telematics | Complete Fleet Management Solution | Samsara. URL: https://www.samsara.com/products/telematics(дата звернення: 18.03.2026).
19. One Platform - Total Fleet Management | Geotab. URL: https://www.geotab.com/(дата звернення: 18.03.2026).

ДОДАТОК А

МОНІТОРИНГ ТЕЛЕМЕТРІЇ АВТОМОБІЛЯ ЗАСОБАМИ ПЛАТФОРМИ THINGSBOARD ТА ПРОТОКОЛУ MQTT

Корнійчук Олександр Васильович
Рівненський державний гуманітарний університет, бакалавр,
korniichuk.oleksandr.ipz@rshu.edu.ua
Шинкарчук Назар Володимирович
Рівненський державний гуманітарний університет, доцент кафедри інформаційних технологій та моделювання, кандидат технічних наук, доцент, nazar.shynkarchuk@rshu.edu.ua

MONITORING VEHICLE TELEMETRY USING THE THINGSBOARD PLATFORM AND MQTT PROTOCOL

Korniichuk Oleksandr Vasylovych
Rivne State University of the Humanities, Bachelor, korniichuk.oleksandr.ipz@rshu.edu.ua
Shynkarchuk Nazar Volodymyrovych
Rivne State University of the Humanities, Associate Professor of the Department of Information Technologies and Modeling, Candidate of Technical Sciences, Associate Professor,
nazar.shynkarchuk@rshu.edu.ua

This paper describes the design and testing of a real-time vehicle telemetry system built on the ThingsBoard IoT platform. A Python simulator generates vehicle data (GPS position, speed, RPM, engine temperature and fuel level) transmitted via MQTT. The system demonstrates live dashboarding and automated alarm notification using open-source tools.

Keywords: Internet of Things(IoT), MQTT, ThingsBoard, vehicle monitoring, telemetry, real-time visualization, Python.

З розвитком Індустрії 4.0, підключений до мережі транспорт перестав бути теоретичною концепцією, на сьогоднішній день – це складна інженерна задача. Сучасний автомобіль, генерує велику кількість інформації: GPS-координати, стан двигуна та батареї, швидкість, температура, рівень та витрата палива, тощо. Ці дані генеруються щосекунди та вимагають значних ресурсів для їх збору, обробки та зберігання. Організація збору даних, доставка їх до сервера і відображення в зручному та інтуїтивно зрозумілому інтерфейсі ще кілька років тому вимагало суттєвих витрат на підписку до великих хмарних платформ або написання власного проєкту «з нуля». Саме тут на допомогу приходять готові розробки та IoT-платформи з відкритим кодом, вони дозволяють зосередитись на прикладній задачі, а не на розгортанні дороговартісної інфраструктури.

Одним з хороших прикладів таких платформ є Thingsboard Community Edition. Thingsboard CE – це платформа Інтернету речей з відкритим кодом, що розробляється з 2016 року під ліцензією Apache 2.0. Основні задачі: збір, обробка, візуалізація та управління пристроями. ThingsBoard надає вбудований MQTT-брокер, сховище даних (через інтеграцію як з реляційними (PostgreSQL), так і з нереляційними базами даних (Cassandra)) і конструктор дашбордів без написання серверного коду та без оренди хмарного середовища. ThingsBoard підтримує такі протоколи для підключення пристроїв: MQTT, CoAP, HTTP, MQTT Sparkplug, LwM2M, SNMP [1]. У ThingsBoard усе влаштовано наступним чином: платформа здатна приймати дані через будь-який транспортний протокол, адже вона автоматично приводить ці дані до одного вигляду. Це зручно, розробник може у будь-який момент змінити протокол на самому пристрої, при цьому не доведеться переналаштовувати сервер чи дашборд.

Головною особливістю платформи Thingsboard є здатність створювати та налаштовувати власні дашборди для зручної візуалізації даних. Такий функціонал має зручний та інтуїтивно

зрозумілий інтерфейс і не вимагає знання фронтенд-розробки. Бібліотека вбудованих віджетів охоплює: цифрові та аналогові індикатори, картографічні віджети на базі Google Maps, лінійні та стовпчикові графіки часових рядів, таблиці подій і журнали сповіщень, а також Trip Animation (плеєр-відтворювач геотреків із керуванням).

Rule Engine є основою для логіки обробки подій у ThingsBoard. Він дозволяє будувати ланцюжки правил із вузлів фільтрації, збагачення, трансформації та дії. Для кожного пристрою або групи пристроїв можна задати порогові значення та умови. Також, є можливість призначити реакцію на події та ситуації: створення alarm, відправку листа на email. У тестовому проєкті налаштовано правило, що генерує «Fuel Alert» рівня «Critical» при падінні рівня палива нижче критичного значення.

Розроблена система побудована за наступною схемою: джерело даних (симулятор автомобіля) → транспортний протокол MQTT → платформа обробки та відображення. Джерелом даних є симулятор транспортного засобу, реалізований мовою Python. Клас VehicleSimulator моделює поведінку автомобіля, його розміщення, стан, температуру і швидкість обертів двигуна, швидкість його руху.

Основним мережевим протоколом було обрано MQTT на основі бібліотеки paho-mqtt. Дані публікуються кожну секунду у форматі JSON. ThingsBoard розгорнуто локально. Пристрій автентифікується через унікальний Access Token. Уся конфігурація (правила обробки, дашборди, тригери) налаштовується засобами вебінтерфейсу платформи.

Після запуску програми-симулятора, дані генеруються та надсилаються до IoT-платфроми ThingsBoard, де і відображаються у режимі реального часу на дашборді (рис. 1.).

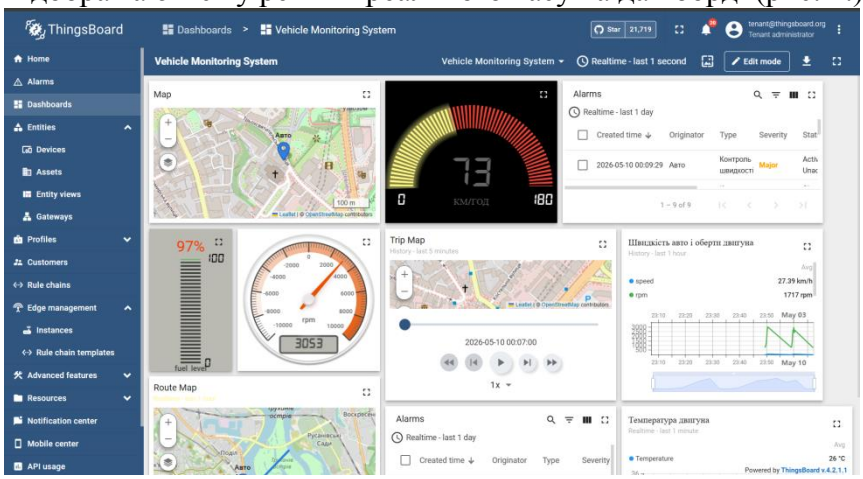


Рисунок 1. Візуалізація отриманих даних в ThingsBoard

Також, відмітимо систему алярмів (alarm) і сигнальних повідомлень. Наприклад, система надсилає алерт «Перевищення швидкості», якщо водій перевищує ліміт у 60 км/год. При цьому, Rule Engine виявляв перевищення допустимої швидкості та сформував відповідне сповіщення. Це підтверджує коректність налаштованих умов та придатність платформи до моніторингу параметрів транспортного засобу у режимі реального часу.

Слід відмітити, що програмний продукт Community Edition, має надто обмежені можливості по масштабуванню та аналітиці, тому при необхідності обробляти дані з великої кількості пристроїв, рекомендуємо використовувати хмарні платформи рівня «ентерпрайз» (наприклад, Microsoft Azure чи Amazon Web Services).

Підсумовуючи, IoT-платформи рівня ThingsBoard придатні для створення системи транспортного моніторингу у проєктах, де головним пріоритетом є швидкість та відносна простота розробки і мінімізація витрат.

Список використаних джерел

1. Device Connectivity Protocols | ThingsBoard Community Edition. URL: https://thingsboard.io/docs/reference/protocols/ (дата звернення: 26.04.2026).