МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра комп'ютерних наук

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА ВИСОКОПРОДУКТИВНОЇ СИСТЕМИ ЗБОРУ ТА АНАЛІЗУ ГЕОЛОКАЦІЙНИХ ДАНИХ»**

Виконав: студент 4 курсу, групи 6.1222
спеціальності 122 Комп'ютерні науки
(шифр і назва спеціальності)
освітньої програми Комп'ютерні науки
(назва освітньої програми)
Д. О. Кравцов
(ініціали та прізвище)
Керівник доцент кафедри комп'ютерних наук, к. т. н. Добровольський Г. А.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент доцент кафедри програмної інженерії, к.ф.-м.н. Мильцев О. М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра комп'ютерних наук

Рівень вищої освіти бакалавр

Спеціальність 122 Комп'ютерні науки
(шифр і назва)

Освітня програма Комп'ютерні науки

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри комп'ютерних наук, к.пед.н., доцент

_____ Пшенична О.С.
(підпис)

" ____ " _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Кравцову Данилу Олександровичу
(прізвище, ім'я та по–батькові)

1. Тема роботи Розробка високопродуктивної системи збору та аналізу геолокаційних даних

керівник роботи Добровольський Геннадій Анатолійович, к. т. н.
(прізвище, ім'я та по–батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 02 » грудня 2025 року № 1741-с

2. Строк подання студентом роботи 31.05.2026

3. Вихідні дані до роботи 1. Постановка задачі.
2. Технічна документація СУБД ClickHouse та фреймворку FastAPI
3. Протоколи та метрики експериментального навантажувального тестування

4. Зміст розрахунково–пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз предметної області та обґрунтування вибору технологій.
2. Проєктування та архітектура інформаційної системи.
3. Експериментальне дослідження та тестування системи.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) презентація, рисунки архітектури системи та результати тестування

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 05.12.2025

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	06.12.2025	
2.	Збір вихідних даних.	07.12.2025 – 20.12.2025	
3.	Обробка методичних та теоретичних	21.12.2025 – 18.02.2026	
	джерел.		
4.	Розробка першого та другого розділу.	18.02.2026 – 20.03.2026	
5.	Розробка третього розділу та тестування системи	21.03.2026 – 08.05.2026	
6.	Оформлення та нормоконтроль	08.05.2026 – 17.05.2026	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	19.06.2026	

Студент _____ Д. О. Кравцов
(підпис) (ініціали та прізвище)

Керівник роботи _____ Г. А. Добровольський
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ О. Г. Спиця
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка високопродуктивної системи збору та аналізу геолокаційних даних»: 52 с., 13 рис., 4 табл., 23 джерела, 3 додатки.

БУФЕРУВАННЯ ДАНИХ, ВЕБСЕРВІС, ВИСОКОНАВАНТАЖЕНА СИСТЕМА, ГЕОЛОКАЦІЙНІ ДАНІ, ІНТЕРНЕТ РЕЧЕЙ, ТЕЛЕМЕТРІЯ, CLICKHOUSE, FASTAPI, GPS-МОНІТОРИНГ, REST API.

Об’єкт дослідження – процес збору, зберігання та обробки телеметричної інформації у високонавантажених логістичних системах.

Предмет дослідження – алгоритми, мережеві протоколи та програмні засоби, що забезпечують високопродуктивну обробку та візуалізацію геолокаційних даних.

Мета роботи – проєктування та програмна реалізація високопродуктивної системи для надійного збору та оперативного аналізу геолокаційних даних у режимі реального часу.

Методи дослідження – системний аналіз, проєктування реляційних та колонкових баз даних, об’єктно-орієнтоване програмування, навантажувальне і стрес-тестування програмного забезпечення.

У результаті виконання кваліфікаційної роботи розроблено високопродуктивну систему збору та аналізу геолокаційних даних на базі фреймворку FastAPI та колонкової СУБД ClickHouse. Система забезпечує стабільну обробку телеметрії в режимі реального часу, підтримує локальне буферування координат у разі втрати мережевого з’єднання та дозволяє виконувати швидкі аналітичні запити до великих масивів історичних даних із коефіцієнтом стиснення 7–10 разів. Розроблена система може бути використана як основа для комерційних рішень у сфері логістики, GPS-моніторингу та IoT-платформ.

SUMMARY

Bachelor's Qualifying Thesis «Development of a High-Performance System for Collecting and Analyzing Geolocation Data»: 52 pages, 13 figures, 4 tables, 23 references, 3 supplements.

BUFFERING, CLICKHOUSE, FASTAPI, GEOLOCATION DATA, GPS MONITORING, HIGH-LOAD SYSTEM, INTERNET OF THINGS, REST API, TELEMETRY, WEB SERVICE.

Object of the study – the process of collecting, storing, and processing telemetry information in high-load logistics systems.

Subject of the study – algorithms, network protocols, and software tools that provide high-performance processing and visualization of geolocation data.

Aim of the study – design and software implementation of a high-performance system for reliable collection and real-time analysis of geolocation data.

Methods of research – system analysis, design of relational and column-oriented databases, object-oriented programming, load testing, and stress testing of software systems.

As a result of this bachelor's thesis, a high-performance system for collecting and analyzing geolocation data based on the FastAPI framework and the ClickHouse column-oriented DBMS was developed. The system provides stable real-time telemetry processing, supports local coordinate buffering in case of network connection loss, and enables fast analytical queries for large volumes of historical data with a compression ratio ranging from 7x to 10x. The proposed system can serve as a basis for commercial solutions in the fields of logistics, GPS monitoring, and IoT platforms.

ЗМІСТ

Завдання на кваліфікаційну роботу……..2
Реферат 4
Summary 5
Вступ..... 8
1 Аналіз предметної області та обґрунтування вибору технологій 11
1.1 Огляд предметної області та актуальність розробки 11
1.2 Порівняльний аналіз СУБД для зберігання часових рядів та геоданих 13
1.3 Аналіз мережевих протоколів передачі телеметричних даних..... 15
1.4 Концепція стійкості клієнтської частини до розривів зв’язку 17
1.5 Постановка задачі та технічні вимоги до системи 17
2 Проєктування та архітектура інформаційної системи 19
2.1 Загальна архітектура системи та опис компонентів..... 19
2.2 Проєктування бази даних та вибір структур зберігання..... 19
2.3 Механізми компресії та низькорівневого кодування даних у ClickHouse..... 23
2.4 Розробка моделі валідації вхідних даних та взаємодії з СУБД..... 24
2.5 Алгоритмічне забезпечення стійкості клієнтської частини 26
2.6 Проєктування програмного інтерфейсу (API) та структура ендпоінтів..... 29
2.7 Концепція інформаційної безпеки та захист персональних даних.... 30
2.8 Обробка виняткових ситуацій та відмовостійкість серверної частини..... 32
3 Експериментальне дослідження та тестування системи..... 33
3.1 Конфігурація тестового стенда та параметри середовища..... 33
3.2 Тестування функціональних можливостей аналітичного модуля 34
3.3 Аналіз продуктивності системи при високому навантаженні..... 35

3.4 Перевірка стійкості клієнтської частини до розривів зв’язку 38
3.5 Оптимізація клієнтської візуалізації великих масивів геоданих у браузері 39
3.6 Візуалізація геолокаційних даних у реальному часі 41
Висновки 42
Перелік посилань 43
Додаток А Лістинг серверної частини 45
Додаток Б Лістинг клієнтської частини та симулятора навантаження 48
Додаток В Лістинг веб-інтерфейсу візуалізації 51

ВСТУП

Бурхливий розвиток систем моніторингу та телеметрії призвів до появи значних масивів просторово-часової інформації. Основну частину таких даних становлять геолокаційні координати. Їх обробка в режимі реального часу є складним технічним завданням. Система має не лише безперебійно приймати тисячі точок щосекунди, але й паралельно виконувати ресурсомісткі аналітичні запити по вже накопичених історичних даних.

Інтенсивний запис просторових координат швидко виснажує ресурси класичних реляційних баз. Саме тому для подібних завдань доцільно розгортати колонкові СУБД, зокрема ClickHouse. Ще одна вразливість систем моніторингу – зони нестабільного або відсутнього мобільного покриття. Відповідно, зараз виникає потреба в інженерних рішеннях, які б об’єднали високопродуктивну серверну частину на базі FastAPI із клієнтськими застосунками, здатними зберігати дані локально до моменту відновлення зв’язку.

Мета роботи полягає у проєктуванні та програмній реалізації високопродуктивної системи для надійного збору та оперативного аналізу геолокаційних даних.

Для досягнення поставленої мети необхідно вирішити такі завдання:

– проаналізувати сучасні методи та технології обробки великих масивів геолокаційних даних;
– розробити архітектуру системи на базі асинхронного фреймворку FastAPI;
– спроєктувати структуру бази даних у СУБД ClickHouse для оптимізації аналітичних запитів;
– реалізувати алгоритми буферування на стороні клієнта для забезпечення стійкості до розривів зв’язку;
– провести тестування системи під високим навантаженням та оцінити її ефективність.

Об’єкт дослідження: процес збору, зберігання та обробки телеметричної інформації у високонавантажених логістичних системах.

Предмет дослідження: алгоритми, мережеві протоколи та програмні засоби, що забезпечують високопродуктивну обробку та візуалізацію геолокаційних даних.

Методи дослідження: системний аналіз предметної області, принципи проєктування реляційних та колонкових баз даних, об’єктно-орієнтоване програмування, а також методи навантажувального і стрес-тестування програмного забезпечення.

Особливістю запропонованого рішення є поєднання механізмів локального файлового буферування на стороні клієнта з високопродуктивною аналітичною СУБД ClickHouse для забезпечення цілісності геолокаційних даних в умовах нестабільного мережевого з’єднання. Практична цінність роботи полягає у створенні робочого прототипу. Розроблене програмне рішення може бути використане як основа для комерційних систем трекінгу вантажівок, IoT-платформ або моніторингу транспорту. Запропонований підхід суттєво знижує ризик втрати координат на маршруті та значно скорочує час генерації аналітичних звітів на основі історичних даних.

Робота структурована за логікою системного підходу і складається зі вступу, трьох взаємопов’язаних розділів, загальних висновків, списку використаних джерел та трьох додатків.

У вступі обґрунтовано актуальність теми, окреслено об’єкт і предмет дослідження, а також сформульовано мету та ключові завдання розробки.

Перший розділ присвячено комплексному аналізу предметної області, дослідженню ринку сучасних логістичних систем моніторингу та порівняльному аналізу СУБД і протоколів передачі геоданих.

У другому розділі розкрито безпосередні інженерні аспекти проєктування бази даних ClickHouse, наведено архітектуру серверних ендпоінтів на FastAPI, описано концепцію безпеки та алгоритмічні рішення для файлового буферування на стороні клієнта.

Третій розділ містить опис конфігурації тестового стенда та результати експериментальної перевірки системи, включаючи стрес-тестування меж її продуктивності.

У додатках наведено повні лістинги програмного коду серверного модуля, клієнтського скрипта та веб-інтерфейсу інтерактивної карти.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОБҐРУНТУВАННЯ ВИБОРУ ТЕХНОЛОГІЙ

1.1 Огляд предметної області та актуальність розробки

Сучасний ринок транспортної логістики значною мірою спирається на системи супутникового моніторингу. Запровадження технологій Інтернету речей (IoT) дозволило компаніям контролювати переміщення автопарків у режимі реального часу. За оцінками галузевих аналітичних звітів, рівень оснащення комерційних вантажних транспортних засобів телематичними трекерами в Європі оцінюється на рівні 45–50 % [1]. В Україні, враховуючи необхідність жорсткої оптимізації витрат на пальне та забезпечення безпеки вантажів, впровадження систем GPS-моніторингу стає базовою вимогою для управління автопарком.

На сьогодні ринок пропонує широкий спектр готових комерційних платформ, які здатні закрити більшість типових потреб перевізників. Для розуміння контексту розробки доцільно проаналізувати три найбільш відомі світові рішення: Wialon, Navixy та Fleet Complete.

Платформа Wialon є одним із найбільш поширених рішень на ринку [7]. Система підтримує інтеграцію з понад 3000 типами апаратного забезпечення і пропонує широкий набір аналітичних інструментів. Проте її недоліком для певних компаній є модель монетизації: ліцензійний платіж стягується за кожен підключений об’єкт, що для великого автопарку призводить до значних операційних витрат. Крім того, інтерфейс Wialon може бути перевантаженим для вузькоспеціалізованих логістичних сценаріїв, де диспетчеру потрібен лише базовий моніторинг.

Система Navixy відрізняється сучасним веб-інтерфейсом та орієнтованістю на зручність користувача. Її архітектура переважно спирається на хмарну модель SaaS-розгортання [10]. Якщо транспортна компанія вимагає

зберігання геолокаційних треків виключно на власних серверах з міркувань інформаційної безпеки, розгортання локальної версії Navixy може вимагати додаткових витрат та складнішого адміністрування.

Платформа Fleet Complete фокусується на глибокому зборі телеметрії безпосередньо з CAN-шин автомобілів [11]. Одним із обмежень платформи є прив'язка до пропрієтарного апаратного забезпечення. Клієнт стає технічно залежним від конкретного виробника обладнання, що ускладнює заміну пошкоджених трекерів на універсальні аналоги від інших брендів.

Аналіз існуючих рішень демонструє потребу у більш гнучких та спеціалізованих системах, не обтяжених зайвим функціоналом. Порівняльна характеристика розглянутих платформ та запропонованої системи наведена у таблиці 1.1.

Таблиця 1.1 – Порівняння існуючих платформ моніторингу та запропонованої системи

Характеристика	**Wialon**	**Navixy**	**Fleet Complete**	**Запропонована розробка**
Модель монетизації	Плата за об'єкт (SaaS)	Плата за об'єкт (Ліцензія)	Пропрієтарні підписки	Відсутні ліцензійні платежі
Апаратна прив'язка	Універсальна	Універсальна	Висока (пропрієтарне ПЗ)	Універсальна (JSON/HTTP)
Тип розгортання	SaaS / On-Premise	Переважно SaaS	SaaS	On-Premise (Docker)
Складність інтерфейсу	Висока	Середня	Середня	Низька (лише базовий трекінг)

З огляду на вищезазначене, розробка власного програмного рішення є доцільною. Створення вузькоспеціалізованої системи на базі асинхронного вебфреймворку FastAPI та колонкової СУБД ClickHouse дозволяє вирішити

одразу кілька інженерних завдань. По-перше, власна архітектура дозволяє уникнути ліцензійних обмежень на кількість підключених об’єктів. По-друге, компанія отримує суверенний контроль над даними. По-третє, фокус на мінімалістичному функціоналі дозволяє підтримувати стабільну роботу інтерфейсу при значній кількості активних об’єктів.

1.2 Порівняльний аналіз СУБД для зберігання часових рядів та геоданих

Специфіка систем супутникового моніторингу полягає у безперервній генерації величезних масивів часових рядів (time-series data). Кожен підключений пристрій щосекунди або щохвилини надсилає пакет просторових координат. Використання класичних реляційних баз даних, таких як MySQL або базова версія PostgreSQL, у цьому випадку є менш ефективним. При накопиченні десятків мільйонів записів швидкість вставки нових даних різко падає через необхідність постійної перебудови B-Tree індексів, а виконання аналітичних запитів може тривати хвилинами.

Для вирішення подібних завдань архітектура повинна спиратися на спеціалізовані колонкові аналітичні СУБД (OLAP) або бази даних для часових рядів (TSDB) [3]. У межах даного дослідження було проведено порівняльний аналіз трьох провідних рішень: TimescaleDB, QuestDB та ClickHouse.

TimescaleDB є потужним розширенням для PostgreSQL, оптимізованим для роботи з часовими рядами. Її головна перевага – повна підтримка стандарту SQL та екосистеми PostgreSQL. Завдяки механізму гіпертаблиць (hypertable) система автоматично партиціює дані. Проте в основі TimescaleDB лежить рядкова архітектура. Згідно з результатами незалежних тестувань (зокрема Time Series Benchmark Suite), швидкість вставки даних у цю СУБД відчутно поступається суто колонковим рішенням, а рівень стиснення даних вимагає значних апаратних ресурсів під час фонових операцій [23].

QuestDB – це відносно нова, але надзвичайно продуктивна реляційна база даних для часових рядів, написана на Java та C++. Вона демонструє визначні результати швидкості запису, використовуючи оптимізацію на рівні SIMD-інструкцій процесора. За результатами відкритих benchmark-тестів QuestDB здатна обробляти мільйони записів за секунду [20]. Однак її екосистема ще перебуває на стадії активного формування. Деякі складні аналітичні агрегації та геопросторові функції, які критично важливі для рендерингу маршрутів на карті, реалізовані не в повному обсязі порівняно зі зрілішими OLAP-системами.

ClickHouse – це колонкова СУБД з відкритим вихідним кодом, розроблена спеціально для виконання аналітичних запитів у режимі реального часу. Фізичне зберігання даних по стовпцях дозволяє системі зчитувати з диска лише ті атрибути, які фігурують у запиті (наприклад, лише координати, ігноруючи дані про швидкість чи рівень палива). Завдяки сімейству рушіїв таблиць MergeTree ClickHouse здійснює фонове злиття та стиснення шматків даних, досягаючи ступеня компресії до 10–15 разів порівняно з сирими даними [13].

Для об’єктивного вибору СУБД було проаналізовано метрики продуктивності баз даних на основі відкритого набору тестів ClickBench при виконанні стандартних операцій з масивами телеметрії. Результати порівняння наведено у таблиці 1.2.

Таблиця 1.2 – Бенчмарки продуктивності СУБД при обробці 1 мільйона записів телеметрії [13]

Характеристика	**TimescaleDB**	**QuestDB**	**ClickHouse**
Архітектура зберігання	Рядкова	Колонкова	Колонкова
Час вставки 1 млн рядків	~ 8500 мс	~ 1200 мс	~ 1800 мс
Коефіцієнт стиснення на диску	3x – 4x	2x – 3x	7x – 10x
Час аналітичної агрегації (SUM/AVG)	~ 450 мс	~ 80 мс	~ 40 мс
Підтримка складних гео-функцій	Висока	Базова	Висока

Спираючись на наведені технічні характеристики, для розробки серверної частини програмного комплексу було обрано ClickHouse. Хоча QuestDB демонструє трохи вищу пікову швидкість чистої вставки (ingest), ClickHouse пропонує кращий баланс між продуктивністю запису, надзвичайно ефективним стисненням історичних даних та швидкістю виконання важких аналітичних запитів. Крім того, наявність зрілих і стабільних асинхронних драйверів для мови Python (зокрема clickhouse-connect) робить цю СУБД доцільним вибором для інтеграції з фреймворком FastAPI.

1.3 Аналіз мережевих протоколів передачі телеметричних даних

Вибір мережевого протоколу прикладного рівня є одним із найважливіших архітектурних рішень під час проєктування систем Інтернету речей (IoT). На сьогодні в індустрії транспортного моніторингу домінують два основні підходи до організації зв’язку між бортовими пристроями та сервером: використання спеціалізованого протоколу MQTT та застосування класичної моделі HTTP/REST.

Протокол MQTT (Message Queuing Telemetry Transport) вважається галузевим стандартом для комунікації телематичного обладнання [8]. Він працює за моделлю «видавець-підписник» і розроблений спеціально для умов нестабільних мереж із низькою пропускною здатністю. Його основною перевагою є мінімальний розмір заголовків пакетів (від 2 байт) та підтримці постійного TCP-з’єднання. Крім того, MQTT має вбудований механізм контролю якості доставки (QoS 0, 1, 2), який дозволяє суттєво підвищити надійність доставки повідомлень. Проте впровадження цього протоколу вимагає розгортання та підтримки додаткового інфраструктурного вузла – MQTT-брокера.

Протокол HTTP (HyperText Transfer Protocol) у парадигмі REST функціонує за принципом «запит-відповідь». Він є протоколом без збереження

стану, у якому кожний запит обробляється незалежно від попередніх, а для зменшення накладних витрат може використовуватися механізм Keep-Alive. Традиційно HTTP вважається більш ресурсоємним для мікроконтролерів через великий обсяг текстових заголовків [2].

Незважаючи на статус MQTT як стандарту де-факто, для програмної реалізації даного дипломного проєкту було обрано протокол HTTP/REST. Таке рішення базується на кількох вагомих інженерних аргументах, які враховують специфіку розробленої архітектури:

– характер передачі даних. Розроблений клієнтський модуль реалізує механізм локального файлового буферування. У разі втрати зв’язку координати накопичуються у файлі buffer.json. Після відновлення мережі пристрій не відправляє точки поштучно, а формує єдиний масив з історичними даними. HTTP POST-запити добре підходять для передачі великих структурованих JSON-документів (Bulk Ingest);
– односпрямований потік телеметрії. Система орієнтована виключно на збір координат. Зворотна комунікація у межах цієї роботи не передбачена. Відповідно, необхідність у постійно відкритому двосторонньому каналі зв’язку, який забезпечує MQTT-брокер, відсутня;
– мінімізація інфраструктурної складності. Використання HTTP дозволяє напряму взаємодіяти із серверним застосунком, реалізованим на базі фреймворку FastAPI, без необхідності розгортати додатковий проміжний шар у вигляді брокера повідомлень. Це суттєво спрощує топологію системи та полегшує її контейнеризацію за допомогою Docker.

Отже, застосування протоколу HTTP у комбінації з асинхронною обробкою запитів на стороні сервера є обґрунтованим технічним компромісом. Такий підхід дозволяє забезпечити надійну передачу телеметричних даних без суттєвого ускладнення серверної інфраструктури.

1.4 Концепція стійкості клієнтської частини до розривів зв’язку

Мобільні трекери на вантажівках регулярно потрапляють у зони без покриття. Щоб не втрачати координати маршруту, клієнтську частину побудовано за концепцією «стійкого клієнта». Було вирішено відмовитись від ресурсомістких локальних баз даних на користь алгоритму проміжного локального буферування у файли структурованого формату на постійному носії пристрою.

Логіка відмовостійкості працює за таким алгоритмом:

– якщо HTTP-з’єднання переривається, нові просторові точки записуються до локального файлового буфера;
– клієнт постійно моніторить статус мережі. Підставою для переходу є аналіз кодів відповідей сервера при спробах відправити черговий POST-запит;
– коли зв’язок з’являється, застосунок починає послідовно вивантажувати накопичений буфер на сервер із фіксованими інтервалами.

Такий підхід дозволяє реалізувати концепцію енергонезалежного сховища без розгортання ресурсомістких локальних СУБД, що важливо для пристроїв із обмеженими ресурсами. Застосування файлового буфера суттєво знижує ризик втрати накопиченої телеметрії навіть у разі повного розряду акумулятора або раптового перезавантаження бортового комп’ютера вантажівки.

1.5 Постановка задачі та технічні вимоги до системи

Отже, робота спрямована на створення комплексного рішення для збору геоданих. Проєктування системи спирається на наступні базові обмеження та вимоги:

- функціонал: архітектура повинна підтримувати реєстрацію нових трекерів і валідацію вхідних JSON пакетів із координатами. Дані накопичуються у ClickHouse, після чого через окреме API генеруються звіти по швидкості та кілометражу;
- швидкодія: навантажувальний мінімум встановлено на рівні 200 RPS. Щоб черга запитів не блокувалася, середній час відповіді не повинен перевищувати 100 мс;
- відмовостійкість: система повинна забезпечувати роботу при півгодинній відсутності зв’язку. Протягом цього часу мобільний клієнт повинен акумулювати всю зібрану геолокацію в локальному енергонезалежному файловому буфері, мінімізуючи втрати точок маршруту та забезпечуючи збереження даних при можливому перезапуску пристрою.

2 ПРОЄКТУВАННЯ ТА АРХІТЕКТУРА ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1 Загальна архітектура системи та опис компонентів

Високі вимоги до швидкодії зумовили перехід до розподіленої архітектури. Щоб уникнути вузьких місць під час збору геолокації, логіку прийому пакетів, їх буферування та аналізу рознесено по ізольованих вузлах. Компоненти системи мають низький рівень взаємозалежності та можуть масштабуватися незалежно один від одного.

Фізично програмний комплекс поділено на три зони відповідальності:

– клієнтський рівень – мобільний додаток або скрипт, завданням якого є безперервна генерація точок маршруту. Цей компонент має вбудований механізм відмовостійкості для локального буферування координат під час перебування в офлайні;
– транспортний шлюз – серверний застосунок, реалізований на базі фреймворку FastAPI. Він виконує роль первинного фільтра, який проводить перевірку вхідного JSON-повідомлення та пересилає лише коректні дані далі по ланцюжку;
– рівень зберігання – колонкова СУБД ClickHouse. Особливості архітектури ClickHouse дозволяють одночасно виконувати високошвидкісний запис нових даних та аналітичну обробку накопиченої телеметрії.

2.2 Проєктування бази даних та вибір структур зберігання

Швидкість генерації аналітичних звітів у високонавантажених системах моніторингу критично залежить від фізичного формату зберігання інформації на

диску. Для оптимізації процесів запису та читання архітектуру бази даних було побудовано навколо колонкової СУБД ClickHouse.

Для формалізації структури даних було розроблено ER-діаграму, яка відображає концептуальну схему зберігання телеметрії. Оскільки система орієнтована на максимальну швидкість вставок, архітектура бази є денормалізованою [5] – усі необхідні атрибути зберігаються в межах єдиної "широкої" таблиці locations. Модель даних наведено на рисунку 2.1.

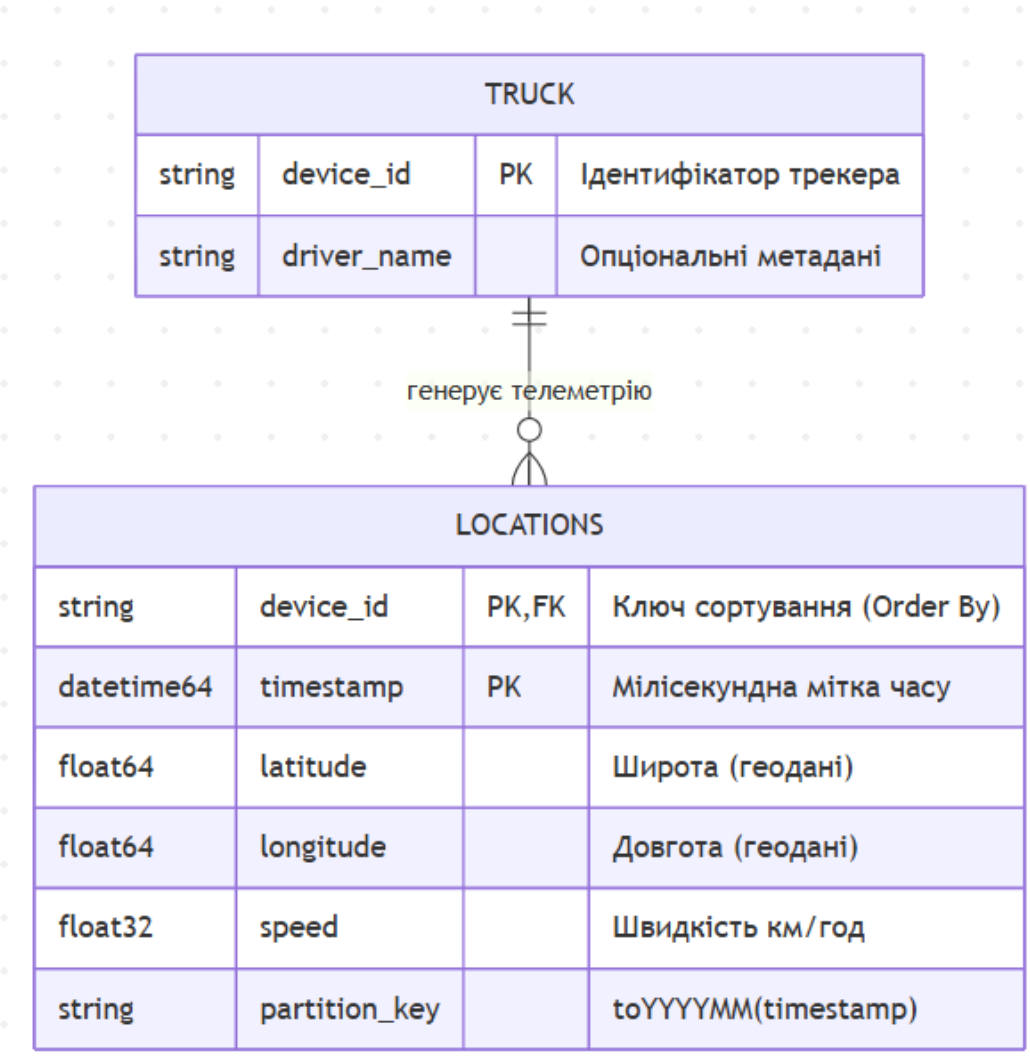


Рисунок 2.1 – ER-діаграма структури бази даних телеметрії

Фізична схема головної таблиці спроєктована з урахуванням мінімізації витрат оперативної пам’яті. Перелік полів та їх типів наведено у таблиці 2.1.

Таблиця 2.1 – Фізична схема та типи даних таблиці locations

Назва поля	Тип даних	Опис
device_id	String	Унікальний текстовий ідентифікатор бортового термінала
timestamp	DateTime64(3, ‘UTC’)	Часова мітка фіксації координат із мілісекундною точністю
latitude	Float64	Географічна широта
longitude	Float64	Географічна довгота
speed	Float32	Поточна швидкість транспортного засобу

Фундаментом для таблиці виступає базовий рушій MergeTree. Під час проєктування розглядалися альтернативні варіанти з цього сімейства, такі як ReplacingMergeTree та SummingMergeTree. Проте ReplacingMergeTree використовується для дедуплікації записів під час операцій злиття частин таблиці та зберігає лише актуальну версію рядка, що категорично не підходить для поставленого завдання, де необхідно зберігати повну історичну лінію треку вантажівки. Рушій SummingMergeTree також був відхилений, оскільки предметна область не потребує автоматичного сумування числових показників під час злиття даних. Тому класичний MergeTree є оптимальним рішенням для незмінного журналу часових рядів.

У системах моніторингу обсяг бази даних неминуче зростає до десятків гігабайт на місяць. Для запобігання деградації продуктивності було застосовано механізм партиціювання за часом: PARTITION BY toYYYYMM(timestamp). Цей підхід розбиває єдину фізичну таблицю на ізольовані шматки за календарними місяцями. Це дає дві критичні переваги:

– запити, що стосуються побудови треку за конкретний день, зчитують лише одну місячну партицію, ігноруючи решту даних на диску;

– реалізація політики утримання даних стає максимально дешевою – система може миттєво видалити застарілу партицію цілком, не навантажуючи процесор операціями DELETE.

На відміну від традиційних реляційних СУБД, ClickHouse не використовує класичні B-Tree індекси для організації первинного доступу до даних. Замість цього тут працює механізм розріджених індексів. Для оптимізації доступу до даних використано ключ сортування ORDER BY (device_id, timestamp), який одночасно формує первинний індекс таблиці. Фізично на диску всі записи спочатку групуються за ідентифікатором вантажівки, а потім – за часом. Індексний файл зберігає посилання не на кожен рядок, а лише на кожну 8192-гу. Таке архітектурне рішення дозволяє базі миттєво знаходити потрібний діапазон даних на жорсткому диску під час фільтрації треків конкретного автомобіля *WHERE device_id = 'truck-01'*, споживаючи при цьому мінімум оперативної пам'яті.

Програмну реалізацію створення цієї таблиці з усіма описаними структурними обмеженнями наведено на рисунку 2.2.

```
    startup_db.command("""
    CREATE TABLE IF NOT EXISTS locations (
    device_id String,
    timestamp DateTime64(3, 'UTC') CODEC(DoubleDelta, LZ4),
    latitude Float64 CODEC(Gorilla, LZ4),
    longitude Float64 CODEC(Gorilla, LZ4),
    speed Float32 CODEC(Gorilla, LZ4)
)
ENGINE = MergeTree()
PARTITION BY toYYYYMM(timestamp)
ORDER BY (device_id, timestamp)
    """)
```

Рисунок 2.2 – Програмна реалізація структури таблиці locations

2.3 Механізми компресії та низькорівневого кодування даних у ClickHouse

Оскільки системи моніторингу транспорту оперують колосальними обсягами інформації, критично важливим аспектом проєктування є мінімізація фізичного обсягу, який база даних займає на постійному носії. Традиційні реляційні СУБД зберігають дані рядками, через що суміжні типи даних у межах однієї сторінки пам’яті постійно змінюються (наприклад, за рядком типу String йде DateTime, а потім Float64). Це унеможливує ефективне застосування алгоритмів стиснення.

Колонкова архітектура ClickHouse кардинально вирішує цю проблему [14]. Завдяки тому, що кожна колонка таблиці locations зберігається в окремому фізичному файлі на SSD-накопичувачі Kingston NV2, архітектура отримує можливість аналізувати послідовності однорідних типів даних. Для підвищення коефіцієнта стиснення в системі реалізовано дворівневу схему оптимізації: спеціалізовані кодеки кодування на рівні типів даних та універсальні алгоритми компресії загального призначення.

Під час проєктування схеми даних для кожної колонки було підібрано індивідуальні компресійні кодеки:

– колонка timestamp (DateTime64): Значення часових міток у системах телеметрії завжди зростають монотонно з фіксованим або прогнозованим інтервалом. Для цієї колонки було застосовано спеціалізований кодек DoubleDelta. Замість збереження повного 64-бітного значення кожної мітки, кодек обчислює різницю (дельту) між сусідніми мітками, а потім – різницю між дельтами. У більшості випадків друга дельта дорівнює нулю або мінімальному числу, що дозволяє стиснути масив чисел до кількох біт на запис;
– колонки latitude та longitude (Float64): Географічні координати вантажного автомобіля під час рейсу змінюються плавно і без різких стрибків. Для оптимізації зберігання цих полів використано

архітектурний кодек Gorilla. Цей алгоритм, розроблений компанією Facebook для стиснення чисел із плаваючою комою, базується на операції виключного «АБО» (XOR) над послідовними значеннями. Якщо координати близькі, результат XOR містить велику кількість провідних нулів, які ефективно пакуються в бітові прапори;

– колонка speed (Float32): Для показників швидкості використано кодек Gorilla. Алгоритм оптимізований для роботи з числами з плаваючою комою та ефективно стискає послідовності близьких за значенням величин. Оскільки швидкість транспортного засобу змінюється поступово, використання Gorilla дозволяє зменшити обсяг даних без втрати точності та підвищити ефективність зберігання телеметрії.

Після відпрацювання кодеків кодування, ClickHouse застосовує до блоків даних універсальний алгоритм компресії LZ4 за замовчуванням або ZSTD (Zone for Testing Data). У межах даної роботи для стиснення даних було обрано алгоритм LZ4, оскільки він забезпечує максимальну швидкість декомпресії під час читання, практично не створюючи значного навантаження на центральний процесор. Це дозволяє досягти компресії даних у 7–10 разів порівняно із сирими JSON-пакетами, суттєво заощаджуючи ресурс PCIe 4.0 NVMe диска.

2.4 Розробка моделі валідації вхідних даних та взаємодії з СУБД

Основою серверної частини системи виступає FastAPI [21]. У межах цієї роботи використовувався синхронний клієнт СУБД ClickHouse, тому функції обробки ендпоінтів були реалізовані за допомогою стандартних декларацій def. Незважаючи на використання синхронного клієнта ClickHouse, продуктивність системи залишається достатньою для виконання поставлених завдань завдяки невеликому часу обробки окремих запитів. Сам API-інтерфейс спроєктовано з дотриманням базових REST-принципів [22]. Увесь потік координат від терміналів маршрутизується через єдиний POST-ендпоінт /ingest.

Під високим навантаженням сервер повинен швидко відсіювати некоректні пакети даних. Тому вхідна інформація підлягає обов'язковій перевірці через схеми Pydantic [18]. Бібліотека контролює типи та структуру геолокаційних пакетів. Структуру моделі валідації наведено на рисунку 2.3.

```
class LocationData(BaseModel):
    device_id: str
    lat: float
    lon: float
    speed: float
```

Рисунок 2.3 – Програмна модель валідації вхідних даних

Попередня валідація ізолює СУБД від невалідних JSON-пакетів. Як наслідок, обчислювальні потужності сервера не витрачаються марно, що є критично важливим для збереження цілісності архіву координат.

Для усунення накладних витрат на повторну ініціалізацію мережевих сесій та мінімізації затримок при обробці вхідних пакетів телеметрії в системі реалізовано механізм повторного використання підключень до аналітичного сховища [6]. Замість розгортання класичного пулу з'єднань у програмному коді бекенду використано архітектурний патерн Singleton на рівні потоку (per-thread singleton). Даний підхід реалізовано за допомогою стандартного інструменту threading.local(), який забезпечує ізоляцію клієнтського підключення до СУБД ClickHouse в межах кожного робочого потоку. Це дозволяє уникнути конфліктів під час паралельної обробки запитів та зменшити витрати на створення нових підключень до бази даних. Програмну реалізацію такого рішення наведено на рисунку 2.4, а повний лістинг серверної частини міститься у додатку А.

```
thread_local = threading.local()

def get_db():
    if not hasattr(thread_local, "client"):
        thread_local.client = clickhouse_connect.get_client(
            host='localhost',
            port=8123,
            username='default',
            password='admin'
        )
    return thread_local.client
```

Рисунок 2.4 – Реалізація механізму повторного використання підключень

2.5 Алгоритмічне забезпечення стійкості клієнтської частини

Специфіка експлуатації логістичних систем моніторингу вимагає стабільної роботи в умовах регулярних розривів мережевого з'єднання. Тому під час розробки клієнтського ПЗ першочергову увагу приділили архітектурним рішенням, які мінімізують ризик втрати просторових координат на глухих ділянках маршруту.

Основою розробленого механізму став локальний енергонезалежний файловий буфер. Стан мережевого підключення контролюється через аналіз відповідей транспортного сервера. Якщо POST-запити починають завершуватися помилками або таймаутами, застосунок автоматично переходить в автономний режим роботи.

Для детального аналізу динаміки взаємодії об'єктів у часі та верифікації логіки роботи файлового буфера було побудовано діаграму послідовності (Sequence Diagram). Вона покроково відображає життєвий цикл обміну повідомленнями між модулями системи у трьох станах: при штатній передачі телеметрії, у разі аварійної втрати зв'язку та під час фінальної синхронізації накопиченої черги. Графічну модель взаємодії представлено на рисунку 2.5.

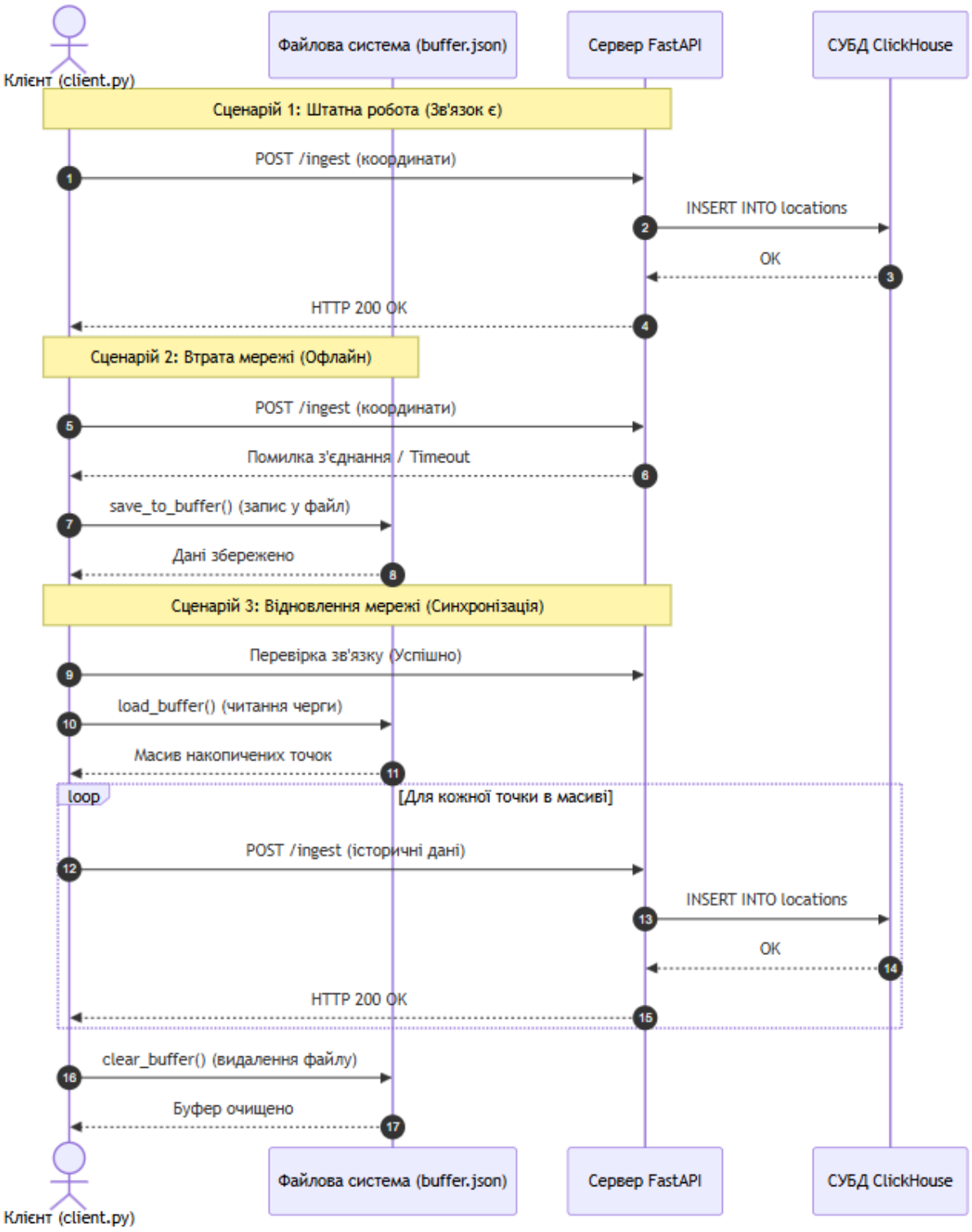


Рисунок 2.5 – Діаграма послідовності процесів передачі та буферування геоданих

За умов стабільного покриття мережі (Сценарій 1) процес обмежується прямою трансляцією координат від клієнта до FastAPI з миттєвим логуванням у

колонкову таблицю ClickHouse та поверненням клієнту підтвердження у вигляді HTTP-статусу 200. Сценарій відсутності зв'язку (Сценарій 2) демонструє ізоляцію бекенду: клієнтський скрипт перехоплює мережевий таймаут, блокує відправку в мережу та переспрямовує об'єкт у локальний структурований файл buffer.json.

Процес фіксації точок треку продовжується у штатному режимі, проте координати більше не надсилаються в мережу, а додаються до структурованого JSON-файлу на дисковому просторі пристрою. Фрагмент програмного коду, що відповідає за верифікацію підключення, локальне збереження об'єктів та синхронізацію, наведено на рисунку 2.6.

```
while True:
    current_lat += random.uniform(-0.005, 0.005)
    current_lon += random.uniform(-0.005, 0.005)

    if send_data(current_lat, current_lon, 60.0):
        saved_data = load_buffer()
        if saved_data:
            print(f"[*] Знайдено {len(saved_data)} записів у буфері. Синхронізую...")
            successful_count = 0

            for item in saved_data:
                if send_data(item['lat'], item['lon'], item['speed']):
                    successful_count += 1
                else:
                    break

            if successful_count == len(saved_data):
                clear_buffer()
                print("[*] Буфер очищено.")
            elif successful_count > 0:

                remaining_data = saved_data[successful_count:]
                with open(BUFFER_FILE, "w", encoding="utf-8") as f:
                    json.dump(remaining_data, f)
                print(f"[*] Перезаписано {len(remaining_data)} точок у буфер.")
    else:
        save_to_buffer({"lat": current_lat, "lon": current_lon, "speed": 60.0})

    time.sleep(3)
```

Рисунок 2.6 – Реалізація механізму файлового буферування на стороні клієнта

Блок відновлення (Сценарій 3) ілюструє циклічне вивантаження: коли пристрій повертається в зону стабільного покриття, запускається зворотний процес вивантаження черги. Клієнт послідовно читає локальний файл і надсилає пакети на сервер. Видалення тимчасового файлу виконується лише після отримання успішних статусів доставки для кожної точки, що мінімізує ризик втрати інформації через апаратні збої. Такий підхід робить накопичений трек стійким до перебоїв живлення автомобіля під час рейсів. Повний текст скриптів клієнтського модуля та генератора навантаження міститься у додатку Б.

2.6 Проєктування програмного інтерфейсу (API) та структура ендпоінтів

Комунікаційним ядром системи є REST API, розроблене на базі фреймворку FastAPI. Вибір цього інструменту зумовлений його високою продуктивністю завдяки підтримці асинхронного стандарту ASGI [12], а також автоматичній генерації інтерактивної документації за стандартом OpenAPI (Swagger).

Для забезпечення суворої типізації вхідних даних на рівні сервера використовується бібліотека Pydantic. Вона автоматично перевіряє структуру JSON-пакетів та типи даних ще до моменту їх обробки серверною частиною. Це захищає систему від критичних збоїв у разі надходження некоректно сформованих або "сміттєвих" пакетів.

Архітектура програмного інтерфейсу складається з чотирьох ключових маршрутів, кожен з яких виконує ізольовану функцію. Детальну специфікацію ендпоінтів наведено у таблиці 2.2.

Така структура API дозволяє зменшити мережеві затримки та спростити подальше масштабування системи. Завдяки розмежуванню маршрутів для запису (/ingest) та читання (/stats, /all_locations) закладається фундамент для

подальшого балансування навантаження, коли вузли обробки трафіку та вузли видачі аналітики можна буде масштабувати незалежно один від одного.

Таблиця 2.2 – Специфікація маршрутів програмного інтерфейсу (REST API)

Метод та URL	**Призначення**	**Тіло запиту / Параметри**	**Формат відповіді (JSON)**	**Коди стану**
POST /ingest	Прийом телеметрії від бортових пристроїв.	Тіло запиту (JSON): device_id (string), lat (float), lon (float), speed (float), заголовок Authorization: Bearer token.	{"status": "success"}	200 – Успішно 500 – Внутрішня помилка сервера
GET /stats/{device_id}	Отримання аналітичного зведення по конкретному автомобілю.	Параметр шляху: {device_id}, заголовок Authorization: Bearer token	{"device_id": str, "average_speed": float, "max_speed": float, "total_points": int}	200 – Успішно
GET /all_locations	Вивантаження останніх відомих координат усіх транспортних засобів для рендерингу карти.	Заголовок Authorization: Bearer token	[{"id": str, "lat": float, "lon": float}]	200 – Успішно
GET /	Відкриття веб-інтерфейсу моніторингу та завантаження HTML-сторінки карти	Без параметрів	HTML-документ	200 – успішно, 404 – Файл не знайдено

2.7 Концепція інформаційної безпеки та захист персональних даних

Важливою складовою проєктування сучасних систем супутникового моніторингу є забезпечення конфіденційності та цілісності інформації. Згідно з

положеннями європейського Загального регламенту про захист даних (GDPR), а також Закону України «Про захист персональних даних», точні геолокаційні координати, які дозволяють відстежити переміщення конкретної особи (водія) або прив'язаного до неї транспортного засобу, класифікуються як персональні дані [4, 9]. Це накладає суворі юридичні та технічні вимоги до архітектури сховищ.

У поточній реалізації прототипу для захисту від надсилання фіктивних POST-запитів із довільними значеннями device_id реалізовано механізм автентифікації клієнтів. Кожен бортовий термінал використовує унікальний токен доступу (Bearer Token). Сервер FastAPI верифікує цей токен у заголовку Authorization кожного запиту за допомогою вбудованих механізмів безпеки. У разі надходження запиту з недійсним або відсутнім токеном система автоматично відхиляє його зі статусом HTTP 401 Unauthorized, що забезпечує базовий рівень захисту від несанкціонованого втручання в архів телеметрії.

Для переведення системи у стан комерційної експлуатації (Production-Ready) архітектура передбачає впровадження багаторівневої концепції безпеки:

– захист транспортного рівня: Увесь трафік між бортовими пристроями та сервером має шифруватися за протоколом TLS 1.3 (HTTPS), що унеможливлює перехоплення координат методом "людина посередині" (Man-in-the-Middle) під час передачі через відкриті стільникові мережі 3G/4G;
– автентифікація пристроїв: Кожен бортовий термінал на етапі ініціалізації на виробництві повинен отримувати унікальний криптографічний API-ключ або використовувати підписані JWT-токени (JSON Web Tokens). Сервер FastAPI верифікуватиме цифровий підпис токена у заголовку Authorization кожного запиту. У разі компрометації пристрою, адміністратор зможе миттєво відкликати токен, не змінюючи логіку роботи всієї системи;
– мережева ізоляція (VPC): Аналітична база даних ClickHouse не повинна мати відкритого доступу до глобальної мережі (Public IP). У межах

Docker-середовища вона обмінюється даними виключно із сервером FastAPI через внутрішню ізольовану bridge-мережу.

Застосування цих превентивних заходів створює основу для підвищення рівня безпеки системи та її відповідності законодавству у сфері захисту персональних даних.

2.8 Обробка виняткових ситуацій та відмовостійкість серверної частини

Окрема увага під час проєктування приділялася сценаріям апаратної або програмної відмови бази даних. Якщо СУБД ClickHouse стає тимчасово недоступною (наприклад, через перезавантаження контейнера або збій дискової підсистеми), сервер FastAPI перехоплює виняток драйвера бази даних і повертає клієнту статус-код HTTP 500 Internal Server Error.

У поточній реалізації прототипу механізм відмовостійкості делеговано клієнтській частині: отримавши помилку від сервера або тайм-аут мережевого з’єднання, бортовий термінал не відкидає координати, а зберігає їх у локальному файловому буфері (buffer.json), ініціюючи повторні спроби (Retry Pattern) після відновлення нормальних відповідей від бекенду.

Водночас для переведення системи у стан комерційної експлуатації архітектура передбачає впровадження проміжного шару брокера повідомлень (Message Broker) на базі Redis або RabbitMQ. У такому разі FastAPI функціонуватиме виключно як Producer, миттєво складаючи вхідні JSON-пакети в оперативну чергу. Окремі фонові процеси будуть асинхронно вичитувати цю чергу і виконувати пакетну вставку в ClickHouse. Застосування такого підходу дозволить серверу безперебійно приймати телеметрію навіть при повному падінні бази даних, формуючи надійну серверну чергу очікування.

3 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ТЕСТУВАННЯ СИСТЕМИ

3.1 Конфігурація тестового стенда та параметри середовища

Для забезпечення репрезентативності та можливості відтворення результатів експериментів навантажувальне тестування програмного комплексу проводилося на виділеному апаратному стенді. Фізична конфігурація сервера базувалася на шестиядерному процесорі AMD Ryzen 5 5600 (архітектура Zen 3) з базовою тактовою частотою 3.5 ГГц. Обсяг оперативної пам’яті становив 32 ГБ (стандарт DDR4 з частотою 3200 МГц), що гарантувало відсутність використання файлу підкачки (swap) під час пікових навантажень на базу даних.

Оскільки колонкові бази даних є вкрай чутливими до швидкості дискової підсистеми, як постійний носій використовувався твердотільний накопичувач Kingston NV2 ємністю 1 ТБ (форм-фактор M.2 2280) з інтерфейсом NVMe PCIe 4.0 x4.

Програмне середовище функціонувало під управлінням операційної системи Windows 11. Серверна частина, включаючи СУБД та бекенд, була ізольована за допомогою платформи контейнеризації Docker Desktop версії 4.63.0 [15]. Для виконання серверної логіки застосовувався інтерпретатор мови Python версії 3.11 у зв’язці з ASGI-сервером Uvicorn. Як аналітичне сховище використовувався офіційний Docker-образ ClickHouse Server (версії 24.3-alpine). Усі компоненти розгорталися у межах єдиної віртуальної підмережі (bridge network), що мінімізувало затримки на маршрутизацію трафіку між застосунком та базою даних.

3.2 Тестування функціональних можливостей аналітичного модуля

На першому етапі перевірялася здатність ClickHouse віддавати коректну аналітику через API системи. Спочатку за допомогою клієнтського скрипта було завантажено до бази даних тестовий набір координат для віртуальної вантажівки truck-01. Далі ініціювали GET-запит до ендпоінта /stats/truck-01. Відповідь сервера у форматі JSON зображено на рисунку 3.1.

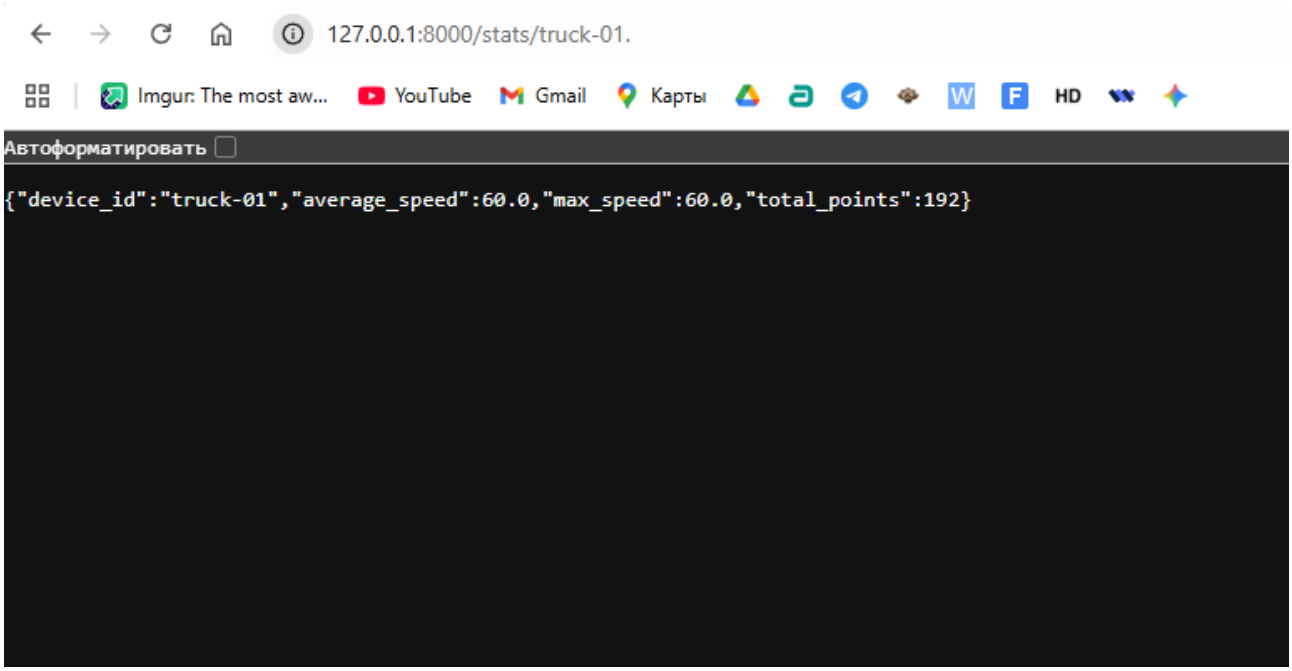
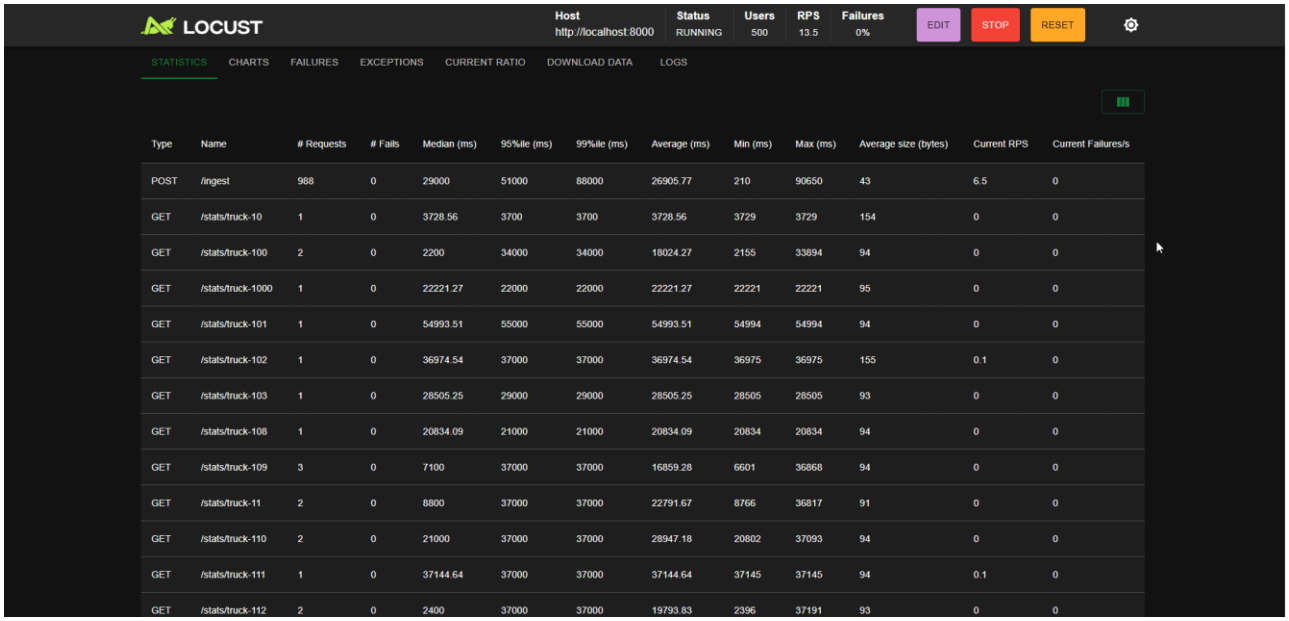


Рисунок 3.1 – Результат виконання аналітичного запиту щодо статистики об’єкта

Як видно, серверна частина успішно виконує агрегацію в режимі реального часу. Система виконує розрахунок середньої та максимальної швидкості транспортного засобу на основі всіх накопичених записів телеметрії.

3.3 Аналіз продуктивності системи при високому навантаженні

Для оцінки поведінки системи під високим навантаженням було розгорнуто стрес-тести через інструмент Locust [17]. У сценарії задано імітацію 500 конкурентних користувачів. Вони одночасно генерували безперервний потік POST-запитів з координатами та GET-запитами для читання статистики. Результати першого прогону зведено на рисунку 3.2.



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/ingest	988	0	29000	51000	88000	26905.77	210	90650	43	6.5	0
GET	/stats/truck-10	1	0	3728.56	3700	3700	3728.56	3729	3729	154	0	0
GET	/stats/truck-100	2	0	2200	34000	34000	18024.27	2155	33894	94	0	0
GET	/stats/truck-1000	1	0	22221.27	22000	22000	22221.27	22221	22221	95	0	0
GET	/stats/truck-101	1	0	54993.51	55000	55000	54993.51	54994	54994	94	0	0
GET	/stats/truck-102	1	0	36974.54	37000	37000	36974.54	36975	36975	155	0.1	0
GET	/stats/truck-103	1	0	28505.25	29000	29000	28505.25	28505	28505	93	0	0
GET	/stats/truck-108	1	0	20834.09	21000	21000	20834.09	20834	20834	94	0	0
GET	/stats/truck-109	3	0	7100	37000	37000	16859.28	6601	36868	94	0	0
GET	/stats/truck-11	2	0	8800	37000	37000	22791.67	8766	36817	91	0	0
GET	/stats/truck-110	2	0	21000	37000	37000	28947.18	20802	37093	94	0	0
GET	/stats/truck-111	1	0	37144.64	37000	37000	37144.64	37145	37145	94	0.1	0
GET	/stats/truck-112	2	0	2400	37000	37000	19793.83	2396	37191	93	0	0

Рисунок 3.2 – Зведена статистика навантажувального тестування

Перші метрики виявилися проблемними: середній час відповіді (ART) зріс до 26,9 с. Аналіз результатів показав, що основною причиною деградації продуктивності стала велика кількість одночасних HTTP-запитів, які створювали значне навантаження на серверний застосунок FastAPI та мережевий стек операційної системи. Графіки динаміки запитів і зміни часу відгуку наведено на рисунку 3.3.

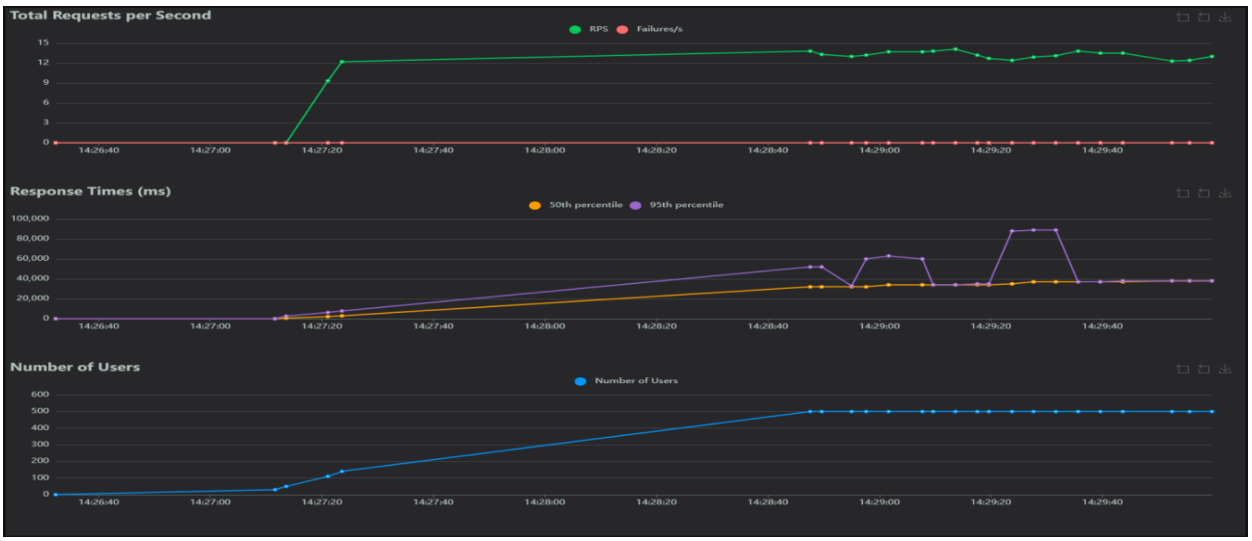
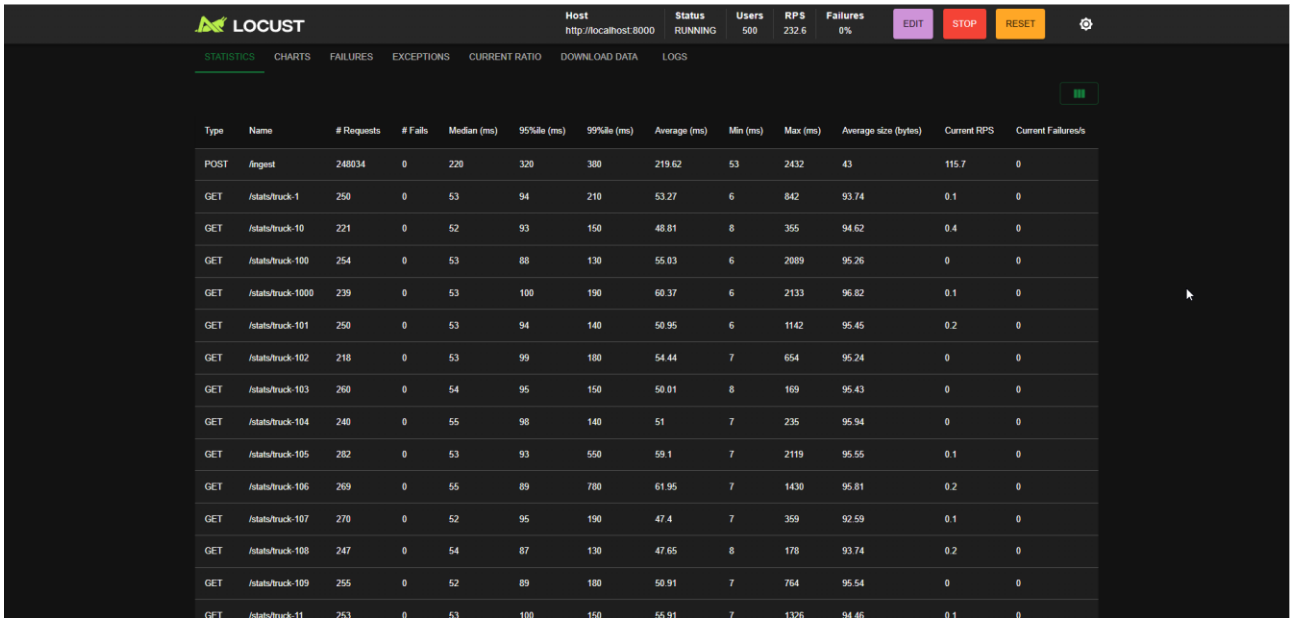


Рисунок 3.3 – Графіки динаміки запитів та часу відгуку системи

Було запущено повторне тестування, яке тривало протягом 30 хвилин безперервної роботи. Воно продемонструвало стабілізацію часу відгуку та збільшення пропускної здатності. Статистику оптимізованої системи та графік стабільної пропускної здатності наведено на рисунках 3.4 та 3.5.



Рисунок 3.4 – Зведена статистика оптимізованої системи



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/ingest	248034	0	220	320	380	219.62	53	2432	43	115.7	0
GET	/stats/truck-1	250	0	53	94	210	53.27	6	842	93.74	0.1	0
GET	/stats/truck-10	221	0	52	93	150	48.81	8	355	94.62	0.4	0
GET	/stats/truck-100	254	0	53	88	130	55.03	6	2089	95.26	0	0
GET	/stats/truck-1000	239	0	53	100	190	60.37	6	2133	96.82	0.1	0
GET	/stats/truck-101	250	0	53	94	140	50.95	6	1142	95.45	0.2	0
GET	/stats/truck-102	218	0	53	99	180	54.44	7	654	95.24	0	0
GET	/stats/truck-103	260	0	54	95	150	50.01	8	169	95.43	0	0
GET	/stats/truck-104	240	0	55	98	140	51	7	235	95.94	0	0
GET	/stats/truck-105	282	0	53	93	550	59.1	7	2119	95.55	0.1	0
GET	/stats/truck-106	269	0	55	89	780	61.95	7	1430	95.81	0.2	0
GET	/stats/truck-107	270	0	52	95	190	47.4	7	359	92.59	0.1	0
GET	/stats/truck-108	247	0	54	87	130	47.65	8	178	93.74	0.2	0
GET	/stats/truck-109	255	0	52	89	180	50.91	7	764	95.54	0	0
GET	/stats/truck-11	253	0	53	100	150	55.91	7	1326	94.46	0.1	0

Рисунок 3.5 – Графік стабільної пропускної здатності RPS та часу відгуку

Пропускна здатність зафіксувалася на стабільній позначці 230-240 RPS. Сервер успішно обробив увесь тестовий потік запитів без помилок. Отже, проведена оптимізація дозволила стабілізувати роботу системи.

Для визначення граничної межі працездатності системи навантажувальне тестування було розширено до екстремальних значень конкурентних підключень. Утиліта Locust генерувала одночасні запити від 1000 та 2000 віртуальних клієнтів.

Під час тестування із навантаженням 1000 одночасних користувачів було зафіксовано різке погіршення якості обслуговування запитів. Частка невдалих запитів (Failures) перевищувала 79%, а середній час відповіді зростав до неприйнятних значень. При цьому апаратні ресурси тестового стенда не були повністю вичерпані, що свідчить про наявність архітектурних обмежень серверної частини застосунку.

Подальше збільшення навантаження до 2000 одночасних підключень не призвело до пропорційного зростання продуктивності та супроводжувалося подальшою деградацією стабільності роботи системи. Аналіз результатів тестування показав, що вузьким місцем виступає серверний застосунок FastAPI та механізм обробки великої кількості одночасних HTTP-з’єднань, а не СУБД

ClickHouse, яка продовжувала коректно виконувати операції запису та агрегації даних.

Отримані результати демонструють, що поточна реалізація системи придатна для роботи під помірним навантаженням, однак для експлуатації в середовищах із тисячами одночасно активних об’єктів потребує подальшої оптимізації та горизонтального масштабування серверної інфраструктури.

3.4 Перевірка стійкості клієнтської частини до розривів зв’язку

Робота енергонезалежного файлового буфера також потребувала практичної перевірки. З метою імітації критичного збою мережі та перевірки стійкості системи, під час активної сесії генерації телеметрії транспортний шлюз було примусово вимкнено.

Клієнтський додаток зафіксував помилку відсутності з’єднання за таймаутом і автоматично перейшов в автономний режим роботи. Процес генерації координат при цьому не припинявся, проте замість відправки HTTP-запитів клієнт почав послідовно записувати просторово-часові точки [19] у локальний файл buffer.json на постійному носії пристрою. Логи перемикання станів та фіксації збоїв зв’язку зафіксовано на рисунку 3.6.

Для перевірки ключової переваги нової архітектури-енергонезалежності – під час перебування в офлайні клієнтський пристрій було штучно перезавантажено. Після повторного старту операційної системи та запуску клієнтського скрипта, додаток успішно перевірив наявність файлу buffer.json, зчитав його вміст у чергу та продовжив очікування мережі без втрати жодного попереднього запису.

Щойно зв’язок із сервером було відновлено, клієнт автоматично ініціював процес зворотної синхронізації. Увесь накопичений файловий буфер був порційно та послідовно переданий до СУБД ClickHouse, після чого локальний файл було успішно видалено. Тестування підтвердило цілісність підсумкового

треку маршруту та стійкість накопичених даних до збоїв живлення бортового обладнання вантажівки.

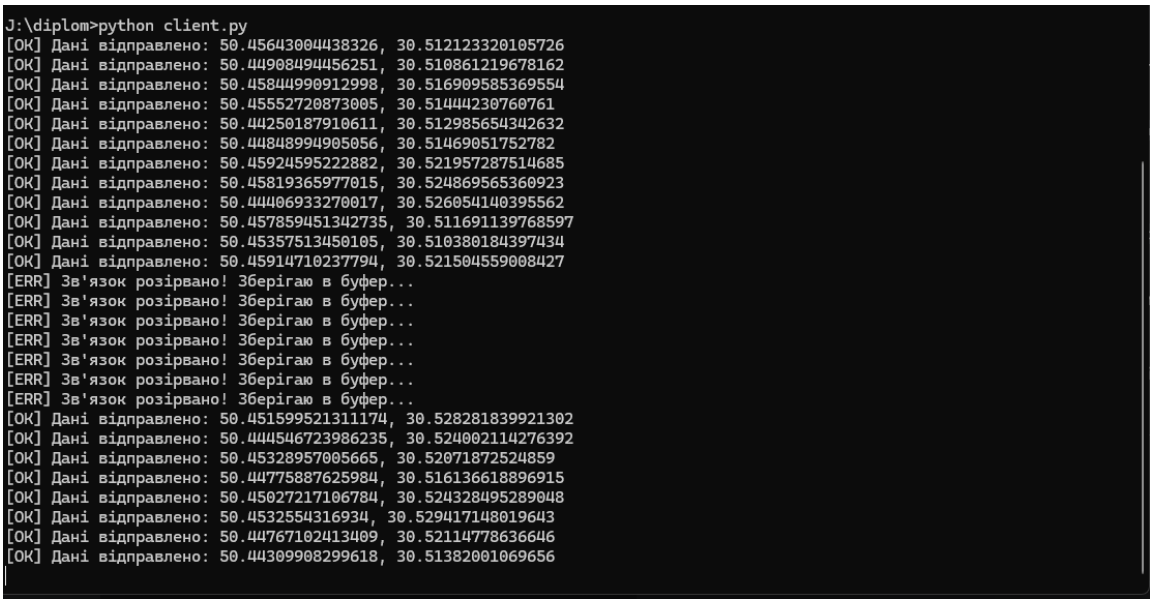


Рисунок 3.6 – Робота клієнтського модуля в умовах відсутності зв’язку

Щодо ємнісних показників, то під час тестування було змодельовано стан повної відсутності мережевого покриття протягом 90 хвилин. За умови генерації телеметрії з частотою одна точка кожні 3 секунди, у локальному файлі buffer.json накопичилося 5400 записів. Враховуючи, що один серіалізований об’єкт JSON із координатами займає приблизно 75 байт, накопичений масив зайняв близько 400 КБ дискового простору. Оскільки максимальна ємність файлового буфера програмно не обмежується, пристрій здатний тижнями акумулювати координати в офлайн-режимі без ризику переповнення пам’яті.

3.5 Оптимізація клієнтської візуалізації великих масивів геоданих у браузері

Важливим етапом експериментального дослідження розробленого програмного комплексу став аналіз продуктивності клієнтської частини системи моніторингу. Оскільки диспетчерський інтерфейс призначений для

відображення великої кількості транспортних засобів у режимі реального часу, необхідно було забезпечити прийнятну швидкодію браузера під час регулярного оновлення координат.

Для реалізації картографічного інтерфейсу було використано JavaScript-бібліотеку Leaflet.js та відкриті картографічні тайли OpenStreetMap. Такий підхід дозволяє відображати географічні об’єкти без необхідності використання комерційних картографічних сервісів та забезпечує достатню продуктивність для задач диспетчерського моніторингу.

Оновлення положення транспортних засобів реалізовано за допомогою періодичних HTTP-запитів до серверного ендпоінта /all_locations. Клієнтська частина кожні дві секунди виконує асинхронний запит до сервера, отримує актуальні координати всіх активних пристроїв у форматі JSON та автоматично оновлює положення відповідних маркерів на карті.

Для мінімізації навантаження на браузер застосовано механізм повторного використання маркерів. Після першого створення маркер зберігається у локальній структурі даних JavaScript. Під час чергового оновлення координат система не створює новий графічний об’єкт, а лише змінює географічне положення вже існуючого маркера методом setLatLng(). Це дозволяє уникнути накопичення зайвих DOM-елементів та забезпечує плавне оновлення положення транспортних засобів на карті.

Під час тестування було згенеровано значну кількість віртуальних транспортних засобів, координати яких зберігалися у базі даних ClickHouse та відображалися через веб-інтерфейс. Аналіз роботи браузера показав відсутність помітних затримок під час масштабування карти, переміщення області перегляду та автоматичного оновлення координат об’єктів.

Таким чином, використання бібліотеки Leaflet.js у поєднанні з асинхронним отриманням даних через REST API забезпечило достатню продуктивність клієнтської частини системи та дозволило реалізувати оперативний моніторинг транспортних засобів у режимі реального часу.

3.6 Візуалізація геолокаційних даних у реальному часі

Координати самі по собі не мають цінності без зручної панелі моніторингу. Для візуалізації треків було зібрано веб-інтерфейс за допомогою бібліотеки Leaflet.js [16]. Клієнтська частина браузера кожні 2 секунди періодично надсилає GET-запити до програмного інтерфейсу /all_locations. З боку бази тут працює функція argMax, яка отримує з ClickHouse виключно останні відомі координати для кожного пристрою моніторингу.

Для тестування мапи в умовах, наближених до реальної експлуатації, було створено генератор руху. Він імітує набір віртуальних транспортних об’єктів міжнародної логістичної компанії, імітуючи розташування автомобілів у ключових транспортних вузлах України та ЄС. Інтерфейс системи наведено на рис. 3.7.

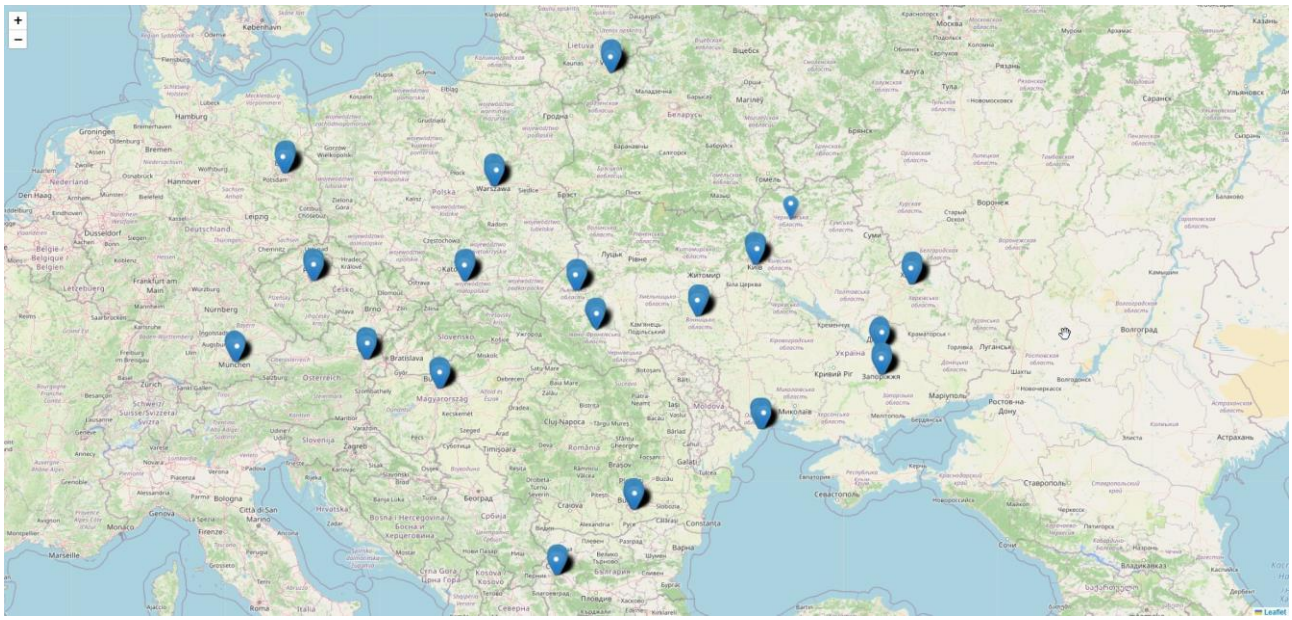


Рисунок 3.7 – Інтерфейс моніторингу вантажівок у реальному часі

Тестування карти показало, що всі компоненти системи працюють злагоджено. Дані без втрат проходять весь шлях: від збереження на пристрої під час офлайну до запису в базу та оперативного відображення на екрані диспетчера. Програмний код веб-інтерфейсу візуалізації наведено у додатку В.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було спроєктовано та реалізовано високопродуктивну систему збору та аналізу геолокаційних даних. На основі проведеного дослідження можна зробити наступні висновки:

- розроблено гібридну архітектуру серверної частини на базі фреймворку FastAPI. Використання FastAPI та оптимізація обробки запитів дозволили забезпечити високу пропускну здатність системи під час прийому інтенсивних потоків просторово-часових даних;
- обґрунтовано та впроваджено використання СУБД ClickHouse як основного сховища даних. Використання колонкового зберігання та рушія MergeTree дозволило забезпечити швидке виконання агрегаційних запитів (середня швидкість, пробіг) навіть при накопиченні значних масивів інформації;
- створено стійкий клієнтський додаток із алгоритмом енергонезалежного файлового буферування. Експериментально підтверджено, що такий підхід забезпечує безперервне накопичення координат під час автономного режиму, гарантує збереження даних при критичних збоях живлення пристрою та суттєво знижує ризик втрати телеметрії після стабілізації мережевого з’єднання;
- проведено навантажувальне тестування системи. Результати показали стабільну роботу програмного комплексу при навантаженні до 500 одночасних користувачів без втрати запитів та суттєвої деградації продуктивності. Подальше збільшення навантаження до 1000 одночасних клієнтів призводить до появи помилок з’єднання та поступового зниження пропускної здатності, що свідчить про необхідність подальшого горизонтального масштабування серверної інфраструктури;
- розроблений програмний комплекс може бути використаний як основа для побудови комерційних рішень у сфері логістики, GPS-моніторингу та IoT-платформ.

ПЕРЕЛІК ПОСИЛАНЬ

1. Аналітичний звіт щодо ринку телематики комерційного транспорту в Європі (Berg Insight). URL : https://www.berginsight.com/fleet-management-in-europe (дата звернення : 15.05.2026).

2. Архітектурні шаблони проектування IoT-рішень: REST API проти MQTT. URL : https://cloud.google.com/architecture/iot-core-mqtt-vs-http (дата звернення : 20.05.2026).

3. Бойко О. В., Коваленко О. С. Аналіз сучасних колонкових СУБД для обробки та зберігання просторово-часових даних. *Сучасні інформаційні системи*. 2023. Т. 7, № 2. С. 45–52.

4. Загальний регламент про захист даних (GDPR) – Регламент (ЄС) 2016/679 Європейського Парламенту і Ради від 27 квітня 2016 року. URL : https://eur-lex.europa.eu/eli/reg/2016/679/oj (дата звернення : 22.05.2026).

5. Клеппман М. Проєктування систем, що працюють з даними. Надійні, масштабовані та зручні у підтримці програми. Київ : Форс Україна, 2020. 608 с.

6. Мельник А. О., Шевченко В. І. Оптимізація обробки запитів та керування з'єднаннями у високонавантажених REST API системах. *Вісник комп'ютерних технологій*. 2024. № 14. С. 112–118.

7. Офіційна документація та архітектурні рішення платформи Wialon. URL : https://help.wialon.com/en/wialon-platform/user-guide (дата звернення : 16.05.2026).

8. Офіційна специфікація протоколу MQTT версії 5.0 (OASIS Standard). URL : https://mqtt.org/mqtt-specification/ (дата звернення : 20.05.2026).

9. Про захист персональних даних: Закон України від 01.06.2010 № 2297-VI. URL : https://zakon.rada.gov.ua/laws/show/2297-17 (дата звернення : 22.05.2026).

10. Специфікація розгортання On-Premise платформи Navixy. URL : https://www.navixy.com/ (дата звернення : 16.05.2026).

11. Технічні характеристики апаратних рішень Fleet Complete. URL : https://www.fleetcomplete.com/ (дата звернення : 17.05.2026).

12. Caleb H. Python Concurrency with asyncio. Shelter Island : Manning Publications, 2022. 352 p.

13. ClickBench: A Benchmark for Analytical Databases. URL : https://benchmark.clickhouse.com/ (дата звернення : 19.05.2026).

14. ClickHouse. Fast Open-Source OLAP DBMS. ClickHouse Documentation. URL : https://clickhouse.com/docs/en/ (дата звернення : 20.04.2026).

15. Docker Compose. Docker Documentation. URL : https://docs.docker.com/compose/ (дата звернення : 15.04.2026).

16. Leaflet. An open-source JavaScript library for mobile-friendly interactive maps. URL : https://leafletjs.com/ (дата звернення : 22.04.2026).

17. Locust. An open source load testing tool. URL : https://locust.io/ (дата звернення : 21.04.2026).

18. Pydantic. Data validation using Python type hints. URL : https://docs.pydantic.dev/ (дата звернення : 19.04.2026).

19. Python requests module. Requests : HTTP for Humans. URL : https://requests.readthedocs.io/ (дата звернення : 17.04.2026).

20. QuestDB Performance Benchmarks. URL : https://questdb.io/docs/reference/benchmarks/ (дата звернення : 19.05.2026).

21. Ramírez S. FastAPI. FastAPI Official Documentation. URL : https://fastapi.tiangolo.com/ (дата звернення : 18.04.2026).

22. Richardson L., Amundsen M., Ruby S. RESTful Web APIs : Services for a Changing World. Sebastopol : O’Reilly Media, 2013. 406 p.

23. Time Series Benchmark Suite (TSBS). URL : https://github.com/timescale/tsbs (дата звернення : 18.05.2026).

ДОДАТОК А

Лістинг серверної частини

Серверна частина на FastAPI з налаштуванням пулу з’єднань та ендпоінтами для збору й аналізу даних.

```
from contextlib import asynccontextmanager
from fastapi import FastAPI, HTTPException, Security, status
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
from fastapi.responses import HTMLResponse
from pydantic import BaseModel
import clickhouse_connect
from datetime import datetime, timezone
import threading
import os

security = HTTPBearer()
EXPECTED_TOKEN = os.getenv("API_TOKEN", "secret_device_token_123")

def verify_token(credentials: HTTPAuthorizationCredentials =
Security(security)):
    if credentials.credentials != EXPECTED_TOKEN:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid or missing token"
        )
    return credentials.credentials

@asynccontextmanager
async def lifespan(app: FastAPI):
    startup_db = clickhouse_connect.get_client(
        host='localhost',
    port=8123,
    username='default',
    password='admin'
    )
    startup_db.command("""
    CREATE TABLE IF NOT EXISTS locations (
    device_id String,
    timestamp DateTime64(3, 'UTC') CODEC(DoubleDelta, LZ4),
    latitude Float64 CODEC(Gorilla, LZ4),
    longitude Float64 CODEC(Gorilla, LZ4),
    speed Float32 CODEC(Gorilla, LZ4)
)
ENGINE = MergeTree()
PARTITION BY toYYYYMM(timestamp)
ORDER BY (device_id, timestamp)
    """)
    yield

app = FastAPI(title="Geo-Ingestor API", lifespan=lifespan)
```

```
thread_local = threading.local()

def get_db():
    if not hasattr(thread_local, "client"):
        thread_local.client = clickhouse_connect.get_client(
             host='localhost',
              port=8123,
              username='default',
               password='admin'
        )
    return thread_local.client

class LocationData(BaseModel):
    device_id: str
    lat: float
    lon: float
    speed: float

@app.post("/ingest")
def ingest_location(data: LocationData, token: str =
Security(verify_token)):
    try:
        db = get_db()
        current_time = datetime.now(timezone.utc)
        row = [data.device_id, current_time, data.lat, data.lon, data.speed]

        db.insert(
            'locations',
            [row],
            column_names=['device_id', 'timestamp', 'latitude', 'longitude',
'speed']
        )
        return {"status": "success"}

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.get("/stats/{device_id}")
def get_device_stats(device_id: str, token: str = Security(verify_token)):
    clean_id = device_id.strip().rstrip('.')
    db = get_db()

    query = """
    SELECT avg(speed), max(speed), count(*)
    FROM locations
    WHERE device_id = {did:String}
    """

    try:
        result = db.query(query, parameters={'did': clean_id})
        row = result.result_rows[0]
        total_points = row[2]

        if total_points == 0:
            return {
                "device_id": clean_id,
                "total_points": 0,
                "message": "Дані відсутні"
            }
```

```
        return {
            "device_id": clean_id,
            "average_speed": round(row[0], 2),
            "max_speed": row[1],
            "total_points": total_points
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.get("/all_locations")
def get_all_locations(token: str = Security(verify_token)):
    db = get_db()
    query = """
    SELECT
        device_id,
        argMax(latitude, timestamp),
        argMax(longitude, timestamp)
    FROM locations
    GROUP BY device_id
    """
    try:
        result = db.query(query)
        return [
            {"id": r[0], "lat": r[1], "lon": r[2]}
            for r in result.result_rows
        ]
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.get("/", response_class=HTMLResponse)
def get_map_page():
    try:
        with open("index.html", "r", encoding="utf-8") as f:
            return f.read()
    except FileNotFoundError:
        raise HTTPException(status_code=404, detail="Файл index.html не
знайдено")
```

Конфігурація для розгортання контейнера з базою даних ClickHouse.

```
services:
  clickhouse:
    image: clickhouse/clickhouse-server:latest
    container_name: clickhouse_diplom
    environment:
      - CLICKHOUSE_PASSWORD=admin
    ports:
      - "8123:8123"
      - "9000:9000"
    volumes:
      - ch_data:/var/lib/clickhouse
    ulimits:
      nofile:
        soft: 262144
        hard: 262144

volumes:
  ch_data:
```

ДОДАТОК Б

Лістинг клієнтської частини та симулятора навантаження

Скрипт клієнтської частини з реалізацією енергонезалежного файлового буфера для забезпечення стійкості до розривів зв’язку.

```
import requests
import time
import random
import json
import os

SERVER_URL = "http://localhost:8000/ingest"
DEVICE_ID = "truck-01"
BUFFER_FILE = "buffer.json"

def send_data(lat, lon, speed):
    payload = {
        "device_id": DEVICE_ID,
        "lat": lat,
        "lon": lon,
        "speed": speed
    }

    headers = {
        "Authorization": "Bearer secret_device_token_123"
    }

    try:
        response = requests.post(SERVER_URL, json=payload, headers=headers,
timeout=2)
        if response.status_code == 200:
            print(f"[OK] Дані відправлено: {lat}, {lon}")
            return True
    except requests.RequestException:
        print("[ERR] Зв’язок розірвано! Зберігаю у файл...")
    return False

def save_to_buffer(data):
    buffer = load_buffer()
    buffer.append(data)
    with open(BUFFER_FILE, "w", encoding="utf-8") as f:
        json.dump(buffer, f)

def load_buffer():
    if os.path.exists(BUFFER_FILE):
        with open(BUFFER_FILE, "r", encoding="utf-8") as f:
            try:
                return json.load(f)
            except json.JSONDecodeError:
                return []
```

```
    return []

def clear_buffer():
    if os.path.exists(BUFFER_FILE):
        os.remove(BUFFER_FILE)

current_lat = 48.4647
current_lon = 35.0462

while True:
    current_lat += random.uniform(-0.005, 0.005)
    current_lon += random.uniform(-0.005, 0.005)

    if send_data(current_lat, current_lon, 60.0):
        saved_data = load_buffer()
        if saved_data:
            print(f"[*] Знайдено {len(saved_data)} записів у буфері.
Синхронізую...")
            successful_count = 0

            for item in saved_data:
                if send_data(item['lat'], item['lon'], item['speed']):
                    successful_count += 1
                else:
                    break

            if successful_count == len(saved_data):
                clear_buffer()
                print("[*] Буфер очищено.")
            elif successful_count > 0:

                remaining_data = saved_data[successful_count:]
                with open(BUFFER_FILE, "w", encoding="utf-8") as f:
                    json.dump(remaining_data, f)
                print(f"[*] Перезаписано {len(remaining_data)} точок у
буфер.")
    else:
        save_to_buffer({"lat": current_lat, "lon": current_lon, "speed":
60.0})

    time.sleep(3)
```

Сценарій для навантажувального тестування з координатами міст України та Європи.

```
import random
from locust import HttpUser, task, between

CITIES = [
    (50.4501, 30.5234),
    (49.8397, 24.0297),
    (46.4825, 30.7233),
    (48.4647, 35.0461),
    (49.9935, 36.2304),
    (47.8388, 35.1396),
    (49.2331, 28.4682),
    (48.9226, 24.7111),
    (52.2297, 21.0122),
    (50.0647, 19.9450),
```

```
    (56.9496, 24.1052),
    (54.6872, 25.2797),
    (42.6977, 23.3219),
    (59.4370, 24.7536),
    (52.5200, 13.4050),
    (48.1351, 11.5820),
    (48.8566, 2.3522),
    (50.0755, 14.4378),
    (44.4268, 26.1025),
    (48.2082, 16.3738),
    (47.4979, 19.0402),
]

HEADERS = {
    "Authorization": "Bearer secret_device_token_123"
}

class GeoClientUser(HttpUser):

    wait_time = between(1.0, 3.0)

    @task(5)
    def send_geo_data(self):

        truck_id = f"truck-{random.randint(1, 1000)}"

        base_lat, base_lon = random.choice(CITIES)

        lat = base_lat + random.uniform(-0.1, 0.1)
        lon = base_lon + random.uniform(-0.1, 0.1)

        payload = {
            "device_id": truck_id,
            "lat": lat,
            "lon": lon,
            "speed": random.uniform(40, 90)
        }

        self.client.post(
            "/ingest",
            json=payload,
            headers=HEADERS
        )

    @task(1)
    def get_stats(self):

        truck_id = f"truck-{random.randint(1, 1000)}"

        self.client.get(
            f"/stats/{truck_id}",
            headers=HEADERS
        )
```

ДОДАТОК В

Лістинг веб-інтерфейсу візуалізації

Фронтенд-частина з інтерактивною картою на Leaflet.js для візуалізації вантажівок.

```
<!DOCTYPE html>
<html>
<head>
    <title>Моніторинг вантажівок</title>
    <link rel="stylesheet"
href="https://unpkg.com/leaflet@1.9.4/dist/leaflet.css" />
    <script
src="https://unpkg.com/leaflet@1.9.4/dist/leaflet.js"></script>
    <style> #map { height: 100vh; width: 100%; } body { margin: 0; }
</style>
</head>
<body>
    <div id="map"></div>
    <script>
    var map = L.map('map').setView([48.37, 31.16], 6);

    L.tileLayer(
        'https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png'
    ).addTo(map);

    var markers = {};

    async function updateMarkers() {

        const response = await fetch('/all_locations', {
            headers: {
                'Authorization': 'Bearer secret_device_token_123'
            }
        });
```

```
        const data = await response.json();

        data.forEach(truck => {

            if (markers[truck.id]) {

                markers[truck.id].setLatLng([
                    truck.lat,
                    truck.lon
                ]);

            } else {

                markers[truck.id] = L.marker([
                    truck.lat,
                    truck.lon
                ])
                .addTo(map)
                .bindPopup("Вантажівка: " + truck.id);
            }
        });
    }

    updateMarkers();
    setInterval(updateMarkers, 2000);
</script>
</body>
</html>
```