Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Автоматизації, комп'ютерно-інтегрованих технологій та систем
(повна назва)

Кафедра Комп'ютерно-інтегрованих технологій, автоматизації робототехніки та безпекової інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Розроблення системи віддаленого керування промисловим обладнанням з використанням IoT-технології.
(тема)

Виконав:
здобувач 4 року навчання,
групи АКТСІ-22-1
Денис ЛЯХОВ
(власне ім'я, прізвище)

Спеціальність 151 Автоматизація та комп'ютерно-інтегровані технології
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Системна інженерія
(повна назва освітньої програми)

Керівник доц. Артем БРОННІКОВ
(посада, власне ім'я, прізвище)

Допускається до захисту
Завідувач кафедри КІТАРБІ _____ Ігор НЕВЛЮДОВ
(підпис) (власне ім'я, прізвище)

2026 р.

Я, Ляхов Денис Володимирович, як здобувач вищої освіти ХНУРЕ, розумію та підтримую політику закладу з академічної доброчесності. Я не надавав і не одержував недозволену допомогу під час підготовки кваліфікаційної роботи. Я не використовував штучний інтелект для підготовки кваліфікаційної роботи. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

«28» червня 2026 р.  Денис ЛЯХОВ

(підпис)

Харківський національний університет радіоелектроніки

Факультет АКІТС

Кафедра КІТАРБІ

Рівень вищої освіти перший (бакалаврський)

Спеціальність 151 Автоматизація та комп'ютерно-інтегровані технології

(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Системна інженерія

(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«__»___20 _____ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Ляхову Денису Володимировичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розроблення системи віддаленого керування промисловим обладнанням з використанням IoT-технології
затверджена наказом університету від "04" червня 2026р. № 473 Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії 10 червня 2026 р.
3. Вихідні дані до роботи
3.1 Віддалене керування промисловим обладнанням;
3.2 Програмний симулятор насосної станції;
3.3 IoT-взаємодія між симулятором, серверною частиною та веб-інтерфейсом.
4. Перелік питань, що потрібно опрацювати в роботі
4.1 Аналіз стану систем віддаленого моніторингу та керування промисловим обладнанням;
4.2 Вибір та обґрунтування програмно-технічних засобів системи;
4.3 Розроблення прототипу системи віддаленого керування насосною станцією з використанням IoT-технології, який включає симулятор обладнання, MQTT-обмін, серверну частину, базу даних, веб-інтерфейс та аварійну логіку;
4.4 Оформлення пояснювальної записки кваліфікаційної роботи.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри)

Графічний матеріал у вигляді презентації – 17 арк. ф. А4,

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	Дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Аналіз літератури за темою	01.05.2026	Виконано
2	Аналіз літератури та предметної області віддаленого керування промисловим обладнанням	06.05.2026	Виконано
3	Вибір та обґрунтування технічних засобів системи	19.05.2026	Виконано
4	Розробка програмного забезпечення системи віддаленого керування обладнанням	04.06.2026	Виконано
5	Подання роботи на перевірку Інтернет-сервісом StrikePlagiarism	17.06.2026	Виконано
6	Оформлення пояснювальної записки	20.06.2026	Виконано
7	Подання роботи на рецензію	25.06.2026	Виконано
8	Подання роботи на підпис зав. кафедри	30.06.2026	Виконано
9	Подання кваліфікаційної роботи в ЕК	10.07.2026	Виконано

Дата видачі завдання 27 квітня 2026 р.

Здобувач _____ Денис ЛЯХОВ
(підпис) (власне ім'я, прізвище)

Керівник роботи _____ доц. Артем БРОННІКОВ
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка: 119 с.; 7 табл.; 19 рис.; 3 дод.; 19 джерел.

АВАРІЙНА ЛОГІКА, ВЕБ-ІНТЕРФЕЙС, ВІДДАЛЕНЕ КЕРУВАННЯ, DOCKER COMPOSE, IOT-ТЕХНОЛОГІЯ, MQTT, НАСОСНА СТАНЦІЯ, СИМУЛЯТОР, ТЕЛЕМЕТРІЯ.

Об’єкт розробки – процес віддаленого моніторингу та керування промисловим обладнанням.

Предмет розробки – програмна система віддаленого керування насосною станцією з використанням IoT-технології.

Мета роботи – підвищення ефективності керування насосною станцією за рахунок розробки локального прототипу системи віддаленого моніторингу її параметрів

У першому розділі було проведено аналіз систем віддаленого моніторингу та керування промисловим обладнанням.

У другому розділі було обрано та обґрунтовано програмно-технічні засоби системи, розроблено схеми взаємодії компонентів і виконано розрахунки параметрів телеметрії.

У третьому розділі було розроблено та перевірено прототип системи на базі Python, FastAPI, SQLite, React, TypeScript, MQTT, WebSocket і Docker Compose.

Бакалаврська кваліфікаційна робота передбачає розробку, магістерська - дослідження.

У результаті розроблена система може використовуватися як навчальний або демонстраційний стенд для пояснення принципів побудови IoT-систем віддаленого керування промисловим обладнанням.

ABSTARCT

Explanatory note: 119 p.; 7 tables; 19 figures; 3 appendices; 19 sources.

ALARM LOGIC, WEB INTERFACE, REMOTE CONTROL, DOCKER COMPOSE, IOT TECHNOLOGY, MQTT, PUMP STATION, SIMULATOR, TELEMETRY.

The object of the development is the process of remote monitoring and control of industrial equipment.

The subject of the development is a software system for remote control of a pump station using IoT technology.

The purpose of the work is to develop a local prototype of a remote monitoring and control system for a pump station.

In the first section, remote monitoring and control systems for industrial equipment were analyzed.

In the second section, the software and technical tools of the system were selected and justified, component interaction schemes were developed, and telemetry parameter calculations were performed.

In the third section, a system prototype based on Python, FastAPI, SQLite, React, TypeScript, MQTT, WebSocket and Docker Compose was developed and tested.

Sustainable Development Goal 9 “Industry, Innovation and Infrastructure”, in particular the direction of developing innovative and resilient infrastructure.

As a result, the developed system can be used as an educational or demonstration stand for explaining the principles of building IoT-based remote control systems for industrial equipment.

ЗМІСТ

Перелік скорочень 8
Вступ 9
1 Аналіз літератури та предметної області віддаленого керування обладнанням 12
1.1 Системи віддаленого моніторингу та керування обладнанням 12
1.2 Застосування IoT-технологій у промислових системах керування 14
1.3 Протоколи передавання даних і засоби побудови IoT-систем керування .. 16
2 Вибір та обґрунтування технічних засобів системи 21
2.1 Вибір програмно-технічних засобів розробки системи 21
2.2 Розробка функціональної схеми системи віддаленого керування 26
2.3 Розробка схеми взаємодії та розгортання системи 29
2.4 Розрахунок параметрів передавання, зберігання та обробки телеметрії 35
2.5 Розрахунок системи автоматичного керування електродвигуном насоса .. 40
3 Розробка програмного забезпечення системи 45
3.1 Розробка блок-схеми алгоритму роботи системи 45
3.2 Розробка програмних компонентів системи 52
3.3 Розробка моделі даних системи 57
3.4 Розробка обміну повідомленнями та оновлення даних у реальному часі .. 62
3.5 Практична перевірка роботи системи віддаленого керування 69
3.6 Охорона праці 79
Висновки 84
Перелік джерел посилання 87
Додаток А Графічні матеріали 90
Додаток Б Фрагменти програмного коду 99

ПЕРЕЛІК СКОРОЧЕНЬ

ПК – персональний комп'ютер;

ПЛК – програмований логічний контролер;

СКБД – система керування базами даних;

ЦСР 9 – Ціль сталого розвитку 9;

API – Application Programming Interface (програмний інтерфейс застосунку);

HTTP – HyperText Transfer Protocol (протокол передавання гіпертекстових даних у веб-застосунках);

IoT – Internet of Things (Інтернет речей);

MQTT – Message Queuing Telemetry Transport (легкий протокол обміну повідомленнями за моделлю публікації та підписки);

OPC UA – OPC Unified Architecture (промислова архітектура обміну даними між пристроями та програмними системами);

PID-регулятор – Proportional-Integral-Derivative controller (пропорційно-інтегрально-диференціальний регулятор);

QoS – Quality of Service (рівень якості доставлення MQTT-повідомлення);

REST API – Representational State Transfer API (інтерфейс взаємодії клієнтської та серверної частин через HTTP-запити);

SCADA – Supervisory Control And Data Acquisition (система диспетчерського керування та збирання даних).

ВСТУП

Промислове обладнання потребує постійного контролю, оскільки від його стану залежить безперервність технологічного процесу, безпека персоналу та своєчасне реагування на аварійні ситуації. У традиційних системах керування оператор часто отримує інформацію через локальні панелі, пульти або окремі програмні засоби, що обмежує можливості віддаленого нагляду. Через це зростає потреба в рішеннях, які дозволяють передавати телеметрію, надсилати команди керування та бачити поточний стан обладнання через мережевий інтерфейс.

Одним із напрямів розвитку таких рішень є використання IoT-технологій. Вони дозволяють пов’язати обладнання, серверну частину, базу даних і веб-інтерфейс користувача в одну систему обміну даними. Для промислових задач важливо не лише отримувати показники від обладнання, а й перевіряти команди, обмежувати доступ за ролями, зберігати історію роботи та своєчасно реагувати на критичні відхилення. Тому віддалене керування доцільно розглядати як поєднання моніторингу, комунікаційного протоколу, серверної логіки, аварійного захисту та інтерфейсу оператора.

Актуальність теми зумовлена потребою у створенні зрозумілих і доступних засобів віддаленого моніторингу та керування промисловим обладнанням. У роботі таким обладнанням виступає насосна станція, стан якої можна оцінювати за тиском, температурою, вібрацією, швидкістю двигуна, струмом двигуна та витратою рідини. Реальне обладнання, фізичні датчики та програмований логічний контролер у роботі не використовуються. Їх роль виконує програмний симулятор, який генерує телеметрію насосної станції та реагує на команди керування.

Об’єктом дослідження є процес інформаційної взаємодії під час віддаленого моніторингу та керування промисловим обладнанням. Предметом дослідження є програмні засоби та архітектурні рішення побудови локального прототипу системи

віддаленого керування насосною станцією з використанням IoT-технології. Такий підхід дозволяє розглядати не лише окремий веб-інтерфейс, а повний шлях даних від симулятора обладнання до серверної частини, бази даних і робочого екрана оператора.

Розроблюваний прототип показує шлях даних від програмного симулятора до веб-інтерфейсу оператора. Симулятор передає телеметрію через MQTT-брокер Eclipse Mosquitto, а серверна частина на основі FastAPI зберігає записи в SQLite, перевіряє команди та обробляє аварійні стани. Поточні значення надходять до веб-застосунку на React і TypeScript через WebSocket, а REST API використовується для авторизації, історії, аварій, налаштувань і команд.

Метою роботи є підвищення ефективності керування насосною станцією за рахунок розробки локального прототипу системи віддаленого моніторингу її параметрів

Для досягнення поставленої мети необхідно вирішити такі завдання:

– проаналізувати сучасний стан систем віддаленого моніторингу та керування промисловим обладнанням;

– розглянути особливості застосування IoT-технологій у промислових системах керування;

– обґрунтувати вибір програмно-технічних засобів для розробки системи віддаленого керування насосною станцією;

– розробити функціональну схему системи та схему інформаційної взаємодії її компонентів;

– виконати орієнтовний розрахунок параметрів передавання, зберігання та обробки телеметрії;

– описати модель керування електродвигуном насоса як основним динамічним елементом насосної станції;

– реалізувати програмні компоненти системи, зокрема симулятор, серверну частину, базу даних, механізми MQTT-обміну, WebSocket-оновлення та веб-інтерфейс користувача;

– реалізувати авторизацію, рольовий доступ, обробку команд керування, аварійну логіку та збереження історії роботи системи;

– провести практичну перевірку роботи розробленого прототипу в локальному середовищі Docker Compose.

Результати роботи можуть використовуватися як навчальний або демонстраційний стенд для пояснення принципів віддаленого керування промисловим обладнанням. Запропонована структура може бути основою для подальшого розвитку систем моніторингу насосних агрегатів, компресорного, вентиляційного або іншого технологічного обладнання. Для реального впровадження такої системи потрібно буде додати фізичні датчики, контролер, промислові вимоги до безпеки, надійності та захисту мережевої взаємодії.

Кваліфікаційна робота пов’язана з попередньо виконаними практичними напрацюваннями зі створення прототипу віддаленого керування насосною станцією. У цій роботі ці напрацювання доповнено розрахунковими підрозділами, описом архітектури, моделлю даних, аварійною логікою та практичною перевіркою роботи системи. Це дозволяє розглядати розробку не як окремий веб-застосунок, а як узгоджену систему з симулятором обладнання, серверною частиною, базою даних, каналом реального часу та інтерфейсом оператора.

Кваліфікаційну роботу виконано згідно з ДСТУ 3008:2015 [1], а також з методичними вказівками з підготовки й оформлення кваліфікаційної роботи здобувачами першого бакалаврського рівня вищої освіти спеціальності 151 Автоматизація та комп’ютерно-інтегровані технології освітньої програми «Системна інженерія» [2].

1 АНАЛІЗ ЛІТЕРАТУРИ ТА ПРЕДМЕТНОЇ ОБЛАСТІ ВІДДАЛЕНОГО КЕРУВАННЯ ОБЛАДНАННЯМ

1.1 Системи віддаленого моніторингу та керування обладнанням

Системи керування промисловим обладнанням розвивалися від локальних щитів керування до розподілених автоматизованих систем, у яких оператор отримує інформацію про стан об’єкта через програмний інтерфейс. У таких системах основну роль відіграють засоби вимірювання, виконавчі механізми, контролери, канали передавання даних і програмні засоби відображення стану об’єкта. У навчальному посібнику з основ автоматизації технологічних процесів автоматизація контролю та керування розглядається як частина загальної автоматизації технологічних об’єктів, де інформація від об’єкта використовується для прийняття керувальних рішень [3].

Для промислового обладнання важливо не тільки отримувати значення окремих параметрів, а й правильно інтерпретувати їх у зв’язку зі станом об’єкта. Наприклад, для насосної станції значення тиску має різний зміст залежно від того, зупинений насос чи перебуває в роботі. Те саме стосується температури, вібрації, струму двигуна та витрати рідини. У літературі з автоматизації типових технологічних процесів окрема увага приділяється технологічним об’єктам керування та схемам автоматизації, оскільки саме структура вимірювань і каналів керування визначає, які рішення може приймати оператор або автоматичний регулятор [4].

У виробничих умовах віддалене керування не означає пряме підключення оператора до обладнання без проміжного рівня перевірок. Зазвичай між інтерфейсом оператора та виконавчим механізмом є контролер, сервер або промисловий шлюз, який перевіряє допустимість команди, фіксує подію та передає

її нижче до обладнання. Такий підхід зменшує ризик некоректних дій, дає змогу вести журнал команд і відокремлює людино-машинний інтерфейс від нижнього рівня керування. У навчальному посібнику з автоматизації виробництва в машинобудуванні показано, що автоматизація виробничих процесів пов’язана не тільки з механічним виконанням операцій, а й з організацією контролю, керування та взаємодії окремих технічних засобів [5]. Узагальнений принцип такого керування показано на рисунку 1.1. На схемі видно, що вимірюване значення повертається до регулятора через зворотний зв’язок, а керувальний вплив передається на об’єкт через виконавчу частину.

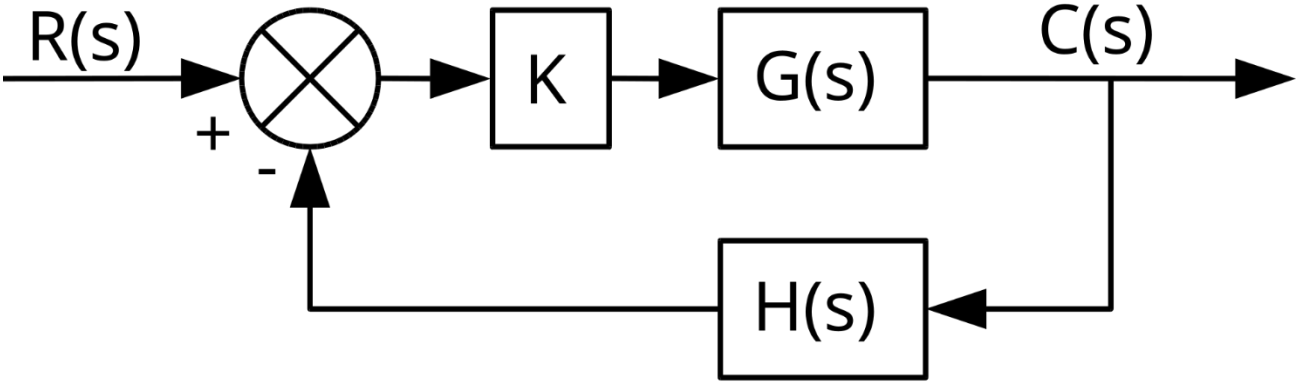


Рисунок 1.1 – Узагальнений контур керування зі зворотним зв’язком

Окремий напрям розвитку таких систем пов’язаний з переходом від ізольованих локальних систем до рішень, які використовують мережеві технології, віддалений доступ і накопичення даних. У стратегії розвитку Індустрії 4.0 в Україні підкреслено значення промислових високотехнологічних сегментів, інтеграції ІТ-сектору з промисловістю та цифровізації виробничих процесів [6]. Для теми цієї кваліфікаційної роботи це означає, що система віддаленого керування повинна розглядатися не як окремий веб-інтерфейс, а як частина інформаційного контуру промислового об’єкта.

Предметною областю цієї роботи є віддалений моніторинг і керування промисловим обладнанням на прикладі насосної станції. У фізичній реалізації така

станція могла б містити датчик тиску, датчик температури, датчик вібрації, датчик струму двигуна, витратомір, електродвигун насоса, пускову апаратуру та контролер або мікроконтролер з мережевим модулем. Контролер збирав би сигнали з датчиків, формував поточний стан обладнання та приймав команди від верхнього рівня. У цій роботі конкретна модель мікроконтролера та конкретні моделі датчиків не обираються, оскільки апаратну реалізацію нижнього рівня свідомо винесено за межі розроблення. У розробленому прототипі роль нижнього рівня виконує програмний симулятор, який генерує телеметрію насосної станції та реагує на команди через MQTT.

Такий підхід не замінює реальну апаратну частину, але дає змогу перевірити повний інформаційний ланцюг без підключення фізичного стенда. Для дипломного прототипу це особливо важливо, оскільки можна показати не тільки інтерфейс користувача, а й передавання телеметрії, збереження історії, виявлення аварій, блокування небезпечних команд і оновлення стану в реальному часі. Практичний зміст роботи полягає в тому, що симулятор імітує поведінку обладнання, а всі інші рівні системи працюють так, як вони могли б працювати з реальним контролером.

1.2 Застосування IoT-технологій у промислових системах керування

IoT-технології дають змогу об’єднувати датчики, виконавчі пристрої, контролери, сервери та користувацькі інтерфейси в єдину систему обміну даними. На відміну від звичайного веб-застосунку, IoT-система має нижній рівень пристроїв, які постійно формують дані про стан об’єкта. У навчальному посібнику з технологій Інтернету речей розглянуто IoT як напрям, у якому поєднуються пристрої збирання даних, канали зв’язку, обробка інформації та програмні засоби взаємодії з користувачем [7]. На рисунку 1.2 наведено приклад шарової структури промислової IoT-архітектури. Для цієї роботи він корисний тим, що відокремлює

рівень обладнання і датчиків від мережевого обміну, обробки даних та прикладного інтерфейсу оператора.

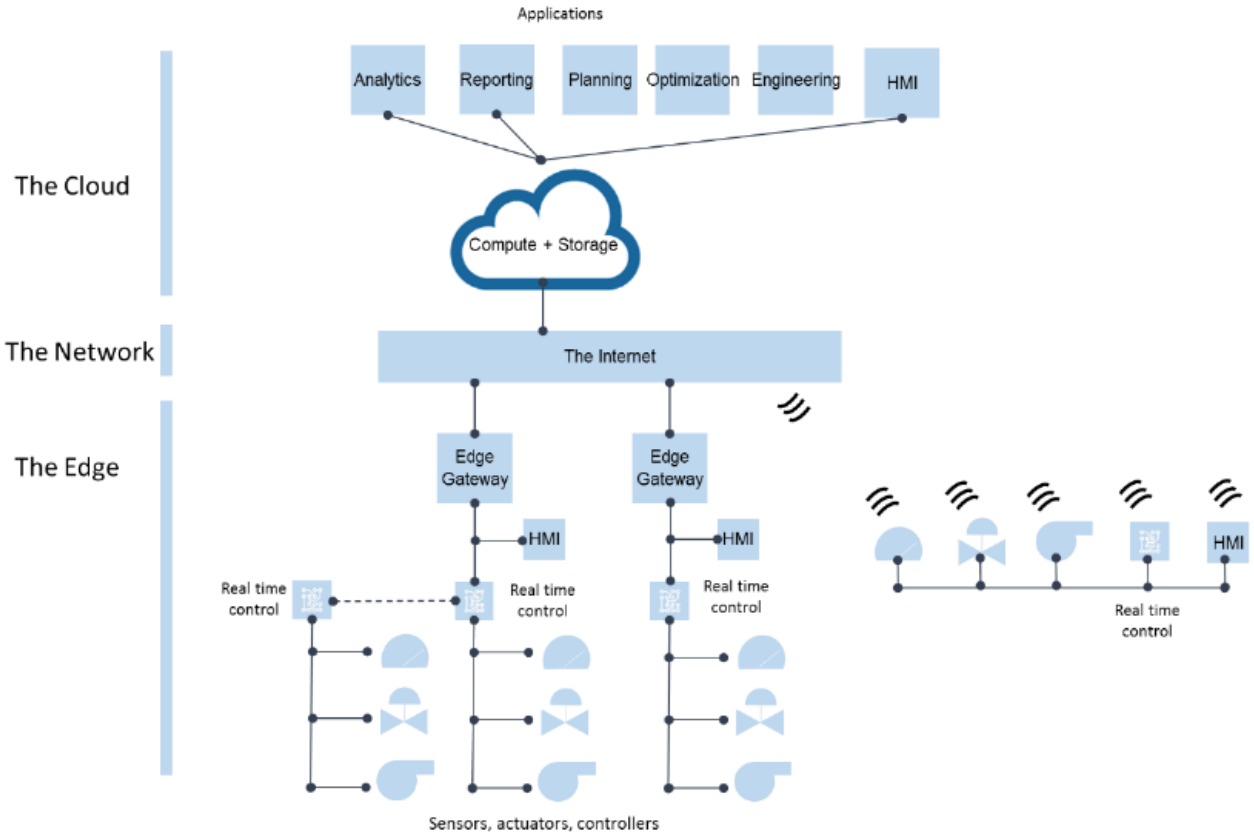


Рисунок 1.2 – Узагальнена структура промислової IoT-архітектури

У промисловому застосуванні IoT має додаткові обмеження. Дані від обладнання повинні надходити регулярно, а втрата зв’язку повинна бути помітною для оператора. Команди керування мають проходити перевірку, тому що помилка в команді може призвести не лише до некоректного відображення стану, а й до небезпечної роботи обладнання. У посібнику “Вступ до Інтернету речей” розглянуто структуру систем IoT та промислового Інтернету речей, їхні складові, сфери застосування, датчики і канали зв’язку [8].

Для насосної станції IoT-підхід проявляється в тому, що телеметрія передається від нижнього рівня до серверної частини без ручного введення

оператором. Значення температури, тиску, вібрації, швидкості двигуна, струму двигуна та витрати рідини формують поточну картину роботи обладнання. На основі цих даних серверна частина може оновлювати інтерфейс, зберігати історію, перевіряти порогові значення та створювати аварійні повідомлення.

Віддалене керування в такій системі має зворотний напрям передавання даних. Оператор або адміністратор взаємодіє з веб-інтерфейсом, але команда не переходить одразу до обладнання. Спочатку вона потрапляє на сервер, де перевіряється роль користувача, стан зв’язку, поточний режим роботи і допустимість параметрів. Тільки після цього команда передається на нижній рівень через протокол обміну повідомленнями. Для цієї роботи такий принцип важливий, оскільки він відокремлює логіку безпеки і перевірки команд від інтерфейсу користувача.

Промислові IoT-системи також потребують збереження історичних даних. Поточне значення тиску або температури показує стан об’єкта лише в один момент часу, але для аналізу роботи обладнання потрібна послідовність вимірювань. Історія телеметрії дає змогу перевірити, як змінювалися параметри після запуску насоса, перемикання режиму, зміни цільового тиску або виникнення аварійної ситуації. У розробленому прототипі це реалізується через збереження телеметрії, аварій, команд і налаштувань у SQLite.

IoT-рішення не скасовує потребу в автоматичній логіці на рівні обладнання або серверної частини. Навпаки, воно робить таку логіку більш помітною, тому що оператор бачить не лише результат дії, а й проміжні стани. Для насосної станції це означає наявність режимів MANUAL, AUTO та EMERGENCY, станів насоса STOPPED і RUNNING, а також стану зв’язку ONLINE або OFFLINE. Таке розділення станів потрібне для коректної обробки аварій і для того, щоб аварійний режим не змішувався зі звичайною зупинкою насоса.

1.3 Протоколи передавання даних і засоби побудови IoT-систем керування

Під час побудови IoT-системи керування потрібно розділяти протоколи нижнього рівня обладнання, протоколи передавання телеметрії та протоколи взаємодії з веб-інтерфейсом. Один протокол рідко закриває всі задачі одночасно. Для промислового об’єкта потрібна передача регулярних вимірювань, передача команд, отримання історії з бази даних, авторизація користувачів і оновлення інтерфейсу без ручного перезавантаження сторінки. Через це доцільно використовувати кілька каналів з чітким розподілом ролей.

Одним з поширених протоколів для IoT є MQTT. Він побудований за моделлю publish/subscribe, де клієнти не передають повідомлення напряму один одному, а працюють через брокер. Офіційна специфікація MQTT 5.0 описує його як легкий клієнт-серверний протокол обміну повідомленнями, придатний для M2M та IoT-сценаріїв з обмеженими ресурсами і вимогами до невеликого мережевого навантаження [9]. Для насосної станції така модель підходить, тому що симулятор може регулярно публікувати телеметрію, а серверна частина може підписуватися на відповідну тему та незалежно обробляти отримані повідомлення. Модель publish/subscribe показано на рисунку 1.3. Вона наочно пояснює, чому симулятор і backend не повинні мати прямого з’єднання один з одним: обмін між ними виконується через MQTT-брокер і наперед визначені теми.

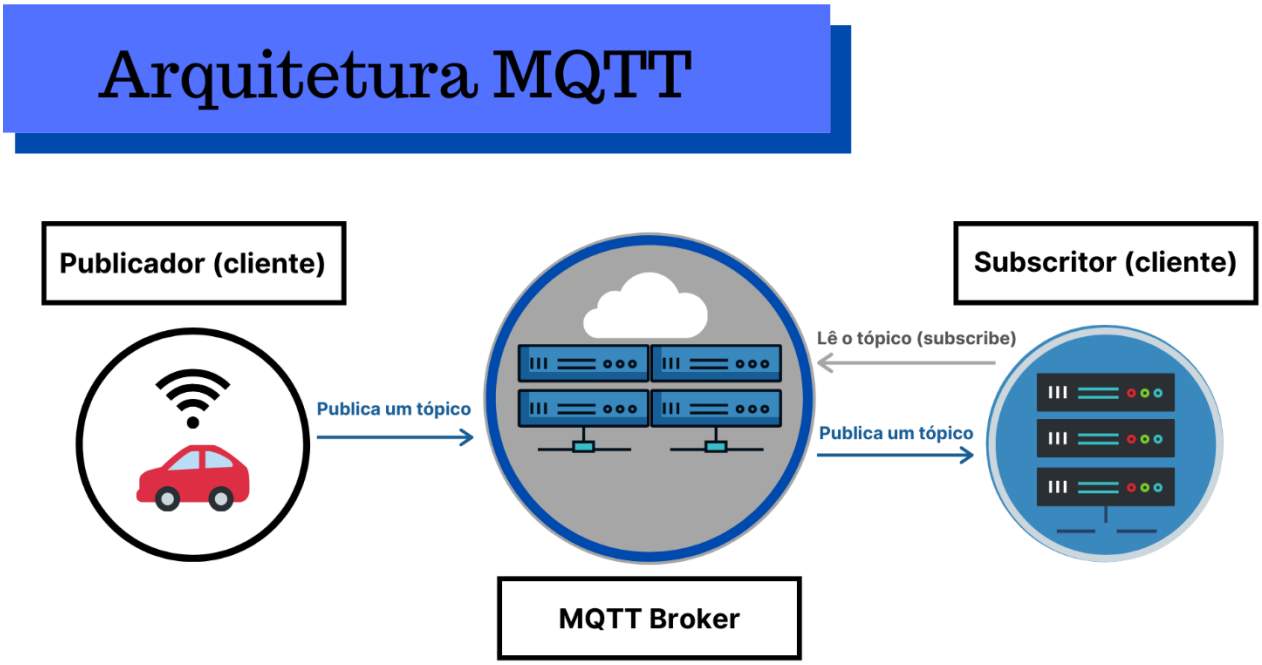


Рисунок 1.3 – Приклад архітектури MQTT з видавцем, брокером і підписником

MQTT також зручний для передавання команд на нижній рівень. У цій роботі для телеметрії доцільно використовувати QoS 0, оскільки окрема втрачена телеметрична точка не є критичною, а наступне повідомлення надходить через короткий інтервал. Для команд керування логічним є QoS 1, тому що команда запуску, зупинки або аварійної зупинки повинна бути доставлена хоча б один раз. Через можливість повторної доставки команда повинна мати ідентифікатор, а нижній рівень має відкидати дублікати за commandId.

OPC UA також застосовується у промислових системах, але його призначення ширше, ніж проста передача телеметрії. За описом OPC Foundation, OPC UA є платформонезалежною сервісно-орієнтованою архітектурою, яка об’єднує функції попередніх специфікацій OPC в розширювану модель [10]. Цей протокол доречний у випадках, коли потрібно формалізувати інформаційну модель обладнання, інтегрувати різні промислові пристрої та забезпечити сумісність між виробниками.

Для дипломного прототипу з однією насосною станцією така повнота була б надмірною, тому використання MQTT виглядає простішим і достатнім.

Modbus належить до класичних промислових протоколів і часто використовується для обміну між контролерами, перетворювачами, лічильниками та іншими пристроями. У специфікації Modbus Application Protocol зазначено, що це протокол прикладного рівня для клієнт-серверної взаємодії між пристроями, підключеними через різні типи шин або мереж [11]. Його можна розглядати як варіант для реальної апаратної частини насосної станції, наприклад для зчитування регістрів контролера або перетворювача частоти. У цьому прототипі Modbus не використовується, тому що нижній рівень реалізовано програмним симулятором, а обмін між симулятором і сервером побудовано через MQTT.

Для взаємодії веб-інтерфейсу з серверною частиною MQTT напряму не використовується. Це пов’язано з тим, що браузерний інтерфейс не повинен підключатися до брокера і самостійно обробляти промислові повідомлення. У розробленій архітектурі веб-інтерфейс працює тільки з backend через REST API та WebSocket. REST API зручний для запитів, які мають чітку дію і відповідь, наприклад авторизація, отримання історії, перегляд аварій, зміна налаштувань або надсилання команди.

WebSocket потрібний для тих даних, які мають надходити в інтерфейс без постійного опитування сервера. Документація MDN описує WebSocket API як механізм відкриття двосторонньої інтерактивної сесії між браузером і сервером, що дає змогу отримувати відповіді без регулярного polling [12]. У цій роботі такий канал використовується для оновлення поточного стану насосної станції, нової телеметрії, аварій, результатів команд і стану зв’язку. Це дає оператору відчуття роботи з живим об’єктом, а не зі статичною сторінкою.

Окремо потрібно розглядати засоби розгортання, оскільки IoT-система складається з кількох процесів. Для прототипу потрібні MQTT-брокер, backend, simulator, frontend і файл бази даних. Docker Compose підходить для такого випадку,

оскільки дає змогу описати багато контейнерів, мережі та томи в одному конфігураційному файлі, а потім запускати їх однією командою [13]. Для дипломної роботи це важливо не як окрема технологічна перевага, а як спосіб зробити демонстрацію відтворюваною на іншому комп’ютері.

За результатами аналізу предметної області можна виділити кілька рішень, які мають бути збережені в подальшому проєктуванні. Нижній рівень повинен генерувати телеметрію та приймати команди через окремий канал обміну повідомленнями. Серверна частина має бути центральним місцем перевірки команд, збереження даних, виявлення аварій і передавання оновлень до інтерфейсу. Веб-інтерфейс повинен показувати стан об’єкта в реальному часі, але не повинен напряму підключатися до MQTT-брокера або бази даних. Саме така логіка покладена в основу системи віддаленого керування насосною станцією, що розробляється в цій кваліфікаційній роботі.

2 ВИБІР ТА ОБҐРУНТУВАННЯ ТЕХНІЧНИХ ЗАСОБІВ СИСТЕМИ

2.1 Вибір програмно-технічних засобів розробки системи

Після аналізу предметної області для реалізації системи віддаленого керування промисловим обладнанням було обрано архітектуру локального IoT-прототипу. У ній технологічний об’єкт представлено насосною станцією, а апаратний рівень замінено програмним симулятором. Такий підхід дає змогу перевірити повний шлях передавання даних від умовного обладнання до веб-інтерфейсу оператора без придбання датчиків, мікроконтролера, контролера ПЛК або реального насосного стенда. Для теми роботи важливим є не сам факт використання конкретного датчика, а побудова каналу віддаленого моніторингу, передавання команд, збереження телеметрії та оновлення інтерфейсу в режимі, близькому до реального часу.

Обрана структура відповідає типовій логіці IoT-систем, у яких є пристрій або його програмна модель, мережевий канал передавання даних, серверна частина, база даних і клієнтський інтерфейс. У роботі ці рівні реалізовано через Python-симулятор насосної станції, брокер повідомлень Eclipse Mosquitto, backend на Python з FastAPI, базу даних SQLite та frontend на React з TypeScript. Для передавання телеметрії та команд між симулятором і backend використано MQTT, а для взаємодії веб-інтерфейсу з backend використано REST API та WebSocket.

Застосування програмного симулятора замість реального обладнання не змінює логіку IoT-взаємодії. Симулятор виконує роль пристрою нижнього рівня, який формує телеметрію, має власний стан, приймає команди та реагує на них. У реальній системі на його місці міг би бути мікроконтролер або ПЛК, до якого підключаються датчики температури, тиску, вібрації, струму двигуна та витрати рідини. У межах цієї кваліфікаційної роботи ці вимірювальні канали описано через

програмні параметри temperature, pressure, vibration, motorCurrent, flowRate та motorSpeed, які передаються до backend у складі MQTT-повідомлень.

Узагальнений вибір програмно-технічних засобів наведено у таблиці 2.1. У таблиці вказано не тільки використані технології, а й причину їх застосування саме в цій роботі.

Таблиця 2.1 – Вибір програмно-технічних засобів системи віддаленого керування

Елемент системи	Обраний засіб	Призначення	Обґрунтування вибору
Симулятор обладнання	Python	Формування телеметрії насосної станції, обробка команд, моделювання стану насоса та електродвигуна	Python дає змогу швидко описати логіку зміни параметрів, аварійні режими та інерційну зміну швидкості двигуна без додаткового апаратного стенда
Серверна частина	Python, FastAPI	Приймання телеметрії, перевірка команд, обробка аварій, робота з REST API та WebSocket	FastAPI підходить для побудови асинхронного веб-сервера з перевіркою даних, авторизацією, REST-запитами та WebSocket-каналом
База даних	SQLite	Збереження користувачів, телеметрії, аварій, команд, налаштувань і сесій оновлення токенів	SQLite не потребує окремого серверного процесу, достатня для одного демонстраційного об’єкта та спрощує локальний запуск через Docker Compose
Брокер повідомлень	Eclipse Mosquitto	Передавання телеметрії, статусів і команд між симулятором та backend	Mosquitto є легким MQTT-брокером, який підходить для локального IoT-прототипу та не потребує складної конфігурації

Продовження таблиці 2.1

Елемент системи	Обраний засіб	Призначення	Обґрунтування вибору
Клієнтська частина	React, TypeScript	Веб-інтерфейс оператора й адміністратора, відображення телеметрії, графіків, аварій і елементів керування	React дає змогу побудувати інтерактивний інтерфейс, а TypeScript зменшує ризик помилок під час роботи з типізованими API-даними
Канал оновлення інтерфейсу	WebSocket	Передавання поточного стану станції, телеметрії, аварій і результатів команд від backend до frontend	WebSocket усуває потребу в постійному опитуванні backend і дає змогу швидко оновлювати дані на панелі оператора
Розгортання	Docker Compose	Одночасний запуск frontend, backend, simulator і mosquitto	Docker Compose забезпечує повторюваний локальний запуск усіх компонентів однією командою та спрощує демонстрацію роботи системи

Для обміну між симулятором і backend обрано MQTT, оскільки цей протокол добре підходить для обміну короткими повідомленнями між пристроями та серверними компонентами IoT-систем. У стандарті MQTT описано модель публікації та підписки, у якій клієнти не передають повідомлення напряму один одному, а працюють через брокер. Для цієї роботи така модель зручна, оскільки симулятор не повинен знати, як влаштований backend, а backend не повинен напряму керувати внутрішньою логікою симулятора. Обидва компоненти взаємодіють через наперед визначені теми MQTT.

OPC UA і Modbus також можуть застосовуватися у промислових системах автоматизації, але для цього прототипу вони не були основними засобами взаємодії. OPC UA орієнтований на розширену промислову інтеграцію, інформаційні моделі, сумісність різних виробників і складніші сценарії обміну даними. Modbus є

поширеним промисловим протоколом нижчого рівня, який часто застосовується для взаємодії з контролерами та пристроями введення-виведення. У цій роботі немає реального ПЛК, промислової мережі та набору фізичних регістрів пристрою, тому використання MQTT є простішим і більш придатним для демонстрації IoT-логіки.

Серверна частина реалізує центральну логіку системи. Вона приймає MQTT-повідомлення від симулятора, перевіряє структуру телеметрії, зберігає записи в SQLite, аналізує пороги аварій і передає оновлення до frontend. Команди від оператора не надсилаються безпосередньо в симулятор. Спочатку вони потрапляють у backend через REST API, де проходять перевірку користувача, ролі, типу команди, вмісту payload і поточного стану станції. Лише після цього backend публікує команду в MQTT-тему для симулятора.

SQLite обрано як вбудовану базу даних, тому що прототип працює з одним об’єктом керування і не потребує окремого сервера бази даних. Для збереження телеметрії з частотою 1 запис за секунду, журналу команд, аварій, користувачів і налаштувань можливостей SQLite достатньо. Використання PostgreSQL або іншої серверної СКБД збільшило б кількість контейнерів, ускладнило розгортання та не дало б суттєвої переваги для локальної демонстрації. Якщо систему розширювати до кількох станцій або тривалого промислового зберігання даних, перехід на окрему серверну СКБД може бути обґрунтованим, але для поточної роботи це виходить за межі потрібного обсягу.

Frontend розробляється як веб-інтерфейс оператора й адміністратора. Він не підключається до MQTT-брокера і не має прямого доступу до бази даних. Така межа потрібна для того, щоб усі перевірки команд, прав доступу та стану обладнання виконувалися на backend. Frontend отримує історичні дані через REST API, а поточні оновлення отримує через WebSocket. WebSocket API підтримує відкриття інтерактивного каналу між браузером і сервером без постійного

опитування сервера, тому його використано для відображення живої телеметрії, аварій і результатів обробки команд.

Docker Compose обрано для локального розгортання всіх компонентів системи. Docker Compose використано для опису та запуску застосунку, який складається з кількох контейнерів. У цій роботі окремими контейнерами запускаються frontend, backend, simulator і mosquitto. База SQLite не виділяється в окремий контейнер, оскільки вона зберігається як файл усередині backend-контейнера на підключеному томі Docker. Це спрощує структуру запуску і водночас залишає дані після перезапуску контейнерів.

Повноцінна SCADA-система не застосовується, оскільки мета роботи полягає у створенні власного прототипу віддаленого керування, а не в налаштуванні готової промислової платформи. SCADA-рішення мають значно ширший набір можливостей, включно з диспетчеризацією багатьох об’єктів, архівацією великих обсягів технологічних даних, промисловими драйверами, правами доступу підприємства та засобами візуалізації для операторських станцій. Для демонстраційного прототипу з одним об’єктом керування такі можливості були б надмірними. У роботі реалізовано тільки ті функції, які потрібні для підтвердження теми, а саме віддалений моніторинг, передавання команд, автоматичну реакцію на аварії, журналювання та візуалізацію поточного стану.

Окремо потрібно пояснити відмову від повного PID-регулятора. У системі є автоматичний режим підтримання тиску, але він реалізований як спрощений алгоритм поступового коригування ефективного керувального впливу в симуляторі. Такий підхід достатній для демонстрації автоматизованої поведінки насосної станції, плавної зміни швидкості двигуна та реакції на зміну цільового тиску. Розрахунок повного PID-регулятора, його налаштування, перевірка перерегулювання та дослідження частотних характеристик потребували б іншого обсягу роботи і точнішої фізичної моделі насосної станції. У цій роботі

розрахункова частина з теорії автоматичного керування зосереджується на електродвигуні насоса як основному динамічному елементі.

2.2 Розробка функціональної схеми системи віддаленого керування

Функціональна схема потрібна для відображення загальної взаємодії користувача, веб-застосунку, серверної частини, MQTT-брокера, симулятора обладнання та бази даних. Вона не деталізує внутрішній алгоритм перевірки аварій і не замінює блок-схему роботи системи, яка буде наведена в розділі 3. У цьому підрозділі розглядається, які функціональні блоки входять до системи та які дані проходять між ними під час моніторингу й керування.

На рисунку 2.1 наведено функціональну схему системи віддаленого керування насосною станцією. Вона показує два основні напрями взаємодії. Перший напрям пов’язаний із передаванням телеметрії від симулятора до інтерфейсу оператора. Другий напрям пов’язаний із передаванням команд від оператора або адміністратора до симулятора через backend і MQTT-брокер.

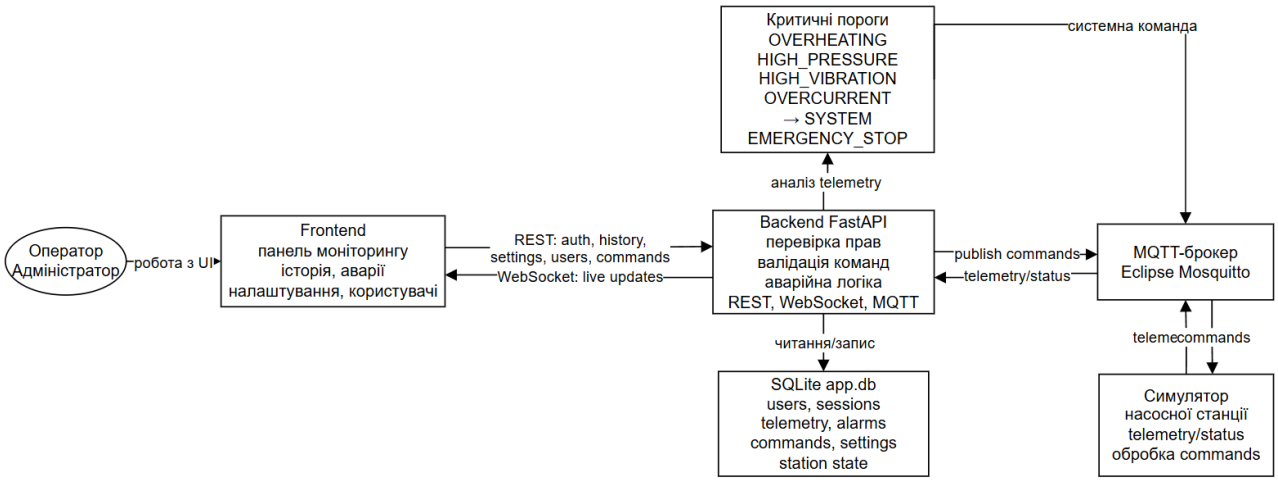


Рисунок 2.1 – Функціональна схема системи віддаленого керування насосною станцією

Користувачами системи є оператор і адміністратор. Оператор працює з панеллю моніторингу, переглядає поточні значення температури, тиску, вібрації, швидкості двигуна, струму двигуна та витрати рідини. Також оператор може запускати й зупиняти насос, перемикати режим MANUAL або AUTO, задавати цільовий тиск, виконувати аварійну зупинку та скидати аварійний режим після стабілізації параметрів. Адміністратор має ті самі можливості керування, але додатково може змінювати налаштування порогів і керувати користувачами.

Frontend є єдиним компонентом, з яким безпосередньо працює користувач. Він відображає сторінку входу, панель поточного стану, історію телеметрії, перелік аварій, сторінку налаштувань, журнал команд і сторінку керування користувачами. Frontend не має прямого каналу до MQTT-брокера та симулятора. Усі запити проходять через backend, тому саме серверна частина контролює права доступу, допустимість команд і коректність даних.

Backend виконує роль центрального вузла системи. Він приймає HTTP-запити від frontend, підтримує WebSocket-з’єднання для оновлення інтерфейсу, підписується на MQTT-теми телеметрії та статусу, а також публікує команди в MQTT-тему керування. Крім того, backend зберігає дані в SQLite, веде журнал команд, створює записи аварій і підтримує останній відомий стан насосної станції. Через це backend є єдиною точкою, де об’єднуються користувацькі дії, телеметрія, правила безпеки та збереження даних.

MQTT-брокер Eclipse Mosquitto розміщується між backend і симулятором. Він не аналізує значення тиску, температури або команд, а тільки доставляє повідомлення за темами. Такий поділ відповідає моделі публікації та підписки. Симулятор публікує телеметрію і статус, backend підписується на ці повідомлення. Коли користувач надсилає команду, backend публікує її в тему команд, а симулятор отримує її як підписник.

Симулятор насосної станції виконує роль умовного обладнання. Він зберігає поточний режим роботи, стан насоса, цільовий тиск, швидкість двигуна та значення

технологічних параметрів. Після отримання команди симулятор змінює внутрішній стан, а наступні повідомлення телеметрії відображають результат цієї зміни. Наприклад, після команди START_PUMP стан pumpState переходить у RUNNING, швидкість двигуна починає зростати поступово, а тиск і витрата рідини змінюються відповідно до моделі симуляції.

База даних SQLite входить до функціональної схеми як сховище довготривалих даних. У ній зберігаються користувачі, хеші паролів, refresh-сесії, записи телеметрії, аварії, команди та налаштування станції. Симулятор не має доступу до цієї бази. Frontend також не звертається до бази напряму. Усі операції читання й запису виконує backend, що спрощує контроль цілісності даних.

Під час передавання телеметрії симулятор формує JSON-повідомлення з часовою міткою, ідентифікатором stationId, вимірюваними параметрами та поточними станами mode, pumpState і targetPressure. Повідомлення надходить у MQTT-брокер, після чого його приймає backend. Далі backend перевіряє структуру повідомлення, записує телеметрію до SQLite, оновлює останній відомий стан станції, перевіряє аварійні пороги та передає оновлення до frontend через WebSocket. На панелі оператора це відображається як зміна карток параметрів, графіків, індикатора режиму, стану насоса та аварійних повідомлень.

Передавання команд відбувається у зворотному напрямі. Коли оператор натискає кнопку запуску, зупинки або зміни режиму, frontend надсилає REST-запит до backend. Backend перевіряє access token користувача, роль, тип команди, payload і поточний стан станції. Якщо команда дозволена, backend створює commandId, записує команду в журнал і публікує MQTT-повідомлення для симулятора. Якщо команда не дозволена, наприклад через режим EMERGENCY або відсутність зв’язку із симулятором, backend не передає її в MQTT і повертає причину відхилення.

REST-відповідь backend не означає, що команда вже фізично застосована симулятором. Вона показує тільки результат обробки команди на сервері, тобто

ACCEPTED, REJECTED або FAILED_TO_PUBLISH. Фактичне застосування підтверджується наступними status або telemetry повідомленнями від симулятора. Саме після них frontend оновлює відображений стан насосної станції. Це рішення запобігає ситуації, коли інтерфейс показує запуск насоса тільки на підставі прийнятого REST-запиту, хоча симулятор ще не встиг обробити команду.

Функціональна схема також враховує аварійні дії backend. Якщо під час аналізу телеметрії температура, тиск, вібрація або струм двигуна перевищують критичні пороги, backend створює аварію і формує системну команду EMERGENCY_STOP. Така команда має походження SYSTEM, оскільки її створює не користувач, а логіка захисту. Після обробки команди симулятор переходить у режим EMERGENCY, зупиняє насос і поступово зменшує швидкість двигуна до нуля.

2.3 Розробка схеми взаємодії та розгортання системи

Схема інформаційної взаємодії уточнює, як саме компоненти системи з’єднуються під час запуску через Docker Compose. На відміну від функціональної схеми, у цьому підрозділі основна увага приділяється контейнерам, мережевим каналам, портам, MQTT-темам і розміщенню бази даних. Такий опис потрібний для того, щоб роботу системи можна було відтворити на іншому ПК без ручного налаштування кожного компонента окремо.

На рисунку 2.2 наведено схему інформаційної взаємодії та розгортання компонентів. У ній потрібно показати чотири контейнери Docker Compose, а саме frontend, backend, mosquitto і simulator. Окремо на схемі показується файл бази даних SQLite app.db, який зберігається не як самостійний контейнер, а як файл у томі, підключеному до backend.

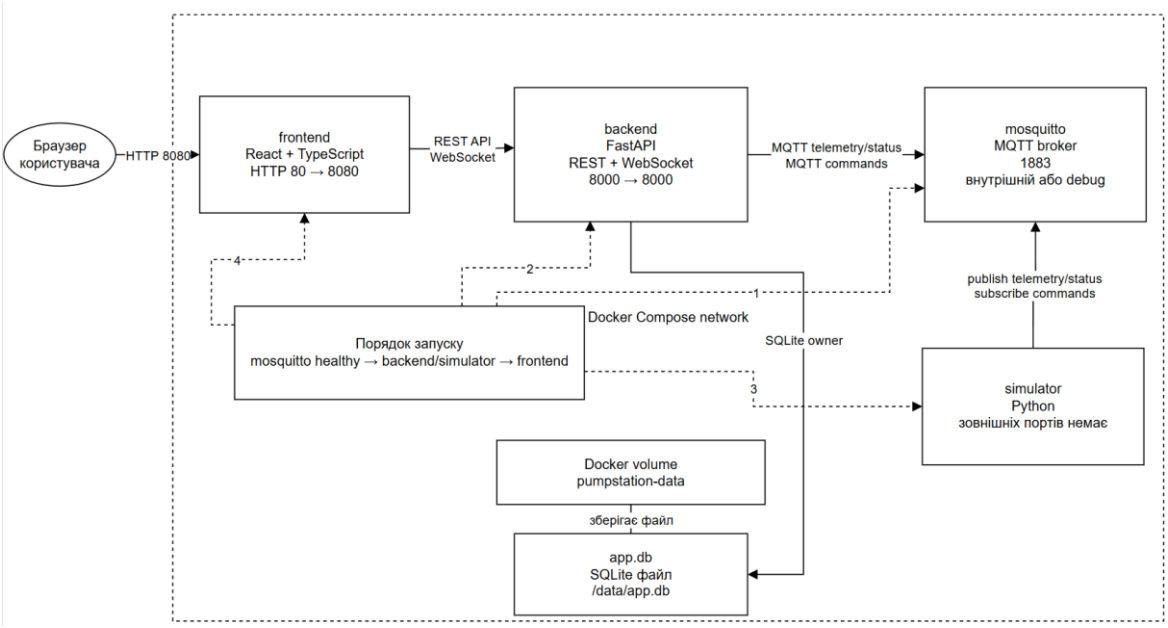


Рисунок 2.2 – Схема інформаційної взаємодії та розгортання компонентів системи

Усі контейнери працюють в одній мережі Docker Compose. Frontend відкривається користувачем у браузері та має зовнішній HTTP-порт. Backend також має зовнішній порт для REST API і WebSocket, оскільки саме до нього звертається frontend. Mosquitto може мати відкритий порт 1883 для налагодження через зовнішній MQTT-клієнт, але для звичайної демонстрації достатньо внутрішньої взаємодії між контейнерами. Simulator не відкриває зовнішніх портів, бо він не приймає HTTP-запити та не має власного веб-інтерфейсу. Загальний принцип поділу застосунку на сервіси, мережі та томи в Docker Compose показано на рисунку 2.3. У розробленій системі цей підхід конкретизується через сервіси frontend, backend, simulator і mosquitto, а також через том для збереження файла SQLite.

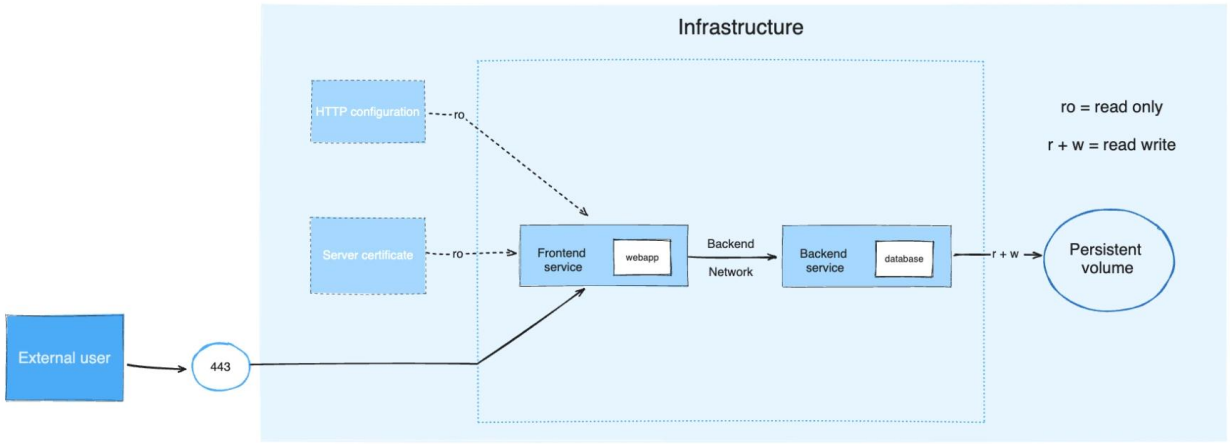


Рисунок 2.3 – Приклад поділу застосунку Docker Compose на сервіси, мережі та томи

Інформаційна взаємодія між frontend і backend поділена на два канали. REST API використовується для входу в систему, оновлення токена, виходу, отримання історії телеметрії, перегляду аварій, зміни налаштувань, керування користувачами та надсилання команд. WebSocket використовується для поточного стану станції, нової телеметрії, аварій, результатів команд і зміни стану з’єднання. Такий поділ зручний, оскільки звичайні запити залишаються у форматі REST, а дані, які мають оновлюватися без ручного перезавантаження сторінки, передаються через постійний канал.

Для взаємодії backend із симулятором використано три MQTT-теми. Їх наведено у таблиці 2.2. Телеметрія та статус передаються від симулятора до backend, а команди передаються від backend до симулятора.

Таблиця 2.2 – MQTT-теми інформаційної взаємодії між backend і симулятором

MQTT-тема	Напрям передавання	Призначення	Основний вміст повідомлення
pumpstation/telemetry	simulator -> backend	Передавання періодичної телеметрії насосної станції	timestamp, stationId, temperature, pressure, vibration, motorSpeed, motorCurrent, flowRate, pumpState, mode, targetPressure, simulatorState
pumpstation/status	simulator -> backend	Передавання стану після зміни режиму, стану насоса або цільового тиску	timestamp, stationId, pumpState, mode, targetPressure, lastCommandId, message
pumpstation/commands	backend -> simulator	Передавання команд керування до симулятора	commandId, type, issuedAt, commandOrigin, requestedByUserId, systemReason, payload

Тема pumpstation/telemetry використовується з частотою приблизно 1 повідомлення за секунду. У цьому повідомленні містяться всі вимірювані параметри, які потім зберігаються в таблиці TelemetryRecord та відображаються на панелі оператора. Тема pumpstation/status потрібна для швидкого повідомлення про зміну режиму, стану насоса або цільового тиску після обробки команди. Навіть якщо окреме status-повідомлення буде втрачено, наступне telemetry-повідомлення все одно міститиме mode, pumpState і targetPressure.

Тема pumpstation/commands використовується для всіх команд керування. До неї можуть потрапляти команди START_PUMP, STOP_PUMP, EMERGENCY_STOP, SET_MODE, SET_TARGET_PRESSURE і RESET_EMERGENCY. Кожна команда має commandId, що дає симулятору змогу не застосовувати одну й ту саму команду повторно. Це важливо, оскільки для команд використовується надійніший режим доставлення MQTT, а повторне доставлення повідомлення не повинно призводити до повторної зміни стану.

Схему розгортання компонентів можна деталізувати через перелік контейнерів, наведений у таблиці 2.3. У ній відображено, які сервіси відкриті для користувача, а які працюють тільки всередині мережі Docker Compose.

Таблиця 2.3 – Компоненти розгортання системи в Docker Compose

Компонент	Тип розгортання	Зовнішній доступ	Основні зв’язки
frontend	Окремий контейнер зі статичним веб-застосунком	HTTP-порт для браузера користувача	Надсилає REST-запити до backend, відкриває WebSocket-з’єднання з backend
backend	Окремий контейнер FastAPI	HTTP-порт для REST API та WebSocket	Працює з SQLite, підписується на MQTT-теми telemetry і status, публікує commands
mosquitto	Окремий контейнер MQTT-брокера	Може бути відкритий порт 1883 для налагодження	Приймає і маршрутизує MQTT-повідомлення між simulator і backend
simulator	Окремий контейнер Python-симулятора	Зовнішній доступ не потрібний	Публікує telemetry і status, підписується на commands
app.db	Файл SQLite у томі backend	Прямий зовнішній доступ не потрібний	Зберігає користувачів, телеметрію, аварії, команди, налаштування та сесії

Backend є єдиним компонентом, який має доступ одночасно до MQTT і SQLite. Це спрощує відповідальність компонентів. Симулятор не записує дані в базу, бо його завданням є імітація обладнання. Frontend не читає базу напряму, бо всі дані мають проходити через перевірені API-запити. Mosquitto не зберігає телеметрію як архів, оскільки його роль полягає в доставленні повідомлень між клієнтами.

База даних SQLite розміщується у файлі app.db в директорії, підключеній як Docker volume до backend-контейнера. Це означає, що після звичайного

перезапуску контейнерів дані не зникають. У базі зберігаються не тільки історичні значення телеметрії, а й користувачі, refresh-сесії, записи аварій, журнал команд і налаштування порогів. Для демонстраційної системи з одним об’єктом керування та частотою телеметрії 1 запис за секунду такий спосіб зберігання є достатнім.

Під час запуску Docker Compose спочатку має бути доступний MQTT-брокер, після чого backend і simulator підключаються до нього. Backend також створює або відкриває SQLite-базу, перевіряє схему, додає початкових користувачів і налаштування, а потім стає доступним для frontend. Simulator після підключення до Mosquitto починає публікувати telemetry-повідомлення. Frontend відкривається в браузері користувача, виконує вхід або оновлення сесії та підключається до WebSocket-каналу.

Окремою частиною інформаційної взаємодії є автентифікація. Після входу користувача backend повертає короткочасний access token, який frontend зберігає в пам’яті та використовує для REST-запитів і WebSocket-підключення. Refresh token передається через httpOnly cookie, тому JavaScript-код frontend не має до нього доступу. Після перезавантаження сторінки frontend звертається до endpoint оновлення токена, і користувач не втрачає сесію, якщо cookie ще чинний.

У схемі розгортання також враховано втрату зв’язку із симулятором. Якщо backend не отримує телеметрію довше за налаштований telemetryTimeout, він змінює connectionState на OFFLINE, створює аварію CONNECTION_LOSS і передає connectionUpdate на frontend. Команди в такому стані блокуються, оскільки backend не може гарантувати, що симулятор отримає їх і застосує. Коли telemetry-повідомлення знову надходить, backend переводить connectionState в ONLINE, оновлює стан станції та передає це до інтерфейсу оператора.

Розроблена схема інформаційної взаємодії показує, що система не є просто веб-сторінкою з випадковими значеннями. Дані проходять повний шлях від симулятора обладнання через MQTT-брокер до backend, зберігаються в SQLite та передаються у frontend через WebSocket. Команди проходять у зворотному напрямі

через REST API, перевіряються backend і тільки після цього публікуються в MQTT. Такий поділ відповідає темі віддаленого керування промисловим обладнанням з використанням IoT-технології та дає можливість захистити роботу як завершений локальний прототип.

2.4 Розрахунок параметрів передавання, зберігання та обробки телеметрії

Параметри передавання телеметрії визначаються логікою роботи симулятора, MQTT-каналу та backend-частини. У розробленій системі симулятор виконує роль пристрою нижнього рівня, який формує значення параметрів насосної станції та передає їх до backend через брокер Eclipse Mosquitto. Backend приймає повідомлення з теми pumpstation/telemetry, перевіряє їх структуру, зберігає запис у SQLite, оновлює стан станції та передає нові дані на frontend через WebSocket. Такий порядок потрібний для того, щоб веб-інтерфейс не підключався напряму до MQTT і не мав доступу до бази даних.

У розрахунку прийнято, що симулятор надсилає одне телеметричне повідомлення за секунду. Це відповідає демонстраційному режиму роботи, у якому оператор бачить зміну температури, тиску, вібрації, швидкості двигуна, струму двигуна та витрати рідини майже в реальному часі. Частоту формування телеметрії визначено за формулою (2.1).

$$f_{\text{тел}} = \frac{1}{T_{\text{тел}}} = \frac{1}{1} = \ 1 \text{ Гц}, \tag{2.1}$$

де $f_{\text{тел}}$ – частота формування телеметричних повідомлень, Гц;

$T_{\text{тел}}$ – період формування одного повідомлення, с.

Кількість записів, які можуть бути збережені у базі даних протягом доби, визначено за формулою (2.2). Розрахунок виконано для однієї насосної станції з

stationId = "main", оскільки підтримка кількох станцій у цьому прототипі не передбачена.

$$N_{\text{доб}} = f_{\text{тел}} \cdot 60 \cdot 60 \cdot 24 = 1 \cdot 60 \cdot 60 \cdot 24 = 86400 \frac{\text{записів}}{\text{добу}}, \qquad (2.2)$$

де $N_{\text{доб}}$ – кількість телеметричних записів за добу;

$f_{\text{тел}}$ – частота формування телеметричних повідомлень, Гц.

Один телеметричний запис містить часову мітку timestamp, ідентифікатор станції stationId, числові значення temperature, pressure, vibration, motorSpeed, motorCurrent, flowRate, а також станові поля pumpState, mode, targetPressure і, за потреби, simulatorState. У SQLite ці дані зберігаються не як один текстовий блок, а як окремі поля сутності TelemetryRecord. Через це фактичний розмір запису залежить від службових полів таблиці, способу зберігання часу, індексів і внутрішніх витрат SQLite. Для орієнтовної оцінки прийнято, що один запис займає від 0,5 КБ до 1,0 КБ.

Мінімальний і максимальний добовий обсяг телеметрії розраховано за формулами (2.3) і (2.4).

$$V_{\text{доб},min} = N_{\text{доб}} \cdot S_{\text{зап},min} = 86400 \cdot 0{,}5 = 43200 \text{ КБ} \approx 43 \text{ МБ}, \qquad (2.3)$$

де $V_{\text{доб},min}$ – мінімальна оцінка добового обсягу телеметрії, МБ;

$N_{\text{доб}}$ – кількість телеметричних записів за добу;

$S_{\text{зап},min}$ – мінімальна оцінка розміру одного запису, КБ.

$$V_{\text{доб},max} = N_{\text{доб}} \cdot S_{\text{зап},max} = 86400 \cdot 1{,}0 = 86400 \text{ КБ} \approx 86 \text{ МБ}, \qquad (2.4)$$

де $V_{\text{доб},max}$ – максимальна оцінка добового обсягу телеметрії, МБ;

$N_{\text{доб}}$ – кількість телеметричних записів за добу;

$S_{\text{зап},max}$ – максимальна оцінка розміру одного запису, КБ.

За такого режиму роботи база даних отримує приблизно від 43 МБ до 86 МБ телеметричних даних за добу. За 30 діб без очищення історії цей обсяг можна оцінити за формулою (2.5).

$$V_{\text{міс}} = 30 \cdot V_{\text{доб}} \approx 30 \cdot [43;\ 86] \approx [1290;\ 2580]\text{МБ} \approx [1{,}3;\ 2{,}6]\text{ГБ}, \quad (2.5)$$

де $V_{\text{міс}}$ – орієнтовний обсяг телеметричних даних за 30 діб, ГБ;

$V_{\text{доб}}$– орієнтовний добовий обсяг телеметричних даних, МБ.

Отримане значення не є точним вимірюванням розміру файлу app.db. Воно показує порядок обсягу даних, який може накопичуватися за умови збереження кожного повідомлення. Для локального дипломного прототипу такий обсяг є прийнятним, оскільки SQLite-файл зберігається у Docker volume і не потребує окремого серверу бази даних. Якщо після демонстрації потрібно зменшити розмір історії, backend може зберігати телеметрію з меншою частотою або видаляти старі записи після заданого періоду.

Затримка оновлення даних в інтерфейсі складається з кількох частин. Спочатку симулятор очікує наступного моменту формування телеметрії, потім повідомлення проходить через MQTT-брокер, обробляється backend і передається на frontend через WebSocket. Для оцінки використано формулу (2.6).

$$\tau_{\text{оновл}} = T_{\text{тел}} + \tau_{MQTT} + \tau_{backend} + \tau_{WS}, \quad (2.6)$$

де $\tau_{\text{оновл}}$ – орієнтовна затримка появи нового стану в інтерфейсі, с;

$T_{\text{тел}}$ – період формування телеметрії, с;

τ_{MQTT} – затримка передавання повідомлення через MQTT-брокер, с;

$\tau_{backend}$ – час перевірки, збереження та обробки повідомлення у backend, с;

τ_{WS} – час передавання оновлення через WebSocket, с.

У локальному Docker Compose середовищі затримки MQTT, backend і WebSocket зазвичай менші за період телеметрії. Через це основним обмеженням для візуального оновлення є $T_{\text{тел}} = 1$ с. Практично оператор отримує новий стан після надходження наступного повідомлення або після найближчого WebSocket-оновлення. Для такого прототипу розрахункова затримка оновлення приймається на рівні приблизно від 1 с до 2 с.

Окремо розраховано реакцію на втрату зв'язку із симулятором. Backend не використовує MQTT Last Will, а контролює час останнього отриманого телеметричного повідомлення. Якщо нова телеметрія не надходить довше за telemetryTimeout, стан connectionState переходить в OFFLINE, активується аварія CONNECTION_LOSS, а frontend отримує connectionUpdate. Для демонстраційної конфігурації прийнято telemetryTimeout = 5 с. Час виявлення втрати зв'язку визначено за формулою (2.7).

$$\tau_{\text{втр}} = telemetryTimeout + \tau_{WS} = 5 + \tau_{WS} \approx 5 \ldots 6 \text{ с}, \qquad (2.7)$$

де $\tau_{\text{втр}}$ – орієнтовний час від останнього телеметричного повідомлення до відображення втрати зв'язку, с;

$telemetryTimeout$ – допустимий інтервал без телеметрії, с;

τ_{WS} – час передавання повідомлення connectionUpdate до frontend, с.

Значення $telemetryTimeout = 5$ с є компромісом між швидким виявленням проблеми та стійкістю до випадкової затримки одного або кількох повідомлень. Якщо встановити менше значення, коротка пауза в роботі контейнера або MQTT-

клієнта може помилково відобразитися як втрата зв'язку. Якщо встановити значно більше значення, оператор пізніше побачить, що телеметрія вже не надходить, а backend довше дозволятиме роботу зі застарілим станом. У стані OFFLINE нові команди керування блокуються, оскільки backend не може підтвердити, що симулятор їх отримає.

Реакція на аварійний технологічний стан залежить від періоду телеметрії і швидкості обробки повідомлення у backend. Якщо значення temperature, pressure, vibration або motorCurrent перевищує критичний поріг, backend створює або оновлює запис Alarm і формує команду EMERGENCY_STOP з commandOrigin = SYSTEM. Ця команда надсилається до теми pumpstation/commands, а симулятор після її застосування переводить станцію в mode = EMERGENCY і pumpState = STOPPED. Видимий для оператора перехід підтверджується наступним telemetryUpdate або stationSnapshot, а не самим фактом прийняття REST-команди.

Для аварійного стану можна прийняти оцінку, наведену у формулі (2.8). Вона враховує один період очікування небезпечного значення в телеметрії та ще один період до появи підтвердженого стану після застосування команди.

$$\tau_{\text{авар}} \approx 2 \cdot T_{\text{тел}} + \tau_{backend} + \tau_{MQTT} = 2 \cdot 1 + \tau_{backend} + \tau_{MQTT} \approx 2 \dots 3 \text{ с}, \quad (2.8)$$

де $\tau_{\text{авар}}$ – орієнтовний час від появи аварійного значення до відображення аварійного стану в інтерфейсі, с;

$T_{\text{тел}}$ – період формування телеметрії, с;

$\tau_{backend}$ – час перевірки порогів, створення аварії та формування команди, с;

τ_{MQTT} – час доставлення команди EMERGENCY_STOP до симулятора, с.

Ця оцінка не означає, що кожна аварія завжди обробляється рівно за 2 с або 3 с. Вона показує очікуваний порядок часу для локального прототипу, де телеметрія надходить щосекунди, а службові затримки менші за період дискретизації. Для дипломної демонстрації такий час реакції достатній, оскільки оператор бачить

зміну стану без ручного оновлення сторінки, а аварійна команда формується backend автоматично після перевищення критичного порогу.

2.5 Розрахунок системи автоматичного керування електродвигуном насоса

У цій роботі об'єктом для розрахунку системи автоматичного керування обрано електродвигун насоса. Насосна станція як технологічний об'єкт залежить від гідравлічної мережі, параметрів рідини, клапанів і режимів споживання, тому повна гідравлічна модель не вводиться. Для дипломного прототипу достатньо описати головний динамічний елемент, який прямо відображений у симуляторі через параметр motorSpeed. Тиск у системі розглядається як похідна технологічна величина, що змінюється залежно від швидкості двигуна, стану насоса та режиму роботи.

Електродвигун насоса подано як аперіодичну ланку першого порядку. Така модель описує інерційне зростання швидкості після пуску і спадання швидкості після зупинки. У ній вхідним сигналом є нормований керуючий вплив u(t) у діапазоні від 0 до 1, а вихідним сигналом є швидкість обертання двигуна n(t), об/хв. Передавальну функцію двигуна записано за формулою (2.9).

$$W_{\text{дв}}(s) = \frac{N(s)}{U(s)} = \frac{K}{T \cdot s + 1}, \tag{2.9}$$

де $W_{\text{дв}}(s)$ – передавальна функція електродвигуна насоса;

$N(s)$ – зображення швидкості обертання двигуна у просторі Лапласа;

$U(s)$ – зображення нормованого керуючого впливу;

K – статичний коефіцієнт підсилення, об/хв;

T – стала часу двигуна, с;

s – оператор Лапласа.

Для моделі прийнято демонстраційні параметри K = 1200 об/хв і T = 2 с. Значення K відповідає усталеній швидкості при одиничному нормованому керуючому впливі. Значення T = 2 с задає помірну інерційність, яку можна побачити на графіках і телеметричних картках під час запуску або зупинки насоса. Після підстановки параметрів передавальна функція має вигляд формули (2.10).

$$W_{\text{дв}}(s) = \frac{1200}{2 \cdot s + 1}, \tag{2.10}$$

Стійкість моделі визначається коренем характеристичного рівняння. Для ланки першого порядку знаменник передавальної функції дорівнює T·s + 1, тому характеристичне рівняння записано у формулі (2.11).

$$T \cdot s + 1 = 0, \tag{2.11}$$

де T – стала часу двигуна, с;

s – корінь характеристичного рівняння.

Корінь характеристичного рівняння обчислено за формулою (2.12).

$$s = -\frac{1}{T} = -\frac{1}{2} = -0{,}5, \tag{2.12}$$

де s – полюс передавальної функції;

T – стала часу двигуна, с.

Отриманий полюс s = -0,5 розміщений у лівій півплощині комплексної площини. Для лінійної неперервної моделі це означає асимптотичну стійкість. Оскільки модель має тільки один дійсний від'ємний полюс, перехідний процес не має коливань і перерегулювання. За фізичним змістом така властивість відповідає

роботі двигуна, швидкість якого плавно наближається до нового усталеного значення.

Перехідну характеристику для одиничного стрибка керуючого впливу u(t) = 1 визначено за формулою (2.13). Початкова швидкість у цьому випадку прийнята рівною нулю, тобто двигун стартує зі стану спокою.

$$n(t) = K \cdot \left(1 - e^{-\frac{t}{T}}\right), \tag{2.13}$$

де $n(t)$ – швидкість обертання двигуна в момент часу t, об/хв;

K – усталена швидкість при u(t) = 1, об/хв;

t – час від початку перехідного процесу, с;

T – стала часу двигуна, с.

Після підстановки демонстраційних параметрів отримано формулу (2.14).

$$n(t) = 1200 \cdot \left(1 - e^{-\frac{t}{2}}\right), \tag{2.14}$$

За формулою (2.14) швидкість через 2 с після пуску становить приблизно 758 об/хв, тобто близько 63% від усталеного значення. Через 4 с швидкість зростає приблизно до 1037 об/хв, через 6 с до 1140 об/хв, а через 8 с до 1178 об/хв. Після 8 с двигун досягає близько 98 % від усталеного значення 1200 об/хв, тому перехідний процес можна вважати практично завершеним для демонстраційної моделі.

Для часткового керуючого впливу форма перехідного процесу не змінюється, змінюється тільки усталена швидкість. Якщо керуючий вплив дорівнює u_0, то швидкість двигуна визначається за формулою (2.15).

$$n(t) = K \cdot u_0 \cdot \left(1 - e^{-\frac{t}{T}}\right), \tag{2.15}$$

де $n(t)$ – швидкість обертання двигуна в момент часу t, об/хв;

K – статичний коефіцієнт підсилення, об/хв;

u_0 – значення нормованого керуючого впливу;

t – час від початку перехідного процесу, с;

T – стала часу двигуна, с.

Ця залежність використовується для пояснення поведінки motorSpeed у симуляторі. Якщо насос зупинений або станція перебуває в режимі EMERGENCY, ефективний керуючий вплив прямує до нуля, а швидкість двигуна спадає до 0 об/хв. У ручному режимі під час роботи насоса використовується номінальний керуючий вплив, тому швидкість зростає до відповідного сталого значення. В автоматичному режимі AUTO симулятор поступово змінює ефективний керуючий вплив залежно від різниці між поточним тиском і targetPressure.

Оскільки телеметрія формується з періодом 1 с, безперервну модель двигуна зручно пов'язати з дискретним оновленням motorSpeed у симуляторі. Для одного кроку моделювання коефіцієнт наближення до нового усталеного значення визначено за формулою (2.16).

$$\alpha = 1 - e^{-\frac{T_S}{T}} = 1 - e^{-\frac{1}{2}} \approx 0{,}393, \tag{2.16}$$

де α – частка наближення швидкості до нового цільового значення за один крок моделювання;

T_s – період дискретного оновлення, с;

T – стала часу двигуна, с.

З урахуванням коефіцієнта α дискретне оновлення швидкості двигуна можна записати формулою (2.17).

$$n_k = n_{k-1} + \alpha \cdot \left(K \cdot \mathrm{u}_{\mathrm{eff,k}} - n_{k-1}\right), \tag{2.17}$$

де n_k – швидкість двигуна на поточному кроці, об/хв;

n_{k-1} – швидкість двигуна на попередньому кроці, об/хв;

α – коефіцієнт наближення до цільового значення за один крок;

K – статичний коефіцієнт підсилення, об/хв;

$u_{eff,k}$ – ефективний нормований керуючий вплив на поточному кроці.

Формула (2.17) пояснює, чому motorSpeed у симуляторі не змінюється стрибком після команди START_PUMP або STOP_PUMP. Після пуску швидкість поступово наближається до $K \cdot u_eff$, а після аварійної зупинки або звичайної зупинки так само поступово спадає до нуля. Саме ця інерційність робить поведінку симулятора узгодженою з передавальною функцією (2.10).

Автоматичне утримання тиску в режимі AUTO не подається як повний PID-регулятор. У симуляторі тиск використовується для поступової зміни u_eff у межах допустимого діапазону. Якщо pressure нижчий за targetPressure поза мертвою зоною, u_{eff} збільшується. Якщо pressure вищий за targetPressure поза мертвою зоною, u_{eff} зменшується. Крок зміни обмежений, тому динаміка motorSpeed залишається плавною і визначається сталою часу T = 2 с.

Розрахунок показує, що обрана модель електродвигуна є стійкою, має монотонну перехідну характеристику без перерегулювання і дає зрозумілий час встановлення швидкості приблизно від 8 с до 10 с. Це узгоджується з призначенням дипломного прототипу, де треба показати не точну модель конкретного промислового двигуна, а фізично зрозумілу інерційну поведінку керованого обладнання. Через це motorSpeed у телеметрії можна використовувати як прямий зв'язок між теоретичним розрахунком і роботою симулятора під час демонстрації системи.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ СИСТЕМИ

3.1 Розробка блок-схеми алгоритму роботи системи

На відміну від функціональної схеми, наведеної у підрозділі 2.2, яка відображає загальну взаємодію компонентів системи, блок-схема алгоритму деталізує внутрішню логіку обробки телеметрії, перевірки аварійних умов, формування команд керування та оновлення стану інтерфейсу користувача. У цьому підрозділі розглядається не розміщення контейнерів і не загальний обмін між компонентами, а послідовність дій, яку виконує програмне забезпечення після запуску системи та надходження нових даних від симулятора насосної станції.

Алгоритм роботи системи починається із запуску компонентів, після чого backend ініціалізує базу даних, готує REST API, WebSocket-канал і підключення до MQTT-брокера. Симулятор після підключення до брокера починає періодично формувати телеметрію насосної станції. Користувач отримує доступ до панелі керування тільки після авторизації через frontend. Після успішного входу frontend зберігає access token у пам’яті, відкриває WebSocket-з’єднання з backend і отримує початковий знімок стану станції.

Основний цикл роботи системи пов’язаний з обробкою telemetry-повідомлень. Симулятор публікує значення temperature, pressure, vibration, motorSpeed, motorCurrent, flowRate, pumpState, mode, targetPressure і simulatorState у MQTT-тему pumpstation/telemetry. Backend приймає це повідомлення, перевіряє його структуру, записує новий запис телеметрії до SQLite і оновлює останній відомий стан насосної станції. У фрагменті коду, наведеному в лістингу Б.2, показано, що під час приймання телеметрії створюється запис TelemetryRecord, оновлюється StationState, стан з’єднання переводиться в ONLINE, а потім викликається функція evaluate_telemetry для перевірки аварійних умов.

Після збереження телеметрії backend переходить до перевірки технологічних параметрів. Для кожного параметра, який має порогове значення, система порівнює поточне значення з відповідним налаштуванням. До таких параметрів належать температура, тиск, вібрація та струм двигуна. Якщо значення перевищує критичний поріг, backend створює новий активний запис аварії або оновлює вже активний запис такого самого типу. Це важливо, оскільки система не повинна створювати велику кількість однакових аварій при кожному новому повідомленні телеметрії. Для кожного типу аварії в один момент часу може існувати тільки один активний запис.

Окремо в алгоритмі враховано аварію низького тиску. Вона не перевіряється тоді, коли насос зупинений, оскільки низький тиск у стані STOPPED є нормальним наслідком відсутності роботи насоса. Також ця аварія не повинна спрацьовувати одразу після запуску насоса, коли тиск ще не встиг піднятися до робочого рівня. Через це в алгоритмі застосовується grace period після переходу насоса в RUNNING. Така перевірка дає змогу не створювати помилкові аварії під час нормального запуску обладнання.

Для зменшення хибних спрацювань біля порогових значень у системі застосовано механізм hysteresis/debounce. Його призначення полягає в тому, щоб аварія не перемикалася між ACTIVE і RESOLVED через невеликі коливання телеметрії або шум симуляції. Значення має не просто повернутися нижче або вище порога, а перейти в безпечну область з урахуванням hysteresisMargin. Крім цього, безпечний стан повинен зберігатися протягом кількох послідовних повідомлень телеметрії. У лістингу Б.3 наведено фрагмент реалізації функцій _safe і _has_safe_debounce, у яких перевіряється відступ від порога та наявність трьох послідовних безпечних вимірювань.

Критичні аварії мають додатковий вплив на алгоритм керування. Якщо backend уперше активує OVERHEATING, HIGH_PRESSURE, HIGH_VIBRATION або OVERCURRENT, і станція ще не перебуває в режимі EMERGENCY, система

формує намір на аварійну зупинку. Після цього створюється системна команда EMERGENCY_STOP з commandOrigin = SYSTEM. Така команда відрізняється від дії оператора тим, що requestedByUserId не заповнюється, а в systemReason записується причина автоматичного спрацювання. Фрагмент формування користувацьких і системних команд наведено в лістингу Б.5.

Команди оператора обробляються іншим напрямом алгоритму. Коли користувач натискає кнопку запуску, зупинки, зміни режиму або зміни цільового тиску, frontend надсилає REST-запит до backend. Backend спочатку перевіряє access token, роль користувача та коректність структури запиту. Після цього виконується перевірка поточного стану станції. Команда START_PUMP відхиляється в режимі EMERGENCY або тоді, коли насос уже працює. Команда SET_MODE відхиляється під час EMERGENCY або за наявності активної критичної аварії. Команда SET_TARGET_PRESSURE додатково перевіряється на входження значення в налаштований діапазон targetPressureMin і targetPressureMax. Ці перевірки реалізовано у функції _validate_state, фрагмент якої включено до лістингу Б.5.

Якщо команда проходить перевірку, backend створює запис CommandLog зі статусом ACCEPTED і публікує MQTT-повідомлення в тему pumpstation/commands. Якщо публікація не виконалася, статус команди змінюється на FAILED_TO_PUBLISH. У payload MQTT-команди передаються commandId, type, issuedAt, commandOrigin, requestedByUserId і payload. Для системної команди додатково передається systemReason. Це дає симулятору змогу розрізняти звичайні команди користувача та команду аварійної зупинки, створену backend після критичної аварії. Формування MQTT-payload і обробку помилки публікації також наведено в лістингу Б.5.

Після отримання MQTT-команди симулятор перевіряє commandId і не застосовує команду повторно, якщо вона вже була оброблена раніше. Якщо команда є новою та допустимою для поточного стану симулятора, він змінює mode, pumpState або targetPressure, а наступні status і telemetry повідомлення

відображають змінений стан. Фрагмент перевірки commandId і запам’ятовування оброблених команд наведено в лістингу Б.7, а детальніше механізм ідемпотентності команд розглянуто в підрозділі 3.4.

У системі не використовується окрема MQTT-тема для підтвердження виконання команди. REST-відповідь backend фіксує результат серверної обробки команди, але фактичний стан насоса підтверджується наступними telemetry або status повідомленнями від симулятора. Детальніше цей асинхронний механізм підтвердження команд описано в підрозділі 3.4.

Окремий блок алгоритму відповідає за втрату зв’язку із симулятором. Backend зберігає час останнього telemetry-повідомлення. Якщо нові повідомлення не надходять довше, ніж дозволяє параметр telemetryTimeout, система переводить connectionState у OFFLINE і створює аварію CONNECTION_LOSS. У цьому стані нові команди керування відхиляються, тому що backend не може гарантувати їх доставлення до симулятора. У лістингу Б.4 наведено метод _run, який періодично викликає перевірку втрати зв’язку, порівнює попередній і новий стан connectionState та передає оновлення в WebSocket-канал.

Після відновлення телеметрії стан з’єднання повертається в ONLINE. Це відбувається не вручну, а під час приймання нового коректного telemetry-повідомлення. Backend знову оновлює StationState, закриває активну аварію CONNECTION_LOSS і передає поточний стан на frontend. Для оператора це відображається як зникнення індикатора втрати зв’язку та повернення доступності команд, якщо немає інших обмежень за режимом або аваріями.

Оновлення інтерфейсу виконується через WebSocket-канал backend -> frontend. Після підключення користувач отримує початковий stationSnapshot, а далі backend передає зміни стану станції, аварій, команд і з’єднання через окремі типи WebSocket-повідомлень. Серверна частина WebSocket-каналу наведена в лістингу Б.8, де показано реєстрацію клієнта, перевірку origin і token, надсилання

початкового snapshot та функції broadcast. Повний перелік типів WebSocket-повідомлень наведено в таблиці 3.4.

На frontend отримані WebSocket-повідомлення не обробляються окремо в кожному компоненті інтерфейсу. Вони передаються в stationReducer, який оновлює спільний стан станції та забезпечує узгоджену зміну карток телеметрії, графіків, аварій, індикаторів стану і результату останньої команди. Відповідний фрагмент frontend-обробки наведено в лістингу Б.9, а детальніше типи WebSocket-повідомлень розглянуто в підрозділі 3.4.

Блок-схема алгоритму роботи системи наведена на рисунку 3.1. У ній показано основні гілки роботи: запуск системи та авторизацію користувача, приймання телеметрії, перевірку порогів, формування аварій, створення системної команди EMERGENCY_STOP, обробку користувацьких команд, перевірку втрати зв’язку та оновлення frontend через WebSocket.

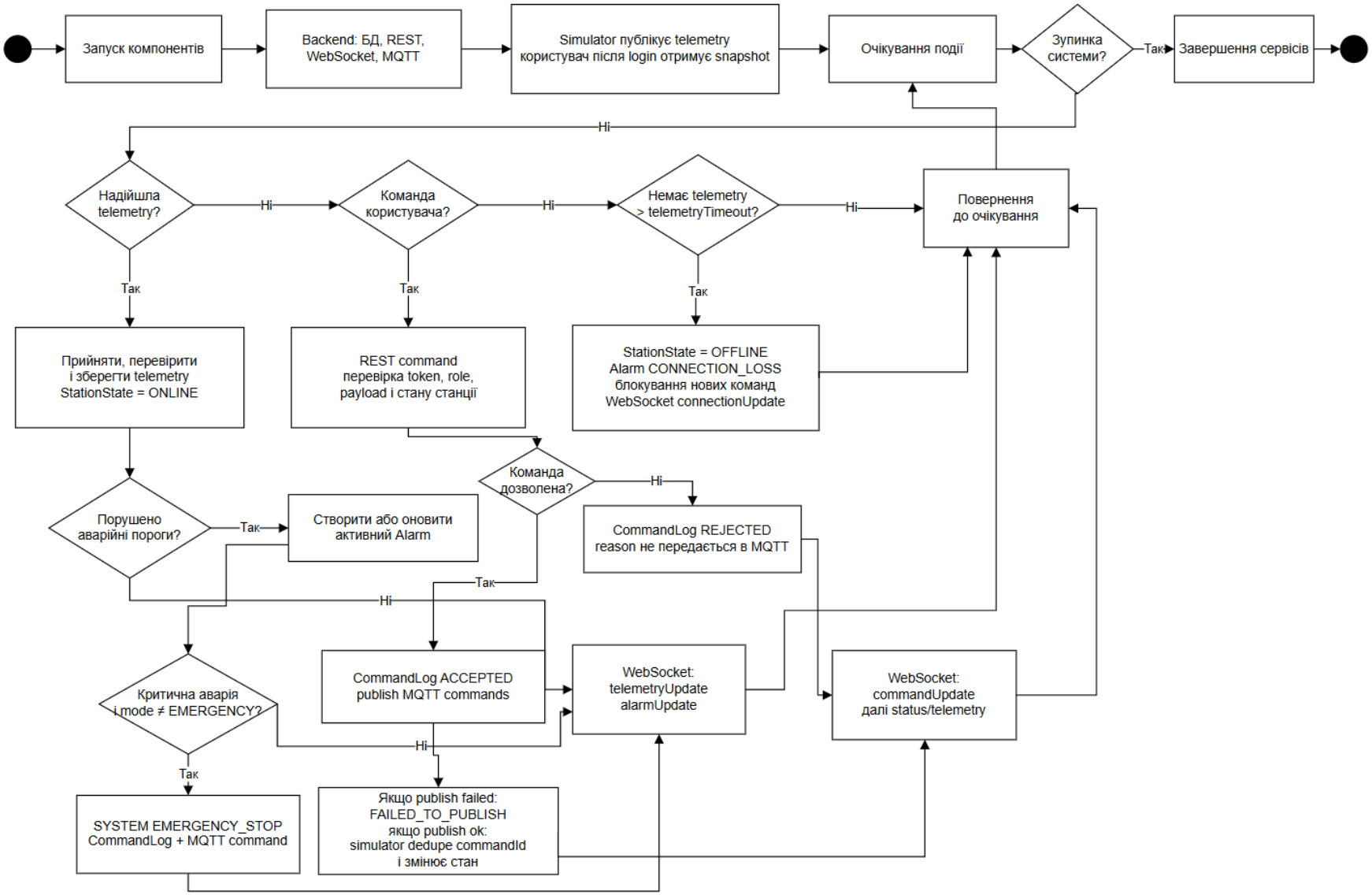


Рисунок 3.1 – Блок-схема алгоритму обробки телеметрії, аварійних станів і команд керування насосною станцією

За блок-схемою видно, що backend є центральною частиною алгоритму. Він не тільки пересилає повідомлення між frontend і симулятором, а й приймає рішення про допустимість команди, стан аварії, втрату зв’язку та необхідність аварійної зупинки. Симулятор відповідає за стан умовного обладнання та формування телеметрії, але не має доступу до бази даних і не перевіряє права користувачів. Frontend відображає стан і надсилає дії користувача, але не приймає остаточних рішень щодо безпеки керування.

Розроблений алгоритм забезпечує узгоджену роботу режимів MANUAL, AUTO та EMERGENCY. У режимах MANUAL і AUTO оператор може керувати насосом відповідно до поточного стану станції. У режимі EMERGENCY звичайні команди блокуються, а повернення до MANUAL можливе тільки через RESET_EMERGENCY після стабілізації критичних параметрів. Така логіка не дає запустити насос або змінити режим керування тоді, коли система ще перебуває в аварійному стані.

Побудована блок-схема також показує різницю між моніторингом і керуванням. Моніторинг працює постійно, оскільки кожне telemetry-повідомлення зберігається, перевіряється та передається в інтерфейс. Керування виконується тільки після дії авторизованого користувача або після автоматичного спрацювання критичної аварії. Усі команди проходять через backend, фіксуються в журналі та передаються до симулятора тільки після перевірок. Це відповідає загальній логіці системи віддаленого керування промисловим обладнанням, де інтерфейс оператора не повинен напряму керувати нижнім рівнем без серверної перевірки.

3.2 Розробка програмних компонентів системи

Програмне забезпечення системи розроблено як набір окремих компонентів, кожен з яких має власну зону відповідальності. Такий поділ потрібний для того, щоб логіка симуляції обладнання, серверна обробка, збереження даних, інтерфейс

користувача та обмін повідомленнями не змішувалися в одному модулі. У підрозділах 2.2 і 2.3 уже було розглянуто загальну функціональну схему та розгортання компонентів, тому далі увага зосереджується саме на програмній реалізації.

У розробленій системі можна виділити чотири основні програмні частини: симулятор насосної станції, backend, frontend і конфігурацію контейнерного запуску. Симулятор відповідає за поведінку умовного обладнання, backend приймає дані, перевіряє команди та зберігає інформацію, frontend забезпечує взаємодію з оператором і адміністратором, а Docker Compose об’єднує всі частини в єдиний локальний прототип. Узагальнений склад програмних компонентів наведено в таблиці 3.1.

Таблиця 3.1 – Програмні компоненти системи віддаленого керування

Компонент	Основна технологія	Основні файли	Призначення
Симулятор	Python	simulator/app/main.py, simulator/app/state.py, simulator/app/mqtt_client.py	Формування телеметрії, підтримання стану насосної станції, обробка команд MQTT
Backend	Python, FastAPI	backend/app/main.py, backend/app/mqtt, backend/app/commands, backend/app/alarms, backend/app/realtime	Приймання телеметрії, збереження даних, перевірка команд, аварійна логіка, REST API і WebSocket
Frontend	React, TypeScript	frontend/src/pages, frontend/src/dashboard, frontend/src/station, frontend/src/api	Відображення телеметрії, аварій, історії, команд і адміністративних сторінок
База даних	SQLite	backend/app/db/models.py, backend/app/db/enums.py	Збереження користувачів, телеметрії, аварій, команд, налаштувань і стану станції
Контейнеризація	Docker Compose	docker-compose.yml, Dockerfile компонентів	Локальний запуск mosquitto, backend, simulator і frontend

Backend є центральним програмним компонентом системи. Його запуск починається з ініціалізації налаштувань, підготовки бази даних, створення об’єкта RealtimeHub, запуску монітора втрати зв’язку та старту MQTT-сервісу. Такий порядок важливий, тому що backend не повинен приймати запити як готовий компонент до того, як створена база даних, підготовлені початкові записи і запущені внутрішні служби. Реалізацію цього життєвого циклу показано у фрагменті Б.1.

У файлі backend/app/main.py також формується FastAPI-застосунок, підключаються CORS-правила, обробники помилок і маршрути окремих модулів. До застосунку підключено маршрути auth, station, settings, users, commands і realtime. Через це main.py не містить бізнес-логіку керування насосом, але виконує роль точки збирання всіх backend-модулів. Окремий endpoint /health використовується для перевірки готовності backend під час контейнерного запуску.

Модуль авторизації забезпечує вхід користувача, оновлення access token і вихід із системи. У системі використано дві ролі: OPERATOR і ADMIN. Оператор має доступ до панелі керування, історії телеметрії та аварій, а адміністратор додатково може керувати користувачами, налаштуваннями та журналом команд. Access token використовується для захищених REST-запитів і WebSocket-підключення. Refresh token зберігається у httpOnly cookie, тому frontend не має доступу до нього через JavaScript.

Модуль приймання MQTT-повідомлень відповідає за обробку телеметрії та статусів від симулятора. Під час надходження telemetry-повідомлення backend створює запис TelemetryRecord, оновлює StationState і передає дані в модуль аварій. Ця логіка не винесена у frontend, тому що саме backend повинен бути джерелом істини для стану станції, аварій і журналу команд. Фрагмент приймання телеметрії наведено в лістингу Б.2.

Модуль аварій перевіряє значення технологічних параметрів і створює записи Alarm. У ньому реалізовано правила для OVERHEATING, HIGH_PRESSURE,

LOW_PRESSURE, HIGH_VIBRATION, OVERCURRENT і CONNECTION_LOSS. Для критичних аварій backend формує системну команду EMERGENCY_STOP. Окремо реалізовано hysteresis/debounce, щоб аварія не перемикалася через випадковий шум у телеметрії. Цей фрагмент наведено в лістингу Б.3.

Модуль commands відповідає за приймання команд від користувача, перевірку стану станції та публікацію MQTT-команд. У ньому розділено користувацькі команди і системні команди. Користувацькі команди надходять через REST API, а системна команда EMERGENCY_STOP створюється backend після критичної аварії. Усі семантично коректні спроби команд фіксуються в CommandLog. Якщо команда відхиляється через стан станції або недопустиме значення targetPressure, вона не передається в MQTT, але запис у журналі зберігається зі статусом REJECTED. Фрагмент перевірки, журналювання та публікації команд керування наведено в лістингу Б.5.

Симулятор реалізовано як окремий Python-компонент. Він не має REST API, WebSocket-каналу і доступу до SQLite. Його основне завдання полягає в тому, щоб виконувати роль нижнього рівня IoT-системи. Після запуску симулятор завантажує налаштування, створює об’єкт PumpStationState, підключається до MQTT-брокера, підписується на тему команд і починає періодично виконувати цикл оновлення стану. Цей порядок роботи відображено у фрагменті Б.7.

Внутрішній стан симулятора містить режим роботи mode, стан насоса pumpState, цільовий тиск targetPressure, швидкість двигуна motorSpeed, температуру, тиск, вібрацію, струм двигуна, витрату рідини та службовий стан simulatorState. Команди START_PUMP, STOP_PUMP, EMERGENCY_STOP, SET_MODE, SET_TARGET_PRESSURE і RESET_EMERGENCY змінюють ці поля відповідно до правил роботи насосної станції. Симулятор також запам’ятовує recent_command_ids, щоб повторне доставлення MQTT-команди з тим самим commandId не призводило до повторної зміни стану.

Окремою частиною симулятора є модель зміни швидкості двигуна. У методі step() використано обчислення коефіцієнта alpha = 1.0 - exp(-elapsed / motor_time_constant_sec), після чого motorSpeed наближається до цільового значення за експоненційним законом. Це відповідає моделі аперіодичної ланки першого порядку, яка використана в розрахунку системи автоматичного керування електродвигуном насоса у підрозділі 2.5. Завдяки цьому зміна motorSpeed у симуляторі не є довільним згладжуванням, а пов’язана з обраною передавальною функцією двигуна.

Автоматичний режим роботи реалізовано у симуляторі, а не в backend. Під час mode = AUTO і pumpState = RUNNING симулятор порівнює поточний pressure з targetPressure та поступово змінює effective_control_input. Якщо тиск нижчий за цільовий, вплив збільшується, якщо вищий, зменшується. Зміна обмежується невеликим кроком, тому тиск не повинен агресивно коливатися навколо заданого значення. Фрагмент оновлення effective_control_input наведено в лістингу Б.7.

Frontend реалізовано як односторінковий веб-застосунок на React і TypeScript. Його структура поділена на модулі API, auth, station, dashboard, pages, layout і ui. Модуль API містить функції звернення до REST endpoint-ів backend. Модуль auth зберігає access token у пам’яті, виконує bootstrap через refresh-запит і забезпечує login/logout. Модуль station підтримує стан насосної станції на основі WebSocket-повідомлень. Компоненти dashboard відповідають за відображення поточних параметрів, графіків, стану станції, аварій і панелі команд.

Панель керування на frontend не є джерелом остаточного рішення про допустимість команди. Вона може блокувати кнопки або показувати повідомлення користувачу, але backend повторно перевіряє команду незалежно від інтерфейсу. Це зроблено для того, щоб користувач не міг обійти правила керування через зміну запиту в браузері або прямий виклик API. Такий поділ особливо важливий для команд START_PUMP, SET_MODE, SET_TARGET_PRESSURE і RESET_EMERGENCY.

Для відображення поточного стану frontend використовує stationReducer. Він обробляє stationSnapshot, telemetryUpdate, alarmUpdate, commandUpdate, connectionUpdate, error і liveStatus. Після отримання telemetryUpdate оновлюються картки параметрів, графіки і стан станції. Після alarmUpdate змінюється список активних аварій і журнал останньої аварійної активності. Після commandUpdate користувач бачить результат обробки команди. Відповідний фрагмент наведено в лістингу Б.9.

Розгортання системи виконано через Docker Compose. У конфігурації описано чотири сервіси: mosquitto, backend, simulator і frontend. Mosquitto має healthcheck на порт 1883. Backend залежить від готовності mosquitto і перевіряється через endpoint /health. Simulator також запускається після готовності mosquitto і має власний healthcheck через файл стану. Frontend залежить від готовності backend. Фрагмент конфігурації розгортання наведено в лістингу Б.10.

Програмні компоненти системи розроблено так, щоб кожен з них можна було пояснити окремо під час демонстрації. Симулятор показує, де формується телеметрія і як змінюється стан насосної станції. Backend показує, де відбувається перевірка команд, аварій і прав доступу. Frontend показує, як користувач взаємодіє з системою. Docker Compose показує, як ці частини запускаються разом і утворюють працездатний локальний IoT-прототип.

3.3 Розробка моделі даних системи

Після оцінювання обсягу телеметрії в підрозділі 2.4 потрібно визначити, які дані система повинна зберігати постійно та як вони пов’язані між собою. Модель даних системи побудовано з урахуванням того, що backend є єдиним компонентом, який працює з базою даних. Симулятор передає дані через MQTT і не має прямого доступу до сховища. Frontend отримує інформацію через REST API та WebSocket і також не звертається до SQLite напряму. Через це модель даних відображає не

тільки історію телеметрії, а й користувачів, сесії, аварії, команди, налаштування та останній відомий стан насосної станції.

Для збереження даних використано SQLite. Такий вибір відповідає характеру прототипу, оскільки система працює з одним об’єктом керування stationId = main і не потребує окремого серверного процесу бази даних. Файл app.db зберігається в томі Docker, підключеному до backend-контейнера. Після перезапуску контейнерів історія телеметрії, аварії, користувачі, налаштування і журнал команд залишаються доступними.

У коді модель описано через ORM-класи SQLAlchemy. Для обмеження допустимих значень застосовано enum-и та CheckConstraint. Це зменшує ризик появи в базі некоректних режимів, станів насоса, ролей, типів аварій або типів команд. Основні enum-значення подано у фрагменті Б.6. Вони відповідають тим значенням, які використовуються в MQTT-повідомленнях, REST API, WebSocket-оновленнях і frontend-стані.

Основні сутності моделі даних наведено в таблиці 3.2. У таблиці вказано призначення кожної сутності та її зв’язок із роботою системи.

Таблиця 3.2 – Основні сутності моделі даних системи

Сутність	Призначення	Основні дані
User	Зберігання облікових записів користувачів	username, passwordHash, role, isActive, createdAt
RefreshSession	Підтримка оновлення access token після перезавантаження сторінки	userId, refreshTokenHash, issuedAt, expiresAt, revokedAt
TelemetryRecord	Історія телеметрії насосної станції	timestamp, stationId, temperature, pressure, vibration, motorSpeed, motorCurrent, flowRate, pumpState, mode, targetPressure
Alarm	Активні та завершені аварійні стани	type, severity, state, activatedAt, resolvedAt, currentValue, threshold

Продовження таблиці 3.2

Сутність	Призначення	Основні дані
CommandLog	Журнал команд користувача і системних команд	commandId, type, payload, commandOrigin, requestedByUserId, systemReason, status
StationSetting	Налаштування порогів і параметрів станції	temperatureMax, pressureMax, vibrationMax, telemetryTimeout, hysteresisMargin, targetPressureMin, targetPressureMax
StationState	Останній відомий стан насосної станції	mode, pumpState, targetPressure, connectionState, lastTelemetryAt

Сутність User використовується для збереження користувачів системи. Кожен користувач має унікальне ім’я, хеш пароля, роль і ознаку активності. У базі не зберігається відкритий пароль. Роль користувача визначає, які сторінки та операції йому доступні. Оператор може працювати з панеллю керування, історією телеметрії та аваріями, а адміністратор додатково отримує доступ до налаштувань, користувачів і журналу команд.

RefreshSession потрібна для підтримки сесії після перезавантаження сторінки. Access token зберігається тільки в пам’яті frontend, тому після оновлення сторінки він втрачається. Щоб користувач не проходив вхід повторно, backend перевіряє refresh token, який передається через httpOnly cookie, і видає новий access token. У таблиці RefreshSession зберігається не сам refresh token, а його хеш, час створення, час завершення дії та ознака відкликання.

TelemetryRecord є найбільш часто заповнюваною сутністю. Новий запис створюється під час кожного коректного telemetry-повідомлення від симулятора. У записі зберігаються значення температури, тиску, вібрації, швидкості двигуна, струму двигуна, витрати рідини, режиму, стану насоса і цільового тиску. Для прискорення запитів за періодом часу використано індекс за stationId і timestamp. Такі записи потрібні для сторінки історії та для аналізу зміни параметрів після команд керування.

У TelemetryRecord передбачено обмеження, які захищають модель від некоректних станів. Поле stationId обмежене значенням main, оскільки система працює з однією насосною станцією. Для pumpState дозволено тільки STOPPED або RUNNING, а для mode тільки MANUAL, AUTO або EMERGENCY. Також на рівні таблиці заборонено комбінацію mode = EMERGENCY і pumpState = RUNNING. Це важливо, оскільки аварійний режим не повинен поєднуватися з працюючим насосом.

Сутність Alarm зберігає аварійні стани. Вона містить тип аварії, рівень критичності, стан ACTIVE або RESOLVED, час активації, час завершення, поточне значення, поріг і стан насосної станції на момент спрацювання. Для аварій також використано індекси, які прискорюють пошук активної аварії за типом і перегляд історії за часом активації.

Особливістю таблиці Alarm є частковий унікальний індекс uq_alarms_one_active_per_type. Він дозволяє мати не більше одного активного запису для кожного типу аварії. Це означає, що правило single-active захищене не тільки логікою модуля alarms, а й схемою SQLite. Якщо аварія такого типу вже активна, backend оновлює її currentValue і lastUpdatedAt, а не створює новий рядок. Відповідний фрагмент моделі Alarm наведено в лістингу Б.6.

CommandLog зберігає журнал команд. У цю таблицю потрапляють користувацькі команди, які пройшли рівень семантичної обробки, а також системні команди backend. Запис містить commandId, тип команди, payload, походження команди, користувача або системну причину, час створення, статус і причину відхилення. Для USER-команди обов’язково вказується requestedByUserId і не заповнюється systemReason. Для SYSTEM-команди requestedByUserId має бути порожнім, а systemReason обов’язковим. Це обмеження реалізовано на рівні CheckConstraint.

Статус команди може мати значення ACCEPTED, REJECTED або FAILED_TO_PUBLISH. ACCEPTED означає, що backend прийняв команду і зміг

опублікувати її в MQTT. REJECTED означає, що команда була розпізнана як правильна за формою, але відхилена через стан станції, аварійний режим або недопустиме значення. FAILED_TO_PUBLISH означає, що команда пройшла перевірки, але backend не зміг передати її в MQTT. Завдяки такій моделі адміністратор може перевірити не тільки успішні дії, а й відхилені спроби керування.

StationSetting зберігає пороги та параметри станції. До цієї сутності входять temperatureMax, pressureMax, pressureMinOperating, vibrationMax, motorCurrentMax, telemetryTimeout, lowPressureGracePeriod, hysteresisMargin, targetPressureDefault, targetPressureMin і targetPressureMax. Ці значення використовуються модулем аварій, перевіркою команд і сторінкою налаштувань. Для StationSetting задано обмеження додатності значень, перевірку targetPressureMin < targetPressureMax і pressureMinOperating < pressureMax.

StationState зберігає останній відомий стан насосної станції. На відміну від TelemetryRecord, ця таблиця не є історією вимірювань. Вона потрібна для швидкого отримання поточного режиму, стану насоса, цільового тиску, стану з’єднання і часу останньої телеметрії. Під час обробки telemetry-повідомлення StationState оновлюється, а під час втрати телеметрії connectionState переходить в OFFLINE.

Зв’язки між сутностями в моделі даних залишаються простими. RefreshSession пов’язана з User через userId. CommandLog для користувацьких команд також посилається на User через requestedByUserId. StationSetting може зберігати updatedByUserId, якщо налаштування були змінені адміністратором. TelemetryRecord і Alarm не мають зв’язку з користувачем, оскільки вони формуються не діями людини, а телеметрією та логікою backend.

Структура моделі даних наведена на рисунку 3.2. На схемі потрібно показати основні таблиці, їхні поля та зв’язки між User, RefreshSession, CommandLog і StationSetting. TelemetryRecord, Alarm і StationState можна показати як окремі сутності, пов’язані зі stationId = main.

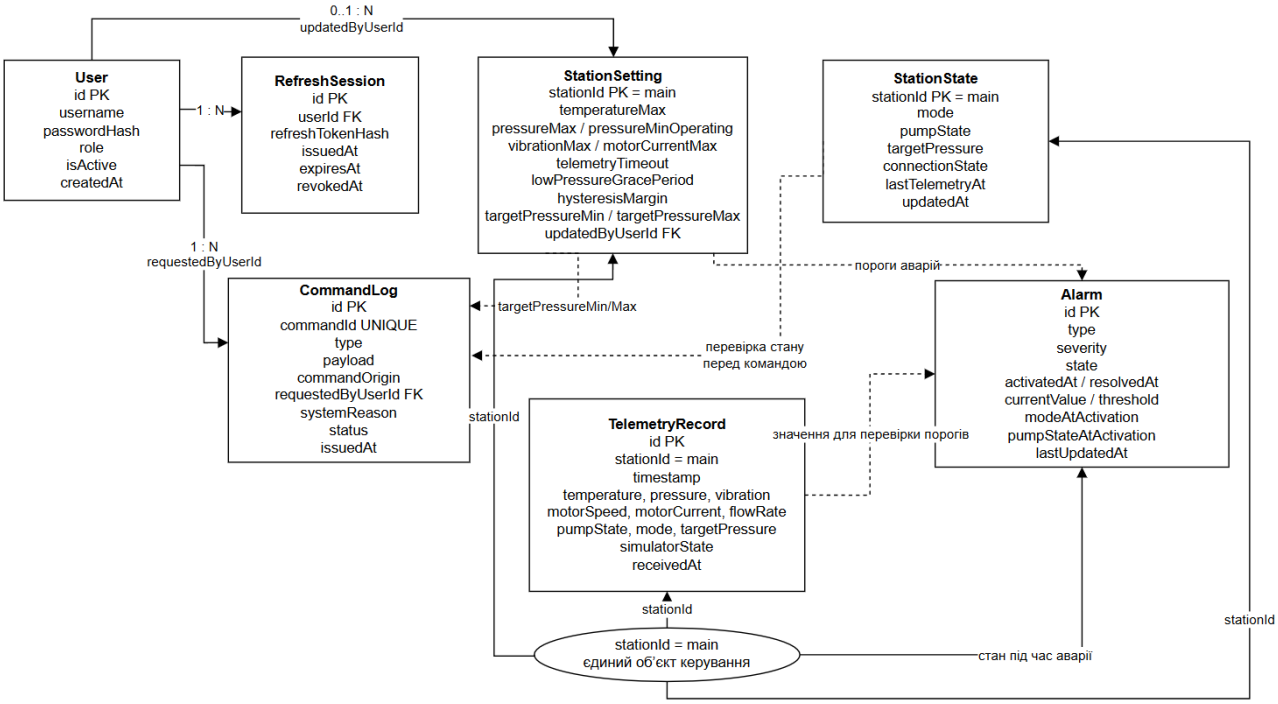


Рисунок 3.2 – Модель даних системи віддаленого керування насосною станцією

Розроблена модель даних відповідає логіці роботи системи. Вона дозволяє зберігати історію телеметрії, відстежувати аварії, фіксувати команди, підтримувати користувацькі сесії та зберігати поточні налаштування станції. Обмеження на рівні бази даних доповнюють перевірки backend і зменшують ризик появи некоректних станів у сховищі.

3.4 Розробка обміну повідомленнями та оновлення даних у реальному часі

Механізми обміну повідомленнями в системі поділено на три рівні. Перший рівень відповідає за обмін між симулятором і backend через MQTT. Другий рівень забезпечує запити frontend до backend через REST API. Третій рівень передає поточні оновлення від backend до frontend через WebSocket. Такий поділ дозволяє використовувати кожний канал для того типу взаємодії, для якого він найбільше підходить.

MQTT використовується тільки між симулятором і backend. Frontend не підключається до MQTT-брокера, тому що браузерний інтерфейс не повинен напряму працювати з нижнім рівнем обладнання. Через MQTT проходять telemetry, status і commands. REST API використовується для дій, які мають чіткий запит і відповідь: авторизація, перегляд історії, отримання аварій, зміна налаштувань, керування користувачами та надсилання команд. WebSocket використовується для тих подій, які повинні потрапляти до інтерфейсу без ручного оновлення сторінки.

Загальна схема обміну повідомленнями наведена на рисунку 3.3. На ній потрібно показати напрям передавання telemetry і status від simulator до backend, напрям commands від backend до simulator, REST-запити від frontend до backend і WebSocket-оновлення від backend до frontend.

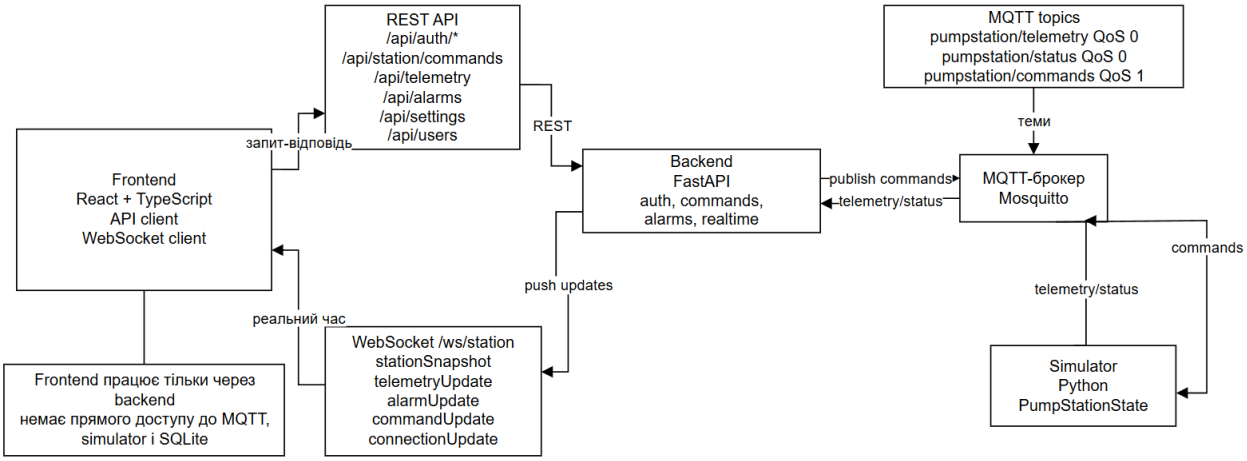


Рисунок 3.3 – Схема обміну повідомленнями між simulator, backend і frontend

MQTT-рівень використовує три теми: pumpstation/telemetry, pumpstation/status і pumpstation/commands. Тема telemetry містить періодичні вимірювання насосної станції. Тема status передає зміну стану після обробки команди або внутрішньої події симулятора. Тема commands передає керувальні команди від backend до симулятора. Ці теми вже були показані в таблиці 2.2, тому

в цьому підрозділі розглядається не їх загальний зміст, а порядок обробки повідомлень у програмному забезпеченні.

Під час надходження telemetry-повідомлення backend виконує кілька послідовних дій. Спочатку MQTT-сервіс приймає JSON-повідомлення та перетворює його в структуру TelemetryPayload. Далі викликається ingest_telemetry, де створюється запис TelemetryRecord і оновлюється StationState. Після цього виконується evaluate_telemetry, яка перевіряє пороги аварій і може повернути наміри на системну команду EMERGENCY_STOP. Цей шлях показано в лістингах Б.2 і Б.3.

Status-повідомлення має допоміжну роль. Воно передає mode, pumpState, targetPressure і lastCommandId після зміни стану симулятора. Навіть якщо status-повідомлення буде втрачено, наступне telemetry-повідомлення все одно містить основні поля стану. Через це status не використовується як єдине джерело істини. Backend може оновити стан за status, але остаточно стан станції регулярно підтверджується telemetry-повідомленнями.

Команди передаються в MQTT тільки після обробки в backend. Frontend не створює MQTT-повідомлення. Користувач натискає кнопку в інтерфейсі, після чого frontend надсилає REST-запит до POST /api/station/commands. Backend перевіряє користувача, роль, тип команди, payload і поточний стан станції. Якщо команда допустима, backend створює CommandLog, формує MQTT-payload і публікує повідомлення в pumpstation/commands. Формування такого payload наведено в лістингу Б.5.

У системі реалізовано асинхронне підтвердження команд. REST-відповідь повертає commandId, status і rejectionReason, але вона не підтверджує фізичне застосування команди симулятором. Якщо статус ACCEPTED, це означає, що backend перевірив команду і зміг передати її в MQTT. Фактичний стан станції підтверджується пізніше, коли симулятор обробить команду і опублікує нові status

або telemetry повідомлення. Такий підхід узгоджується з тим, що окремого ACK-топіка в MQTT-рівні не передбачено.

Для команд використовується commandId. Симулятор перевіряє, чи не обробляв він таку команду раніше. Якщо commandId уже є в recent_command_ids, команда не застосовується повторно. Це важливо для MQTT-команд, тому що доставка з QoS 1 може повторити повідомлення після тимчасового збою. Ідемпотентність команд реалізовано на рівні симулятора, а у backend commandId додатково зберігається в CommandLog як унікальний ідентифікатор.

REST API використовується не тільки для команд. Через нього виконуються login, refresh, logout, отримання поточного користувача, перегляд стану станції, історії телеметрії, аварій, налаштувань, користувачів і журналу команд. Для частини endpoint-ів достатньо ролі OPERATOR, а адміністративні endpoint-и доступні тільки ролі ADMIN. Усі захищені REST-запити використовують access token у заголовку Authorization.

Основні типи взаємодії через REST API наведено в таблиці 3.3. Вона показує, які дії виконуються у форматі запит-відповідь і не потребують постійного каналу.

Таблиця 3.3 – Основні REST-взаємодії frontend з backend

Група запитів	Приклади endpoint-ів	Призначення
Авторизація	/api/auth/login, /api/auth/refresh, /api/auth/logout, /api/auth/me	Вхід, оновлення токена, вихід і отримання поточного користувача
Стан станції	/api/station/state	Отримання останнього відомого стану насосної станції
Команди	/api/station/commands	Надсилання команди керування та отримання результату її обробки backend
Історія	/api/telemetry, /api/alarms	Перегляд збереженої телеметрії та аварій
Налаштування	/api/settings	Перегляд і зміна порогів та параметрів станції

Продовження таблиці 3.3

Група запитів	Приклади endpoint-ів	Призначення
Адміністрування	/api/users, /api/station/commands	Керування користувачами та перегляд журналу команд

WebSocket-канал використовується для оновлення інтерфейсу в реальному часі. Після авторизації frontend відкриває підключення до /ws/station і передає access token у query parameter. Backend перевіряє origin і token, після чого реєструє клієнта в RealtimeHub. Одразу після підключення клієнт отримує stationSnapshot. Цей snapshot потрібний для того, щоб frontend мав повний стан станції навіть до надходження наступного telemetryUpdate. Серверну частину WebSocket-каналу наведено в лістингу Б.8.

Повідомлення WebSocket мають спільну структуру: type, timestamp і payload. Поле type визначає, як frontend має обробити повідомлення. Для stationSnapshot payload містить повний знімок стану. Для telemetryUpdate передається нова телеметрія і стан станції. Для alarmUpdate передається запис аварії. Для commandUpdate передається результат обробки команди. Для connectionUpdate передається ONLINE або OFFLINE. Для error передається повідомлення про помилку live-каналу.

Типи WebSocket-повідомлень наведено в таблиці 3.4. Ця таблиця показує, які події backend передає у frontend без опитування REST API.

Таблиця 3.4 – Типи WebSocket-повідомлень backend -> frontend

Тип повідомлення	Коли надсилається	Що оновлює у frontend
stationSnapshot	Одразу після успішного WebSocket-підключення або перепідключення	Повний стан станції, останню телеметрію, активні аварії
telemetryUpdate	Після приймання і обробки нового telemetry-повідомлення	Картки параметрів, графіки, режим, стан насоса, цільовий тиск

Продовження таблиці 3.4

Тип повідомлення	Коли надсилається	Що оновлює у frontend
alarmUpdate	Після створення, оновлення або завершення аварії	Список активних аварій і останню аварійну активність
commandUpdate	Після обробки користувацької або системної команди	Останній результат команди, статус ACCEPTED, REJECTED або FAILED_TO_PUBLISH
connectionUpdate	Після переходу simulator у ONLINE або OFFLINE	Індикатор стану з’єднання і доступність команд
error	Після помилки live-каналу	Повідомлення про помилку або стан перепідключення

RealtimeHub на backend відповідає за розсилання WebSocket-повідомлень усім підключеним клієнтам. Він зберігає активні WebSocket-з’єднання, надсилає початковий snapshot і виконує broadcast для telemetryUpdate, alarmUpdate, commandUpdate та connectionUpdate. Така логіка потрібна, щоб кілька відкритих інтерфейсів бачили однаковий стан станції. Наприклад, якщо адміністратор і оператор одночасно відкрили dashboard, обидва отримають однакове повідомлення про аварію або результат системної команди EMERGENCY_STOP.

На frontend WebSocket-повідомлення надходять у stationReducer. Цей reducer не виконує мережевих запитів і не приймає рішень про керування насосом. Його завдання полягає в тому, щоб перетворити повідомлення backend на новий стан інтерфейсу. Після stationSnapshot стан повністю замінюється на отриманий snapshot. Після telemetryUpdate оновлюється остання телеметрія та буфер графіків. Після alarmUpdate змінюється список аварій. Після commandUpdate оновлюється результат останньої команди. Фрагмент обробки повідомлень наведено в лістингу Б.9.

Окремо обробляється liveStatus. Це не повідомлення від backend у тому самому сенсі, що telemetryUpdate або alarmUpdate, а внутрішній стан WebSocket-клієнта frontend. Він потрібний для того, щоб інтерфейс міг показувати

підключення, перепідключення, помилку або втрату live-каналу. Завдяки цьому користувач бачить не тільки стан насосної станції, а й стан каналу, через який надходять оновлення.

Механізм оновлення в реальному часі не замінює збереження історії. TelemetryUpdate потрібний для поточного dashboard, але історичні записи все одно зберігаються в SQLite і запитуються через REST API. Це розділення важливе, оскільки dashboard працює з коротким буфером останніх значень, а сторінка History має отримувати дані за вибраний період із бази даних. Аналогічно alarmUpdate показує зміну аварії в моменті, але повний перелік аварій доступний через REST-запит до /api/alarms.

У разі втрати WebSocket-з'єднання frontend не повинен мовчки показувати дані як живі. Він переводить liveStatus у стан помилки або перепідключення, показує відповідний індикатор і намагається відновити канал. Після повторного підключення backend надсилає новий stationSnapshot, який стає авторитетним станом для frontend. Це дозволяє інтерфейсу відновитися навіть після тимчасової втрати live-каналу або перезапуску backend.

Розроблений механізм обміну повідомленнями забезпечує розділення відповідальності між каналами. MQTT використовується для зв'язку з нижнім рівнем, REST API для контрольованих запитів frontend до backend, WebSocket для поточних оновлень інтерфейсу. Команди не вважаються застосованими до появи наступного стану від симулятора, аварії фіксуються в базі даних і передаються у frontend, а стан з'єднання контролюється backend. Така організація дає змогу показати систему як повний IoT-прототип віддаленого керування насосною станцією.

3.5 Практична перевірка роботи системи віддаленого керування

Практична перевірка розробленої системи виконувалася після реалізації основних програмних компонентів, описаних у підрозділах 3.2–3.4. Метою перевірки було підтвердити, що система працює не як набір окремих сторінок або модулів, а як єдиний IoT-прототип віддаленого керування насосною станцією. Під час перевірки контролювався повний шлях даних: формування телеметрії в симуляторі, передавання повідомлень через MQTT-брокер, обробка даних у backend, збереження в SQLite, оновлення frontend через WebSocket і передавання команд у зворотному напрямі.

Перевірка виконувалася в локальному середовищі Docker Compose. У запуску брали участь чотири контейнери: mosquitto, backend, simulator і frontend. База даних SQLite не запускалася як окремий сервіс, а зберігалася у файлі app.db у томі, підключеному до backend-контейнера. Такий спосіб запуску відповідає схемі розгортання, розглянутій у підрозділі 2.3, і дає змогу відтворити демонстрацію системи на іншому ПК без ручного запуску кожної частини окремо.

Після виконання команди docker compose up --build перевірявся стан усіх сервісів. Mosquitto мав приймати MQTT-підключення, backend мав відкрити REST API та WebSocket-канал, симулятор мав почати публікувати телеметрію, а frontend мав бути доступним у браузері. Стан контейнерів після запуску системи наведено на рисунку 3.4.

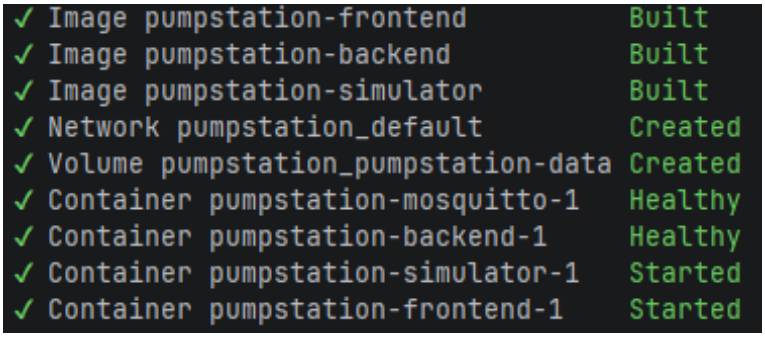


Рисунок 3.4 – Стан контейнерів Docker Compose після запуску системи

Першим кроком користувацької перевірки був вхід до системи з обліковим записом оператора. На сторінці входу користувач вводив логін і пароль, після чого frontend надсилав запит до backend. У разі успішної автентифікації backend повертав access token, а refresh token встановлювався через httpOnly cookie. Після цього frontend переходив на панель керування і відкривав WebSocket-з'єднання для отримання поточного стану станції. Сторінка входу оператора наведена на рисунку 3.5.

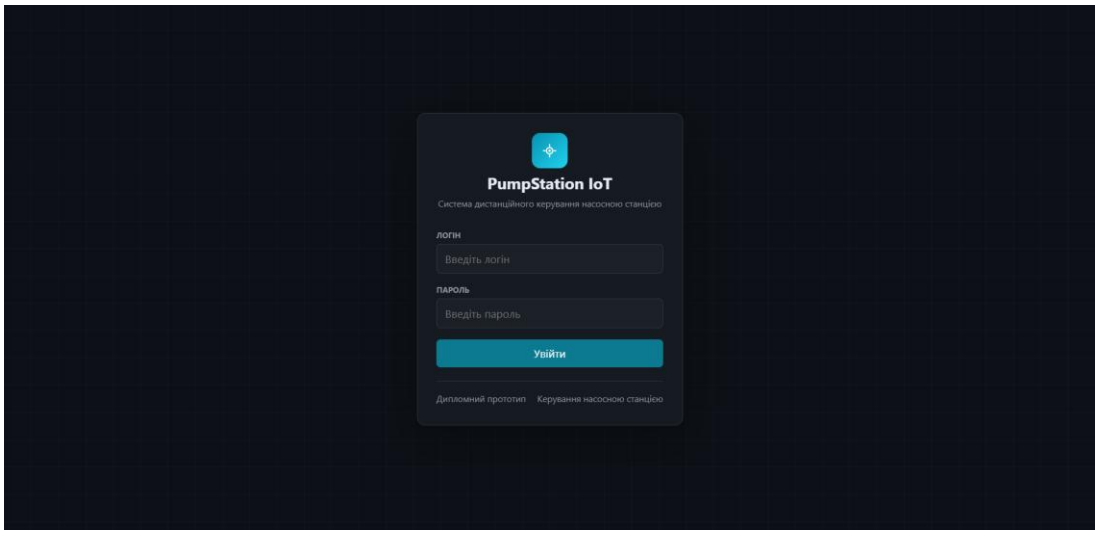


Рисунок 3.5 – Сторінка входу оператора до системи PumpStation IoT

Після входу оператор потрапляв на dashboard. У початковому стані насосна станція перебувала в режимі MANUAL, насос мав стан STOPPED, а з'єднання із симулятором було ONLINE. На сторінці відображалися картки поточних параметрів: temperature, pressure, vibration, motorSpeed, motorCurrent і flowRate. Також були доступні індикатори режиму роботи, стану насоса, стану з'єднання, активних аварій і панель команд керування. Початковий стан панелі оператора після авторизації показано на рисунку 3.6.



Рисунок 3.6 – Панель оператора з поточною телеметрією насосної станції

На цьому етапі перевірялося, що веб-інтерфейс отримує дані не напряму від симулятора, а через backend. Симулятор формував telemetry-повідомлення і публікував їх у MQTT-тему pumpstation/telemetry. Backend приймав ці повідомлення, перевіряв їхню структуру, створював записи TelemetryRecord, оновлював StationState і передавав поточний стан у WebSocket-повідомленнях. У frontend ці повідомлення потрапляли до stationReducer, після чого змінювалися картки параметрів, графіки та індикатори стану. Така перевірка підтвердила правильність інформаційного шляху від симулятора через mosquitto і backend до frontend.

Далі перевірявся запуск насоса оператором. Після натискання кнопки запуску frontend надсилав REST-запит із командою START_PUMP до endpoint /api/station/commands. Backend перевіряв access token, роль користувача, тип команди та поточний стан станції. Оскільки станція не перебувала в режимі EMERGENCY і з’єднання було ONLINE, команда приймалася, записувалася в CommandLog зі статусом ACCEPTED і публікувалася в MQTT-тему pumpstation/commands. Симулятор отримував команду, змінював pumpState на

RUNNING і передавав оновлений стан у наступних status та telemetry повідомленнях.

Після запуску насоса на dashboard змінювався стан pumpState, а значення motorSpeed починало зростати поступово. Це підтверджує зв’язок між практичною поведінкою симулятора та розрахунком електродвигуна, наведеним у підрозділі 2.5. Швидкість не змінювалася миттєво, а наближалася до нового значення за інерційним законом. Разом зі швидкістю двигуна змінювалися тиск і витрата рідини, що показує залежність технологічних параметрів від стану насоса. Стан системи після запуску насоса наведено на рисунку 3.7.



Рисунок 3.7 – Стан насосної станції після виконання команди START_PUMP

Наступним етапом була перевірка переходу до автоматичного режиму. Оператор перемикав mode з MANUAL на AUTO. Команда SET_MODE також проходила через backend, а не надсилалася безпосередньо до симулятора. Після перевірки поточного стану backend публікував MQTT-команду, а симулятор змінював внутрішній режим роботи. У режимі AUTO під час RUNNING активувався спрощений алгоритм підтримання тиску, який поступово змінює

ефективний керувальний вплив залежно від різниці між поточним pressure і targetPressure.

Після переходу в AUTO перевірялася зміна цільового тиску. Оператор вводив нове значення targetPressure і надсилав команду SET_TARGET_PRESSURE. Frontend міг виконати попередню перевірку введеного значення, але остаточне рішення приймав backend. Якщо значення входило в межі targetPressureMin і targetPressureMax, команда приймалася і передавалася до симулятора. Після її застосування новий targetPressure відображався на dashboard, а pressure поступово наближався до заданого значення. Результат роботи автоматичного режиму після зміни цільового тиску показано на рисунку 3.8.

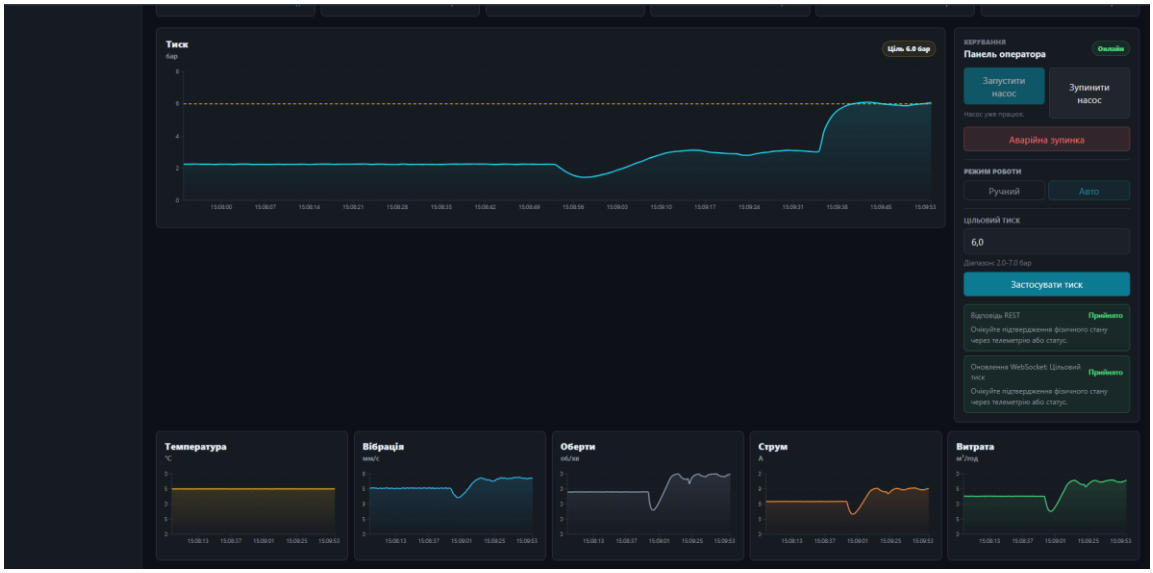


Рисунок 3.8 – Робота автоматичного режиму після зміни цільового тиску

Практична перевірка підтвердила, що REST-відповідь backend не використовується як остаточне підтвердження фізичного стану насоса. Якщо команда отримала статус ACCEPTED, це означає тільки те, що backend перевірив її, записав у журнал і зміг опублікувати в MQTT. Фактична зміна стану відображалася лише після того, як симулятор обробляв команду і надсилав нові status або telemetry повідомлення. Завдяки цьому інтерфейс не показував запуск,

зміну режиму або зміну targetPressure тільки на підставі натискання кнопки користувачем.

Далі перевірялася аварійна зупинка. Для цього використовувалася команда EMERGENCY_STOP або демонстраційне аварійне спрацювання симулятора. У разі ручної аварійної зупинки оператор натискав відповідну кнопку на dashboard, після чого backend приймав команду і передавав її в MQTT. У разі автоматичного спрацювання backend створював системну команду EMERGENCY_STOP після виявлення критичного значення в телеметрії, наприклад перегріву, підвищеного тиску, вібрації або струму двигуна. В обох випадках симулятор переводив станцію в mode = EMERGENCY і pumpState = STOPPED.

Після переходу в аварійний режим на панелі оператора з’являвся аварійний індикатор, а звичайні команди керування блокувалися. Команди START_PUMP, SET_MODE і SET_TARGET_PRESSURE не повинні виконуватися, поки система перебуває в EMERGENCY. Швидкість двигуна при цьому спадала до нуля не миттєво, а з інерційною динамікою, як було описано в підрозділі 2.5. Стан системи після аварійної зупинки наведено на рисунку 3.9.



Рисунок 3.9 – Перехід насосної станції в аварійний режим після команди EMERGENCY_STOP

Після аварійної зупинки перевірялося скидання аварійного режиму. Команда RESET_EMERGENCY дозволяється не в будь-який момент, а тільки після стабілізації критичних параметрів. Backend перевіряє, що mode = EMERGENCY, pumpState = STOPPED, connectionState = ONLINE, а критичні технологічні значення повернулися в безпечну область з урахуванням hysteresisMargin і debounce. Якщо хоча б одна умова не виконана, команда відхиляється, а frontend показує причину відхилення.

Після виконання умов безпечного скидання оператор натискав Reset Emergency. Backend приймав команду, записував її в CommandLog і передавав до симулятора. Симулятор переводив mode у MANUAL, залишав насос у стані STOPPED, а backend переводив відповідні критичні аварії в RESOLVED. Після цього панель оператора знову дозволяла звичайні команди керування. Результат успішного скидання аварійного режиму показано на рисунку 3.10.

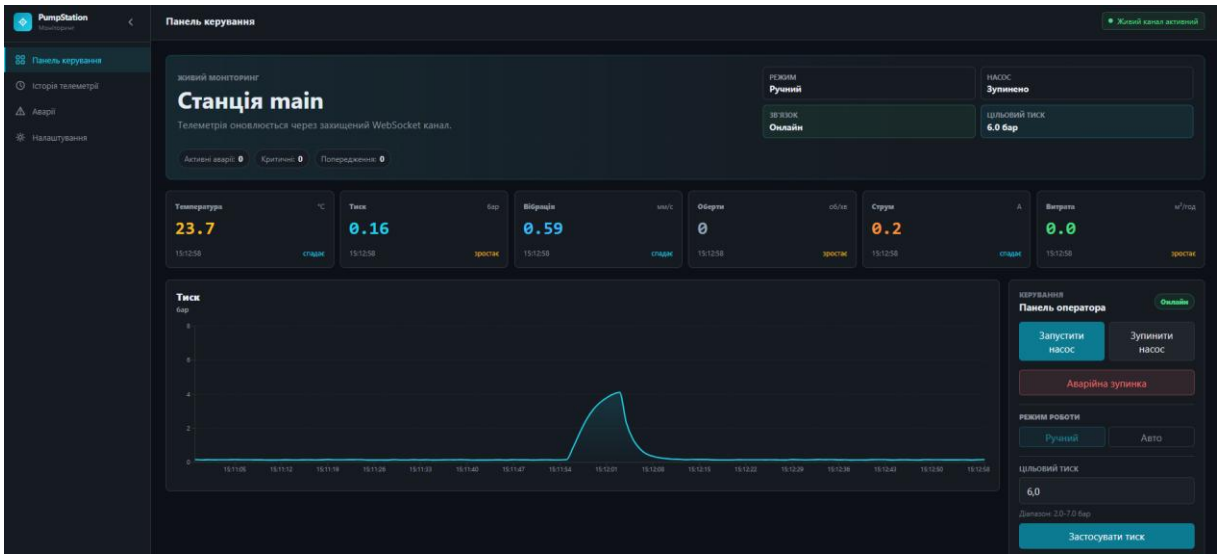


Рисунок 3.10 – Стан системи після успішного виконання RESET_EMERGENCY

Окремо перевірялася реакція системи на втрату зв’язку із симулятором. Для цього simulator-контейнер тимчасово зупинявся. Після того як backend не отримував telemetry-повідомлення довше за налаштований telemetryTimeout,

connectionState переходив у OFFLINE, створювалася аварія CONNECTION_LOSS, а frontend отримував connectionUpdate. У цьому стані команди керування блокувалися, оскільки backend не міг гарантувати доставлення команд до симулятора. Після повторного запуску simulator-контейнера нове telemetry-повідомлення переводило connectionState назад в ONLINE. Відображення втрати зв’язку із симулятором наведено на рисунку 3.11.

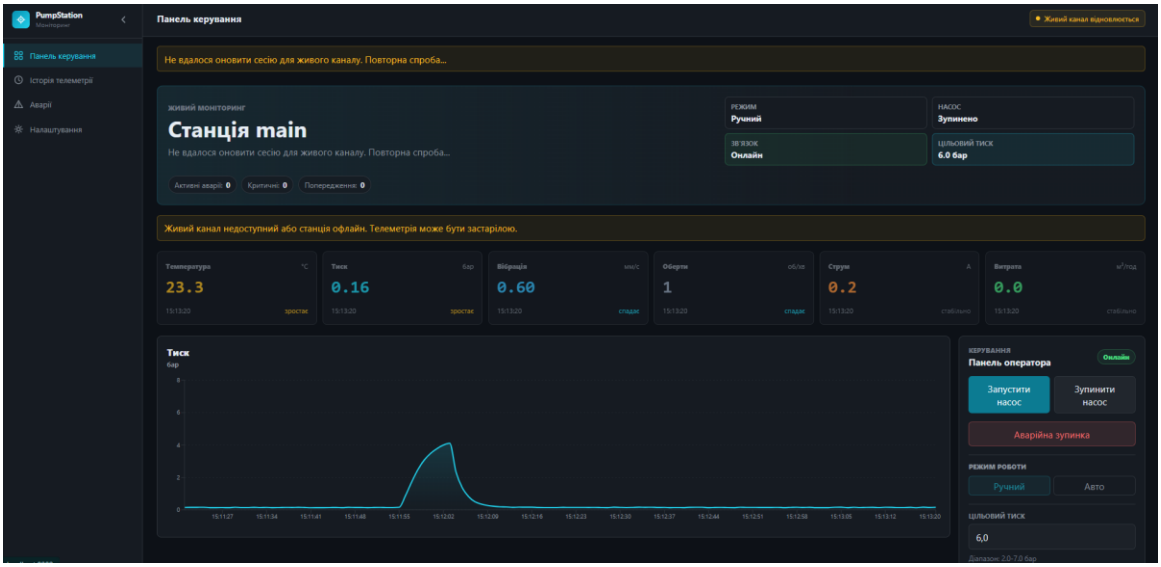


Рисунок 3.11 – Відображення стану OFFLINE після зупинки симулятора

Після перевірки дій оператора виконувалася перевірка входу з роллю адміністратора. Адміністратор мав доступ до тих самих функцій керування насосом, що й оператор, але додатково міг відкривати сторінки Settings, Users і Command History. Це дозволило перевірити рольову модель доступу не тільки на рівні інтерфейсу, а й через backend. Якщо оператор намагався відкрити адміністративні сторінки напряму, frontend мав заблокувати доступ, а backend мав повернути заборону для відповідного захищеного endpoint.

На сторінці Users перевірялася наявність списку користувачів, їхніх ролей і стану активності. Адміністратор міг створити нового користувача, змінити роль або деактивувати обліковий запис. На сторінці Settings перевірялися налаштування

порогів температури, тиску, вібрації, струму двигуна, telemetryTimeout, lowPressureGracePeriod, hysteresisMargin і допустимого діапазону targetPressure. Спроба зберегти некоректні значення мала відхилятися backend, оскільки frontend-перевірка не є єдиним захистом. Адміністративна сторінка налаштувань наведена на рисунку 3.12.



Рисунок 3.12 – Сторінка налаштувань порогів і параметрів насосної станції

Останнім етапом перевірявся журнал команд. На сторінці Command History адміністратор переглядав користувацькі команди та системні команди backend. У журналі мали відображатися START_PUMP, SET_MODE, SET_TARGET_PRESSURE, EMERGENCY_STOP і RESET_EMERGENCY, а також автоматична системна команда EMERGENCY_STOP, якщо вона була створена через критичну аварію. Для кожного запису перевірялися commandId, type, commandOrigin, payload, користувач або systemReason, час створення, status і rejectionReason за наявності. Сторінка журналу команд адміністратора наведена на рисунку 3.13.

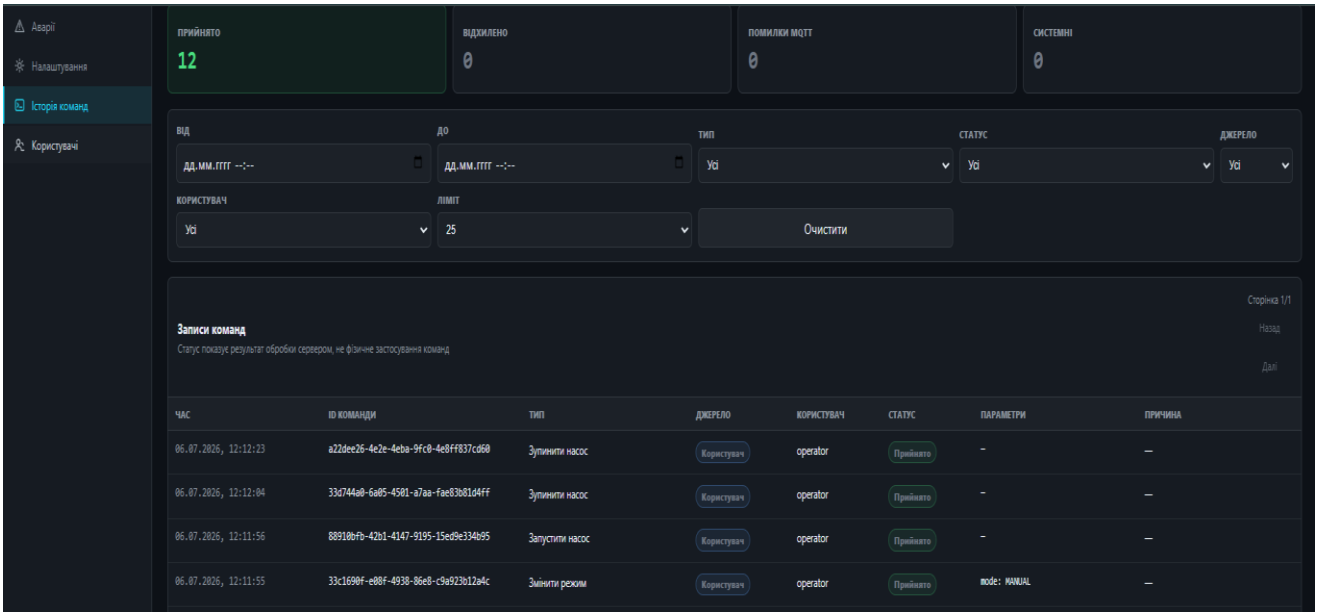


Рисунок 3.13 – Журнал користувацьких і системних команд у режимі адміністратора

Під час практичної перевірки також було підтверджено, що симулятор виконує роль нижнього рівня IoT-системи. У поточній реалізації фізичний мікроконтролер, ПЛК і датчики не використовуються, оскільки їхню функцію замінено програмною моделлю насосної станції. Саме в симуляторі формується телеметрія, змінюються mode, pumpState, targetPressure і motorSpeed, а також моделюються аварійні стани.

Якщо переносити розроблену систему на фізичний стенд, місце симулятора зайняв би апаратний вузол на базі мікроконтролера або ПЛК. До нього могли б підключатися датчик тиску на трубопроводі, датчик температури корпусу двигуна або насоса, датчик вібрації на корпусі агрегату, датчик струму в колі живлення двигуна, витратомір на вихідній лінії та датчик швидкості обертання двигуна, наприклад тахогенератор або енкодер на валу. Ці сигнали після вимірювання перетворювалися б у ті самі логічні поля, які використані в прототипі: temperature, pressure, vibration, motorCurrent, flowRate і motorSpeed.

У такому апаратному варіанті frontend усе одно не отримував би дані напряму з датчиків. Вимірювальний вузол формував би JSON-повідомлення і передавав їх через MQTT-брокер у тему pumpstation/telemetry. Backend приймав би ці повідомлення, зберігав їх у SQLite, перевіряв аварійні умови і передавав стан у frontend через WebSocket. Команди проходили б у зворотному напрямі: frontend надсилав би REST-запит до backend, backend перевіряв би команду і публікував її в MQTT-тему pumpstation/commands, а апаратний вузол уже керував би виконавчою частиною насоса. У дипломному прототипі цей апаратний вузол замінено Python-симулятором, тому для демонстрації коду нижнього рівня використовуються файли теки simulator.

Практична перевірка показала, що система виконує основні функції, необхідні для теми кваліфікаційної роботи. Оператор може увійти в систему, переглядати живу телеметрію, запускати насос, переводити станцію в AUTO, змінювати targetPressure, виконувати аварійну зупинку і скидати аварійний режим після стабілізації параметрів. Адміністратор може додатково перевіряти користувачів, налаштування і журнал команд. Дані проходять через MQTT, REST API і WebSocket відповідно до розробленої архітектури, а backend залишається центральним компонентом, який перевіряє права доступу, стан обладнання, аварії та допустимість команд.

3.6 Охорона праці

Під час виконання кваліфікаційної роботи було розроблено програмний прототип системи віддаленого керування промисловим обладнанням. Практична частина роботи виконувалася за персональним комп’ютером із використанням середовища розробки, браузера, Docker Compose, MQTT-брокера, backend, frontend і програмного симулятора насосної станції. Реальна насосна станція, фізичні датчики, ПЛК, силові шафи та електродвигун у роботі не використовувалися, тому

небезпечні виробничі чинники, пов’язані з рухомими частинами насоса, підвищеним тиском у трубопроводі або роботою силового електрообладнання, у практичній перевірці не виникали.

Основними чинниками, які потрібно враховувати для цієї роботи, є тривала робота з екранним пристроєм, статичне навантаження на опорно-руховий апарат, напруження зору, електрична небезпека під час використання комп’ютерного обладнання та пожежна небезпека в приміщенні. Загальні питання організації безпечної праці, виробничої санітарії та профілактики небезпек розглядаються в навчальній літературі з охорони праці [14]. Нормативну основу підрозділу становлять Закон України «Про охорону праці» [15], вимоги щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями [16], правила безпечної експлуатації електроустановок споживачів [17], правила пожежної безпеки в Україні [18] і вимоги до природного та штучного освітлення приміщень [19].

Робоче місце розробника має бути організоване так, щоб зменшити навантаження на зір і не створювати незручної робочої пози. Монітор потрібно розміщувати перед користувачем, без різких відблисків від вікон або світильників. Клавіатура та миша мають бути розташовані так, щоб руки не перебували в постійному напруженні. Робоче крісло повинно підтримувати спину, а висота стола має давати змогу працювати без надмірного нахилу корпуса. Під час тривалої роботи з кодом, тестуванням веб-інтерфейсу та переглядом журналів контейнерів потрібно робити короткі перерви, оскільки безперервна робота з екраном підвищує втому очей і погіршує концентрацію.

Для зменшення зорового навантаження приміщення має мати достатнє природне або штучне освітлення. Освітлення повинно бути рівномірним, без мерехтіння та різких контрастів між екраном і навколишніми поверхнями. Екран не бажано розміщувати напроти вікна або прямо під джерелом світла, оскільки це створює відблиски і ускладнює читання тексту. Під час розробки цієї системи

значна частина роботи пов’язана з дрібними елементами інтерфейсу, JSON-повідомленнями, логами backend і MQTT-повідомленнями, тому якість освітлення прямо впливає на точність роботи та зменшує ймовірність помилок.

Електробезпека під час роботи над програмною системою пов’язана з використанням персонального комп’ютера, монітора, блока живлення, мережевого обладнання та периферійних пристроїв. Перед початком роботи потрібно переконатися, що кабелі живлення не мають видимих пошкоджень, розетки не перевантажені, а обладнання не розміщене поруч із рідинами. Не можна самостійно розбирати блок живлення, монітор або мережевий адаптер, якщо вони підключені до електромережі. У разі появи запаху горіння, іскріння або нестабільної роботи обладнання роботу потрібно припинити, від’єднати живлення безпечним способом і повідомити відповідальну особу.

Пожежна небезпека для такого робочого місця пов’язана переважно з електрообладнанням, подовжувачами, зарядними пристроями та можливим перегріванням блоків живлення. Для її зменшення не можна використовувати пошкоджені кабелі, саморобні подовжувачі, перевантажені розетки або залишати обладнання в несправному стані. Після завершення роботи бажано вимикати обладнання, яке не повинно працювати постійно. У приміщенні мають бути доступні первинні засоби пожежогасіння, а проходи до них не повинні бути заставлені меблями або обладнанням. Для електрообладнання небезпечно застосовувати воду, тому потрібно використовувати засоби пожежогасіння, придатні для гасіння електроустановок.

Окремо потрібно врахувати особливості теми кваліфікаційної роботи. Розроблена система призначена для віддаленого моніторингу та керування насосною станцією. У програмному прототипі нижній рівень замінено симулятором, але в реальній системі замість нього міг би працювати мікроконтролер або ПЛК, підключений до датчиків і виконавчих пристроїв. У такій реалізації віддалений інтерфейс не повинен замінювати локальні засоби безпеки.

Фізична насосна станція має мати локальну аварійну зупинку, захист від перевантаження двигуна, захист від небезпечного тиску, коректне заземлення, справні шафи керування та засоби блокування під час обслуговування.

У розробленому прототипі частина вимог безпеки відображена на програмному рівні. Backend перевіряє роль користувача, стан з'єднання із симулятором, режим роботи станції та допустимість команд перед їх передаванням у MQTT. Команди блокуються, якщо станція перебуває в режимі EMERGENCY або якщо backend втратив телеметрію від симулятора. Для критичних аварій передбачено автоматичне формування команди EMERGENCY_STOP, після чого насос переходить у зупинений аварійний стан. Така логіка не є повною промисловою системою захисту, але вона показує правильний принцип: небезпечна команда не повинна передаватися на нижній рівень без перевірки.

Вплив розробленого програмного прототипу на навколишнє середовище є мінімальним, оскільки робота не передбачає використання хімічних речовин, промислових викидів, мастил або реального насосного обладнання. Основними екологічними аспектами є споживання електроенергії комп'ютером і правильне поводження з електронним обладнанням після завершення строку його експлуатації. Несправні кабелі, батареї, мережеві пристрої та інші електронні компоненти не можна викидати разом із побутовими відходами, їх потрібно передавати на відповідну утилізацію.

За результатами аналізу можна зробити висновок, що для виконання цієї кваліфікаційної роботи основними заходами з охорони праці є правильна організація робочого місця за комп'ютером, дотримання режиму роботи з екранним пристроєм, забезпечення безпечного використання електрообладнання, дотримання правил пожежної безпеки та врахування майбутніх вимог до безпечної роботи фізичної насосної станції. Програмний прототип не створює промислових небезпек безпосередньо, але його логіка має бути побудована з урахуванням того,

що в реальному впровадженні команди керування можуть впливати на фізичне обладнання.

ВИСНОВКИ

У кваліфікаційній роботі розроблено локальний прототип системи віддаленого керування промисловим обладнанням з використанням IoT-технології. Як приклад промислового об’єкта використано насосну станцію, а її нижній рівень реалізовано програмним симулятором. Це дало змогу перевірити повний шлях даних без використання фізичного насоса, датчиків, ПЛК або окремого апаратного стенда.

Під час виконання роботи було проаналізовано предметну область віддаленого моніторингу і керування промисловим обладнанням, розглянуто застосування IoT-технологій у таких системах і порівняно протоколи передавання даних. Для обміну між симулятором і серверною частиною обрано MQTT, оскільки він підходить для регулярної передачі телеметрії та команд через брокер повідомлень. Для взаємодії веб-інтерфейсу з backend використано REST API та WebSocket, що дозволило розділити звичайні запити й оновлення поточного стану в реальному часі.

У роботі обґрунтовано вибір програмно-технічних засобів системи. До складу прототипу входять Python-симулятор, MQTT-брокер Eclipse Mosquitto, backend на FastAPI, база даних SQLite, frontend на React і TypeScript та локальне розгортання через Docker Compose. Система запускається як набір контейнерів, а база даних зберігається у файлі SQLite на підключеному томі Docker. Така структура є достатньою для демонстраційного прототипу з однією насосною станцією та не потребує окремого сервера бази даних.

Розроблена система підтримує два типи користувачів, оператора й адміністратора. Оператор може переглядати поточну телеметрію, історію, аварії та надсилати команди керування насосною станцією. Адміністратор додатково має доступ до налаштувань, журналу команд і керування користувачами. Для захисту

доступу використано автентифікацію з короткочасним JWT access token і refresh token в httpOnly cookie, а перевірка прав виконується на backend.

У практичній частині реалізовано передавання шести параметрів телеметрії: температури, тиску, вібрації, швидкості двигуна, струму двигуна та витрати рідини. Система працює з режимами MANUAL, AUTO і EMERGENCY, станами насоса STOPPED і RUNNING, а також зі станом зв'язку ONLINE або OFFLINE. Для керування передбачено команди START_PUMP, STOP_PUMP, EMERGENCY_STOP, SET_MODE, SET_TARGET_PRESSURE і RESET_EMERGENCY. Команди не надсилаються напряму з frontend до симулятора, а проходять перевірку на backend, що зменшує ризик некоректного керування.

Окрему увагу приділено аварійній логіці. Backend аналізує телеметрію, створює записи аварій, відстежує втрату зв'язку із симулятором і формує системну команду EMERGENCY_STOP у разі критичних аварій. У режимі EMERGENCY насос зупиняється, а звичайні команди блокуються до стабілізації параметрів і виконання RESET_EMERGENCY. Це не замінює промислові апаратні засоби захисту, але демонструє правильну логіку програмного рівня для віддаленого керування обладнанням.

Для розділу з теорії автоматичного керування було обрано електродвигун насоса як основний динамічний елемент. Його описано аперіодичною ланкою першого порядку з коефіцієнтом підсилення K = 1200 об/хв і сталою часу T = 2 с. За результатами аналізу полюс моделі дорівнює s = -0,5, тому модель є асимптотично стійкою. Перехідний процес має монотонний характер без перерегулювання, а швидкість двигуна наближається до усталеного значення приблизно за 8-10 с. Ця модель узгоджена з логікою симулятора, у якому motorSpeed не змінюється миттєво, а наближається до нового значення інерційно.

Практична перевірка підтвердила працездатність основних частин прототипу. Було перевірено запуск системи через Docker Compose, вхід користувача,

отримання телеметрії через MQTT, збереження записів у SQLite, оновлення інтерфейсу через WebSocket, виконання команд керування, роботу автоматичного режиму, аварійну зупинку, втрату зв’язку із симулятором і доступ адміністратора до службових сторінок. Отримані результати відповідають поставленому завданню, оскільки система демонструє повний цикл віддаленого моніторингу, керування, збереження даних і реакції на аварійні стани.

Практична цінність роботи полягає в тому, що розроблений прототип можна використовувати як навчальний або демонстраційний стенд для пояснення структури IoT-систем керування промисловим обладнанням. Він показує взаємодію нижнього рівня, брокера повідомлень, серверної частини, бази даних і веб-інтерфейсу. Матеріали роботи можуть бути використані у навчальному процесі під час вивчення IoT-систем, віддаленого моніторингу, MQTT-обміну, WebSocket-оновлень, ролей користувачів і програмної логіки аварійного захисту.

Фізичну апаратуру в роботі не створювали, а вимірювання з реальних датчиків не виконували. Для перенесення системи на реальний стенд потрібно замінити симулятор на мікроконтролер або ПЛК, підключити датчики тиску, температури, вібрації, струму і витрати, додати апаратні блокування, локальну аварійну зупинку, промислові вимоги до електробезпеки та захищене мережеве з’єднання. Також доцільним напрямом подальшої роботи є підтримка кількох станцій, перехід на серверну СКБД для тривалого зберігання даних, розширення журналювання подій і тестування системи з реальним контролером.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Методичні вказівки з підготовки кваліфікаційної роботи для здобувачів першого (бакалаврського) рівня вищої освіти спеціальності 151 Автоматизація та комп'ютерно-інтегровані технології освітньої програми «Системна інженерія» / Упоряд.: І.Ш. Невлюдов, О.М. Цимбал, О.В. Токарєва, А.І. Бронніков. Харків: ХНУРЕ, 2023. 65 с.

2. Навчальний посібник з підготовки кваліфікаційної роботи бакалавра для здобувачів вищої освіти денної і заочної форм навчання спеціальності 151 «Автоматизація та комп'ютерно-інтегровані технології» освітньої програми «Автоматизація та комп'ютерно-інтегровані технології» : Навчальний посібник / І. Ш. Невлюдов, В. А. Андрусевич, О. В. Токарєва, С. П. Новоселов, О. В. Сичова. – Харків : Видавництво Іванченка І. С., 2022. 151 с.

3. ДСТУ 3008-15. Документація. Звіти у сфері науки та техніки. структура та правила оформлення. – Введ. 2015-06-22. – К. Держстандарт України, 2017 – 29 с

4. Основи автоматизації технологічних процесів. Конспект лекцій : навч. посіб. / уклад.: В. В. Шевченко, Г. С. Тимчик ; КПІ ім. Ігоря Сікорського. Київ, 2023. 111 с. URL: https://ela.kpi.ua/handle/123456789/57481 (дата звернення: 27.06.2026).

5. Лукінюк М. В. Автоматизація типових технологічних процесів: технологічні об'єкти керування та схеми автоматизації : навч. посіб. Київ : Політехніка, 2008. 236 с. URL: https://ela.kpi.ua/handle/123456789/49706 (дата звернення: 27.06.2026).

6. Муляр Ю. І., Репінський С. В. Автоматизація виробництва в машинобудуванні. Частина ІІ : навч. посіб. Вінниця : ВНТУ, 2020. 123 с. URL: https://ir.lib.vntu.edu.ua/handle/123456789/34726 (дата звернення: 27.06.2026).

7. Стратегія розвитку «Індустрія 4.0» в Україні. АППАУ, 2018. URL: https://mautic.appau.org.ua/asset/42%3Astrategia-rozvitku-4-0-v3.pdf (дата звернення: 27.06.2026).

8. Жураковський Б. Ю., Зенів І. О. Технології інтернету речей : навч. посіб. Київ : КПІ ім. Ігоря Сікорського, 2021. 271 с. URL: https://ela.kpi.ua/handle/123456789/42078 (дата звернення: 27.06.2026).

9. Вступ до Інтернету речей : навч. посіб. / уклад. В. В. Макаренко ; КПІ ім. Ігоря Сікорського. Київ, 2025. 536 с. URL: https://ela.kpi.ua/handle/123456789/74124 (дата звернення: 27.06.2026).

10. MQTT Version 5.0 / ed. A. Banks, E. Briggs, K. Borgendale, R. Gupta. OASIS Standard, 2019. URL: https://www.oasis-open.org/standard/mqtt-v5-0-os/ (дата звернення: 27.06.2026).

11. OPC Unified Architecture. OPC Foundation. URL: https://opcfoundation.org/about/opc-technologies/opc-ua/ (дата звернення: 27.06.2026).

12. Modbus Application Protocol Specification V1.1b3. Modbus Organization, 2012. URL: https://www.modbus.org/modbus-specifications (дата звернення: 27.06.2026).

13. WebSocket API. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API (дата звернення: 27.06.2026).

14. Docker Compose. Docker Docs. URL: https://docs.docker.com/compose/ (дата звернення: 27.06.2026).

15. Охорона праці та цивільний захист : підручник / О. Г. Левченко та ін. ; за ред. О. Г. Левченка. Київ : КПІ ім. Ігоря Сікорського, 2019. 420 с. URL: https://ela.kpi.ua/handle/123456789/26895 (дата звернення: 05.07.2026).

16. Про охорону праці : Закон України від 14.10.1992 № 2694-XII / Верховна Рада України. URL: https://zakon.rada.gov.ua/laws/show/2694-12 (дата звернення: 05.07.2026).

17. Про затвердження Вимог щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями : Наказ Міністерства соціальної політики України від 14.02.2018 № 207. URL: https://zakon.rada.gov.ua/go/z0508-18 (дата звернення: 05.07.2026).

18. Про затвердження Правил безпечної експлуатації електроустановок споживачів (ДНАОП 0.00-1.21-98) : Наказ Мінпраці України від 09.01.1998 № 4. URL: https://zakon.rada.gov.ua/go/z0093-98 (дата звернення: 05.07.2026).

19. Про затвердження Правил пожежної безпеки в Україні : Наказ Міністерства внутрішніх справ України від 30.12.2014 № 1417. URL: https://zakon.rada.gov.ua/go/z0252-15 (дата звернення: 05.07.2026).

20. ДБН В.2.5-28:2018. Природне і штучне освітлення. Київ : Мінрегіон України, 2018. URL: https://e-construction.gov.ua/laws_detail/3074958732556240833?doc_type=2 (дата звернення: 05.07.2026).

ДОДАТОК А

Графічні матеріали

