

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ
СІКОРСЬКОГО»**

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці

ДО ЗАХИСТУ ДОПУЩЕНО

В.о. завідувача кафедри

Олександр КОВАЛЬ

«__» _____ 2026 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інженерія програмного забезпечення
інтелектуальних кібер-фізичних систем в енергетиці»
спеціальності 121 Інженерія програмного забезпечення
на тему: «Розробка системи моніторингу та збору помилок для
розподілених додатків»

Виконав:

студент IV курсу, групи ТВ-23

Чуйко Нікіта Олегович

(прізвище, ім'я, по батькові)

(підпис)

Керівник:

Старший викладач, PhD

Колумбет Вадим Петрович

(посада, науковий ступінь, вчене звання, прізвище та ініціали)

(підпис)

Рецензент:

Доцент, к.ф.-м., доцент кафедри ЦТЕ

Донець Андрій Георгійович

(посада, науковий ступінь, вчене звання, прізвище та ініціали)

(підпис)

Засвідчую, що у цій дипломній роботі
немає запозичень із праць інших авторів
без відповідних посилань.

Студент _____

(підпис)

Київ – 2026

**Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»**

Навчально-науковий інститут атомної та теплової енергетики

Кафедра інженерії програмного забезпечення в енергетиці

Рівень вищої освіти перший (бакалаврський)

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення
інтелектуальних кібер-фізичних систем в енергетиці»

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри

_____ Олександр КОВАЛЬ

«___» _____ 2026р.

ЗАВДАННЯ

на дипломну роботу студенту

Чуйко Нікіті Олеговичу

(прізвище, ім'я, по батькові)

1. Тема роботи: Розробка системи моніторингу та збору помилок для розподілених додатків

керівник роботи Колумбет Вадим Петрович, старший викладач, PhD

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «28» травня 2026 р. № 1992-с

2. Строк подання студентом роботи «10» червня 2026 р.

3. Вихідні дані до роботи: платформа Node.js, система керування базами даних PostgreSQL (формат JSONB), бібліотека React, технологія контейнеризації Docker, хмарний сервіс штучного інтелекту Google Gemini API.

4. Зміст (дипломної роботи) пояснювальної записки (перелік завдань, які потрібно розробити): проаналізувати предметну область та існуючі підходи до моніторингу розподілених систем, визначити вимоги до системи, спроектувати архітектуру та логічну структуру бази даних, описати вибраний стек технологій, розробити алгоритм дедуплікації помилок, реалізувати серверну частину (REST API колектор) та клієнтський веб-інтерфейс (Dashboard), налаштувати інтеграцію зі штучним інтелектом та системою сповіщень, провести функціональне та навантажувальне тестування.

5. Перелік ілюстративного матеріалу: UML-діаграми (Use-case, IDEF0),

архітектурна модель системи, концептуальна та логічна схеми бази даних, алгоритм дедуплікації інцидентів, графічний інтерфейс користувача (дашборд, аналітика, налаштування).

6. Дата видачі завдання «31» жовтня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Строки виконання етапів роботи	Примітка
1	Отримання завдання	31.10.2025	Виконано
2	Дослідження предметної області	01.11.2025 – 31.11.2025	Виконано
3	Дослідження існуючих рішень	01.12.2025 – 31.12.2025	Виконано
4	Постановка вимог до проєктування системи	01.01.2026 – 31.01.2026	Виконано
5	Розробка програмного продукту	01.02.2026 – 03.05.2026	Виконано
6	Тестування	04.05.2026 – 10.05.2026	Виконано
7	Захист програмного продукту	11.05.2026 – 15.05.2026	Виконано
8	Оформлення дипломної роботи	18.05.2026 – 31.05.2026	Виконано
9	Передзахист	01.06.2026 – 05.06.2026	Виконано
10	Захист	15.06.2026 – 19.06.2026	Виконано

Студент

(підпис)

Нікіта ЧУЙКО

(ім'я, ПРІЗВИЩЕ)

Керівник роботи

(підпис)

Вадим КОЛУМБЕТ

(ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

Структура та обсяг дипломної роботи. Робота містить 78 сторінок, 17 рисунків, 1 таблицю, 2 додатки та 21 посилання.

Метою роботи є створення легковисної системи централізованого моніторингу та збору помилок для розподілених додатків екосистеми «ЄКафедра», що дозволить значно знизити середній час відновлення системи після збоїв (MTTR). Для досягнення цієї мети розроблено REST API колектор для прийому інцидентів у реальному часі, впроваджено алгоритм дедуплікації на базі MD5-хешування та створено інтерактивний клієнтський дашборд для візуалізації зібраної аналітики.

Практичне значення одержаних результатів полягає в застосуванні Push-моделі збору даних та алгоритмів криптографічного хешування для оптимізації процесу відстеження помилок у мікросервісних архітектурах з обмеженими апаратними ресурсами. Завдяки інтеграції з хмарним сервісом Google Gemini API реалізовано модуль інтелектуального аналізу, який автоматично генерує рекомендації щодо виправлення багів. Створено готовий до впровадження програмний продукт із вбудованою системою розмежування доступу (RBAC) та багатоканальними сповіщеннями (Telegram, Discord), який суттєво спрощує підтримку надійності та стабільності IT-інфраструктури кафедри.

Ключові слова: розподілені системи, моніторинг помилок, дедуплікація, Node.js, PostgreSQL, штучний інтелект, Gemini API.

ABSTRACT

Structure and scope of the thesis. The work contains 78 pages, 17 figures, 1 table, 2 appendices and 21 references.

The purpose of the work is to create a lightweight centralized error monitoring and collection system for distributed applications of the "e-Kafedra" ecosystem, which will significantly reduce the Mean Time To Recovery (MTTR) after failures. To achieve this goal, a REST API collector for real-time incident reception was developed, an MD5 hashing-based deduplication algorithm was implemented, and an interactive client dashboard for analytics visualization was created.

The practical value of the results obtained lies in the application of the Push data collection model and cryptographic hashing algorithms to optimize the error tracking process in microservice architectures with limited hardware resources. Thanks to the integration with the Google Gemini API cloud service, an intelligent analysis module was implemented that automatically generates bug-fixing recommendations. A ready-to-implement software product was created with a built-in Role-Based Access Control (RBAC) system and multi-channel notifications (Telegram, Discord), which significantly simplifies maintaining the reliability and stability of the department's IT infrastructure.

Keywords: distributed systems, error tracking, deduplication, Node.js, PostgreSQL, artificial intelligence, Gemini API.

ЗМІСТ

	Стор.
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ.....	8
ВСТУП.....	9
1 ПОСТАНОВКА ЗАДАЧІ СТВОРЕННЯ СИСТЕМИ МОНІТОРИНГУ ТА ЗБОРУ ПОМИЛОК.....	11
1.1 Аналіз інфраструктури екосистеми «єКафедра» та актуальність розробки .	11
1.2 Обґрунтування мети та завдань дипломної роботи.....	12
1.3 Функціональні вимоги до програмної системи	13
1.4 Нефункціональні вимоги та критерії якості.....	14
Висновки до розділу 1	15
2 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ДЛЯ МОНІТОРИНГУ ТА ЗБОРУ ПОМИЛОК У РОЗПОДІЛЕНИХ СИСТЕМАХ.....	17
2.1 Огляд промислових систем збору та аналізу логів (Sentry, ELK, Prometheus)	17
2.2 Порівняльний аналіз архітектурних підходів (Pull/Push) та транспортних протоколів	18
2.3 Обґрунтування вибору власного рішення для потреб кафедри	21
Висновки до розділу 2	22
3 ЗАСОБИ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	23
3.1 Обґрунтування вибору платформи Node.js для реалізації серверної частини	23
3.2 Проектування сховища даних на базі реляційної СУБД PostgreSQL.....	25
3.3 Використання бібліотеки React для побудови клієнтського інтерфейсу	25
3.4 Інтеграція хмарного сервісу штучного інтелекту Google Gemini API	27
3.5 Застосування технології контейнеризації Docker для розгортання системи	27
Висновки до розділу 3	28
4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ	30
4.1 Архітектурна модель системи та схема взаємодії компонентів.....	30
4.2 Реалізація алгоритму дедуплікації інцидентів на базі MD5	31
4.3 Розробка модуля інтелектуального аналізу помилок через Gemini API	33
4.4 Розмежування прав доступу на основі моделі RBAC	35
4.5 Організація багатоканальної системи сповіщень	36

4.6 Забезпечення відказостійкості колектора та механізми Rate Limiting	38
Висновки до розділу 4	39
5 РОБОТА КОРИСТУВАЧА ІЗ ПРОГРАМНОЮ СИСТЕМОЮ	40
5.1 Порядок розгортання та конфігурування середовища	40
5.2 Керування проектами та реєстрація нових сервісів	42
5.3 Візуалізація аналітики та інтерфейс моніторингу інцидентів.....	44
5.4 Взаємодія з ШІ-асистентом та системами сповіщень	46
Висновки до розділу 5	47
ВИСНОВКИ.....	49
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	51
ДОДАТОК А Лістинг розробленої системи.....	53
ДОДАТОК Б Презентація.....	71

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

REST (Representational State Transfer) — архітектурна парадигма побудови веб-сервісів, що спирається на використання базових протоколів HTTP для виконання операцій з даними інцидентів.

API (Application Programming Interface) — стандартизований набір інструментів та ендпоінтів, що забезпечує інформаційну взаємодію між незалежними мікросервісами екосистеми та центральним колектором моніторингу.

ELK (Elasticsearch, Logstash, Kibana) — комплексне індустріальне рішення для агрегації та візуалізації журналів подій, що розглядається в роботі як альтернативний, важковаговий підхід до відстеження логів.

JSON (JavaScript Object Notation) — легковажний текстовий формат обміну даними, який є стандартом для формування та передачі payload-повідомлень про інциденти через мережу.

JSONB (JSON Binary) — специфічний бінарний тип даних у реляційній моделі PostgreSQL, який застосовується для ефективного збереження та швидкого індексування ієрархічних метаданих і стек-трейсів.

MD5 (Message-Digest algorithm 5) — алгоритм хешування, який генерує 128-бітне значення і застосовується в архітектурі системи як базовий механізм для ідентифікації та дедуплікації ідентичних програмних винятків.

SPA (Single Page Application) — підхід до розробки фронтенд-додатків, за якого веб-сторінка не перезавантажується повністю, що забезпечує високу швидкість та реактивність інтерфейсу аналітичного дашборду.

ВСТУП

Сучасний етап розвитку інформаційних технологій невіддільний від переходу до мікросервісних архітектур та розподілених програмних систем [11]. Екосистема «єКафедра», що розробляється та впроваджується на кафедрі інженерії програмного забезпечення в енергетиці (ІПЗЕ), є яскравим прикладом такої складної інфраструктури, де відмовостійкість кожного окремого вузла безпосередньо визначає життєздатність та функціональність усієї системи.

Одним із найскладніших викликів при експлуатації сучасних розподілених додатків є забезпечення високого рівня спостережуваності (Observability) згідно з методологією Site Reliability Engineering [2]. Традиційні підходи до аналізу програмних збоїв, що зводяться до ручного перегляду логів в ізольованих Docker-контейнерах, на сьогодні є вкрай неефективними. Вони спричиняють значні часові витрати на локалізацію першопричин (root cause analysis) та суттєво погіршують показник середнього часу відновлення (MTTR), що негативно позначається на загальній надійності. Відповідно, автоматизація процесів агрегації, централізованого зберігання та інтелектуальної візуалізації інцидентів виступає пріоритетним науково-технічним завданням у межах розвитку кафедральної ІТ-інфраструктури.

Об'єктом дослідження є процеси забезпечення надійності, спостережуваності та відказостійкості розподілених програмних систем.

Предметом дослідження є методи, алгоритми та програмні засоби проектування і реалізації легковисної системи централізованого моніторингу на базі платформи Node.js [10].

Головна мета роботи полягає у суттєвому підвищенні стабільності функціонування сервісів екосистеми «єКафедра» шляхом створення програмного продукту NodeWatch, який забезпечує автоматизовану фіксацію інцидентів у реальному часі, їх криптографічну дедуплікацію та інтелектуальний аналіз із залученням сучасних моделей штучного інтелекту [19].

Для досягнення поставленої мети в ході виконання дипломної роботи було проведено комплексне дослідження предметної області та існуючих інструментів моніторингу, що дозволило обґрунтувати вибір оптимального технологічного стеку. До нього увійшли: платформа Node.js для високопродуктивного бекенду [4], реляційна СУБД PostgreSQL, бібліотека React для клієнтського інтерфейсу [6] та технологія контейнеризації Docker.

У процесі проектування особливу увагу було приділено архітектурній моделі системи та структурі бази даних із використанням специфічного типу JSONB [18]. Це стало міцним фундаментом для реалізації алгоритму дедуплікації помилок на базі хешування MD5 [9] та успішної інтеграції хмарного сервісу Google Gemini API [8] для генерації автоматичних рекомендацій з відлагодження коду. Підсумком роботи стала розробка інтерактивної панелі керування (Dashboard) та впровадження системи багатоканальних сповіщень.

Результати проведеної роботи мають безпосередню практичну цінність для системних адміністраторів та розробників кафедри ІПЗЕ. Впровадження системи NodeWatch в інфраструктуру дозволяє не лише мінімізувати час реакції на виникнення критичних помилок, а й забезпечити глибокий проактивний аналіз стану розподіленої системи, що є критично необхідною умовою для підтримки високих стандартів якості програмного забезпечення.

1 ПОСТАНОВКА ЗАДАЧІ СТВОРЕННЯ СИСТЕМИ МОНІТОРИНГУ ТА ЗБОРУ ПОМИЛОК

Експлуатація сучасних програмних рішень вимагає впровадження надійних механізмів контролю їхньої працездатності. У даному розділі проводиться аналіз предметної області, досліджуються проблеми існуючої ІТ-інфраструктури кафедри ІПЗЕ та обґрунтовується необхідність створення власної системи збору інцидентів. На основі виявлених недоліків формується мета дипломної роботи та визначається комплекс завдань, необхідних для її досягнення.

1.1 Аналіз інфраструктури екосистеми «єКафедра» та актуальність розробки

Кафедра інженерії програмного забезпечення в енергетиці (ІПЗЕ) Навчально-наукового інституту атомної та теплової енергетики КПІ ім. Ігоря Сікорського поєднує фундаментальну освітню діяльність із розробкою реальних ІТ-продуктів. Головним об'єктом автоматизації виступає екосистема «єКафедра» — комплексна розподілена система, побудована за принципами мікросервісної архітектури [11]. Інфраструктура даного об'єкта розгорнута на локальних серверних потужностях під керуванням ОС Ubuntu Server, де десятки незалежних мікросервісів функціонують в ізольованих Docker-контейнерах та взаємодіють між собою через REST API.

Незважаючи на архітектурні переваги мікросервісного підходу, такі як гнучкість розгортання та ізольованість компонентів, виникає складний виклик щодо забезпечення наскрізної спостережуваності (Observability) системи [3]. На сьогоднішній день процес виявлення та аналізу програмних збоїв у екосистемі «єКафедра» базується переважно на реактивній моделі. Поточний сценарій обробки інциденту (підхід «As-Is») виглядає наступним чином: адміністратор або розробник дізнається про помилку від кінцевого користувача, після чого змушений

підключатися до сервера через протокол SSH, ідентифікувати проблемний Docker-контейнер та вручну шукати причину збою, переглядаючи неструктуровані текстові логи через консольний інтерфейс.

Такий підхід фундаментально суперечить сучасним інженерним практикам Site Reliability Engineering (SRE), які вимагають проактивного моніторингу та автоматизованого реагування на інциденти [2]. Ручний пошук першопричин (root cause analysis) у розподіленому середовищі призводить до критичного збільшення показника середнього часу відновлення системи (Mean Time To Recovery, MTTR). Оскільки помилка в одному вузлі може бути каскадним наслідком збою в іншому мікросервісі, відсутність єдиної точки агрегації телеметрії унеможливорює швидке формування повного контексту інциденту.

Отже, розробка спеціалізованої системи моніторингу NodeWatch є актуальним та невідкладним завданням. Перехід до цільової моделі управління інфраструктурою (підхід «To-Be») передбачає автоматичне перехоплення програмних винятків на рівні кожного сервісу, миттєву передачу структурованого стек-трейсу до центрального колектора та генерацію проактивного сповіщення. Це дозволить трансформувати процес підтримки програмного забезпечення кафедри з реактивного виправлення наслідків у керований процес безперервного контролю якості [2].

1.2 Обґрунтування мети та завдань дипломної роботи

Головною метою дипломної роботи є підвищення рівня надійності, відказостійкості та стабільності функціонування розподілених додатків екосистеми «єКафедра». Досягнення цієї мети реалізується шляхом проектування та розробки легковисної програмної системи централізованого моніторингу та збору помилок NodeWatch, яка адаптована до специфіки мікросервісної взаємодії та апаратних обмежень IT-інфраструктури кафедри ІПЗЕ.

Відповідно до поставленої мети, у межах дипломної роботи необхідно вирішити наступний комплекс науково-практичних завдань:

1. Дослідити існуючі архітектурні патерни та індустріальні програмні рішення для забезпечення спостережуваності розподілених систем [11].
2. Сформулювати детальні функціональні та нефункціональні вимоги до програмного продукту.
3. Спроекувати архітектурну модель системи, а також концептуальну та логічну схеми реляційної бази даних із застосуванням типу JSONB для зберігання динамічних метаданих [5].
4. Розробити алгоритмічне забезпечення системи, зокрема алгоритм дедуплікації програмних інцидентів на основі криптографічного хешування [9].
5. Програмно реалізувати серверну частину (REST API колектор) на базі середовища Node.js та клієнтський односторінковий додаток (Dashboard) з використанням бібліотеки React.
6. Інтегрувати хмарну платформу Google Gemini API для створення модуля автоматизованої інтелектуальної діагностики причин програмних збоїв [8].
7. Впровадити систему розмежування прав доступу та налаштувати механізми миттєвого сповіщення розробників через сторонні месенджери.

1.3 Функціональні вимоги до програмної системи

Для успішної реалізації цільової моделі управління інфраструктурою необхідно чітко визначити функціональні межі програмного продукту. Спираючись на специфіку екосистеми «єКафедра», до розроблюваної системи моніторингу NodeWatch висувається наступний комплекс функціональних вимог:

а. Прийом та обробка інцидентів: Система повинна забезпечувати безперебійний прийом даних від зовнішніх клієнтських мікросервісів через стандартизований REST API. Вхідні payload-повідомлення мають передаватися у

форматі JSON, що гарантує сумісність із будь-якими технологіями, які використовуються на кафедрі.

б. Збереження даних: Агреговані інциденти повинні нормалізуватися та зберігатися у СУБД PostgreSQL. Для забезпечення гнучкості при роботі з неструктурованими стек-трейсами та змінними оточення вимагається використання бінарного формату JSONB [18].

с. Дедуплікація подій: Для запобігання переповненню бази даних ідентичними записами під час масованих програмних збоїв, система повинна реалізовувати алгоритм дедуплікації на основі MD5-хешування [9].

д. Інтелектуальний аналіз: Вимагається інтеграція з хмарною нейронною мережею Google Gemini API для автоматичної генерації рекомендацій щодо виправлення зафіксованих помилок.

е. Керування доступом: Безпека платформи повинна базуватися на моделі RBAC [13]. Система повинна підтримувати ролі «Адміністратор» (повне керування проектами), «Розробник» (робота з інцидентами та сповіщеннями) та «Гість» (лише перегляд метрик).

ф. Візуалізація та взаємодія: Клієнтська частина повинна бути реалізована як SPA-додаток [14]. Це забезпечить створення сучасного та реактивного UI, що значно покращить загальний UX адміністраторів під час аналізу телеметрії.

г. Система сповіщень: Програмний продукт повинен підтримувати конфігурацію веб-хуків для миттєвої відправки повідомлень про критичні помилки у месенджери (Telegram, Discord) [20].

1.4 Нефункціональні вимоги та критерії якості

Окрім функціональних можливостей, життєво важливим аспектом є забезпечення відповідності програмного продукту технічним та експлуатаційним

обмеженням кафедральної IT-інфраструктури. До ключових нефункціональних вимог належать:

1. Апаратна ефективність: Оскільки колектор помилок розгортатиметься на тих самих серверах, що й основні навчальні сервіси, його архітектура має бути максимально легковисною. Споживання оперативної пам'яті (RAM) компонентами системи не повинно перевищувати 1 ГБ у режимі стандартного навантаження.

2. Продуктивність та пропускна здатність: Серверна частина (колектор) повинна мати високу швидкість обробки вхідних HTTP-запитів. Затримка (latency) при реєстрації інциденту має бути мінімальною, щоб операція логування не блокувала роботу цільових мікросервісів.

3. Безпека та авторизація: Доступ до API та аналітичних дашбордів має бути захищеним. Для перевірки прав доступу без збереження стану сесії на сервері (stateless) вимагається використання криптографічних JWT-токенів. Реєстрація інцидентів від зовнішніх сервісів повинна здійснюватися виключно за умови наявності валідного API-ключа проекту.

4. Портативність: Усі компоненти системи (колектор, база даних, веб-інтерфейс) повинні бути контейнеризовані за допомогою Docker для забезпечення швидкого розгортання та ізоляції середовищ [21].

Висновки до розділу 1

У першому розділі проведено детальний аналіз предметної області та досліджено поточний стан IT-інфраструктури екосистеми «е-Кафедра». У результаті дослідження виявлено, що стрімке масштабування та зростання кількості незалежних мікросервісів неминуче призводить до критичного ускладнення процесів виявлення та локалізації програмних збоїв. Встановлено, що відсутність єдиних централізованих інструментів спостережуваності (observability) та використання застарілих підходів, орієнтованих на ручний аналіз розрізнених логів, прямо суперечить сучасним методологіям Site Reliability Engineering (SRE).

Це, у свою чергу, спричиняє неприпустиме збільшення середнього часу відновлення працездатності систем (MTTR) та знижує загальну надійність програмного комплексу.

З метою ефективного вирішення виявленої проблематики обґрунтовано нагальну необхідність проектування та створення спеціалізованої, апаратно-ефективної системи моніторингу та збору помилок NodeWatch. У межах даного розділу чітко сформовано мету дипломної роботи, а також визначено ключові науково-практичні завдання, необхідні для її досягнення. Окрім цього, розроблено та систематизовано вичерпний перелік функціональних і нефункціональних вимог до майбутнього програмного продукту.

Доведено, що практична реалізація висунутих вимог — зокрема, впровадження алгоритмів криптографічної дедуплікації інцидентів, інтеграція модуля інтелектуального аналізу помилок за допомогою штучного інтелекту та розробка клієнтської частини за архітектурою SPA (Single Page Application) — забезпечить якісний перехід процесу технічного обслуговування кафедральних ІТ-сервісів від застарілої моделі реактивного реагування до стратегії проактивного моніторингу та превентивного усунення вразливостей.

2 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ДЛЯ МОНІТОРИНГУ ТА ЗБОРУ ПОМИЛОК У РОЗПОДІЛЕНИХ СИСТЕМАХ

2.1 Огляд промислових систем збору та аналізу логів (Sentry, ELK, Prometheus)

На сьогоднішній день ринок програмного забезпечення для забезпечення спостережуваності (Observability) пропонує декілька потужних індустріальних стандартів. Найбільш розповсюдженими серед них є Sentry, стек ELK та система Prometheus. Кожен із цих інструментів вирішує специфічний клас завдань та має власні архітектурні обмеження.

Система Sentry спеціалізується безпосередньо на відстеженні програмних винятків та помилок у реальному часі. Вона надає розробникам деталізовані стек-трейси, дані про оточення та контекст виконання коду, що робить її потужним інструментом для комерційних проєктів [15]. Однак використання Sentry у межах локальної інфраструктури кафедри ІПЗЕ є вкрай ускладненим. Головною перешкодою виступають високі вимоги до апаратних ресурсів при розгортанні локальної (Self-hosted) версії, яка потребує значних обсягів оперативної пам'яті для підтримки своїх внутрішніх мікросервісів.

Іншим популярним рішенням є стек ELK, який поєднує можливості Elasticsearch, Logstash та Kibana. Ця комбінація інструментів дозволяє збирати, індексувати та візуалізувати будь-які типи логів у величезних масштабах [16]. Проте, незважаючи на безпрецедентну гнучкість, стек ELK не є вузькоспеціалізованою платформою саме для відстеження помилок (Error Tracking). Його конфігурація вимагає побудови складних пайплайнів для парсингу стек-трейсів. Крім того, ресурсоємність пошукового рушія робить його практично непридатним для розгортання на легковажних кафедральних серверах.

Система Prometheus, у свою чергу, є стандартом де-факто для збору часових рядів та метрик продуктивності [17]. Вона ідеально підходить для моніторингу завантаження процесора чи пам'яті, але не призначена для агрегації об'ємних текстових даних, якими є звіти про програмні інциденти.

На рисунку 2.1 наведено порівняння споживання оперативної пам'яті різними інструментами моніторингу, що наочно ілюструє доцільність розробки легковажної альтернативи.

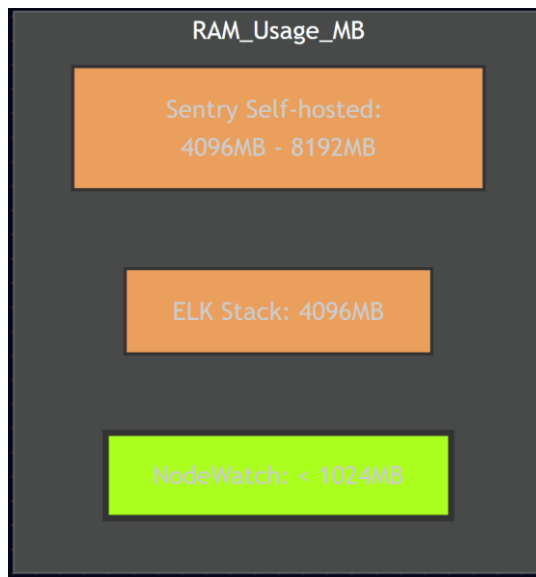


Рисунок 2.1 – Діаграма порівняння мінімальних системних вимог (RAM) для Sentry та NodeWatch

Як видно з наведеної діаграми, індустріальні рішення є надто важкими для поточної IT-інфраструктури, де кожен мегабайт пам'яті є критичним для підтримки функціонування основних сервісів екосистеми.

2.2 Порівняльний аналіз архітектурних підходів (Pull/Push) та транспортних протоколів

При проектуванні систем моніторингу ключовим архітектурним рішенням є вибір вектора передачі телеметрії. Зазвичай розглядають два фундаментальні підходи: Pull-модель та Push-модель.

У Pull-моделі (яку використовує, наприклад, Prometheus) система моніторингу самостійно звертається до цільових сервісів через певні задані проміжки часу для опитування їхнього стану. Цей підхід є високоефективним для збору статичних метрик, проте він є абсолютно непридатним для фіксації випадкових програмних винятків. Оскільки помилка може статися і зникнути між циклами опитування, існує висока ймовірність незворотної втрати критичних даних про інцидент.

Натомість Push-модель передбачає принципово інший механізм: клієнтський додаток самостійно ініціює відправку даних до централізованого колектора рівно в момент виникнення помилки. Це дозволяє миттєво фіксувати повний контекст збою (стан змінних, послідовність викликів функцій), що кардинально знижує показник MTTR під час відлагодження системи.

Схематичне порівняння архітектурної взаємодії компонентів при використанні різних моделей наведено на рисунку 2.2.

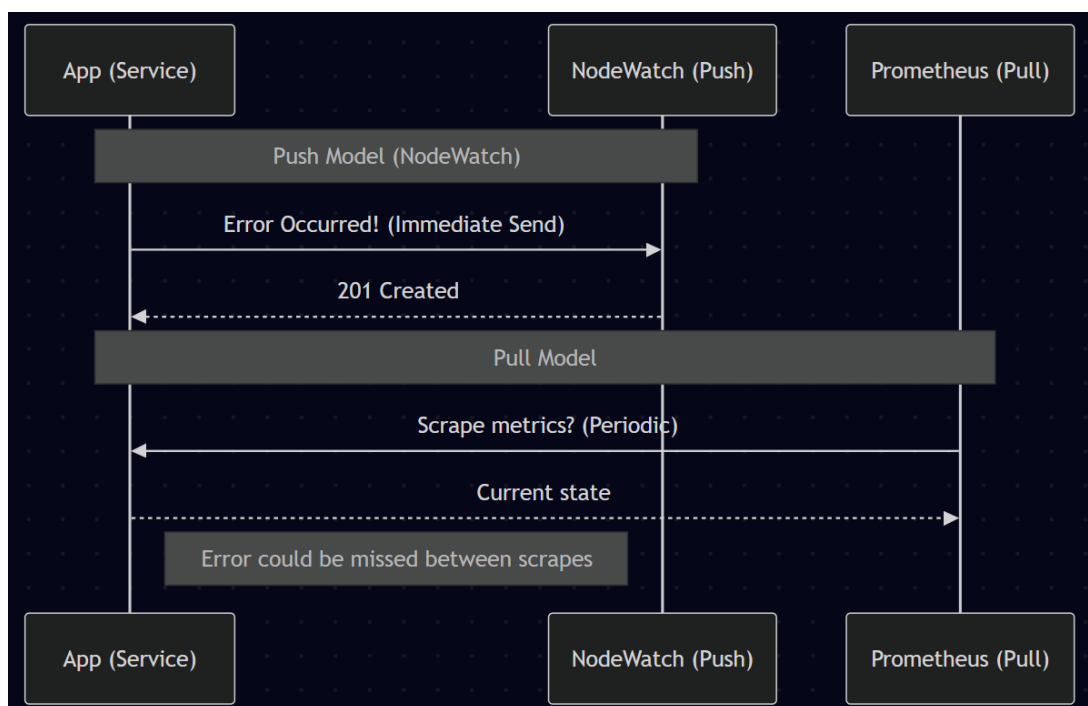


Рисунок 2.2 – Порівняльна схема взаємодії компонентів при використанні Pull та Push моделей моніторингу

Окрім вектора передачі, критичну роль відіграє вибір транспортного протоколу. Враховуючи гетерогенність мікросервісів у екосистемі «єКафедра», централізована архітектура з використанням Push-моделі була посилена застосуванням архітектурного стилю REST. Використання базових протоколів HTTP та формату обміну даними JSON забезпечує універсальну сумісність [12]. Будь-який новий мікросервіс може бути легко інтегрований із системою моніторингу шляхом простої відправки POST-запиту на відповідний API-ендпоінт колектора.

Узагальнений порівняльний аналіз можливостей промислових систем та проектованого рішення наведено нижче (табл. 2.1).

Таблиця 2.1 — Порівняльна характеристика функціональних можливостей систем моніторингу

Характеристика	Sentry	ELK Stack	NodeWatch
Спеціалізація	Error Tracking	Log Management	Error Tracking & AI Analysis
Ресурсомісткість	Висока (4-8 ГБ RAM)	Дуже висока (8+ ГБ RAM)	Низька (<1 ГБ RAM)
Складність розгортання	Складна (багато компонентів)	Дуже складна	Легка (Docker Compose)
Інтеграція з ШІ	Обмежена (платна)	Відсутня	Вбудована (Gemini API)
Вартість	Безкоштовно (Self-hosted)	Безкоштовно (Open Source)	Безкоштовно/ Власна розробка

Як демонструє таблиця, промислові аналоги значно програють у контексті ресурсомісткості та безкоштовної інтеграції інструментів штучного інтелекту, що робить розробку кастомного рішення найбільш доцільним кроком.

2.3 Обґрунтування вибору власного рішення для потреб кафедри

Вибір на користь розробки власної системи NodeWatch зумовлений нагальною необхідністю досягнення жорсткого балансу між широкою функціональністю та ефективним використанням апаратних ресурсів. Як показав попередній аналіз, існуючі промислові аналоги є надто важкими для поточної ІТ-інфраструктури кафедри ІПЗЕ, де абсолютний пріоритет віддається стабільній роботі основних навчальних сервісів.

Розробка кастомного рішення на базі платформи Node.js [10] дозволяє створити максимально легковажний сервіс. Це дає можливість розгортати колектор NodeWatch на тому ж обладнанні, де працюють інші мікросервіси екосистеми, не створюючи при цьому суттєвого навантаження на серверні потужності.

Крім того, власна розробка дозволяє впровадити специфічні інноваційні функції, які або повністю відсутні, або є платними Enterprise-доповненнями у комерційних продуктах. Фундаментальною перевагою проектованої системи є інтеграція з хмарним сервісом штучного інтелекту Google Gemini API [8, 19], що перетворює NodeWatch зі звичайного агрегатора логів на повноцінного автоматизованого асистента розробника. Модуль інтелектуального аналізу здатний не лише фіксувати факт збою, але й пояснювати технічну причину виникнення помилки, пропонуючи конкретні ділянки коду для відлагодження.

Додатковою перевагою є можливість розробки гнучкої системи сповіщень із безшовною інтеграцією у робочі месенджери адміністраторів (наприклад, за допомогою Telegram Bot API [20]). Таким чином, NodeWatch стає вузькоспеціалізованим інструментом, що ідеально враховує специфіку

мікросервісної архітектури [11], вимоги до ресурсоефективності та гостру потребу у використанні сучасних технологій штучного інтелекту для зменшення показника MTTR.

Висновки до розділу 2

У другому розділі було проведено комплексний аналіз існуючих підходів та індустріальних рішень для моніторингу та збору помилок у розподілених програмних системах. Дослідження популярних платформ, таких як Sentry, стек **ELK** та Prometheus, показало, що незважаючи на їхню функціональну потужність, вони є надлишковими, ресурсомісткими та складними в адмініструванні для масштабів локальної інфраструктури кафедри.

Порівняльний аналіз архітектурних моделей збору телеметрії підтвердив, що для гарантованої фіксації випадкових програмних винятків найбільш ефективним є використання **Push-моделі** у поєднанні з архітектурною парадигмою **REST**. Це дозволяє клієнтським додаткам миттєво відправляти контекст помилки до централізованого колектора, мінімізуючи ризик втрати критично важливих даних між циклами опитування.

На основі проведеного дослідження було повністю обґрунтовано рішення щодо розробки власного програмного продукту NodeWatch. Створення кастомної системи дозволяє досягти оптимального балансу між низьким споживанням апаратних ресурсів та наявністю інноваційних функцій, таких як інтелектуальний аналіз інцидентів за допомогою нейронних мереж та миттєва багатоканальна відправка сповіщень. Це повністю підтверджує доцільність обраного технічного шляху для подальшої програмної реалізації завдань дипломної роботи.

3 ЗАСОБИ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У даному розділі наведено обґрунтування вибору технологічного стеку, використаного для реалізації розроблюваної системи моніторингу. Детально розглядаються архітектурні особливості обраних інструментів, їхня сумісність із ресурсообмеженим середовищем екосистеми «єКафедра» та ефективність при вирішенні завдань агрегації та візуалізації програмних інцидентів.

3.1 Обґрунтування вибору платформи Node.js для реалізації серверної частини

Вибір програмного середовища для бекенд-частини (колектора інцидентів) є критичним етапом проектування, оскільки саме цей компонент бере на себе основне мережеве навантаження під час масованих збоїв у цільових мікросервісах. Для програмної реалізації серверної логіки NodeWatch було обрано платформу Node.js.

Цей вибір обумовлений високою ефективністю середовища при обробці великої кількості одночасних мережевих з'єднань [10]. В основі архітектури платформи лежить асинхронна неблокуюча модель вводу-виводу та подійний цикл (Event Loop), який дозволяє виконувати операції читання-запису без створення окремих апаратних потоків для кожного вхідного запиту (рис. 3.1).

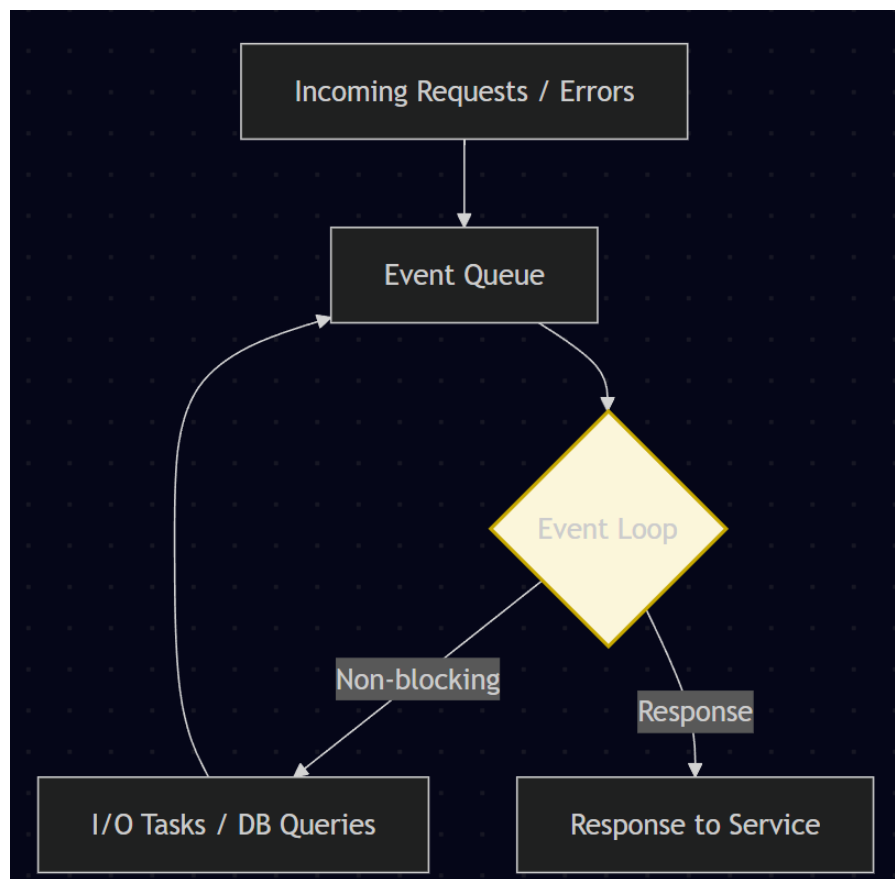


Рисунок 3.1 – Схема роботи подійного циклу (Event Loop) у платформі Node.js

Такий архітектурний патерн є критично важливим для колектора помилок, який повинен миттєво приймати та обробляти вхідні HTTP-запити від десятків мікросервісів, не блокуючи при цьому основний потік виконання інструкцій [4]. Крім того, використання високопродуктивного рушія V8 забезпечує швидку компіляцію та виконання коду, що в поєднанні з багатою екосистемою пакетних рішень дозволяє оперативно впроваджувати необхідний функціонал для обробки мережових пакетів.

3.2 Проектування сховища даних на базі реляційної СУБД PostgreSQL

Організація надійного та ефективного рівня персистенції (збереження даних) є наступною фундаментальною задачею. Для збереження нормалізованих звітів про помилки було обрано реляційну СУБД PostgreSQL. На перший погляд, жорстка реляційна схема таблиць є неефективною для логування, оскільки кожна програмна помилка генерує унікальний набір метаданих (різні версії бібліотек, специфічні заголовки запитів, динамічний стан оточення).

Проте основним критерієм вибору на користь PostgreSQL стала нативна підтримка вдосконаленого бінарного типу даних JSONB [18]. Цей тип дозволяє зберігати неструктуровані ієрархічні дані про інциденти безпосередньо у реляційних таблицях, забезпечуючи гнучкість, притаманну NoSQL-рішенням. Водночас система зберігає всі переваги транзакційної цілісності на рівні бази даних, що виключає можливість втрати звітів під час їх паралельного запису.

Використання формату JSONB надає можливість створювати спеціалізовані інвертовані індекси для швидкого повнотекстового пошуку за глибоко вкладеними полями об'єктів помилок [5]. Це суттєво оптимізує процеси дедуплікації та фільтрації інцидентів при обробці аналітичних запитів від клієнтського інтерфейсу.

3.3 Використання бібліотеки React для побудови клієнтського інтерфейсу

Для забезпечення зручної візуалізації зібраної аналітики та керування конфігураціями проектів було розроблено інтерактивну панель керування (Dashboard). Розробка клієнтської частини системи моніторингу була реалізована з використанням бібліотеки React, що дозволило створити сучасний та швидкий SPA-додаток [14].

Головною перевагою вибору React є його компонентний підхід до побудови інтерфейсу. Дана архітектура дозволяє розділити складний логічний шар аналітичної панелі на ізольовані елементи, такі як графіки статистики, списки інцидентів, модальні вікна та форми налаштувань (рис. 3.2).

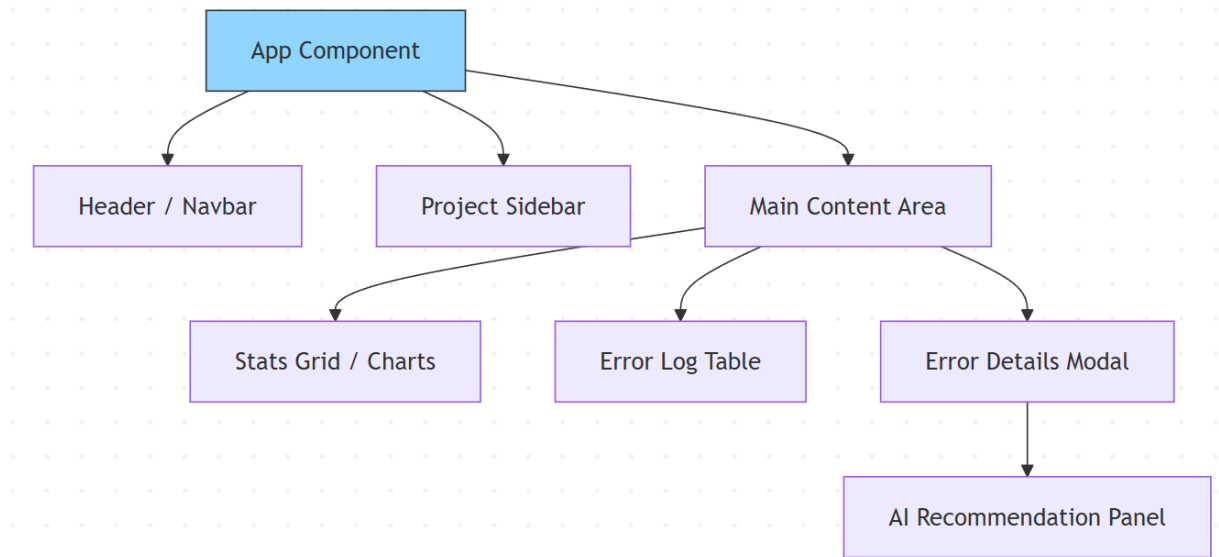


Рисунок 3.2 – Структура компонентної моделі фронтенд-додатка на базі React

Така компонентна декомпозиція не лише значно спрощує підтримку та масштабування програмного коду, але й дозволяє повторно використовувати елементи UI на різних сторінках платформи.

Крім того, використання механізму Virtual DOM забезпечує високу швидкість оновлення інтерфейсу за рахунок мінімізації прямих ресурсомістких маніпуляцій з реальним деревом об'єктів браузера [6]. Це є особливо критичним фактором для забезпечення плавного UX під час відображення потоку даних про помилки, який оновлюється у режимі реального часу.

3.4 Інтеграція хмарного сервісу штучного інтелекту Google Gemini API

Інноваційною складовою проекрованої системи моніторингу є впровадження модуля інтелектуальної діагностики на базі хмарного сервісу Google Gemini API. Традиційні системи збору логів обмежуються лише наданням сирих даних про збій, залишаючи процес пошуку причини на розсуд розробника. Однак сучасні великі мовні моделі (LLM) демонструють надзвичайно високу ефективність у задачах синтаксичного та семантичного аналізу програмного коду [19].

При реєстрації нового унікального інциденту система NodeWatch автоматично формує структурований контекстний запит, який містить технічні метадані та деталізований стек-трейс помилки. Цей запит передається до нейронної мережі через офіційний API платформи [8]. Штучний інтелект проводить глибокий аналіз наданої інформації, виявляє логічні хиби у послідовності викликів функцій та генерує структуровану пораду щодо виправлення багу.

Такий архітектурний крок трансформує систему зі звичайного пасивного агрегатора на повноцінного автоматизованого експерта-помічника. Це кардинально знижує когнітивне навантаження на інженерів кафедри, оскільки вони отримують готові гіпотези та шляхи вирішення проблеми безпосередньо у вікні аналітики.

3.5 Застосування технології контейнеризації Docker для розгортання системи

Для забезпечення надійної ізоляції середовища виконання та стандартизації процесу розгортання в IT-інфраструктурі кафедри ІПЗЕ було застосовано технологію контейнеризації Docker. Цей підхід гарантує, що розроблений програмний продукт функціонуватиме абсолютно ідентично як на локальній машині розробника під час тестування, так і на продуктовому сервері під

управлінням Ubuntu Server [21]. Завдяки контейнеризації вдається повністю уникнути проблеми конфліктів версій системних залежностей.

Управління багатоконтейнерною архітектурою здійснюється за допомогою інструменту оркестрації Docker Compose. Даний інструмент дозволяє описати всі компоненти інфраструктури у вигляді єдиного конфігураційного файлу [7]. Завдяки цьому колектор на базі Node.js, база даних PostgreSQL та фронтенд-сервер Nginx об'єднуються у спільну віртуальну ізольовану мережу.

Схему розгортання та взаємодії контейнеризованих компонентів системи наведено на рисунку 3.3.

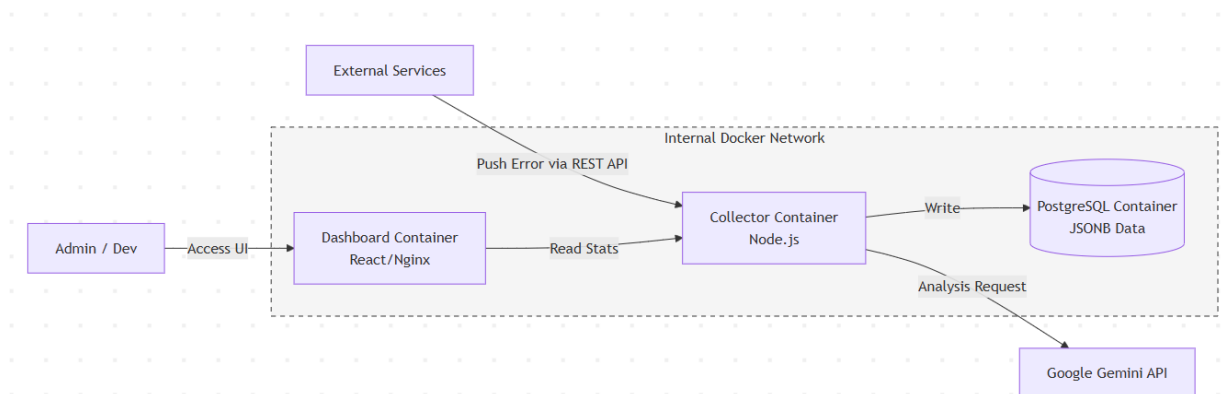


Рисунок 3.3 – Діаграма розгортання системи NodeWatch за допомогою Docker Compose

Така архітектура розгортання дозволяє інтегрувати NodeWatch поруч з існуючими сервісами екосистеми «єКафедра», мінімізуючи витрати часу на налаштування серверного оточення та забезпечуючи високий рівень портативності програмного рішення.

Висновки до розділу 3

У третьому розділі було проведено обґрунтування вибору технологічного стеку, необхідного для реалізації легковисної системи централізованого моніторингу. Кожен обраний інструмент вирішує специфічний клас інженерних

завдань, що в сукупності забезпечує високу продуктивність та надійність розробленого програмного продукту.

Використання серверної платформи Node.js дозволило створити асинхронний неблокуючий колектор, здатний витримувати значні мережеві навантаження під час масованих збоїв клієнтських додатків. Застосування реляційної СУБД PostgreSQL у поєднанні з бінарним типом даних JSONB успішно вирішило проблему ефективного та безпечного збереження неструктурованих метаданих інцидентів.

Клієнтська частина системи, побудована на базі бібліотеки React, гарантує створення реактивного SPA-додатка, що забезпечує комфортний UX для адміністраторів завдяки компонентній декомпозиції та використанню Virtual DOM. Доведено високу доцільність інтеграції хмарного сервісу Google Gemini API, що дозволило впровадити функціонал інтелектуальної діагностики та автоматичної генерації порад з відлагодження коду. Крім того, застосування технології Docker забезпечило швидке, ізольоване та портативне розгортання всіх компонентів системи в інфраструктурі кафедри.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

У даному розділі детально розглядається архітектурна та алгоритмічна складова розробленої системи NodeWatch. Описуються принципи взаємодії внутрішніх компонентів, логічна структура збереження даних, а також наводяться схеми програмної реалізації ключових інноваційних модулів, зокрема механізму криптографічної дедуплікації та підсистеми інтелектуального аналізу програмних інцидентів.

4.1 Архітектурна модель системи та схема взаємодії компонентів

Програмна реалізація системи моніторингу базується на класичній багаторівневій мікросервісній архітектурі, що дозволяє забезпечити високий рівень масштабованості та ізоляції процесів [11]. Архітектурно екосистема NodeWatch поділяється на три фундаментальні рівні: рівень агрегації даних (Collector), рівень персистенції (база даних) та рівень візуалізації (Dashboard).

Основний компонент системи — Collector — реалізований як високопродуктивний HTTP-сервер. Він надає уніфіковані REST ендпоінти, через які зовнішні клієнтські мікросервіси екосистеми «єКафедра» передають інформацію про виникнення критичних помилок. Використання стандартизованого архітектурного стилю взаємодії гарантує, що система моніторингу не залежить від мови програмування клієнтського додатка [12].

Процес обробки вхідного запиту будується за принципом конвеєра (Pipeline). Кожен вхідний пакет даних спочатку проходить етап валідації, де перевіряється наявність обов'язкових полів (назва проекту, рівень критичності, текстове повідомлення та детальний стек-трейс).

Для збереження валідованих даних використовується СУБД PostgreSQL. Концептуальна схема бази даних включає три основні сутності: Projects (реєстр

підключених мікросервісів), Users (облікові записи адміністраторів) та Issues (унікальні помилки). Ключовою архітектурною особливістю логічної схеми є використання бінарного типу JSONB у таблиці інцидентів [5]. Це дозволяє динамічно зберігати контекст виконання коду (наприклад, змінні середовища або специфічні заголовки запитів) без необхідності жорстко декларувати ці колонки під час створення таблиць.

Загальна схема структурної взаємодії компонентів та напрямки потоків даних відображені на відповідній діаграмі (рис. 4.1).

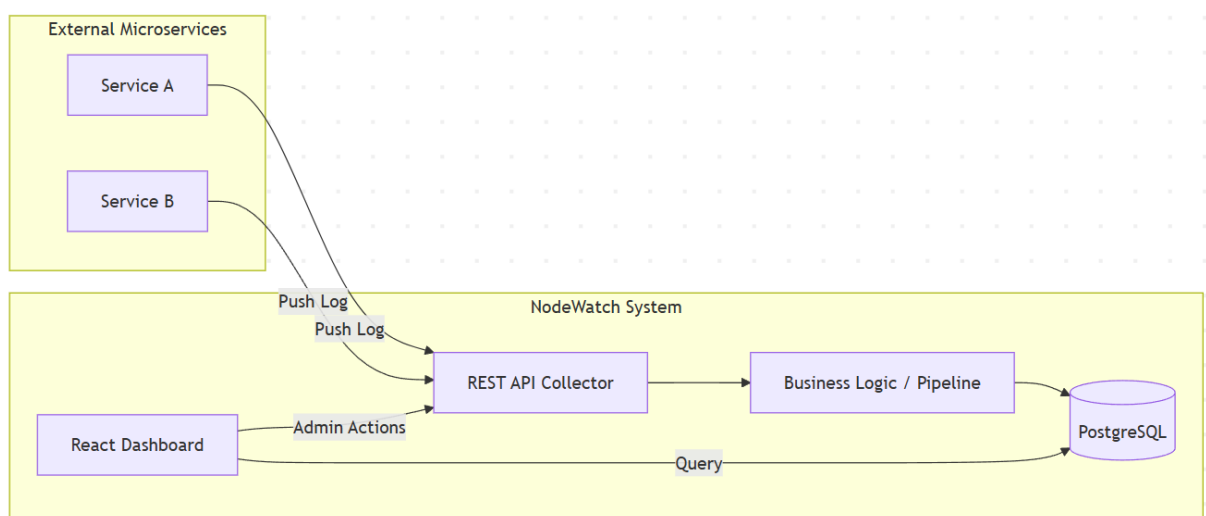


Рисунок 4.1 – Схема архітектурної взаємодії компонентів системи NodeWatch

Така ізольована побудова гарантує, що навіть у випадку критичного навантаження на рівень візуалізації, процес збору телеметрії від кафедральних мікросервісів функціонуватиме безперебійно, забезпечуючи високу відказостійкість усієї системи моніторингу.

4.2 Реалізація алгоритму дедуплікації інцидентів на базі MD5

Однією з найпоширеніших проблем у розподілених архітектурах є ефект «лавини помилок», коли збій базового сервісу провокує тисячі ідентичних

циклічних винятків у залежних вузлах. Для запобігання надмірному розростанню бази даних та уникнення інформаційного перевантаження адміністраторів, у системі NodeWatch реалізовано алгоритм жорсткої дедуплікації.

Програмна реалізація цього модуля базується на генерації унікального криптографічного хеш-ідентифікатора (fingerprint) для кожного вхідного інциденту за допомогою алгоритму MD5 [9]. Логіка формування хешу полягає у конкатенації ключових параметрів помилки: ідентифікатора проекту, типу виключення (наприклад, `TypeError` або `DatabaseConnectionError`) та нормалізованого тексту стек-трейсу. Нормалізація передбачає програмне видалення динамічних значень (таких як випадкові таймстампи або ідентифікатори сесій), що гарантує генерацію однакового хешу для ідентичних за своєю природою помилок, навіть якщо вони сталися у різний час.

При надходженні нового повідомлення система обчислює його MD5-суму та виконує пошук у базі даних. Якщо інцидент із таким ідентифікатором вже існує, замість створення нового запису колектор лише інкрементує лічильник подій (occurrences) та оновлює час останнього виникнення помилки (last_seen). Блок-схему даного алгоритмічного процесу наведено нижче (рис. 4.2).

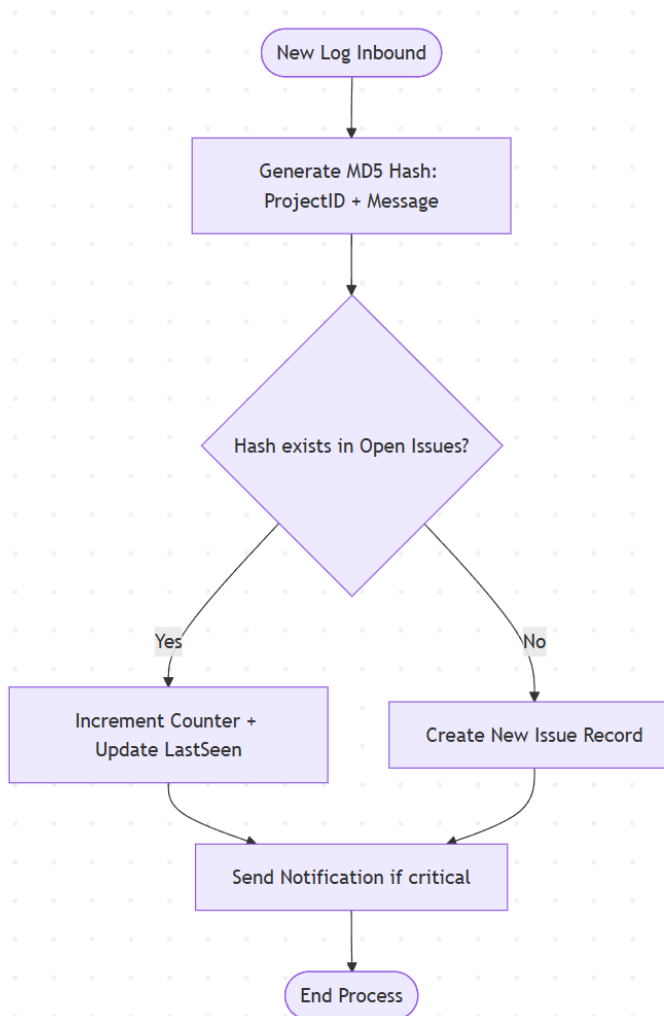


Рисунок 4.2 – Блок-схема алгоритму дедуплікації інцидентів

Такий підхід кардинально економить дисковий простір серверів та значно спрощує роботу розробника, оскільки він бачить одну зведену картку проблеми зі статистикою її частотності, а не розрізнені тисячі логів.

4.3 Розробка модуля інтелектуального аналізу помилок через Gemini API

Інноваційною складовою платформи є використання великих мовних моделей для автоматизації рутинних процесів відлагодження коду. Реалізація цього модуля передбачає асинхронну взаємодію між ядром NodeWatch та хмарною нейронною мережею Google Gemini через її захищений API [8].

Процес ініціюється автоматично в момент реєстрації нового унікального інциденту. Програмний модуль формує спеціалізований структурований промпт (prompt), який містить технічну інструкцію для штучного інтелекту, назву мови програмування, з якої надійшов лог, та безпосередньо сам стек-трейс помилки. Згенерований запит відправляється до сервісу Gemini, де неймережа проводить глибокий семантичний аналіз тексту, виявляє ймовірну причину збою на рівні логіки коду та повертає текстову рекомендацію щодо її усунення.

Логіку інформаційного обміну між компонентами під час інтелектуального аналізу відображено на схемі (рис. 4.3).

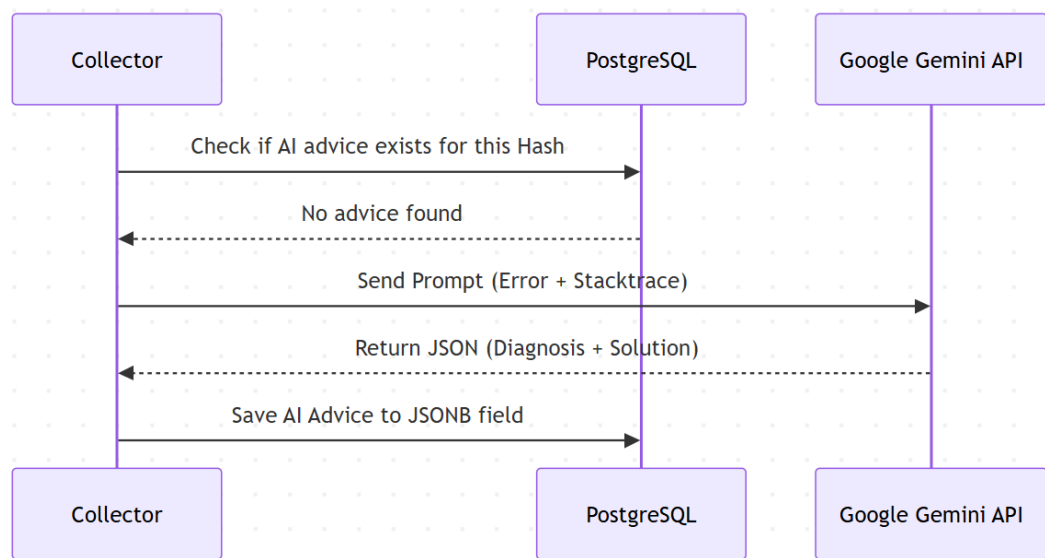


Рисунок 4.3 – Схема інтеграції та передачі даних до Google Gemini API

Отриманий результат зберігається в окремому текстовому полі бази даних і миттєво відображається у SPA-додатку адміністратора на сторінці деталей інциденту. Це дозволяє розробникам кафедри отримати експертну підказку щодо вирішення проблеми за лічені секунди, оптимізуючи загальні витрати часу на підтримку працездатності екосистеми.

4.4 Розмежування прав доступу на основі моделі RBAC

Безпека доступу до даних та захист конфігурацій є критично важливими вимогами для будь-якої корпоративної системи моніторингу. Оскільки з панеллю керування NodeWatch можуть працювати співробітники кафедри з різними посадовими обов'язками, програмна реалізація клієнт-серверної взаємодії була посилена системою Role-Based Access Control (RBAC) [13].

Ця модель безпеки базується на жорсткому розмежуванні прав користувачів на рівні проміжного програмного забезпечення (middleware) серверного API. Під час кожного запиту до сервера система перевіряє криптографічний маркер сесії та визначає рівень повноважень ініціатора. У межах розробленої системи виділено три ієрархічні рівні доступу:

1. **Адміністратор (Admin):** має повний контроль (Full Access) над системою. Дозволено створювати нові проекти, генерувати ключі доступу для мікросервісів, а також керувати обліковими записами інших користувачів.

2. **Розробник (Developer):** має права на читання та запис (Read/Write) у межах призначених проектів. Дозволено переглядати деталі інцидентів, змінювати їхній статус (наприклад, на «Вирішено»), взаємодіяти з модулем штучного інтелекту та налаштовувати канали сповіщень.

3. **Гість (Guest):** має права виключно на читання (Read Only). Доступ обмежений переглядом загальної статистики на дашборді без можливості перегляду конфіденційних стек-трейсів чи зміни конфігурацій.

Ієрархічну схему управління правами доступу та розподіл зон відповідальності користувачів наведено на рисунку 4.4.

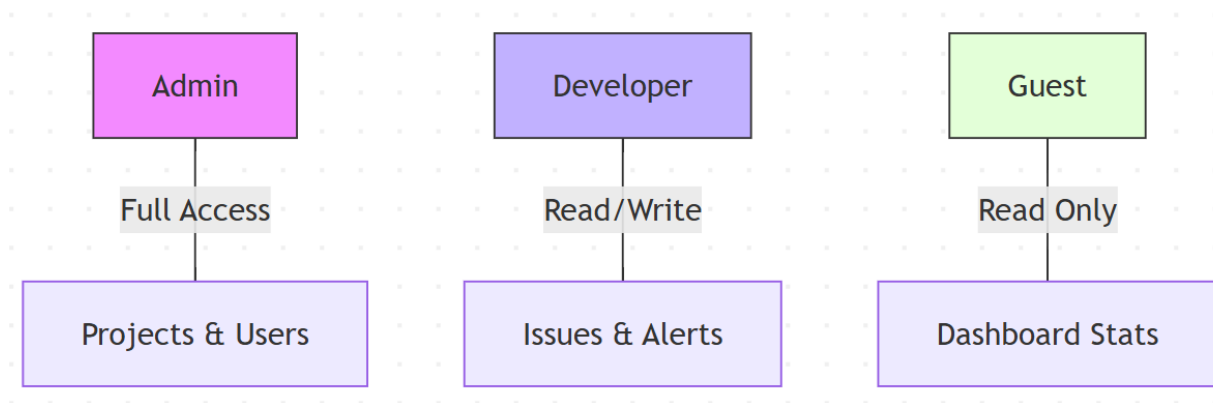


Рисунок 4.4 – Схема управління правами доступу на основі моделі RBAC

Таке розмежування гарантує цілісність даних, захищає конфігурації екосистеми «єКафедра» від несанкціонованих або випадкових змін та є необхідною умовою для безпечного корпоративного використання програмного продукту [13].

4.5 Організація багатоканальної системи сповіщень

Критично важливою вимогою до сучасних систем моніторингу є забезпечення мінімальної затримки між моментом виникнення програмного збою та початком його усунення. Для реалізації цієї вимоги у системі NodeWatch розроблено модуль проактивного багатоканального сповіщення (Alerting System), який працює як асинхронний фоновий процес.

Основною платформою для доставки сповіщень розробникам кафедри ІІЗЕ було обрано месенджер Telegram, інтеграція з яким здійснюється за допомогою офіційного Bot API [20]. Програмна логіка модуля передбачає перехоплення події створення нового інциденту в базі даних та автоматичне формування структурованого HTTP POST-запиту до серверів месенджера.

Фрагмент програмного коду, що демонструє логіку формування та відправки HTTP-запиту з використанням веб-хуків, наведено на рисунку 4.5.

```

async function sendTelegramAlert(chatId, projectName, log) {
  const token = configService.get('telegram_bot_token');
  if (!token) return;

  const emoji = log.level === 'critical' ? '🔴' :
    log.level === 'error' ? '🔴' :
    log.level === 'warn' ? '🟡' : '🟢';

  const dashboardUrl = configService.get('dashboard_url', 'http://localhost:5173');
  const incidentUrl = `${dashboardUrl}/incidents/${log.id}`;

  const message = `${emoji} *NodeWatch Alert*\n*Project:* ${projectName}\n*Level:*  

  ${log.level.toUpperCase()}\n\n*Message:*${log.message}\n\n[🔗  

  View Incident in Dashboard](${incidentUrl})`;

  try {
    const url = `https://api.telegram.org/bot${token}/sendMessage`;
    const response = await fetch(url, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        chat_id: chatId,
        text: message,
        parse_mode: 'Markdown'
      })
    });

    if (!response.ok) {
      throw new Error(`Telegram API responded with ${response.status}  

      ${response.statusText}`);
    }
  } catch (err) {
    console.error(`🔴 Telegram alert failed for chat ${chatId}:`, err.message);
  }
}

```

Рисунок 4.5 – Лістинг функції генерації та відправки сповіщень через Telegram Bot API

Як видно з наведеного лістингу, алгоритм форматує вихідне повідомлення з використанням синтаксису Markdown або HTML. Сповідження містить ключову інформацію: назву мікросервісу, де стався збій, рівень критичності (наприклад, Fatal або Error), короткий опис проблеми та згенероване гіперпосилання. Це посилання веде безпосередньо на детальну сторінку інциденту у веб-інтерфейсі (Dashboard), що позбавляє адміністратора необхідності шукати помилку вручну.

Для запобігання ефекту «втоми від сповіщень» (alert fatigue), модуль обладнано системою фільтрації. Розробник має змогу вказати в налаштуваннях проекту мінімальний поріг критичності, при досягненні якого система ініціюватиме відправку повідомлення, ігноруючи при цьому незначні попередження (Warnings) [2].

4.6 Забезпечення відказостійкості колектора та механізми Rate Limiting

Оскільки колектор NodeWatch виступає централізованою точкою агрегації телеметрії для всієї екосистеми «єКафедра», він стає потенційно вразливим до каскадних відмов. У випадку критичного збою в інфраструктурі (наприклад, падіння спільної бази даних), десятки мікросервісів можуть одночасно почати генерувати тисячі повідомлень про помилки щосекунди. Це створює ефект ненавмисної DDoS-атаки на REST API моніторингу.

Для забезпечення високої доступності (High Availability) та захисту серверної платформи Node.js від вичерпання ресурсів, у програмну архітектуру було інтегровано механізми обмеження частоти запитів (Rate Limiting) [10]. Даний функціонал реалізовано на рівні проміжного програмного забезпечення (middleware).

Алгоритм працює за принципом «маркерного кошика» (Token Bucket): для кожного унікального API-ключа проекту встановлюється ліміт на кількість запитів за певне вікно часу (наприклад, не більше 100 інцидентів на хвилину). Якщо мікросервіс перевищує цей ліміт, колектор негайно припиняє обробку його корисного навантаження і повертає клієнту HTTP-статус 429 Too Many Requests.

Додатковим рівнем захисту є жорстке обмеження розміру вхідного JSON-пакета (Payload Size Limit) до 2 МБ. Це унеможливорює ситуацію, коли злоумисник або некоректно налаштований сервіс спробує відправити гігантський стек-трейс,

що могло б призвести до переповнення оперативної пам'яті (RAM) на сервері та аварійної зупинки подійного циклу (Event Loop) [4].

Висновки до розділу 4

У четвертому розділі було здійснено глибокий опис архітектурних рішень та алгоритмічної бази програмного продукту NodeWatch. Реалізована багаторівнева мікросервісна архітектура з використанням REST-інтерфейсів забезпечила надійну та ізольовану взаємодію між кафедральними сервісами та центральним колектором. Вибір реляційної СУБД PostgreSQL із типом даних JSONB довів свою ефективність для збереження неструктурованих стек-трейсів із підтримкою транзакційної цілісності.

Ключовим досягненням у площині оптимізації стало впровадження криптографічного алгоритму дедуплікації на базі MD5, який ефективно усуває циклічне дублювання помилок. Вагому наукову новизну забезпечила успішна інтеграція модуля штучного інтелекту на основі Google Gemini API, що дозволило автоматизувати складний процес семантичного аналізу збоїв та генерації порад для розробників.

Безпека та операційність системи були суттєво підвищені за рахунок впровадження моделі контролю доступу RBAC та багатоканальної системи сповіщень через Telegram. Завершальним етапом проектування стала розробка захисних механізмів Rate Limiting, які гарантують відказостійкість колектора навіть в умовах аномальних пікових навантажень. Комплексна реалізація цих інноваційних підходів підтверджує готовність системи NodeWatch до повноцінного впровадження у робоче середовище екосистеми кафедри.

5 РОБОТА КОРИСТУВАЧА ІЗ ПРОГРАМНОЮ СИСТЕМОЮ

У даному розділі наведено детальну інструкцію з експлуатації розробленого програмного продукту NodeWatch. Описується життєвий цикл роботи із системою: від початкового розгортання інфраструктури та конфігурування змінних оточення до реєстрації мікросервісів та практичної взаємодії з клієнтським інтерфейсом під час аналізу реальних програмних інцидентів.

5.1 Порядок розгортання та конфігурування середовища

Процес підготовки системи NodeWatch до роботи починається з етапу розгортання інфраструктури. З метою забезпечення максимальної портативності та незалежності від характеристик цільової хост-машини, розгортання здійснюється за допомогою технології контейнеризації.

Користувач взаємодіє з єдиним конфігураційним файлом Docker Compose, який автоматизує процес завантаження, збирання та запуску всіх необхідних вузлів: сервера збору даних (Collector), реляційної бази даних PostgreSQL та клієнтського веб-інтерфейсу (Dashboard). Для успішного старту системи адміністратору достатньо виконати стандартну команду `docker-compose up -d` у терміналі (рис. 5.1).

```
PS C:\Users\nikic\Desktop\NodeWatch> docker-compose up -d --build
#24 2.194
#24 2.195 [plugin builtin:vite-reporter]
#24 2.195 (!) Some chunks are larger than 500 kB after minification. Consider:
#24 2.195 - Using dynamic import() to code-split the application
#24 2.195 - Use build.rolldownOptions.output.codeSplitting to improve chunking: https://ro
lldown.rs/reference/OutputOptions.codeSplitting
#24 2.195 - Adjust chunk size limit for this warning via build.chunkSizeWarningLimit.
#24 2.197 ✓ built in 1.51s
#24 DONE 2.3s

#25 [frontend stage-1 3/3] COPY --from=build /app/dist /usr/share/nginx/html
#25 DONE 0.0s

#26 [frontend] exporting to image
#26 exporting layers 0.1s done
#26 exporting manifest sha256:47acccf642ede47e9c37d2848f1047d73a4b9197db7b5b196e6be3f1180b
df90 0.0s done
#26 exporting config sha256:58cf495cb2e44375802d03e410e63a4f00ef53573f82ed73b2d6a8fd80261e
bf 0.0s done
#26 exporting attestation manifest sha256:e0effbf27d5a4745fe2e354be1b1170f48827deade280656
10f28e808b927208 0.0s done
#26 exporting manifest list sha256:d53511bb47f7f406f963bc4f8943a4679c8a8913d99d77d8c3b5246
3d471c6ad 0.0s done
#26 naming to docker.io/library/nodewatch-frontend:latest done
#26 unpacking to docker.io/library/nodewatch-frontend:latest 0.0s done
#26 DONE 0.3s

[+] up 20/20d] resolving provenance for metadata file
✓ Image postgres:16-alpine Pulled 16.1s
✓ Image nodewatch-backend Built 99.0s
✓ Image nodewatch-frontend Built 99.0s
✓ Network nodewatch_default Created 0.0s
✓ Container nodewatch_db Healthy 11.3s
✓ Container nodewatch_backend Started 11.4s
✓ Container nodewatch_frontend Started 11.5s
○ PS C:\Users\nikic\Desktop\NodeWatch>
```

Рисунок 5.1 – Процес ініціалізації та запуску контейнерів NodeWatch у терміналі

Як видно з процесу виконання команди, Docker самостійно завантажує образи, створює ізольовану віртуальну мережу для безпечної комунікації між внутрішніми контейнерами та ініціалізує постійні томи (Volumes). Використання томів є критично важливим архітектурним кроком, оскільки це гарантує збереження агрегованих даних у базі PostgreSQL навіть у випадку повного перезавантаження або оновлення контейнерів системи [21].

Важливим етапом налаштування є робота з файлом змінних оточення (.env). Архітектура NodeWatch побудована згідно з індустріальною методологією «12-factor app», яка вимагає суворої ізоляції конфігурації від програмного коду. У цьому файлі користувач визначає параметри безпеки та ключі інтеграції (рис. 5.2).

```

2 .env
3
4 # NodeWatch - Environment Variables
5 # Copy this file to .env and fill in your values
6
7 # — Database —
8 DB_HOST=localhost
9 DB_PORT=5432
10 DB_USER=nodewatch
11 DB_PASSWORD=nodewatch_secret
12 DB_NAME=nodewatch_db
13
14 # — Backend —
15 PORT=3000
16 NODE_ENV=development
17 DASHBOARD_URL=http://localhost:5173
18
19 # Secret for signing JWT tokens (change in production!)
20 JWT_SECRET=change_me_in_production_super_secret_key
21 JWT_EXPIRES_IN=7d
22
23 # — Telegram Alerts —
24 # Get this from @BotFather on Telegram
25 TELEGRAM_BOT_TOKEN=8571041401:AAG4Ay1z1eaPZzDF6rP1M1LaJbYeLr7t0v0
26
27 # — AI Assistant (Google Gemini) —
28 GEMINI_API_KEY=AIzaSyDGBbjwBoUbzO4SzoEPVTnkT3ia_GJGPvc
29

```

Рисунок 5.2 – Фрагмент конфігураційного файлу .env з параметрами інтеграції

У конфігураційному файлі вказуються реквізити доступу до бази даних, секретний криптографічний ключ для генерації JWT-токенів авторизації, а також унікальний токен для доступу до хмарної нейромережі Google Gemini API. Такий розподіл дозволяє адміністратору швидко змінювати параметри роботи (наприклад, замінити API-ключ або оновити паролі) без необхідності перезбирання всього програмного продукту.

5.2 Керування проектами та реєстрація нових сервісів

Після первинного налаштування та авторизації в системі адміністратор потрапляє до модуля керування проектами, який є відправною точкою для початку моніторингу. Оскільки екосистема «Кафедра» складається з великої кількості

мікросервісів, користувач має можливість створити окрему логічну сутність для кожного з них, вказавши унікальну назву та опис (рис. 5.3).

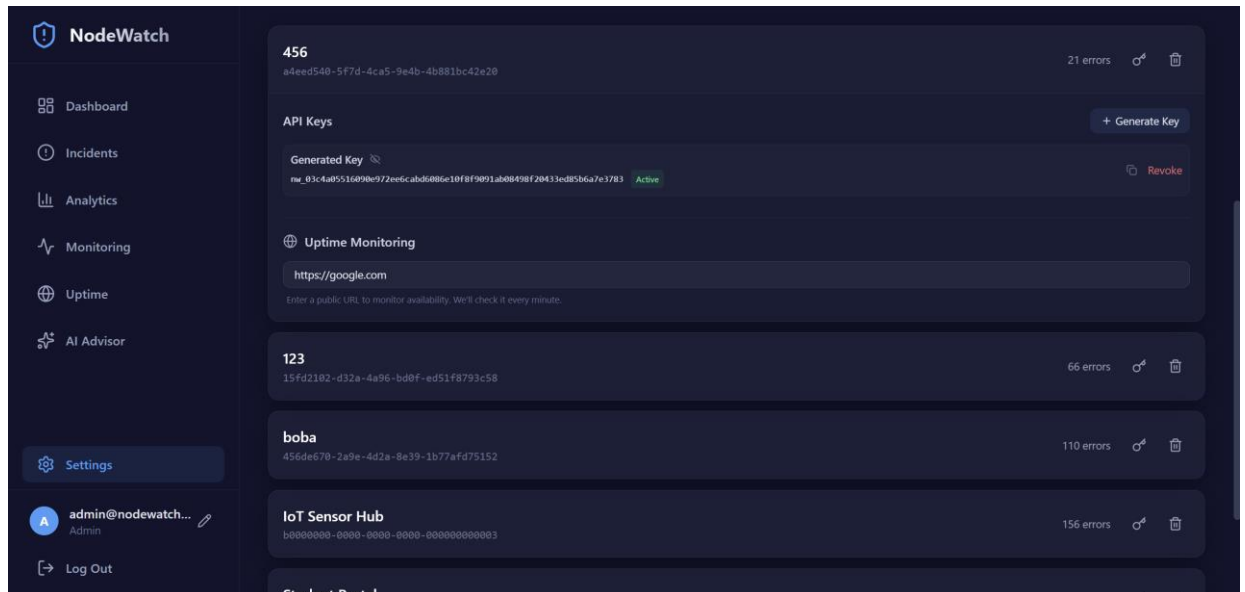


Рисунок 5.3 – Інтерфейс створення нового проекту та керування API-ключами

Під час збереження нового проекту система генерує унікальний криптографічний API-ключ. Цей маркер є обов'язковим ідентифікатором, який клієнтські мікросервіси повинні передавати у заголовках своїх запитів при відправці повідомлень про помилки. Це дозволяє колектору розмежувати потоки даних, ідентифікувати джерело інциденту та забезпечити цілісність аналітики для різних програмних модулів.

Процес практичної інтеграції цільового додатка із системою моніторингу вимагає від користувача (розробника) додавання невеликого фрагмента коду у глобальний обробник винятків свого проекту. Система NodeWatch підтримує прийом даних через стандартизований REST API, тому розробник має налаштувати відправку HTTP POST-запиту на адресу колектора. Приклад такої інтеграції наведено на рисунку 5.4.

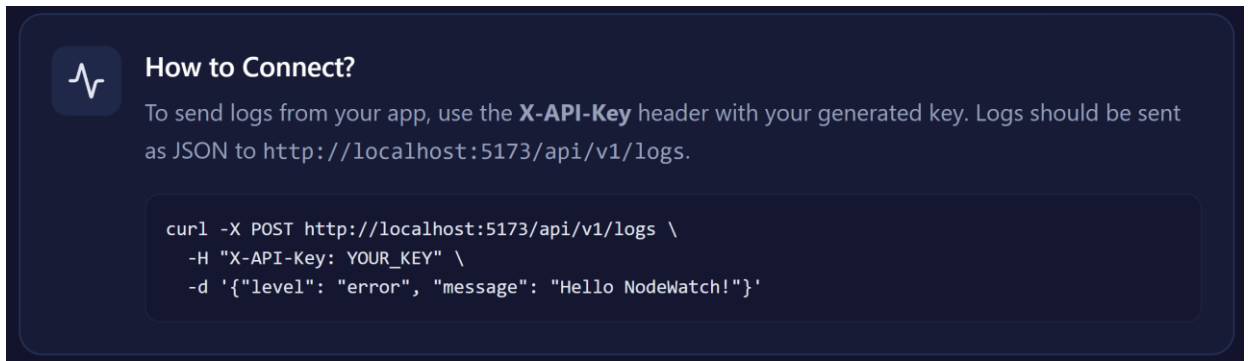


Рисунок 5.4 – Приклад коду для інтеграції клієнтського сервісу з API NodeWatch

Для успішної реєстрації інциденту розробник повинен передати згенерований ключ доступу у заголовок X-API-Key. Тіло запиту формується у форматі JSON та обов'язково включає три ключові поля: level (рівень критичності помилки, наприклад fatal або error), error (назва винятку) та message (повний текстовий стек-трейс).

Використання універсального формату JSON робить систему абсолютно сумісною з будь-якими технологіями, що використовуються на кафедрі, чи то мікросервіс мовою Node.js, Python, чи додаток на базі .NET [12]. Користувач може легко протестувати правильність з'єднання за допомогою базової утиліти curl або вбудованих інструментів середовища розробки, переконавшись, що інциденти успішно потрапляють до бази даних колектора.

5.3 Візуалізація аналітики та інтерфейс моніторингу інцидентів

Головна панель керування (Dashboard) надає адміністратору комплексний огляд стану працездатності всієї інфраструктури в режимі реального часу. Оскільки клієнтська частина системи NodeWatch побудована за принципами SPA-додатка, взаємодія з інтерфейсом відбувається без повного перезавантаження веб-сторінок [14]. Це забезпечує безперервний та плавний UX, що є критично важливим при роботі з великими масивами динамічних даних.

На головному екрані візуалізуються ключові показники ефективності: загальна кількість зафіксованих помилок за обраний період часу, розподіл інцидентів за рівнем критичності та графік активності збоїв (рис. 5.5).

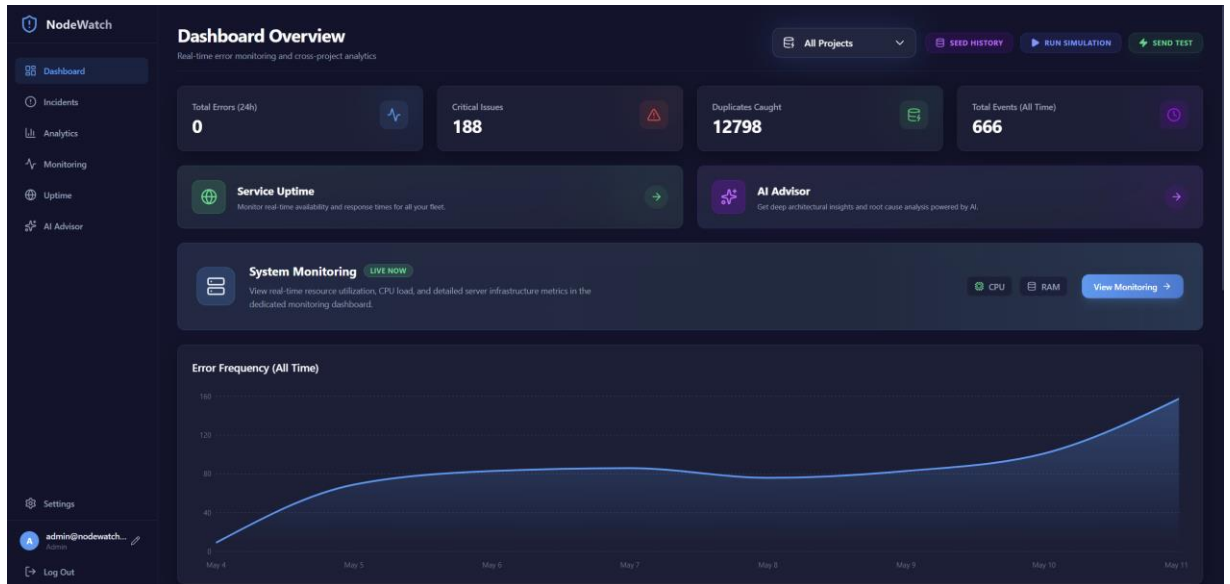


Рисунок 5.5 — Головний екран Dashboard з візуалізацією статистики інцидентів

Завдяки використанню інтерактивних графіків, користувач може миттєво ідентифікувати аномальні сплески активності, що свідчать про критичні збої або невдалі оновлення мікросервісів. Веб-інтерфейс дозволяє гнучко фільтрувати дані за датою, статусом вирішення та рівнем загрози, надаючи детальну картину по кожному окремому вузлу системи.

Для більш глибокого аналізу стабільності користувач використовує сторінку розширеної аналітики. Тут доступні кругові діаграми розподілу винятків та стовпчикові графіки найчастіших помилок (рис. 5.6).

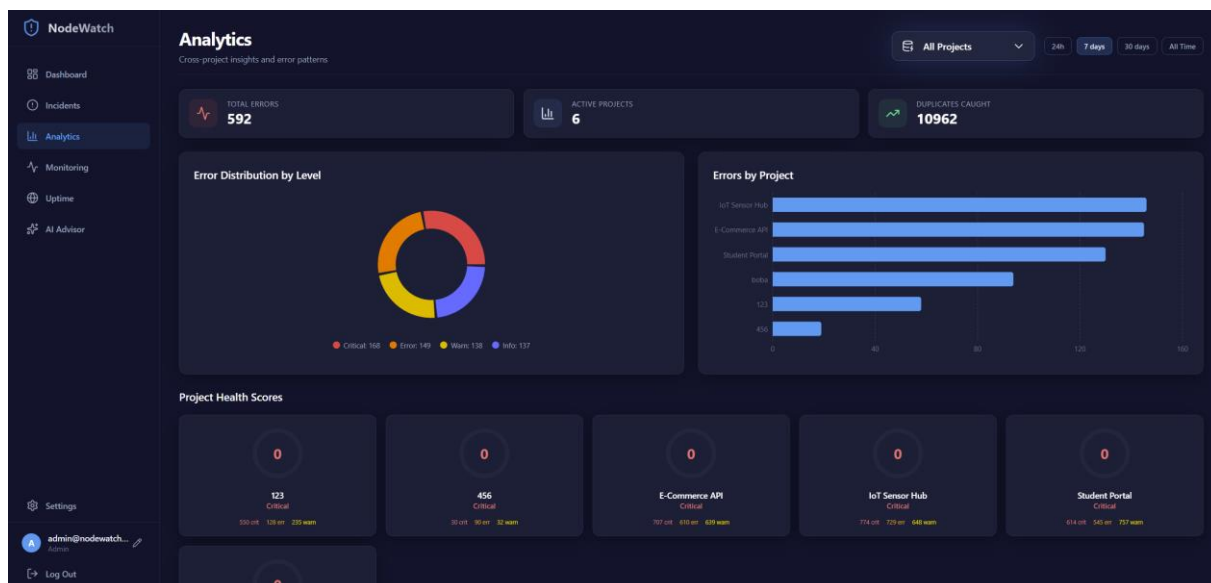


Рисунок 5.6 – Сторінка аналітики NodeWatch

Користувач може взаємодіяти з елементами графіків для швидкого переходу до списку конкретних логів, що були зафіксовані у вибраний проміжок часу. Такий рівень деталізації значно спрощує роботу системних адміністраторів кафедри при плануванні технічного обслуговування серверів та пріоритезації завдань з рефакторингу коду [2].

5.4 Взаємодія з ШІ-асистентом та системами сповіщень

Повний життєвий цикл обробки програмної помилки (User Journey) в системі NodeWatch починається з моменту отримання автоматичного сповіщення. Коли на колектор надходить критичний інцидент, система миттєво генерує алерт у корпоративний месенджер (Telegram або Discord).

Кожне сповіщення містить пряме гіперпосилання. Перейшовши за ним, розробник одразу потрапляє на сторінку детального аналізу конкретного інциденту в панелі керування. Центральним елементом цієї сторінки є блок інтелектуальних рекомендацій, згенерований нейромережею Google Gemini (рис. 5.7).

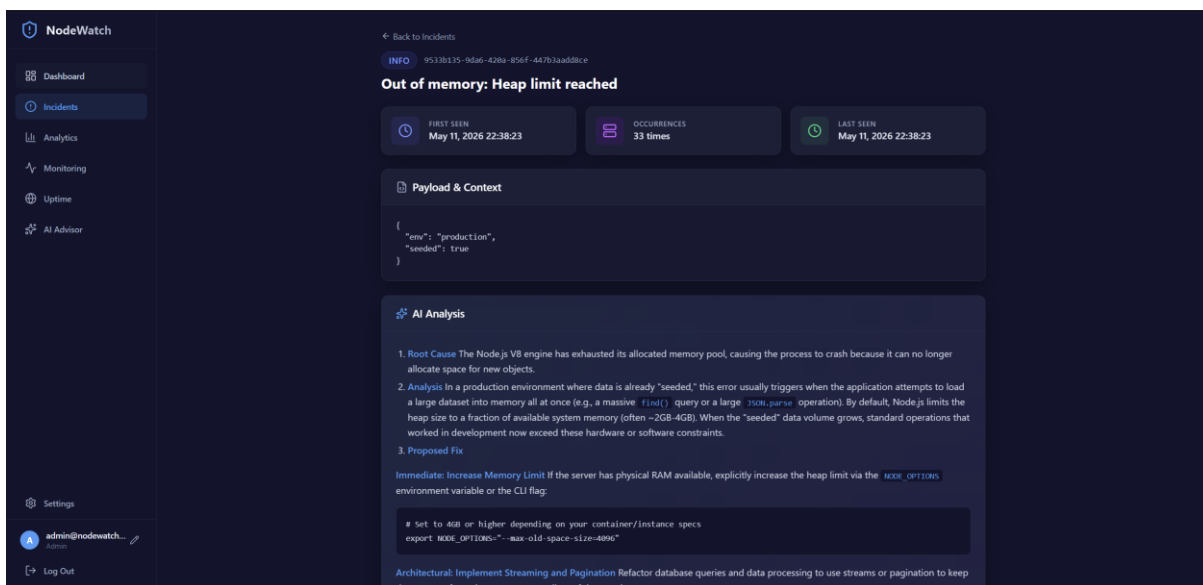


Рисунок 5.7 – Відображення деталей помилки та інтелектуальної поради від Gemini AI

Користувач бачить структуровану відповідь, де штучний інтелект не просто констатує факт збою, а здійснює семантичний розбір наданого стек-трейсу, пояснює фундаментальну причину виникнення помилки та пропонує конкретні фрагменти коду для її виправлення [19]. Це дозволяє розробнику не витратити час на пошук документації чи аналіз суміжних модулів, а одразу переходити до корекції вихідного коду, керуючись експертними підказками.

Завершенням циклу роботи користувача є зміна статусу помилки в інтерфейсі на «Вирішено» (Resolved). Ця дія супроводжується очищенням лічильника дедуплікації та архівуванням інциденту в базі даних. Якщо помилка виникне повторно після оновлення системи, колектор зареєструє її як нову подію та ініціює новий цикл сповіщення.

Висновки до розділу 5

У п'ятому розділі було наведено вичерпний опис процесів експлуатації розробленої системи моніторингу. Продемонстровано, що використання технології Docker суттєво спрощує етап розгортання колектора, бази даних та веб-інтерфейсу,

гарантуючи ідентичність робочого середовища незалежно від апаратних характеристик серверів кафедри. Конфігурування за методологією «12-factor app» забезпечило гнучке та безпечне управління параметрами інтеграції без втручання у вихідний код.

Процес реєстрації нових сервісів реалізовано максимально прозоро: генерація унікальних **API**-ключів та використання стандартизованих **JSON**-пакетів дозволяють підключити будь-який клієнтський додаток до системи NodeWatch з мінімальними витратами часу розробника.

Огляд клієнтського **UI** підтвердив високу ефективність обраної **SPA**-архітектури, яка надає адміністраторам швидкий та зручний доступ до інтерактивної аналітики. Розглянутий сценарій користувацької взаємодії довів, що синергія миттєвих сповіщень через месенджери та модуля інтелектуального аналізу коду на базі ШІ кардинально оптимізує процес відлагодження. Це дозволяє розробникам кафедри миттєво локалізувати проблему, отримувати готові рекомендації щодо її усунення та ефективно керувати життєвим циклом програмних інцидентів.

ВИСНОВКИ

У результаті виконання дипломної роботи було успішно вирішено актуальне науково-практичне завдання — спроектовано та програмно реалізовано легковисну систему централізованого моніторингу та збору помилок NodeWatch, адаптовану для розподілених додатків екосистеми «єКафедра». Проведений на початку дослідження порівняльний аналіз індустріальних стандартів (таких як Sentry та стек ELK) довів, що вони є надто ресурсомісткими для локальної інфраструктури кафедри ІПЗЕ. Це повністю підтвердило доцільність розробки власного кастомного рішення.

У процесі проектування було обґрунтовано та застосовано сучасний технологічний стек. Побудова серверної частини на базі платформи Node.js із використанням Push-моделі та REST-архітектури забезпечила високу пропускну здатність колектора. Проектування логічної структури на основі СУБД PostgreSQL із застосуванням бінарного типу JSONB дозволило ефективно зберігати неструктуровані метадані інцидентів, зберігаючи при цьому транзакційну цілісність. Завдяки компонентному підходу бібліотеки React створено реактивний SPA-додаток, що гарантує високий рівень користувацького досвіду під час аналізу телеметрії. Уся система успішно контейнеризована за допомогою Docker, що забезпечило її портативність та ідентичність середовищ розгортання.

Ключовим інженерним досягненням стала реалізація алгоритму криптографічної дедуплікації на базі MD5-хешування. Цей механізм надійно захищає базу даних від переповнення під час виникнення циклічних помилок, суттєво оптимізуючи дисковий простір.

Наукову новизну роботи було значно розширено за рахунок інноваційної інтеграції модуля штучного інтелекту через Google Gemini API. Це перетворило NodeWatch зі звичайного агрегатора логів на інтелектуального помічника розробника, здатного автоматично здійснювати семантичний аналіз стек-трейсів та генерувати експертні рекомендації щодо виправлення багів.

Для забезпечення корпоративних стандартів безпеки впроваджено систему контролю доступу на основі моделі RBAC та розроблено механізми захисту від мережеских перевантажень. Багатоканальна система сповіщень через месенджери, обладнана гнучкими фільтрами критичності, звела до мінімуму затримку між виникненням збою та інформуванням адміністратора.

Підсумовуючи результати, можна стверджувати, що всі завдання, поставлені в дипломній роботі, виконані в повному обсязі. Впровадження системи NodeWatch в екосистему «єКафедра» дозволяє відмовитися від ручного пошуку проблем, значно скоротити показник середнього часу відновлення (MTTR) та перевести процес технічної підтримки програмного забезпечення на рівень проактивного моніторингу. Розроблений програмний продукт має високу практичну цінність, відповідає всім заявленим нефункціональним вимогам щодо споживання ресурсів і є повністю готовим до промислової експлуатації у навчальному та робочому процесах кафедри.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Great Britain: Prentice Hall, 2018. 352 p.
2. Beyer B., Jones C., Petoff J., Murphy N. R. Site Reliability Engineering: How Google Runs Production Systems. O'Reilly Media, 2016. 552 p.
3. Burns B. Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services. O'Reilly Media, 2018. 166 p.
4. Node.js v20.12.2 documentation. Node.js. URL: <https://nodejs.org/docs/latest-v20.x/api/> (дата звернення: 12.05.2026).
5. PostgreSQL 16.2 Documentation. Chapter 8. Data Types. JSON Types. PostgreSQL. URL: <https://www.postgresql.org/docs/16/datatype-json.html> (дата звернення: 12.05.2026).
6. React Documentation. URL: <https://react.dev/> (дата звернення: 12.05.2026).
7. Docker Documentation. Orientation and setup. URL: <https://docs.docker.com/get-started/> (дата звернення: 12.05.2026).
8. Google Gemini API Documentation. URL: <https://ai.google.dev/docs> (дата звернення: 12.05.2026).
9. MD5 Message-Digest Algorithm. IETF RFC 1321. URL: <https://datatracker.ietf.org/doc/html/rfc1321> (дата звернення: 12.05.2026).
10. Casciaro M., Mammino L. Node.js Design Patterns: Design and implement production-grade Node.js applications. 3rd ed. Packt Publishing, 2020. 664 p.
11. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. O'Reilly Media, 2021. 614 p.
12. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : Ph.D. thesis. University of California, Irvine, 2000. 162 p.
13. Ferraiolo D. F., Kuhn D. R., Chandramouli R. Role-Based Access Control. 2nd ed. Artech House, 2007. 404 p.

14. Banks A., Porcello E. Learning React: Modern Patterns for Developing React Apps. 2nd ed. O'Reilly Media, 2020. 341 p.
15. Sentry Documentation. Sentry. URL: <https://docs.sentry.io/> (дата звернення: 12.05.2026).
16. Elasticsearch Guide. Elastic. URL: <https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html> (дата звернення: 12.05.2026).
17. Prometheus Overview. Prometheus. URL: <https://prometheus.io/docs/introduction/overview/> (дата звернення: 12.05.2026).
18. Schönig H.-J. Mastering PostgreSQL 13: Build, administer, and maintain enterprise-class databases, high performance, and high availability. Packt Publishing, 2020. 724 p.
19. Gemini Team. Gemini: A Family of Highly Capable Multimodal Models. arXiv preprint arXiv:2312.11805. 2023. URL: <https://arxiv.org/abs/2312.11805> (дата звернення: 12.05.2026).
20. Telegram Bot API. Telegram. URL: <https://core.telegram.org/bots/api> (дата звернення: 12.05.2026).
21. Mouat I. Using Docker: Developing and Deploying Software with Containers. O'Reilly Media, 2015. 354 p.

ДОДАТОК А Лістинг розробленої системи

server.js

```
const path = require('path');
require('dotenv').config({ path: path.join(__dirname, '../.env') });
const express = require('express');
const cors = require('cors');

// DB connection (establishes pool on startup)
require('./db');

const { errorHandler, notFound } = require('./middleware/errorHandler');
const configService = require('./services/config');
const uptimeService = require('./services/uptime.service');

// Load dynamic config from DB
configService.loadFromDb();

// Start background services
uptimeService.start();

const app = express();
const PORT = process.env.PORT || 3000;

// _____ Global _____ Middleware
app.use(cors());
app.use(express.json({ limit: '1mb' })); // Parse JSON bodies

// _____ Health _____ Check
app.get('/health', (req, res) => {
```

```

res.json({
  success: true,
  service: 'NodeWatch Collector API',
  version: '1.0.0',
  timestamp: new Date().toISOString(),
});
});

```

//	—	API	Routes
----	---	-----	--------

```

app.use('/api/v1/logs',    require('./routes/logs.routes'));
app.use('/api/v1/auth',    require('./routes/auth.routes'));
app.use('/api/v1/dashboard', require('./routes/dashboard.routes'));
app.use('/api/v1/admin',    require('./routes/admin.routes'));
app.use('/api/v1/analytics', require('./routes/analytics.routes'));
app.use('/api/v1/system',    require('./routes/system.routes'));
app.use('/api/v1/ai',        require('./routes/ai.routes'));

```

//	—	404	&	Error	Handlers	(must	be	LAST)
----	---	-----	---	-------	----------	-------	----	-------

```

app.use(notFound);
app.use(errorHandler);

```

//	—	Start	Server
----	---	-------	--------

```

app.listen(PORT, () => {
  console.log(`🔗 NodeWatch API running on http://localhost:${PORT}`);
  console.log(` Health check: http://localhost:${PORT}/health`);
  console.log(` Environment: ${process.env.NODE_ENV || 'development'}`);
});

```

Dashboard.jsx

```
import React, { useState, useEffect } from 'react';
import { Link } from 'react-router-dom';
import client from '../api/client';
import StatsCard from '../components/StatsCard';
import ErrorsChart from '../components/ErrorsChart';
import IncidentsTable from '../components/IncidentsTable';
import AdminControls from '../components/AdminControls';
import { AlertTriangle, Activity, DatabaseZap, Clock, ArrowRight, BarChart3, Cpu, Server, Database,
Globe, Sparkles } from 'lucide-react';
import { PieChart, Pie, Cell, ResponsiveContainer } from 'recharts';

const LEVEL_COLORS = { critical: '#ef4444', error: '#f97316', warn: '#eab308', info: '#3b82f6' };

const AnalyticsBanner = ({ projectId }) => {
  const [data, setData] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const pid = projectId || 'all';
        const res = await client.get(`/analytics/overview?range=7d&project_id=${pid}`);
        if (res.data.success) setData(res.data.overview);
      } catch (err) { /* silent */ }
    };
    fetchData();
  }, [projectId]);

  const pieData = (data?.by_level || []).map(item => ({
    name: item.level,
    value: parseInt(item.count),
  }));

  const totalErrors = data?.totals?.total_errors || 0;
```

```
const projectCount = data?.totals?.active_projects || 0;

return (
  <div className="relative overflow-hidden rounded-2xl border border-slate-700/50 bg-gradient-to-r
from-dark-900 via-dark-800 to-brand-900/20">
    { /* Decorative glow */ }
    <div className="absolute -right-20 -top-20 w-60 h-60 bg-brand-500/10 rounded-full blur-3xl" />
    <div className="absolute -left-10 -bottom-10 w-40 h-40 bg-purple-500/10 rounded-full blur-3xl"
/>

    <div className="relative flex flex-col sm:flex-row items-center gap-6 p-6">
      { /* Mini Pie Chart */ }
      <div className="w-28 h-28 shrink-0">
        {pieData.length > 0 ? (
          <ResponsiveContainer width="100%" height="100%">
            <PieChart>
              <Pie
                data={pieData}
                cx="50%" cy="50%"
                innerRadius={28} outerRadius={48}
                paddingAngle={3} dataKey="value" stroke="none"
              >
                {pieData.map((entry) => (
                  <Cell key={entry.name} fill={LEVEL_COLORS[entry.name] || '#64748b'} />
                ))}
              </Pie>
            </PieChart>
          </ResponsiveContainer>
        ) : (
          <div className="w-full h-full rounded-full border-4 border-slate-700 flex items-center justify-
center">
            <BarChart3 className="w-8 h-8 text-slate-600" />
          </div>
        )}
      </div>
    </div>
  )
)
```

```

</div>

{/* Text */}
<div className="flex-1 text-center sm:text-left">
  <h3 className="text-lg font-bold text-white mb-1">Cross-Project Analytics</h3>
  <p className="text-sm text-slate-400">
    {totalErrors > 0
      ? `${totalErrors} errors across ${projectCount} project${projectCount !== 1 ? 's' : ''} in the last
7 days. View heatmaps, health scores, and trends.`
      : 'Explore error patterns, project health scores, and heatmaps across all your projects.'
    }
  </p>
  {/* Mini legend */}
  {pieData.length > 0 && (
    <div className="flex flex-wrap justify-center sm:justify-start gap-3 mt-2">
      {pieData.map(entry => (
        <span key={entry.name} className="flex items-center gap-1 text-xs text-slate-400">
          <span className="w-2 h-2 rounded-full" style={{ backgroundColor:
LEVEL_COLORS[entry.name] }} />
          {entry.name}: {entry.value}
        </span>
      ))}
    </div>
  )}
</div>

{/* CTA Button */}
<Link
  to="/analytics"
  className="shrink-0 inline-flex items-center gap-2 px-5 py-2.5 rounded-xl bg-gradient-to-r from-
brand-500 to-brand-600 text-white text-sm font-semibold hover:from-brand-600 hover:to-brand-500
transition-all shadow-lg shadow-brand-500/20 hover:shadow-brand-500/40"
  >
  View Analytics

```

```

        <ArrowRight className="w-4 h-4" />
      </Link>
    </div>
  </div>
);
};

```

```

const Dashboard = () => {
  const [stats, setStats] = useState(null);
  const [projects, setProjects] = useState([]);
  const [selectedProjectId, setSelectedProjectId] = useState("");
  const [isLoading, setIsLoading] = useState(true);
  const [refreshKey, setRefreshKey] = useState(0);

```

// Fetch projects on mount

```

useEffect(() => {
  const fetchProjects = async () => {
    try {
      const res = await client.get('/dashboard/projects');
      if (res.data.success && res.data.projects.length > 0) {
        setProjects(res.data.projects);
        setSelectedProjectId('all'); // Default to All Projects
      }
    } catch (err) {
      console.error('Failed to fetch projects', err);
    }
  };
  fetchProjects();
}, []);

```

// Fetch stats when selected project changes

```

useEffect(() => {
  if (!selectedProjectId) return;

```

```

const fetchStats = async () => {
  setIsLoading(true);
  try {
    const res = await client.get(`/dashboard/stats?project_id=${selectedProjectId}`);
    if (res.data.success) {
      setStats(res.data.stats);
    }
  } catch (err) {
    console.error('Failed to fetch stats', err);
  } finally {
    setIsLoading(false);
  }
};

fetchStats();

// Auto-refresh every 30 seconds
const interval = setInterval(fetchStats, 30000);
return () => clearInterval(interval);
}, [selectedProjectId, refreshKey]);

return (
  <div className="space-y-6">
    {/* Header & Project Selector */}
    <div className="flex flex-col sm:flex-row sm:items-center justify-between gap-6 pb-4 border-b
border-white/5">
      <div>
        <h1 className="text-3xl font-bold text-white tracking-tight">Dashboard Overview</h1>
        <p className="text-slate-400 text-sm mt-1.5">Real-time error monitoring and cross-project
analytics</p>
      </div>

      <div className="flex flex-col sm:flex-row items-stretch sm:items-center gap-4">
        {projects.length > 0 && (

```

```

<div className="relative group min-w-[240px]">
  <div className="absolute inset-0 bg-brand-500/20 blur-xl rounded-xl group-hover:bg-brand-
500/30 transition-all duration-300"></div>
    <div className="relative flex items-center bg-dark-800/80 border border-brand-500/30
rounded-xl overflow-hidden backdrop-blur-sm hover:border-brand-500/50 transition-colors">
      <div className="pl-4 pr-2 py-3 flex items-center text-brand-400">
        <DatabaseZap className="w-5 h-5" />
      </div>
      <select
        value={selectedProjectId}
        onChange={(e) => setSelectedProjectId(e.target.value)}
        className="bg-transparent text-white font-medium text-base appearance-none py-3 pr-10
pl-2 outline-none cursor-pointer w-full"
      >
        <option value="all" className="bg-dark-900 text-brand-400 font-semibold">All
Projects</option>
        <option disabled className="bg-dark-900 text-slate-500">—————</option>
        {projects.map(p => (
          <option key={p.id} value={p.id} className="bg-dark-900 text-slate-
200">{p.name}</option>
        ))}
      </select>
      <div className="absolute right-4 pointer-events-none text-brand-400/70 group-hover:text-
brand-400 transition-colors">
        <svg className="w-5 h-5" fill="none" viewBox="0 0 24 24" stroke="currentColor">
          <path strokeLinecap="round" strokeLinejoin="round" strokeWidth={2} d="M19 9l-7 7-7
7" />
        </svg>
      </div>
    </div>
  </div>
  <AdminControls projectId={selectedProjectId} onEventSent={() => setRefreshKey(k => k + 1)}
/>

```

```

</div>
</div>

{isLoading && !stats ? (
  <div className="flex justify-center py-20">
    <div className="animate-spin rounded-full h-12 w-12 border-t-2 border-b-2 border-brand-500"></div>
  </div>
) : stats ? (
  <>
    { /* Stats Grid */ }
    <div className="grid grid-cols-1 sm:grid-cols-2 lg:grid-cols-4 gap-6">
      <StatsCard
        title="Total Errors (24h)"
        value={stats.errors_last_24h}
        icon={Activity}
        colorClass="bg-brand-500 text-brand-500"
      />
      <StatsCard
        title="Critical Issues"
        value={stats.by_level?.find(l => l.level === 'critical')?.count || 0}
        icon={AlertTriangle}
        colorClass="bg-red-500 text-red-500"
      />
      <StatsCard
        title="Duplicates Caught"
        value={stats.duplicates_caught}
        icon={DatabaseZap}
        colorClass="bg-emerald-500 text-emerald-500"
      />
      <StatsCard
        title="Total Events (All Time)"
        value={stats.total_errors}
        icon={Clock}

```

```

        colorClass="bg-purple-500 text-purple-500"
    />
</div>

{/* Feature Banners Grid */}
<div className="grid grid-cols-1 lg:grid-cols-2 gap-6">
    {/* Uptime Banner */}
    <Link to="/uptime" className="relative group overflow-hidden rounded-xl border border-emerald-500/20 bg-dark-800/80 hover:border-emerald-500/50 transition-all duration-300">
        <div className="absolute inset-0 bg-gradient-to-br from-emerald-500/10 via-brand-500/5 to-transparent opacity-50 group-hover:opacity-100 transition-opacity"></div>
        <div className="absolute -right-10 -top-10 w-40 h-40 bg-emerald-500/10 rounded-full blur-3xl group-hover:bg-emerald-500/20 transition-all"></div>

        <div className="relative p-6 flex items-center justify-between">
            <div className="flex items-center gap-5">
                <div className="h-14 w-14 rounded-xl bg-emerald-500/20 flex items-center justify-center border border-emerald-500/30 group-hover:scale-110 transition-transform">
                    <Globe className="w-7 h-7 text-emerald-400" />
                </div>
                <div>
                    <h3 className="text-lg font-bold text-white group-hover:text-emerald-300 transition-colors">Service Uptime</h3>
                    <p className="text-xs text-slate-400 mt-1 max-w-[200px] sm:max-w-none">Monitor real-time availability and response times for all your fleet.</p>
                </div>
            </div>
            <div className="hidden sm:flex shrink-0 w-10 h-10 rounded-full bg-emerald-500/10 items-center justify-center text-emerald-400 group-hover:bg-emerald-500 group-hover:text-white transition-all">
                <ArrowRight className="w-5 h-5" />
            </div>
        </div>
    </Link>

```

```

    { /* AI Advisor Banner */}

    <Link to="/advisor" className="relative group overflow-hidden rounded-xl border border-
purple-500/20 bg-dark-800/80 hover:border-purple-500/50 transition-all duration-300">

        <div className="absolute inset-0 bg-gradient-to-br from-purple-500/10 via-brand-500/5 to-
transparent opacity-50 group-hover:opacity-100 transition-opacity"></div>

        <div className="absolute -right-10 -top-10 w-40 h-40 bg-purple-500/10 rounded-full blur-3xl
group-hover:bg-purple-500/20 transition-all"></div>


    <div className="relative p-6 flex items-center justify-between">
        <div className="flex items-center gap-5">
            <div className="h-14 w-14 rounded-xl bg-purple-500/20 flex items-center justify-center
border border-purple-500/30 group-hover:scale-110 transition-transform">
                <Sparkles className="w-7 h-7 text-purple-400" />
            </div>
            <div>
                <h3 className="text-lg font-bold text-white group-hover:text-purple-300 transition-
colors">AI Advisor</h3>
                <p className="text-xs text-slate-400 mt-1 max-w-[200px] sm:max-w-none">Get deep
architectural insights and root cause analysis powered by AI.</p>
            </div>
            <div>
                <div className="hidden sm:flex shrink-0 w-10 h-10 rounded-full bg-purple-500/10 items-
center justify-center text-purple-400 group-hover:bg-purple-500 group-hover:text-white transition-all">
                    <ArrowRight className="w-5 h-5" />
                </div>
            </div>
        </div>
    </Link>
</div>

    { /* System Monitoring CTA Banner */}

    <Link to="/monitoring" className="block relative group overflow-hidden rounded-xl border
border-brand-500/20 bg-dark-800/80 hover:border-brand-500/50 transition-all duration-300">

```

```
<div className="absolute inset-0 bg-gradient-to-r from-brand-500/10 to-emerald-500/10
opacity-50 group-hover:opacity-100 transition-opacity"></div>
```

```
<div className="absolute right-0 top-0 bottom-0 w-1/3 bg-gradient-to-l from-brand-500/10 to-
transparent pointer-events-none"></div>
```

```
<div className="relative p-6 sm:p-8 flex items-center justify-between">
  <div className="flex items-center gap-6">
    <div className="hidden sm:flex h-16 w-16 rounded-2xl bg-brand-500/20 items-center justify-
center border border-brand-500/30 group-hover:scale-110 transition-transform">
      <Server className="w-8 h-8 text-brand-400" />
    </div>
    <div>
      <div className="flex items-center gap-3 mb-2">
        <h3 className="text-xl font-bold text-white group-hover:text-brand-400 transition-
colors">System Monitoring</h3>
        <span className="px-2.5 py-0.5 rounded-full text-xs font-bold tracking-wide uppercase
bg-emerald-500/20 text-emerald-400 border border-emerald-500/30">Live Now</span>
      </div>
        <p className="text-slate-400 max-w-xl text-sm leading-relaxed">View real-time resource
utilization, CPU load, and detailed server infrastructure metrics in the dedicated monitoring
dashboard.</p>
      </div>
    </div>
    <div className="hidden md:flex items-center gap-6 text-slate-400 transition-colors">
      <div className="flex items-center gap-3 group-hover:text-white">
        <div className="flex items-center gap-2 bg-dark-900/50 px-3 py-1.5 rounded-lg border
border-white/5">
          <Cpu className="w-4 h-4 text-emerald-400" />
          <span className="text-sm font-medium">CPU</span>
        </div>
        <div className="flex items-center gap-2 bg-dark-900/50 px-3 py-1.5 rounded-lg border
border-white/5">
          <Database className="w-4 h-4 text-brand-400" />
          <span className="text-sm font-medium">RAM</span>
        </div>
      </div>
    </div>
  </div>
```

```

        </div>
    </div>

    <div className="inline-flex items-center gap-2 px-5 py-2.5 rounded-xl bg-gradient-to-r from-brand-500 to-brand-600 text-white text-sm font-semibold hover:from-brand-600 hover:to-brand-500 transition-all shadow-lg shadow-brand-500/20 hover:shadow-brand-500/40">
        View Monitoring
        <ArrowRight className="w-4 h-4" />
    </div>
</div>
</div>
</Link>

{/* Main Chart */}
<div className="glass-panel p-4 sm:p-6">
    <h2 className="text-lg font-semibold text-white mb-6">Error Frequency (All Time)</h2>
    <ErrorsChart data={stats.timeline_24h} />
</div>

{/* Analytics Banner */}
<AnalyticsBanner projectId={selectedProjectId} />

{/* Incidents Table */}
<IncidentsTable projectId={selectedProjectId} refreshKey={refreshKey} />
</>
) : null}
</div>

);
};

export default Dashboard;

alerts.js
/**
 * NodeWatch - Alert Service

```

```

*
* Checks if incoming logs trigger any alert configurations for the project.
* Supports sending notifications via Telegram.
*/
const TelegramBot = require('node-telegram-bot-api');
const https = require('https');
const db = require('../db');
const configService = require('./config');

let bot = null;
let currentToken = null;

/**
 * Get or initialize the Telegram bot instance.
 */
function getBot() {
  const token = configService.get('telegram_bot_token');

  if (!token) return null;

  // If token changed, re-initialize
  if (token !== currentToken) {
    currentToken = token;
    bot = new TelegramBot(token, { polling: false });
    console.log('✅ Telegram Bot (re)configured with dynamic token');
  }

  return bot;
}

// Map levels to numeric weights for comparison
const LEVEL_WEIGHTS = {
  info: 10,
  warn: 20,

```

```

    error: 30,
    critical: 40,
};

/**
 * Process alerts for a given log entry.
 * Checks active alert configurations for the project and sends notifications
 * if the log level is high enough.
 *
 * @param {Object} project - { id, name }
 * @param {Object} log - The saved log record
 * @param {boolean} isDuplicate - Whether this is a duplicate occurrence
 */
async function processAlerts(project, log, isDuplicate) {
    // To avoid spam, only alert on the FIRST occurrence of an error
    if (isDuplicate) {
        return;
    }

    try {
        // Find all active alert configs for this project
        const result = await db.query(
            `SELECT channel, recipient_id, min_level
            FROM alert_configs
            WHERE project_id = $1 AND is_enabled = true`,
            [project.id]
        );

        if (result.rows.length === 0) return;

        const logWeight = LEVEL_WEIGHTS[log.level] || 0;

        for (const config of result.rows) {
            const configWeight = LEVEL_WEIGHTS[config.min_level] || 0;

```

```

// Check if log level meets or exceeds the minimum level required for this alert
if (logWeight >= configWeight) {
  if (config.channel === 'telegram') {
    const activeBot = getBot();
    if (activeBot) {
      await sendTelegramAlert(config.recipient_id, project.name, log, activeBot);
    }
  } else if (config.channel === 'discord') {
    await sendDiscordAlert(config.recipient_id, project.name, log);
  }
}
} catch (err) {
  console.error(' ✖ Failed to process alerts:', err.message);
}
}

/**
 * Send a formatted message to a Telegram chat.
 *
 * @param {string} chatId - Telegram chat ID
 * @param {string} projectName - Name of the project
 * @param {Object} log - The log entry
 */
async function sendTelegramAlert(chatId, projectName, log, activeBot) {
  const emoji = log.level === 'critical' ? '💣' :
    log.level === 'error' ? '✖' :
    log.level === 'warn' ? '⚠' : 'i';

  const message = `${emoji} *NodeWatch Alert*\n*Project:* ${projectName}\n*Level:*
  ${log.level.toUpperCase()}\n\n*Message:* \n\`\`\`text\n${log.message}\n\`\`\`;

```

```

try {
  await activeBot.sendMessage(chatId, message, { parse_mode: 'Markdown' });
} catch (err) {
  console.error(` ✖ Telegram alert failed for chat ${chatId}:`, err.message);
}
}

/**
 * Send a rich embed message to a Discord Webhook.
 *
 * @param {string} webhookUrl - Discord Webhook URL
 * @param {string} projectName - Name of the project
 * @param {Object} log - The log entry
 */
async function sendDiscordAlert(webhookUrl, projectName, log) {
  const color = log.level === 'critical' ? 0xff4757 : // Red
    log.level === 'error' ? 0xffa502 : // Orange
    log.level === 'warn' ? 0xeccc68 : // Yellow
    0x2f3542; // Dark

  const payload = JSON.stringify({
    embeds: [{
      title: `NodeWatch Alert: ${projectName}`,
      description: `**${log.message}**`,
      color: color,
      fields: [
        { name: 'Level', value: log.level.toUpperCase(), inline: true },
        { name: 'Project', value: projectName, inline: true },
        { name: 'Time', value: new Date(log.last_seen_at || log.created_at).toLocaleString(), inline: true }
      ],
      footer: { text: 'NodeWatch Monitoring' },
      timestamp: new Date().toISOString(),
      url: `${configService.get('dashboard_url', 'http://localhost:5173')}/incidents/${log.id}`
    }]
  })
}

```

```

});

try {
  const url = new URL(webhookUrl);
  const options = {
    hostname: url.hostname,
    path: url.pathname + url.search,
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      'Content-Length': Buffer.byteLength(payload)
    }
  };

  return new Promise((resolve, reject) => {
    const req = https.request(options, (res) => {
      if (res.statusCode >= 200 && res.statusCode < 300) resolve();
      else reject(new Error(`Discord API returned status ${res.statusCode}`));
    });

    req.on('error', (err) => reject(err));
    req.write(payload);
    req.end();
  });
} catch (err) {
  console.error(` ✖ Discord alert failed for ${projectName}:`, err.message);
}

module.exports = {
  processAlerts,
};

```

ДОДАТОК Б Презентація



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці

РОЗРОБКА СИСТЕМИ МОНІТОРИНГУ ТА ЗБОРУ ПОМИЛОК ДЛЯ РОЗПОДІЛЕНИХ ДОДАТКІВ

Виконав: студент групи ТВ-23 Чуйко Нікіта Олегович

Керівник: Колумбет Вадим Петрович, старший викладач, PhD

2026

АКТУАЛЬНІСТЬ ТЕМИ ТА ПРОБЛЕМАТИКА

Ускладнення архітектури:

Зростання кількості мікросервісів в екосистемі «єКафедра» ускладнює локалізацію програмних збоїв.

Критичний показник MTTR:

Відсутність єдиної точки збору телеметрії призводить до великих витрат часу на пошук та усунення багів (Mean Time To Recovery).

Апаратні обмеження кафедри:

Індустріальні гіганти моніторингу (Sentry, ELK) є ресурсомісткими (потребують від 4 до 8 ГБ RAM для розгортання).

Висновок:

Гостра потреба у розробці власного легковисного рішення (< 1 ГБ RAM) з функціями проактивного аналізу.



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

1/13

МЕТА ТА ЗАВДАННЯ ДОСЛІДЖЕННЯ

Об'єкт: процес моніторингу та збору помилок у розподілених програмних системах.

Предмет: алгоритми агрегації, дедуплікації та інтелектуального аналізу програмних інцидентів.

Мета: розробка легковисної програмної системи NodeWatch для централізованого збору та проактивного аналізу помилок в екосистемі «єКафедра».



Основні завдання:

1. Проектування архітектури колектора та бази даних (JSONB).
2. Реалізація криптографічного алгоритму дедуплікації (MD5).
3. Інтеграція ШІ-асистента для семантичного аналізу логів.
4. Розробка SPA-додатка та багатоканальної системи сповіщень.



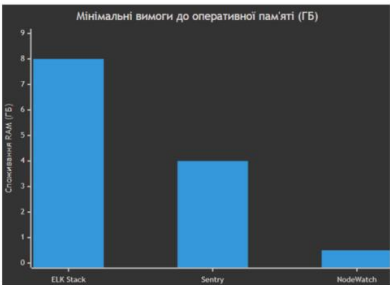
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

2/13

ПОРІВНЯЛЬНИЙ АНАЛІЗ ТА ОБҐРУНТУВАННЯ РОЗРОБКИ

Індустріальні рішення (Sentry, ELK Stack)

- ❌ Потужний функціонал, але **критично висока ресурсоємність** (потребують від 4 до 8 ГБ RAM для локального розгортання).
- ❌ Перевантаженість зайвими функціями та складна підтримка інфраструктури.



Власна розробка (NodeWatch)

- ✅ **Легковаговість:** мінімальне споживання ресурсів (< 1 ГБ RAM), що ідеально для серверів кафедри.
- ✅ **Спеціалізованість:** адаптовано виключно під потреби екосистеми «єКафедра».
- ✅ **Інноваційність:** вбудована безкоштовна інтеграція зі штучним інтелектом (Google Gemini API) для аналізу помилок.



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

3/13

ВИМОГИ ДО СИСТЕМИ

Функціональні вимоги:

- a) Централізований збір логів від різних мікросервісів у реальному часі.
- b) Автоматична дедуплікація ідентичних помилок з використанням MD5.
- c) Інтелектуальний аналіз стек-трейсів за допомогою ШІ (Gemini).
- d) Генерація сповіщень у месенджери (Telegram/Discord) за заданими правилами.
- e) Розмежування рівнів доступу користувачів (RBAC).

Нефункціональні вимоги:

- a) **Продуктивність:** Мінімальне споживання ОЗП (до 1 ГБ RAM).
- b) **Надійність:** Висока доступність завдяки контейнеризації в Docker.
- c) **Масштабованість:** Здатність обробляти великі потоки логів без затримок.
- d) **Безпека:** Шифрування API-ключів та безпечне керування правами доступу.
- e) **Сумісність:** Підтримка стандартизованих JSON-запитів.



ВИКОРИСТАНИЙ СТЕК ТЕХНОЛОГІЙ



Серверна частина (Backend): Node.js



База даних (Database): PostgreSQL

Клієнтська частина (Frontend): React

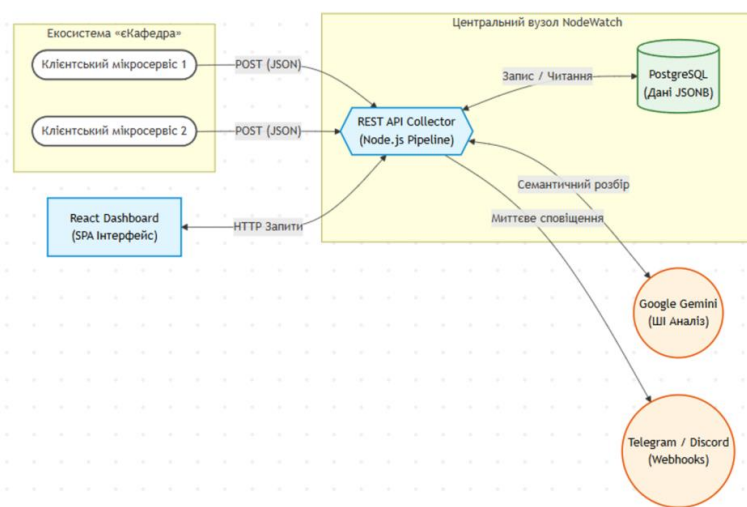
Інфраструктура: Docker



Інтеграції (ШІ та Сповіщення): Google Gemini, Telegram та Discord



АРХІТЕКТУРНА МОДЕЛЬ СИСТЕМИ

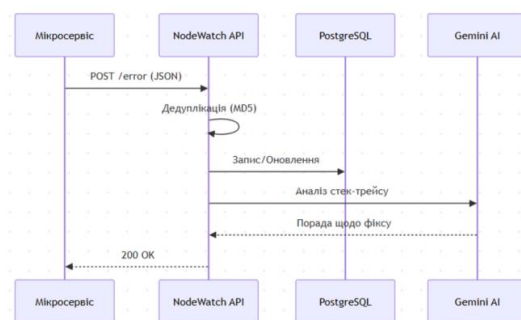


UML-МОДЕЛЮВАННЯ СИСТЕМИ

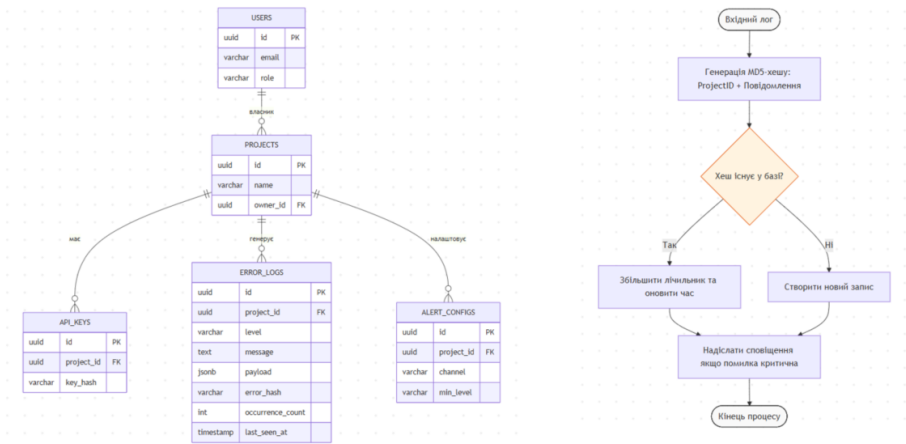
Діаграма прецедентів (Use Case)



Діаграма послідовності (Sequence)



СТРУКТУРА БД ТА АЛГОРИТМ ДЕДУПЛІКАЦІЇ

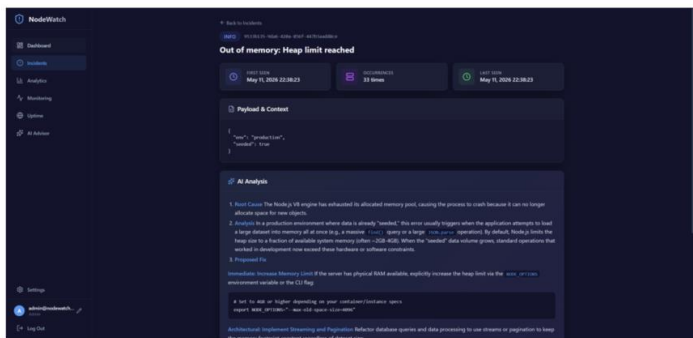


ER-діаграма (для СУБД PostgreSQL)

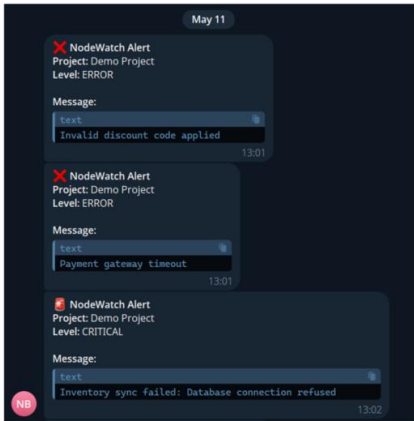
Блок-схема алгоритму дедуплікації

- 1. **Гнучкість:** PostgreSQL + JSONB для динамічних логів без зміни структури БД.
- 2. **Оптимізація:** MD5-хешування для автоматичної дедуплікації інцидентів.
- 3. **Економічність:** Інкрементація лічильника замість дублювання — економія ресурсів.

ІНТЕГРАЦІЯ ШІ ТА СПОВІЩЕННЯ

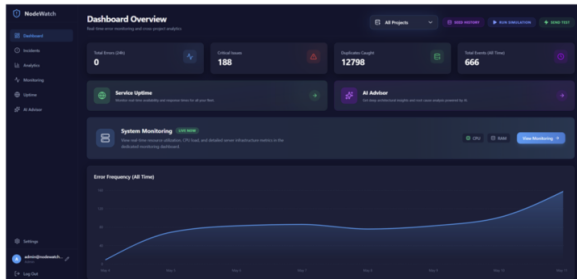


Семантичний аналіз помилки за допомогою Gemini API

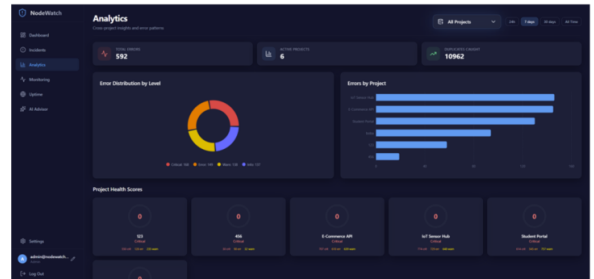


Миттєвий алерт у Telegram-каналі проекту

ІНТЕРФЕЙС КОРИСТУВАЧА



Головна панель (Dashboard) зі статистикою помилок



Сторінка аналітики



ТЕСТУВАННЯ СИСТЕМИ

```
PS C:\Users\nikic\Desktop\NodeWatch\backend> npm test

> nodewatch-backend@1.0.0 test
> jest

PASS  __tests__/config.test.js
PASS  __tests__/deduplication.test.js

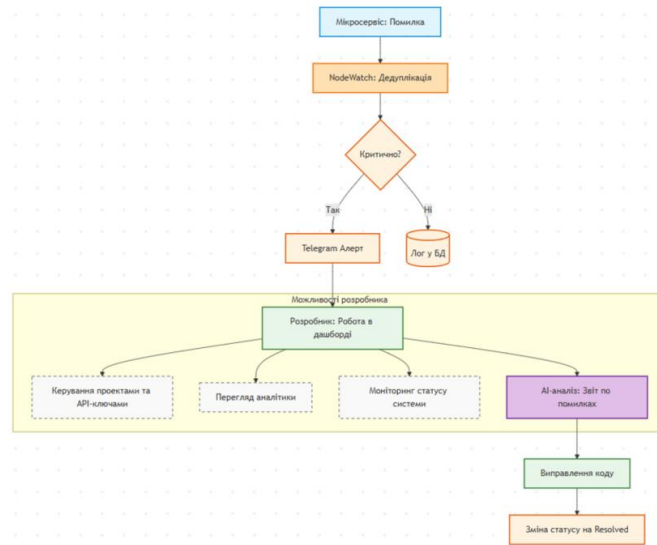
Test Suites: 2 passed, 2 total
Tests:       7 passed, 7 total
Snapshots:  0 total
Time:        1.926 s
Ran all test suites.
PS C:\Users\nikic\Desktop\NodeWatch\backend>
```

Успішне виконання модульних тестів (Jest) та перевірка стабільності API.

- **Модульне тестування:** Перевірка коректності алгоритму дедуплікації (MD5) та парсингу JSON-метаданих.
- **Навантажувальне тестування:** Перевірка стабільності колектора при одночасних запитах від 50+ сервісів.
- **Інтеграційна перевірка:** Верифікація успішного ланцюжка: Колектор -> Gemini API -> Telegram Bot.



ПРАКТИЧНИЙ СЦЕНАРІЙ РОБОТИ



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

12/13

ВИСНОВКИ ТА ПЕРСПЕКТИВИ РОЗВИТКУ

 **Автоматизація моніторингу:** Централізований збір та обробка логів без ручного перегляду файлів.

 **Інтелектуалізація діагностики:** Отримання аналізу помилок та готових порад через Gemini API.

 **Масштабована інтеграція:** Легке підключення будь-яких сервісів завдяки адаптивному API

 **Безпека та контроль:** Модель доступу (RBAC), шифрування ключів та система сповіщень.

 **Аналітика:** Прогнозування збоїв та візуалізація трендів на базі ML.

 **Multi-tenancy:** Повна ізоляція даних для корпоративних клієнтів.

 **Екосистема:** Розробка готових SDK для популярних фреймворків.



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

13/13



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute"