Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Навчально-науковий інститут комп'ютерних наук та штучного інтелекту

Рекомендовано до захисту:
протокол засідання кафедри
№ ___ від ____ ____ 2025 р.
Завідувач кафедри ІПСіТ
_____ Олешко О.І.
(підпис) (прізвище та ініціали)

Пояснювальна записка

до кваліфікаційної роботи бакалавра

на тему Застосування ELK-стеку для аналізу системних даних для виявлення аномалій у процедурах машинного навчання на Python

Захищено на засіданні ЕК № _____
протокол № ___ від ___ ___ 2025 р.
Оцінка ___ / _____
Голова ЕК
_____ Фесенко Г.В.
(підпис) (прізвище та ініціали)

Виконала:
студентка 4 курсу, групи КС-42
спеціальності:
122 Комп'ютерні науки
(шифр і назва спеціальності підготовки)
Гамідова Сабіна Русланівна
(прізвище, ім'я та по батькові, підпис)
Керівник ст.викл. Паршенцев Б. В.
(ступень, звання, прізвище та ініціали, підпис)
Рецензент _____
(ступень, звання, прізвище та ініціали, підпис)

Харків – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В.Н.Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Кафедра інтелектуальних програмних систем і технологій
Рівень вищої освіти (освітньо-кваліфікаційний рівень) бакалавр
Спеціальність F3 Комп'ютерні науки

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри ІПСіТ

_____ Олег ОЛЕШКО
підпис ім'я, прізвище

"____" _____2025 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Гамідової Сабіни Русланівни
(прізвище, ім'я, по батькові студента)

1. Тема роботи: «Застосування ELK-стеку для аналізу системних даних для виявлення аномалій у процедурах машинного навчання на Python».

керівник роботи ст. викладач КІПСіТ Паршенцев Богдан Володимирович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від "16" квітня 2025 року № 4101-5/962

2. Строк подання студентом роботи 02.06.2025 р.

3. Перелік питань, які потрібно розробити:

Ознайомитися з особливостями мікросервісної архітектури в контексті розподілених застосунків та їх логування; провести аналіз сучасних підходів і рішень для централізованого збору логів із мікросервісів; дослідити можливості застосування ELK-стеку та Kafka як основи для побудови системи обробки логів; розробити кастомну бібліотеку логування із підтримкою формату ECS для уніфікованого представлення подій; створити програмне середовище з

мікросервісною архітектурою для проведення експериментів; побудувати Python-модуль для виявлення аномалій у логах за допомогою методів машинного навчання; провести експериментальне дослідження ефективності побудованої системи для виявлення відхилень у поведінці застосунку на основі аналізу логів.

4. План роботи

№ з/п	Назви етапів роботи
1	Формулювання мети та повного переліку завдань для виконання кваліфікаційної роботи (КР).
2	Аналіз джерел з тем мікросервісної архітектури, логування та машинного навчання для програм, які виявляють аномалії.
3	Вивчення принципів централізованого логування з ELK-стеком.
4	Розробка кастомної бібліотеки логування у форматі JSON відповідно до ECS.
5	Реалізація пайплайну збору логів з Kafka → Logstash → Elasticsearch.
6	Налаштування Kibana-дешбордів для візуалізації логів.
7	Збір логів з Elasticsearch та підготовка даних для ML-аналізу.
8	Реалізація алгоритму машинного навчання для виявлення аномалій (на Python).
9	Інтеграція модуля аналізу з основною системою логування.
10	Експериментальне дослідження ефективності виявлення аномалій.
11	Аналіз результатів, формування висновків.
12	Підсумок отриманих результатів та оформлення пакету документів для процедури захисту КР в екзаменаційній комісії ННІ КН та ШІ.

5. Дата видачі завдання 10.12.2024 р.

Студент _____ Гамідова С.Р.
(підпис) (ініціали, прізвище)

Керівник роботи _____ Паршенцев Б.В.
(підпис) (ініціали, прізвище)

АНОТАЦІЯ

Пояснювальна записка охоплює 95 сторінок основного тексту, складається з 5 розділів, містить 24 рисунків, 6 таблиць, 3 додатки та 26 використаних джерел. У додатках подано фрагменти конфігурацій, коду логування, код машинного навчання (моделей).

У роботі розглянуто створення системи централізованого логування на базі ELK-стеку з інтеграцією моделей машинного навчання для виявлення аномалій у мікросервісному застосунку MovieMingle.

Мета — підвищити «спостережуваність», зменшити час реагування на збої та автоматизувати виявлення нетипової поведінки в логах.

Застосовано методи логічного аналізу, експериментального моделювання, візуального й алгоритмічного аналізу логів. Серед використаних інструментів - Spring Cloud, Apache Kafka, Filebeat, Logstash, Elasticsearch, Kibana, Docker і Python. У Python реалізовано моделі Isolation Forest, Local Outlier Factor та KMeans для виявлення уніваріантних і мультиваріантних відхилень.

Результати дослідження показали ефективність інтеграції логування з алгоритмами машинного навчання. Новизна полягає в поєднанні традиційного логування з ML-аналізом у єдиному пайплайні, адаптованому до реального мікросервісного середовища.

Результати можуть бути впроваджені (рекомендується впроваджувати) в ІТ-компаніях, DevOps-командах і використані в навчальних або дослідницьких цілях як приклад комплексного підходу до логування та аналізу логів.

Ключові слова: ЛОГУВАННЯ, МІКРОСЕРВІСИ, ELK, КАКА, АНАЛІЗ ЛОГІВ, ВИЯВЛЕННЯ АНОМАЛІЙ, ISOLATION FOREST, PYTHON, JSON, ВІЗУАЛІЗАЦІЯ, ДОКЕР, SPRING CLOUD, MACHINE LEARNING, KIBANA.

ABSTRACT

The explanatory note comprises 95 pages of the main text, includes 5 chapters, 24 figures, 6 tables, 3 appendices, and 26 sources listed in the references. The appendices contain configuration fragments, logging code, and machine learning model implementations.

This work explores the development of a centralized logging system based on the ELK stack with integrated machine learning models for anomaly detection in the MovieMingle microservices-based application.

The aim of the project is to improve observability, reduce response time to failures, and automate the detection of abnormal behavior in log data.

The study employs methods of logical analysis, experimental modeling, and both visual and algorithmic log analysis. The technological stack includes Spring Cloud, Apache Kafka, Filebeat, Logstash, Elasticsearch, Kibana, Docker, and Python. The Python module implements Isolation Forest, Local Outlier Factor, and KMeans models for detecting univariate and multivariate anomalies.

The results confirm the effectiveness of combining logging with machine learning algorithms. The novelty of the project lies in integrating conventional logging with ML-based analysis into a unified pipeline tailored for real-world microservice environments.

The results are recommended for implementation in IT companies and DevOps teams and can be used in educational or research settings as an example of a comprehensive approach to logging and log analysis.

Keywords: LOGGING, MICROSERVICES, ELK, KAFKA, LOG ANALYSIS, ANOMALY DETECTION, ISOLATION FOREST, PYTHON, JSON, VISUALIZATION, DOCKER, SPRING CLOUD, MACHINE LEARNING, KIBANA.

ЗМІСТ

ПЕРЕЛІК ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

AGPLv3 (Affero General Public License version 3)	– Загальна громадська ліцензія Affero версії 3
AMQP (Advanced Message Queuing Protocol)	– Протокол передавання повідомлень із чергами
API (Application Programming Interface)	– Програмний інтерфейс застосунку
AWS (Amazon Web Services)	– Хмарні сервіси Amazon
CI/CD (Continuous Integration / Continuous Delivery)	– Безперервна інтеграція / Безперервне доставлення
CPU (Central Processing Unit)	– Центральний процесор
CSS (Cascading Style Sheets)	– Таблиці каскадних стилів
CSV (Comma-Separated Values)	– Значення, розділені комами
DSL (Domain Specific Language)	– Мова, специфічна для предметної області
ELK (Elasticsearch, Logstash, Kibana)	– Стек інструментів для пошуку, обробки та візуалізації логів
Faiss (Facebook AI Similarity Search)	– Бібліотека для пошуку векторної схожості від Facebook
F1 (F1-score)	– Гармонічне середнє між точністю (precision) та повнотою (recall)
Git (Git Version Control System)	– Система керування версіями Git
HTML (HyperText Markup Language)	– Мова розмітки гіпертексту
HTTP (HyperText Transfer Protocol)	– Протокол передавання гіпертексту
ID (Identifier)	– Ідентифікатор

IF (Isolation Forest)	– Ліс ізоляції – метод виявлення аномалій
ILM (Index Lifecycle Management)	– Керування життєвим циклом індексів
IQR (Interquartile Range)	– Міжквартильний розмах
IoT (Internet of Things)	– Інтернет речей
IP (Internet Protocol)	– Інтернет-протокол
JSON (JavaScript Object Notation)	– Формат обміну даними JavaScript
JRuby (Java Ruby)	– Реалізація мови Ruby для Java Virtual Machine
JWT (JSON Web Token)	– Веб-токен у форматі JSON
KQL (Kibana Query Language)	– Мова запитів Kibana
LOF (Local Outlier Factor)	– Локальний коефіцієнт викидів
LRD (Local Reachability Density)	– Локальна досяжна щільність
MAD (Median Absolute Deviation)	– Медіанне абсолютне відхилення
ML (Machine Learning)	– Машинне навчання
nmslib (Non-Metric Space Library)	– Бібліотека для пошуку у просторах без метрики
PDF (Portable Document Format)	– Портативний формат документів
PNG (Portable Network Graphics)	– Портативний графічний формат з прозорістю
REST (Representational State Transfer)	– Архітектурний стиль взаємодії систем
S3 (Simple Storage Service)	– Просте хмарне сховище Amazon
SLA (Service-Level Agreement)	– Угода про рівень обслуговування
SQL (Structured Query Language)	– Мова структурованих запитів
SSPL (Server Side Public License)	– Публічна ліцензія для серверного коду
UI (User Interface)	– Користувацький інтерфейс
URL (Uniform Resource Locator)	– Уніфікований локатор ресурсу

YAML (YAML Ain't Markup Language)	– Формат конфігураційних файлів, орієнтований на читання людиною
XML (eXtensible Markup Language)	– Розширювана мова розмітки

ВСТУП

Більшість вебзастосунків, які ми використовуємо щодня - від соцмереж до сервісів перегляду фільмів - працюють не як одна велика програма, а як набір окремих маленьких сервісів, які між собою взаємодіють. Це називається мікросервісною архітектурою. Такий підхід дуже зручний, на прикладі тієї ж платформи для перегляду фільмів: один сервіс відповідає, наприклад, за профіль користувача, інший — за коментарі, ще інший — за рейтинг фільмів. Якщо щось ламається в одному з них, інші можуть працювати далі. Але разом з тим виникає інша проблема — як зрозуміти, що саме пішло не так, якщо всі сервіси живуть своїм окремим життям?

Кожен мікросервіс веде свої «щоденники» — тобто логи. В цих логах записується, що відбувається: хто заходив, що запитував, які помилки виникали. І ось уявимо: якщо у нас 10, 15 або 20 сервісів — то й логів буде стільки ж. Вони можуть зберігатися у різних місцях, мати різний формат, бути нечитабельними або дублювати одне й те саме. У такому хаосі знайти джерело проблеми дуже складно. А якщо сайт щойно «впав», а тисячі користувачів не можуть подивитись фільм — часу на довгі розслідування нема.

Саме тут на допомогу приходить централізоване логування. Це підхід, при якому всі логи з усіх сервісів збираються в одному місці, обробляються, фільтруються та стають зручними для перегляду. Існує цілий набір популярних інструментів, які для цього використовуються разом — один із найпопулярніших сетів — так званий ELK-стек: Elasticsearch, Logstash і Kibana. Перший зберігає інформацію, другий обробляє її на вході, а третій дозволяє переглядати все через зручний інтерфейс — наприклад, графіки кількості помилок, списки логів з фільтрами, тощо.

Але і цього іноді недостатньо. Наприклад, система працює, логи приходять, все виглядає нормально — але раптом зростає кількість помилок або

зменшується швидкість відповіді сервісу. Людина не завжди встигне це помітити. Тому в наш час усе частіше використовують машинне навчання — алгоритми, які вміють самостійно знаходити відхилення у поведінці. Вони аналізують звичайний стан системи і можуть сказати: «щось не так», ще до того, як користувачі почнуть скаржитися.

У цій роботі розглядаємо все це не в теорії, а на прикладі мого реального додатку — MovieMingle, платформи для перегляду фільмів. Додаток має десятки сервісів, серед яких — управління користувачами, профілями, рейтингами, вподобаннями, коментарями, зберігання медіа в Amazon S3 та інші. Для такого застосунку дуже важливо мати систему, яка не просто збирає логи, а ще й допомагає швидко виявляти, коли щось іде не так.

Також було створено власну систему логування: розроблена кастомна бібліотека, яка може працювати з Kafka, обробляти логи в Logstash, зберігати їх у Elasticsearch і візуалізувати через Kibana. Але на цьому ще не все —доданий Python-модуль, який використовує алгоритми машинного навчання (Isolation Forest, LOF тощо), щоби автоматично виявляти аномалії у логах.

Взагалі, актуальність даної теми обумовлена тим, що більшість сучасних IT-систем використовують мікросервісну архітектуру. Проблема логування в таких системах виникає дуже часто — ігнорувати її не можна. Якщо немає централізованого логування, розробники буквально «ходять навпомацки». А якщо немає автоматичного аналізу — будь-яку аномалію можна помітити надто пізно.

Об’єктом дослідження є система логування в мікросервісному застосунку.

Предметом дослідження — застосування стеку ELK разом із засобами машинного навчання для виявлення аномалій у логах.

Мета роботи — створити практичну систему, яка дозволить не лише зручно збирати і переглядати логи, а й вчасно виявляти підозрілу активність або

відхилення в роботі сервісів. Все це буде реалізовано в контексті реального додатку — MovieMingle.

1 АНАЛІЗ СУЧАСНИХ ПІДХОДІВ ДО РОЗРОБКИ ТА ЛОГУВАННЯ У МІКРОСЕРВІСАХ

1.1 Сучасні підходи до архітектури: моноліт vs мікросервіси

На сьогоднішній день при створенні програмного забезпечення архітектура має найкритичніше значення. Саме вона визначає, як компоненти системи взаємодіють між собою, як легко вдаватиметься оновлювати функціонал, масштабувати проект, знаходити помилки та впроваджувати нові технології. Існує два головних архітектурних підходи, що частіше за всі інші використовуються у розробці – монолітна та мікросервісна архітектури.

Що таке монолітна архітектура?

Моноліт – це така архітектура, по якій вся логіка програми зосереджена в одній, єдиній структурі. Усі компоненти, такі як контролери, сервіси, доступ до бази даних, загалом вся бізнес-логіка, об’єднані в одному додатку. Це мається на увазі, що якщо потрібно змінити якусь частину системи, наприклад, якусь логіку в одному з контролерів потрібно переписати, то часто доводиться перекомпілювати та розгортати весь застосунок заново.

Переваги монолітної архітектури:

– легше стартувати розробку, особливо для невеликої команди;

– набагато простіше зібрати та розгорнути один додаток, аніж десятки дрібних сервісів;

– швидший початковий час розробки, бо все в одному місці, не має потреби налаштовувати зв’язки між модулями;

Недоліки монолітної архітектури:

– будь-яка зміна в одному компоненті потенційно може вплинути на інші частини програми;

– з часом система ускладнюється, тому новим розробникам важко розібратися, де і що працює;

– неможливо легко використовувати різні технології в різних частинах системи, усе має бути побудовано за єдиною технологічною платформою;

– тільки вертикальне масштабування, що обмежує адаптивність, пристосовуваність у роботі з великими навантаженнями;

Що таке мікросервісна архітектура?

Мікросервіси – це сучасний підхід, у якому весь застосунок поділяється на невеликі незалежні сервіси, кожен із яких відповідає за свою певну частину бізнес-логіки, наприклад, сервіс авторизації, сервіс перегляду фільмів, сервіс юзерів тощо). Кожен сервіс – автономний, самодостатній і може розроблятися, тестуватися та розгортатися окремо від інших.

Серед основних рис міксросервісів можу виділити такі:

– за патерном «база даних для сервісу» - кожен сервіс має власну базу даних, що дозволяє уникати залежностей;

– усі сервіси взаємодіють через мережу, використовуючи API або HTTP-запити;

– можна масштабувати лише той сервіс, який «відчуває» навантаження, замість масштабування всього застосунку;

– можна використовувати різні мови програмування, фреймворки, бази даних у різних сервісах;

– таку архітектуру легше впроваджувати у хмарному середовищі – сервіси легко масштабуються горизонтально (запускати кілька копій одного сервісу одночасно);

Переваги мікросервісної архітектури:

– кожен сервіс має чітке функціональне призначення – легше тестувати, змінювати, оновлювати;

– свого роду незалежність компонентів дозволяє оновлювати один сервіс без зупинки всього застосунку;

– можна залучати окремі команди до окремих сервісів, це призводить до розподілу обов’язків та вищої масштабованості команди;

– простіше впровадження CI/CD, автоматизоване тестування, логування та моніторинг;

Недоліки мікросервісної архітектури:

– складніший старт розробки, бо потрібно продумати взаємодію між сервісами, безпеку, конфігурацію;

– необхідна певна інфраструктура, а саме сервіс конфігурацій, служба відкриття (discovery), логування, трасування запитів;

– комунікація між сервісами через HTTP може бути повільнішою та складнішою в дебагінгу і не тільки;

Виходячи з усього цього, можна зробити коротке порівняння даних архітектур.

Таблиця 1.1.1 – Порівняльна таблиця монолітної та мікросервісної архітектур

Критерій	Моноліт	Мікросервіси
Масштабування	Вертикальне	Горизонтальне по сервісах
Залежність компонентів	Тісно пов’язані	Незалежні
Технологічна адаптивність	Один стек технологій	Можна змішувати різні
Розгортання	Один великий додаток	Окреме для кожного сервісу
Обслуговування	Важко змінювати окремі частини	Легко оновлювати локальні зміни
Початкова складність	Менша	Вища
Спостережуваність	Простіша логіка	Потрібні централізовані інструменти

Мій вибір (для мого додатку – платформа для перегляду фільмів MovieMingle) – це мікросервісна архітектура. Після детального аналізу особливостей обох архітектурних підходів я обрала саме такий варіант, бо:

– моя платформа MovieMingle складається з кількох незалежних модулів, а саме – управління профілями користувачів, рейтингами, коментарями, вподобаннями тощо. Кожен із цих модулів легко виділити в окремий мікросервіс;

– я хочу мати відмовостійку систему, тобто щоб відмова одного модуля не вплинула на всю платформу. Наприклад, якщо сервіс коментарів буде недоступний, користувач все одно зможе переглядати фільми та ставити свої оцінки;

– завдяки мікросервісам я можу масштабувати лише ті сервіси, які найбільше навантажені (наприклад, сервіс фільмів);

– у моїй архітектурі використовується шлюз API Gateway, система відкриття Eureka, централізоване логування через ELK-стек, та адаптивне, гнучке зберігання конфігурацій через Spring Cloud Config, усе це органічно вписується саме в мікросервісний підхід;

1.2 Технологічні рішення для логування в мікросервісах

Коли система побудована за мікросервісною архітектурою, кожен сервіс працює незалежно, виконує окрему функцію та має власні логи. Це створює складність, тому що логи зберігаються окремо, їх важко зібрати разом і побачити повну картину того, що відбувається в системі. Через це виникає потреба у централізованому логуванні - це коли всі логи з мікросервісів потрапляють в одне місце, де їх можна зручно зберігати, переглядати і аналізувати.

Для цього використовуються спеціальні інструменти. Найпоширеніше рішення - **ELK-стек**. Він складається з трьох компонентів:

– **Logstash** відповідає за збір логів із різних сервісів;

– **Elasticsearch** зберігає логи у структурованому вигляді, що дозволяє швидко їх шукати;

– **Kibana** надає зручний інтерфейс для візуалізації логів — графіки, фільтри, дашборди тощо;

Це підходить, коли в системі багато різних сервісів, і потрібно швидко, постійно знаходити помилки або відстежувати активність користувачів.

Окрім ELK, є й інші варіанти. Наприклад, **Graylog**, який також збирає логи з різних джерел і має власний інтерфейс для перегляду. Він простіший у налаштуванні, але менш гнучкий. Інший варіант — **Splunk**, який має великий функціонал і гарну підтримку, але зазвичай вимагає ліцензії.

Важливу роль також відіграють агенти – це такі програми, які передають логи. Наприклад, **Fluentd** - дуже гарний адаптивний інструмент з багатьма плагінами для збору та обробки логів. Його часто порівнюють із Logstash, і кожен має свої переваги залежно від задачі.

Щоб не втратити логи під час збоїв або пікових навантажень, дані часто спочатку передаються в **чергу повідомлень**, таку як **Kafka**. Вона виступає чимось на кшталт буфером, завдяки чому система не пропускає події, навіть якщо основне сховище тимчасово недоступне.

Також важливо, щоб усі логи мали єдиний формат. Це полегшує аналіз і дозволяє відслідковувати, як обробляється запит через усю систему - від одного сервісу до іншого.

У моїй платформі MovieMingle я використовую саме ELK-стек з Kafka, бо це достатньо надійне рішення. Воно добре підходить під задачі масштабованого мікросервісного застосунку та дозволяє швидко знаходити причини збоїв, викидів.

1.3 Роль Apache Kafka у транспортуванні логів

У звичайних вебзастосунках або монолітах передача даних між модулями зазвичай відбувається напряму - один компонент робить запит до іншого і чекає на відповідь. Це називається синхронна взаємодія. Але коли мова йде про мікросервіси, де кожен працює окремо, така модель уже не завжди зручна.

Уявімо ситуацію: один сервіс платформи MovieMingle, наприклад, відповідає за обробку фільмів або логування, хоче передати повідомлення одразу кільком іншим сервісам. Якщо це зробити напряму - потрібно знати адреси кожного, хто має отримати це повідомлення, перевірити, чи вони активні, чи не зависли тощо. А якщо один із них не працює - запит може «зависнути» або

втратитися. Це сильно ускладнює систему й створює залежність між сервісами, тоді якщо буде «зависання», то це стосується всієї логіки, бо є взаємозв’язок.

Щоб уникнути таких проблем, у мікросервісній архітектурі часто використовують **асинхронну передачу повідомлень** - і ось тут у справу входить **Apache Kafka**. У випадку MovieMingle Kafka використовується як транспорт для логів: замість того щоб напряму надсилати логи в систему зберігання або аналітики, мікросервіси просто відправляють їх у Kafka. А вже інші сервіси (наприклад, Logstash) отримують ці повідомлення з Kafka, коли готові.

Kafka працює як така собі «черга повідомлень», або навіть краще сказати - **«поштовий ящик», який ніколи не втрачає листів**. Сервіси MovieMingle, які надсилають логи, є **продюсерами** (producer), а ті, що їх зчитують - **споживачами** (consumer). Продюсери просто надсилають події (наприклад, «Користувач залишив коментар», «Стався виняток у сервісі рейтингу») у певну **тему (topic)** Kafka, і більше ні про що не турбуються. Вони не знають, хто буде читати цю подію і коли саме - головне, що вона збережена.

Це дозволяє **роз'єднати сервіси між собою**, зробити їх незалежними. Наприклад, якщо Logstash або Elasticsearch тимчасово недоступні - Kafka все одно зберігає лог, і він буде оброблений пізніше. Навіть якщо споживачів стане більше - Kafka без проблем обробить їх одночасно, розподіливши події між ними. Також важливо отмітити, що події у Kafka зберігаються у спеціальних «розділах» (partitions), що дозволяє читати їх **паралельно** - тобто система масштабується й не просідає по швидкості.

Також Kafka підтримує **реплікацію даних** - тобто кожне повідомлення зберігається одразу на кількох серверах (Kafka-брокерах), щоб у разі збою нічого не загубилося. Тобто навіть якщо один із брокерів «впаде», дані все одно будуть доступні з іншого. Це важливо, бо в логах можуть бути критично важливі дані для аналітики або безпеки.

Ще одна велика перевага — **розширюваність**. Наприклад, зараз платформа надсилає логи лише для зберігання. Але в майбутньому може з’явитись ще один мікросервіс, який аналізує логи через машинне навчання. І все, що йому потрібно - просто підписатися на ту ж Kafka-тему, і він буде бачити всі логи автоматично, без змін у логіці решти системи.

Тож Kafka у моєму проекті виконує роль **центрального посередника**, який приймає всі події від мікросервісів і передає їх туди, де вони потрібні - швидко, й без ризику втрати.

1.4 Аналіз компонентів ELK-стеку

OpenSearch vs Elasticsearch

При розробці системи централізованого логування для моєї платформи MovieMingle постає важливе завдання вибору між двома основними системами пошуку та аналітики - OpenSearch та Elasticsearch. Обидві платформи є інструментами для збору, індексації, пошуку та візуалізації логів, однак мають різні підходи до ліцензування, архітектури, масштабованості та підтримки з боку спільноти. Для того, щоб вирішити, що ж все ж таки використовувати у моєму додатку, розглянемо порівняльну характеристику даних технологічних рішень.

Історичний контекст

Elasticsearch був запущений у 2010 році як частина відомого стеку ELK (Elasticsearch, Logstash, Kibana) і швидко здобув популярність завдяки своїй адаптивності та продуктивності. Проте у 2021 році компанія Elastic змінила модель ліцензування з Apache 2.0 на власну комерційну (Elastic License та SSPL), що спричинило появу OpenSearch - повністю відкритої альтернативи, створеної Amazon на базі останньої відкритої версії Elasticsearch (7.10.2).

Після цього обидва продукти почали розвиватися окремо. Elasticsearch отримав доволі потужні корпоративні функції (наприклад, вбудовану аналітику, машинне навчання, розширену безпеку), але частина з них доступна лише в платних версіях. Водночас OpenSearch залишився вільним і відкритим для всіх охочих, отримуючи активну підтримку з боку спільноти, зокрема під егідою Linux Foundation.

Функціональні можливості

Архітектурно обидва рішення базуються на Apache Lucene і підтримують розподілену обробку даних із шардованою структурою - це означає, що обидві системи можуть ефективно працювати з великими обсягами логів та складними запитами. Проте підходи до реалізації функцій дещо відрізняються:

- **Elasticsearch** надає розширену підтримку для enterprise-рішень: засоби спостереження, інтеграцію з машинним навчанням, просунуту безпеку, візуалізацію через Kibana тощо;
- **OpenSearch** зосереджується на відкритості, тобто має власні інструменти для автентифікації, алертів, керування індексами та підтримку векторного пошуку, він особливо добре інтегрується з сервісами AWS, як-от OpenSearch Service;

Продуктивність і масштабованість

Тестування, проведені у 2024-2025 роках, показують, що Elasticsearch, як правило, демонструє кращу продуктивність у складних сценаріях запитів - від 40% до 140% швидше, ніж OpenSearch, із меншими витратами ресурсів. Це пов’язано з тим, що Elastic інвестує значні зусилля у власну оптимізацію та підтримку. Однак у невеликих або середніх системах ця різниця може бути не суттєвою, і OpenSearch часто вважається «достатньо хорошим» рішенням.

Ліцензування

Одним із головних моментів при виборі платформи є тип ліцензії:

– **OpenSearch** - повністю відкрите програмне забезпечення за ліцензією Apache 2.0, що дозволяє вільно використовувати, змінювати й поширювати код без ризику потрапити в залежність від одного постачальника;

– **Elasticsearch** - використовує декілька моделей: SSPL, Elastic License, а також нещодавно додану AGPLv3; частина функцій доступна лише у комерційних версіях;

Для систем, що орієнтовані на відкритість і уникають vendor lock-in, OpenSearch є привабливим вибором, водночас у критичних бізнес-сценаріях Elasticsearch може бути вигіднішим через просунутіші функції та підтримку.

Нові можливості: векторний пошук і AI

У 2025 році обидві системи вже мають підтримку векторного пошуку - це особливо актуально для проектів, пов’язаних із генеративним AI, semantic search або recommendation-системами:

– **Elasticsearch** має нативну підтримку векторного пошуку з оптимізаціями під ML-навантаження;

– **OpenSearch** використовує бібліотеки Faiss та nmslib, що дозволяють обробляти вектори до 16 000 вимірів, хоч і поступається за швидкістю;

Хмарні рішення

Обидві системи можна розгорнути локально або використовувати як хмарну послугу:

– **Amazon OpenSearch Service** - повністю інтегрована з AWS, проста в розгортанні й обслуговуванні, але орієнтована лише на один хмарний провайдер;

– **Elastic Cloud** - доступний для AWS, Google Cloud і Microsoft Azure, забезпечує більше адаптивності для мультихмарної стратегії, хоч і вимагає більшого бюджету;

Мій вибір

Після глибокого аналізу я дійшла висновку, що **для логування в системі MovieMingle найбільш підходящим є саме Elasticsearch**. Причина проста: для мого випадку важлива максимальна продуктивність, інтеграція з Kibana, стабільність, підтримка enterprise-функцій, можливість використовувати машинне навчання - і все це я отримую у зручному вигляді в рамках Elastic Stack. До того ж, навіть у безкоштовній версії Elasticsearch надає достатньо функціональності для ефективного логування, особливо якщо правильно налаштувати логгер, індексацію та візуалізацію.

Таблиця 1.4.1 - Порівняння характеристик Elasticsearch та OpenSearch

Критерій	**Elasticsearch**	**OpenSearch**
Ліцензія	SSPL / Elastic / AGPLv3	Apache 2.0
Швидкодія	Вища (до 140% швидше)	Задовільна
Підтримка ML	Так (вбудована)	Обмежено через сторонні бібліотеки
Векторний пошук	Нативний	Через Faiss / nmslib
Хмарна інтеграція	Multi-cloud (AWS, GCP, Azure)	AWS OpenSearch Service
Візуалізація	Kibana	OpenSearch Dashboards
Платна функціональність	Частково	Відсутня

Підсумовуючи, обидві системи мають свої переваги й недоліки, але в умовах мікросервісної архітектури з підвищеними вимогами до швидкодії, адаптивності аналізу логів та візуалізації подій **Elasticsearch виявився найкращим варіантом для мого проекту**.

Kibana vs Grafana

Також потрібно визначитись, що краще використати для візуалізації - Kibana чи Grafana. Обидва сервіси мають відкритий код, активні спільноти й підтримують створення дашбордів, однак їхні основні цілі та підходи суттєво відрізняються. Оскільки в моєму застосунку важливо було не лише показувати технічні метрики, а й **аналізувати логи з Elasticsearch**, шукати **аномалії** та формувати **візуалізації логів**, я провела детальне порівняння, щоб обґрунтовано обрати інструмент, який найкраще підходить саме для таких задач.

Kibana - це інструмент для візуалізації логів і аналітики, який є частиною стеку **ELK**. Він розроблений спеціально для роботи з великими обсягами логових даних, які надходять із сервісів, обробляються Logstash і зберігаються в Elasticsearch. Завдяки цьому, Kibana дозволяє будувати адаптивні запити до логів, створювати графіки, діаграми, геокарти, таблиці та навіть виявляти закономірності або **аномалії у поведінці систем**.

Окрім стандартної візуалізації, Kibana підтримує:

- власну мову запитів **KQL**, зручну для легкого аналізу логів;
- фільтри, агрегації, побудову часових серій та трендів;
- інтеграцію з **Alerting**, Machine Learning і Security-модулями Elastic;
- збереження та публікацію дашбордів у вигляді PDF або PNG;
- підтримку геоданих (наприклад, карта входів користувачів);

Це робить Kibana незамінною для **такого рода аналізу**, виявлення проблем і причин помилок. Це ідеально вписується у мікросервісну архітектуру з великою кількістю взаємодій між сервісами.

Grafana - це ще один популярний інструмент візуалізації, орієнтований більше на **моніторинг технічних метрик у реальному часі**. Його головна перевага - адаптивна підтримка різноманітних джерел даних: Prometheus, InfluxDB, Graphite, MySQL, PostgreSQL та навіть Elasticsearch. Grafana створена переважно для **метричних даних**, таких як використання CPU, кількість запитів на секунду, об’єм оперативної пам’яті тощо.

Grafana також підтримує:

- детальні кастомні панелі для дашбордів;
- потужну систему сповіщень (Slack, email, Telegram, Webhook);
- шаблони дашбордів і зручний інтерфейс налаштування;
- плагіни для додаткових візуалізацій;
- автоматичне створення (provisioning) дашбордів і джерел даних;

Чому для MovieMingle обрана саме Kibana?

Оскільки основна мета логування в MovieMingle - **виявлення аномалій у поведінці мікросервісів**, **запис помилок**, **аналіз подій**, що надходять із Kafka через Logstash до Elasticsearch - саме Kibana надала найзручніші інструменти для цих задач. Можливість писати запити до логів, комбінувати фільтри, переглядати часові лінії подій і розміщувати все це в одному дашборді дозволяє мені **продуктивно аналізувати логічні патерни, шукати збої та порушення**.

Grafana, хоч і має красиву візуалізацію та зручне налаштування, більше підходить для **метричного моніторингу**, а не для **подібного аналізу логів**, якого потребує мій додаток.

Таблиця 1.4.2 - Порівняльна таблиця Kibana vs Grafana

Характеристика	Kibana	Grafana
Основне призначення	Аналіз логів з Elasticsearch	Візуалізація метрик у реальному часі
Джерела даних	Elasticsearch	Prometheus, InfluxDB, SQL, Elasticsearch тощо
Типи візуалізацій	Лінії, стовпці, пай, хмари тегів, heatmap, карти	Лінії, heatmap, singlestat, gauge, freetext
Підтримка запитів	KQL, Lucene, Elasticsearch DSL	Query editor на основі джерела
Візуалізація логів	Наявна повна	Обмежена
Геопросторові карти	Наявні повні	Обмежено
Сповіщення	Через Watcher, ElastAlert, X-Pack	Вбудовані, з умовами та Webhooks
Плагіни та розширення	Обмежена кількість	Дуже велика екосистема
Зручність налаштування	Середня (через YAML)	Висока (через UI або .ini файли)
Повнотекстовий пошук	Так	Ні
Підтримка часових рядів	Обмежена	Повна
Вартість	У складі ELK Stack (платні та безкоштовні версії)	Є безкоштовний тариф, Pro від $8/місяць
Спільнота	Велика (Elastic)	Дуже активна (Grafana Labs)

Fluentd vs Logstash

Ще один вибір інструменту, який буде відповідати всім вимогам: збір логів з різних мікросервісів, їхня обробка, фільтрація, збагачення та передача в Elasticsearch для подальшого аналізу в Kibana. Основними варіантами були **Fluentd** і **Logstash**. Обидва інструменти широко, часто використовуються, однак вони мають різні підходи, архітектуру та можливості.

Що таке лог-колектори і навіщо вони потрібні?

Коли в системі багато сервісів (як у MovieMingle), кожен з них генерує власні логи: про помилки, події, запити, відповіді тощо. Якщо ці логи не збирати централізовано, знайти причину збою або простежити, що сталося з користувачем у певний момент, практично неможливо. Тут на допомогу приходять лог-колектори. Вони забирають логи з усіх джерел, фільтрують, формують зручну структуру й відправляють у центральне сховище (у моєму випадку - Elasticsearch).

Fluentd - це інструмент з відкритим кодом, який розробила компанія Treasure Data. Його головна сила - гнучкість завдяки **плагінам** (більше 500), які дозволяють працювати з різними джерелами даних. Fluentd чудово справляється з потоками даних у реальному часі, підтримує збагачення, буферизацію та збереження при збоях. Він підходить для IoT, контейнеризованих додатків, систем з великою кількістю подій. Його конфігурація відбувається через файли YAML, а маршрутизація даних - через «теги». Підтримує збереження логів у пам'яті, на диск або в хмару.

Logstash - один з основних компонентів **ELK-стеку** (Elasticsearch, Logstash, Kibana). Це також інструмент з відкритим кодом, який чудово працює з логами, що надходять з серверів, додатків, хмарних сервісів. Він підтримує

величезну кількість плагінів (більше 250) для обробки, фільтрації, збагачення даних. Одним з головних плюсів Logstash є **тісна інтеграція з Elasticsearch**, що значно полегшує подальший аналіз даних у Kibana. Logstash використовує «конвеєри» з трьох частин: input, filter, output. Завдяки плагінам grok, mutate, dissect можна зручно витягувати потрібні поля з логів, формувати структуру й передавати далі.

Чому я обрала саме Logstash?

Хоча **Fluentd** є більш легким та простим для початку, **Logstash** забезпечує **глибшу та точнішу обробку логів**, особливо для складних форматів і сценаріїв. У платформі MovieMingle дані надходять з різних мікросервісів: профілі, фільми, рейтинги, вподобання, коментарі. Кожен мікросервіс має свої особливості у логах, тому мені було важливо:

- зручно парсити й структурувати нестандартні логи (тут grok - ідеальний);
- мати **стабільну доставку даних в Elasticsearch**;
- використовувати **один стек технологій** - Kibana вже в проекті, тому Logstash природно підходить;
- мати можливість масштабування і потужну підтримку від Elastic;

Також Logstash добре працює з Kafka, яку я використовую для транспортування логів. Конфігурація трохи складніша, але як результат, то це є надійна система логування, яка не потребує зовнішніх залежностей для буферизації або збереження даних під час збоїв.

Таблиця 1.4.3 - Порівняльна таблиця Fluentd vs Logstash

Критерій	Fluentd	Logstash
Платформа	C + Ruby, кросплатформенний	JRuby + Java, кросплатформенний
Продуктивність	Висока (менше ресурсів)	Висока (але більший обсяг пам'яті)
Плагіни	>500, децентралізовані	>250, централізовані в Elastic
Парсинг логів	Вбудовані парсери (JSON, regex, CSV)	Потужний grok + mutate
Маршрутизація подій	Теги	Умови (if-else)
Транспортування	Буферизація (диск/пам’ять/хмара)	Черга + інтеграція з Kafka/Redis
Інтерфейс	Є веб UI	Конфіг файли + Kibana як інтерфейс
Ліцензія/Ціна	Apache 2.0 (безкоштовно)	Apache 2.0 (безкоштовно)

1.5 Порівняльний аналіз систем централізованого логування: ELK-стек vs Splunk vs Graylog

Тепер прийшов час порівняти цілісний ELK-стек з альтернативами. У MovieMingle я детально ознайомилася з трьома основними інструментами для централізованого логування - це **ELK Stack**, **Splunk** та **Graylog**. Кожен з них має свої переваги, але для цілей мого застосунку найбільш зручним і гнучким виявився саме **ELK стек**, про що розповім далі.

Ще раз**, ELK Stack** - це комбінація трьох компонентів: Elasticsearch, Logstash і Kibana. Вони працюють разом як єдина система. Logstash відповідає за збір логів із різних джерел, обробку та передачу далі. Elasticsearch зберігає ці дані

у вигляді, зручному для швидкого пошуку, а Kibana дозволяє бачити все це в зручному веб-інтерфейсі - у вигляді графіків, діаграм, таблиць тощо.

Що мені найбільше сподобалося — це свого рода адаптивність. Наприклад, я можу налаштувати власні дашборди в Kibana, змінювати вигляд графіків або фільтрувати лише ті події, які мене цікавлять. Також можна легко розширити систему через плагіни або підключити машинне навчання для виявлення аномалій, що є великою перевагою для мого проєкту.

Splunk - це зручний та міцний інструмент. Він також вміє збирати логи, індексувати їх, дозволяє працювати з даними за допомогою спеціальної мови запитів. Багато хто обирає саме його через зручний інтерфейс і гарну підтримку. Але в нього є один великий мінус - він є платним (є безкоштовна версія, але з суттєвими обмеженнями), і чим більший обсяг даних, тим більше доведеться платити. Тому якщо в проекті багато мікросервісів, як у мене, — Splunk швидко стане дорогим задоволенням.

Graylog - це також хороше рішення, яке зосереджене саме на зборі і базовому аналізі логів. У ньому є дашборди, можна налаштовувати алерти (сповіщення), інтегрувати різні джерела логів. Він простий у налаштуванні, не потребує багато ресурсів і також є open-source. Але в порівнянні з ELK, Graylog менш масштабований і обмежений у можливостях візуалізації та розширення. Для невеликих застосунків або початку роботи з логами - він може бути вдалим вибором, але, як на мене, то цьому інструменту не вистачає гнучкості, адаптивності.

У підсумку, хоча всі три системи виконують подібні завдання, я вважаю (та обрала саме його), що саме **ELK Stack** найкраще підходить для великих і

розгалужених застосунків, де важлива масштабованість, візуалізація, точність пошуку і можливість розширення функцій.

Таблиця 1.5.1 - Порівняльна таблиця: ELK vs Splunk vs Graylog

Критерій	**ELK Stack**	**Splunk**	**Graylog**
Основна функція	Збір, агрегація і аналіз логів	Збір, аналіз і візуалізація великих обсягів даних	Централізований збір і аналіз логів
Масштабованість	Висока (можна обробляти великі обсяги даних)	Висока (але потребує більше ресурсів і коштів)	Середня
Удобство використання	Середнє (потрібне налаштування Logstash/Kibana)	Високе (простий інтерфейс)	Високе (легко налаштовується)
Пошук і аналіз	Потужний пошук через Elasticsearch, фільтри, агрегації	Складні пошукові запити, зручні алерти	Ефективний пошук і базові алерти
Візуалізація	Гнучкі дашборди, графіки, інтерактивна робота з Kibana	Потужна візуалізація, кастомізовані звіти	Стандартні графіки і дашборди

Закінчення таблиці 1.5.1

Розширюваність	Висока (велика кількість плагінів і інтеграцій)	Гнучка модульна система плагінів	Підтримка плагінів, але менша кількість
Безпека	Є можливості шифрування, керування доступом	Розширені функції безпеки (але у платній версії)	Є базові механізми доступу і фільтрації
Ліцензування	Open Source (з комерційними доповненнями)	Пропрієтарна (є безкоштовна обмежена версія)	Open Source (з опціями платних доповнень)

1.6 Висновки з аналітичного огляду

Після детального аналізу сучасних підходів до архітектури та логування в мікросервісах, можна зробити кілька важливих висновків.

По-перше, мікросервісна архітектура виявилась набагато зручнішою для розробки масштабованих систем, у порівнянні з класичним монолітом. Так, вона вимагає більше налаштувань і має певну складність у плані взаємодії між сервісами, зате дає змогу кожному компоненту працювати незалежно, оновлюватись окремо й не залежати від інших.

Що стосується логування, то тут ситуація теж змінилася. Якщо в монолітах достатньо було просто писати логи у файл або консоль, то в мікросервісах усе ускладнюється. Логи надходять із різних джерел, у різному форматі, їх треба збирати, обробляти, фільтрувати та зберігати так, щоб потім можна було швидко знайти потрібну інформацію.

Я розглянула кілька інструментів для централізованого логування й порівняла їх між собою. Зокрема, вивчала можливості Fluentd і Logstash як колекторів логів, а також різні платформи - Graylog, Splunk та ELK-стек. Після аналізу переваг і недоліків стало зрозуміло, що для мого додатку найкраще підходить саме **ELK-стек**. Він добре масштабується, дозволяє адаптивно налаштувати обробку логів, має міцний інструмент візуалізації (Kibana), і, що важливо (саме для мене), є відкритим і безкоштовним для використання в базовому варіанті.

Окремо варто відзначити роль Apache Kafka саме в контексті асинхронного збору логів, яку я також детально дослідила. Kafka дозволяє організувати передачу лог-файлів таким чином, щоб сервіси не надсилали їх безпосередньо до систем обробки (наприклад, Logstash чи Elasticsearch), а передавали в спеціальну чергу повідомлень. Це означає, що логування відбувається асинхронно - сервіси не витрачають час на очікування обробки логів і можуть продовжувати виконувати свою основну логіку - без затримок. Такий підхід значно зменшує навантаження на сервіси та підвищує надійність загального процесу логування. Саме тому у своєму додатку я вирішила використати Apache Kafka як проміжну ланку для збору та передачі логів до наступних компонентів системи моніторингу.

У результаті, цей аналітичний огляд допоміг мені чітко сформулювати технічні рішення для подальшої реалізації системи логування: вибір на користь мікросервісів, впровадження Kafka для передачі логів і використання ELK-стеку для збирання, обробки та візуалізації інформації.

2 ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ ІНФРАСТРУКТУРИ MOVIEMINGLE

2.1 Проектування бази даних

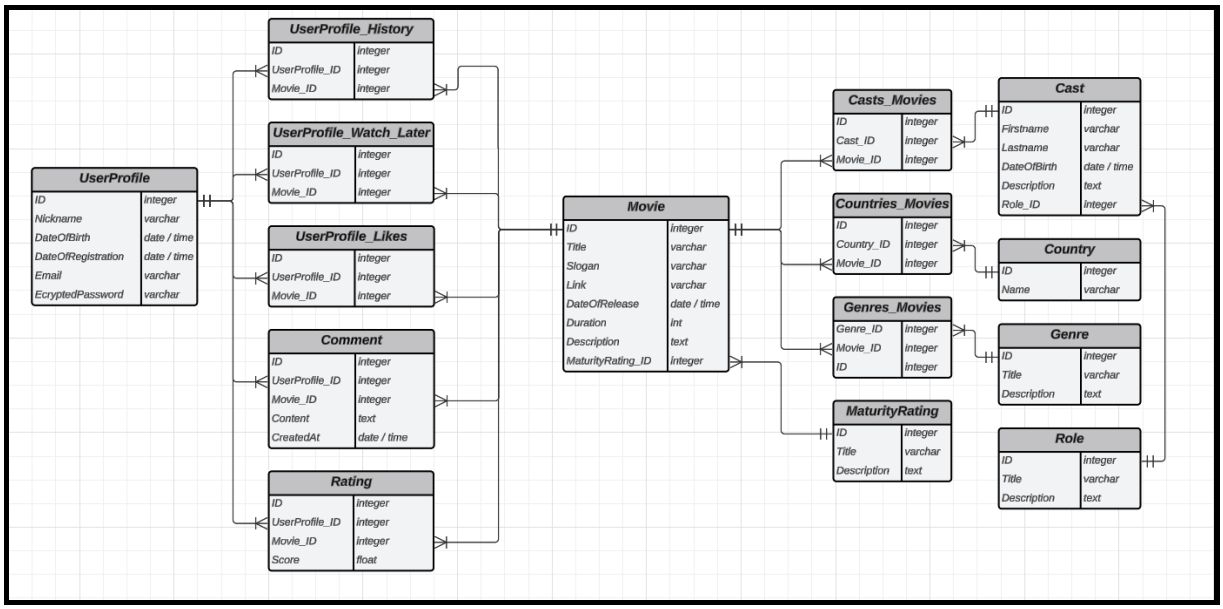


Рисунок 2.1.1 – ER-діаграма бази даних застосунку MovieMingle

У центрі схеми - таблиця Movie, яка зберігає основну інформацію про фільми: назву, слоган, посилання на сторінку перегляду, дату виходу, тривалість, опис та рівень вікових обмежень (через зовнішній ключ на таблицю MaturityRating). Ця таблиця є основною, оскільки вона безпосередньо пов’язана з великою кількістю інших таблиць - користувацькою активністю, акторським складом, країнами, жанрами тощо.

Кожен користувач платформи представлений у таблиці UserProfile, яка містить основні атрибути: нікнейм, дату народження, дату реєстрації, електронну пошту та зашифрований пароль. Усі дії користувача щодо взаємодії з фільмами реалізовані через окремі таблиці, пов’язані з UserProfile та Movie. Зокрема:

– UserProfile_History - відображає історію переглядів фільмів користувачем;

- UserProfile_Watch_Later - список запланованих до перегляду фільмів;
- UserProfile_Likes - фільми, які сподобались користувачу;
- Comment - зберігає текстові коментарі користувачів до фільмів, а також дату створення;
- Rating - дозволяє користувачам залишати оцінки (у вигляді числа з плаваючою комою) для кожного фільму;

Така декомпозиція користувацької активності на окремі таблиці допомагає підтримувати базу в нормалізованому вигляді, що означає, що одна й та сама інформація зберігається лише в одному місці, без повторів у різних частинах структури. Через це зменшується надлишковість даних. Наприклад, інформацію про користувача не потрібно дублювати в таблицях історії переглядів, вподобань чи коментарів - достатньо посилання на його ідентифікатор. Такий підхід також значно спрощує оновлення, в плані, якщо змінюється якась інформація про користувача (наприклад, нікнейм чи email), її потрібно змінити лише в одній таблиці, і всі пов’язані з нею записи залишаються актуальними без необхідності повторного редагування в інших місцях бази.

Ще один важливий блок - **акторський склад фільмів**. Таблиця Cast містить інформацію про акторів (ім’я, прізвище, дату народження, опис), а також зовнішній ключ на таблицю Role, яка вказує на роль (наприклад, актор або режисер). Зв’язок між фільмами та акторами реалізовано через таблицю Casts_Movies, що виконує функцію зв’язуючої таблиці для реалізації зв’язку багато-до-багатьох.

Аналогічно організовано зв’язки з країнами (Countries_Movies) та жанрами (Genres_Movies). Таким чином, кожен фільм може мати **декілька жанрів** та бути **створеним у кількох країнах**.

Окрема таблиця MaturityRating містить рівень вікових обмежень (наприклад, 0+, 12+, 18+), який зв’язується з фільмами через поле

MaturityRating_ID. Це дає змогу обмежувати доступ до окремих фільмів залежно від віку користувача.

У результаті спроектована база даних вийшла достатньо практичною для використання в умовах мікросервісної архітектури. Завдяки розділенню інформації на логічні блоки вдалося досягти більш-менш зрозумілої структури, що дозволяє легко працювати як з даними про користувачів і їхню активність, так і з фільмами, акторами, жанрами, країнами тощо. Для зв'язків типу «багато до багатьох» використано окремі таблиці, що не тільки допомагає уникнути дублювання інформації, а й дозволяє підтримувати дані в цілісному вигляді.

2.2 Основні мікросервіси та їх функціональні можливості

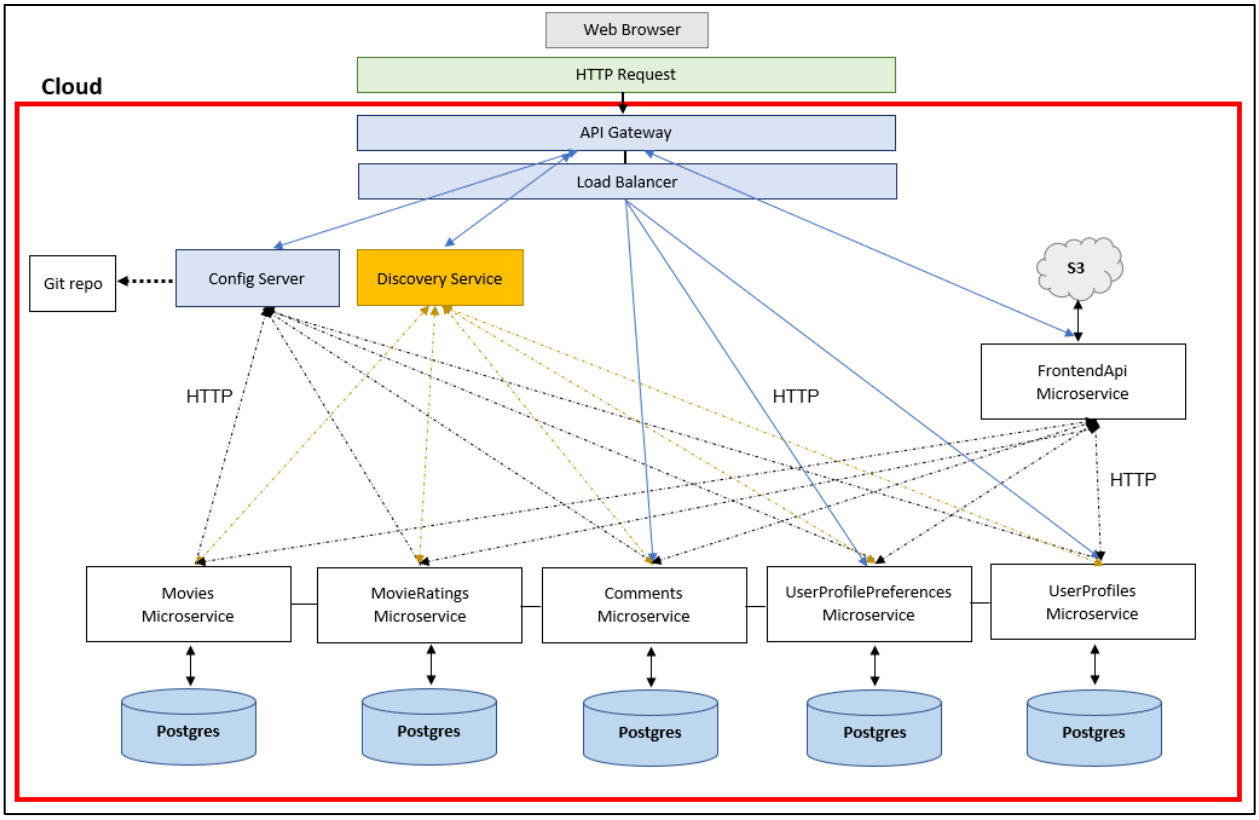


Рисунок 2.2.1 – Скріншот загальної архітектури мікросервісної системи MovieMingle

Взагалі, у цьому проекті я реалізувала повноцінну мікросервісну архітектуру, де кожен мікросервіс виконує свою окрему функцію. Вся система побудована так, щоб була масштабованою, зрозумілою та адаптивною в налаштуванні. Запити користувача завжди починаються з браузера, але далі вони не йдуть одразу в якийсь мікросервіс - замість цього всі HTTP-запити обробляє API Gateway. Цей компонент виконує дуже важливу роль: по-перше, він контролює, куди саме має піти запит (тобто перенаправляє його в потрібний мікросервіс), а по-друге - через нього простіше реалізувати безпеку, логування, обмеження доступу та аналіз трафіку.

Далі є Load Balancer - балансувальник навантаження. Його завдання - розподіляти вхідні запити між кількома екземплярами сервісів, щоб не було перевантаження. Якщо, наприклад, у якийсь момент сервіс фільмів отримує дуже багато запитів, Load Balancer автоматично передає частину з них на інші його копії. Це потрібно для стабільності роботи всієї системи.

Два важливих компоненти, які сильно спрощують управління мікросервісами - це Config Server і Discovery Service.

Discovery Service (Eureka) - це як телефонна книга, яка зберігає, де саме (на яких адресах і портах) зараз доступні всі мікросервіси. Завдяки цьому, сервіси можуть «знаходити» один одного автоматично, без хардкоджених IP. Це особливо зручно при деплойменті в хмару, де адреси можуть змінюватися.

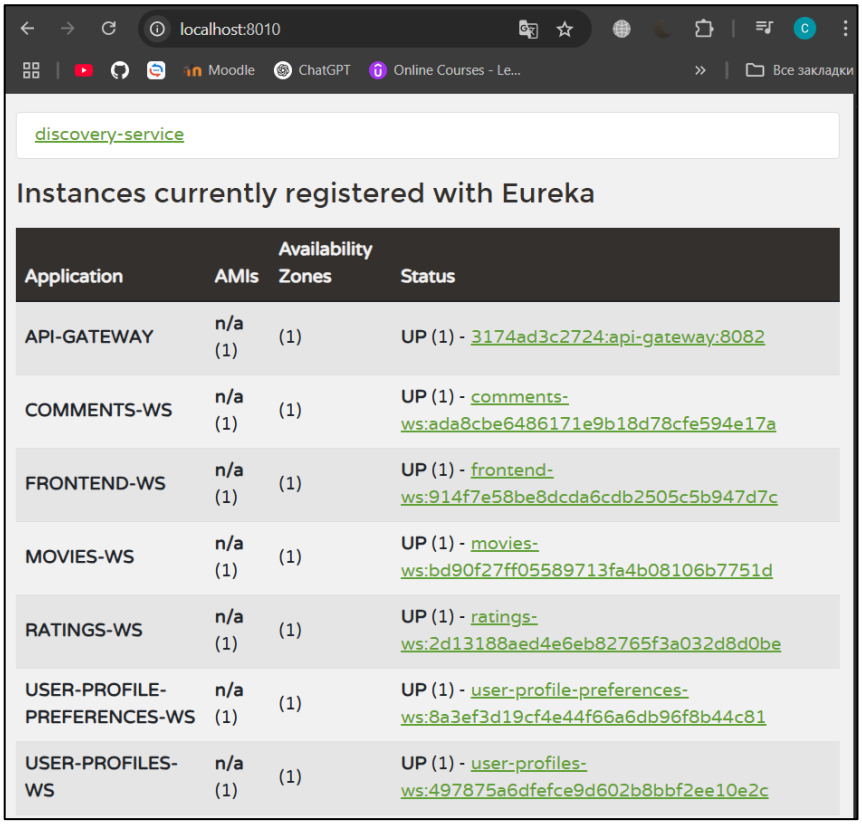


Рисунок 2.2.2 – Скріншот дашборду Eureka Discovery Service

Config Server - це центр конфігурацій. Я підключила до нього Git-репозиторій, у якому зберігаються всі .properties або .yml файли конфігурації. Якщо потрібно змінити порт, логін до бази даних, або змінну середовища - достатньо змінити файл у Git, і всі мікросервіси самі підтягнуть ці зміни. Завдяки цьому не треба перезапускати сервіси вручну або у гортати їх заново, і це дуже зручно при розробці.

Мікросервіси побудовані відповідно до принципу «один сервіс - одна відповідальність»:

– Movies Microservice зберігає всю інформацію про фільми - їхні назви, опис, жанри, рік випуску тощо;

– MovieRatings Microservice відповідає за збереження і підрахунок рейтингу, який залишають користувачі;

– Comments Microservice реалізує додавання, редагування та перегляд коментарів;

– UserProfilePreferences Microservice зберігає вподобання користувача, які потім можна враховувати при формуванні рекомендацій;

– UserProfiles Microservice обробляє все, що пов’язано з акаунтами користувачів: логін, реєстрація, редагування профілю тощо;

– FrontendApi Microservice - це міст між фронтендом (тобто інтерфейсом користувача) і всіма бекенд-сервісами, особливо він використовується для обробки запитів на завантаження або вивід зображень (наприклад, фото фільмів або аватарки користувачів);

Кожен мікросервіс має свою власну базу даних на Postgres. Це було зроблено свідомо, щоб розділити відповідальність і уникнути ситуацій, коли один збій в одній БД впливає на всю систему. Також це дозволяє кожному сервісу масштабуватися окремо й уникати конфліктів транзакцій.

Зберігання зображень винесено в окремий компонент - S3. Це хмарне сховище від Amazon, у якому зберігаються всі великі файли, які не варто тримати в базі (наприклад, постери фільмів, фото профілю, банери жанрів тощо). Фронтенд через FrontendApi Microservice має доступ до S3 і може завантажувати або отримувати ці файли напряму через відповідні URL. Це дозволяє значно зменшити навантаження на базу даних і підвищити швидкість роботи системи.

Уся ця архітектура дозволяє окремо оновлювати або перезапускати будь-який мікросервіс, не зачіпаючи інші. При цьому всі компоненти мають спільну конфігурацію та можуть знаходити одне одного без ручного налаштування.

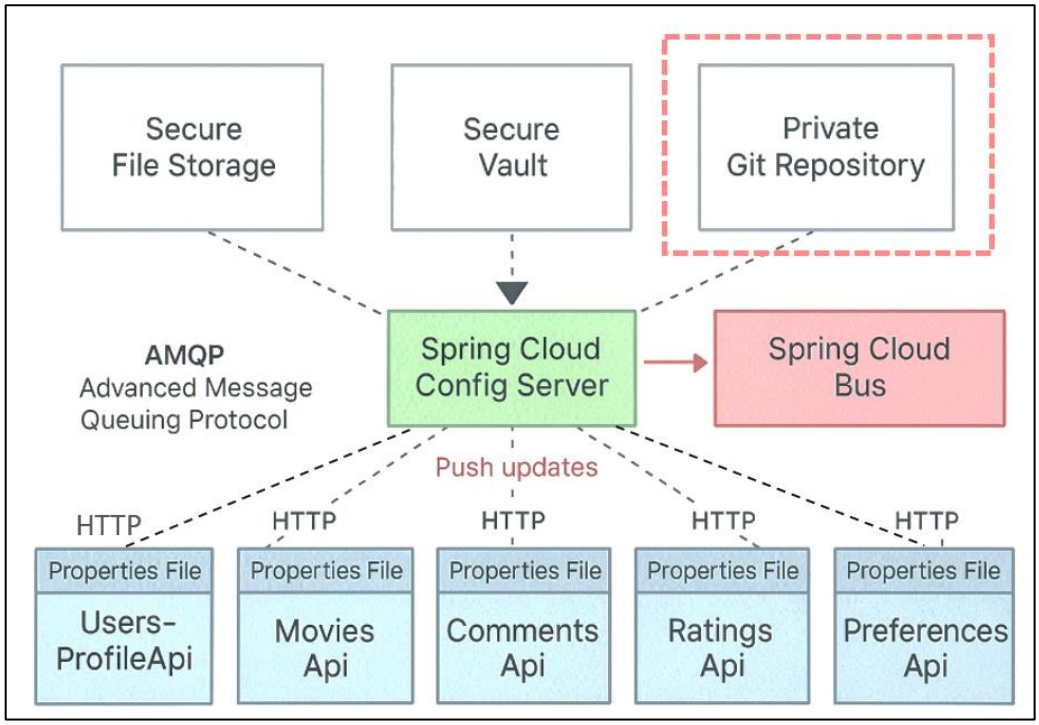


Рисунок 2.2.3 – Скріншот механізму конфігурації через Spring Cloud Config Server та Git-репозиторій

Усі мікросервіси використовують спільну систему для зберігання й оновлення конфігурацій, і головну роль у цьому відіграє Spring Cloud Config Server. Його основне завдання - це централізоване керування параметрами, які потрібні мікросервісам для нормальної роботи. Щоб не зберігати одні й ті самі конфігурації в кожному сервісі окремо, я винесла всі .properties та .yml файли в окремий приватний Git-репозиторій. У ньому зберігаються всі налаштування: порти, адреси БД обмеження для логів, доступ до сторонніх сервісів тощо. Цей репозиторій пов’язаний із Spring Cloud Config Server, і він постійно стежить за тим, чи не з’явились якісь зміни.

Як тільки я змінюю файл у Git, Config Server одразу бачить це і завдяки Spring Cloud Bus може повідомити всім мікросервісам, що з’явилися оновлення. Spring Cloud Bus - це компонент, який забезпечує механізм сповіщення. У моєму випадку він працює поверх RabbitMQ (протокол AMQP), і саме через нього

Config Server «пушить» повідомлення про зміну конфігів. Це означає, що сервіси не просто періодично перевіряють, чи змінився файл - їм приходить конкретне повідомлення: «оновись, бо в тебе нові параметри».

У схемі (див. рисунок 2.2.3) видно, як саме це працює. Config Server через HTTP надає доступ до актуальних файлів конфігурації кожному з мікросервісів. Наприклад, коли запускається UsersProfileApi, він звертається до Config Server і запитує файл зі своїми налаштуваннями. Те саме стосується MoviesApi, CommentsApi, RatingsApi та PreferencesApi - усі вони отримують власні properties-файли через HTTP. Це дозволяє підтримувати налаштування в одному місці, не дублювати їх у коді і спрощує обслуговування. Якщо, наприклад, у якийсь момент я вирішую, що база даних має працювати на іншому порту або логування має бути детальнішим - я змінюю параметри в Git, і всі сервіси автоматично це підхоплюють, без перезапуску.

Отож, ця структура дозволяє мені керувати конфігураціями у зручний, безпечний та контрольований спосіб. Замість того, щоб редагувати файли окремо в кожному сервісі або перезапускати систему після кожної зміни, усе відбувається централізовано, прозоро та без збоїв.

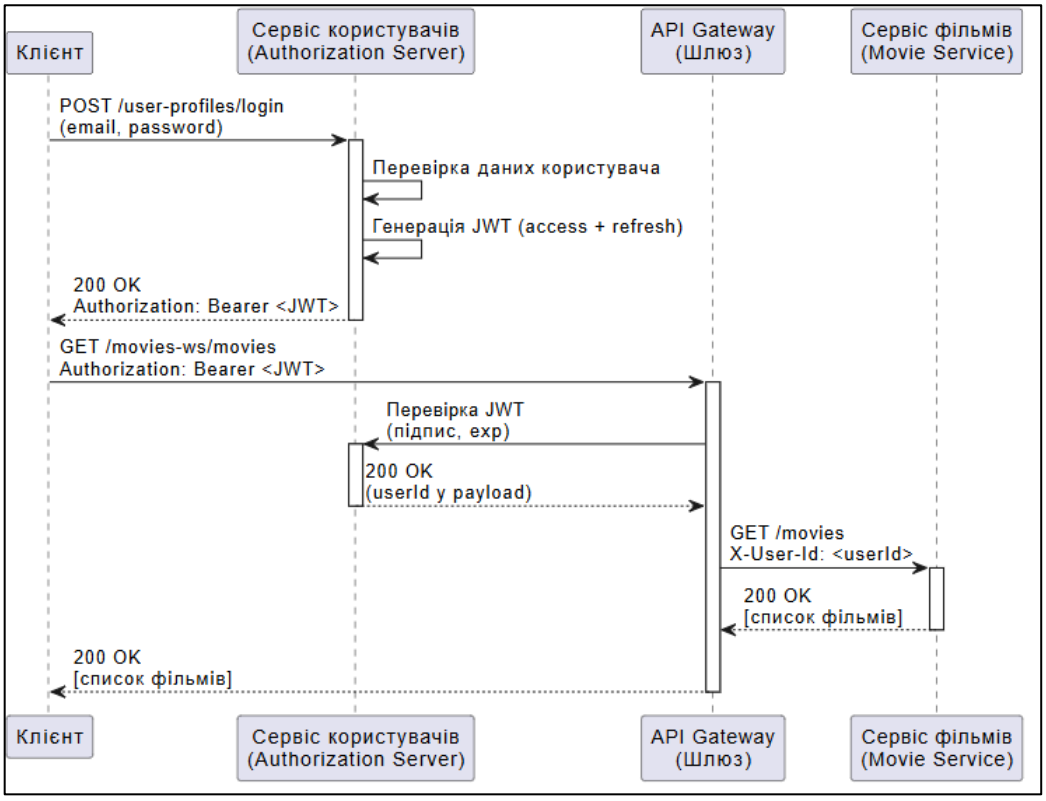


Рисунок 2.2.4 – Скріншот діаграми послідовності авторизації користувача та перевірки JWT у мікросервісах

У MovieMingle процес доступу користувача до певного ресурсу (наприклад, списку фільмів) побудований на основі JWT-аутентифікації, і вся ця логіка реалізована з використанням API Gateway.

Коли користувач хоче увійти в систему, він спочатку надсилає POST-запит на адресу /user-profiles/login, передаючи свої облікові дані - електронну пошту та пароль. Цей запит обробляє окремий мікросервіс, який я умовно називаю сервісом користувачів або авторизаційним сервером.

Він перевіряє правильність введених даних, звіряє їх із базою й, якщо все добре, генерує два токени: access-token і refresh-token. У відповіді користувач отримує access-token у форматі JWT, який буде використовуватись у подальших запитах як доказ того, що він успішно авторизований.

Після цього, коли користувач хоче отримати список фільмів, він робить GET-запит на адресу /movies-ws/movies, і обов’язково додає в заголовку Authorization значення Bearer <JWT> , тобто той самий токен, який йому повернули після входу. Далі цей запит потрапляє не напряму до сервісу фільмів, а спершу до API Gateway - це такий проміжний шар між зовнішнім світом і мікросервісами. Саме тут і відбувається основна перевірка безпеки: шлюз дивиться, чи є в запиті JWT-токен, і якщо він є - перевіряє його справжність. Тобто gateway розпаковує токен, перевіряє його цифровий підпис і термін дії (чи не закінчився він). Якщо токен невалідний або відсутній - користувач отримає помилку й доступу до сервісу не буде. Але якщо все в порядку шлюз пропускає запит далі.

2.3 Технології, використані в реалізації

У реалізації мого мікросервісного застосунку я використала сучасний стек технологій, який дозволив побудувати масштабовану та зручну систему. В основі всієї архітектури лежить Spring Boot - це фреймворк на Java, який значно спрощує створення мікросервісів, бо дає готові рішення для REST API, підключення баз даних, безпеки, обробки помилок та інших важливих аспектів.

Кожен мікросервіс був реалізований окремо, має свою логіку та базу даних, що дозволяє змінювати або оновлювати їх незалежно один від одного.

Для обміну конфігурацією між мікросервісами я використала Spring Cloud Config Server. Це окремий компонент, який завантажує загальні параметри (наприклад, шляхи до баз, порти, змінні середовища) з централізованого Git-репозиторію. Тобто я зберігаю конфігурацію в окремому файлі у Git, і всі сервіси під час запуску звертаються до Config Server, щоб отримати свої налаштування. Це зручно, бо я можу змінити параметри в одному місці - і всі сервіси автоматично оновляться, не потрібно редагувати їх вручну.

Щоб мікросервіси могли знаходити один одного динамічно, без «зашитих» IP-адрес, я використала Eureka Discovery Service. Коли сервіс запускається, він реєструється в Eureka, і тоді інші сервіси можуть його знайти просто за назвою. Наприклад, сервіс фільмів може звертатися до «comments-service», не знаючи точну адресу. Це особливо важливо, коли система розгортається у хмарі або на декількох серверах.

Всі запити від клієнтів потрапляють у систему через Spring Cloud Gateway, який працює як шлюз. Саме він відповідає за маршрутизацію запитів до відповідних мікросервісів, а також за безпеку. Через нього йде вся перевірка JWT-токена: якщо токен невалідний або прострочений - користувач отримує помилку, і запит далі не проходить. Це дозволяє ізолювати логіку безпеки в одному місці й не дублювати її в кожному мікросервісі.

Для зберігання даних я обрала PostgreSQL - це надійна реляційна база даних, яка дуже добре підходить для зберігання структурованої інформації. У моєму застосунку кожен мікросервіс має свою окрему базу, тобто вони не ділять спільний доступ до однієї схеми. Такий підхід дозволяє уникнути конфліктів, спрощує розгортання і покращує масштабованість.

Щодо зберігання зображень - я не зберігаю їх у базі, а використовую хмарне сховище Amazon S3. Це дозволяє зберігати фото профілю користувачів, постери до фільмів, іконки жанрів тощо в окремому безпечному просторі. Доступ до цих файлів здійснюється через URL, які генеруються у Frontend APl.

Щоб забезпечити логування та моніторинг, я інтегрувала власну кастомну бібліотеку логування, яка формує уніфіковані логи та передає їх через Kafka у систему збору логів. Там вони обробляються за допомогою Logstash,

зберігаються в Elasticsearch і візуалізуються через Kibana. Це дозволяє в режимі реального часу бачити, що відбувається у системі, аналізувати помилки, навантаження та навіть виявляти аномалії в поведінці мікросервісів.

На фронтенді, тобто з боку інтерфейсу, я використала JavaScript, HTML, CSS для побудови клієнтської частини, яка взаємодіє з API-шлюзом. Усі запити від користувача йдуть через нього, і на основі токенів відбувається доступ до потрібного ресурсу. Для обробки зображень у браузері використовуються прямі посилання на S3.

Уся ця комбінація технологій дала змогу створити систему, яка є не тільки зручною для користувача, а й простою в адмініструванні, масштабованою та здатною витримувати великі навантаження.

3 РОЗРОБКА ТА ІНТЕГРАЦІЯ КАСТОМНОЇ БІБЛІОТЕКИ ЛОГУВАННЯ

3.1 Формат логів і шаблон уніфікованого логування

Рисунок 3.1.1 – Скріншот прикладу сформованих JSON-логів у консольному виводі сервісу коментарів

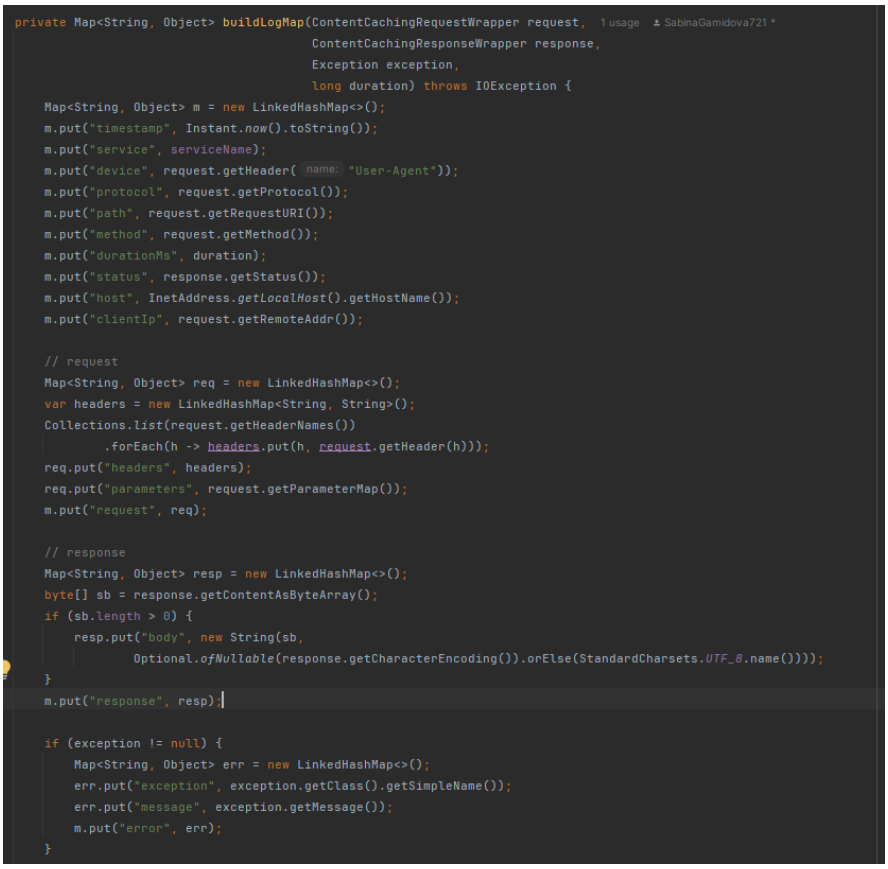
У моїй системі логування (див. лістинги А.1 та А.2) реалізовано чітко структурований підхід до створення та запису логів на основі власного шаблону. Основна ідея полягала в тому, щоб у кожному лог-записі містилася максимально корисна інформація про HTTP-запит, його обробку, відповідь і можливі виключення, і все це в єдиному, уніфікованому форматі, придатному для подальшого аналізу - наприклад, з використанням Elasticsearch та Kibana.

Фільтр doFilterInternal (див. рисунок 3.1.2 чи лістинг А.1), що перехоплює запити, обгортає як сам запит, так і відповідь у відповідні обгортки - ContentCachingRequestWrapper та ContentCachingResponseWrapper. Це дозволяє зчитувати повне тіло як запиту, так і відповіді, навіть після того, як вони були передані далі по ланцюжку обробки. Після цього фіксується момент початку виконання, а в блоці finally - момент завершення, що дозволяє обчислити тривалість у мілісекундах. Якщо виникає виняток, він перехоплюється і також включається до логу. Уся інформація збирається в одному методі buildLogap (див. рисунок 3.1.3 чи лістинг А.1). Він формує LinkedHashMap, де чітко

розділено дані про запит (request), відповідь (response), а також супровідну інформацію - час, назву сервісу, заголовки, шлях, метод, статус, IP-клієнта, хост, тривалість, протокол, тощо. Лог також включає рівень важливості (INFO/WARN/ERROR), який визначається на основі статус-коду відповіді або наявності винятку: якщо код >= 500 або є виняток - це ERROR, якщо 400-499 - WARN, усе інше - INFO.

```java
@Override  no usages  SabinaGamidova721
protected void doFilterInternal(HttpServletRequest request0,
                                HttpServletResponse response0,
                                FilterChain chain)
        throws ServletException, IOException {

    var request = new ContentCachingRequestWrapper(request0);
    var response = new ContentCachingResponseWrapper(response0);
    long start = System.currentTimeMillis();
    Exception exception = null;

    try {
        chain.doFilter(request, response);
    } catch (Exception ex) {
        exception = ex;
        throw ex;
    } finally {
        long duration = System.currentTimeMillis() - start;
        var m = buildLogMap(request, response, exception, duration);

        String level;
        int status = response.getStatus();
        if (exception != null || status >= 500)      level = "ERROR";
        else if (status >= 400)                      level = "WARN";
        else                                         level = "INFO";

        m.put("level", level);
        String json = mapper.writeValueAsString(m);

        switch (level) {
            case "ERROR" -> logger.error(json);
            case "WARN"  -> logger.warn(json);
            default      -> logger.info(json);
        }

        response.copyBodyToResponse();
    }
```

Рисунок 3.1.2 – Скріншот логуючого фільтра з обробкою виключень та визначенням рівня логування (фільтр із кастомної бібліотеки логування)



```
private Map<String, Object> buildLogMap(ContentCachingRequestWrapper request,  1 usage  SabinaGamidova721 *
                                        ContentCachingResponseWrapper response,
                                        Exception exception,
                                        long duration) throws IOException {
    Map<String, Object> m = new LinkedHashMap<>();
    m.put("timestamp", Instant.now().toString());
    m.put("service", serviceName);
    m.put("device", request.getHeader( name: "User-Agent"));
    m.put("protocol", request.getProtocol());
    m.put("path", request.getRequestURI());
    m.put("method", request.getMethod());
    m.put("durationMs", duration);
    m.put("status", response.getStatus());
    m.put("host", InetAddress.getLocalHost().getHostName());
    m.put("clientIp", request.getRemoteAddr());

    // request
    Map<String, Object> req = new LinkedHashMap<>();
    var headers = new LinkedHashMap<String, String>();
    Collections.list(request.getHeaderNames())
            .forEach(h -> headers.put(h, request.getHeader(h)));
    req.put("headers", headers);
    req.put("parameters", request.getParameterMap());
    m.put("request", req);

    // response
    Map<String, Object> resp = new LinkedHashMap<>();
    byte[] sb = response.getContentAsByteArray();
    if (sb.length > 0) {
        resp.put("body", new String(sb,
                Optional.ofNullable(response.getCharacterEncoding()).orElse(StandardCharsets.UTF_8.name())));
    }
    m.put("response", resp);

    if (exception != null) {
        Map<String, Object> err = new LinkedHashMap<>();
        err.put("exception", exception.getClass().getSimpleName());
        err.put("message", exception.getMessage());
        m.put("error", err);
    }
```

Рисунок 3.1.3 – Скріншот методу buildLogMap, який формує структуру лог-запису

Особливу увагу приділено серіалізації - лог формується у форматі JSON, що значно спрощує подальшу обробку у системах аналітики. Серіалізація виконується через Jackson (mapper.writeValueAsString(m)), що забезпечує правильний формат незалежно від складності об’єкта. Лог потім виводиться через відповідний рівень логування .info(), .warn() або .error() - у залежності від раніше визначеного рівня.

Таке рішення дозволяє точно бачити, який саме запит надійшов, як він був оброблений, скільки часу це зайняло, що повернулося у відповіді і чи була помилка. У логах присутня й інформація про user-agent, що допомагає відстежити, з якого клієнта чи пристрою надходить трафік. Загалом така схема дозволяє не лише проводити моніторинг, а й будувати складні аналітичні запити

в Kibana, наприклад: «знайти всі запити з тривалістю > 500 мс і статусом 200, що надходили з певного user-agent» або «відобразити всі помилки, що виникали при POST-запитах».

3.2 Архітектура передачі логів: Kafka-продюсер та пайплайн

Було реалізовано повноцінну архітектуру збору, транспортування, зберігання та подальшого аналізу логів. Ця архітектура охоплює кастомну бібліотеку логування, лог-шипери (Filebeat), систему передачі повідомлень Kafka, компонент Logstash для попередньої обробки та фільтрації, і нарешті Elasticsearch та Kibana - як основні інструменти збереження та візуалізації. Також логі передаються до Python-моделі машинного навчання для виявлення аномалій у поведінці мікросервісів.

Рисунок 3.2.1 - Скріншот архітектури лог-пайплайну на базі Kafka, Beats, Logstash, Elasticsearch та Kibana, загальна схема

На Рисунку 3.2.1 у центрі знаходиться брокер Kafka, який виступає в ролі «центру обміну» повідомленнями. Кожен мікросервіс генерує структуровані логи у форматі JSON за допомогою спеціально розробленої бібліотеки (JsonLogging). Ці логи зберігаються у відповідних локальних файлових логах, які потім зчитуються лог-шиперами - Filebeat-агентами. Кожен Filebeat налаштований як

Kafka-продюсер, що надсилає логи безпосередньо у відповідний Kafka-топік - наприклад, comments-logs, gateway-logs, movies-logs тощо. Це дає можливість відокремити потоки логів для кожного мікросервісу та спростити обробку.

Рисунок 3.2.2 - Скріншот обробленого лог-запису у Logstas (мікросервіс коментарів)

На Рисунку 3.2.2 видно, як один із логів, що надійшов із Kafka, був трансформований у структуровану подію. Logstash відповідає за парсинг, перетворення та збагачення цих логів, після чого вони надсилаються - Elasticsearch. У прикладі зображено JSON-об’єкт із полями: рівень логування, сервіс, метод, статус, клієнтська IP-адреса, тривалість запиту, тіло відповіді тощо. Це свідчить про успішне збереження і розбір логів.

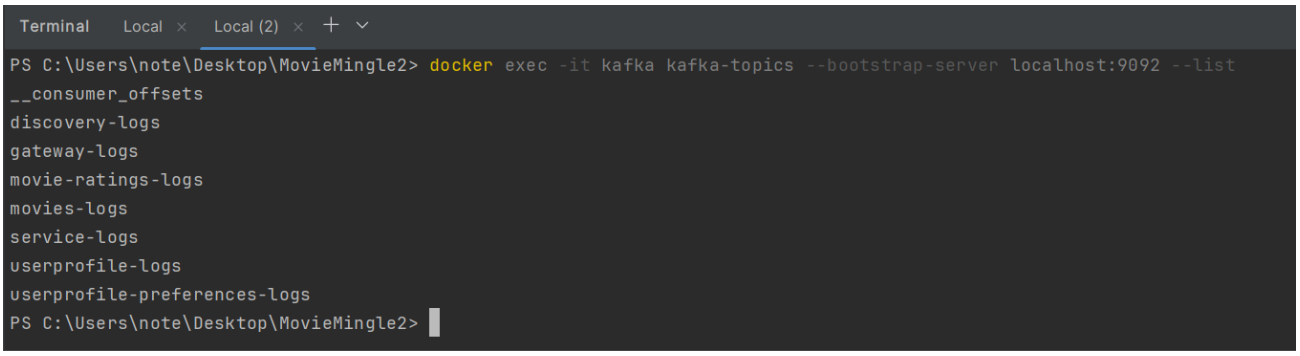


Рисунок 3.2.3 - Скріншот перевірки наявності відповідних топіків у Kafka

Для перевірки наявних топіків Kafka використано консольну команду, показану на скріншоті вище. Тут можна побачити всі створені топіки, серед яких comments-logs, movies-logs, gateway-logs, userprofile-logs та інші. Кожен із них отримує дані з відповідного сервісу.

Рисунок 3.2.4 - Скріншот читання логів із Kafka-топіка через консоль

На Рисунку 3.2.4 продемонстровано, як можна у реальному часі прочитати повідомлення з Kafka за допомогою kafka-console-consumer. У прикладі прочитано лог-повідомлення з топіка comments-logs. Кожен рядок - це окремий лог-запис, що містить інформацію про сервіс, назву операції, вхідні параметри, результат виконання та часові мітки.

Тож можна в реальному часі слідкувати за активністю системи або використовувати ці повідомлення як вхідні дані для аналізу аномалій.

Уся ця архітектура дозволяє не просто зберігати та переглядати логи, а й аналізувати їх у реальному часі, виявляти підозрілу активність, стежити за змінами навантаження або виявляти помилки.

4 АНАЛІЗ ЛОГІВ ЗА ДОПОМОГОЮ МАШИННОГО НАВЧАННЯ ТА ОЦІНКА РЕЗУЛЬТАТІВ

4.1 Що таке виявлення аномалій?

Виявлення аномалій (іноді також називається виявленням викидів) - це процес пошуку шаблонів або окремих прикладів у наборі даних, які суттєво відрізняються від очікуваної або «нормальної» поведінки.

Варто зазначити, що поняття «нормальних» і «аномальних» даних може змінюватися залежно від контексту. Ось кілька прикладів реального застосування виявлення аномалій:

Фінансові транзакції

– норма: регулярні покупки в Лондоні та стабільна витрата коштів;

– аномалія: раптове велике зняття грошей в Ірландії з того ж рахунку можлива ознака шахрайства;

Мережевий трафік у кібербезпеці

– норма: стабільна передача даних, звичайні протоколи;

– аномалія: різкий стрибок у кількості переданих даних або використання невідомого протоколу - потенційна атака або зараження системи;

Моніторинг життєвих показників пацієнта

– норма: стабільний пульс і тиск;

– аномалія: раптове зростання пульсу і падіння тиску — можливий сигнал про надзвичайну ситуацію або збій обладнання;

Виявлення аномалії найчастіше здійснюється **за допомогою моделей навчання без учителя (unsupervised learning)**. Аналітики обирають відповідний підхід залежно від типу аномалії, структури, властивостей даних і контексту.

Крім фінансів і боротьби з шахрайством, внявлення аномалій застосовується у:

- кібербезпеці;
- охороні здоров'я;
- моніторингу промислового обладнання;
- виявленні мережевих вторгнень;
- енергетичних системах;
- електронній комерції та аналізі поведінки користувачів;
- контролі якості у виробництві;

І найголовніше - ці механізми вже інтегровані в багато повсякденних сервісів, навіть якщо їх не помічаємо.

4.2 Роль виявлення аномалій у Data Science

Дані є найціннішим ресурсом у сфері Data Science, і будь-які аномалії в цих даних можуть серйозно підірвати всю аналітичну роботу. Якщо дані містять несподівані або невідповідні значення, це призводить до помилок у статистичних висновках, недостовірних візуалізаціях та неправильних прогнозах моделей машинного навчання. На виході такі помилки можуть стати підставою для хибних рішень і призвести до суттєвих втрат. Аномалії здатні спотворювати ключові показники статистичного опису розподілів (зокрема середнє значення та стандартне відхилення), на яких базуються більшість алгоритмів машинного

навчання. Саме тому виявлення аномалій є необхідним етапом обробки даних. Якщо не визначити їх вчасно, то навіть ідеальні моделі працюватимуть із хибними вхідними даними і даватимуть недостовірні результати.

4.3 Класифікація аномалій

4.3.1 Типи аномалій

Існує дві основні категорії аномалій: **викиди** (outliers) та **новинки** (novelties).

Викиди — це точки або випадки, які вже присутні у тренувальних даних, але відхиляються від звичних значень настільки, що виходять за межі очікуваного розподілу. Наприклад, якщо щоденні температури в місті звичайно коливаються між 20°C і 30°C, то раптовий стрибок до 40°C у рамках тих самих вимірювань буде вважатися викидом.

Новинки — це нові випадки, які виникають після тренування моделі і раніше не зустрічалися в даних. Наприклад, уявімо, що місто встановило нову, більш точну метеостанцію, яка тепер вимірює температуру в діапазоні 25-35°C. Ця нова тенденція не є поодиноким стрибкоподібним відхиленням, а показує зміну звичної закономірності. То це не викид, а саме новинка.

Саме слово «аномалія» охоплює обидва поняття — будь-яку ненормальну поведінку, яка порушує очікуваний шаблон. Важливо вміти відрізняти викиди від новинок, адже виявлення викидів зазвичай здійснюються за допомогою статистичних методів або простих непараметричних правил (наприклад, z-оцінок), а для новинок потрібен окремий механізм контролю над вихідними даними. Якщо упустити цей етап, слабкі або хибні дані потраплять у дашборди, звіти і моделі, що призведе до невірних висновків і рішень.

4.3.2 Типи викидів: уніваріантні та мультиваріантні

У процесі аналізу даних дуже важливо розуміти, що не всі незвичні значення однакові за своєю природою. Коли говоримо про «аномалії» або «викиди», маємо на увазі ті спостереження, які помітно відрізняються від основної маси. Але є різні типи таких відхилень, і для того, щоб правильно з ними працювати, варто розрізняти, що саме бачимо — уніваріантний чи мультиваріантний викид.

Уніваріантні викиди - це ті, що можна побачити, аналізуючи лише одну змінну.

Наприклад, уявімо, що маємо інформацію про ціні на будинки в одному районі. Більшість із них коштують десь між 200 і 400 тисяч доларів. Але раптом серед цього списку бачимо будинок за мільйон. Навіть не знаючі нічого більше про його розмір чи стан, вже розуміємо: щось тут незвично. І це якраз приклад уніваріантного викиду - дивимося лише на одну характеристику (у цьому випадку - ціну) і бачимо, що вона сильно виривається з загального ряду.

Мультиваріантні викиди - трохи складніші. Вони не завжди виглядають підозрілими, якщо дивитися лише на одну ознаку. Їхня аномальність виявляється лише тоді, коли **беремо до уваги кілька змінних одночасно.**

Наприклад, уявімо інший будинок, який коштує $380,000. На перший погляд, усе нормально, адже ціна вписується у загальні межі. Але якщо ми подивимося ближче і побачимо, що цей будинок має лише одну спальню та вдвічі меншу площу порівняно з іншими будинками в тому ж районі, то картина вже виглядає інакше. Такий будинок (його ціна) вже не здається логічним при такому співвідношенні площі, кількості кімнат і ціни. І ось тут вже маємо справу з мультиваріантним викидом — його відхилення проявляється тільки тоді, колі оцінюємо комбінацію кількох характеристик.

4.3.2.1 Методи для унівaріантних викидів

Z-оцінка (Z-score)

Z-оцінка показує, наскільки далеко значення відхиляється від середнього арифметичного значення, вимірюючи це в одиницях стандартного відхилення.

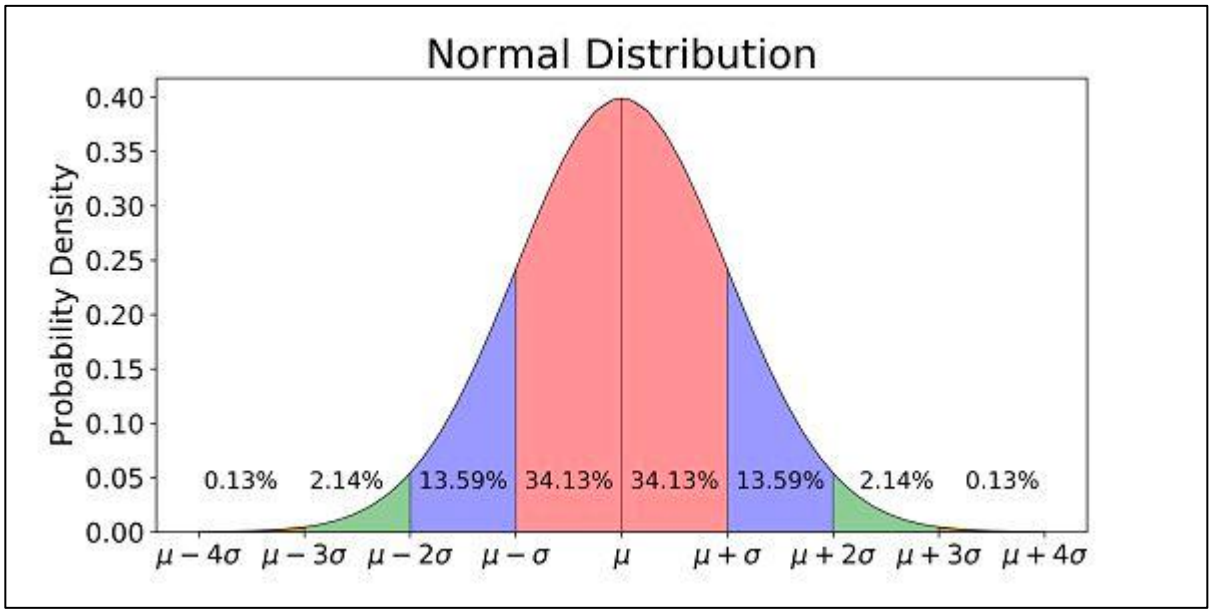


Рисунок 4.3.2.1.1 – Скріншот прикладу Z-оцінки (запозичено з [22])

Формула:

$$Z = \frac{x-\mu}{\sigma}, \qquad (4.3.2.1.1)$$

де x - конкретне значення, яке аналізується,

p - середнє значення всіх даних,

o - стандартне відхилення.

Наприклад, якщо Z > 3, це означає, що значення більше ніж на 3 стандартні відхилення від середнього - імовірно, це аномалія.

Треба використовувати, коли дані мають нормальний розподіл (плавну «дзвінкоподібну» форму).

Міжквартильний розмах (Interquartile Range, IQR)

IOR допомагає виявити відхилення, орієнтуючись не на середнє, як попередня Z-оцінка, а на медіану і розподіл квартилей (тобто розбиття вибірки на чверті).

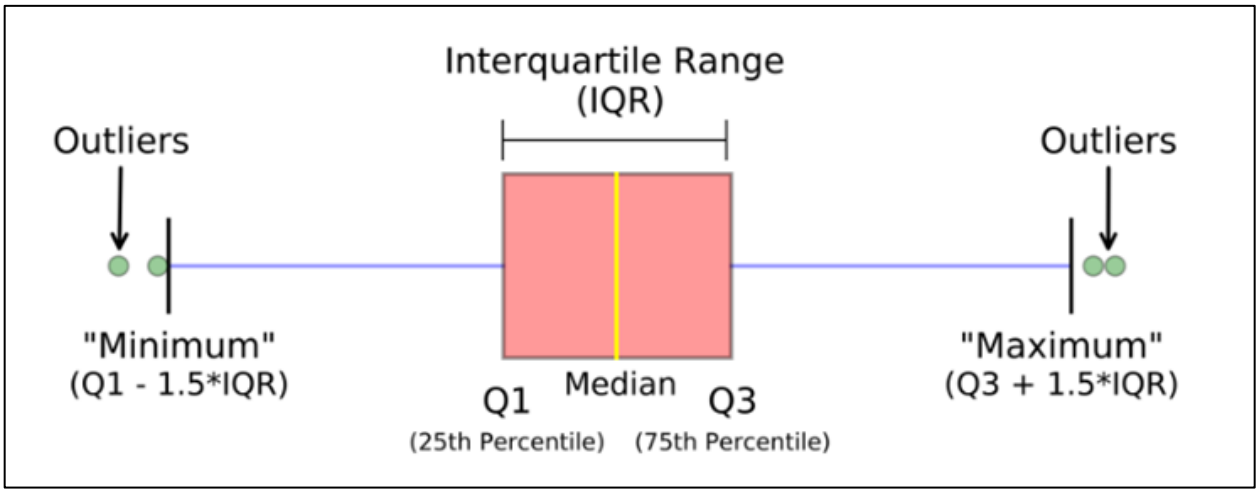


Рисунок 4.3.2.1.2 – Скріншот прикладу міжквартильного розмаху (запозичено із [23])

Формули:

$$IQR = Q_3 - Q_1, \quad (4.3.2.1.2)$$

$$\text{Аномальні межі} = [Q_1 - 1.5 * IQR,\ Q_3 + 1.5 * IQR) \quad (4.3.2.1.3)$$

де Q1 — перший квартиль (25% даних менші або рівні йому),

Q3 — третій квартиль (75% даних менші або рівні йому).

Усе, що менше за нижню межу або більше за верхню, вважається аномалією.

Використовуємо, коли розподіл даних не обов’язково нормальний, є викиди або «перекоси» (наприклад до великих значень).

Модифікована Z-оцінка (Modified Z-score)

Це більш стійка версія звичайної Z-оцінки, яка використовує медіану замість середнього та MAD замість стандартного відхилення - тобто менше піддається впливу самих викидів.

Формула:

$$M_1 = \frac{0{,}6745 \cdot (x_i - med)}{MAD}, \qquad (4.3.2.1.4)$$

де x_i — конкретне значення,

med — медіана всіх значень,

MAD — середнє абсолютне відхилення, тобто медіана від $|x_i - med|$,

коефіцієнт 0.6745 — щоб значення були сумісні зі звичайною Z-оцінкою при нормальному розподілі.

Використовуємо, коли у вибірці багато викидів або вона не є нормально розподіленою

4.3.2.2. Методи для мультиваріантних викидів

Виявлення аномалій за допомогою Isolation Forest

Isolation Forest (IF) — це метод, який базується на принципі, що аномальні точки ізолюються швидше, ніж звичайні. На відміну від багатьох інших алгоритмів, які моделюють «нормальну» поведінку і виявляють відхилення, IF безпосередньо моделює аномальність, спрощуючи завдання.

Дані ітеративно розділяються за випадковими ознаками і випадковими пороговими значеннями. Утворюється бінарне дерево, де кожен розділ (node) ділить дані надвоє. Точка, яка потрапляє в окрему гілку за менше кроків, вважається ізольованою - і, найймовірніше за все, аномальною.

Чим менше глибина дерева для даної точки, тим вища її «аномальність». Глибина усереднюється за кількома ізоляційними деревами (наприклад, 100 чи 200).

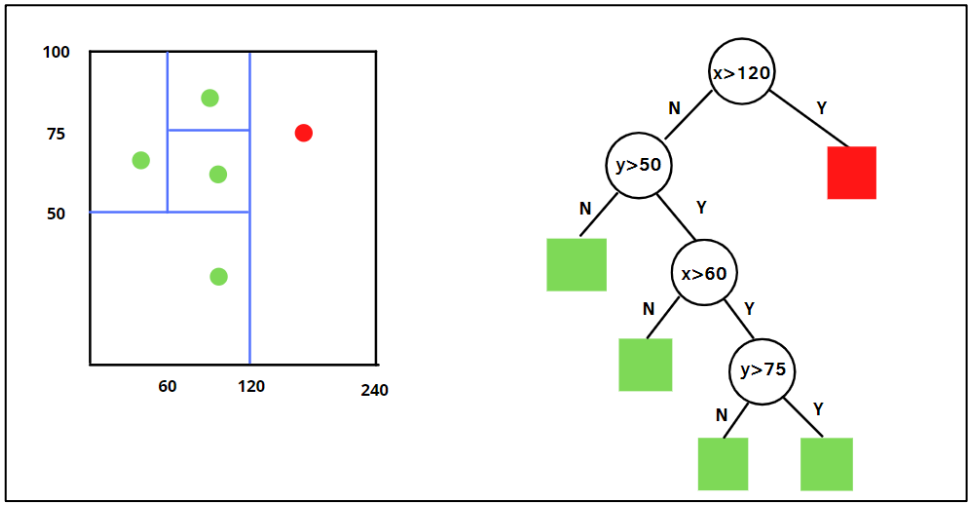


Рисунок 4.3.2.2.1 – Скріншот прикладу побудови бінарного дерева у моделі IF (запозичено із [24])

Формула аномальності:

$$s(x,n) = 2^{-\frac{E(h(x))}{c(n)}}, \qquad (4.3.2.2.1)$$

де $s(x,n)$ - оцінка аномальності точки x при n зразках,

$E(h(x))$ - середня довжина шляху до ізоляції точки x у всіх деревах,

$c(n)$ - нормалізуючий коефіцієнт:

$$c(n) = 2H(n-1) - \frac{2(n-1)}{n}; H(n) \approx ln(n) + \gamma, \qquad (4.3.2.2.2)$$

де γ – стала Ейлера.

Із гіперпараметрів дана модель має:

– n_estimators — кількість дерев у лісі (типове значення — 100), більше дерев => точніше, але довше;

– max_samples - скільки випадкових точок вибирати для побудови кожного дерева;

– contamination - очікувана частка аномалій у даних (використовується для визначення порогу відсічення);

– max_features — скільки ознак використовувати при кожному поділі;

Із позитивних сторін, то дана модель швидка, не потребує масштабування даних та добре працює на великих вибірках.

А ось із мінусів, то може втрачати ефективність на дуже малих наборах.

Виявлення аномалій за допомогою Local Outlier Factor

Local Outlier Factor (LOF) - це метод, який оцінює аномальність точки, порівнюючи щільність її оточення зі щільністю сусідніх точок. Якщо об’єкт знаходиться в області з помітно нижчою щільністю, ніж у його сусідів — він вважається викидом. Тобто основна ідея => вимірювання локальної досяжної щільності (Local Reachability Density, LRD).

Потім обчислюється коефіцієнт LOF, який показує, наскільки точка менш «щільна», ніж її сусіди.

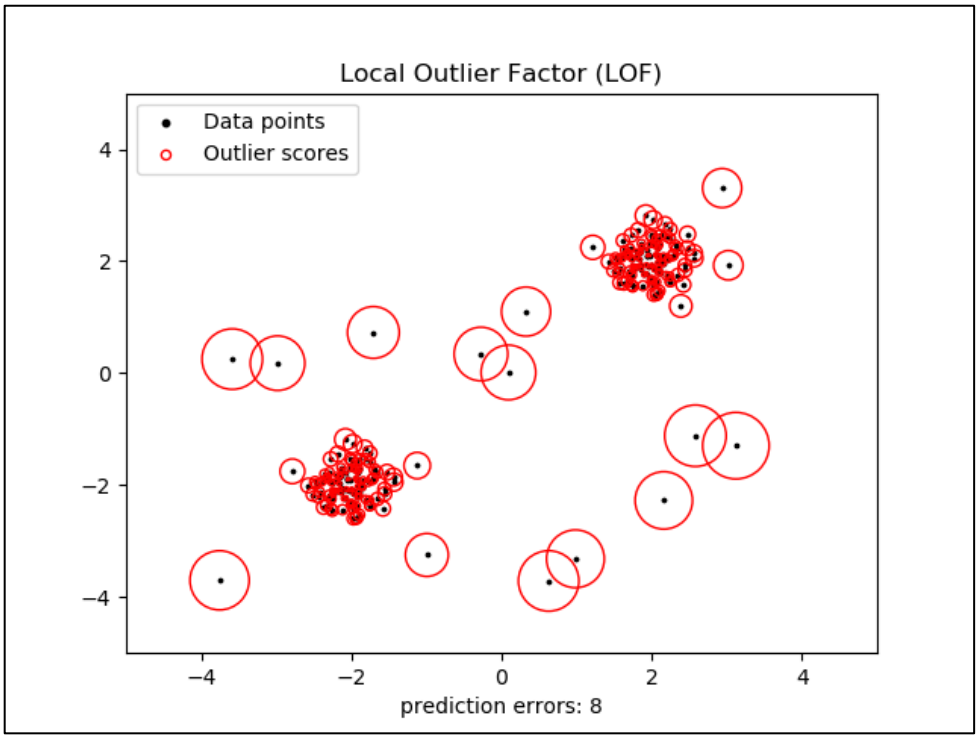


Рисунок 4.3.2.2.2 – Скріншот результату роботи моделі LOF (запозичено із [25])

Формула відстані до k-го сусіда:

kdist(p) = відстань до k-го найближчого сусіда точки p (4.3.2.2.3)

Формула обчислення відстані від точки p до всіх інших у вибірці (Евклідова відстань):

$$d(p,o) = \sqrt{\sum_{i=1}^{n}(p_i - o_i)^2}, \quad (4.3.2.2.4)$$

де p_i та o_i – значення відповідних ознак для точок p та o,

n – кількість ознак.

Формула досяжної відстані:

$$rd_k(p,o) = max\big(kdist(o), d(p,o)\big), \quad (4.3.2.2.5)$$

де $d(p,o)$ – евклідова відстань між p та o.

Формула локальної досяжної щільності (LRD):

$$LRD_k(p) = \left(\frac{1}{\sum_{o\in N_k(p)} rd_k(p,o)} \cdot \frac{1}{|N_k(p)|}\right)^{-1}, \quad (4.3.2.2.6)$$

де $N_k(p)$ – множина k найближчих сусідів точки p.

Формула LOF-оцінки:

$$LOF_k(p) = \left(\sum_{o\in N_k(p)} \frac{LRD_k(o)}{LRD_k(p)}\right) \cdot \frac{1}{|N_k(p)|} \quad (4.3.2.2.7)$$

Якщо LOF(p) > 1, то точка має нижчу щільність, ніж її оточення, і вважається підозрілою.

Якщо до 1 – норма. Якщо < 1 – дуже щільна зона.

Із гіперпараметрів маємо такі:

– n_neighbors (тобто k) – скільки сусідів брати до уваги (типово беруться значення 10-30);

– metrics – тип відстані, тут найчастіше використовується саме евклідова;

– contamination – частка очікуваних викиді, що впливає на поріг для класифікації;

Із плюсів цієї моделі можна виділити те, що вона добре працює з різною щільністю кластерів. Із мінусів – чутливість до вибору k, а також складність масштабування для великих даних.

Виявлення аномалій за допомогою кластерізаційній методу K-means

Метод K-means традиційно використовується для звичайної кластеризації, але його також можна застосовувати для виявлення аномалій. Основна ідея => точки, що знаходяться далеко від центрів кластерів, можуть бути аномальними.

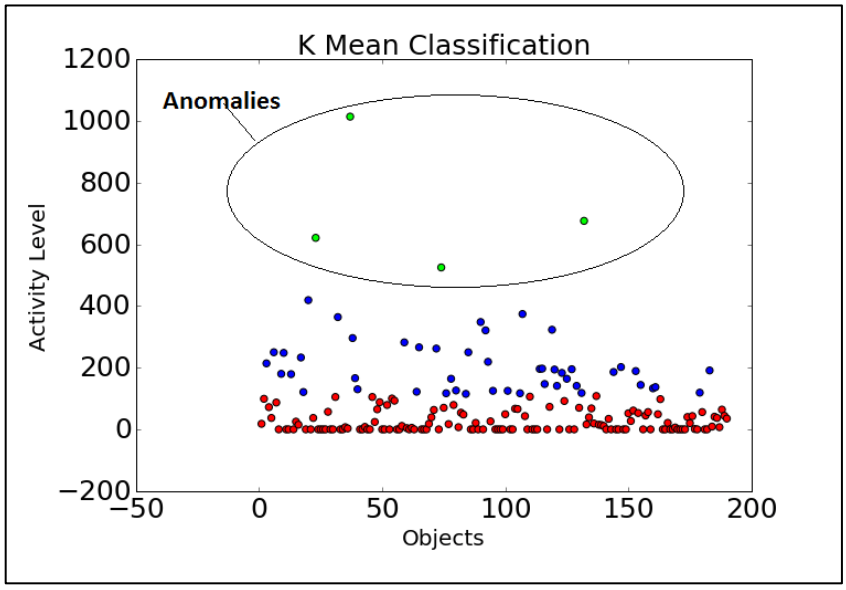


Рисунок 4.3.2.2.3 – Скріншот результату роботи моделі K-means для виявлення аномалій (запозичено із [26])

Принцип роботи K-means:

1. визначити кількість кластерів k;
2. ініціалізувати к випадкових центрів (центроїдів);
3. призначити кожну точку до найближчого центру;
4. обчислити нові центроїди як середнє значення точок у кожному кластері;
5. повторювати кроки 3-4, допоки центри не стабілізуються;

Формула функції втрат (яку мінімізує K-means):

$$J = \sum_{i=1}^{k} \sum_{x_j \in C_i} \left\| x_j - \mu_i \right\|^2, \qquad (4.3.2.2.8)$$

де C - множина точок у кластері i,

μ - центроїд кластера,

$\left\| x_j - \mu_i \right\|^2$ - квадрат евклідової відстані.

Для аномалій після кластеризації можна розрахувати відстань кожної точки до її центру. Якщо вона значно більша за середню у кластері — точка вважається відхиленням.

Із гіперпараметрів маємо такі:

– n_clusters - кількість кластерів к, важливо підібрати правильно (можна за допомогою «методу ліктя»);

– init — спосіб ініціалізації центрів (найкращий – «k-means++»);

– max_iter — максимальна кількість ітерацій для сходження;

– tol - точність зупинки (якщо зміни стали мінімальними);

Даний метод/модель простий у реалізації, добре працює на структурованих даних. Із мінусів, то не виявляє складні аномалії, залежить від кількості кластерів, не працює добре при різній щільності або формі кластерів.

Таблиця 4.3.2.2.1 - Порівняльний аналіз моделей для мультиваріантних викидів

Модель	Стійкість до шуму	Робота з великими даними	Інтерпретація	Обчислювальна вартість
Isolation Forest	Висока	Добра	Проста	Середня
Local Outlier Factor	Чутлива	Погано масштабується	Складна	Висока
K-means	Не стійка	Добра	Проста	Швидка

У моєму проекті вважаю, **найдоцільніше використовувати саме мультиваріантні викиди**, а не уніваріантні. Причина в тому, що логування у такій системі охоплює багато різних параметрів одночасно. Наприклад, один лог-запис може містити час запиту, IP-адресу, HTTP-метод, домен, мікросервіс, який його обробляє, код відповіді, тривалість обробки запиту, користувацький ID, повідомлення про помилку, тип дії рівень логування (INFO, WARN, ERROR тощо) - тобто десятки полів, які тільки в сукупності дають повне уявлення про поведінку системи,

Уніваріантний аналіз міг би розглядати лише одну змінну - наприклад виключно час відповіді або тільки HTTP-статус. Але це не дає повної картини. Може бути ситуація, коли код відповіді 200 виглядає нормальним, але при цьому поєднується з підозріло довгим часом обробки, незвичним шляхом запиту або з абсолютно новим шаблоном поведінки користувача. Кожен окремий параметр може виглядати нормально, але в сукупності — це вже аномалія.

Саме тому для такого типу систем, як у застосунку, потрібно обов’язково використовувати мультиваріантний підхід. Він дозволяє виявляти нетипові поєднання параметрів, які порушують нормальну поведінку, навіть якщо кожен

з них окремо не викликає підозр. Наприклад запит, що обробився менш ніж за секунду, повернув успішний статус і не мав помилок, може все одно бути аномальним, якщо він виник з невідомого IP, у незвичний час і спричинив навантаження на кілька мікросервісів одночасно. Такий рівень аналізу можливий для розгляду лише через мультивимірні алгоритми.

У межах мого додатку я реалізувала виявлення аномалій у логах мікросервісів за допомогою моделі машинного навчання Isolation Forest. Через обмеження Elasticsearch щодо типів підтримуваних ML-моделей, такі моделі безнаглядового навчання не можуть бути завантажені безпосередньо у elastic кластер.

Тому модель була натренована локально в Python, після чого результати її передбачення - тобто мітки, які вказують, чи є конкретний лог-запис аномальним чи ні - були додані до потоку логів. Ці збагачені логи потрапили до Elasticsearch, де вже могли бути візуалізовані через Kibana. Так вдалося зробити в певному роді колаборацію точності машинного виявлення аномалій із сильними інструментами аналітики та моніторингу в Kibana.

4.4 Оцінка якості моделей виявлення аномалій: Precision, Recall, F1-score. Порівняння алгоритмів за точністю.

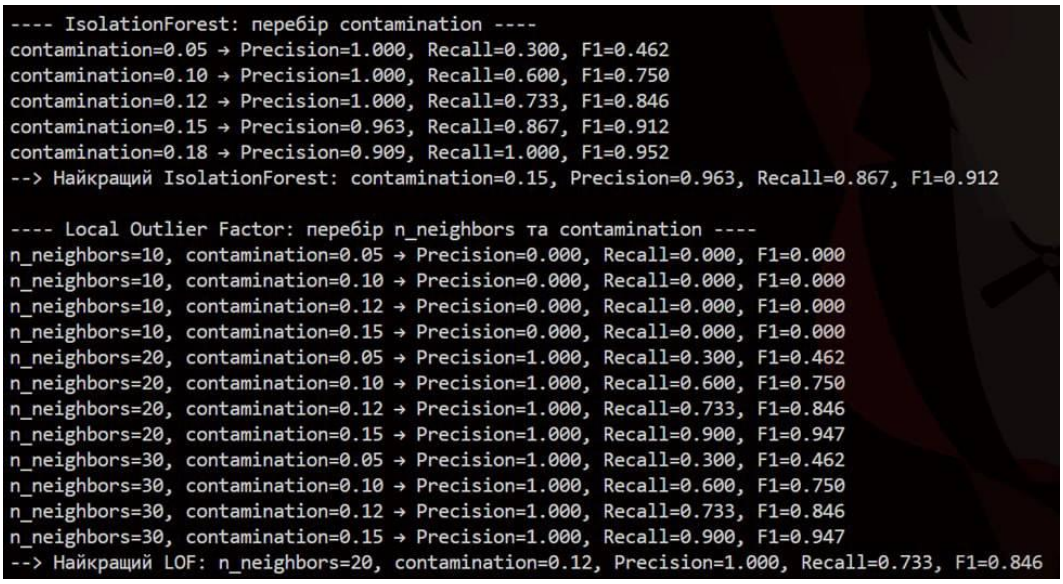


Рисунок 4.4.1 - Тестування параметрів моделей Isolation Forest і Local Outlier Factor для досягнення найвищих показників точності

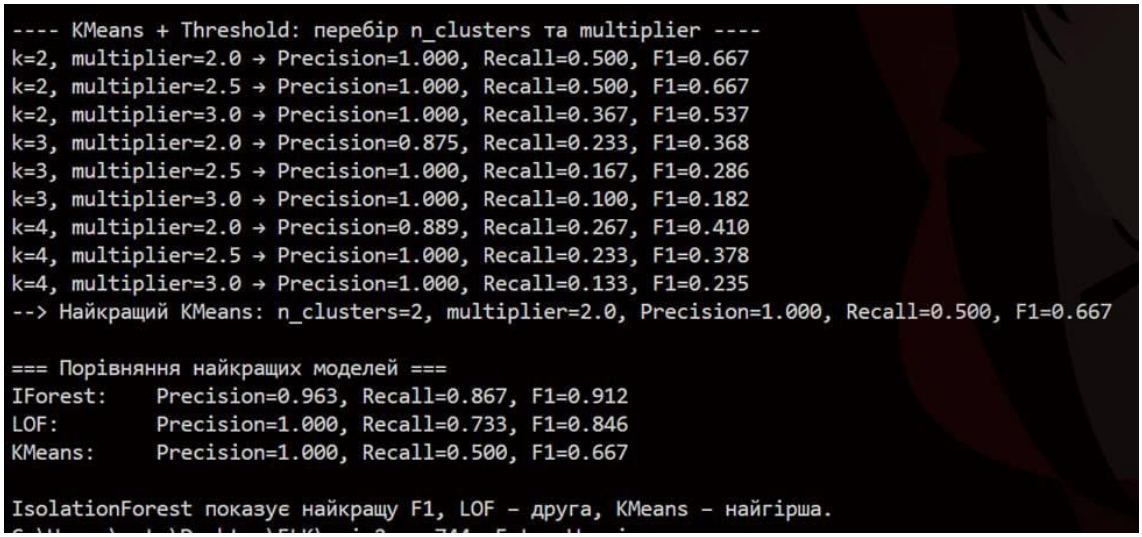


Рисунок 4.4.2 - Підбір гіперпараметрів і порівняння кластеризаційного підходу KMeans з моделями Isolation Forest та Local Outlier Factor

На наведених рисунках (4.4.1 та 4.4.2) я провела порівняння трьох моделей для виявлення аномалій: Isolation Forest, Local Outlier Factor (LOF) та

кластеризаційного підходу KMeans у зв’язці з пороговим відсіюванням. Для кожної моделі я підбирала гіперпараметри, а потім оцінила якість виявлення аномалій за класичними метриками - Precision, Recall та F1-score. Precision показує, який відсоток з передбачених як аномальні дійсно є аномаліями (тобто точність безпомилкових «тривог»). Recall вказує, яку частину всіх справжніх аномалій модель змогла знайти. А F1-score - це гармонійне середнє між Precision і Recall, яке дозволяє оцінити баланс між точністю і повнотою.

У випадку з KMeans я тестувала різні комбінації кількості кластерів (від 2 до 4) та множників (threshold-множників для виявлення викидів за відстанню від центру кластера). Найкращий результат показала конфігурація з двома кластерами і множником 2.0, де Precision становила 1.000 (жодного хибнопозитивного спрацювання), але Recall був лише 0.500 - тобто модель пропустила половину справжніх аномалій. Відповідно, F1-score дорівнював 0.667, що вказує на середню ефективність.

У моделі Isolation Forest я перебирала значення параметра contamination (очікувана частка аномалій у даних). Найкращим виявився варіант з contamination=0.15: Precision склала 0.963, Recall — 0.867, а F1-score — 0.912. Це означає, що модель зберігає високу точність, але й при цьому виявляє більшість реальних аномалій, на відміну від KMeans, де повнота була значно гіршою. У LOF також тестувався підбір кількості сусідів і параметра contamination. Найкраща конфігурація — n_neighbors=20 і contamination=0.12 — дала Precision 1.000, Recall 0.733, F1-score 0.846. Це свідчить про високу точність без хибних спрацьовувань, але менш повне охоплення справжніх аномалій, ніж у Isolation Forest.

Якщо підсумувати, **Isolation Forest виявився найкращим** за збалансованістю метрик, оскільки його F1-score був найвищим — 0.912. LOF показав хорошу точність, але нижчу повноту, що трохи знижує загальну ефективність. KMeans хоч і забезпечив абсолютну точність у найкращій

конфігурації, однак провалився за Recall - модель занадто обережно визначала аномалії, через що втрачала значну частину справжніх.

4.5 Інтеграція з Kibana: візуалізація аномалій

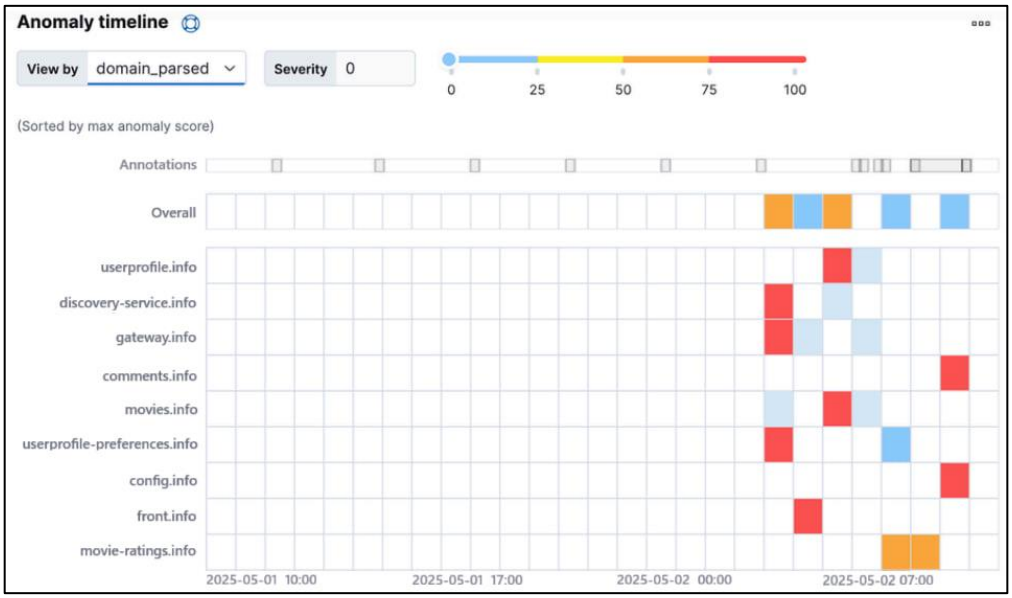


Рисунок 4.5.1 – Скріншот візуалізації аномалій у логах мікросервісів на основі передбачень моделі IF

На рисунку 4.5.1, який отримала в результаті, зображено часову шкалу аномалій, що були виявлені у логах окремих мікросервісів. Цей графік дозволяє наочно побачити, коли саме та в яких компонентах системи виникала нестандартна поведінка. Зліва у таблиці відображаються назви доменів логів, що відповідають конкретним мікросервісам (наприклад, comments.info - сервіс коментарів, movies.info — фільми, userprofile.info — профілі користувачів, gateway.info - шлюз тощо), а горизонтально тягнеться шкала часу, яка охоплює період з 1 по 2 травня 2025 року. Кожна клітинка на перетині сервісу та відрізку часу показує, чи були зафіксовані аномалії, і якщо так — наскільки серйозними вони були. Колір клітинки вказує на рівень аномальності: білий - усе в межах норми, блакитний - слабке відхилення, оранжевий - помірне, а червоний -

критичне. Саме червоні клітинки є найбільш тривожними, оскільки означають максимальний рівень аномалії. На графіку чітко видно, що вранці 2 травня одночасно кілька мікросервісів — таких як gateway.info, comments.info, discovery-service.info, movies.info, userprofile.info - почали демонструвати різкий стрибок аномалій. Ймовірною причиною появи таких масових аномалій у логах одразу кількох мікросервісів може бути системний збій у роботі одного з центральних компонентів, наприклад, API Gateway або Discovery Service. Якщо шлюз перестає правильно маршрутизувати запити, або Eureka неправильно реєструє сервіси, це може спричинити ланцюгову реакцію збоїв: сервіси не можуть знайти одне одного, зростає кількість помилок у відповідях, повторних спроб, нестандартних логів тощо. Також не можна виключати зміну конфігурації, оновлення або навантажувальний пік, який призвів до перевантаження системи. Оскільки аномалії виявлені одночасно в логах декількох ключових мікросервісів, це точно свідчить не про локальну помилку, а про загальний зсув у поведінці системи.

Рисунок 4.5.2 – Скріншот візуалізації аномалій за IP-адресами клієнтів на основі результатів моделі IF

Даний рисунок 4.5.2 також відображає візуалізацію аномалій, проте цього разу вони згруповані не за сервісами, а за IP-адресами клієнтів, тобто за параметром client_ip_parsed. Це означає, що на графіку показано, з яких саме IP-адрес надходив трафік, що мав аномальні характеристики, і в які саме моменти часу це відбувалося.

По вертикалі розміщені самі IP-адреси, з яких надходили запити. Наприклад: 192.168.168.20 , 192.168.169.5, 192.168.168.50 тощо. По горизонталі — часовий ряд з 1 по 2 травня. Кожна клітинка на перетині IP та моменту часу показує, чи були у цей проміжок часу запити з цієї IP-адреси аномальними. Колір клітинки знову ж таки вказує на рівень аномальності: білий - все в межах норми, блакитний - слабке відхилення, жовтий - помірне, оранжевий - суптєве, а червоний - критичне.

Судячи з графіка, найінтенсивніша активність з аномальними ознаками була виявлена 2 травня в ранкові години, і вона походила від IP-адрес 192.168.168.20, 192.168.161.5 . 192.168.168.50 та 192.168.168.37 - саме там видно найбільше червоних клітинок. Це може свідчити про кілька речей: або з цих адрес надходив надмірний трафік (наприклад, через DDoS-подібну активність або помилку в клієнтському коді), або ці клієнти почали надсилати запити, які не відповідали нормальній структурі (наприклад, з атиповими параметрами, підозрілою частотою або з великою кількістю помилоку відповідях сервера).

Також варто звернути увагу, що IP-адреси, які проявили аномалії, не є послідовними за підмережею - тобто це не одна підсистема чи серверна група, а можливо, різні користувачі або різні машини одного кластера. Це свідчить про те, що проблема могла виникнути або на рівні загального доступу до сервісу (наприклад, якщо один мікросервіс некоректно обробляв запити), або з боку окремих клієнтів, які через неправильне оновлення або конфігурацію почали надсилати аномальний трафік.

То цей графік допомагає точно локалізувати джерело аномалій — вже не за логікою сервісу, як у попередньому прикладі, а за кінцевими клієнтами, які спричинили нестандартну активність у системі.

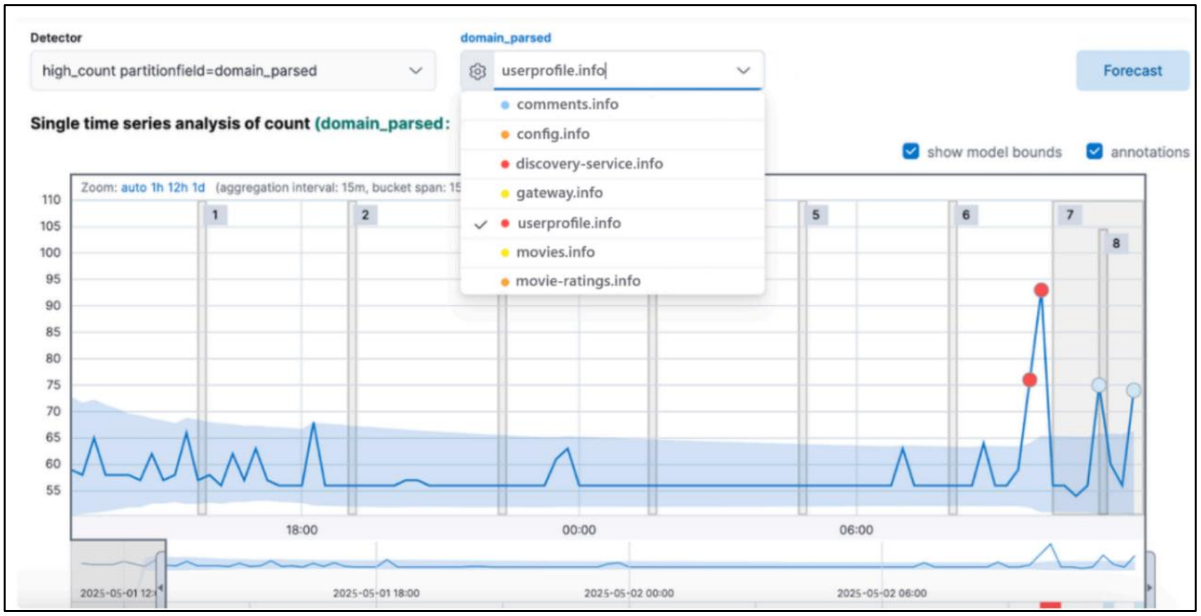


Рисунок 4.5.3 – Скріншот прогнозного графіка з аналізом кількості логів у сервісі userprofile.info з виявленням піків аномальної активності

На даному скрині 4.5.3 зображено одномірний часовий аналіз частоти логів для userprofile.info. У верхньому лівому куті видно, що обрано детектор high_count з параметром partitionfield-domain_parsed, тобто модель аналізує кількість логів, розбиту по окремих сервісах. У випадаючому списку можна змінити лог-домен (тобто обрати інший мікросервіс), але наразі активний саме userprofile.info, що відповідає за обробку профілів користувачів.

Синя лінія на графіку - це кількість логів у часі, а світло-блакитна область навколо неї - це границі моделі, в яких передбачалася нормальна поведінка (тобто нормальний рівень логів). Якщо фактична кількість логів виходить за межі цієї області, система розпізнає це як аномалію. Такі аномальні піки позначено червоними крапками. Видно, що найвиразніше відхилення сталося 2 травня

вранці - модель зафіксувала різкий стрибок активності, який суттєво вийшов за межі очікуваного.

На графіку також присутні вертикальні сіро-затемнені ділянки — це анімації аномальних сегментів, які дозволяють виділити періоди з найбільш критичними відхиленнями. У нижній частині - спрощене зображення всього тренду, яке допомагає зорієнтуватися по шкалі часу.

Найімовірніше, сплеск активності в сервісі userprofile.info пов’язаний із помилками взаємодії з іншими мікросервісами або підвищеним навантаженням через неправильну маршрутизацію запитів (наприклад, з боку шлюзу). Важливо, що система автоматично обчислює межі «нормальної» частоти логів і самостійно сигналізує про порушення, що суттєво полегшує моніторинг.

Рисунок 4.5.4 - Скріншот аналізу кількості помилок клієнта за IP-адресами в логах gateway.info з використанням моделі сумарного виявлення аномалій у Kibana

На цьому графіку (див. рисунок 4.5.4) зображено детальний аналіз помилок клієнтів, що проходять через API Gateway.

Детектор high_sum(client_error) використовується для визначення, коли певна IP-адреса генерує аномально високу кількість клієнтських помилок (наприклад, статуси 4xx). Ці помилки не вказують на проблеми сервера, а свідчать, що клієнт надсилає некоректні запити (наприклад, з помилковими параметрами або на неіснуючі маршрути).

У полі domain parsed обрано лог-домен gateway.info, що відповідає шлюзу, а в полі client_ip_parsed вибрано конкретну IP-адресу - 192.168.169.5 . Система будує часовий графік, на якому видно, як змінювалася кількість клієнтських помилок, що надходили з цієї адреси, у кожному 15-хвилинному проміжку. Синя лінія - це реальні дані, а світло-блакитна зона - це допустимі межі поведінки, які модель вважає нормальними. Після тривалого періоду повної відсутності помилок, ближче до 06:00 2 травня відбувається різкий стрибок до 7 помилок за 15 хвилин, що позначено червоною точкою — тобто критична аномалія.

На рисунку також видно випадаючий список з іншими IP-адресами (192.168.168.20, 192.168.168.37, 192.168.168.50 та інші), які також мають позначки з різним рівнем аномальності (жовтим, оранжевим, синім), що свідчить про масовий сплеск некоректної клієнтської активності. Це може бути спричинено неправильним оновленням клієнтської частини, помилками в маршрутизації запитів або навіть автоматичними скриптами, які викликають неправильні endpointin.

Цей тип графіка дозволяє швидко ідентифікувати, яка саме IP-адреса генерує найбільше проблем, у який момент часу, і наскільки це вибивається з нормальної поведінки.

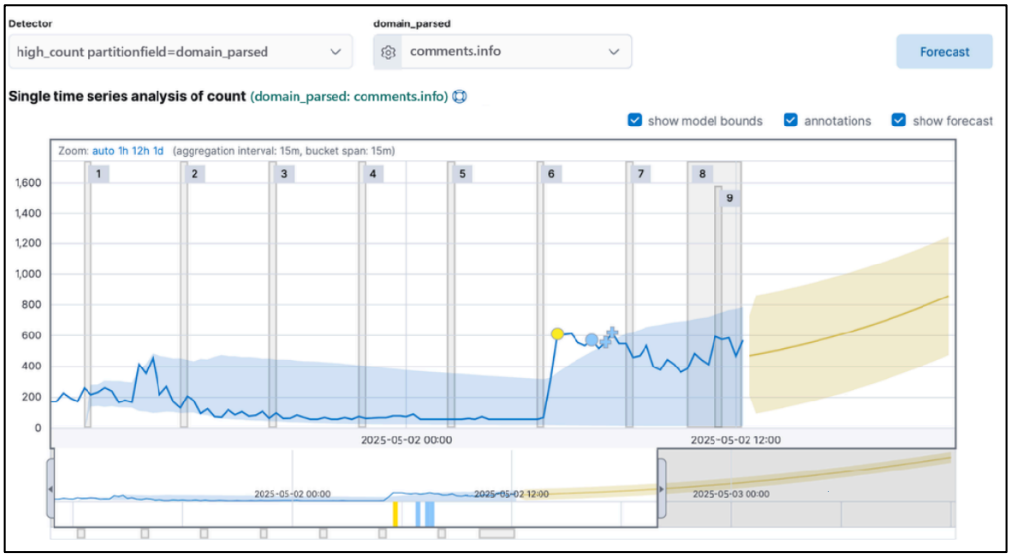


Рисунок 4.5.5 — Скріншот прогнозного графіка Kibana для сервісу comments.info з аналізом частоти логів та побудовою прогнозу на основі тренду

На цьому скріншоті 4.5.5 зображено графік, що ілюструє часовий аналіз частоти логів, отриманих від мікросервісу comments.info. У даному випадку використовується той самий детектор high_count partitionfield-domain_parsed, який відстежує збільшення кількості лог-записів у межах одного сервісу.

Обрана метрика - це кількість логів кожні 15 хвилин, а модель фіксує, чи ця кількість відповідає очікуваному діапазону.

Синя лінія на графіку - це реальна кількість логів, що надходили від сервісу. Світло-блакитна область навколо неї - межі, в яких система вважає поведінку нормальною. У певний момент, ближче до полудня 2 травня, спостерігається різке зростання активності - кількість логів зростає з майже нульового рівня до понад 400 записів за 15-хвилинний період. Цей стрибок був розпізнаний як аномальний, про що свідчить жовта точка на піку — вона позначає незначне, але суттєве відхилення від очікуваного.

Особливістю цього графіка є те, що включено прогноз (show forecast). Права частина графіка, починаючи після 2 травня 12:00, показує жовту область, яка відображає очікуване майбутнє значення кількості логів. Жовта крива - це

середній прогноз, а світло-жовта зона навколо - межі довіри (довірчий інтервал), що вказує на можливу варіацію. Видно, що прогноз моделі передбачає подальше зростання активності, що може свідчити про очікуваний підвищений трафік або повторну появу помилок.

Можна зробити висновок, що сервіс comments.info або почав масово генерувати логи через збільшення кількості запитів, або зазнав змін у поведінці - наприклад, після перезапуску чи оновлення. Попередній період до піку характеризувався майже повною відсутністю активності, що виглядає підозріло — можливо, сервіс був тимчасово недоступний або відключений. Потім — раптовий сплеск, що не вписується у нормальну модель.

5 ПЕРСПЕКТИВИ РОЗВІТКУ

Налаштована система вже добре справляється з виявленням аномалій у логах мого мікросервісного застосунку MovieMingle. Проте я розумію, що з часом навантаження на систему зростатиме, з’являтиметься більше логів і ситуацій, які потрібно обробляти швидше. Тому для подальшого розвитку важливо не зупинятись, а розширювати можливості – а саме, автоматизувати сповіщення про проблеми та зробити систему здатною працювати зі значно більшими обсягами даних.

5.1 Впровадження алертингу (Elastic Watcher, Prometheus Alertmanager)

На цьому етапі система вже виявляє аномалії та відображає їх у вигляді графіків у Kibana, але цього недостатньо для оперативного реагування. Якщо ніхто не відкриє Kibana саме в той момент, коли з’явиться проблема, то аномалія залишиться непоміченою. Щоб цього уникнути, потрібно реалізувати механізм автоматичних сповіщень, який повідомлятиме про підозрілу активність одразу після її виявлення. Для цього можна використати Elastic Watcher - вбудований функціонал у Elasticsearch, який дозволяє створювати спеціальні правила (так звані watch-и). Ці правила можуть перевіряти наявність аномалій у логах, і якщо умова спрацьовує (наприклад, виявлено різке зростання кількості помилок), то система автоматично надсилає повідомлення. Це може бути електронна пошта, повідомлення в Telegram, Slack або інший зручний канал. Ще один варіант - це зв’язка Prometheus + Alertmanager, де Prometheus слідкує за системними метриками сервісів, а Alertmanager займається логікою алертів: фільтрацією, згортанням однакових сповіщень і надсиланням повідомлень у потрібні канали. У майбутньому впровадження алертингу дозволить не тільки бачити проблему

після факту, а й оперативно отримувати повідомлення про неї — без потреби постійно моніторити вручну.

5.2 Масштабування обробки логів до мільйонів подій

На поточному етапі система працює стабільно при помірному обсязі логів, однак у майбутньому кількість подій у логах буде лише зростати. Коли в застосунку з’явиться більше користувачів, більше мікросервісів і більше запитів, це автоматично призведе до різкого збільшення обсягу логів - можливо навіть до мільйонів подій на день. Щоб система залишалася продуктивною та не «впиралася» в ліміти, потрібно заздалегідь продумати масштабування. У першу чергу це стосується Kafka, яка передає логи між компонентами - її можна масштабувати горизонтально, додавши кілька брокерів та налаштувавши партиції. Це дозволить розподілити потік логів рівномірно між кількома вузлами.

Logstash теж потрібно масштабувати, запустивши кілька екземплярів із паралельною обробкою - інакше він може стати вузьким місцем. Elasticsearch повинен працювати як кластер з кількома нодами і шардованими індексами — це дасть змогу розподіляти навантаження при індексації та запитах на пошук. Також варто налаштувати ILM (Index Lifecycle Management) - політику життєвого циклу індексів, яка автоматично видалятиме старі логи або переміщатиме їх в архів, щоб уникнути переповнення диска і зниження продуктивності. Усе це разом дозволить системі обробляти великі обсяги даних, не втрачаючи швидкість та стабільність.

ВИСНОВКИ

У ході цієї роботи була розроблена повноцінна система централізованого логування для мікросервісного застосунку MovieMingle, що не тільки збирає й обробляє логи, а й дає змогу автоматично виявляти аномалії за допомогою алгоритмів машинного навчання. У процесі реалізації я пройшла всі основні етапи - від проектування логуючої бібліотеки й налаштування Kafka до візуалізації даних у Kibana та побудови моделей у Python.

Завдяки впровадженню ELK-стеку вдалося зручно централізувати всі логи з різних мікросервісів. Це значно спростило аналіз поведінки системи, дозволило швидко знаходити джерела помилок і переглядати повну картину того, що відбувається «під капотом» додатку. Додавши машинне навчання, я зробила ще один крок уперед: тепер система може сама звертати увагу на підозрілі дії, навіть якщо вони не кидаються в очі людині.

Під час виконання проекту я отримала глибше розуміння, як влаштовані сучасні підходи до логування, з якими проблемами стикаються DevOps-команди (в плані виявлення аномалій) та як важливо не лише збирати інформацію, а й вміти її правильно інтерпретувати. Особисто для мене ця робота стала не просто технічним завданням, а справжнім досвідом побудови складного, але дуже корисного рішення для реального мікросервісного середовища.

Отримані результати можна впроваджувати в аналогічних системах, особливо там, де потрібна висока надійність та швидка реакція на будь-які збої. У майбутньому систему можна ще доповнювати - наприклад, розширити набір алгоритмів, додати більш адаптивну аналітику чи автоматичне оповіщення про знайдені проблеми. Але головне - фундамент закладено, і він доволі стабільно працює.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1 Mark Richard. Microservices vs. Service-Oriented Architecture, 1st Edition Gravenstein Highway Press, 2016 568c.

2 **Roman Beekeeper.** Що таке логування, де і як його використовувати. JavaRush. URL: https://javarush.com/groups/posts/2388-logirovanie-chto-kak-gde-i-chem. Дата звернення: 04.03.2025.

3 **Denis Tsyplakov.** Логування в Java. Habr. URL: https://habr.com/ru/articles/130195/. Дата звернення: 04.03.2025.

4 **Vlasov Pavel.** Реальне логування методів у Java/logback. Habr. URL: https://habr.com/ru/articles/463601/. Дата звернення: 04.03.2025.

5 **Кирило Грищук.** Про логування із прикладами. Kirya Tech Blog. URL: https://kirya522.tech/posts/logging/. Дата звернення: 10.04.2025.

6 **Stanley Ulili.** How to collect, process, and ship log data with vector. URL: https://betterstack.com/community/guides/logging/vector-explained/. Дата звернення: 10.04.2025.

7 **RisingWave Team.** How to Send Data from Kafka to Elasticsearch: A Step-by-Step Guide. URL: https://risingwave.com/blog/how-to-send-data-from-kafka-to-elasticsearch-a-step-by-step-guide/. Дата звернення: 10.04.2025.

8 **Elastic.** Just Enough Kafka for the Elastic Stack, Part 1. URL: https://www.elastic.co/blog/just-enough-kafka-for-the-elastic-stack-part1. Дата звернення: 10.04.2025.

9 **Ayooluwa Isaiah.** A beginners's guide to log management. URL: https://betterstack.com/community/guides/logging/log-management/. Дата звернення: 27.04.2025.

10 **Sematext.** Kafka Connect Elasticsearch: How to Guide. URL: https://sematext.com/blog/kafka-connect-elasticsearch-how-to/. Дата звернення: 27.04.2025.

11 **GeeksforGeeks.** Parse and Clean Log Files in Python. URL: https://www.geeksforgeeks.org/parse-and-clean-log-files-in-python/. Дата звернення: 27.04.2025.

12 **CodezUp.** From Logs to Insights: Effective Elasticsearch Log Analysis Techniques. URL: https://codezup.com/from-logs-to-insights-effective-elasticsearch-log-analysis-techniques/. Дата звернення: 27.04.2025.

13 **Ritvik Khanna.** How to Use Elasticsearch, Logstash, and Kibana to Visualise Logs in Python in Realtime. URL: https://www.freecodecamp.org/news/how-to-use-elasticsearch-logstash-and-kibana-to-visualise-logs-in-python-in-realtime-acaab281c9de/. Дата звернення: 27.04.2025.

14 **Elastic.** Getting started with search use cases in Python. URL: https://www.elastic.co/guide/en/cloud/current/ec-getting-started-search-use-cases-python-logs.html. Дата звернення: 27.04.2025.

15 **Badcasedaily1.** Виявлення проблем у log-файлах за допомогою аналітики. Habr. URL: https://habr.com/ru/companies/otus/articles/781598/. Дата звернення: 27.04.2025.

16 **Elastic.** Getting Started with Anomaly Detection. URL: https://www.elastic.co/docs/explore-analyze/machine-learning/anomaly-detection/ml-getting-started. Дата звернення: 27.04.2025.

17 **Mayur Koshti.** Real-Time Log Analysis with Elasticsearch, Python, and Kibana. Medium**.** URL: https://medium.com/pythoneers/real-time-log-analysis-with-elasticsearch-python-and-kibana-d51b3b966ebd. Дата звернення: 18.05.2025.

18 **David Braslow.** How to Build a Live Time Series Anomaly Detection Model. Metaplane**.** URL: https://www.metaplane.dev/blog/how-to-build-a-live-time-series-anomaly-detection-model. Дата звернення: 18.05.2025.

19 **Gavin Cohen.** Elasticsearch Machine Learning: An Improved Approach Using Correlated Anomaly Detection to Find Root Cause. ScienceLogic**.** URL:

https://sciencelogic.com/blog/elasticsearch-machine-learning-an-improved-approach-using-correlated-anomaly-detection-to-find-root-cause. Дата звернення: 18.05.2025.

20 **Ashnik Team.** Step-by-Step Guide: How to Configure Elastic Machine Learning for Anomaly Detection. URL: https://www.ashnik.com/step-by-step-guide-how-to-configure-elastic-machine-learning-for-anomaly-detection/. Дата звернення: 18.05.2025.

21 **Ahmad Bamie.** Anomaly Detection with Elasticsearch & Kibana (Part 1). Medium**.** URL: https://medium.com/bamieh-tech/anomaly-detection-with-elasticsearch-kibana-part-1-a0037b9607b6. Дата звернення: 18.05.2025.

22 **Kassie Khoo.** Anomaly detection with Z-score. URL: https://loyalty.dev/posts/anomaly-detection-with-z-score. Дата звернення: 18.05.2025.

23 **Karankajrolkar**. WHY “1.5” in IQR Method of Outlier Detection. URL: https://medium.com/@karankajrolkar/why-1-5-in-iqr-method-of-outlier-detection-b9301a95c704. Дата звернення: 18.05.2025.

24 **Geeksforgeeks**. What is isolation Forest? URL: https://www.geeksforgeeks.org/what-is-isolation-forest/. Дата звернення: 18.05.2025.

25 **Scikit**. Outlier detection with local outlier. URL: https://scikit-learn.org/stable/auto_examples/neighbors/plot_lof_outlier_detection.html. Дата звернення: 18.05.2025.

26 **Hazrat Ali.** Anomalu detection k-means clustering. URL: https://www.researchgate.net/figure/Anomaly-detection-through-k-means-clustering-first-dataset_fig4_326619835. Дата звернення: 18.05.2025.

ДОДАТОК А
ВИХІДНИЙ КОД КАСТОМНОЇ БІБЛІОТЕКИ

Лістинг А.1 – Основний код класу JsonLoggingFilter

```
package com.example.logging.filter;

import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
import org.springframework.web.util.ContentCachingRequestWrapper;
import org.springframework.web.util.ContentCachingResponseWrapper;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.net.InetAddress;
import java.nio.charset.StandardCharsets;
import java.time.Instant;
import java.util.*;


@Component
public class JsonLoggingFilter extends OncePerRequestFilter {

    private static final ObjectMapper mapper = new ObjectMapper();

    @Value("${spring.application.name:unknown-service}")
    private String serviceName;

    @Override
    protected void doFilterInternal(HttpServletRequest request0,
                                    HttpServletResponse response0,
                                    FilterChain chain)
            throws ServletException, IOException {

        var request = new ContentCachingRequestWrapper(request0);
        var response = new
ContentCachingResponseWrapper(response0);
        long start = System.currentTimeMillis();
        Exception exception = null;

        try {
            chain.doFilter(request, response);
        } catch (Exception ex) {
```

```
            exception = ex;
            throw ex;
        } finally {
            long duration = System.currentTimeMillis() - start;
            var m = buildLogMap(request, response, exception,
duration);

            String level;
            int status = response.getStatus();
            if (exception != null || status >= 500)      level =
"ERROR";
            else if (status >= 400)                      level =
"WARN";
            else                                         level =
"INFO";

            m.put("level", level);
            String json = mapper.writeValueAsString(m);

            switch (level) {
                case "ERROR" -> logger.error(json);
                case "WARN"  -> logger.warn(json);
                default      -> logger.info(json);
            }

            response.copyBodyToResponse();
        }
    }

    private Map<String, Object>
buildLogMap(ContentCachingRequestWrapper request,

ContentCachingResponseWrapper response,
                                          Exception exception,
                                          long duration) throws
IOException {
        Map<String, Object> m = new LinkedHashMap<>();
        m.put("timestamp", Instant.now().toString());
        m.put("service", serviceName);
        m.put("device", request.getHeader("User-Agent"));
        m.put("protocol", request.getProtocol());
        m.put("path", request.getRequestURI());
        m.put("method", request.getMethod());
        m.put("durationMs", duration);
        m.put("status", response.getStatus());
        m.put("host", InetAddress.getLocalHost().getHostName());
        m.put("clientIp", request.getRemoteAddr());

        // request
```

```
        Map<String, Object> req = new LinkedHashMap<>();
        var headers = new LinkedHashMap<String, String>();
        Collections.list(request.getHeaderNames())
                .forEach(h -> headers.put(h,
request.getHeader(h)));
        req.put("headers", headers);
        req.put("parameters", request.getParameterMap());
        m.put("request", req);

        // response
        Map<String, Object> resp = new LinkedHashMap<>();
        byte[] sb = response.getContentAsByteArray();
        if (sb.length > 0) {
            resp.put("body", new String(sb,

Optional.ofNullable(response.getCharacterEncoding()).orElse(Standar
dCharsets.UTF_8.name())));
        }
        m.put("response", resp);

        if (exception != null) {
            Map<String, Object> err = new LinkedHashMap<>();
            err.put("exception",
exception.getClass().getSimpleName());
            err.put("message", exception.getMessage());
            m.put("error", err);
        }
        return m;
    }
}
```

Лістинг А.2 – Основний код класу JsonServiceLogger

```
package com.example.logging.service;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.SerializationFeature;
import com.fasterxml.jackson.datatype.jsr310.JavaTimeModule;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import java.time.Instant;
import java.util.LinkedHashMap;
import java.util.Map;

@Component
```

```
public class JsonServiceLogger {
    private static final ObjectMapper MAPPER = new ObjectMapper();
    private final Logger log =
LoggerFactory.getLogger(JsonServiceLogger.class);

    @Value("${spring.application.name:unknown-service}")
    private String serviceName;

    public void info(String operation, Map<String,Object> input,
Map<String,Object> output) {
        log("INFO", operation, input, output, null);
    }
    public void warn(String operation, Map<String,Object> input,
Map<String,Object> output) {
        log("WARN", operation, input, output, null);
    }
    public void error(String operation, Map<String,Object> input,
Throwable ex) {
        log("ERROR", operation, input, Map.of(), ex);
    }

    private void log(String level,
                     String operation,
                     Map<String,Object> input,
                     Map<String,Object> output,
                     Throwable error) {
        try {
            MAPPER.registerModule(new JavaTimeModule());

MAPPER.disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);
            var m = new LinkedHashMap<String,Object>();
            m.put("timestamp", Instant.now().toString());
            m.put("service", serviceName);
            m.put("level", level);
            m.put("operation", operation);
            m.put("input", input);
            m.put("output", output);
            if (error != null) {
                m.put("error", Map.of(
                        "exception",
error.getClass().getSimpleName(),
                        "message", error.getMessage()
                ));
            }
            String json = MAPPER.writeValueAsString(m);
            switch (level) {
                case "ERROR" -> log.error(json);
                case "WARN"  -> log.warn(json);
                default      -> log.info(json);
```

```
            }
        } catch (Exception ex) {
            log.error("Failed to serialize service log", ex);
        }
    }
}
```

ДОДАТОК Б
ВИХІДНИЙ КОД РОБОТИ З PYTHON (ML)

Лістинг Б.1 – Основний код anomaly-detection.py

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.metrics import precision_score, recall_score, f1_score
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor
from sklearn.cluster import KMeans
from elasticsearch import Elasticsearch, helpers
from matplotlib.lines import Line2D

ES_HOST = "http://localhost:9200"
SOURCE_INDEX = "moviemingle-logs-2025.05.01"
DEST_INDEX = "moviemingle-logs-enriched"

es = Elasticsearch(ES_HOST)

def fetch_logs(index, size=10000):
    query = {"query": {"match_all": {}}}
    resp = helpers.scan(es, index=index, query=query)
    data = [doc['_source'] for doc in resp]
    return pd.DataFrame(data)

df = fetch_logs(SOURCE_INDEX)
df = df.dropna(subset=["duration_ms", "body_size", "status"])
df["duration_ms"] = df["duration_ms"].astype(float)
df["body_size"] = df["body_size"].astype(float)
df["status"] = df["status"].astype(float)

features = ["duration_ms", "body_size", "status"]
X_scaled = StandardScaler().fit_transform(df[features])
X_pca = PCA(n_components=2).fit_transform(X_scaled)
df["pca_1"] = X_pca[:, 0]
df["pca_2"] = X_pca[:, 1]

def evaluate_model(y_pred):
    return {
        "count_anomalies": int(np.sum(y_pred)),
        "percent": round(100 * np.mean(y_pred), 2)
```

```
    }

model_if = IsolationForest(contamination=0.12, random_state=42)
df["iforest_anomaly"] = (model_if.fit_predict(X_scaled) == -
1).astype(int)
score_if = evaluate_model(df["iforest_anomaly"])

model_lof = LocalOutlierFactor(n_neighbors=20, contamination=0.10)
df["lof_anomaly"] = (model_lof.fit_predict(X_scaled) == -
1).astype(int)
score_lof = evaluate_model(df["lof_anomaly"])

kmeans = KMeans(n_clusters=3, random_state=42)
labels = kmeans.fit_predict(X_scaled)
dist = np.linalg.norm(X_scaled - kmeans.cluster_centers_[labels],
axis=1)
threshold = dist.mean() + 2.5 * dist.std()
df["kmeans_anomaly"] = (dist > threshold).astype(int)
score_km = evaluate_model(df["kmeans_anomaly"])

df["is_anomaly"] = df["iforest_anomaly"]

def send_back_to_elastic(df, index):
    actions = [
        {"_index": index, "_source": row}
        for row in df.to_dict(orient="records")
    ]
    helpers.bulk(es, actions)

send_back_to_elastic(df, DEST_INDEX)

def plot_scatter(method, column, color):
    plt.figure(figsize=(6, 5))
    for idx, row in df.iterrows():
        if row[column] == 0:
            plt.scatter(row["pca_1"], row["pca_2"], c="blue",
marker="o", edgecolor="k", s=60)
        else:
            plt.scatter(row["pca_1"], row["pca_2"], c=color,
marker="X", edgecolor="k", s=60)
    plt.title(f"{method}: PCA-проєкція (○ Норма, ✕ Аномалія)")
    plt.xlabel("PCA 1")
    plt.ylabel("PCA 2")
    legend_elements = [
        Line2D([0], [0], marker='o', color='w', label='Норма',
               markerfacecolor='blue', markersize=10,
markeredgecolor='k'),
        Line2D([0], [0], marker='X', color='w', label='Аномалія',
```

```
                markerfacecolor=color, markersize=10,
markeredgecolor='k')
    ]
    plt.legend(handles=legend_elements)
    plt.tight_layout()
    plt.show()

plot_scatter("Isolation Forest", "iforest_anomaly", "red")
plot_scatter("Local Outlier Factor", "lof_anomaly", "orange")
plot_scatter("KMeans + Threshold", "kmeans_anomaly", "green")

if "domain" in df.columns:
    domain_counts = df[df["is_anomaly"] ==
1].groupby("domain").size().reset_index(name="anomaly_count")
    domain_counts["domain"] =
domain_counts["domain"].replace("config.info", "userprofile-
preferences.info")
    domain_counts = domain_counts.sort_values("anomaly_count",
ascending=False)

    plt.figure(figsize=(8, 5))
    sns.barplot(data=domain_counts, x="domain", y="anomaly_count",
palette="magma")
    plt.title("Кількість аномалій по доменах")
    plt.xlabel("Домен")
    plt.ylabel("К-сть аномалій")
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()

if "timestamp" in df.columns:
    df["timestamp"] = pd.to_datetime(df["timestamp"])
    df["hour_bucket"] = df["timestamp"].dt.floor("H")
    hourly = df.groupby("hour_bucket").agg(
        total_requests=("is_anomaly", "count"),
        anomalies=("is_anomaly", "sum")
    ).reset_index()

    plt.figure(figsize=(10, 4))
    plt.plot(hourly["hour_bucket"], hourly["total_requests"],
label="Усього", color="gray", linewidth=1.5)
    plt.bar(hourly["hour_bucket"], hourly["anomalies"], width=0.03,
label="Аномалії", color="red", alpha=0.6)
    plt.title("Таймсерія: загальна кількість запитів і аномалій")
    plt.xlabel("Час")
    plt.ylabel("Кількість")
    plt.xticks(rotation=45)
    plt.legend()
    plt.tight_layout()
```

```
plt.show()
```

ДОДАТОК В
ВИХІДНИЙ КОД DOCKER-COMPOSE ФАЙЛУ МІКРОСЕРВІСНОГО ДОДАТКУ

Лістинг В.1 – Основний код docker-compose.yml

```
services:
  rabbitmq:
    image: 'rabbitmq:3.6-management-alpine'
    container_name: rabbitmq_container
    ports:
      - '5672:5672'
      - '15672:15672'
    environment:
      AMQP_URL:
'amqp://rabbitmq?connection_attempts=5&retry_delay=5'
      RABBITMQ_DEFAULT_USER: "Sabina"
      RABBITMQ_DEFAULT_PASS: "Sabina2004Lena"
    networks:
      - microservices-network


  postgres:
    image: postgres:14.8-alpine3.18
    container_name: postgres_container
    environment:
      POSTGRES_DB: "movie_mingle_app_db"
      POSTGRES_USER: "postgres"
      POSTGRES_PASSWORD: "postgres"
      PGDATA: "/var/lib/postgresql/data/pgdata"
    volumes:
      - ./init-scripts:/docker-entrypoint-initdb.d
      - db-data:/var/lib/postgresql/data
    ports:
      - "5432:5432"
    networks:
      - microservices-network


  discovery-service:
    build:
      context: ./DiscoveryService
      dockerfile: Dockerfile
    container_name: discovery_container
    ports:
      - "8010:8010"
    environment:
```

```
      EUREKA_CLIENT_SERVICE_URL_DEFAULTZONE: "http://discovery-
service:8010/eureka"
    networks:
      - microservices-network


#  admin-service:
#    build:
#      context: ./AdminService
#      dockerfile: Dockerfile
#    container_name: admin_container
#    ports:
#      - "8085:8085"
#    networks:
#      - microservices-network


  config-service:
    build:
      context: ./ConfigServerApi
      dockerfile: Dockerfile
    container_name: config_container
    ports:
      - "8012:8012"
    depends_on:
      - rabbitmq
    networks:
      - microservices-network


  gateway-service:
    build:
      context: ./GatewayApi
      dockerfile: Dockerfile
    container_name: gateway_container
    ports:
      - "8082:8082"
    environment:
      - eureka.client.service-url.defaultZone=http://discovery-
service:8010/eureka
      #- eureka.client.serviceUrl.defaultZone=http://discovery-
service:8010/eureka
    depends_on:
      - discovery-service
      - rabbitmq
    networks:
      - microservices-network
```

```
  userprofile-service:
    build:
      context: ./UsersProfileApi
      dockerfile: Dockerfile
    container_name: userprofile_container
    environment:
      - PORT=0
      - eureka.client.service-url.defaultZone=http://discovery-
service:8010/eureka
      #- spring.cloud.config.uri=http://config-service:8012
      #- spring.boot.admin.client.url=http://admin-service:8085
    depends_on:
      - rabbitmq
      - postgres
      - discovery-service
      - gateway-service
      #- config-service
      #- admin-service
    networks:
      - microservices-network
    expose:
      - "8083"


  movie-service:
    build:
      context: ./MoviesApi
      dockerfile: Dockerfile
    container_name: movie_container
    environment:
      - PORT=0
      - eureka.client.service-url.defaultZone=http://discovery-
service:8010/eureka
#      - spring.cloud.config.uri=http://config-service:8012
    depends_on:
      - rabbitmq
      - postgres
      - discovery-service
      - gateway-service
#      - config-service
    networks:
      - microservices-network
    expose:
      - "8086"


  frontend-service:
    build:
      context: ./FrontedApi
```

```
      dockerfile: Dockerfile
    container_name: frontend_container
    environment:
      - PORT=0
      - eureka.client.service-url.defaultZone=http://discovery-
service:8010/eureka
#      - spring.cloud.config.uri=http://config-service:8012
    depends_on:
      - rabbitmq
      - postgres
      - discovery-service
      - gateway-service
#      - config-service
    networks:
      - microservices-network
    volumes:
      - ./default_pics:/app/default_pics
    expose:
      - "8084"


  comment-service:
    build:
      context: .
      dockerfile: CommentsApi/Dockerfile
#      context: ./CommentsApi
#      dockerfile: Dockerfile
    container_name: comment_container
    environment:
      - PORT=0
      - eureka.client.service-url.defaultZone=http://discovery-
service:8010/eureka
#      - spring.cloud.config.uri=http://config-service:8012
    depends_on:
      - rabbitmq
      - postgres
      - discovery-service
      - gateway-service
#      - config-service
    networks:
      - microservices-network
    expose:
      - "8087"
    volumes:
      - ./logs/comments:/logs/comments


  filebeat:
    image: docker.elastic.co/beats/filebeat:6.3.2
```

```
    container_name: filebeat
    user: root
    volumes:
      - ./filebeat/filebeat.yml:/usr/share/filebeat/filebeat.yml:ro
      - ./filebeat/logs:/usr/share/filebeat/logs:ro
      - /var/lib/docker/containers:/var/lib/docker/containers:ro
      - /var/run/docker.sock:/var/run/docker.sock:ro
    depends_on:
      - elasticsearch
      - logstash
    networks:
      - elk

  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:6.3.2
    container_name: elasticsearch
    environment:
      - discovery.type=single-node
      - xpack.security.enabled=false
      - bootstrap.memory_lock=true
      - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
    ulimits:
      memlock:
        soft: -1
        hard: -1
    ports:
      - "9200:9200"
    networks:
      - microservices-network

  logstash:
    image: docker.elastic.co/logstash/logstash:6.3.2
    container_name: logstash
    volumes:
      -
./logstash/config/logstash.yml:/usr/share/logstash/config/logstash.
yml:ro
      -
./logstash/pipeline/logstash.conf:/usr/share/logstash/pipeline/logs
tash.conf:ro
      - ./logs/comments:/logs/comments:ro
    depends_on:
      - elasticsearch
    networks:
      - microservices-network

  movie-rating-service:
    build:
      context: ./MovieRatingsApi
```

```
      dockerfile: Dockerfile
    container_name: movie_rating_container
    environment:
      - PORT=0
      - eureka.client.service-url.defaultZone=http://discovery-
service:8010/eureka
#      - spring.cloud.config.uri=http://config-service:8012
    depends_on:
      - rabbitmq
      - postgres
      - discovery-service
      - gateway-service
#      - config-service
    networks:
      - microservices-network
    expose:
      - "8088"

  userprofile-preference-service:
    build:
      context: ./UserProfilePreferences
      dockerfile: Dockerfile
    container_name: userprofile_preference_container
    environment:
      - PORT=0
      - eureka.client.service-url.defaultZone=http://discovery-
service:8010/eureka
#      - spring.cloud.config.uri=http://config-service:8012
    depends_on:
      - rabbitmq
      - postgres
      - discovery-service
      - gateway-service
#      - config-service
    networks:
      - microservices-network
    expose:
      - "8089"

  kafka:
    image: confluentinc/cp-kafka:7.2.1
    container_name: kafka
    ports:
      - "9092:9092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
```

```
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    depends_on:
      - zookeeper
    networks:
      - microservices-network

  zookeeper:
    image: confluentinc/cp-zookeeper:7.2.1
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    networks:
      - microservices-network


  kibana:
    image: docker.elastic.co/kibana/kibana:6.3.2
    container_name: kibana
    ports:
      - "5601:5601"
    environment:
      ELASTICSEARCH_URL: http://elasticsearch:9200
      ELASTICSEARCH_HOSTS: http://elasticsearch:9200
    networks:
      - microservices-network
    depends_on:
      - elasticsearch


volumes:
  db-data:


networks:
  microservices-network:
    driver: bridge
```