Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Кваліфікаційна робота

освітньо-кваліфікаційний рівень - бакалавр

на тему: «**Розробка веб-застосування на мікросервісній архітектурі та порівняння Microsoft Azure та AWS для його розгортання**»

Виконав: студент 4-го року навчання,

Освітньої програми «Інженерія програмного забезпечення», 121

Ахмадов Олексій Сергійович

Керівник Гречко А.В., кандидат фіз.-мат. наук, ст. викладач.

Рецензент _____
(прізвище та ініціали)

Кваліфікаційна робота захищена

з оцінкою _____

Секретар ЕК _____

«____» _____ 20____ р.

Київ – 2025

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав. Кафедри мультимедійних систем,

_____ А.В. Гречко

(підпис)

“____” _____ 2025 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

Для кваліфікаційної роботи студенту Ахмадову Олексію Сергійовичу
факультету інформатики 4-го курсу БП

ТЕМА: Розробка веб-застосування на мікросервісній архітектурі та порівняння Microsoft Azure та AWS для його розгортання

Зміст ТЧ до курсової роботи:

- Індивідуальне завдання
- Календарний план
- Вступ
- Розділ 1. Аналітичний огляд та вимоги до системи
- Розділ 2. Теоретичні основи мікросервісних архітектур
- Розділ 3. Проєктування та реалізація MentorMatch
- Розділ 4. Деплой мікросервісів у хмарах: Microsoft Azure vs AWS
- Висновки
- Список використаної літератури

Дата видачі “____” _____ 2025р. Керівник_____

(підпис)

Завдання отримав _____

(підпис)

Тема: Розробка веб-застосування на мікросервісній архітектурі та порівняння Microsoft Azure та AWS для його розгортання

Календарний план виконання роботи:

п/п	Назва етапу	Термін виконання етапу	Примітка
1.	Отримання теми курсової роботи	01.11.2024 – 05.11.2024	
2.	Вивчення предметної області	06.11.2024 – 20.12.2024	
3.	Вивчення технологій для розробки	21.12.2024 – 02.01.2025	
4.	Побудова технічного завдання	03.01.2025 – 13.01.2025	
5.	Написання практичної частини роботи	20.01.2025 – 25.03.2025	
6.	Написання текстової частини роботи	01.04.2025 – 15.05.2025	
7.	Перегляд змісту роботи з керівником	16.05.2025	
8.	Внесення змін до курсової роботи відповідно до зауважень наукового керівника	16.05.2025 – 20.05.2025	
9.	Створення презентації	20.05.2025 – 28.05.2025	
10.	Захист роботи	02.06.2025 – 04.06.2025	

Студент Ахмадов О.С.

Керівник Гречко А.В.

“_____” _____

Анотація

У дипломній роботі розглянуто процес розробки веб-застосування на основі мікросервісної архітектури з порівнянням хмарних платформ Microsoft Azure та Amazon Web Services (AWS) для його розгортання. Як приклад реалізовано систему MentorMatch — платформу для взаємодії менторів і менті, побудовану з окремих сервісів для аутентифікації, профілювання, обробки сесій, рейтингів і сповіщень.

Усі компоненти контейнеризовано за допомогою Docker і розгорнуто в Kubernetes-кластері. Для обміну подіями використано Apache Kafka. Зберігання даних організовано через керовані інстанси PostgreSQL. Інтеграцію з хмарними менеджерами секретів реалізовано на основі Azure Key Vault та AWS Secrets Manager.

Проведено порівняння платформ за критеріями зручності конфігурування, безпеки, масштабованості та автоматизації розгортання. У результаті показано, що обидва середовища придатні для створення cloud-native застосунків, хоча відрізняються ступенем автоматизації та підходами до керування інфраструктурою.

Зміст

Вступ

Сучасні веб-застосунки повинні швидко адаптуватися до зростаючого навантаження та змін бізнес-вимог. Традиційна монолітна архітектура часто не дозволяє оперативно реагувати на такі зміни, внаслідок цього дедалі частіше використовується мікросервісний підхід. Мікросервіси дають змогу розбити складну систему на менші компоненти, які можуть незалежно масштабуватись, оновлюватись та розгортатись. Це робить застосунок гнучким, стійким до збоїв та простим у підтримці.

Тому метою дипломної роботи є розробка веб-застосунку на мікросервісній архітектурі та його подальше розгортання у двох провідних хмарних середовищах — Microsoft Azure та Amazon Web Services (AWS). Вибір цих платформ пояснюється їх популярністю, широкими можливостями для побудови cloud-native застосунків та суттєвими відмінностями в підходах до налаштування інфраструктури.

У сучасному IT-середовищі мікросервісна архітектура стала стандартом для побудови масштабованих, гнучких і надійних програмних систем. Її поява змінила підходи до розробки веб-застосунків, дозволяючи створювати незалежні сервіси, які легко підтримуються, оновлюються та масштабуються окремо один від одного.

Зрозуміємо, де і коли треба використовувати Microsoft Azure та Amazon Web Services, яка з хмарних платформ більш зручна для швидкого старту, яка забезпечує більш глибокий контроль над інфраструктурою, та які інструменти краще підходять для підтримки й розвитку високонавантажених мікросервісних веб-застосунків.

Розділ 1. АНАЛІТИЧНИЙ ОГЛЯД ТА ВИМОГИ ДО СИСТЕМИ

Протягом останнього десятиріччя cloud-native development із використанням мікросервісної архітектури сформувався як галузевий стандарт для масштабованих веб-платформ. У різних галузях від FinTech до e-Learning поступово відмовляються від монолітної архітектури через потребу в:

- еластичному горизонтальному масштабуванні під пікові навантаження;
- CI/CD-циклі з можливістю випуску оновлень кілька разів на день;
- fault-isolation, що запобігає повному відмовленню системи у разі збою окремого сервісу.

Водночас український IT-ринок характеризується браком якісних менторських платформ, адаптованих до локального контексту: україномовного інтерфейсу, правових і культурних особливостей, потреб корпоративного навчання. Наявні міжнародні сервіси як: ADPList, MentorCruise, GrowthMentor, не враховують ці особливості: мають обмежену підтримку локалізації, не підтримують кастомізацію або фінансово недоступні для невеликих організацій.

Актуальність створення MentorMatch як open-source менторської платформи з гнучким налаштуванням, можливістю white-label-розгортання та безкоштовними базовими функціональними можливостями зумовлена зазначеними викликами. Вибір мікросервісної архітектури та хмарного розгортання на Azure та AWS забезпечує відповідність вимогам production-рівня: високу доступність, спостережуваність та масштабованість.

1.1 Аналіз існуючих рішень

Присутні на ринку системи аналоги менторських платформ можна умовно поділити дві групи:

- волонтерські (наприклад ADPList) - безкоштовні, але переважно орієнтовані на загальний ринок, без SLA й корпоративних інтеграцій;
- комерційні SaaS (MentorCruise, GrowthMentor) - пропонують широкий набір функціональних можливостей, проте лишаються англомовними й працюють за підпискою на користувача, що робить їх менш придатними для освітніх закладів чи невеликих компаній.

Основні проблеми, які не вирішують існуючі альтернативи розроблюваної в даній роботі системи:

- відсутність підтримки української мови (локалізації);
- неможливість самостійного розгортання або використання в режимі white-label;
- закриті API, що ускладнюють інтеграцію з внутрішніми HR-системами компаній.

Вказані виклики усуваються в MentorMatch завдяки відкритому ядру, підтримці white-label-розгортання та гнучким алгоритмам підбору менторів відповідно до потреб конкретної організації.

1.2 Вибір технологій розробки

Вибір технологій розробки зумовлений наступними ключовими потребами:

- **Контейнеризація.** Використовується Docker — забезпечує reproducible builds, дозволяє швидко розгорнути середовище розробника та спростити CI/CD-процеси. Це стандарт де-факто у production для мікросервісної архітектури.
- **Оркестрація.** Обрано Kubernetes (через AKS або EKS залежно від хмарного провайдера). Його переваги включають self-healing механізми, автоматичне масштабування та широку підтримку через Helm-чарти. Kubernetes забезпечує стабільність і гнучкість у production.

- **Event streaming.** Використовується Apache Kafka — платформа з високою пропускною здатністю, гарантіями збереження (durability) та підтримкою exactly-once семантики. Такий підхід є необхідним для підтримки аудиту, аналітики подій і масових розсилок.
- **Транзакційна база даних.** Основна база — PostgreSQL. Вона поєднує ACID-гарантії з підтримкою напівструктурованих даних через JSONB. Завдяки цьому забезпечується можливість одночасного використання реляційної моделі для core-даних і зберігання атрибутів, як-от навички (skills), у гнучкому форматі.
- **Аутентифікація та авторизація.** Використовуються JWT tokens. Вони stateless і добре інтегруються в мікросервісну архітектуру. Claims у токенах дозволяють реалізувати RBAC (Role-Based Access Control) без додаткового звернення до бази.

Observability. Пара Prometheus + Grafana — це Kubernetes-native рішення для збору метрик і побудови дашбордів. Вони безкоштовні, масштабовані, підтримують алерти через Alertmanager, і є стандартним стеком у більшості production-систем.

1.3 Функціональні вимоги

В результаті аналізу систем аналогів було сформульовано такі функціональні вимоги до системи:

- Реєстрація та авторизація через JWT, підтримка ролей: MENTEE / MENTOR / ADMIN
- Профілі менторів і менті з фільтрацією за навичками, компаніями, рейтингом
- Бронювання сесій: запит, підтвердження, скасування, завершення
- Система відгуків і рейтингів для обох сторін

- Сповіщення подій у системі

1.4 Нефункціональні вимоги

Окрім основної функціональності, до системи було висунуто певні нефункціональні вимоги:

1. **Масштабованість**
    - система має підтримувати горизонтальне масштабування мікросервісів до 10 реплік (перевірено в тестовому середовищі з Kubernetes);
    - час запуску нової репліки ≤ 60 с при використанні Kubernetes (EKS/AKS) та Docker-контейнерів.
2. **Продуктивність**
    - середній час відповіді API ≤ 400 мс при середньому навантаженні (до 100 одночасних користувачів).
3. **Надійність і доступність**
    - система забезпечує базову доступність ≥ 99.5%;
    - автоматичне відновлення подів після збою ≤ 60 с реалізоване через механізми Kubernetes
4. **Безпека**
    - всі API виклики захищено за допомогою HTTPS;
    - використовується JWT-токен з терміном дії 60 хвилин; можливе оновлення через refresh-токени;
    - реалізована авторизація на основі ролей з обмеженням доступу до ендпоінтів.

5. **Контейнеризація та оркестрація**
    - кожен мікросервіс пакується в Docker-контейнер з окремим Dockerfile;
    - застосунок оркеструється через Kubernetes (AKS або EKS);
    - включено HPA, що забезпечує масштабування на основі CPU;
    - час перезапуску контейнера ≤ 30 с.
6. **Інтеграція і сумісність**
    - взаємодія між мікросервісами через REST і Kafka виконується з часом відгуку ≤ 300 мс у більшості сценаріїв;
    - Інтеграційні тести виконуються вручну на pre-prod середовищі перед кожним релізом.
7. **Відмовостійкість**
    - kafka Consumer повторно обробляє повідомлення після помилки кожні ≤ 1 секунд у межах постійного циклу споживання, за умови неуспішної обробки події;
    - health-check усіх мікросервісів виконується через Kubernetes probes (перевірка кожні 10 с).

1.5 Постановка задачі й критерії успіху

Постановка задачі

У рамках курсової роботи передбачено створити хмарну мікросервісну платформу для менторства MentorMatch, орієнтовану на український ринок і внутрішні освітньо-корпоративні ініціативи. Проект буде реалізовано з урахуванням вимог до локалізації, відкритої архітектури та можливості масштабування у хмарних середовищах.

Окрему увагу буде зосереджено на технічному порівнянні особливостей розгортання платформи у Microsoft Azure та Amazon Web Services (AWS).

Реалізація проекту охоплюватиме такі завдання:

1. Розробка застосування:
    - Побудувати набір мікросервісів: auth, profile, session_rating, notification;
    - Застосувати Docker, PostgreSQL, Apache Kafka та JWT для реалізації безпечної аутентифікації;
    - Забезпечити українську локалізацію інтерфейсу та можливість використання white-label-брендування;
2. Підготовка інфраструктури:
    - Створити конфігурацію інфраструктури у форматі Infrastructure-as-Code (Helm/YAML);
    - Додати механізми автоматичного масштабування (HPA), самовідновлення, управління секретами;
3. Хмарні сценарії розгортання
    - Розгортання в Azure: ACR + AKS з використанням Load Balancer і Ingress-контролера;
    - Розгортання в AWS: ECR + EKS з AWS Load Balancer Controller;
4. Моніторинг і спостережуваність
    - Інтегровати Prometheus, Grafana, Alertmanager;
    - Зібрати базові метрики: HTTP latency, Kafka backlog, використання ресурсів CPU/RAM;

5. Порівняльний аналіз Azure та AWS
    - Оцінити час налаштування кластерів, зручність інструментів, автоматизацію масштабування та базову вартість експлуатації (Free Tier + on-demand).

Критерії успіху

1. Production-ready інфраструктура:
    - усі сервіси контейнеризовано в Docker, час старту ≤ 60 с.
    - kubernetes підтримує HPA; відновлення pod після збою ≤ 30 с.
    - hobservability: щонайменше 5 дашбордів Grafana.
2. Функціональна цілісність MVP:
    - реалізовано реєстрацію, логін (JWT із терміном дії 60 хв + refresh), розмежування прав доступу (MENTEE, MENTOR, ADMIN);
    - функціонує пошук менторів із фільтрацією за скілами, компанією, рейтингом;
    - працює повний життєвий цикл менторської сесії: створення запиту → підтвердження → завершення/скасування;
    - реалізовано рейтинг і відгуки, показ середнього рейтингу;
    - notification Service обробляє події Kafka; повідомлення створюються ≤ 1 с після події.
3. Деплой у хмарі:
    - повний стек успішно розгортається в Azure та AWS з використанням автоматизованого сценарію;

- ingress-контролер надає публічний доступ до застосунку, перевірено через curl.

4. Відкрита архітектура:
    - усі REST-ендпоінти документовано у форматі OpenAPI 3.0 (Swagger).

Розділ 2. ТЕОРЕТИЧНІ ОСНОВИ МІКРОСЕРВІСНИХ АРХІТЕКТУР

2.1 Еволюція архітектур ПЗ від моноліту до мікросервісів

Розробка програмного забезпечення пройшла довгий шлях еволюції архітектур – від монолітних застосувань до сучасних розподілених мікросервісів та хмарних рішень. Цей процес зумовлений потребами технологічного прогресу, бізнесу та масштабування [1].

У таблиці нижче наведено короткий огляд основних архітектур, що сформували підхід до побудови програмних систем на різних етапах розвитку IT-індустрії:

Тип архітектури	**Період активного використання**	**Ключові особливості**
Монолітна	1950–2010	Весь застосунок — одна єдина система
Модульна (модульний моноліт)	1980–досі	Розділення коду на модулі, але спільне розгортання
Service-Oriented Architecture (SOA)	2000–2010	Орієнтація на сервіси з важкою middleware
Мікросервісна	2010–досі	Кожен сервіс — окремий компонент із власним деплоєм

Тип архітектури	**Період активного використання**	**Ключові особливості**
Хмарно-орієнтована (Cloud-native)	2015–досі	Використання контейнерів, Kubernetes, серверлесс-підходів

Кожна архітектура відображає еволюцію мислення у створенні програмних систем — від централізованих рішень до гнучких і розподілених. Ці підходи демонструють, як змінювалися технологічні та бізнес-пріоритети у розробці ПЗ залежно від вимог часу.

У межах наступних підрозділів буде послідовно розглянуто кожен з представлених архітектурних стилів. Ми почнемо з монолітної архітектури, яка історично є найстарішою, та поступово перейдемо до більш сучасних підходів, зосереджених на гнучкості, масштабуванні й автономності компонентів.

2.1.1 Монолітна архітектура

Монолітна архітектура означає, що всі компоненти системи — бізнес-логіка, інтерфейс, доступ до бази даних — об’єднані в одну програму. Такий підхід тривалий час залишався домінуючим у корпоративному середовищі. Застосунок будується за рівневою схемою: презентаційний рівень (UI, контролери), прикладний рівень (сервіси, транзакції), доменний рівень (сутності, правила), інфраструктурний рівень. Усі шари збираються в єдиний артефакт, який розгортається як одне ціле.

Перевагами цього підходу є простота організації коду та деплою: один репозиторій, одна збірка, єдиний процес розгортання. Висока швидкодія досягається завдяки локальним викликам. Налагодження прозоре, бо все виконується в одному середовищі. Проте зростання команди та системи виявляє

обмеження. Зміни в одному модулі потребують перескладання всього застосунку. Масштабування відбувається лише на рівні всього контейнера. Втрата чіткої структури з часом призводить до появи "Big Ball of Mud" — хаотичної системи без чітких меж. Оновлення технологій ускладнюється через тісне переплетення компонентів [2].

2.1.2 Модульна архітектура

Модульна архітектура зберігає цілісність моноліту, але вводить логічне розділення на ізольовані модулі. Кожен модуль відповідає за конкретну частину бізнес-логіки та має чітко визначені межі відповідальності. Усі модулі працюють у межах одного процесу, зберігаючи спільну інфраструктуру та середовище виконання. Це дозволяє покращити структуру коду та спростити тестування.

Завдяки модульності з’являється можливість поступового переходу до мікросервісної архітектури — модуль можна винести в окремий сервіс без значної переробки. Підтримується єдина транзакція, що знижує складність. DevOps-навантаження залишається мінімальним: один контейнер, один процес.

Водночас у модульного підходу зберігаються деякі недоліки моноліту. Спільний життєвий цикл не дозволяє оновлювати модулі незалежно. Масштабування відбувається на рівні контейнера. Якщо не контролювати залежності, модулі можуть почати напряму звертатися до внутрішніх елементів інших модулів, що руйнує модульність і ускладнює підтримку [3].

2.1.3 Service-Oriented Architecture (SOA)

Сервісно-орієнтована архітектура побудована навколо окремих сервісів, кожен з яких реалізує конкретну бізнес-функцію. Взаємодія між сервісами відбувається через мережу за допомогою стандартизованих протоколів, найчастіше SOAP, WSDL, XML. Центральну роль у взаємодії виконує Enterprise

Service Bus (ESB), що відповідає за маршрутизацію, трансформацію даних і управління повідомленнями.

До переваг SOA належать модульність, повторне використання компонентів, зручна інтеграція різнорідних систем, масштабованість окремих сервісів, незалежне оновлення компонентів і підтримка індустріальних стандартів.

Натомість складність архітектури, необхідність налаштування ESB, повільна комунікація через обсяг XML-даних і централізована точка відмови створюють суттєві проблеми. Згодом SOA втратила популярність через обмежену гнучкість і складну підтримку, особливо у хмарних середовищах. У результаті архітектура була витіснена мікросервісним підходом [4].

2.1.4 Мікросервісна архітектура

Історично більшість програмних систем створювалися у вигляді монолітної архітектури, де вся логіка — бізнес, авторизація, інтерфейс, робота з базою даних — об'єднувалася в єдиний виконуваний застосунок. Такий підхід був простим у реалізації на початковому етапі, але з часом масштабування, тестування, супровід та впровадження змін ускладнювалися. Навіть незначне оновлення вимагало повторного розгортання всього застосунку, а збій одного модуля міг зупинити всю систему.

Відповіддю на ці обмеження стала сервіс-орієнтована архітектура (SOA), яка передбачала розбиття системи на логічно ізольовані сервіси. Проте SOA, особливо з використанням ESB, часто виявлялася надто громіздкою для сучасних гнучких продуктів. У 2010-х роках компанії-новатори, такі як Netflix, Amazon та Spotify почали шукати нові підходи, що дозволили б швидше впроваджувати нові функції. Так виникла мікросервісна архітектура — наступна еволюційна сходинка у проектуванні ПЗ.

Термін **microservices** набув популярності після архітектурного воркшопу у 2011 році, хоча перші згадки, зокрема термін “micro web services”, ще у 2005-му приписують доктору Пітеру Роджерсу — архітектору програмного забезпечення, який одним із перших описав ідею розподілених сервісів, кожен з яких виконує одну бізнес-функцію.

Концептуально, мікросервіси — це набір невеликих, незалежно розгортуваних сервісів, кожен з яких виконує чітко визначену бізнес-функцію. Такі сервіси комунікують між собою через легковагі протоколи — HTTP/REST, gRPC або асинхронно через брокери подій, наприклад Apache Kafka.

Кожен мікросервіс має власний контекст даних — тобто окрему базу даних або сховище. Це дозволяє уникнути спільного «мутуючого стану» та дає можливість реалізовувати архітектуру polyglot persistence, коли різні сервіси можуть використовувати PostgreSQL, MongoDB, DynamoDB залежно від задачі.

У порівнянні з монолітною архітектурою, мікросервіси мають переваги, які роблять їх ефективним рішенням для складних і динамічних систем. Однією з ключових переваг є незалежне масштабування: навантаження на систему не потребує масштабування всього застосунку — масштабуються лише ті сервіси, які дійсно цього потребують. Це дозволяє раціонально використовувати ресурси та гнучко реагувати на зміну навантаження. Також мікросервіси забезпечують стійкість до збоїв — у разі помилки одного сервісу загальна робота системи не припиняється. Гнучкість у розробці досягається завдяки тому, що кожна команда працює автономно над власним сервісом і може впроваджувати зміни без потреби узгодження з іншими модулями.

Попри перелічені переваги, мікросервісна архітектура має і низку недоліків. Найпомітніше — зростання загальної складності системи. Замість одного процесу з’являється десятки сервісів, кожен із яких потребує окремого CI/CD-пайплайна, а також централізованого логування й моніторингу. Питання узгодженості даних також стає критичним: класичні транзакції, що охоплюють

кілька сервісів, неможливі, тому виникає потреба у використанні патернів забезпечення цілісності, як-от saga або event sourcing. Крім того, мережеві виклики, на яких базується взаємодія між сервісами, мають більшу затримку порівняно з локальними. У результаті стає важливо правильно налаштувати таймаути, політики повторів і обробку помилок на рівні інфраструктури.

Тому мікросервісна архітектура виправдана у випадках, коли система потребує високої масштабованості, гнучкого розширення, частих оновлень та незалежної роботи команд. Вона забезпечує технічну та організаційну автономію, що особливо важливо для великих продуктів із динамічним розвитком.

Однак перехід до мікросервісів потребує зрілості як у технічному стеку, такі як контейнеризація, оркестрація, моніторинг, так і в організації процесів. Успішне впровадження такої архітектури можливе лише тоді, коли її складність виправдана бізнес-вимогами [5 - 6].

2.1.5 Хмарно-орієнтована архітектура (Cloud-native architecture)

Хмарно-орієнтована архітектура — це підхід до побудови програмного забезпечення й інфраструктури з урахуванням можливостей автоматичного масштабування, швидкого розгортання та відмовостійкості в середовищі хмарного провайдера. Логіка застосунку, процеси CI/CD та засоби моніторингу об’єднуються в цілісну систему практик, що дозволяє максимально використовувати переваги хмарних платформ.

Серед основних характеристик cloud-native архітектури вирізняється використання контейнерів і систем оркестрації. Код ізолюється в Docker-контейнери, управління виконується через Kubernetes [7], що забезпечує функції самовідновлення та безперервного оновлення (rolling update). Підхід Infrastructure-as-Code реалізується за допомогою інструментів на кшталт

Terraform або CloudFormation — інфраструктура описується як код, зберігається в репозиторії та може бути відтворена в будь-який момент.

Системи створюються на основі мікросервісів або serverless-функцій, які масштабуються незалежно й можуть оновлюватися без зупинки роботи. За спостережуваність відповідають такі інструменти, як Prometheus, Grafana та засоби трасування, які збирають метрики та журнали, що дає змогу швидко локалізувати проблеми. Операційні завдання делегуються керованим сервісам — наприклад, базам даних, брокерам повідомлень і файловим сховищам, що дозволяє знизити навантаження на команду розробки. Залежно від потреб використовуються моделі SaaS, PaaS або IaaS, що дає змогу гнучко балансувати між контролем і швидкістю запуску.

Серед основних переваг хмарно-орієнтованої архітектури — горизонтальне масштабування без зупинки сервісів, відмовостійкість через реплікацію та автоматичний перезапуск, а також чітке розмежування зон відповідальності між командами. Водночас підхід має і певні обмеження: високий поріг входу через потребу володіння знаннями з Docker, Kubernetes, IaC, CI/CD та складність архітектурної топології. Також існує ризик прив’язки до конкретного хмарного провайдера (vendor lock-in).

У результаті cloud-native архітектура дає змогу будувати гнучкі, масштабовані системи з мінімальним простоєм, але вимагає зрілого DevOps-процесу та продуманого підходу до вибору хмарних сервісів [8].

2.2 Принципи проектування

На відміну від традиційних монолітних архітектур, сучасні мікросервісні та cloud-native системи потребують глибшої уваги до проектувальних рішень. Для підтримки цілісності, масштабованості й стабільної еволюції, використовуються структуровані принципи проектування. У цьому підрозділі зосередимося на ключових архітектурних підходах (Domain-Driven Design, 12-

Factor App, Clean Architecture), а також розглянемо базові принципи, що є спільними для більшості виробничих систем.

2.2.1 Domain-Driven Design

Domain-Driven Design (DDD) — предметно-орієнтоване проектування, сформульоване Еріком Евансом. Основна ідея полягає в тому, що програмна система має будуватися навколо предметної області, з активною участю доменних експертів. Модель предметної області стає центром архітектури, а її терміни і правила — основою для структури коду.

DDD є особливо корисним у мікросервісній архітектурі, оскільки дозволяє чітко виділити окремі функціональні, або предметні області (bounded contexts), у межах яких терміни мають однозначне значення. Це допомагає уникнути змішування понять та зменшує залежності між сервісами.

Ключові поняття DDD:

- Bounded Context — обмежений контекст, у межах якого модель має єдине тлумачення.
- Entities — об’єкти з унікальною ідентичністю, які змінюються в часі (наприклад, User).
- Value Objects — об’єкти, що характеризуються лише набором атрибутів і не мають ідентичності (наприклад, Грошова сума або Адреса).
- Aggregates — групи сутностей і об'єктів-значень, що мають один вхідний інтерфейс “Aggregate Root” і гарантують цілісність бізнес-інваріантів.
- Repositories — інтерфейси для доступу до агрегатів (читання/запис).
- Domain Services — операції, що не належать жодній конкретній сутності, але важливі з точки зору домену.

DDD дозволяє побудувати чисту доменну модель, із чітким розділенням бізнес-логіки та інфраструктурного коду. Це сприяє модульності, підвищує масштабованість системи і полегшує тестування. Завдяки DDD, кожен сервіс у системі може відповідати за реалізацію конкретної бізнес-функції, зберігаючи незалежність від інших сервісів [9].

2.2.2 12 Factor App

Методологія Twelve-Factor App — це набір із 12 практик розробки сучасних програмних застосунків як сервісів (SaaS), сформульований командою платформи Heroku на чолі з Адамом Віггінсом. Основною метою є створення масштабованих, портативних і надійних веб-застосунків, які легко розгортаються у хмарному середовищі.

Хоча спочатку методологія орієнтувалася на SaaS-додатки, вона ідеально підходить для мікросервісної архітектури, де кожен мікросервіс є окремим самостійним процесом.

Принципи Twelve-Factor охоплюють різні аспекти розробки і розгортання, включаючи управління залежностями, конфігураціями, логами та масштабуванням. Нижче наведено стислий опис кожного з дванадцяти факторів:

1. Кодовий базис – одна кодова база під контролем версій, яка може бути розгорнута у декількох середовищах, наприклад: dev/stage/prod.
2. Залежності – усі залежності мають бути явно оголошені та ізольовані, не використовуються глобальні пакети.
3. Конфігурація – зберігається поза кодом, наприклад у змінних середовища (environment variables).
4. Служби як ресурси – бази даних, черги повідомлень та інші сервіси мають оброблятися як зовнішні ресурси.

5. Build, release, run – чітке розділення етапів збірки, релізу та виконання застосунку.
6. Процеси без збереження стану – стан (наприклад, сесії) зберігається у зовнішніх сховищах, що спрощує масштабування.
7. Експортування сервісу через порт – застосунок сам слухає HTTP-порт, не покладаючись на веб-сервер хоста.
8. Конкурентність – масштабується через запуск декількох процесів, екземплярів.
9. Disposability – швидкий старт і завершення процесів забезпечують гнучкість при оновленнях та аваріях.
10. Однаковість середовищ – максимальна подібність між середовищами розробки та продакшну.
11. Логування як події – усі логи виводяться у stdout/stderr і збираються зовнішніми сервісами, наприклад: Elasticsearch, Grafana.
12. Адміністративні процеси – міграції баз даних або одноразові команди виконуються як окремі процеси того ж середовища [10].

2.2.3 Clean Architecture

Clean Architecture — це популярний підхід до проектування програмних систем, запропонований Робертом Мартіном. Його головна ідея полягає в чіткому розмежуванні відповідальності між компонентами системи з метою підвищення гнучкості, тестованості та підтримки коду.

Основні принципи Clean Architecture передбачають організацію коду у вигляді декількох шарів, які утворюють ієрархію залежностей:

- Entities — внутрішній шар, що містить ключові бізнес-об'єкти та їх базову поведінку. Сутності визначають основні правила бізнесу, незалежно від зовнішніх сервісів чи технологій.
- Use Cases — описують конкретні сценарії бізнес-логіки. Вони взаємодіють із сутностями, координуючи виконання бізнес-процесів. В цьому шарі не повинно бути деталей реалізації інтерфейсів, баз даних або фреймворків.
- Interface Adapters — відповідають за конвертацію даних між внутрішніми шарами та зовнішніми компонентами системи. До цього шару належать такі елементи, як репозиторії для роботи з базами даних, контролери API, презентери та інші адаптери.
- Frameworks & Drivers — зовнішній шар, що взаємодіє із зовнішніми системами та службами, такими як веб-фреймворки, системи зберігання даних, UI та інші зовнішні інструменти.

Ключовим у Clean Architecture є правило залежностей (Dependency Rule), згідно з яким залежності завжди повинні спрямовуватися всередину, до більш абстрактних шарів, а не навпаки. Це означає, що бізнес-логіка, зокрема сутності та сценарії використання, має залишатися максимально незалежною від конкретних реалізацій баз даних, веб-фреймворків або інших зовнішніх сервісів.

Clean Architecture підтримує та розширює принципи SOLID, які є основоположними у проектуванні гнучких, масштабованих і підтримуваних програмних систем.

Принципи SOLID

Це абревіатура з п’яти ключових принципів об’єктно-орієнтованого програмування:

- S — Single Responsibility Principle: кожен клас повинен мати лише одну причину для змін;

- O — Open/Closed Principle: сутності мають бути відкриті для розширення, але закриті для модифікації;
- L — Liskov Substitution Principle: підкласи повинні замінювати батьківські класи без порушення логіки;
- I — Interface Segregation Principle: краще багато вузьких інтерфейсів, ніж один загальний;
- D — Dependency Inversion Principle: залежності повинні будуватись від абстракцій, а не реалізацій.

Отже, Clean Architecture забезпечує чітке розділення відповідальностей, сприяє високій гнучкості та дозволяє будувати системи, що легко масштабуються і підтримуються. Завдяки опорі на принципи SOLID, цей підхід добре підходить для сучасних розподілених застосунків і особливо актуальний у мікросервісному середовищі.

Більше того, Clean Architecture природно поєднується з іншими загальновизнаними принципами розробки — такими як розділення відповідальностей, низьке зв’язування, висока згуртованість, унікнення повторень тощо. У наступних підрозділах розглянемо ці принципи детальніше та проаналізуємо їхнє значення для побудови надійної мікросервісної архітектури.

2.2.4 Separation of Concerns та High Cohesion / Low Coupling

Одним з базових підходів до структурування програмного коду є принцип Separation of Concerns (SoC), згідно з яким кожен компонент або шар системи повинен відповідати лише за одну категорію задач. Наприклад: інтерфейс користувача — за візуалізацію, доменна логіка — за обробку правил бізнесу, а сховище даних — за їх зберігання.

Цьому підходу відповідає також принцип високої згуртованості (High Cohesion), згідно з яким усі класи всередині одного модуля або сервісу мають бути об’єднані спільною метою. Це покращує зрозумілість коду та спрощує його тестування.

У свою чергу, слабке зв’язування (Low Coupling) означає, що різні сервіси або модулі повинні взаємодіяти між собою лише через чітко визначені інтерфейси (наприклад, REST API або події через Kafka), без залежності від внутрішньої реалізації один одного. Це дозволяє змінювати або переміщати сервіси (наприклад, між Azure AKS та AWS EKS) без зміни логіки взаємодії.

2.2.5 DRY, KISS та YAGNI

Принцип DRY (Don't Repeat Yourself) заохочує уникати дублювання коду. Будь-яке бізнес-правило повинно мати лише одне джерело істини. Це значно полегшує супровід системи та зменшує ризик помилок при внесенні змін.

KISS (Keep It Simple, Stupid) наголошує на важливості простоти: рішення повинні бути настільки простими, наскільки це можливо. Надмірна складність ускладнює деплоймент, налагодження та розвиток системи.

YAGNI (You Aren’t Gonna Need It) застерігає від реалізації функціоналу «про запас», якого ще не потребує користувач або бізнес. Краще сфокусуватися на тому, що вже має підтверджене використання.

Наведені принципи проектування становлять основу для побудови надійних, гнучких та масштабованих мікросервісних систем і активно використовуються в індустрії розробки програмного забезпечення.

2.3 Патерни взаємодії сервісів

У мікросервісній архітектурі застосунок розділяється на багато незалежних сервісів, кожен з яких виконує окрему бізнес-функцію. Ці сервіси спілкуються між собою через мережу, що докорінно відрізняється від викликів функцій у монолітному застосунку. Така взаємодія накладає додаткові вимоги до надійності, узгодженості та відмовостійкості. Для ефективного проектування взаємодії сервісів використовують два основні підходи:

- Синхронна взаємодія — коли один сервіс очікує негайну відповідь від іншого. Найчастіше реалізується через REST або gRPC. Це зручно для запитів, що потребують швидкої реакції.
- Асинхронна взаємодія — коли сервіс надсилає повідомлення або подію, не чекаючи відповіді. Використовуються брокери повідомлень, такі як Kafka або RabbitMQ. Це дозволяє масштабувати обробку подій, зменшувати залежність між сервісами та покращувати відмовостійкість.

Обидва підходи мають свої сценарії застосування, і найкращі практики мікросервісного дизайну рекомендують комбінувати їх. Для цього в архітектурі використовуються типові патерни — шаблони проектування, які допомагають упорядкувати й уніфікувати міжсервісну взаємодію.

Нижче розглянуто більш детально найпоширеніші патерни та технології взаємодії між сервісами, які забезпечують масштабованість, спрощують інтеграцію та знижують ризики при відмовах окремих компонентів. Серед них — синхронні API-запити за допомогою REST і gRPC, обробка подій через Kafka, централізоване керування точками входу через API Gateway, а також гарантована доставка повідомлень завдяки патерну Outbox.

2.3.1 REST

REST (Representational State Transfer) — це архітектурний стиль для побудови розподілених систем, який функціонує поверх протоколу HTTP. Його концепцію запропонував Рой Філдінг у своїй докторській дисертації у 2000 році. REST широко використовується у веб-розробці, зокрема для синхронної взаємодії між сервісами, коли один сервіс викликає інший, надсилаючи запит і очікуючи негайну відповідь.

У REST-архітектурі клієнт звертається до ресурсів, представлених у вигляді уніфікованих URI-ідентифікаторів, використовуючи стандартні методи HTTP: GET, POST, PUT / PATCH, DELETE.

Сервер, у відповідь, повертає представлення ресурсу у форматі JSON або XML, хоча у сучасних системах переважає JSON через його компактність і зручність.

REST є безстановим (stateless): кожен запит не залежить від попередніх і містить всю необхідну інформацію. Такий підхід полегшує масштабування та спрощує обробку запитів [11].

2.3.2 gRPC

gRPC (Google Remote Procedure Call) — це фреймворк від Google для реалізації високопродуктивної взаємодії між сервісами у розподілених системах. На відміну від REST, який базується на ресурсах і HTTP-методах, gRPC реалізує концепцію віддаленого виклику процедур (RPC): клієнт викликає метод, який фізично виконується на іншому сервері, як звичайну локальну функцію.

gRPC використовує:

- Protocol Buffers (protobuf) — мову опису інтерфейсів і формат серіалізації даних, який забезпечує компактність і швидкість передачі;

- HTTP/2 як транспортний протокол — що дозволяє мультиплексування запитів, стиснення заголовків і підтримку потоків.

gRPC реалізує модель синхронної взаємодії між клієнтом і сервером, але завдяки використанню HTTP/2 також підтримує асинхронні потоки даних, включаючи одно- та двосторонній стрімінг повідомлень. Це робить gRPC універсальним для широкого спектра сценаріїв — від простих CRUD-операцій до real-time обміну [12].

2.3.3 Kafka

Kafka — це розподілений брокер повідомлень, який широко використовується для реалізації асинхронної взаємодії між сервісами у мікросервісній архітектурі.

Зокрема, Kafka підтримує патерн "publish/subscribe", за якого один сервіс публікує (publish) події, а інші — підписані (subscribe) на їх отримання. Така модель дозволяє досягти низького зв’язування між компонентами, забезпечити високу надійність доставки та гнучке масштабування [12].

Розгорнутий опис принципів роботи Kafka, а також її ролі в архітектурі подієвої інтеграції наведено у підрозділі 2.5.

2.3.4 API Gateway

API Gateway – це шаблон, при якому вводиться спеціальний фасадний сервіс-шлюз, що є єдиною точкою входу для клієнтів в систему мікросервісів. Замість того, щоб зовнішній клієнт, наприклад веб-браузер користувача, звертався окремо до різних сервісів, всі запити надходять на API Gateway, а вже він маршрутизує їх до відповідних внутрішніх сервісів.

API Gateway виступає своєрідним «єдиним вікном» для системи: клієнт звертається до нього по визначеним URI, а Gateway визначає, куди

перенаправити запит, можливо агрегує дані з кількох сервісів, повертаючи єдиний консолідований відгук. Зазвичай це окремий сервіс, що функціонує на рівні API (часто використовують Nginx або спеціалізовані рішення типу Kong, Tyk, AWS API Gateway).

Переваги API Gateway:

Спрощення клієнта: клієнт не потребує знати про всі внутрішні сервіси і їхні адреси/інтерфейси. Він завжди працює лише з одним endpoint (Gateway), що знижує складність на фронтенді.

- Можливість зробити композицію запитів: Gateway може отримати запит, який вимагає дані з кількох сервісів, паралельно звернутися до них і об’єднати результати. Це знижує кількість «раундів» звернень від клієнта і оптимізує трафік.
- Єдине місце для реалізації загальних послуг: аутентифікація, авторизація, ліміт запитів, кешування, трансформація даних – усе це можна зосередити в API Gateway, замість дублювання в кожному мікросервісі. Зокрема, Gateway може перевіряти JWT токен у заголовку і відразу відхиляти неавторизовані запити, не пропускаючи їх до внутрішньої мережі сервісів. Також можна централізовано логувати звернення, здійснювати моніторинг трафіку, тощо.
- Сховування внутрішньої структури: мікросервіси можуть змінюватися без впливу на зовнішніх клієнтів, якщо Gateway продовжує підтримувати стабільний зовнішній API. Це додає гнучкості еволюції системи [14 - 15].

2.3.5 Transactional Outbox

Transactional Outbox — це архітектурний патерн, призначений для надійного обміну подіями між сервісами у мікросервісній архітектурі за допомогою асинхронних механізмів, таких як Kafka. Його основна мета — забезпечити узгодженість даних та уникнути так званої проблеми подвійного

запису, яка виникає при спробі одночасно змінити локальні дані в базі та відправити повідомлення про цю зміну до брокера подій.

Приклад потенційної проблеми з узгодженістю:

Сервіс виконує транзакцію в локальній базі даних і хоче повідомити інші сервіси через Kafka. Якщо спочатку зафіксувати зміну в базі, а потім надіслати повідомлення — існує ризик аварійного завершення роботи системи після фіксації змін, унаслідок чого повідомлення не буде доставлене

Інші сервіси залишаться не поінформованими про зміни, що порушить консистентність. Якщо ж навпаки — спочатку надіслати повідомлення, а потім завершити транзакцію — може статися зворотна проблема: повідомлення буде опубліковане, але зміна в базі не збережеться.

Патерн Outbox вирішує це шляхом використання транзакційної природи бази даних. У межах однієї транзакції сервіс:

- Виконує основну зміну бізнес-даних;
- Додає до спеціальної таблиці outbox запис про подію, яка має бути надіслана.

Завдяки цьому гарантується атомарність — або обидві дії виконуються, або жодна. Таблиця outbox виступає як буфер для вихідних повідомлень.

Пізніше окремий фоновий процес (ретранслятор) або інструмент типу Change Data Capture (CDC), наприклад Debezium, зчитує ці записи й надсилає події до брокера повідомлень, наприклад до Kafka. Після успішної публікації подія видаляється або позначається як оброблена.

Хоча впровадження цього патерну потребує додаткової інфраструктури (таблиця outbox, ретранслятор подій), він значно підвищує надійність системи. Це особливо важливо у розподілених системах, де транзакції не можуть охопити кілька сервісів.

Для побудови надійної мікросервісної системи важливо правильно обрати способи взаємодії між сервісами. Синхронні патерни, такі як REST і gRPC, добре підходять для швидких запитів і відповіді в режимі реального часу. У свою чергу, асинхронні механізми Kafka та Outbox допомагають зменшити зв’язність між сервісами, підвищити їхню незалежність і забезпечити обробку подій навіть у разі збоїв. API Gateway дозволяє централізовано керувати маршрутизацією, безпекою та агрегацією відповідей. Разом ці шаблони створюють архітектурний каркас, який робить систему стійкою до змін і зручною для масштабування [16].

2.4 Надійність та спостережуваність

Розподілена природа мікросервісної архітектури висуває підвищені вимоги до надійності системи та її спостережуваності (observability). Надійність передбачає здатність системи продовжувати коректно функціонувати навіть у разі часткових збоїв, помилок мережі чи недоступності окремих сервісів — це реалізація принципу fault tolerance. Спостережуваність, у свою чергу, забезпечує глибинне розуміння стану системи на основі зовнішніх сигналів, таких як логи, метрики й трасування, що є критично важливим у розподілених середовищах.

У мікросервісних системах навіть незначна помилка одного компонента може спричинити каскадні збої, тому кожен сервіс повинен бути обладнаний механізмами обробки винятків, повторних спроб, тайм-аутів та резервування. Водночас логування та моніторинг відіграють ключову роль у спостережуваності: вони дозволяють не лише виявляти проблеми постфактум, а й завчасно помічати аномалії в роботі системи. Наприклад, аналізуючи час відповіді, рівень навантаження або зростання кількості помилок, можна прогнозувати потенційні збої й оперативно реагувати на них до того, як вони вплинуть на користувача.

У цьому розділі розглянуто підходи до забезпечення fault tolerance, практики ефективного логування та побудови моніторингової інфраструктури з

використанням сучасних інструментів, таких як Prometheus, Grafana та системи alerting.

2.4.1 Fault tolerance

Кожен мікросервіс проектують із припущенням, що суміжні сервіси чи інфраструктурні компоненти можуть тимчасово бути недоступними або працювати некоректно. Щоб уникнути ефекту доміно, застосовують такі патерни відмовостійкості:

1. **Circuit Breaker**

“Вимикач ланцюга” (Closed → Open → Half-open). Це програмний компонент, що огортає виклики до зовнішнього сервісу і відстежує їх успішність. Якщо зовнішній сервіс починає повільно відповідати або зникає зв’язок, наприклад: таймаути або помилки, Circuit Breaker спершу почне відхиляти частину запитів (напіввідкрите становище, дозволяючи періодично пробні запити), а при стійкій проблемі – повністю “розірве коло” і миттєво відмовлятиме у викликах, не чекаючи таймаутів. Це запобігає накопиченню запитів і дозволяє системі швидше відреагувати на збій замість марного очікування.

2. **Bulkheads**

Це шаблон, який був запозичений з кораблебудування: відсіки з перегородками не дозволяють воді затопити весь корабель при пробоїні. У програмуванні ідея така ж: розділити ресурси між частинами системи, щоб збій в одному компоненті не вплинув на інші.

3. Timeouts

Необхідно встановлювати таймаути на всі зовнішні виклики: HTTP-запити, звернення до бази даних, сторонні API тощо, щоб уникнути ситуацій, коли сервіс зависає в очікуванні відповіді, яка може так і не надійти.

Timeouts дозволяють:

- швидко виявляти проблеми у зовнішніх залежностях;
- звільняти ресурси (потоки, з’єднання) для інших запитів;
- активувати резервні сценарії (fallback) або механізми повтору (retry).

У професійній розробці рекомендовано використовувати різні значення таймаутів для різних типів операцій: коротші для критичних маршрутів і довші — для менш важливих фонових процесів.

4. Retries with Exponential Backoff

Переклад з англійської означає “Повторні спроби з експоненційним збільшенням інтервалу”. Цей шаблон передбачає повторення запиту після невдалої спроби, з кожним разом збільшуючи інтервал очікування між спробами, наприклад: 1 секунда → 2 секунди → 4 секунди тощо. Такий підхід дозволяє:

- уникати перевантаження сервісу, що вже знаходиться в нестабільному стані;
- перечекати короткочасні збої в мережі або інфраструктурі;
- зменшити кількість одночасних повторів, що можуть погіршити ситуацію.

5. Fallback

Це заздалегідь визначена альтернативна стратегія обробки запиту у випадку, коли основна логіка виклику не спрацювала через збій зовнішнього сервісу, таймаут чи іншу помилку. Метою fallback-механізму є забезпечення безперервності обслуговування, навіть якщо дані будуть неповними або менш актуальними.

Типові приклади реалізації fallback:

- повернення кешованих даних, збережених під час попередніх успішних запитів;
- використання заглушкових значень;
- переадресація запиту до резервного сервісу або іншої репліки;
- логування помилки і безпечне завершення запиту, якщо немає змоги надати навіть часткову відповідь.

Fallback є останнім рубежем fault tolerance — він не усуває проблему, але дозволяє системі відреагувати на неї контрольовано та без критичних наслідків для користувача.

2.4.2 Спостережуваність

У мікросервісній архітектурі, де система складається з десятків або сотень автономних сервісів, важливо не лише забезпечити їхню відмовостійкість, а й мати чітке уявлення про внутрішній стан кожного з них у будь-який момент часу. Саме це забезпечує спостережуваність — здатність системи відповідати на запитання “що відбувається всередині” на основі зовнішніх сигналів.

На відміну від традиційного моніторингу, який фокусується лише на перевірці “живий сервіс чи ні”, сучасна спостережуваність дозволяє глибше

аналізувати причини збоїв, виявляти приховані проблеми, оцінювати продуктивність та виявляти аномалії ще до того, як вони вплинуть на кінцевого користувача.

Щоб досягти високого рівня спостережуваності, система має бути розроблена з урахуванням телеметрії на всіх рівнях: від окремого запиту до комплексного бізнес-процесу. Цей підхід критично важливий для команд, які підтримують продакшн-середовище у режимі 24/7, де швидке виявлення і локалізація проблеми є запорукою стабільної роботи.

Спостережуваність охоплює три основні категорії: **логування, метрики, трасування.**

2.4.3 Логування

Логування — процес фіксації подій, помилок, запитів і технічної інформації про функціонування системи. У мікросервісній архітектурі воно відіграє ключову роль, забезпечуючи спостережуваність за кожним окремим сервісом і формуючи цілісне уявлення про загальний стан застосунку.

На відміну від монолітних рішень, де всі логи зберігаються централізовано, у мікросервісному середовищі кожен компонент формує власні журнали подій. Це ускладнює обробку, внаслідок чого впроваджується централізоване логування — об’єднання записів у спільному сховищі з можливістю фільтрації, пошуку та встановлення зв’язків між подіями.

Для підвищення ефективності логування застосовується структурований підхід, що дозволяє автоматизувати обробку, аналіз і візуалізацію даних. Часто використовується формат JSON — він полегшує фільтрацію, пошук за ключовими параметрами та побудову дашбордів для моніторингу.

Лог-повідомлення класифікуються за рівнями важливості: DEBUG, INFO, WARNING, ERROR або CRITICAL. Така ієрархія полегшує обробку великого

обсягу даних і дає змогу оперативно виявляти критичні ситуації чи аномальну поведінку.

Контекст у логах має визначальне значення. У записах фіксуються ідентифікатор користувача, пов’язаний запит, назва сервісу, код відповіді, тривалість обробки та параметри вхідного запиту — ці дані істотно спрощують аналіз роботи системи.

Особливу увагу приділено питанням безпеки. У логах не повинна міститися конфіденційна інформація — паролі, токени, фінансові реквізити або персональні дані. У результаті зміст логів формується з урахуванням вимог інформаційної безпеки та конфіденційності.

2.4.4 Моніторинг метрик

У мікросервісній архітектурі спостережуваність вважається одним із ключових чинників для забезпечення стабільної роботи системи, а метрики є її центральною складовою.

Метрики — це числові показники, які дозволяють відстежувати стан компонентів у реальному часі. Завдяки їм виявляються технічні проблеми, прогнозуються збої та оцінюється продуктивність окремих сервісів.

У межах мікросервісної архітектури налаштовується збір базових технічних метрик, серед яких: час відповіді, кількість запитів, частота помилок, завантаження процесора, використання пам’яті та статуси HTTP-відповідей. Поряд із ними враховуються бізнес-метрики, що демонструють реальні сценарії використання функціональності. Це дозволяє контролювати не лише технічну справність, а й ефективність взаємодії з боку кінцевих користувачів.

Збір, зберігання та аналіз метрик забезпечується за допомогою спеціалізованих інструментів. Стандартом у мікросервісних системах виступає Prometheus — система моніторингу, що застосовує pull-підхід для регулярного збору даних з endpoint’ів /metrics. Кожен мікросервіс публікує власні показники

через клієнтські бібліотеки. Типові приклади — кількість HTTP-запитів, тривалість їх обробки, кількість з’єднань з базою даних або оброблених повідомлень у Kafka.

Для візуалізації та аналізу застосовується Grafana — платформа, що надає інтерактивні дашборди з графіками, таблицями та індикаторами. Це значно спрощує моніторинг і виявлення критичних зон у системі.

Важливою складовою є налаштування порогових значень метрик для автоматичної генерації сповіщень. Система моніторингу має реагувати на відхилення, наприклад, коли частка помилок HTTP-запитів перевищує заданий рівень протягом певного періоду. У результаті з’являється змога вчасно локалізувати проблему ще до того, як вона вплине на користувачів.

Метрики становлять основу моніторингу мікросервісів і сприяють підтриманню стабільної та передбачуваної роботи. Їх використання є необхідною умовою для побудови надійної та масштабованої архітектури.

2.4.5 Розподілене трасування

У мікросервісній архітектурі, де обробка одного запиту може охоплювати десятки або навіть сотні незалежних сервісів, складно отримати повну картину його виконання, спираючись лише на логи чи метрики. В умовах такої складності вирішальне значення має розподілене трасування — інструмент, що дає змогу відстежити повний маршрут запиту в межах усієї системи, виявити затримки, вузькі місця або збої на будь-якому з етапів.

Щоб бачити весь шлях запиту, кожен мікросервіс додає до нього спеціальні ідентифікатори — trace ID і span ID. Вони передаються разом із запитом через HTTP-заголовки або Kafka headers. Це дає змогу визначити, які сервіси були залучені, скільки часу тривала кожна операція та де саме виникла затримка або помилка.

Під час обробки кожен сервіс створює власний запис — span, у якому зберігається інформація про початок і завершення дії, її назву, статус та інші технічні деталі. Усі ці спани об’єднуються в єдину трасу (trace), що показує повну картину проходження запиту. У результаті з'являється можливість виявити проблемну ділянку навіть у випадку успішної відповіді системи.

Розподілене трасування реалізується за допомогою таких інструментів, як OpenTelemetry — відкритого стандарту, що забезпечує збір і передачу даних у системи на зразок Jaeger, Zipkin або Grafana Tempo. OpenTelemetry підтримує клієнтські бібліотеки для різних мов програмування, зокрема Python, Java, Go. Це відкриває можливість інтегрувати метрики, логи та трасування в єдину систему спостережуваності. Наприклад, у Django використовується opentelemetry-django, а для Flask чи FastAPI — відповідні middleware, які автоматично створюють спани для HTTP-запитів, роботи з базами даних і зовнішніми сервісами.

У масштабних системах, де зростає кількість сервісів і залежностей, значення розподіленого трасування посилюється. Особливо воно корисне у випадках, коли причина погіршення продуктивності не є очевидною — наприклад, спостерігається збільшення часу відповіді з 200 мс до 2 секунд без видимих помилок у логах.

2.5 Роль брокера повідомлень Kafka у подієвій інтеграції сервісів

У системах з мікросервісною архітектурою взаємодія сервісів реалізується як синхронно через HTTP-виклики, так і асинхронно — за допомогою подій або повідомлень. Подієво-орієнтована інтеграція забезпечує реакцію сервісів на події без жорсткої прив’язки у часі. Центральне місце в такій моделі посідає брокер повідомлень — проміжна ланка, що приймає події від відправників і передає їх отримувачам. У межах дипломної роботи як брокер повідомлень використано

Apache Kafka — розподілену платформу обробки потоків подій, що активно застосовується в мікросервісних системах.

Kafka працює за принципом pub/sub (публікація-підписка): події зберігаються у спеціальних категоріях — топіках. Продюсери публікують події у відповідний топік, а споживачі підписуються на нього й отримують події у порядку надходження. Повідомлення зберігаються на диску, а завдяки можливості горизонтального масштабування та підтримці реплікації забезпечується висока пропускна здатність і відмовостійкість. Кожен топік може бути розбитий на партиції, що дозволяє обробляти події паралельно на кількох брокерах. У межах однієї партиції повідомлення зберігають чітку послідовність [13, 18].

Серед ключових переваг подієвої інтеграції та Kafka:

- **Асинхронна комунікація.** Сервіси не зобов’язані працювати одночасно. Подія зберігається у Kafka до моменту її обробки споживачем. Це знижує ризик блокувань і підвищує надійність, навіть у разі тимчасової недоступності окремих компонентів.
- **Масштабованість.** Kafka здатна обробляти тисячі подій за секунду в реальному часі, розподіляючи навантаження між кількома брокерами. Споживання масштабується за допомогою consumer group — декількох екземплярів, що паралельно читають один і той самий топік.
- **Надійність доставки.** Kafka гарантує доставку принаймні одного разу (at-least-once), а за потреби — точно один раз (exactly-once). Наприклад, подія про призначення ментора зберігається до моменту підтвердження її обробки сервісом сповіщень. У разі збою повідомлення не втрачаються та дочитуються після перезапуску.

Розділ 3. ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ MentorMatch

3.1 Архітектура системи

3.1.1 Загальна логіка побудови системи

MentorMatch реалізовано як систему з мікросервісною архітектурою, де кожна функціональна частина винесена в окремий сервіс. Кожен сервіс виконує чітко визначене завдання: автентифікація, управління профілями, обробка сесій, повідомлення, рейтингування тощо. Компоненти розгортаються у вигляді окремих контейнерів і комунікують між собою через REST-запити або брокер повідомлень Kafka. Така побудова дозволяє масштабувати окремі частини системи незалежно, зменшує зв'язність між компонентами й підвищує надійність у разі збоїв.

3.1.2 Обґрунтування вибору мікросервісної архітектури

Мікросервісний підхід було обрано через потребу у високій гнучкості, модульності й масштабованості. На відміну від моноліту, у якому зміна одного компонента потребує перескладання всієї системи, мікросервіси дозволяють оновлювати окремі частини без зупинки інших. Крім того, розділення на сервіси полегшує командну розробку: над кожним мікросервісом може працювати незалежна команда. У результаті це пришвидшує впровадження змін і дає змогу адаптуватися до змін бізнес-логіки без суттєвого ризику для стабільності системи.

3.1.3 Компоненти системи

До складу системи входять такі основні сервіси:

Сервіс	**Призначення**
Auth-service	Автентифікація та видача токенів

Profile-service	Зберігання й редагування профілів користувачів
Session-rating-service	Створення сесій та їх оцінювання
Notification-service	Формування та надсилання сповіщень
Frontend (Web Client)	Користувацький інтерфейс

Для маршрутизації запитів використовується API Gateway, який приймає зовнішні HTTP-запити та передає їх відповідним сервісам. Зберігання даних реалізовано через окремі керовані хмарні бази даних для кожного сервісу. Для асинхронної взаємодії використовується Apache Kafka.

3.1.4 Взаємодія між сервісами

Комунікація між мікросервісами реалізована у двох формах. Основні операції виконуються через REST API, де кожен запит спрямовується до конкретного сервісу. Для подій, які не потребують негайної відповіді або можуть бути оброблені асинхронно, наприклад надсилання повідомлення після видалення користувача, застосовується Kafka. Події публікуються у відповідні топіки, а інші сервіси-споживачі обробляють їх незалежно (Додаток Б). У результаті забезпечується більша стійкість до збоїв, оскільки сервіси не блокують один одного.

3.1.5 Загальна схема архітектури

На рис. 3.1 зображено загальну архітектуру системи MentorMatch. Веб-клієнт надсилає запити до API Gateway, який маршрутизує їх до відповідних мікросервісів. Кожен сервіс розгорнуто в окремому контейнері та підключено до власної хмарної бази даних (Cloud DB). Залежно від середовища розгортання використовуються Azure Database for PostgreSQL Flexible Server або Amazon RDS for PostgreSQL. Для асинхронної взаємодії між сервісами застосовується Kafka,

розгорнута як окремий контейнер. Така структура забезпечує ізольованість, масштабованість і стійкість до збоїв.

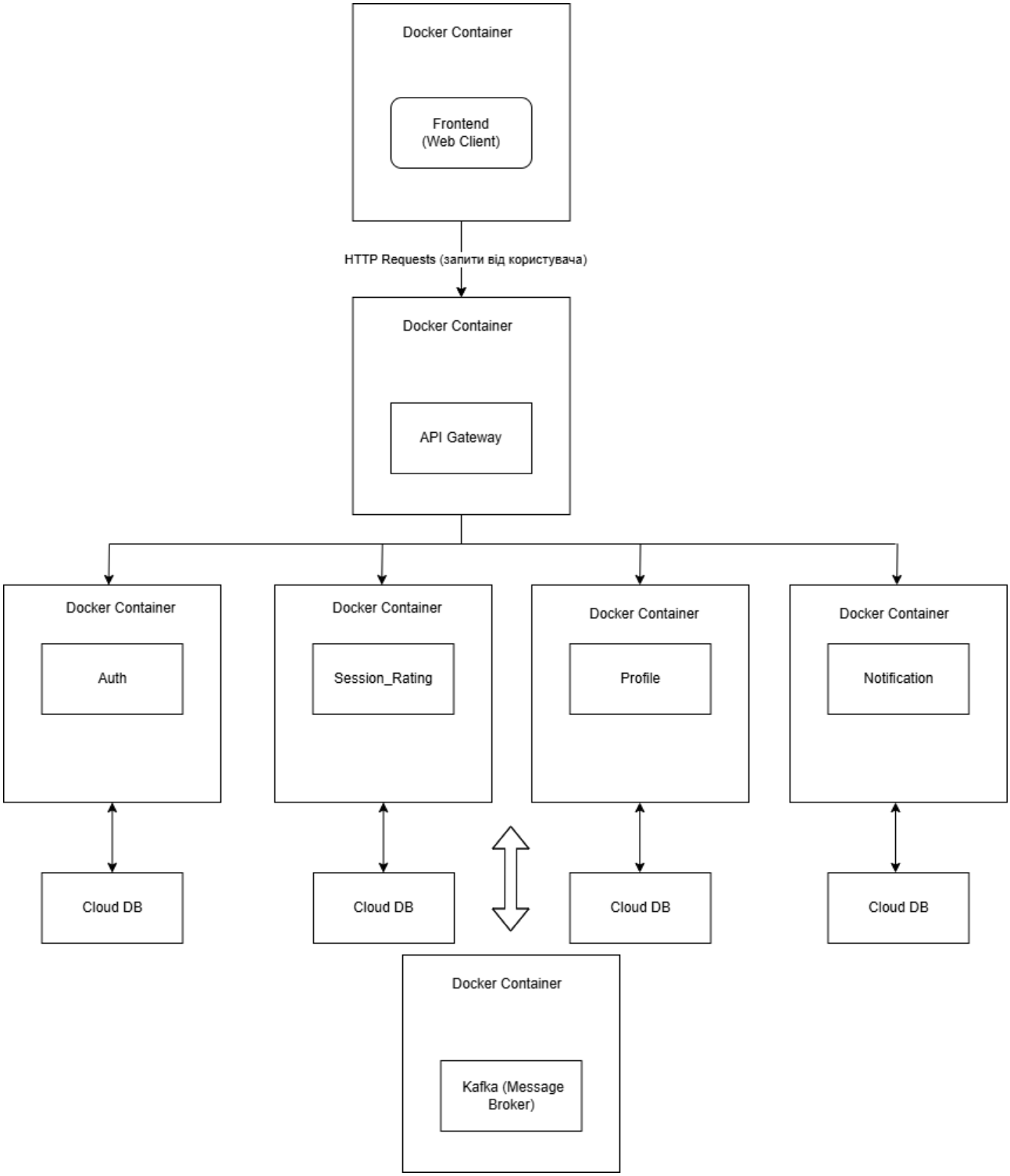


Рисунок 3.1 – Архітектура системи MentorMatch

Крім загальної архітектури компонентів системи, ключову роль відіграє структура даних, що зберігаються та обробляються всередині кожного сервісу. На рис. 3.2 ілюстровано ER-модель, логічну організацію сутностей, які використовуються в платформі MentorMatch. Незважаючи на поділ на окремі мікросервіси з індивідуальними базами даних, створення єдиної узгодженої моделі було необхідним для забезпечення цілісності й несуперечливості бізнес-логіки.

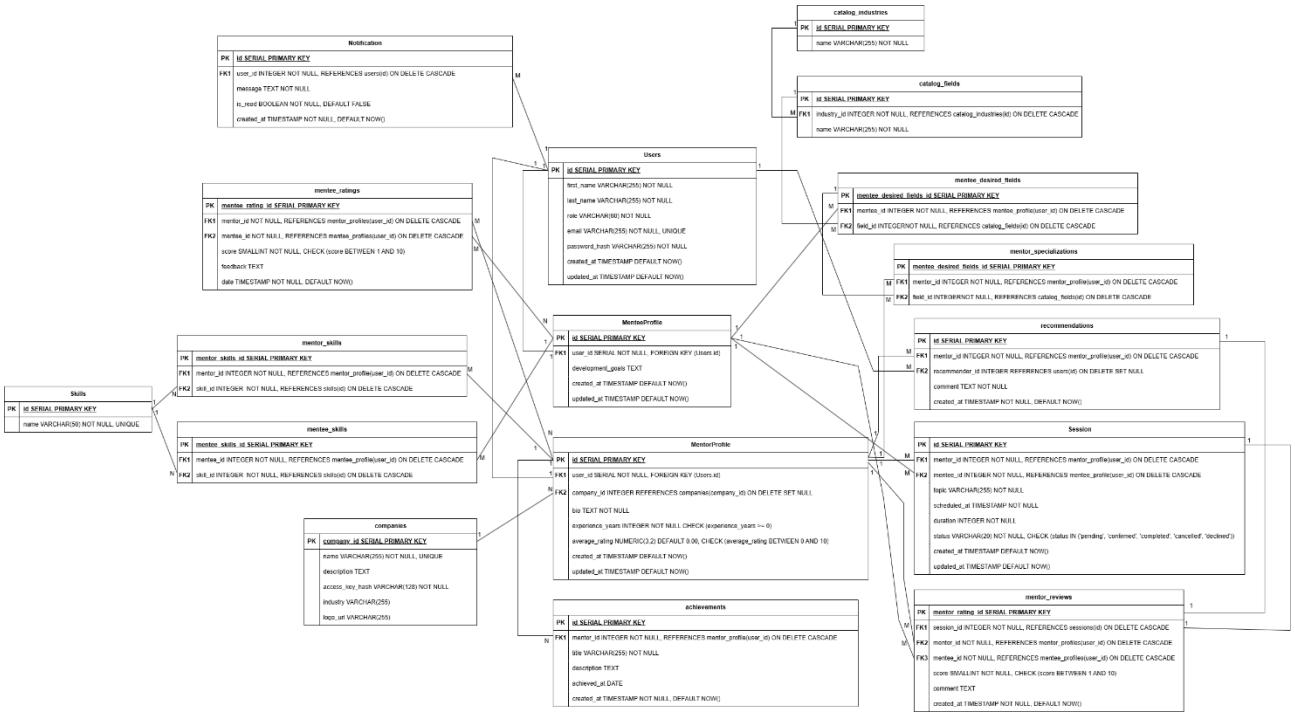


Рисунок 3.2 - ER-модель бази даних системи MentorMatch

3.2 Технологічний стек

3.2.1 Обрані інструменти

Для реалізації системи MentorMatch було використано сучасний і широко підтримуваний технологічний стек. Основна логіка сервісів реалізована мовою програмування Python з використанням фреймворку Django та бібліотеки Django REST Framework, що дозволяє швидко створювати REST API та забезпечує підтримку масштабованої структури. Контейнеризація виконана за допомогою Docker, що спрощує розгортання кожного сервісу як окремого ізольованого

середовища. Для оркестрації та управління конфігурацією використовуються YAML-файли та Helm-шаблони. Аутентифікація реалізована на основі JWT-токенів, що забезпечує безпечну авторизацію між сервісами. Моніторинг реалізовано за допомогою Prometheus для збору метрик та Grafana для їх візуалізації.

3.2.2 Керовані бази даних

Замість локального або контейнерного розгортання баз даних було прийнято рішення використовувати керовані хмарні СУБД. У середовищі Microsoft Azure застосовується Azure Database for PostgreSQL Flexible Server, а в середовищі Amazon Web Services — Amazon RDS for PostgreSQL. Кожен мікросервіс підключається до окремої бази даних, що дозволяє зберігати ізольований контекст даних. Такий підхід знижує навантаження на команду розробки, оскільки інфраструктурні задачі: резервне копіювання, масштабування, автоматичне оновлення виконуються хмарною платформою. Це підвищує надійність і зменшує ризики втрати даних або збоїв під час обробки запитів.

3.2.3 Обробка подій через Kafka

Для реалізації асинхронної взаємодії між мікросервісами застосовується Apache Kafka. Події, що виникають у процесі обробки користувацьких дій (наприклад, створення сесії, оновлення профілю), публікуються у відповідні топіки Kafka. Сервіси-споживачі (Kafka consumers) обробляють ці події незалежно від основного процесу, що дозволяє уникнути блокування та підвищує стабільність системи. Такий підхід зменшує залежність між сервісами та підвищує гнучкість архітектури. Використання брокера повідомлень також полегшує масштабування обробки, оскільки кількість споживачів можна регулювати відповідно до навантаження.

3.3 Cloud-ready реалізація

3.3.1 Архітектура, готова до хмарного розгортання

Архітектура системи MentorMatch з самого початку проектувалася з урахуванням принципів cloud-native. Усі сервіси реалізовано як окремі ізольовані контейнери, які можна масштабувати незалежно один від одного. Комунікація між компонентами здійснюється через мережу, що унеможливлює прив'язку до фізичної інфраструктури. У системі передбачено централізоване управління конфігурацією, використання секретів, підтримку асинхронного обміну подіями та зовнішніх керованих СУБД. У результаті вся структура повністю сумісна з хмарною інфраструктурою та легко переноситься між платформами Microsoft Azure і Amazon Web Services без зміни бізнес-логіки.

3.3.2 Інфраструктура як код

Для опису конфігурації системи використовується підхід IaaS. Усі сервіси конфігуруються за допомогою YAML-файлів та Helm-шаблонів, які зберігаються в репозиторії поряд з кодом застосунку. Це дозволяє створити, оновити або відновити всю інфраструктуру в автоматизований спосіб. Описуються не лише мікросервіси, а й об'єкти Kubernetes: ingress, volume claims, health checks, конфігурації Kafka. Завдяки використанню Helm створюється шаблонізована система, де можна швидко змінювати параметри розгортання без дублювання коду. Це суттєво спрощує CI/CD-процеси та пришвидшує підготовку середовищ.

3.3.3 Спрощення підтримки завдяки керованим сервісам

Система активно використовує керовані сервіси обох хмарних платформ. Для зберігання даних застосовуються Azure Database for PostgreSQL Flexible Server або Amazon RDS for PostgreSQL, залежно від середовища розгортання. Розгортання та масштабування Kafka виконується через Helm, що спрощує

конфігурацію. Усі бази даних підключаються як зовнішні сервіси, а не запускаються в межах кластера, що дозволяє досягти більшої стабільності, знизити навантаження на DevOps-команду та уникнути втрати даних у разі аварій. У результаті адміністрування значно спрощується, а підтримка критичних компонентів перекладається на хмарного провайдера.

3.4 Спостережуваність та надійність

3.4.1 Моніторинг системи

У системі MentorMatch реалізовано повноцінну спостережуваність за допомогою комбінації інструментів Prometheus та Grafana. Prometheus відповідає за збір і зберігання метрик з кожного мікросервісу. До основних показників належать: кількість HTTP-запитів, статуси відповідей, час обробки, навантаження на базу даних і тривалість затримок між Kafka-подіями. Усі сервіси інтегровані з Prometheus через відповідні експортери. Зібрані дані виводяться на дашборди в Grafana, що дає змогу швидко оцінити поточний стан системи, виявити пікові навантаження й локалізувати проблеми продуктивності. Такий підхід дозволяє вчасно реагувати на відхилення в роботі окремих компонентів без потреби ручного моніторингу логів.

3.4.2 Відстеження доступності та відновлення після збоїв

У системі реалізовано базовий рівень стійкості до збоїв завдяки використанню Kafka як асинхронного брокера подій. Усі ключові дії, що запускають вторинну логіку, публікуються у топіки Kafka, а сервіси-споживачі обробляють події у власному ритмі. У разі збою або недоступності окремого споживача повідомлення не втрачається, а зберігається в Kafka до моменту повторного зчитування. Це дозволяє уникнути ситуацій, коли критичні дії губляться через тимчасову недоступність одного з сервісів.

Повторне споживання подій реалізовано через логіку обробки Kafka consumer'ів. У разі виникнення помилки під час обробки повідомлення застосовується механізм повторної обробки, що фіксується у логах. Завдяки персистентному сховищу Kafka події не видаляються одразу після публікації, а зберігаються у відповідному топіку протягом налаштованого часу, що забезпечує можливість їх повторного використання після відновлення сервісу.

3.4.3 Захист API та безпека

Усі запити до внутрішніх та зовнішніх API проходять перевірку через JWT-аутентифікацію. При кожному зверненні перевіряється дійсність токена, його підпис, строк дії та відповідність ролі. Для обмеження доступу реалізовано рольову модель (RBAC), де кожен тип користувача має власні дозволи. Крім того, всі конфіденційні дані, зокрема змінні середовища для підключення до баз даних та JWT-ключі, зберігаються у керованих сервісах для управління секретами — Azure Key Vault та AWS Secrets Manager. Завдяки інтеграції з Kubernetes через CSI драйвер, ці секрети динамічно монтуються у поди й не зберігаються у вигляді звичайних Kubernetes secrets або у репозиторії коду. Усі сервіси працюють у межах окремих ізольованих контейнерів, що також підвищує рівень безпеки.

Розділ 4. ДЕПЛОЙ МІКРОСЕРВІСІВ У ХМАРАХ: MICROSOFT AZURE vs AWS

4.1 Цілі хмарного розгортання

Хмарне розгортання системи MentorMatch було реалізовано для перевірки, наскільки сучасні хмарні сервіси спрощують процес розгортання, масштабування та супроводу мікросервісного веб-застосунку. Основним завданням стало перенесення системи з локального середовища, побудованого на Docker та Kubernetes, у хмарну інфраструктуру з використанням керованих сервісів. Замість розгортання всіх компонентів вручну, як у локальному кластері, було вирішено використати максимально доступні можливості Azure та AWS для зниження операційних витрат та покращення відмовостійкості.

Окрему увагу було приділено перевірці можливості використання керованих баз даних замість контейнеризованих PostgreSQL-інстансів. У Microsoft Azure для цього застосовувався Azure Database for PostgreSQL Flexible Server, а в Amazon — Amazon RDS for PostgreSQL. Це дозволило винести з кластера зберігання даних і покласти відповідальність за резервне копіювання, оновлення та масштабування на хмарного провайдера.

Ще однією ключовою метою стало забезпечення cloud-ready підходу до архітектури. Усі сервіси розгорнуті як окремі контейнери, збережені в хмарному реєстрі (Azure Container Registry або Amazon ECR) і автоматизовано запущені у Kubernetes-кластері. Для зручного маршрутизування зовнішніх запитів використовувався Ingress Controller у поєднанні з хмарним Load Balancer.

Також було поставлено завдання зібрати й порівняти досвід використання Azure та AWS для однакової системи: MentorMatch. Для цього у двох середовищах було розгорнуто ідентичні компоненти: API Gateway, Kafka, мікросервіси, моніторингова система на базі Prometheus та Grafana, а також керовані БД. Це дало змогу порівняти відмінності в інструментах, автоматизації, конфігурації, документації, зручності керування та стабільності.

У результаті вдалося сформувати повну картину технічних відмінностей між двома провідними хмарними платформами у контексті реального прикладу веб-застосунку. Зібрані дані стали основою для детального порівняння, яке наведено в наступних підрозділах.

4.2 Розгортання у Microsoft Azure

4.2.1 Компоненти, що було розгорнуто

У середовищі Microsoft Azure було розгорнуто повноцінну інфраструктуру для роботи застосунку MentorMatch. Основу становить Azure Kubernetes Service (AKS), у межах якого кожен мікросервіс системи — зокрема auth-service, profile-service, notification-service, session-rating-service та frontend — розгорнуто у вигляді окремого контейнера. Всі Docker-образи були попередньо зібрані локально й завантажені до Azure Container Registry (ACR), звідки вони автоматично підтягуються у кластер при розгортанні.

Замість локальних або контейнеризованих СУБД у межах Azure було використано керований сервіс Azure Database for PostgreSQL Flexible Server. Для кожного мікросервісу створено окрему базу даних, розміщену на одному сервері. Доступ до баз даних здійснюється через DNS-імена, а конфіденційні параметри — зокрема облікові дані — зберігаються у відповідних хмарних сховищах секретів: Azure Key Vault для розгортання в Microsoft Azure та AWS Secrets Manager для середовища Amazon Web Services. Замість Kubernetes Secret реалізовано автоматичне монтування секретів у вигляді змінних середовища за допомогою інтеграції з CSI-драйверами.

Крім основних сервісів, у кластері було розгорнуто Apache Kafka як брокер подій. Kafka працює у парі з Zookeeper і розгорнута в AKS за допомогою Helm-чарту [18]. Для маршрутизації HTTP-запитів і доступу до веб-інтерфейсу та API використовується Ingress Controller, який пов’язаний з Azure Load Balancer і виконує роль єдиного вхідного шлюзу.

4.2.2 Особливості реалізації

Процес розгортання системи в Azure реалізовано за допомогою CLI-команд (az cli) та конфігураційних YAML-файлів. Усі компоненти, зокрема деплойменти мікросервісів, сервіси, інгреси, секрети та конфігурації, були описані у вигляді окремих Kubernetes-ресурсів. Структура проєкту дозволяє централізовано керувати конфігурацією кожного сервісу, що значно спрощує підтримку та оновлення.

Контейнери було завантажено до ACR, після чого кожен мікросервіс автоматично підтягує свій образ з приватного реєстру під час розгортання в AKS. Параметри підключення до бази даних передаються через змінні середовища у вигляді секретів, зокрема: назва бази, хост, логін, пароль. З'єднання з Azure PostgreSQL реалізовано через стандартні TCP-запити по порту 5432.

Інтеграція з Kafka відбувається через змінні середовища, у яких вказано брокер, порт і топіки. Споживачі Kafka реалізовані як окремі поди, які запускаються паралельно з основними мікросервісами та обробляють події незалежно. Конфігурація Kafka передбачає використання персистентного сховища, що дозволяє зберігати події навіть після рестарту брокера.

Ingress Controller налаштовано на маршрутизацію запитів до різних сервісів за допомогою path-based routing. Усі зовнішні запити надходять через єдиний вхідний домен, після чого передаються до відповідного сервісу згідно з конфігурацією Ingress. Це спрощує доступ до API та фронтенду без потреби в окремих публічних IP для кожного компонента.

Для безпечного зберігання конфіденційних даних, зокрема паролів до бази даних, секретних ключів та конфігурацій, було інтегровано Azure Key Vault. У зв’язці з Secrets Store CSI Driver секрети з Key Vault автоматично монтуються у контейнери у вигляді змінних середовища. Такий підхід дозволив централізовано керувати секретами для кожного мікросервісу без використання відкритих значень у YAML-файлах. Замість статичних Kubernetes Secrets

використовувалися посилання на об’єкти в Key Vault, що зменшує ризик компрометації та спрощує ротацію ключів.

У результаті вдалося реалізувати повноцінне хмарне розгортання мікросервісної системи з ізольованими сервісами, керованими базами даних та автоматизованим обробленням подій.

4.3 Розгортання в Amazon Web Services (AWS)

4.3.1 Компоненти, що було розгорнуто

У середовищі Amazon Web Services розгортання системи MentorMatch здійснювалося з використанням основних інструментів платформи для контейнеризації, зберігання даних та обробки подій. Як основа кластерної інфраструктури використовувався Amazon EKS (Elastic Kubernetes Service), який забезпечує запуск контейнеризованих застосунків з автоматичним керуванням масштабуванням і оновленнями. Для зберігання Docker-образів застосовувався Amazon Elastic Container Registry (ECR), у який було завантажено всі мікросервіси системи.

Кожен мікросервіс — auth-service, profile-service, notification-service, session-rating-service та frontend — було описано за допомогою YAML-файлів із зазначенням конфігурації Deployment, Service та Ingress. Доступ до баз даних реалізовано через Amazon RDS for PostgreSQL, де для кожного сервісу створено окрему базу. Підключення здійснюється через DNS-ім’я сервера, а параметри авторизації зберігаються у хмарних менеджерах секретів Azure Key Vault та AWS Secrets Manager.

Для асинхронної обробки подій, як і в Azure, використовується Apache Kafka, розгорнута у кластері за допомогою Helm-чарту Bitnami. Разом із Kafka було розгорнуто Zookeeper. Для маршрутизації HTTP-запитів і публічного доступу до застосунку використовувався AWS Load Balancer Controller, який створює відповідний Network Load Balancer, прив’язаний до Ingress-ресурсів.

4.3.2 Особливості реалізації

Процес створення кластеру в AWS передбачав налаштування декількох критичних компонентів, зокрема OIDC-провайдера, IAM-ролей для сервісних акаунтів та налаштування VPC. Для автоматизації розгортання кластера

використовувався інструмент eksctl, який дозволяє швидко створювати EKS-кластери з урахуванням безпекових політик і параметрів масштабування [19].

Після створення кластера до нього було підключено CLI-контекст, що дало змогу розгорнути всі компоненти через kubectl і Helm.

Docker-образи мікросервісів попередньо було зібрано локально та завантажено в Amazon ECR. Під час розгортання кожен сервіс отримує відповідний образ з реєстру та автоматично розгортається в EKS. Доступ до баз даних здійснюється через Azure Key Vault та AWS Secrets Manager, інтегровані з Kubernetes за допомогою Secrets Store CSI Driver та IRSA відповідно.

Kafka розгорнута як Helm-чарт із налаштуванням персистентного сховища для збереження подій, навіть у разі рестарту брокера. Споживачі Kafka реалізовано як окремі поди, що працюють незалежно від основних мікросервісів. Застосовано логіку повторного споживання подій при збої, що дозволяє зберігати цілісність бізнес-процесів.

Для маршрутизації HTTP-запитів застосовано AWS Load Balancer Controller, що автоматично створює Network Load Balancer на основі Ingress-ресурсів. Це рішення забезпечує коректну обробку запитів до API й фронтенду, без потреби у ручному керуванні балансуванням.

Для підвищення безпеки конфігурацій було налаштовано інтеграцію з AWS Secrets Manager. Для цього у кластері встановлено Secrets Store CSI Driver у поєднанні з провайдером AWS, що забезпечує автоматичне підключення секретів до подів без зберігання чутливої інформації у звичайних Kubernetes Secret-об’єктах. Доступ до секретів було реалізовано через IAM-ролі сервісних акаунтів, які обмежують доступ лише до дозволених секретів. Усі критичні змінні середовища — включно з паролями, токенами та ключами — надходять у сервіси безпосередньо з Secrets Manager, що спрощує керування конфігурацією та ротацію значень.

У результаті реалізовано повноцінне розгортання мікросервісної архітектури в AWS з повною підтримкою подієвої взаємодії, керованих баз даних та балансування навантаження.

4.4 Порівняння Azure та AWS за ключовими аспектами

Для оцінки хмарної готовності системи MentorMatch було виконано повноцінне розгортання в обох провідних хмарних середовищах — Microsoft Azure та Amazon Web Services (AWS). У кожній платформі застосовано однакову технологічну основу: мікросервіси, Apache Kafka, керовані бази даних PostgreSQL, система моніторингу та маршрутизація трафіку через Ingress-контролер.

У процесі реалізації було виокремлено технічні параметри, що впливають на ефективність розгортання мікросервісних веб-застосунків. Порівняльна таблиця нижче демонструє реальні відмінності, виявлені під час практичного впровадження.

Критерії	Microsoft Azure	AWS
Створення кластера	Просте, через Azure CLI або портал	Більш складне: через “eksctl”, IAM, VPC
Завантаження Docker-образів	Azure Container Registry (ACR)	Amazon Elastic Container Registry (ECR)
Бази даних	Azure Database for PostgreSQL Flexible Server	Amazon RDS for PostgreSQL
Розгортання Kafka	Через Helm у AKS	Через Helm в EKS
Обробка Ingress	LoadBalancer-сервіс для API Gateway	Ingress-контролер (Load Balancer Controller) + Application Load Balancer

Параметри безпеки (Secrets)	Azure Key Vault + Secrets Store CSI Driver	AWS Secrets Manager + IAM (IRSA)
Простота старту для новачків	Вища	Нижча
Гнучкість та контроль	Менша — фокус на автоматизацію	Вища — більша кількість ручних параметрів

4.5 Підсумки аналізу платформ

У межах розділу реалізовано повноцінне хмарне розгортання мікросервісної системи MentorMatch у двох середовищах — Microsoft Azure та Amazon Web Services (AWS). Для обох платформ використано однакову архітектурну модель, що включає Kubernetes-кластер (AKS або EKS), приватний контейнерний реєстр, керовані бази даних PostgreSQL, брокер повідомлень Kafka, систему маршрутизації HTTP-запитів та інструменти моніторингу.

Під час реалізації було зафіксовано помітні відмінності у підходах до конфігурування інфраструктури. Azure забезпечує вищий рівень автоматизації та спрощене створення кластерів і ресурсів за допомогою CLI або графічного інтерфейсу. AWS, навпаки, вимагає більшої кількості ручних налаштувань, зокрема через eksctl, IAM-ролі та VPC-конфігурацію, але надає більший контроль над мережею, доступами та безпекою.

Незважаючи на різницю в інструментах, у обох середовищах вдалося досягти повної функціональної відповідності: мікросервіси працюють автономно, база даних кожного сервісу розміщена в окремому керованому інстансі, Kafka передає події між сервісами, а моніторинг дозволяє відстежувати продуктивність і стан системи в реальному часі.

У результаті було підтверджено, що обидві хмарні платформи повністю відповідають вимогам cloud-native архітектури та дозволяють реалізувати масштабовану, ізольовану, спостережувану і стабільну інфраструктуру для веб-застосунку. Зібрані у процесі розгортання спостереження лягли в основу

порівняльного аналізу, поданого у таблиці, і стали вихідною точкою для формування технічних висновків у завершальному розділі дипломної роботи.

Висновки

У межах дипломної роботи реалізовано платформу MentorMatch, побудовану на мікросервісній архітектурі з незалежними сервісами для аутентифікації, роботи з профілями, керування сесіями, рейтингами й сповіщеннями. Для кожного сервісу організовано окрему базу даних, використано брокер Kafka для передачі подій, усі компоненти контейнеризовано у Docker й розгорнуто у Kubernetes-кластері.

Під час реалізації було налаштовано розгортання платформи у двох хмарних середовищах — Microsoft Azure і Amazon Web Services. У процесі конфігурування інфраструктури використано керовані бази даних (Azure Database for PostgreSQL, Amazon RDS for PostgreSQL), секрети зберігалися у відповідних сховищах (Azure Key Vault, AWS Secrets Manager), організовано балансування навантаження, налаштовано моніторинг через Prometheus і Grafana.

Azure забезпечив більш простий старт: усі сервіси інтегрувалися без зайвих складнощів, а робота з Key Vault і налаштуванням бази даних не викликала питань. Усі основні дії виконувалися через зрозумілий інтерфейс, що пришвидшило запуск проєкту. В Amazon Web Services налаштування конфігурацій вимагало більше часу через налаштування IAM-ролей, інтеграцію секретів, підключення до RDS і управління доступом. Однак у підсумку саме AWS дозволив отримати гнучкіші налаштування й ширший контроль над параметрами інфраструктури.

У результаті розгортання платформи в обох хмарах довелося пройти повний цикл роботи з реальними сервісами, налаштувати ізоляцію компонентів, захистити секрети, забезпечити масштабування і моніторинг системи. Порівняння платформ дозволило побачити відмінності підходів, оцінити швидкість старту, легкість налаштування, можливості кастомізації й рівень автоматизації процесів.

Отриманий досвід із побудови мікросервісної системи й інтеграції з хмарними платформами підтвердив актуальність такого підходу для створення сучасних веб-застосувань і заклав фундамент для подальшого професійного розвитку у сфері розробки та DevOps.

Список використаної літератури

1. Evolving Architecture [Електронний ресурс] / доступно за адресою: https://www.linkedin.com/pulse/evolving-architecture-journey-from-monolithic-soa-avita-katal/

2. Monolith [Електронний ресурс] / доступно за адресою: https://martinfowler.com/bliki/MonolithFirst.html

3. Modular Architecture [Електронний ресурс] / доступно за адресою: https://martinfowler.com/articles/linking-modular-arch.html

4. What Is SOA [Електронний ресурс] / доступно за адресою: https://www.oracle.com/ua/service-oriented-architecture-soa/#:~:text=Service%2Doriented%20architecture%20(SOA),a%20variety%20of%20business%20applications.

5. Microservices [Електронний ресурс] / доступно за адресою: https://martinfowler.com/articles/microservices.html

6. Microservices Architecture [Електронний ресурс] / доступно за адресою: https://www.vsourz.com/blog/microservices-explained-all-you-ever-wanted-to-know-about-microservices-architecture/

7. Kubernetes [Електронний ресурс] / доступно за адресою: https://kubernetes.io/docs/concepts/overview/

8. Cloud-native architectures [Електронний ресурс] / доступно за адресою: https://www.liquidweb.com/blog/cloud-native-architectures/

9. Design a DDD-oriented microservice [Електронний ресурс] / доступно за адресою: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice

10. The Twelve Factors [Електронний ресурс] / доступно за адресою: https://12factor.net/

11. REST [Електронний ресурс] / доступно за адресою: https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

12. gRPC [Електронний ресурс] / доступно за адресою: https://grpc.io/docs/what-is-grpc/introduction/

13. Kafka In Microservices [Електронний ресурс] / доступно за адресою: https://medium.com/@ozziefel/kafka-in-microservices-architecture-enabling-scalable-and-event-driven-systems-7ff474de49f4

14. API gateway [Електронний ресурс] / доступно за адресою: https://learn.microsoft.com/en-us/azure/architecture/microservices/design/gateway

15. Microservice Architecture API gateway [Електронний ресурс] / доступно за адресою: https://microservices.io/patterns/apigateway.html

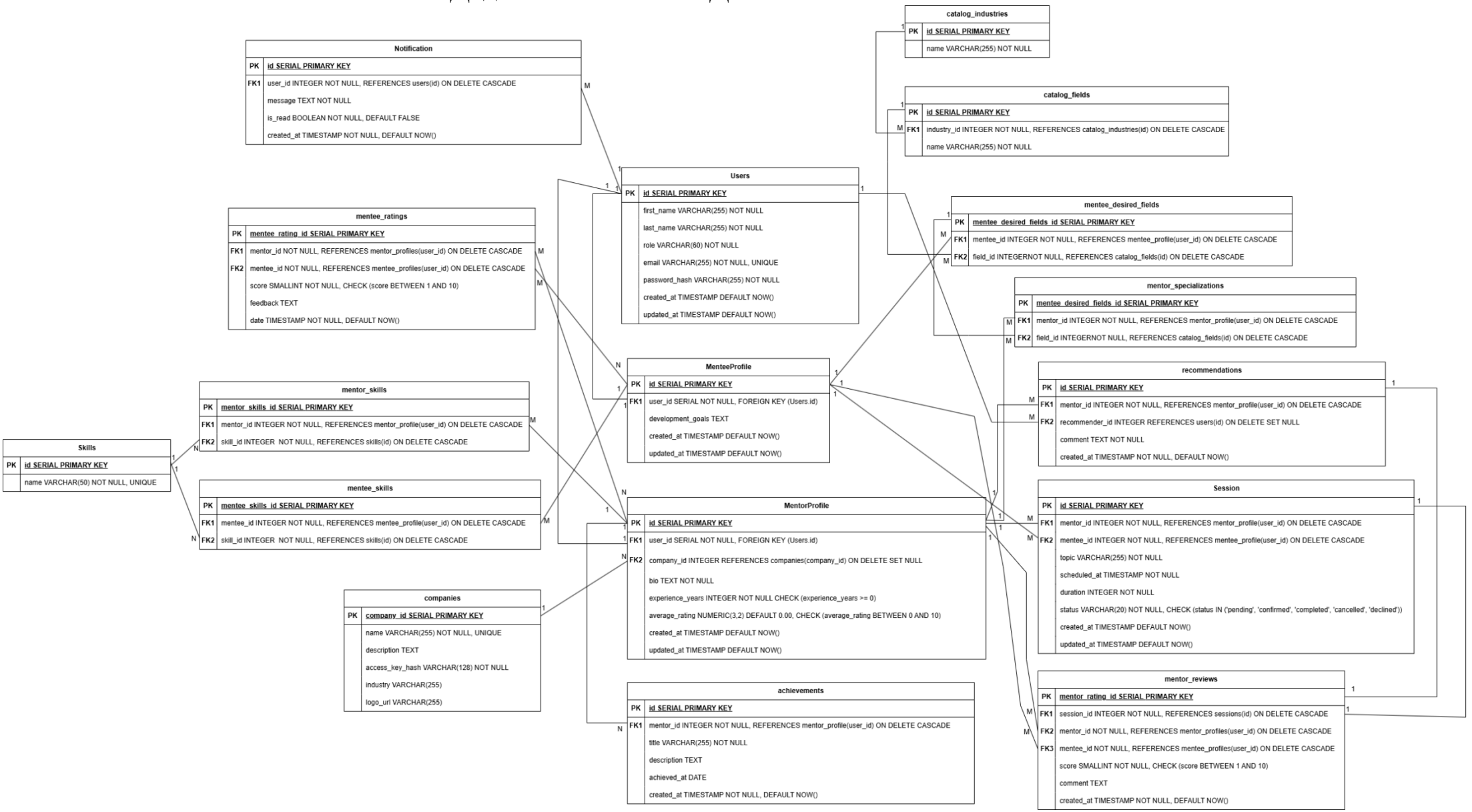
16. Outbox Pattern [Електронний ресурс] / доступно за адресою: https://medium.com/design-microservices-architecture-with-patterns/outbox-pattern-for-microservices-architectures-1b8648dfaa27

17. Apache Kafka [Електронний ресурс] / доступно за адресою: https://medium.com/@platform.engineers/event-driven-microservices-architecture-with-apache-kafka-0d52361c2624

18. Azure Apache Kafka cluster [Електронний ресурс] / доступно за адресою: https://learn.microsoft.com/en-us/azure/hdinsight/kafka/apache-kafka-get-started

19. Amazon EKS [Електронний ресурс] / доступно за адресою: https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html

Додаток А. Схема БД системи MentorMatch



Додаток Б

producer_delete_profile

```
KAFKA_BROKER = os.environ.get("KAFKA_BROKER")

TOPIC_PROFILE_DELETED = os.environ.get("KAFKA_TOPIC_PROFILE_DELETED")

producer = KafkaProducer(

    bootstrap_servers=[KAFKA_BROKER],

    value_serializer=lambda v: json.dumps(v).encode("utf-8")

)

def publish_profile_deleted_event(user_id):

    event = {

        "user_id": user_id,

        "event": "profile.deleted"

    }

    producer.send(TOPIC_PROFILE_DELETED, event)

    producer.flush()

    print(f"Published profile deleted event: {event}")

# If the profile creation function already exists, it will remain unchanged:

def publish_new_user_event(user):

    event = {

        "user_id": user.id,

        "email": user.email,

        "first_name": user.first_name,

        "last_name": user.last_name,

        "role": user.role,

    }
```

```
producer.send(os.environ.get("KAFKA_TOPIC_NEW_USER", "new_user_topic"), event)
producer.flush()
print(f"Published new user event: {event}")
```

consumer_profile_delete

```
def handle(self, *args, **options):
    kafka_broker = os.environ.get("KAFKA_BROKER")
    topic = os.environ.get("KAFKA_TOPIC_PROFILE_DELETED")
    consumer = KafkaConsumer(
        topic,
        bootstrap_servers=[kafka_broker],  auto_offset_reset='earliest',
        enable_auto_commit=True,  group_id='auth_service_profile_deletion',
        value_deserializer=lambda x: json.loads(x.decode("utf-8"))
    )
    self.stdout.write(self.style.SUCCESS("Auth Service Kafka consumer for profile deletion started."))
    for message in consumer:
        event = message.value
        user_id = event.get("user_id")
        if event.get("event") == "profile.deleted" and user_id:
            self.stdout.write(f"Received deletion event for user_id: {user_id}")
            try:
                user = CustomUser.objects.get(id=user_id)
                user.delete()
                self.stdout.write(self.style.SUCCESS(f'Deleted user with id: {user_id}"))
```

```
except CustomUser.DoesNotExist:
    self.stdout.write(f"User with id {user_id} not found.")
```