Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти перший (бакалаврський)

Методи інтеграції та узгодження мікросервісів в хмарній архітектурі

(тема)

Виконав:

студент IV курсу, групи СПм-22-3

Дюльгер В.Д.
(прізвище, ініціали)

Спеціальність 123 - Комп’ютерна інженерія

123 «Комп’ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування

Системне програмування
(повна назва освітньої програми)

Керівник: Ст. викл. Сорокін А.Р.
(посада, прізвище, ініціали)

Допускається до захисту

зав. кафедри ЕОМ _____ Коваленко А.А.
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп’ютерна інженерія»
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав.
кафедри _____
(підпис)

“ ___ ”_____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Дюльгеру Владиславу Дмитровичу
(прізвище, ім’я, по батькові)

1. Тема роботи Методи інтеграції та узгодження мікросервісів в хмарній архітектурі

затверджена наказом по університету від “ 1 ” квітня 2024 р. № 257 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 15 червня 2024 р.

3. Вхідні дані до роботи 1) Тема роботи; 2) Середовище розробки
3) Мікросервіси

4. Перелік питань, що потрібно опрацювати у роботі
1) Аналіз предметної області;
2) Вибір і обґрунтування обладнання;
3) Вибір і обґрунтування програмних засобів;
4) Побудова системи позиціонування
5) Аналіз розміщення маяків в приміщенні
6) Висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) 12 слайдів

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

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

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз проблеми та огляд існуючих рішень	01.04.24-15.14.24	
2	Вибір технології розробки та	16.04.24-30.04.24	
	інструментальних засобів		
3	Розробка алгоритмічного забезпечення	01.05.24-15.05.24	
4	Розробка програмних модулів	16.05.24-25.05.24	
5	Відлагодження програмних модулів	26.05.24-28.05.24	
6	Оформлення матеріалів кваліфікаційної роботи	29.05.24-05.06.24	
7	Подання кваліфікаційної роботи керівникові та	07.06.24-08.06.24	
	її попередній захист		
8	Подання кваліфікаційної роботи на рецензу-	09.06.24-12.06.24	
	вання		

Дата видачі завдання 1 квітня 2024 р.

Студент _____
(підпис)

Керівник роботи _____ Ст. викл. Сорокін А.Р.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Кваліфікаційна робота: 121 стор., 17 рис., 1 додаток, 10 джерел. Графічний матеріал кваліфікаційної роботи містить 13 аркушів.

МІКРОСЕРВІСНА АРХІТЕКТУРА, ХМАРНІ ОБЧИСЛЕННЯ, AWS LAMBDA, POSTGRESQL, CI/CD, МАСШТАБОВАНІСТЬ, АНАЛІТИКА ОБМІННИХ КУРСІВ

Об’єкт дослідження – процес розробки та впровадження платформи для аналітики обмінних курсів криптовалют, побудованої за принципами мікросервісної архітектури.

Предмет дослідження – методи та технології реалізації мікросервісної архітектури, використання хмарних обчислень для збору та обробки даних, інтеграція AWS сервісів для забезпечення масштабованості та надійності.

Мета роботи – аналіз існуючих методів реалізації мікросервісної архітектури та хмарних обчислень, а також їх застосування для створення оптимізованої платформи аналітики обмінних курсів.

Методи дослідження – методи мікросервісного проектування: декомпозиція системи на сервіси, визначення межі функціональності сервісів, комунікація між ними; використання хмарних сервісів AWS для реалізації серверлесс функцій, моніторинг та оптимізація за допомогою AWS CloudWatch та X-Ray; принципи DevOps для безперервної інтеграції та безперервного розгортання (CI/CD).

Результати роботи – модель мікросервісної архітектури для платформи аналітики обмінних курсів, яка описує компоненти системи, їхню взаємодію, принципи розгортання та підтримки системи.

Область застосування – розробка та оптимізація програмного забезпечення для платформ аналітики фінансових даних та обмінних курсів криптовалют.

ABSTRACT

Qualification work: 121 p., 17 pic., 10 sources, 1 appendix. Graphic material of the attestation work contains 13 posters.

MICROSERVICE ARCHITECTURE, CLOUD COMPUTING, AWS LAMBDA, POSTGRESQL, CI/CD, SCALABILITY, CRYPTO EXCHANGE ANALYTICS

Research object – the process of developing and implementing a platform for cryptocurrency exchange analytics, built on the principles of microservice architecture.

Subject of research – methods and technologies for implementing microservice architecture, utilizing cloud computing for data collection and processing, integrating AWS services to ensure platform scalability and reliability.

Purpose of the work – to analyze existing methods of implementing microservice architecture and cloud computing, and their application to create an optimized analytics platform for exchange rates.

Research methods – microservice design methods: decomposing the system into services, defining the boundaries of service functionalities, communication between them; utilizing AWS cloud services for implementing serverless functions, monitoring, and optimization with AWS CloudWatch and X-Ray; DevOps principles for continuous integration and continuous deployment (CI/CD).

Work results – a model of microservice architecture for the exchange rate analytics platform, which describes the system components, their interaction, and the principles of deployment and maintenance of the system.

Scope – development and optimization of software for financial data analytics platforms, focusing on cryptocurrency exchange rates.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

CI/CD - Continuous Integration and Continuous Deployment (Безперервна інтеграція та безперервне розгортання)

HTTP - HyperText Transfer Protocol (Протокол передачі гіпертексту)

API - Application Programming Interface (Інтерфейс програмування застосунків)

AWS - Amazon Web Services (Хмарні сервіси Амазон)

S3 - Simple Storage Service (Простий сервіс зберігання)

RDS - Relational Database Service (Сервіс реляційних баз даних)

SNS - Simple Notification Service (Простий сервіс сповіщень)

VPC - Virtual Private Cloud (Віртуальна приватна хмара)

JSON - JavaScript Object Notation (Нотація об'єктів JavaScript)

DynamoDB - Dynamo Database (База даних Dynamo)

SQL - Structured Query Language (Структурована мова запитів)

JVM - Java Virtual Machine (Віртуальна машина Java)

IDE - Integrated Development Environment (Інтегроване середовище розробки)

TLS - Transport Layer Security (Безпека транспортного рівня)

SSL - Secure Sockets Layer (Рівень захищених сокетів)

IaaS - Infrastructure as a Service (Інфраструктура як послуга)

PaaS - Platform as a Service (Платформа як послуга)

SaaS - Software as a Service (Програмне забезпечення як послуга)

FaaS - Function as a Service (Функція як Послуга)

REST - Representational State Transfer (Передача репрезентативного стану)

URL - Uniform Resource Locator (Уніфікований покажчик ресурсу)

VPN - Virtual Private Network (Віртуальна Приватна Мережа)

XML - Extensible Markup Language (Розширювана Мова Розмітки)

OS - Operating System (Операційна Система)

DNS - Domain Name System (Система Доменної Імені)

TCP - Transmission Control Protocol (Протокол Управління Передачею)

HTTPS - Hypertext Transfer Protocol Secure (Безпечний Протокол Передачі Гіпертексту)

YAML - YAML Ain't Markup Language (YAML Не Є Мовою Розмітки)

JWT - JSON Web Token (Веб-Токен JSON)

CDN - Content Delivery Network (Мережа Доставки Контенту)

VM - Virtual Machine (Віртуальна машина)

IAM - Identity and Access Management (Управління ідентифікацією та доступом)

SLA - Service Level Agreement (Угода про рівень обслуговування)

QoS - Quality of Service (Якість обслуговування)

KPI - Key Performance Indicator (Ключовий показник ефективності)

DDoS - Distributed Denial of Service (Розподілена відмова в обслуговуванні)

FTP - File Transfer Protocol (Протокол передачі файлів)

ВСТУП

У сучасному світі технологій мікросервіси та хмарні обчислення стали важливими інструментами для створення ефективних, гнучких та масштабованих програмних рішень. Ці технології дозволяють компаніям розробляти складні системи, що легко адаптуються до змін, забезпечуючи високу продуктивність та надійність. Дана робота присвячена вивченню та оптимізації мікросервісів у хмарному середовищі на прикладі розробки платформи для аналітики обмінних курсів криптовалют.

Цей звіт надає детальний опис процесу розробки та впровадження платформи, аналізує ключові аспекти оптимізації мікросервісів у хмарному середовищі.

Метою даного дослідження є розробка та впровадження оптимізованої платформи, яка використовує мікросервісну архітектуру та хмарні обчислення для збору, обробки та аналізу даних з криптовалютних бірж. У цьому контексті були обрані сучасні технології та сервіси для забезпечення максимальної продуктивності, надійності та масштабованості системи.

Практична частина роботи включає розробку платформи, що складається з кількох мікросервісів, кожен з яких виконує окрему функцію, таку як збір даних, обробка та збереження інформації, та надання аналітичних звітів. В рамках розробки платформи були використані такі сервіси, як AWS Lambda для реалізації серверлесс функцій, що дозволяє автоматично масштабувати систему відповідно до навантаження, забезпечуючи високу продуктивність при мінімальних витратах, а також AWS API Gateway для створення та управління RESTful API, що забезпечує ефективний інтерфейс між мікросервісами та зовнішніми клієнтами.

Для зберігання структурованих даних була використана база даних Postgres, що дозволяє швидко та ефективно обробляти великі обсяги інформації. Моніторинг та трасування запитів здійснюються за допомогою

AWS CloudWatch та X-Ray, що забезпечує глибокий аналіз продуктивності системи та допомагає виявляти та усувати вузькі місця. Моніторинг забезпечує виявлення аномалій у реальному часі, а трасування запитів дозволяє глибоко аналізувати кожен етап обробки даних у системі, що є критично важливим для своєчасного виявлення та усунення проблем. У процесі реалізації проекту були вирішені ряд технічних викликів, таких як забезпечення безперебійної роботи мікросервісів, ефективне управління даними та інтеграція різних компонентів системи.

Нарешті, у роботі обговорюються можливі шляхи подальшого розвитку та вдосконалення платформи, включаючи впровадження додаткових аналітичних інструментів, розширення функціональності системи та покращення методів обробки та аналізу даних.

1 АНАЛІЗ МІКРОСЕРВІСІВ ТА ПАРАДИГМИ ХМАРНИХ ОБЧИСЛЕНЬ

1.1 Визначення мікросервісів

Мікросервіси, або архітектура мікросервісів, - це особливий метод розробки програмних систем, який фокусується на створенні однофункціональних модулів з чітко визначеними операціями та інтерфейсами. Це модель для розробки складних програмних систем як набору невеликих, незалежних і слабко пов'язаних між собою сервісів. Кожен з цих сервісів має специфічну, єдину бізнес-можливість, яку він досягає найкращим чином. Вони розгортаються незалежно, можуть взаємодіяти один з одним за допомогою простих, загальнодоступних API і можуть бути написані різними мовами програмування.

Архітектура мікросервісів організовує додатки як набори слабко пов'язаних між собою сервісів. На відміну від монолітного архітектурного стилю, де додаток будується як єдине ціле, мікросервіси організовують додаток як набір пов'язаних між собою сервісів. Цей архітектурний стиль - це підхід до розробки одного додатку як набору невеликих сервісів, кожен з яких виконує свій власний процес і взаємодіє з легкими механізмами, найчастіше з HTTP API ресурсу. [3]

Вона заохочує використання невеликих, автономних сервісів, які працюють разом, де "невеликий" означає розмір сервісу, а не розмір бази даних. Ці сервіси можуть підтримуватися різними командами розробників і можуть бути розроблені з використанням різних технологій, включаючи різні мови програмування і програмні середовища.

Коротко кажучи, мікросервіси - це метод розробки програмних додатків, які складаються з незалежних модульних сервісів, що розгортаються незалежно один від одного.

1.1.1 Ключові характеристики мікросервісів

Розробка сучасних програмних додатків для великомасштабних систем може бути складним і непростим завданням. Однак поява архітектури мікросервісів, безсумнівно, спростила цей процес. Мікросервіси, як модульні компоненти загальної системи, мають чіткі характеристики, які відрізняють їх від інших структур розробки програмного забезпечення. Ці характеристики передають їх унікальні можливості, які ілюструють причини, що лежать в основі їх зростаючої популярності у сфері системної розробки та інженерії програмного забезпечення.

Почнемо з того, що характерною рисою мікросервісів є їхня автономна природа. Мікросервіси - це незалежно розгорнуті одиниці програмного забезпечення, які працюють дискретно, але гармонійно в рамках більшої прикладної системи. Ця характеристика полегшує ізоляцію сервісів, дозволяючи кожному з них працювати у своєму окремому середовищі. Коли один сервіс зазнає збою або проблеми, це не призводить до падіння всієї системи. Натомість, несправний сервіс можна обслуговувати або замінити незалежно, не спричиняючи перебоїв у роботі всієї системи. Незалежне розгортання також дозволяє створювати версії сервісів. Розробники можуть оновлювати, розгортати та масштабувати сервіси незалежно, зменшуючи ризики, пов'язані з модифікаціями в монолітних системах, де навіть для невеликих змін потрібно перерозгортати весь додаток.

Крім того, архітектурні структури мікросервісів особливим чином організовані навколо окремих бізнес-можливостей. Це означає, що кожна послуга інкапсулює одну бізнес-функцію або процес, наприклад, управління запасами або автентифікацію користувачів. Ця особлива організаційна стратегія дає змогу організаціям тісніше узгоджувати свою програмну архітектуру з бізнес-цілями.[1] Вона забезпечує чітку видимість того, як додатки сприяють розвитку бізнес-можливостей, узгоджуючи технічну реалізацію з бізнес-потребами більш ефективно, ніж інші архітектурні патерни.

Окрім того, що мікросервіси орієнтовані на бізнес, вони є самодостатніми сутностями, які інкапсулюють дані, з якими вони працюють. Ця особливість забезпечує автономність в управлінні даними, де кожен сервіс має свою ізольовану базу даних або сховище даних. Взаємодія між сервісами зазвичай відбувається за допомогою мовно-діагностичних API, часто через RESTful

HTTP або черги повідомлень. Такий дизайн гарантує, що кожен мікросервіс є єдиним джерелом істини для конкретного домену.

Нарешті, мікросервіси вирізняються своєю поліглотською природою, завдяки потужній підтримці багатьох технологій і платформ. Ця особливість дає розробникам свободу використовувати найкращий стек технологій, що відповідає унікальним потребам кожного мікросервісу. Отже, один сервіс може бути написаний на Java, інший - на Python, а третій - на Node.js, залежно від вимог конкретної бізнес-можливості, яку інкапсулює сервіс.

Використовуючи ці визначальні характеристики, архітектура мікросервісів забезпечує більш гнучкий, адаптивний та масштабований підхід до розробки програмного забезпечення. Цей дизайн сприяє стійкості до системних збоїв і забезпечує архітектурну структуру, яка тісно пов'язана з бізнес- можливостями, одночасно пристосовуючись до різноманітних технологічних стеків. Як наслідок, архітектура мікросервісів набуває все більшої популярності і все частіше використовується компаніями, які вимагають від своїх програмних систем гнучкості, масштабованості та надійності.

Однією з основоположних характеристик мікросервісної архітектури є те, що вона значною мірою складається з відокремлених сервісів, що є значним відходом від переплетеної структури, яка зазвичай спостерігається в монолітній архітектурі. Розділені сервіси означають категорично незалежну природу кожного сервісу, що означає, що вони можуть працювати автономно один від одного, не впливаючи на роботу інших сервісів у системі і не зазнаючи впливу з боку інших сервісів.

Цей особливий дизайн характеризується інкапсуляцією специфічних

бізнес- можливостей в окремих сервісах, які є самодостатніми, тобто працюють незалежно. Кожен сервіс претендує на недоторканність своєї функціональності та супровідної бази даних, тим самим зменшуючи ступінь внутрішньої залежності, яка зазвичай притаманна монолітним системам. Отже, модифікації, вдосконалення або проблеми, що виникають в межах одного сервісу, не поширюються на інші частини системи завдяки цій характеристиці відокремленості. Цей принцип суттєво стримує поширення проблем між сервісами, додаючи додатковий рівень стійкості до потенційних збоїв системи. [2]

Більше того, концепція відокремлення надає цим сервісам свободу розгортання та оновлення в ізоляції. На відміну від монолітних додатків, оновлення або розширення функцій мікросервісу не вимагає повного перерозгортання програми. Аналогічно, масштабування певного сервісу може бути виконано незалежно, залежно від попиту на його функціональність, а не масштабування всього додатку, таким чином підвищуючи ефективність та продуктивність системи.

Важливо, що зв'язок між відокремленими сервісами, як правило, здійснюється через легкі API або механізми, керовані подіями. Ці механізми полегшують контрольований обмін інформацією, зберігаючи при цьому автономність цих сервісів. API забезпечують рівень абстракції, який приховує складність базових сервісів і надає спрощений інтерфейс для міжсервісної комунікації, таким чином гарантуючи, що кожен сервіс не знає про внутрішню роботу інших сервісів, з якими він взаємодіє.

Однак тут слід зробити застереження: відокремлення має бути реалізоване з розумом. Хоча цей принцип є основоположним для філософії незалежних мікросервісів, доведення його до крайнощів може призвести до розпорошеності та неефективності системи. Повне відокремлення може призвести до того, що сервіси стануть надмірно самодостатніми, що призведе до потенційного дублювання або неузгодженості. Рохробники повинні знайти баланс між адекватним розділенням і необхідністю підтримувати певний рівень

координації та узгодженості між послугами.

На завершення слід зазначити, що відокремлені сервіси залишаються однією з центральних визначальних особливостей архітектури мікросервісів. Вона надає організаціям більшої гнучкості, стійкості та ефективності в сучасну динамічну цифрову епоху. Однак її слід впроваджувати стратегічно, пам'ятаючи про загальну узгодженість системи, щоб гарантувати, що система залишається єдиною у своїй меті, зберігаючи при цьому свою гнучкість.

Архітектура мікросервісів тісно пов'язана з потребами бізнесу, що відрізняється від традиційних схем розробки програмного забезпечення, орієнтованих на технології. У середовищі мікросервісів акцент зміщується в бік бізнес-орієнтованої перспективи, використовуючи принцип проектування мікросервісів, де кожна послуга структурована навколо певних бізнес-можливостей або операцій. Цей тонкий, але потужний зсув ставить бізнес-потреби на передній план при розробці програмного забезпечення, керуючи формуванням послуг, які безпосередньо відповідають конкретним бізнес-вимогам.

Бізнес-орієнтований підхід у контексті мікросервісів означає, що кожна мікропослуга в архітектурі інкапсулює певну бізнес-можливість. Ці можливості відповідають окремим бізнес-операціям або функціям, які можуть варіюватися від процесів управління персоналом, управління заємовідносинами з клієнтами до більш специфічних завдань, таких як обробка платежів або управління замовленнями клієнтів. Ця специфіка відводить від широкого, монолітного підходу до розробки програмного забезпечення, пропонуючи більш точне узгодження технологічних стратегій з бізнес-цілями і завданнями.

Заглиблюючись у значення та прояви бізнес-орієнтованих мікросервісів, ми можемо виділити кілька ключових характеристик.

Значною перевагою застосування бізнес-орієнтованого підходу є відчутна відповідність, яка встановлюється між функціональними можливостями програмного забезпечення та бізнес-цілями. Розбиваючи додаток на мікросервіси, кожен з яких паралельно виконує окремі бізнес-операції, ми

створюємо більш прямий зв'язок між бізнес-стратегією та реалізацією програмного забезпечення. Таке пряме узгодження сприяє прозорому розумінню того, як технічні ресурси сприяють досягненню бізнес-цілей, роблячи його доступним не лише для технічної команди, а й для інших зацікавлених сторін, включаючи керівників вищого рівня, менеджерів продуктів і навіть зовнішніх інвесторів. [7]

Іншим важливим наслідком бізнес-орієнтованого підходу в архітектурі мікросервісів є підвищена гнучкість у реагуванні на динаміку ринку та бізнесу. Враховуючи, що кожна мікропослуга по суті є самодостатньоюсутністю, яка втілює одну бізнес-операцію, зміни, що стосуються конкретної бізнес-вимоги, можуть бути швидко вирішені шляхом модифікації відповідної мікропослуги, не спричиняючи перебоїв в роботі інших послуг або системи в цілому. Така гнучкість має вирішальне значення в сучасному бізнес-ландшафті, що швидко розвивається і постійно змінюється.

Бізнес-орієнтованість мікропослуг також сприяє створенню міжфункціональних команд, які зосереджуються на комплексній відповідальності за певну мікропослугу. Кожна команда складається з представників різних секторів, які володіють багатогранними навичками, що разом охоплюють весь життєвий цикл мікропослуги, від розробки до розгортання та обслуговування. Така інтегрована структура команди зменшує залежність від інших команд, підвищує ефективність комунікації та прискорює цикл розробки.

Нарешті, бізнес-орієнтований характер мікросервісів позитивно впливає на вдосконалення бізнес-процесів. Оскільки кожна мікропослуга підкреслює певну бізнес-можливість, вони забезпечують вищий рівень деталізації процесів налаштування, полегшуючи функціональні налаштування та цільові вдосконалення. Отже, організації можуть оптимізувати свої операції на основі показників ефективності та зворотного зв'язку від кожного мікросервісу, тим самим підвищуючи загальну операційну ефективність і цінність бізнесу.

На завершення, покладання поняття бізнес-орієнтованого підходу в

основу архітектури мікросервісів докорінно змінює динаміку та результати розробки програмного забезпечення. Він сприяє розвитку культури гнучкості, покращує узгодженість з бізнес-цілями, сприяє оптимізації операцій і, зрештою, сприяє покращенню бізнес-результатів, оскільки в центрі уваги залишаються бізнес-можливості.

Розподілена розробка у сфері мікросервісів означає здатність розробляти, розгортати і підтримувати окремі незалежні сервіси окремими командами, які можуть бути розподілені по різних географічних точках або функціональних підрозділах організації. Такому способу дій сприяють і надихають притаманні мікросервісам принципи - окремішність, відокремленість і автономність сервісів. [5] Використання методології розподіленої розробки суттєво реструктурує традиційну динаміку розробки програмного забезпечення і відкриває нову еру спільної ефективності та технологічної незалежності.

Нижче наведені ключові аспекти та наслідки, які забарвлюють полотно розподіленої розробки мікросервісів:

Модель розподіленої розробки створює передумови для ефективної співпраці між командами, розкиданими по різних географічних точках. Це особливо важливо в нинішню епоху глобалізації та віддаленої роботи, коли фізичне місцезнаходження команди більше не є обмежуючим фактором. Свобода працювати в різних часових поясах, культурах і кадрових резервах може призвести до збагачення розмаїття думок і, як наслідок, до підвищення потенціалу для інновацій та продуктивності. Швидкість процесу розробки також може отримати поштовх, оскільки кілька команд, що працюють у різних часових поясах, можуть цілодобово сприяти просуванню загального проекту.

Фундаментальним для архітектури мікросервісів є принцип незалежних сервісів, де кожен сервіс є самодостатнім і може бути змінений або оновлений без ініціювання ефекту доміно для всієї системи. Така автономія полегшує розподілену розробку, оскільки значно зменшує залежність між командами, тим самим уможливлюючи паралельні та швидші цикли розробки.

Розподілена розробка за своєю суттю є гнучкою, що дозволяє

організаціям швидко адаптуватися до мінливих вимог ринку та потреб клієнтів. [4] Легкість масштабування окремих мікросервісів при дотриманні дискретних потреб кожного мікросервісу призводить до створення надзвичайно гнучкої та масштабованої системи, яка може відповідати темпам динаміки бізнесу.

Важливим наслідком розподіленої розробки є поява сприятливої екосистеми для постійного вдосконалення та інновацій. Надаючи можливість окремим командам взяти на себе відповідальність за свій мікросервіс, від проектування через розгортання до обслуговування, вони можуть приймати обґрунтовані рішення щодо оптимального технологічного стеку і методологій для своїх унікальних потреб. Це сприяє створенню сприятливого середовища для інновацій і запускає механізм безперервного повторення і вдосконалення.

Розподіл функціональності системи між різними автономними сервісами, розробленими окремими командами, розподіляє системні ризики та підвищує загальну стійкість системи. Якщо збій відбувається в одному мікросервісі, він не виходить за межі цього сервісу, тобто інші сервіси, а отже, і робота інших команд, залишаються непорушними.

Хоча розподілена розробка має значні переваги, організаціям, які використовують цей підхід, вкрай важливо забезпечити ефективну комунікаційну структуру, надійні протоколи стандартизації та надійну і безпечну інтеграційну платформу для підтримки цілісного, ефективного і безперебійного середовища розробки програмного забезпечення. [8]

Отже, впровадження розподіленої розробки в рамках архітектури мікросервісів може відкрити величезні переваги, включаючи посилену співпрацю, мінімізацію залежностей, підвищену гнучкість і збагачену арену для інновацій. Це означає перехід від монолітних, обмежувальних стратегій розвитку до гнучкого, стійкого і розподіленого підходу, який краще відповідає динамізму сучасного цифрового бізнес-середовища. Однак для того, щоб повною мірою скористатися його потенційними перевагами,необхідно впроваджувати його вдумливо і стратегічно.

Впровадження продуктово-орієнтованого мислення в архітектуру

мікросервісів означає помітний перехід від традиційних парадигм розробки програмного забезпечення на основі проектів до більш стійких, орієнтованих на користувача і націлених на майбутнє практик. В основі продуктового мислення лежить концепція кожного мікросервісу як окремого продукту зі своїм життєвим циклом. Кожна послуга інкапсулює конкретні бізнес- можливості і пов'язана з окремими командами розробників, які несуть повну відповідальність, починаючи з початкового проектування і розробки, розгортання і закінчуючи безперервним обслуговуванням і розвитком. Це різко контрастує з проектно-орієнтованим мисленням, де після завершення певного завдання чи результату команда часто розпадається, а безперервність не обов'язково підтримується чи є пріоритетною. [9]

Продуктово-орієнтоване мислення, фундаментальне для ефективної організації ландшафту мікросервісів, вносить кілька трансформаційних елементів у культуру розробки програмного забезпечення:

У продуктовому мисленні команди отримують право власності на свій конкретний мікросервісний "продукт". Таке відчуття відповідальності, як правило, сприяє формуванню глибокої відданості команді, оскільки успіх їхнього продукту є прямим віддзеркаленням їхньої професійної майстерності. Така відповідальність спонукає до цілеспрямованого підходу, а не до ефемерної проектно-орієнтованої участі, культивуючи почуття відповідальності та постійної взаємодії для забезпечення довгострокової життєздатності продукту та його ціннісної пропозиції.

Визначальною характеристикою продуктового мислення є прагнення до довгострокового та сталого створення цінності. На відміну від проектного мислення, де завершення проекту означає кінець відповідальності, продуктовий підхід розглядає розробку програмного забезпечення як ітеративний і динамічний процес. Команди розробників віддані своєму "продукту" протягом усього його життєвого циклу і відповідають за розгортання оновлень, удосконалень або змін, щоб задовольнити потреби бізнесу, що змінюються, або відгуки клієнтів.

В основі продуктового мислення лежить незмінна орієнтація на кінцевого користувача - його вподобання, потреби та задоволеність. Таке мислення спонукає команди розробників постійно відстежувати відгуки користувачів, бути в курсі мінливих ринкових тенденцій і гарантувати, що їхній "продукт" відповідатиме теперішнім і майбутнім потребам користувачів. [5] Основна мета виходить за рамки простого створення коду і полягає у вирішенні проблем та створенні цінності для кінцевого користувача.

Продуктовий підхід мотивує команди надавати пріоритет якості, функціональності та зручності програмного забезпечення, а не суворим строкам виконання проекту. Фокус зміщується з "вчасної доставки" на "цінну доставку", де кінцевим мірилом успіху є реальна продуктивність продукту, його здатність задовольняти потреби користувачів та здатність генерувати цінність з часом.

Підсумовуючи, продуктовий підхід у поєднанні з архітектурою мікросервісів перетворює розробку програмного забезпечення на безперервний, ціннісно- орієнтований та орієнтований на користувача процес. Воно переосмислює відносини між командами розробників та їхньою роботою, розвиваючи відповідальність, заохочуючи інновації та забезпечуючи адаптивність до динаміки ринку. Він вдихає життя в програмне забезпечення, перетворюючи його з тимчасового проекту на сталий продукт, розроблений для розвитку та створення цінності в постійно мінливому цифровому ландшафті. В результаті організації можуть підвищити рівень задоволеності клієнтів, зробити інвестиції в програмне забезпечення перспективними, а також сприяти формуванню культури розробки, яка передбачає відповідальність та зацікавленість.

1.1.2 Типи архітектури

Концепція архітектури мікросервісів, яка поступово стає нормою для розробки сучасних програмних систем, являє собою безліч шаблонів і практик проектування, що киплять під її егідою. Вибір архітектурного дизайну

безпосередньо залежить від конкретних потреб організації, рівня зрілості, технологічних можливостей та бізнес-цілей. Розуміння різних типів архітектури мікросервісів є ключовим для ефективного вибору.

Ця модель розробки програмного забезпечення, яка, по суті, формує основу, з якої розвинулася концепція мікросервісів, розглядає програмний додаток як єдину цілісну одиницю. Екземпляри кожного компонента - чи то бази даних, чи то шари користувацького інтерфейсу, чи то серверні додатки - є частиною єдиної програмної сутності, що функціонує та розвивається разом. Економічно ефективна і проста для невеликих додатків, монолітна архітектура забезпечує узгодженість, простоту розгортання і легкість у наскрізному тестуванні. [6]

Однак притаманна їй відсутність модульності та масштабованості може стати вузьким місцем для більших і складніших додатків, оскільки зміни в одному компоненті потенційно можуть порушити роботу всієї системи, а масштабованість обмежена системою в цілому, а не окремими компонентами.

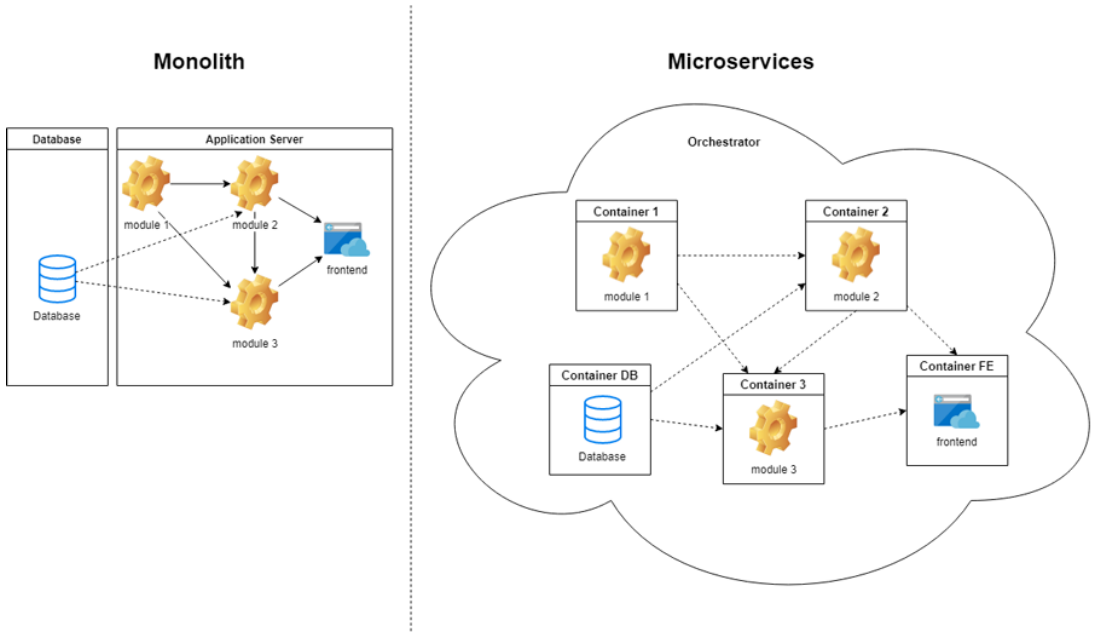

Рисунок 1.1 – Порівняння монолітної і мікросервісної архітектур

Генеалогічний попередник сучасних архітектурних парадигм, таких як

мікросервіси, монолітна архітектура є одним з найпростіших, найбільш лінійних підходів до організації системи. У цій архітектурній моделі програмний додаток сприймається і працює як єдине, неподільне ціле, де всі пов'язані з ним функції - від інтерфейсу користувача до серверних додатків, від управління базами даних до проміжного програмного забезпечення - функціонують в унісон, тісно пов'язані між собою в єдиний програмний блок.

Обтічний форм-фактор Monolithic Architecture створює унікальне згуртоване середовище розробки. Проектування, розробка, тестування та розгортання програми виконуються як єдине ціле на єдиній кодовій базі. Це призводить до більш централізованого і простого процесу, що робить його особливо придатним для відносно скромних додатків з нескладною бізнес-логікою. Додаток, побудований на монолітній архітектурі, розробляється, управляється і розміщується як єдине ціле, створюючи чіткий, прямий шлях від розробки до розгортання, зменшуючи складність міжпроцесної комунікації.

Кожен елемент, кожна функція в монолітній структурі взаємопов'язані та взаємозалежні. Цей симбіотичний зв'язок означає, що зміни в одному компоненті вимагають змін в інших модулях, що вимагає комплексної перестановки та тестування всієї системи щоразу, коли впроваджуються зміни, оновлення або вдосконалення. Така тісна взаємозалежність може суттєво сповільнити цикли розробки та оновлення, впливаючи на життєздатність програмного забезпечення в динамічному бізнес-середовищі, що швидко змінюється.

Природа монолітної архітектури створює специфічні виклики, коли стикається з вимогами до масштабованості. Збільшення потужності програмного забезпечення передбачає масштабування всієї архітектури, незалежно від інтенсивності навантаження на окремі функції або відносних вимог. Це призводить до неефективного використання ресурсів, оскільки масштабування тягне за собою збільшення потужності всіх функцій програми, а не фокусується на конкретних компонентах з високим попитом.

Враховуючи високу взаємозалежність компонентів у монолітній

архітектурі, помилки або аномалії в одному компоненті можуть призвести до ефекту доміно, коли проблеми поширюються на різні частини системи, ставлячи під загрозу загальну стабільність програми. Ізоляція проблем та усунення несправностей може стати складним завданням, оскільки діагностика проблеми одного компонента може ненавмисно вплинути на інші через їх внутрішню залежність. [10]

У динамічно мінливому світі розробки програмного забезпечення монолітна архітектура може не забезпечити бажаної гнучкості, швидкості та оперативності. Впровадження змін, додавання нових функцій, оновлення технологічного стеку займає значну кількість часу через необхідність перебудови всієї системи, що перешкоджає гнучкості та потенційно уповільнює інновації.

Незважаючи на те, що монолітна архітектура є значно менш універсальною та гнучкою порівняно з мікросервісною, вона є логічним, спрощеним та економічно ефективним архітектурним вибором для невеликих додатків або на початковій стадії складних програмних проектів. Це сходинка, яка пропонує цінні уроки під час еволюції до більш сегментованих, масштабованих та гнучких архітектур, таких як сервіс-орієнтована архітектура та архітектура мікросервісів. Таким чином, він вважається життєво важливою сходинкою на шляху до розуміння еволюції архітектури програмного забезпечення у сфері складних розподілених систем. [9]

Як проміжний крок між монолітною архітектурою та архітектурою мікросервісів, SOA сприяє більшій модульності, ніж монолітна архітектура, розбиваючи програмний додаток на окремі сервіси. Ці сервіси, однак, ширші і більш грубозернисті, часто інкапсулюють набір пов'язаних функцій. Вони, як правило, використовуються в межах підприємства і охоплюють такі функції, як виставлення рахунків, логістика або обслуговування клієнтів, вільно пов'язані через API або протоколи обміну повідомленнями, і можуть незалежно масштабуватися і розгортатися. Однак, будучи більшими, ніж одноцільові мікросервіси, вони все ще можуть стикатися зі значною складністю, меншою

гнучкістю і проблемами в масштабуванні та управлінні окремими послугами, специфічними для конкретного бізнесу.

У більш широкому контексті розробки програмного забезпечення вибір певного архітектурного стилю не повинен бути жорстким, одноразовим рішенням. Натомість, часто це поступовий перехід з часом та складністю. Організації часто починають свій шлях з монолітної архітектури, яка забезпечує простіший підхід для менших, менш складних додатків, і мігрують до моделі SOA в міру зростання розміру, складності бізнесу та технологічної зрілості. Зрештою, вони можуть перейти до архітектури мікросервісів, оскільки їх складність і масштаб вимагають подальшої модульності, незалежності та масштабованості. [10]

Вибір правильної архітектури - це балансування між перевагами модульності та масштабованості і проблемами, пов'язаними з ремонтопридатністю та складністю. Ключовим моментом є розуміння того, що кожен тип архітектури є лише засобом для досягнення мети, інструментом, розробленим для ефективного проектування та надання програмного забезпечення, яке генерує високу бізнес-цінність, тісно пов'язане з вимогами користувачів, що змінюються, та витримує випробування часом, масштабом та складністю.

Знаходячи золоту середину між простотою монолітної архітектури та складністю архітектури мікросервісів, сервіс-орієнтована архітектура (SOA) пропонує більш гнучку архітектурну парадигму. SOA структурує додаток як набір сервісів, які чітко визначені, функціонально незалежні і спільно працюють через мережу за допомогою стандартного протоколу зв'язку. Кожна з цих послуг, більша і більш прикладна порівняно з мікросервісами, втілює певну бізнес-операцію або набір пов'язаних функціональних можливостей.

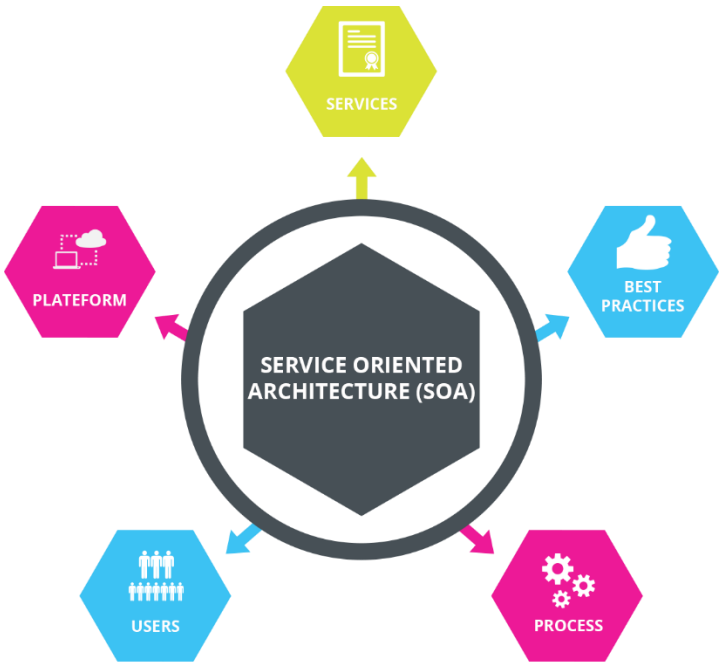

Рисунок 1.2 – SOA архітектура

На відміну від монолітної архітектури, яка сприймає весь додаток як неподільну одиницю, SOA розбиває додаток на незалежні сервіси. Кожен з цих сервісів, принципово більший і автономніший за мікросервіси, інкапсулює окрему бізнес-операцію або набір пов'язаних функціональних можливостей. Вони перетинаються, утворюючи добре інтегровану систему, кожна з яких має повне право власності на свої дані і працює незалежно від інших.

Інтероперабельність лежить в основі SOA, уникаючи жорстких залежностей. Дизайн SOA дозволяє сервісам, навіть якщо вони реалізовані різними мовами програмування або працюють на різних платформах, підтримувати безперебійну та узгоджену взаємодію. [6] Ця взаємодія відбувається за допомогою відкритих стандартів і протоколів, що усуває жорстку залежність і заохочує диверсифікований технологічний ландшафт.

Сервіси в рамках SOA навмисно розроблені таким чином, щоб бути вільно пов'язаними, тобто вони взаємопов'язані, зберігаючи при цьому певний ступінь незалежності. Такий вільний зв'язок гарантує, що зміни, оновлення або збої в одному сервісі не поширюються і не впливають на інші сервіси. Крім

того, SOA сприяє багаторазовому використанню послуг, коли послуга, розроблена для одного процесу, може бути повторно використана в іншому, зменшуючи надмірність і підвищуючи продуктивність.

SOA пропонує значно більш модульну структуру, ніж монолітні системи, що забезпечує підвищену гнучкість і стійкість. Зміни та оновлення можуть бути обмежені конкретним сервісом, який цього потребує, не спричиняючи збоїв у роботі всієї системи. Така модульність підвищує універсальність, надійність та адаптивність системи в умовах мінливого бізнес-ландшафту.

Незважаючи на свої переваги, SOA-системи стикаються з унікальними викликами. Широкий спектр послуг може ускладнити координацію послуг, призвести до потенційних проблем з узгодженістю даних і ускладнити комунікацію в системі. Однак глибоке розуміння принципів SOA та ретельне архітектурне планування можуть допомогти уникнути цих складнощів і скористатися всіма перевагами SOA. [8]

SOA являє собою еволюційний крок у послідовному переході від монолітної до більш модульної архітектури. Вона пропонує гнучкість і модульність, яких часто бракує монолітним системам, і зменшує складність управління, пов'язану з архітектурою мікросервісів. В результаті ми отримуємо збалансоване, гнучке та ефективне програмне рішення, яке може адаптуватися до динамічних потреб бізнесу.

У міру того, як світ переходить від традиційних локальних систем до більш масштабованих та економічно ефективних рішень, парадигма хмарних обчислень стала трансформаційною силою, що формує ландшафт інформаційних технологій. Ця парадигма зміщує інфраструктуру, платформи та програмне забезпечення, які традиційно підтримувалися власними силами, до високомасштабованих, віртуалізованих ресурсів, що надаються як послуга через Інтернет. Розміщення ресурсів у хмарних дата-центрах дозволяє організаціям використовувати і платити саме за те, що їм потрібно, тим самим підвищуючи свою операційну гнучкість і фінансову ефективність.

1.2 Визначення та характеристики хмарних обчислень

Цей розділ має на меті пролити світло на те, що таке хмарні обчислення, сформулювати їх визначення та зрозуміти їх основні характеристики. У ньому підкреслюється, що хмарні обчислення приносять колосальні зміни в тому, як підприємства сприймають і використовують IT-ресурси. [4]

Поява хмарних обчислень зробила революцію у світі технологій, запропонувавши надійну платформу для різноманітних додатків і систем. Однак, щоб повністю зрозуміти можливості та особливості хмарних обчислень, необхідно заглибитися в їхні основні характеристики, які відрізняють їх від традиційних IT-моделей.

Хмарні обчислення дозволяють запитувати ресурси за потребою. Це означає, що вони надають користувачам однозначні засоби для управління своїми обчислювальними ресурсами без будь-якого втручання людини з боку постачальника хмарних послуг. Ці ресурси можуть включати обчислювальні потужності, сховище або додаткову пропускну здатність мережі.

Оскільки ці сервіси базуються на хмарних технологіях, доступ до них можна отримати через мережу за допомогою стандартних механізмів, що робить їх легко доступними для різних пристроїв, таких як мобільні телефони, ноутбуки, планшети або виділені робочі станції. Ця особливість забезпечує доступність і гнучкість з точки зору робочих місць і сприяє використанню гетерогенних платформ.

Згідно з цією характеристикою, модель хмарних обчислень розширює можливості декількох клієнтів для спільного використання спільного пулу ресурсів. Це забезпечує ефективне використання ресурсів, дозволяючи клієнтам отримати максимальну віддачу від своїх інвестицій. Ресурси можуть динамічно призначатися або перепризначатися на основі попиту споживачів, часто без точного контролю з боку клієнта над точним розташуванням наданих ресурсів. [5]

Однією з головних переваг хмарних обчислень є їхня еластичність, що

забезпечує можливість швидкого та автоматичного масштабування ресурсів у відповідь на зміни робочого навантаження. Ця характеристика дозволяє хмарі гнучко реагувати на мінливі вимоги, забезпечуючи оптимальну продуктивність при мінімізації витрат.

У хмарних обчисленнях використання ресурсів ретельно відстежується, контролюється та відображається у звітах, що забезпечує прозорість як для постачальника, так і для споживача. Така можливість обліку уможливлює модель "оплата за фактом", що є особливо привабливою для компаній, які прагнуть привести витрати на IT у відповідність до фактичного попиту.

Дійсно, ці основні характеристики хмарних обчислень забезпечують міцний фундамент для їхнього успіху. Вони являють собою парадигми, які забезпечують перехід від традиційного до сучасного, демонструючи потенціал технології у застосуванні до бізнес-моделей та операційних інструментів.

1.3 Типи хмарних сервісів

Заглиблюючись у специфіку спектру хмарних обчислень, цей модуль ретельно вивчає різні типи хмарних сервісів, а саме: інфраструктуру як послугу (IaaS), платформу як послугу (PaaS) та програмне забезпечення як послугу (SaaS). Він проливає світло на відмінні аспекти та випадки використання кожної з цих моделей послуг. [2]

Незважаючи на складнощі, які ховаються за її концепцією, основна пропозиція парадигми хмарних обчислень створює середовище з майже безмежною масштабованістю, структурою витрат за принципом "платити по мірі використання" та доступністю ресурсів на вимогу. Це підхід, який за своєю суттю розроблений для задоволення динамічних вимог та розвитку сучасного бізнесу, діючи як еліксир для вирішення його найнагальніших проблем.

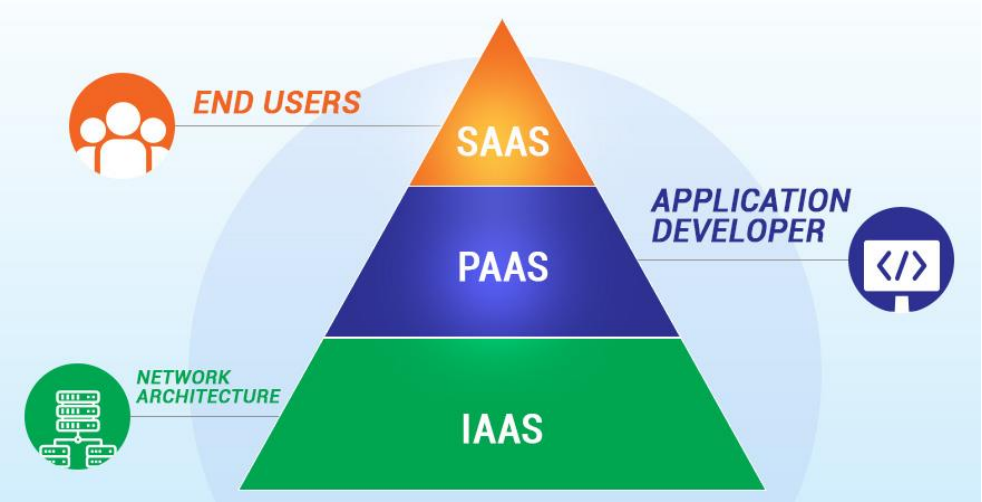

Рисунок 1.3 – Типи хмарних сервісів

Парадигма хмарних обчислень охоплює низку сервісів, які відповідають різним вимогам користувачів, від індивідуальних розробників до глобальних підприємств. Ці послуги зазвичай поділяються на три категорії: інфраструктура як послуга (IaaS), платформа як послуга (PaaS) і програмне забезпечення як послуга (SaaS). Ці категорії суттєво відрізняються за обсягом контролю, який мають користувачі, та обсягом необхідних накладних витрат на управління. [2]

1.3.1 Інфраструктура як послуга (IaaS)

IaaS - це найбільш гнучка категорія хмарних сервісів, що надає користувачам повністю контрольоване, конфігуроване та автоматизоване середовище. Ця модель надає інфраструктуру, таку як віртуальні машини, та інші ресурси, такі як мережі та сховища, з пулу апаратних ресурсів. Це вигідно для бізнесу, оскільки зменшує капітальні витрати та складність придбання, управління та модернізації фізичних серверів.

Інфраструктура як послуга, часто відома як IaaS, формує фундаментальний рівень у моделі хмарних обчислень. IaaS надає бізнесу можливість орендувати IT-інфраструктуру - сервери, віртуальні машини (ВМ), сховища, мережі, операційні системи - у хмарного провайдера на умовах

оплати за фактом використання.

Сприяючи створенню гнучкого середовища, IaaS спрощує операційні складнощі, усуваючи необхідність обслуговування та управління фізичними центрами обробки даних і серверами, тим самим зменшуючи витрати на обладнання, технічне обслуговування та персонал, забезпечуючи при цьому масштабованість, високу продуктивність та безперервність бізнесу.

IaaS усуває обмеження, що накладаються фізичним обладнанням, надаючи ресурси за потребою. Це дає організаціям можливість швидко масштабувати ресурси, щоб впоратися зі сплесками трафіку, і зменшувати їх, коли попит падає, що сприяє ефективному використанню ресурсів. [9]

Розгортання моделі IaaS дозволяє уникнути значних капітальних витрат на встановлення та обслуговування апаратного та програмного забезпечення. Натомість вони замінюються керованим і постійно контрольованим середовищем, яке працює на основі гнучкої підписки з оплатою по мірі використання.

Хоча провайдер хмарних послуг підтримує інфраструктуру, користувачі все одно мають повний контроль над конфігурацією інфраструктури, а також несуть відповідальність за запуск додатків, дані та проміжне програмне забезпечення.

Завдяки IaaS плани забезпечення високої доступності, аварійного відновлення та безперервності бізнесу стають доступнішими та простішими в реалізації, оскільки дані можна копіювати та реплікувати на декількох майданчиках за допомогою постачальника IaaS. [5]

Таким чином, впровадження послуг IaaS дозволяє організаціям підвищити функціональність, зменшити витрати та оптимізувати розподіл ресурсів, досягаючи раніше віддалених цілей продуктивності.

1.3.2 Платформа як послуга (PaaS)

PaaS призначена для підтримки повного життєвого циклу веб-додатків:

створення, тестування, розгортання, управління та оновлення. Ця платформа забезпечує середовище, в якому користувачі можуть розробляти, запускати та керувати додатками без складнощів, пов'язаних з побудовою та підтримкою інфраструктури. Це ідеальний інструмент для розробників, що дозволяє їм зосередитися на проектуванні та створенні додатків, а не на завданнях управління інфраструктурою, таких як планування потужностей, обслуговування програмного забезпечення, виправлення та інші завдання з управління IT.

Платформа як послуга (PaaS) - це інноваційна хмарна пропозиція, яка надає повне середовище для розробки та розгортання в хмарі. Завдяки PaaS розробники можуть спільно створювати, тестувати, запускати та керувати додатками, не турбуючись про базову інфраструктуру. Мета PaaS - абстрагуватися від складнощів управління апаратним та програмним забезпеченням, дозволяючи розробникам зосередитися на процесі кодування та проектування.

PaaS прискорює розробку, тестування та розгортання додатків, надаючи зрозумілу та просту у використанні платформу. Вона включає в себе інструменти розробки, управління базами даних, послуги бізнес-аналітики (BI) та багато іншого безпосередньо в хмарі. [8]

Завдяки тому, що платформа виконує більшість завдань системного адміністрування, розробники можуть зосередитися на написанні коду та вдосконаленні функцій додатків. Такі завдання, як налаштування баз даних, серверів або контейнерів, систем зберігання та мережевих ресурсів, абстрагуються від розробників, що прискорює цикл розробки.

Рішення PaaS за своєю суттю є масштабованими, що дозволяє компаніям збільшувати або зменшувати ресурси відповідно до їхніх потреб. Гнучкість в управлінні ресурсами дозволяє компаніям підтримувати оптимальний баланс між економічною ефективністю та продуктивністю.

Оскільки PaaS абстрагується від дріб'язкових деталей управління інфраструктурою та платформою, організації можуть досягти подальшого

зниження витрат. Моделі ціноутворення, як правило, базуються на використанні, гарантуючи, що бізнес платить лише за ті ресурси, які він фактично споживає. [1]

Рішення PaaS часто використовують багатокористувацьку архітектуру, що означає, що кілька користувачів можуть виконувати свої програми та робочі навантаження на одній інфраструктурі. Така модель спільної інфраструктури зменшує операційні витрати та підвищує оптимізацію ресурсів.

PaaS слугує потужним каталізатором, який прискорює інтеграцію додатків у хмару. Пом'якшуючи перешкоди в процесі розробки, вона забезпечує плавний перехід від традиційних методів розробки програмного забезпечення до сучасної, хмаро орієнтованої моделі. Тому розуміння PaaS залишається життєво важливим для будь-якої організації, яка хоче скористатися перевагами гнучкості, ефективності та масштабованості, що надаються хмарою. [1]

1.3.3 Програмне забезпечення як послуга (SaaS)

SaaS дозволяє користувачам підключатися до хмарних додатків через Інтернет. Часто це веб-програми, які виконують широкий спектр завдань. Хмарні провайдери керують інфраструктурою та платформами, на яких працюють додатки, що дозволяє компаніям легко відмовитися від дорогих інсталяцій та обслуговування програмного забезпечення власними силами.

Вивчення цих різних рівнів послуг хмарних обчислень має важливе значення для розуміння значних переваг, таких як скорочення витрат, масштабованість і варіативність потужностей, які можуть запропонувати хмарні технології. Вибравши відповідні хмарні сервіси, організації можуть зосередитися на своїх основних операціях, підтвердити бізнес-цілі та відкрити нові можливості для зростання та інновацій. Отже, розуміння того, який тип сервісу найкраще відповідає бізнес-вимогам, є ключовим рішенням на шляху міграції до хмарних технологій.

Програмне забезпечення як послуга (SaaS) - це верхній рівень стеку

хмарних обчислень. Це модель розповсюдження програмного забезпечення, за якої додатки розміщуються у постачальника послуг і стають доступними для користувачів через Інтернет. По суті, замість того, щоб встановлювати і підтримувати програмне забезпечення, користувач просто отримує доступ до нього через Інтернет, що звільняє його від складних обов'язків з управління програмним і апаратним забезпеченням.

Оскільки додатки SaaS надаються через Інтернет, користувачам для доступу до них потрібен лише браузер і підключення до Інтернету. Така простота доступу забезпечує більшу гнучкість працівників і сприяє віддаленій роботі.

SaaS може забезпечити значну економію, оскільки усуває початкові витрати на придбання/інсталяцію, поточні витрати на обслуговування та оновлення. Оскільки додатки готові до використання, це також значно скорочує час впровадження та розгортання.

Провайдери SaaS зазвичай пропонують багато варіантів підписки та гнучкість у зміні підписників за потреби. Крім того, більшість SaaS-додатків розроблені таким чином, щоб підтримувати певний ступінь кастомізації способу їхнього функціонування, часто за допомогою конфігурації налаштувань програми або використання програмних інтерфейсів (API).

За допомогою SaaS постачальник послуг керує всіма оновленнями та модернізаціями, пропонуючи нові випуски, які включають інноваційні та вдосконалені функції. Це звільняє користувача від необхідності та відповідальності за обслуговування програмного забезпечення. [2]

Провайдери SaaS відповідають за безпеку програмного забезпечення та його відповідність нормативним вимогам, що дозволяє компаніям витрачати менше часу на турботи про безпеку і зосередитися на своїй основній діяльності.

Отже, модель "Програмне забезпечення як послуга" (SaaS) пропонує значні переваги, зокрема доступ до Інтернету з будь-якої точки світу, надійні заходи безпеки та аутсорсинг IT-відповідальності. Це дозволяє організаціям оптимізувати свої бізнес-процеси і зосередитися на основних бізнес-функціях,

що підкреслює важливість розуміння і використання пропозицій SaaS в сучасному бізнес-середовищі.

2 РОЛЬ МІКРОСЕРВІСІВ У ХМАРНИХ СЕРЕДОВИЩАХ ТА ДОСЛІДЖЕННЯ ПОТОЧНИХ ТЕНДЕНЦІЙ

Мікросервіси, інкапсульовані як автономні одиниці, відіграють ключову роль у парадигмі хмарних обчислень, забезпечуючи деталізацію, гнучкість та універсальність. Впровадження архітектури мікросервісів у хмарному середовищі може призвести до більш стійкого, масштабованого і швидкого циклу розробки програмного забезпечення, що робить її ідеальним підходом для сучасної розробки додатків.

2.1 Переваги розгортання мікросервісів у хмарі

При розгортанні в хмарі мікросервіси отримують численні переваги від широких можливостей та інфраструктури хмарних обчислень.

2.1.1 Масштабованість

Хмарне середовище надає мікросервісам можливість самостійно масштабуватися відповідно до їхніх індивідуальних потреб. Наприклад, якщо певний сервіс користується високим попитом, то лише цей сервіс можна масштабувати, не змінюючи інші, заощаджуючи ресурси та зменшуючи витрати.

Хмара сприяє необмеженій масштабованості, а в поєднанні з архітектурою мікросервісів дозволяє здійснювати дрібнозернисте масштабування окремих сервісів. Це означає, що сервісам, які користуються більшим попитом, можна виділити більше ресурсів, уникаючи необхідності масштабування всієї програми, а отже, заощаджуючи витрати.

Масштабованість є однією з найважливіших переваг впровадження мікросервісів у хмарному середовищі. Масштабованість - це здатність системи

справлятися зі зростаючим навантаженням шляхом додавання ресурсів, іноді автоматично, не впливаючи на продуктивність системи та не спричиняючи перебоїв у роботі сервісу. По суті, масштабованість дозволяє додатку ефективно адаптуватися до змін попиту. [8]

Коли справа доходить до розгортання мікросервісів у хмарі, масштабованість набуває особливої переваги, оскільки вона дозволяє окремим сервісам масштабуватися незалежно. Цю особливість часто називають "масштабованістю для конкретного сервісу" і вона є характеристикою, притаманною мікросервісам. Це відрізняється від традиційних монолітних додатків, де збільшення попиту змушує масштабувати весь додаток, що є ресурсоємним і трудомістким процесом.

В архітектурі мікросервісів, якщо на певний сервіс зростає попит, масштабувати можна лише ресурси цього сервісу. Це може означати додавання більшої кількості екземплярів певного сервісу або виділення більшої обчислювальної потужності для сервісу, залежно від можливостей хмарного провайдера. Відповідно, коли попит зменшиться, ресурси можна буде масштабувати назад. [8]

Гранульована масштабованість хмари на вимогу забезпечує ефективне використання ресурсів та економічну ефективність. Це надійне рішення для роботи зі змінним робочим навантаженням і піковими навантаженнями, що сприяє безперебійній та безперебійній роботі користувачів і, таким чином, забезпечує задоволеність клієнтів. Тому розуміння переваг масштабованості при розгортанні мікросервісів у хмарі є фундаментальним для оптимізації та ефективного управління ресурсами.

2.1.2 Стійкість

Мікросервіси в хмарі підвищують відмовостійкість системи. Якщо сервіс виходить з ладу, це впливає лише на функціональність, що надається цим сервісом, а не на весь додаток. Крім того, хмарні платформи часто надають

автоматизовані інструменти для відновлення роботи сервісу та скорочення часу простою.

Вбудована в хмару надмірність у поєднанні з ізольованим характером мікросервісів гарантує, що збій одного сервісу не призведе до зупинки всієї програми, таким чином підвищуючи надійність і стійкість системи. [3]

Відмовостійкість - ще одна вагома перевага розгортання мікросервісів у хмарному середовищі. Відмовостійкість у контексті програмного забезпечення означає здатність системи швидко відновлюватися після збоїв і продовжувати виконувати свої функції. Це вирішальний фактор у забезпеченні доступності та надійності системи, особливо в умовах неочікуваних збоїв.

В архітектурі мікросервісів, розгорнутих у хмарі, відмовостійкість підкреслюється кількома способами.

По-перше, мікросервіси працюють як незалежні процеси - збій в одному сервісі не призводить безпосередньо до відмови інших. Така ізоляція запобігає швидкому поширенню несправностей між сервісами, на відміну від монолітних архітектур, де одна точка відмови може вивести з ладу всю систему. [7]

По-друге, хмарні платформи надають вбудовані інфраструктурні можливості для управління та пом'якшення наслідків збоїв. Вони часто мають автоматизоване масштабування, процедури обходу відмов, сервіси резервного копіювання та протоколи відновлення після збоїв. Таким чином, якщо мікросервіс з якихось причин виходить з ладу, хмарна платформа може автоматично створити новий екземпляр сервісу майже миттєво, підтримуючи функціональність додатків з мінімальними перебоями.

По-третє, з мікросервісами ви можете використовувати стратегію розгортання декількох екземплярів одного і того ж сервісу. У разі відмови одного екземпляра сервісу, інші екземпляри зможуть продовжити обробку запитів. Це сприяє високій доступності сервісів.

Нарешті, для досягнення відмовостійкості в архітектурі мікросервісів часто використовуються такі патерни, як автоматичні вимикачі, повторні спроби, резервні копії та тайм-аути. Ці патерни можуть запобігти каскадному

поширенню збою одного сервісу на інші сервіси і допомогти зберегти загальну цілісність програми, навіть у випадку помилки.

По суті, відмовостійкість в контексті хмарних мікросервісів - це розробка та впровадження стратегій, спрямованих на передбачення, подолання, відновлення та адаптацію до будь-якого виду системних збоїв. Це гарантує надання безперебійних послуг користувачам і підвищує довіру до розгорнутих додатків. Таким чином, розуміння та підвищення відмовостійкості є ключовим аспектом оптимізації та інтеграції мікросервісів у хмарне середовище. [9]

2.1.3 Свобода вибору технології

Завдяки децентралізованій природі мікросервісів розробники мають більше свободи у виборі найкращої технології для кожного сервісу. Такий поліглотський підхід дозволяє використовувати різні мови програмування, технології зберігання даних або програмні середовища для різних сервісів.

Хмарне середовище в поєднанні з архітектурою мікросервісів надає свободу вибору оптимального технологічного стеку (мови програмування, бази даних, серверні середовища) для кожного окремого сервісу. Ця свобода допомагає у виборі найбільш підходящих інструментів для кожного завдання, що потенційно призводить до більшої ефективності та результативності послуг.

Одним з найбільш вигідних аспектів розгортання мікросервісів у хмарі є свобода вибору найбільш підходящої технології для кожного мікросервісу. Враховуючи, що кожен мікросервіс стоїть окремо і працює незалежно, розробники не обмежені єдиним технологічним стеком, як у традиційних монолітних архітектурах. Таким чином, мікросервісна архітектура в хмарному середовищі дозволяє використовувати декілька мов програмування, баз даних і фреймворків в одному додатку.

Ця свобода вибору технологій має кілька наслідків і переваг.

По-перше, це дає змогу розробникам обрати найефективніший стек технологій для функціональності, яку пропонує кожен окремий сервіс.

Наприклад, сервіс з високими обчислювальними вимогами може бути розроблений на високопродуктивній мові, такій як C або Go, в той час як сервіс, що маніпулює текстом, може скористатися перевагами потужних бібліотек обробки тексту Python.

По-друге, така гнучкість сприяє підвищенню продуктивності та залученості команд розробників. Розробники можуть використовувати свій досвід у різних технологіях, що призводить до більш ефективних та результативних процесів розробки.

По-третє, свобода технологічного вибору сприяє еволюційному розвитку мікросервісів. Зі зміною бізнес-потреб і розвитком технологій можна створювати нові сервіси з використанням новітніх технологій. Існуючі сервіси можуть бути переписані на різні технологічні стеки, якщо це необхідно, не впливаючи на загальну роботу програми. [10]

Нарешті, це дозволяє розробникам експериментувати та впроваджувати інновації з новими технологіями в невеликому масштабі в межах одного сервісу, перш ніж вирішувати, чи варто впроваджувати ці технології ширше.

З точки зору теми диплому свобода вибору технології є критично важливим фактором для оптимізації продуктивності, ефективності та адаптивності мікросервісів та хмарних додатків в цілому. Правильно обравши технологію, розробники можуть створювати елегантні, ефективні та високофункціональні сервіси, які відповідають унікальним потребам конкретної бізнес-сфери.

2.1.4 Швидкі інновації та швидкий вихід на ринок

Інтеграція мікросервісів у хмарне середовище дає можливість командам працювати над різними сервісами одночасно, що сприяє пришвидшенню циклів розробки та розгортання. Це сприяє швидким ітераціям, інноваціям та скороченню часу виходу на ринок.

2.1.5 Простота обслуговування та оновлення

Дезінтегрована природа мікросервісів та широкі можливості хмарних інструментів і автоматизації роблять обслуговування та оновлення додатків значно простішими та ефективнішими. Окремі сервіси можна оновлювати незалежно, не впливаючи на всю програму, що зменшує час простою. [3]

Модель хмарних обчислень, що працює на вимогу та оплачується по мірі використання, разом із масштабованістю мікросервісів, може призвести до значної економії коштів. Підприємствам потрібно платити лише за ті ресурси, які вони використовують, забезпечуючи ефективне використання ресурсів та зменшуючи непотрібні витрати.

Використання цих переваг вимагає цілісного розуміння як мікросервісів, так і хмарних концепцій. Разом вони сприяють підвищенню операційної ефективності, стійкості та гнучкості в постійно мінливому середовищі розробки програмного забезпечення.

2.2 Проблеми розгортання мікросервісів у хмарі

Незважаючи на численні переваги, розгортання мікросервісів у хмарі не позбавлене проблем.

Оскільки мікросервіси потенційно можуть використовувати різні бази даних, підтримка узгодженості даних у всій розподіленій системі стає проблемою.

Підтримка узгодженості даних між модульованими сервісами є складним завданням в архітектурі мікросервісів. Кожен мікросервіс може мати власну базу даних, щоб забезпечити відокремлення від інших сервісів, але це означає, що спільні дані між сервісами можуть стати неузгодженими. Необхідно застосовувати стратегії, щоб забезпечити узгодженість у всій системі. [8]

При розгортанні мікросервісів у хмарному середовищі забезпечення

узгодженості даних стає значним викликом. Кожен мікросервіс має власну унікальну базу даних для збереження своєї незалежності. Хоча така інкапсуляція є ключем до гнучкості та стійкості архітектури мікросервісів, вона також створює перешкоди для синхронізації даних між різними сервісами.

У монолітній архітектурі часто існує єдина, уніфікована база даних, з якою взаємодіють усі модулі. Такий дизайн по суті забезпечує узгодженість: зміна даних одразу ж відображається у всіх частинах програми. Однак в архітектурі мікросервісів узгодженість даних не є автоматичною через розподілену природу сховищ даних.

Розробники повинні впроваджувати стратегії, які гарантують, що всі сервіси мають узгоджене уявлення про дані. Це стає критично важливим, коли бізнес- процес охоплює кілька сервісів, вимагаючи від них співпраці та коректного обміну даними, що може бути складним завданням.

Два найпоширеніші рішення цієї проблеми - це моделі Eventual Consistency та Transactional Consistency. У моделі Eventual Consistency деякі сервіси можуть бачити застарілі дані на певний час, розуміючи, що з часом вони будуть оновлені в повному обсязі. З іншого боку, модель транзакційної узгодженості вимагає, щоб кожна транзакція між кількома сервісами оброблялася як одна атомарна одиниця, забезпечуючи узгодженість даних у будь-який момент часу.

У хмарному середовищі мікросервісів проблему узгодженості даних не можна ігнорувати, якщо ви прагнете до цілісності та надійності всієї системи. Це також є життєво важливим фактором при прийнятті методів інтеграції та оптимізації підходу мікросервісів. Ретельне планування, розуміння конкретних потреб організації та обґрунтована стратегія управління узгодженістю даних мають важливе значення для успішного подолання цього виклику.

Архітектура мікросервісів додає операційної складності, оскільки з'являється більше незалежних компонентів, якими потрібно керувати. Це проявляється у таких проблемах, як міжсервісний зв'язок, виявлення сервісів, відмовостійкість та розподілене зберігання даних.

Хоча розгортання мікросервісів у хмарному середовищі надає безліч переваг для бізнесу, воно не обходиться без перешкод. Однією з помітних проблем є додаткова операційна складність, пов'язана з управлінням кількома слабко пов'язаними незалежними сервісами.

Мікросервіси збільшують операційну складність порівняно з монолітними архітектурами. Управління численними сервісами, кожен з яких може бути написаний іншою мовою і працювати на різних машинах, може бути складним завданням. Виклики пов'язані з міжсервісною комунікацією, автоматизацією, розгортанням, моніторингом, тестуванням та усуненням несправностей. [6]

Обробка численних сервісів окремо може бути більш об'ємною і складною у порівнянні з традиційною монолітною архітектурою. Кожен мікросервіс, який може бути написаний іншою мовою і розгорнутий в різних середовищах, вимагає індивідуального підходу при розгортанні, масштабуванні, моніторингу та ізоляції несправностей. Така сегментація може значно підвищити операційну складність.

Крім того, залежність між сервісами може створювати труднощі, особливо при виконанні таких завдань, як впровадження змін, оновлення або налагодження системи. Кожна зміна, зроблена в одному сервісі, може вплинути на інші взаємопов'язані сервіси, що вимагає ретельної координації та постійної комунікації з різними командами.

Міжсервісний зв'язок через мережу, яка бажано повинна бути безпечною, швидкою, послідовною і масштабованою, додає ще один рівень складності. Протоколи, формати даних і ланцюжки запитів повинні підтримуватися належним чином, щоб забезпечити оптимальний зв'язок, що може бути складним завданням.

Зменшення цієї операційної складності має вирішальне значення. Методи, що використовуються для мінімізації складності, можуть включати впровадження стандартизованих протоколів для міжсервісного зв'язку, застосування автоматизації для рутинних операційних завдань, використання

централізованого ведення журналів, а також використання надійних інструментів моніторингу та візуалізації для відстеження всієї системи.

Вирішення проблеми операційної складності має вирішальне значення для забезпечення успішної інтеграції та оптимізації мікросервісів у хмарному середовищі, підвищуючи таким чином загальну надійність, доступність і ремонтопридатність системи. Вдумливий підхід до зменшення операційної складності може дозволити організації отримати всі переваги хмарної архітектури мікросервісів.

Оскільки мікросервіси розподілені і взаємодіють один з одним через мережу, затримка в мережі є проблемою. Хоча хмарне середовище пропонує високошвидкісні мережі, затримка все одно може впливати на час відгуку сервісу, особливо коли в обробці одного запиту беруть участь кілька сервісів.

Важливою проблемою при розгортанні мікросервісів у хмарному середовищі є управління мережевими затримками. Оскільки мікросервіси взаємодіють один з одним через мережу, час, необхідний для передачі даних від одного сервісу до іншого, стає важливим фактором, який може вплинути на загальну продуктивність системи. [5]

В архітектурі мікросервісів процеси, які раніше виконувалися в рамках одного монолітного додатку, тепер розподілені між різними сервісами, часто працюючими на різних машинах або навіть в різних географічних точках хмари. Такий розподіл процесів природно призводить до зростання важливості мережевої затримки, оскільки для виконання однієї бізнес-операції дані повинні переміщатися туди і назад між цими сервісами.

Хоча кожен окремий запит може зайняти мінімальну кількість часу, кумулятивний ефект цих численних запитів може сповільнити роботу системи, впливаючи на час відгуку та якість обслуговування користувачів. Проблема стає критичною, коли в обробці одного запиту беруть участь кілька сервісів, що призводить до так званої проблеми "балакучості мікросервісів".

Проблема подолання мережевих затримок вимагає відвертого розгляду та продуманих рішень. Використання асинхронного зв'язку та буферів

повідомлень, консолідація пов'язаних операцій або кешування можуть допомогти зменшити витрати на міжсервісний зв'язок.Ретельне тестування продуктивності, моніторинг шаблонів мережевого трафіку та розуміння поведінки програми в різних мережевих умовах також можуть допомогти розробникам оптимізувати програми для вирішення проблем затримок у мережі.

Хоча затримка є невід'ємною частиною мережевого зв'язку, можна зменшити її вплив і зберегти переваги розгортання мікросервісів у хмарному середовищі за допомогою стратегій, які вирішують цю життєво важливу проблему. Підтримка оптимальної продуктивності системи, незважаючи на проблеми затримок, має вирішальне значення для надання якісної та швидкої відповіді користувачам, що є важливою вимогою в конкурентному цифровому середовищі.

При розгортанні мікросервісів у хмарі, безпека міжсервісна та мережева безпека вимагає ретельного розгляду та проектування. Кожен мікросервіс має більшу вразливу область, що потенційно пропонує зловмисникам більше потенційних вразливостей для використання. [6]

Глибоке розуміння симбіотичного зв'язку між мікросервісами та хмарою є життєво важливим для використання їхнього спільного потенціалу та уникнення потенційних пасток. Це важливий концептуальний крок на шляху до інтеграції та оптимізації мікросервісів у хмарному середовищі.

Цілісне розуміння мікросервісів у хмарному середовищі буде неповним без урахування поточних і нових тенденцій у цій галузі. Ці тенденції формують спосіб, у який підприємства підходять до інтеграції та оптимізації мікросервісів та виконують їх.

2.2 Огляд нових тенденцій

У цьому розділі ми розглянемо три основні тенденції, які відіграють важливу роль у формуванні сучасного ландшафту хмарних мікросервісів.

Кожна тенденція вказує на сферу змін і розвитку, яка впливає на те, як бізнес і команди розробників підходять до використання мікросервісів.

2.2.1 Швидка розробка та тестування

Однією з основних тенденцій є перехід до швидкої та безперервної розробки та тестування, що стало можливим завдяки децентралізації кодових баз та можливостям автономного розгортання архітектури мікросервісів.

2.2.2 Більший фокус на користувацькому досвіді

Покращення користувацького досвіду стало центральним питанням, оскільки бізнес переходить на клієнтоорієнтований підхід. Необхідність швидко і надійно надавати високоякісний користувацький досвід впливає на те, як розробляються і розгортаються мікросервіси в хмарі.

2.2.3 Посилення заходів безпеки

Оскільки мікросервіси мають більшу площу поверхні для потенційних атак, акцент на впровадженні більш жорстких заходів безпеки в рамках архітектури та всієї діяльності є помітною тенденцією.

2.3 Потенційний вплив та наслідки нових тенденцій

Оцінка потенційних наслідків цих тенденцій зобов'язує нас розуміти їхній ширший вплив на IT-сектор, очікування щодо продуктивності та залучення клієнтів.

2.3.1 Зміна робочих ролей в IT-секторі

У світлі цих тенденцій робочі ролі в IT-секторі змінюються, вимагаючи нових навичок і сфер знань.

2.3.2 Підвищені очікування щодо ефективності

Очікування щодо продуктивності зростають, оскільки компанії відчувають все більший тиск з боку необхідності швидко та ефективно надавати своїм користувачам високопродуктивні, надійні та безпечні послуги.

2.3.3 Підвищені очікування клієнтів

Очікування клієнтів зростають завдяки широким можливостям кастомізації, персоналізованому досвіду та надійним заходам безпеки, які стають стандартами.

Метою вивчення цих тенденцій є усвідомлення ширшого контексту, в якому працюють мікросервіси в хмарному середовищі. Розуміючи сучасні тенденції та їхні наслідки, ми краще позиціонуємо себе для спрощення інтеграції, оптимізації операцій та передбачення майбутніх напрямків у цій сфері, що швидко розвивається.

Цей розділ містить огляд сучасних тенденцій, які формують ландшафт мікросервісів у хмарному середовищі. Ці тенденції не лише впливають на технології, що використовуються, але й змінюють методології та принципи, якими керуються при розробці та управлінні мікросервісами. [9]

Перехід до більш гнучких практик продовжує набирати обертів в IT-секторі. Підхід безперервної інтеграції та безперервного розгортання (CI/CD) набуває все більшої популярності завдяки потенціалу, який він пропонує для швидшого та ефективнішого життєвого циклу програмного забезпечення. Завдяки модульності мікросервісів, оновлення або зміни можуть бути виконані на ізольованих елементах, не порушуючи роботу всієї системи, що призводить

до частіших випусків і швидкого реагування на запити на зміни або виявлені проблеми.

Поширення цифрових послуг та загострення конкуренції на ринку змушують компанії приділяти більше уваги користувацькому досвіду, щоб відрізнятись від конкурентів. Технології мікросервісів у поєднанні з хмарними можливостями дозволяють компаніям пропонувати своїм користувачаміндивідуальний, динамічний та захоплюючий досвід.

Враховуючи розширену площу потенційних атак через розподілену природу мікросервісів, впровадження передових заходів безпеки стало життєвою необхідністю. Нові тенденції включають в себе комплексні проекти безпеки, які враховують кожен аспект системи, від рівня даних до мережевого зв'язку і між сервісами. [3]

Вивчення цих нових тенденцій допомагає нам залишатися в курсі подій, готуватися до змін і використовувати можливості, які вони приносять. Це дозволяє нам вести дискусію про методи інтеграції та оптимізації мікросервісів у контексті найновіших, найцікавіших та найвпливовіших розробок у цій галузі.

В рамках сучасних тенденцій розвитку мікросервісів та хмарних додатків все більше уваги приділяється необхідності швидкої розробки та тестування. У цифрову епоху швидкість є конкурентною перевагою, і організації постійно шукають шляхи прискорення циклів розробки.

Швидка розробка вказує на здатність проектувати, створювати прототипи та впроваджувати нові функції або послуги в обмежений часовий проміжок.

Завдяки мікросервісам команди мають перевагу зосереджуватися на одній послузі за раз, що дозволяє їм впроваджувати нові функції або модифікувати існуючі паралельно, не впливаючи на решту системи. У хмарі команди можуть використовувати величезні ресурси для прискорення процесів кодування, тестування та розгортання.

Тестування, яке є не менш важливим, також зазнало значних змін. Філософія "Тестуй раніше і частіше" набирає обертів, переходячи від

традиційних етапів тестування до постійної, інтегрованої частини циклу розробки. Автоматизоване тестування, що є частиною конвеєра безперервної інтеграції та безперервного розгортання (CI/CD), гарантує, що кожна зміна буде перевірена, щоб виявити і виправити потенційні проблеми на ранніх стадіях. [6]

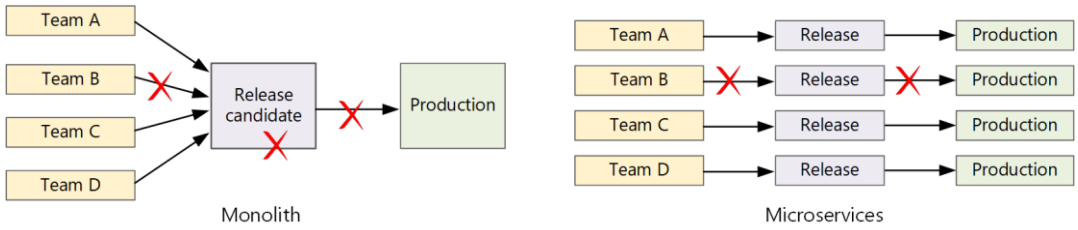

Рисунок 2.1 – Процес CI/CD для мікосервісних і мнолітних архітектур

У хмарних мікросервісах такий підхід означає значне підвищення продуктивності. Розробники можуть створювати і тестувати невеликі фрагменти функціональності та швидко переміщувати їх у хмару. Автоматизовані інструменти тестування можуть переглядати цю нову функціональність, негайно відзначаючи будь-які збої, що дозволяє розробникам вирішувати ці проблеми, поки завдання ще свіже в їхній пам'яті.

Перевагою такої практики є не лише скорочення часу виходу на ринок, але й покращення якості програмного забезпечення, оскільки проблеми швидко виявляються та виправляються. Отже, швидка розробка та тестування є важливими тенденціями, що пропонують організаціям гнучкість та ефективність, необхідні для швидкої адаптації та інновацій у сучасному конкурентному середовищі.

Оскільки очікування клієнтів постійно зростають, тенденція до приділення більшої уваги створенню видатного користувацького досвіду (UX) стає все більш помітною. Гнучкість і масштабованість, які забезпечують мікросервіси та хмарна парадигма, відіграють важливу роль у підтримці цього напрямку.

У цьому контексті користувацький досвід стає важливим фактором, що

визначає цінність бізнесу та задоволеність клієнтів. Мікросервіси дозволяють цілеспрямовано вдосконалювати окремі елементи продукту без необхідності перебудови всієї системи. Як наслідок, розробники можуть зосередитися на оптимізації окремих аспектів користувацького досвіду, які пов'язані з конкретними бізнес-результатами, такими як коефіцієнт конверсії, утримання користувачів або швидкість реагування системи. [4]

Хмарні мікросервіси ще більше посилюють цю тенденцію, пропонуючи необхідну еластичність інфраструктури для швидкого тестування, отримання зворотного зв'язку та ітерацій над UX-дизайном. Вони дають змогу бізнесу оперативно налаштовувати та розгортати оновлення на основі відгуків користувачів, забезпечуючи кращий та постійно вдосконалюваний досвід для кінцевих користувачів.

Крім того, мікросервіси пропонують розподіл обов'язків. Команди можуть зосередитися на різних аспектах користувацького досвіду, клієнтських подорожах або точках контакту і працювати над ними незалежно. Одна група може вдосконалювати процес оформлення замовлення на сайті електронної комерції, в той час як інша працює над оптимізацією перегляду та пошуку товарів - і все це відбувається одночасно, не впливаючи одна на одну.

В епоху, коли користувацький досвід є важливим фактором диференціації, тенденція зосередження на користувацькому інтерфейсі при розробці та розгортанні мікросервісів у хмарному середовищі дозволяє використовувати більш тонкий контроль і підвищену гнучкість, які пропонує ця архітектура, тим самим гарантуючи, що організації відповідають очікуванням користувачів і навіть перевершують їх.

У світі мікросервісів і хмарних додатків безпека має першочергове значення через розподілений характер архітектури. Оскільки інформація проходить через безліч вузлів і зберігається в різних місцях, потенціал вразливості зростає. Як наслідок, у сфері мікросервісів з'являється значна тенденція до запровадження сильніших і складніших заходів безпеки.

Одним з ключових аспектів прийняття більш жорстких заходів безпеки є

впровадження принципу найменших привілеїв (PoLP). У середовищі мікросервісів цей принцип означає, що сервіс повинен мати лише ті дозволи, які необхідні для виконання визначених завдань, і не більше. Якщо сервіс скомпрометований, принцип найменших привілеїв може допомогти обмежити шкоду, яку може завдати зловмисник. [1]

Крім того, використання складних методів управління ідентифікацією та доступом (IAM), шифрування, належне управління ключами та ведення аудиторських журналів для всіх транзакцій вважаються важливими заходами безпеки. Хмарне середовище допомагає підтримувати ці заходи безпеки високого рівня за допомогою різних вбудованих інструментів і сервісів, тим самим зменшуючи складність, пов'язану з їх реалізацією.

Іншим фундаментальним аспектом є включення сканування та перевірок безпеки в конвеєр CI/CD. Ці перевірки гарантують, що в системі немає відомих вразливостей, і забезпечують використання безпечних методів кодування.

Крім того, з появою таких технологій, як штучний інтелект (ШІ) і машинне навчання (МН), все більшого поширення набувають предиктивні заходи безпеки для моніторингу та оповіщення про незвичайну поведінку.

Ця тенденція до посилення заходів безпеки підкреслює критичну важливість захисту даних і сервісів. Високий рівень безпеки - це не просто бажане вдосконалення, а фундаментальна вимога до сучасних мікросервісів і хмарних архітектур для забезпечення безперервності бізнес-операцій і збереження довіри клієнтів.

Тенденції, що розвиваються у сфері архітектури мікросервісів у хмарі, мають широкий вплив на різні аспекти бізнес-операцій та IT-відділів. У цьому розділі ми розглянемо, як ці тенденції змінюють IT-ландшафт і трансформують традиційні очікування.

Поява мікросервісів та хмарних технологій спричиняє значну зміну робочих ролей в IT-секторі. У зв'язку з цим IT-персонал повинен володіти необхідними навичками для розробки, впровадження та управління мікросервісами в хмарному середовищі. Зросла потреба у фахівцях з досвідом

роботи в таких сферах, як хмарні засоби автоматизації, оркестрування контейнерів та вдосконалення методів безпеки.

Гнучкість, яку забезпечує архітектура мікросервісів та хмарне середовище, підвищила очікування щодо продуктивності. Акцент на забезпеченні високої швидкості реагування, чудового користувацького досвіду та безперервної еволюції для задоволення мінливих потреб клієнтів означає, що від бізнесу очікують постійної роботи на оптимальному рівні.

Оскільки бізнес все більше покладається на хмарні мікросервіси для надання послуг, очікування клієнтів також стрімко зростають. Клієнти прагнуть персоналізованого та розширеного досвіду, швидкого реагування та безперешкодної взаємодії через різні канали. Вони також очікують надійного захисту своєї конфіденційної інформації.

Розуміння цих впливів та наслідків має вирішальне значення для того, щоб успішно орієнтуватися в мінливому середовищі IT-сфери. Воно допомагає компаніям належним чином розробити стратегію своїх трансформаційних ініціатив, гарантуючи, що вони будуть добре підготовлені, щоб скористатися перевагами, пом'якшуючи при цьому виклики, які несуть ці нові тенденції. Здатність передбачати та адаптуватися до цих змін впливатиме на ефективність інтеграції та оптимізації мікросервісів у хмарному середовищі, що постійно розвивається.

Зміна динаміки розвитку IT, спричинена поширенням мікросервісів та хмарних технологій, провіщає зміну робочих ролей у цьому секторі. Повсюдний вплив цих технологій зумовлює необхідність еволюції існуючих ролей і появу нових. [9]

З більш деталізованою та розподіленою структурою мікросервісів зростає потреба у фахівцях, які можуть проектувати, розробляти та керувати цими модульними компонентами. Спеціалізовані ролі, такі як архітектор мікросервісів та розробник мікросервісів, стають все більш затребуваними. Вони часто вимагають глибоких знань практик DevOps, інструментів контейнеризації, таких як Docker, платформ оркестрування, таких як

Kubernetes, а також ґрунтовного розуміння мережевих протоколів і моделей зв'язку.

Крім того, розвиток хмарних мікросервісів також призвів до зростання попиту на архітекторів хмарних рішень, експертів з хмарної безпеки та інженерів з надійності сайтів. Очікується, що ці фахівці будуть орієнтуватися в складних умовах хмарних середовищ, підтримуючи ефективну роботу, надійну безпеку та високу доступність.

Більше того, оскільки безперервна інтеграція, безперервне розгортання та автоматизоване тестування стають нормою, такі ролі, як CI/CD інженери, стають все більш важливими. Ці фахівці спрощують процеси розробки програмного забезпечення та скорочують час виходу на ринок, автоматизуючи ключові етапи життєвого циклу.

Такі зміни в робочих ролях підкреслюють гостру потребу в безперервному навчанні та підвищенні кваліфікації в IT-секторі. Оскільки технологія мікросервісів продовжує розвиватися і вдосконалюватися, затребувані навички, ймовірно, змінюватимуться. Розуміння цього мінливого ландшафту може допомогти окремим особам переорієнтувати свій кар'єрний ріст, а організаціям - розробити стратегію найму та навчання персоналу. Оскільки архітектура мікросервісів стає все більш звичним явищем, ці еволюціонуючі робочі ролі відіграватимуть ключову роль у забезпеченні успішного впровадження та управління цими системами. Підвищені очікування щодо ефективності

Розвиток мікросервісів та хмарних технологій призвів до значних змін в очікуваннях щодо продуктивності. Компанії, кінцеві користувачі і навіть IT-спеціалісти в організаціях тепер очікують більш високих рівнів продуктивності завдяки потенціалу, який пропонують ці технології.

З точки зору бізнесу, мікросервіси та хмарні парадигми породили очікування швидшого розгортання нових функцій, більшої масштабованості та швидшого реагування на мінливі ринкові умови або вимоги клієнтів. Модульність мікросервісів сприяє цьому, дозволяючи ізольовано оновлювати

або змінювати окремі сервіси, не порушуючи екосистему всієї системи, а отже, скорочуючи час виходу на ринок. [5]

Кінцеві користувачі, з іншого боку, вимагають більш багатого, безперебійного та надійного користувацького досвіду, який формується завдяки високошвидкісній, безперебійній та безпечній взаємодії з додатками. Мікросервіси, розгорнуті в хмарі, відповідають цим очікуванням завдяки притаманній їм масштабованості, відмовостійкості та здатності виконувати оновлення без простоїв. Як результат, компанії бачать пряму залежність між продуктивністю додатків і задоволеністю клієнтів, що зумовлює потребу в оптимальній продуктивності.

В організаціях IT-відділи також відчувають дедалі більший тиск з боку необхідності підтримувати високу продуктивність системи. Життєво важлива роль моніторингу, збору та аналізу даних, усунення несправностей і швидкого вирішення проблем зростає в середовищі мікросервісів і хмарних технологій. Ці потреби задовольняються шляхом впровадження надійних інструментів системного моніторингу, підвищення кваліфікації для роботи з архітектурою мікросервісів та впровадження культури швидкого реагування в IT-командах.

Таким чином, поява мікросервісів і хмарних технологій значно підвищила очікування щодо продуктивності на всіх рівнях, впливаючи на стратегії і практику бізнесу і формуючи ставлення кінцевих користувачів до цифрових додатків. Створення методів, що відповідають цим підвищеним очікуванням, є ключовим аспектом інтеграції та оптимізації мікросервісів у хмарному середовищі.

Оскільки мікросервіси та хмарні обчислення продовжують впливати на цифровий ландшафт, очікування клієнтів значно змінилися. Сучасні цифрові клієнти очікують цікавого досвіду, надійних заходів безпеки, безперебійної взаємодії в режимі реального часу, персоналізованих послуг та багато іншого - і все це на тлі високопродуктивних цифрових додатків.

Враховуючи можливості архітектури мікросервісів та хмарних середовищ, бізнес має розширені можливості для задоволення цих підвищених

очікувань. Мікросервіси надають можливість адаптувати та оптимізувати кожну послугу для надання бажаної функціональності або сервісу, забезпечуючи персоналізований досвід та швидке розгортання нових функцій. Крім того, збій в роботі одного сервісу не впливає на роботу всього додатку, забезпечуючи клієнтам безперебійну роботу.

Хмарні середовища, з іншого боку, покращують масштабованість, гарантуючи користувачам безперебійну роботу без затримок навіть у пікові періоди використання. Вони також забезпечують доступність, тобто клієнти можуть отримати доступ до своїх послуг у будь-який час і з будь-якого місця. Більше того, притаманні хмарі можливості інтегрувати передові технології аналітики, машинного навчання та штучного інтелекту пропонують компаніям більше можливостей для кращого розуміння та задоволення потреб клієнтів. [5]

У той же час, технологічний прогрес призвів до того, що клієнти постійно очікують вищих стандартів. Наприклад, сценарій простою системи зараз майже не існує завдяки виявленню та виправленню помилок у режимі реального часу за допомогою мікросервісів, розгорнутих у хмарі.

Однак природа цих очікувань, що постійно змінюється, покладає на бізнес постійний тягар, який змушує його розширювати свої цифрові кордони та безперервно впроваджувати інновації. Тому розуміння та відповідність очікуванням клієнтів, що постійно змінюються, є невід'ємною частиною інтеграції та оптимізації мікросервісів у хмарі. Це дозволяє компаніям надавати винятковий досвід, якого прагнуть клієнти, і залишатися попереду в умовах жорсткої конкуренції на цифровому ринку.

2.4 Огляд архітектури мікросервісів Netflix

Netflix, провідний світовий стрімінговий сервіс, був в авангарді впровадження архітектури мікросервісів. Компанія перейшла від монолітної архітектури до архітектури мікросервісів, щоб підтримати свою швидко зростаючу клієнтську базу та підвищену складність своїх послуг.

Архітектура мікросервісів у Netflix побудована за принципом "слабко пов'язаних, але дуже згуртованих" сервісів. Кожен мікросервіс - це невеликий додаток, який має власну гексагональну архітектуру, що складається з бізнес-логіки та різних адаптерів. Деякі з цих мікросервісів включають, зокрема, сервіс рекомендацій, сервіс каталогу фільмів та сервіс обслуговування клієнтів.

Мікросервіси Netflix взаємодіють один з одним через чітко визначені API і використовують HTTP/REST з JSON або двійковий протокол. Вони організовані навколо бізнес-можливостей, і кожен сервіс розробляється, розгортається та масштабується незалежно.

Компанія використовує різноманітні інструменти для управління архітектурою мікросервісів. Наприклад, вона використовує Eureka для виявлення сервісів, Hystrix для затримки та відмовостійкості, Ribbon для балансування навантаження та Zuul для динамічної маршрутизації, серед інших.

Архітектура мікросервісів Netflix розміщена на Amazon Web Services (AWS), що робить її яскравим прикладом хмарних мікросервісів. Це дозволяє Netflix використовувати еластичність хмари для масштабування своїх послуг відповідно до попиту.

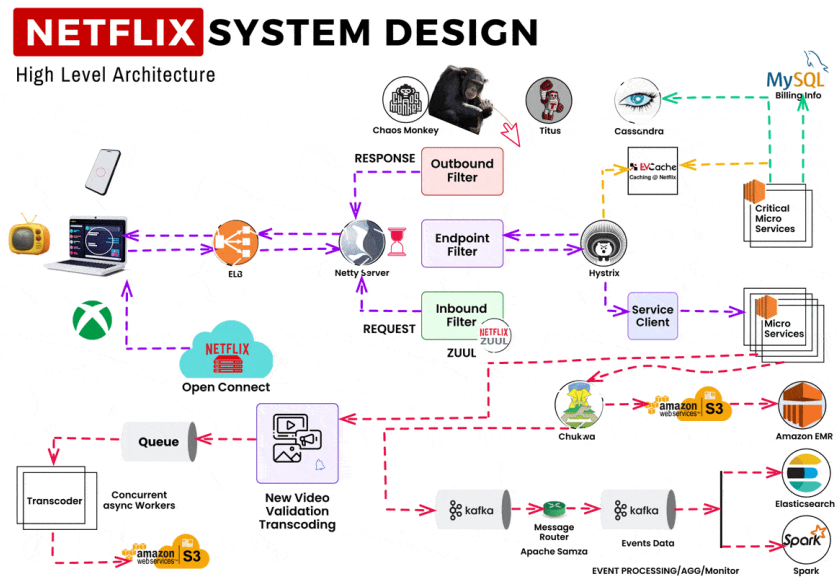

Рисунок 2.2 – План систем Netflix

Архітектура мікросервісів Netflix - це складна екосистема невеликих незалежних сервісів, які працюють разом, щоб забезпечити безперебійний потоковий досвід для мільйонів користувачів по всьому світу.

2.4.1 Підхід Netflix до інтеграції мікросервісів

Підхід Netflix до інтеграції мікросервісів базується на використанні API та набору інструментів з відкритим вихідним кодом, які вони розробили. Компанія використовує RESTful API для забезпечення зв'язку між різними мікросервісами. Ці API надають контракт на те, як мікросервіси взаємодіють один з одним, гарантуючи, що зміни в одному сервісі не порушують роботу інших.

Одним з ключових інструментів, які Netflix використовує для інтеграції, є Eureka - інструмент виявлення сервісів. Eureka дозволяє кожному мікросервісу знаходити та спілкуватися один з одним без жорсткого кодування їхнього розташування. Це дуже важливо в хмарному середовищі, де розташування сервісів може швидко змінюватися.

Netflix також використовує Zuul, периферійний сервіс, який забезпечує динамічну маршрутизацію, моніторинг, відмовостійкість і безпеку. Zuul обробляє всі запити від пристроїв і веб-сайтів до внутрішнього додатку Netflix. Він маршрутизує і фільтрує запити до відповідного мікросервісу, діючи як привратник для всього трафіку між мікросервісами.

Інший інструмент, Ribbon, використовується для міжпроцесної комунікації. Ribbon - це балансувальник навантаження на стороні клієнта, який дає змогу Netflix контролювати поведінку клієнтів HTTP і TCP під час здійснення віддалених дзвінків. Він допомагає спрямовувати запити до відповідного екземпляра мікросервісу на основі декількох факторів, таких як доступність сервера, шардеризація даних і балансування навантаження.

Netflix також використовує Hystrix, бібліотеку затримок і

відмовостійкості, щоб ізолювати точки доступу між сервісами, зупинити каскадні збої між ними і надати запасні варіанти, забезпечуючи більш стійку систему.

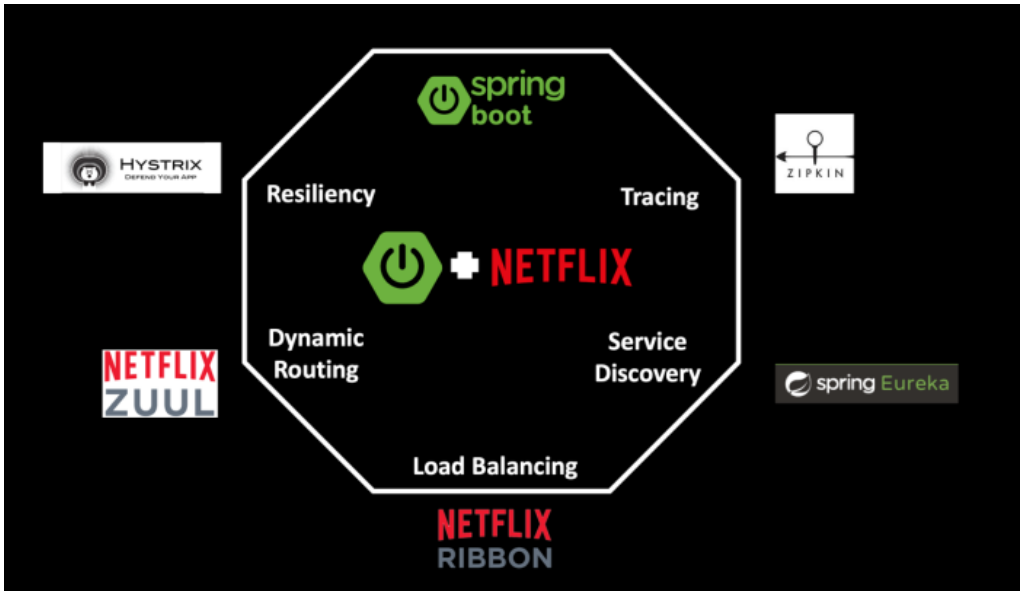

Рисунок 2.3 – Інструменти оптимізації Netflix

Підхід Netflix до інтеграції мікросервісів - це поєднання чітко визначених API та набору інструментів, які дозволяють виявляти сервіси, динамічно маршрутизувати їх, балансувати навантаження та забезпечувати відмовостійкість. Такий підхід дозволяє Netflix підтримувати дуже роз'єднану, але згуртовану систему мікросервісів.

2.4.2 Підхід Netflix до оптимізації мікросервісів

Підхід Netflix до оптимізації архітектури своїх мікросервісів є багатогранним, зосередженим на продуктивності, масштабованості та відмовостійкості. Однією з ключових стратегій оптимізації, яку застосовує Netflix, є використання автомасштабування. Розміщений на Amazon Web Services (AWS), Netflix використовує можливості автомасштабування AWS для динамічного регулювання кількості екземплярів серверів у відповідь на

структуру трафіку. Це гарантує, що в періоди пікового трафіку система може впоратися з навантаженням, а в більш спокійні періоди ресурси не витрачаються даремно.

Netflix також використовує інструмент під назвою Conductor, механізм оркестрування мікросервісів, для оптимізації виконання складних бізнес-потоків, які охоплюють кілька мікросервісів. Conductor допомагає керувати взаємодією між мікросервісами, забезпечуючи їхню ефективну спільну роботу.

Для оптимізації роботи своїх мікросервісів Netflix використовує поєднання аналітики в реальному часі та машинного навчання. Вони збирають величезну кількість даних про те, як працюють їхні сервіси, і використовують ці дані для виявлення вузьких місць і сфер для вдосконалення. Потім алгоритми машинного навчання використовуються для прогнозування та запобігання потенційних проблем до того, як вони вплинуть на користувацький досвід.

Netflix також робить сильний акцент на стійкості. Вони використовують інструмент під назвою Chaos Monkey, який є частиною більшої Simian Army, для випадкового завершення роботи сутності у продакшені, щоб гарантувати, що інженери реалізують свої сервіси стійкими до збоїв сутностей.

Підхід Netflix до оптимізації мікросервісів передбачає поєднання автомасштабування, оркестровки, аналітики в реальному часі, машинного навчання та хаос-інженерії. Такий підхід гарантує, що їхня архітектура мікросервісів є не лише продуктивною та масштабованою, але й стійкою.

2.4.3 Результати та переваги, досягнуті Netflix

Перехід на архітектуру мікросервісів приніс Netflix значні результати та переваги. Одним з найпомітніших результатів є можливість швидкого та ефективного масштабування. Маючи понад 200 мільйонів підписників по всьому світу, архітектура мікросервісів Netflix довела, що здатна витримувати величезні масштаби, з можливістю запуску тисяч екземплярів за лічені хвилини, щоб впоратися з піковими навантаженнями.

Використання мікросервісів також покращило швидкість впровадження інновацій у Netflix. Завдяки роз'єднанню сервісів команди можуть працювати незалежно над різними мікросервісами, що дозволяє їм швидше ітерації та скоротити час виведення на ринок нових функцій. Це дозволило Netflix постійно вдосконалювати свою платформу і випереджати конкурентів. Стійкість Netflix також значно покращилася. Такі інструменти, як Hystrix і Chaos Monkey, зробили систему більш надійною і здатною справлятися зі збоями, не впливаючи на користувацький досвід. Це призвело до збільшення часу безвідмовної роботи та підвищення надійності сервісу для мільйонів користувачів Netflix.

Крім того, використання аналітики в режимі реального часу та машинного навчання для оптимізації продуктивності призвело до підвищення ефективності системи. Це допомогло Netflix забезпечити безперебійний потоковий досвід для своїх користувачів навіть у періоди пікового трафіку.

Впровадження архітектури мікросервісів дозволило Netflix ефективно масштабуватися, швидко впроваджувати інновації, підвищувати стійкість і оптимізувати продуктивність, надаючи користувачам чудовий сервіс і зберігаючи свою позицію лідера в індустрії стрімінгових сервісів.

2.5 Огляд архітектури мікросервісів Amazon

Amazon, один з найбільших світових онлайн-рітейлерів, став піонером у впровадженні архітектури мікросервісів. Компанія перейшла від монолітної архітектури до архітектури мікросервісів для підтримки широкого і різноманітного спектру послуг, від електронної комерції до хмарних обчислень, цифрового потокового мовлення та штучного інтелекту.

Архітектура мікросервісів Amazon побудована за принципом "єдиної відповідальності", коли кожен мікросервіс відповідає за окрему бізнес-можливість. Ці мікросервіси невеликі, незалежні та слабо пов'язані між собою, що дозволяє розробляти, розгортати та масштабувати їх незалежно.

Мікросервіси Amazon взаємодіють один з одним через API, використовуючи такі протоколи, як REST і gRPC. Вони організовані навколо бізнес-можливостей, і кожен сервіс належить невеликій команді, яка відповідає за повний життєвий цикл сервісу.

Архітектура мікросервісів Amazon розміщена на Amazon Web Services (AWS), власній платформі хмарних обчислень. Це дозволяє Amazon використовувати масштабованість, надійність та широту послуг, що пропонуються AWS.

Компанія використовує різноманітні інструменти для управління своєю архітектурою мікросервісів. Наприклад, вона використовує AWS Lambda для безсерверних обчислень, Amazon API Gateway для управління API, Amazon ECS та EKS для оркестрування контейнерів та AWS X-Ray для моніторингу та налагодження, серед іншого.

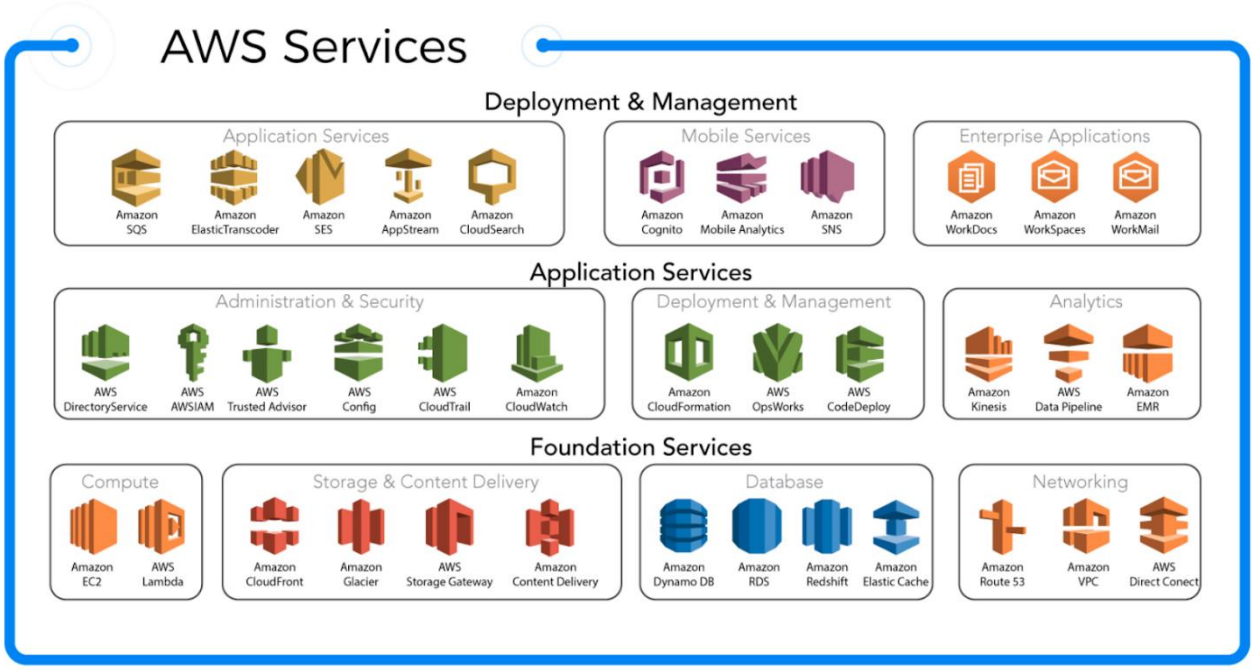

Рисунок 2.4 – Сервіси Amazon Web Services

Архітектура мікросервісів Amazon - це складна екосистема невеликих незалежних сервісів, які працюють разом, щоб надавати широкий спектр послуг

мільйонам клієнтів по всьому світу.

2.5.1 Підхід Amazon до інтеграції мікросервісів

Підхід Amazon до інтеграції мікросервісів базується на використанні API та набору інструментів AWS. Компанія використовує RESTful API для забезпечення зв'язку між різними мікросервісами. Ці API надають договір про те, як мікросервіси взаємодіють один з одним, гарантуючи, що зміни в одному сервісі не порушують роботу інших.

Amazon API Gateway відіграє важливу роль в управлінні цими API. Він виконує всі завдання, пов'язані з прийомом та обробкою паралельних викликів API, включаючи управління трафіком, авторизацію та контроль доступу, моніторинг та управління версіями API.

Для пошуку сервісів Amazon використовує AWS Cloud Map, сервіс пошуку хмарних ресурсів. Це дозволяє їхнім мікросервісам визначати власні імена для своїх ресурсів і підтримувати актуальне місцезнаходження, що полегшує сервісам пошук і зв'язок один з одним.

Amazon також використовує AWS Step Functions для координації компонентів розподілених додатків і мікросервісів. Цей сервіс надає візуальний робочий процес, який допомагає керувати залежностями, ланцюжками, розгалуженнями та паралельним виконанням завдань між кількома мікросервісами.

Для міжпроцесної комунікації Amazon використовує Amazon SQS (Simple Queue Service) та Amazon SNS (Simple Notification Service). SQS - це повністю керована служба черги повідомлень, яка дозволяє мікросервісам спілкуватися один з одним, тоді як SNS - це повністю керована служба обміну повідомленнями як для спілкування між додатками, так і для спілкування між користувачами.

Таким чином, підхід Amazon до інтеграції мікросервісів передбачає поєднання чітко визначених API, пошуку сервісів, координації робочих

процесів та послуг обміну повідомленнями. Такий підхід дозволяє Amazon підтримувати дуже роз'єднану, але згуртовану систему мікросервісів.

2.5.2 Підхід Amazon до оптимізації мікросервісів

Підхід Amazon до оптимізації своєї архітектури мікросервісів зосереджений на продуктивності, масштабованості та відмовостійкості, використовуючи різноманітні інструменти та сервіси AWS.

Для оптимізації продуктивності Amazon використовує AWS X-Ray - сервіс, який надає уявлення про поведінку своїх додатків, допомагаючи зрозуміти, як працює їхній додаток і де виникають вузькі місця. Це дозволяє Amazon проактивно вирішувати проблеми продуктивності та оптимізувати свої мікросервіси для кращої роботи.

Масштабованість є ключовим аспектом оптимізації мікросервісів Amazon. Amazon використовує AWS Lambda для безсерверних обчислень, який автоматично масштабує додаток у відповідь на вхідний трафік запитів. Це гарантує, що їхні мікросервіси можуть ефективно обробляти різні навантаження.

Amazon також використовує AWS Fargate, безсерверний обчислювальний движок для контейнерів, який працює з Amazon Elastic Container Service (ECS) та Amazon Elastic Kubernetes Service (EKS). Fargate усуває необхідність у наданні та управлінні серверами, дозволяючи Amazon зосередитися на розробці та створенні своїх додатків.

Для забезпечення стійкості Amazon використовує AWS Shield, керовану службу захисту від розподілених відмов у обслуговуванні (DDoS), яка захищає додатки, що працюють на AWS. Це гарантує, що їхні мікросервіси стійкі до DDoS-атак, підвищуючи їхню загальну надійність.

Крім того, Amazon використовує AWS Auto Scaling для автоматичного регулювання потужності, щоб підтримувати стабільну, передбачувану продуктивність за найнижчих можливих витрат. Це не лише підвищує

доступність додатків, але й знижує витрати, збільшуючи або зменшуючи потужність за потреби.

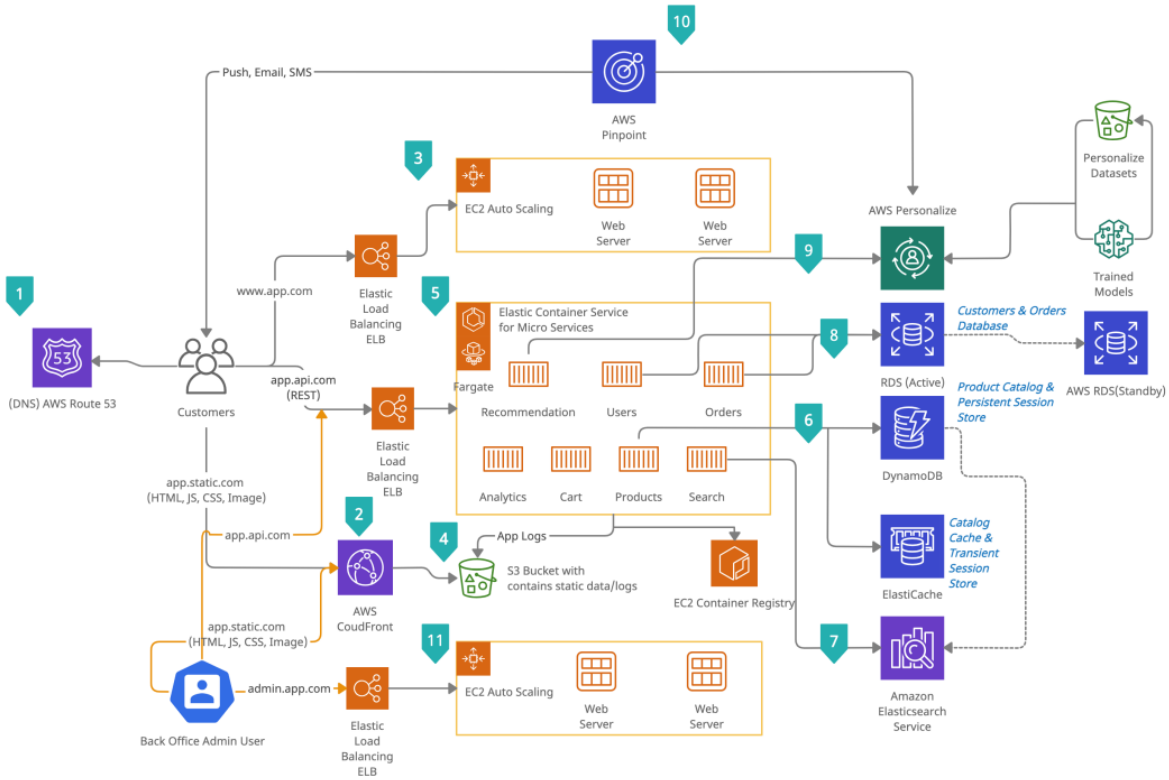

Рисунок 2.5 – Типова архітектура на платформі AWS

Таким чином, підхід Amazon до оптимізації мікросервісів передбачає поєднання моніторингу продуктивності, безсерверних обчислень, контейнеризації, захисту від DDoS-атак та автоматичного масштабування. Такий підхід гарантує, що їхня архітектура мікросервісів є продуктивною, масштабованою та відмовостійкою.

2.5.3 Результати та переваги, досягнуті Amazon

Перехід на архітектуру мікросервісів приніс значні результати та переваги для Amazon. Одним з найпомітніших результатів є можливість швидкого та ефективного масштабування. З мільйонами клієнтів по всьому світу та широким спектром послуг, архітектура мікросервісів Amazon довела, що здатна витримувати величезні масштаби, з можливістю швидко регулювати

кількість серверів у відповідь на трафік.

Використання мікросервісів також покращило швидкість впровадження інновацій в Amazon. Завдяки роз'єднанню сервісів команди можуть працювати незалежно над різними мікросервісами, що дозволяє їм пришвидшити ітерації та скоротити час виведення на ринок нових функцій. Це дозволило Amazon постійно вдосконалювати свою платформу і випереджати конкурентів.

Стійкість Amazon також значно покращилася. Такі інструменти, як AWS Shield, зробили систему більш надійною і здатною протистояти потенційним загрозам, не впливаючи на користувацький досвід. Це призвело до збільшення часу безвідмовної роботи та підвищення надійності сервісу для мільйонів користувачів Amazon.

Крім того, використання AWS X-Ray для оптимізації продуктивності призвело до підвищення ефективності системи. Це допомогло Amazon забезпечити безперебійну роботу своїх користувачів навіть у періоди пікового трафіку.

З точки зору економічної ефективності, використання AWS Lambda та AWS Fargate дозволило Amazon платити лише за час, який вони споживають, що призвело до значної економії коштів.

Таким чином, впровадження архітектури мікросервісів дозволило Amazon ефективно масштабуватися, швидко впроваджувати інновації, підвищити стійкість, оптимізувати продуктивність і знизити витрати, надаючи користувачам чудовий сервіс і зберігаючи свою позицію лідера в індустрії онлайн-рітейлу.

2.6 Огляд архітектури мікросервісів Twitter

Twitter, провідна соціальна мережа, також перейшла на архітектуру мікросервісів, щоб підтримувати масовість своїх послуг та їхню роботу в режимі реального часу. Компанія перейшла від монолітної архітектури до архітектури мікросервісів, щоб впоратися зі зростаючою складністю своїх

послуг і швидким зростанням користувацької бази.

Архітектура мікросервісів Twitter побудована за принципом "робіть щось одне і робіть це добре". Кожен мікросервіс відповідає за окрему функцію, наприклад, обробку твітів, управління профілями користувачів або обслуговування графіків. Ці мікросервіси невеликі, незалежні та слабо пов'язані між собою, що дозволяє розробляти, розгортати та масштабувати їх незалежно.

Мікросервіси Twitter взаємодіють один з одним через чітко визначені API, використовуючи такі протоколи, як Thrift (програмний фреймворк для розробки масштабованих міжмовних сервісів) і HTTP/REST. Вони організовані навколо бізнес-можливостей, і кожен сервіс належить невеликій команді, яка відповідає за повний життєвий цикл сервісу.

Архітектура мікросервісів Twitter базується на Mesos, ядрі розподілених систем, та Aurora, фреймворку Mesos для довготривалих сервісів та завдань, що виконуються за допомогою cron. Це дозволяє Twitter керувати своїми мікросервісами в масштабі і гарантує, що вони розподілені по спільному пулу машин, покращуючи використання ресурсів.

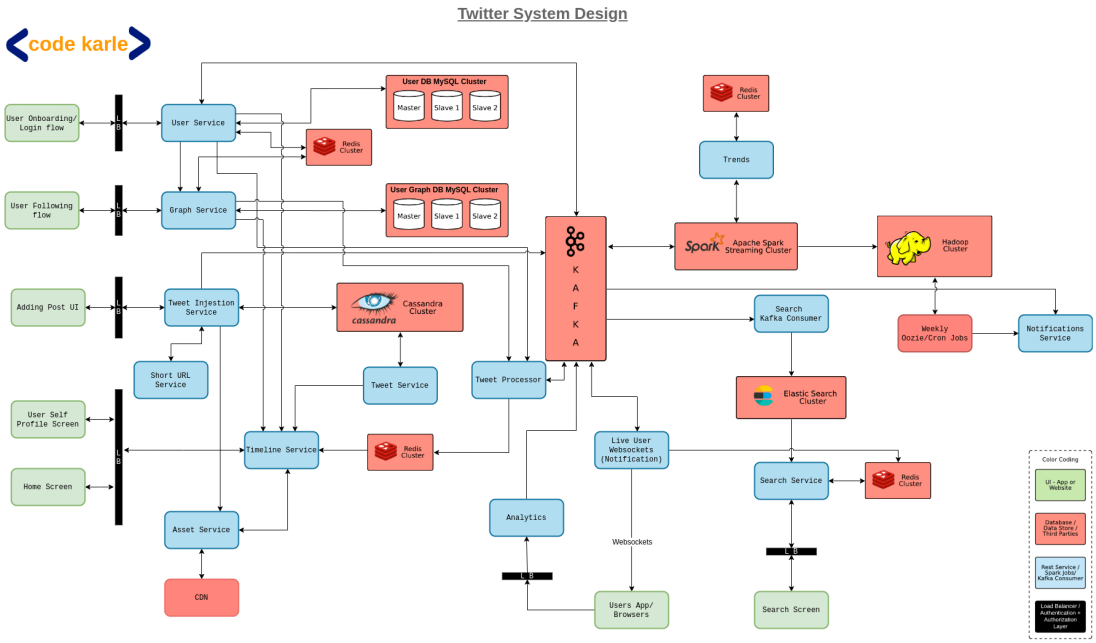

Рисунок 2.6 – План систем Twitter

Архітектура мікросервісів Twitter - це складна екосистема невеликих незалежних сервісів, які працюють разом, щоб забезпечити мільйони користувачів по всьому світу платформою соціальних медіа в режимі реального часу.

2.6.1 Підхід Twitter до інтеграції мікросервісів

Підхід Twitter до інтеграції мікросервісів базується на використанні API та набору інструментів власної розробки. Компанія використовує API-інтерфейси Thrift для забезпечення зв'язку між різними мікросервісами. Ці API надають договір про те, як мікросервіси взаємодіють один з одним, гарантуючи, що зміни в одному сервісі не порушують роботу інших.

Одним з ключових інструментів, які Twitter використовує для інтеграції, є Finagle, розширювана система RPC для JVM, що використовується для побудови високопродуктивних серверів. Finagle забезпечує єдину модель для обробки збоїв на рівні додатків, що полегшує створення надійних серверів і клієнтів.

Twitter також використовує Mesos та Aurora для управління та оркестрування своїх мікросервісів. Mesos забезпечує рівень абстракції між додатками та пулом машин, на яких вони працюють, що дозволяє ефективно ізолювати ресурси та спільно використовувати їх між розподіленими додатками. Aurora, з іншого боку, використовується для планування завдань та управління ресурсами, забезпечуючи ефективний розподіл мікросервісів у спільному пулі машин.

Для виявлення сервісів Twitter використовує ZooKeeper, централізований сервіс для підтримки інформації про конфігурацію, іменування, забезпечення розподіленої синхронізації та надання групових послуг. Це дозволяє їхнім мікросервісам ефективно знаходити та спілкуватися один з одним.

Таким чином, підхід Twitter до інтеграції мікросервісів передбачає поєднання чітко визначених API, розширюваної системи RPC, інструментів управління ресурсами та оркестрування, а також системи виявлення сервісів.

Такий підхід дозволяє Твіттеру підтримувати дуже відокремлену, але згуртовану систему мікросервісів.

2.6.2 Підхід Twitter до оптимізації мікросервісів

Підхід Twitter до оптимізації архітектури своїх мікросервісів зосереджений на продуктивності, масштабованості та відмовостійкості, використовуючи різноманітні інструменти власної розробки та технології з відкритим кодом.

Для оптимізації продуктивності Twitter використовує інструмент під назвою Zipkin, який являє собою розподілену систему трасування, що допомагає збирати дані про час виконання кожного запиту, який поширюється через систему. Це дозволяє Twitter виявляти вузькі місця в роботі та оптимізувати свої мікросервіси для кращої продуктивності.

Масштабованість є ключовим аспектом оптимізації мікросервісів Twitter. Twitter використовує Mesos та Aurora для управління та оркестрування своїх мікросервісів. Mesos забезпечує ефективну ізоляцію та спільне використання ресурсів між розподіленими додатками, в той час як Aurora планує завдання та керує ресурсами, гарантуючи, що мікросервіси ефективно розподіляються між спільним пулом машин.

Для забезпечення відмовостійкості Twitter використовує комбінацію Finagle та Hystrix. Finagle забезпечує єдину модель для обробки збоїв на рівні додатків, що полегшує створення надійних серверів і клієнтів. Hystrix, з іншого боку, - це бібліотека затримок і відмовостійкості, призначена для ізоляції точок доступу між сервісами, зупинки каскадних збоїв між ними та забезпечення запасних варіантів.

Крім того, Twitter використовує техніку під назвою "шардінг" для розподілу даних між декількома базами даних, зменшуючи навантаження на кожну базу даних і підвищуючи загальну продуктивність і надійність своєї системи.

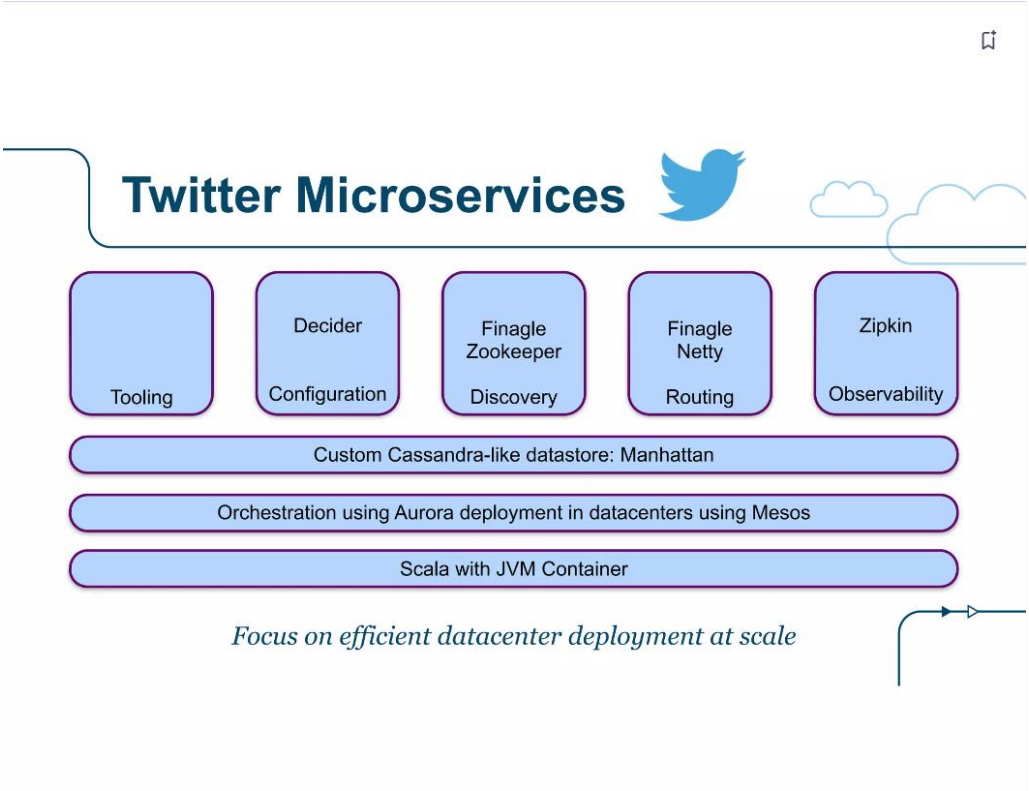

Рисунок 2.7 – Інструменти оптимізації Twitter

Підхід Twitter до оптимізації мікросервісів передбачає поєднання розподіленого трасування, управління ресурсами та оркестровки, відмовостійкості та шардеризації даних. Такий підхід гарантує, що їхня архітектура мікросервісів є продуктивною, масштабованою та відмовостійкою.

2.6.3 Результати та переваги, досягнуті Твіттером

Перехід до архітектури мікросервісів приніс значні результати та переваги для Twitter. Одним з найпомітніших результатів є здатність справлятися з величезними масштабами. Маючи сотні мільйонів активних користувачів і понад 500 мільйонів твітів на день, архітектура мікросервісів Twitter довела, що здатна впоратися з такими масштабами, надаючи оновлення в режимі реального часу своїм користувачам по всьому світу.

Використання мікросервісів також покращило швидкість впровадження інновацій у Twitter. Завдяки роз'єднанню сервісів команди можуть працювати

незалежно над різними мікросервісами, що дозволяє їм швидше ітерації та скоротити час виведення на ринок нових функцій. Це дозволило Twitter постійно вдосконалювати свою платформу і випереджати конкурентів.

Стійкість Twitter також значно покращилася. Такі інструменти, як Finagle і Hystrix, зробили систему більш надійною і здатною справлятися зі збоями, не впливаючи на користувацький досвід. Це призвело до збільшення часу безвідмовної роботи та підвищення надійності сервісу для мільйонів користувачів Twitter.

Крім того, використання Zipkin для оптимізації продуктивності призвело до підвищення ефективності системи. Це допомогло Twitter забезпечити безперебійну роботу користувачів навіть у періоди високого трафіку.

З точки зору економічної ефективності, використання Mesos і Aurora дозволило Twitter ефективно управляти ресурсами спільного пулу машин, що призвело до значної економії коштів.

Впровадження архітектури мікросервісів дозволило Twitter ефективно масштабуватися, швидко впроваджувати інновації, підвищити стійкість, оптимізувати продуктивність і знизити витрати, надаючи користувачам чудовий сервіс і зберігаючи свою позицію лідера в індустрії соціальних медіа.

2.7 Аналіз отриманих результатів

На завершення, тематичні дослідження Netflix, Amazon і Twitter розкривають кілька найкращих практик і стратегій, які сприяли їхньому успіху у впровадженні та оптимізації мікросервісів у хмарному середовищі.

Усі три компанії розробили свої мікросервіси навколо конкретних бізнес-можливостей, що дозволило їм розробляти, розгортати та масштабувати їх незалежно. Таке розділення послуг дозволило швидше впроваджувати інновації та ефективніше масштабуватися.

API мають вирішальне значення для забезпечення зв'язку між мікросервісами. Вони надають договір про те, як мікросервіси взаємодіють

один з одним, гарантуючи, що зміни в одному сервісі не порушують роботу інших.

Для виявлення сервісів були використані такі інструменти, як Eureka (Netflix), AWS Cloud Map (Amazon) та ZooKeeper (Twitter), а для балансування навантаження - Ribbon (Netflix) та сервіси AWS (Amazon). Ці інструменти забезпечують ефективну маршрутизацію запитів і балансування навантаження між різними екземплярами сервісів.

Для моніторингу продуктивності та виявлення вузьких місць були використані системи аналітики в реальному часі та розподіленої трасування, такі як AWS X-Ray (Amazon) та Zipkin (Twitter). Це допомогло оптимізувати продуктивність мікросервісів.

Такі інструменти, як Hystrix (Netflix і Twitter) і AWS Shield (Amazon), були використані для підвищення стійкості системи і плавного реагування на збої. Це призвело до збільшення часу безвідмовної роботи та підвищення надійності сервісу.

Такі інструменти, як Mesos і Aurora (Twitter), а також сервіси AWS (Amazon), використовувалися для управління ресурсами та їх оркестрування, забезпечуючи ефективне використання ресурсів.

І Netflix, і Amazon використовують можливості автоматичного масштабування AWS для динамічного налаштування кількості серверів у відповідь на трафік. Це забезпечує ефективну обробку пікових навантажень та економію коштів у більш спокійні періоди.

Ці найкращі практики та стратегії не лише дозволили цим компаніям успішно впровадити та оптимізувати мікросервіси, але й надали цінні уроки для інших компаній, які прагнуть впровадити архітектуру мікросервісів у хмарному середовищі.

3 ТЕОРЕТИЧНІ АСПЕКТИ РЕАЛІЗАЦІЇ СИСТЕМИ

3.1 Огляд системи Crypto Rate

Для демонстрації найкращих практик інтеграції та оптимізації мікросервісів в хмарі була створена аналітична платформа Crypto Rate, яка призначена для надання користувачам даних про курси криптовалют в реальному часі, що підвищує їхню здатність приймати обґрунтовані торгові рішення. Платформа задовольняє потребу в надійній, своєчасній і точній інформації про криптовалюти на все більш волатильному і динамічному ринку. Використовуючи архітектуру мікросервісів і хмарні технології, платформа забезпечує масштабованість, гнучкість і високу доступність.

Архітектура програми складається з декількох мікросервісів, реалізованих на основі AWS Lambda, кожен з яких відповідає за певну функцію:

DataFetchService: Періодично отримує курси криптовалют із зовнішніх API за допомогою планувальника.

DataProcessorService: Обробляє та зберігає отримані дані, забезпечуючи їх доступність для швидкого пошуку.

UserService: Керує даними користувача та обробляє аутентифікацію і авторизацію.

CoinRateService: Надає кінцеві точки для отримання оброблених даних про курси криптовалют.

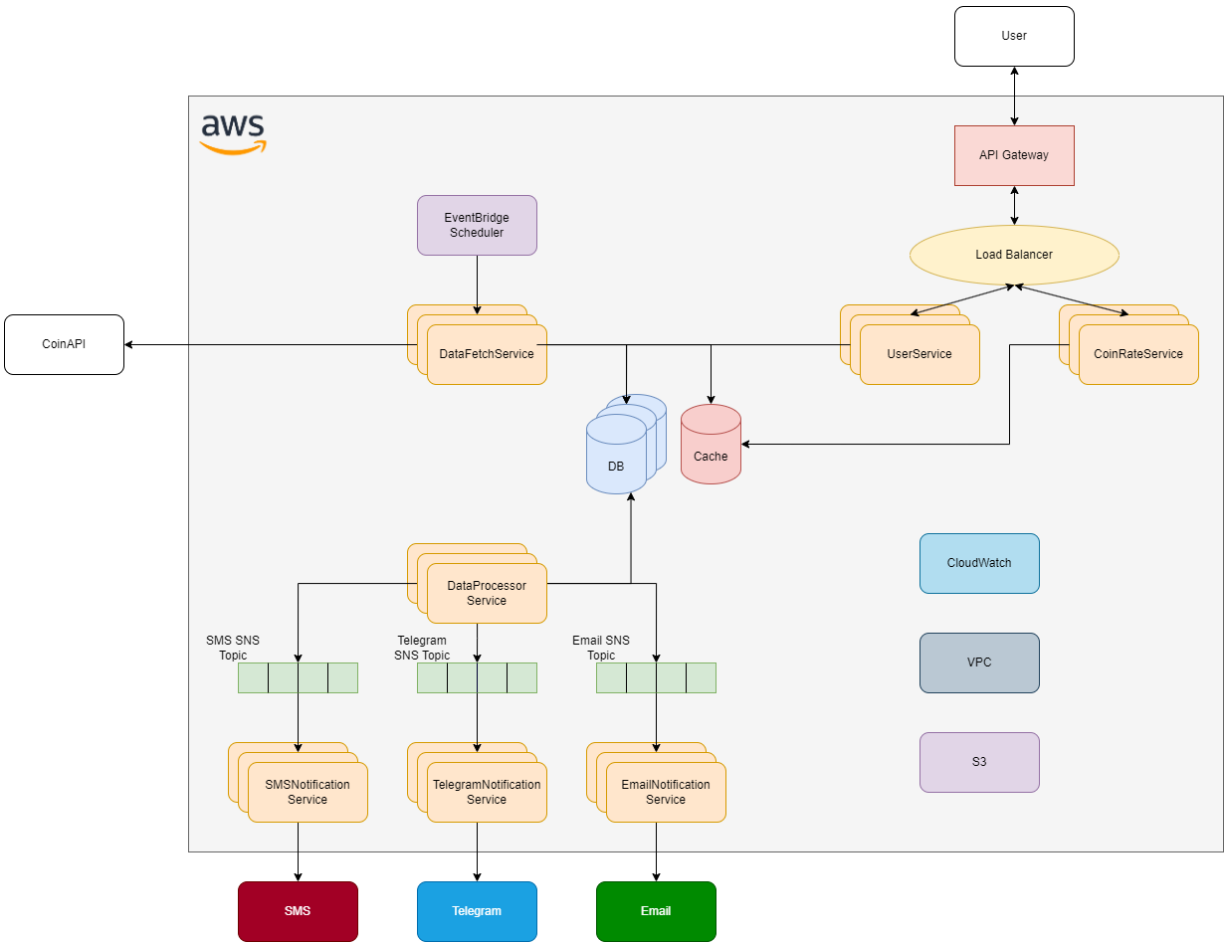

Рисунок 3.1 – План системи Crypto Rate

Служби сповіщень: Надсилають сповіщення користувачам через SMS, Telegram та електронну пошту на основі попередньо визначених тригерів.

Ці мікросервіси взаємодіють з центральною базою даних і рівнем кешування, щоб забезпечити узгодженість даних і швидкий час відгуку. Архітектура також включає API-шлюз для обробки вхідних запитів користувачів, забезпечуючи уніфіковану та безпечну точку входу на платформу.

При практичній реалізації Crypto Rate Platform було використано кілька стратегій оптимізації та методів хмарної інтеграції, щоб забезпечити відповідність системи високим стандартам продуктивності, надійності та масштабованості.

Для оптимізації управління запитами та підвищення безпеки в

архітектуру було інтегровано Amazon API Gateway. Цей компонент слугує точкою входу для всіх взаємодій користувачів з платформою, спрямовуючи запити до відповідних внутрішніх сервісів. API-шлюз виконує такі завдання, як перевірка запитів, обмеження швидкості та автентифікація, гарантуючи, що лише авторизовані та правильно сформовані запити потрапляють до внутрішніх служб. Це не тільки захищає систему від потенційних загроз, але й допомагає ефективно керувати трафіком.

Виклики DataFetchService плануються за допомогою Amazon EventBridge, який запускає вибірку даних через регулярні проміжки часу. Цей сервіс отримує курси криптовалют із зовнішніх API, таких як CoinAPI, і зберігає дані в центральній базі даних і кеші. EventBridge дозволяє здійснювати точне планування, гарантуючи, що вибірка даних відбувається без ручного втручання, і підлаштовується під зміну структури навантаження, оптимізуючи використання ресурсів.

Сервіс DataProcessorService обробляє вхідні дані, виконуючи необхідні перетворення та зберігаючи результати в базі даних, оптимізованій для інтенсивних операцій зчитування. Цей сервіс розроблений таким чином, щоб не залежати від стану і бути горизонтально масштабованим, забезпечуючи можливість регулювання потужності обробки відповідно до обсягу вхідних даних.

Платформа включає в себе надійну систему сповіщень, яка сповіщає користувачів через SMS, Telegram та електронну пошту, коли виконуються певні умови. Amazon SNS (Simple Notification Service) використовується для управління та розповсюдження повідомлень серед різних служб сповіщень. Кожна служба сповіщень підписується на теми SNS, забезпечуючи ефективну доставку повідомлень користувачам різними каналами.

Мікросервіси, такі як UserService та CoinRateService, розгорнуті в групі з автоматичним масштабуванням за еластичним балансувальником навантаження. Таке налаштування гарантує, що сервіси можуть масштабуватися в періоди високого попиту і збільшуватися в більш спокійні

періоди, оптимізуючи використання ресурсів і витрати. Крім того, розгортання сервісів у декількох зонах доступності підвищує відмовостійкість системи, забезпечуючи високу доступність і відмовостійкість.

AWS CloudWatch використовується для моніторингу продуктивності та працездатності сервісів. Відстежуються такі показники, як затримка запитів, рівень помилок і використання ресурсів, що дає уявлення про робочий стан системи. Журнали різних сервісів агрегуються та аналізуються для виявлення аномалій та оперативного усунення несправностей.

Для ефективного управління операційними витратами платформа використовує стратегії оптимізації витрат AWS, такі як використання зарезервованих екземплярів для передбачуваних робочих навантажень, надання рекомендацій щодо правильного вибору розміру та регулярний аналіз використання ресурсів. Такий підхід гарантує, що платформа залишається економічно ефективною, зберігаючи при цьому високу продуктивність і доступність.

Інтегруючи ці хмарні сервіси та дотримуючись найкращих практик архітектури мікросервісів Crypto Rate Platform створює надійне, масштабоване та оптимізоване рішення для надання користувачам аналітики курсів криптовалют. Ця реалізація демонструє ефективність сучасних хмарних технологій у створенні та оптимізації додатків на основі мікросервісів.

4 ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ

4.1.1 Реалізація Сервісу Отримання Даних

Лістинг 4.1 – Код реалізації сервісу «DataFetchHandler»

```
      class   DataFetchHandler   :   RequestHandler<APIGatewayProxyRequestEvent,
APIGatewayProxyResponseEvent> {

          private lateinit var client: HttpClient
          private   val   dbUrl:   String   =   System.getenv("DB_URL")   ?:   throw
IllegalArgumentException("DB_URL is not set")
          private   val   dbUser:   String   =   System.getenv("DB_USER")   ?:   throw
IllegalArgumentException("DB_USER is not set")
          private val dbPassword: String = System.getenv("DB_PASSWORD") ?: throw
IllegalArgumentException("DB_PASSWORD is not set")
          private   val   apiUrl:   String   =   System.getenv("API_URL")   ?:   throw
IllegalArgumentException("API_URL is not set")
          private   val   apiKey:   String   =   System.getenv("API_KEY")   ?:   throw
IllegalArgumentException("API_KEY is not set")
          private            val            logger:            Logger            =
LoggerFactory.getLogger(DataFetchHandler::class.java)

          override fun handleRequest(
              input: APIGatewayProxyRequestEvent?,
              context: Context?
          ): APIGatewayProxyResponseEvent {
              client = createHttpClient()
              val  response  =  fetchExchangeRate()  ?:  ExchangeRateResponse("",
emptyList())
              saveToDatabase(response)
              client.close()

              return createResponse(response)
          }

          private fun createHttpClient(): HttpClient {
              return HttpClient {
                  install(ContentNegotiation) {
                      json(Json { ignoreUnknownKeys = true })
                  }
              }
          }

          private fun fetchExchangeRate(): ExchangeRateResponse? = runBlocking {
              return@runBlocking try {
                  client.get(apiUrl) {
                      header("X-CoinAPI-Key", apiKey)
                      header("Accept", "text/plain")
                  }.body()
              } catch (e: Exception) {
                  logger.error("Error fetching exchange rate: ${e.message}", e)
                  null
              }
          }
```

```
        private fun saveToDatabase(response: ExchangeRateResponse) {
            try {
                DriverManager.getConnection(dbUrl, dbUser, dbPassword).use {
connection ->
                    createTableIfNotExists(connection)
                    insertExchangeRates(connection, response)
                }
            } catch (e: SQLException) {
                logger.error("Error interacting with the database:
${e.message}", e)
            }
        }

        private fun createTableIfNotExists(connection: Connection) {
            val createTableSQL = """
                CREATE TABLE IF NOT EXISTS exchange_rates (
                    asset_id_base VARCHAR(255),
                    asset_id_quote VARCHAR(255),
                    rate DOUBLE PRECISION,
                    PRIMARY KEY (asset_id_base, asset_id_quote)
                )
            """
            connection.createStatement().use { it.execute(createTableSQL) }
        }

        private fun insertExchangeRates(connection: Connection, response:
ExchangeRateResponse) {
            val insertSQL = """
                INSERT INTO exchange_rates (asset_id_base, asset_id_quote,
rate)
                VALUES (?, ?, ?)
                ON CONFLICT (asset_id_base, asset_id_quote) DO UPDATE SET rate
= EXCLUDED.rate
            """
            connection.prepareStatement(insertSQL).use { preparedStatement ->
                for (rate in response.rates) {
                    preparedStatement.setString(1, response.assetIdBase)
                    preparedStatement.setString(2, rate.assetIdQuote)
                    preparedStatement.setDouble(3, rate.rate)
                    preparedStatement.addBatch()
                }
                preparedStatement.executeBatch()
            }
        }

        private fun createResponse(response: ExchangeRateResponse):
APIGatewayProxyResponseEvent {
            return APIGatewayProxyResponseEvent().apply {
                body = response.toString()
            }
        }
    }
```

Клас `DataFetchHandler` є ключовим компонентом Crypto Rate Платформи, відповідальним за періодичне отримання курсів криптовалют з публічного API та збереження їх у базі даних. Цей сервіс реалізований за допомогою AWS Lambda, яка тригерується CloudWatch Events для забезпечення

регулярного отримання та оновлення даних. Ось детальний опис його призначення та реалізації.

Основна мета сервісу `DataFetchHandler` – автоматизувати отримання актуальних курсів криптовалют та зберігати цю інформацію в базі даних для подальшого аналізу та використання іншими сервісами платформи. Цей сервіс допомагає забезпечити платформу точними та актуальними даними, покращуючи можливості користувачів приймати обґрунтовані торгові рішення.

Клас `DataFetchHandler` успадковує `RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent>`, що робить його сумісним з подійно-орієнтованою моделлю виконання AWS Lambda:

1) клас використовує змінні оточення для налаштування параметрів підключення до бази даних (`DB_URL`, `DB_USER`, `DB_PASSWORD`) та доступу до API (`API_URL`, `API_KEY`). Це необхідно для безпечного підключення до зовнішнього API та бази даних;

2) метод `createHttpClient` налаштовує `HttpClient` з підтримкою JSON-контенту, що дозволяє обробляти JSON-відповіді від API;

3) метод `fetchExchangeRate` здійснює HTTP GET-запит до зовнішнього API, використовуючи налаштовані `apiUrl` та `apiKey`. Він виконується в межах корутини `runBlocking` для обробки асинхронних мережевих операцій;

4) операції з Базою Даних:

- saveToDatabase: Цей метод встановлює з'єднання з базою даних і забезпечує збереження отриманих даних. Він викликає `createTableIfNotExists` для створення необхідної таблиці, якщо вона не існує, та `insertExchangeRates` для вставки або оновлення даних;

- createTableIfNotExists: Створює таблицю `exchange_rates`, якщо вона ще не існує;

- insertExchangeRates: Вставляє нові курси валют або оновлює існуючі записи, використовуючи конструкцію `ON CONFLICT` для обробки конфліктів первинного ключа;

5) метод `createResponse` генерує об'єкт відповіді, який повертається функцією Lambda. Ця відповідь містить отримані дані про курси валют у своєму тілі;

6) реалізація включає всебічну обробку помилок для журналювання помилок, що виникають під час запитів до API або операцій з базою даних, забезпечуючи надійність і стійкість системи.

`DataFetchHandler` розгорнуто як функцію AWS Lambda. Lambda надає безсерверну обчислювальну службу, яка виконує функцію у відповідь на певні тригери, у цьому випадку – CloudWatch Events.

CloudWatch Events (тепер частина EventBridge) використовуються для запланованого виконання функції Lambda через регулярні інтервали. Це забезпечує періодичне отримання та оновлення курсів валют без ручного втручання.

Дані, отримані з публічного API біржі, включають курси обміну для різних пар криптовалют.

Лістинг 4.2 – Приклад відповіді від CoinAPI

```json
[
    {
        "time": "2024-06-02T13:30:13.0000000Z",
        "asset_id_quote": "ETH",
        "rate": 0.00026374548937274153025172 59
    },
    {
        "time": "2024-06-02T13:30:13.0000000Z",
        "asset_id_quote": "BTC",
        "rate": 0.00001469798689280394454312 07
    }
]
```

Ці дані отримуються, обробляються та зберігаються в базі даних, роблячи їх доступними для інших мікросервісів та користувачів платформи.

Клас `DataFetchHandler`, реалізований як функція AWS Lambda, що тригерується CloudWatch Events, ефективно автоматизує процес отримання та

збереження даних про курси криптовалют. Цей сервіс забезпечує актуальність даних на Платформі Аналітики Криптовалютної Біржі, надаючи користувачам надійні та точні дані. Використання змінних оточення, надійної обробки помилок та інтеграція з сервісами AWS демонструє найкращі практики у побудові масштабованих та надійних мікросервісних застосунків у хмарному середовищі.

4.1.2 Реалізація Сервісу Створення Користувачів

Лістинг 4.3 – Код реалізації сервісу «UserHandler»

```
data class CreateUserResponse(val userId: Int?, val error: String?)

class CreateUserHandler : RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> {
    private val logger = LoggerFactory.getLogger(CreateUserHandler::class.java)
    private val dbUrl = System.getenv("DB_URL") ?: throw IllegalArgumentException("DB_URL is not set")
    private val dbUser = System.getenv("DB_USER") ?: throw IllegalArgumentException("DB_USER is not set")
    private val dbPassword = System.getenv("DB_PASSWORD") ?: throw IllegalArgumentException("DB_PASSWORD is not set")
    private val objectMapper = ObjectMapper()
    override fun handleRequest(input: APIGatewayProxyRequestEvent, context: Context): APIGatewayProxyResponseEvent {
        val requestBody = input.body
        val createUserRequest = parseRequest(requestBody, context)
            ?: return createErrorResponse(400, "Invalid input JSON format")
        val response = saveUserToDatabase(createUserRequest)
        return createSuccessResponse(response)
    }
    private fun parseRequest(requestBody: String?, context: Context): CreateUserRequest? {
        return try {
            objectMapper.readValue(requestBody, CreateUserRequest::class.java)
        } catch (e: Exception) {
            context.logger.log("Error parsing input JSON: ${e.message}")
            null
        }
    }
    private fun createErrorResponse(statusCode: Int, message: String): APIGatewayProxyResponseEvent {
        return APIGatewayProxyResponseEvent().apply {
            this.statusCode = statusCode
            body = message
        }
    }
    private fun createSuccessResponse(response: CreateUserResponse): APIGatewayProxyResponseEvent {
        return APIGatewayProxyResponseEvent().apply {
```

```
                statusCode = if (response.userId != null) 200 else 500
                body = objectMapper.writeValueAsString(response)
            }
        }

        private fun saveUserToDatabase(request: CreateUserRequest):
CreateUserResponse {
            val connection: Connection?
            val preparedStatement: PreparedStatement?

            return try {
                connection = DriverManager.getConnection(dbUrl, dbUser,
dbPassword)
                connection.use { conn ->
                    conn.createStatement().use { stmt ->
                        createTablesIfNotExist(stmt)
                    }
                    preparedStatement = conn.prepareStatement(
                        """
                        INSERT INTO users (name, telegram_id, email,
phone_number)
                        VALUES (?, ?, ?, ?) RETURNING id
                        """
                    )
                    preparedStatement.use { stmt ->
                        stmt.setString(1, request.name)
                        stmt.setString(2, request.telegramId)
                        stmt.setString(3, request.email)
                        stmt.setString(4, request.phoneNumber)
                        val resultSet = stmt.executeQuery()
                        if (resultSet.next()) {
                            val userId = resultSet.getInt(1)
                            insertNotifications(conn, userId,
request.notifications)
                            insertTrackedAssets(conn, userId,
request.trackedAssets)
                            CreateUserResponse(userId, null)
                        } else {
                            CreateUserResponse(null, "User creation failed")
                        }
                    }
                }
            } catch (e: SQLException) {
                logger.error("SQL Exception: ${e.message}", e)
                CreateUserResponse(null, "An error occurred: ${e.message}")
            }
        }
        private fun createTablesIfNotExist(statement: PreparedStatement) {
            val createUserTableSQL = """
                CREATE TABLE IF NOT EXISTS users (
                    id SERIAL PRIMARY KEY,
                    name VARCHAR(255) NOT NULL,
                    telegram_id VARCHAR(255),
                    email VARCHAR(255),
                    phone_number VARCHAR(255)
                )
            """
            statement.execute(createUserTableSQL)

            val createNotificationsTableSQL = """
                CREATE TABLE IF NOT EXISTS notifications (
                    user_id INT REFERENCES users(id) ON DELETE CASCADE,
                    channel VARCHAR(255),
                    PRIMARY KEY (user_id, channel)
```

```
                )
            """
            statement.execute(createNotificationsTableSQL)
            val createTrackedAssetsTableSQL = """
                CREATE TABLE IF NOT EXISTS tracked_assets (
                    user_id INT REFERENCES users(id) ON DELETE CASCADE,
                    asset_id_base VARCHAR(255),
                    asset_id_quote VARCHAR(255),
                    PRIMARY KEY (user_id, asset_id_base, asset_id_quote),
                    FOREIGN KEY (asset_id_base, asset_id_quote) REFERENCES
exchange_rates(asset_id_base, asset_id_quote)
                )
            """
            statement.execute(createTrackedAssetsTableSQL)
        }
        private fun insertNotifications(connection: Connection, userId: Int,
notifications: List<String>?) {
            notifications?.let {
                val insertNotificationSQL = "INSERT INTO notifications
(user_id, channel) VALUES (?, ?)"
                connection.prepareStatement(insertNotificationSQL).use { stmt
->
                    for (channel in notifications) {
                        stmt.setInt(1, userId)
                        stmt.setString(2, channel)
                        stmt.addBatch()
                    }
                    stmt.executeBatch()
                }
            }
        }
        private fun insertTrackedAssets(connection: Connection, userId: Int,
trackedAssets: List<TrackedAsset>?) {
            trackedAssets?.let {
                val insertAssetSQL = "INSERT INTO tracked_assets (user_id,
asset_id_base, asset_id_quote) VALUES (?, ?, ?)"
                connection.prepareStatement(insertAssetSQL).use { stmt ->
                    for (asset in trackedAssets) {
                        stmt.setInt(1, userId)
                        stmt.setString(2, asset.assetIdBase)
                        stmt.setString(3, asset.assetIdQuote)
                        stmt.addBatch()
                    }
                    stmt.executeBatch()
                }
            }
        }
    }
    data class CreateUserRequest @JsonCreator constructor(
        @JsonProperty("name") val name: String,
        @JsonProperty("telegramId") val telegramId: String?,
        @JsonProperty("email") val email: String?,
        @JsonProperty("phoneNumber") val phoneNumber: String?,
        @JsonProperty("notifications") val notifications: List<String>?,
        @JsonProperty("trackedAssets") val trackedAssets: List<TrackedAsset>?
    )
    data class TrackedAsset @JsonCreator constructor(
        @JsonProperty("assetIdBase") val assetIdBase: String,
        @JsonProperty("assetIdQuote") val assetIdQuote: String
    )
```

Клас `CreateUserHandler` є важливим компонентом Платформи Аналітики

Криптовалютної Біржі, відповідальним за реєстрацію нових користувачів. Цей сервіс реалізований за допомогою AWS Lambda, що дозволяє обробляти запити на створення нових користувачів та зберігати їх дані в базі даних. Ось детальний опис його призначення та реалізації.

Основна мета сервісу `CreateUserHandler` – забезпечити можливість реєстрації нових користувачів на платформі. Користувачі можуть вказувати свої контактні дані та налаштовувати параметри сповіщень і відслідковуваних активів. Цей сервіс забезпечує збереження всіх цих даних у базі даних для подальшого використання іншими сервісами платформи.

Клас `CreateUserHandler` успадковує `RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent>`, що робить його сумісним з подійно-орієнтованою моделлю виконання AWS Lambda:

1) клас використовує змінні оточення для налаштування параметрів підключення до бази даних (`DB_URL`, `DB_USER`, `DB_PASSWORD`). Це необхідно для безпечного підключення до бази даних;

2) метод `parseRequest` розбирає JSON-тіло запиту, перетворюючи його на об'єкт `CreateUserRequest`. У разі помилки парсингу створюється відповідний об'єкт помилки;

3) збереження Користувача у Базі Даних:

- saveUserToDatabas: Цей метод з'єднується з базою даних та забезпечує збереження даних користувача. Він створює таблиці, якщо вони не існують, і вставляє нові записи у таблиці користувачів, сповіщень та відслідковуваних активів;

- createTablesIfNotExist: Створює необхідні таблиці (`users`, `notifications`, `tracked_assets`), якщо вони ще не існують;

- insertNotifications: Вставляє записи сповіщень у таблицю `notifications`;

- insertTrackedAssets: Вставляє записи відслідковуваних активів у таблицю `tracked_assets`;

4) методи `createErrorResponse` та `createSuccessResponse` генерують

відповідні об'єкти відповіді, які повертаються функцією Lambda. Відповідь включає інформацію про успішне створення користувача або повідомлення про помилку;

5) реалізація включає всебічну обробку помилок для журналювання помилок, що виникають під час запитів до бази даних, забезпечуючи надійність і стійкість системи.

`CreateUserHandler` розгорнуто як функцію AWS Lambda. Lambda надає безсерверну обчислювальну службу, яка виконує функцію у відповідь на події, такі як HTTP-запити, що надходять через API Gateway.

API Gateway використовується для маршрутизації HTTP-запитів до функції Lambda. Це дозволяє викликати функцію Lambda за допомогою REST API-запитів, забезпечуючи інтеграцію з іншими компонентами платформи та зручність використання для клієнтів.

Лістинг 4.5 – Приклад вхідних даних для сервісу UserHandler

```json
{
    "name": "Vladyslav Diulher",
    "telegramId": "valdyslav_diulher_telegram",
    "email": "valdyslav.diulher@example.com",
    "phoneNumber": "1234567890",
    "notifications": ["email", "telegram"],
    "trackedAssets": [
        {
            "assetIdBase": "BTC",
            "assetIdQuote": "USD"
        },
        {
            "assetIdBase": "ETH",
            "assetIdQuote": "USD"
        }
    ]
}
```

Клас `CreateUserHandler`, реалізований як функція AWS Lambda, ефективно автоматизує процес реєстрації нових користувачів на Платформі Аналітики Криптовалютної Біржі. Використання змінних оточення, надійної обробки помилок та інтеграція з API Gateway демонструє найкращі практики у

побудові масштабованих та надійних мікросервісних застосунків у хмарному середовищі.

4.1.3 Реалізація Сервісу Обробки Даних

Лістинг 4.6 – Код реалізації сервісу « DataProcessorHandler»

```
      class DataProcessorHandler : RequestHandler<APIGatewayProxyRequestEvent,
APIGatewayProxyResponseEvent> {
          private    val    dbUrl    =    System.getenv("DB_URL")    ?:    throw
IllegalArgumentException("DB_URL is not set")
          private    val    dbUser    =    System.getenv("DB_USER")    ?:    throw
IllegalArgumentException("DB_USER is not set")
          private  val  dbPassword  =  System.getenv("DB_PASSWORD")  ?:  throw
IllegalArgumentException("DB_PASSWORD is not set")
          private          val          snsClient:          AmazonSNS          =
AmazonSNSClientBuilder.defaultClient()
          private val objectMapper = jacksonObjectMapper()

          private val snsTopics = mapOf(
              "telegram"    to    System.getenv("TELEGRAM_TOPIC")    ?:    throw
IllegalArgumentException("TELEGRAM_TOPIC is not set"),
              "sms"      to      System.getenv("SMS_TOPIC")      ?:      throw
IllegalArgumentException("SMS_TOPIC is not set"),
              "email"     to     System.getenv("EMAIL_TOPIC")     ?:     throw
IllegalArgumentException("EMAIL_TOPIC is not set")
          )

          override    fun    handleRequest(input:    APIGatewayProxyRequestEvent,
context: Context): APIGatewayProxyResponseEvent {
              val connection = getConnection()
              val notifications = try {
                  queryUserNotifications(connection)
              } catch (e: SQLException) {
                  throw RuntimeException("Error querying user notifications", e)
              } finally {
                  connection.close()
              }

              publishNotifications(notifications)

              return APIGatewayProxyResponseEvent().apply {
                  statusCode = 200
                  body = "Notifications published successfully"
              }
          }

          private fun getConnection(): Connection {
              return DriverManager.getConnection(dbUrl, dbUser, dbPassword)
          }

          private    fun    queryUserNotifications(connection:    Connection):
List<UserNotification> {
              val query = """
                  SELECT  u.id  AS  user_id,  u.name  AS  user_name,  n.channel  AS
notification_channel,
```

```
                        COALESCE(u.telegram_id,  u.phone_number,  u.email)  AS
notification_id,
                        ta.asset_id_base, ta.asset_id_quote, er.rate
                FROM users u
                JOIN notifications n ON u.id = n.user_id
                JOIN tracked_assets ta ON u.id = ta.user_id
                JOIN exchange_rates er ON ta.asset_id_base = er.asset_id_base
AND ta.asset_id_quote = er.asset_id_quote
                ORDER BY u.id, n.channel
            """
            val statement = connection.createStatement()
            val resultSet = statement.executeQuery(query)
            val   notifications   =   mutableMapOf<Int,   MutableMap<String,
UserNotification>>()

            while (resultSet.next()) {
                val userId = resultSet.getInt("user_id")
                val userName = resultSet.getString("user_name")
                val channel = resultSet.getString("notification_channel")
                val notificationId = resultSet.getString("notification_id")
                val assetIdBase = resultSet.getString("asset_id_base")
                val assetIdQuote = resultSet.getString("asset_id_quote")
                val rate = resultSet.getDouble("rate")

                val  userNotification  =  notifications.getOrPut(userId)  {
mutableMapOf() }
                    .getOrPut(channel) {
                        UserNotification(userId,  userName,  notificationId,
channel, mutableListOf())
                    }

                userNotification.assets.add(Asset(assetIdBase,  assetIdQuote,
rate))
            }

            resultSet.close()
            statement.close()

            return notifications.flatMap { it.value.values }
        }

        private           fun           publishNotifications(notifications:
List<UserNotification>) {
            notifications.forEach { notification ->
                val message = objectMapper.writeValueAsString(notification)
                snsClient.publish(snsTopics[notification.channel], message)
            }
        }
    }

    data class UserNotification(
        val userId: Int,
        val userName: String,
        val notificationId: String,
        val channel: String,
        val assets: MutableList<Asset>
    )

    data class Asset(
        val assetIdBase: String,
        val assetIdQuote: String,
        val rate: Double
    )
```

Клас `DataProcessorHandler` є ключовим компонентом Платформи Аналітики Криптовалютної Біржі, відповідальним за обробку даних, отриманих з API, та сповіщення користувачів про зміни в курсах криптовалют. Цей сервіс реалізований за допомогою AWS Lambda, що забезпечує безсерверне виконання обчислень, та інтеграції з Amazon SNS (Simple Notification Service) для відправки сповіщень користувачам.

Мета сервісу `DataProcessorHandler` – обробка даних про курси криптовалют, отриманих з зовнішнього API, та надсилання сповіщень користувачам відповідно до їх налаштувань. Сервіс отримує дані про курси, обробляє їх і відправляє сповіщення через різні канали (телеграм, SMS, email).

Клас `DataProcessorHandler` успадковує `RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent>`, що робить його сумісним з подійно-орієнтованою моделлю виконання AWS Lambda:

1) використання змінних оточення для налаштування параметрів підключення до бази даних (`DB_URL`, `DB_USER`, `DB_PASSWORD`) та визначення тем SNS для різних каналів сповіщень (`TELEGRAM_TOPIC`, `SMS_TOPIC`, `EMAIL_TOPIC`);

2) підключення до Бази Даних - метод `getConnection` встановлює підключення до бази даних, використовуючи JDBC;

3) запит Сповіщень Користувачів - метод `queryUserNotifications` виконує SQL-запит, щоб отримати дані про користувачів, їхні налаштування сповіщень та відслідковувані активи, дані організовані в структуру `UserNotification`, що містить інформацію про користувача, канал сповіщень та список активів з відповідними курсами;

4) відправка сповіщень - метод `publishNotifications` перетворює кожне сповіщення в JSON-формат за допомогою `jacksonObjectMapper` і публікує повідомлення в відповідні теми SNS. В залежності від каналу (`telegram`, `sms`, `email`), повідомлення відправляються на відповідні теми SNS, що забезпечує

доставку повідомлень користувачам через обрані ними канали;

5) використання Amazon SNS для відправки сповіщень дозволяє легко масштабувати систему та забезпечує надійну доставку повідомлень через різні канали (телеграм, SMS, email).

Клас `DataProcessorHandler` забезпечує обробку даних про курси криптовалют та сповіщення користувачів відповідно до їхніх налаштувань. Використання AWS Lambda для виконання безсерверних обчислень та інтеграція з Amazon SNS для відправки сповіщень дозволяє легко масштабувати систему та забезпечує надійну доставку повідомлень користувачам через різні канали. Цей підхід демонструє найкращі практики у побудові масштабованих та надійних мікросервісних застосунків у хмарному середовищі.

4.1.3 Реалізація Сервісу Сповіщень Telegram

Лістинг 4.7 – Код реалізації сервісу «TelegramNotificationHandler»

```
    class TelegramNotificationHandler : RequestHandler<SNSEvent, Unit> {
        private val objectMapper = jacksonObjectMapper()
        private val client = OkHttpClient()
        private val telegramBotToken = System.getenv("TELEGRAM_BOT_TOKEN") ?:
throw IllegalArgumentException("TELEGRAM_BOT_TOKEN is not set")

        override fun handleRequest(event: SNSEvent, context: Context) {
            event.records.forEach { record ->
                try {
                    val message = record.sns.message
                    val notification: UserNotification =
objectMapper.readValue(message)
                    sendTelegramMessage(notification)
                } catch (e: Exception) {
                    context.logger.log("Error processing record:
${e.message}")
                }
            }
        }

        private fun sendTelegramMessage(notification: UserNotification) {
            val telegramApiUrl =
"https://api.telegram.org/bot$telegramBotToken/sendMessage"
            val chatId = notification.notificationId
            val text = buildMessageText(notification)

            val requestBody = """
                {
```

```
                    "chat_id": "$chatId",
                    "text": "$text"
                }
            """.trimIndent().toRequestBody("application/json;     charset=utf-
8".toMediaType())

            val request = Request.Builder()
                .url(telegramApiUrl)
                .post(requestBody)
                .build()

            client.newCall(request).execute().use { response ->
                if (!response.isSuccessful) {
                    throw    RuntimeException("Failed    to    send    message:
${response.body?.string()}")
                }
            }
        }

        private fun buildMessageText(notification: UserNotification): String {
            val  assetsInfo  =  notification.assets.joinToString(separator  =
"\n") {
                "Asset:      ${it.assetIdBase}/${it.assetIdQuote},      Rate:
${it.rate}"
            }
            return "Hello ${notification.userName},\n\nHere are your tracked
assets:\n$assetsInfo"
        }
    }

    data class UserNotification(
        val userId: Int,
        val userName: String,
        val notificationId: String,
        val channel: String,
        val assets: List<Asset>
    )

    data class Asset(
        val assetIdBase: String,
        val assetIdQuote: String,
        val rate: Double
    )
```

Клас `TelegramNotificationHandler` є важливим компонентом Coin Rate Платформи, який відповідає за надсилання сповіщень користувачам через Telegram. Цей сервіс реалізовано за допомогою AWS Lambda, що дозволяє автоматизувати процес надсилання сповіщень, та Amazon SNS (Simple Notification Service), який забезпечує розповсюдження повідомлень.

Метою сервісу `TelegramNotificationHandler` є відправка сповіщень користувачам через Telegram у випадках, коли відбуваються зміни в курсах криптовалют, які користувачі відстежують. Сповіщення допомагають користувачам оперативно отримувати інформацію про актуальні курси їхніх

активів.

Клас `TelegramNotificationHandler` успадковує `RequestHandler<SNSEvent, Unit>`, що робить його сумісним з подіями, які генерує Amazon SNS:

1) використовуються змінні оточення для отримання токену бота Telegram (`TELEGRAM_BOT_TOKEN`), що дозволяє надсилати повідомлення через API Telegram;

2) обробка Події - метод `handleRequest` обробляє подію `SNSEvent`, яка містить повідомлення від Amazon SNS. Кожне повідомлення обробляється окремо: зчитується повідомлення та перетворюється у об'єкт `UserNotification`;

3) надсилання Повідомлення через Telegram - метод `sendTelegramMessage` формує та надсилає HTTP POST-запит до API Telegram для відправки повідомлення користувачу. Також використовується бібліотека OkHttp для відправки HTTP-запиту. Також, повідомлення формується у форматі JSON, який містить ID чату (ідентифікатор користувача) та текст повідомлення;

4) метод `buildMessageText` формує текст повідомлення, який містить привітання користувача та список відстежуваних активів з їхніми поточними курсами.

Ось приклад JSON-формату повідомлення, яке надсилається через Telegram:

Лістинг 4.8 – Приклад півдомлення для Telegram

```
```json
{
    "chat_id": "123456789",
    "text": "Hello John Doe,\n\nHere are your tracked assets:\nAsset: BTC/USD, Rate: 35000.5\nAsset: ETH/USD, Rate: 2500.75"
}
```
```

Сервіс `TelegramNotificationHandler` забезпечує ефективне та своєчасне інформування користувачів про зміни в курсах криптовалют через Telegram.

Використання AWS Lambda для безсерверного виконання обчислень та Amazon SNS для розповсюдження повідомлень дозволяє легко масштабувати систему та забезпечити надійну доставку сповіщень користувачам. Цей підхід демонструє ефективне використання хмарних сервісів для побудови сучасних мікросервісних застосунків.

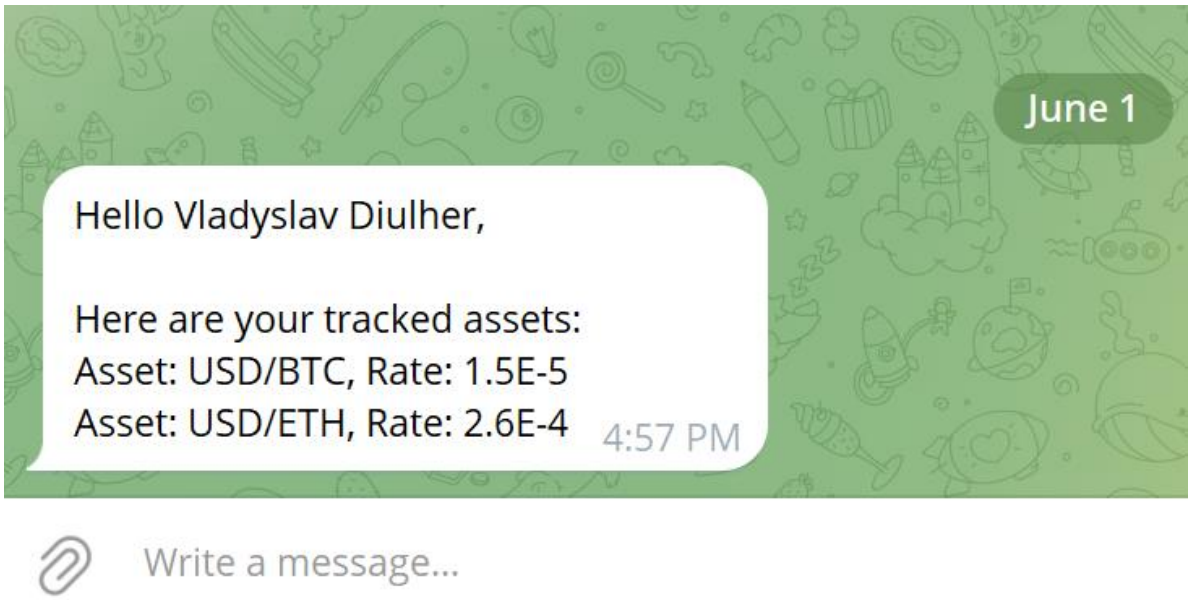

Рисунок 4.1 – Приклад повідомлення від Telegram бота

4.1.4 Роль API Gateway та Load Balancer

У нашій архітектурі API Gateway використовується для управління вхідними запитами користувачів та маршрутизації їх до відповідних мікросервісів. Це забезпечує безпечний і надійний доступ до наших API, дозволяє здійснювати аутентифікацію користувачів та обмежувати кількість запитів. Наприклад, запити до `UserService` або `CoinRateService` проходять через API Gateway, який перевіряє правильність запиту, аутентифікує користувача та пересилає запит до потрібного сервісу.

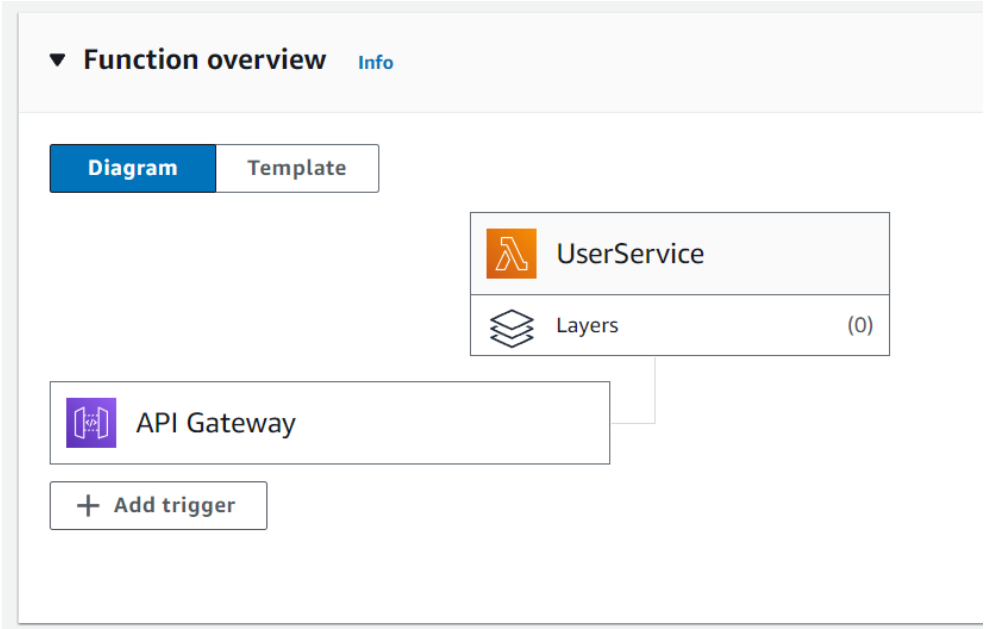

Рисунок 4.2 – API Gateway тригер

Elastic Load Balancer використовується для розподілу трафіку між різними інстансами мікросервісів, таких як `UserService` та `CoinRateService`. Це дозволяє забезпечити рівномірний розподіл навантаження на всі інстанси та підвищити продуктивність системи. Load Balancer також забезпечує високу доступність, автоматично перенаправляючи трафік у разі збою одного з інстансів.

Поєднання API Gateway та Load Balancer дозволяє створити масштабовану, надійну та безпечну архітектуру для нашої Платформи Аналітики Криптовалютної Біржі. API Gateway керує вхідними запитами та забезпечує безпеку, тоді як Load Balancer розподіляє трафік і підвищує доступність сервісів. Це сприяє оптимізації роботи мікросервісів у хмарі та забезпечує якісне обслуговування користувачів.

4.1.5 Використання PostgreSQL для зберігання отриманих та оброблених даних

Для зберігання даних, отриманих і оброблених нашою системою, ми

використовуємо базу даних PostgreSQL. Нижче наведено структуру таблиць, яка відповідає вимогам нашої архітектури.

Лістинг 4.9 – Таблиця `users`

```
- Primary Key: `id`
- Columns:
  - `id SERIAL PRIMARY KEY`
  - `name VARCHAR(255) NOT NULL`
  - `telegram_id VARCHAR(255)`
  - `email VARCHAR(255)`
  - `phone_number VARCHAR(255)`
```

Лістинг 4.10 – Таблиця `notifications`

```
- Primary Key: Composite Key, що складається з `user_id` та `channel`.
- Columns:
  - `user_id INT REFERENCES users(id) ON DELETE CASCADE`
  - `channel VARCHAR(255)`
  - `PRIMARY KEY (user_id, channel)`
```

Лістинг 4.11 – Таблиця `tracked_assets`

```
- Primary Key: Composite Key, що складається з `user_id`, `asset_id_base` та `asset_id_quote`.
- Columns:
  - `user_id INT REFERENCES users(id) ON DELETE CASCADE`
  - `asset_id_base VARCHAR(255)`
  - `asset_id_quote VARCHAR(255)`
  - `PRIMARY KEY (user_id, asset_id_base, asset_id_quote)`
  - `FOREIGN KEY (asset_id_base, asset_id_quote) REFERENCES exchange_rates(asset_id_base, asset_id_quote)`
```

Лістинг 4.12 – Таблиця `exchange_rates`

```
- Primary Key: Composite Key, що складається з `asset_id_base` та `asset_id_quote`.
- Columns:
  - `asset_id_base VARCHAR(255)`
  - `asset_id_quote VARCHAR(255)`
  - `rate DOUBLE PRECISION`
  - `timestamp TIMESTAMP`
  - `PRIMARY KEY (asset_id_base, asset_id_quote)`
```

Причини вибору такого дизайну:

1 нормалізація допомагає уникнути дублювання даних і забезпечити цілісність даних;

2 використання зовнішніх ключів забезпечує референційну цілісність між таблицями;

3 PostgreSQL забезпечує масштабованість за рахунок можливості створення індексів, партиціонування таблиць і використання реплікації;

4 PostgreSQL підтримує виконання складних SQL-запитів, що дозволяє ефективно витягувати необхідні дані для аналітики.

Використання PostgreSQL для зберігання даних в нашій архітектурі забезпечує надійність, масштабованість та гнучкість. Це дозволяє ефективно керувати великими обсягами даних, швидко виконувати запити та забезпечувати цілісність даних. PostgreSQL також легко інтегрується з іншими компонентами нашої системи, що дозволяє забезпечити безперебійну роботу платформи аналітики криптовалютної біржі.

4.2 Використання AWS CloudWatch та X-Ray для моніторингу та трасування сервісів

AWS CloudWatch — це сервіс моніторингу та управління, який дозволяє збирати та відслідковувати метрики, збирати та моніторити файли журналів, налаштовувати оповіщення та автоматично реагувати на зміни у вашому середовищі.

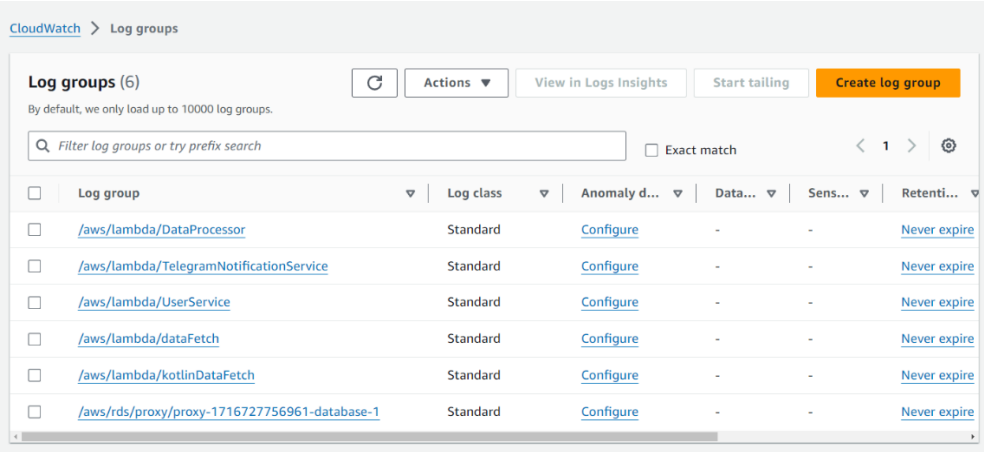

Рисунок 4.3 – Групи логів CloudWatch

1 Моніторинг метрик:

- ми налаштували CloudWatch для збору метрик з наших сервісів, таких як використання CPU, пам'яті, кількість запитів до API, час відгуку сервісів та кількість помилок;

- ці метрики дозволяють нам отримувати уявлення про продуктивність наших сервісів у реальному часі.

2 Журнали (Logs):

- логи з наших AWS Lambda функцій автоматично надсилаються до CloudWatch Logs;

- ми налаштували групи логів та створили правила фільтрації для виявлення специфічних подій або помилок.

3 Оповіщення (Alarms):

- налаштовані оповіщення CloudWatch попереджають нас про критичні події, такі як високий рівень помилок або перевищення лімітів використання ресурсів;

- оповіщення можуть бути налаштовані для відправлення повідомлень через Amazon SNS, електронну пошту або інші засоби зв'язку.

Лістинг 4.9 – Приклад метрик CloudWatch

```plaintext
- Invocation count: 500 invocations/hour
- Error count: 5 errors/hour
- Average latency: 200ms
- Memory usage: 128MB
```

AWS X-Ray дозволяє нам аналізувати та налагоджувати розподілені програми. Це допомагає виявляти проблеми в сервісах та відстежувати запити через всю архітектуру мікросервісів.

1 Трасування запитів:

- всі запити до наших сервісів трасуються за допомогою X-Ray, що дозволяє бачити кожен етап обробки запиту;

- ми налаштували Lambda функції для інтеграції з X-Ray, що дозволяє автоматично трасувати всі запити.

2 Аналіз продуктивності:

- X-Ray надає детальні дані про час виконання кожного сегмента запиту, що дозволяє виявляти затримки та неефективності;

- ми використовуємо ці дані для оптимізації коду та налаштувань інфраструктури.

3 Виявлення вузьких місць (bottlenecks):

- аналіз трасування допоміг нам виявити проблемні місця, такі як повільні запити до бази даних або висока затримка мережі;

- ми провели оптимізацію, наприклад, шляхом використання кешування, оптимізації запитів до бази даних та налаштування пулу з'єднань.

Лістинг 4.13 – Приклад даних трасування X-Ray

```plaintext
- Total request time: 500ms
- Lambda execution time: 200ms
- Database query time: 150ms
- External API call time: 100ms
- Overhead: 50ms
```

Використання AWS CloudWatch та X-Ray дозволяє нам ефективно моніторити наші сервіси, виявляти та усувати проблеми продуктивності. Це допомагає забезпечити стабільну та надійну роботу нашої платформи аналітики криптовалют. Завдяки цим інструментам ми можемо швидко реагувати на проблеми та оптимізувати наші сервіси для досягнення кращих результатів.

4.3 Використання VPC у сервісах

Amazon VPC дозволяє запускати AWS ресурси у віртуальній мережі, яку ви визначаєте. Вона забезпечує контроль над мережею та ізоляцію ресурсів, що є критичним для безпеки та продуктивності додатків.

1 Ізоляція ресурсів:

- ми створили окрему VPC для нашої платформи аналітики криптовалют, що дозволяє ізолювати наші ресурси від інших проектів та середовищ;

- використання приватних і публічних підмереж для забезпечення безпеки

та доступу до ресурсів.

2 Безпека:

- використання правил безпеки (Security Groups) та списків контролю доступу (NACLs) для контролю трафіку між ресурсами;

- лямбда-функції, бази даних і інші ресурси розміщуються в приватних підмережах, де до них неможливо отримати доступ безпосередньо з інтернету;

- публічні підмережі використовуються для ресурсів, які потребують доступу до інтернету, таких як API Gateway.

3 З'єднання з іншими сервісами:

- використання Amazon VPC Endpoints для безпечного з'єднання з іншими сервісами AWS, такими як S3 та DynamoDB, без виходу в інтернет;

- використання NAT Gateway для надання лямбда-функціям та іншим приватним ресурсам доступу до інтернету.

4 Трасування запитів та моніторинг:

- всі ресурси в VPC інтегровані з AWS CloudWatch та X-Ray для моніторингу та трасування запитів;

- це дозволяє виявляти проблеми мережевого трафіку, такі як затримки або втрати пакетів, і оперативно їх усувати;

- лямбда-функції мають Security Groups, що дозволяють трафік тільки від інших внутрішніх ресурсів або через VPC Endpoints;

- бази даних мають Security Groups, що дозволяють трафік тільки від лямбда-функцій;

- NAT Gateway розташована в публічній підмережі та забезпечує вихід лямбда-функцій у інтернет для виконання зовнішніх запитів, таких як доступ до зовнішніх API.

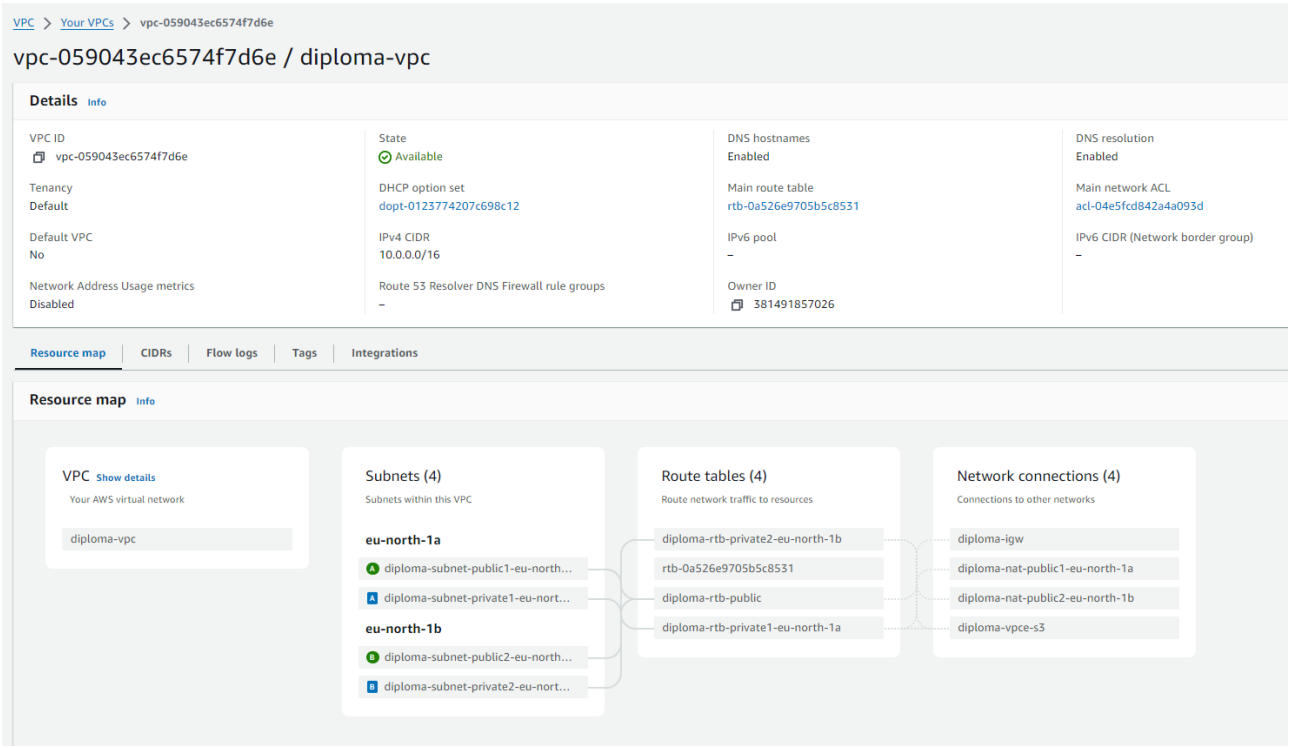

Рисунок 4.4 – Налаштований VPC

Використання VPC дозволяє нам забезпечити високий рівень безпеки та ізоляції для наших сервісів. Це важливо для захисту чутливих даних і забезпечення безперебійної роботи платформи аналітики криптовалют. Завдяки продуманій конфігурації підмереж, правил безпеки та використанню VPC Endpoints, ми можемо забезпечити ефективне та безпечне середовище для виконання наших сервісів.

4.4 Використання Amazon S3 для зберігання jar-файлів Kotlin Lambda

Amazon Simple Storage Service (S3) є надійним та масштабованим сховищем об'єктів, що ідеально підходить для зберігання різних файлів, включаючи jar-файли для AWS Lambda. Використання S3 для зберігання jar-файлів дозволяє легко завантажувати, оновлювати та розгортати функції Lambda безпосередньо з хмари.

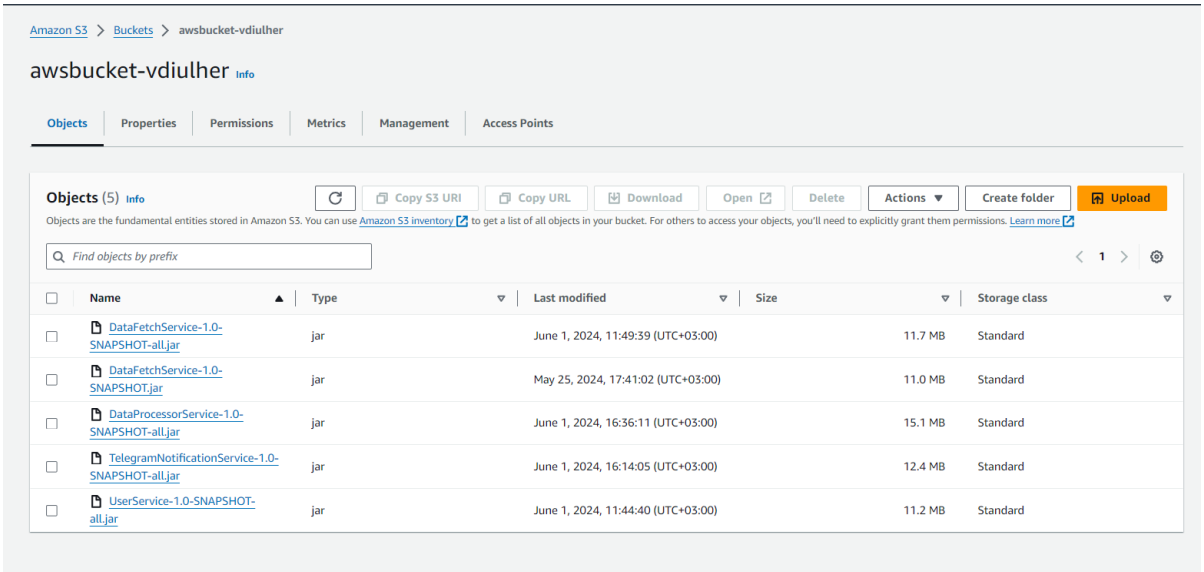

Рисунок 4.5 – S3 bucket з файлами проекту

Як ми використовуємо S3 для зберігання jar-файлів для наших Kotlin Lambda.

1 Завантаження jar-файлів на S3:

- під час процесу розробки та побудови ми створюємо jar-файли для наших Kotlin Lambda функцій;

- ці jar-файли завантажуються до певного бакету S3 за допомогою AWS CLI, SDK або CI/CD системи, такої як Jenkins або GitHub Actions.

2 Бакети S3 налаштовані з увімкненим версіонуванням, що дозволяє зберігати кілька версій одного jar-файлу. Це дозволяє нам відслідковувати зміни та відновлювати попередні версії у разі потреби.

3 Ми налаштовуємо політики доступу S3, щоб забезпечити безпеку та обмежити доступ тільки до певних IAM ролей або користувачів. Це запобігає несанкціонованому доступу до наших jar-файлів.

4 Розгортання Lambda з S3:

- коли ми створюємо або оновлюємо Lambda функцію, ми вказуємо S3 шлях до jar-файлу. AWS Lambda автоматично завантажує jar-файл з S3 при створенні або оновленні функції;

- це здійснюється за допомогою AWS Management Console, CLI або

інфраструктури як код (IaC) інструментів, таких як AWS CloudFormation або Terraform.

Використання Amazon S3 для зберігання jar-файлів для Kotlin Lambda функцій забезпечує простоту керування та розгортання, надійність зберігання та безпеку доступу. Інтеграція S3 з AWS Lambda дозволяє легко оновлювати функції, мінімізуючи зусилля, необхідні для керування інфраструктурою. Завдяки цьому підходу ми можемо ефективно керувати нашими Lambda функціями, забезпечуючи високу продуктивність та надійність нашої платформи аналітики криптовалют.

4.5 Аналіз досягень

У цьому практичному розділі ми розглянули розробку та впровадження платформи аналітики криптовалют, зосереджуючись на оптимізації та інтеграції мікросервісів у хмарі. Основні пункти, які ми висвітлили:

- ми створили платформу, яка складається з декількох мікросервісів для збору, обробки та повідомлення даних про курси криптовалют;
- використання AWS Lambda для обробки даних і відправки сповіщень дозволило нам досягти масштабованості та гнучкості;
- дані зберігалися в базі даних Postgres, що забезпечує надійне та структуроване збереження інформації;
- реалізовано сервіс для регулярного збору даних про курси криптовалют через API та збереження їх у базі даних;
- використання AWS CloudWatch Events для планування завдань збирання даних дозволило автоматизувати цей процес;
- впроваджено сервіс для обробки зібраних даних та відправлення сповіщень користувачам через різні канали (Telegram, SMS, Email) за допомогою AWS SNS;
- реалізовано окремі Lambda функції для кожного каналу сповіщень, що забезпечує модульність і легкість підтримки;

- використання AWS CloudWatch та X-Ray для моніторингу і відстеження запитів дозволило нам швидко ідентифікувати та усунути вузькі місця у системі;

- аналіз зібраних метрик допоміг оптимізувати продуктивність та зменшити затримки.

Ключові висновки та результати:

- використання AWS Lambda та SNS дозволило створити гнучку, масштабовану систему, яка легко адаптується до змін у навантаженні;

- система демонструє високу продуктивність і стійкість завдяки правильній архітектурі та оптимізації;

- використання AWS CloudWatch для планування та моніторингу забезпечило високу надійність і автоматизацію процесів збирання та обробки даних;

- інтеграція з S3 для зберігання jar-файлів Lambda функцій спростила розгортання та оновлення функцій.

4.6 Виклики та їх подолання

1 Інтеграція Мікросервісів:

- виклик: Забезпечення ефективної взаємодії між різними мікросервісами;

- рішення: Використання AWS SNS для координації та відправки сповіщень, що дозволило забезпечити надійну інтеграцію та мінімізувати залежності між сервісами.

2 Оптимізація продуктивності:

- виклик: Виявлення та усунення вузьких місць у системі;

- рішення: Використання AWS CloudWatch та X-Ray для моніторингу продуктивності, що дозволило ідентифікувати проблеми та впровадити оптимізації.

Потенційні покращення та розширення.

1 Машинне Навчання для Прогнозування:

- впровадження алгоритмів машинного навчання для прогнозування змін курсів та надання аналітичних прогнозів користувачам.

2 Покращення безпеки:

- додаткові заходи безпеки для захисту даних та забезпечення конфіденційності користувачів.

У підсумку, розробка платформи аналітики криптовалют дозволила нам не лише створити потужний інструмент для моніторингу та аналізу ринкових даних, але й здобути цінний досвід у оптимізації та інтеграції мікросервісів у хмарному середовищі.

ВИСНОВКИ

У ході роботи було успішно розроблено та впроваджено платформу для аналітики обмінних курсів криптовалют, що базується на мікросервісній архітектурі та використовує хмарні обчислення. Основною метою було створення масштабованої, надійної та продуктивної системи, яка забезпечує швидкий та ефективний збір, обробку та аналіз великих обсягів даних.

Однією з головних переваг розробленої платформиє використання AWS Lambda для реалізації серверлесс функцій, що дозволяє автоматично масштабувати систему відповідно до навантаження, знижуючи витрати на інфраструктуру та забезпечуючи високу продуктивність. AWS API Gateway було використано для створення та управління RESTful API, що забезпечує ефективний інтерфейс між мікросервісами та зовнішніми клієнтами.

Зберігання даних було реалізовано на базі Postgres, що дозволило ефективно обробляти та зберігати великі обсяги структурованої інформації. Для моніторингу та трасування запитів використовувалися AWS CloudWatch та X-Ray, що дозволило забезпечити високу надійність та своєчасне виявлення та усунення проблем у роботі системи.

Результати впровадження платформи показали її високу ефективність у зборі та обробці даних, що дозволяє користувачам отримувати актуальну інформацію про обмінні курси криптовалют та акцій у реальному часі. Система продемонструвала високу стабільність та надійність під час тестування, забезпечуючи безперебійну роботу навіть при значних навантаженнях.

Під час розробки було подолано низку технічних викликів, зокрема забезпечення безперебійної роботи мікросервісів, оптимізація процесів обробки даних та інтеграція різних компонентів системи. Це дозволило створити гнучку та масштабовану платформу, здатну адаптуватися до змінних вимог користувачів та зростаючих обсягів даних.

Основні висновки з реалізації проекту включають важливість вибору

відповідних технологій для забезпечення масштабованості та надійності системи, а також необхідність ретельного моніторингу та оптимізації всіх її компонентів. Успішна інтеграція AWS Lambda, API Gateway, CloudWatch та X-Ray дозволила досягти поставлених цілей та забезпечити високий рівень продуктивності та надійності платформи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Мікросервіси: Архітектура для розробки сучасних подільних систем / Н. Ньюмен; пер. з англ. – К. : Дія, 2019. – 280 с.

2. Обчислення в хмарах: принципи, системи та застосування / Г. Хотенія, Кадір Оакалаг ; пер. з англ. – К. : Техніка, 2018. – 392 с.

3. Проектування і декомпозиція мікросервісів: практичний підхід / С. Річардс; пер. з англ. – К. : Наук. думка, 2021. – 216 с.

4. Хмарні технології: Архітектура та розробка хмарних застосунків / Л. Вашкевич; пер. з англ. – К. : Наук. думка, 2020. – 352 с.

5. Хмароорієнтовані мережі: Архітектура, послуги та управління / П. Мелл, Т. Гренс; пер. з англ. – К. : Дія, 2019. – 416 с.

6. Сучасні комп'ютерні системи: Масштабування, мережі та хмарні технології / В. Гординський ; пер. з англ. − К. : Видавничий дім "Професіонал", 2017. − 672 с.

7. Адаптивний керований управлінський процесс інтеграції та оптимізації мікросервісів в хмарному середовищі / А. Сахаров ; пер. з англ. − К. : Видавничий дім "Професіонал", 2019. − 300 с.

8. Розробка мікросервісів: Створення, випробування та впровадження масштабованих розподілених систем / С. Ньюман ; пер. з англ. – К. : Техніка, 2018. – 312 с.

9. Програмування хмарних застосунків: Кращі практики для великомасштабних служб / Д. Вілдермут; пер. з англ. – К. : Дія, 2020. – 392 с.

10. Модель типових служб хмарних обчислень для навчальних цілей / О. Шульга; пер. з англ. – К. : Наук. думка, 2020. – 192 с.