Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії на управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти перший (бакалаврський)

Метод забезпечення відмовостійкості мікросервісів у AWS

(тема)

Виконав:
здобувач 3 року навчання,
групи КІУКІу-22-2
Олександр ТКАЧЕНКО
(власне ім’я, прізвище)

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

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

Освітня програма
Комп’ютерна інженерія
(повна назва освітньої програми)

Керівник: ас. Ірина ЧЕПУРНА
(посада, власне ім’я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ _____ Андрій КОВАЛЕНКО
(підпис) (власне ім’я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії на управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти перший (бакалаврський)

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

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

Освітня програма Комп’ютерна інженерія
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
“____” _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Ткаченку Олександру Віталійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Метод забезпечення відмовостійкості мікросервісів у AWS

затверджена наказом по університету від “ 26 ” травня 2025 р. № 425Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 14 липня 2025 р.

3. Вхідні дані до роботи 1)хмарні платформи: AWS$ 2) сервіси AWS: API Gateway; Lambda, CloudWatch, IAM, DynamoDB, ElactiCashe, Redis; 3)методи забезпечення відмовостійкості: балансування навантаження, застосування CDN, масштабування, кешування; 4) методи тестування та аналітичного моделювання роботи вебдодатку з мікросервісною архітектурою.

4. Перелік питань, що потрібно опрацювати у роботі
1) аналіз предметної області
2) методи обробки запитів у хмарних мікросервісах
3) розробка методу забезпечення відмовостійкості хмарних мікросервісів на AWS
4) моделювання методу
5) висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій

Слайд-презентація – 10 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	Строк / терміни виконання етапів роботи	Примітка
	Аналіз проблеми та огляд існуючих рішень	10.06.25-13.06.25	
	Вибір технології розробки та інструментальних засобів	14.06.25-16.06.25	
	Розробка та тестування методу	17.06.25-30.06.25	
	Оформлення матеріалів кваліфікаційної роботи	01.07.25-09.07.25	
	Подання кваліфікаційної роботи керівникові	10.07.25-09.07.25	
	Підготовка презентації та попередній захист	10.07.25 – 11.07.25	
	Подання кваліфікаційної роботи на рецензування	12.07.25-13.07.25	

Дата видачі завдання " 09 " червня 2025 р.

Здобувач _____ (підпис)

Керівник роботи _____ ас. Ірина ЧЕПУРНА
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 53 с., 13 рис., 1 табл., 3 дод., 29 джерел.

МІКРОСЕРВІСИ, ХМАРНІ ОБЧИСЛЕННЯ, AWS, ВІДМОВОСТІЙКІСТЬ, МАСШТАБОВАНІСТЬ

Метою кваліфікаційної роботи є розробка методу забезпечення відмовостійкості мікросервісів у середовищі AWS, який забезпечує стабільну та безперервну роботу вебдодатків з мікросервісною архітектурою та мінімізацію затримок в умовах динамічно змінного навантаження.

В ході виконання роботи проведено аналіз сучасних методів забезпечення відмовостійкості у хмарних мікросервісах, зокрема розглянуто підходи до розподіленого кешування, реплікації даних, а також застосування механізмів балансування навантаження та автоматичного масштабування. Особливу увагу приділено інтеграції сервісів AWS з безсерверними обчисленнями на базі Lambda.

Отримані результати підтверджують доцільність використання хмарної інфраструктури для забезпечення стабільної та безперервної роботи вебдодатків з мікросервісною архітектурою.

ABSTRACT

Bachelor`s thesis: 53 pages, 13 figures, 1 table, 3 appendices, 29 sources.

MICROSERVICE, CLOUD COMPUTING, AWS, FAULT RESILIENCE, SCALABILITY

The major goal of this thesis is to develop a method for ensuring the fault tolerance of microservices within the AWS environment, which guarantees stable and continuous operation of web applications based on microservice architecture, while minimizing latency under conditions of dynamically changing workloads.

In order to achieve this goal, the thesis analyzes contemporary approaches to fault tolerance in cloud-based microservices, focusing on distributed caching, data replication, as well as the implementation of load balancing and automatic scaling mechanisms. Special attention is paid to the integration of AWS services with serverless computing based on Lambda functions.

The results obtained confirm the feasibility and effectiveness of using cloud infrastructure to ensure stable and uninterrupted operation of web applications utilizing a microservice architecture.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

API – програмний інтерфейс прикладного рівня (англ. Application Programming Interface)

ARN – уніфіковане ім’я ресурсу в AWS (англ. Amazon Resource Name)

AWS – хмарна платформа Amazon (англ. Amazon Web Services)

AWS SDK – набір інструментів розробника для роботи з сервісами AWS (англ. AWS Software Development Kit)

AZ – зона доступності в хмарній інфраструктурі (англ. Availability Zone)

CDN – мережа доставки контенту (англ. Content Delivery Network)

CORS – політика кросдоменної взаємодії (англ. Cross-Origin Resource Sharing)

CPU – центральний процесор (англ. Central Processing Unit)

ECS – сервіс керування контейнерами в AWS (англ. Elastic Container Service)

FaaS – модель безсерверних обчислень «Функція як сервіс» (англ. Function as a Service)

FIFO – структура черги «перший прийшов – перший обслуговується» (англ. First In – First Out)

GCP – хмарна платформа Google (англ. Google Cloud Platform)

HTTP – протокол передачі гіпертексту (англ. HyperText Transfer Protocol)

HTTPS – захищений протокол передачі гіпертексту (англ. HyperText Transfer Protocol Secure)

IAM – сервіс управління ідентифікацією та доступом (англ. Identity and Access Management)

JWT – токен для передачі підтверджених даних користувача (англ.

JSON Web Token)

KMS – сервіс керування ключами шифрування в AWS (англ. Key Management Service)

NoSQL – тип баз даних без табличної структури (англ. Not Only SQL)

OAuth – протокол авторизації з делегованим доступом (англ. Open Authorization)

REST – архітектурний стиль взаємодії між компонентами (англ. Representational State Transfer)

S3 – сервіс об’єктного зберігання в AWS (англ. Simple Storage Service)

SNS – сервіс розсилки повідомлень в AWS (англ. Simple Notification Service)

SQS – сервіс черг повідомлень в AWS (англ. Simple Queue Service)

SSL –протокол захисту передавання даних (англ. Secure Sockets Layer)

TLS – протокол захисту передавання даних (англ. Transport Layer Security)

TTL – час життя пакета або кешу (англ. Time To Live)

URI – уніфікований ідентифікатор ресурсу (англ. Uniform Resource Identifier)

VPC – ізольована віртуальна мережа в AWS (англ. Virtual Private Cloud)

VPN – віртуальна приватна мережа (англ. Virtual Private Network)

ВСТУП

Сучасні інформаційні технології активно переходять до хмарних обчислень, що забезпечують гнучкість, масштабованість та доступність сервісів. Завдяки своїй ефективності, мікросервісна архітектура стала стандартним підходом для розробки складних розподілених систем, оскільки вона дозволяє виділяти функціональні компоненти в незалежні сервіси. Проте розподілені сервіси стикаються з низкою проблем, зокрема зростає складність управління відмовами, збільшується ймовірність помилок, а також виникають затримки при взаємодії між сервісами. З огляду на стрімке зростання онлайн-додатків та сервісів до розподілених систем висуваються високі вимоги до надійної та неперервної роботи через необхідність забезпечення високої якості обслуговування користувачів.

Хмарні платформи стають ефективним рішенням для розгортання мікросервісної архітектури завдяки широкому спектру інструментів і сервісів, що підтримують створення та управління масштабованими та відмовостійкими додатками. Такі розподілені системи забезпечують можливість використання контейнеризації, безсерверних обчислень, автоматичного масштабування та моніторингу, що підвищує доступність і стійкість до збоїв. Проте для гарантування стабільної та безперервної роботи мікросервісів необхідно впроваджувати механізми резервування компонентів і балансування навантаження, які сприяють зниженню затримок при обробці запитів і забезпечують відмовостійкість мікросервісної архітектури.

Метою даної роботи є розробка методу забезпечення відмовостійкості мікросервісів в середовищі AWS, який дозволяє гарантувати стабільне функціонування додатків в умовах високого навантаження, відмов окремих компонентів або кіберзагроз. Запропонований підхід має забезпечити високу доступність, безперервність та надійність роботи сервісів, побудованих на основі мікросервісної архітектури.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Особливості мікросервісної архітектури

В умовах стрімкого розвитку інформаційних технологій та збільшення попиту на високонадійні та масштабовані мережні інфраструктури, хмарні платформи, зокрема Amazon Web Services (AWS), Google Cloud Platform (GCP) та Microsoft Azure, набувають все більшої популярності. Одним з основних напрямків у розвитку хмарних рішень є розгортання мікросервісної архітектури, що дозволяє створювати гнучкі, масштабовані та незалежні сервіси, кожен з яких виконує конкретну функцію в межах загальної системи.

Особливої актуальності хмарні технології набувають в зв’язку зі зростаючими вимогами до безперервності роботи, високої доступності та швидкої адаптації до змін у бізнес-середовищі.

Одним з найбільш ефективних архітектурних підходів в побудові сучасних розподілених систем є мікросервісна архітектура. Вона дозволяє розділити складні додатки на низку автономних сервісів, кожен з яких реалізує окрему бізнес-функцію та може масштабуватися та обслуговуватися незалежно від інших компонентів (рисунок 1.1).

Хмарні сервіси надають широкий спектр переваг, які суттєво сприяють ефективному впровадженню мікросервісної архітектури. В порівнянні з традиційною монолітною архітектурою, мікросервісний підхід демонструє низку важливих переваг, що зумовлюють його зростаючу популярність в сучасній програмній інженерії:

- гнучкість, що полягає в спрощенні процесу оновлення окремих компонентів програмного забезпечення, та забезпечує прискорення циклів розробки та впровадження нових функцій;

- кросплатформність, що дає змогу створювати вебзастосунки з використанням різноманітних програмних середовищ і мов програмування, сприяючи незалежності окремих сервісів;

- підвищений рівень безпеки, який забезпечується завдяки можливостям управління та моніторингу мережної інфраструктури за допомогою вбудованих інструментів, інтерфейсів та сервісів, що надаються хмарною платформою;

- стабільність та відмовостійкість, що досягаються через застосування механізмів балансування навантаження, засобів оркестрації, а також технологій віртуалізації й контейнеризації, які сприяють забезпеченню безперервної та стабільної роботи програмного забезпечення.

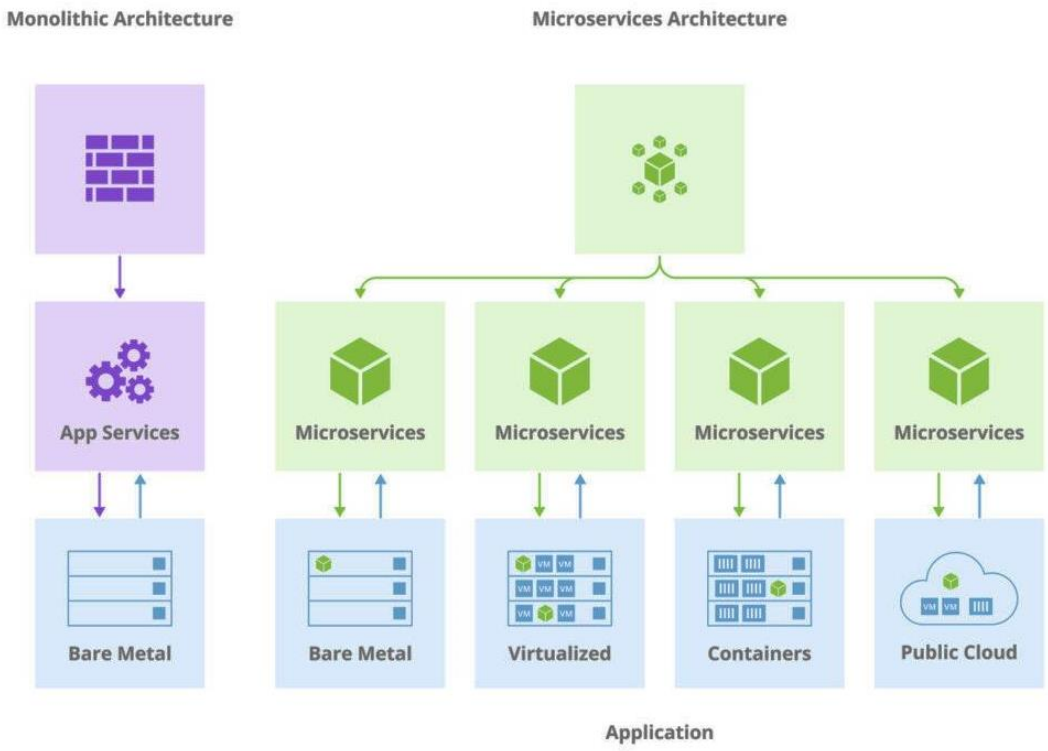

Рисунок 1.1 – Схема порівняння монолітної та мікросервісної архітектури

Порівняльні характеристики монолітної та мікросервісної архітектури наведені в таблиці 1.1, що дозволяє визначити основні причини, через які монолітна архітектура стає менш ефективна через вимоги до масштабованості, попри простоту реалізації та централізоване управління.

Таблиця 1.1 – Порівняльні характеристики архітектур для вебдодатків

Категорія	Мікросервісна архітектура	Монолітна архітектура
Тип архітектури	розподілена система, що складається з множини автономних, взаємодіючих сервісів	уніфікована система з централізованою кодовою базою, яка охоплює всю функціональність
Зв’язність компонентів	висока ізольованість; взаємодія реалізується через API	тісна інтеграція компонентів з прямими міжкомпонентними залежностями
Процес розгортання	незалежне розгортання окремих функціональних одиниць	розгортання всієї системи як єдиного програмного блоку
Технологічний стек	можливість використовувати гетерогенні технології для реалізації різних сервісів	Обмеження уніфікованим технологічним стеком, спільним для всієї системи
Масштабованість	незалежне масштабування окремих сервісів залежно від навантаження	Масштабування можливо лише на рівні всієї програми
Стійкість до відмов (збоїв)	відмова окремого сервісу ізольована і не впливає на інші компоненти	Вихід з ладу одного з компонентів може спричинити зупинку функціонування всієї системи

Ключова відмінність між традиційними монолітними системами та мікросервісами полягає в їх підході до розробки додатків, оскільки мікросервіси розділяють функціонал на окремі об'єкти, які ефективно співпрацюють разом, що зазначено в роботі [1]. Сервіси працюють незалежно для досягнення окремих функцій, водночас зберігаючи можливість для відокремленої розробки та розгортання.

Кожен мікросервіс виконує чітко визначену задачу і має власний набір залежностей, що забезпечує гнучкість у виборі технологій. Завдяки цьому внесення змін в один сервіс не потребує оновлення всієї системи, що пришвидшує розробку та спрощує підтримку [2].

Ізоляція мікросервісів досягається за допомогою контейнеризації або віртуалізації, що гарантує незалежність середовища виконання для кожного сервісу. Кожний сервіс має власний життєвий цикл і ресурси; за потреби він може масштабуватися, незалежно від інших компонентів, на що акцентовано увагу в роботі [3]. Взаємодія між мікросервісами здійснюється через інтерфейси. Такий підхід забезпечує слабке зв’язування компонентів і дає змогу безперешкодно інтегрувати нові сервіси в систему, що зазначено в роботі [4].

Для масштабування мікросервісних додатків використовуються горизонтальне додавання екземплярів сервісу та балансування навантаження. Платформи оркестрації контейнерів, зокрема Kubernetes, Amazon ECS, автоматично розгортають додаткові контейнери при зростанні кількості звернень, що дозволяє ефективніше використовувати ресурси, та запобігти втрат у швидкості та доступності. Масштабування також може здійснюватися автоматично або на основі метрик продуктивності, що сприяє адаптивній роботі системи, крім ручного налаштування, як зазначається в роботі [5].

Безпека в мікросервісній архітектурі гарантується окремими механізмами аутентифікації та авторизації для кожного сервісу. Для цього використовуються міжсервісні токени, зокрема JWT, шифрування

транспортного каналу для захисту даних при передачі, а також мережне сегментування, зокрема використання VPN, правил доступу, на що вказано у роботах [6, 7]. Таким чином, компрометація одного сервісу не відкриває вільного доступу до всієї системи. Також потрібно застосовувати регулярний аудит і моніторинг дій користувачів, використовуючи інструменти AWS IAM, CloudWatch, зокрема платформи AWS, що підвищує загальний рівень захисту, що зазначено в роботі [8].

Відмовостійкість в мікросервісній архітектурі забезпечується використанням механізмів автоматичного відновлення, простого перезапуску сервісів після збоїв, дублюванням критичних компонентів та черги повідомлень для перекриття затримок, що описано в роботі [9]. Використання обробки виключних ситуацій дозволяє ізолювати відмови та мінімізувати помилки, обмежуючи збої в мережі.

Моніторинг стану сервісів та логування допомагають швидко виявляти проблеми та оперативно реагувати на зміни в показниках, забезпечуючи високу стійкість всієї системи, як зазначено в роботі [10].

1.2 Архітектура AWS для мікросервісів

З огляду на високі вимоги до адаптивності, масштабованості та безперервної доступності сучасних розподілених систем, використання хмарних платформ для розгортання мікросервісної архітектури набуває дедалі більшої актуальності. Серед відомих хмарних платформ, які надають широкий набір керованих сервісів для побудови, запуску та масштабування мікросервісів із вбудованою підтримкою високої доступності та стійкості до збоїв, найбільш популярною є AWS, яка задовольняє потреби в розгортанні сервісів компаній малого та середнього бізнесу.

Одним з зовнішніх сервісів, що зменшує затримки та навантаження на систему, є Cloudflare CDN – глобальна мережа доставки контенту та забезпечення високого рівня безпеки мережної інфраструктури. Cloudflare

має можливість кешувати статичний контент на своїх серверах якомога ближче до користувачів, що дозволяє швидше доставляти ресурси веб додатків та веб-сайтів, знижуючи затримки в обробці запитів. Ще одна перевага застосування Cloudflare полягає в забезпеченні захисту від DDoS-атак та фільтрації вхідного трафіку на прикладному рівні, що зазначено в роботі [11].

В хмарному середовищі AWS для обробки зовнішніх запитів використовується сервіс Amazon API Gateway, який забезпечує експлуатацію, моніторинг т а безпеку API будь-якого масштабу. API Gateway виступає в ролі єдиного інтерфейсу до мікросервісної архітектури, яка приймає HTTP/HTTPS запити від користувачів, здійснюючи автентифікацію та авторизацію, зокрема за допомогою AWS IAM, OAuth або ключів доступу, а також подальшу маршрутизацію до кінцевих сервісів, зокрема AWS Lambda, Amazon ECS та інші, що зазначено в роботі [12]. Використання Amazon API Gateway дозволяє ефективно керувати навантаженням, забезпечуючи високу продуктивність та безпеку системи.

AWS Lambda представляє собою безсерверну обчислювальну платформу, що дозволяє запускати код у відповідь на події без необхідності керування серверами. Lambda-функції запускаються миттєво при надходженні запитів або подій, а кількість екземплярів автоматично масштабується залежно від навантаження [13].

Функціонування AWS Lambda базується на концепції функцій як сервісу (FaaS), де кожна функція виконується як окремий фрагмент коду для визначеного завдання. AWS Lambda підтримує різні мови програмування, та визначає умови його активації, зокрема HTTP-запит, зміну об’єкта в Amazon S3, повідомлення з черги Amazon SQS, подію в DynamoDB, та автоматично виконує його у відповідь на цю подію. До основних переваг AWS Lambda належать автоматичне масштабування у відповідь на збільшення навантаження, оплата лише за використання, що дозволяє знизити витрати, висока доступність, що забезпечує відмовостійкість, та інтеграція з різними

сервісами, зокрема Amazon API Gateway, S3, DynamoDB, SNS, CloudWatch, що дозволяє використовувати її в мікросервісній архітектурі [14].

Для прискореного доступу до даних в AWS застосовується сервіс ElastiCache (Redis), що працює як кеш-пам’ять. Redis дозволяє зберігати тимчасові дані та відповіді запитів у оперативній пам’яті, що значно скорочує затримки і знижує навантаження на базу даних, що відображено в роботі [15].

Основні дані зберігаються в Amazon DynamoDB – повністю керованій NoSQL базі даних, яка забезпечує низьку латентність запитів та автоматичне горизонтальне масштабування. DynamoDB використовується для довготривалого зберігання інформації:, зокрема сесій користувачів, метаданих, транзакцій, коли потрібна висока доступність та відмовостійкість. Завдяки можливості реплікації даних в декількох зонах доступності, DynamoDB гарантує збереження інформації навіть за відмови окремих серверів, що вказано в роботі [16].

Ідентифікація та контроль доступу в AWS здійснюються за допомогою сервісу IAM, який дозволяє створювати користувачів, ролі та привілеї для кожного користувача відповідно до його ролі. Кожен мікросервіс або Lambda-функція виконується з визначеною IAM-роллю, яка обмежує доступ до інших ресурсів, наприклад, до певних таблиць DynamoDB або AWS S3. Також в хмарній платформі AWS використовується підтримка шифрування даних та захищеної передачі за протоколом TLS за допомогою сервісу AWS KMS. Поєднання цих механізмів гарантує, що в разі компрометації окремого компонента буде знижено ризик витоку чи модифікації конфіденційної інформації, що зазначено авторами роботи [17].

Моніторинг та логування в AWS здійснюються за допомогою Amazon CloudWatch, який збирає основні метрики системи, зокрема CPU, пам’яті, вводу/виводу, та журнали потоку виконання – лог-файли контейнерів чи Lambda, що дозволяє формувати звіти продуктивності та виконувати додаткові дії при перевищенні встановлених лімітів при використанні

пам’яті, процесору та інших ресурсів. Впровадження аналітичних інструментів є вимогою до сучасної мікросервісної архітектури, оскільки постійний моніторинг та контроль за системою забезпечують своєчасне реагування на потенційні відмови та загрози, що зазначено в роботі [18].

1.3 Постановка задачі

В умовах зростаючих вимог до масштабованості, гнучкості та стійкості сучасних інформаційних систем, мікросервісна архітектура демонструє високу ефективність завдяки можливості ізоляції компонентів, незалежного масштабування та автоматизованого управління інфраструктурою. Особливої актуальності набуває застосування хмарних сервісів, зокрема Amazon Web Services (AWS), які надають широкий спектр інструментів для розгортання та підтримки мікросервісних рішень з вбудованими механізмами відмовостійкості та автоматичного масштабування.

Метою кваліфікаційної роботи є розробка методу забезпечення відмовостійкості мікросервісів в хмарному середовищі AWS, що дозволяє забезпечити стабільну та неперервну роботу мікросервісів в умовах динамічної зміни навантаження.

2 МЕТОДИ ОБРОБКИ ЗАПИТІВ В ХМАРНИХ МІКРОСЕРВІСАХ

В хмарній мікросервісній архітектурі ключову роль відіграють ефективні та масштабовані методи обробки запитів. Сучасні сервіси повинні виконувати бізнес-логіку та бути здатними обробляти велику кількість запитів з мінімальними затримками, забезпечуючи при цьому відмовостійкість та високу доступність.

2.1 Маршрутизація запитів через API Gateway

Amazon API Gateway представляє собою повністю керований сервіс, призначений для створення, публікації, підтримки, моніторингу та захисту REST, HTTP та WebSocket API будь-якого масштабу. Цей сервіс функціонує як єдина точка входу для всіх API-викликів, забезпечуючи безпечний та контрольований доступ до backend-сервісів (рисунок 2.1).

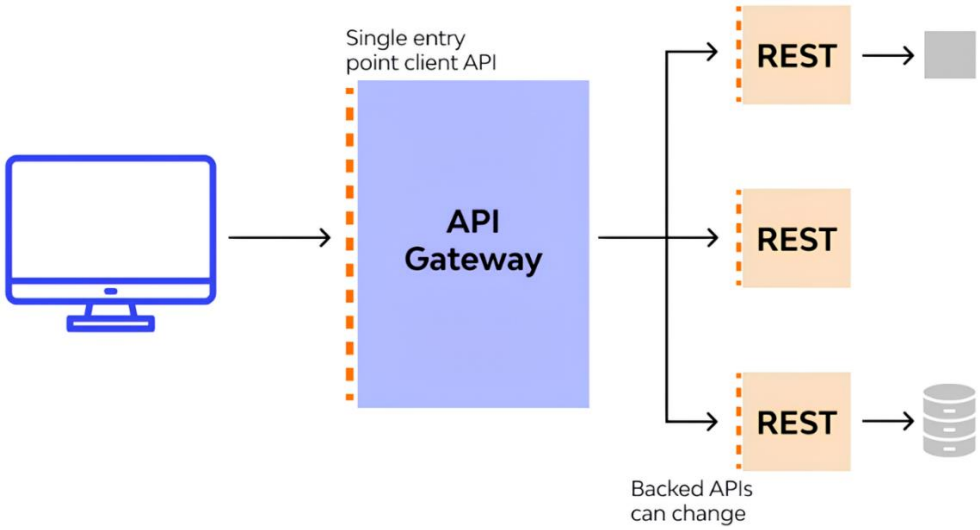

Рисунок 2.1 – Схема використання API Gateway в інфраструктурі

Основне призначення API Gateway полягає в централізованому управлінні взаємодією між клієнтськими застосунками та внутрішніми

сервісами інформаційної системи. Цей компонент виконує низку важливих функцій:

- маршрутизація запитів, що полягає в прийманні запитів від користувачів або клієнтських застосунків та перенаправленні їх до відповідних внутрішніх сервісів, які відповідають за обробку визначених функцій або даних;

- перевірка доступу, що забезпечує автентифікацію та авторизацію користувача;

- регулювання трафіку, що полягає в контролі кількості запитів, які можуть бути виконані за певний проміжок часу. Це включає обмеження швидкості та захист від надмірного навантаження, що важливо для стабільної роботи системи при великій кількості користувачів;

- моніторинг та аналітика, що полягає в зборі даних про роботу API, включаючи кількість запитів, час їх обробки, частоту помилок тощо. Ці дані використовуються для покращення якості обслуговування та оптимізації роботи всієї системи;

- інтеграція з іншими сервісами. API Gateway може запускати обчислювальні функції, зокрема безсерверні функції lambda, а також передавати повідомлення до систем обробки черг [19].

Принцип роботи API Gateway полягає у визначенні кінцевих точок, які відповідають певним шляхам URI та методам HTTP, що дозволяє логічно організувати інтерфейс взаємодії з системою, приховуючи складність внутрішньої реалізації від клієнта.

Застосування API Gateway сприяє підвищенню відмовостійкості інформаційної системи завдяки реалізації механізмів балансування навантаження між сервісами. Цей компонент виконує функції попередньої обробки вхідних запитів, зокрема їх валідації та автентифікації, що дозволяє знизити обчислювальне навантаження на основні компоненти бізнес-логіки. Водночас забезпечується ізоляція прикладної логіки від процесів контролю доступу, що сприяє підвищенню масштабованості, безпеки та структурної

гнучкості системи [20].

Окрім основних функцій маршрутизації, API Gateway забезпечує підтримку кешування відповідей безпосередньо на рівні шлюзу, що дозволяє значно скоротити час обробки повторюваних запитів. Інтеграція з сервісами типу Cloudflare Workers сприяє реалізації додаткового рівня оптимізації, зменшуючи затримки при доступі до часто запитуваних ресурсів і підвищуючи загальну продуктивність і відгук системи [21].

API Gateway є ключовим компонентом сучасної розподіленої архітектури, який забезпечує централізоване управління взаємодією між клієнтами та сервісами. Завдяки функціям маршрутизації, безпеки, балансування навантаження та кешування, він сприяє підвищенню надійності, масштабованості та продуктивності інформаційних систем, водночас спрощуючи підтримку та розвиток програмної інфраструктури.

2.2 Застосування AWS Lambda для обробки запитів

AWS Lambda є безсерверним обчислювальним сервісом, який забезпечує виконання програмного коду у відповідь на події без необхідності управління фізичною або віртуальною серверною інфраструктурою. Сервіс автоматично управляє обчислювальними ресурсами, динамічно масштабуючи їх залежно від обсягу запитів, що забезпечує високу доступність та ефективність виконання функцій.

Основна функціональна роль AWS Lambda полягає в реалізації моделі «функція як сервіс» (FaaS), що дозволяє розробникам зосередитися безпосередньо на бізнес-логіці без зайвих витрат часу на налаштування та підтримку серверних середовищ. Використання сервісу охоплює обробку подій з різних джерел, зокрема HTTP-запитів через API Gateway (рисунок 2.2), змін в базах даних, операцій з файлами в сховищі S3 та інших тригерних подій. AWS Lambda підтримує декілька мов програмування, зокрема Python, Node.js, Java, Go та C#, що забезпечує

гнучкість у виборі технологічного стеку для розробки [22].

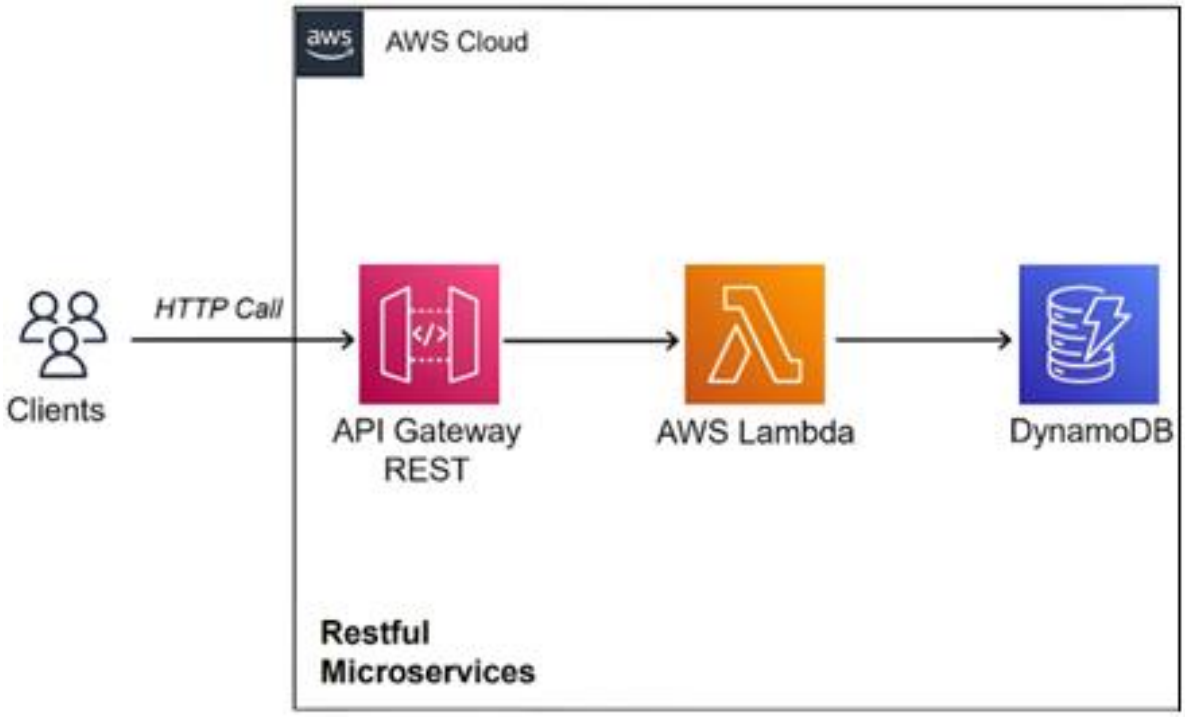

Рисунок 2.2 – Приклад схеми використання AWS Lambda

Функції AWS Lambda відповідають за обробку вхідних запитів та реалізацію основної бізнес-логіки системи. Після отримання події або запиту з відповідного джерела, функція виконує обробку вхідних даних, проводить необхідні обчислення, взаємодіє з зовнішніми сервісами, формує відповідь та повертає результат клієнту.

Завдяки подієво-орієнтованій архітектурі, lambda-функції активуються виключно за потреби, що забезпечує ефективне використання обчислювальних ресурсів та зниження операційних витрат [23].

Крім того, функції Lambda можуть бути інтегровані з іншими сервісами AWS в межах віртуальної приватної мережі (VPC), що гарантує безпечний та ізольований доступ до сервісів, зокрема Redis та DynamoDB [24]. Підтримка асинхронної обробки дозволяє паралельно виконувати численні запити, запобігаючи перевантаженню системи та забезпечуючи стабільність її роботи навіть під час пікових навантажень.

Таким чином, AWS Lambda забезпечує гнучке, масштабоване та

ефективне виконання бізнес-логіки на основі подієвого підходу, що сприяє оптимізації ресурсів, підвищенню безпеки та підтримці стабільності функціонування інформаційних систем в умовах змінного навантаження.

2.3 Кешування даних для зменшення затримок

Redis є високопродуктивною системою керування структурами даних в оперативній пам’яті, яка може функціонувати як база даних, кеш-пам’ять, брокер повідомлень або сховище сесій. В хмарному середовищі AWS Redis реалізується у вигляді керованого сервісу Amazon ElastiCache, який забезпечує автоматизоване управління, моніторинг та масштабування кластерів Redis.

Основне призначення Redis полягає в забезпеченні надшвидкого доступу до даних шляхом зберігання інформації в оперативній пам’яті, що дає змогу значно зменшити затримки при обробці запитів. Redis активно використовується для кешування часто запитуваних ресурсів, зберігання сесійної інформації, реалізації черг повідомлень, а також для механізмів блокування в розподілених системах [25]. Система підтримує різноманітні структури даних, зокрема рядки, хеші, списки, множини та відсортовані множини, що забезпечує її універсальність для різних сценаріїв використання.

Використання Redis як проміжного шару зберігання дозволяє значно знизити час відповіді на запити за рахунок уникнення повторних звернень до повільніших джерел даних. В разі наявності потрібної інформації в кеші вона негайно повертається клієнту; якщо дані відсутні, система отримує їх з основного джерела та зберігає в кеші з визначеним терміном життя (TTL), оптимізуючи подальші запити.

В мікросервісних архітектурах Redis забезпечує суттєві переваги для підвищення продуктивності та відмовостійкості. Завдяки своїй високій швидкодії при інтенсивному навантаженні Redis є ефективним

інструментом для досягнення масштабованості та балансування навантаження в системі [26]. На відміну від кешування на рівні API Gateway або CDN, використання ElastiCache забезпечує централізоване керування кешованими даними, з можливістю встановлення індивідуальних TTL, інвалідації кешу та інших політик керування, що є важливим для підтримання узгодженості даних в розподіленому середовищі.

Таким чином, Redis в поєднанні з Amazon ElastiCache виступає ключовим інструментом оптимізації доступу до даних у хмарних мікросервісних архітектурах, забезпечуючи високу продуктивність, централізоване керування кешем та ефективне масштабування системи.

2.4 Зберігання та доступ до даних

Amazon DynamoDB є повністю керованою нереляційною базою даних типу NoSQL, розробленою для високопродуктивних застосунків, які вимагають стабільної продуктивності при динамічному масштабуванні. Сервіс гарантує передбачувану затримку на рівні одиниць мілісекунд незалежно від обсягу даних чи кількості одночасних запитів, а також автоматично керує розподілом навантаження та збереженням даних в хмарному середовищі AWS.

Ключове призначення DynamoDB полягає в забезпеченні високо доступного та масштабованого середовища для зберігання структурованої та напівструктурованої інформації. База даних використовує модель зберігання «ключ-значення» та документоорієнтовану модель, що забезпечує високошвидкісне виконання операцій читання та запису за умови обробки мільйонів запитів на секунду (рисунок 2.3). Функціональність сервісу включає автоматичне масштабування, вбудоване шифрування даних, а також можливості резервного копіювання та відновлення, що робить його придатним для критично важливих застосунків з підвищеними вимогами до надійності [27].

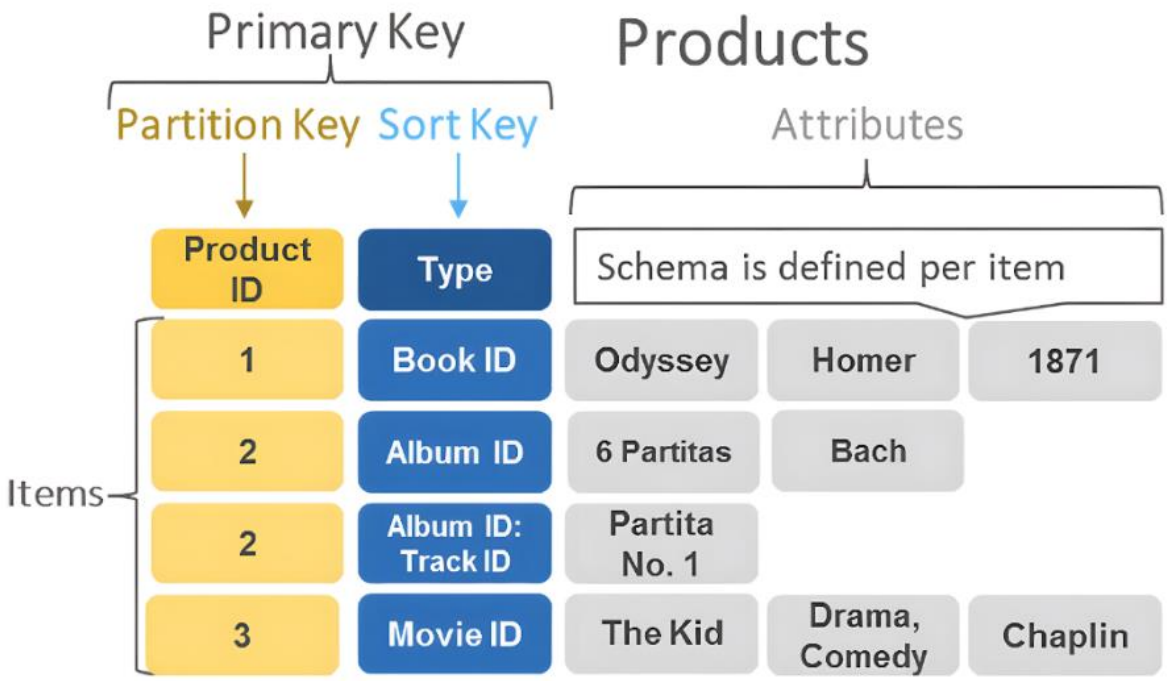

Рисунок 2.3 – Схема зберігання даних в DynamoDB

В AWS DynamoDB може виступати основним сховищем для постійного зберігання даних з низькою затримкою доступу. AWS Lambda-функції мають можливість здійснювати основні операції, зокрема створення, читання, оновлення та видалення, в DynamoDB за допомогою AWS SDK, використовуючи ролі системи IAM для безпечного доступу до ресурсів. Автоматична адаптація сервісу до зміни навантаження без необхідності ручного втручання сприяє підтриманню стабільної роботи в періоди пікового трафіку.

Однією з ключових переваг DynamoDB є глибока інтеграція з сервісами керування доступом, моніторингу та обробкою подій. Завдяки своїй архітектурі, орієнтованій на ключі, запити до DynamoDB виконуються з передбачуваною мінімальною затримкою, що є важливим для побудови високонавантажених розподілених систем з великою кількістю одночасних користувачів [28].

Отже, Amazon DynamoDB забезпечує надійне, масштабоване й безпечне середовище для зберігання даних у хмарній інфраструктурі. Завдяки низькій затримці доступу, автоматичному масштабуванню та

інтеграції з іншими сервісами AWS, DynamoDB виступає оптимальним рішенням для застосунків із високими вимогами до доступності, продуктивності та гнучкості управління даними.

Таким чином, використання API Gateway дозволяє централізувати керування трафіком, контролювати доступ, виконувати попередню обробку запитів та забезпечувати балансування навантаження, що є основою для побудови відмовостійких систем.

AWS Lambda забезпечує безсерверне виконання коду у відповідь на події, дозволяючи розробникам зосередитися на бізнес-логіці без необхідності адміністрування інфраструктури. Подієво-орієнтована модель Lambda сприяє оптимізації ресурсів та зменшенню експлуатаційних витрат, особливо в умовах змінного навантаження.

Інтеграція Redis через Amazon ElastiCache як проміжного шару зберігання, дозволяє значно знизити затримки доступу до даних та забезпечити високу продуктивність при повторюваних запитах. Можливість детального налаштування політик кешування робить Redis незамінним елементом для масштабованих мікросервісних систем.

Amazon DynamoDB виступає надійним та масштабованим рішенням для зберігання структурованих та напівструктурованих даних з гарантовано низькою затримкою. Його глибока інтеграція з іншими сервісами AWS, автоматичне масштабування та підтримка безпечного доступу забезпечують стабільність та ефективність роботи систем, навіть у періоди пікового навантаження.

Сукупне використання зазначених сервісів дозволяє створювати високопродуктивні, адаптивні та відмовостійкі хмарні мікросервісні системи з гнучкими можливостями масштабування, що відповідають сучасним вимогам до швидкості, надійності та безпеки обробки запитів.

3 РОЗРОБКА МЕТОДУ ЗАБЕЗПЕЧЕННЯ ВІДМОВОСТІЙКОСТІ МІКРОСЕРВІСНОГО ВЕБДОДАТКУ НА AWS

3.1 Моделювання роботи мікросервісного вебдодатку на AWS

Для побудови відмовостійкої мікросервісної архітектури вебдодатку в хмарному середовищі AWS, першочергово необхідно забезпечити стабільне інтернет-з'єднання зі швидкістю не менш ніж 100 Мбіт/с. Стабільність та доступність хмарних сервісів гарантується постачальником, що є ключовим фактором для підтримання безперервності роботи системи. Схему загальної архітектури подано на рисунку 3.1.

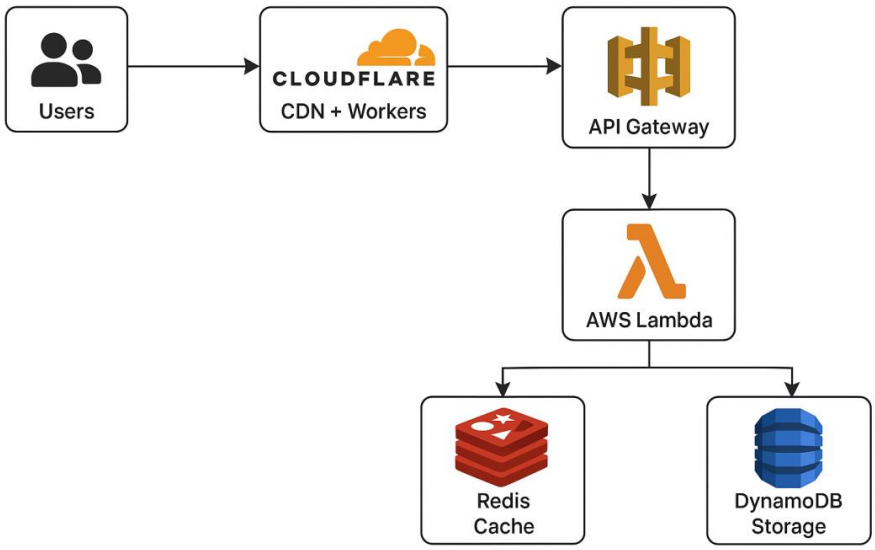

Рисунок 3.1 – Архітектурна схема

Налаштування мережної інфраструктури для розгортання вебдодатку з мікросервісною архітектурою здійснюється через покроковий алгоритм з наступними етапами:

1. На цьому етапі виконується конфігурація проміжного шару на базі Cloudflare, який водночас виконує роль CDN та кешуючого проксі-сервера. Це

забезпечує зменшення затримок, покращення глобальної доступності та базовий рівень захисту. На рисунках 3.2 – 3.4 наведено створення Cloudflare Workers з відповідним виділенням ресурсів CPU та оперативної пам’яті до очікуваного навантаження, а також механізм кешування у Cloudflare Workers, що забезпечує маршрутизацію та обробку вхідних запитів. Додатково в середовищі Workers конфігурується змінна середовища ORIGIN_URL, що вказує на адресу шлюзу API Gateway та визначаються обробники помилок 502 та 504, що визначають реакцію системи на типові збої, які полягають у перебільшені часу очікування відповіді або в невалідній відповіді API Gateway або Lambda. Надалі налаштовується Cache API з політикою зберігання кешу, яка включає TTL для статичних ресурсів, а також реалізується логіка перевірки наявності відповіді у кеші. У випадку кеш-влучання дані повертаються негайно; у випадку кеш-промаху – запит спрямовується до API Gateway, а відповідь автоматично кешується (лістинг 3.1).

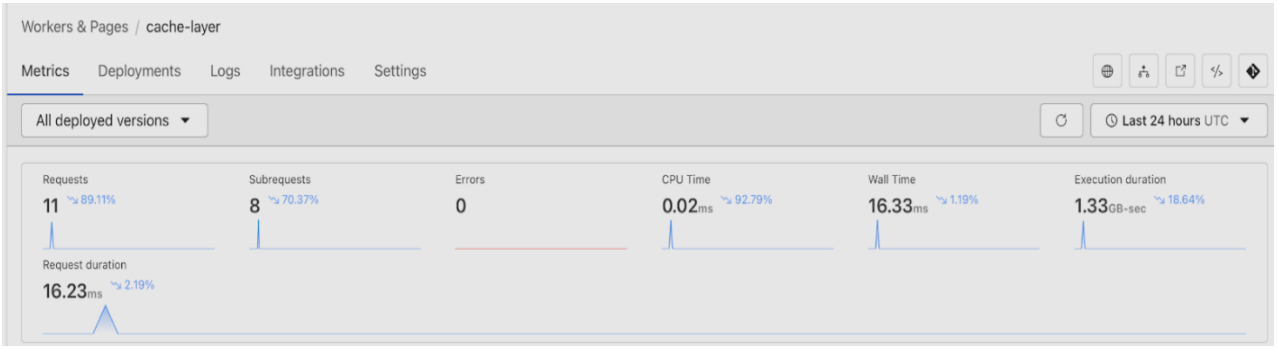

Рисунок 3.2 – Створений Cloudflare Worker

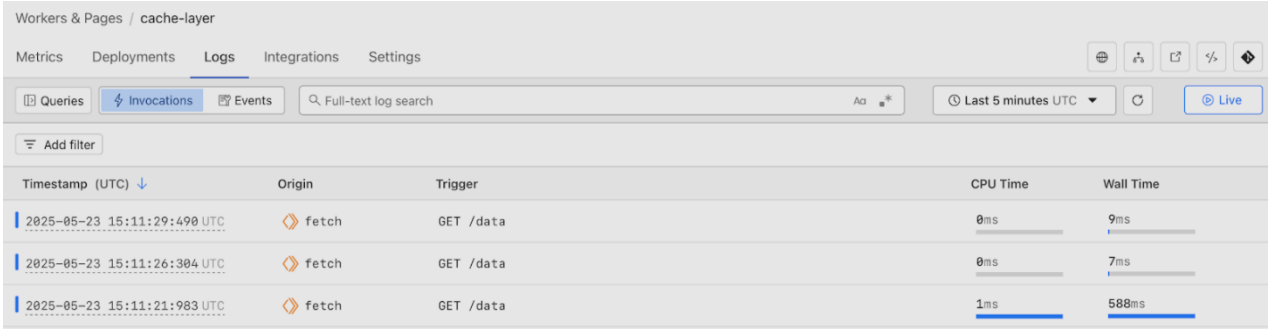

Рисунок 3.3 – Логування запитів у Worker

а) б)

Рисунок 3.4 – Заголовки відповідей: а) кеш-промах; б) кеш-влучення

Лістинг 3.1 – Механізм кешування у Cloudflare Workers

```
export default {
  async fetch(request, env, ctx) {

    const cache = caches.default;
    const cacheKey = new Request(request.url, request);
    const cachedResponse = await cache.match(cacheKey);

    if (cachedResponse) {
      return cachedResponse;
    }

    const apiUrl = ORIGIN_URL;
    const backendResponse = await fetch(apiUrl, {
      method: request.method,
      headers: request.headers,
    });

    const newHeaders = new Headers(backendResponse.headers);
    newHeaders.set('Cache-Control', 'public, max-age=TTL_TIME');

    const responseToCache = new Response(backendResponse.body, {
      status: backendResponse.status,
      statusText: backendResponse.statusText,
      headers: newHeaders,
    });
```

```
    ctx.waitUntil(cache.put(cacheKey, responseToCache.clone()));

    return responseToCache;
  },
};
```

2. На цьому етапі налаштовується сервіс Amazon API Gateway для обробки запитів до мікросервісу. Створюється REST API типу Regional, щоб зменшити затримки в межах обраного регіону. Далі додається маршрут та методи запитів, зокрема GET або POST, з типом інтеграції AWS_PROXY, що дозволяє напряму передавати дані в Lambda-функцію. В налаштуваннях інтеграції зазначається ARN цієї Lambda. Для безпеки вмикається перевірка за API-ключем, та створюється Usage Plan з обмеженням у 1000 запитів на секунду та 100 000 запитів на добу. Також додаються параметри CORS для підтримки доступу з інших доменів (рисунок 3.5).

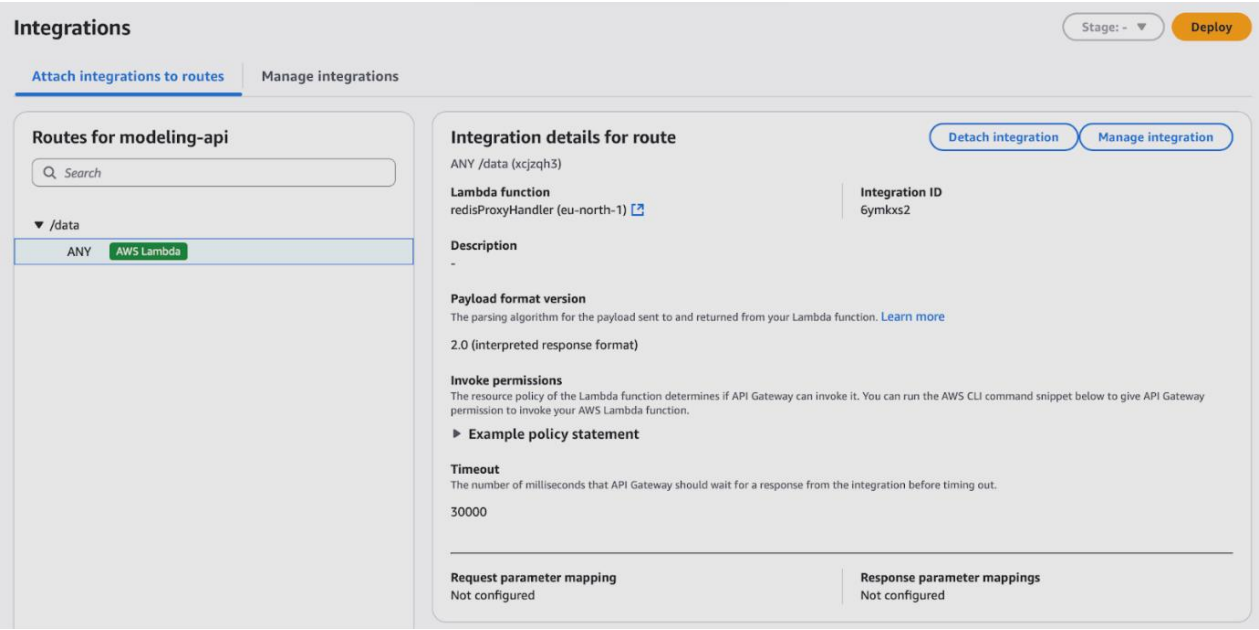

Рисунок 3.5 – Додавання ARN для Lambda

3. На цьому етапі виконується розгортання AWS Lambda функції. Для цього обирається середовище Node.js 18.x, виділяється 512 МБ оперативної пам’яті та встановлюється таймаут на 10 секунд. Функція розміщується в тій самій приватній VPC, де знаходиться Redis, з

підключенням до приватних підмереж в кількох Availability Zones для підвищення надійності. Надалі налаштовується Security Group з дозволами на вихідний трафік до портів 6379 для Redis), 443 для HTTPS-запитів та 80 для HTTP-запитів в межах VPC. Також встановлюються змінні середовища, зокрема REDIS_HOST, REDIS_PORT, DYNAMODB_TABLE_NAME та CACHE_TTL. В AWS Lambda функції реалізована логіка обробки запитів, за якою спочатку перевіряється наявність даних в Redis (лістинг 3.2). Якщо дані відсутні, вони отримуються з DynamoDB та зберігаються в кеш (лістинг 3.3).

Лістинг 3.2 – Перевірка наявності даних в Redis

```
...
const redisClient = createClient({
  socket: {
    host: process.env.REDIS_HOST,
    port: REDIS_PORT,
    tls: true
  }
});

...

if (!isConnected) {
  await redisClient.connect();
  isConnected = true;
  }

const cached = await redisClient.get(redisKey);
if (cached) {
  return {
    statusCode: 200,
    body: JSON.stringify({
      source: 'redis',
      data: cached
    })
  };
}
```

Лістинг 3.3 – Отримання даних з DynamoDBз подальшим записом в кеш

```
...
const dynamoClient = new DynamoDBClient({ region: 'eu-north-1'
});

const command = new GetItemCommand({
```

```
  TableName: DYNAMODB_TABLE_NAME,
  Key: {
    id: { S: key }
  }
});

try {
  const result = await dynamoClient.send(command);

  if (result.Item && result.Item.data && result.Item.data.S) {
    const data = result.Item.data.S;

    await redisClient.set(redisKey, data, { EX: CACHE_TTL });
    return {
      statusCode: 200,
      body: JSON.stringify({
        source: 'dynamodb',
        data
      })
    };
...
```

4. На цьому етапі для забезпечення доступу Lambda-функції до необхідних сервісів створюється IAM роль LambdaExecutionRole із політиками доступу, що дозволяє службі lambda.amazonaws.com отримати цю роль. До ролі прикріплюється базова політика AWSLambdaVPCAccessExecutionRole для роботи у VPC, а також створена політика DynamoDBAccessPolicy з дозволами dynamodb:GetItem, dynamodb:PutItem, dynamodb:UpdateItem, dynamodb:DeleteItem та dynamodb:Query для визначеної таблиці. Додатково створено політики доступу з дозволами ec2:CreateNetworkInterface, ec2:DescribeNetworkInterfaces та ec2:DeleteNetworkInterface для взаємодії з Redis (рисунок 3.6).

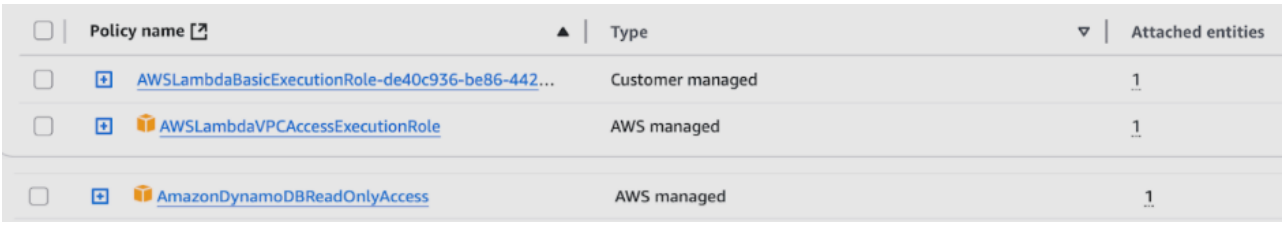

Рисунок 3.6 – надання політик доступу для Lambda-функції

5. НА цьому етапі здійснюється налаштування Redis (Amazon ElastiCache), яке включає створення кластера Redis та портом 6379 в приватній VPC, де розміщена Lambda-функція. Кластер прив’язується з увімкненими опціями автоматичного перемикання та розгортання у кількох зонах доступності для підвищення відмовостійкості. В межах Security Group дозволяється лише вхідний трафік на 6379 від Lambda.

6. На цьому етапі виконується налаштування бази даних Amazon DynamoDB, що передбачає створення таблиці з визначеним іменем та ключем секції. В якості режиму оплати обирається “on-demand” для автоматичного масштабування відповідно до навантаження. Також активується функція відновлення стану даних на певний момент часу для підвищення надійності зберігання. Для забезпечення ефективного виконання запитів за часовими мітками додається глобальний вторинний індекс з назвою TimestampIndex. Додатково вмикається шифрування даних в стані спокою для гарантування конфіденційності та цілісності інформації.

7. На цьому етапі виконується налаштування системи моніторингу та сповіщень. Для Lambda-функції активується ведення логів у CloudWatch Logs з періодом зберігання 14 днів. Додатково створюються власні метрики для аналізу співвідношення кеш-влучань (HIT) та кеш-промахів (MISS), що дозволяє оцінювати ефективність кешування. Конфігуруються сповіщення через CloudWatch Alarms для виявлення критичних подій, зокрема збої у виконанні Lambda-функції, зростання затримок під час доступу до DynamoDB або недоступність Redis. В разі потреби активується трасування AWS X-Ray для детального аналізу продуктивності системи та виявлення вузьких місць у ланцюгу обробки запитів.

Завершальним етапом виконується тестування за допомогою утиліти навантажувального тестування, яка генерує від 100 до 1000 HTTP-запитів (рисунки 3.7, 3.8). Тестування дозволяє оцінити ефективність кешування та відмовостійкість системи.

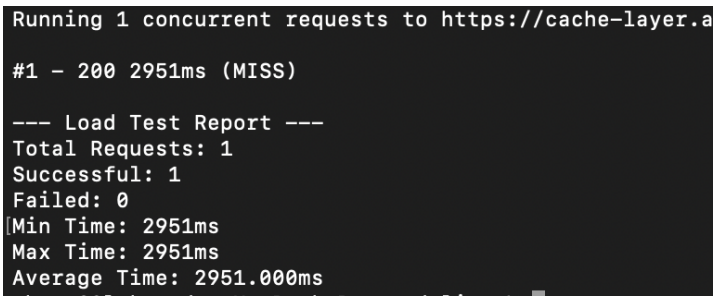

Рисунок 3.7 – Час обробки одиничного запиту з відсутністю даних у кеші

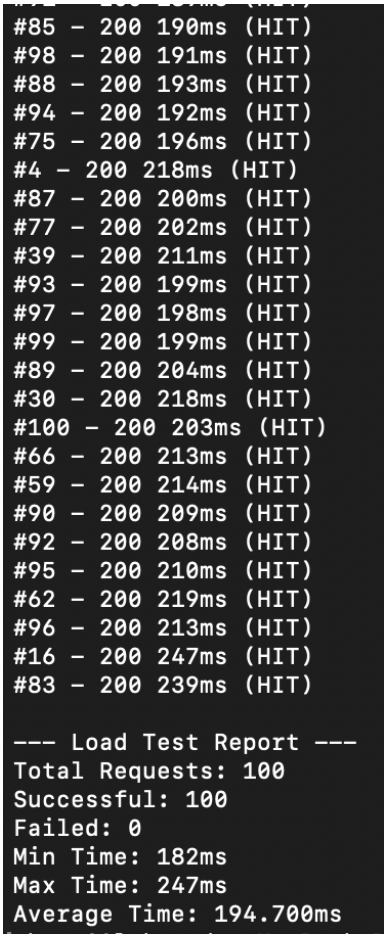

Рисунок 3.8 – Часові характеристики обробки повторюваних запитів з кеш-влучанням

3.2 Аналітичне моделювання роботи вебдодатку

В межах даної роботи система обробки запитів може бути змодельована як одноканальна система масового обслуговування типу M/M/1, що характеризує взаємодію клієнтських запитів з хмарною

мікросервісною архітектурою, яка включає в себе API Gateway як вхідну точку, AWS Lambda як виконавче середовище, Redis як кешуючий проміжний шар та Amazon DynamoDB як основне сховище даних.

В моделі M/M/1 вхідний потік запитів λ=5 зап/с вважається потоком Пуассона, тобто інтервали між запитами розподілені за експоненційним законом. Час обслуговування одного запиту відповідає випадковому часу обробки запиту серверною частиною, а інтенсивність обслуговування системи становить μ=15 зап/с.

Алгоритм обробки запитів виконується за принципом «перший прийшов – перший обслуговується» (FIFO), що відповідає поведінці HTTP-запитів у вебдодатках.

Основними показниками систем масового обслуговування моделі M/M/1, які характеризують роботу вебдодатку в режимі реального часу є середні значення запитів вхідного та вихідного потоків. При цьому вихідний потік запитів складається з кількості запитів, які перебувають в черзі на обробку сервером та кількості вже оброблених запитів [29].

Ймовірність надходження запитів ρ до системи характеризує її стабільність і визначається як відношення інтенсивності вхідного потоку запитів до інтенсивності обробки. Цей показник дозволяє оцінити ступінь завантаженості архітектури, яка включає вхід через API Gateway, обробку за допомогою AWS Lambda, кешування в Redis та звернення до бази даних DynamoDB. Ймовірність надходження пакетів характеризує завантаженість системи та розраховується як:

$$\rho = \frac{\lambda}{\mu}, \tag{3.1}$$

де λ – інтенсивність вхідного потоку запитів, од;

μ – інтенсивність вихідного потоку запитів, од.

Середня кількість запитів L в системі включає як ті запити, що

перебувають у черзі, так і ті, що наразі обслуговуються. Цей показник відображає загальну завантаженість усієї хмарної архітектури та дозволяє прогнозувати потребу в масштабуванні обчислювальних ресурсів, зокрема AWS Lambda, Redis-кешу або пропускної здатності API Gateway. Середня кількість запитів L розраховується як:

$$L = \frac{\lambda}{\mu - \lambda}, \tag{3.2}$$

де λ – інтенсивність вхідного потоку запитів, од;

μ – інтенсивність вихідного потоку запитів, од.

Середня кількість запитів в черзі L_q є показником, що дозволяє оцінити наявність затримок на вході до Lambda або Redis та вказує на необхідність впровадження додаткових оптимізацій, зокрема кешування або асинхронної обробки. Вона є корисною для моніторингу стабільності мікросервісної архітектури при високому навантаженні. Середня кількість запитів в черзі L_q розраховується як:

$$L_q = \frac{\lambda^2}{\mu(\mu - \lambda)}, \tag{3.3}$$

де λ – інтенсивність вхідного потоку запитів, од;

μ – інтенсивність вихідного потоку запитів, од.

Середній час перебування запитів в системі W включає як час очікування в черзі, так і час активної обробки, наприклад, виконання Lambda-функції або звернення до DynamoDB. Зменшення цього показника свідчить про ефективну роботу кешу Redis та низький рівень навантаження. Середній час перебування запитів в системі розраховується як:

$$W = \frac{1}{\mu - \lambda}, \tag{3.4}$$

де λ – інтенсивність вхідного потоку запитів, од;

μ – інтенсивність вихідного потоку запитів, од.

Середній час очікування в черзі W_q, показує, скільки часу проходить між надходженням запиту до системи та початком його обробки. Цей параметр є важливим індикатором оперативності реагування системи на зростаюче навантаження та дозволяє оцінити ефективність кешування, обмеження запитів та балансування трафіку на рівні API Gateway. Середній час очікування в черзі розраховується як:

$$W_q = \frac{\lambda}{\mu(\mu - \lambda)}, \tag{3.5}$$

де λ – інтенсивність вхідного потоку запитів, од;

μ – інтенсивність вихідного потоку запитів, од.

Для виконання розрахунків показників моделі M/M/1 використано середовище MATLAB, лістинг коду розрахунку яких наведено в додатку В.

За результатами аналітичного моделювання роботи запропонованої хмарної інфраструктури мікросервісного додатку було отримано наступні результати, що наведені в лістингу 3.4.

Лістинг 3.4 – Результати аналітичного моделювання хмарної інфраструктури

```
Інтенсивність надходження (λ): 5.00 зап/с
Інтенсивність обслуговування (μ): 15.00 зап/с
Коефіцієнт завантаження (ρ): 0.333
Середня кількість запитів у системі (L): 0.500
Середня кількість у черзі (Lq): 0.167
Середній час перебування (W): 0.100 сек
Середній час в черзі (Wq): 0.033 сек
>>
```

За результатами тестування та аналітичного моделювання було побудовано графік залежності середнього часу перебування запитів у системі за моделлю СМО M/M/1 та фактично виміряного середнього часу обробки кешованих запитів в тестовому середовищі (рисунок 3.9).

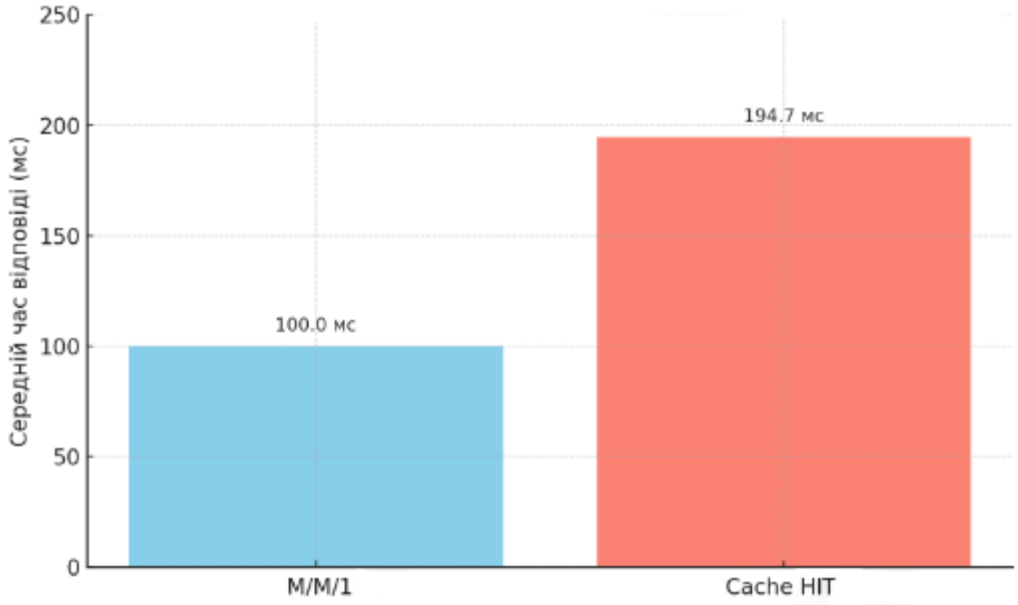

Рисунок 3.9 – Графік залежності середнього часу перебування запитів у системі за моделлю СМО M/M/1 та фактично виміряного середнього часу обробки кешованих запитів в тестовому середовищі

Аналітичне моделювання хмарної мікросервісної архітектури, побудованої за принципом API Gateway → AWS Lambda → Redis/DynamoDB, було здійснено на основі класичної моделі системи масового обслуговування типу M/M/1. Незважаючи на те, що практичний середній час відповіді перевищує теоретичне значення, це пояснюється реальними обмеженнями інфраструктури, зокрема мережними затримками, обробки на рівні Cloudflare, Lambda та Redis. Проте співвідношення часу в межах допустимого, а стабільність результатів свідчить про ефективність кешування та масштабованості архітектури.

Таким чином, результати моделювання демонструють, що запропонований підхід до побудови відмовостійкої хмарної архітектури із

застосуванням кешування, балансування навантаження та динамічного масштабування здатний забезпечити стійку роботу системи в умовах змінного навантаження, при цьому оптимізуючи споживання обчислювальних ресурсів. Це підтверджує ефективність використання розробленого методу забезпечення відмовостійкості.

ВИСНОВКИ

В ході кваліфікаційної роботи було досягнуто основну мету – розроблено метод забезпечення відмовостійкості хмарних мікросервісних вебзастосунків на базі сервісів AWS, що дозволяє зберігати високу продуктивність при масштабованому обслуговуванні запитів в режимі реального часу та динамічно змінному навантаженні.

На основі аналізу наукових публікацій та технічної документації було запропоновано мікросервісну архітектуру вебдодатку, яка поєднує механізми API Gateway, безсерверних обчислень за допомогою AWS Lambda, кешування в Redis та зберігання даних в базі Amazon DynamoDB.

Запропонований підхід може бути використаний для побудови високонавантажених вебзастосунків, що працюють в хмарному середовищі з підвищеними вимогами до відмовостійкості, масштабованості та швидкодії. Отримані результати підтверджують доцільність впровадження розробленої архітектури в практичних сценаріях.

Подальші дослідження доцільно спрямувати на інтеграцію архітектури з інструментами автоматичного масштабування та системами моніторингу.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Thota R. C. Cost optimization strategies for micro services in AWS: Managing resource consumption and scaling efficiently //International Journal of Science and Research Archive. – 2023. – Т. 10. – №. 2. – С. 1-12.

2. Christian J. et al. Analyzing Microservices and Monolithic Systems: Key Factors in Architecture, Development, and Operations // 2023 6th International Conference of Computer and Informatics Engineering (IC2IE). – IEEE, 2023. – С. 64-69.

3. Kamisetty A. et al. Microservices vs. Monoliths: Comparative Analysis for Scalable Software Architecture Design //Engineering International. – 2023. – Т. 11. – №. 2. – С. 99-112.

4. Lercher A. et al. Microservice api evolution in practice: a study on strategies and challenges //Journal of Systems and Software. – 2024. – Т. 215. – С. 112110.

5. Nunes, Joao Paulo KS, et al. "State of the art on microservices autoscaling: An overview." *Seminário Integrado de Software e Hardware (SEMISH)* (2021): 30-38.

6. Matias M. et al. Evaluating Effectiveness and Security in Microservices Architecture //Procedia Computer Science. – 2024. – Т. 237. – С. 626-636.

7. de Almeida M. G., Canedo E. D. Authentication and authorization in microservices architecture: A systematic literature review //Applied Sciences. – 2022. – Т. 12. – №. 6. – С. 3023.

8. Leocadio P. H. AWS Master Class Chapter 05: Security, Identity and Compliance //Authorea Preprints. – 2025.

9. Tadi S. Architecting Resilient Cloud-Native APIs: Autonomous Fault Recovery in Event-Driven Microservices Ecosystems //Journal of Scientific and Engineering Research. – 2022. – Т. 9. – №. 3. – С. 293-305.

10. John T. Enhancing Microservices Architecture with AI-Based

Monitoring and Self-Healing Systems. – 2023.

11. Shethiya A. S. Scalability and Performance Optimization in Web Application Development //Integrated Journal of Science and Technology. – 2025. – Т. 2. – №. 1.

12. Oyeniran, Oyekunle Claudius, et al. "Microservices architecture in cloud-native applications: Design patterns and scalability." International Journal of Advanced Research and Interdisciplinary Scientific Endeavours 1.2 (2024): 92-106.

13. Eassa, Ahmed M. "Optimizing Web Application Development: A Proposed Architecture Integrating Headless CMS and Serverless Computing." IJCI. International Journal of Computers and Information 12.1 (2025): 103-119.

14. Paul J. J. Distributed Serverless Architectures on AWS // Berkeley, CA. – 2023.

15. Leocadio P. H. AWS Master Class Chapter 14: AWS Well-Architected Framework //Authorea Preprints. – 2025.

16. Elhemali M. et al. Amazon {DynamoDB}: A scalable, predictably performant, and fully managed {NoSQL} database service //2022 USENIX Annual Technical Conference (USENIX ATC 22). – 2022. – С. 1037-1048.

17. Arif T., Jo B., Park J. H. A Comprehensive Survey of Privacy-Enhancing and Trust-Centric Cloud-Native Security Techniques Against Cyber Threats //Sensors. – 2025. – Т. 25. – №. 8. – С. 2350.

18. Diagboya E. Infrastructure Monitoring with Amazon CloudWatch: Effectively monitor your AWS infrastructure to optimize resource allocation, detect anomalies, and set automated actions. – Packt Publishing Ltd, 2021.

19. Imran, M., Bashir, F., Jafri, A. R., Rashid, M., & ul Islam, M. N. (2021). A systematic review of scalable hardware architectures for pattern matching in network security. *Computers & Electrical Engineering*, *92*, 107169.

20. Gadia R. et al. A System on Automated Database and API (Application Programming Interface) Management //Int. J. Res. Appl. Sci. Eng. Technol. – 2022. – Т. 10. – №. 4. – С. 3226-3234.

21. Gorantla V. K. C., Madala S. C. Serverless PWAs: Reducing Backend Load with Cloud Functions and Edge Computing //International Journal of Emerging Research in Engineering and Technology. – 2022. – Т. 3. – №. 2. – С. 39-48.

22. Thota R. C. Efficient serverless architectures: Leveraging AWS Lambda and SageMaker for scalable workflow solutions //Journal of Science & Technology. – 2024. – Т. 5. – №. 3. – С. 133-152.

23. Syed A. A. M., Ali S. Kubernetes and AWS Lambda for Serverless Computing: Optimizing Cost and Performance Using Kubernetes in a Hybrid Serverless Model //International Journal of Emerging Trends in Computer Science and Information Technology. – 2024. – Т. 5. – №. 4. – С. 50-60.

24. Nugroho E. C. A Horizontally Scalable WebSocket Architecture for Cost-Effective Online Examination Proctoring System on AWS Cloud Infrastructure //Engineering, MAthematics and Computer Science Journal (EMACS). – 2025. – Т. 7. – №. 1. – С. 115-126.

25. Sanka, Abdurrashid Ibrahim, Mehdi Hasan Chowdhury, and Ray CC Cheung. "Efficient high-performance FPGA-Redis Hybrid NoSQL caching system for blockchain scalability." *Computer Communications* 169 (2021): 81-91.

26. Joshi P. K. Redis Cache Optimization for Payment Gateways in the Cloud //International Journal of Emerging Trends in Computer Science and Information Technology. – 2023. – Т. 4. – №. 2. – С. 28-36.

27. Carvalho, Inês, Filipe Sá, and Jorge Bernardino. "Performance evaluation of NoSQL document databases: couchbase, CouchDB, and MongoDB." *Algorithms* 16.2 (2023): 78.

28. Sadat, Nazmus, and Rui Dai. "A Survey of Quality-of-Service and Quality-of-Experience Provisioning in Information-Centric Networks." *Network* 5.2 (2025): 10.

29. Moltafet, M., Leinonen, M., & Codreanu, M. (2020). Average Age of Information for a Multi-Source M/M/1 Queueing Model With Packet Management. In 2020 IEEE International Symposium on Information Theory (ISIT), Los Angeles, CA, USA. https://doi.org/10.1109/isit44484.2020.9174099.