МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ПРИРОДОКОРИСТУВАННЯ
ФАКУЛЬТЕТ МЕХАНІКИ, ЕНЕРГЕТИКИ
ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

другого (магістерського) рівня вищої освіти

на тему:

“РОЗРОБКА СИСТЕМИ ДЛЯ БЕЗПЕРЕБІЙНОЇ РОБОТИ ВЕБ-ДОДАТКІВ ІЗ ВИКОРИСТАННЯМ ТЕХНОЛОГІЙ БЕЗСЕРВЕРНИХ ОБЧИСЛЕНЬ”

Виконав: студент групи Іт-61

Спеціальності 126 «Інформаційні системи та технології»

Мартинів Б.Я.
(Прізвище та ініціали)

Керівник: Смолінський В.Б.
(Прізвище та ініціали)

ДУБЛЯНИ-2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ПРИРОДОКОРИСТУВАННЯ
ФАКУЛЬТЕТ МЕХАНІКИ, ЕНЕРГЕТИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Другий (магістерський) рівень вищої освіти
Спеціальність 126 «Інформаційні системи та технології»

"ЗАТВЕРДЖУЮ"
Завідувач кафедри_____
д.т.н., проф. А.М. Тригуба
"____"_____2024 р.

ЗАВДАННЯ

на кваліфікаційну роботу студенту
Мартиніву Богдану Ярославовичу

1. Тема роботи: *«Розробка системи для безперебійної роботи веб-додатків із використанням технологій безсерверних обчислень»*
Керівник роботи: *к.е.н., доцент Смолінський В.Б.*
(наук.ступінь, вч. звання, прізвище та ініціали)
затверджені наказом по університету від 28.04.2023 року № 133/к-с.
2. Строк подання здобувачем роботи 06.12.2024 р.
3. Вихідні дані до роботи: *Основні поняття та інструменти для роботи з безсерверними обчисленнями, розроблення веб-додатків, AWS Lambda, API Gateway, DynamoDB, S3, AWS SAM*
4. Зміст розрахунково-пояснювальної записки (перелік питань, які необхідно розробити)
1. Аналіз технологій безсерверних обчислень та їх застосування у веб-додатках.
2. Методи та інструменти для побудови архітектури веб-додатків.
3. Розробка та впровадження архітектури веб-додатку.
4. Тестування продуктивності та аналіз результатів.
5. Охорона праці та безпека в надзвичайних ситуаціях.
Висновки
Список використаних джерел
Додатки
5. Перелік ілюстраційного матеріалу (з точним зазначенням обов'язкових схем та моделей): *Модель ціноутворення хмарних провайдерів. Порівняння часу виконання та паралелізм. Приклад роботи AWS Lambda у веб-додатку. Процес автоматичного перемикання з основного сервера на резервний у разі збою. Схема уникнення єдиної точки відмови (SPOF). Діаграма мікросервісної архітектури. Схема аутентифікації та контролю доступу через API Gateway. Схема взаємодії API Gateway та Lambda. Діаграма процесу деплойменту SAM.*

Діаграма налаштування метрик та алертів у AWS CloudWatch. CloudWatch Lambda Dashboard. Діаграма автоматичного масштабування Lambda. Діаграма резервування та відновлення даних у DynamoDB. Діаграма архітектури додатку. Головна сторінка додатку. Вікно JMeter

6. Консультанти з розділів:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
1, 2, 3, 4	*Смолінський В.Б., доцент кафедри інформаційних технологій*		
5	*Городецький І.М., доцент кафедри фізики, інженерної механіки та безпеки виробництва*		

7. Дата видачі завдання 28.04.2023 р.

Календарний план

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	*Написання першого розділу: Аналіз технологій безсерверних обчислень та їх застосування у веб-додатках*	28.04-20.06.23	
2	*Виконання другого розділу: Методи та інструменти для створення архітектури веб-додатків*	21.06-14.08.23	
3.	*Виконання другого розділу: Методи та інструменти для створення архітектури веб-додатків*	15.08-31.10.23	
4.	*Виконання четвертого розділу: Тестування продуктивності та аналіз результатів*	01.10 - 01.11.24	
5.	*Написання розділу: «Охорона праці та безпека в надзвичайних ситуаціях»*	01.10 - 01.11.24	
6.	*Написання висновків та пропозицій. Завершення оформлення пояснювальної записки та презентаційних матеріалів*	01.11 - 01.12.24	
7.	*Завершення роботи вцілому*	01-10.12.24	

Студент _____ Мартинів Б.Я.
(підпис)

Керівник роботи _____ Смолінський В.Б.
(підпис)

УДК: 004.94: 631.1

Розробка системи для безперебійної роботи веб-додатків із використанням безсерверних обчислень. Мартинів Б.Я. Кафедра інформаційних технологій. Дубляни, Львівський національний університет природокористування, 2024.

Кваліфікаційна робота викладена на 73 сторінках, містить 5 розділів, 28 рисунків, 30 літературних джерел.

У роботі наведено основні поняття безсерверних обчислень, розглянуто архітектурні особливості та переваги цих технологій. Проведено аналіз сучасних інструментів, таких як AWS Lambda, API Gateway, DynamoDB та S3, для реалізації безсерверної архітектури.

Розроблено систему для забезпечення високої доступності та відмовостійкості веб-додатків. Використано AWS SAM для автоматизації інфраструктури, CloudWatch для моніторингу та управління метриками, а також налаштовано безпеку доступу за допомогою IAM.

Описано процес розробки MVP системи, протестовано її продуктивність та надійність. Система демонструє переваги безсерверної архітектури: автоматичне масштабування, економія ресурсів, спрощення розробки та підтримки.

Предмет дослідження – розробка системи для безперебійної роботи веб-додатків із використанням безсерверної архітектури на базі AWS.

Мета роботи – створення веб-додатку із високою надійністю, який реалізований на базі безсерверної архітектури AWS, із використанням AWS Lambda, API Gateway, DynamoDB та S3.

Для досягнення поставленої мети у роботі виконані наступні завдання:

- Дослідження основних понять та принципів безсерверних обчислень.
- Аналіз існуючих технологій і вибір відповідних інструментів для реалізації проекту.

- Розробка системи із застосуванням AWS SAM для автоматизації інфраструктури.
- Проведення тестування надійності та продуктивності системи.

Практична значимість кваліфікаційної роботи полягає у створенні інноваційної системи для веб-додатків, що забезпечує їхню безперебійну роботу навіть під значним навантаженням. Розроблений веб-додаток демонструє переваги безсерверної архітектури, такі як висока масштабованість, автоматизація процесів та економія ресурсів. Реалізація цієї системи може слугувати основою для створення подібних рішень у сфері веб-розробки та інфраструктури.

Ключові слова: AWS Lambda, безсерверні обчислення, веб-додатки, API Gateway, хмарні обчислення, продуктивність, надійність, IAM, AWS SAM, S3, DynamoDB.

ЗМІСТ

ВСТУП

Сучасний розвиток інформаційних технологій спричинив значне зростання вимог до веб-додатків, які повинні забезпечувати стабільну та надійну роботу за будь-яких умов. У зв'язку з постійним збільшенням навантаження на цифрові сервіси, бізнес та інші галузі стикаються з викликом забезпечення безперебійної роботи своїх систем, особливо в умовах зростаючого трафіку та збільшення кількості користувачів.

Основна важливість полягає в тому, щоб розробити систему, яка забезпечить безперебійну роботу веб-додатків, навіть при високих навантаженнях та можливих відмовах окремих компонентів. Така система повинна забезпечувати автоматичне масштабування та розподіл навантаження, забезпечуючи стабільну та надійну роботу додатку навіть в умовах збільшення обсягу користувацьких запитів та високого трафіку.

Одним із перспективних рішень цієї проблеми є впровадження безсерверних обчислень, які надають можливість автоматичного масштабування ресурсів і мінімізації витрат на інфраструктуру. Такі технології, як AWS Lambda, API Gateway та AWS SAM, дозволяють створювати високопродуктивні, надійні й економічно ефективні веб-додатки, що відповідають сучасним потребам.

Проте, незважаючи на очевидні переваги безсерверних архітектур, існують і значні виклики. Серед них можна виділити питання оптимізації витрат, стабільності систем при високих навантаженнях та забезпечення відмовостійкості компонентів. Особливе значення має створення систем, здатних адаптуватися до динамічних умов експлуатації, зберігаючи при цьому високу продуктивність і якість обслуговування користувачів [5].

Актуальність цієї роботи визначається необхідністю розробки системи, яка забезпечувала б безперебійну роботу веб-додатків, ефективно використовуючи обчислювальні ресурси та забезпечуючи автоматичне масштабування. Така система має бути здатною адаптуватися до умов високого навантаження, залишаючись надійною навіть у разі виникнення відмов окремих компонентів.

Мета роботи полягає в розробці архітектури веб-додатка з використанням технологій безсерверних обчислень, яка забезпечить безперервність його роботи, оптимізацію витрат і високу продуктивність.

Об’єктом дослідження є процеси розробки та впровадження безсерверної архітектури для веб-додатків. Предметом дослідження виступають методи та технології, що дозволяють забезпечити надійність і ефективність функціонування таких систем.

Запропонована робота має практичну значущість, оскільки розроблена система може знайти застосування у сфері електронної комерції, освіти, медицини, фінансів та інших галузях, де важливими є стабільність та ефективність веб-додатків.

Таким чином, вирішення проблеми пов'язано з розробкою алгоритмів та стратегій управління ресурсами, які дозволяють автоматично масштабувати та зменшувати кількість ресурсів в залежності від навантаження. Така система повинна бути здатною ефективно використовувати ресурси, що дозволить забезпечити стабільність роботи веб-додатків без зайвих витрат.

РОЗДІЛ 1
АНАЛІЗ ТЕХНОЛОГІЙ БЕЗСЕРВЕРНИХ ОБЧИСЛЕНЬ ТА ЇХ ЗАСТОСУВАННЯ У ВЕБ-ДОДАТКАХ

1.1. Огляд технологій безсерверних обчислень

У сучасному світі з кожним днем зростає вимога до технологій, що забезпечують безперебійність роботи веб-додатків. Висока конкуренція та швидкий розвиток інтернет-простору вимагають від розробників здатності створювати додатки, які працюють стійко та надійно в будь-який час.

Технології безсерверних обчислень, які дозволяють автоматично масштабувати ресурси та забезпечувати безперебійну роботу додатків, стають ключовим інструментом для вирішення цієї проблеми.

Значимість роботи проявляється в її потенційному внеску у різні сфери. Стабільні та ефективні веб-додатки є критично важливими для бізнесу, освіти, медицини та інших галузей. Вони визначають конкурентоспроможність компаній, які залежать від цифрового простору .

З розширенням обсягу онлайн-сервісів та цифровізації різних аспектів життя, можливість забезпечити безперебійну роботу веб-додатків стає стратегічною. Така робота може позитивно вплинути на користувачів, покращуючи їхній досвід та надійність взаємодії з додатками.

Безсерверні технології, або FaaS (Function-as-a-Service), являють собою новий підхід до розробки додатків. У двох словах, FaaS - це форма безсерверних обчислень, яка використовує інфраструктуру, повністю керовану провайдером, для завантаження функцій і їх запуску на основі оплати за запит. На відміну від інших підходів до хмарних обчислень, безсерверні повністю абстрагують розробників від серверів і дозволяють їм зосередитися на бізнес-логіці.

Одним із ключових принципів безсерверних обчислень є те, що розробник вже не відповідає за управління серверами чи серверними ресурсами. Весь цикл життя програми, від розгортання до масштабування, адмініструється хостинг-

постачальником. Це робить процес розробки та утримання системи більш простим та спрощеним для розробників.

Ще однією ключовою рисою є принцип оплати за фактом використання. Користувач оплачує лише реальний час виконання коду та використані ресурси, замість сталої кількості серверних інстанцій. Це сприяє ефективнішому використанню ресурсів та зменшенню витрат.

Ще однією концепцією безсерверних обчислень є концентрація на функціях. Програмний код розглядається як набір невеликих, незалежних функцій. Розробники фокусуються на написанні функцій, які виконують конкретні завдання, і вони можуть бути викликані лише при необхідності, сприяючи простоті та ефективності коду.

Безсерверні обчислення автоматично масштабуються в залежності від обсягу роботи. Хостинг-постачальник реагує на зміни в обсягу запитів, забезпечуючи еластичність та високу доступність системи. Це дозволяє ефективно використовувати ресурси, запускаючи лише необхідну кількість функцій.

Ще однією особливістю безсерверних функцій є їхня безстандовість. Вони не зберігають стану між викликами, і кожен виклик обробляється ізольовано. Це спрощує розробку та масштабування системи, зберігаючи її простою та надійною [18].

Безсерверність спрощує адміністрування та розгортання, оскільки розробники можуть зосередитися на написанні коду, а хостинг-постачальник бере на себе відповідальність за інфраструктуру. Це полегшує процес розробки та зменшує навантаження на команду адміністраторів.

В основі безсерверних обчислень лежить принцип реагування на події чи виклики. Функції автоматично викликаються при настанні певних подій, таких як нові дані чи завершення завдань. Це дозволяє системі реагувати динамічно та ефективно, створюючи гнучку та високопродуктивну інфраструктуру.

1.2. Архітектурні особливості безсерверних систем

У своєму основному виявленні, безсерверні системи часто використовують мікросервісну архітектуру. Це означає, що весь функціонал розбивається на невеликі, незалежні мікросервіси, кожен з яких може бути реалізований як окрема функція. Такий підхід дозволяє досягти великої функціональної незалежності та полегшує утримання системи [3].

Ще однією ключовою архітектурною особливістю є принцип функціональної незалежності. Кожна функція в системі повинна бути самодостатньою та незалежною, забезпечуючи спрощення розробки та підтримки. Це стає можливим завдяки розбиттю системи на невеликі функціональні одиниці, які можуть працювати автономно.

Архітектура безсерверних систем також передбачає динамічне масштабування. Система може автоматично змінювати кількість ресурсів в залежності від обсягу роботи, забезпечуючи еластичність та ефективне використання ресурсів.

Ключовим принципом також є те, що розробники не повинні взаємодіяти напряму з серверним середовищем. Хостинг-постачальник приховує деталі інфраструктури, що дозволяє спростити процес розгортання та управління.

Асинхронна обробка є ще однією характерною особливістю. Багато безсерверних систем використовують асинхронний підхід до обробки запитів, де функції викликаються при настанні подій, а обробка відбувається асинхронно, що сприяє оптимізації використання ресурсів.

Нарешті, архітектура безсерверних систем дозволяє зберігати функції безстандартними, тобто без зберігання стану між викликами. Це сприяє простоті масштабування та розподілу роботи [9].

Нижче наведено ключові аспекти, що свідчать на користь використання безсерверних систем:

- Нижчі витрати та масштабованість. Є кілька причин, чому дешевше запускати свій додаток на основі FaaS. Порівняно з традиційним підходом, це зменшує витрати на експлуатацію та обслуговування серверів. Порівняно з іншими типами хмарних обчислень, більшість провайдерів FaaS працюють за моделлю "оплата за запит". Це означає, що ви платите лише за час, коли функція була викликана, і за кількість викликів. Крім того, ви можете виділити певну кількість пам'яті та процесорних ресурсів для функції та масштабувати її за потреби.

- Швидша розробка та розгортання. Замість того, щоб писати монолітну структуру, FaaS пропонує більш гнучку альтернативу. Розробники можуть писати код для набору функцій, а не для всього монолітного додатку, і завантажувати біти коду на сервер. Це дозволяє легко налагоджувати, оновлювати та додавати нові функції.

- Зменшення витрат на людські ресурси. Відсутність серверів означає, що не потрібно наймати DevOps-інженерів для обслуговування або купувати спеціальне обладнання.

- Висока доступність і автоматичне масштабування. Функція стає активною, коли її запитує клієнтська частина. Функція також може працювати після декількох запитів, але все одно вимикається, коли в ній немає потреби. Зі зростанням трафіку сервіс автоматично масштабує ресурси, виділені для певної функції. Такий підхід робить FaaS високодоступним і забезпечує безперебійну роботу під великими навантаженнями.

- Фокус на потребах бізнесу. Абстрагування розробників від роботи на стороні сервера дозволяє вашій команді зосередитися на бізнес-логіці вашого додатку.

Недосконалості підходу:

- Деяка затримка при старті функцій.
- Обмежена контроль низькорівневих параметрів.

1.3. Популярні платформи безсерверних обчислень

Amazon Web Services (AWS) Lambda – одна з найпопулярніших платформ безсерверних обчислень. Вона надає широкий спектр можливостей для виконання функцій відповідно до потреб користувача. AWS Lambda підтримує велику кількість мов програмування, включаючи Python, Node.js, Java, та інші. Його висока масштабованість, швидкість реакції та інтеграція з іншими сервісами AWS роблять його популярним вибором для розробників [2].

Microsoft Azure Functions – ще одна важлива платформа, яка надає можливості безсерверних обчислень. Azure Functions підтримує широкий спектр мов програмування та інтегрується з іншими сервісами Azure, такими як Azure Storage, Azure Cosmos DB та інші. Ця платформа також надає можливість вибору між різними типами тригерів, такими як HTTP-тригери, таймери, або тригери, пов'язані з подіями [8].

Google Cloud Functions – третя ключова платформа, що пропонує можливості безсерверних обчислень. Вона інтегрується з іншими продуктами Google Cloud, такими як Google Cloud Storage, Firebase, та інші. Google Cloud Functions також підтримує кілька мов програмування та дозволяє розробникам легко створювати та виконувати функції [6].

Порівняльний аналіз:

Моделі ціноутворення та фактори виставлення рахунків. Більшість FaaS-провайдерів використовують модель ціноутворення з оплатою за запит, яка є досить економічно вигідною. Для розрахунку вартості вашого додатку існують сервіси, які досить точно прогнозують ваші потенційні витрати.

FaaS PRICING COMPARISON

	Free-tier	Pricing
AWS Lambda	1 million/month 400,000 GB-s	$0.00001667 per GB-s
Microsoft Azure	1 million/month 400,000 GB-s	$0.000016 per GB-s
Google Cloud Functions	2 million/month 400,000 GB-s	$0.0000004 per GB-s (separate billing for memory and CPU)

Рисунок 1.1 - Модель ціноутворення хмарних провайдерів

AWS Lambda пропонує безкоштовний рівень, який включає 1 мільйон запитів і 400 000 ГБ-секунд обчислювального часу на місяць. Всі запити, що перевищують ліміт безкоштовного рівня, оплачуються за ціною $0.00001667/GB-s, що є найнижчою ціною на ринку. У реальній практиці безкоштовний рівень дозволяє запускати ваш додаток досить довго, перш ніж почнеться виставлення рахунків. Виділені ресурси (пам'ять і процесор) виставляються як одна одиниця, оскільки обидва ресурси зростають пропорційно. Додаткові витрати можуть бути пов'язані з використанням інших сервісів AWS в рамках вашої лямбда-функції.

Azure тарифікується так само, як і Lambda, з єдиною різницею в $0,000016/ГБ-с, але безкоштовний рівень є ідентичним. Вартість великих навантажень у Azure трохи нижча, ніж у Lambda, і дорівнює Lambda для середніх навантажень. Але Microsoft вважає за краще виставляти рахунки за спожиту пам'ять, а не за виділену. Azure також пропонує нижчі ціни за використання Windows і SQL, що цілком логічно. Отже, вибір між ними залежить більше від середовища, яке ви використовуєте, ніж від витрат, які ви несете.

Безкоштовний рівень GCF – це 2 мільйони запитів на місяць з тими ж 400 000 ГБ, і $0,0000004 за запит після нього, з урахуванням мережевого трафіку. Враховуючи тривалість роботи функції та кількість запитів, витрати на Google Cloud Functions значно вищі. Що стосується ресурсів, то GCF відрізняється тим, що виставляє рахунки за виділену пам'ять і процесор окремо.

Підсумовуючи, AWS Lambda пропонує золоту середину в ціноутворенні, в той час як Azure може варіювати витрати в залежності від використовуваного процесора та пам'яті. Але для середовищ Windows Azure пропонує найнижчу ціну [13].

Мови програмування: Провайдер FaaS - це публічна хмара, а це означає, що ви запускаєте свій додаток у керованому середовищі, і кожен постачальник пропонує підтримку різних мов.

Lambda охоплює широкий спектр мов програмування, включаючи середовище виконання Node.js, Python, Java та мови, скомпільовані на ньому, а також мови .NET (C#, Visual Basic та F#).

Функції Azure, очевидно, зосереджені на сімействі мов Microsoft і включають JavaScript та мови, скомпільовані на ньому, середовище виконання Node.js, C#, F#, Python, PHP, Bash, Batch і PowerShell.

Раніше Google Cloud Functions підтримував лише JavaScript, але було оголошено, що багато інших мов проходять бета-тестування, так що в довгостроковій перспективі сервіс GCF має шанс не відставати від інших великих вендорів. Але поки що це не виглядає надійним вибором.

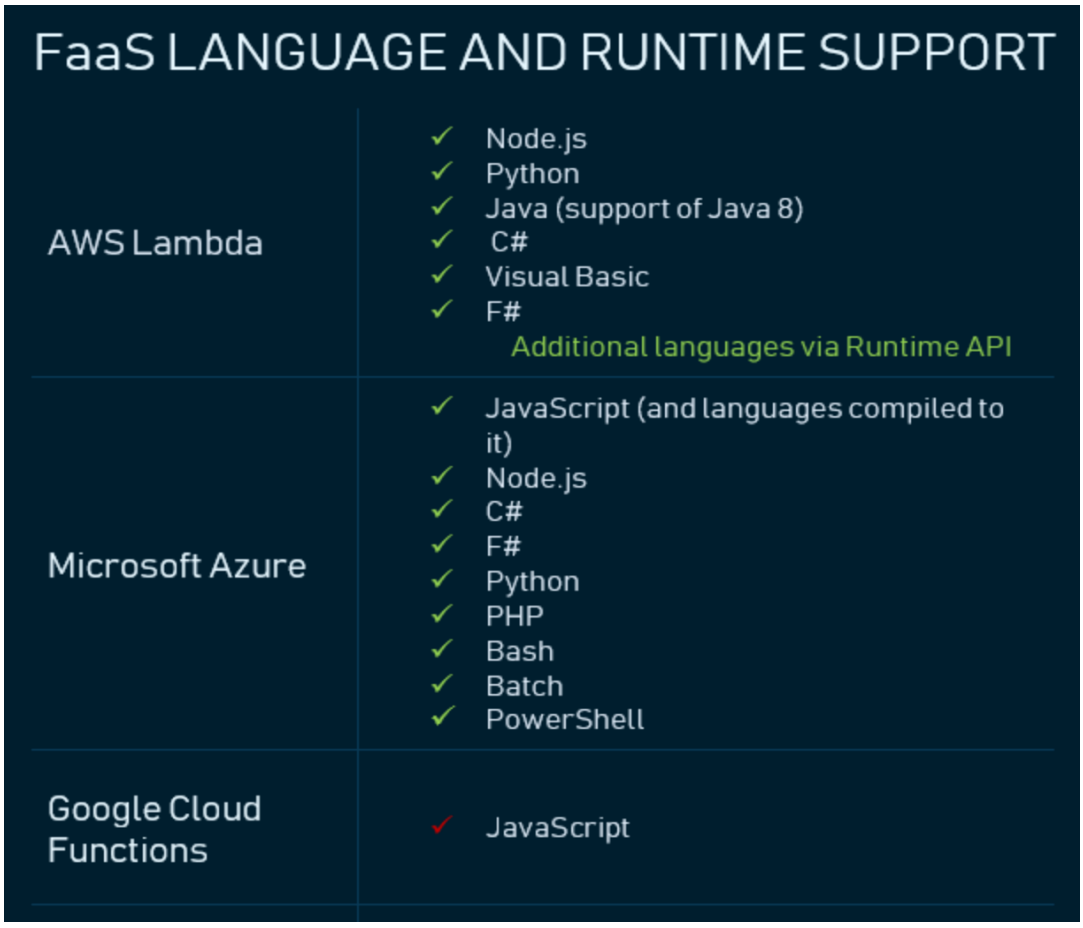

Рисунок 1.2 - Список підтримуваних мов

Час виконання та паралелізм: Іншим важливим аспектом, пов'язаним з викликом функції, є час, протягом якого функція може бути активною, а також паралельність. Паралельність означає паралельне виконання різних функцій протягом певного періоду часу.

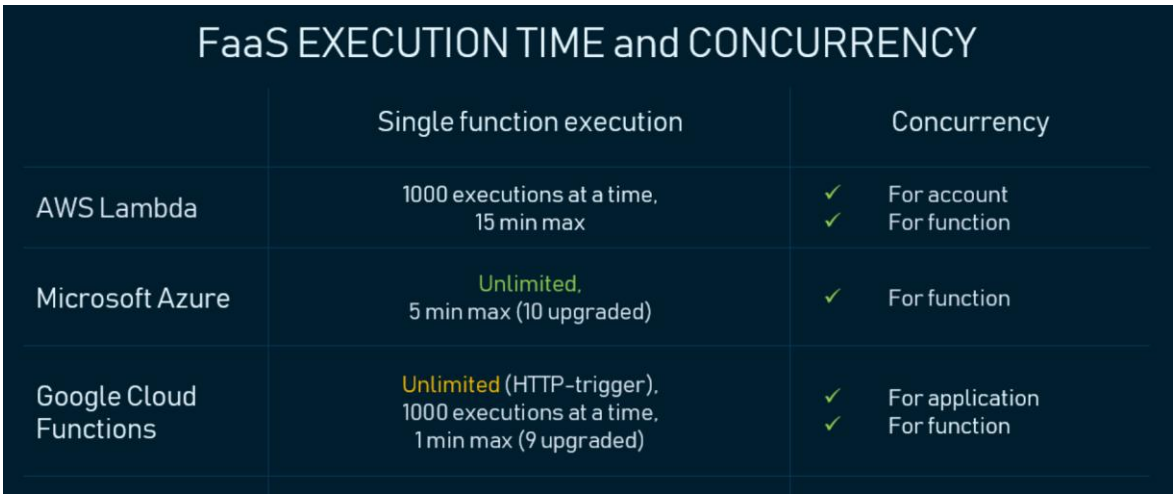

Рисунок 1.3 - Порівняння часу виконання та паралелізм

Lambda обмежує рівень паралелізму до 1,000 виконаннь за раз, з максимальним часом виконання 15 хвилин. Паралелізм можна налаштувати для всього облікового запису або для окремої функції.

Azure пропонує необмежену кількість одночасних запусків в одному додатку, але обмежує максимальний час виконання однієї функції до 5 хвилин, а в разі оновлення - до 10 хвилин.

GCF дозволяє необмежену кількість викликів для типу тригера HTTP, що є хорошим варіантом. Що стосується інших методів запуску, то паралельність така ж, як і у Lambda - 1,000 виконаннь за раз. Час виконання однієї функції обмежений 60 секундами, але може бути збільшений до майже 9 хвилин. Важливо зазначити, що AWS Lambda рахує паралельні функції в межах облікового запису, тоді як GCF робить те ж саме в межах проекту. Це означає, що на AWS ви можете запустити лише одну функцію з 1000 одночасних викликів, тоді як на GCF можна запустити декілька функцій з тим самим паралельним виконанням.

1.4. Інструменти для розробки та управління безсерверними додатками

Цей розділ присвячений огляду основних інструментів, які спрощують розробку, тестування та управління безсерверними додатками. Вибір

правильного інструменту є важливим кроком у створенні ефективних та надійних додатків.

При розробці безсерверних додатків можна скористатись таким інструментарієм:

1. AWS Toolkit – надає інтеграцію з популярними інтегрованими середовищами розробки, такими як Visual Studio та IntelliJ IDEA. Цей інструмент дозволяє створювати, розгортати та керувати безсерверними додатками безпосередньо з робочого середовища розробки.

2. Serverless Framework - це відкрите та розширюване середовище для розробки безсерверних додатків. За допомогою конфігураційних файлів, розробники можуть визначати функції, тригери та інші аспекти додатка. Цей інструмент підтримує різні провайдери, включаючи AWS, Azure та Google Cloud.

3. Azure Functions Tools призначений для розробників, які працюють в середовищі Microsoft Azure, інструмент Azure Functions Tools дозволяє легко створювати, розгортати та керувати Azure Functions безпосередньо з Visual Studio.

4. Google Cloud SDK – це інструментарій для розробки та управління додатками на платформі Google Cloud. За допомогою SDK розробники можуть створювати та розгортати безсерверні функції на платформі Google Cloud Functions.

5. Postman – це інструмент для тестування та взаємодії з API. Для безсерверних додатків, які використовують HTTP тригери, Postman може бути використаний для відлагодження та тестування взаємодії з функціями.

6. AWS Management Console – веб-інтерфейс для керування різними сервісами AWS, включаючи Lambda. Розробники можуть використовувати консоль для моніторингу та керування функціями.

7. AWS CloudWatch та AWS X-Ray надають інструменти для моніторингу та відладки безсерверних додатків. CloudWatch дозволяє вам відстежувати логи та метрики, тоді як X-Ray забезпечує можливість аналізу взаємодії компонентів додатка.

Ці інструменти роблять процес розробки, тестування та управління безсерверними додатками більш зручним та ефективним. Вибір конкретних інструментів може залежати від ваших потреб та вибору платформи хмарних обчислень [1].

1.5. Масштабування та автоматизація в безсерверних системах

Безсерверні обчислення володіють унікальними можливостями масштабування та автоматизації, що визначає їхню ефективність та гнучкість у відповіді на зміни обсягу роботи.

Безсерверні обчислення автоматично масштабуються в залежності від потреб обсягу роботи. Система реагує на збільшення навантаження, запускаючи додаткові екземпляри функцій, тим самим забезпечуючи високу доступність та ефективність.

Безсерверні системи легко горизонтально масштабуються, дозволяючи створювати паралельні екземпляри функцій для обробки більшої кількості запитів. Це забезпечує рівномірний розподіл роботи та дозволяє легко реагувати на зміни в обсязі роботи [10].

Модель ціноутворення багатьох безсерверних платформ базується на кількості викликів функцій. Збільшення кількості запитів веде до збільшення вартості обчислень, що є вигідним для проектів з непостійними часами активності.

Безсерверні системи швидко масштабуються вгору та вниз відповідно до обсягу роботи. Це забезпечує ефективне використання ресурсів та високу відповідь системи при різких змінах навантаження.

Безсерверні системи легко інтегруються з автоматизованими засобами управління, такими як AWS CloudWatch, які надають інструменти для моніторингу та автоматизованого масштабування, що сприяє ефективному керуванню ресурсами.

Безсерверні платформи часто мають можливості автоматичного резервного копіювання та відновлення функцій, що допомагає забезпечити надійність та швидке відновлення системи в разі виникнення проблем.

Інфраструктурний код дозволяє автоматизувати конфігурацію та управління ресурсами безсерверних додатків, спрощуючи розгортання та масштабування.

1.6. Використання безсерверних обчислень в реальних проектах

Безсерверні обчислення набувають популярності завдяки своїй гнучкості та ефективності. Нижче представлені приклади успішного впровадження та використання безсерверних технологій у реальних проектах:

1. Ефективне Споживання Ресурсів: У проекті електронної комерції було використано безсерверні обчислення для обробки та відображення статистики відвідувань та покупок. Функції викликалися лише при необхідності, що дозволило економити ресурси та забезпечувати швидку відповідь системи.

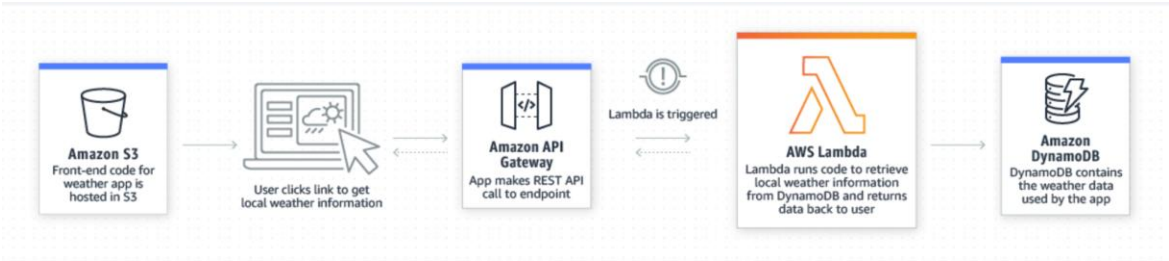

Рисунок 1.4 - Приклад роботи AWS Lambda у веб додатку

2. Системи Обробки Зображень: У медичному проекті безсерверність була використана для створення системи обробки зображень для діагностики. Функції обробки викликалися при завантаженні нових зображень, забезпечуючи швидке та ефективне їхнє оброблення.

Рисунок 1.5 - Приклад роботи AWS Lambda в обробці зображень

3. Аналіз Даних у Реальному Часі: Проект з моніторингу використовував безсерверні функції для аналізу та візуалізації даних у реальному часі. Запити на обчислення створювалися тільки при отриманні нових даних, що дозволяло швидко реагувати на зміни у великому потоці інформації.

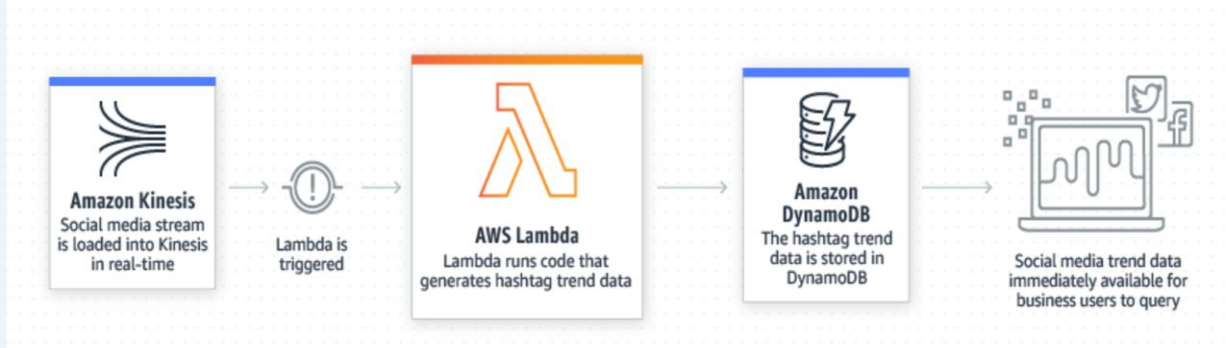

Рисунок 1.6 - Приклад роботи AWS Lambda з даними в реальному часі

4. Системи Інтеграції Та Постачання: У проекті автоматизації бізнес-процесів використовувалась безсерверність для реалізації системи інтеграції та постачання (CI/CD). Функції автоматично викликаються при зміні коду або запиті на розгортання, спрощуючи процес розробки та впровадження.

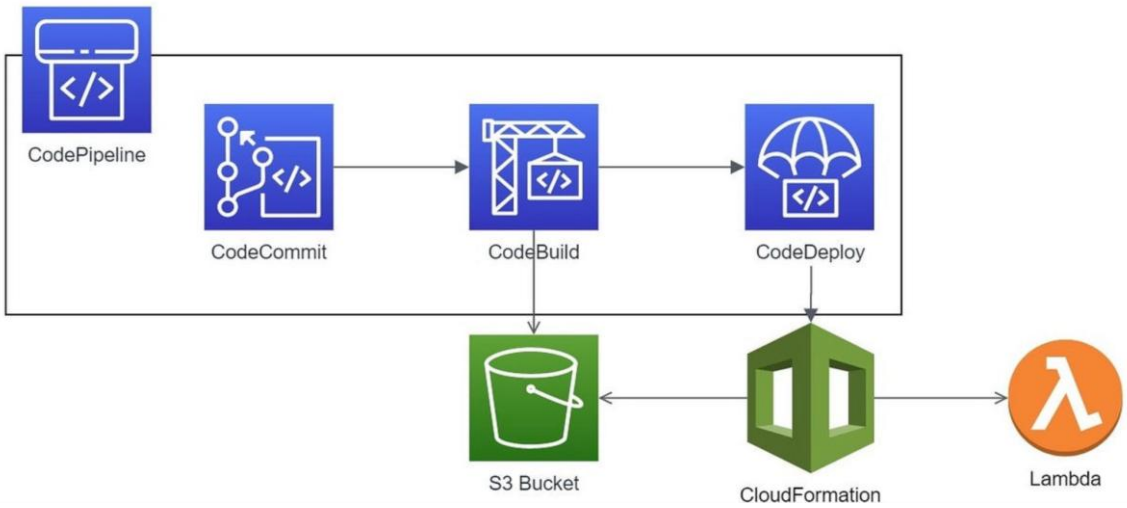

Рисунок 1.7. - Приклад роботи AWS Lambda з сервісами CI/CD

Ці приклади демонструють різноманітність використання безсерверних обчислень у реальних проектах, від оптимізації ресурсів до швидкого реагування на зміни в середовищі [22].

1.7. Переваги та недоліки безсерверних архітектур для веб-додатків

Безсерверні архітектури для веб-додатків пропонують ряд переваг та відкривають нові можливості, але вони також пов'язані з певними викликами та недоліками. У цьому розділі розглянемо як переваги, так і недоліки безсерверних архітектур у контексті веб-додатків.

До основних переваг безсерверних архітектур у контексті веб-додатків можна віднести:

1. Ефективне використання ресурсів. Безсерверні архітектури реалізують концепцію "плати лише за використання", де функції викликаються лише при наявності конкретного запиту чи завдання для виконання. У порівнянні з традиційними серверними моделями, де сервери постійно увімкнені та готові обробляти запити, безсерверні архітектури дозволяють максимально раціонально використовувати ресурси хмарних сервісів.

Це призводить до оптимізації витрат та забезпечує ефективне використання хмарних ресурсів. Основний вигідний момент полягає в тому, що користувач оплачує лише реальний час виконання коду та використані ресурси під час виклику функції. При неактивності додатка ресурси не витрачаються, що призводить до значного зниження витрат, особливо у ситуаціях, коли обсяг роботи має значні піки та періоди спокою. Такий підхід робить безсерверні архітектури привабливими для проектів зі змінними навантаженнями та піковими обсягами викликів.

2. Гнучкість та швидкість розробки. Безсерверна архітектура створює сприятливий контекст для гнучкості та швидкості розробки, дозволяючи розробникам фокусуватися на самому коді та функціональності, а не на інфраструктурних аспектах. Розробники можуть швидко створювати новий функціонал та впроваджувати зміни, не хвилюючись про складність конфігурації чи налаштувань інфраструктури.

Ця гнучкість підвищує швидкість реагування на зміни вимог та сприяє швидкому виводу нового функціоналу. Розробники можуть експериментувати з

ідеями та проводити швидкі ітерації без необхідності витрачати час на ручну настройку серверів чи інших інфраструктурних елементів. Безсерверність дозволяє зосередитися на креативному процесі розробки, що робить її ідеальним вибором для проектів, де важливо швидко впроваджувати та адаптуватися до змін у вимогах.

3. Автоматичне масштабування. Однією з ключових переваг безсерверних обчислень є можливість автоматичного масштабування. Це означає, що система може динамічно адаптувати кількість ресурсів, необхідних для обробки завдань, в залежності від обсягу роботи. Коли збільшується кількість запитів або завдань, система автоматично вирішує розподілити їх між вузлами чи іншими ресурсами для забезпечення ефективності та швидкості обробки.

Це дозволяє забезпечити високий рівень доступності та ефективності системи, оскільки ресурси автоматично масштабуються в залежності від навантаження. При піках активності система може використовувати більше ресурсів для обробки запитів, тоді як у періоди меншого навантаження може автоматично зменшувати кількість активних ресурсів, щоб зменшити витрати. Це робить безсерверні обчислення ефективним рішенням для проектів зі змінною або несприятливою навантаженістю.

4. Спрощена адміністрація. Спрощена адміністрація у безсерверних обчисленнях виникає через відсутність необхідності розглядати та управляти інфраструктурними аспектами, що відбувається автоматично на рівні хостинг-постачальника. Розробники можуть зосередитися на написанні коду та функціоналу, оскільки багато інфраструктурних питань, таких як масштабування, резервне копіювання та керування ресурсами, вже вирішено на рівні платформи.

Це робить адміністрування додатків простішим та менш часовитратним завданням для розробників. Вони можуть уникнути рутинної роботи з конфігурацією та підтримкою інфраструктури, що є зокрема важливим у сфері безсерверних обчислень. Спрощена адміністрація також забезпечує швидше

розгортання та оновлення додатків, оскільки розробники можуть зосередитися на розробці, уникнувши важкостей налаштування та обслуговування інфраструктури.

Також можливі недоліки безсерверних архітектур у контексті веб-додатків, а саме:

1. Затримки при холодному старті. Затримки при холодному старті виникають при першому виклику функції після тривалого періоду неактивності. Оскільки безсерверні функції масштабуються вгору або вниз в залежності від навантаження, під час неактивності ресурси зменшуються, і для наступного виклику функції потрібно часу на їхнє відновлення.

Ця затримка може призвести до несприятливого впливу на час відповіді додатка та загальний користувацький досвід. Особливо це може бути помітно в сценаріях, де важлива митлива відповідь на запити. Затримки при холодному старті можуть впливати на продуктивність додатку, що важливо враховувати при виборі безсерверної архітектури для проекту.

2. Вартість при великих обсягах роботи. Хоча безсерверні обчислення відзначаються тим, що користувач оплачує лише реальний час виконання коду та використані ресурси, великі обсяги роботи можуть призвести до неочікувано високих витрат. Зокрема, при постійному великому навантаженні може виявитися, що вартість безсерверних обчислень перевищує витрати на традиційні серверні рішення.

Це може стати суттєвим обмеженням для проектів з великим обсягом роботи або стабільним високим навантаженням. Розробники та організації повинні уважно враховувати можливі витрати при масштабуванні проектів та оцінювати, чи безсерверні обчислення є економічно вигідним рішенням у конкретному контексті.

3. Відсутність повного контролю. В безсерверних обчисленнях розробники обмежені можливостями повного контролю над інфраструктурними аспектами. Оскільки хостинг-постачальник бере на себе відповідальність за управління

ресурсами, розробники мають менше можливостей для налаштування окремих параметрів середовища виконання та інших інфраструктурних параметрів.

Це може бути проблемою для проектів, де важливий повний контроль над інфраструктурою, такий як налаштування оптимізації чи специфічні вимоги до безпеки. Розробники повинні усвідомлювати обмеження та вибирати безсерверні обчислення лише для тих проектів, де відсутність повного контролю не є критичним фактором.

4. Обмежена тривалість виконання. Безсерверні функції обмежені максимальною тривалістю виконання. Це стосується часу, протягом якого функція може залишатися активною. Наприклад, в багатьох хмарних сервісах це обмеження може становити 15 хвилин.

Це може стати проблемою для завдань, які вимагають тривалої обробки чи довгих періодів активності. Розробники повинні ретельно планувати архітектуру свого додатку, враховуючи обмеження часу виконання, і вибирати безсерверні обчислення тільки для завдань, які можуть бути виконані в межах цього обмеження.

5. Системи комплексного стану. Системи комплексного стану визначаються як системи, які забезпечують стеження, аналіз та управління станом різних компонентів системи. У контексті безсерверних обчислень це може включати в себе моніторинг використання ресурсів, обробку помилок, аналіз логів та інші аспекти, щоб забезпечити ефективність та доступність системи.

Використання систем комплексного стану важливо для забезпечення надійності та продуктивності безсерверних додатків. Ці системи можуть допомагати виявляти проблеми, оптимізувати використання ресурсів та забезпечувати оперативне втручання в разі необхідності. Розробники повинні уважно розглядати включення систем комплексного стану у свої проекти для покращення їхньої ефективності та надійності

РОЗДІЛ 2
МЕТОДИ ТА ІНСТРУМЕНТИ ДЛЯ ОРГАНІЗАЦІЇ БЕЗСЕРВЕРНОГО СЕРЕДОВИЩА З ВИКОРИСТАННЯМ AWS

2.1 Теоретичні основи забезпечення високої доступності та відмовостійкості

Висока доступність (High Availability, HA) означає, що система залишається доступною і функціональною навіть у разі технічних проблем або значного навантаження. Це досягається завдяки резервуванню, кластеризації, а також розподілу ресурсів, які забезпечують стабільну роботу системи без перебоїв. Відмовостійкість (Fault Tolerance) дозволяє системі продовжувати роботу навіть у разі відмови окремих її компонентів. Для цього в архітектурі передбачається резервування та дублювання компонентів, що гарантує безперервність роботи.

Ключовими принципами для досягнення високої доступності та відмовостійкості є реплікація даних і компонентів, які можуть автоматично замінити пошкоджені елементи; використання failover – швидкого автоматичного перемикання на резервний сервер у разі збою основного; балансування навантаження (load balancing) для рівномірного розподілу трафіку між серверами, що запобігає перевантаженню окремих компонентів. Крім того, важливим елементом є збереження стану системи, що дозволяє зберегти дані користувацьких сесій і забезпечити стабільну роботу навіть у випадку перенаправлення трафіку [19].

Серед переваг реалізації високої доступності та відмовостійкості можна виділити мінімізацію часу простою, зниження ризику втрати даних та загальне підвищення надійності системи.

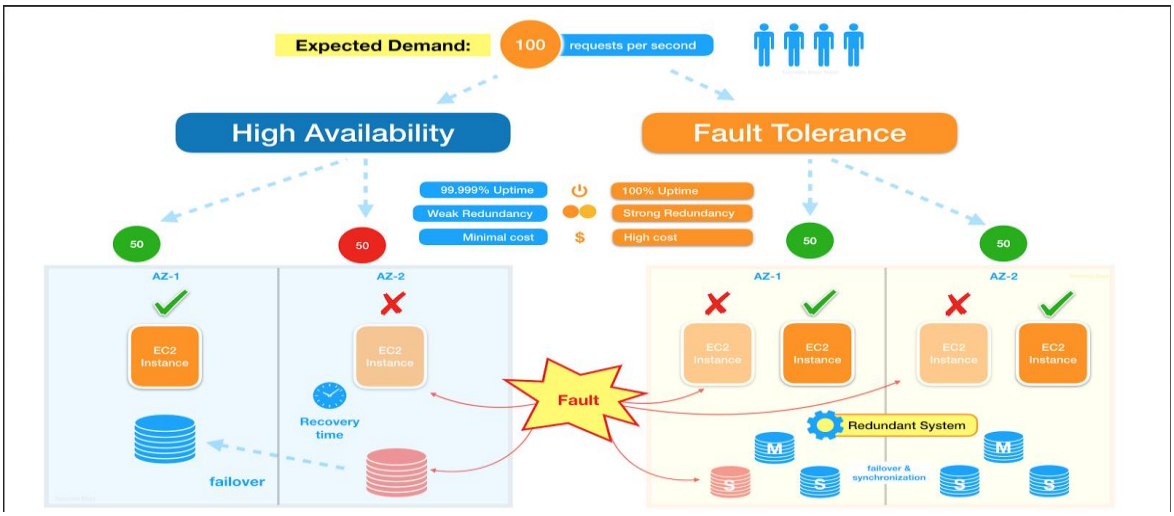

Рисунок 2.1 - Процес автоматичного перемикання з основного сервера на резервний у разі збою

Однак такі рішення мають і свої виклики, зокрема складність налаштування, високу вартість резервування і зберігання, а також необхідність регулярного тестування компонентів та сценаріїв відмов.

2.2 Основні підходи до побудови відмовостійких систем

Відмовостійкі системи – це архітектури, які здатні зберігати функціональність навіть у разі відмови окремих компонентів, мінімізуючи простої та втрату даних. Головним завданням у створенні таких систем є зменшення ризику виникнення критичних ситуацій та забезпечення стабільної роботи. Для досягнення цього використовуються різні методи, включаючи реплікацію даних, дублювання сервісів, а також стратегії для уникнення єдиної точки відмови (Single Point of Failure, SPOF).

2.2.1 Реплікація даних і дублювання сервісів

Реплікація даних і дублювання сервісів є ключовим підходом у створенні відмовостійкої системи. Реплікація даних полягає в тому, що критично важлива інформація зберігається на декількох серверах або у різних дата-центрах, забезпечуючи її доступність у разі виходу з ладу одного з компонентів.

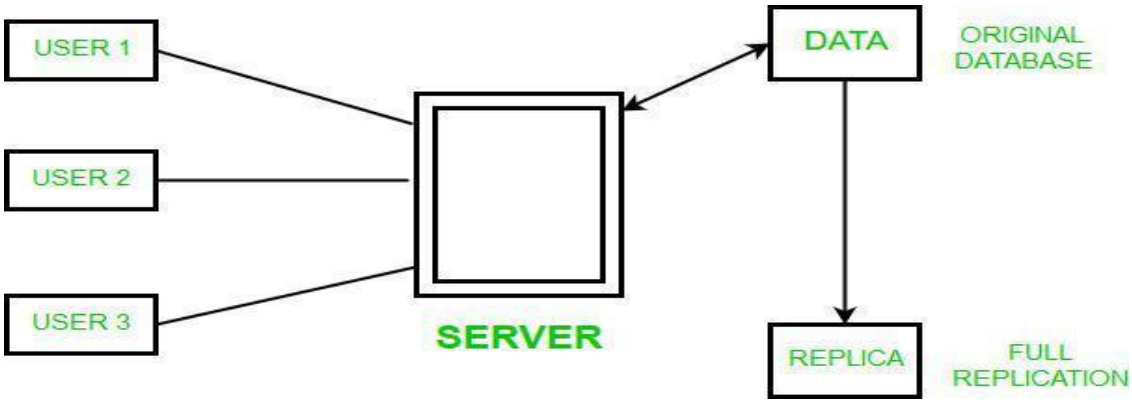

Рисунок 2.2 - Реплікація бази даних

Існує кілька підходів до реплікації, зокрема синхронна та асинхронна реплікація. Синхронна реплікація передбачає одночасне оновлення даних на всіх серверах, що гарантує їх узгодженість, але може впливати на продуктивність системи. Асинхронна реплікація дозволяє швидше обробляти операції, однак може спричинити невеликі затримки в оновленні даних. Дублювання сервісів, у свою чергу, забезпечує резервні екземпляри всіх критично важливих сервісів, що дозволяє системі автоматично перемикатися на дубльований сервіс у разі відмови основного.

2.2.2 Стратегії для уникнення єдиної точки відмови (SPOF)

Уникнення єдиної точки відмови (Single Point of Failure, SPOF) є критично важливим підходом у побудові надійних систем. Єдина точка відмови – це компонент, відмова якого може призвести до зупинки всієї системи. Щоб запобігти такому сценарію, системи будуються так, щоб уникнути залежності від одного критичного компоненту. Наприклад, у випадку серверів баз даних SPOF можна уникнути за рахунок створення кластерів, де дані зберігаються на кількох серверах. У разі відмови одного з серверів трафік автоматично перенаправляється на інший, забезпечуючи безперебійну роботу.

Іншим прикладом стратегії уникнення SPOF є балансування навантаження (load balancing), яке дозволяє розподілити запити між декількома серверами. Це забезпечує не тільки оптимізацію використання ресурсів, але й запобігає ситуації, коли відмова одного серверу може вплинути на доступність усієї

системи. Балансувальники навантаження можуть автоматично перенаправляти запити на активні сервери, знижуючи ризик простоїв.

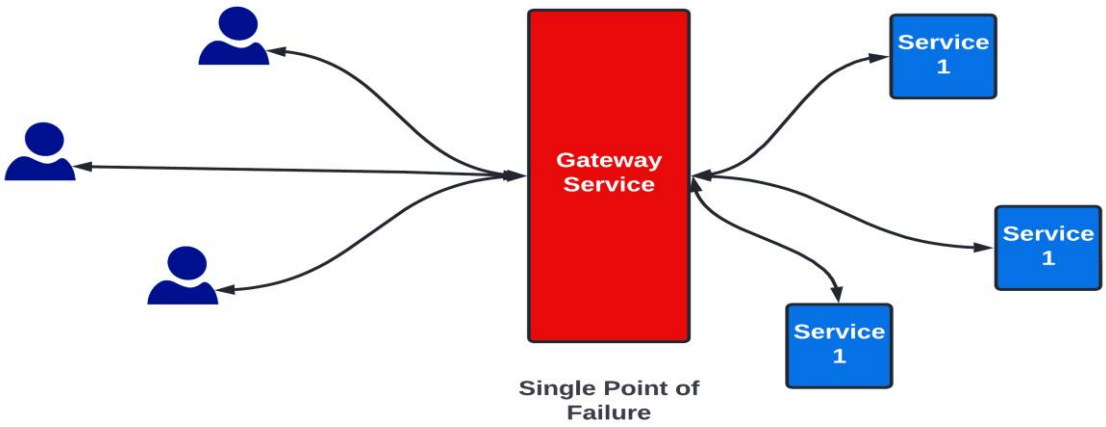

Рисунок 2.3 - Схема уникнення єдиної точки відмови (SPOF)

2.3. Вибір AWS як основної платформи для безсерверної реалізації

Amazon Web Services (AWS) є провідною платформою для реалізації безсерверних рішень завдяки своєму широкому набору сервісів, які забезпечують надійність, масштабованість та гнучкість. AWS надає можливості для побудови інфраструктури, яка дозволяє компаніям зосередитись на розробці продукту без необхідності керування серверами. Для цього AWS пропонує сервіси, як-от AWS Lambda, API Gateway, DynamoDB, S3 та багато інших, які дозволяють легко інтегрувати різні компоненти в межах єдиної платформи. Використання AWS як основної платформи для безсерверних обчислень обумовлено її здатністю надавати сервіси «на вимогу», що дозволяє оптимізувати використання ресурсів і знижувати витрати.

AWS дозволяє уникнути проблем, пов'язаних із традиційними підходами до серверної інфраструктури, оскільки управління обчислювальними ресурсами, масштабування та безпека виконуються автоматично. Безсерверна архітектура в AWS знімає з розробників необхідність планувати серверні потужності, що є особливо важливим для високонавантажених або динамічних систем, де навантаження змінюється у реальному часі. Крім того, AWS надає можливості

для налаштування автоматичного масштабування, що забезпечує високу доступність та відмовостійкість додатка.

AWS також інтегрується з інструментами для моніторингу та управління безпекою, такими як AWS CloudWatch та AWS IAM, що дає змогу адміністраторам контролювати стан і доступ до системи. Вибір AWS як платформи для безсерверних обчислень створює основу для розгортання надійної та захищеної архітектури, яка відповідає вимогам сучасних додатків.

2.4. Використання AWS Lambda для виконання функцій на вимогу

AWS Lambda є одним із найважливіших компонентів у безсерверній архітектурі AWS, оскільки дозволяє виконувати функції на вимогу без необхідності керування серверною інфраструктурою. Lambda дозволяє запускати функції у відповідь на події, такі як HTTP-запити, завантаження файлів у S3, або зміни у базі даних DynamoDB. Це забезпечує ефективне використання ресурсів, оскільки Lambda функції активуються тільки при виникненні події і виконуються із заданими обчислювальними параметрами, що дозволяє зменшити витрати на інфраструктуру.

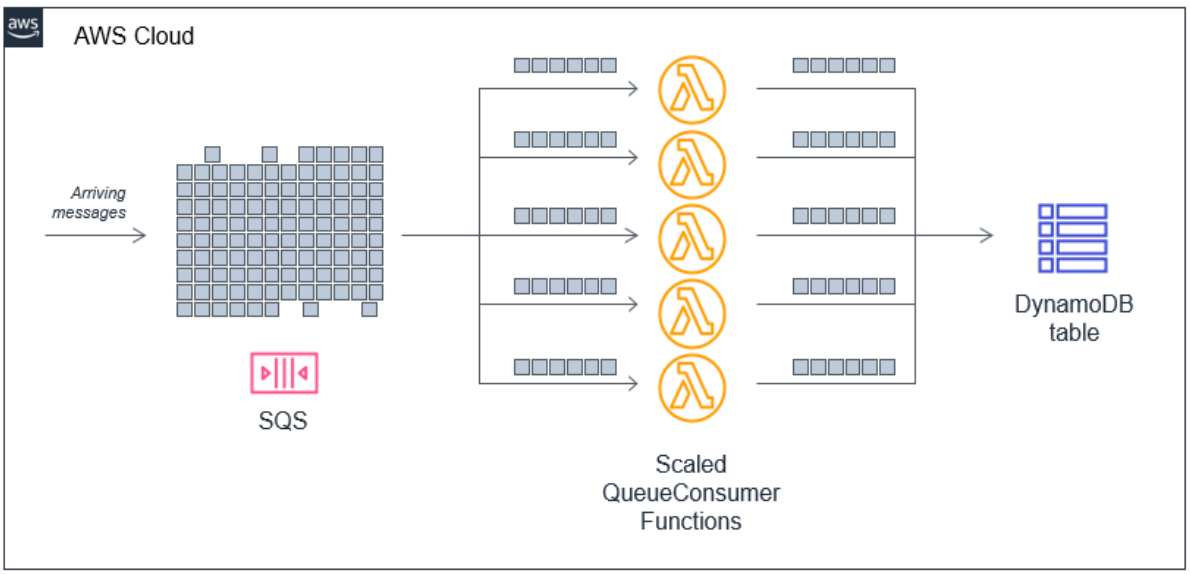

Рисунок 2.4 - Діаграма автоматичного масштабування Lambda

AWS Lambda автоматично масштабується під навантаження, що забезпечує відмовостійкість та швидку обробку запитів навіть при пікових навантаженнях. Наприклад, у системі, що обслуговує тисячі запитів щомиті, Lambda функції можуть динамічно створювати нові екземпляри для обробки запитів без жодного впливу на продуктивність додатка. AWS Lambda також підтримує інтеграцію з іншими сервісами AWS, такими як S3, DynamoDB та API Gateway, що створює єдине середовище для розробки та впровадження функціональності.

2.5. API Gateway як інструмент для маршрутизації запитів

AWS API Gateway є ключовим компонентом, що забезпечує маршрутизацію запитів у безсерверній архітектурі. Він дозволяє створювати, розгортати та обслуговувати RESTful API, які виступають як інтерфейс для користувацьких запитів, спрямованих до Lambda функцій чи інших сервісів. API Gateway допомагає розподілити запити на різні сервіси, забезпечуючи гнучку оркестрацію мікросервісів та управління трафіком.

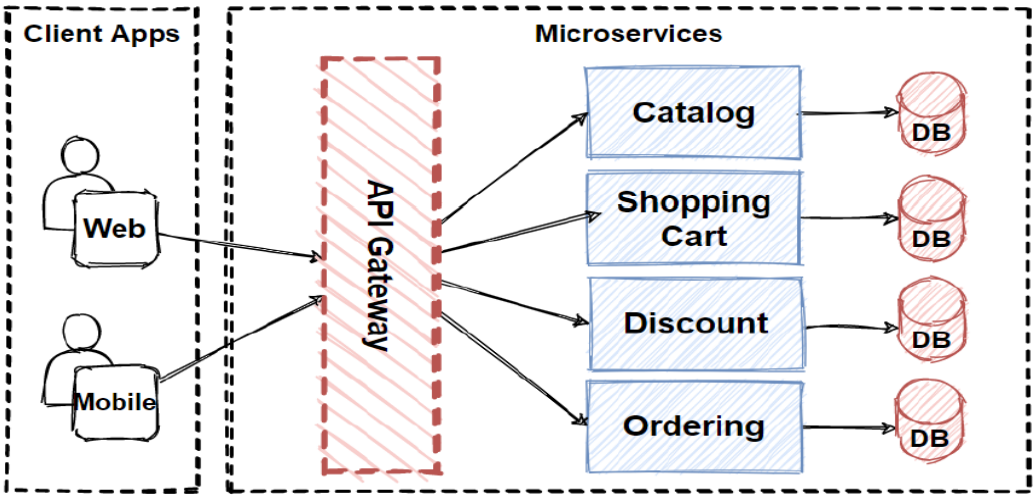

Рисунок 2.5 - Схема аутентифікації та контролю доступу через API Gateway

API Gateway підтримує як синхронні, так і асинхронні виклики, що дозволяє обробляти різні типи запитів з урахуванням потреб користувачів і характеристик додатка. Наприклад, для обробки важливих запитів у режимі

реального часу API Gateway направляє їх до Lambda функцій з найвищим пріоритетом.

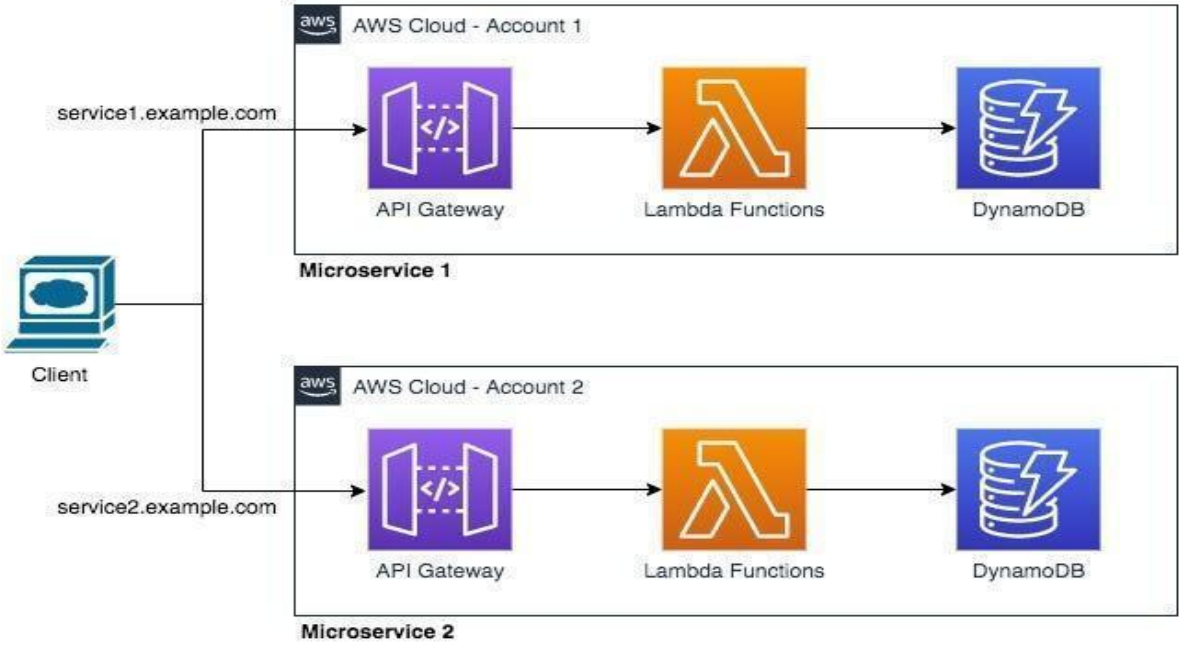

Рисунок 2.6 - Схема взаємодії API Gateway та Lambda.

Крім того, API Gateway забезпечує безпеку доступу до сервісів, підтримуючи різні механізми аутентифікації та авторизації, включаючи IAM, OAuth та Cognito.

2.6 Використання AWS SAM (Serverless Application Model) для розробки та управління інфраструктурою

AWS Serverless Application Model (SAM) є фреймворком, який спрощує процес розробки, тестування та управління безсерверними додатками. SAM дозволяє описувати інфраструктуру додатка у вигляді коду за допомогою шаблонів, що полегшує розгортання та управління ресурсами. Основною перевагою SAM є можливість інтеграції з іншими сервісами AWS, такими як AWS Lambda, API Gateway, DynamoDB та S3, що дозволяє будувати цілісні безсерверні рішення, не залишаючи меж AWS.

Однією з важливих особливостей SAM є використання декларативного підходу до опису інфраструктури. За допомогою шаблонів у YAML-форматі

можна описати ресурси додатка, включно з Lambda-функціями, API Gateway, базами даних і правами доступу, які необхідні для виконання цих функцій. Такий підхід дає змогу легко зберігати всю інфраструктуру додатка у вигляді коду (Infrastructure as Code, IaC), що забезпечує відтворюваність і прискорює процес розгортання, оскільки конфігурацію можна відновити в будь-який час, просто використавши шаблон SAM.

Після написання шаблону, SAM надає команди для компіляції, тестування, локального запуску та розгортання додатка на AWS. Використовуючи команду *sam deploy*, розробники можуть автоматично створити необхідні ресурси, і SAM забезпечить їхнє коректне налаштування та підключення. Це знімає з команди розробників необхідність вручну створювати кожен ресурс і налаштовувати його, що знижує ймовірність помилок та пришвидшує процес розгортання. Також SAM автоматично застосовує найкращі практики AWS для безпеки та масштабування, забезпечуючи відповідність вимогам продуктивності та безпеки додатка.

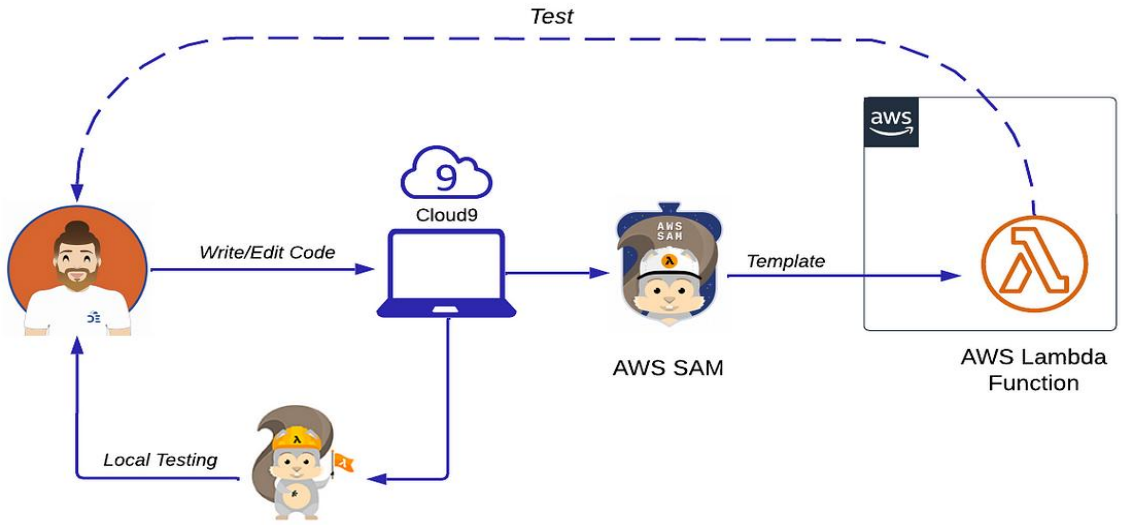

Рисунок 2.7 - Діаграма процесу деплойменту SAM

Шаблони SAM дозволяють чітко та лаконічно описати архітектуру додатка, створюючи структурований опис ресурсів у вигляді YAML-документів. Наприклад, для Lambda-функцій можна вказати назву, обчислювальні ресурси (пам'ять, таймаути) та тригери, такі як HTTP-запити через API Gateway. SAM

також підтримує специфічні параметри, які полегшують опис складних додатків з кількома компонентами, дозволяючи налаштувати зв'язки між ними.

При розгортанні додатка за допомогою SAM, фреймворк автоматично створює та налаштовує всі ресурси відповідно до описаних у шаблоні параметрів. Це включає не лише створення ресурсів, а й забезпечення їхнього коректного зв'язку, що особливо корисно для мікросервісних додатків з високим рівнем взаємодії між компонентами. Крім того, SAM дозволяє виконувати оновлення конфігурації з мінімальними перебоями в роботі додатка, завдяки чому зміни можна впроваджувати без суттєвих збоїв у продуктивному середовищі.

SAM також надає можливості для управління версіями та контролю доступу до ресурсів, що підвищує безпеку інфраструктури. За допомогою SAM Policies можна визначати права доступу для кожної функції Lambda, що дозволяє обмежити доступ лише до тих ресурсів, які необхідні для виконання певного завдання. Це відповідає принципу найменших привілеїв, що є важливим для захисту безсерверних додатків.

2.7. Інструменти для моніторингу та логування (AWS CloudWatch). Налаштування метрик та алертів у CloudWatch для моніторингу стану системи

Amazon CloudWatch є потужним інструментом для моніторингу та управління логами в хмарі AWS. Він дозволяє збирати та відстежувати метрики, журнали та події з різних сервісів AWS, таких як Lambda, EC2, DynamoDB та багато інших, надаючи можливість отримувати вичерпну інформацію про стан додатків, серверів та інфраструктури. CloudWatch допомагає адміністраторам і розробникам виявляти проблеми в реальному часі, оптимізувати продуктивність і забезпечувати високу доступність систем.

CloudWatch дозволяє здійснювати моніторинг ключових метрик, таких як використання ресурсів, швидкість обробки запитів, кількість помилок і затримка

в системі. Завдяки гнучким налаштуванням, користувачі можуть створювати алерти, які сповіщають про непередбачувані зміни в метриках, що може свідчити про проблеми в додатку або інфраструктурі. Наприклад, можна налаштувати тривожні сигнали для ситуацій, коли використання пам'яті або CPU перевищує певний поріг, або коли кількість помилок у логах перевищує задану кількість.

Завдяки CloudWatch, розробники можуть отримувати інформацію про роботу окремих компонентів безсерверних додатків, таких як функції AWS Lambda, що дозволяє точно визначати джерело проблем. Інтеграція CloudWatch з іншими сервісами AWS дозволяє створювати автоматизовані дії для усунення проблем, наприклад, перезапуск функцій Lambda, збільшення кількості обчислювальних ресурсів або перенаправлення трафіку.

Для ефективного моніторингу важливо налаштувати метрики, які дозволяють оцінити як загальний стан системи, так і окремі показники. Важливими є такі метрики як Latency, Error Count, Throttles (для Lambda-функцій), а також метрики для API Gateway, такі як Count і 4xx/5xx Error Rates. Це дає змогу не тільки моніторити поточний стан, а й прогнозувати можливі проблеми заздалегідь.

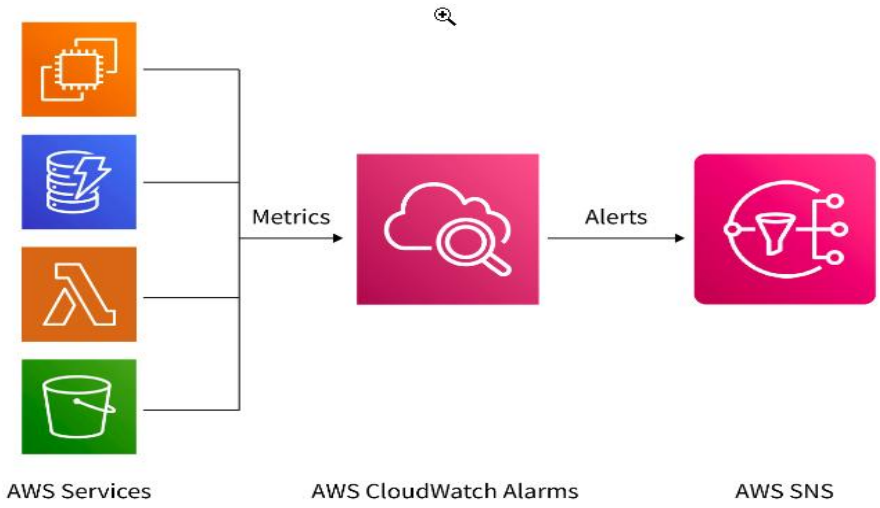

Рисунок 2.8 - Діаграма налаштування метрик та алертів у AWS CloudWatch

Налаштування метрик та алертів у AWS CloudWatch є ключовим для підтримки належного стану безсерверних систем. Для цього необхідно

визначити основні показники продуктивності (KPI), на основі яких будуть відстежуватися й оцінюватися всі критичні аспекти роботи системи. CloudWatch дозволяє налаштувати моніторинг для конкретних функцій або сервісів, таких як AWS Lambda, API Gateway, DynamoDB чи S3.

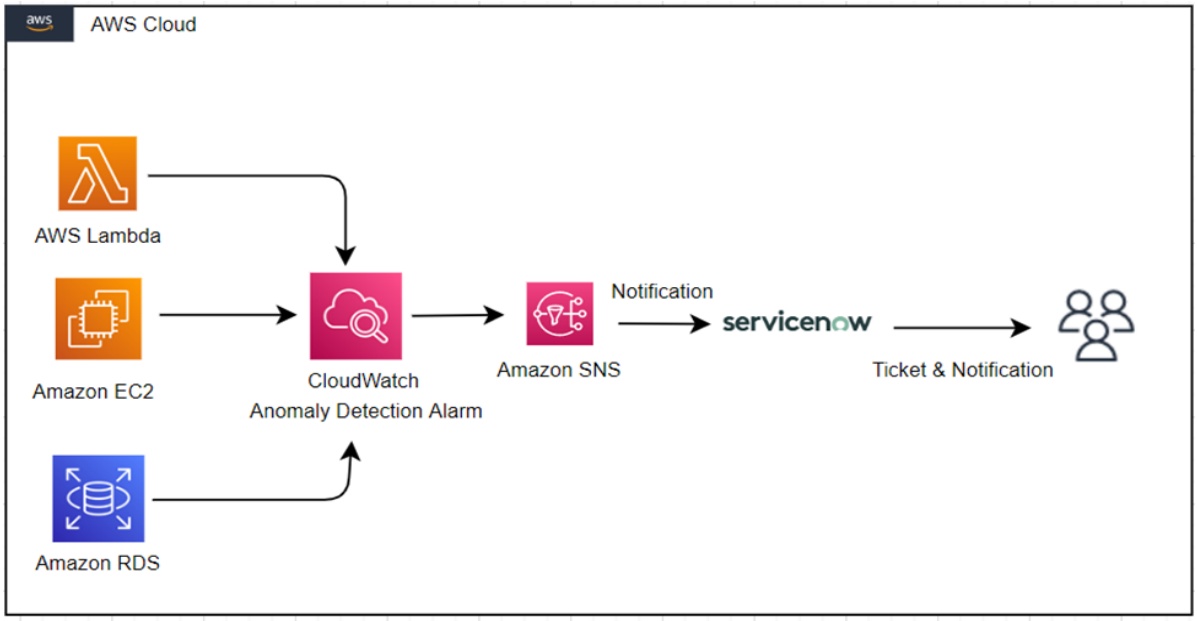

Рисунок 2.9 - Діаграма інтеграції AWS CloudWatch з іншими сервісами

Для того щоб налаштувати метрики, потрібно здійснити такі дії:

1. Визначити, які параметри є критичними для вашого додатку. Для функцій Lambda це можуть бути Duration, Error count, Throttles, а для API Gateway – Count, 4xx та 5xx error rates.

2. Створити CloudWatch Alarm – систему сповіщень, яка буде автоматично реагувати на зміни у метриках, перевищення порогових значень або критичні помилки. Наприклад, можна налаштувати alarm для сповіщення через email або SMS, коли кількість помилок у функціях Lambda перевищить 5% або коли затримка обробки запиту API стане більше 1 секунди.

3. Використовувати CloudWatch Logs для аналізу детальних логів, що генеруються сервісами, і створювати алерти на основі певних шаблонів у логах. Це дозволяє виявляти та автоматично реагувати на помилки або аномальні події, які можуть бути не помітні через метрики [11].

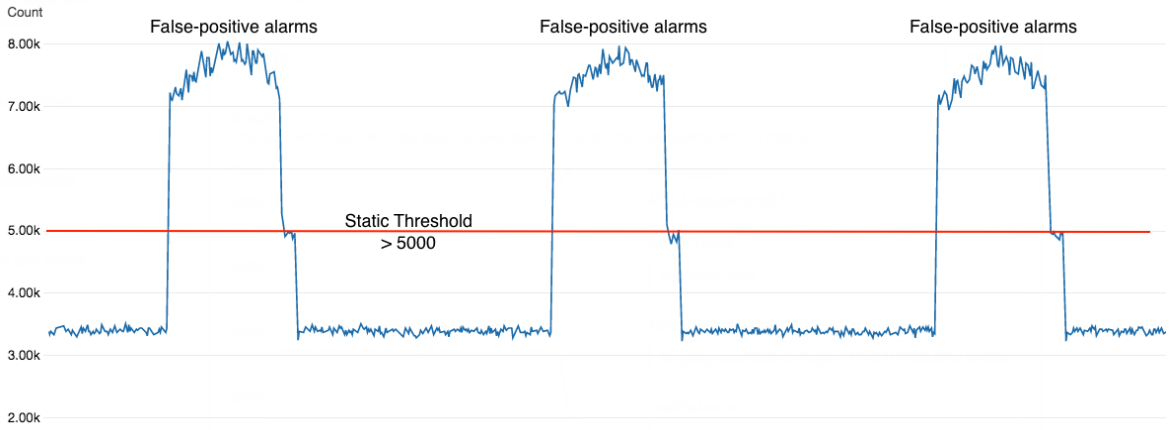

Рисунок 2.10 - Приклад аномалій в CloudWatch

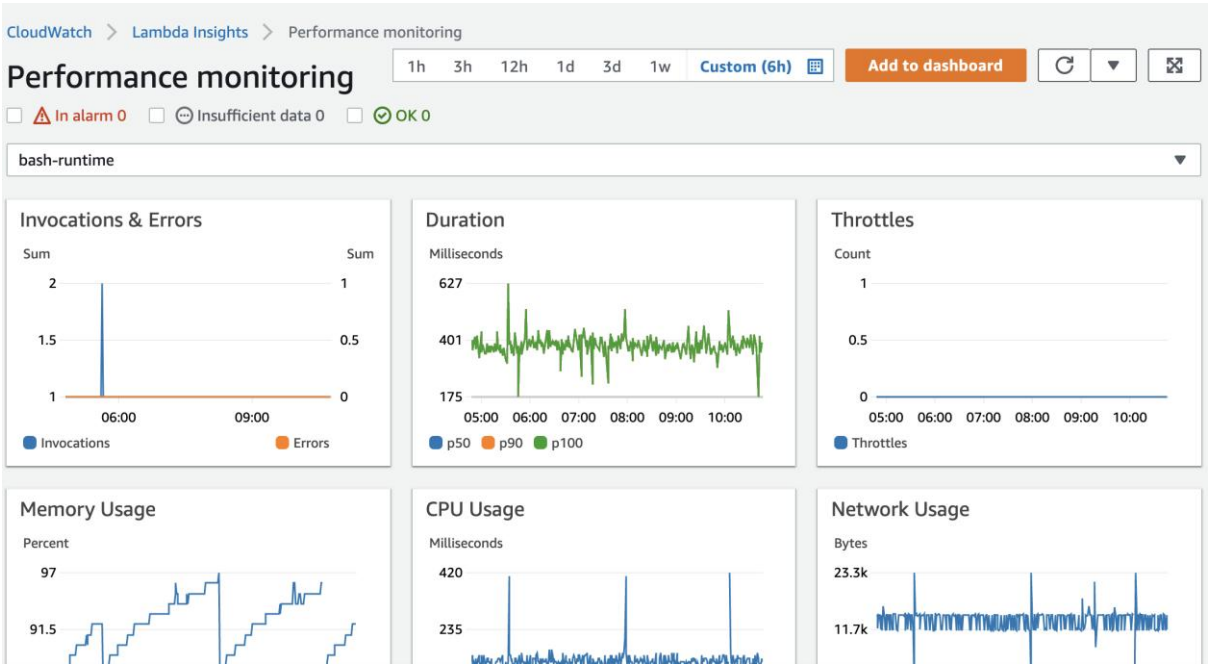

Рисунок 2.11 - CloudWatch Lambda Dashboard

CloudWatch також підтримує створення **дашбордів**, які дозволяють візуалізувати показники в реальному часі. Це дає змогу адміністраторам легко відстежувати стан системи і швидко реагувати на зміни в метриках. Для безсерверних додатків важливо, щоб дашборди включали метрики на рівні функцій Lambda та їх інтеграцій з іншими сервісами, наприклад, API Gateway та базами даних.

РОЗДІЛ 3
РОЗРОБКА ТА ВПРОВАДЖЕННЯ АРХІТЕКТУРИ ВЕБ-ДОДАТКУ

3.1 Вибір існуючого веб-додатку та адаптація його до безсерверної архітектури за допомогою AWS Lambda та API Gateway

У цьому розділі я обґрунтую вибір існуючого веб-додатку для адаптації до безсерверної архітектури та опишу, як я використовую AWS Lambda та API Gateway для реалізації безсерверного рішення. З огляду на те, що мій проект передбачає створення системи для безперебійної роботи веб-додатку з використанням AWS, вибір існуючого веб-додатку був обґрунтований кількома факторами.

Перш за все, вибір веб-додатку обумовлений його здатністю працювати в безсерверному середовищі. Для цієї мети було обрано веб-додаток, який має добре розвинену архітектуру для інтеграції з API, що дозволяє без проблем використовувати функціональність AWS Lambda. Веб-додаток повинен бути достатньо легким для масштабування і здатним підтримувати інтеграцію з різними AWS-сервісами, такими як API Gateway для маршрутизації запитів і DynamoDB для зберігання даних.

Після вибору веб-додатку на черзі стояла задача адаптації його архітектури до безсерверного середовища, використовуючи можливості AWS. Я вирішив використовувати AWS Lambda для обробки бізнес-логіки веб-додатку. Lambda дозволяє виконувати функції на вимогу, що робить її ідеальним вибором для реалізації безсерверної архітектури, де функції запускаються лише за необхідністю, без необхідності постійного перебування сервера в роботі.

У рамках адаптації веб-додатку було визначено, які частини функціоналу додатку можуть бути переведені в Lambda-функції. Це включає обробку запитів від користувачів, взаємодію з базою даних, обробку файлів, а також різноманітні фонові задачі. Кожну з цих задач я інкапсулював в окрему Lambda-функцію, що

дозволяє масштабувати її відповідно до навантаження та забезпечує відмовостійкість.

Оскільки веб-додаток вимагає взаємодії з користувачем через інтерфейс, для цього був обраний AWS API Gateway, що служить точкою входу для всіх HTTP-запитів. API Gateway забезпечує маршрутизацію запитів до відповідних Lambda-функцій і виконує роль проксі-сервера, який передає запити від користувачів до відповідних обробників. Це дає змогу масштабувати веб-додаток, обробляючи велику кількість запитів без необхідності управління фізичними серверами.

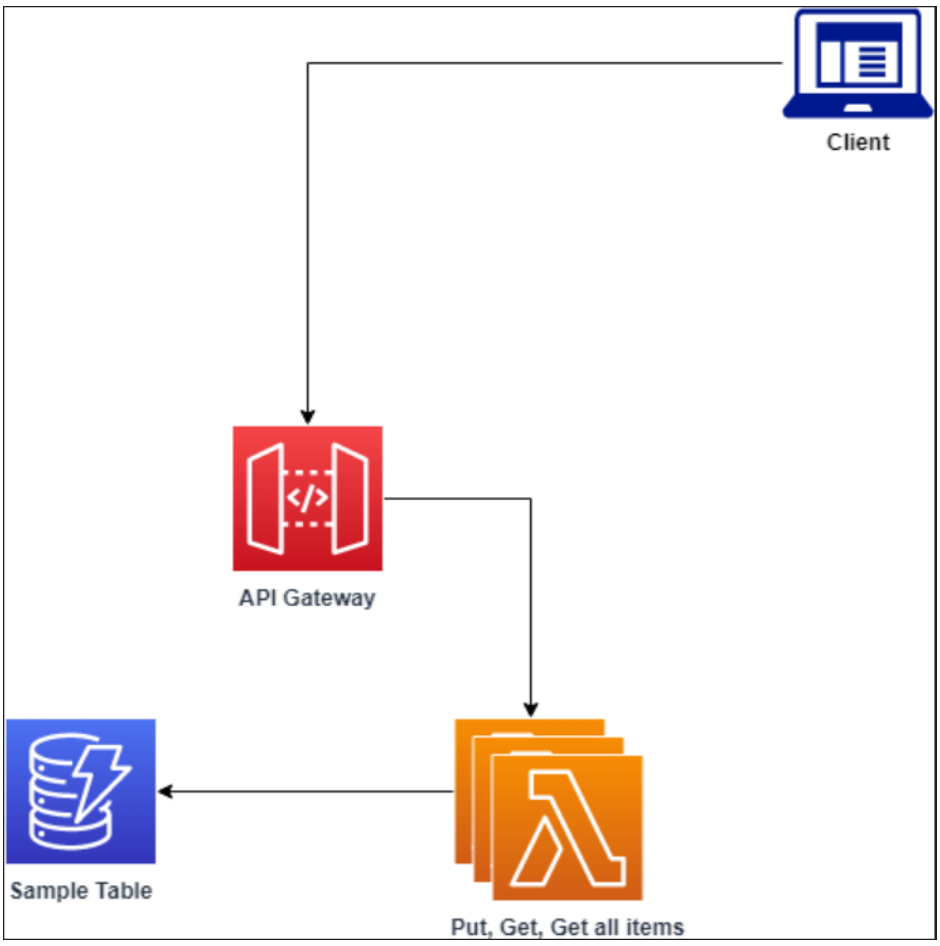

Рисунок 3.1 - Діаграма back-end архітектури додатку

Для зберігання даних, що використовуються веб-додатком, було вирішено використовувати Amazon DynamoDB – повністю керовану NoSQL базу даних. DynamoDB має високу швидкість обробки запитів і автоматично масштабується відповідно до навантаження, що є важливим для підтримки безперебійної роботи додатку. Застосування DynamoDB дозволяє також уникнути необхідності

налаштування складної інфраструктури для бази даних, оскільки всі аспекти її управління беруть на себе сервіси AWS.

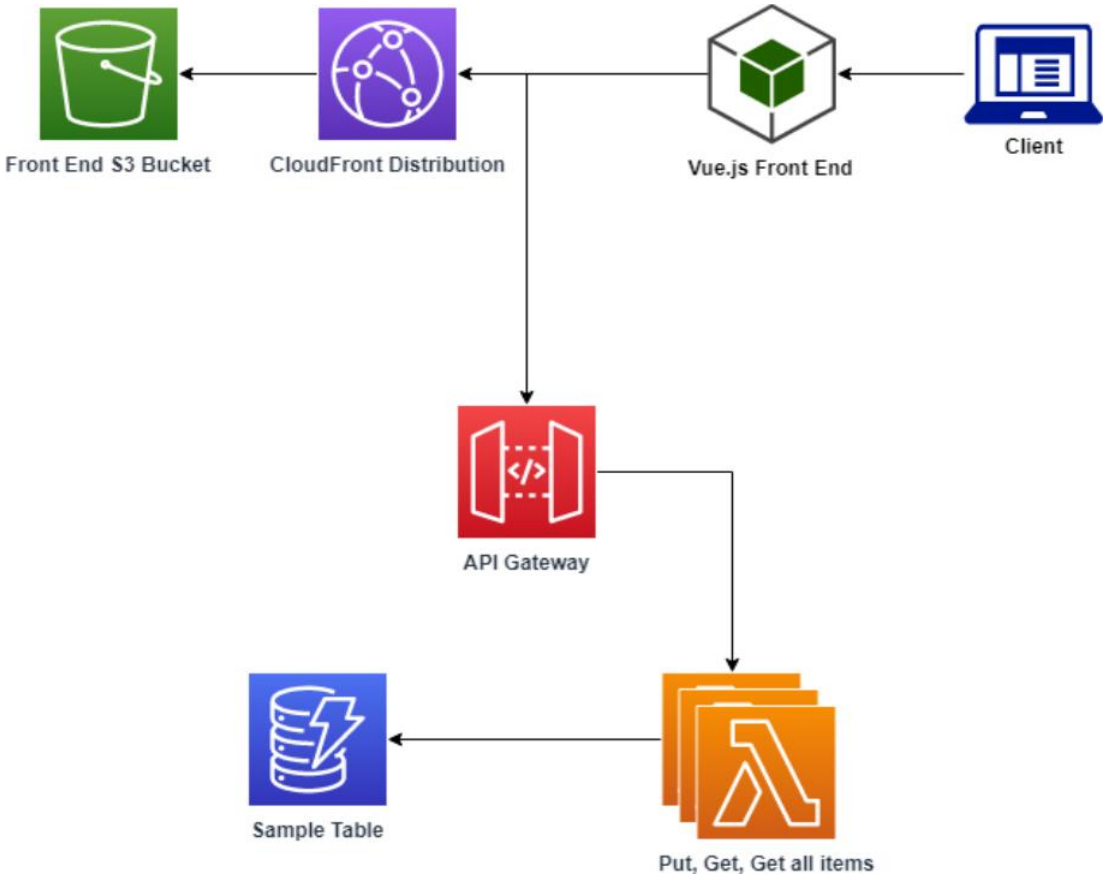

Рисунок 3.2 - Діаграма архітектури додатку

Завдяки використанню AWS Lambda, API Gateway та DynamoDB, система здатна працювати в безсерверному середовищі, забезпечуючи високу доступність, масштабованість та надійність. Веб-додаток, після адаптації до безсерверної архітектури, працює без необхідності в управлінні фізичними серверами і може швидко реагувати на зміну навантаження.

У наступних розділах я розгляну додаткові аспекти безперебійної роботи веб-додатку, зокрема, забезпечення високої доступності, відмовостійкості та безпеки на основі використовуваних технологій AWS.

3.2 Побудова мінімально життєздатного продукту (MVP) з використанням SAM для автоматизації створення ресурсів

Для реалізації мінімально життєздатного продукту (MVP) я вирішив використовувати AWS Serverless Application Model (SAM), оскільки це дозволяє ефективно автоматизувати створення та управління ресурсами. У SAM шаблоні визначені всі необхідні компоненти та ресурси, що дозволяє швидко розгорнути інфраструктуру без зайвих ручних налаштувань. Таким чином, MVP базується на чіткій архітектурі, яка включає необхідні функції для базової роботи системи.

Ключовим елементом SAM-шаблону є файл *template.yaml* (Додаток А), де визначені всі ресурси, які необхідні для функціонування веб-додатку. У цьому файлі я зазначив основні компоненти для MVP, включаючи:

- Lambda-функції для обробки запитів. Кожна Lambda-функція реалізує певну частину бізнес-логіки веб-додатку.
- API Gateway для маршрутизації запитів до Lambda-функцій. API Gateway працює як точка входу для HTTP-запитів, дозволяючи користувачам взаємодіяти з додатком через веб-інтерфейс. У SAM-шаблоні цей компонент налаштований для передачі запитів до відповідних Lambda-функцій залежно від кінцевої точки запиту.
- S3 Bucket з CloudFront для розміщення фронтенду веб-додатку. У шаблоні SAM передбачено створення S3 Bucket для зберігання статичних файлів фронтенду. Додатково інтегрований CloudFront Distribution для забезпечення швидкого доступу до веб-додатку користувачам з різних регіонів.
- DynamoDB як база даних для зберігання даних додатку. В SAM-шаблоні налаштована DynamoDB таблиця, яка є основним сховищем даних для веб-додатку. DynamoDB забезпечує надійне та масштабоване зберігання даних без необхідності додаткового управління інфраструктурою.

У SAM-шаблоні були реалізовані основні функції для автоматизації процесу розгортання, а саме:

1. Автоматичне створення та оновлення Lambda-функцій. SAM дозволяє визначити всю конфігурацію Lambda-функцій в шаблоні, включно з необхідними правами доступу (IAM ролями) і змінними середовища, що спрощує підтримку додатку.

2. Налаштування маршрутів для API Gateway. Вказані кінцеві точки API Gateway та їхнє відповідне призначення для кожної Lambda-функції, що дозволяє легко масштабувати API, додаючи нові маршрути в шаблоні.

3. Розгортання статичних ресурсів. S3 Bucket автоматично створюється для розміщення статичних файлів фронтенду, а CloudFront забезпечує глобальне кешування контенту.

4. Конфігурація бази даних DynamoDB. SAM також забезпечує визначення схеми таблиць DynamoDB, що дозволяє миттєво створити потрібну базу даних при розгортанні.

Використання SAM значно спрощує автоматизацію створення ресурсів, необхідних для MVP. Завдяки цій автоматизації я можу швидко розгорнути повнофункціональну інфраструктуру, забезпечуючи її масштабованість, безпеку та високу доступність. Використовуючи SAM, я можу вносити зміни в архітектуру додатку, лише оновлюючи *template.yaml*, що особливо зручно для процесу швидкого прототипування та покращення продукту.

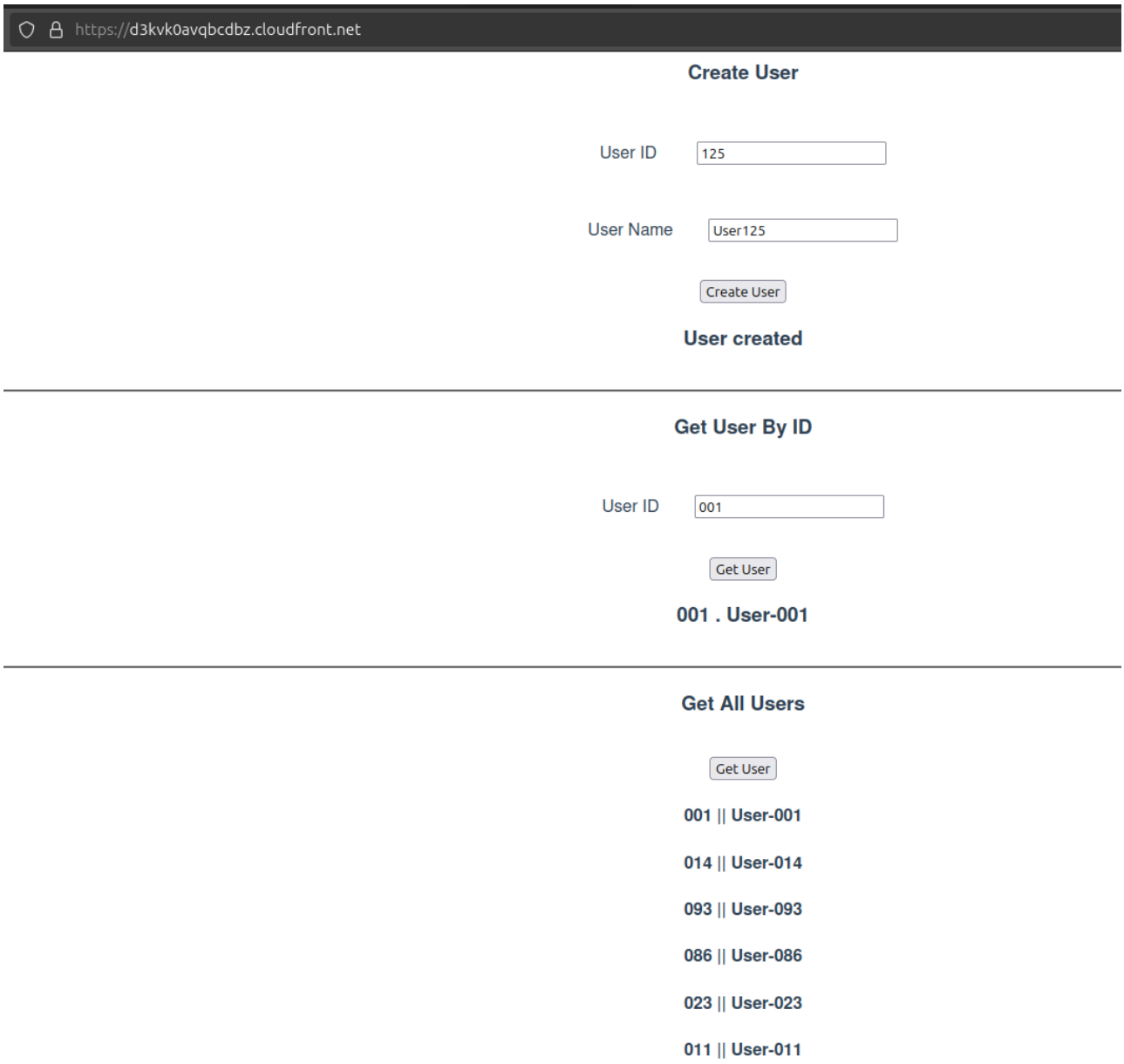

Рисунок 3.3 - Головна сторінка веб-додатку

Далі, у наступному розділі я детально розгляну процес навантажувального тестування для оцінки продуктивності безсерверної архітектури, а також зроблю підсумки щодо ефективності та надійності конфігурації SAM у розробленій системі.

РОЗДІЛ 4
ТЕСТУВАННЯ ПРОДУКТИВНОСТІ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

Тестування надійності та продуктивності є важливим етапом оцінки готовності серверлес-додатка до високих навантажень і забезпечення стабільної роботи системи. Для цього проекту, який побудований на базі AWS Lambda, я провів стрес-тестування та тестування на навантаження, щоб перевірити масштабованість і стійкість до зростання трафіку.

Для перевірки надійності та продуктивності системи в умовах різноманітного навантаження було важливо забезпечити повне тестування масштабованості функцій AWS Lambda, що є ядром безсерверної архітектури мого веб-додатку. Основною метою таких тестів було гарантувати стабільну роботу та швидке реагування сервісу при збільшенні кількості запитів і забезпечити користувачам якісний досвід навіть при високому навантаженні. Я вирішив використовувати JMeter для тестування навантаження та стрес-тестування, оскільки він надає гнучкі можливості для створення детальних сценаріїв тестування та автоматизації процесу [12].

JMeter дозволив мені створити кілька типів навантаження, імітуючи різноманітні реальні сценарії, зокрема:

1. Тестування нормального навантаження – сценарій, що відображав типову кількість запитів від середньостатистичних користувачів. Це тестування забезпечувало впевненість у стабільності додатку в умовах звичайної експлуатації.

2. Тестування пікових навантажень – сценарії, що симулювали різке збільшення запитів, яке може виникнути, наприклад, під час рекламних кампаній чи інших масових заходів. Це дозволяло оцінити реакцію системи на раптовий ріст навантаження, виявити затримки або збої, які могли б порушити її стабільність.

3. Стрес-тестування – імітація максимального навантаження, що перевищувало очікуваний рівень використання, з метою виявлення критичних

точок, коли продуктивність Lambda починає деградувати. Це дозволило встановити ліміти використання та визначити межі, за якими необхідно оптимізувати архітектуру.

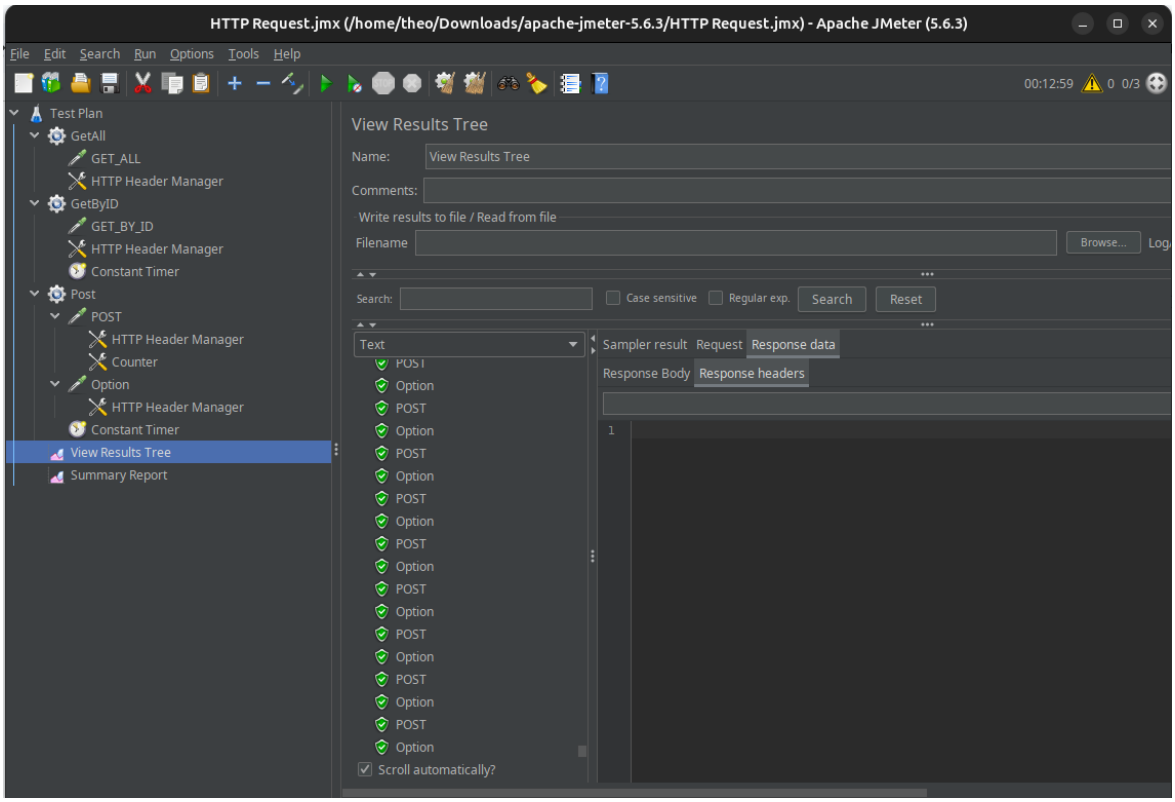

Рисунок 4.1 - Вінко Jmeter

У рамках проекту для оцінки продуктивності AWS Lambda я провів серію тестів з використанням JMeter. Тестування було спрямоване на перевірку масштабованості та відмовостійкості серверлес-архітектури під різними рівнями навантаження.

Нижче наведено результати тестування.

1. Тест 1.

Умови: 1 користувач, 1 запит в секунду, тривалість — 1 хвилина.

Отримані результати:

- Середній час відповіді: 168 мс.
- Відсутність помилок (0% errors).
- Максимальний час відповіді: 2 секунди на функцію GET_ALL.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughp...	Received ...	Sent KB/s...	Avg. Bytes
POST	60	161	85	311	58.49	0.00%	51.2/min	0.53	0.51	636.0
GET_BY_ID	60	150	64	305	62.88	0.00%	1.1/sec	1.35	0.72	1230.0
Option	60	25	9	247	40.78	0.00%	51.4/min	0.51	0.49	604.0
GET_ALL	60	338	164	2032	263.18	0.00%	2.9/sec	101.02	1.88	35072.0
TOTAL	240	168	9	2032	178.88	0.00%	3.4/sec	30.87	2.07	9385.5

Рисунок 4.2 - Результати тесту 1

2. Тест 2.

Умови: 3 користувачі, 1 запит в секунду, тривалість — 1 хвилина.

Отримані результати:

- Середній час відповіді: 158 мс.
- Відсутність помилок (0% errors).
- Максимальний час відповіді: 1.5 секунди на функцію GET_ALL.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Thro
POST	180	157	83	770	77.23	0.00%	
GET_BY_ID	180	136	64	465	54.38	0.00%	
Option	180	24	8	1072	81.28	0.00%	
GET_ALL	180	314	150	1519	216.45	0.00%	
TOTAL	720	158	8	1519	162.18	0.00%	1

Рисунок 4.3 - Результати тесту 2

3. Тест 3:

Умови: 10 користувачів, 1 запит в секунду, тривалість — 10 хвилин.

Отримані результати:

- Середній час відповіді: 258 мс.
- Помилки: 3%.
- Максимальний час відповіді: 5 секунд на функцію GET_ALL.

Label	# Samples	Average ↓	Min	Max	Std. Dev.	Error %	Throughp...	Received ...	Sent KB/s...	Avg. Bytes
GET_ALL	600	451	178	4421	366.25	10.67%	2.2/sec	72.99	1.41	33769.8
GET_BY_ID	600	284	121	5354	343.04	3.83%	58.0/min	1.16	0.62	1232.7
POST	600	255	118	5298	303.32	0.00%	46.3/min	0.48	0.46	636.0
Option	600	42	20	1135	98.47	0.00%	46.4/min	0.46	0.45	604.0
TOTAL	2400	258	20	5354	330.94	3.62%	3.1/sec	27.25	1.89	9060.6

Рисунок 4.4 - Результати тесту 3

У процесі проведених тестів були отримані наступні результати:

- При низькому навантаженні (Тести 1 та 2), серверлес-архітектура продемонструвала високу ефективність. Час відповіді залишався стабільно низьким, а помилок не було.

- При значному збільшенні навантаження (Тест 3) середній час відповіді зріс, а відсоток помилок досяг 3%. Максимальний час відповіді також суттєво збільшився до 5 секунд.

Нижче наведені причини можливих збоїв у Тесті 3:

1. Межі одночасного виконання Lambda. AWS Lambda має обмеження на кількість одночасно виконуваних функцій. Якщо кількість запитів перевищує цей поріг, нові запити очікують, поки звільняться ресурси, що призводить до затримок.

2. Холодні старти: Під час різкого збільшення навантаження AWS Lambda може створювати нові контейнери для обробки запитів. Ініціалізація нових контейнерів (холодний старт) додає затримки, особливо якщо обробка даних включає складні залежності або великий обсяг ініціалізації.

3. Обмеження бази даних (DynamoDB). DynamoDB, яка використовується як сховище, також має обмеження на швидкість читання/запису. При одночасних запитах ці обмеження могли вплинути на продуктивність.

4. Обробка функції GET_ALL. Функція GET_ALL, яка має максимальний час відповіді у всіх тестах, можливо, не оптимізована для роботи під великим навантаженням. Наприклад, якщо функція виконує сканування великого обсягу даних, це може призводити до затримок.

На основі отриманих результатів від проведених тестів оцінки продуктивності AWS Lambda можна зробити наступні висновки і надати такі рекомендації, зокрема:

1. Оптимізація функції GET_ALL. Переглянути логіку обробки запитів. Можливо, слід впровадити фільтри або індексацію в DynamoDB, щоб зменшити обсяг оброблюваних даних.

2. Збільшення лімітів виконання Lambda. Перевірити та за необхідності збільшити ліміти одночасного виконання функцій у налаштуваннях AWS.

3. Активація теплих контейнерів. Використання підходів для підтримки теплих контейнерів (наприклад, періодичне викликання функцій) може зменшити вплив холодних стартів.

4. Додавання CloudWatch алертів. Налаштувати алерти для моніторингу затримок та відсотків помилок, щоб мати можливість оперативно реагувати на збої.

5. Автоматичне масштабування DynamoDB. Увімкнути автоматичне масштабування продуктивності DynamoDB, щоб уникнути зниження швидкості читання/запису під час пікового навантаження.

Це тестування продемонструвало здатність серверлес-архітектури AWS Lambda адаптуватися до середнього навантаження та виявило можливості для подальшої оптимізації, які дозволять створити більш стабільну та продуктивну систему.

В ході тестів я спостерігав поведінку Lambda в умовах зростання навантаження, оцінюючи такі показники, як час відгуку, затримки та споживання ресурсів. JMeter зібрав детальні метрики, що допомогло визначити точки оптимізації для підвищення ефективності масштабування.

РОЗДІЛ 5
ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

Охорона праці регулюється рядом законодавчих і нормативних документів, які визначають основні вимоги до забезпечення безпеки праці. В Україні до ключових документів належать :

● Кодекс законів про працю України (КЗпП) — визначає права працівників і роботодавців у сфері охорони праці.

● Закон України "Про охорону праці" — встановлює загальні принципи організації охорони праці, права працівників на безпечні умови праці та обов’язки роботодавців [17].

● Державні санітарні норми і правила — забезпечують регулювання гігієнічних умов праці, які попереджують вплив шкідливих факторів на здоров’я працівників.

● Нормативні акти Держпраці України — регламентують спеціалізовані аспекти охорони праці, такі як безпека при роботі з електрообладнанням, організація робочих місць, технічні стандарти тощо.

В контексті розробки програмного забезпечення та безсерверних обчислень, увага зосереджується на інформаційній безпеці та ергономіці робочого місця для працівників, які працюють переважно за комп’ютерами.

Оцінка ризиків є ключовим етапом у забезпеченні охорони праці, і яка складається з таких кроків [7]:

1. Визначення потенційних небезпек.

○ Тривале сидіння за комп’ютером, що викликає м’язові напруження та проблеми з хребтом.

○ Високий рівень стресу через дедлайни та відповідальність за проекти.

○ Ризики кіберзагроз, які можуть спричинити втрату даних або компрометацію систем.

2. Оцінка ймовірності та наслідків небезпек.

○ Висока ймовірність виникнення проблем із зором через тривалу роботу з монітором.

○ Потенційно серйозні наслідки через витік конфіденційних даних компанії.

3. Розробка заходів для мінімізації ризиків.

○ Організація регулярних перерв і фізичних вправ для працівників.

○ Встановлення захищених мережевих протоколів і використання шифрування.

○ Навчання персоналу правилам інформаційної безпеки.

Також, немаловажним є ррозробка профілактичних заходів, які повинні включати наступні аспекти:

○ Ергономіка робочого місця [25] (регульовані крісла та столи, які підтримують правильну поставу; монітори на рівні очей, щоб уникнути перенапруги шиї; достатнє освітлення, щоб зменшити навантаження на очі.

○ Психологічна безпека (підтримка здорового робочого клімату; регулярні індивідуальні зустрічі для обговорення труднощів і проблем; забезпечення програм психологічної підтримки).

○ Інформаційна безпека (встановлення сучасного антивірусного програмного забезпечення; регулярне оновлення систем безпеки; використання двофакторної автентифікації).

Охорона праці та безпека в надзвичайних ситуаціях є важливими аспектами організації роботи, які мають забезпечити як комфорт працівників, так і стійкість технологічних процесів. У сфері інформаційних технологій, де основна діяльність виконується за комп’ютерами, ці аспекти охоплюють широкий спектр заходів – від ергономіки робочого місця до запобігання кібератакам.

5.1 Охорона праці під час роботи над проєктом

Робота розробників програмного забезпечення супроводжується значними навантаженнями на зір, хребет та нервову систему, тому належна організація

робочого місця є першочерговим завданням. Освітлення має бути достатнім і рівномірним, щоб зменшити втому очей. Робочий стіл і крісло повинні відповідати вимогам ергономіки: забезпечувати правильну посадку та підтримку спини. Важливо робити регулярні перерви для фізичної активності, щоб запобігти захворюванням опорно-рухового апарату.

Додатково слід дотримуватися вимог електробезпеки. Усі електроприлади, зокрема комп'ютери, монітори та сервери, мають бути справними і регулярно перевірятися. Користування подовжувачами повинно бути мінімальним, а їх розташування – зручним і безпечним, щоб уникнути ризику пошкодження кабелів чи спотикання.

Одним із ключових завдань є забезпечення готовності до надзвичайних ситуацій, таких як пожежі, відключення електроенергії чи природні катастрофи. Приміщення повинно бути обладнане справними вогнегасниками, планами евакуації та сигналізацією. Персонал має бути проінструктований про дії у разі пожежі, а також про безпечний спосіб покидання будівлі.

Для запобігання втратам даних та перебоїв у роботі системи доцільно використовувати джерела безперебійного живлення (UPS) та резервні сервери. Це дозволить зберегти працездатність проєкту навіть у разі раптового відключення електроенергії. Крім того, варто впроваджувати резервне копіювання даних на регулярній основі, зберігаючи їх у захищених хмарних сховищах.

5.2 Захист інформації та кібербезпека

Особливу увагу слід приділити захисту даних від кіберзагроз. У проєкті необхідно використовувати сучасні методи шифрування для збереження конфіденційності даних, налаштовувати багаторівневий контроль доступу до ресурсів, а також впроваджувати регулярний моніторинг безпеки системи. У разі виявлення підозрілої активності повинні бути передбачені алгоритми негайного

реагування, які включають ізоляцію уражених сегментів та повідомлення відповідальних осіб.

Також важливо враховувати можливість техногенних чи природних катастроф. У разі землетрусу, затоплення чи інших форс-мажорів персонал має бути ознайомлений із планами евакуації та правилами збереження життя і здоров’я. Для швидкого відновлення роботи системи необхідно мати резервні сервери, які знаходяться в іншому географічному регіоні, що забезпечує відмовостійкість системи.

Виконання правил охорони праці та заходів безпеки не лише забезпечує збереження здоров’я працівників, але й сприяє ефективній та безперебійній роботі технологічних процесів. Надійна інфраструктура, правильна організація робочого середовища та підготовленість до надзвичайних ситуацій є ключовими елементами стійкої роботи будь-якого проєкту.

Забезпечення охорони праці та безпеки в надзвичайних ситуаціях вимагає комплексного підходу, що враховує нормативно-правову базу, оцінку ризиків, профілактику захворювань та реагування на надзвичайні ситуації. У сфері розробки веб-додатків важливо приділяти увагу як фізичному, так і психологічному стану працівників, забезпечуючи їм безпечні умови для продуктивної діяльності [26].

5.3 Безпека в надзвичайних ситуаціях

Планування дій у разі надзвичайних ситуацій є важливою складовою охорони праці. Для ІТ-компаній у цьому напрямку повинні бути здійсненні наступні кроки:

1. Планування дій на випадок пожежі.
    - Інструктаж персоналу щодо евакуаційних маршрутів.
    - Оснащення приміщень вогнегасниками та протипожежними системами.
    - Регулярні тренування з евакуації.

2. Планування дій у разі кіберзагрози.
    - Резервне копіювання даних на хмарні та локальні сервери.
    - Регулярне оновлення політик доступу до систем.
    - Використання програмного забезпечення для виявлення кіберзагроз.
3. Здійснення медичних заходів.
    - Наявність аптечок першої допомоги на всіх робочих локаціях.
    - Організація тренінгів із надання домедичної допомоги.
    - Впровадження систем моніторингу здоров’я працівників.
4. Вчасне реагування на природні катастрофи.
    - Розробка плану дій для випадків землетрусів, повеней чи інших стихійних лих.
    - Забезпечення працівників необхідними засобами захисту.
    - Створення запасів води, їжі та інших ресурсів у офісах.
5. Проведення інструктажів щодо надзвичайних ситуацій.
    - Регулярне проведення навчань для всіх працівників.
    - Впровадження систем сповіщення про надзвичайні ситуації.
    - Забезпечення доступу до інструкцій і контактів служб екстреної допомоги.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи "Розробка системи для безперебійної роботи веб-додатку із використанням безсерверних обчислень" було успішно реалізовано всі етапи проєкту – від вибору архітектурних підходів до тестування продуктивності й оптимізації інфраструктури. Мета роботи полягала у створенні системи, здатної забезпечувати безперебійну роботу та високу доступність веб-додатку навіть за умов змінних та інтенсивних навантажень.

Кваліфікаційна робота присвячена розробці системи для забезпечення безперебійної роботи веб-додатків, використовуючи технології безсерверних обчислень. Основний акцент робиться на використанні AWS Lambda, як потужного інструменту для виконання функцій в реальному часі.

Переваги безсерверних обчислень в контексті стабільності та оптимізації ресурсів ретельно розглядалися. Аналіз технологій, принципів та платформ дозволяє визначити їхню відповідність потребам розробки веб-додатків.

Зазначено важливість систем комплексного стану для забезпечення ефективного моніторингу та управління безсерверними додатками, що є критичним для забезпечення надійності та продуктивності.

Цей дослідження вказує на перспективність використання безсерверних обчислень у сфері веб-розробки і підкреслює важливість правильного підбору платформ та розробки системи, що забезпечує безперебійну роботу веб-додатків.

У контексті сучасних технологічних тенденцій, використання безсерверних обчислень, зокрема AWS Lambda, може значно полегшити розгортання та адміністрування веб-додатків, знизити витрати та забезпечити ефективність використання ресурсів.

Важливість правильного вибору платформи та належного розгортання системи комплексного стану підкреслюється як ключовий аспект успішної реалізації безсерверних рішень.

Зроблене дослідження має за мету стимулювання подальших робіт у цьому напрямку та внесення вагомого внеску в сучасну парадигму розробки веб-додатків. Впровадження запропонованих рішень може відкрити нові можливості для розробників та покращити загальну ефективність та стабільність веб-додатків в умовах зростаючого обсягу та складності завдань.

На початкових етапах роботи був здійснений огляд існуючих технологій безсерверних обчислень, а також обґрунтований вибір AWS як основної платформи для реалізації проєкту. Завдяки архітектурі на базі AWS Lambda та API Gateway вдалося створити систему, яка може автоматично масштабуватися залежно від рівня навантаження, забезпечуючи стабільну продуктивність та економічну ефективність. Ретельний вибір компонентів архітектури, таких як S3 для зберігання фронтенд-ресурсів, DynamoDB для зберігання даних, та CloudFront для кешування й швидкого доступу до контенту, забезпечив високу доступність і швидкість роботи додатка.

У процесі розробки було створено мінімально життєздатний продукт (MVP) з використанням AWS SAM, який автоматизував процес створення та оновлення ресурсів, спростив управління інфраструктурою та значно пришвидшив розробку. Такий підхід дозволив швидко впроваджувати зміни, забезпечуючи гнучкість і масштабованість проєкту.

Тестування продуктивності за допомогою Jmeter показало високу здатність додатка обробляти великі обсяги одночасних запитів, що підтверджує його готовність до реальних умов використання.

Рекомендації для подальшого вдосконалення

Хоча додаток досягнув основних цілей, існують шляхи для його подальшого вдосконалення. Серед можливих напрямків покращення:

1. Використання мульти-регіональної архітектури: Розгортання системи в кількох AWS-регіонах підвищить її надійність та доступність, зменшуючи затримки для користувачів з різних регіонів і забезпечуючи стійкість до відмов на рівні регіонів.

2. Покращення безпеки за допомогою додаткових інструментів AWS: Використання таких сервісів, як AWS WAF (Web Application Firewall) для захисту API Gateway від небажаного трафіку та AWS Shield для захисту від DDoS-атак, підвищить рівень безпеки веб-додатку.

3. Впровадження більш детального моніторингу та аналізу логів: Додаткові можливості для моніторингу, такі як розширене налаштування метрик у CloudWatch та інтеграція з AWS X-Ray, дозволять отримати глибший аналіз продуктивності й виявити можливі затримки або проблеми на різних етапах обробки запитів.

4. Розширення функціоналу за допомогою нових сервісів AWS: У майбутньому можна розглянути можливість інтеграції з іншими сервісами AWS, такими як Step Functions для управління складними робочими процесами або Amazon SQS для покращення обробки асинхронних запитів.

5. Підвищення ефективності кешування: Оптимізація CloudFront та конфігурації S3 для більш ефективного кешування контенту може значно знизити затримки та покращити швидкість завантаження додатка, особливо для користувачів, розташованих далеко від основного регіону розгортання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Apache JMeter Documentation. (2024). The Definitive Guide to Load Testing. Apache Software Foundation. https://jmeter.apache.org.

2. Aditya Akella, Peter Bailis. Cloud Computing and Serverless Systems Research // ACM SIGCOMM Computer Communication Review. 2019.

3. Amazon Web Services (AWS). Документація [Електронний ресурс]. – Режим доступу: https://docs.aws.amazon.com.

4. Erez Tager. Serverless Architectures with AWS: Solutions for Real-World Scenarios. O'Reilly Media, 2020.

5. Gene Kim, Patrick Debois, John Willis, Jez Humble. The DevOps Handbook. IT Revolution, 2016.

6. Gojko Adzic, Robert Chatley. Serverless Design Patterns and Best Practices. Packt Publishing, 2019.

7. Google Cloud Functions Documentation [Електронний ресурс]. – Режим доступу: https://cloud.google.com/functions/docs.

8. ISO 45001:2018 Occupational health and safety management systems. International Organization for Standardization [Електронний ресурс]. – Режим доступу: https://www.iso.org/standard/63787.html.

9. Microsoft Azure Documentation. Azure Functions Overview [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/azure/azure-functions.

10. Mike Roberts, John Chapin. Serverless Architectures on AWS. Manning Publications, 2019.

11. Sbarski, P., & Kroonenburg, P. (2017). *Serverless Architectures on AWS: With Examples Using AWS Lambda*. Manning Publications. ISBN: 978-1617293827.

12. TechBeacon. (2024). Top Tools for Load Testing: A Deep Dive into Apache JMeter. TechBeacon. https://techbeacon.com/load-testing-tools.

13. Load Testing Academy. (2024). Understanding Load Testing Metrics: Using JMeter Effectively. https://loadtestingacademy.com.

14. Yan Cui. Production-Ready Serverless: Learn How to Build Reliable Serverless Applications. Manning Publications, 2020.

15. Zhu K., Fu J., Li Y. Research the Performance Testing and Performance Improvement Strategy in Web Application [Електронний ресурс] // 2nd International Conference on Education Technology and Computer. 2010. – Режим доступу: https://www.researchgate.net/publication/224161514_Research_the_performance_testing_and_performance_improvement_strategy_in_web_application.

16. Understanding Your Reports - Part 4: How to Read Your Load Testing Reports on BlazeMeter [Електронний ресурс] // BlazeMeter. 2016. – Режим доступу: https://www.blazemeter.com/blog/understanding-your-reports-part-4-how-read-your-load-testing-reports-blazemeter.

17. Власна публікація: AWS Lambda проти ECS проти EC2: Комплексний аналіз продуктивності та вартості. 2024. – [Електронний ресурс] // Режим доступу: https://medium.com/@bohdan-martyniv/aws-lambda-проти-ecs-проти-ec2-комплексний-аналіз-продуктивності-та-вартості-ba2d2018932f.

18. Гоголь, Є. В. Охорона праці в галузі ІТ. Методичні рекомендації. Київ: КПІ, 2022.

19. Джошуа Хілтон. Cloud Native Transformation: Practical Patterns for Innovation. O'Reilly Media, 2019. 328 с.

20. Джоель Спольски. Основи програмування на роботі. К.: Видавництво Сталкер, 2018. 320 с.

21. Державні санітарні норми і правила. Міністерство охорони здоров’я України [Електронний ресурс]. – Режим доступу: https://moz.gov.ua.

22. Дружиніна О.О., Квєтний Р. Н. Підвищення ефективності функціонування веб-серверів з використанням технології прогнозування часових рядів на основі нейромереж // Інформаційні технології та комп'ютерна інженерія. 2013. № 1. С. 15–21. – Режим доступу: http://nbuv.gov.ua/UJRN/Itki_2013_1_5.

23. Дан Шлеттер. Безсерверна архітектура: створення веб-додатків на AWS. СПб.: Пітер, 2019. 320 с.

24. Дональд Е. Кнут. Мистецтво програмування. М.: Біном, 2006. Т.1: Основні алгоритми. 640 с.

25. Коваль, О. В. Інформаційна безпека у роботі з даними. Київ: Фенікс, 2020.

26. Марк Массі. AWS Lambda: Розробка безсерверних додатків на прикладах. К.: Добра книга, 2020. 352 с.

27. Мартін Ореллі. Архітектура веб-додатків. К.: Діалектика, 2020. 400 с.

28. Мартін Фаулер. Архітектура корпоративних програмних додатків. К.: БІНОМ. Лабораторія знань, 2019. 576 с.

29. Полянський, В. М. Ергономіка робочого місця в умовах сучасних офісів. Львів: Світ, 2021.

30. Посібник з оцінки ризиків для робочого середовища. Європейське агентство з безпеки та здоров’я на роботі [Електронний ресурс]. – Доступно: https://osha.europa.eu.

ДОДАТКИ

ДОДАТОК А

Код головного SAM-шаблону.

(формат – YAML, оболонка – VS Code)

```
AWSTemplateFormatVersion: 2010-09-09
Description: >-
  sam-app
Transform:
- AWS::Serverless-2016-10-31

Resources:

  ApiGatewayApi:
        Type: AWS::Serverless::Api
        Properties:
        StageName: Prod
        Cors:
        AllowMethods: "'OPTIONS, POST, GET'"
        AllowHeaders: "'Content-Type'"
        AllowOrigin: "'*'"


  # This is a Lambda function config associated with the source code: get-by-id.js
  getAllItemsFunction:
        Type: AWS::Serverless::Function
        Properties:
        CodeUri: backend/
        Handler: src/handlers/get-all-items.getAllItemsHandler
        Runtime: nodejs20.x
        Architectures:
        - x86_64
        MemorySize: 128
        Timeout: 100
        Description: A simple example includes a HTTP get method to get all items by
        id from a DynamoDB table.
        Policies:
        # Give Create/Read/Update/Delete Permissions to the SampleTable
        - DynamoDBCrudPolicy:
```

```
        TableName: !Ref SampleTable
        Environment:
        Variables:
        # Make table name accessible as environment variable from function code during execution
        SAMPLE_TABLE: !Ref SampleTable
        # Make DynamoDB endpoint accessible as environment variable from function code during execution
        ENDPOINT_OVERRIDE: ''
        Events:
        Api:
        Type: Api
        Properties:
        Path: /
        Method: GET
        RestApiId:
        Ref: ApiGatewayApi

  # This is a Lambda function config associated with the source code: get-by-id.js
  getByIdFunction:
        Type: AWS::Serverless::Function
        Properties:
        CodeUri: backend/
        Handler: src/handlers/get-by-id.getByIdHandler
        Runtime: nodejs20.x
        Architectures:
        - x86_64
        MemorySize: 128
        Timeout: 100
        Description: A simple example includes a HTTP get method to get one item by
        id from a DynamoDB table.
        Policies:
        # Give Create/Read/Update/Delete Permissions to the SampleTable
        - DynamoDBCrudPolicy:
        TableName: !Ref SampleTable
        Environment:
        Variables:
        # Make table name accessible as environment variable from function code during execution
```

```
        SAMPLE_TABLE: !Ref SampleTable
        # Make DynamoDB endpoint accessible as environment variable from function
code during execution
        ENDPOINT_OVERRIDE: ''
        Events:
        Api:
        Type: Api
        Properties:
        Path: /{id}
        Method: GET
        RestApiId:
        Ref: ApiGatewayApi

  # This is a Lambda function config associated with the source code: put-item.js
  putItemFunction:
        Type: AWS::Serverless::Function
        Properties:
        CodeUri: backend/
        Handler: src/handlers/put-item.putItemHandler
        Runtime: nodejs20.x
        Architectures:
        - x86_64
        MemorySize: 128
        Timeout: 100
        Description: A simple example includes a HTTP post method to add one item
to
        a DynamoDB table.
        Policies:
        # Give Create/Read/Update/Delete Permissions to the SampleTable
        - DynamoDBCrudPolicy:
        TableName: !Ref SampleTable
        Environment:
        Variables:
        SAMPLE_TABLE: !Ref SampleTable
        # Make DynamoDB endpoint accessible as environment variable from function
code during execution
        ENDPOINT_OVERRIDE: ''
        Events:
        Api:
```

```
        Type: Api
        Properties:
        Path: /
        Method: POST
        RestApiId:
        Ref: ApiGatewayApi

# DynamoDB table to store item: {id: &lt;ID&gt;, name: &lt;NAME&gt;}
SampleTable:
        Type: AWS::Serverless::SimpleTable
        Properties:
        PrimaryKey:
        Name: id
        Type: String
        ProvisionedThroughput:
        ReadCapacityUnits: 2
        WriteCapacityUnits: 2

# S3 Bucket to host single page app website
S3Bucket:
        Type: AWS::S3::Bucket
        Properties:
        BucketName: !Sub "${AWS::StackName}-bucket-bohdan-marty"
        AccessControl: Private

# S3 Bucket Policy
S3BucketPolicy:
        Type: AWS::S3::BucketPolicy
        Properties:
        Bucket: !Ref S3Bucket
        PolicyDocument:
        Version: "2012-10-17"
        Statement:
        - Sid: AllowCloudFrontAccess
        Effect: Allow
        Principal:
        Service: cloudfront.amazonaws.com
        Action: "s3:GetObject"
        Resource: !Sub "${S3Bucket.Arn}/*"
```

```
        Condition:
        StringEquals:
            "AWS:SourceArn": !Sub
"arn:aws:cloudfront::${AWS::AccountId}:distribution/${CloudFrontDistribution}"

  # CloudFront Distribution for hosting the single page app website
  CloudFrontDistribution:
        Type: AWS::CloudFront::Distribution
        Properties:
        DistributionConfig:
        Enabled: true
        DefaultCacheBehavior:
        TargetOriginId: S3Origin
        ViewerProtocolPolicy: redirect-to-https
        AllowedMethods:
        - DELETE
        - GET
        - HEAD
        - OPTIONS
        - PATCH
        - POST
        - PUT
        CachedMethods:
        - GET
        - HEAD
        ForwardedValues:
        QueryString: false
        Compress: true
        Origins:
        - Id: S3Origin
        DomainName: !GetAtt S3Bucket.RegionalDomainName
        OriginAccessControlId: !Ref CloudFrontOriginAccessControl
        S3OriginConfig: # Add this to resolve the error
        OriginAccessIdentity: ""
        ViewerCertificate:
        CloudFrontDefaultCertificate: true
        DefaultRootObject: index.html

  CloudFrontOriginAccessControl:
```

```
        Type: AWS::CloudFront::OriginAccessControl
        Properties:
        OriginAccessControlConfig:
        Name: !Sub "${AWS::StackName}-OAC"
        OriginAccessControlOriginType: s3
        SigningBehavior: always
        SigningProtocol: sigv4
  ApplicationResourceGroup:
        Type: AWS::ResourceGroups::Group
        Properties:
        Name:
        Fn::Sub: ApplicationInsights-SAM-${AWS::StackName}
        ResourceQuery:
        Type: CLOUDFORMATION_STACK_1_0
  ApplicationInsightsMonitoring:
        Type: AWS::ApplicationInsights::Application
        Properties:
        ResourceGroupName:
        Ref: ApplicationResourceGroup
        AutoConfigurationEnabled: 'true'

Outputs:
  APIGatewayEndpoint:
        Description: API Gateway endpoint URL for Prod stage
        Value: !Sub "https://${ApiGatewayApi}.execute-
api.${AWS::Region}.amazonaws.com/Prod/"
  CloudFrontDistributionId:
        Description: CloudFront Distribution ID for hosting web front end
        Value: !Ref CloudFrontDistribution
  CloudFrontDistributionDomainName:
        Description: CloudFront Distribution Domain Name for accessing web front
end
        Value: !GetAtt CloudFrontDistribution.DomainName
  WebS3BucketName:
        Description: S3 Bucket for hosting web frontend
        Value: !Ref S3Bucket

Globals:
  Function:
```

```
        Tracing: Active
        # You can add LoggingConfig parameters such as the Logformat, Log Group, and SystemLogLevel or ApplicationLogLevel. Learn more here https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/sam-resource-function.html#sam-function-loggingconfig.
        LoggingConfig:
        LogFormat: JSON
  Api:
        TracingEnabled: true
```

ДОДАТОК Б

Код Lambda Get-All-Users

(програмне середовище – Java Script, оболонка – VS Code)

```
import { DynamoDBClient } from '@aws-sdk/client-dynamodb';
import { DynamoDBDocumentClient, ScanCommand } from '@aws-sdk/lib-dynamodb';

const ENDPOINT_OVERRIDE = process.env.ENDPOINT_OVERRIDE;
let ddbClient = undefined;

if (ENDPOINT_OVERRIDE) {
	ddbClient = new DynamoDBClient({ endpoint: ENDPOINT_OVERRIDE });
}
else {
	ddbClient = new DynamoDBClient({});
	console.warn("No value for ENDPOINT_OVERRIDE provided for DynamoDB, using default");
}

const ddbDocClient = DynamoDBDocumentClient.from(ddbClient);

const tableName = process.env.SAMPLE_TABLE;

export const getAllItemsHandler = async (event) => {
	if (event.httpMethod !== 'GET') {
	throw new Error(`getAllItems only accept GET method, you tried: ${event.httpMethod}`);
	}
	console.info('received:', event);

	var params = {
	TableName : tableName
	};

	try {
	const data = await ddbDocClient.send(new ScanCommand(params));
	var items = data.Items;
```

```
        } catch (err) {
        console.error("Error retrieving all items:", err.message);
        console.error("Error code:", err.code);
        console.error("Error name:", err.name);
        console.error("Error stack:", err.stack);

        throw err;
        }

        const response = {
        statusCode: 200,
        headers: {
        "Access-Control-Allow-Headers" : "Content-Type",
        "Access-Control-Allow-Origin": "*",
        "Access-Control-Allow-Methods": "OPTIONS,POST,GET"
        },
        body: JSON.stringify(items)
        };

        console.info(`response from: ${event.path} statusCode:
${response.statusCode} body: ${response.body}`);
        return response;
}
```

ДОДАТОК В

Код Lambda Get-User-By-Id

(програмне середовище – Java Script, оболонка – VS Code)

```
import { DynamoDBClient } from '@aws-sdk/client-dynamodb';
import { DynamoDBDocumentClient, GetCommand } from '@aws-sdk/lib-
dynamodb';

const ENDPOINT_OVERRIDE = process.env.ENDPOINT_OVERRIDE;
let ddbClient = undefined;

if (ENDPOINT_OVERRIDE) {
  ddbClient = new DynamoDBClient({ endpoint: ENDPOINT_OVERRIDE });
}
else{
  ddbClient = new DynamoDBClient({});
  console.warn("No value for ENDPOINT_OVERRIDE provided for DynamoDB,
using default");
}

const ddbDocClient = DynamoDBDocumentClient.from(ddbClient);

const tableName = process.env.SAMPLE_TABLE;

export const getByIdHandler = async (event) => {
  if (event.httpMethod !== 'GET') {
        throw new Error(`getMethod only accept GET method, you tried:
${event.httpMethod}`);
  }

  console.info('received:', event);

  const id = event.pathParameters.id;

  var params = {
        TableName : tableName,
        Key: { id: id },
  };
```

```
  try {
        const data = await ddbDocClient.send(new GetCommand(params));
        var item = data.Item;
  } catch (err) {
        console.error("Error retrieving item:", err.message);
        console.error("Error code:", err.code);
        console.error("Error name:", err.name);
        console.error("Error stack:", err.stack);

        throw err;
  }

  const response = {
        statusCode: 200,
        headers: {
        "Access-Control-Allow-Headers" : "Content-Type",
        "Access-Control-Allow-Origin": "*",
        "Access-Control-Allow-Methods": "OPTIONS,POST,GET"
  },
        body: JSON.stringify(item)
  };

  console.info(`response from: ${event.path} statusCode: ${response.statusCode}
body: ${response.body}`);
  return response;
}
```

ДОДАТОК Г

Код Lambda Create-User

(програмне середовище – Java Script, оболонка – VS Code)

```
import { DynamoDBClient } from '@aws-sdk/client-dynamodb';
import { DynamoDBDocumentClient, PutCommand } from '@aws-sdk/lib-
dynamodb';

const ENDPOINT_OVERRIDE = process.env.ENDPOINT_OVERRIDE;
let ddbClient = undefined;

if (ENDPOINT_OVERRIDE) {
  ddbClient = new DynamoDBClient({ endpoint: ENDPOINT_OVERRIDE });
}
else {
  ddbClient = new DynamoDBClient({});
  console.warn("No value for ENDPOINT_OVERRIDE provided for DynamoDB,
using default");
}

const ddbDocClient = DynamoDBDocumentClient.from(ddbClient);

const tableName = process.env.SAMPLE_TABLE;

export const putItemHandler = async (event) => {
        if (event.httpMethod !== 'POST') {
        throw new Error(`postMethod only accepts POST method, you tried:
${event.httpMethod} method.`);
        }
        console.info('received:', event);

        const body = JSON.parse(event.body);
        const id = body.id;
        const name = body.name;

        var params = {
        TableName : tableName,
        Item: { id : id, name: name }
```

```
        };

        try {
        const data = await ddbDocClient.send(new PutCommand(params));
        console.log("Success - item added or updated", data);
        } catch (err) {
        console.error("Error adding or updating item:", err.message);
        console.error("Error code:", err.code);
        console.error("Error name:", err.name);
        console.error("Error stack:", err.stack);
        throw err;
        }
        const response = {
        statusCode: 200,
        headers: {
        "Access-Control-Allow-Headers" : "Content-Type",
        "Access-Control-Allow-Origin": "*",
        "Access-Control-Allow-Methods": "OPTIONS,POST,GET"
        },
        body: JSON.stringify(body)
        };
        console.info(`response from: ${event.path} statusCode:
${response.statusCode} body: ${response.body}`);
        return response;
};
```