Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Дослідження моделі побудови мікросервісної архітектури за допомогою безсерверних обчислень
(тема)

Виконав:
студент 2 курсу, групи ІПЗм-22-5

Гавриш Д.Л.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова

Керівник д.т.н проф. Смеляков К. С.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ З.В.Дудар
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет	Комп'ютерних наук
Кафедра	Програмної інженерії
Рівень вищої освіти	другий (магістерський)
Спеціальність	121 – Інженерія програмного забезпечення
Тип програми	освітньо-наукова програма
Освітня програма	Інженерія програмного забезпечення

Курс 2 Группа ІПЗм-22-5 Семестр 2

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Студенту Гавришу Д. Л. групи ІПЗм-22-5

1. Тема роботи: Дослідження моделі побудови мікросервісної архітектури за допомогою безсерверних обчислень

Затверджена наказом по університету від 29.03. 2024 р. № 250 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 18.06.2024

3. Вихідні дані до роботи *Архітектура розроблюваної системи: мікросервісна; інфраструктура системи: хмарна; критерії оцінювання архітектури системи: швидкодія на великих об'ємах даних, масштабованість, стійкість до помилок. Перелік використовуваних програмних засобів: ОС Microsoft Windows 7 та вище, AWS Management Console, AWS Command Line Interface. Технічне забезпечення: IBM-сумісний ПК з МП Pentium II та вище.*

4. Перелік питань, що потрібно опрацювати в роботі: *теоретичні аспекти мікросервісної архітектури та безсерверних обчислень, побудова мікросервісної архітектури з використанням aws безсерверних обчислень, опис прийнятих проектних рішень, функціонально-вартісний аналіз програмного продукту.*

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Видача завдання	30.03.2024	*виконано*
2	Аналіз предметної галузі	01.04.2024	*виконано*
3	Постановка задачі	01.04.2024	*виконано*
4	Експериментальні дослідження	02.04 – 20.04.24	*виконано*
5	Аналіз результатів експериментальних досліджень та розробка рекомендацій	20.04 – 23.04.24	*виконано*
6	Написання та оформлення статті та тез доповіді	17.04 – 23.04.24	*виконано*
7	Підготовка пояснювальної записки	01.04 – 26.04.24	*виконано*
8	Підготовка презентації та доповіді	26.04 – 2.05.24	*виконано*
9	Нормоконтроль	03.05 – 31.05.24	*виконано*
10	Рецензування	01.06 – 6.06.24	*виконано*
11	Занесення диплома в електронний архів	07.06.2024	*виконано*
12	Попередній захист	07.06.2024	*виконано*
13	Допуск до захисту у зав. кафедри	10.06.2024	*виконано*

Дата видачі завдання «30» березня 2024 р.

Студент _____ Гавриш Д. Л.

(підпис)

Керівник роботи _____ д.т.н проф. Смеляков К. С.

(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить 85 ст., 16 рис., 6 табл., 24 джерел.

БЕЗСЕРВЕРНІ ОБЧИСЛЕННЯ, ЕФЕКТИВНЕ ВИКОРИСТАННЯ РЕСУРСІВ, МІКРОСЕРВІСНА АРХІТЕКТУРА, СТІЙКІСТЬ ТА НАДІЙНІСТЬ, ХМАРНІ ОБЧИСЛЕННЯ, AMAZON API GATEWAY, AWS LAMBDA, DYNAMODB.

Об'єктом дослідження роботи є побудова мікросервісної архітектури за допомогою безсерверних обчислень на платформі AWS.

В ході дослідження у роботі вивчатиметься та аналізуватиметься використання ключових сервісів Amazon Web Services (AWS) для розробки та розгортання мікросервісів. Зокрема, розглядатиметься використання AWS Lambda для реалізації безсерверних обчислень, Amazon API Gateway для створення та управління API, а також DynamoDB як безсерверну базу даних NoSQL.

В результаті проведеного дослідження буде отримано глибокий інсайт у побудову мікросервісної архітектури за допомогою безсерверних обчислень на платформі Amazon Web Services (AWS). Дослідження дозволило ретельно розглянути та визначити переваги та недоліки використання ключових сервісів, таких як AWS Lambda, Amazon API Gateway та DynamoDB, в контексті створення та управління мікросервісами.

AMAZON API GATEWAY, AWS LAMBDA, CLOUD COMPUTING, DYNAMODB, MICROSERVICE ARCHITECTURE, RESILIENCE AND RELIABILITY, RESOURCE EFFICIENCY, SERVERLESS COMPUTING.

The object of research of the course work is the construction of a microservice architecture using serverless computing on the AWS platform.

During the research, the coursework will study and analyze the use of key services of Amazon Web Services (AWS) for the development and deployment of

microservices. In particular, it will cover the use of AWS Lambda to implement serverless computing, Amazon API Gateway to create and manage APIs, and DynamoDB as a serverless NoSQL database.

As a result of the conducted research, a deep insight into the construction of a microservice architecture using serverless computing on the Amazon Web Services (AWS) platform will be obtained. The study allowed us to carefully consider and identify the advantages and disadvantages of using key services such as AWS Lambda, Amazon API Gateway, and DynamoDB in the context of creating and managing microservices.

Я, Гавриш Дмитро Леонідович, студент(ка) гр. ІПЗм-22-5, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження моделі побудови мікросервісної архітектури за допомогою безсерверних обчислень», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений(на) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

СУБД – Система Управління Базами Даних;

ЕОМ – Електронно-Обчислювальна Машина;

AMI – Amazon Machine Image;

API – Application Programming Interface;

AWS – Amazon Web Services;

CLI – Command Line Interface;

GCP – Google Cloud Platform;

HTTP – Hyper Text Transfer Protocol;

JSON – JavaScript Object Notation;

MVC – Model-View-Controller;

MS – Microsoft;

NoSQL – Not Only SQL;

REST – Representational State Transfer;

SDK – Software Development Kit;

S3 – Simple Storage Service;

SQL – Structured Query Language.

ВСТУП

У сучасному світі високих технологій парадигма розробки програмного забезпечення стрімко змінюється, визначаючи нові стандарти ефективності та гнучкості. Два основні напрямки цих змін – мікросервісна архітектура та безсерверні обчислення – представляють собою ключові концепції, що революціонізують спосіб створення та підтримки програмних систем.

Мікросервісна архітектура визначається як архітектурний підхід, де великі програмні системи розбиваються на невеликі незалежні сервіси, що взаємодіють між собою через легкий механізм комунікації. Цей підхід забезпечує гнучкість, масштабованість та легкість розгортання та розвитку програмних компонентів.

Безсерверні обчислення (Serverless), в свою чергу, не означають відсутність серверів, але скоріше вказують на те, що розробникам не потрібно вручну керувати інфраструктурою. Один із провідних постачальників безсерверних рішень – Amazon Web Services (AWS) – надає низку сервісів, таких як AWS Lambda, API Gateway та DynamoDB, для спрощення створення та управління безсерверними застосунками.

У даному дослідженні буде задача об'єднати ці дві концепції, досліджуючи можливості побудови мікросервісної архітектури з використанням безсерверних обчислень на платформі AWS. Докладно розглянето ключові аспекти та переваги обох підходів, а також проведено порівняльний аналіз вартості та продуктивності, спрямований на визначення оптимального шляху для побудови сучасних та ефективних інформаційних систем.

1 ТЕОРЕТИЧНІ АСПЕКТИ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ТА БЕЗСЕРВЕРНИХ ОБЧИСЛЕНЬ

1.1 Основні концепції мікросервісної архітектури

Архітектура мікросервісів означає структуру, яка відображає різні етапи або процеси роботи, зазвичай охоплюючи конкретну галузь бізнесу. Наприклад, уявімо програму для роздрібної торгівлі, яка включає кошик для покупок, управління замовленнями, обробку платежів, формування замовлень, каталоги продуктів, інтеграцію з внутрішніми фінансами/обліком та обслуговування клієнтів. Ці послуги функціонують незалежно одна від одної, представляючи відокремлені області знань, але вони повинні взаємодіяти для створення повнофункціональної програми мікросервісів.

Незалежні та автономні групи розробників програмного забезпечення керують, тестують і розгортають ці відокремлені служби, не впливаючи на решту системи в цілому. Результатом цього розподілу є структура, яка функціонує як більша архітектура. Якщо це ефективно впроваджено, це розгалуження робить операції більш гнучкими, дозволяючи командам створювати власні цикли випуску, прискорюючи виробництво та зменшуючи час виходу на ринок за рахунок спрощення операцій з многоразовими артефактами.

У мікросервісній архітектурі кожна окрема служба відповідає за конкретну бізнес-можливість або домен і може бути розроблена та розгорнута незалежно від інших служб. Цей підхід стоїть у контрасті до монолітної архітектури, де всі функціональні можливості програми упаковані в єдиний блок.

Кожна з цих служб спроектована навколо конкретних бізнес-вимог і може бути автономно розгорнута за допомогою повністю автоматизованого процесу розгортання. Важливою рисою є відсутність централізованого керування цими службами, які можуть бути написані на різних мовах програмування та використовувати різні технології для зберігання даних. Архітектура мікросервісів розбиває програму на невеликі, автономні служби, які ефективно взаємодіють через чітко визначені API.

Ці сервіси створені відповідно до бізнес-можливостей і можуть бути самостійно розгортані за допомогою повністю автоматизованого процесу розгортання. Ця архітектура характеризується мінімальним централізованим керуванням цими службами, які можуть бути написані на різних мовах програмування та використовувати різноманітні технології зберігання даних. Мікросервісна архітектура розбиває програму на компактні, автономні служби, що взаємодіють через чітко визначені API (див. рис. 1.1).

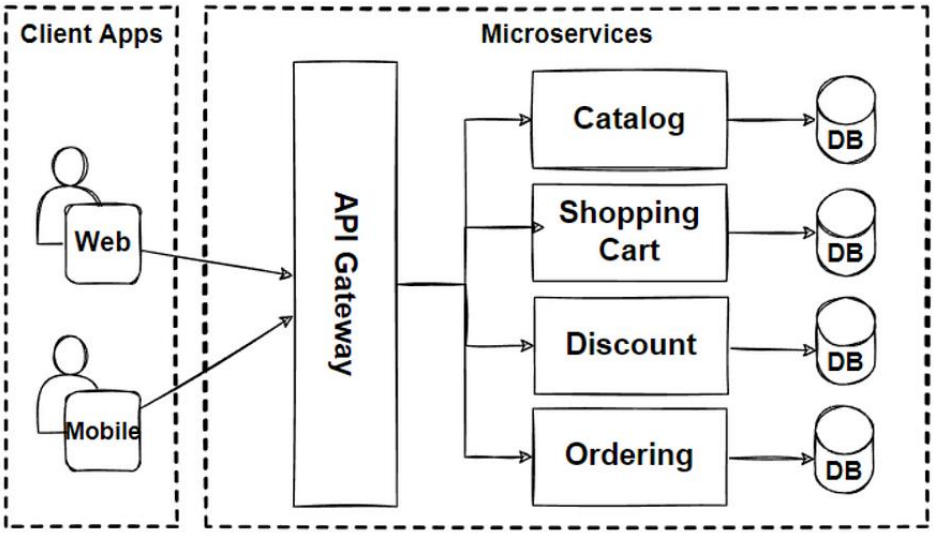

Рисунок. 1.1 – Архітектура мікросервісів (за даними[1])

Оскільки кожен сервіс може бути розроблений та підтриманий автономними командами, це представляє собою найбільш масштабований метод розробки програмного забезпечення. Кожна з цих служб відноситься до невеликих самостійних команд. Архітектура мікросервісів полегшує масштабування додатків та прискорює їх розробку, сприяючи інноваціям і швидкому введенню нових функцій на ринок.

У такий спосіб, можна сказати, що мікросервісна архітектура – це хмарний архітектурний підхід, де програми складаються з багатьох слабко пов'язаних менших компонентів, що можуть бути розгортані незалежно. Мікросервіси володіють власним стеком технологій, включаючи базу даних і модель управління даними, вони спілкуються один з одним через комбінацію REST API, потокової передачі подій та посередників повідомлень. Вони організовані за бізнес-

можливостями, а роздільні лінії між службами часто називають обмеженим контекстом.

Термін "принцип єдиної відповідальності", введений Робертом Мартіном, основною мірою стосується розподілу обов'язків між службами. Отже, при вирішенні питань мікросервісів можна керуватися принципами SRP.

Архітектура мікросервісів використовує той самий підхід та розширює його на слабко пов'язані служби, що можуть бути розроблятися, розгортатися та підтримуватися незалежно. Кожна з цих служб відповідає за різні завдання та може взаємодіяти з іншими службами через RESTful API для вирішення більш складних бізнес-проблем.

1.2 Постачальники безсерверних рішень

Хмарна модель безсерверних обчислень перш за все спрямована на швидкий запуск проектів та вигідну цінову політику для клієнтів. Проте вибір платформи має вирішальне значення для цієї моделі, тому розглянемо рішення провідних учасників ринку.

AWS Lambda – це безсерверний обчислювальний сервіс від Amazon, який запускає код у відповідь на події і автоматично керує необхідними обчислювальними ресурсами. Випущений у листопаді 2014 року, цей сервіс лідирує серед платформ з моделлю "Функція як послуга". Кожен екземпляр AWS Lambda – це контейнер, створений на AMI Amazon Linux, з оперативною пам'яттю від 128 до 3008 МБ, віртуальним сховищем розміром 512 МБ та часом виконання від 1 до 900 секунд.

Google Cloud Functions – це безсерверна платформа від Google, яка керується подіями. Вона підключає ваш код до Google Cloud Platform, налаштовуючи тригери, щоб реагувати на дії користувача та обробляти зміни програми. У поєднанні з Firebase, Cloud Functions дозволяють надсилати швидкі повідомлення про оновлення даних у додатку.

Azure Functions – це безсерверний обчислювальний сервіс від Microsoft, який, як і попередні платформи, реагує на внутрішні події на платформі Azure та

обробляє їх. Однією з особливостей цього сервісу є його тісна інтеграція з інструментами та сервісами Azure, пов'язаними з кібербезпекою, такими як захист мережі, захист даних, управління ризиками та ідентифікація.

1.3 Визначення та переваги безсерверних обчислень

Переваги архітектури мікросервісів включають наступне.

Масштабованість: Завдяки мікросервісам кожну службу можна масштабувати незалежно, що робить можливим легше впоратися зі збільшеним трафіком та навантаженням.

Гнучкість: Архітектура мікросервісів забезпечує швидші цикли розробки, оскільки кожен сервіс можна розробляти, тестувати та розгортати незалежно від інших.

Стійкість: Оскільки кожна служба є незалежною, збої в одній службі не обов'язково впливають на решту програми, що підвищує стійкість системи.

Гнучкість: Мікросервіси забезпечують більшу гнучкість у виборі технологій, оскільки кожна служба може бути розроблена з використанням технології, яка найбільше відповідає її конкретним вимогам.

Проста інтеграція зі сторонніми службами. Мікросервіси можна легко інтегрувати зі сторонніми службами, оскільки кожна служба має власний API, що спрощує процес взаємодії з іншими системами.

Недоліки архітектури мікросервісів включають нижче.

Підвищена складність: архітектура мікросервісів є більш складною порівняно з традиційною монолітною архітектурою, оскільки вона включає кілька служб, які взаємодіють між собою. Це може ускладнити розробку, обслуговування та вимагати додаткових ресурсів та досвіду.

Проблеми з розподіленою системою: розподілені служби архітектури мікросервісів можуть призводити до проблем, пов'язаних з узгодженістю даних, затримками мережі та виявленням служб. Це може виникнути через розподілений характер системи та залежність від мережевого взаємодії.

Проблеми з тестуванням і налагодженням: тестування та налагодження можуть бути складнішими в архітектурі мікросервісів через необхідність тестування та налагодження кількох незалежних служб. Інтеграційне тестування між службами також може виявитися викликом.

Накладні витрати на керування: архітектура мікросервісів вносить додаткові накладні витрати на керування кількома службами. Це включає в себе завдання, такі як розгортання, масштабування та моніторинг кожної окремої служби, що може призвести до збільшення адміністративних та інфраструктурних витрат.

Архітектура мікросервісів становить вдалий вибір для складних, великомасштабних програм, що потребують високого рівня масштабованості, доступності та гнучкості. Вона також може ідеально підійти для організацій, які мають потребу в інтеграції з численними сторонніми службами чи системами.

Однак слід відзначити, що архітектура мікросервісів не є універсальним рішенням і може не бути оптимальним вибором для всіх програм. Вона вимагає додаткових зусиль у справах проектування, впровадження та підтримки служб, а також ефективного управління їх взаємозв'язком. Крім того, накладні витрати на координацію між службами можуть викликати збільшення затримок та зниження продуктивності, тому в конкретних випадках це може бути менш оптимальним вибором для програм, які вимагають високої продуктивності або низької затримки.

1.4 Огляд ключових сервісів AWS для безсерверних рішень

Амазонська веб-служба (AWS) пропонує розширений спектр сервісів для реалізації безсерверних рішень, що дозволяє розробникам будувати ефективні та високопродуктивні додатки без необхідності керування інфраструктурою.

Обчислення:

– AWS Lambda – це оплачуваний у міру використання обчислювальний сервіс на основі подій, за допомогою якого можна виконувати код, не виділяючи сервери та не керуючи ними;

– AWS Fargate – це ядро для безсерверних обчислень, яке працює з Amazon Elastic Container Service (ECS) та Amazon Elastic Kubernetes Service (EKS).

Інтеграція додатків:

– Amazon EventBridge – це безсерверна шина подій, яка дозволяє створювати програми на основі подій в AWS та існуючих системах у будь-якому масштабі;
– AWS Step Functions – це засіб оркестрації візуальних робочих процесів, що полегшує створення послідовностей безлічі сервісів AWS для додатків, які є критично важливими для діяльності компанії;
– Amazon Simple Queue Service (SQS) – це сервіс черг повідомлень, за допомогою якого можна ізолювати та масштабувати мікросервіси, розподілені системи та безсерверні програми;
– Amazon Simple Notification Service (SNS) – це повністю керований сервіс обміну повідомленнями для зв'язку між програмами (A2A), а також між програмами та користувачами (A2P) ;
– Amazon API Gateway – це повністю керований сервіс, за допомогою якого простіше створювати та публікувати API у будь-яких масштабах;
– AWS AppSync – це повністю керований сервіс, який прискорює розробку програм за допомогою масштабованих API GraphQL.

Сховище даних:

– Amazon S3 – це сервіс об'єктного сховища, призначений для збереження та захисту будь-якого обсягу даних;
– Amazon EFS – це безсерверна, повністю еластична файлова система для спрощення налаштування, масштабування високодоступного сховища із загальним доступом та оптимізації витрат на нього;
– Amazon DynamoDB – це сервіс бази даних пар «ключ-значення» та документів, що забезпечує затримку менше 10 мілісекунд при роботі в будь-якому масштабі;

- Amazon RDS Proxy – це керований проксі-сервер бази даних для Amazon Relational Database Service (RDS), який забезпечує велику масштабованість та безпеку додатків;
- Amazon Aurora Serverless – це сумісна з MySQL та PostgreSQL реляційна база даних, яка автоматично масштабує ресурси залежно від потреб програми;
- Amazon Redshift безсерверний дозволяє запускати і масштабувати аналітичні робочі навантаження без налаштування інфраструктури сховища даних при оплаті ресурсів, що тільки використовуються;
- Amazon Neptune безсерверний – це масштабована графова база даних, що надається на вимогу, яка автоматично виділяє клієнтам ресурси залежно від потреб;
- безсерверний Amazon OpenSearch – це безсерверна опція у сервісі Amazon OpenSearch, що дозволяє надавати аналітику пошуку та журналів без виділення та налаштування ресурсів;
- безсерверний Amazon ElastiCache дозволяє легко створити високодоступний кеш менш ніж за хвилину, масштабується відповідно до вимог додатків та забезпечує надшвидку продуктивність;
- AWS Glue – це повністю керований сервіс інтеграції даних, який дозволяє легко підготувати та завантажити дані для аналітики. AWS Glue пропонує інтерфейс без коду для перетворення даних та обробки ETL-завдань;
- Amazon Managed Streaming for Apache Kafka (MSK) – це повністю керований сервіс, який полегшує створення та виконання додатків реального часу з використанням Apache Kafka. Він дозволяє зосередитися на логіці обробки даних замість управління інфраструктурою Kafka.

Приклад використання таких сервісів можна переглянути на рисунку 1.2.

Інтернет-додаток на основі подій може використовувати AWS Lambda та Amazon API Gateway для бізнес-логіки, Amazon DynamoDB як базу даних, а консоль AWS Amplify для розміщення статичного контенту.

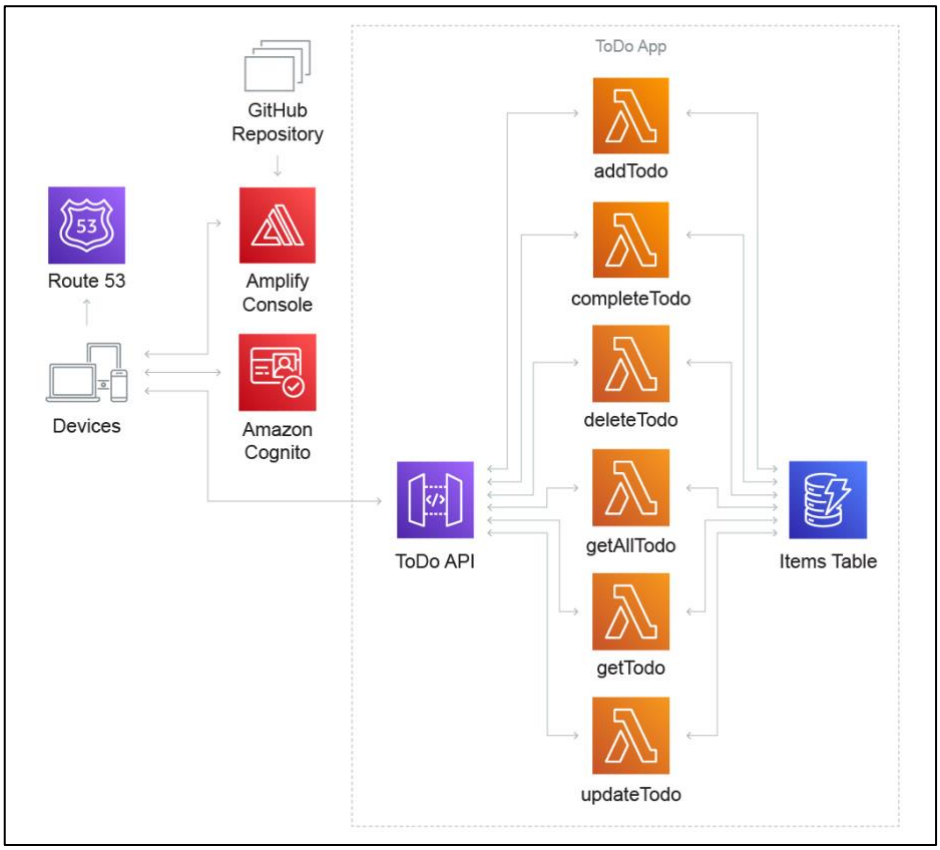

Рисунок 1.2 – Приклад створення інтернет-додатків (за даними [2])

Ці ключові сервіси створюють інфраструктурний фундамент для ефективної реалізації безсерверних архітектур в середовищі AWS.

1.5 Постановка задачі

У даній роботі буде досліджено теоретичні аспекти мікросервісної архітектури та безсерверних обчислень, а також їх практичне застосування з використанням сервісів AWS. Це дослідження має на меті вивчити сучасні підходи до побудови програмних систем, які дозволяють підвищити гнучкість, масштабованість та ефективність управління ресурсами.

Завдання дослідження:

а) теоретичні аспекти мікросервісної архітектури та безсерверних обчислень:

– розглянути основні концепції мікросервісної архітектури;

– проаналізувати постачальників безсерверних рішень;

– визначити переваги безсерверних обчислень;

– провести огляд ключових сервісів aws для безсерверних рішень.

б) побудова мікросервісної архітектури з використанням AWS безсерверних обчислень:

– розробити проект мікросервісної архітектури;

– реалізувати мікросервіси за допомогою AWS Lambda;

– налаштувати та управляти API за допомогою API Gateway;

– використовувати DynamoDB для зберігання даних мікросервісів.

в) опис прийнятих проектних рішень:

– обґрунтувати вибір мікросервісної архітектури;

– проаналізувати вибір хмарного провайдера та стеку сервісів;

– описати обґрунтування вибору постачальника хмарних обчислень;

– розглянути вибір необхідного стеку хмарних сервісів;

– дослідити застосування Amazon Step Functions.

г) функціонально-вартісний аналіз програмного продукту:

– провести аналіз витрат і вигод створеного програмного продукту на основі мікросервісної архітектури та безсерверних обчислень з використанням сервісів AWS.

Результатом даного дослідження має стати розробка ефективної мікросервісної архітектури з використанням безсерверних обчислень, що забезпечить підвищену масштабованість, гнучкість та оптимізацію витрат. У роботі буде представлено обґрунтовані проектні рішення, описано їх реалізацію та проведено функціонально-вартісний аналіз.

2 ПОБУДОВА МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ З ВИКОРИСТАННЯМ AWS БЕЗСЕРВЕРНИХ ОБЧИСЛЕНЬ

2.1 Проектування мікросервісної архітектури

Проектування мікросервісної архітектури – це ключовий етап у розробці додатків, які базуються на принципах розподіленої системи та самостійно працюючих компонентів. Детальне планування і структурування мікросервісів визначає ефективність, гнучкість та стійкість системи. Основні кроки при проектуванні мікросервісної архітектури включають:

- аналіз бізнес-потреб: визначення бізнес-потреб і функціональних вимог є важливим етапом, дає зрозуміти області бізнесу та їх взаємозв'язки допомагає належно ідентифікувати мікросервіси, які краще підходять для розв'язання конкретних завдань;
- граничні контексти: визначення граничних контекстів для кожного мікросервісу – ключовий аспект, який дозволяє ясно визначити обов'язки та функціональні можливості кожного сервісу, розділяючи їх від інших;
- збереження даних: вибір правильного механізму зберігання даних для кожного мікросервісу є критичним, наприклад, використання бази даних типу NoSQL, такої як Amazon DynamoDB, для сервісу, який потребує гнучкого та масштабованого зберігання;
- API та інтерфейси: ясно визначити API та інтерфейси для взаємодії між сервісами, яка включає в себе формати обміну даними, протоколи та стандарти, що дозволяють ефективну комунікацію між мікросервісами;
- масштабованість та витривалість: забезпечення можливості масштабованості та витривалості для кожного мікросервісу, яка включає в себе обробку навантаження, балансування навантаження та забезпечення доступності.
- моніторинг та логування: впровадження ефективної системи моніторингу та логування для всіх мікросервісів, яка сприяє вчасному виявленню проблем та усуненню неполадок в системі;

– безпека: забезпечення адекватного рівня безпеки для кожного сервісу, включаючи валідацію вхідних даних, захист від атак та управління доступом;
– документація: створення чіткої та повної документації для кожного мікросервісу, яка описує його функціональність, взаємодію та вимоги.

Ефективне проектування мікросервісної архітектури дозволяє створити систему, яка легко масштабується, гнучка та відповідає конкретним бізнес-вимогам.

2.2 Реалізація мікросервісів за допомогою AWS Lambda

Архітектурний шаблон для мікросервісів з використанням AWS Lambda.

Шаблон архітектури мікросервісу не обмежується типовою трирівневою архітектурою; проте цей шаблон може взяти значні вигоди від використання безсерверних ресурсів (див. рис. 2.1).

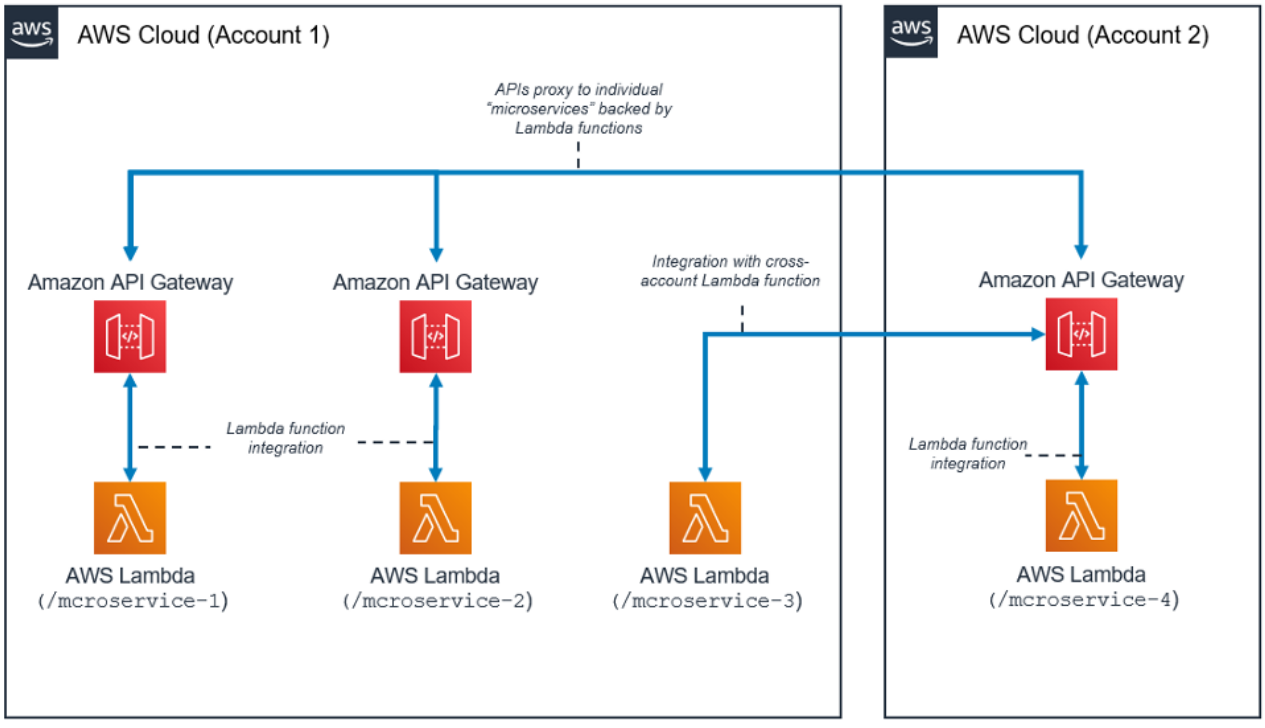

Рисунок 2.1 – Архітектурний шаблон для мікросервіса з Lambda (за даними [3])

У цій архітектурі кожен компонент програми розглядається як відокремлений і незалежно розгорнутий та працюючий. API, створений за

допомогою Amazon API Gateway, і функції, які потім запускаються за допомогою AWS Lambda, – це все, що потрібно для створення мікросервісу.

Команда може використовувати ці служби, щоб розділити та фрагментувати середовище на бажаний рівень деталізації.

Загалом в середовищі мікросервісів можуть виникнути такі труднощі: повторювані накладні витрати на створення кожного нового мікросервісу, проблеми з оптимізацією щільності та використання серверів, складність одночасного запуску кількох версій кількох мікросервісів і збільшення вимог до клієнтського коду для інтеграції з багатьма окремими послугами.

При створенні мікросервісів за допомогою безсерверних ресурсів ці проблеми стає легше вирішувати, а в деяких випадках навіть просто зникають. Шаблон безсерверних мікросервісів знижує бар'єр для створення кожного наступного мікросервісу (API Gateway навіть дозволяє клонувати існуючі API та використовувати функції Lambda в інших облікових записах). Оптимізація використання сервера більше не є актуальною з цим шаблоном. Нарешті, Amazon API Gateway надає програмно згенеровані клієнтські SDK для декількох популярних мов, щоб зменшити витрати на інтеграцію.

Огляд ключових етапів реалізації мікросервісів за допомогою AWS Lambda:

- створення функції AWS Lambda: спочатку необхідно створити функцію AWS Lambda для кожного мікросервісу, це можна зробити через консоль AWS або за допомогою інструментів розробки, таких як AWS CLI або AWS SDK.
- визначення тригерів: AWS Lambda може бути спрацьований різними тригерами, наприклад, виклик функції може бути ініційований зміною в об'єкті Amazon S3, новим записом у DynamoDB, або через API Gateway. Визначення тригерів залежить від конкретного використання мікросервісу.
- конфігурація та налаштування: для кожної функції можна налаштовувати параметри, такі як обсяг виконань (виділені ресурси), доступ до інших

служб AWS, обробка помилок та інші налаштування згідно з вимогами мікросервісу.

- інтеграція з іншими службами AWS: AWS Lambda може легко інтегруватися з іншими службами AWS, такими як Amazon DynamoDB для зберігання даних, Amazon S3 для зберігання об'єктів, чи Amazon API Gateway для виклику мікросервісу через API.
- моніторинг та логування: налаштуйте систему моніторингу та логування для функцій AWS Lambda. AWS CloudWatch може використовуватися для збору та відстеження метрик виконання функцій.
- безпека та управління доступом: забезпечте безпеку мікросервісів через правильне налаштування управління доступом та використання AWS Identity and Access Management (IAM), та визначте права доступу для функцій та інших AWS ресурсів.
- тестування та відлагодження: проведіть тестування функцій, які реалізують мікросервіси, використовуючи різні сценарії та вхідні дані, AWS Lambda дозволяє легко відлагоджувати функції за допомогою інструментів відлагодження.
- масштабування та оптимізація: оцініть потреби у масштабуванні для кожного мікросервісу та налаштуйте параметри відповідно, та використовути можливості автоматичного масштабування AWS Lambda для ефективного використання ресурсів.

Реалізація мікросервісів за допомогою AWS Lambda дозволяє побудувати гнучкі, масштабовані та витривалі системи без необхідності управління інфраструктурою.

2.3 Конфігурація та управління API за допомогою API Gateway

У сучасному цифровому оточенні інтерфейси програмування застосунків (API) визначають основні елементи для створення та інтеграції новітніх програм. Шлюз API виступає як платформа, яка забезпечує єдиний інтерфейс для зовнішніх користувачів для звертання до різних API. Один з найвідоміших API-

шлюзів – Amazon API Gateway, це повністю управляємий сервіс, який дає розробникам можливість легко створювати, публікувати та управляти безпечними API будь-якого розміру.

У зв'язку з ростом електронної комерції, мобільних додатків та інших цифрових сервісів зростає потреба в API експоненційно. API дозволяють компаніям безперешкодно надавати свої послуги іншим підприємствам, розробникам і кінцевим користувачам. Тим не менше, управління API може бути не простим завданням через проблеми, такі як масштабованість, безпека та інтеграція.

Amazon API Gateway розв'язує ці проблеми, надаючи повністю управляючий, масштабований і безпечний шлюз, що спрощує управління API. Завдяки Amazon API Gateway розробники можуть зосередитися на створенні та наданні своїх API, в той час як Amazon відповідає за інфраструктуру та масштабування, забезпечуючи стабільну доступність API (див. рис. 2.2).

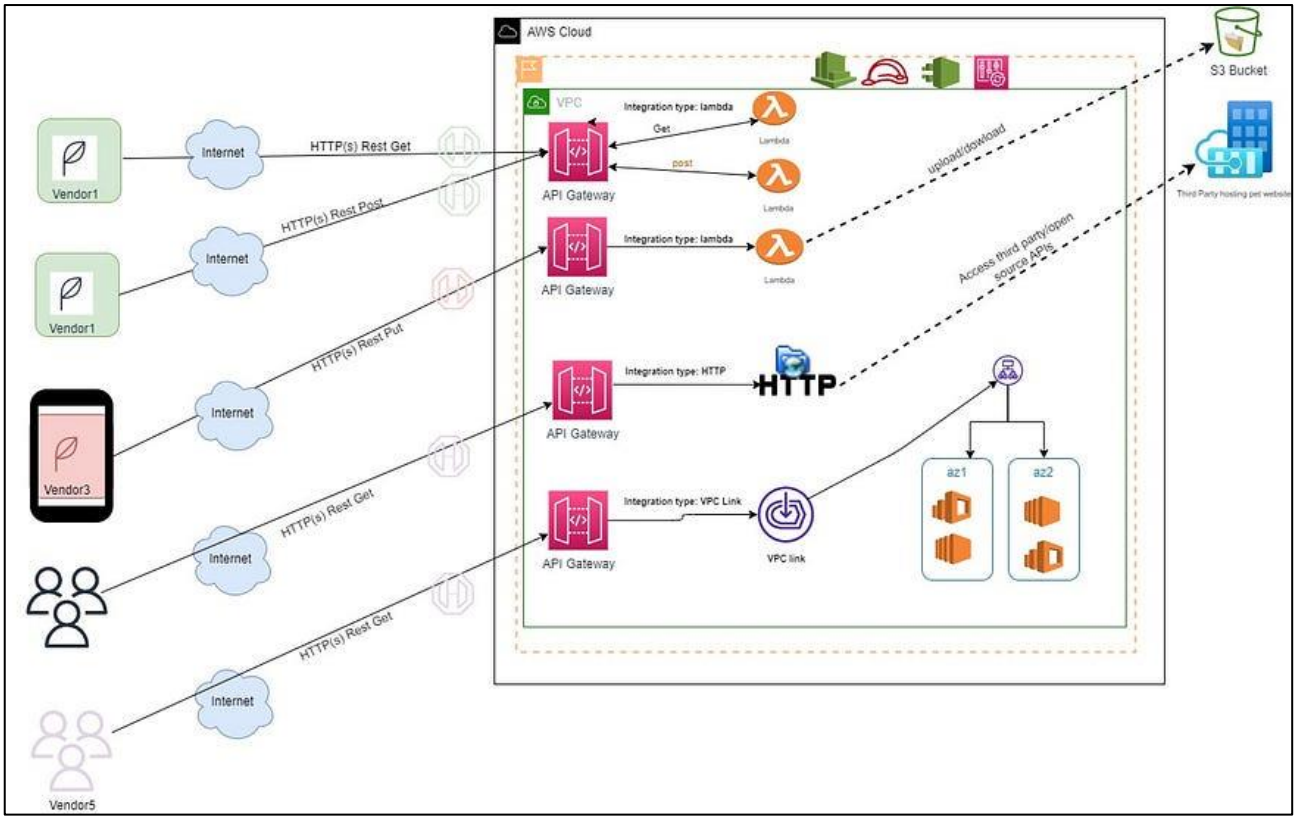

Рисунок 2.2 – Amazon API Gateway – REST API працює із запитами інтеграції (за даними [4])

Amazon API Gateway надає можливість створення та налаштування API, які відображають функціональність окремих мікросервісів. Встановлення параметрів маршрутизації, обробка запитів та визначення вихідних форматів даних – це лише кілька аспектів, які можна налаштовувати відповідно до вимог конкретного мікросервісу.

Ефективне управління API включає в себе кілька важливих аспектів. Відстеження та аналіз вхідних запитів, моніторинг пропускної здатності та забезпечення безпеки API – це ключові завдання, які можна вирішити за допомогою Amazon API Gateway.

Додатково, інструментарій API Gateway дозволяє здійснювати контроль доступу до ресурсів, застосовувати обмеження на рівні API ключів та використовувати політики автентифікації для забезпечення безпеки API.

Підсумовуючи, налаштування та управління API Gateway стає ключовим компонентом в реалізації мікросервісної архітектури, забезпечуючи гнучкість та безпеку взаємодії між мікросервісами.

2.4 Використання DynamoDB для зберігання даних мікросервісів

DynamoDB представляє собою безсерверну базу даних NoSQL від AWS. Безсерверність сприяє використанню DynamoDB для безсерверних мікросервісів, оскільки вона відповідає шаблонам і практикам розробки безсерверної архітектури в AWS (див. рис. 2.3).

DynamoDB також пропонує різноманітні інструменти для управління даними, такі як DynamoDB Streams для обробки змін у реальному часі, а також можливість інтеграції з іншими сервісами AWS через AWS Lambda, що дозволяє створювати потужні безсерверні додатки. Завдяки цим можливостям розробники можуть швидко і ефективно реагувати на події, що відбуваються у базі даних, забезпечуючи високу адаптивність і продуктивність додатків. Крім того, DynamoDB підтримує транзакції, що дозволяє виконувати кілька операцій як єдине ціле, забезпечуючи атомарність, узгодженість, ізольованість і надійність

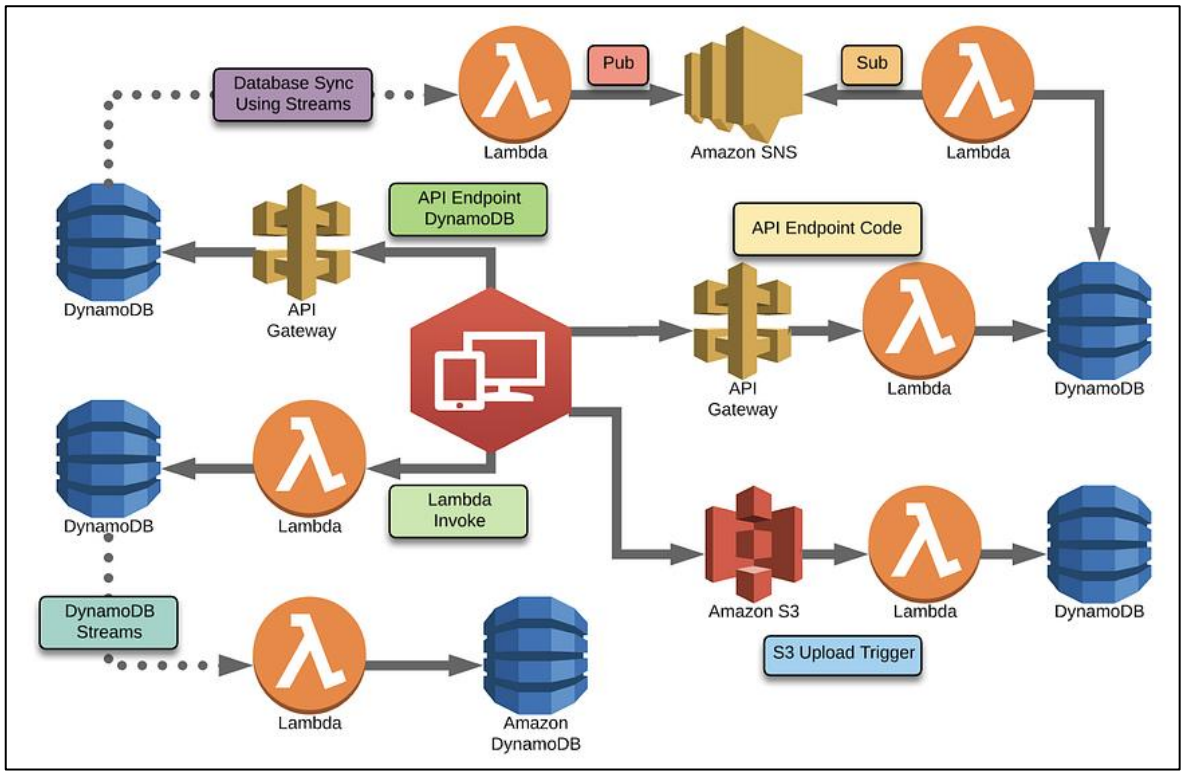

Рисунок 2.3 – Шаблони конфігурації AWS DynamoDB (за даними [5])

Щодо DynamoDB, AWS взяла на себе управління основною інфраструктурою та програмним забезпеченням, надаючи абстрактне представлення про таблиці, індекси (GSI, LSI), пропускну здатність, автоматичне масштабування та політики безпеки. Ці високорівневі конструкції бази даних NoSQL стають доступними завдяки послугам AWS.

AWS DynamoDB оптимально використовується для зберігання документів у форматі JSON та як сховище пар ключ-значення. З різними типами індексів та можливостей запитів, він стає зручним для різноманітних завдань зберігання та пошуку.

Важливо враховувати, що DynamoDB – це база даних NoSQL, і порівняння її з реляційною базою даних виявляється важким завданням. Це може бути особливо важливо для розробників, звикших до роботи з реляційними базами даних. Тому розуміння кількох основних принципів використання DynamoDB стає критичним. Ось 12 принципів, які слід дотримуватися при створенні таблиць і запитів DynamoDB:

- використовуйте guid або унікальні атрибути замість додаткових ідентифікаторів;
- уникайте нормалізації таблиць;
- допускайте наявність повторюваних атрибутів у кількох таблицях і реалізуйте способи синхронізації змін;
- зберігайте попередньо обчислені дані після оновлення для ефективних запитів;
- обмежте кількість зв'язків між таблицями, оскільки для отримання атрибутів доведеться запитувати кілька таблиць;
- використовуйте остаточну послідовність;
- створюйте власні транзакції для роботи з умовними записами;
- проектуйте таблиці, атрибути та індекси, враховуючи природу запитів;
- використовуйте тригери та потоки dynamodb для розповсюдження змін та потоків проектних даних через події;
- розглядайте розмір елементів і ефективне використання індексів під час переліку елементів для мінімізації вимог до пропускної здатності;
- враховуйте можливість зростання розміру атрибутів для визначення, чи слід зберігати їх як вкладений об'єкт, чи використовувати окрему таблицю;
- якщо можливо, уникайте використання операції сканування dynamodb.

Amazon DynamoDB володіє простотою налаштування та гнучкістю у роботі з різними видами даних. Він дозволяє легко створювати таблиці, розподіляти та масштабувати їх, що робить його ідеальним вибором для зберігання даних, які використовуються в мікросервісах [6].

DynamoDB пропонує автоматичне масштабування, що означає, що обсяг даних може зростати або зменшуватися без втрати продуктивності. Це ідеально підходить для мікросервісів, які можуть динамічно змінювати свої потреби в обсязі даних.

DynamoDB легко інтегрується з іншими сервісами AWS, такими як Lambda і API Gateway, створюючи потужну екосистему для розробки мікросервісів. Це дозволяє ефективно керувати взаємодією між різними компонентами системи.

DynamoDB надає засоби моніторингу та автоматичного шкалювання, спрощуючи управління базою даних. Це особливо важливо для забезпечення безперебійної роботи мікросервісів в умовах збільшення навантаження або змін у вимогах до даних.

Загалом використання Amazon DynamoDB для зберігання даних мікросервісів дозволяє ефективно керувати інформацією, забезпечуючи масштабованість та високу доступність для надійної роботи системи.

3 ОПИС ПРИЙНЯТИХ ПРОЕКТНИХ РІШЕНЬ

3.1 Вибір мікросервісної архітектури: важливість та обґрунтування

Запропоновано розглянути, чому вибір мікросервісної архітектури є доцільним для підсистеми пошуку. Незважаючи на широке поширення монолітної архітектури, яка є найбільш традиційним підходом у розробці програмного забезпечення на даний момент, вона також має свої недоліки. Наприклад, одна помилка може вплинути на всю систему, і для внесення змін розробники повинні розгортати нову версію додатка на стороні сервера. Це може створювати проблеми, які характеризують традиційний підхід. Монолітні програми можуть перерости у "велику кулю грязі", коли ні один розробник або група розробників не має повного розуміння всієї програми. У монолітних додатках обмежена можливість повторного використання модулів через сильний зв'язок між ними; їх масштабування часто стає проблемою, а оптимальна швидкість при багаторазовому розгортанні монолітних артефактів також важко досягти (див. рис. 3.1).

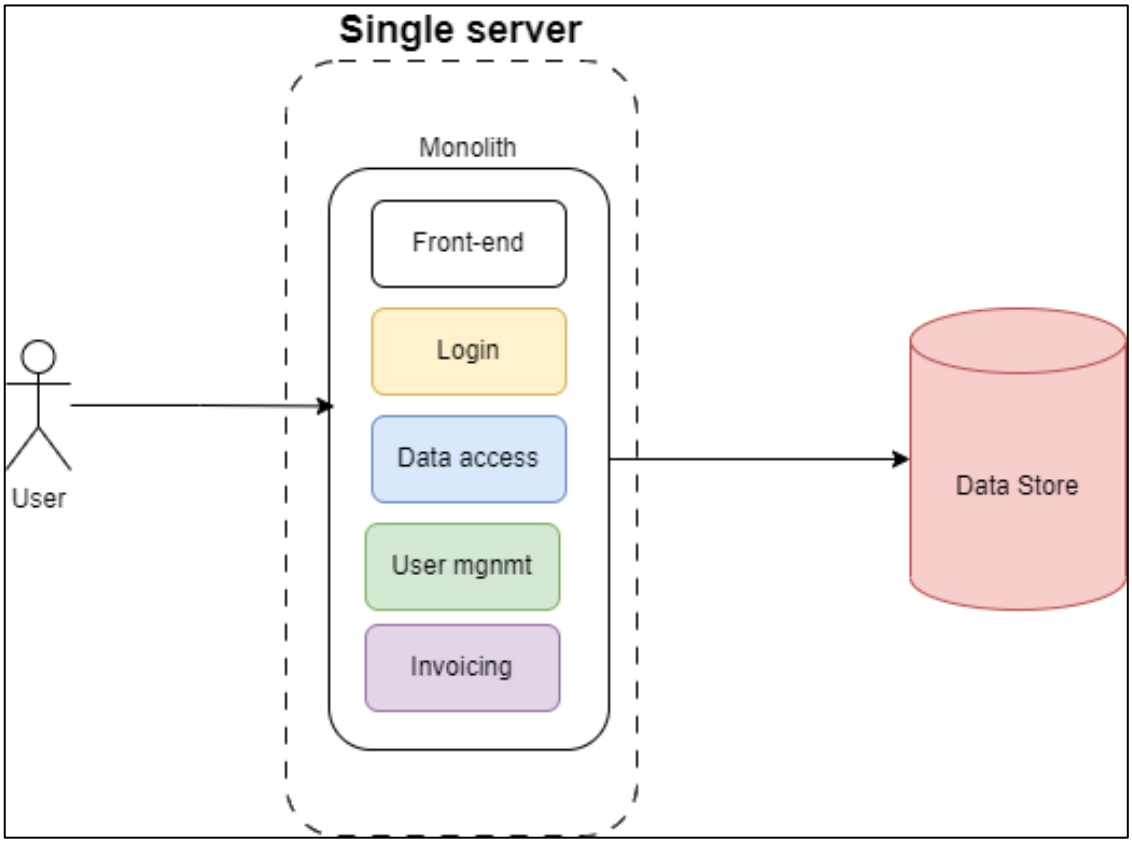

Рисунок 3.1 – Система монолітної архітектури (за даними [7])

Цей малюнок ілюструє, що у монолітній системі зазвичай логіка додатку розподілена між модулями, які взаємодіють між собою, а також з базою даних. В цій ситуації можуть виникнути наступні проблеми при розробці системи пошуку.

Неможливість незалежної реалізації: через те, що сервіс є лише модулем усієї системи, його не можна розробляти окремо від неї. Потрібно мати доступ до коду всього продукту для розробки певної частини, і з архітектурної точки зору, потрібно враховувати всі аспекти вже побудованої системи та обмеження (мову програмування, технології і т. д.).

Відсутність версіонування змін конкретної підсистеми пошуку: кожен раз, коли потрібно буде змінити пошуковий алгоритм або щось інше, потрібно оновлювати всю систему.

Неефективне використання бази даних: без системи кешування даних пошук може працювати повільно, а використання сторонніх рішень може бути ускладненим через обмеження технологій.

Неефективна робота самого пошуку: мова програмування, в якій написана монолітна система, може уповільнювати пошук.

Бізнес-переваги мікрослужб порівняно з монолітною архітектурою є значними. При правильному розгортанні архітектури на основі мікросервісів можна досягти значного підвищення ефективності та уникнути технічного боргу.

Технічне обслуговування: ця перевага, безумовно, іде разом з попередніми двома. Архітектура мікросервісів спрощує обслуговування та управління послугами. Насправді, кожну службу може керувати інша команда, оскільки це унікальний процес з власною базою даних. Крім того, можна виконувати окремі оновлення, не зупиняючи всю систему програмного забезпечення [8].

Адаптованість: мікросервіси не мають жорсткої взаємозалежності та мають стандартизований спосіб взаємодії. Оскільки вони побудовані окремо, вони можуть використовувати різні мови або технології програмування. Це означає, що кожна послуга може бути розроблена таким чином, щоб максимально відповідати її функціональності.

Еволюційність: потреби бізнесу можуть змінюватися, що вимагає додавання нових функцій до існуючих послуг. У традиційних архітектурних підходах це може призвести до втручання у всю бізнес-логіку та порушення існуючих функцій. Архітектура мікросервісів дозволяє уникнути цієї проблеми завдяки своєму модульному підходу.

Масштабованість: однією з головних переваг мікросервісів є їх можливість масштабуватися горизонтально. Це означає, що будь-яку розгорнуту службу можна легко дублювати для уникнення вузьких місць через повільне виконання. Іншим методом підвищення продуктивності є запуск служби на більш потужному обладнанні або на кількох машинах, які обробляють дані паралельно.

Тестування та моніторинг: окремі програмні компоненти набагато легше тестувати, ніж складні програми. Якщо кожну мікрослужбу перевірено належним чином, програмне забезпечення в цілому стає більш надійним та стабільним. Можливість ізолювати лише частини програми дозволяє поліпшити моніторинг та управління безпекою.

Постачання: завдяки автоматизації інфраструктури та безперервній доставці, операційна складність створення, впровадження та тестування мікросервісів значно зменшилась.

Інтеграція: програмне забезпечення повинно мати можливість інтегрувати стільки нових функцій та послуг, скільки потрібно. Архітектура мікросервісів значно спрощує цей процес завдяки використанню інструментів безперервної інтеграції.

Багаторазове використання: великою перевагою мікросервісів є можливість повторного використання незалежних компонентів для майбутніх проектів.

Дохід: швидші ітерації та зменшення простоїв можуть призвести до збільшення доходу, що сприяє залученню користувачів та поліпшенню залучення.

У рамках даного дослідження ми маємо розробити архітектуру окремого сервісу, який буде інтегрований у систему електронної бібліотеки. Нам важливо, щоб цей сервіс працював незалежно від бібліотеки, оскільки помилки у пошуковому алгоритмі не повинні впливати на роботу самої бібліотеки. Це

означає, що ми мусимо використовувати концепцію мікросервісів. Також важливо, щоб алгоритм не зазнавав збоїв через помилки в фільтрах або перенавантаження сервера, де він розташований, оскільки робота цього сервісу є його основною функцією для системи бібліотеки.

Мікросервісна архітектура має безліч переваг, які привертають увагу розробників по всьому світу. Однією з основних переваг цієї архітектури є її масштабованість і гнучкість. Кожен мікросервіс можна масштабувати незалежно від інших, враховуючи його власні потреби. Така деталізована масштабованість дозволяє ефективно розподіляти ресурси та забезпечує оптимальну продуктивність, особливо при змінних вимогах до робочого навантаження (див. рис. 3.2).

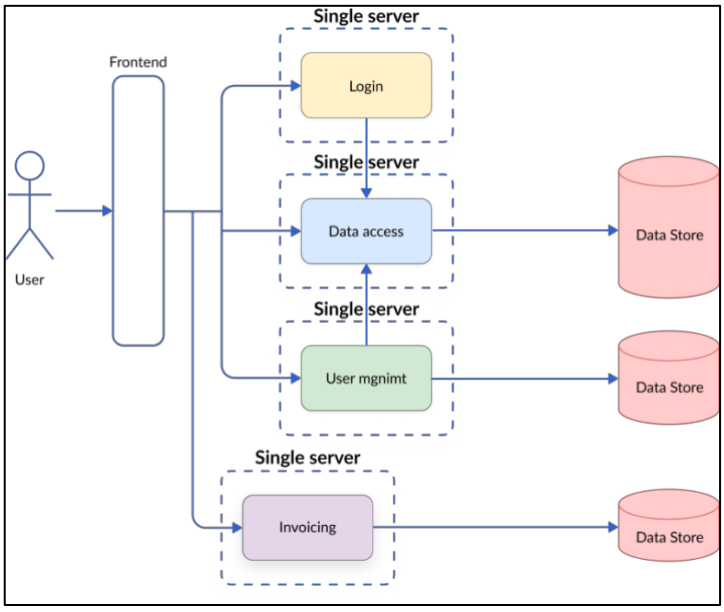

Рисунок 3.2 – Мікросервісна архітектура (за даними [7])

Також, мікросервіси відрізняються технологічною різноманітністю та автономією. Кожен сервіс функціонує самостійно, тому можна використовувати різні технології та фреймворки, якщо їхні інтерфейси не залежать від конкретних технологій (наприклад, використовують HTTP API або веб-сокети). Це відкриває безмежні можливості та дає змогу командам обирати найкращі інструменти для вирішення конкретних завдань. Мікросервіси подібні до динамічної екосистеми, в якій кожен сервіс може розвиватись та покращуватись у своєму власному темпі.

Безперервна доставка та розгортання є ще однією важливою перевагою мікросервісної архітектури. Сервіси не зв'язані між собою, тому їх можна розробляти, тестувати та розгортати окремо. Це прискорює виведення на ринок нових функцій та оновлень, оскільки їх можна розгортати без зміни всього додатку. У вас є гнучкий набір сервісів, готових адаптуватися та покращуватися в будь-який момент. Крім того, такий підхід дозволяє працювати над кожним мікросервісом одночасно кільком незалежним командам, що значно прискорює процес розробки.

3.2 Використання хмарних обчислень: вибір провайдера та стеку сервісів

Хмарні обчислення можна вважати однією з найяскравіших технологічних інновацій 21 століття. Це обумовлено швидким та широким поширенням цієї технології серед загальної публіки порівняно з іншими інноваціями. Цей широкий прийом головним чином обумовлений постійним зростанням кількості смартфонів та мобільних пристроїв, які мають доступ до Інтернету. Хмарні обчислення не лише корисні для організацій та бізнесу, але й для звичайних людей. Вони дозволяють запускати програми без необхідності встановлення їх на комп’ютерах, зберігати та отримувати доступ до мультимедійного вмісту через Інтернет, а також розробляти та тестувати програми без необхідності володіти власними серверами. Хмарні обчислення – це справжнє чудо 21 століття, яке залишається важливим практично в усіх сферах діяльності, про які можна подумати [9].

Чому хмарні обчислення важливі для бізнесу?

Об'єднані обчислювальні ресурси, що надаються хмарними технологіями, приносять значні переваги бізнес-організаціям. Ці переваги можна розділити на три великі категорії: ефективність, гнучкість і стратегічна цінність.

Ефективність: хмарні обчислення роблять бізнес-операції більш ефективними наступними способами

Доступність: хмарні обчислення забезпечують легкий доступ до додатків і даних з будь-якого пристрою, підключеного до Інтернету, в будь-якій точці світу.

Зниження витрат: хмарні обчислення надають компаніям масштабовані обчислювальні ресурси, що зменшує витрати на їх придбання та обслуговування. Оскільки ці ресурси оплачуються по мірі використання, компанії платять лише за ті ресурси, які вони використовують, що виявилося набагато дешевше, ніж купувати ресурси.

Безпека: хмарні провайдери, особливо ті, що пропонують послуги приватної хмари, зобов'язуються впроваджувати найвищі стандарти та процедури безпеки для захисту даних клієнтів, що зберігаються в хмарі.

Відновлення після збоїв: хмарні обчислення надають малим і середнім підприємствам і навіть великим підприємствам найефективніший засіб для швидкого і надійного резервного копіювання та відновлення даних і додатків.

Гнучкість: хмарні обчислення забезпечують наступну гнучкість

Масштабованість: хмарні обчислення є найкращим варіантом для підприємств з мінливим робочим навантаженням, оскільки хмарна інфраструктура може масштабуватися відповідно до потреб бізнесу.

Чому хмарні обчислення – це майбутнє?

З огляду на численні переваги, які хмарні обчислення дають організаціям, цілком логічно, що хмарні обчислення все більше стають новою нормою. Хмарні обчислення можуть допомогти суспільству вирішити такі майбутні проблеми, як управління великими даними, кібербезпека та управління якістю. Крім того, нові технології, такі як штучний інтелект, технологія розподілених бухгалтерських книг і багато інших, стають доступними у вигляді послуг за допомогою хмарних обчислень.

У результаті ці технології мають бути адаптовані до різних платформ, наприклад, до мобільних пристроїв, і їхнє використання зростає. Також розробляються інновації на основі хмарних обчислень, як-от хмарна автоматизація та промислові хмари, які дають змогу інтегрувати хмарні обчислення в більш конкретні види промислової діяльності. Остаточний вердикт щодо хмарних обчислень такий: це технологія, що перетворює, дає змогу

організаціям у всіх юрисдикціях надавати більш якісні продукти та послуги, ніж будь-коли раніше.

Крім того, за даними журналу Forbes, хмарні рішення стають дедалі популярнішими в багатьох технологічних проєктах, і очікується, що їхня популярність зростатиме з кожним роком (див. рис. 3.3).

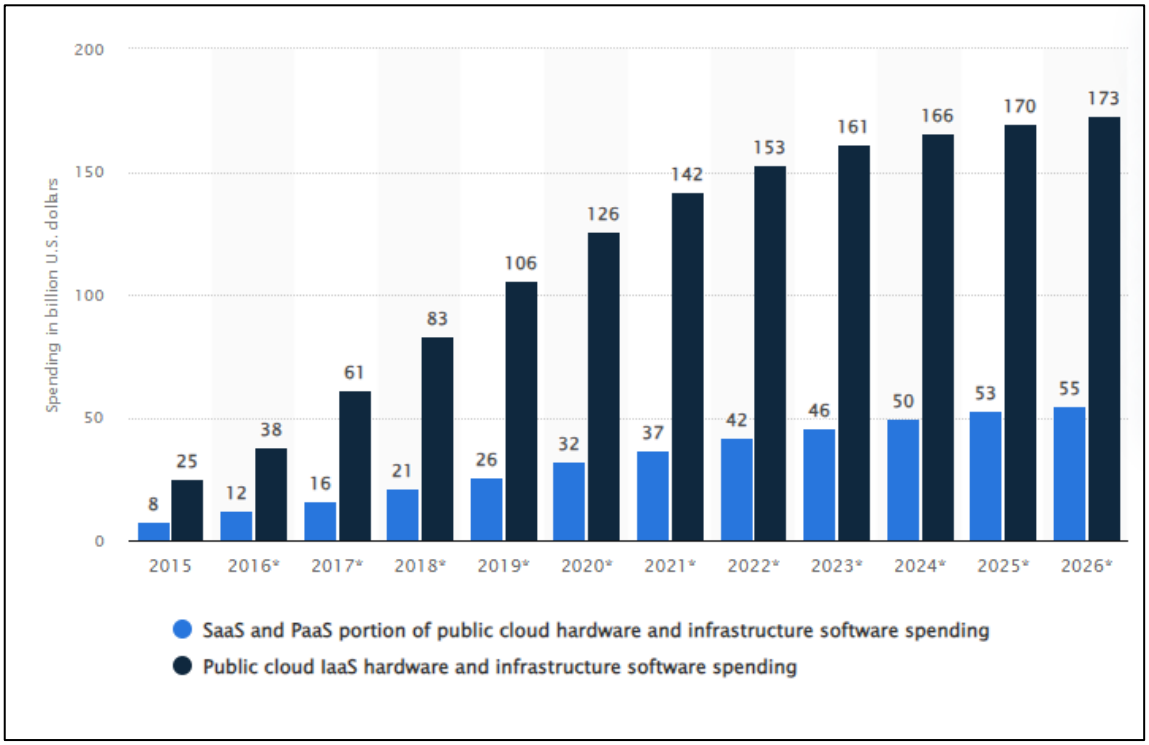

Рисунок 3.3 – Зростаюча популярність хмарних обчислень (за даними [10])

Принцип роботи пошукових систем полягає в тому, що вони є незалежними системами. Тому компанії не повинні знати, якою мовою написана пошукова система і які технології вона використовує, а пошуковий сервіс має бути масштабованим, швидким, відмовостійким і легко змінюваним. Саме це і дають нам хмарні обчислення. Тому в цій роботі було вирішено розробити незалежний пошуковий модуль із використанням хмарних обчислень.

3.3 Обгрунтування вибору постачальника хмарних обчислень

Пропонується розглянути основні провайдери хмарних обчислень.

До них треба віднести Amazon Web Services (AWS), AWS Microsoft Azure та Google Cloud Platform. Усі вони пропонують багато в чому схожих базових

можливостей щодо гнучких обчислень, зберігання та мереж. Всі вони мають спільні елементи загальнодоступної хмари: самообслуговування та миттєве надання, автоматичне масштабування, плюс функції безпеки, відповідності та управління ідентифікацією.

Всі три постачальники запустили послуги та інструменти, орієнтовані на найсучасніші технологічні галузі, такі як безсерверні обчислення (Lambda для AWS, Функції з Azure та Google), що надає клієнтам змоги «торкнутися» будь-якої хмари, щоб по-різному створити мобільний додаток або навіть створити високопродуктивне обчислювальне середовище залежно від їхніх потреб.

Зазначимо деяку інформацію про обчислення, зберігання, бази даних та мережі.

Для обчислень основною пропозицією AWS є екземпляри EC2 (Elastic Cloud Computing), які можуть бути адаптовані до великої кількості варіантів. Він також надає супутні послуги, такі як Elastic Beanstalk для розгортання додатків, послугу EC2 Container, ECS для Kubernetes (EKS), AWS Lambda та Autoscaling.

Тим часом обчислювальна пропозиція Azure зосереджена навколо її віртуальних машин (ВМ) з іншими інструментами, такими як хмарні служби та диспетчер ресурсів, які допомагають розгортати програми в хмарі, та службою автоматичного масштабування Azure.

Масштабований Compute Engine від Google забезпечує віртуальні машини в центрах обробки даних Google. Вони швидко завантажуються, оснащені постійним дисковим сховищем, обіцяють стабільну продуктивність і можуть бути легко налаштовані залежно від потреб замовника.

Всі три провайдери підтримують реляційні бази даних – це Azure SQL Database, Amazon Relational Database Service, Redshift і Google Cloud SQL – а також бази даних NoSQL з Azure DocumentDB, Amazon DynamoDB і Google Bigtable.

Зберігання AWS включає просте сховище (S3 – Simple Storage Service), еластичне блокове зберігання (EBS – Elastic Block Storage), еластичну файлову систему (EFS – Elastic File System), послугу імпорту / експорту великих обсягів

даних, резервне копіювання архівів Glacier та шлюз зберігання, який інтегрується з локальними середовищами [11].

Пропозиції Microsoft включають основну службу зберігання Azure, блочне сховище Azure Blob, а також сховище таблиць, черг та файлів. Він також пропонує відновлення сайтів, імпорт експорту та резервне копіювання Azure.

Всі три, як правило, пропонують чудові можливості роботи в мережі за допомогою автоматизованого балансування навантаження сервера та підключення до локальних систем, що підводить нас до гібридних варіантів.

Однією з тенденцій зростання серед гіпермасштабних державних хмарних провайдерів протягом останніх років було збільшення фокусу на допомозі в задоволенні гібридних та мультихмарних потреб клієнтів. Це, як правило, застосовується там, де клієнти розгортають інфраструктуру декількох постачальників, а також потребують постійного обслуговування деяких додатків. Постачальники відповіли цілою низкою рішень, щоб допомогти цим клієнтам, які ще не готові перейти на загальну хмару, що, звичайно, є більшістю великих підприємств.

Microsoft вже давно є варіантом гібридних розгортань серед великих трьох з її добре зарекомендованим Azure Stack. Це надає клієнтам апаратне та програмне забезпечення, необхідне для розгортання служб загальнодоступних хмарних служб Azure з локального центру обробки даних із спільним порталом управління, кодом та API для простої взаємодії.

AWS ознаменував свій перший серйозний перехід до гібридних розгортань 2018 році запуском Outposts – повністю керованої послуги, де постачальник доставляє заздалегідь налаштовані стелажі до ваших приміщень, де послуги AWS можна запускати як у їх дата-центр.

Основними постачальниками хмарних обчислень, які пропонується розглянути, є. Amazon Web Services (AWS), Microsoft Azure і Google Cloud Platform.

Усі вони пропонують схожі базові можливості в плані гнучких обчислень, зберігання даних і мережевої взаємодії. У них є загальні для публічних хмар

елементи, як-от самообслуговування, миттєве надання та автоматичне масштабування, а також функції безпеки, відповідності вимогам та управління ідентифікацією.

Усі три компанії анонсували сервіси та інструменти, орієнтовані на передові технологічні напрямки, як-от безсерверні обчислення (Lambda від AWS, Functions від Azure і Google), що дають змогу клієнтам "доторкнутися" до будь-якої хмари та створювати мобільні додатки по-іншому, високопродуктивне обчислювальне середовище, що відповідає їхнім потребам.

Нижче наведено інформацію про обчислення, сховища, бази даних і мережі.

Що стосується обчислень, то основним сервісом AWS є екземпляр EC2 (Elastic Cloud Computing), який може бути налаштований на безліч варіантів. Також пропонуються Elastic Beanstalk для розгортання додатків, сервіси EC2 Container, ECS for Kubernetes (EKS), AWS Lambda, Autoscaling та інші супутні послуги.

Обчислювальні послуги Azure, з іншого боку, зосереджені на віртуальних машинах (ВМ) і таких інструментах, як Cloud Services, Resource Manager і Azure Auto Scaling для підтримки розгортання хмарних додатків.

Scalable Compute Engine від Google надає віртуальні машини в центрах обробки даних Google. Ці віртуальні машини швидко запускаються, мають постійне дискове сховище, забезпечують стабільну продуктивність і легко налаштовуються під потреби замовника.

Усі три компанії пропонують реляційні бази даних (Azure SQL Database, Amazon Relational Database Service, Redshift і Google Cloud SQL), а також бази даних NoSQL (Azure DocumentDB, Amazon DynamoDB і Google Bigtable). DynamoDB і Google Bigtable).

Сховища AWS охоплюють служби Simple Storage Service (S3), Elastic Block Storage (EBS), Elastic File System (EFS), сервіси імпорту/експорту великих даних і архівного резервного копіювання Glacier, шлюзи зберігання, інтегровані з локальними середовищами [11].

Послуги, пропоновані Microsoft, включають Azure Basic Storage Service, блочне сховище Azure Blob, сховище таблиць, черг і файлів. Вона також пропонує відновлення сайтів, імпорт та експорт і резервне копіювання Azure.

Усі три сервіси, як правило, пропонують чудові мережеві можливості з автоматичним балансуванням навантаження на сервери і підключенням до локальних систем.

Однією з тенденцій останніх років серед постачальників гіпермасштабних публічних хмар є підвищена увага до задоволення потреб клієнтів у гібридних і мультихмарних рішеннях. Зазвичай це відбувається, коли замовники розгортають інфраструктуру від кількох постачальників, а також потребують постійного обслуговування деяких додатків. У відповідь на це постачальники запропонували цілу низку рішень, покликаних допомогти клієнтам, які ще не готові до переходу в публічну хмару.

Компанія Microsoft з її стеком Azure Stack, що добре зарекомендував себе, довгий час залишалася гібридним варіантом розгортання серед великої трійки. Вона надає клієнтам апаратне і програмне забезпечення, необхідне для розгортання публічних хмарних сервісів Azure з локальних центрів обробки даних, а також загальний портал управління, код і API для зручної взаємодії.

AWS відзначила перше серйозне зрушення в бік гібридного розгортання 2018 року, запустивши Outposts – повністю керовану послугу, що дає змогу провайдерам запускати сервіси AWS як власні центри оброблення даних, з попередньо налаштованими стійками на території замовника для доставки. Це фактично ребрендинг платформи Google Cloud Services, що об'єднує наявні Google Kubernetes Engine (GKE), GKE On-Prem і Anthos Config Management Console. Вона обіцяє уніфіковане управління, політики та безпеку для гібридних розгортань Kubernetes.

Далі необхідно розглянути досить важливе питання ціни.

Для тих, хто розглядає можливість переходу в хмару, ціна може стати основним привабливим фактором, і на те є вагомі причини.

Ціни зазвичай приблизно можна порівняти, особливо відтоді, як у 2017 році AWS перевела свої ціни на послуги EC2 і EBS на багаторівневу систему, привівши їх у відповідність з Azure і Google.

Однак чітке порівняння може бути ускладнене, оскільки всі три компанії пропонують дещо різні моделі ціноутворення, знижки та часте зниження цін. Звісно, не всі клієнти платять повну ціну, і з торговими представниками можна домовитися про знижки за обсяг, особливо на рівні підприємства.

Усі постачальники пропонують безкоштовні пробні версії для випробування своїх послуг перед покупкою, що дає змогу залучити на свої платформи інноваційні стартапи, а також зазвичай пропонують "завжди безкоштовні" рівні з жорсткими обмеженнями щодо використання.

Давайте розглянемо плюси і мінуси AWS.

Як уже йшлося вище, у різних клієнтів є свої причини обирати одного постачальника замість іншого. Однак є аспекти конкуруючих хмар, які можуть бути корисні в певних ситуаціях.

Широта і глибина пропозиції AWS розглядається як позитивний момент для AWS.

З 2006 року AWS стала провідним постачальником, створивши цілий набір хмарних сервісів. Усі вони розроблені так, щоб бути зручними для підприємств, привабливими як для основної бази розробників, так і для IT-директорів.

Вендор здобув визнання завдяки можливостям налаштування платформи, моніторингу та політик, безпеки та надійності. Його партнерська екосистема та загальна продуктова стратегія також лідирують на ринку: на AWS Marketplace доступна низка програмних сервісів сторонніх розробників.

Однак масштабність пропозицій AWS іноді розглядається як недолік. Хоча вибір цих опцій привабливий, може бути складно зорієнтуватися в безлічі пропонованих функцій, і деякі люди вважають AWS складним постачальником для управління.

Розглянемо плюси та мінуси Microsoft Azure.

Основна привабливість Azure полягає в тому, що компанія Microsoft вже міцно влаштувалася в організаціях і може легко зіграти свою роль у наданні допомоги цим компаніям у переході на хмарні технології. Azure можна використовувати спільно з основними локальними системами Microsoft, такими як Windows Server, System Centre і Active Directory. з основними локальними системами Microsoft, такими як Windows Server, System Centre і Active Directory. з основними локальними системами Microsoft, такими як Windows Server, System Centre і Active Directory.

За даними опитування 100 керівників вищої ланки IT-компаній, проведеного Goldman Sachs у січні 2020 року, 56 осіб обрали Azure порівняно з 48, які обрали AWS.

Однак одним із недоліків є низка збоїв протягом багатьох років, включно з великим глобальним збоєм у травні 2021 року.

Ще один недолік – "залежність від платформи". Тоді як інші хмарні провайдери можуть взаємодіяти з будь-якою системою, незалежно від її виробника, Microsoft характеризується високою надійністю тільки з локальними системами.

Плюси та мінуси хмарної платформи Google.

Google має великий досвід співпраці з інноваційними компаніями, що працюють у хмарі, і посідає міцні позиції в співтоваристві розробників відкритого коду, але традиційно зазнає труднощів із виходом на корпоративний ринок.

Стратегія компанії щодо виходу на ринок була спрямована скоріше на те, щоб довести свою спроможність у невеликих інноваційних проектах для великих підприємств, ніж на те, щоб стати стратегічним хмарним партнером. Якщо GCP хоче залучити більш традиційні компанії, їй слід розширити партнерські відносини та зосередитися на підтримці бізнес- та IT-процесів до хмари. процесів, що передують хмарі.

Однак, оскільки Google Cloud Platform – відносно новий постачальник послуг і його популярність тільки набирає обертів, складно визначити недоліки і, в даному випадку, чи можна довірити йому впровадження пошукової системи.

Якого провайдера обрати?

Загалом, AWS продовжує лідирувати за широтою можливостей і зрілістю; AWS залишається явним лідером ринку, але розрив скорочується.

Широкий спектр інструментів і послуг AWS, а також зручні для підприємств функції роблять його сильною пропозицією для великих підприємств. Водночас її велика і зростаюча інфраструктура забезпечує економію від масштабу, що дає змогу активно знижувати ціни.

Необхідно визначити постачальника. Для цього пропонується звернутися до графіків популярності використання конкретних хмарних провайдерів [12]: Amazon Web Services, Google Cloud Platform і Microsoft Azure. Як пошукові запити тут обрано три теми: Amazon Web Services, Google Cloud Platform і Microsoft Azure. Результати побудови графа показано на рисунку 3.4.

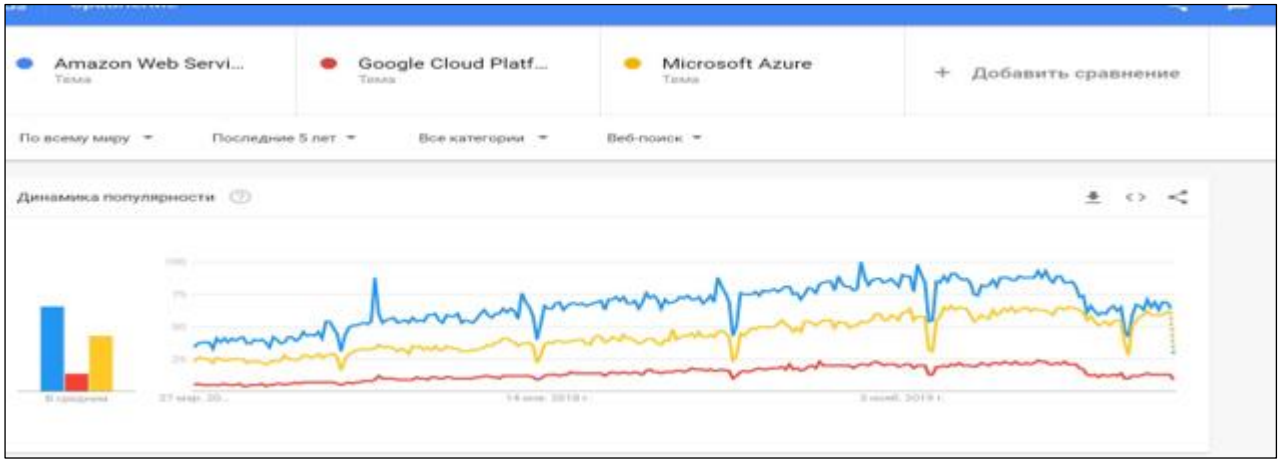

Рисунок 3.4 – Порівняння AWS, GCP, Microsoft Azure

Опитування і графіки популярності показують, що всі хмарні провайдери доволі популярні і мають усі сервіси, необхідні для вирішення завдання побудови архітектури пошукової системи.

Примітно, що сервіси, які використовуються для пошукових систем, доступні на Amazon, Azure і Google Cloud Platform. Це означає, що незалежно від того, яку платформу ви оберете, ви зможете перейти на іншу платформу без втрати коду, без проблем із написанням або використанням інших сервісів і без необхідності все переналаштовувати.

Тому ви можете вибрати будь-якого хмарного провайдера. У даній роботі було обрано Amazon Web Services, оскільки він найбільш популярний і має зрозумілу документацію з діаграмами та прикладами використання тих чи інших сервісів (див. рис. 3.5)

Рисунок 3.5 – AWS як найпопулярніший хмарний провайдер

Крім того, за даними авторитетних веб-журналів, AWS залишається найпопулярнішим вибором серед технічних рішень для різних проєктів.

3.4 Вибір потрібного стеку хмарних сервісів

Для того щоб ефективно використовувати хмарні обчислення для побудови архітектури системи, необхідно вибрати всі необхідні сервіси, які можуть знадобитися для цієї реалізації [13].

Почнемо з варіанта, коли пошукова підсистема є незалежним мікросервісом, що містить весь пошук, тобто фактично являє собою моноліт пошукових модулів. Це означає, що підсистему розгортають і збирають як єдиний виконуваний файл, вона відповідає за інтеграцію з корпоративною системою, приймає запити, які містять усі необхідні користувацькі дані, система під'єднується до корпоративної

бази даних і витягає всі необхідні дані та обрані умови, Це означає, наприклад, фільтрацію за всіма характеристиками користувачів і повернення відфільтрованого списку на підприємство.

На практиці це буде додаток клієнт-сервер, доступний підприємству за посиланням. Такі додатки мають відповідати всім принципам SOLID і бути побудовані з використанням архітектури REST:

- вона має багаторівневу структуру застосунку, де контролер на першому рівні приймає всі запити до системи та повертає результати;
- другий шар сервісів застосовує бізнес-логіку;
- третій рівень доступу до даних звертається до бази даних підприємства і витягує всі необхідні дані.

Зрозуміло, для реалізації такого застосунку необхідна серверна мова програмування з можливістю підключення до бази даних і швидкого опрацювання даних, бажано паралельного. До таких мов належить Java і його основний фреймворк Spring. Використовуючи цю мову та її технології, подібні додатки можна створювати за допомогою збирачів проектів, таких як Maven.

Складальник проєктів містить усі бібліотеки, від яких залежить застосунок, і за допомогою його команд збирає весь застосунок в один виконуваний файл для розгортання та запуску пошукової системи.

Усе, що необхідно зробити в цьому випадку побудови архітектури, – це створити сервер, на якому працюватиме додаток. Для цього в обраного провайдера AWS є окремий сервіс під назвою EC2.

Amazon Elastic Compute Cloud (Amazon EC2) надає масштабовані обчислювальні можливості в хмарі Amazon Web Services (AWS), оскільки використання Amazon EC2 не вимагає інвестицій в устаткування, Ви можете швидко розробляти та розгортати додатки Використовуючи Amazon EC2, ви можете запускати стільки віртуальних серверів, скільки вам потрібно, налаштовувати безпеку та мережі, а також управляти сховищем Amazon EC2 дає змогу адаптуватися до мінливих вимог та стрибків популярності Його можна

масштабувати або зменшувати, це дає змогу знизити необхідність прогнозування трафіку [1].

Amazon EC2 надає такі можливості:

- віртуальне обчислювальне середовище, відоме як екземпляр;
- заздалегідь сконфігуровані шаблони для екземплярів, відомі як образи машин Amazon (AMI), які містять усі необхідні для сервера компоненти (включно з операційною системою та додатковим програмним забезпеченням);
- різні конфігурації процесора, пам'яті, сховища і мережевих можливостей для екземпляра, які називаються типами екземплярів.
- сховище для тимчасових даних, які видаляються під час зупинки, сплячого режиму або вимкнення екземпляра (відомі як томи зберігання екземпляра);
- різні фізичні місця розташування для таких ресурсів, як екземпляри і томи Amazon EBS, звані регіонами або зонами доступності;
- статичні IPv4-адреси для динамічних хмарних обчислень, відомі як еластичні IP-адреси.

Можна створити віртуальну приватну хмару (VPC) – віртуальну мережу, логічно ізольовану від решти хмари AWS і здатну підключатися до вашої власної мережі в міру необхідності.

Іншими словами, AWS надає вам обмежену кількість безкоштовних серверів, на яких ви можете запустити свій пошуковий додаток.

Після запуску сервера, незалежно від його архітектури, залишається питання доступу до бази даних. База даних підприємства, звісно, не розташована на тому самому сервері, що й пошукова підсистема, і швидкість запитів буде низькою, реляційна вона чи ні.

Це означає, що щоразу, коли потрібно отримати якусь інформацію з бази даних, пошукова система витрачає час на під'єднання до сервера бази даних, надсилання запиту та отримання запиту, який може бути завантажений запитами підприємства на тому самому сервері бази даних. Існує ймовірність того, що один

і той самий сервер бази даних може бути завантажений корпоративними запитами. Що слід зробити в цей час?

Як уже зазначалося, вирішення цієї проблеми полягає в налаштуванні кешу бази даних. Для цього AWS надає можливості сервісів ElastiCache або DAX.

ElastiCache / DAX – це сховище даних in-memory, і одним з основних варіантів його використання є використання як кешу.

Прямого з'єднання з базою даних немає, а значення, отримані з бази даних, додаються в кластер ElastiCache/DAX.

Спочатку додаток перевіряє кеш-пам'ять і, якщо вона порожня, запитує базу даних для отримання даних. Успішні результати записуються в сховище ElastiCache/DAX, тому під час наступної спроби застосунок просто витягує результати з кешу.

Також слід розглянути варіант архітектури пошукової системи як повноцінного додатка мікросервісів, а не як незалежного моноліту. У цьому разі кожну функцію пошукового фільтра можна розглядати як незалежну сервісну функцію, тому немає потреби у фізичному, постійно працюючому сервісі.

Безсерверний – це зручний термін, який зазвичай відноситься до безсерверних додатків. Безсерверні додатки – це додатки, які не потребують виділення серверів та управління ними.

Усі послідовні функції замінюються окремими безсерверними функціями AWS Lambda. Тепер корпоративна система просто запускає легковагий застосунок за посиланням і чекає на виконання AWS Lambda – витрати трохи збільшилися, оскільки вам доведеться платити за кожен виклик Lambda, але тільки за пам'ять, використовувану під час виконання, тому за умови правильної оптимізації – витрати мінімальні. За правильної оптимізації витрати можна звести до мінімуму.

AWS Lambda – це сервіс безсерверних обчислень, що надається Amazon Web Services (AWS). Користувачі AWS Lambda створюють функції, які являють собою автономні програми, написані однією з підтримуваних мов і режимів виконання. Lambda і завантажують їх у Lambda.

Функції Lambda можуть виконувати будь-які обчислювальні завдання, від обслуговування веб-сторінок і обробки потоків даних до виклику API та інтеграції з іншими сервісами AWS [14].

Концепція "безсерверних" обчислень означає, що вам не потрібно утримувати власні сервери для запуску цих функцій; AWS Lambda – це повністю керований сервіс, який бере на себе всю інфраструктуру. "Безсерверність" не означає відсутність серверів, це означає, що сервери, операційна система, мережевий рівень та інша інфраструктура вже керовані, що дає вам змогу зосередитися на написанні коду програми.

Загальне порівняння AWS Lambda і AWS EC2 показано на рисунку 3.6.

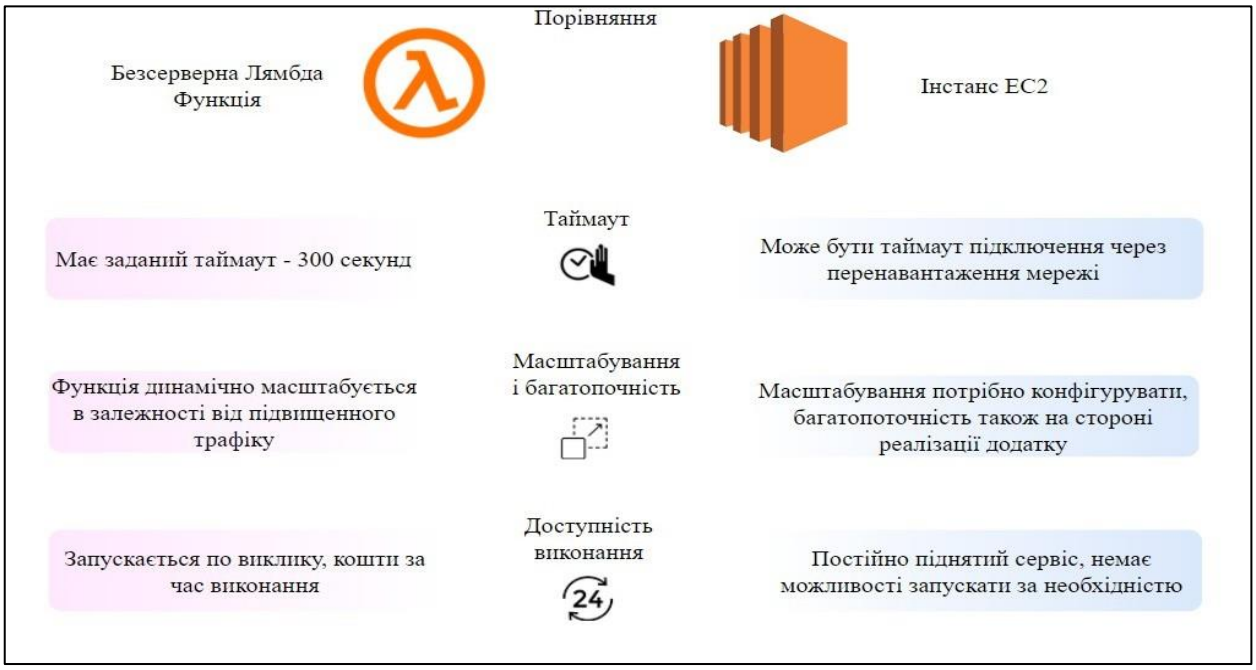

Рисунок 3.6 – Порівняння фізичних екземплярів серверів і безсерверних функцій

Це означає, що вся архітектура пошукової системи тепер може складатися з великої кількості послідовних або паралельних безсерверних функцій, які можна написати незалежно, розгорнути незалежно і які не впливають на поведінку всієї пошукової системи.

Проблема такого підходу полягає в тому, що AWS Lambda виконується після виклику і потім "закривається", тому стан її виконання не можна побачити або відстежити; AWS Lambda не має контексту, тому якщо будь-яка з них дасть

збій, то яким був вихід функції? Дуже складно дізнатися, чи були якісь помилки. Чи означає це, що нам потрібно повернутися до колишнього підходу, коли фізичні сервери завжди були запущені, історія запитів тощо. AWS пропонує рішення під назвою Step Functions, яке є новітньою технологією.

3.5 Застосування Amazon Step Functions

Вивчивши потребу мікросервісних додатків у послідовному виконанні частин системної логіки та поділі її на окремі сервіси, можна зробити висновок, що найпоширенішим рішенням є синхронний виклик хмарної функціональності як сервісу. Вивчення наявних досліджень показало такі особливості:

"Багато організацій, що використовують Amazon Web Services (AWS), спочатку використовували продукт Amazon Function-as-a-Service, AWS Lambda, для автоматизації робочих процесів, пов'язаних із запуском масштабованого коду. Хоча цей сервіс економить час розробників, у багатьох компаній виникли проблеми з використанням AWS Lambda для більш складних додатків".

Незважаючи на те що AWS Lambda є чудовим сервісом, у багатьох користувачів виникають проблеми у двох ключових сферах. Що стосується часу виконання, то AWS Lambda може працювати не більше 15 хвилин. Потім сервіс завершує роботу, залишаючи незавершене завдання.

Очевидно, що це не підходить для організацій зі складними додатками, які залежать від тривалого виконання завдань. Крім того, користувачі не можуть повернутися і повторно виконати невдалий код. Це відбувається тому, що, як уже зазначалося вище, кожне нове виконання є окремим новим виконанням зі своїм власним станом, оскільки функція, що виконується, завершується після виконання завдання і її не можна викликати знову.

Можна додати контейнери за допомогою таких інструментів, як Amazon Elastic Container Service (ECS), для виконання таких довгих завдань, але що станеться, коли контейнер буде зайнятий іншими завданнями? Доводиться додавати стани.

Проблема в тому, що AWS Lambda не призначена для ефективного управління станами. Розробникам доводиться писати код у своїх додатках для зберігання цих самих станів, що ускладнює управління і збільшує час обробки. У результаті користувачі змушені обирати між запуском високонавантаженого застосунку і забезпеченням достатнього стану для максимального використання [15].

Що таке AWS Step Functions.AWS Step Functions – це сервіс, що надається Amazon Web Services, який спрощує організацію декількох сервісів AWS для виконання завдань. Step Functions дозволяють використовувати результати одного кроку в іншому кроці процесу, так що результат одного кроку стає входом для іншого кроку, за допомогою візуального редактора робочих процесів.

Функції кроків надають безліч функціональних можливостей: автоматична обробка повторних спроб, активація та відстеження кожного кроку робочого процесу, а також забезпечення правильного порядку виконання кроків. Спочатку цей список може здатися не вражаючим, але в робочому процесі з десятками кроків і сотнями одночасних виконань виявляється, що гарантувати правильність їх виконання – завдання не з легких. Крокові функції виконують роботу, яку раніше робили Додатки [16].

Базова абстракція називається станом: конфігурація покрокових функцій – це карта всіх можливих кроків і переходів між ними. Стани і переходи між ними визначаються за допомогою мови станів Amazon. Ця мова заснована на JSON і є унікальною для Amazon.

Чому покрокові функції AWS важливі для безсерверної екосистеми При створенні безсерверних застосунків розробники зазвичай хочуть масштабувати застосунок зі зростанням навантаження і дати змогу кільком командам одночасно працювати над різними частинами застосунку, зберігаючи при цьому низькі витрати. Вони хочуть мати можливість зробити це. У безсерверній моделі й однією з найкращих практик для досягнення цих цілей є ізоляція бізнес-логіки застосунку в групу відокремлених сервісів. Великі безсерверні застосунки можуть складатися з десятків або сотень безсерверних сервісів.

Проблеми виникають, коли такій великій кількості сервісів потрібен доступ до різних частин загального стану. Щоб ці сервіси працювали ефективно, команда повинна мати можливість організувати потік даних між усіма сервісами програми в одному місці. Step Function – ключовий елемент безсерверної екосистеми, що забезпечує управління станом системи та даними, необхідними для ефективного масштабування безсерверних систем.

Step Function полегшує життя розробникам безсерверних систем, даючи їм змогу швидко створювати складні послідовності завдань в AWS, брати на себе обробку помилок і логіку повторних спроб, а також відокремлювати бізнес-логіку застосунку від логіки оркестровки. Далі ми розглянемо, як ця технологія може допомогти в проєктуванні та розробці систем [17].

Складні послідовності завдань можна створювати швидко. Організація послідовностей з десяти окремих безсерверних додатків, управління повторними спробами та налагодження будь-яких помилок можуть бути надзвичайно складним завданням. У міру додавання нових функцій складність управління зростає в геометричній прогресії.

Step Functions керує послідовністю завдань і знижує операційне навантаження завдяки графічному інтерфейсу і вбудованим засобам оперативного управління.

Управління станом між різними функціями без стану. У багатьох безсерверних робочих процесах налаштування черг і баз даних для взаємодії між усіма безсерверними сервісами може забирати багато часу і призводити до помилок.

Крокові функції дають змогу легко налаштувати управління станом на ранніх етапах і продовжують добре працювати навіть у разі збільшення обсягу завдання в міру спільного використання безлічі сервісів.

Ця техніка дає змогу відокремити логіку робочого процесу застосунку від бізнес-логіки. Це ще одна хороша практика безсерверної розробки. Додавання логіки робочих процесів і складної конфігурації сервера в застосунок, який має займатися тільки бізнес-логікою, збільшує складність застосунку і робить його

більш схильним до проблем. Крім того, управління станом окремо від бізнес-логіки дає змогу розробникам зберігати ясність під час роботи з безсерверними системами.

Серед інших переваг – ефективні робочі процеси за рахунок паралельного виконання. Управління станом різних функцій незалежно одна від одної може ускладнити досягнення високої продуктивності системи. Деякі оркестровані фрагменти можуть одночасно виконувати тільки одне завдання, що може уповільнити роботу всього додатка.

Крокові функції дають змогу одночасно виконувати безліч паралельних робочих процесів, тому продуктивність масштабується залежно від навантаження на додаток. Усі перераховані вище переваги можуть виявитися корисними для багатьох команд і сценаріїв використання, але перш ніж запускати Step Functions у виробничому середовищі, слід також розглянути потенційні недоліки використання Step Functions у застосунку. AWS Step Functions вирішує безліч проблем для спільноти serverless, але перенесення всього рівня оркестровки на Step Functions може не завжди бути правильним рішенням для вас. Перш ніж почати використовувати Step Functions у виробничому середовищі, необхідно запам'ятати кілька моментів.

Конфігурації мовою станів Amazon дуже складні, а їхній синтаксис ґрунтується на JSON, який оптимізовано для читання машинами, а не людьми. Ця мова може бути складною для вивчення і є унікальною для Amazon, тому отримані навички не застосовні поза контекстом AWS Step Functions [18].

Мова є власністю Amazon і може використовуватися тільки на AWS. Тому при переході до інших хмарних провайдерів шар оркестровки необхідно реалізувати заново або замінити на альтернативний варіант від іншого провайдера.

Ось кілька сценаріїв використання, в яких крокова функціональність AWS може підвищити продуктивність команди.

Обробка даних. Інші типи обробки даних також підходять для функції AWS step, особливо якщо джерело і пункт призначення знаходяться в AWS. Функція

Step спрощує об'єднання декількох етапів обробки даних, включно з тими, які виконуються функціями AWS Lambda, Simple Queue Service і Simple Notification Service.

Організація робочих процесів без серверів Основна цінність функції AWS Step полягає в простому налаштуванні застосунків, що вимагають об'єднання безлічі безсерверних функцій. Якщо у вас є бізнес-процес, що вимагає комбінації декількох безсерверних додатків для досягнення кінцевого результату, Step може стати правильним вибором для простої організації [19]. Безкоштовний рівень AWS включає 4000 переходів стану функції AWS Step на місяць. Ця частина безкоштовного рівня не має терміну дії і може бути використана, навіть якщо у вас немає нового облікового запису AWS.

На додаток до безкоштовного рівня вартість Step-функцій становить 0,025 долара США за 1000 переходів стану. USD за 1000 переходів станів.

Можливо, важко уявити, як це вплине на ваш щомісячний рахунок за послуги AWS.

Наприклад, припустімо, що у вас є робочий процес, який перетворює зображення для веб-сайту на чотири різні розміри, і цей робочий процес складається в середньому з 10 переходів станів для кожного зображення, причому приблизно 10 % виконання робочого процесу включає одну повторну спробу. Припустимо, ви завантажуєте 100 000 зображень на місяць.

У цьому випадку вартість покрокової функції AWS буде виглядати наступним чином: (10 переходів станів x 100 000 виконань + 1 повторна спроба x (100 000 x 10 % виконань)) x 0,025 дол. США за 1000 переходів станів = 25,25 дол. США.

Зверніть увагу, що плату за виконання покрокових функцій AWS стягують на додаток до плати за дані та плати за послуги AWS, які використовуються в робочому процесі. Наприклад, якщо при виконанні 100 000 робочих процесів плата за AWS Lambda становить 600 доларів США на місяць, а плата за дані – 100 доларів США на місяць, то загальні витрати будуть такими.

Плата за AWS Lambda $600 + плата за дані $100 + плата за покрокову функцію $25,25 = $725,25 на місяць за всю систему.

Пропонується переглянути це базове питання з урахуванням усіх переваг покрокових функцій, щоб визначити, чи є ще необхідність у використанні цієї технології.

По-перше, технологія мікросервісів передбачає, що кожен сервіс є незалежною частиною системи і може бути написаний різними мовами програмування. Тому у випадку з AWS – Lambda, де кілька функцій мають виконуватися послідовно як сервіси, реалізація Step Functions є необхідним кроком для встановлення порядку виконання, зберігаючи незалежність сервісів. Без цієї техніки ім'я наступної функції на кожному кроці довелося б задавати в кожній функції, а це значить, що сервіси знають один одного і тісно пов'язані між собою, що порушує концепцію мікросервісної архітектури.

По-друге, одним із недоліків Step Functions було те, що їх досить складно налаштовувати, оскільки вони вимагають знання нової мови AWS, Amazon State Language. Однак, як уже йшлося, уявіть собі структуру з функцій, які викликаються послідовно і знають одна про одну. Один з етапів можна виключити або змінити так, щоб, наприклад, шостий етап викликався після третього, а потім четвертий. Як уже говорилося, функція, яка викликає наступну функцію у своєму власному коді, тісно пов'язана з наступною функцією. Тому зміна порядку викликів або видалення одного з кроків виконання означатиме зміну коду майже кожної функції окремо, але якщо ви розумієте ASL[20] і налаштуєте покрокові функції, то зможете легко маніпулювати порядком тощо, не маючи потреби чіпати кожну окрему функцію.

По-третє, покрокові функції AWS вирішують проблему контексту між усіма послідовними функціями. Це, безумовно, плюс, тому що концепція Lambda говорить про те, що це швидка функція без контексту, тому розробка логіки для її передавання без покрокової функції суперечить самому принципу безсерверних функцій і робить логіку виконання та обслуговування складнішою і повільнішою. Як свідчить один із принципів розробки SOLID, S означає Single Responsibility

(єдина відповідальність). Іншими словами, якщо лямбди не можуть обробляти контекст і за своєю природою не повинні цього робити, то нехай цим займається створений для цього сервіс – AWS Step Functions, а не змушує його це робити. І нарешті, ціноутворення. Як уже говорилося вище, Step Functions – не дуже дорогий сервіс. Будь-яка компанія, яка може дозволити собі використовувати AWS для систем, побудованих на архітектурі мікросервісів, повинна розглянути можливість використання технології кінцевих автоматів Amazon для послідовного виконання функцій. Зрештою, навіть якщо потрібно послідовно виконати понад 100 функцій, машина виконає їх тільки один раз, якщо вона налаштована на Step Functions, і, як уже говорилося вище, 1000 функцій обійдуться лише в 0,025 долара для Amazon Web Services THE US [21].

4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗПРОГРАМНОГО ПРОДУКТУ

Функціонально-вартісний аналіз програмних продуктів досліджує доцільність застосування технологій, обраних у процесі роботи, та оцінює їхній вплив на розробку кінцевого продукту.

Функціонально-вартісний[22] аналіз сам по собі є методикою оцінювання вартості як продуктів, так і послуг, і не залежить від специфікацій компанії-виробника оцінюваного продукту або компанії, що надає відповідну послугу. Для того щоб отримати чіткі результати, необхідно

- визначити план розвитку продукту;
- вивчити весь спектр річних витрат і трудогодин, необхідних для реалізації проєкту;
- оцінити джерела витрат;
- розрахувати кінцевий результат.

Тепер перейдемо до алгоритму техніко-економічного аналізу методу FBA:

Розіб'ємо програмний продукт – чат (веб-додаток для обміну повідомленнями між користувачами) – на функції для подальшого аналізу:

а) вибір хмарного провайдера для безсерверних сервісів:

1) AWS Lambda;

2) Google Cloud Functions;

б) вибір СУБД:

1) FaunaDB;

2) Amazon DynamoDB;

в) вибір мови програмування:

1) Python;

2) JavaScript.

Нехай наведене вище дерево варіантів є морфологічною[23] картою системи (див. рис. 4.1).

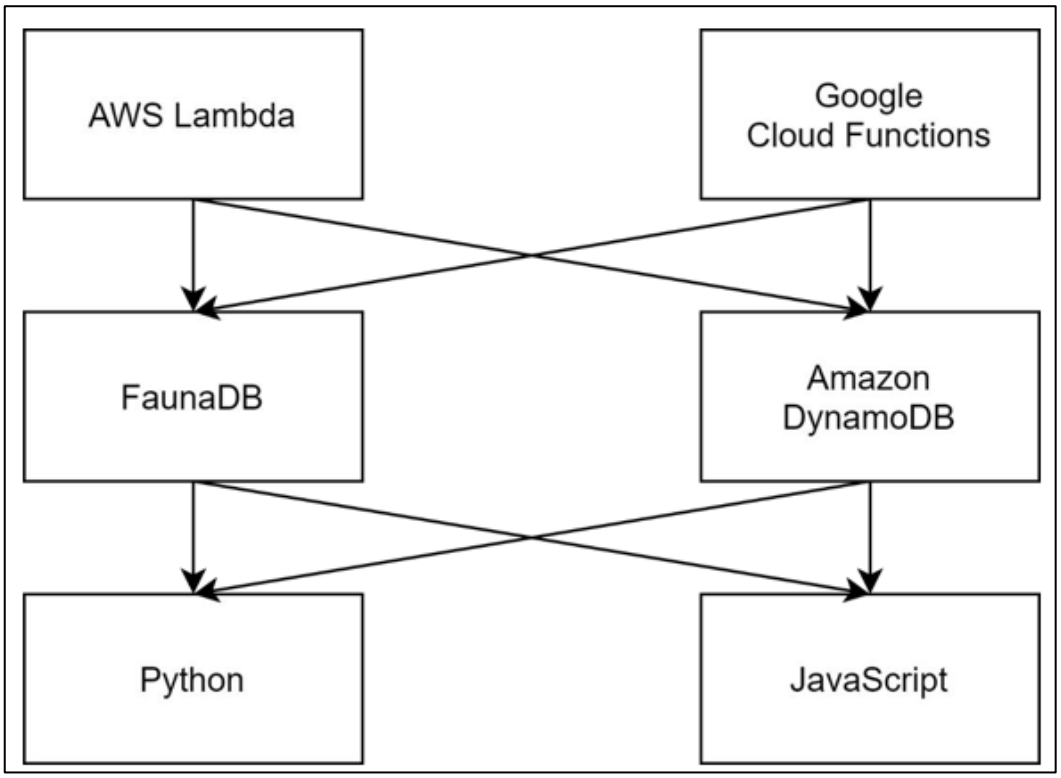

Рисунок 4.1 – Морфологічна карта

Зведемо всі варіанти в одну таблицю (див. табл. 4.1)

Таблиця 4.1 – Плюси і мінуси обраних технологій

Функція	Опція	Переваги	Недоліки
F1	A	Вартість запитів поза безкоштовним планом знижено вдвічі, підтримка всіх провідних мов програмування	Дуже глибока взаємодія з іншими сервісами AWS в екосистемі
	B	Кількість безкоштовних запитів збільшилася вдвічі.	Існує широкий спектр середовищ для виконання коду та стандартів його написання, кожне з них має власні особливості та рівень підтримки функціональності

Кінець таблиці 4.1

Функція	Опція	Переваги	Недоліки
F2	A	Архітектура без сервера, транзакційна незалежність, гнучкість та підтримка різних парадигм	Домінування одного постачальника, відсутність управління ресурсами та складність створення запитів
	B	Архітектура без сервера, транзакційна надійність, швидкість, глибока взаємодія з сервісами AWS та загальна популярність	Домінування одного постачальника та проблеми впровадження складних данихових структур
F3	A	Темпи розробки, доступність якісних бібліотек, мінімальні затримки у безсерверних умовах та можливість роботи на кількох платформах	Деякі завдання виконуються повільніше.
	B	Темпи розвитку, доступність великої кількості високоякісних бібліотек, мінімальні затримки у безсерверних середовищах, і можливість кросплатформенної роботи	Існує різноманітність середовищ виконання коду та стандартів його написання, кожен з яких має відмінності у підтримці функціоналу

Виключимо найгірший варіант на основі таблиці:

– для функції F1 варіант A є кращим за багатьма параметрами, тому ми обираємо його;

– розглядаючи варіанти для функції F2, ми бачимо, що обидва варіанти дуже схожі, але через відсутність вичерпних даних та тісну інтеграцію між додатком та AWS, ми обираємо варіант B;

– що стосується функції F3, то можна чітко сказати, що обидва варіанти майже схожі, але ми обираємо варіант А через відсутність суттєвих недоліків.

Це означає, що варіантами реалізації програмного продукту є: F1(A) – F2(B) – F3(A).

Визначимо параметри вибору та розглянемо систему параметрів програмного продукту.

Характерними параметрами є наступні (див. табл. 4.2):

– X1 – час затримки функції;
– X2 – швидкість обробки даних мовою програмування;
– X3 – швидкість роботи з базою даних;
– X4 – потенційний розмір коду.

Таблиця 4.2 – Характеристики програмних продуктів

Характеристика	Позначення	Одиниця виміру	Значення параметру		
			Гірші	Середні	Кращі
Затримка роботи функцій	X1	мс	120	10	5
Швидкість обробки даних мовою програмування	X2	операції в мс	10000	14000	21000
Швидкість роботи бази даних	X3	мільйонів операцій в секунду	18	20	23
Потенційний об’єм коду	X4	рядок	4000	2500	1000

За даними таблиці 4.2 побудуємо параметрів графіки характеристик(див. рис. 4.2 – 4.5).

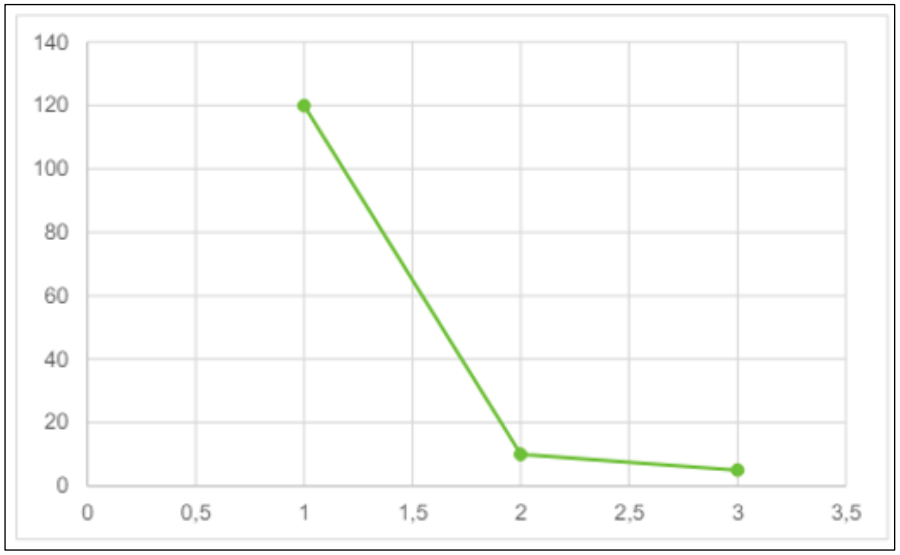

Рисунок 4.2 – Графік X1, затримка роботи функцій

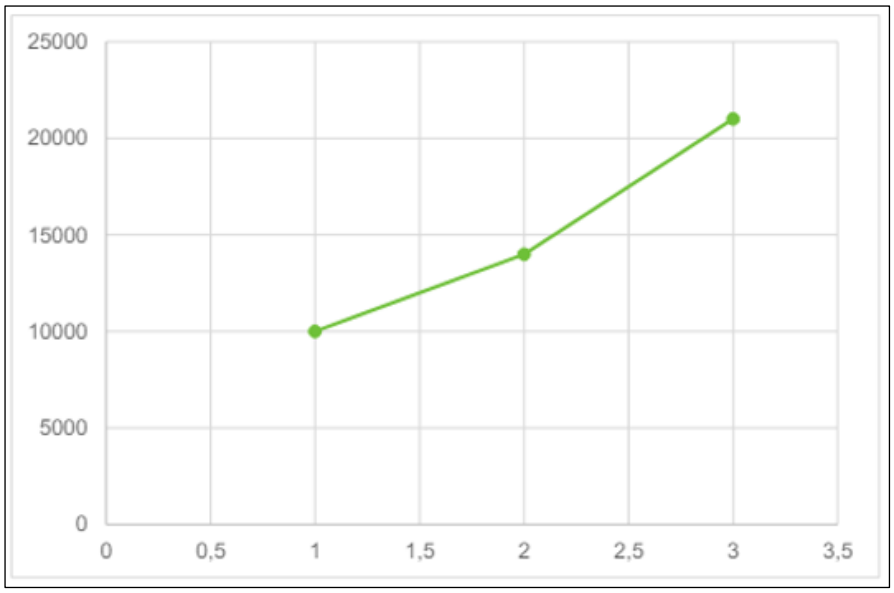

Рисунок 4.3 – Графік X2, швидкість обробки даних мовою програмування

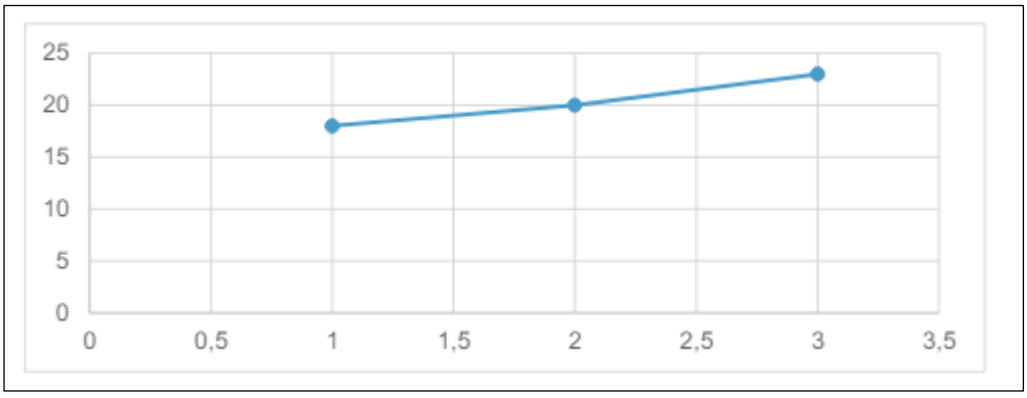

Рисунок 4.4 – Графік X3, швидкість роботи бази даних

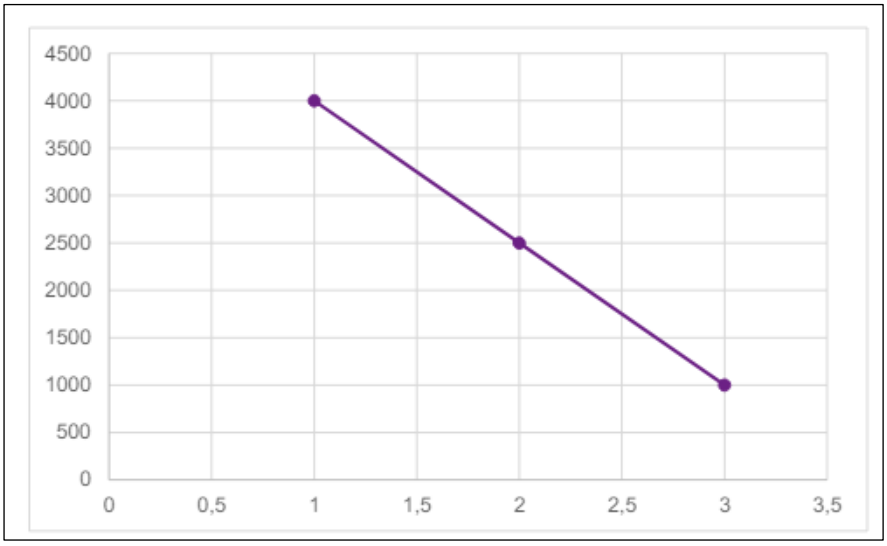

Рисунок 4.5 – Графік X4, потенційний об’єм коду

Проаналізуємо результати, отримані за допомогою методу експертних оцінок.

I. Визначення важливості характеристик шляхом їх ранжування.

II. Перевірка адекватності експертної думки.

III. Призначення пріоритетності параметрів та обробка результатів.

Таблиця 4.3 – Експертні оцінки

Позначення	Одиниці виміру	Ранг за оцінкою експерта						Сума рангів R_i	Відхилення Δ_i	Δ_i^2
		1	2	3	4	5	6			
X1	мс	4	4	5	3	3	4	23	3,25	10,56
X2	операції в мс	5	5	4	5	5	4	28	8,25	68,06
X3	мільйонів операцій в секунду	2	1	1	2	1	3	10	-9,75	95,06
X4	Потенційний об’єм коду	3	4	5	2	1	3	18	-1,75	3

Кінець таблиці 4.3

Позначення	Одиниці виміру	Ранг за оцінкою експерта						Сума рангів R_i	Відхилення Δ_i	Δ_i^2
		1	2	3	4	5	6			
Разом		14	14	15	12	10	14	79	0	176,75

Для перевірки достовірності експертної оцінки вводяться наступні параметри, які наведено нижче.

Загальний ранг оцінки (формула 1):

$$R_i = \sum_{j=1}^{N} r_{ij} R_{ij} = \frac{N_{n(n+1)}}{2} = 79 \quad (1)$$

де N – число експертів;

n – кількість параметрів.

Середня сума рангів (формула 2):

$$T = \frac{1}{n} R_{ij} = 19.75 \quad (2)$$

Відхилення суми рангів від середньої суми (формула 3):

$$\Delta_i = R_i - T \quad (3)$$

Сума квадратів відхилення (формула 4):

$$S = \sum_{i=1}^{N} \Delta_i^2 \quad (4)$$

Користуючись всіма параметрами оцінимо коефіцієнт узгодженості (формула 5):

$$W = \frac{12S}{N^2(n^3 - n)} = 2.398 > 0.67 \quad (5)$$

Отримані коефіцієнти перевищили нормативне значення 0,67, що свідчить про достовірність оцінок.

Проведемо попарне порівняння оцінених характеристик.

Числове значення aij, яке свідчить про те, що і-й параметр переважає j-й параметр, отримуємо за формулою 6 (див. табл. 4.4):

$$a_{ij} = \begin{cases} 1.5 \text{ при } X_i > X_j \\ 1.0 \text{ при } X_i = X_j \\ 0.5 \text{ при } X_i < X_j \end{cases} \quad (6)$$

Таблиця 4.4 – Попарне порівняння оцінок

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 і X2	>	>	<	<	>	>	>	>	1,5
X1 і X3	<	>	<	=	<	<	>	<	0,5
X1 і X4	>	>	<	=	<	=	>	>	1,5
X2 і X3	<	<	<	<	<	<	<	<	0,5
X2 і X4	<	<	<	<	<	<	<	<	0,5
X3 і X4	>	<	>	>	<	>	<	>	1,5

З отриманих числових оцінок переваги складемо матрицю $A = \| a_{ij} \|$.

Для кожного параметра зробимо розрахунок вагомості Кві за наступними формулами 7 та 8:

$$K_{\text{вi}} = \frac{b_i}{\sum_{i=1}^{n} b_i} \quad (7)$$

$$b_i = \sum_{i=1}^{N} a_{ij} \quad (8)$$

Відносна оцінка розраховується кілька разів, поки наступне значення не буде незначно відрізнятися (менше ніж на 2%) від попереднього; на другому та наступних кроках відносна оцінка розраховується за такою формулою 9 та 10:

$$K_{\text{вi}} = \frac{b_i'}{\sum_{i=1}^{n} b_i'} \quad (9)$$

$$b_i' = \sum_{i=1}^{N} a_{ij} b_j \quad (10)$$

Як видно з таблиці 4.5, різниця між значеннями вагових коефіцієнтів не перевищує 2%, тому подальші ітерації не потрібні.

Таблиця 4.5 – Розрахунок вагових коефіцієнтів параметрів

Параметри x_i	Параметри x_j				Перша ітер.		Друга ітер.		Третя ітер	
	X1	X2	X3	X4	b_i	$K_{\text{вi}}$	b^1_i	$K^{1\text{вi}}$	b^2_i	$K^{2\text{вi}}$
X1	1,0	1,5	0,5	1,5	4,5	0,36	17,75	0,25	73,38	0,25
X2	0,5	1,0	1,5	0,5	3,5	0,28	15,75	0,22	65,63	0,23
X3	1,5	1,5	1,0	1,5	5,5	0,44	22,75	0,33	93,6	0,32
X4	0,5	1,5	0,5	1,0	3,5	0,28	13,75	0,2	57,63	0,2
Всього:					12,5	1	70	1	290,25	1

Проаналізуємо якість варіантів реалізації функцій.

Коефіцієнт технічного рівня для кожного варіанта реалізації ПП розраховується так (формула 11) (див. табл. 4.6):

$$K_K(j) = \sum_{i=1}^{n} K_{вi,j} B_{i,j} \quad (11)$$

де n – кількість параметрів;

$K_{вi}$– коефіцієнт вагомості i–го параметра;

B_i – оцінка i–го параметра в балах.

Таблиця 4.6 – Розрахунок показників рівня якості варіантів реалізації основнихфункцій ПП

Основні функції	Варіант реалізації функції	Параметри	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1	A	X1	10000	6	0,25	1,5
F2	B	X2	64	5	0,23	1,15
F3	A	X3	1000	8	0,2	1,6

За даними з таблиці 4.7 за формулою (12) можна було б знайтирішення для спірних варіантів, якби такі були б.

$$K_K = K_{\text{ТУ}}[F_{1k}] + K_{\text{ТУ}}[F_{2k}] + \ldots + K_{\text{ТУ}}[F_{zk}] \quad (12)$$

Проведення економічного аналізу варіантів розробки програмного продукту.

Для визначення вартості розробки програмного продукту спочатку розраховується трудомісткість.

Всі варіанти передбачають виконання двох окремих завдань:

– розробка проекту програмного продукту;

– розробка програмної оболонки.

За ступенем новизни завдання 1 належить до групи А, а завдання 2 – до групи В. За складністю алгоритм, використаний у завданні 1, належить до групи 1, а у завданні 2 – до групи 3.

У завданні 1 використовується довідкова інформація, а у завданні 2 – інформація у форматі даних.

Для кожної задачі розрахуємо критерії часу розробки та програмування.

Загальна трудомісткість розраховується наступним чином (формула 13):

$$T_O = T_P \cdot K_П \cdot K_{СК} \cdot K_М \cdot K_{СТ} \cdot K_{СТ.М} \quad (13)$$

де T_P – трудомісткість розробки ПП;

$K_П$ – поправочний коефіцієнт;

$K_{СК}$ – коефіцієнт на складність вхідної інформації;

$K_М$ – коефіцієнт рівня мови програмування;

$K_{СТ}$ – коефіцієнт використання стандартних модулів і прикладнихпрограм;

$K_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення.

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру степеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює: T_P = 90 людино-днів. Поправочний коефіцієнт[24], який враховує вид нормативно-довідкової інформації для першого завдання: $K_П$ = 1.7. Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації для всіх семи завдань рівний 1: $K_{СК}$ = 1. Оскільки при розробці першого завдання використовуються стандартні модулі, врахуємо це за допомогою коефіцієнта $K_{СТ}$ = 0.8. Тоді загальна трудомісткість програмування першого завдання дорівнює:

$$T_1 = 90 \cdot 1.7 \cdot 0.8 = 122.4 \text{ людино-днів.}$$

Проведемо аналогічні розрахунки для подальших завдань.

Для другого завдання (використовується алгоритм третьої групи складності, степінь новизни Б), тобто ТР =27 людино-днів, КП =0.9, КСК = 1, КСТ = 0.8:

$T_2 = 27 \cdot 0.9 \cdot 0.8 = 19.44$ людино-днів.

Складаємо трудомісткість відповідних завдань для кожного з обраних варіантів реалізації програми, щоб отримати їх трудомісткість:

$T_I = (122.4 + 19.44 + 4.8 + 19.44) \cdot 8 = 1328{,}64$ людино-годин.

$T_{II} = (122.4 + 19.44 + 6.91 + 19.44) \cdot 8 = 1345.52$ людино-годин.

Варіант II є найбільш трудомістким.

У розробці беруть участь два програмісти з зарплатою 10 000 грн і аналітик даних з зарплатою 12 500 грн. Знайдіть середню заробітну плату за годину за формулою 14 та 15:

$$С_ч = \frac{М}{T_m \cdot t} \text{грн.} \quad (14)$$

де М – місячний оклад працівників;

T_m – кількість робочих днів тиждень;

t – кількість робочих годин в день.

$$С_ч = \frac{10000 + 10000 + 12500}{3 \cdot 21 \cdot 8} = 64{,}48 \text{ грн.} \quad (15)$$

Тоді, розрахуємо заробітну плату за формулою 16:

$$С_{ЗП} = С_ч \cdot T_i \cdot К_Д \quad (16)$$

де $С_ч$– величина погодинної оплати праці програміста;

T_i – трудомісткість відповідного завдання;

$К_Д$ – норматив, який враховує додаткову заробітну плату.

Зарплата розробників за варіантами становить:

I. $С_{ЗП} = 64.48 \cdot 1328.64 \cdot 1.2 = 102811{,}43$ грн.

II. $С_{ЗП} = 64.48 \cdot 1345.52 \cdot 1.2 = 104117{,}62$ грн.

Відрахування на єдиний соціальний внесок становить 22%:

I. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 102811,43 \cdot 0.22 = 22618,51$ грн.

II. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 104117,62 \cdot 0.22 = 22905,88$ грн.

Тепер визначимо витрати на оплату однієї машино-години. ($С_М$)

Так як одна ЕОМ обслуговує одного програміста з окладом 10000 грн., з коефіцієнтом зайнятості 0,2 то для однієї машини отримаємо:

$$С_Г = 12 \cdot М \cdot К_З = 12 \cdot 10000 \cdot 0,2 = 24000 \text{ грн.}$$

З урахуванням додаткової заробітної плати:

$$С_{ЗП} = С_Г \cdot (1 + К_З) = 24000 \cdot (1 + 0.2) = 28800 \text{ грн.}$$

Відрахування на соціальний внесок:

$$С_{ВІД} = С_{ЗП} \cdot 0.22 = 28800 \cdot 0,22 = 6336 \text{ грн.}$$

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 20000 грн.

$$С_А = К_{ТМ} \cdot К_А \cdot Ц_{ПР} = 1.15 \cdot 0.25 \cdot 20000 = 5750 \text{ грн.},$$

де $К_{ТМ}$ – коефіцієнт, який враховує витрати на транспортування та монтажприладу у користувача;

$К_А$ – річна норма амортизації; $Ц_{ПР}$ – договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо як:

$$С_Р = К_{ТМ} \cdot Ц_{ПР} \cdot К_Р = 1.15 \cdot 20000 \cdot 0.05 = 1150 \text{ грн.},$$

де $К_Р$ – відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо наступним чином:

$$Т_{ЕФ} = (Д_К – Д_В – Д_С – Д_Р) \cdot t_З \cdot К_В = (365 – 104 – 12 – 16) \cdot 8 \cdot 0.9 = 1677.6 \text{ годин},$$

де $Д_К$ – календарна кількість днів у році;

$Д_В$, $Д_С$ – відповідно кількість вихідних та святкових днів;

$Д_Р$ – кількість днів планових ремонтів устаткування;

t –кількість робочих годин в день;

$К_В$ – коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо наступним чином:

$$С_{ЕЛ} = Т_{ЕФ} \cdot N_С \cdot К_З \cdot Ц_{ЕН} = 1677.6 \cdot 0,3 \cdot 2,48 \cdot 2 = 2496,27 \text{ грн.},$$

де $N_С$ – середньо-споживча потужність приладу;

$К_З$– коефіцієнтом зайнятості приладу;

$Ц_{ЕН}$ – тариф за 1 КВт-годин електроенергії.

Накладні витрати розраховуємо наступним чином:

$$С_Н = Ц_{ПР} \cdot 0.67 = 20000 \cdot 0,67 = 13400 \text{ грн.}$$

Тоді, річні експлуатаційні витрати будуть (формула 17):

$$С_{ЕКС} = С_{ЗП} + С_{ВІД} + С_А + С_Р + С_{ЕЛ} + С_Н \quad (17)$$

$$С_{ЕКС} = 28800 + 6336 + 10589,76 + 5750 + 1150 + 2496,27 + 13400 = 57932,27 \text{ грн.}$$

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$$С_{М\text{-}Г} = С_{ЕКС} / Т_{ЕФ} = 57932,27 / 1677.6 = 34,53 \text{ грн/год.}$$

Оскільки в даному випадку всі роботи, які пов‘язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу, в залежності від обраного варіанта реалізації, складає (формула 18):

$$С_М = С_{М-Г} \cdot Т \quad (18)$$

I. $С_М = 34,53 \cdot 1328,64 = 45881,69$ грн.

II. $С_М = 34,53 \cdot 1345.52 = 46464,61$ грн.

Накладні витрати складають 67% від заробітної плати (формула 19):

$$С_Н = С_{ЗП} \cdot 0,67 \quad (19)$$

I. $С_Н = 102811,43 \cdot 0,67 = 68883,66$ грн.

II. $С_Н = 104117,62 \cdot 0,67 = 69758,80$ грн.

Отже, вартість розробки ПП за варіантами становить (формула 20):

$$C_{ПП} = C_{ЗП} + C_{ВІД} + C_{М} + C_{Н} \quad (20)$$

I. $C_{ПП} = 102811,43 + 22618,51 + 45881,69 + 68883,66 = 240195,29$ грн

II. $C_{ПП} = 104117,62 + 22905,88 + 46464,61 + 69758,80 = 243246,91$ грн.

Вибір найкращого варіанту програмного продукту на техніко-економічному рівні.

В результаті проведеного функціонального та вартісного аналізу програмного комплексу, що розробляється, можна зробити висновок, що перший варіант реалізації програмного комплексу є оптимальним серед варіантів, що залишилися після першого з двох варіантів реалізації програмного комплексу. Цей варіант програмного продукту включає безсерверну платформу AWS Lambda, базу даних Amazon DynamoDB та мову програмування Python [21-22].

ВИСНОВКИ

У підсумку дослідження можна визначити, що використання архітектури мікросервісів, спеціально на основі AWS і з використанням служби AWS Lambda, може принести значні переваги для розробки та розгортання додатків.

Архітектура мікросервісів дозволяє незалежно масштабувати кожну службу, полегшуючи обробку збільшеного трафіку та сприяючи швидшим циклам розробки. AWS Lambda дозволяє автоматизовано масштабувати функції, що реалізують окремі мікросервіси.

Незалежність служб робить систему менш вразливою до збоїв, оскільки проблеми в одній службі не впливають на інші. AWS Lambda, разом з Amazon API Gateway, дозволяє легко інтегрувати та публікувати API для кожного мікросервісу.

Використання безсерверних ресурсів, таких як AWS Lambda, дозволяє ефективно використовувати ресурси, оскільки обчислювальні ресурси витрачаються тільки при виклику функції.

Використання AWS Lambda та інших сервісів AWS дозволяє спростити управління інфраструктурою, оскільки AWS бере на себе багато аспектів, таких як масштабування, моніторинг та управління.

Незважаючи на ці переваги, важливо враховувати обмеження та виклики, пов'язані з архітектурою мікросервісів та використанням безсерверних технологій. Високий рівень децентралізації може призводити до збільшених витрат на управління та деяку складність в інтеграції.

Враховуючи ці аспекти, вибір архітектури мікросервісів на базі AWS, особливо з використанням AWS Lambda, має потенціал значно полегшити розробку, масштабування та управління додатками, забезпечуючи ефективне використання ресурсів і сприяючи інноваціям.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Microservices Architecture for Enterprise Large-Scaled Application URL – https://medium.com/design-microservices-architecture-with-patterns/microservices-architecture-for-enterprise-large-scaled-application-825436c9a78a (дата звернення: 27.11.2023);

2. Безсерверні обчислення на AWS URL – https://aws.amazon.com/ru/serverless/#:~:text=Serverless%20on%20AWS&text=AWS%20offers%20technologies%20for%20running,increase%20agility%20and%20optimize%20costs (дата звернення: 27.11.2023);

3. Microservices with Lambda URL – https://docs.aws.amazon.com/whitepapers/latest/serverless-multi-tier-architectures-api-gateway-lambda/microservices-with-lambda.html (дата звернення: 27.11.2023);

4. Amazon API Gateway: Effortless API Management URL – https://medium.com/cloud-native-daily/amazon-api-gateway-effortless-api-management-2bfa6e50f5ac (дата звернення: 27.11.2023);

5. AWS DynamoDB for Serverless Microservices URL – https://enlear.academy/aws-dynamodb-for-serverless-microservices-2acbbbff1bca (дата звернення: 27.11.2023);

6. Martin Fowler – Microservices URL – http://martinfowler.com/articles/microservices.html (дата звернення: 24.02.2024),

7. Monolithic and microservice architecture. Comparison URL – https://habr.com/en/companies/haulmont/articles/758780/ (дата звернення: 24.02.2024);

8. I. Nadareishvili. Microservice Architecture: Aligning Principles, Practices, and Culture / I. Nadareishvili, R. Mitra, M. McLarty, M. Amundsen – O'Reilly Media, 2016 – 146 p.;

9. Introduction to microservices. URL – https://nginx.com/blog/introduction-to-microservices/ (дата звернення: 24.02.2024);

10. Public cloud Infrastructure as a Service (IaaS) hardware and software spending from 2015 to 2026, by segment URL –

https://www.statista.com/statistics/507952/worldwide-public-cloud-infrastructure-hardware-and-software-spending-by-segment/ (дата звернення: 24.02.2024) ;

11. Офіційний сайт Google: Compare AWS and Azure services to Google Cloud. URL – https://cloud.google.com/free/docs/aws-azure-gcp-service-comparison (дата звернення: 24.02.2024);

12. Google trends URL – https://trends.google.com/ (дата звернення: 24.02.2024);

13. Jurg van Vliet. Programming Amazon EC2: Survive your Success / J. Vliet, F. Paganelli, 2016 – O'Reilly Media, 133p.;

14. Using an API Gateway. URL – https://nginx.com/blog/buildingmicroservices-using-an-api-gateway/ (дата звернення: 24.02.2024);

15. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions / Gregor Hohpe, Bobby Woolf – Addison-Wesley, 2004 – 736 p.;

16. AWS Step Functions: Developer Guide URL – https://www.amazon.com/AWS-Step-Functions-Developer-Guide-ebook/dp/B078XBSLY5 (дата звернення: 24.02.2024);

17. Офіційний сайт C++ Micro Services. URL – http://cppmicroservices.org/ (дата звернення: 24.02.2024);

18. Andreas Witting. Amazon Web Services in Action / A. Witting, M/ Witting – Shelter Island, 2015 – 200p.;

19. Vijin Boricha. Learn AWS Serverless Computing / V. Boricha, M. Rajani, Alokita Amanna – Packt Publishing, 2021 – 174 p.;

20. Chris Richardson. From Design to Deployment / Chris Richardson, Floyd Smith, 2016. – 74 p.

21. Kiriy V.V, Sheiko, I., Petrova, R., Optimization of management information support as a basis for organizational transformations at an enterprise, Periodicals of Engineering and Natural Sciences. 2019. V.7. N. 2. P. 679-689

22. Кравець Н. С., Інтерактивна імітація та аналіз розкрашеної сітки Петрі з використанням алгебри предикатних операцій, Праці п'ятої всеукраїнської міжнародної конференції 27 листопада -1 грудня 2000р.,Україна,Київ. С.313-316.

23. Shubin I.Yu, A. Kozyriev, V. Liashik, G. Chetverykov, Methods of Adaptive Knowledge Testing Based on the Theory of Logical Networks, Proceedings of the 5th International Conference on Computational Linguistics and Intelligent Systems (COLINS-2021), Kharkiv, Ukraine, April 23-24, 2021. – Volume I, P. 1184-1193. (CEUR, SCOPUS)

24. K. Smelyakov, A. Chupryna, o. Bohomolov and N. Hunko, “The Neural Network Models Effectiveness for Face Detection and Face Recognition”, 2021 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream), Vilnius, Lithuania, 2021, pp. 1-7, doi: 10.1109/eStream53087.2021.9431476